A Physical Design Verification Framework for Superconducting Electronics

by

Ruben van Staden

Dissertation presented for the degree of Doctor of Philosophy in Electrical and Electronic Engineering in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof. C. J. Fourie

December 2019

Declaration

By submitting this dissertation 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: December 2019

Abstract

A Physical Design Verification Framework for Superconducting Electronics

R. van Staden

Dissertation: PhD

December 2019

A new parameterized methodology for solving physical design verification for superconductor digital electronics (SDE) is proposed. Circuit verification tools forms an important part in integrated circuit (IC) design. This dissertation introduces a new physical verification framework, called SPiRA, that is composed of three modules: Parameterized Cells (PCells), Design Rule Checking (DRC) and Layout-versus-Schematic (LVS). These form part of the electronic design automation (EDA) verification toolchain for SDE. The proposed framework uses a combination of Python and metaprogramming to create a modular approach to verifying SDE circuits. The goal of the PCells framework is to create efficient layout generators for superconductor electronics, while at the same time checking for design rule violations. The LVS module is responsible for verifying if the designed circuit layout corresponds to the original simulated schematic. Parameter extraction for superconductor circuit technologies, such as single flux quantum (SFQ), requires an input netlist that corresponds to the circuit layout. The parameter extraction model is only as good as the given netlist, which makes LVS an essential piece in the parameter extraction phase. The proposed LVS module uses a parameterized-hierarchical methodology, which is process independent. The framework is capable of supporting any kind of superconducting- or quantum circuit technology, such as Rapid-Single-Flux-Quantum (RSFQ), Energy-effiecient RSFQ (ERSFQ, eSFQ) or Adiabatic Quantum Flux Parametron (AQFP).

Uittreksel

'n Raamwerk vir die Verifikasie van Fisiese Ontwerp vir Supergeleier Elektronika

("A Physical Design Verification Framework for Superconducting Electronics")

R. van Staden

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa. Proefskrif: PhD

Desember 2019

'n Nuwe geparameteriseerde metode vir die oplos van fisiese ontwerp verifikasie vir supergeleier digitale elektronika (SDE) word voorgestel. Stroombaan verifikasie gereedskap vorm 'n belangrike deel in die geïntegreerde stroombaan (IC) ontwerp. Hierdie desertasie stel bekend 'n nuwe fisiese verifikasie raamwerk, genaamd SPiRA, wat bestaan uit drie modules: "Parameterized Cells (PCells)", "Design Rule Checking (DRC)", en "Layout-versus-Schematic (LVS)". Dit maak deel van die elektroniese ontwerp outmatisering (EDA) verifikasie sagteware vir SDE. Die voorgestelde raamwerk gebruik 'n kombinasie van Python en meta-programming om 'n modulere benadering te skep om SDE-stroombane te verifieer. Die doel van die PCell-raamwerk is om doeltreffende uitleggenerators vir supergeleier electronika te skep, terwyl ontwerpreël oortredings terselfdetyd nagegaan word.

Die LVS-module is verantwoordelik vir die verifiëring van die ontwerpuitleg en bepaal of dit ooreenstem met die oorspronklike gesimuleerde uitleg.

Parameteronttrekking vir supergeleier stroombaan tegnologie, soos "Single Flux Quantum (SFQ)", vereis dat 'n stroombaan ooreenstem met die fisiese uitleg. Die akkuraatheid van die parameter ekstraksie model word grootliks bepaal deur die gegewe stroombaan, daarom is LVS 'n noodsaaklike aspek in die parameter ontrekking fase.

Die voorgestelde LVS-module gebruik 'n parameter-hierargiese metodologie, wat proses onafhanklik is. Die raamwerk is in staat om enige soort supergeleidende stroombaan tegnologie te ondersteun, soos *Rapid-Single-Flux-Quantum* (RSFQ), Energie-effektiewe RSFQ (ERSFQ, eSFQ) of *Adiabatic Quantum Flux Parametron* (AQFP).

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations: My supervisor Professor Coenrad J. Fourie. Thank you for your patience and faith in times when even I was doubtful. Thank you for the travels around the world, it has been a journey the past seven years we have worked together.

Thank you Kyle Jackman and Joey Delport for your showing extreme discipline in any task we took on. Also, thank you to Lieze Schindler for layouts and testing. Finally, I would like the Intelligence Advanced Research Projects Activity (IARPA) for financing my post-graduate studies.

Contents

List of Figures

Acronyms

ADP Advanced Process
AIST Advanced Industrial Science and Technology
AQFP Adiabatic Quantum Flux Parameteron
CAD Computer Aided Design
DFF D Flip-Flop
DRC Design Rule Checking
EDA Electronic Design Automation
ERC Electrical Rule Checking
ERSFQ Energy-Efficient Rapid Single Flux Quantum
eSFQ Energy-Efficient Rapid Single Flux Quantum
IARPA Intelligence Advanced Research Projects Activity
IC Integrated Circuit
IPHT Institute of Photonic Technology
JJ Josephson Junction
JTL Josephson Transmission Line
JTLT Josephson Transmission Line Transmitter
LDF Layer Definition File
LVS Layout-versus-Schematic
MITLL MIT Lincoln Laboratory
PCell Parameterized Cell
PDK Process Design Kit
PDV Physical Design Verification
PTLRX Passive Transmission Line
RDD Rule Deck Database
RSFQ Rapid Single Flux Quantum
RQL Reciprocal Quantum Logic
SDE Superconductor Digital Electronics
SDP Standard Process
SFQ Single Flux Quantum

SIC Superconducting Integrated Circuit

S-EDA Superconductor Electronic Design Automation

VLSI Very Large Scale Integrated

Chapter 1

Introduction

Significant improvements have been made in the field of Superconducting Digital Electronics (SDE) technologies, such as Single Flux Quantum Electronics (SFQ). More specifically eSFQ, ERSFQ, AQFP, and RQL, have led to an increasing demand for the development of low-power technologies [3], [4], [5]. Recently an IARPA funded project, called SuperTools, was initiated to help with the development of software for the SFQ field [6].

The rapid development in very large scale integrated (VSLI) circuit in the semiconductor industry has largely been influenced by utilizing a large variety of software tools and methodologies which are defined as computer-aided design (CAD). The current state of CAD software for the superconductor industry is depicted in [7], with focus falling largely on numerical solvers and physical parameter extraction [8], [9], [10].

As the technology progress and the complexity of SDE circuit designs increase, it is no longer feasible to perform extractions by hand. Therefore, automated layout extraction software is required to accelerate the design process of large-scale SDE circuits.

1.1 SDE Fabrication Processes

The current superconductor fabrication industry is largely comprised of five fabrication processes: D-wave Systems, MIT Lincoln Lab, AIST (ADP and SDP), Hypres, and IPHT. Due to SDE still being its early phase, the data available from fabrication facilities is extremely limited.

The fabrication of superconducting integrated circuits comprises the deposition and etching of various layers of different materials on top of each other. These layers are either deposited dielectric layers, contact layers or metal layers. In SDE, contact layers represents via or Josephson junction structures, while metal layers can be either superconductive or resistive. Each layer type has a set of different design characteristics specific to the process, such as their width and minimum size. Each process also has a set of design rules that

restricts the user from designing layouts that have a high chance of failure due to manufacturing errors. Process related data are stored in a separate set of files, called a Process Design Kit (PDK).

MIT Lincoln Laboratory (MITLL) fabrication process [11] for superconductor electronics is a niobium-based integrated circuit technology appropriate for RSFQ, RQL, ERSFQ, and other superconductor electronic circuits. The MITLL process is used as the base process in this dissertation. It is based on Nb/Al-AlOx/Nb tri-layer technology with a critical current density, J_c, of $100\,\mu A/\mu m^2$. It utilises $248\,nm$ photolithography and planarization with chemical-mechanical polishing (CMP) for wiring-layer feature sizes down to $350\,nm$ and Josephson junction diameters down to $700\,nm$.

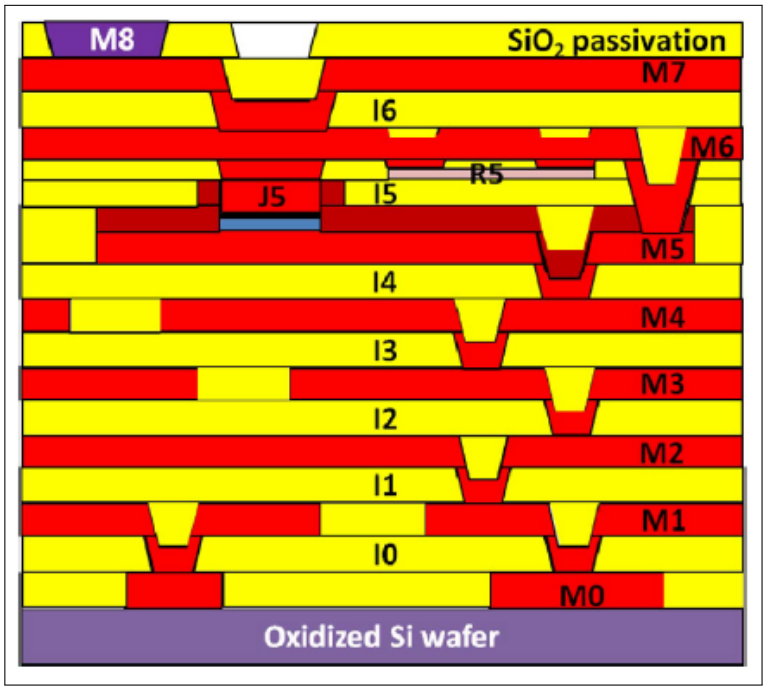


Figure 1.1: Cross-section of 8-metal-layer (8M) fabrication process developed at MITLL [1]

The MITLL process consists of eight superconducting layers (M_0-M_7), eight contact layers (I_0-I_7), one junction layer, and one resistive layer. Layer M_4 is the ground-plane layer, M_7 is the sky-plane layer, and layers M_5 and M_6 are considered to the two main inductive wiring layers. Layer R_5 is the resistive layer, and J_5 is the Josephson junction layer. A cross-sectional view of this process is depicted in Fig. 1.1.

1.2 Integrated Circuit Design Flow

The semiconductor design process has made significant progress over the last few decades. With the invention of modern scripting languages, such as

Python, new software packages have recently been developed [12]. Integrated circuit design engineers rely heavily on software to prove the stability of their designs. Designing an integrated circuit (IC) requires several steps [13] and can be broken down into the following categories:

- *Circuit simulation*: Simulate the logical behaviour of a circuit.
- *Layout design*: The design of a physical layout of a circuit.
- *Rule checking*: Checks whether any design rules specified by the fabrication process is violated. This step consists of Electrical Rule Checking (ERC) and Design Rule Checking (DRC).
- *Layout versus Schematic*: The topology of the circuit layout is compared to that of the circuit schematic. Layout-versus-Schematic consists of two part: netlist extraction and netlist comparison.
- *Parameter extraction*: The parameters of the layout is extracted and compared to the designed circuit parameters.

