A Physical Design and Layout Versus Schematic Framework for Superconducting Electronics

by

Johannes Coetzee

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electric and Electronic) in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof. C.J Fourie

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2020/12/10 .....

Abstract

A Physical Design and Layout Versus Schematic Framework for Superconducting Electronics

J.A Coetzee

Department of Electric and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (EE)

March 2021

This dissertation presents a PCell synthesis and layout vs schematic extraction framework, named *SPiRA*. This framework allows the user to create a PCell-based layout, creating parameters to adjust polygon positions, sizes and presence. All polygons are connected to a specific l ayer i n t he fabrication process by means of a suggested Rule Deck Database containing process information. During the creation of a PCell, an undirected graph or *node graph*, showing all the interconnections present in the layout, is generated. Furthermore a *SPICE*-like netlist (a list containing information about the elements contained in the circuit and how element ports are connected) is generated by parsing this node network allowing the user to see if the extracted elements match up with the initial design. *SPiRA* is a Python framework, allowing for dynamicity in the creation of the layout, giving the user feedback along the way. Design rule checking (DRC) is implemented by means of different parameter types, allowing the user to get feedback during the creation of the layout about broken design rules. Further, full post-layout DRC is implemented by means of the KLayout DRC engine. As a futher extension of the framework, *SPiRA-tools* is introduced. This collection of tools allows the user to modify a layout, to prepare it for simulation by means of the InductEx simulation engine. *SPiRA-tools* also brings to life a schematic generator, that reads in a netlist file to produce a Standard Vector Graphics schematic, allowing the user to visually compare initial design, with the newly generated output allowing for true Layout vs Schematic comparison.

Opsomming

Die dissertasie bied 'n geparametriseerde sell (*PCell*) sintese raamwerk met *LVS* (*Layout vs Schematic)* funksionaliteit ingebou, genaamd *SPiRA*. Hierdie raamwerk gee die gebruiker die funksionaliteit om 'n *PCell* te maak, wat verstelbaarheid aan die posisie, grote en teenwoordigheid van enige veelhoek in die stroombaan gee. Vervaardigingsprosesreëls en prosesdata word in 'n Reëldatabase (*RDD*) gestoor, om hergebruik te word deur stroombaanontwerpers gedurende die uitleg van 'n geparameteriseerde sel. 'n Ongerigte node grafiek/netwerk word gegenereer wanneer 'n gaparametriseerde sel geïnstansieer word. Hierdie grafiek dui al die interkonneksies van die gegewe stroombaan aan. Hierdie netwerk word dan verder reduseer om 'n geskikte *netlist* (soortgelyk aan *SPICE*) te produseer wat werk met die simulasiesagteware, *InductEx*. Hierdie *netlist* kan met die oorspronklike ontwerp vergelyk word om te bepaal of al die konneksies en grote van die stroombaanelemente ooreenstem. Aangesien *SPiRA* op die skriptaal, Python, gebaseer is, kan dinamiese terugvoer vir die gebruiker gegee word tydens seluitleg. Ontwerpreëlkontrole is in plek gestel deur middel van gespesialiseerde *SPiRA* parameters, wat die gebruiker inkennis stel wanneer 'n ontwerpreël gebreek word. Verder het *SPiRA* die funksionaliteit om volle selontwerpe te kan analiseer met behulp van *KLayout* se losstaande ontwerpreëlkontrole sagteware. *SPiRA-tools* is 'n ekstensie van *SPiRA* wat streef om 'n *Standard Vector Graphic* lêer te produseer, wat die visuele voorstelling van 'n gegewe *netlist* is. Hierdie visuele voorstelling van die stroombaan kan dan direk met die oorspronklike ontwerp vergelyk word, vir ware *Layout vs Schematic* vergelykbaarheid.

Acknowledgements

First and foremost I would like to thank Prof Fourie and Dr. Delport for the help, motivation and wisdom when it comes to the field of superconductivity. I would like to thank the IARPA for the funding, making this research dissertation possible. To all my friends and family who stood by me these last two years, I thank you all dearly for every word of motivation and comfort during the hardships and struggles. A big thank you to all my research colleagues who have turned into the friends and always contributed to solving any problems involving *SPiRA*. Finally, I have to thank Dr. Ruben van Staden for planting the seeds and laying the foundation for *SPiRA* and guiding me.

Contents

Chapter 1

1. Introduction

Since the discovery of superconductivity in Mercury by Heike Kamerlingh Onnes in 1911, our understanding of the superconductor electronics (SCE) has increased dramatically and as we reach the end of the semi-conductor era of computing, new software and design methods are sought after to further increase our control over this physical phenomenon [3]. Process and design engineers have formulated different methods to synthesize digital superconductor circuits and formed both magnetic and physical design rules on how to appropriately design a circuit from the ground up to ensure the best possible margins for operation.

Today's SCE fabrication industry largely consists of 5 technology families [4] (AIST, Hypres, IPHT, D-Wave and MIT-Lincoln Lab). Even though the fabrication information for most of these processes is not available to the public, test results not including any parameters can still be shown.

As most of these processes are still in early stages of development, information is very limited to the public or lies behind expensive, closed source software/-documentation or non-disclosure arrangements. The Intelligence Advanced Research Projects Activity (IARPA) thus created the ColdFlux SuperTools project. The aim of this project is to develop comprehensive software tools to aid the designing of SCE circuits. This software suite should not only aid in the design of the circuit, but also supply comprehensive timing and inductance analysis [5]. The goal of this project is to create a full EDA (electronic design automation) suite with LVS (Layout vs. Schematic) functionality. LVS is considered a type of EDA, as a netlist can be generated and compared to the design autonomously from a given layout.

1.1.1 Core differences between SCE and CMOS

CMOS (Complimentary Metal Oxide Semiconductors) is a very established technology that is commonly used for every single piece of electronics on the market. CMOS today is almost exclusively developed around field-effect tran-

sistors (FETs). The idea behind performance gain for this technology type is purely based on the size scaling of junction areas. Area is decreased to fit more transistors onto the same package to simultaneously increase performance, while bringing down power consumption. However we are nearing the end of the physical limitations of just shrinking down junction size to increase performance. Not only is the rise in clock speeds slowing down, the rate of shrink and efficiency is decreasing. These problems gave light to the development of SCE and their respective design processes.

Superconducting electronics make use of a different junction model as well as different mesh and node equations. Instead of using a transistor as a switch or junction, a Josephson Junction is used. This junction is described by a different set of equations than a transistor, as resistance is non-existing if a metal is in it's superconductive state. Inductive elements form the core of SCE circuits and thus circuits are described by means of inductance loops as opposed to capacitance loops generally used in CMOS. Due to the aforementioned, most commercial-grade software's efforts are focused towards the design and development of CMOS-based ICs (integrated circuits), leaving a gap not only in the commercial software market, but also in OS (open-source) software solutions.

1.1.2 Current Layout vs Schematic solutions

The process of layout vs schematic entails the comparison of a generated undirected graph (or similar mathematical representations such as *undirected graphs*) from a desired layout to the theoretical design of the same circuit. Currently two main methods are used to determine if layouts are equal: Formal Equivalence Checking [6], where the logical operations of a circuit is compared to the desired response of a designed circuit and *graph isomorphism*, where two graphs are compared to one another to detect resemblances or *isomorphism* between vertices and edges.

Formal Equivalence Checking:

Equivalence checking has a wide range of definitions based on the level of abstraction at which the checking takes place. Most commonly, the desired clock-for-clock output of the given circuit is compared to the simulated output of the design. Alternatively, on a lower level, the execution of instructions from CIS (CPU instruction set) can be compared to ensure sequential execution of instructions are the same.

Software such as qEC has been created to try and apply the principles of FEC (Formal Equivalence Checking) to SFQ (Single-Flux Quantum), a popular superconducting technology branch [7]. This tool is built on the CMOS

logical synthesis tool, ABC [8]. This tool however is only verified for the Sport lab SFQ logic circuit benchmark suite of cells and does not support other technology types such as Hypres or AIST.

However, the Python3 framework, *SPiRA*[4][9][10], takes another approach, namely graph isomorphism. This framework is built to generate a graph that can be used during LVS checking.

Graph Isomorphism:

Graph Isomorphism is the mathematical process of comparing two undirected or directed graph's edges, vertices and labels to determine if they are equivalent. When two graphs are isomorphic, it can be denoted as $G(f) \cong H(f)$. The figure below shows how two graphs can be isomorphic despite how they are drawn.

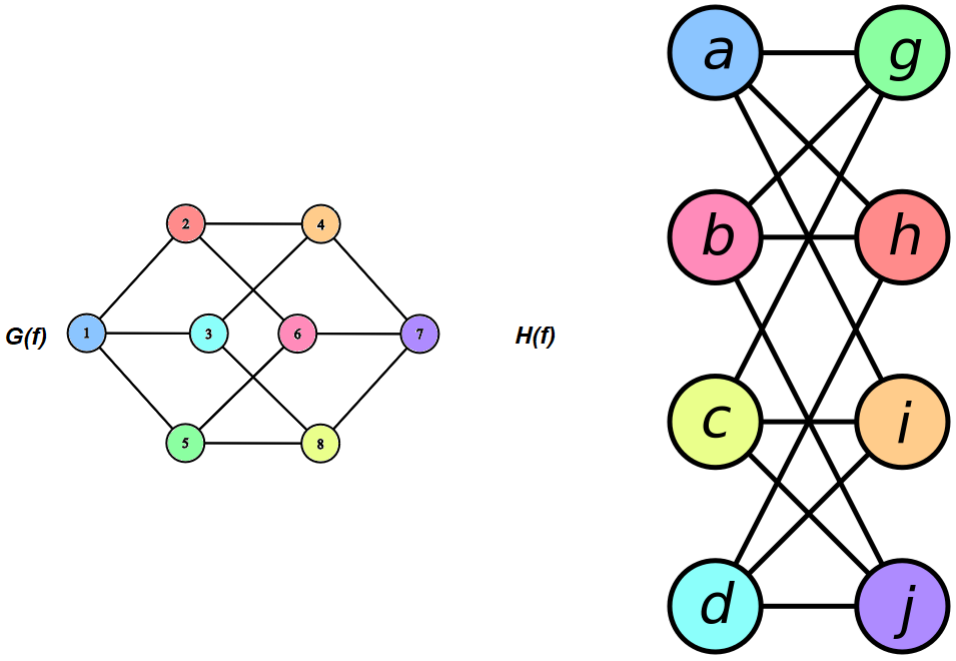


Figure 1.1 Graphs $G(f)$ and $H(f)$

The idea is to establish an undirected graph showing all the connections contained in a layout. This is done by meshing an entire layout by means of software such as Gmsh. Some restriction algorithms and graph reduction methods are used to generate an undirectional node graph showing where connections are made between metal layers. These connections are then tested against the theoretical or designed layout to detect isomorphism between the aforemen-

tioned. In this case each function should return the same circuit element for the same value of f (node position, number or label).

Although *SPiRA* does not directly apply graph isomorphism, the general idea of forming a network containing the information contained in the circuit and filtering out non-essential connections to produce a circuit netlist that can be directly compared to the original design.

1.1.3 Available Design Software for SCE

A few open source software solutions, such as KLayout and Magic do provide some form of LVS, however all these efforts are focused towards semiconductor design standards and do require some form of user intervention during the layout phase of the design or compares a user-supplied netlist to ensure the circuit design coheres with it.

Xic is almost exclusively the only open-source layout and editor software that focuses on superconducting electronics. In the next chapter *Xic* is thoroughly discussed and attention is given to cell design and the software's short-comings.

This aforementioned short-comings gave rise to the synthesis and LVS framework, *SPiRA* [4]. At the core of *SPiRA* lies Python3, which allows for rapid expansion and allows for the extensive variety of libraries to be used and for tight integration with the Python language itself. In later chapters, explanations are given on how the *SPiRA* core was used and expanded to create an LVS cycle from parameterized cell (PCell) principles.

1.1.3 SCE Synthesis Process

The physical synthesis of SCE is a multilayer process of different elements. Currently most fabrication processes are Niobium based and deposited on a Silicone substrate. Silicone Dioxide (SiO_2) is used to fill the layers and create interconnections between inductive or resistive layers. Each layer serves a purpose and has its on permeability, London penetration depth, thickness and design rules. These layers (thin film) are deposited by using methods such as low-pressure metal organic chemical vapor deposition (CVD) [11]. Each process is also designed for a specific current density (usually in $\mu A/\mu m^2$).

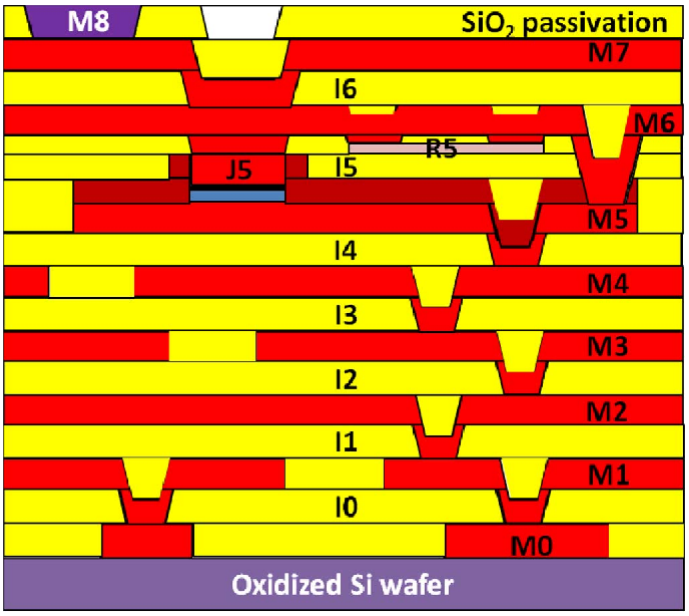


Figure 1.2: 8-Layer process designed by MITLL [12]

Every aspect of the process data needs to be taken into account when working on an LVS solution as information about connections between layers, layer type (inductive, resistive, etc.) and material types are required to create an accurate representation of the circuit, by means of a netlist or a graph. This problem gave rise to the idea of a PDK (process design kit)

2. Process Design Kit

A Process Design Kit should be an ensemble of technology files, such as layer definitions, in which all the information about the process should be present. These files should be used by the SuperTools tool-chain [5] during the design and simulation of SCE circuits. The PDK should be a toolkit containing information such as example cells, calibrated inductance definitions files, design rules and layer definitions which should allow someone with no prior knowledge of the fabrication process to design circuits adhering to all the required rules. The format for a superconducting technology PDK is still undecided; because of this *SPiRA* makes use of it's own PDK format, namely a Rule Deck Database (RDD). This database not only contains the layer map used by the desired process, it also contains metadata about the process, such as maximum polygon thickness allowed on each layer and the type of layer (metal, resistive, connection layer, interconnection layer, etc.). Figure 1.3 below shows a simple example of how the *SPiRA* RDD is imported and simple layer rule definitions are given.

```
from spira.yevon.process.all import *
from spira.yevon.process import get_rule_deck

RDD = get_rule_deck()

RDD.M0 = ParameterDatabase()
RDD.M0.MIN_SIZE = 0.5
RDD.M0.MAX_WIDTH = 20.0
RDD.M0.LAYER = 0.0
RDD.M0.MIN_DENSITY = 25.0
RDD.M0.MAX_DENSITY = 55.0

```

Figure 1.3: Extract from a sample RDD to show Parameters

Objective of Dissertation

This dissertation proposes an integrated software tool-chain, that will allow for the synthesis and rule-checking of SC integrated circuits. At the core of this software stack lies *SPiRA* and the standalone KLayout DRC engine. The *SPiRA* framework has been modified and expanded to bring to life an electrical netlist that is generated by means of an hierarchical algorithm. This means an InductEx compatible netlist can be generated along with a circuit file (in the GDSII format) that has been modified with the correct ports and junction labeling needed for inductance extraction by means of the InductEx engine. A modified PDK format is suggested and makes use of all the GDSII file-type features, such as hierarchy, different layers and datatypes. This industry standard circuit file is commonly supported by Computer Aided Design (CAD) software such as *Xic*, Layout Editor and KLayout and will thus make it easier to implement different types of fabrication processes, or make changes to existing ones within the entire SuperTools tool-chain. As a final step and extension of the *SPiRA* core, *SPiRA-tools* is created. This piece of software is meant to bridge the gap between InductEx and *SPiRA*. This extension also includes a circuit schematic generator to create a visual representation of the extracted netlist.

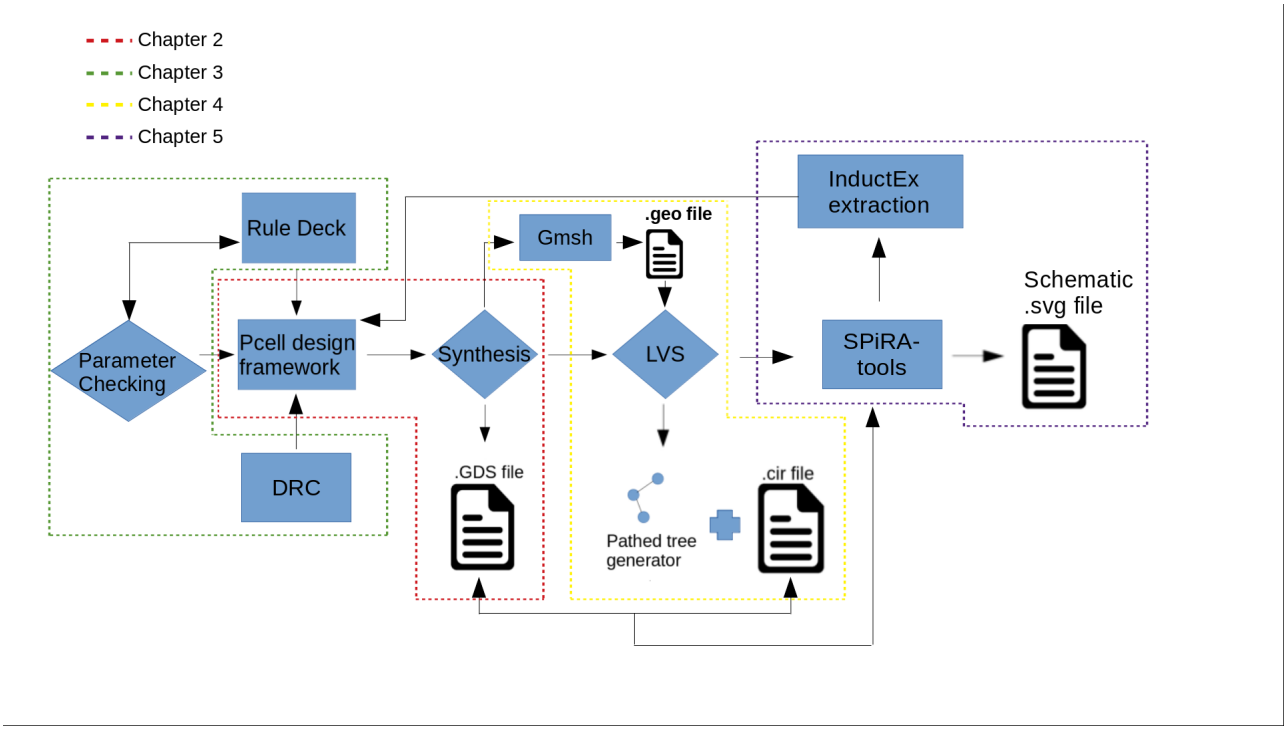


Figure 1.4: SPiRA LVS cycle

Chapter 2. Understanding Parameterized Cell (PCell) Design

2.1 History of PCells

The first concept of a parameterized cell was developed by Cadence, in their popular industry layout editor, Virtuoso [13]. The idea behind a parameterized cell, is to be able to design a cell layout once, then draw the geometry, labels, paths and ports based on the given parameter inputs. Attributes such as track width, resistor and inductor sizes can be changes by merely calling the PCell with the correct parameters, changing the layout dynamically to accommodate the new parameters. The scripting language, Skill, was used in Virtuoso and it remains an industry golden standard. However the Skill scripting language is not supported outside of the Cadence EDA suite and generally is not directly portable to other tools. This problem lead to the development of *Xic*, by Stephen R. Whiteley.

The idea was to make use of the widely used Python scripting language to implement PCell design into *Xic* and be able to export and use the PCells in other tools, by means of the OpenAccess plug-in, written for Python. This includes open-source software solutions such as PyCell or PyCell Studio.

Since the release of *Xic*, not much progress has been made to other open-source solutions. For this reason the ColdFlux project managers agreed to make use of *Xic* as the standard cell editor as it is the only PCell editor package that focuses on SCE design and IC layouts. *Xic* however did not age well due to lack of optimization for ever-expanding layout sizes and the libraries used by *Xic* are not all constantly updated, making the installation on modern systems or operating systems daunting.

These short-comings inspired the idea of an easily maintainable PCell framework that made use of modern environments such as Ruby and Python3. *SPiRA* aims to create a simple scripting standard that can be used to create PCells and generating full circuit layouts.

2.2 Equation Based Cell Design

Unlike CMOS or semi-conductor circuit design, superconducting electronic are design with the inductor phase-based circuit equation, rather than loop or node voltage-based design.

The current and voltage inside of a Joshpson junction can be described by the following diagram and equation:

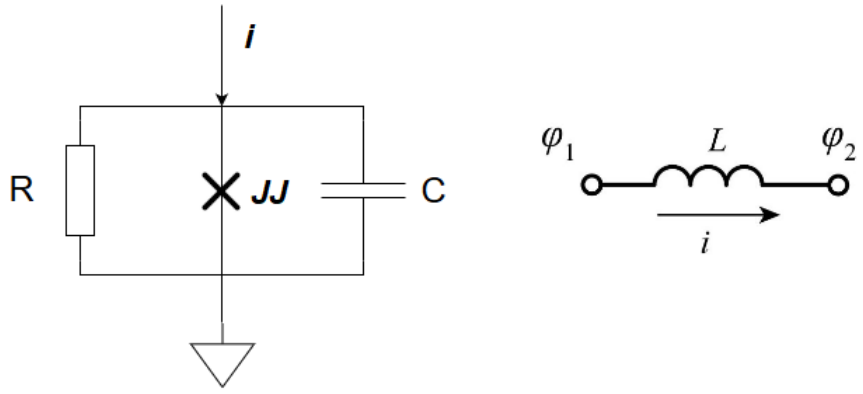


Figure 2.1 Josephson junction equivalent circuit

$$i = I_c sin\phi + \frac{v}{R} + C\frac{dv}{dt} \qquad Eqn.\ (1)$$

$$v = \frac{\Phi_0}{2\pi}\frac{d\phi}{dt} \qquad Eqn.\ (2)$$

And from Faraday's law, the following voltage equation can be deduced in differential form.

$$v = L\frac{di}{dt} \qquad Eqn.\ (3)$$

By combining these 3 equation, the current going through an inductor can be determined as a function of it's phase:

$$\Phi_L = \frac{2\pi L}{\Phi_0} i \quad OR \quad i = \frac{\Phi_0}{2\pi}\frac{\phi_1 - \phi_2}{L} \qquad Eqn.\ (4)$$

Generally the whole circuit is described by the phase-base equations shown. This is required to calculate all the unknown inductances and currents, as it will translate to physical changes in the layout.

