OpenPCells

PCell Design Guide and API

Patrick Kurth

June 29, 2023

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. The name for this function is `layout()`. Additional functions such as `parameters()` are also understood.

1.1 First Example – 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 and add it to the main object
    geometry.rectanglebltr(
        obj,
        generics.metal(1),
        point.create(0, 0),
        point.create(_P.width, _P.height)
    )
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:

```
geometry.rectanglebltr(
    obj,
    generics.metal(1),
    point.create(0, 0),
    point.create(_P.width, _P.height)
)
```

There is a lot going on here: We use the `geometry.rectanglebltr` function to create a rectangle with two corner points (bottom-left, bl and top-right, tr). 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 pass the main object as first argument to the function, which places the rectangle within this object.

[1] Well, this is not entirely sure. Only integers are allowed and the base unit is assumed to be nanometer. This is also currently reflected for example in the GDSII export, where the scaling is done approriately. However, it is planned that this will change in the future, making the base unit in opc arbitrary.

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

```
opc --technology opc --export gds --cell library/simple_rectangle
```

where `library` is a folder where the cell is placed in and lies in the cell search path.

Now you already now how to create simple rectangles with generic layers. As integrated circuits are mostly made up of rectangles, 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. Furthermore, to reduce complexity hierarchical layout design is also important. We will talk about these topics in the remaining cell tutorials.

1.2 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 width of the connecting rails and the used metals. The shown cell is a simplified version of the actual momcap implementation in opc:

```
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 },
        { "rext(Rail Extension)",       0 },
        { "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 listing the defined parameters of a cell, the display names are used, but within the cell the regular names are significant. The parameters are stored in a lua table and can be accessed in two ways: `_P.fingers` and `_P["fingers"]`. Usually, the first way is easier, but it requires the parameter name to be a valid lua identifier. Names like `foo-bar` (with a hyphen) are not valid identifiers. In this case, the second way would have to be used.

In `layout()` first the metals are resolved. This makes sure that only positive integers are used (for instance, with a metal stack of five metals, the index -2 is resolved to 4). This is done in order to have properly-defined values for the following for-loop iterating over all metals.

```
local firstmetal = technology.resolve_metal(_P.firstmetal)
local lastmetal = technology.resolve_metal(_P.lastmetal)
```

Then we can set up the loop:

```
for m = firstmetal, lastmetal do
```

At first, we create the rails (upper and lower):

```
    geometry.rectanglebltr(
        momcap, generics.metal(m),
        point.create(-_P.rext, 0),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.rwidth)
    )
    geometry.rectanglebltr(
        momcap, generics.metal(m),
        point.create(-_P.rext, _P.fheight + 2 * _P.foffset +
            _P.rwidth),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.fheight + 2 * _P.foffset + 2 *
            _P.rwidth)
    )
```

Then we create the fingers separately. A second loop represents the fingers. Every finger has to be moved right, every second finger has to be moved up:

```
    for f = 1, _P.fingers do
        local xshift = (f - 1) * pitch
        local yshift = (f % 2 == 0) and 0 or _P.foffset
        geometry.rectanglebltr(
            momcap, generics.metal(m),
            point.create(xshift, _P.rwidth + yshift),
            point.create(xshift + _P.fwidth, _P.rwidth + yshift +
                _P.fheight + _P.foffset)
        )
    end
```

What remains is the drawing of the vias between the metals. For this we introduce a new `geometry` function: `geometry.viabltr`. It takes a rectangular area and creates individual cuts as well as surrounding metals. There has to be some technology translation for this (proper layer generation as well as calculating the proper geometry of the cuts). The details on this are not important for this discussion. It is covered more in-depth in the technology translation manual. For this case it is enough to know that an appropriate and manufacturable amount of via cuts is placed within this rectangular area. Since the region is a rectangular, `geometry.viabltr` takes almost the same arguments as `geometry.rectanglebltr`. Only the metal layer is changed into two indices for the first and the last metal of the stack. This means that `geometry.viabltr` does *not* expect a generic layer as input. All layer creation is done by the function itself. Furthermore, we don't have to specify the individual vias between each layer in the stack, this is resolved later by the technology translation. For the capacitor, the vias are placed in the rails:

```
if firstmetal ~= lastmetal then
    geometry.viabltr(
        momcap, firstmetal, lastmetal,
        point.create(-_P.rext, 0),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.rwidth)
    )
    geometry.viabltr(
        momcap, firstmetal, lastmetal,
        point.create(-_P.rext, _P.fheight + 2 * _P.foffset +
            _P.rwidth),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.fheight + 2 * _P.foffset + 2 *
            _P.rwidth)
    )
end
```

With this the pcell is almost finished, the remaining code defines so-called *anchors*, which are used for relative positioning. These will be discussed further down this document and therefore are skipped in this section. A cell similar to this is bundled in this release of openPCells (`cells/passive/capacitor/mom.lua`). A few optimizations and additional parameters are added, but the here shown implementation is the basic structure of this capacitor.

1.3 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 (a *path*) routed in a special manner, therefore we will describe the inductor as a `path`. This is a series of points that describe a line with a certain width. To create a path, we have to pass the points to `geometry.path`, which we will store in a `table`. The cell for the octagonal inductor requries some calculations for the right point positions and uses some helper functions. We won't discuss the entire cell here as some issues are not important for general advice on building cells. The main purpose is to show how to handle points, point lists and how to draw paths.

First, some information on points: Points are a structure with an x- and a y coordinate. They represent absolute locations in the layout and many layout- and object-related take either a pair of x/y coordinates or a point. Points are simply created by `point.create`, which takes two numbers for the coordinates (x comes first). The coordinates can be queried:

```
local pt = point.create(100, 100) -- create a point
local x, y = pt:unwrap() -- get both x and y
```

```
x = pt:getx() -- get only x
y = pt:gety() -- get only y
```

For some layouts, combinations of points are needed, where for instance the x-coordinate of one point should be combined with the y-coordinate of another point. There are two ways to achieve this:

```
-- newpt == point.create(pt1.x, pt2.y)
local newpt = point.combine_12(pt1, pt2)
-- newpt == point.create(pt2.x, pt1.y)
local newpt = point.combine_21(pt1, pt2)
```

Both functions do the same, they just differ on the order of their arguments. Furthermore, some mathematical operators are defined for points:

```
local pt1 = point.create(100, 100)
local pt2 = point.create(20, -100)
print(pt1 + pt2)  -- (60, 0)       -> arithmetic average
print(pt1 - pt2)  -- (80, 200)     -> difference
print(-pt1)       -- (-100, -100) -> unary minus
print(pt1 .. pt2) -- (100, -100)  -> point.combine_12(pt1, pt2)
```

Lastly, there is a shorthand for the scalar distance in either x or y:

```
point.xdistance(pt1, pt2)
point.ydistance(pt1, pt2)
```

Let us get back to the inductor. It has the number of turns as a parameter. For every turn the points for one half of the turn are calculated, then the path is drawn twice, one time with a mirrored version of the points.

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

The points are stored in the `table` `pathpts`, `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`.

Then we add points:

```
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)
```

Now the cell adds points for the underpass (for the crossing of both sides) as well as the extension for the connections. This discussion will skip these parts as they don't add any value for learning about `geometry.path`. The entire code generating the path points is a bit complex and involves some trigonometric calculations.

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):

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

The `geometry.path` function takes five arguments: the cell, 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. The layers where created earlier as

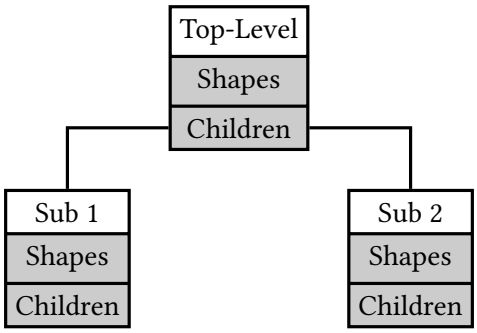
```
local mainmetal = generics.metal(_P.metalnum)
local auxmetal = generics.metal(_P.metalnum - 1)
```

1.4 Integrating Other Cell Layouts

Layouts of integrated circuits can get very complex quite easily. The typical way to deal with high complexity is by building individual cell layouts and placing them in the appropriate location. Partitioning and hierarchy are good tools to tackle circuit complexity. In essence: Divide and conquer. Therefore, this section will discuss how to integrate existing cell layouts and how to find the correct location by relative placement. But first an introduction to how openPCells represents layouts will be given, as this will be important to understand advanced topics.

1.4.1 Representation of Hierarchical Layouts

In integrated circuits, layouts are in general represented by a mixture of instantiations of other layouts as well as shapes (such as rectangles, polygons, paths, etc.). Typically, there is a root cell for the layout, which is called top-level. The top-level layout then has some shapes as well as instantiations of other layouts, which in turn have shapes and Instantiations of other layouts and so on. This is quite similar to a filesystem, where you have a root directory with sub-directories, where every directory can (but not necessarily) contain files that are not directories. The following picture shows an example of this:



Every layout cell holds a list of shapes and a list of children (references to other layout cells). Furthermore, layout cells can be transformed, that is translated in x and y or rotated, mirrored etc. This affects all parts of a cell, so shapes and children are also transformed, which is just what you would expect. Object transformation is a very cheap operation, much cheaper than operating on every single shape in a layout. This is the reason why layout hierarchies are usually much more efficient to work on than so-called flat layouts. How cell hierarchies are created is discussed in section 1.5. In this section we will continue to look at flat layouts but still using other cells.

1.4.2 Adding Cell Contents

For this example, we assume that we have a cell containing some layout in a variable:

```
function layout(toplevel)
    local cell = -- get the layout from somewhere
```

Now we want to place the contents of `cell` in `toplevel`. The method for this is `merge_into`. This method takes all shapes from a cell, copies them and places them in the cell that called `merge_into`. In our example, in order to place the contents from `cell` in `toplevel`, we do the following:

```
toplevel:merge_into(cell)
```

This dissolves the cell and places the content (the shapes and the children) into the toplevel cell.

1.4.3 Moving Cells

Now that we saw how to add cell content to other cells, the question arises how we can place this content at the correct position. As an example, let's say we want to build a ring oscillator. Here, *N* copies of the same inverter are placed. The inverter has a certain width so we know how far we have to move it so that it does not overlap with the surrounding inverters. For the example, we will place three inverters (of which we will assume that a layout is available):

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter and translate it
    local width = 1000 -- the width needs to be known
    local inverter1 = inverter:copy():translate(0 * width, 0)
    local inverter2 = inverter:copy():translate(1 * width, 0)
    local inverter3 = inverter:copy():translate(2 * width, 0)
    -- merge into toplevel
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

In this example, the original cell is copied three times, since every call to `translate` changes the internal state of the cell. It is possible to do this without intermediate variables and copying:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    local width = 1000 -- the width needs to be known
    -- translate and merge into toplevel
    toplevel:merge_into(inverter)
    toplevel:merge_into(inverter:translate(width, 0))
    toplevel:merge_into(inverter:translate(width, 0))
end
```

Here, `translate` is called on the original cell, so the movements in x and y accumulate. This is better in regards of processing performance, as copies of cells are expensive.

The approach of explicitly translating cells to move them to the right location works, but requires the knowledge of some parameters of the cell. Usually these values are known, since the layout of a single inverter was also created in this cell. However, for more abstraction it is desirable to describe these kind of layouts in a relative way. For this, the cells should know their own width and height, so they could be placed aligned to each other in an automatic way. There are two mechanisms in openPCells to help with these placements: anchors and alignment boxes.