1.3 Design Environment

An IC design environment is an environment were all the design software are meticulously integrated into a single workspace. This workspace includes all steps in designing a circuit layout, from creating the physical layout, to layout verification, to electrical simulations. In an IC design environment a circuit design is done from the cell level, which requires defining individual layout components, such as transistors in semiconductor circuits and Josephson junctions in superconductor circuits.

The IC design industry for semiconductor electronics have grown in complexity over the past few decades [14]. As any technology evolves new software solutions to fit the changing needs of the technology are developed. Standalone tools can be used to create a custom design environment, but in industry commercial tools are typically used that offers a fully featured design environment [15].

Once a circuit schematic has been designed and simulated to validate its logic operations, a layout engineer can design a physical layout representation (also called a cell) of the circuit schematic. The physical layout can be designed using one of the following methods available in an IC design environment:

1.3.1 Hand-designed Layout

A hand-designed layout, or hand-crafted layout, is created using the GDSII file format in conjunction with a layout editor. Polygon objects are placed to represent the presence or absence of layers [16]. These layers are defined by

the specific fabrication process, which includes representing conducting layers (metals) and contacting layers (vias).

1.3.2 Parameterized Layout

A parameterized layout, or parameterized cell (PCell), is created using a scripting language. The design patterns to create a PCell is similar to that of creating a hand-designed layout. However, by creating a PCell a designer has more control over the design, since polygon coordinates are programmatically defined and layout components can be parameterized.

1.4 Physical Design Verification Overview

Physical Design Verification (PDV) is a set of validation tests applied to a layout. Traditionally, PDV operations are performed after a layout has been designed and consists of the following two steps:

1.4.1 Design Rule Checking

Design rules are layout restrictions that ensure that the manufactured circuit will not violate any process rules. These rules are based on known parameters of the manufacturing process, to ensure a margin of safety and to allow for process tolerances.

1.4.2 Layout-versus-Schematic

The purpose of Layout-versus-Schematic (LVS) is to ensure that the designed physical layout represents the designed circuit schematic. LVS verification is the final step in the PDV process. Sub-circuit extraction is the key problem to be solved in LVS. Graph theory techniques, such as sub-graph isomorphism [17], are used to check layout correctness. A graph-based approach for LVS verification first requires extracting a netlist from the layout, using a netlist extractor tool [16]. This netlist can then be visually presented to the layout designer in a graph format, which can then be manually compared to the schematic netlist for layout correctness. This comparison process can be automated using sub-graph isomorphic checks.

1.5 Semiconductor vs SDE Layout Verification

Semiconductor circuits uses resistors as its primary lump element with the active component being the transistor. Components in a semiconductor circuit are connected using planar resistive polygon structures. Very few conducting polygons with the same process layer are interconnected between different

components. Therefore, node detection algorithms when extracting a layout graph are specified by component port objects.

Superconductor circuits uses inductors as the primary lump element with the active component being the Josephson junction. Components in a superconductor circuit are connected using inductive polygon structures. Ground return currents plays an important role in SDE electronics, which is not present in semiconductor circuits. SDE verification has to be able to extract complex inductance networks that must take ground return currents into account [18]. Therefore, traditional node finding algorithms for graph creation as used for semiconductor circuits is not applicable for SDE circuits.

1.5.1 Branch Detection for Circuit Netlist Extraction

Inductance extraction plays a far bigger role in SDE compared to semiconductor electronics. Different inductors in a SDE circuit can interconnect to create complex polygon structures, which makes identifying individual inductor branches much more difficult than is the case with semiconductor layouts [19].

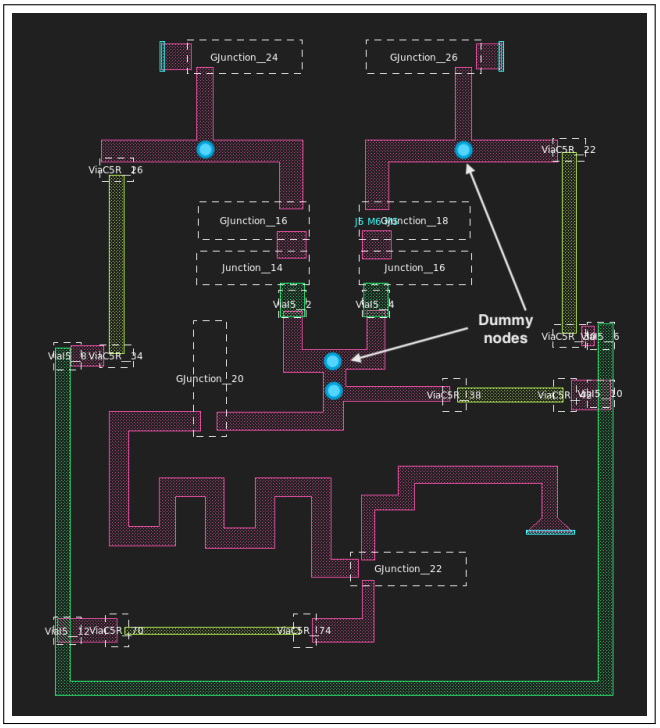


Figure 1.2: RSFQ circuit showing the manually added *dummy nodes*.

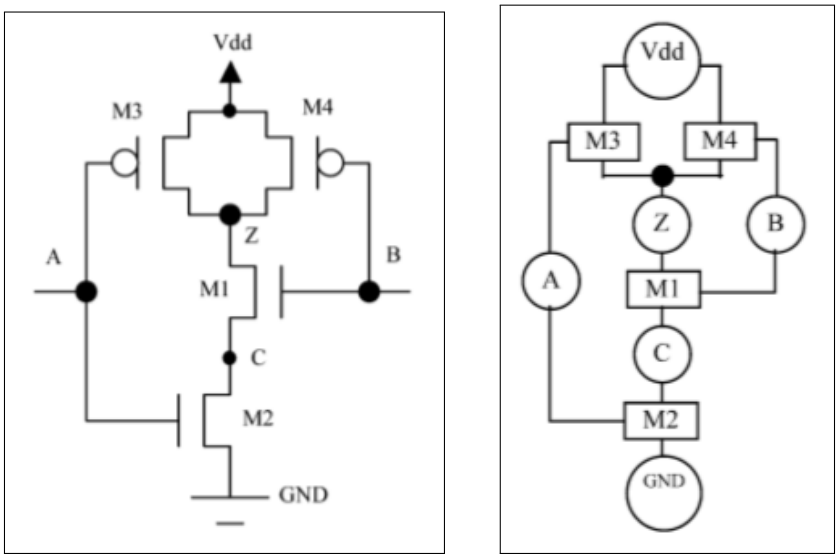
In 2015 a new method for solving LVS specific to SDE was proposed [20], but still requires manual layout editing by the designer to distinguish inductor branches. A layer with a unique GDSII number, called a *dummy node*, has

to be added on a metal polygon to represent a branch cross-over, as shown in Fig. 1.2.

1.5.2 Device Netlist Extraction

In semiconductor technology it is common to represent components (transistors) and ports as a single node in a graph when doing netlist extraction. Graph edges connecting these nodes represent the routing topology.

A semiconductor circuit graph contains two types of nodes: device and port. It is a *bipartite* graph, in which device vertices connect to only port vertices, and port vertices connect only to device vertices. A device is represented by a square, while a port is represented by a circle. Therefore, in semiconducting circuits it is possible to convert the 2-input NAND circuit in Fig. 1.3a to a graph in Fig. 1.3b [2].



(a) (b)

Figure 1.3: (a) 2-Input NAND gate. (b) Graph representation of a 2-input NAND gate. A device is represented by a square, and a terminal is represented by a circle [2].

The active component in SDE is the Josephson junction (JJ) which requires a new approach for device netlist extraction for the following reasons:

- A JJ device is not required to connect to lump elements through a port object.
- It is important to extract the individual inductor branches inside the JJ device for parameter extraction. Therefore, a JJ cannot simply be added as a single node in a graph network when doing a netlist extraction.

1.6 Parameterized Cells for SDE

Over the past few decades PCells have started to play an intricate role in the semiconductor industry [12]. This is largely due to the fact that using modern programming scripting languages has become significantly easier to use by inexperienced programmers. The PCells currently used in semiconductor electronics are developed using proprietary software frameworks, such as SKILL [21] and PyCells [12], and are only usable as a feature of an already existing design environment. Currently, the XIC [22] tool is the only PCell package focussing on superconductor IC layouts. Drawbacks of this API includes:

- XIC requires designing PCells using its own native scripting language, which has a steep learning curve.
- The scripting language limits a layout designer to a small set of parameter types (only integers and string).
- No parameters type-checking or parameter-restriction checking is supported by the scripting language.
- The flexibility of this scripting language is extremely limited and difficult to extend.

With the popularity growth of Python in recent years the heuristic of using Python as the basis for PCell creation has become ever more attractive. Developing a fully functional parameterized framework for SDE that includes parameter type-checking and parameter restriction-checking becomes important for the following reasons:

- Superconducting electronic design is in its early stages of development and therefore still requires a lot of tasks to be completed manually. These tasks include hand-designed layouts, manual netlist extraction from a completed layout, and parameter adjustments, such as resistor or inductance values, made using layout editing software. Manually editing layout parameters by hand becomes a daunting task and is prone to editing errors. Developing a parameterized framework for SDE will aid in automating a lot of these tasks.
- Traditional PDV software applies validation checking algorithms post-layout creation. This typically requires that a cell must first be completely designed before applying rule checking, or rule checking must be applied at intermediate steps during the design process, similar to debugging a piece of source code. However, when designing a layout using the parameterized methodology it becomes possible to leverage the dynamic nature of a modern programming language, such as Python, to

applying rule checking during the PCell creation process. For example, when editing a PCell instance the designer can be warned when they are breaking a specific design rule.

1.7 Creating a PDK for SDE

The level of integration of SDE are limited by the relative immaturity of the fabrication process technology. Developing a physical design verification solution for SDE is also highly limited by this lack of process information. Currently, there is no available PDKs produced by superconductor fabrication plants, nor are there any freely available PDK schemes from the semiconductor industry. However, there is a set of comparable patterns between different technology processes in both semiconducting or superconducting technologies.

By studying these process design patterns it becomes possible to develop a *skeleton view* of what a finished PDK for SDE should look like. A heuristic is made that developing a PDK scheme using a bottom-up approach (starting from the basics and building up) will converge to a perennial solution or standard. Further, as the superconducting industry matures this newly proposed PDK scheme will evolve with it.

1.8 Proposed Framework

Integrated circuit design requires many different levels of analysis. A lot of component/device design is done using a manual design flow, from SPICE simulations, to parameter extraction. While at the same time, circuit design requires abstraction at a much higher level.

Seamlessly designing SDE circuits requires a design flow that is analogous to that of the semiconductor industry. A dedicated verification methodology for SDE must be developed due to the physical differences between superconductor and semiconductor electronics.

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. Creating a parameterized cell integrates all design information into one place. Therefore, it is desirable to convert a hand-designed layout into a parameterized layout, which includes device detection and electrical rule checking information.