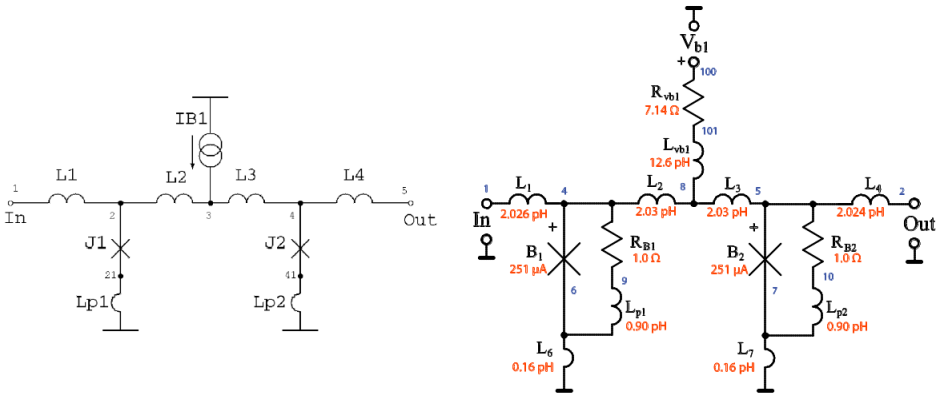


Figure 2.2 JTL schematic (left), with calculated parameters (right)

These equation form the base of superconducting analogue electronics design and a combination of these equations parameters (Φ, v, L, etc) are used to create the desired frequency response, critical switching current for each Josephson junction (JJ) and timing response. Gates are clocked in a sequence and of careful thought is given to route pathing, routing distancing and cell placement. Once the theoretical parameters are calculated, the physical geometry can be determined based on the fabrication process at use.

2.3 Converting circuit equations to geometry

To start creating a geometrical representation of a circuit, one generally needs an idea of the inductor and resistor physical sizes. The figure and table below shows how the inductance varies with the size of a physical polygon. For testing purposes, a micro stripline has been placed on a metal layer. Two simulation sets were created: one where the stripline had a ground plane and a second, where both a groundplane and skyplane was added. This was done as fabrication processes such as the MIT-LL process contains both a sky and groundplane. The height of the inductor was kept at 0.5 μm^2 as it is a good nominal width for inductors in practice.

Width (μm)	Height (μm)	Simulated Inductance (pH)	pH/(μm^2)
1.25	0.5	0.675353	1.0805648
2.5	0.5	1.43642	1.149136
5	0.5	2.95889	1.183556
10	0.5	5.6495	1.299

Table 1: Stripline simulation containing only a groundplane

Width (μm)	Height (μm)	Simulated Inductance (pH)	pH/(μm^2)
1.25	0.5	0.654223	1.0467568
2.5	0.5	1.3747	1.09976
5	0.5	2.80703	1.122812
10	0.5	5.70088	1.140176

Table 2: Stripline simulations with skyplane and groundplane

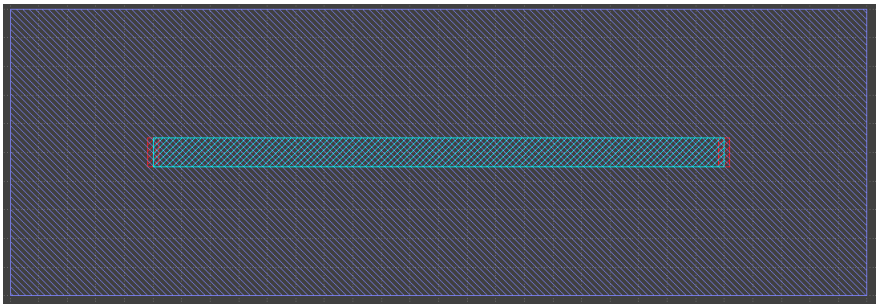


Figure 2.3: Stripline placed on a metal layer with a ground and skyplane

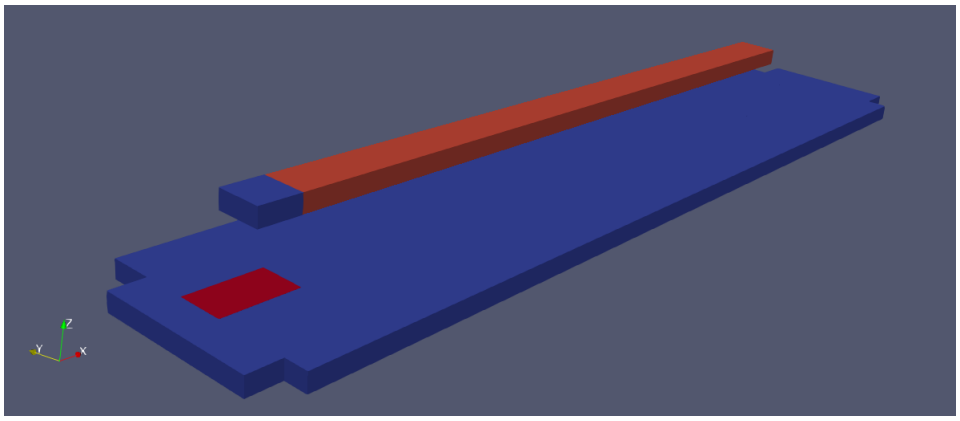


Figure 2.4 3D representation of stripline with groundplane

The results above show a trend of growth in inductance per unit length, as the width of the inductor grows. This growth is non-linear and for this reason, in practice, its best for a circuit designer to use a mean value for inductance

and resistance per unit length, when first doing the circuit layout. Changing resistor values can prove daunting, as in SCE, resistance is modeled as a purely parasitic element [14]. An iterative simulation process thus has to be done, making changes to physical layout after every simulation till the actual simulated parameter values (R, L) best fit the designed values. This tedious process can be circumvented converting the layout to a parameterized layout or PCell.

Once the base geometry for the device has been laid out, a PCell-based design can be created to increase re-usability and allow for easy tweaking of geometrical parameters without redesigning or adjusting the entire layout.

2.4 Converting geometry to PCell design

2.4.1 GDSII file format overview

The GDS or Graphic Design System file format dates back to 80's, when programming was done by means of tapes. This binary file format is very compact, yet has the flexibility to contain almost all the data needed to fabricate an integrated circuit. This file format is an industry standard [15] in circuit design by now and is widely used and supported by most layout editing tools.

The GDS file format contains a layer table which is connected to datatypes. Each layer can have a multitude of different datatypes that can be used to store metadata about the layer as well as the physical geometry needed to construct the circuit. This very robust database-like file format serves as a perfect data structure to store information about a given PCell.

The following figure demonstrates how a resistor (containing the appropriate vias or breach from one metal layer to another) PCell has been generated, making use of different datatypes on the same layer. The different datatypes are used to store edge ports and other information which will be used during routing and netlist extraction.

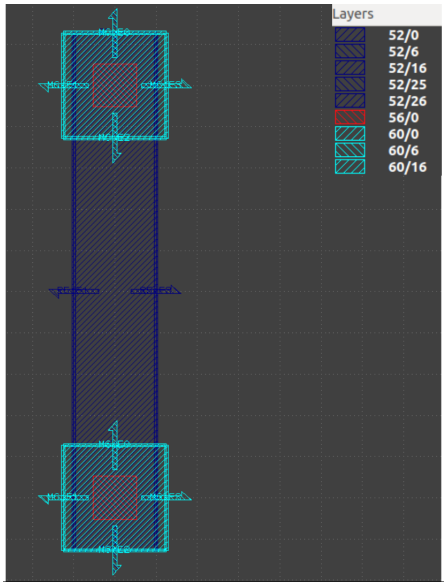


Figure 2.5: Generated PCell showing a resistor connecting two vias

2.4.2 Using *SPiRA* to create PCell layouts

Once the basic geometry has been created in a CAD or scripted environment of choice, a PCell version of the cell can be created. Generally inductor width and length, resistor sizes and junction areas are important parameters within the cell. It is thus intuitive to create these as settable parameters within the PCell. The figures and code extracts below shows how a part of the Process Design Kit, or in this case, the Rule Deck Database is used to generate a PCell containing two vias connected by a resistor (as shown in figure 2.5). The parameters chosen are shown and example outputs containing different parameters are generated.

First the SPiRA framework and desired Rule Deck Database is imported

```
import spira.all as spira
from spira.technologies.mit.process.database import RDD
```

Now construct the geometry for the via device that will be used for the design. In this case the via is used to connect metal layer 6 (M6 in the RDD) to the resistive layer R5.

```
class via(spira.PCell):
    # The name of the cell is 'via'
    __name_prefix__ = "via"

    c5r_length = spira.NumberParameter(
                 default=0.1,
                 restriction=spira.RestrictRange(lower=0,
                 upper=10), doc='C5R length')
```

```
    def __create_elements__(self, elems):
        elems += spira.Box(layer=spira.RDD.PLAYER.M6.METAL,
                 width=1.25,height=1.275,center=(0,0))
        elems += spira.Box(layer = spira.RDD.PLAYER.C5R.VIA,
                 width=self.c5r_length, height=self.c5r_length,
                 center=(0,0))

        return elems
```

In this case one of the many different types of parameters that spira can handle is created, namely a *NumberParameter* with a range restriction set. This tells the *SPiRA* core that this parameter has to be of type float/integer and in a range of 0 to 10. When the parameter is incorrectly, the user will receive an error explaining what violation has been made. Restrictions and design rule checking is fully discussed in chapter 3.

The class 'via' now accepts a C5R length parameter that can be changed every time the via class in initialized. Figure 2.6 shows how the via was generated with different parameter values set.

```
v1 = via(c5r_length = 0.2)
v2 = via(c5r_length = 0.4)
v3 = via(c5r_length = 0.8)
```

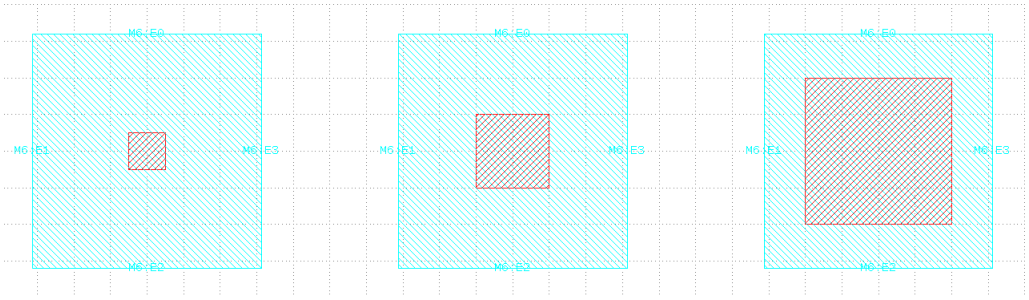


Figure 2.6: Vias with different layer C5R side lengths.

The size of the C5R layer is important, as it needs to change according to the resistor width which will be used for the layout. Two instances of this via will now be instantiated and the structures will be rooted to one another by means of the R5 metal layer. To do so, a main cell or *top level cell* will be created and a reference of the vias will be called and placed inside this cell. The new top cell will contain four new parameters: the width and height of the resistor and the two vias referenced.

```
1 class via_res(spira.Device):
2     __name_prefix__ = 'ib'
3
4     viatop = spira.Parameter(fdef_name = "create_via_top")
5     viabot = spira.Parameter(fdef_name = "create_via_bot")
6
```

```
7
8     height = spira.NumberParameter(default=RDD.R5.MIN_SIZE,
9               restriction=spira.RestrictRange(lower=0,upper=10),
                doc='height of the shunt resistance.RULE 52.1')
10    width = spira.NumberParameter(default = RDD.R5.MIN_SIZE,
11            restriction = spira.RestrictRange(lower=0.1,upper=
12            1.25),
13            doc = 'width of shunt resistor, RULE 52.1')
14
15    def create_ic_top(self):
16        v = via(c5r_length = 0.52)
17        return spira.SRef(reference = v, midpoint =
18        (0,self.height))
19
20    def create_ic_bot(self:
21        v = via(c5r_length = 0.52)
22        return spira.SRef(reference = v, midpoint = (0,0))
```

Now that the two via parameters, height and width parameter has been created, the vias need to be initialized. To do so, one makes use of the *create_structures* function that is inherited by the *spira.Cell* class. This tells the *SPiRA* core that these structures must be placed within the main cell.

```
1     def create_structures(self,elems):
2         elems += self.viatop
3         elems += self.viabot
4         return elems
```

Upon viewing the output, a port list can be printed. The list of ports is of cardinal importance for PCell generation as it is the key to making a layout change dynamically when parameters are varied. Ports can be defined as connection points or a point where an element in a circuit connects to another. In this case it will be the points on the vias to which the resistor will be routed. This will ensure that the resistor will always have the correct length and height, despite of any changes being made to the via locations.The ports of a structure can be accesses by calling the *.ports* attribute of an object.

```
$ python Via_res_Pcell.py

[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
-----------------------------------------------
(0, [SPiRA: Port 'E0'] (name M6:E0, midpoint
   (0.0,9.612499999999999) orientation 90.0 width 1.25, process
    M6, purpose EdgePort))
(1, [SPiRA: Port 'E1'] (name M6:E1, midpoint (-0.625,8.975)
   orientation 180.0 width 1.275, process M6, purpose EdgePort)
   )
(2, [SPiRA: Port 'E2'] (name M6:E2, midpoint (0.0,8.3375)
   orientation 270.0 width 1.25, process M6, purpose EdgePort))
(3, [SPiRA: Port 'E3'] (name M6:E3, midpoint (0.625,8.975)
   orientation 0.0 width 1.275, process M6, purpose EdgePort))
```

Each port contains a set of information. The name of the object, layer on which resides and its midpoint is all encapsulated in the object information. Edge ports will automatically be generated for any polygon in a PCell that is drawn on a metal layer (different port types and their uses is discussed in chapter 3 along with Design Rule Checking, for which ports are essential). This is done to more easily allow for routing. A naming scheme has been put in place to assign a corresponding port type to a port name. The port name will always be the metal layer on which it resides, in this case M6, along with the generated name of that specific port. A port can be either referenced by means of its name, or the number which is assigned to the port in the port list. Taking a closer look at figure 2.6 will show the aforementioned:

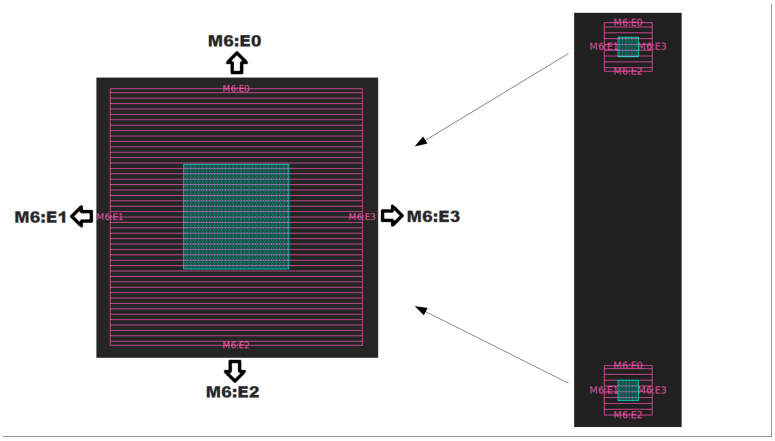


Figure 2.7: Two vias placed, highlighting the port names to be used

Due to the coherence in the naming scheme mentioned previous, it's trivial to determine that the port **M6:E0** of the top via will need to connect to **M6:E2** of the bottom via. This is done using the *create_routing* function of the *cell* class. SPiRA contains different routing algorithms that can be made use of during cell design. The *RouteStraight()* function can be used in the case where the two elements that need to be routed are on the same x-or-y-value. *RouteManhattan()* will create a path from the starting port to the destination port, by only making use of right angle turns.

```
    def create_routes(self,elems):
        elems += spira.RouteManhattan(ports=
        [self.topvia.ports['M6:E0'],
        self.botvia.ports['M6:E2']],
        layer = spira.RDD.PLAYER.R5.METAL,
        width = self.width)
```

It's also important to note that the *self.width* parameter is now used to determine the thickness of the R5 route. In the case of the *RouteStraight* function, the width of the M6 edge port will be used (**M6:E0,M6:E2**) to determine the width of the resistor. Both routing typologies can be used, with the correct adjustments to the elements of the PCell.

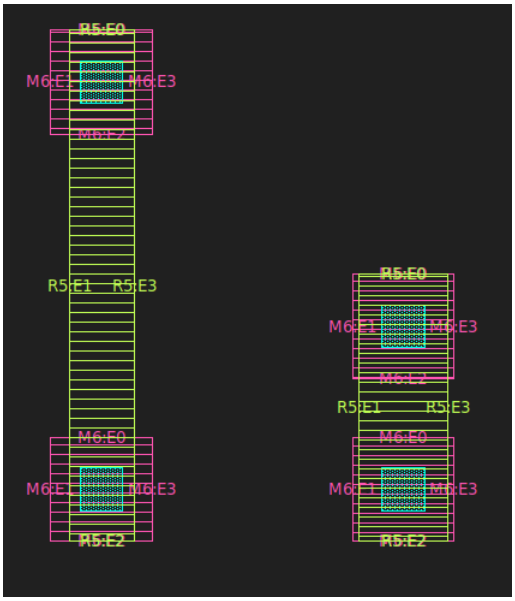


Figure 2.8: Two instances of the *via_res* class with different parameters

```
device1 = via_res(height = 5,width = 0.8)
device2 = via_res(height = 2,width = 1.1)
```

The layout will thus adapt and change as the parameters are changed within in set limits. A cell designer can thus design a full cell of which the cell user does not have to know the intricacies of the design. Instead the user can just change the parameters without being able to exceed the premeditated design limits.

The methods used in the simple example given can be expanded to create not only PCell connections, but junctions, groudplane patterns or full logic gates can be parameterize to allow for re-usability and testing. The advantages of having a purely dynamic cell layout is apparent in many ways. Large layouts containing many reference cell, labels and ports can be regenerated to adapt to the use of the layout, allowing for more expansive IC design. A larger PCell example is given in Appendix E, where a full junction has been parameterized.

Chapter 3: Design Rule Checking

DRC or Design Rule Checks is usually a collection of geometrical tests applies to the physical layout of an integrated circuit or chip to determine of the layout adheres to all the rules determined by the fabrication at use. These rules are generally set in place to ensure the best possible margins for operation after synthesis. These rules account for the variability that occurs in any fabrication process, due to the physical properties of the materials used as well as the physical synthesis process. This means that if a layout fails a design rule check, the chance of that circuit functioning as intended without noise, unwanted magnetic coupling or correct junction switching, decreases.

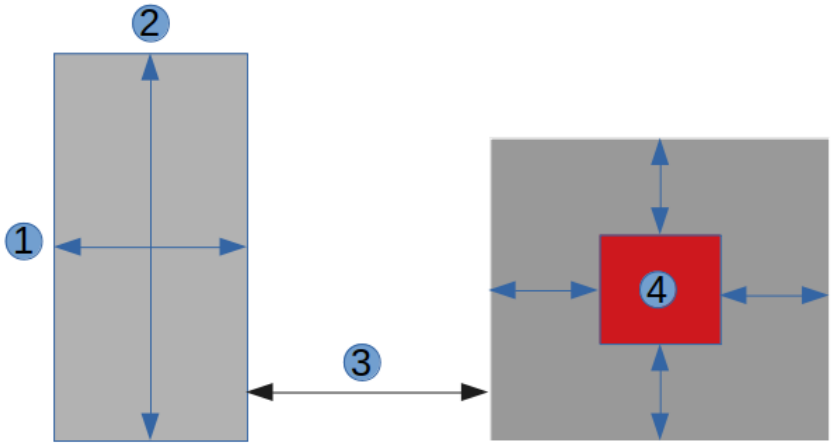


Figure 3.1 Shapes displaying basic DRC rules.
1 - Minimum inner edge distance. 2 - Maximum inner edge instance. 3 - Minimum distance between outside edges or 2 polygons. 4 - Required overhang between two layers.

The figure above shows a few basic design rules that are present in almost every fabrication process. This chapter discusses how *SPiRA* implements DRC by means of it's Rule Deck Database (RDD), different parameter types and extension to KLayout's standalone DRC engine in order to create an output containing all the polygons subject to some sort of design violation.

3.1 Rule Deck Database

As briefly mentioned in the introduction of this dissertation, a process design kit (PDK) is suppose to be a collection of cells, design rules and any other relevant information which can be used by a circuit designer to design a large or more complex circuit. For semiconductor technology, this is a very mature and extensive process used by all large synthesis process designers, however no thorough open-source PDKs are available to the public. *SPiRA* was thus designed with it's own representation of a PDK, namely a rule-deck database. This is a script-like database implementation the connects designed polygons to a specific process data. This is done by means of layer-and-process mapping. It contains all the needed design rules for PCell and different port types that are used during extraction for a specific process. This means that Rule Decks for different processes can be used interchangeably, depending on the fabrication process being used. This allows a cell designer to swap between fabrication processes (Hypres, MITLL, etc), without knowing the exact rules and nuances set in place for each specific process.

The RDD implements a new class, *ParameterDatabase* to store information such as design rules. A parameter database can be connected to another set of parameters, called a *PhysicalLayerDatabase*. This is more thoroughly explained by the figure below.

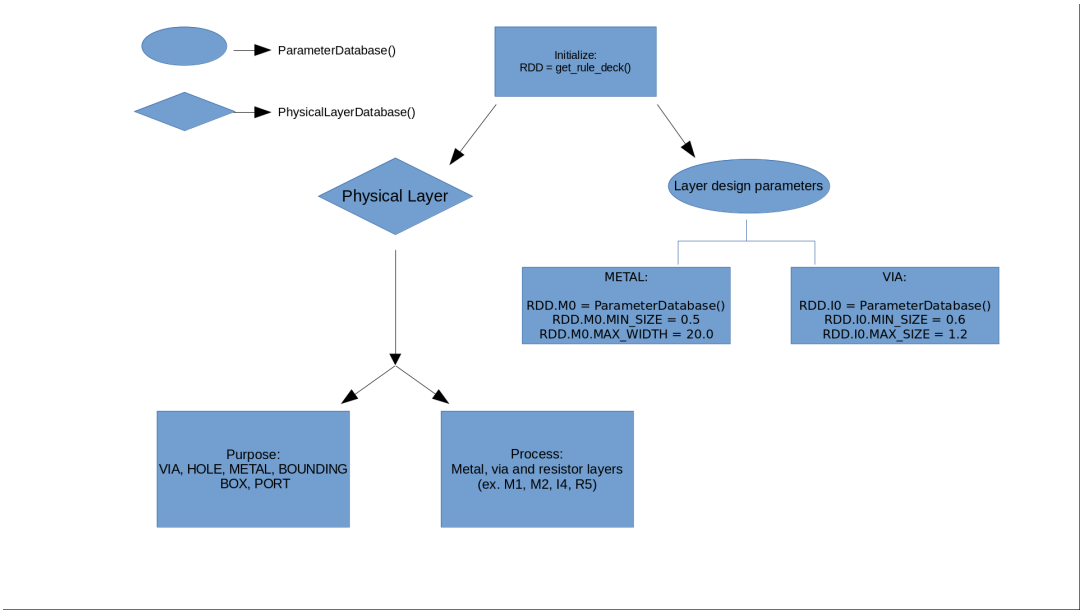


Figure 3.2: Rule Deck Database Relations

The goal of the parameter database is to have one for each layer in the process. This should contain all the basic design rules restrictions for a cell designer to use during PCell creation. The purpose of the physical layer databases are to

connect each layer to a process and gives it a purpose. Each polygon created in any PCell will need to be assigned a process and purpose. This tells the *SPiRA* core what GDS layers and datatypes to use during writing as well as contains some information used during LVS. The layer will be determine by the process to which the layer is connected and the datatype will be assigned based on the purpose of that specific polygon.

```
1     def create_elements(self,elems):
2         new_layer = spira.RDD.PLAYER.M5.METAL
3         polygon = spira.Rectangle(layer = new_layer, p1 = (0,0)
                  ,p2 = (1,1))
4         elems += polygon
5         return elems
```

The code extract above again shows the *create_ elements* function in a PCell. This time a layer is created and assigned to a rectangle polygon that is placed in the layout. The following syntax will always be used in assigning a polygon to a layer or creating a new layer:

DatabaseName.PhysicalLayer.LayerName.Purpose

The format of the RDD is very important when it comes to creating robust PCells. Understanding the workings of the database is cardinal to creation of sensible parameters. The following section will show how the RDD is used during parameters initialization to ensure that design rules are adhered to. It is thus intuitive that these rule decks should be able to be swapped out, depending on the desired synthesis process.