1.4.4 Cell Anchors

In order to place cell at certain points without knowing their exact geometry, *anchors* are introduced. An anchor is a meaningful/important point of a cell and marks a location where other cells can be attached to or where wires can start/end etc. Think of a jigsaw puzzle pieces with their tabs and blanks: Pieces are connected by placing a tab in a blank of another piece. To come back to the example of the ring oscillator, we connect the inverters by placing the inputs on the outputs of the previous cells. The modified example looks like this:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter
    local inverter1 = inverter:copy()

    -- get required displacement for inverter2
    local output1 = inverter1:get_anchor("output")
    local input2 = inverter2:get_anchor("input")
    -- copy and translate inverter2
    local inverter2 = inverter:copy():translate(output1 - input2)
    -- get required displacement for inverter3
    local output2 = inverter2:get_anchor("output")
    local input3 = inverter3:get_anchor("input")
    -- copy and translate inverter3
    local inverter3 = inverter:copy():translate(output2 - input3)
    -- merge inverters into toplevel
```

```
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

The required displacement for each inverter is calculated and then applied to `translate`. This is rather cumbersome. OpenPCells offers a specialized function for this: `move_point`. This takes a layout cell and moves it so that the specified anchor lies at the given location:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter and move it to the origin
    local inverter1 = inverter:copy():move_point("input")
    inverter1:move_point(
        inverter1:get_anchor("input"),
        point.create(0, 0)
    )
    -- copy and translate inverter2
    local inverter2 = inverter:copy()
    inverter2:move_point(
        inverter2:get_anchor("input"),
        inverter1:get_anchor("output")
    )
    -- copy and translate inverter3
    local inverter3 = inverter:copy()
    inverter3:move_point(
        inverter3:get_anchor("input"),
        inverter2:get_anchor("output")
    )
    -- merge inverters into toplevel
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

The function `move_point` takes arbitrary points as arguments. Essentially, it moves the specified cell by the difference of the two points. The interface is a bit akward, because moving cells by their own anchors (a common use case) requires typing out the object name twice. This is due to the generic nature of `move_point`. A more specialized and powerful approach is provided by *area anchors*, as demonstrated in the next section.

1.4.5 Area Anchors

An area anchor is similar to a regular anchor, but it makes up a rectangular area (defined by the bottom-left and the top-right). These anchors can be queried just like regular anchors (but with `get_area_anchor` instead of `get_anchor`). Their two main properties and advantages over regular anchors are that they never change their orientation and cells can be abutted/aligned in various ways with them.

To clarify on the point of orientation: Imagine a rectangular area was made up of two regular anchors. Once the object containing these anchors is rotated by, say, 90 degrees, the bottom-left anchor is no longer the bottom left. Flipping an object for instance reverses the order of these anchors. Area anchors on the other hand always describe, as the name implies, areas. Therefore it does not matter which orientation the respective object has, the bottom-left anchor will always be bottom-left. This enables much simpler approaches, as cell code does not need to keep track of which object is rotated and which one is not.

The second advantage of area anchors are the various abutment/alignment methods for objects. For instance, the `basic/mosfet` cell defines area anchors for every source/drain region. Two devices can be simply abutted by calling

```
mosfet1:align_area_anchor("sourcedrain1", mosfet2, "sourcedrain-1")
```

This places `mosfet1` left of `mosfet2` (as source/drain regions are counted positive beginning from the left and negative beginning from the right). Without area anchors, the call would look like this:

```
mosfet1:move_point(mosfet1:get_anchor("sourcedrain1bl"), mosfet2:
    get_anchor("sourcedrain-1bl"))
```

But this code would break if one of the devices was flipped (for instance to place a gate contact on the top and one on the bottom).

1.4.6 Alignment Boxes

Alignment boxes are tighly connected to area anchors. They provide the same concept but in a simpler and less flexible way. Every cell has at most one alignment box. With this, it can be aligned/abutted to other cells. This makes the most sense for compatible cells (like mosfets and other mosfets).