In this dissertation a new software framework, called SPiRA, is proposed that acts as a system to effectively solve physical design related problems previously discussed for SDE integrated circuit design. SPiRA is a parametric design framework for superconductor integrated circuit design. It revolves around the sound engineering concept that creating circuit layouts are prone

to unexpected design errors. The design process is highly dependent on data provided by the fabrication process. In SPiRA, parameterized cells can be generated that interactively binds data from any fabrication process. The design process of SPiRA introduces parameter restrictions, that limits a designer from breaking process design rules. The proposed framework focuses on solving the following issues in SDE design:

- Develop a parameterized framework that allows layout designer to effectively design layouts using parameters, giving them much more control over their design. SPiRA introduces a newly proposed method, called *validate-by-design*, that gets rid of applying post-layout DRC on a single cell design. This method does not completely replaced traditional design rule checking, since DRC runs still have to be done after a cell has been created to check for violation between different cell instances. Designing a fully functional DRC package is not the core focus of this framework and is left for future development.

- Netlist extraction issues related to SDE, including automatically detecting *dummy nodes*. Complete LVS verification using sub-graph isomorphism are left for future development.

SPiRA integrates different aspects into a single framework, where a parameterized cell can be defined once and then used throughout the design process, performing automatic device detection, netlist extraction, and design rule checking. Consequently, errors are significantly reduced by using the same device definition throughout the entire design flow.

At the core of the SPiRA framework is the Python programming language. Python is easy to read, it is considered to be the industry-standard, and it comes with a rich ecosystem of engineering packages. From Fig. 1.4 the *SPiRA Core* is responsible for parsing a circuit design: If a PCell was designed, the core immediately starts the device detection algorithm. If a GDSII layout is received, the core automatically parameterizes the layout, before running the device detection and netlist extraction algorithms.

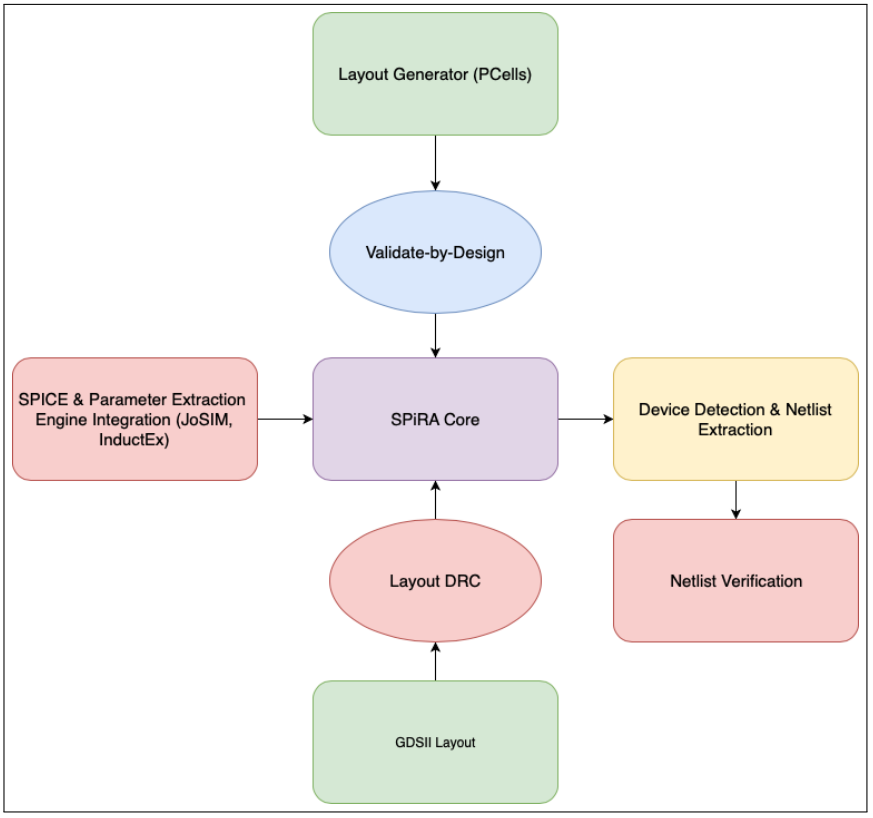


Figure 1.4: SPiRA Framework Design Flow

The macro overview of the SPiRA framework design flow is depicted in Fig. 1.4. From this diagram, the flow diagram elements colored in *red* are still to be implemented in future versions. The *green* elements are circuit models that the designer gives as input: either a parameterized cell, or a designed GDSII layout.

1.9 Objective of Dissertation

The software proposed in this dissertation is not specific to any IC specific technology or fabrication process. The software can be used for other circuit technologies, such as photonics, quantum circuits, analog superconducting circuits, etc. Since this project was funded by IARPA, under the SuperTools project, the focus of this dissertation is Superconductor Integrated Circuits (SICs).

The proposed framework using Python in conjunction with metaprogramming techniques to allow designers to create superconducting and quantum parameterized circuits, while simultaneously checking for design violations—a novel methodology called *validate-by-design*. Using this parameterized frame-

work, a new LVS methodology is proposed that follows a parameterized hierarchical approach to effectively detect layout devices.

Creating PCells and extracting a model from a layout requires data from the fabrication process. A new PDK scheme is introduced, which effectively connects to the SPiRA framework, called a Rule Deck Database (RDD).

This dissertation will show how PCells can be generated and used to compose complex cells, more specifically, how process data from the RDD are connected to layout elementals. Created PCells are then added to the LVS database in the RDD which will be used for device detection in the netlist extraction process. This device detection process uses a parameterized-template method to effectively detect and convert cells in a hand-designed layout to parameterized devices.

A novel *mesh-to-netlist* algorithm is presented that extracts a graph netlist from a circuit layout by first meshing the conducting polygons into a two-dimensional triangular mesh.

Finally, a new graph filtering method is presented that effectively segregates inductive branches. This method is called the *branch-detection* method, since is dissects a raw extracted layout graph into branches before detecting dummy nodes.

In summary, the objective of this dissertation requires the development of a physical verification design environment that can solve the following SDE layout design problems:

1. Present layout designers with an easy to use framework with the focus falling on SDE circuit layout design. The design environment must give a designer full control over their design.

2. Effectively connect process data from a PDK to layout elementals. The data parsing methods of the framework must be generic to be easily adaptable to future technology changes.

3. Easily define technology devices in an LVS database, which can be used as templates for device detection during the netlist extraction phase.

4. Extract a netlist from a physical layout that defines individual conducting branches.

Apart from the technical requirements, there is a few necessary systemic requirements for the future survival of the proposed framework:

1. A development system has to be setup, rather than just developing a collection of single solutions.

2. This system has to be maintainable and extendible, making it self-evolving to some degree.

3. A clear design architecture has to be defined to maintain software cohesion.

Chapter 1 (Introduction): Articulates the problems that have to be solved in this dissertation, and also discusses the limitations of the current software in SDE.

Chapter 2 (PDK Implementation): Gives an overview of the newly proposed PDK scheme, which is a Python-based modular approach to effectively describe fabrication process data.

Chapter 3 (Parameterized Cells): Introduces the SPiRA design environment in which PCells can be created.

Chapter 4 (Validate-by-Design): Introduces the novel design rule checking method when designing parameterized cells. This method leverages the parameterized methodology to check parameter violations as a PCell is designed in the SPiRA environment.

Chapter 5 (Framework Core): An in-depth dissection of the SPiRA core, which is responsible for parsing GDSII layouts and binding process fabrication data.

Chapter 6 (Cell Conversion): This chapter discusses the detection of device cells in a GDSII layout, by connecting to a set of pre-defined PCells defined in the RDD.

Chapter 7 (Netlist Extraction): A novel *mesh-to-graph* netlist extraction algorithm is presented. This method extracts a graph from a layout by first meshing the entire geometry. Then, dummy nodes are automatically detected by first dissecting the graph into individual branches, called the *branch-detection* method.

Chapter 8 (Extraction Results): Discusses the extraction results (device detection and netlist extraction) of a set of GDSII layouts.

Chapter 9 (Conclusions): Conclusion are drawn from the technical methods implemented, and the philosophical constructs that was used to derive these technical solutions.

Chapter 2

PDK Implementation

The process design kit (PDK) is a set of technology files needed to implement the physical aspects of a layout design. Application-specific rules specified in the PDK controls how physical design applications work. The PDK contains the following information.

- **Initialization** Data that is specific to the canvas in which the layout will be generated, such as grid size, coordinate scaling values, etc.
- **Layer Mapping** Contains layer names and GDSII numbers used in the fabrication process. These layers are mapped to purposes and design restrictions. The *layer purpose* defines what role the layer plays in a layout, such as a metal layer, or a contact layer. The *design restrictions* define process constraints on a specific layer, such as minimum and maximum width.
- **DRC Dataset** Data that builds from the design restrictions specified in the layer mapping. The DRC dataset describes how the layer design restrictions should be used in verifying the creation of each layer when designing a layout. It also defines the restriction relations between different layers.
- **LVS Dataset** A class that contains the description of layer formulation rules. For example, it contains information about which metal- and contact layers must overlap in order to detect a via. The LVS dataset consists of parameterized cells used for device detection.

The DRC database specifies certain geometric and connectivity restrictions to ensure sufficient margins to account for variability in the manufacturing processes. Design rule checking plays a major role, during physical verification sign-off, for both a parameterized cell and a hand-design layout. Each process may contain hundreds of physical design rules and multiple device definitions, such as junctions and vias with different design parameters. Similar to process

layers being linked to a DRC database, device cells must be recognized by a LVS database. Creation, modification and maintenance of the complete DRC and LVS database is a complicated and time-consuming process that should be automated.

In this chapter a new PDK scheme is introduced. The Python programming language is used to bind PDK data to a set of classes, called *data trees*, that uniquely categorises PDK data. This new PDK scheme is called the Rule Deck Database (RDD), or the Rule Design Database. By having a native PDK in Python it becomes possible to use methods from the SPiRA framework, and consequently, have a more descriptive PDK database.