3.2 Integrated *SPiRA* parameters

SPiRA contains a multitude of parameters that one can use to describe PCells or use already written cells to create a large layout or IC. These parameters were briefly encountered in chapter 2 and show how any function inside of a *spira.PCell* can be initialized as a parameter in the layout, however *SPiRA* contains set parameter types as well. The most important ones are listed below as they have the most common use cases:

- Integer
- Float
- Complex
- Number range
- String
- Boolean

One of the reasons why it is sensible to use Python at the core of *SPiRA*, is to allow dynamic design rule checking as opposed to static/post-layout rule checking [4]. The scripting nature of the language is what makes the use of the aforementioned parameters so powerful. *SPiRA* will immediately notify the user that a parameter rule has been violated, before a full layout is generated. Taking a look again at the width parameter used during the PCell generation in chapter 2, a number parameters was used to define the width of the resistor. This type of parameter allows any number format (integer, float, complex number). The minimum and maximum values allowed is read from the RDD. Whenever an attempt is made to initialize the PCell (either when writing to GDS, or using it as a subcell in a different circuit) with incorrect values, execution is immediately halted and a value error will be raised notifying the user that a parameter was initialized illegally, as well as which parameter is incorrect and the cell it is contained in.

```
width = spira.FloatParameter(default = 0.2,
        lower = RDD.R5.MIN_SIZE,
        upper = RDD.R5.MAX_WIDTH)
```

```
$python  Via_res_Pcell.py

[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
-----------------------------------------------
ValueError: Invalid parameter assignment 'width' of cell
'via_res' with value '10.0', which is not compatible with
'( Type Restriction: int, float, int32, int64, float and Range
Restriction: [0.5, 5.0) )'.
```

3.3 KLayout Design Rule Checking

As briefly mentioned in chapter 1, KLayout is a simple, yet powerful open-source CAD tool for circuit design [16]. Although the initial intent of the this tool was for creating masks and IC layouts for semiconductor technology, it's robust editing features, along with the integrated Ruby-based DRC scripting extension, makes it usable for SCE implicitly.

Even though the previous sections mention the dynamic rule-checking capabilities of the SPiRA core, full post-layout DRC is still an essential part of the chip check-off process. For this reason, KLayout's DRC functionality has been bridged with *SPiRA*. An essential part of making this possible, is the standalone CLI (Command Line Interface) engine in KLayout. This allows commands to be sent from Python directly to KLayout's DRC engine. All the essential checks mentioned in figured 3.1 and much more is directly included in KLayout.

3.3.1 Using KLayout DRC engine

To do this, scripts have been created and added into the *SPiRA* core. These scripts will be edited as the layout in generated, making the required changes to suite the specific layout and then passed to the CLI of KLayout.

```
input_layer_m0  =input(1,   0).clean,
rule = input_layer_m0.width(0.6)
output(rule, "Min size violation on m0")
```

The code extract shows how a layer and rule is created in the DRC engine. This rule will check the inside edges of all polygons on the specified layer and compare it to the widths value specified. This is done for all the layers in the specific synthesis process at use as well as for different rules. The parameters can be adjusted according to each layer-specific rule. Minimum size and spacing is checked as well as maximum width and overhang between specific layers.

```
drc.width_drc.max_width_drc('layout.GDS')
drc.overlap_drc.no_overlap_drc('layout.GDS')
drc.size_drc.min_size_drc('layout.GDS')
drc.spacing_drc.min_spacing_drc('layout.GDS')
```

```
$python drctests.py

[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
-----------------------------------------------
Maximum width DRC tests are being run...
Overhang/Overlap DRC tests are being run...
Minimum size DRC tests are being run...
Minimum spacing DRC tests are being run...
```

KLayout makes use of a graphic user interface called a Marker Database to view the results of the tests, however for larger layouts, it can be very tedious to go through the list of faulty polygons. Luckily an XML version of the Marker Database can be stored for later use, by means of a file with the extension 'lydb'. This functionality allowed for the creation of a Marker Database parser. This parser is built into *SPiRA* and can automatically be ran after the generation of the result Marker Database. The XML file will be read in by *SPiRA* and the GDS synthesis functions are used to create a masked version of the layout, containing all the faulty polygons placed over the original layout. Two dictionary parameters are created, one containing the GDS layer-mapping and the other, the datatype-mapping. These datatypes were selected as they don't clash with already used datatypes by *SPiRA* or the synthesis process currently used. These datatypes are also not currently used by InductEx [17], which is more extensively explained in chapter 4 and 5.

```
errordict = {
            "Min size violation": "105",
            "Max width violation": "106",
            "Min spacing violation": "107",
```

```
            "Min overhang violation": "108",
            "No overlap violation": "109"
            }
datatype_map = spira.DictParameter(local_name = errordict)
```

Once the parser has been executed, the working directory will now contain the edited/generated scripts with the .lydrc extension. The names of the files will be the drc rule along with the _generated extension, along with a new GDS file with a viewer specified name. The script files are provided so the user can see which tests were actually run and if the parameters used were correct compared to the RDD.

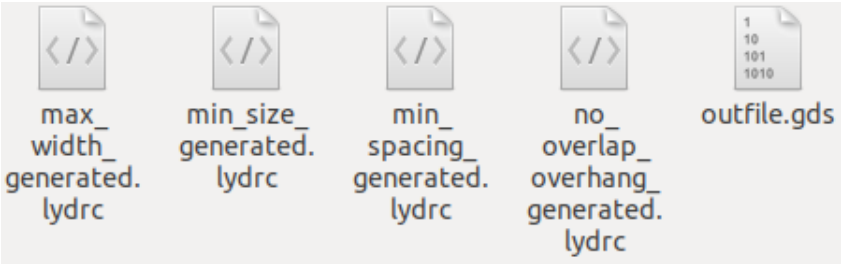


Figure 3.3: Screenshot of working directory after parsing

The tests were conducted on a simple layout to better explain how datatypes were implemented. A few shapes were placed on different layers, some violating set design rules. Below is shown how the masked layout looks (outfile.gds from figure 3.3 in this case).

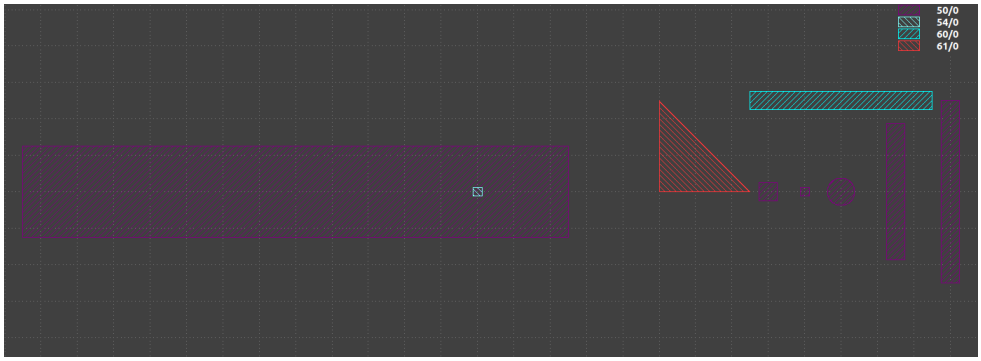


Figure 3.4: Test layout used for DRC testing

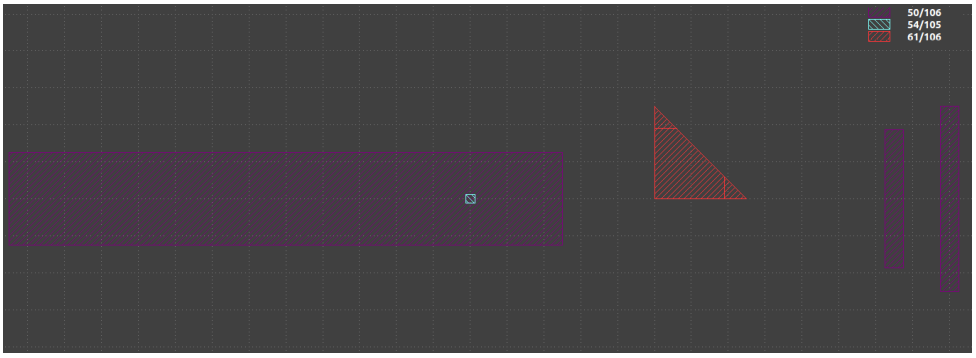


Figure 3.5: Results after the DRC tests

The difference between the two figures show the shapes that failed, or are faulty polygons. By doing this it's much easier to keep track visually of what is incorrect in a layout, rather than using the *Marker Database* from KLayout.

With the KLayout engine in place, *SPiRA* is able to do dynamic layout checking as well as allows for full post-layout DRC once a layout has been exported to GDS.

Chapter 4: Netlist extraction

The process of netlist extraction is complex procedure that involves numerous steps. This chapter will discuss how *meshes* and *nets* are created hierarchically and reduced to create a undirected graph representing physical connections in a given layout. After this process is complete further reduction is done to create a fully InductEx compatible netlist, allowing for full parameter extraction.

4.1 Polygon Operations during extraction

To understand the theories applied in this chapter, the difference in *SPiRA* classes and how it influences the GDS and graph representation of the layout, is explained. In chapter 2 when synthesis was discusses, the *spira.Device* and *spira.PCell/Cell* classes were shown. The Cell class is designed to contain sub-cells or cells that has a function in itself, whoever it can also be a fill structure pattern that spans over multiple layers or stitching vias. Devices form part of a larger sub-class called a *spira.Circuit*. The fundamental difference between a Cell and Circuit, is the polygon operations they are subject to during synthesis and layout extraction. As a *spira.Circuit* is suppose to contain sub-cells (like a Cell, PCell or Device), it is natural that routing and fill structures will be within this class. For this reason the C++ library, Clipper (wrapped in Cython), is used to do certain operations on the polygons in the circuit class. This library allows logic operations to be done on geometry, such as getting the union or intersections between polygons or shapes on the same layer [18]. Polygons connected to one another implicitly (by means of overlap on the same layer for instance) are merged to create shapes. In this case a shape is a complex polygon, or a polygon created from sub-polygons. This highlights the key difference between the Cell class (no polygon merging for shape creation) the Circuit class.

Cell class:

Figure 4.1 shows how two polygons are placed in such a way that they overlap and will of course during physical synthesis, merely be one piece of metal/conductor. As the Cell class is being used, no polygon operations will be done on it, until the cell is placed in a Device or Circuit class as a sub-cell. This also

means that netlist extraction for a cell is not possible as edge polygons are not fused and incoherence will be created in the mesh, not allowing connections to be properly made. At this point it is very important to note that the arrow shapes present in the layout, along with the thick outlines on each polygon, show the orientation of the edge port and the edge port itself. Edge ports play a vital role in this part, as it is used to determine where disjoints are between two or more polygons.

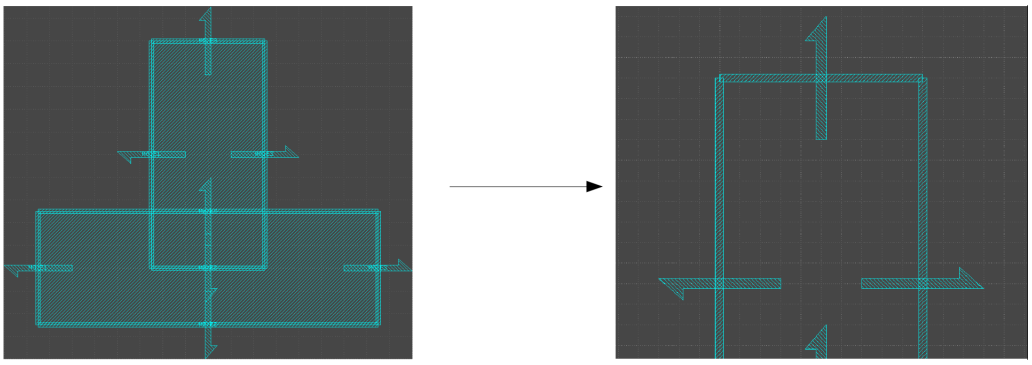


Figure 4.1: Two overlapping polygons (left) with edge port and orientation (right)

Code used to generate figure 4.1:

```
class T-shape(spira.Cell):
    def create_elements(self,elems):
        elems += spira.Box(layer = spira.RDD.PLAYER.M5.METAL,
                  width = 3, height = 1, center = (0,0))
        elems += spira.Box(layer = spira.RDD.PLAYER.M5.METAL,
                  width = 1, height = 2, center = (0,0.5))
        return elems
```

Circuit class:

Whenever a new Device or Circuit class is instantiated, the polygons' edge ports are tested for overlap. Whenever two edge ports overlap on the same layer, the two polygons are sent to the AND operation in Clipper and the resulting shape is placed in the layout as one polygon (as opposed to two).

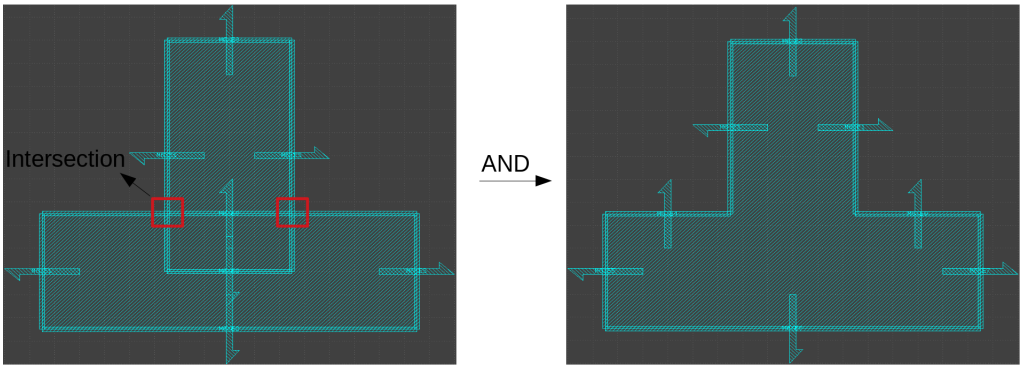


Figure 4.2: Intersection between polygons (left) and the resulting shape after AND operation (right)

Code used to generate figure 4.2:

```
class T-shape(spira.Device):
    def create_elements(self,elems):
        elems += spira.Box(layer = spira.RDD.PLAYER.M5.METAL,
                  width = 3, height = 1, center = (0,0))
        elems += spira.Box(layer = spira.RDD.PLAYER.M5.METAL,
                  width = 1, height = 2, center = (0,0.5))
        return elems
```

4.2 Generating Meshes

A mesh is defined as a collection of edges and vertices that are linked to one another to form a representation of a polygon or polyhedral. Depending of the type of vertices use, meshing can be first, second or third order depending on the mathematical equation used to define the vertices and edges. For the purpose of this dissertation, first order meshing, or triangular meshing was used. This means that all vertices and edges are connected by means of a straight line. The open-source API (application programming interface) PyMesh is used. This API allows for direct interface with the .geo file format, used by Gmsh. Gmsh is a standalone finite element mesh generator, with the required meshing algorithms needed by *SPiRA* for connection detection.

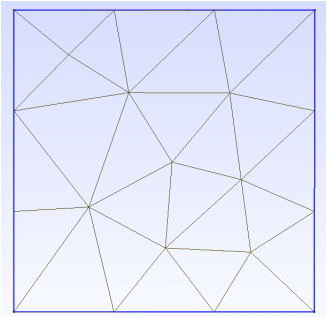


Figure 4.3: Simple first order mesh for a rectangle

4.2.1 Device mesh generation

As mentioned before, once a Cell has been included or a Device has been instantiated, all polygons on a layer will be stitched together with Clipper. Once this has been done, PyMesh is used to generate *.geo* for each individual polygon on all layers on the given layout. If the following code snippet is added to the PCell script (via-resistor connection), the *create_extract_netlist* function in *SPiRA* will be called.

```
D = via_res(kwargs)
netlist = D.extract_netlist
D.netlist_view(net=netlist)
```

This function will read the geometry of all layers to PyMesh, to create a *.geo* or Gmsh geometry file for each polygon. A mesh is then created for each layer. All the layers are then merged in order for connections to be made. In the case of the via-resistor PCell, the resistor is present on the R5 layer and is connected to the M6 layer by means of the C5R via layer.

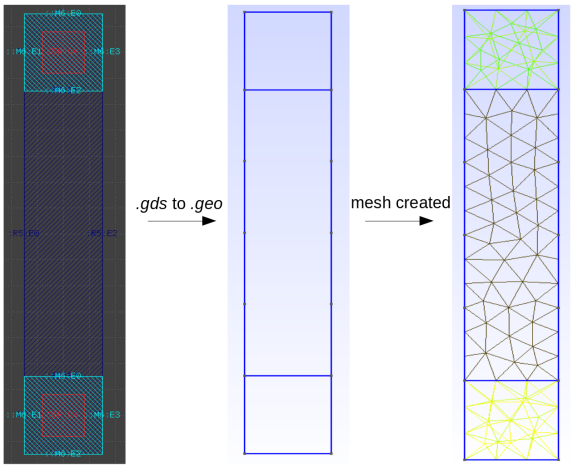


Figure 4.4: Example of how a mesh is created from a .gds file

Figure 4.4 shows how the *gds* representation of the layout is meshed in 2D. At the top and bottom of the resistor, where the connection is being made to a different layer, one can observe that two meshes are overlapping (each colour represents a different layer in the layout):

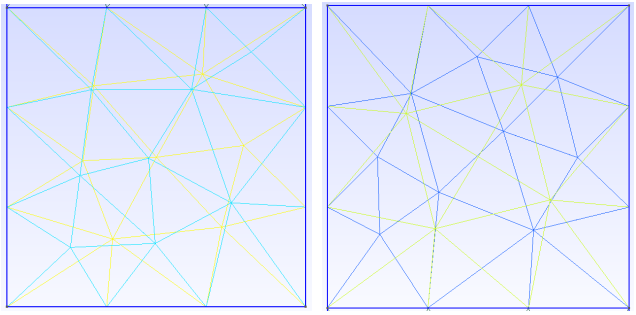


Figure 4.5: Closer view of meshed via connections.

Once the mesh is completed. An overlapping algorithm is run, to see if each mesh has coordinate points that overlap with the connected layer. *SPiRA* then uses the RDD again to determine if the meshes are connected to one another in the fabrication process. In this case, the R5 metal layer is connected to the M6 metal layer by means of the C5R via layer, thus if meshing of the C5R layer is present in the same place as M6 and R5 meshes, a connection is detected. Once a connection has been made, the mesh of each metal polygon is collapsed into a single node (edge) and the appropriate connections (vertices) are added. The simplified network in generated (in the form of an undirected graph) with a library called, NetworkX. This allows for the storing of object information (such as midpoint, layer or purpose of a particular polygon). NetworkX has optimized isomorphic and shortest-path algorithms built into it's functionality, which is cardinal to the netlist extraction process.

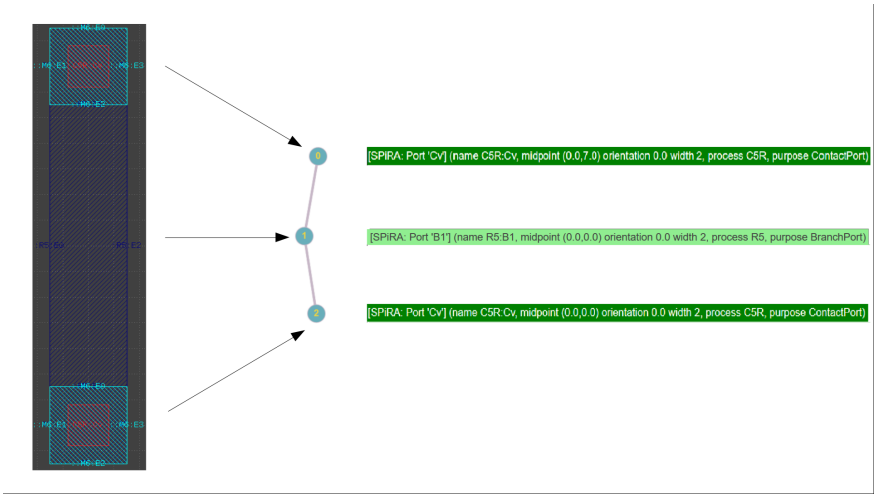


Figure 4.6: Simply geometry to undirected graph example

Once meshes and nets are created for each device class detected in the top level cell (this is the cell that contains the entire layout). These objects are placed into a larger circuit mesh.

4.2.2 Circuit mesh generation

As the name of the class suggests, the *Circuit* class, this class can contain many different devices (different junction size Josephson Junctions (JJs), vias for different layers, etc.) As mentioned before, the Circuit class contains the *create_structures*, *create_routes* and *create_elements* functions that is used to connect Devices/Cells by means of routing, to form a full circuit. Figure 4.7 shows a simple JTL (Josephson Transmission Line) circuit containing three devices - two JJs and the resistor-via device previously used.

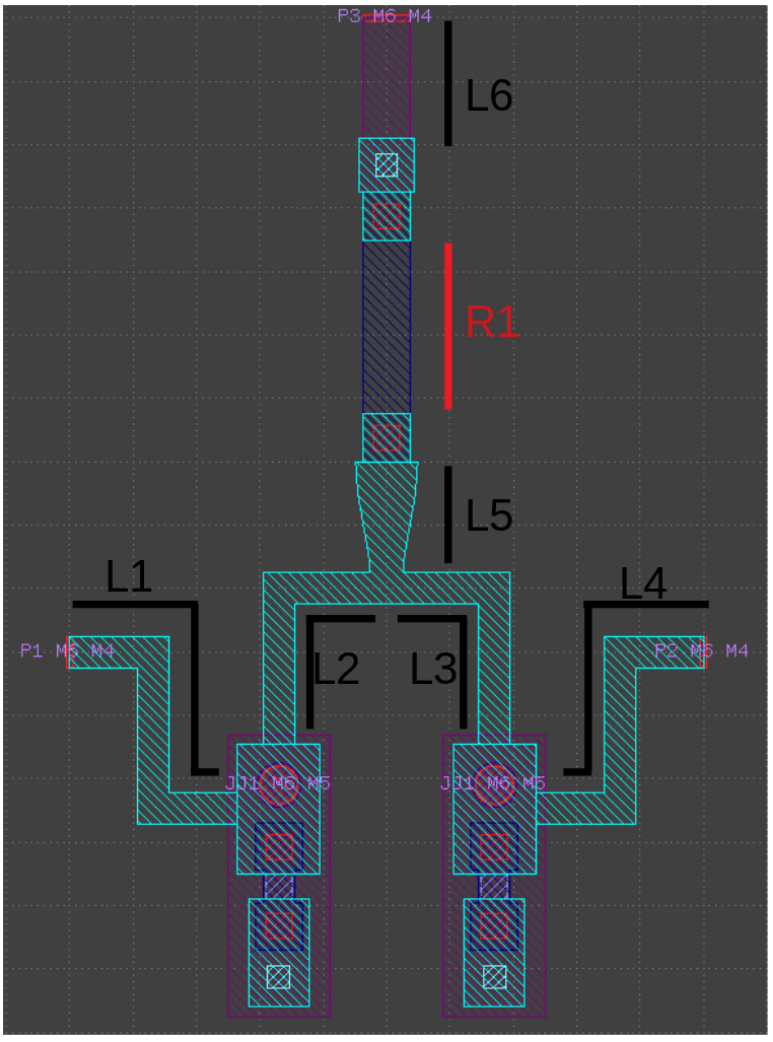


Figure 4.7: JTL layout with component numbers

The purpose of a JTL in a SFQ circuit is to connect cells to one another while keeping them as magnetically isolated from one another as possible. [19]. It is very common for a complex cell to contain more than one JTL and thus for this reason, it will serve as a good example to explain the principles with which a circuit is extracted. The elements in figure 4.7 have been labeled like one would to create a netlist or circuit schematic (full circuit schematic shown in figure 2.2). This means that the inductance of any via in the circuit will be assumed to be negligible .For the purpose of this circuit, the two JJs present in the layout, is added as two device parameters as well as the resistor-via cell.