The above example of abutting two mosfets with area anchors can be re-written as:

```
mosfet1:abut_right(mosfet2)
mosfet1:align_top(mosfet2)
```

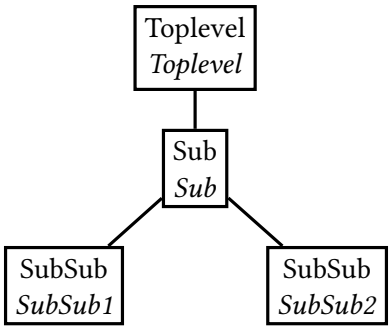
This requires two calls, as every abut/align function with alignment boxes only changes either the x- or the y-coordinate. However, it is anchor-agnostic and therefore does not require any anchors names for the alignment. This also allows writing code that just assembles blocks of layout cells without knowing any anchor names. This code then works for any cells that has an alignment box. This kind of code actually exists in the placement module, which is mainly intended for digital layout.

1.5 Cell Hierarchies

Layouts of integrated circuits usually make great use of repetition/reuse of cells. For instance, a shift register uses the same flip flop over and over again. Creating *flat* layouts (that is, layouts without any hierarchies) for these cells can be quite resource-intense. More shapes have to be

calculated by opc and the resulting layout is very likely to be larger in file size than a hierarchical one. Therefore, opc supports hierachical layouts. Instantiations in a cell of other cells/layouts are called *children*. They are light-weight handles to full cells and don't contain any flat shapes of their own. As example a layout that uses a sub-cell 1000 times it only has to store 1000 handles, but not 1000 versions of the same layout.[2] A layout that makes proper use of hierarchy can me multiple magnitudes faster in processing than a flat version.

Each child has a reference it points to, which is created automatically by adding a child to a cell. As an example, let's create the following hierarchy:



The upright name shows the name of the cell (which can be re-used), the italic name is the instance name (this has to be unique). In this example, the top-level cell instantiates another cell (sub), which in turn instantiates two other cells (subsub). In order to create this hierarchy in code, we have to do the following steps: First we create all cells (which are called *references*):

```
local toplevel = object.create("toplevel")
local sub = object.create("sub")
local subsub = object.create("subsub")
```

After that, we simply add the respective cells as children:

```
sub:add_child(subsub, "subsub1")
sub:add_child(subsub, "subsub2")
toplevel:add_child(sub, "sub")
```

It is important not to confuse the cell names here. All functions that create proper cells (objects) such as `pcell.create_layout` or `object.create` must be supplied with a name for that cell. Functions for adding children to objects (currently the object methods `add_child` and `add_child_array`) take an optional parameter for the instance name[3]. If this name is not given, an automatically generated name will be used.

We can see a simple example of proper instance naming in `analog/ringoscillator.lua`. Here, `string.format` is used to generate unique instance names for the individual inverters of a ring oscillator. First, the inverter reference (the actual cell) is created. This inverter is a CMOS structure with some additional wires, so the basic structure of the inverter is based on `basic/cmos`. For brevity, the additional drawings etc. are not shown here.

[2] More realistically, if the layout allows for that, an arrayed version of the sub-cell is stored.
[3] This instance name is not supported by all exports. For example, GDSII has no notion of an instance name

```
    gatecontactpos = invgatecontacts,
    pcontactpos = invactivecontacts,
    ncontactpos = invactivecontacts,
})
pcell.pop_overwrites("basic/cmos")
```

Then the inverter reference (which is only generated once) can be *instantiated* multiple times, which happens within a loop in this case:

```
for i = 1, _P.numinv do
    inverters[i] = oscillator:add_child(inverterref, string.format("
        inverter_%d", i))
    if i > 1 then
        inverters[i]:move_anchor("left", inverters[i - 1]:get_anchor(
            "right"))
    end
end
```

The above example creates a number of inverters (depending on the parameter `numinv`). To add a child, `add_child` is used, which expects a full object as reference and an instance name. Here, the single reference is instantiated multiple times, therefore the instance name is modified in every call. Additionally, the cells are left-right aligned to build a proper layout.