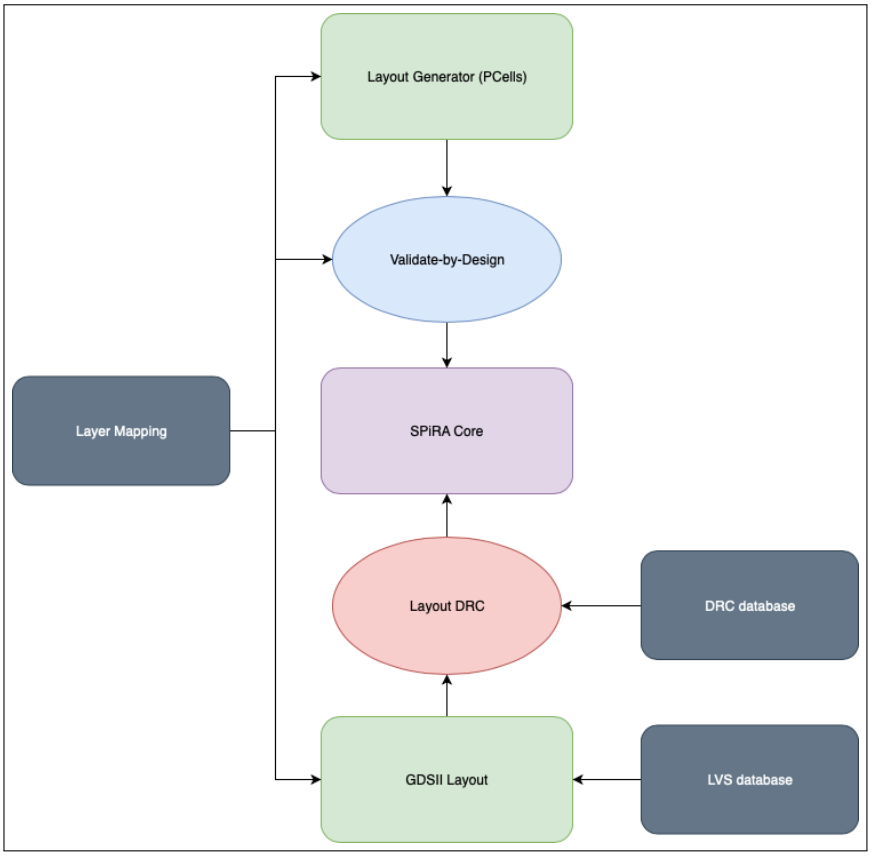


Figure 2.1: Overview of how the different categories of the RDD fits into the SPiRA design environment.

The proposed approach is based on the fact that the set of physical design rules, for any process or technology, usually may be divided into a final set of technology independent categories, such as width, space, enclosure and so on. Moreover, the set of legal devices for any process may be divided into a final set of technology independent categories using parameterized cells. From the SPiRA diagram in Fig. 2.1, data from the *layer mapping* category in the RDD are added to a PCell by defining parameters. The reverse happens for a

GDSII layout, where the layout is automatically parameterized by dissecting and connecting information defined in the RDD.

2.1 Objectives

The objective of this chapter is to introduce a new format for designing a PDK, which solves the following problems:

- Use process data to effectively define parameterized cells, which can be used for device detection.
- Data can directly connect to the SPiRA design environment, since both the framework and the PDK are in Python.
- This novel PDK scheme can use Python features to its advantage.

2.2 Library

In the SPiRA framework, each individual process has its own RDD object. Before describing the structure of creating the RDD, it is important to understand how the RDD connects to the design environment. Each GDSII layout cell is listed in a global class, called the GDSII Library. This library contains all the cell dependencies present in the top-level cell, including the top-level cell itself. The library holds all information required for full physical circuit verification. It consists of the process database, parameterized cells, template cells, and layout cells. It also connects to the GDS Viewer and basic GDSII read/write methods. When creating a layout, a new library is generated if one does not already exist. The specific RDD for a process is automatically connected by importing the specific RDD object. The benefit of adding a cell to a library is to have multiple cells with the same name, but that connect to different fabrication processes.

2.3 The Newly Proposed Process Database

In [23] a transistor PCell for semiconductors is verified using a Generic PDK provided by Cadence [24]. To effectively parse generic metadata, a new scripting format is proposed. Initially, a layer-definition-file (LDF) was used to bind metadata to program objects, such as elementals. This LDF file aggregated layer description data of a specific fabrication process and is the standard used by InductEx [8]. For ease of portability between different programming language, the JSON file format was used and later versions migrated to the YAML format, since it offered more descriptive components. Due to exogenous technological changes defining design rules with just a single file—like a JSON

or YAML file—will at some point, during its future evolution, invalidate some of the embedded assumptions. This explanation is congruous with earlier explanations about inductive reasoning. A more adaptable and complex method has to be implemented, which converges to a more script-like approach.

The lack of interoperable characteristics of a file format (such as effectively describing polygon operations), leads to the conclusion of having to use a scripting approach to effectively describe process data. The aim of the RDD is to develop a script-like interface to intricately describe design details. It therefore becomes axiomatic to use Python as the basis for developing a script-like process database format, since it is currently the most used scripting language used by hardware designers. The RDD defines the parameters used in design sessions. This database includes layer definitions, device definitions, and design rules: all of the information needed to define a parameterized design.

2.4 Creating the RDD

To run a SPiRA design session, the designer must define the process database by importing the specific `RDD` tree. A process tree is constructed when importing the RDD from a specific library. Appropriate parameters and rules can be applied to a design using the RDD object, which contains the constructed process tree. When running the design software on a GDSII layout or a parameterized script, it uses the definitions from the RDD object to define the design. The following explains the implementation of the different PDK categories in the RDD:

2.4.1 Initialization

All caps are used to represent the RDD syntax. The reason being to make the script structure clearly distinguishable from the rest of the framework source code. First, the RDD object is initialized, followed by the process name and description. Second, the GDSII related variables are defined.

```
1  # Create a rule deck object.
2  RDD = RuleDeckDatabase()
3  RDD.name = 'mitll'
4  RDD.desc = 'PDK data for the MiTLL process.'
5
6  # Define the GDSII variables.
7  RDD.GDSII = DataTree()
8  RDD.GDSII.UNIT = 1e-6
9  RDD.GDSII.PRECISION = 1e-9
10 RDD.GDSII.GRID = 1e-12
```

2.4.2 Layer Mapping

A process tree is created for each individual layer. Any number of variables can be added to the tree using the dot operator. A new layer type is defined by creating a `ProcessTree`. This tree structure then contains a `Layer` object in conjunction with a set of design restrictions, as shown below:

```
# Define a new process layer of type M6.
RDD.M6 = ProcessTree()
# Add the layer data to the tree.
RDD.M6.LAYER = Layer(name='M6', number=60, datatype=0)
# Add design data (restrictions) to the tree.
RDD.M6.MIN_SIZE = 0.5
RDD.M6.MAX_WIDTH = 20.0

# Define a new process layer of type I5.
RDD.I5 = ProcessTree()
RDD.I5.LAYER = Layer(name='I5', number=54, datatype=0)
RDD.I5.MIN_SIZE = 0.7
RDD.I5.MAX_WIDTH = 1.2
```

The purpose indicates the use of the layer or material. Multiple layers with the same name but different purposes can be created. Purposes are defined in a `PurposeTree` object:

```
# Define general purpose layers.
RDD.PURPOSE = PurposeTree()
RDD.PURPOSE.METAL = PurposeLayer(
  name='Polygon metal',
  datatype=1,
  symbol='METAL'
)
RDD.PURPOSE.VIA = PurposeLayer(
  name='Via polygon',
  datatype=2,
  symbol='VIA'
)
RDD.PURPOSE.JUNCTION = PurposeLayer(
  name='Junction polygon',
  datatype=3,
  symbol='JJ'
)

# Define error purpose layers.
RDD.PURPOSE.ERROR = ProcessTree()
RDD.PROCESS.ERROR.ENCLOSURE = PurposeLayer(
```

```
22   name='Layer that breaks the enclosure design rule.',
23   datatype=100,
24   symbol='ENC'
25 )
```

SPiRA contains a default purpose tree that will be used for most processes, since these purposes are typically process independent. A user can extend this tree by simply adding new variable definitions. If a specific variable is already defined, an error will be thrown, for example:

```
1 # In a new process file, such as AiST
2 >>> RDD.PURPOSE.ERROR.ENCLOSURE = PurposeLayer(
3   name='Layer that breaks the enclosure design rule.',
4   datatype=103,
5   symbol='ENC'
6 )
7 ENCLOSURE variable already defined.
```

To create layer-purpose mappings a new data set is created that consist of a set of physical layers. Physical layers is unique to SPiRA and is defined as a *layer that has a defined purpose*. The `PhysicalLayer` class accepts the `layer`, `purpose` and `data` parameters to map the purpose to a specific process layer and its design restrictions.

```
1 RDD.PLAYER = PhysicalTree()
2 RDD.PLAYER.M6 = PhysicalLayer(
3   layer=RDD.M6.LAYER,
4   purpose=RDD.PURPOSE.METAL,
5   data=RDD.M6
6 )
7 RDD.PLAYER.I5 = PhysicalLayer(
8   layer=RDD.I5.LAYER,
9   purpose=RDD.PURPOSE.VIA,
10  data=RDD.I5
11 )
```

This code creates a `PhysicalTree` object, and then defines multiple layer-mappings by creating physical layers. The physical layer, `RDD.PLAYER.M6` (lines 2−6), defines that layer M_6 is a metal layer. The physical layer, `RDD.PLAYER.I5` (lines $7-11$), defines that layer I_5 is a via layer.

2.4.3 LVS Database

Any superconductor process or technology only allows a finite set of legal devices. Layout versus Schematic (LVS) comparison run-sets determine one-to-one equivalency between an integrated circuit schematic and an integrated

circuit layout. The correctness and completeness of the LVS run-sets are verified using test cases (set of devices), which contain layout elementals representing failing and passing conditions for each device of the technology. The goal of this section is to describe how this test case suite is setup for LVS detection. The LVS database defines a set of devices that has to be checked when extracting a cell layout. The LVS database is populated by importing the predefined PCells that is representative of the devices present in a specific technology.

```
1 RDD.DEVICES = DeviceTree()
2
3 class JunctionDevice(DynamicDataTree):
4   def initialize(self):
5     from .pcells.junction import Junction
6     self.PCELL = Junction
7
8 RDD.DEVICE.JJ = JunctionDevice()
```

Line 1 initializes the device tree and adds it to the RDD object. The `JunctionDevice` class delays the initialization of the parameterized junction imported in the `initialize` method. Line 8 adds this class to the device tree as the `JJ` variable. A `Junction` instance can then be created as follows:

```
1 >>> RDD.DEVICES.JJ.PCELL()
2 [SPiRA: Device] (name 'Junction')
```

For via devices a template and a device PCell has to be defined. The template cell defines the rules for detecting a metal connection through a via layer.

```
1 RDD.VIAS = DeviceTree()
2
3 class ViaDevice(DynamicDataTree):
4   def initialize(self):
5     from .pcells.via import ViaI5
6     self.PCELL = ViaI5
7     self.TCELL = ViaTemplate(
8       name='I5',
9       via_layer=RDD.I5.LAYER,
10      layer1=RDD.M5.LAYER,
11      layer2=RDD.M6.LAYER
12    )
```

Line 1 defines the via tree (which is also a device tree) and adds it to the RDD object. Line 6 defines the PCell for the specific via device. Line 7 defines that a `ViaTemplate` must be used for device detection. This template will only detect a `ViaI5` device if polygons with layer types M_5, M_6, and I_5 overlaps.