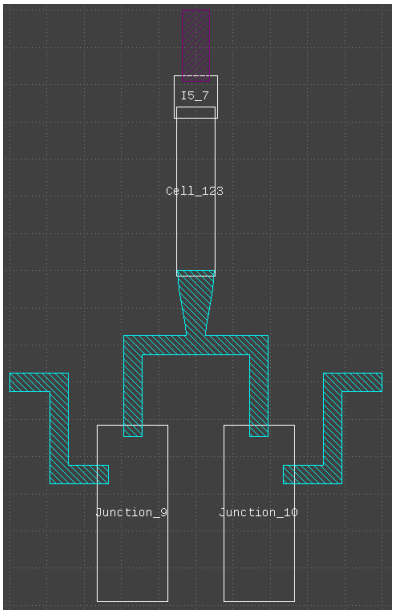


Figure 4.8: Circuit showing only the bounding edges of included devices and routing.

Due to the hierarchical nature of the GDS file format, all the geometry for a device does not have to be created every time, instead a *SRef* or Structure Reference can be used. Merely storing the Device once in binary format, and points towards the structure whenever it is being used. This means that all meshes and nets for Devices classes present in the Circuit class can be generated first and once it has been generated, the full circuit will be meshed to create a net, simply adding the device nets in correct places in the top-level net.

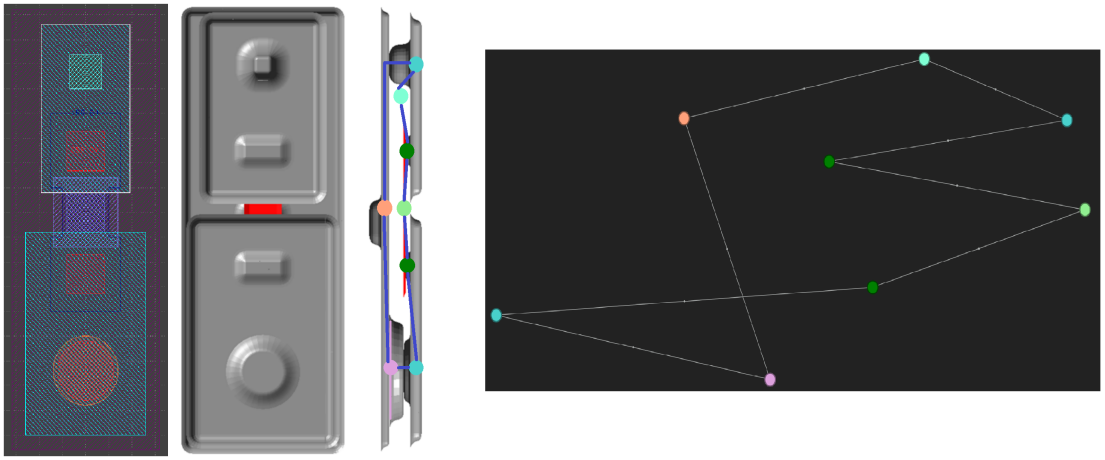


Figure 4.9: Interconnections extracted for JJ device (3D model [20]).

Once the devices have all been meshed, one of the first obstacles is encountered. Depending on the intended purpose of the Josephson junction, the way in which the JJ connects to the rest of the circuit changes. A JJ's use can be changed to either have the junction function as a storage device (stores a magnetic flux to be read out), decision making in the form of logic gates and transferring a signal from one cell to another. For this reason a JJ can not be reduced to a single node in a undirected graph as a logic gate or functional cell, thus one of the nodes in figure 4.9 will be implicitly connected to the rest of the layout. This implicit connection is classified by *SPiRA* as a *Dummy Node/Port* or a branch from one logical netlist element to another on the same metal layer. In the case of figure 4.7, dummy nodes will created for the connection between L2, L3 and L5 as well as the branching from the leftmost JJ to L1 & L2 and the same for the right JJ and L4 & L3. Figure 4.10 shows the schematic with markings where dummy nodes will be detected by the branching algorithm of *SPiRA*.

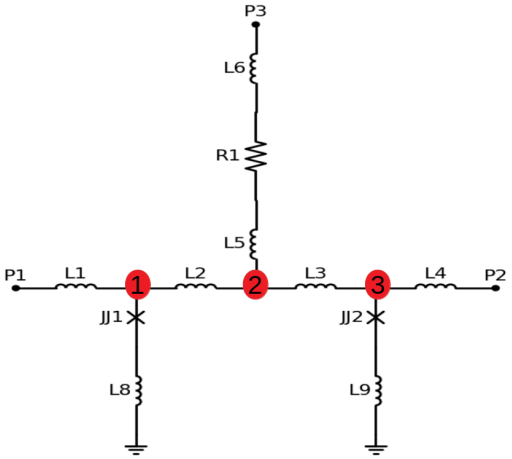


Figure 4.10: JTL schematic with dummy nodes numbered 1 - 3.

Once the dummy nodes/branches in the circuit has been detected, all the elements in the circuit will be meshed and a net will be generated. As mentioned before, the bottom-up (or hierarchical) design of this extraction process allows it to scale to much larger circuits and layouts. The figure below shows the full extracted result is shown below. The dummy ports are again marked in red to show the coherency between figure 4.10 and the extracted net (figure 4.11).

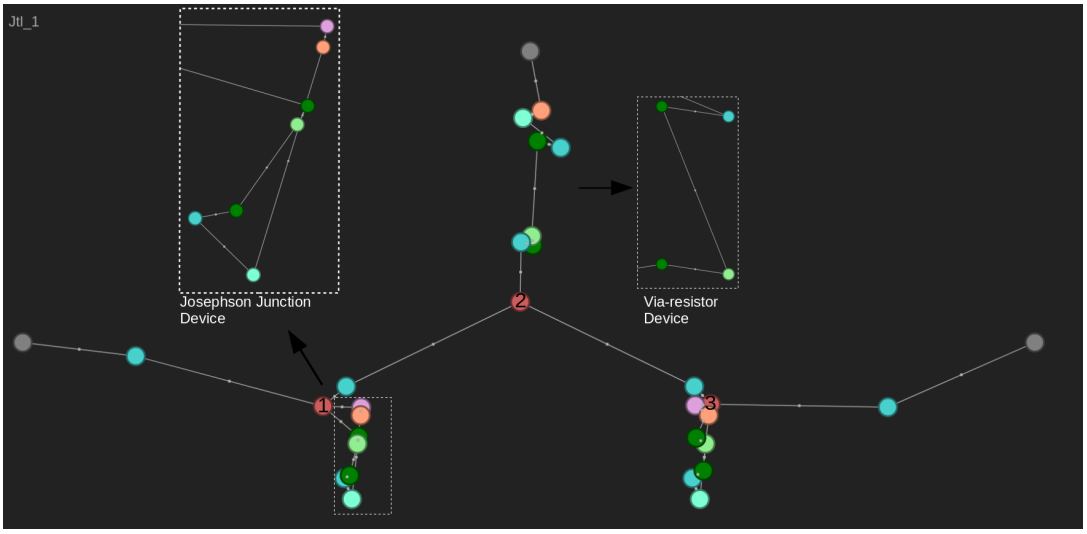


Figure 4.11: Full JTL netlist after extraction.

Once the net has been created for the entire layout, the *create_ electrical_ netlist* will be called. This function will take undirected graph, in the form of a NetworkX graph to create an InductEx compatible netlist which will be used for parameter extraction.

4.3 InductEx Netlist Generator

Once all the information about the physical connections made, are known, it is a matter of finding essential connection points. The essential elements will be all nodes that will represent a singular element in the schematic (fig 4.10). Due to the fact that inductors and other active components have to be placed on a specific layer in a fabrication process, more often than not, vias and other routing elements will need to be found and extracted along with active elements. These nodes contain essential information for proper extraction.
As shown in figure 4.6, each node is an object with it's own attributes. The port type, process and purpose attributes of each node must be taken into consideration. First the outmost terminal ports, branching ports and contact ports must be defined:

4.3.1 Port Types

Terminal port: Described in *SPiRA* as any node containing "T" in the naming scheme upon port creation. These ports are used to connect the layout to other circuits or supply biasing current to the cell. A terminal can be initiated as a parameter.

```
p1 = spira.Parameter(fdef_name='create_p1')
def create_p1(self):
    return spira.Port(name='M6:T1', midpoint=(-10,8),
    orientation=0, width=1)
```

Branch Port: Branch nodes are physical elements spanning over a metal a layer. This type of port represents physical conductors (resistors, inductors). Branch ports are not created by the user explicitly, instead these ports are created during extraction based on where active circuit elements are found. Referenced with "B" in the name.

```
[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
------------------------------------------------
[SPiRA: Port 'B260'](name M6:B260, orientation 0.0, width 2,
process M6,purpose BranchPort)
```

Contact Port: Much like the Branch port, a Contact port is automatically generated by *SPiRA* wherever a polygon is detected on designated Junction layer or whenever a connection is being made through a via layer. A Contact port is denoted in the naming scheme by "Cv".

```
[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
------------------------------------------------
[SPiRA: Port 'Cv'] (name J5:Cv, midpoint (3.4,3.8),
orientation 180.0 width 2, process J5, purpose ContactPort)
```

4.3.2 Extraction Procedure

The first step for creating an electrical netlist is to determine where all the external connection points, or terminal ports are, as every single element in the circuit should contain a path in the tree to either a terminal, or in the case of a shunted Josephson Junction, to ground. The aforementioned fact combined with the fact that each terminal port will have one of it's members be ground, the terminal ports can be reliable generated.

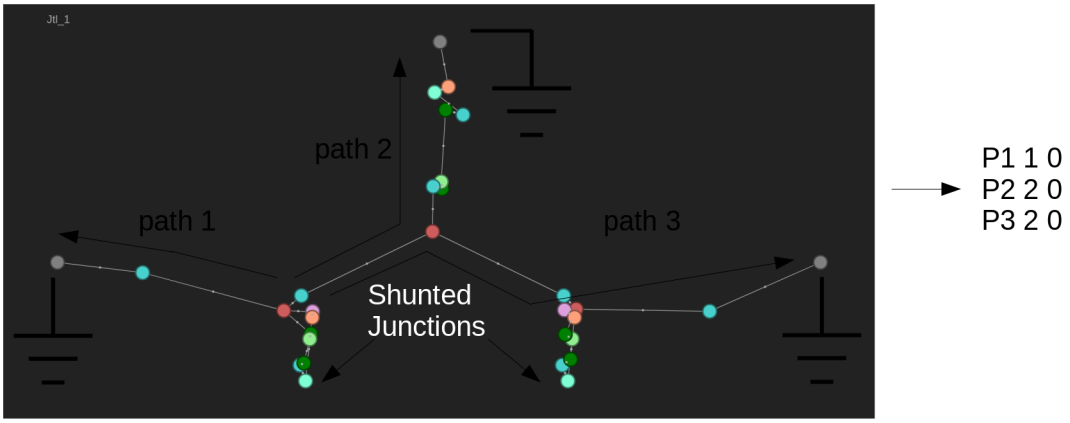


Figure 4.12: Extracted Terminal ("T") Ports / Shortest paths to Terminal ports.

Once all the Terminal ports have been created, all the aforementioned Dummy ports are located in the undirected graph, as logically that is where more than one connection is being made to a single element; or an element branches into two paths.

For each Dummy node found, the shortest path is calculated to each Terminal port or ground. This will result in multiple paths for each dummy node - the shortest of these paths are taken and analysed to determine if there are Branch nodes within in this specific path. Branch nodes by definition should always have a two members, regardless of the purpose of the branch. These members will be either Dummy nodes, Contact ports or Terminals, except when the branching to a JJ is made.

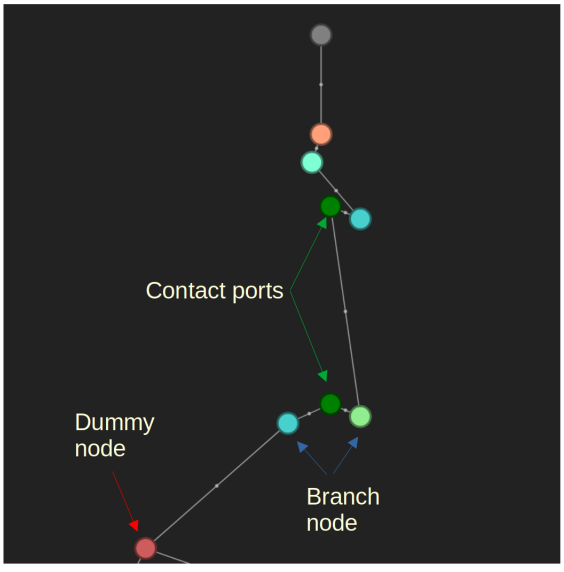


Figure 4.13: Example of Branch, Dummy and Contact ports after extraction.

Once all the Branch nodes and their accompanying via or Dummy nodes in the path in question has been extracted, the elements can be added to the InductEx netlist. This process is repeated till all possible paths and elements have considered. The essential paths detected is shown by figure 4.14.

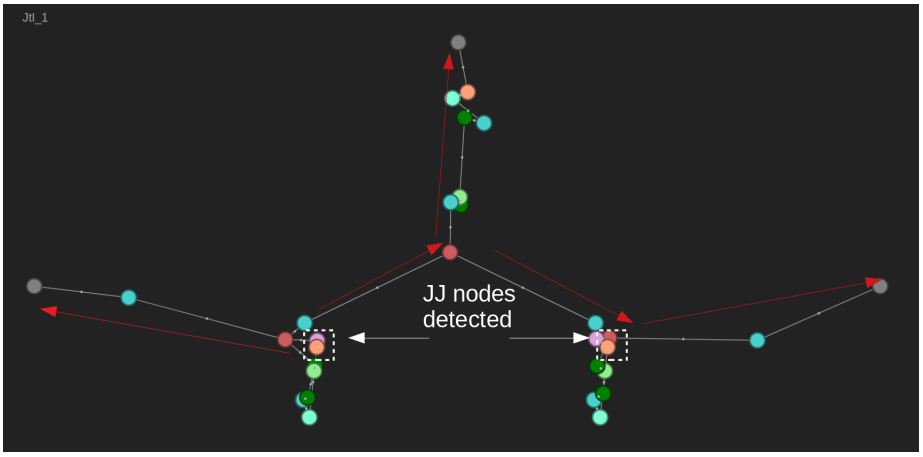


Figure 4.14: Essential Paths and detected JJ nodes.

After the Branch node extraction, the entire netlist should be populated apart from any Josephson Junctions. It is important to note that all the element sizes are a just set to a placeholder for now, as InductEx will be used to find the real (simulated) values. Node numbers are alpha-numeric values and are determined by the ID assigned to the Branch nodes by *SPiRA* and any node values '0' represents a ground connection.

```
Element Name | Node1 | Node2 | Size
P1 176 0
P2 187 0
P3 15 0
L1 244 176 2p
L2 187 257 2p
L3 16 244 2p
L4 16 257 2p
L5 15 299 2p
L6 292 299 2p
L7 285 16 2p
R1 285 292 1
```

The final step is to determine where JJs are present in the layout. To do this, the model used by InductEx for JJs need to established. A undirected graph representation of a junction is made and fitted to the entire layout te determine junction positioning. Fig. 4.14 highlights where the junction Contact ports are detected, which will initialize the process to identify all the junctions.

InductEx JJ model:

InductEx implements JJs by means of Ports. Figure 4.15 shows how InductEx models a standard shunted JJ connection. A pull-down inductor is added after the JJ port to model the internal inductance of the junction itself.

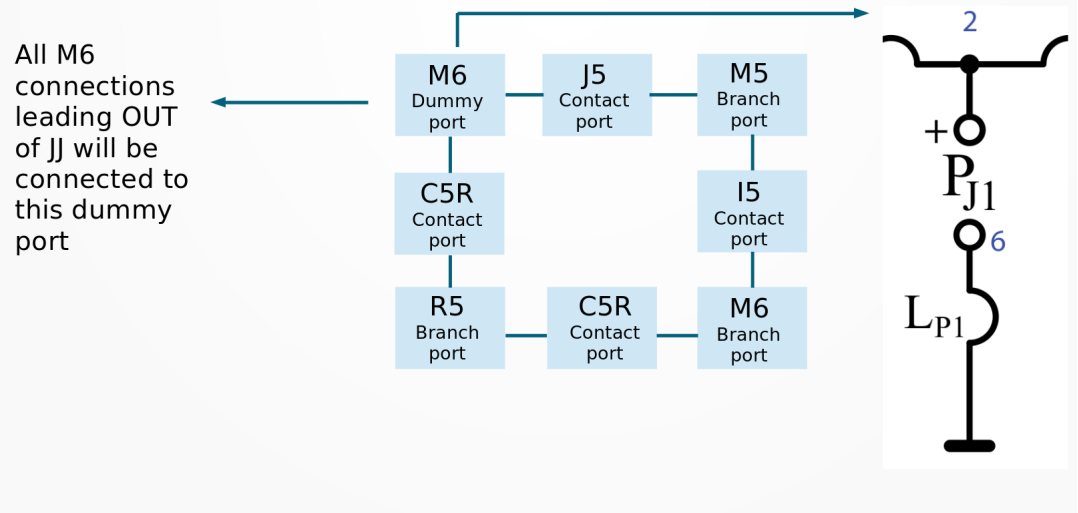


Figure 4.15: Shunted JJ model fitted to layout.

Once all the junctions have been plotted out and the netlist extraction is complete, a netlist file with the *.cir* or InductEx circuit file extension is written and ready to send to InductEx. The junctions added are shown below:

```
1 JJ1 244 240 100
2 L8 240 0 0.1p
3 JJ2 257 250 100
4 L9 250 0 0.1p
```

At this point the *call_ inductex* function is called to send the correct layout file, netlist file and layer definition file to InductEx for simulation. The LDF (Layer Definition File) is proprietary to each process and thus information cannot be quoted from it, however this file is created to contain the layer thicknesses, penetration depths and other parameters required for simulation.

4.3.3 Extraction Results

The functions performed in *call_ inductex* is explain in chapter 5, as it does not directly link to the netlist extraction process, instead needs to perform some functions on the GDS representation of the layout for InductEx to understand it. The extraction results for the JTL does also not contain the resistor element as it does not form a critical part of the simulation as resistors are purely parasitic elements in SCE[21] and the biasing current applied to the resistor (at P3) can just be changed to produce the correct potential difference over

the resistor - which in effect changes the biasing current supplied to the JTL. The following table shows the full extracted parameters along with the newly generated *.cir* file.

Element Name	Node 1	Node 2	Inductance (pH)	Critical Current (μA)
P1	176	0	-	-
P2	187	0	-	-
P3	15	0	-	-
L1	176	244	3.91814	-
L2	244	16	3.36602	-
L3	16	257	3.3874	-
L4	257	187	3.93038	-
L5	16	15	1.59431	-
JJ1	244	240	-	153.93
JJ2	257	250	-	153.93
L8	240	0	0.453305	-
L9	250	0	0.473814	-

Table 3: InductEx extraction results with node numbers assigned by *SPiRA*

The simulated results above directly match up with original design shown in appendix A1, showing that any cell accurately generated in *SPiRA* will yield the same results as a circuit laid out by hand.

The extracted results showed above can then directly be compared to the cell designer's calculated theoretical/designed values, which will indicate if the size of inductors or junctions need to be adjusted. These adjustments can then just be brought forth by adjusting the appropriate *SPiRA* parameter, such as the width of the routing or the size of the junction disc.

Chapter 5: *SPiRA*-tools

The goal of *SPiRA-tools* is to create bridges from *SPiRA* environment to simulation engines such InductEx and JoSim [22]. *SPiRA-tools* also introduces a schematic generator to produce a Standard Vector Graphic file (*.svg*) of the generated InductEx netlist.

Due to the scripting nature of the Python language, *SPiRA-tools* is not strictly tied to the *SPiRA* core and can be used independently by any user who is unfamiliar with InductEx standards and needs a compatible layout for simulation.

5.1 Creating InductEx compatible layouts.

InductEx has certain input requirements when it comes to parsing the .GDS layout. As mentioned before *SPiRA* makes use of the different datatypes available in the GDS file format, however InductEx only reads polygons from layers with datatype 0. For this reason, after the *create_netlist* function, previously discusses, is called the layout can be modified as all the required LVS operations have been performed by *SPiRA*.

All edge ports will be removed from the layout and all polygons will be transferred to the same layer, but with datatype 0. The raw output (as produced by *SPiRA*) is shown in figure 5.1. Throughout chapter 4 the already filtered output was used, as it is easier for demonstrative purposes. Apart from clearing unwanted datatypes, InductEx ports are placed for proper meshing. A *spira.ports* file can be generated during the GDS writing process. This file contains all Terminal port locations as well as their connected process, orientation and width. The same port naming scheme is used as in *SPiRA*. This file is parsed and path elements are drawn over the given terminals on layer 181, as specified by the InductEx manual [17]. Once the terminal ports have been mapped, all Josephson junctions found (by looking for polygons on the designated Junction layer stored in the RDD) are labeled with a name, positive and negative terminals. The positive terminal will be the assigned to the metal layer in the name of the terminal and the ground plane of the loaded fabrication process (RDD) is used as the negative terminal. An example *spira.ports* for the JTL layout used throughout the dissertation follows.

```
M6:T1 (-10,8) 0.0 1
M6:T2 (10,8) 90.0 1
M6:T3 (0,28) 270.0 1.5
```

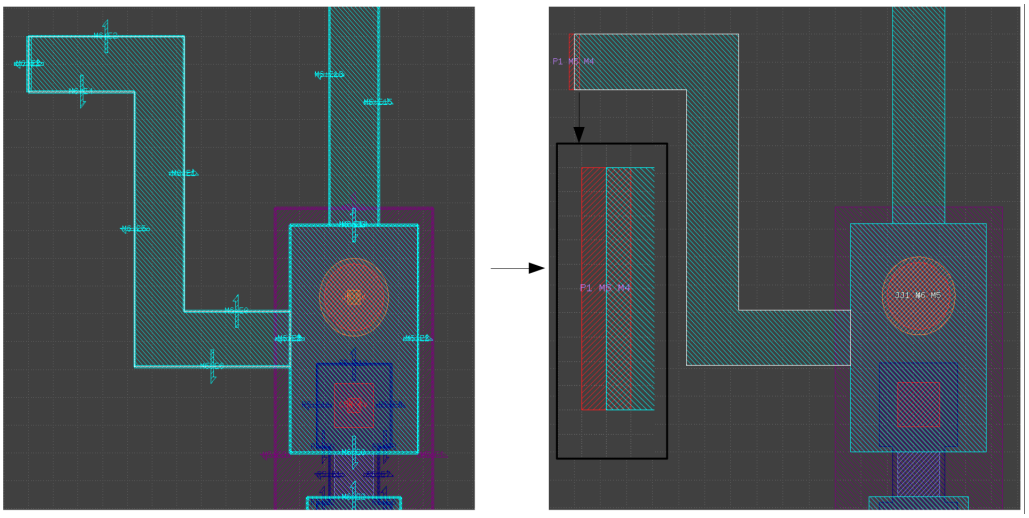


Figure 5.1: Raw *SPiRA* output (left), output from *SPiRA-tools*(right).