In the above ring oscillator example, the return value of `add_child` is stored in a table. The return value of `add_child` is a so-called *proxy object*. This proxy object behaves like a regular object that it can be moved, rotated and its anchors can be queried (as can be seen in the example). These operations only operate on small data structures and a very lightweight in general. It is usually a good idea to put repeated layout structures in sub-cells for them to be re-used, as this very likely leads to a smaller and faster-processed layout.

2 Cell Scripts

The previous section discussed the use of pcell definitions based on the functions `parameters` and `layout`. For re-used cells this is a good aproach, but some layouts are handled with more similarity to a stand-alone program. For this, *cell scripts* are also supported. For the main part, they function like proper cell files, but cellscripts just describe the content of the layout function of cells. This means that some parts are more manual, for instance the main object must be created and returned by the user. An example cell scripts could look like this:

```
local cell = object.create("toplevel")
geometry.rectangle(cell, generics.metal(1), 100, 100)
return cell
```

Cell scripts have the advantage that they don't have to be placed in some path known to opc. The layout-generation call to opc expects a (absolut or relativ to the calling path) path to the cell script, such as

```
opc --technology opc --export gds --cellscript path/to/cell.lua
```

3 Available PCells

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

3.1 Transistor

The transistor might be the most important cell and currently it's also definitely the most complex one. Therefore, 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

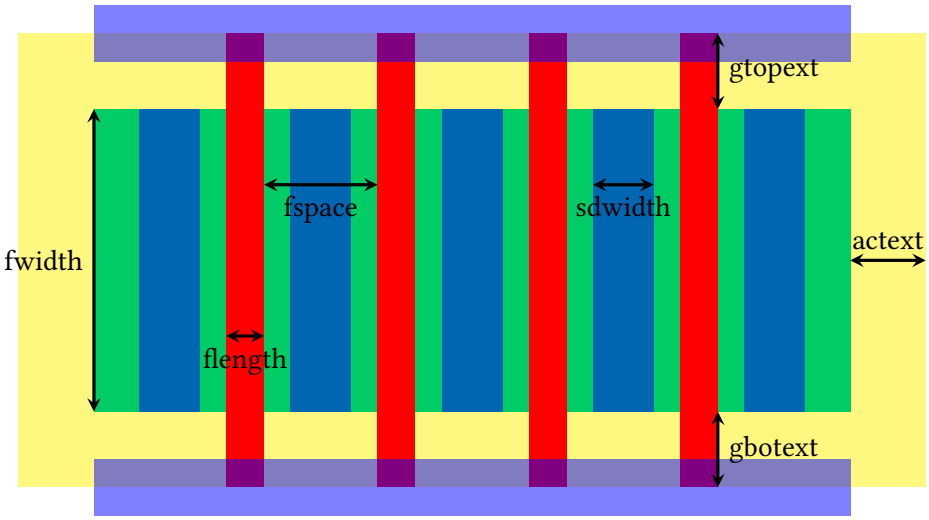


Figure 1: Overview of the transistor

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

4 API Documentation

The following section documents all available API functions for layout creation and manipulation. At the current time, it is far from complete. More up-to-date API documentation can be found by using opc directly with the command-line options `--api-search`, `--api-list` and `--api-help`.

4.1 geometry Module

`rectanglebltr(cell, layer, bl, tr)`

Create a rectangular shape defined by the bottom-left and the top-right corner.

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generic	Layer of the generated rectangle
`bl`	point	Bottom-left (bl) point of the rectangle
`tr`	point	Top-right (tl) point of the rectangle

4.2 Object Module

4.3 Shape Module

4.4 Pointarray Module

4.5 Point Module