2.5 DRC Database

The DRC database is implemented in very much the same way as the LVS database. Instead of defining parameterized cells, the DRC database defines a set of rules. Each rule category, such as minimum/maximum width, is added as a separate list to a *rule tree*. Below is an example of defining the width rules for layer M_5 and M_6. The width rule regulates the distance between two inside edges of the same polygon.

```
1  class DesignRuleTree(DynamicDataTree):
2    def initialize(self):
3      from spira.lrc.rules import Rule
4      from spira.lrc.width import Width
5
6      m6_width = Width(
7        layer1=RDD.PLAYER.M6,
8        minimum=RDD.M6.MIN_SIZE,
9        maximum=RDD.M6.MAX_WIDTH
10     )
11
12     m5_width = Width(
13       layer1=RDD.PLAYER.M5,
14       minimum=RDD.M5.MIN_SIZE,
15       maximum=RDD.M5.MAX_WIDTH
16     )
17
18     self.WIDTH = [
19       Rule(design_rule=m5_width, error_layer=RDD.PURPOSE.ERROR.WIDTH),
20       Rule(design_rule=m6_width, error_layer=RDD.PURPOSE.ERROR.WIDTH),
21     ]
22
23 RDD.RULES = DesignRuleTree()
```

Line $6-10$ defines a width object for layer M_6 using predefined variables from the RDD object. This layer has a minimum width equal to the variable `RDD.M6.MIN_SIZE`, and a maximum width equal to the variable `RDD.M6.MAX_WIDTH`. Line $18-21$ creates a list of design rules using the defined width objects. Each rule are mapped to an error purpose. The defined width rules can be accessed using the dot operator:

```
1 >>> RDD.RULES.WIDTH
2 The 'WIDTH' rule set has a size of 2.
```

2.6 Conclusion

This chapter dissected the basic constituents of a general PDK, followed by an introduction to the newly proposed PDK scheme, called the RDD. The RDD is first initialized by creating a rule deck database object. This object is given a name and a description. Secondly, the layer mapping and physical layers are defined with their corresponding data restrictions as specified by the fabrication process. The LVS database is defined using pre-designed PCell devices. The DRC database is created by defining the different design rules using the corresponding classes from the SPiRA framework. Both these databases uses a delayed initialization class, which solves importing issues and only constructs the physical object once they are called using the dot operator. Next, parameterized cell construction is explained and how the RDD is used to bind process data to layout patterns.

2.7 Future Work

A basic PDK scheme was introduced in this chapter, but there is still a lot of functionality that can be added, such as display resources, material stacking, and InductEx dependent data. Also, future work will involve writing parsers for converting different PDK data structures to that of the RDD structure, including openACCESS, and KLAyout (.lyp) support.

Display resources assigns data to layers for visualization. This includes the color, pattern type (stipple, dots), transparency, etc. *Material stacking* data provides information, such as layer thicknesses and growth order, which are used for three-dimensional model creation. InductEx dependent data, used for physical parameter extraction, can be incorporated into the RDD, once the proposed framework is updated to interface with InductEx.

Chapter 3

Parameterized Cells

The increasing popularity of SDE raises the demand for cheaper S-EDA tools. However, one of the main steps of designing an integrated circuit is drawing the physical layout structure. A structure consists of a combination of elementals, such as polygons, paths and labels. This structure is also called a cell, which can contain other cells, called references. The S-EDA tools should have a feature for automatic generation of parameterized elementals and cells as predefined layouts in a library. The layout generators, which are presently in use in the semiconducting industry, are developed using a proprietary language, SKILL, and are only usable as a feature of a specific EDA-tool [21]. Other layout generators, such as PyCells, are developed in Python, but are also restricted to the open community [12]. On the other hand, the second major drawback of using the proprietary layout generators is the lack of interoperability. The purpose of this chapter is to introduce an open standard for parameterizing layouts for SDE. The more accurate the device description, the more accurate the device detection from a circuit layout, and thus, the more accurate the netlist extraction.

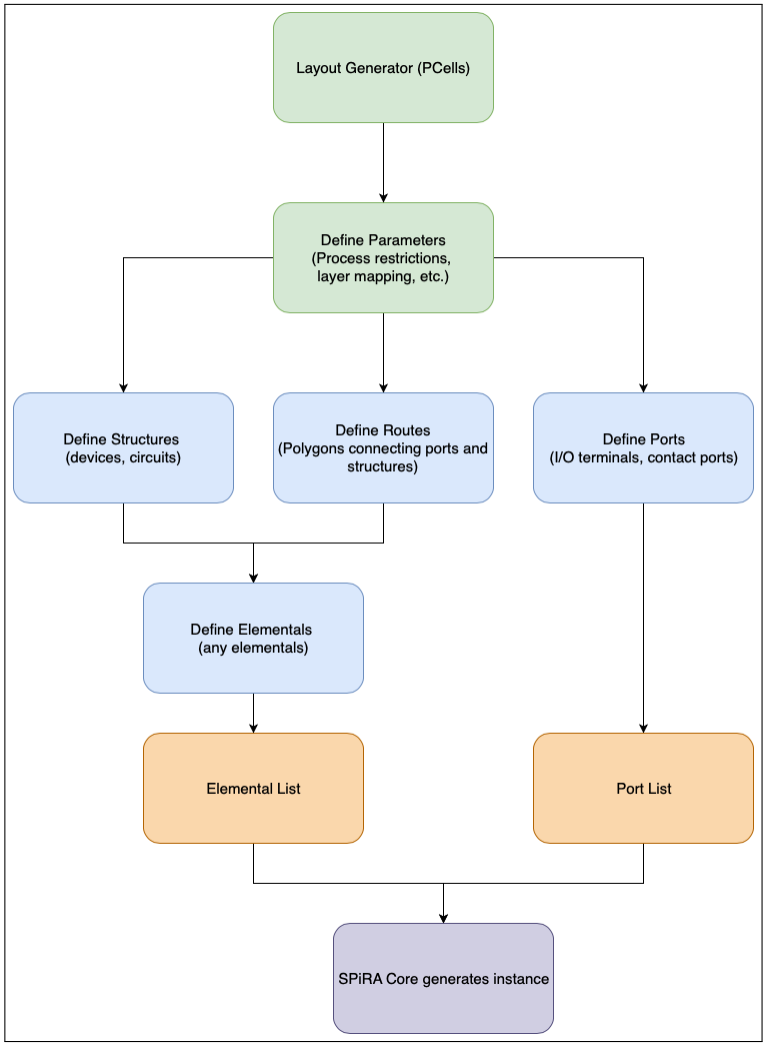


Figure 3.1: Design environment for creating a parameterized cell.

The diagram in Fig. 3.1, shows the expanded diagram of the layout generator concept introduced in Fig. 1.4. Generating a parameterized cells starts by first defining the parameters of the instance. Predefined values from the RDD can be used, along with parameter restrictions. Next, the components (elementals and ports) that constitute the layout are created using the defined parameters. These components are added to a cell instance as custom lists, unique to the SPiRA framework. Finally, the GDSII processing, parameter validation, checking, and binding are done by the SPiRA core.

3.1 Objectives

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, parameterized device cells defined can be added to the LVS database. The SPiRA framework focuses on solving the following programming-related problems when generating a layout using a scripting approach:

- Effectively define and group layout elementals in a templated design environment. This makes it possible to create a system that can both: generate parameterized layouts, and parse GDSII layouts.
- Parameters must easily be used to create elementals. This allows using PDK data to define elementals.
- Ports that can connect different polygons. This is used to provide interconnectivity between different cells.
- Elementals and ports must be easily swapped out between different cells. This is used to parameterize a GDSII layout.

3.2 GDSII Overview

The GDSII file format is the industry standard format used in designing layouts [25]. Different elementals, text and labels are parsed using the GDSII format. This section discusses the interface implementations to parse GDSII related data. The Gdspy library is used for GDSII file parsing. The elements and data structures created by the Gdspy library are inherited by the SPiRA core. Therefore, before parsing data from the Gdspy library, all data has to be congruent with the specific SPiRA data structures. The core interfaces with the Gdspy library. One constraint with the Gdspy library is binding external data to elements, such as data from the PDK. Forking the Gdspy library and implementing parameter properties and extra functionality is a sub-optimal solution for the following reasons:

- **Updates**: The Gdspy library is in active development. Benefiting from future updates is of paramount importance. Loss of robustness can possibly emanate from these updates, but is prevented by the architectural interfaces implemented between SPiRA and Gdspy.
- **Consistency**: The Gdspy library structure instils certain design principles.
- **Stability**: Gdspy has been an opensource project since 2009 and has thus been used and updated by the community for almost ten years. It can therefore be presumed that this library is Lindy compatible [26].
- **Extendibility**: Using the Gdspy library documentation as reference will aid in improving the workflow of new developers. Contributions to the Gdspy library can also be made, providing additional robustness.

The rest of this section gives an overview of the different constituents that make up a GDSII file.

3.2.1 Elementals

SPiRA implements data bindings to Gdspy elements by extending each element class. These newly created elements are called *elementals*. Received parameters are type-checked to make sure it fits the specification requirements, defined in the RDD.

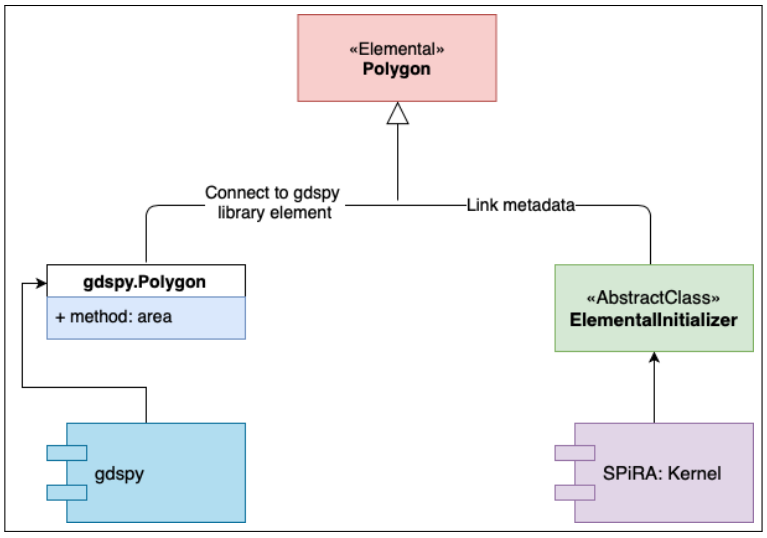


Figure 3.2: Basic elemental class structure

Fig. 3.2 shows the class composition of a basic elemental. The figure uses the `Polygon` elemental as an example. The `Polygon` class inherits from the polygon class, created by the Gdspy library, and extends it by adding functions used for transformation operations. Parameters are dynamically bound, after type-checking to the elemental class through the `FieldInitializer` abstract class—which is a set of base metaclasses explained in Chapter 5.

3.2.2 ElementalList

The elemental list in the cell class is not a normal Python list, but a custom typed list specific to SPiRA, called ElementalList. The ElementalList class inherits and extends from Python's default list class. Elemental types can be easily filtered using *property* methods.