5.2 Schematic Generator

At the heart of the schematic generator included in *SPiRA-tools* is the open-source electrical circuit schematic library, *SchemDraw*[23]. *SchemDraw* allows the user to create high-quality circuit diagrams in a scripting environment and allows the user to make custom circuit element symbols to be used in any schematic. This is cardinal as most symbol libraries will not contain SCE components such as a Josephson Junction. *SPiRA-tools* is meant to encapsulate the functionality of *SchemDraw* to produce a script which will create a Standard Vector Graphics (*.svg*) file of a given netlist.

The schematic generator makes use of some of the same algorithms used during netlist extraction such as finding a path through the netlist to ground or a terminal from a given node and determining the direct members of an element in the netlist.

5.2.1 Drawing Circuit Elements

Before any efforts are put towards parsing the given netlist or applying any pathing algorithms, the basic elements, such as resistors, inductors, ports and junctions need to be define. All of the aforementioned elements are already included in the *SchemDraw*, except for a junction model. An *Element* class is created for this purpose and will just be imported wherever a junction needs to be drawn.

```
class Junction(SchemDraw.elements.Element):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.segments.append(Segment([[-1, 0], [1, 0]]))
        self.segments.append(Segment([[-0.2,-0.2],[0.2,0.2]]))
        self.segments.append(Segment([[-0.2,0.2],[0.2,-0.2]]))

        self.anchors['p1'] = [-1, 0]
        self.anchors['p2'] = [1, 0]
        self.start = self.anchors['p1']
        self.end = self.anchors['p2']
```

The Junction class shown above simply creates a line with a terminal (or *anchor*) on either side with a cross drawn on the midpoint of the line. Upon instantiation of the class, the two anchors labeled *'p1'* and *'p2'* will be attached to desired neighbouring elements' terminals.

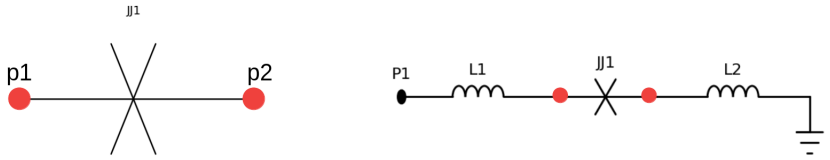


Figure 5.2: Newly generated JJ symbol with anchors shown.

With the junction symbol ready to use, the function wrapping methodology used, can be explained. The *SchemDraw* objects require certain parameters to ensure elements are drawn in the right place, with some parameters being optional. A *Drawing* object will always need to be created to which the elements will be attached. Thereafter new elements can be created and implicitly added to the *Drawing* object.

```
draw = SchemDraw.Drawing()
P1 = draw.add(elm.Dot(label = ''))
J1 = draw.add(Junction,anchor = 'p2',d= 'right',label = 'JJ1')
L1 = draw.add(elm.Inductor(label = 'L1'))
L2 = draw.add(elm.Inductor(xy = J1.p2,d = 'down',label = "L2"))
```

The key difference between object parameters are shown by the code extract above. Whenever an object is created without the *'xy'* or *'d'* parameter, the drawing library will make use of the previously created object's variables to determine the location (*'xy'*) and direction (*'d'*) of the object in question. Or in other words, whenever *xy* and *d* is omitted, the element will automatically be attached to the previously instantiated object in the *Drawing* class, *draw*, thus L1 will automatically be connected to J1 with the direction set to

'right'. However when these parameters are present, *SchemDraw* will place the component at the specified location with the desired direction, regardless of overlapping elements. This means that L2 is also connected to J1 and would overlap with L1 if not for the directional input 'down' that is given.

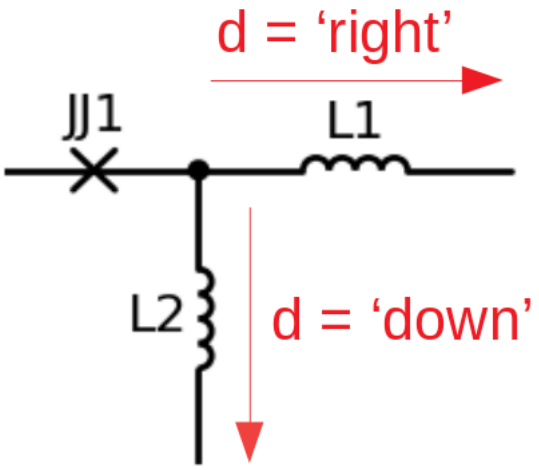


Figure 5.3: Execution of code shown, indicating direction of elements.

With these parameters in mind, functions writing the correct lines to a designated script, were created. These functions take all the aforementioned values ('d', 'xy' and 'achor') as parameters. An example of how the *draw_element* functions work is given below by means of the *draw_inductor()* function. It is important to note that the *'xy'* variable describes where the element will be placed as a function of it's adjacent element ports i.e L2's *xy* parameter will be given as *'J1.p2'* as shown in the previous code extract.

```
def draw_inductor(label,direction,xy):
    s = ""
    if(xy is "" or xy is None):
        s = "{} = d.add(elm.Inductor(label = '{}',d = '{}'))\n"
        .format(label,label,direction)
    else:
        s = "{} = d.add(elm.Inductor(label = '{}',d = '{}'
        ,xy = {}))\n".format(label,label,direction,xy)

    return s
```

This structure of function is created for the following elements:

- Resistor
- Ports/Terminals

- Josephson Junctions
- Ground connections

With all these functions in place, the algorithm used to determine and build a desired schematic, can be explained.

5.2.2 Drawing a Desired Schematic

Before the algorithm that writes the circuit elements to the script is initialized, a desired netlist is read in and saved within a list. Along with the netlist declaration, a global direction variable and a variable that will keep track of how many elements are already drawn, is instantiated and the required imports are already written to the output script, which will be created in the working directory as *draw_schematic.py*. After this process, the *Draw_net()* is called which starts the drawing process. The only user input requirements are the *netlist.cir* and the terminal or port that list LEFTMOST on the physical GDS layout. The leftmost node is merely supplied to attempt drawing the schematic as close the physical layout as possible. In the case of the JTL used, *P1* (shown in figure 5.1) has the smallest x-coordinate and will thus be used.

The *draw_port* function will draw P1 after which a recursive algorithm will be called to draw the rest of the schematic. This function will terminate once the global element counter reaches the same length as the given netlist. This will draw all the elements connected to a given element (in this case, P1, for the first iteration), as long as the element only has a single fan-out. This means that the given element is only connected in a one-to-one fashion and does thus not branch into two segments. Once and element has been found to have more than two members (one element to each terminal), *draw_till_terminal* function will be called.

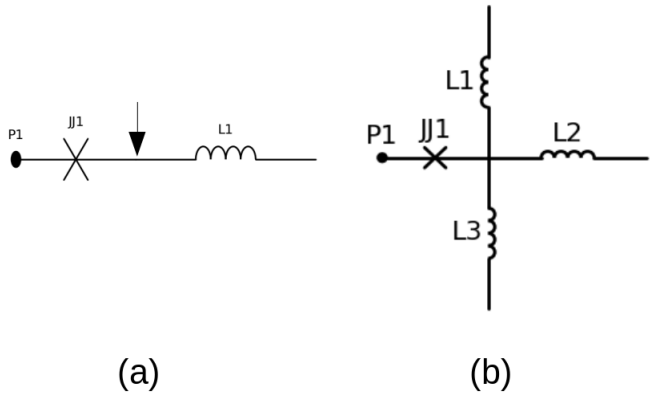


Figure 5.4: Single fan-out element (a), Multi fan-out element (b).

The shortest path for that given element is determined to the closest port/-ground terminal. Once the path has been determined, the appropriate drawing functions are called. By default a branch that terminates to ground is drawn downwards. If more than two branches are detected, the global direction will be set to 'up','down','right', with 'right' being the next single fan-out element.

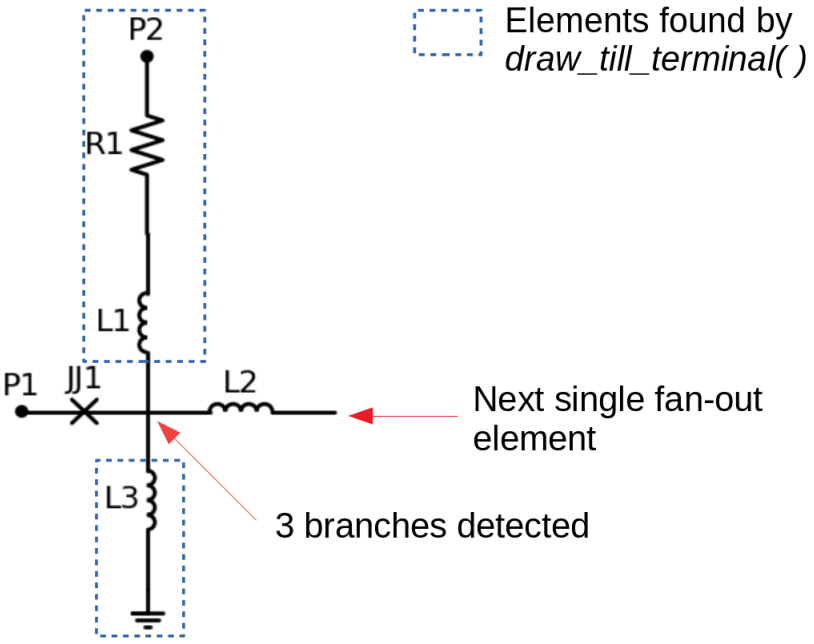


Figure 5.5: Schematic showing *draw_till_terminal* calls.

The 'up' and 'down' branches will be drawn first, after that L2 (from figure 5.5) will trigger the next iteration of the algorithm, starting with the drawing of all single fan-out elements till another branch is detected or till the global element counter has reached a maximum. Whenever branching is detected within the *draw_till_terminal* function, it will call itself to draw the path. The current limitation of the software lies in the amount of branches currently supported in sub-branches. Any given branch can only branch once (changing the global direction to 'right'). In a future iteration of *SPiRA-tools* will work to eliminate this restriction for full, robust schematic drawing.

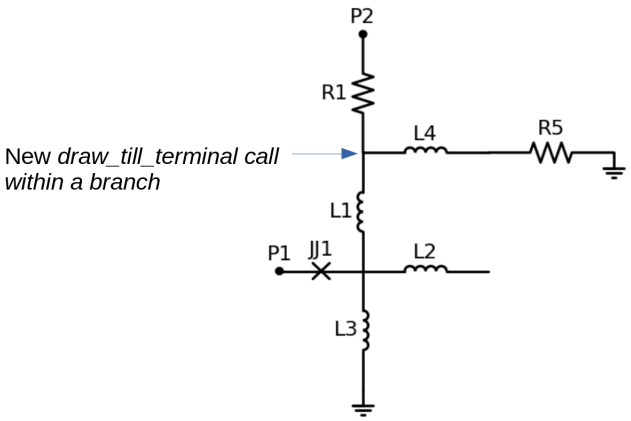


Figure 5.6: New branch detected within a branch.

5.2.3 Resulting Script

Once the drawing algorithm has been terminated and all the elements have been written to the aforementioned *draw_ schematic.py* script. This script can be executed to output a *plotly* schematic of the circuit.

The resulting script for the JTL netlist used throughout the dissertation is shown below along with the schematic. The schematic illustrates which part of the algortihm is used to generate each part of the netlist.

```
draw = SchemDraw.Drawing()
P1 = draw.add(elm.Dot(label ='P1'))
L1 = draw.add(elm.Inductor(label ='L1'))
draw.push()
J1 = draw.add(Junction, anchor ='p1', d='down', label ='JJ1')
L7 = draw.add(elm.Inductor(xy = J1.p2, d ='down', label ="L8"))
draw.add(elm.Ground())
draw.pop()
L2 = draw.add(elm.Inductor(label ='L3'))
draw.push()
L3 = draw.add(elm.Inductor(label ='L7', d ='up'))
R1 = draw.add(elm.Resistor(label ='R1', d ='up'))
L6 = draw.add(elm.Inductor(label ='L6', d ='up'))
L5 = draw.add(elm.Inductor(label ='L5', d='up'))
draw.add(elm.Dot(label ='Pdc'))
draw.pop()
L4 = draw.add(elm.Inductor(label ='L4'))
draw.push()
J2 = draw.add(Junction, anchor ='p1', d ='down', label ='JJ2')
L8 = draw.add(elm.Inductor(label ='L9', xy = J2.p2, d ='down'))
draw.add(elm.Ground())
draw.pop()
```

- P1 - Leftmost node fed to drawing algorithm, initializing it.
- L1 - All single fan-out connected drawn.
- JJ1 - Branch detected, *draw_till_terminal* called. Path to ground is found (through L8) and drawn.
- L3 - New iteration of drawing algorithm started and all single fan-out connected drawn.
- L7 - Branch detected, *draw_till_terminal* called. Path to terminal (through R1, L6 and L5), Pdc, found.
- L4 - New iteration of drawing algorithm started and all single fan-out connected drawn.
- JJ2 - Branch detected, *draw_till_terminal* called. Path to ground is found (through L9) and drawn.
- L2 - New iteration of drawing algorithm started and all single fan-out connected drawn.

The *push()* and *pop()* function is used to store the location of the last drawn member in the *Drawing* class and return to the saved location, respectively.

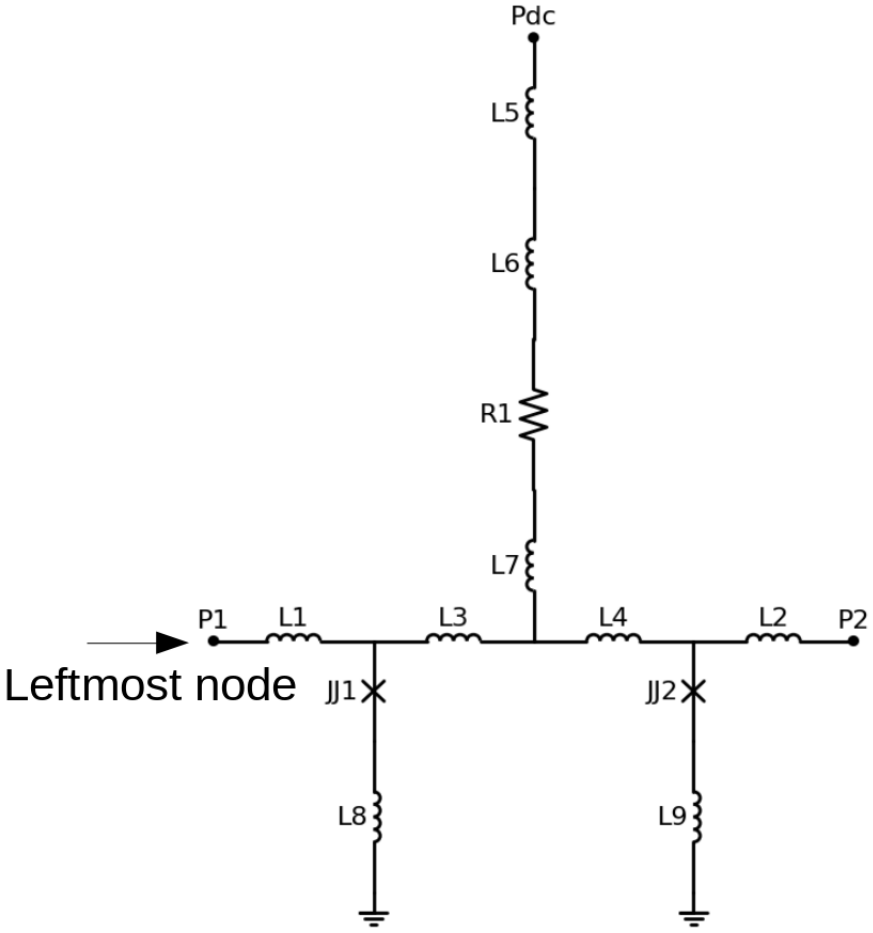


Figure 5.7: Output *plotly* schematic of full JTL

Chapter 6: Results and Applications

6.1: Application of Rule Deck Database

6.1.1 Creating a basic Rule Deck Database

To create a Rule Deck for the user's process of choice, the layers, their purposes and GDSII layers needs to be instantiated. The **get_rule_deck** function will need to imported to create a rule deck and correctly connect the purpose and process data.

```
from spira.yevon.process.all import *
from spira.yevon.process import get_rule_deck

# initialize a Rule Deck wiith name RDD
RDD = get_rule_deck()
```

Once the Rule Deck has been created, the desired layers (processes) and purposes can be assigned. To do so, a **ProcessLayerDatabase** will be defined along with the desired **ProcessLayers** and the required **PurposeLayerDatabase**.

```
RDD.PROCESS = ProcessLayerDatabase()
RDD.PROCESS.M0 = ProcessLayer(name = 'Metal 0', symbol = 'M0')

RDD.PURPOSE = PurposeLayerDatabase()
RDD.PURPOSE.GROUND = PurposeLayer(name='Ground plane polygons',
                     symbol='GND')

RDD.PURPOSE.HOLE = PurposeLayer(name = 'Ground plane hole
                   polygons', symbol = 'HOLE')
```

Once a Process has been put in place, a **PhysicalLayerDatabase** has to be made for each different Process created. The **PhysicalLayerDatabase** connects process and purpose data to one another.

```
RDD.PLAYER.M0 = PhysicalLayerDatabase()
RDD.PLAYER.M0.GND = PhysicalLayer(process=RDD.PROCESS.GND
                    ,purpose=RDD.PURPOSE.GROUND)
RDD.PLAYER.M0.HOLE = PhysicalLayer(process=RDD.PROCESS.HOLE,
                     purpose = RDD.PURPOSE.HOLE)
```

Now parameters of each layer created can be stored by means of a **ParameterDatabase**

```
RDD.M0 = ParameterDatabase()
RDD.M0.MIN_SIZE = 0.5
RDD.M0.MAX_WIDTH = 20.0
```

As a final step a **Layer_Process_Map** and **Purpose_Datatype_Map** will need to created, to tell *SPiRA* what GDS layers and datatypes to use for a given purpose.

```
RDD.GDSII.PROCESS_LAYER_MAP = {RDD.PROCESS.M0 : 10}

RDD.GDSII.PURPOSE_DATATYPE_MAP = {RDD.PURPOSE.GROUND : 0,
                                  RDD.PURPOSE.HOLE : 2}
```

6.1.2 Using parameters from the RDD

Printing the physical layer or the process of a specific layer is as simple as calling the corresponding function to obtain the parameters. These functions are put in place to allow the user to find out what processes are connect to a specific layer without having to open the more complex database file. The GDS layers and datatypes used can be accesses by calling the correct Map.

```
>> spira.RDD.PLAYER.M0.get_physical_layers()

[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
-----------------------------------------------
[SPiRA: PhysicalLayer] (name M4, process 'M0', purpose 'GND')
[SPiRA: PhysicalLayer] (name M4_HOLE, process 'M4', purpose '
   HOLE')
```

```
>> spira.RDD.GDSII.PURPOSE_LAYER_MAP

[SPiRA] Version 0.2.3-Auron [Beta] - MIT License
-----------------------------------------------
[SPiRA: PurposeLayer]('Ground plane polygons', symbol 'GND'): 0
[SPiRA: PurposeLayer]('Polygon holes', symbol 'HOLE'): 2
```

Once the user knows which process and purpose layers are present within the RDD at use, polygons and routing can be added to a specific layer by giving the corresponding **PhysicalLayerDatabase** as the layer parameter within any new element placed inside the *create_elements*, *create_structures* or *create_routing* functions.

```
def create_elements(self,elements_to_be_added):
    a_polygon = spira.Circle(layer = RDD.PLAYER.M0.GND,
                box_size = (1,1))
    elements_to_be_added += a_polygon
    return elements_to_be_added
```

The following code should yield a rectangle with a width/height of 1 database units, placed on the newly created M0 layer (layer 10) with purpose GND (datatype 0).

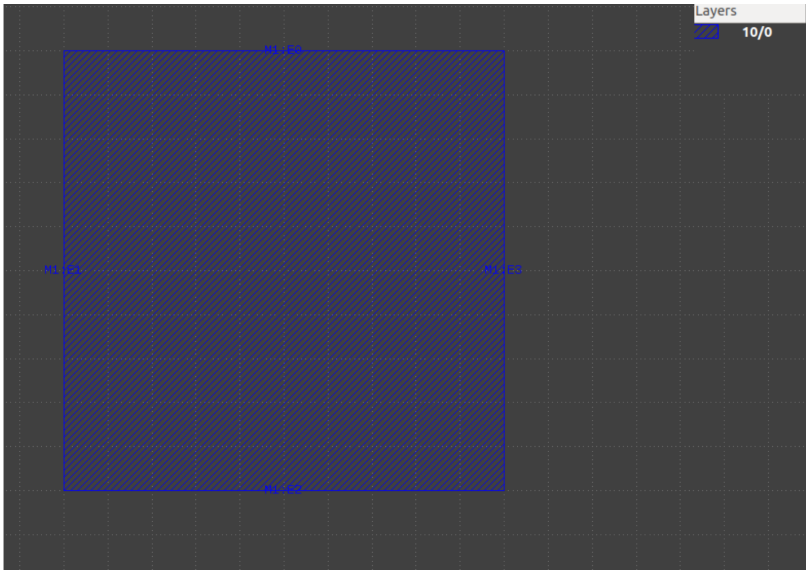


Figure 6.1: Resulting rectangle in GDS format

With these classes and functions already set in place in the *SPiRA* core, a user can add all the required process and purpose layers to the database, depending on the fabrication process at use.

