# OpenPCells

A Technology-Indendent Layout Generator for Integrated Circuits

**Patrick Kurth**
**p.kurth@posteo.de**

2022-08-08

# Motivation

- Design re-use quite poor in analog design
- Still many repetitive tasks, especially in layout
- Designs usually subject to restrictive non-disclosure agreements

# Goals

- Enable design re-use
- Provide reference implementations of common circuits
- Port complex systems to other technology nodes
- Share circuits with other researchers/companies/...

## Difficulties in Layout Automation for Integrated Circuits

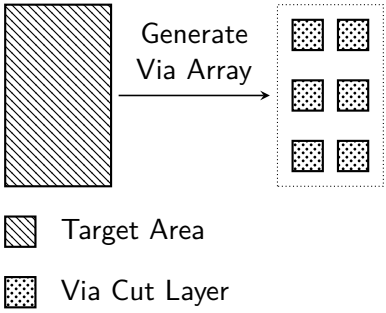
- Integrated circuits are complex
- Many, many, many different ways to do the same thing
- Even more design rules, especially in modern technology nodes

# Principles of Maintainable Circuit Layouts

1. Simplify the schematic
2. Simplify the schematic
3. Simplify the schematic
4. DRY in layout (don't repeat yourself)
   - Draw everything only once (instances, vias, etc.)
   - Make extensive use of symmetry, only draw half/quarter/... layouts
   - Find minimum common representation of sub-blocks (e.g. transistors without implants)
   - Hard to do and can result in very steep instance hierarchies
   - More special cases make this even harder

## Vias and Contacts

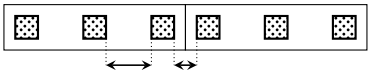
- Resolve via cuts with set of simple rules
- Fit via: $R = \left\lfloor \frac{W + S_{\min} - 2 \cdot E_{\min}}{C + S_{\min}} \right\rfloor$
- Continuous via: $R = \left\lfloor \frac{W + S_{\min} - 2 \cdot E_{\min}}{C + S_{\min}} \right\rfloor$



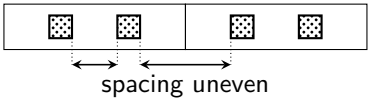
## Continuous Vias

- Simple cut fitting not suitable in some applications (for example merging/aligning guard rings)
- Specialized via translation ensures equal spacing

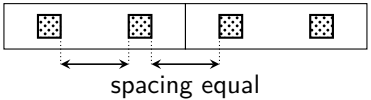
Fitted Via Array
(Enclosure $<$ Space / 2)




Fitted Via Array
(Enclosure $>$ Space / 2)




Continuous Via Array
(Enclosure = Space / 2)



## Basic Cell Code

```
function parameters()
    -- parameter definitions go here
end

function layout(cell, _P, env)
    -- layout descriptions go here
end

function config()
    -- special stuff
end
```

- Cells are described by defining public (global) functions
- Every cell *must* define `layout()`
- `parameters()` defines the parameters of the cell (and some minor other things)
- A cell without any parameters always represents the same layout
- In rare cases, `config()` is used, but this is only required for special cells

# Adding Parameters

```
function parameters()
    pcell.add_parameter("width", 100)
    pcell.add_parameter("height", 100)
end
```

or with a shorthand:

```
function parameters()
    pcell.add_parameters({
        { "width", 100 },
        { "height", 100 } -- <--- uses varargs function call, no
    })                    --      no comma after last argument
end
```

## Parameters – Allowed Values

- Per default, numeric parameters can take any values
- Allowed parameters can be limited to `even` or `odd` values, intervals (`interval`) or explicit sets (`set`)

```
function parameters()
    pcell.add_parameters({
        { "width", 100, posvals = even() },
        { "height", 100, posvals = interval(1, inf) },
        { "rep", 2, posvals = set(2, 4, 8, 16) }
    })
end
```

## The Layout Function

- First argument is the cell buing built (it is created by the calling code)
- The first argument is in theory optional, but a cell without a cell argument makes no sense
- Second (optional) argument is the table holding all parameters with their current value (default or user-given values)
- Third (optional) argument is the environment, similar to parameters but concerns values that are defined by cell higher up in the hierarchy

```
function layout(cell, _P, env)
end
```

## Adding Shapes

- Basic shapes are added with the `geometry` module
- All `geometry` functions take the cell as first argument
- The layer as second argument
- Shape parameters (for example width and height for a rectangle) as remaining parameters

```
function layout(cell)
    geometry.rectangle(cell, layer, 100, 100)
end
```

But what do we put into the layer?

## Digression: Generic Layers

- Shapes need to have layer information
- Can't put in specific technology layers (for example metal 1)
- Can't put in layers for the specific format (for example GDSII)
- Need a system to represent chip layers technology-independent and format-independent
- This is achieved by using *generic* layers

```
function layout(cell)
    geometry.rectangle(cell, generics.metal(1), 100, 100)
end
```

## Digression: Generic Layers

- Metals indexed by a number (1 being at the bottom of the metal stack, closest to the bulk wafer)
- Generic layers for gates, active regions, implants, threshold voltage markings, oxide thickness
- Generics layers are translated into specific technology-bound layers
- Mapping described in layermap file with simple syntax
- Layermap has to be created for every technology node

Layermap example with only one layer (gate):

```
return {
    gate = { layer = { gds = { layer = 6, purpose = 0 } } },
    -- more layers here
}
```

## Digression: Generic Layers

- The technology layermap can define various export types (output formats)
- Most important is currently gds (GDSII)

More complex layermap example with only one layer (gate):

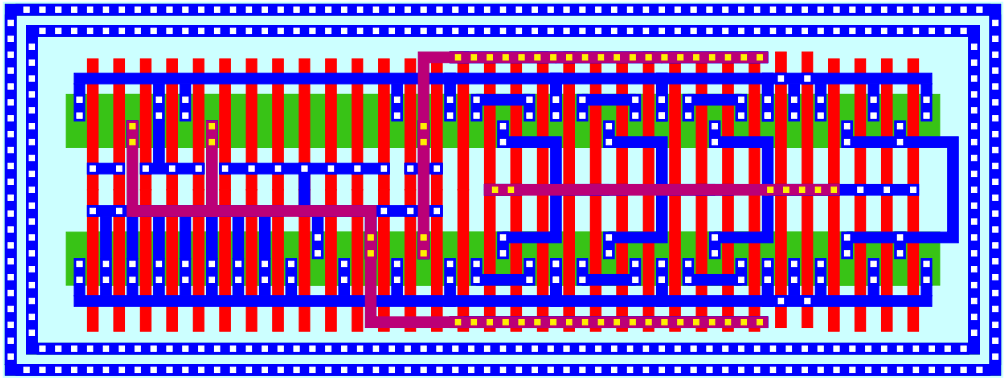
```
return {
    gate = {
        name = "poly",
        layer = {
            gds = { layer = 6, purpose = 0 },
            SKILL = { layer = "poly", purpose = "drawing" },
            svg = { style = "gate", order = 3, color = "ff0000" },
        }
    },
}
```

## Digression: Generic Layers

- Unused layers can be marked as empty
- For example, non-SOI (silicon-on-insulator) technology nodes don't need a buried-oxide break (soiopen)

```
return {
    soiopen = {}, -- will be skipped
}
```

# Current-Starved Ring Oscillator



# Grated Coupler