All elemental objects are stored in an ElementalList parameter, which can either coincide within a cell or be separate. This is an important concept to understand, since using an ElementalList separately enables the manipulation of elementals before committing to a cell instance.

```
# Get all polygon elementals in ‘elems‘
polygons = elems.polygons
# Get polygon set of specific type
m1_plys = elems.polygons[RDD.METALS.M6.LAYER]
```

This custom class consists of a set of magic- and property methods that makes accessing specific data easier. Instead of filtering through all the elementals to retrieve only the polygon elementals, it can be directly accessed, as shown above.

3.2.3 Flattening

Working with the GDSII hierarchical file structure allows for powerful data transfer and manipulation, especially when working with PCells. Flattening circuit layouts plays an intricate role in correctly writing and debugging GDSII files. Sometimes flattening a certain specific device cell, such as a Junction, can be beneficial in simplifying the structure for DRC checking.

The Gdspy library flattens a cell by recursively looping through all the polygon object in the cell instance. All polygon coordinates are recursively added to a global Numpy array, taking into account the cell reference transformations. The new polygon list contains all the coordinates that make up the top-level cell.

Using the Gdspy flattening method through inheritance suffices to take into account the unique polygon parameters for each instance. When flattening a structure, it is required to keep the elemental instances and only apply geometrical transformations to these instances, instead of creating new instances and losing unique metadata. Updating the flattening algorithm required extending the framework with a `Transformation` class. Each elemental is connected to a set of transformation methods through the `Transformation` mixin. The `ElementalList` class is also extended with a flattening method. Elementals in an individual list can be flattened instead of first having to create a cell instance. More important: the flattening level can be set by the user, which is not possible through the Gdspy library.

3.2.4 Cells

The cell class follows much the same structure as that of elementals. It inherits from the Gdspy cell class and binds parameters using the `CellInitializer` abstract class, which then connects to the `FieldInitializer`. Generated layouts connects to the SPiRA core by inheriting from `spira.Cell`, as shown in Fig. 3.3.

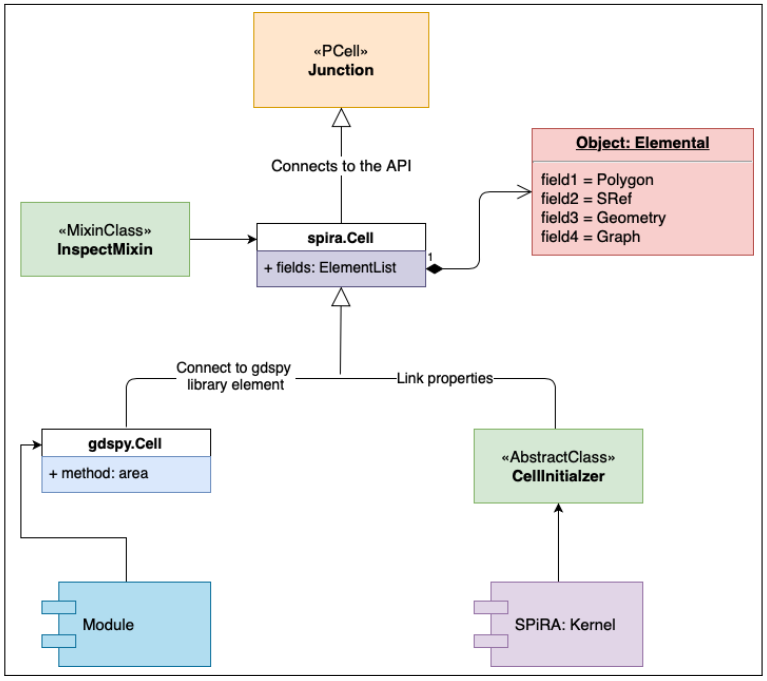
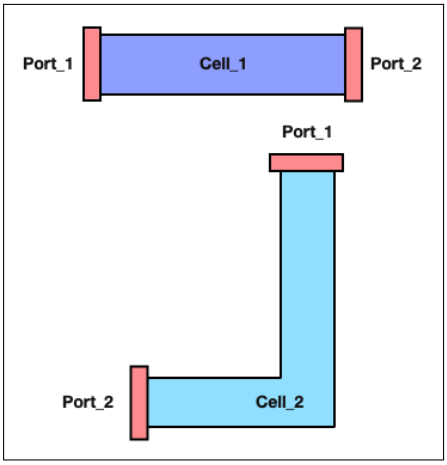


Figure 3.3: The architectural structure of generating a new PCell.

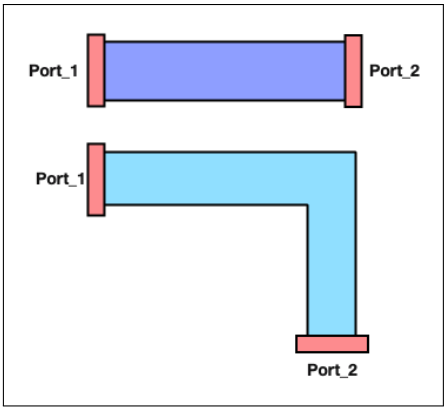
Typically, the designer would want a set of *inspection* functions to programmatically inspect and debug the created cell. An `InspectMixin` class is mixed into the `Cell` class, which adds a set of inspect methods—such as listing all cell references, removing cell references, etc. Ports are introduced as a connection element between different SRef instances.

3.2.5 SRefs

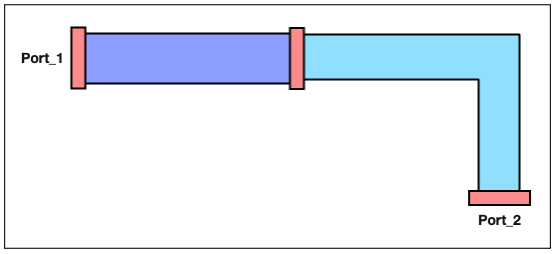
The `SRef` class contains a reference to a `Cell` instance, called the parent cell. In the GDSII format a cell can only be added to the top-level cell through a cell reference. Consequently, all elementals defined in a cell instance are by definition locked to the location at which the SRef object is placed. The reason for discussing `SRef` separately is to explain how it manages `Port` objects. Geometry transformations and binding unique objects to a specific reference 'complexifies' when transferring geometric data through the hierarchical tree. As an example, imagine creating two cells, C_1 and C_2, each consisting of only one polygon P_1 and P_2, as shown in Fig. 3.4. Each polygon has two ports connected to the ends—labeled $Port_1$ and $Port_2$. These two polygons can be connected by moving $Port_1$ of P_2 to $Port_2$ of P_1. Doing so requires the cell reference to be transformed to the correct location. This transformation applied to the port objects also transforms the port instances of the parent cell, which causes recursive problems when more complex hierarchical problems arise.



(a) Two cells consisting of a single polygon and two ports.



(b) $Cell_2$ rotated 90 degrees



(c) $Cell_2$ translated to position

Figure 3.4: $Port_1$ of $Cell_2$ are connected to $Port_2$ of $Cell_1$

Solving this requires that ports are linked to the specific SRef instance, by copying them from the parent cell. These copied ports, called *local ports* are then transformed to the required positions, while at the same time keeping the ports defined in the parent cell unchanged. That is, if copies of a specific SRef instance are made, they still reference back to the same cell. If the port instance from a cell is directly used, then connecting two SRef instances referring to the same cell will not be possible, since the ports belongs to the same objects.

3.2.6 Ports

Port instances are unique to SPiRA. It contains a `Polygon` and a `Label` elemental. This allows the polygon and label elementals to be geometrically transformed as one cohesive structure.

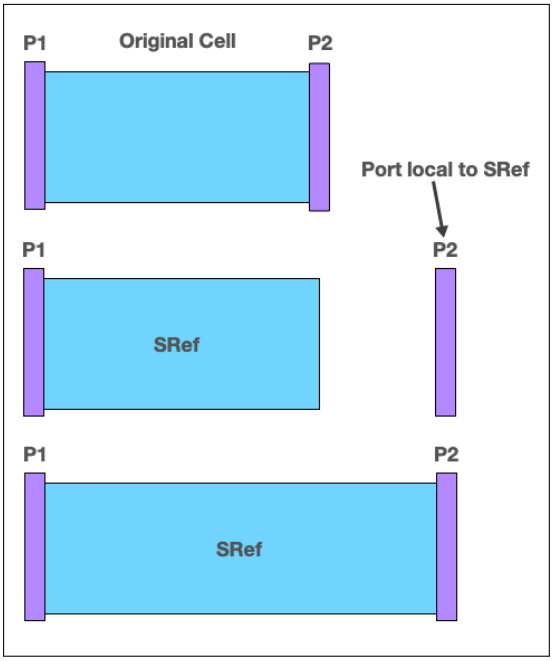


Figure 3.5: Ports are used to stretch a polygon to a specific location.

Consider the basic cell shown in Fig. 3.5 consisting of a basic polygon and two port instances. The cell is added to the top-level cell as a SRef instance. This creates a local copy of the ports from the parent cell to the SRef instance. By simple moving the position of the port, the connected polygon `Edge` will snap to position after the port has been placed.

3.3 Defining Parameterized Cells

The SPiRA definition of a Parameterized Cell (PCell) in general terms: *A PCell is a cell that defines how layout elementals must be generated. When instantiated it constructs itself according to the defined parameters.* Basically, a PCell is simply a cell that changes or generates its elementals based on a set of received parameters.

GDSII layouts encapsulate elemental design in the visual domain. Parameterized cells encapsulates elementals in the programming domain, and utilizes this domain to map external data to elementals. This external data can be data from the PDK or values extracted from an already designed layout using simulation software, such as InductEx. The SPiRA framework uses a scripting framework approach to connect the visual domain with a programming domain. The implemented architecture of SPiRA mimics the physical layout patterns implicit in hand-designed layouts. This framework architecture evolved by developing code heuristics that emerged from the process of creating a PCell. For example, since a layout will always consist of a list of elementals,

the `create_elemental` class method is a special method that generates all the elemental objects that will be added to the cell instance. Elaborate code pattern construction and reconstruction allow for the development of powerful systems to solve device detection, netlist extraction and design rule checking. Without these basic building blocks the system is prone to failure when changes are made to the fabrication process. PCell creation is broken down into the following basic steps:

- Create a new cell by creating a new class that inherits from a base class (that connect to the SPiRA core).
- Define parameters as class attributes through their respective fields.
- Add parameterized layout elementals to a cell based on the received parameterz.

This chapter only dissects PCell examples as an introduction to using the framework and to understand the implemented architecture. The rest of this dissertation will show how these code patterns are used to coalesce other modules in SPiRA.

3.4 Parameterized Elemental Creation

Creating a PCell is done by defining the elementals and parameters required to create the desired layout. The relationship between the elementals and parameters are described in a template format. Template design is an innate feature of parameterizing cell layouts. This heuristic concludes to develop a framework to effectively describe the different constituents of a PCell, rather than developing an API. The SPiRA framework was built from the following concepts:

Defining Elemental Shapes This step defines the geometrical shapes from which an elemental polygon is generated. The supported shapes are rectangles, triangles, circles, as well as regular and irregular polygons. Each of these shapes has a set of parameters that control the pattern dimensions, e.g. the parameterized rectangle has two parameters, `width` and `length`, that defines its length and width, respectively.