6.2: Basic Junction PCell extraction

A Josephson Junction PCell has been set up to demonstrate extraction of a Junction. This is done to show that there should be no discrepancies between a cell created in a CAD tool and *SPiRA*

6.2.1: Geometry and Routing

The geometry for a JJ was placed along with two terminals, T1 and T2. The *RouteManhattan* function previously discussed is used to route the desired ports on the JJ to the created terminals.

```
class Junction(spira.Device):

    t1 = spira.Parameter(fdef_name = 'create_port_1')
    t2 = spira.Parameter(fdef_name = 'create_port_2')
    jj0 = spira.Parameter(fdef_name='create_jj_100sg_0')

    # place the JJ
    def create_structures(self,elems):
        elems += self.jj0
        return elems
    # route T1 & T2 to the JJ
    def create_routes(self,routes):
        routes += spira.RouteManhattan(
                  ports=([self.t1,self.jj0.ports[26]]),
                  layer=spira.RDD.PLAYER.M6.METAL,width = 1)
        routes += spira.RouteManhattan(
                  ports = ([self.jj0.ports[25],self.t2]),
                  layer = spira.RDD.PLAYER.M6.METAL,width = 1)
        return routes
```

Upon calling the GDS output function from *SPiRA*, the following layout will be created:

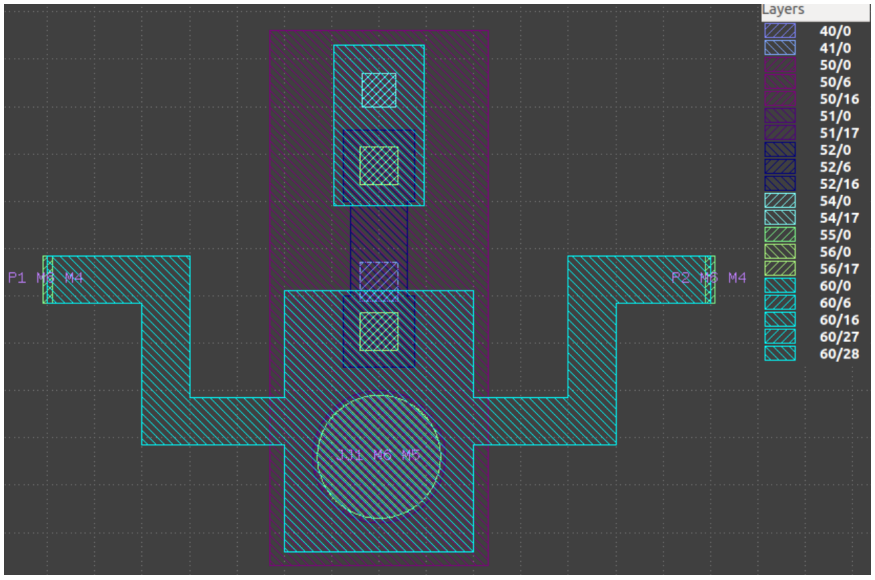


Figure 6.2: Basic Junction output

The output can then be sent to InductEx for simulation. The following solution was generated by InductEx:

```
Name        Design       Extracted
L1          2E-12        3.28966E-12
L2          2E-12        3.171E-12

Junction    Critical current [A]
Name        Design       Extracted
JJ1         100          0.00061571
```

From these results, it's clear that the simulated results should not vary between any CAD tool and *SPiRA* after all the correct adjustments have been made to the layout by *SPiRA-tools*.

6.3: Synthesis and Extraction Results

A Passive Transmission Line or PTL is much like a JTL in functionality. Both of the transmission lines are used to transfer pulses between cells. The main difference between a JTL and PTL is that a JTL is powered by means of a biasing current where as a PTL transfers data passively. A PTL has the advantage of transferring pulses over long distances at rapid speeds by making use of the transmission laye, however this makes the pulse more susceptible to noise and interference.

6.3.1: Parameters

The following code is called inside of the main PTL class to initialize the parameters that will be used:

```
    p1 = spira.Parameter(fdef_name='create_p1')
    p2 = spira.Parameter(fdef_name='create_p2')
    p3 = spira.Parameter(fdef_name='create_p3')

    jj0 = spira.Parameter(fdef_name='create_jj_100sg_0')
    jj1 = spira.Parameter(fdef_name='create_jj_100sg_1')
    jj2 = spira.Parameter(fdef_name='create_jj_100sg_2')
    res0 = spira.Parameter(fdef_name='create_res_0')
    via_i5 = spira.Parameter(fdef_name='create_via_i5')
```

6.3.2: spira.ports File

The *spira.ports* file contains the locations, process layer, width and orientation of all the ports needed for inductance extraction. For the PTL, the following code is executed in the main PTL class to generate the text file.

```
file = open('spira.ports','w')
n1,m1,orientation1,w1 = str(self.create_p1().name),
                        str(self.create_p1().midpoint),
                        str(self.create_p1().orientation),
                        str(self.create_p1().width)

n2,m2,orientation2,w2 = str(self.create_p2().name),
                        str(self.create_p2().midpoint),
                        str(self.create_p2().orientation),
                        str(self.create_p2().width)

n3,m3,orientation3,w3 = str(self.create_p3().name),
                        str(self.create_p3().midpoint),
```

```
                            str(self.create_p3().orientation),
                            str(self.create_p3().width)


file.write(str(n1+ " " + m1+" "+orientation1+" "+w1+"\n"))
file.write(str(n2 + " " + m2+" "+orientation2+" "+w2+"\n"))
file.write(str(n3 + " " + m3+" "+orientation3+" "+w3+"\n"))
file.close()
```

This will produce the following text file:

```
M5:T1 (6.0,0.45) 270.0 1
M6:T2 (-4.95,-6.68) 270.0 1
M6:T3 (10,-23) 180.0 1
```

6.3.3: Filtered GDS Output

SPiRA-tools is used to modify the original layout and add the correct port and junction labels. The modified layout (figure 6.2) shows how all non-zero datatypes have been removed and layer 182 is used for port and junction labels. Layer 19 contains the GDS path polygons, used to indicate the terminal and port widths.

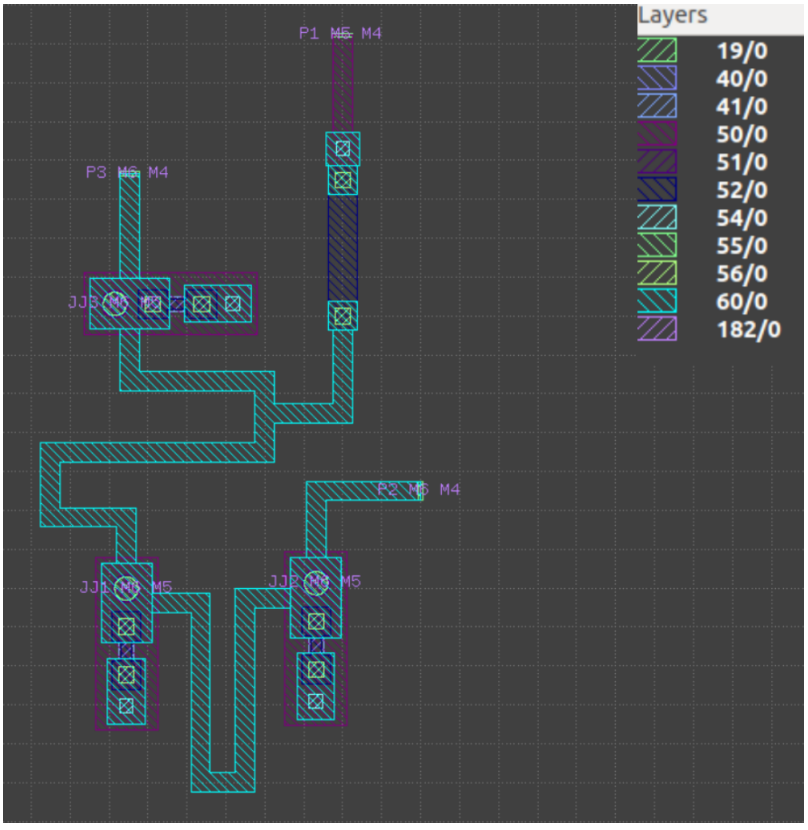


Figure 6.3: Filtered PTL output

6.3.4: Undirected graph representation

Once the geometry of the PTL has been created, an instance of the class needs to be created and filtered before calling the extraction functions. The following code is added to the PCell script.

```
PTL = PtlRX()
PTL.gdsii_output(filename = 'ptlrx')
PTL = RDD.FILTERS.PCELL.MASK(PTL)

net = D.extract_netlist
```

Once the code above is executed, an undirected graph is generated and stored as a *plotly* graph (in the form of an html file). The output is shown in figure 6.4.

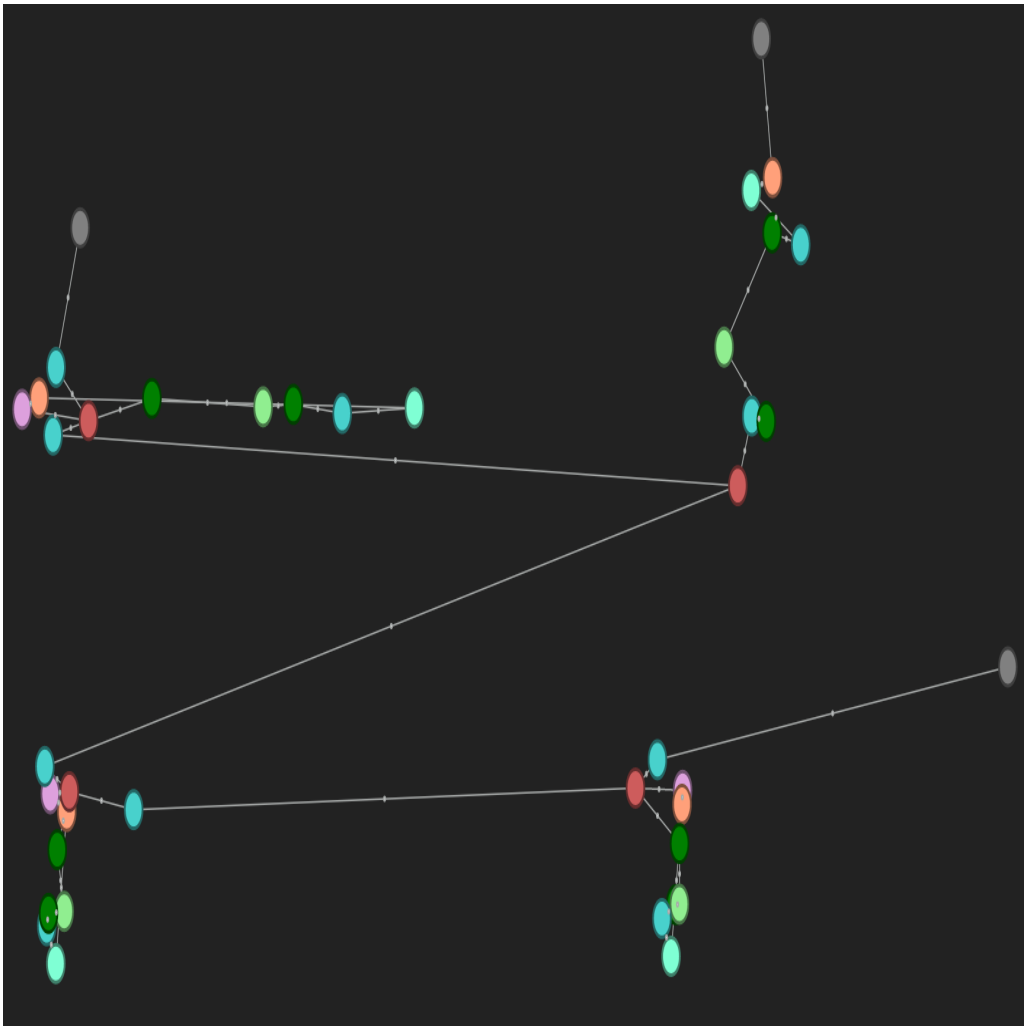


Figure 6.4: Undirected graph representation of PTL.

6.3.5: Extracted netlist.cir File

Along with the *plotly* graph, a new *.cir* netlist file is created in the working directory. This *SPICE*-like netlist contains placeholder quantity values and will be updated once InductEx has been called and the circuit has been simulated.

```
*=========================================================
*
*Netlist extracted by SPiRA for Inductex use on 2020-08-25
*
*=========================================================
P1 17 0
P2 63 0
P3 272 0
L1 421 431 2p
L2 272 431 2p
L3 71 438 2p
L4 438 63 2p
L5 485 17 2p
L6 485 476 2p
L7 71 471 2p
L8 421 71 2p
L9 415 0 0.1p
L10 428 0 0.1p
L11 441 0 0.1p
R1 471 476 1
JJ1 421 415 100
JJ2 431 428 100
JJ3 438 441 100
```

6.3.6: Extracted InductEx results

As is the case with the JTL's biasing resistor, the resistor and succeeding elements are not calculate due to the nature of how resistance is interpreted in superconducting electronics.

```
L1 421 431          9.46824p
L2 272 431          3.59331p
L3 71 438           4.30365p
L4 438 63           2.34948p
L7 71 471           3.17567p
L8 421 71           8.51064p
L9 415 0            0.45315p
L10 428 0           0.414489
L11 441 0           0.363842
JJ1 421 415         0.1539mA
JJ2 431 428         0.1539mA
JJ3 438 441         0.1539mA
```

6.4: Application of *SPiRA*

SPiRA is meant to serve as a step between simulation and initial design. Generating a PCell-based layouts helps shorten the time between initial and final design. *SPiRA*'s RDD allows the user to set up multiple Rule Decks and allows for interchangeability of fabrication processes.

By initializing logical/intuitive parameters, small changes such as changing specific inductor or junction sizes, can be brought forth by simply supplying the correct parameter upon generation. The dynamicity of the layout allows the components to be connected to one another by means of a specific port (regardless of it's location) rather than an X,Y-position. By routing elements by means of ports, the cell designer will not have to manually reroute the entire layout of a small change is made. This is meant to vastly reduce the time between initial design and chip sign-off.

Chapter 7: Conclusions and Recommendations

Conclusion:

This dissertation explains how the Layout vs. Schematic process can be done from hierarchical principles. *SPiRA* proposes a PCell-based synthesis framework which allows for full netlist parameter extraction with the help of a Rule Deck Database (RDD)[4] and simulation engines. The RDD is *SPiRA's* approach to encapsulating a Process Design Kit for a Superconducting Electronics. Existing software packages such as *Gmsh* (and a multitude of Python libraries) is used to generate an undirected graph which will be automatically parsed into an InductEx compatible netlist.

A Design Rule Checking scheme has been put in place by means of different *SPiRA* parameters, and the KLayout stand-alone DRC engine, allowing for dynamic as well as full post-layout DRC.

Once a full PCell layout has been extracted and has passed all the required DRC tests, a schematic can be generated directly from the newly created netlist file. This is done by means of a script generator that makes use of the drawing library, SchemDraw. This new feature brings *SPiRA* one step closer to being a full LVS solution.

Recommendation and Future Work

Future releases of *SPiRA* will contain a more robust schematic generator with less limitations than the current solution and adding features to show magnetic coupling between specified elements as well as hole and fill structures.

Due to the single-threaded finite element meshing engine, *Gmsh*, large layouts require long execution times. Making use of parallel *Gmsh* calls could vastly speed up the LVS process as the meshes are independent from one another.

Extraction of circuits containing two vias directly placed next to one another, will result in the first via being seen as an active inductor, to spite the fact that via resistances and inductances are considered negligible by software such as InductEx. This might lead to rank deficiency when doing simulations. This problem can be resolved by reconsidering the method which is used for via extraction, adding an edge case scenario where *SPiRA* will detect 2 or more vias placed next to one another.

Appendices

Appendix A1

Conference Paper - Layout versus Schematic with Design/Magnetic Rule Checking for Superconducting Integrated Circuit Layouts

Layout versus Schematic with Design/Magnetic Rule Checking for Superconducting Integrated Circuit Layouts

Ruben van Staden, Johannes A Delport, Johannes A Coetzee and Coenrad J Fourie
Department of Electrical and Electronic Engineering
Stellenbosch University
Stellenbosch, South Africa

***Abstract*—The IARPA SuperTools program has accelerated the development of superconductor integrated circuit design tools. Superconductor integrated circuits contain Josephson junctions and rely heavily on inductive interconnects and coupled inductors, all of which are not adequately supported by conventional semiconductor layout-versus-schematic verification (LVS) tools. Such circuits are also susceptible to failure in the presence of magnetic fields above about one tenth of the Earth's field strength and to magnetic flux trapped in layout structures during cool-down, so that magnetic rule checking (MRC) is essential. Under SuperTools we developed an open-source LVS framework, SPiRA, which allows for the parametric creation, alteration and verification of superconductor and quantum circuit layouts. SPiRA is a Python-based framework developed to aid the process of creating parameterized layouts while simultaneously taking into account design rule (DRC) as well as magnetic rule checking. SPiRA is designed to accept any process through a rule deck database (RDD) Python-based PDK schema from which cells are spawned as objects with inherent properties. This process allows rapid implementation of changes to layouts with the ability to extract an electrical netlist that can be simulated, and parameter extraction performed upon. SPiRA creates layouts in the GDSII layout format and allows quick visualization of the layout using the Gdspy library. We present extraction results for examples created parametrically with SPiRA, compare those to results for layouts created by hand and evaluate the capabilities of SPiRA. Finally we show how SPiRA improves models for inductance and compact model extraction with the inductance extraction tool InductEx.**

***Index Terms*—design automation, layout, superconducting devices**

I. Introduction

The history of layout design and automation thereof is plagued with various problems, though one that appears more often than not is the effort required to alter a layout once changes are made to the design or process [1]. Parameterized cells, or PCells, attempt to solve this issue through reuse of layouts across different fabrication processes by automatically adjusting the layout to fit the constraints [2]. These cells are not physical layouts but rather coded scripts that automatically generate layouts. The use of PCells is well documented and form part of large commercial tools such as Cadence Virtuoso through SKILL scripts [3]. PCells can further aid in the extraction of layout-versus-schematic (LVS) by using generated PCells, which are templates for device detection. Design/magnetic rule checks can be applied to each layout cell by restricting instance parameter values. Once a PCell has been created, the physical verification process requires very little human input to automatically adjust individual layout elements. Though automated layout generation has existed in commercial software suites for a few years now, the process is still quite unrefined and requires some work to accommodate superconducting integrated circuits. We present a new software tool, called SPiRA, which is a Python-based framework that enables LVS for superconducting circuits using a *templated-parameterized-methodology*.

The research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via the U.S. Army Research Office grant W911NF-17-1-0120

II. Layout-versus-schematic process optimization with PCells

Though traditional LVS tools merely validates the layout against its analogue netlist counterpart, SPiRA allows the automatic generation of layout elements based on a set of received parameters, by providing the user with a design environment in Python. This design environment dynamically connects to a Rule Deck Database (RDD) that essentially contains, as implied, the rules of the design process. The RDD is a novel approach to develop a generic Process Design Kit (PDK) that can be used for superconducting circuit design. The RDD schema leverages the industry standard Python programming language to use process related data in the SPiRA design environment. The user is then able to systematically create the the layout while the layout elements are validated against defined rules in the RDD using parameter restrictions. This essentially replaces the need for external DRC as the program immediately detects when rules are violated. The cell can then be saved as a Python script (PCell). SPiRA provides functions to produce the layout in GDSII [4] format or a usable SPICE netlist with all the relevant components and values. The resultant parameterized layout can be viewed using a native layout viewer.

A. *Parameterizing simulation models*

Superconducting circuits are highly susceptible to noise and external magnetic interference, therefore the coupling of electromagnetic and inductive structures are designed with the tight timing and current requirements of junctions in mind. By making use of simulation and inductance extraction tools, such as JoSIM [5] and InductEx [6], the inductance of each structure can be determined. The interactions between different layout elements can be extracted and changed to have different timing and current requirements. For example, InductEx receives a generated layout and extracts the values for each element. These values are then used by SPiRA to adjust the required element parameters until the desired lumped element value is extracted.

B. *How PCells speed up the LVS process*

Developing solutions for LVS verification and design rule checking involves effectively parsing the layout to a software domain. By definition a PCell layout is already in the software domain, but a hand-designed layout has to be parsed into a set of programmable objects.

One of the reasons for defining parameterized cells for LVS verification, is to use these PCells as templates for device detection. The more descriptive the PCell design, the more accurate the device extraction for LVS verification. Therefore, defined parameterized device cells are added to the LVS database in the RDD–also known as the LVS rule deck.

III. Available Software and Restrictions

Layout editors, many and varied as they may be (LASI, XIC, KLayout, etc) [7]–[10], have little to no support for automated layout generation using PCells. The best approximation to PCell implementation within a freely available layout editor is that found in KLayout which employs a set of macros for PCell generation. These tools also require the implementation of a separate process specific DRC file to check for design violations.

The available open source and freeware layout design tools are not designed to inherently support parameterized cells. Software such as XIC, rely heavily on open-source code and add-ons to complete DRC tasks. Seeing as circuit fabrication processes are constantly innovating, changing and creating new restraints to ensure optimal circuit operation margins, a flexible DRC system needs to be designed. Without a robust back-end that is designed to support a rule-deck that can be expanded and customized to suit any fabrication process, the change in any fabrication specifications will result in the redesign of all cells, rather than just changing the applicable parameters of the pcell.

IV. SPiRA Design Flow

SPiRA is a Python framework and therefore is not a program that can be executed, but rather a design environment for layout creation. However, certain modules can be separately executed on a GDSII layout, such as LVS verification. A basic design flow of what SPiRA intends to achieve is depicted in Figure 1. The PCell script calls functions from the framework to generate a layout and netlist in *.gds* and *.cir* formats respectively.

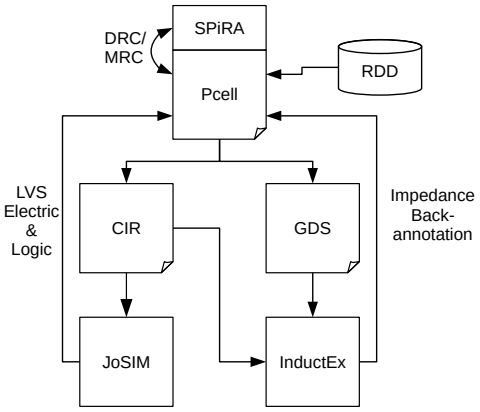


Fig. 1: SPiRA LVS Feedback loop

The impedance values of the layout is then extracted through InductEx and back-annotated into the PCell, which will then make the required adjustments. This back annotation also provides important information regarding the extracted magnetic effects for MRC verification. DRC and MRC happens during generation of the above files and warns the user of violations using the information provided by the RDD.

V. Example

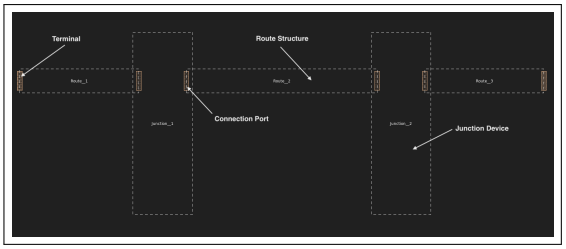
We demonstrate the capabilities of SPiRA by generating the layout of a basic Josephson Transmission Line (JTL) circuit along with the extracted netlist. This layout script is then fed to InductEx which extracts the corresponding inductance and Josephson junction values. We show only the defined parameters of the JTL cell. These three parameters control the width of each inductor branch. Their default values are `RDD.M6.MIN_SIZE` defined in the RDD file. These parameters are restricted to *upper* and *lower* value design rules. We show these inductors in Listing 1.

```
w1 = spira.NumberField(
    default=RDD.M6.MIN_SIZE, restriction=RestrictRange(
        lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH),
    doc='Width of left inductor.')
w2 = spira.NumberField(
    default=RDD.M6.MIN_SIZE, restriction=RestrictRange(
        lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH),
    doc='Width of middle inductor.')
w3 = spira.NumberField(
    default=RDD.M6.MIN_SIZE, restriction=RestrictRange(
        lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH),
    doc='Width of right inductor.')
```

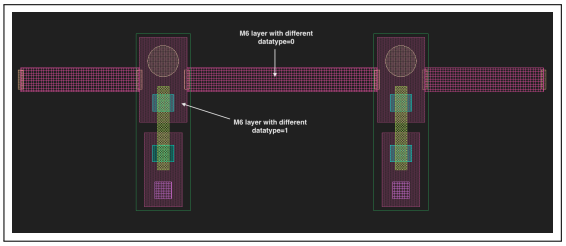
Listing 1: Excerpt of JTL PCell

The resulting layout shown in Fig. 2. The layout view shown in Fig. 2a illustrates the hierarchy of the created PCell. The

layout view in Fig. 2b shows the individual polygon elements represent in the JTL circuit.



(a) JTL parameterized cell, including the cell references used for construction.



(b) Flattened JTL parameterized cell illustrating all the polygon elements.

Fig. 2: Shows the boundaries of the cell references used to construct the junction device.

For the purpose of this exercise we have omitted the bias line to simplify the script. The resulting netlist only contains three inductors and two junctions. The junction layer radius is parameterized and can be dynamically changed, along with the shunt resistor width and length, seen in Listing 2.

```
In[1]: jtl.jj1
[SPiRA: Junction] (name 'JJ1', radius 1, length 2, width 1)
In[2]: jtl.jj1 = Junction(radius=2, length=3, width=0.8)
[SPiRA: Junction] (name 'JJ1', radius 2, length 3, width 0.8)
```

Listing 2: Junctions defined within the JTL PCell

The electrical netlist extracted from the JTL circuit is shown in Listing 3.

```
* JTL.pcell netlist
.subckt JTL 1 2
L1  1   3   6p
L2  3   4   12p
L3  4   2   6p
B1  3   0   jj   area=0.75
B2  4   0   jj   area=0.75
.ends JTL
```

Listing 3: Electrical netlist of the JTL

VI. Conclusion

A new design verification methodology was added to the design of superconductor circuits. Even though, parameterizing cells have been used to a limited extent, it has never been for full circuit design. Individual building blocks need to be designed according to a set of specifications. This is typically a physical design step involving mask layout and parameter extractions. Device design is still very much a part of superconductor circuit design. It is expected, however, that in the future this part of the design flow will be largely done by the fab, which will use it to define standard devices for its PDK. However, full-custom device design for circuits will still play an important role in the foreseeable future.

Designing a circuit layout can be done in an interactive environment that checks for parameter violations, while constructing each individual component of the full circuit. The single model methodology used by the SPiRA framework makes it easy to connect restrictions to defined parameters. These restrictions ensures that no DRC or MRC rule is violated.

The newly proposed rule deck database schema (RDD) leverages the Python programming language to effectively describe the process related details. This database allows the SPiRA framework to perform design rule checking upon layout generation, as well as magnetic rule checking through feedback from impedance extraction tools such as InductEx. Further, SPiRA supports netlist extraction when used in conjunction with circuit simulator JoSIM.

References

[1] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "The magic vlsi layout system," *IEEE Design & Test of Computers*, vol. 2, no. 1, pp. 19–30, 1985.

[2] J. Serras and L. M. Silveira, "Hierarchical analog layout migration with pcells," in *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 197–202, IEEE, 2009.

[3] V. Borisov, "Development of parameterized cell using cadence virtuoso," in *East-West Design & Test Symposium (EWDTS 2013)*, pp. 1–2, IEEE, 2013.

[4] N. B. Cobb and E. Y. Sahouria, "Hierarchical gdsii-based fracturing and job deck system," in *21st Annual BACUS Symposium on Photomask Technology*, vol. 4562, pp. 734–743, International Society for Optics and Photonics, 2002.

[5] J. A. Delport, K. Jackman, P. Le Roux, and C. J. Fourie, "Josim – superconductor spice simulator," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–5, 2019.

[6] C. J. Fourie, "Full-gate verification of superconducting integrated circuit layouts with inductex," *IEEE Transactions on Applied Superconductivity*, vol. 25, no. 1, pp. 1–9, 2014.

[7] D. E. Boyce, "LASI Home Site."

[8] L. P. Heulsman, "LASI - Windows Layout system for individuals," vol. 15, no. 2, pp. 4–22.

[9] S. R. Whiteley, "Xic Reference Manual." http://www.wrcad.com/manual/xicmanual/xicmanual.html, August 2018. Accessed: 2018-08-28.

[10] "KLayout - High Performance Layout Viewer And Editor."

Appendix A2

Conference Paper - Standard Cell Layout Synthesis for Row-Based Placement and Routing of RSFQ and AQFP Logic Families

Standard Cell Layout Synthesis for Row-Based Placement and Routing of RSFQ and AQFP Logic Families

Lieze Schindler
Dept. Elec. and Electron. Engineering
Stellenbosch University
Stellenbosch, 7600, South Africa
17528283@sun.ac.za

Ruben van Staden
Dept. Elec. and Electron. Engineering
Stellenbosch University
Stellenbosch, 7600, South Africa
rubenvanstaden@gmail.com

Coenrad J. Fourie
Dept. Elec. and Electron. Engineering
Stellenbosch University
Stellenbosch, 7600, South Africa
coenrad@sun.ac.za

Christopher L. Ayala
Institute of Advanced Sciences
Yokohama National University
Yokohama, 240-8501, Japan
ayala-christopher-pz@ynu.ac.jp

Johannes A. Coetzee
Dept. Elec. and Electron. Engineering
Stellenbosch University
Stellenbosch, 7600, South Africa
18288928@sun.ac.za

Tomoyuki Tanaka
Dept. Electrical and Comp. Engineering
Yokohama National University
Yokohama, 240-8501, Japan
tanaka-tomoyuki-wy@ynu.ac.jp

Ro Saito
Dept. Electrical and Comp. Engineering
Yokohama National University
Yokohama, 240-8501, Japan
saito-ro-mw@ynu.ac.jp

Nobuyuki Yoshikawa
Dept. Electrical and Comp. Engineering
Yokohama National University
Yokohama, 240-8501, Japan
nyoshi@ynu.ac.jp

***Abstract*—In this work under the IARPA SuperTools program we developed a layout synthesis tool with scripting support. The user specifies the relative positions of Josephson junctions and inductances constrained by a user-defined cell height and cell width. Tight integration with the three-dimensional inductance extraction tool, InductEx, allows inductances to be automatically generated while meeting reasonable design values. Based on these user inputs, the tool can synthesize the physical layout of logic cells for multiple SFQ circuit technologies according to design rules and layer parameters. Furthermore, it enables the straightforward regeneration of entire cell libraries when design rules change or when libraries have to be redesigned for more advanced fabrication processes. We describe the methodology of our synthesis tool and show the results applied to both RSFQ and AQFP logic families.**

***Index Terms*—layout synthesis, paramterized cells, superconductor circuits**

I. Introduction

The SuperTools research program funded by IARPA is a large development effort to produce electronic design automation (EDA) tools for the design of very-large-scale integration (VLSI) superconductor electronics [1]. Under the SuperTools program, the development of standard logic cell libraries that are compatible with automated placement and routing algorithms is required. Standard cell layouts need to conform to row dimensions, as well as to routing track pitch dimensions on which the place-and-route tools operate. The layout revision cost of entire logic cell libraries is inflated by the evolution of the MIT Lincoln Laboratory (MIT-LL) SFQ fabrication process due to changes in design rules such as minimum or maximum dimensions, spacing and surround values to change and layer parameters to shift [2].

We present a layout synthesis tool with scripting support, a subdivision of SPiRA [3], as an alternative to layout synthesis by hand. A set of user-defined parameters forms the basis of a parameterized cell (PCell). A PCell describes how layout elements must be generated according to defined parameters. SPiRA takes a set of user-defined parameters as input, processes the given parameters and automatically generates a layout in GDSII format. SPiRA integrates with InductEx for impedance extraction and has an integrated design rule checker to confirm that no design rules are broken within the layout. The fabrication process can be customized within the script providing the user with the option to simply update the parameters and regenerate the layout if the need arises.

II. Standard Cell Library

Standard RSFQ and AQFP logic cell libraries for layout synthesis have been developed for the ColdFlux project [4], which falls under the IARPA SuperTools program. The RSFQ library follows the fixed-height but variable width methodol-

The research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via the U.S. Army Research Office grant W911NF-17-1-0120, and the South African National Research Foundation, grant number 105859

978-1-7281-1196-4/19/$31.00 ©2019 IEEE

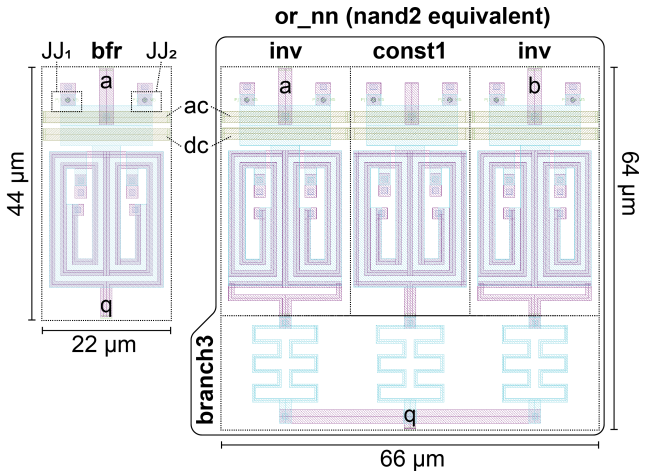


Fig. 1. Layout of the AQFP buffer (bfr) and NAND gate (or_nn). The critical current of both JJ_1 and JJ_2 are 50 μA. The NAND gate consists of two inverters (inv), a single constant-1 (const1) sub-cell, and a branch3 sub-cell which does an additive merge (majority operation) on the connected sub-cells. Note that the sub-cell footprints as well as the ac and dc rails are all uniform in the cell design so that sub-cells can be abutted together to form more complex Boolean logic cells.

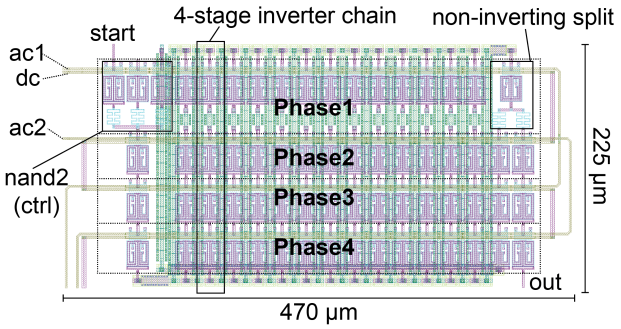


Fig. 2. Row-based layout of an AQFP inverting circular shift register consisting of 59 inverting stages. Cells clocked by the same clock phase co-exist on the same row and are abutted together to form the clock-power networks consisting of ac and dc microstrip lines in each row. A NAND2 gate provides a way to control the oscillator through an active-high 'start' signal.

ogy proposed in [5]. This allows for the cells to be placed in predefined rows for a more area-efficient chip design. This methodology allows for the use of CMOS-based place-and-route tools.

AQFP cells also follow a fixed-height variable-width methodology as shown in Fig. 1. First a set of sub-cells have been designed for AQFP, namely: bfr (buffer), inv (inverter), const1/0 (constant-1 or constant-0), and branch [6], [7]. Each sub-cell has been carefully sized so they can be abutted together to form Boolean logic gates which can be seen in Fig. 1 for the or_nn cell (OR gate with two negative inputs, equivalent to a nand2 gate). For example, the active sub-cells (bfr, inv, and const) are all the same size with identical pin positioning. Each of those active sub-cells have a standardized placement of the power-clock rails (ac and dc) such that by abutting sub-cells (or their higher-level compositions), a power-clock network forms in each row. This is applied in a row-based design of an AQFP inverting circular shift register consisting of 59 inverting stages in a feedback loop as shown in Fig. 2. Cells belonging to the same clock phase share the same row and are abutted together to form the power-clock rails for that row. Data propagates from one phase to the next (modulo 4) through stripline PTLs.

RSFQ cells are connected using passive transmission lines (PTLs) during the routing algorithm. PTL drivers and receivers are therefore required to ensure that a pulse propagates through a PTL. The RSFQ cell library includes additional cells with integrated PTL drivers and receivers. AQFP cells are also connected using PTLs which can be placed both manually or automatically through a channel routing algorithm. Unlike RSFQ cells, AQFP cells can directly drive and receive data using PTLs without a separate driver or receiver circuit. The PTLs are used to propagate positive (or negative) current as data instead of SFQ pulses. The drawback is that the current signals on long PTLs experience attenuation due to large parasitic inductance. This limits cell-to-cell PTL lengths to approximately 1 mm before the AQFP cannot correctly determine the logic state of the signal [7]. The solution for this is to insert another buffer (or several as needed) as a repeater to re-amplify the signal.

III. Layout Synthesis Methodology

Superconducting circuit designers have to currently lay out each cell by hand. This can be extremely time consuming, especially if the user is inexperienced or unfamiliar with the fabrication process. We are developing a tool, named SPiRA [8], which can automatically generate a cell layout utilizing user-defined parameters within a script. A complete introduction to SPiRA and its capabilities is presented in [3].

Cell layouts can be scripted as Python-based parameterized cells (PCells). PCells include user-defined information regarding cell width and height, junction sizes, junction placements, inductor values, width and placement along with port placements and other information required for impedance extraction. The tool uses the PCell script to generate a cell layout in GDSII format. The layout then undergoes design-rule-checking (DRC) and error feedback is collected and stored. If DRC errors are present within the layout, the layout synthesis tool gives error feedback to the user. If no DRC errors are detected, the layout is sent to InductEx [9] for impedance extraction. The results from InductEx can then be processed and the layout adjusted if the extracted values differ from the design values. The layout is once again checked for DRC errors. The iterative process continues until the extracted vales are within a certain tolerance specified by the user.

Once the extracted inductance and resistance values correlate with the designed values, the user can run the built-in SPiRA layout-versus-schematic (LVS) tool to extract the electrical schematic represented in the circuit layout. Electrical simulation can then be used to verify the operation of the circuit.

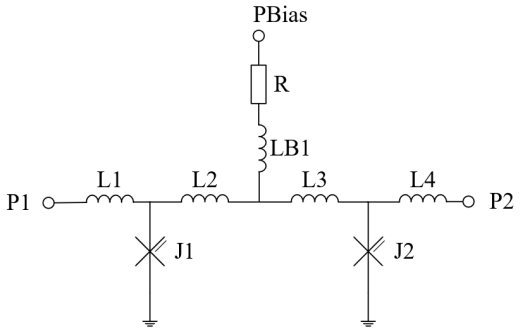


Fig. 3. Schematic of an RSFQ JTL with shunted JJs used as a reference for a layout script. Designed values are $L_1 = L_2 = L_3 = L_4 = 2$ pH, J_c=250 μA and I_b=350 μA (R=7.429 Ω with Vb=2.6 mV)

IV. Layout Script Example

We present an example layout script to show how a user can set up a SPiRA-compatible PCell. The script contains multiple user-defined parameters. It is important to note that SPiRA only generates the layout defined by the user through the PCell script. The current version of SPiRA does not improve the layout itself. The reliability of the design is only as good as the DRC and electrical rule checks as defined in the process design kit. Future work includes the integration with InductEx for automated impedance extraction feedback, and the implementation of this feedback to adjust the layout (for example inductor width) to better correlate the designed and extracted inductance and resistance values.

The following script provides an example of how a Josephson Transmission Line (JTL) PCell can be scripted with SPiRA compatibility. The schematic of the JTL is shown in Fig. 3.

```
class Jtl(spira.Circuit):
    p1 = spira.Parameter(fdef_name='create_p1')
    p2 = spira.Parameter(fdef_name='create_p2')
    p3 = spira.Parameter(fdef_name='create_p3')
    jj1 = spira.Parameter(fdef_name='
        create_jj_100sg_0')
    jj2 = spira.Parameter(fdef_name='
        create_jj_100sg_1')
    res0 = spira.Parameter(fdef_name='create_res_0')
    via_i5 = spira.Parameter(fdef_name='
        create_via_i5')

    def create_p1(self):
        return spira.Port(name='T1', midpoint
            =(-10,8), orientation=0, width=1)

    def create_p2(self):
        return spira.Port(name='T2', midpoint=(10,8)
            , orientation=180, width=1)

    def create_p3(self):
        return spira.Port(name='T3', midpoint=(0,28)
            , orientation=270, width=1.5)

    def create_jj_100sg_0(self):
        jj = dev.Junction(width=1, gnd_via=True,
            sky_via=True)
        T = spira.Translation((-3.4, 1.1)) + spira.
            Rotation(180)
        return spira.SRef(jj, transformation=T)

    def create_jj_100sg_1(self):
        jj = dev.Junction(width=1, gnd_via=True,
            sky_via=True)
        T = spira.Translation((3.4, 1.1)) + spira.
            Rotation(180)
        return spira.SRef(jj, transformation=T)

    def create_res_0(self):
        res = Resistor()
        T = spira.Translation((0, 15)) + spira.
            Rotation(90)
        return spira.SRef(res, transformation=T)

    def create_via_i5(self):
        via = dev.ViaI5()
        V = spira.SRef(via)
        V.connect(port=V.ports['M6_P2'], destination
            =self.res0.ports['M6_P4'])
        return V

    def create_structures(self, elems):
        elems += [self.jj0, self.jj1]
        elems += self.res0
        elems += self.via_i5
        return elems

    def create_routes(self, elems):

        elems += RouteManhattan(
            ports=[self.jj0.ports['M6_P1'], self.p1
                ],
            width=1, layer=RDD.PLAYER.M6.METAL,
            corners=self.corners)

        elems += RouteManhattan(
            ports=[self.jj1.ports['M6_P3'], self.p2
                ],
            width=1, layer=RDD.PLAYER.M6.METAL,
            corners=self.corners)

        elems += RouteStraight(p1=self.p3,
            p2=self.via_i5.ports['M5_P0'].copy(width
                =1.5),
            layer=RDD.PLAYER.M5.METAL)

        elems += RouteStraight(
            p1=self.res0.ports['M6_P2'].copy(width
                =2),
            p2=spira.Port(midpoint=(0,10),
                orientation=90, width=1, port_type='
                dummy'),
            width_type='sine',
            layer=RDD.PLAYER.M6.METAL)

        pl = spira.PortList()
        pl += self.jj0.ports['M6_P0']
        pl += spira.Port(midpoint=(0,10),
            orientation=180, port_type='dummy')
        pl += self.jj1.ports['M6_P0']
        elems += RouteManhattan(ports=pl, width=1,
            layer=RDD.PLAYER.M6.METAL, corners=self.
            corners)

        return elems

    def create_elements(self, elems):
        el = spira.ElementList()
        el += self.structures
        el += self.routes
        margin = 1
```

```
        box_shape = el.bbox_info.bounding_box(margin
            )
        elems += spira.Polygon(shape=box_shape,
            layer=spira.Layer(40))
        elems += spira.Polygon(shape=box_shape,
            layer=spira.Layer(70))
        return elems

    def create_ports(self, ports):
        ports += [self.p1, self.p2, self.p3]
        return ports
```

The script starts off by creating a JTL class. Three ports parameters are defined as `p1`, `p2` and `p3`. These ports are required by InductEx for impedance extraction and, for a JTL circuit, typically represent the input port, the output port and the biasing line port. The Josephson junction (JJ) parameters are then defined by `jj0` and `jj1`. The biasing resistor parameter, `res0`, and a via parameter, `via_i5`, are also defined.

The port parameters also specify a function definition, `fdef_name`. This function definition specifies the port name, the centre point, the width and orientation of the port. Similar functions define the size, placement and orientation of the JJ, biasing resistor and via structures.

The JJ, biasing resistor and via structures are then defined as elements to be connected together through inductors. These connections between structures are spawned through the `create_routes` function - which define routing elements between newly created ports (`M6_P1`, `M6_P3`, `M5_P0`, `M6_P2`) and ports defined at the beginning of the script (`p1`, `p2`, `p3`). The width of the routing is set to 1 μm.

The elements are then created and the polygons representing the ground and sky planes are calculated and created. The port elements are also created to complete the JTL class. The JTL class is called from the main script to create the GDSII layout file.

V. Preliminary Results

An example RSFQ JTL layout was scripted, as described in Section IV, and generated using SPiRA. The resulting layout of the JTL is shown in Fig. 4. The size, orientation and placement of junctions and the biasing resistor were specified. Inductor values were defined through the routing connections and width specifications. Port information, used by InductEx, was also included in the script. The layout underwent DRC and the layer density was extracted. SPiRA currently does not have the functionality to generate additional fill structures or moats within a layout to comply with layer density specifications, but the information can be applied by the user to adjust the PCell script.

InductEx was used for inductance and resistance extraction for the first iteration of layout generation. The extracted impedance values are compared to the designed values in Table I. It is seen that the values inductors L_1 and L_4 are much higher than the designed values. The dimensions of the biasing line resistor, R, also has to be adjusted to reduce the risk of circuit malfunction due to lowered biasing current. This

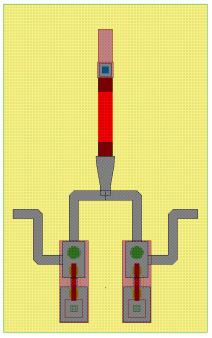


Fig. 4. Layout of JTL with ground plane generated by SPiRA. Extracted values are $L_1 = L_4 = 2.018$ pH, $L_2 = L_3 = 1.724$ pH, J_c=254 μA and I_b=334 μA (R=7.782 Ω with Vb=2.6 mV).

TABLE I
Comparison of Designed and Extracted Parameter Values for the First Iteration of Automated Layout Generation.

Parameter	Design value	Extracted value	% Difference
L1, L4	2.0 pH	3.011 pH	+50.57 %
L2, L3	2.0 pH	2.157 pH	+7.83 %
R_b	7.429 Ω	9.591 Ω	+29.10 %
I_b	350 μA	271.09 μA	-22.55 %

information is an example of necessary feedback to SPiRA required to improve the layout script.

The PCell scripts are not generated by SPiRA, but written by a user and this leads to the possibility of human error. The feedback acquired from inductance and resistance extraction can, in the future, be implemented by SPiRA to adjust the PCell script to better correlate the layout script with the designed inductance values without additional user input.

The main advantages of SPiRA is that a user only has to script a PCell once to represent a cell layout, regardless of the fabrication process. Several fabrication processes are supported by SPiRA and if the cell layout is required for a different fabrication process, the user can simply select a different fabrication process from the rule-deck-database and SPiRA can generate the cell layout for the specified process.

VI. Conclusion

A working layout synthesis tool with scripting support was presented. User-defined parameters are included within a script which SPiRA uses to generate a layout. The tool includes its own DRC and can integrate with InductEx for impedance extraction. The tool provides the user with the opportunity to fully customize the layout through a script, and the layouts can be regenerated with ease if adjustments to user-defined parameters are made.

Additionally, the exploration and co-optimization of logic cell libraries and the "quality-of-result" (QoR) of place-and-route tools can be aided with a layout synthesis tool such as SPiRA. Chip-level QoR is not just a place-and-route optimization problem but it is also influenced by standard cell design. With the synthesis tool we can hereafter explore

good pin placements within the cell, track pitch dimensions, and other standard cell properties by creating a collection of standard cell libraries and evaluating which place-and-route tools give the best QoR results for a set of benchmark circuits. Likewise, it aids in the development of place-and-route tools to try alternative algorithms or strategies that call for more experimental cell design, a feedback process that can be significantly sped up through cell layout synthesis.