Elemental Shape Transformations This step describes the relation between the elementals through a set of operations, that includes transformations of a shape in the x-y plane. Transforming an elemental involves: movement with a specific offset relative to its original location, rotation of a shape around its center with a specific angle, reflection of a shape around a defined line, and aligning a shape to another shape with a specific offset and angle.

PDK Binding The final step is binding data from the PDK to each created pattern. In SPiRA data from the PDK is parsed into the Rule Deck Database. From this database the required process data can be linked to any specific pattern, such as the layer type of the defined rectangle, by defining parameters and placing design restrictions on them.

3.5 Parameterized Elemental Grouping

There are other special shapes that can be used in the pattern creation. These shapes are mainly a combination polygons and relations between polygons. These special shapes are referenced as if they represent a single shape and its outline is determined by its bounding box dimensions. The following elemental groups are defined in the SPiRA framework:

- **Cells:** Is the most generic group that binds different parameterized elementals or clusters, while conserving the geometrical relations between these polygons or clusters.
- **Ports:** A port is simply a polygon with a label on a dedicated process layer. Typically, port elementals are placed on conducting metal layers.
- **Routes:** A route is defined as a cell that consists of a polygon elemental and a set of edge ports, that resembles a path-like structure.
- **Device:** A cell that only consists of metal layers and contact layers, and acts as a device inside a larger circuit. Junctions and vias are examples of device cells.
- **Circuit:** A cell that consist of multiple devices and routes.

3.6 Framework Design Overview

Interfacing with the Gdspy library makes it possible to coherently use PCells in conjunction with a concrete layout. Therefore, mixing PCells and already designed layouts is possible using the SPiRA framework. In other words, the whole circuit layout does not have to be parameterized, only certain cells can be parameterized and imported in a GDSII layout.

```
1 import spira
2 from spira import param, shapes
3 from lib.mit.pdk.process.databases import RDD
```

From the code shown above: When designing a parameterized cell, the `spira` module has to be imported, followed by the `param` and `shapes` namespaces used for defining parameters and geometric shapes. Finally, the global RDD variable is set when imported from a specific fabrication process.

```
class PCell(spira.Cell):
  """ My first parameterized cell. """

  # Define parameters here.
  number = param.IntegerField(default=0, doc='Parameter example number.')

  def create_elementals(self, elems):
    # Define elementals here.
    return elems

  def create_ports(self, ports):
    # Define ports here.
    return ports

>>> pcell = PCell()
[SPiRA: Cell] (name 'PCell', elementals 0, ports 0)
>>> pcell.number
0
>>> pcell.__doc__
My first parameterized cell.
>>> pcell.number.__doc__
Parameter example number.
```

The most basic SPiRA template to generate a PCell is shown above, and consists of three parts:

1. Create a new cell by inheriting from `spira.Cell`. This connects the class to the SPiRA framework when constructed.

2. Define the PCell parameters as class attributes—also called the parameter specification section—using the `param` namespace.

3. Elementals and Ports are defined in the `create_elementals` and `create_ports` class method, which is automatically added to the cell instance. The create methods are special SPiRA class methods that specify how the parameters are used to create the cell. The elementals specified in the `create_elementals` method builds the cell geometry from the parameters identified.

The `spira.Cell` class contains an `elemental` parameter that binds an empty `ElementalList` object to the `create_elemental` method. This custom elemental list acts as a manager object to effectively categorize and filter different elemental types. This prevents a designer from adding elementals that are not supported by the framework, or violate any of the defined parameter restrictions. The same

design pattern is used for connecting ports. Also, SPiRA automatically connects docstrings to parameters and classes using a meta-configuration, which can be used for parameter descriptions as shown in Line 19 and Line 22.

```
1  class Box(spira.Cell):
2
3    width = param.NumberField(default=1)
4    height = param.NumberField(default=1)
5    gds_layer = param.LayerField(number=0, datatype=0)
6
7    def create_elementals(self, elems):
8      shape = shapes.BoxShape(width=self.width, height=self.height)
9      elems += spira.Polygon(shape=shape, gds_layer=self.gds_layer)
10     return elems
11
12   def create_ports(self, ports):
13     ports += spira.Term(name='Input', midpoint=(-0.5,0), orientation=90)
14     ports += spira.Term(name='Output', midpoint=(0.5,0), orientation=-90)
15     return ports
16
17 >>> box = Box()
18 [SPiRA: Cell] (name 'Box', width 1, height 1, number 0, datatype 0)
19 >>> box.width
20 1
21 >>> box.height
22 1
23 >>> box.gds_layer
24 [SPiRA Layer] (name '', number 0, datatype 0)
```

The above example illustrates constructing a parameterized box using the proposed framework: First, defining the parameters that the user would want to change when creating a box instance. Here, three parameter are given namely, the width, the height and the layer properties for GDSII construction. Second, a shape is generated from the defined parameters using the `shape` module. Third, this box shape is added as a polygon elemental to the cell instance. This polygon takes the shape and connects it to a set of methods responsible for converting it to a GDSII elemental. Fourth, two terminal ports are added to the left and right edges of the box, with their directions pointing away from the polygon interior.

```
1 >>> box = Box(width=4, height=2)
2 [SPiRA: Cell] (name 'Box', width 4, height 2, number 0, datatype 0)
3 >>> box.width
4 4
5 >>> box.height
```

```
6  2
7  >>> box.gds_layer
8  [SPiRA Layer] (name '', number 0, datatype 0)
9
10 >>> box = Box(gds_layer=spira.Layer(name='M6', number=60))
11 [SPiRA: Cell] (name 'Box', width 1, height 1, number 60, datatype 0)
12 >>> box.gds_layer
13 [SPiRA Layer] (name 'M6', number 60, datatype 0)
14
15 >>> box = Box(gds_layer=RDD.M6.LAYER)
16 [SPiRA: Cell] (name 'Box', width 1, height 1, number 60, datatype 0)
17 >>> box.gds_layer
18 [SPiRA Layer] (name 'M6', number 60, datatype 0)
```

A box instance is created, line 1, with a width of $4\,\mu m$ and a height of $2\,\mu m$, the result is shown in Fig. 3.6. A default layer parameter is generated with GDSII number 0 and GDSII datatype 0. The generated instance can be inspected as shown in the code above between lines $3-8$. Line 10 creates a box instance with a new layer parameter that has a GDSII number of 60 and the name 'M6'. Line 15 creates a box instance and uses a layer parameter defined in the RDD.

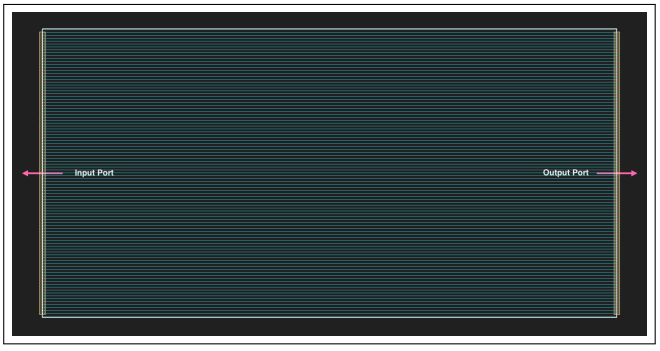


Figure 3.6: Box polygon with two connected terminals.

Next, this basic parameterized methodology can be extended to automatically extract elementals from a layout and parameterize it according to a given algorithm.

3.7 Process Layer

Polygon objects are by far the most used and most important elemental in a GDSII layout. The GDSII file format only connects two data variables to a polygon object; *gdsnumber* and *datatype*. Typically, fabrication process data are connected to polygon structures programmatically, due to the limits of the

GDSII file format. This is highly dependent on the structure of the datafile format used to parse PDK data. Chapter 2 explained the Rule Deck Database scheme, used in SPiRA, to eminently parse PDK data in a generic script, making the proposed package highly independent on future process changes.

The SPiRA core consists of elementals in their most rudimentary form. To maintain a level of abstraction it is important to keep these elemental classes as basic as possible. Changing the classes themselves, every time a new piece of information is obtained or a new implementation method is tested, will quickly lead to overall systemic degradation. Connecting extra data to a polygon object can be done by creating a new polygon class which inherits from `Polygon`. However, it must be possible to add terminal or port objects that are connected to the specific polygon instance. This means elemental data types are getting mixed, and therefore a cell class should be constructed.

With the basic building blocks defined, and parameterization using the RDD described, any cell elemental can be parameterized for a specific fabrication process. This section presents the next hierarchical building block added to the system to automate device detection: The `ProcessLayer` class extends the default `spira.Cell` class to automatically add extra methods to a polygon elemental, depending on data defined by the current fabrication process. Unlike the `spira.Cell` class that contains a list of GDSII elementals, the `ProcessLayer` class only consists of a single polygon.

Below is a basic description of the `ProcessLayer` class that automatically parses a given geometry shape and extends it with a set of functions that will later be used for device detection, and netlist extraction. Designing a physical verification tool requires knowledge of the relations between an elemental and data from any given process. Implicit in the process layer concept is the fact that this class represents a design pattern congruent with any fabrication process.

```
1  class PortConstructor(spira.Cell):
2    edge_ports = param.PortListField()
3    metal_ports = param.PortListField()
4    contact_ports = param.PortListField()
5
6    def create_edge_ports(self, ports):
7      # Algorithm that generates ports along
8      # the edges of the subject polygon.
9      return ports
10
11   def create_metal_ports(self, ports):
12     # If the subject polygon is a metal purpose
13     # layer, a port is added to the center.
14     return ports
15
```

```
16    def create_contact_ports(self, ports):
17      # If the subject polygon is a contact purpose
18      # layer, two ports are added to the center.
19      return ports
20
21  class ProcessLayer(PortConstructor):
22    # Physical Layer of the polygon, i.e. RDD.PLAYER.M6
23    player = param.FunctionField(get_player, set_player)
24
25    def create_ports(self, ports):
26      # Selects which algorithm from the PortConstructor
27      # to call based on a process independent algorithm.
28      return ports
```

Above is an over-simplified pseudocode illustration of the constituents of a process layer. The emphasis is placed on how the proposed framework can be used as building blocks, and how inheritance as a design pattern is used to add extra code artefacts, without compromising the framework core. The `ProcessLayer` class inherits from the `PortConstructor` class that coalesces a set of port methods. These methods contain algorithms to add certain ports depending on the instance parameters. The `create_ports` method in the class `ProcessLayer`, chooses which algorithm in the `PortConstructor` to execute.