REFERENCES

[1] IARPA SuperTools Program. [Online]. Available: https://www.iarpa.gov/index.php/research-programs/supertools

[2] S. K. Tolpygo, V. Bolkhovsky, T. J. Weir, C. J. Galbraith, L. M. Johnson, M. A. Gouker, and V. K. Semenov, "Inductance of Circuit Structures for MIT LL Superconductor Electronics Fabrication Process With 8 Niobium Layers," *IEEE Trans. Appl. Supercond.*, vol. 25, no. 3, June 2015, Art. no. 1100905.

[3] R. van Staden, J. A. Delport, J. A. Coetzee, and C. J. Fourie, "Layout versus schematic with design/magnetic rule checking for superconducting integrated circuit layouts," in *Extended Abstracts of 2019 International Superconductivity Electronics Conf.(Riverside, Los Angeles, 2019)*, 2019.

[4] C. J. Fourie, K. Jackman, M. M. Botha, S. Razmkhah, P. Febvre, C. L. Ayala, Q. Xu, N. Yoshikawa, E. Patrick, M. Law, Y. Wang, M. Annavaram, P. Beerel, S. Gupta, S. Nazarian, and M. Pedram, "Coldflux superconducting eda and tcad tools project: Overview and progress," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–7, Aug 2019, Art. no. 1300407.

[5] S. N. Shahsavani, T. Lin, A. Shafaei, C. J. Fourie, and M. Pedram, "An integrated row-based cell placement and interconnect synthesis tool for large sfq logic circuits," *IEEE Transactions on Applied Superconductivity*, vol. 27, no. 4, pp. 1–8, June 2017, Art. no. 1302008.

[6] N. Takeuchi, S. Nagasawa, F. China, T. Ando, M. Hidaka, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library designed using a 10 ka cm-2 niobium fabrication process," *Supercond. Sci. Technol.*, vol. 30, no. 3, p. 035002, Mar. 2017.

[7] N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library adopting minimalist design," *J. Appl. Phys.*, vol. 117, no. 17, p. 173912, May 2015. [Online]. Available: http://dx.doi.org/10.1063/1.4919838

[8] R. Van Staden, "SPiRA," 2019. [Online]. Available: https://spira.readthedocs.io/en/latest/

[9] C. J. Fourie, "Full-Gate Verification of Superconducting Integrated Circuit Layouts with InductEx," *IEEE Trans. Appl. Supercond.*, vol. 25, no. 1, February 2015, Art. no. 1300209.

Appendix B1:

PCell of RSFQ/AQFP Track Routing Design [1]

```
class Routing(spira.Circuit):
    m0_layer = spira.Parameter(fdef_name='create_m0_track')
    m1_layer = spira.Parameter(fdef_name='create_m1_track')
    m2_layer = spira.Parameter(fdef_name='create_m2_track')
    m4_layer = spira.Parameter(fdef_name='create_m4_track')
    i0_i2_layer = spira.Parameter(
                  fdef_name='create_i0_i2_track')
    i1_i3_layer = spira.Parameter(
                  fdef_name='create_i1_i3_track')
    via_i0_i2_square_length = spira.FloatParameter(restriction=
                              (spira.RDD.I0.MIN_SIZE,
                              spira.RDD.I0.MAX_SIZE))
    via_i1_i3_square_length = spira.FloatParameter(restriction=
                              (spira.RDD.I1.MIN_SIZE,
                              spira.RDD.I1.MAX_SIZE))

    def create_m0_track(self):
        ref = Metal_shape_1(layer_param =
              spira.RDD.PLAYER.M0.GND)
        scale = spira.Magnification(magnification=2)
        return spira.SRef(reference= ref,midpoint=(0,0))

    def create_m1_track(self):
        ref = Metal_shape_2(layer_param =
              spira.RDD.PLAYER.M1.GND)
        return spira.SRef(reference = ref,midpoint=(0,0))

    def create_m2_track(self):
        ref = Metal_shape_2(layer_param=
              spira.RDD.PLAYER.M2.GND)
        return spira.SRef(reference = ref,midpoint=(0,0))

    def create_m4_track(self):
        ref = Metal_shape_1(layer_param
              =spira.RDD.PLAYER.M4.GND)
        return spira.SRef(reference=ref,midpoint=(0,0))

    def create_i0_i2_track(self):
        f = I0_I2_via(i0_bool=True, i1_bool=True,
            via_i0_i2_square_length=
            self.via_i0_i2_square_length)
        return spira.SRef(reference=f,midpoint=(0,0))

    def create_i1_i3_track(self):
        f = I0_I2_via(i0_bool=True, i1_bool=True,
            via_i1_i3_square_length=
```

```
            self.via_i1_i3_square_length)
        return spira.SRef(reference=f,midpoint=(0,0))

    def create_elements(self,elems):
        elems += self.m0_layer
        elems += self.m1_layer
        elems += self.m2_layer
        elems += self.m4_layer
        elems += self.i0_i2_layer
        elems += self.i1_i3_layer
        return elems
```

The **Routing()** is the main class containing the sub-devices for each layer and pairing vias. The image below shows the full cell with via_square_length set to 0.6.

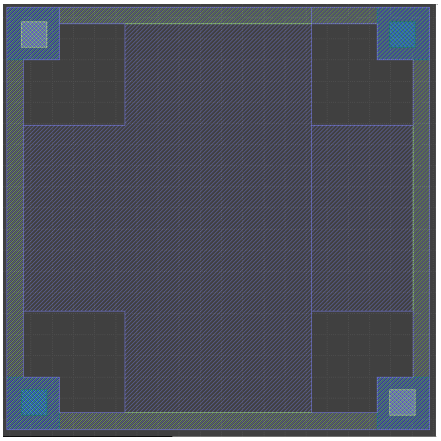
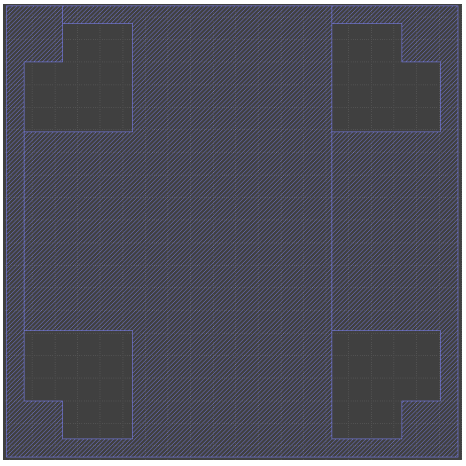


Figure C1: Full track routing cell.

```
class Metal_shape_1(spira.Cell):
    layer_param = spira.LayerParameter()

    def __create_elements__(self,elems):
        elems += spira.Cross(layer = self.layer_param,
                 box_size=9.2,thickness=4.4)
        elems += spira.Box(layer =
                 self.layer_param,width=10,
                 height=0.4,center=(0,4.8))
        elems += spira.Box(layer=self.layer_param, width=10,
                 height=0.4, center=(0, -4.8))
        elems += spira.Box(layer=self.layer_param, width=0.4,
                 height=10, center=(4.8, 0))
        elems += spira.Box(layer=self.layer_param, width=0.4,
                 height=10, center=(-4.8, 0))
        elems += spira.Box(layer= self.layer_param,width=1.25,
                 height=1.25, center=(-4.375,4.375))
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(-4.375, -4.375))
```

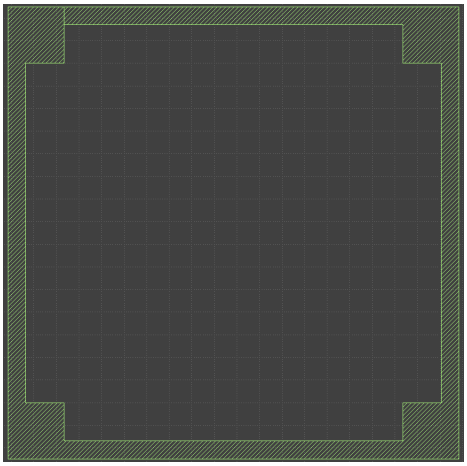
```
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(4.375, 4.375))
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(4.375, -4.375))
        return elems
```

The **Metal_shape_1()** cell is used to generate the base shape for M0 and M4.



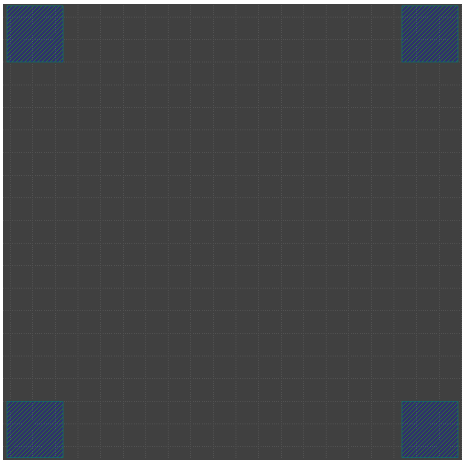
```
class Metal_shape_2(spira.Device):
    layer_param = spira.LayerParameter()

    def __create_elements__(self, elems):
        elems += spira.Box(layer = self.layer_param, width=10,
                 height=0.4, center=(0, 4.8))
        elems += spira.Box(layer=self.layer_param, width=10,
                 height=0.4, center=(0, -4.8))
        elems += spira.Box(layer=self.layer_param, width=0.4,
                 height=10, center=(4.8, 0))
        elems += spira.Box(layer=self.layer_param, width=0.4,
                 height=10, center=(-4.8, 0))
        elems += spira.Box(layer= self.layer_param, width=1.25,
                 height=1.25, center=(-4.375, 4.375))
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(-4.375, -4.375))
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(4.375, 4.375))
        elems += spira.Box(layer=self.layer_param, width=1.25,
                 height=1.25, center=(4.375, -4.375))
        return elems
```

The **Metal_shape_2()** cell is used to generate the outer shape used on M2.



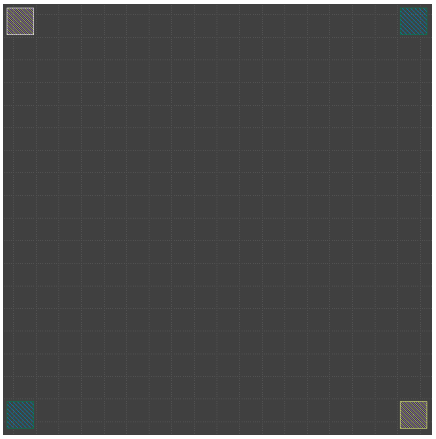
```
class Metal_shape_3(spira.Device):
    layer_param = spira.LayerParameter()

    def __create_elements__(self, elems):
        elems += spira.Box(layer = self.layer_param,
                 width=1.25,height=1.25,center=(-4.375,4.375))
        elems += spira.Box(layer = self.layer_param,
                 width=1.25,height=1.25,center=(4.375,4.375))
        elems += spira.Box(layer=self.layer_param,
                 width=1.25,height=1.25,center=(-4.375,-4.375))
        elems += spira.Box(layer = self.layer_param,
                 width=1.25,height=1.25,center=(4.375,-4.375))
        return elems
```

The **Metal_shape_3()** cell generates the pillars located on M1 and M3.

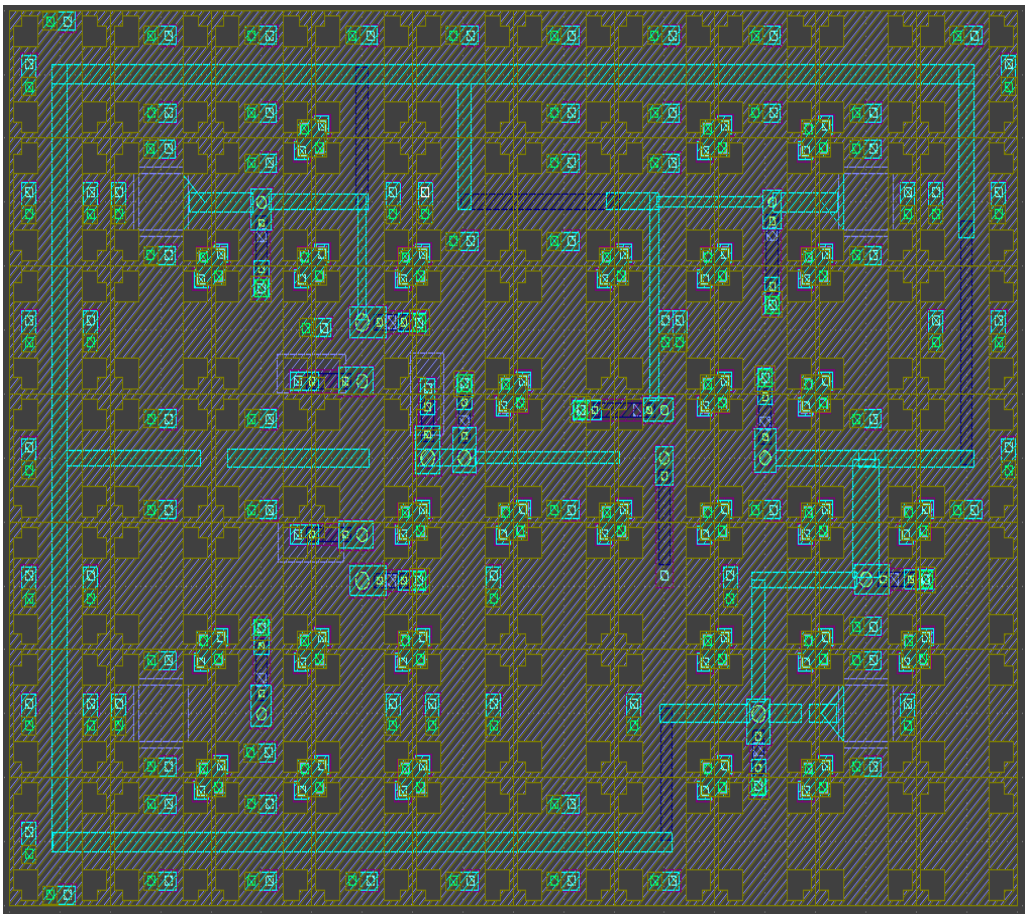


```
class I0_I2_via(spira.Device):
    i0_bool = spira.BoolParameter()
    i1_bool = spira.BoolParameter()
    via_i0_i2_square_length = spira.FloatParameter(
                              restriction=(spira.RDD.I0
                              .MIN_SIZE,spira.RDD.I0.MAX_SIZE))

    via_i1_i3_square_length = spira.FloatParameter(
                              restriction=(spira.RDD.I1.
                              MIN_SIZE,spira.RDD.I1.MAX_SIZE))

    def __create_elements__(self, elems):
        if self.i0_bool == True:
            elems += spira.Box(layer = spira.RDD.PLAYER.I0.VIA,
                     width=self.via_i0_i2_square_length,
                     height=self.via_i0_i2_square_length,
                     center=(-4.35,4.35))
            elems += spira.Box(layer=spira.RDD.PLAYER.I0.VIA,
                     width=self.via_i0_i2_square_length,
                     height=self.via_i0_i2_square_length,
                     center=(4.35, -4.35))
            elems += spira.Box(layer = spira.RDD.PLAYER.I2.VIA,
                     width=self.via_i0_i2_square_length,
                     height=self.via_i0_i2_square_length,
                     center=(-4.35,4.35))
            elems += spira.Box(layer=spira.RDD.PLAYER.I2.VIA,
                     width=self.via_i0_i2_square_length,
                     height=self.via_i0_i2_square_length,
                     center=(4.35, -4.35))
        if self.i1_bool == True:
            elems += spira.Box(layer = spira.RDD.PLAYER.I1.VIA,
                     width=self.via_i1_i3_square_length,
                     height=self.via_i1_i3_square_length,
                     center = (4.35,4.35))
            elems += spira.Box(layer=spira.RDD.PLAYER.I1.VIA,
                     width=self.via_i1_i3_square_length,
                     height=self.via_i1_i3_square_length,
                     center=(-4.35, -4.35))
            elems += spira.Box(layer = spira.RDD.PLAYER.I3.VIA
                     ,width=self.via_i1_i3_square_length,
                     height=self.via_i1_i3_square_length,
                     center = (4.35,4.35))
            elems += spira.Box(layer=spira.RDD.PLAYER.I3.VIA,
                     width=self.via_i1_i3_square_length,
                     height=self.via_i1_i3_square_length,
                     center=(-4.35, -4.35))
        return elems
```

The **I0_I2_via()** function allows the I0/I2 or I1/I3 pair to be disabled by means of the 'i0_bool' and 'i1_bool' Boolean parameter. The float parameters created are used to adjust the size of the via polygons.



Appendix C: Usage of Track Routing in a full Logic Gate[2]



Appendix C: Schematic Generator Functions

Find_members()

The **find_members()** function receives a list with shape [1,3], containing the element name and the two nodes (representing the terminals of the element) present in the netlist. (Eg. L1 1 2). This function will go through the netlist and find any members connected to the sent node and return a [N members,3] list.

```
def find_members(sent_node):

    tempnode = []
    node = sent_node
    memberlist = []
    try:
        if "P" in node[0]:
            relation_node = node[1]
            for n in netlist:
                if n[0] == node[0]:
                    continue
                else:
                    n1 = n[1]
                    n2 = n[2]

                    if n2 == relation_node or
                    n1 == relation_node:
                        memberlist.append([n[0],n[1],n[2]])

        if "L" in node[0] or "R" in node[0] or "J" in node[0]:

            relation_node1 = node[1]
            relation_node2 = node[2]
            for n in netlist:

                if "P" in n[0]:

                    if n[0] != node[0]:

                        leg1, leg2 = n[1],n[2]
                        if leg1 == node[1] or leg2 == node[1]
                        or leg1 == node[2] or leg2 == node[2]:
                            memberlist.append(n)

                if n[0] == node[0] or "P" in n[0]:
                    continue
                else:
                    n1 = n[1]
                    n2 = n[2]

                    if n1 == relation_node1
                    or n1 == relation_node2
```

```
                    or n2 == relation_node1
                    or n2 == relation_node2:
                        memberlist.append([n[0],n[1],n[2]])

    except Exception as e:
        print(e)

    return memberlist
```

Draw_till_terminal()

```
def new_draw_till_terminal(node,predecessor,other_branch_member
    ,direction):
    list_to_be_drawn = []
    list_to_be_drawn.append(node)
    members = find_members(node)

    members.remove(predecessor)

    while 'P' not in list_to_be_drawn[-1][0]:
        for m in members:
            if m not in list_to_be_drawn:
                if 'P' in m[0]:
                    list_to_be_drawn.append(m)
                if 'L' in m[0]:
                    list_to_be_drawn.append(m)
                if 'J' in m[0]:
                    list_to_be_drawn.append(m)
                if 'R' in m[0]:
                    list_to_be_drawn.append(m)


        list_to_be_drawn = remove_drawn(list_to_be_drawn)

        members = find_members(list_to_be_drawn[-1])

    for r in list_to_be_drawn:
        if 'P' in r[0]:
            leg1,leg2 = r[1],r[2]
            for d in list_to_be_drawn:
                if d == r:
                    continue
                dleg1,dleg2 = d[1],d[2]
                if dleg1 == leg1 and dleg1 != 0:
                  continue
                else:
                    if 'P' in d[0]:
                        list_to_be_drawn.remove(d)

    return list_to_be_drawn
```

Bibliography

[1] C. J. Fourie, C. L. Ayala, L. Schindler, T. Tanaka, and N. Yoshikawa, "Design and characterization of track routing architecture for rsfq and aqfp circuits in a multilayer process," *IEEE Transactions on Applied Superconductivity*, vol. 30, no. 6, pp. 1–9, 2020.

[2] L. Schindler, "The Development and Characterisation of a Parameterised RSFQ Cell Library for Layout Synthesis, PhD Thesis, University of Stellenbosch," 2020.

[3] "History of Superconductors." http://cesur.en.ankara.edu.tr/history-of-superconductors/. Accessed: 2020-06-29.

[4] R. Van Staden, "A physical design verification framework for superconducting electronics," 2019.

[5] C. J. Fourie, K. Jackman, M. M. Botha, S. Razmkhah, P. Febvre, C. L. Ayala, Q. Xu, N. Yoshikawa, E. Patrick, M. Law, Y. Wang, M. Annavaram, P. Beerel, S. Gupta, S. Nazarian, and M. Pedram, "Coldflux superconducting eda and tcad tools project: Overview and progress," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–7, 2019.

[6] A. Kuehlmann, F. Somenzi, C.-J. Hsu, and D. Bustan, "Equivalence checking," *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology: Circuit Design, and Process Technology*, p. 77, 2016.

[7] A. Fayyazi, S. Nazarian, and M. Pedram, "qec: A logical equivalence checking framework targeting sfq superconducting circuits," pp. 1–3, 07 2019.

[8] "A system for sequential synthesis and verification."

[9] L. Schindler, R. van Staden, C. J. Fourie, C. L. Ayala, J. A. Coetzee, T. Tanaka, R. Saito, and N. Yoshikawa, "Standard Cell Layout Synthesis for Row-Based Placement and Routing of RSFQ and AQFP Logic Families," in *2019 IEEE International Superconductive Electronics Conference (ISEC)*, pp. 1–5, 2019.

[10] R. van Staden, J. A. Delport, J. A. Coetzee, and C. J. Fourie, "Layout versus schematic with design/magnetic rule checking for superconducting integrated circuit layouts," in *2019 IEEE International Superconductive Electronics Conference (ISEC)*, pp. 1–3, 2019.

[11] H. Huang, X. Wang, E. Tervoort, G. Zeng, T. Liu, X. Chen, A. Sologubenko, and M. Niederberger, "Nano-sized structurally disordered metal oxide composite aerogels as high-power anodes in hybrid supercapacitors," *ACS nano*, vol. 12, no. 3, pp. 2753–2763, 2018.

[12] S. K. Tolpygo, V. Bolkhovsky, T. Weir, C. Galbraith, L. Johnson, M. A. Gouker, and V. Semenov, "Inductance of circuit structures for mit ll superconductor electronics fabrication process with 8 niobium layers," *IEEE Transactions on Applied Superconductivity*, vol. 25, pp. 1–5, 2015.

[13] S. R. Whiteley, "Pcell history and status." http://ftp.srware.com/manual/xicmanual/node123.html, 2019. Accessed: 2020-07-14.

[14] C. J. Fourie, A. Takahashi, and N. Yoshikawa, "Fast and accurate inductance and coupling calculation for a multi-layer nb process," *Superconductor Science and Technology*, vol. 28, no. 3, p. 035013, 2015.

[15] "Gdsii stream format manual." Accessed 2020-07-24.

[16] M. Köfferlein, "Klayout," 2018.

[17] C. J. Fourie, "Inductex," 2012.

[18] A. Johnson, "Clipper library."

[19] O. Mukhanov, V. Semenov, and K. Likharev, "Ultimate performance of the rsfq logic circuits," *IEEE Transactions on Magnetics*, vol. 23, no. 2, pp. 759–762, 1987.

[20] H. Herbst, "Gate-Level Superconductor Integrated Circuit Fabrication Process Modelling for Improved Layout Extraction, MEng Thesis, Stellenbosch University," 2020.

[21] S. K. Tolpygo, V. Bolkhovsky, T. Weir, C. Galbraith, L. M. Johnson, M. A. Gouker, and V. K. Semenov, "Inductance of circuit structures for mit ll superconductor electronics fabrication process with 8 niobium layers," *IEEE Transactions on Applied Superconductivity*, vol. 25, no. 3, pp. 1–5, 2014.

[22] J. A. Delport, K. Jackman, P. le Roux, and C. J. Fourie, "Josimâsuperconductor spice simulator," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–5, 2019.

[23] C. J. Delker, "SchemDraw." https://pypi.org/project/schemdraw/.