3.7.1 Box Process Layer

Basic layout elementals can be constructed by extending the `ProcessLayer` class. The next example demonstrates the creation of a process layer box (by inheriting from `ProcessLayer`), which unlike the vanilla `Box` class created before; can automatically generate edge terminals, and add ports specific to the purpose of the physical layer used for construction the box.

```
1  class Box(ProcessLayer):
2
3    width = param.NumberField(default=1)
4    height = param.NumberField(default=1)
5    gds_layer = param.LayerField(number=0, datatype=0)
6
7    def create_elementals(self, elems):
8      shape = shapes.BoxShape(width=self.width, height=self.height)
9      elems += spira.Polygon(shape=shape, gds_layer=self.ps_layer.layer)
10     return elems
```

When initializing a box process layer, it is required to define the physical layer.

```
1  >>> box = pc.Box(player=RDD.PLAYER.M6, center=(0,0), w=2.0, h=1.0)
```

```
2 [SPiRA: Box] (number 60, datatype 0, center (0,0), width 2.0, height 1.0)
```

This will generate a polygon elemental, see Fig. 3.7, with a width of 2.0 and a height of 1.0. The `player` parameter states that the polygon has a metal purpose and contains a GDSII number and GDSII datatype equal to that of layer M_6, defined in the RDD.

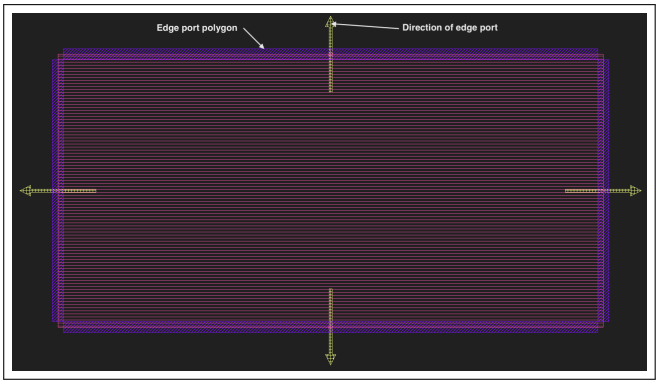


Figure 3.7: Process layer box with automatic generated edge ports.

The key take-away from this section is how using a hierarchical-template-based framework to generate layouts makes it possible to build more complex data structures, without having to compromise lower-level software components in the framework. Before detecting devices from a layout, or extracting a netlist, the underlining data structure must first be put in place.

3.8 Device Cell

The previous section illustrated how a polygon elemental can be extended to contain more complex parameterized data and methods. This section builds from this methodology to introduce a new class type used for device creation.

A Device PCell is defined by inheriting from the `spira.Device` class. This class automatically applies a set of boolean operations on all the different metal and contact layers in a designed cell instance. Once a device PCell has been created, it can be defined in the RDD and added to the LVS database. These parameterized cells can later be used as a template for device detection from a GDSII layout cell. The following two examples illustrates the simplicity of creating a parameterized device using the SPiRA framework.

```
1 class ViaC5R(spira.Device):
2   """ Via component for the AIST process. """
3
4   # Color of the graph node that will represent this
5   # cell in the extracted netlist.
```

```
6   color = param.ColorField(default=color.COLOR_GREY)
7
8   w = param.FloatField(default=RDD.C5R.MIN_SIZE, doc='Width of the via metals.')
9   h = param.FloatField(default=RDD.C5R.MIN_SIZE, doc='Height of the via metals.')
10
11  def create_metals(self, elems):
12    elems += pc.Box(player=RDD.PLAYER.R5, center=(0,0), w=self.w, h=self.h)
13    elems += pc.Box(player=RDD.PLAYER.M6, center=(0,0), w=self.w, h=self.h)
14    return elems
15
16  def create_contacts(self, elems):
17    elems += pc.Box(
18      player=RDD.PLAYER.C5R,
19      center=(0,0),
20      w=RDD.C5R.MIN_SIZE,
21      h=RDD.C5R.MIN_SIZE
22    )
23    return elems
```

The `ViaC5R` cell is parameterized by defining the width and height as parameter fields. In this example, the dimensions of the contact layer are that of its minimum design parameters, line $8-9$. By changing the width parameter of the cell, the width of the metal layers are changed. It is important to understand that this is only one way of parameterizing this specific cell, and that the designer can decide which values of a specific cell they want to parameterize. From the code above, the `create_metals` and `create_contacts` class methods are unique to classes inheriting from `spira.Device`. The `spira.Device` class has a `create_elementals` method that automatically adds the `metals` and `contacts` to the elemental list.

```
1  class Junction(spira.Device):
2    """ Josephon Junction component for the AIST process. """
3
4    color = param.ColorField(default=color.COLOR_PLUM)
5
6    def create_metals(self, elems):
7      elems += pc.Box(name='m5', player=RDD.PLAYER.M5, center=(0, 2.55), w=2.3, h=7.4)
8      elems += pc.Box(name='i5', player=RDD.PLAYER.M6, center=(0, 4.55), w=1.6, h=3.1)
9      elems += pc.Box(name='r5', player=RDD.PLAYER.R5, center=(0, 2.8), w=0.5, h=3.5)
10     elems += pc.Box(name='jj', player=RDD.PLAYER.M6, center=(0, 0.775), w=2.0, h=3.55)
11     return elems
12
13   def create_contacts(self, elems):
14     elems += pc.Box(player=RDD.PLAYER.C5R, center=(0, 3.86), w=0.9, h=0.7)
15     elems += pc.Box(player=RDD.PLAYER.C5R, center=(0, 3.86), w=0.9, h=0.7)
```

```
16      elems += pc.Box(player=RDD.PLAYER.C5R, center=(0, 1.74), w=0.9, h=0.7)
17      elems += pc.Box(player=RDD.PLAYER.I5, center=(0, 5.4), w=0.7, h=0.7)
18      elems += pc.Circle(player=RDD.PLAYER.J5, center=(0, 0), box_size=[1.3, 1.3])
19      return elems
```

The `Junction` cell is created, but not fully parameterized. This means that in order to change the polygon dimensions of this cell, a designer will have to change the physical width, height, center values of that polygon inside this class. Parameterizing all the values in the `Junction` class might be redundant, depending on the design specifications.

In most cases, only the width of the shunt resistance layer and the size of the junction layer will have to be parameterized. All other polygons will automatically adjust their respective parameters accordingly. Before creating a fully parameterized Josephson junction device, parameter restrictions must first be applied. These restrictions are defined in the *layer mapping* section in the RDD, and the next chapter will discuss this in details.

3.9 Circuit Cell

Similar to the `ProcessLayer` and the `spira.Device` class, the `spira.Circuit` class is used for circuit cell construction. A Circuit PCell is defined as a parameterized cell that consists of devices and interconnecting metal layers, called routes.

In Section 3.6 a cell was categorized into two main process independent categories, namely *elementals* and *ports*. The class methods responsible for defining each category is `create_elementals` and `create_ports`, respectively. In Section 3.8 a device cell was categorized into two sub-categories (*metals* and *contacts*), which automatically populates the `create_elementals` method of the device. This section introduces two new categories, namely *routes* and *structures*. Each of them is implemented into a cell, using the `create_routes` and `create_structures` class methods, as illustrated in the code below.

```
1  class Jtl(spira.Circuit):
2
3    def create_structure(self, elems):
4      return elems
5
6    def create_routes(self, elems):
7      return elems
8
9    def create_ports(self, ports):
10     return ports
```

Structures are defined as cell references that has a parent cell of only type `spira.Device` or type `spira.Circuit`. *Routes* are defined as metal layers that inter-

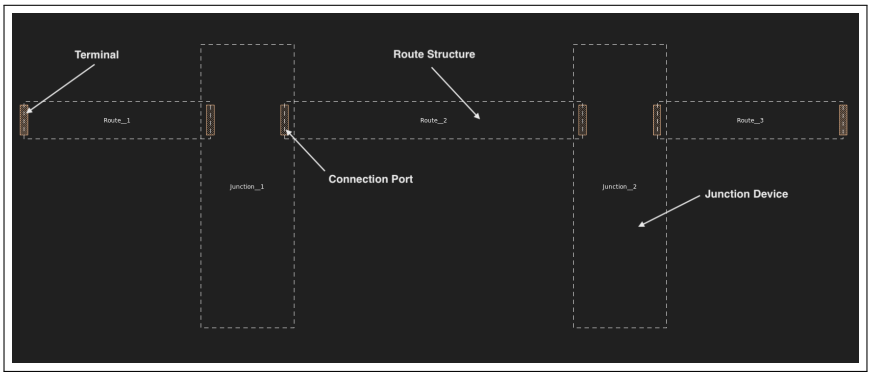
connect different structures. Routes can also connect structures to input/output terminals.

```
class __Devices__(spira.Circuit):

  jj1 = param.DataField(fdef_name='create_junction_one')
  jj2 = param.DataField(fdef_name='create_junction_two')

  def create_junction_one(self):
    jj = mit.Junction()
    return spira.SRef(jj, midpoint=(5, 0), rotation=180)

  def create_junction_two(self):
    jj = mit.Junction()
    return spira.SRef(jj, midpoint=(15, 0), rotation=180)

class __Ports__(__Devices__):
  p1 = param.DataField(fdef_name='create_p1')
  p2 = param.DataField(fdef_name='create_p2')
  p3 = param.DataField(fdef_name='create_p3')

  def create_p1(self):
    midpoint = self.jj1.ports['e3'] + [-5, 0]
    return spira.Term(
      name='P1',
      midpoint=midpoint,
      orientation=-90,
      width=1
    )

  def create_p2(self):
    midpoint = self.jj2.ports['e1'] + [5, 0]
    return spira.Term(
      name='P2',
      midpoint=midpoint,
      orientation=90,
      width=1
    )

class __Routes__(__Ports__):
  p1_jj1 = param.DataField(fdef_name='create_p1_jj1')
  jj1_jj2 = param.DataField(fdef_name='create_jj1_jj2')
  jj2_p2 = param.DataField(fdef_name='create_jj2_p2')

```

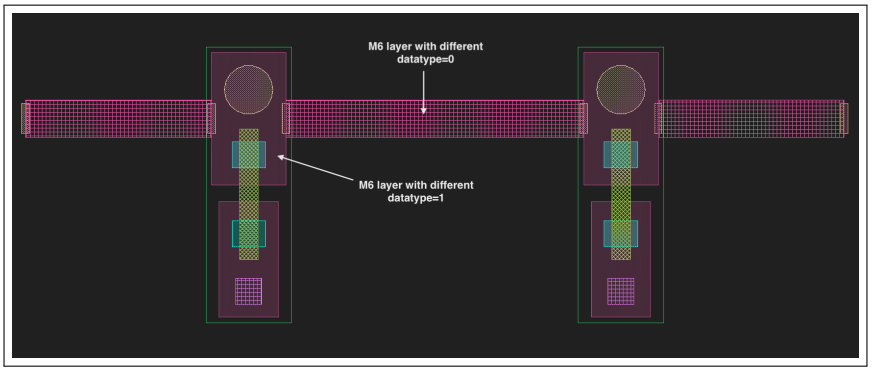
```
42    def create_p1_jj1(self):
43      R1 = spira.Route(
44        port1=self.p1,
45        port2=self.jj1.ports['e3'],
46        player=RDD.PLAYER.M6
47      )
48      r1 = spira.SRef(R1)
49      return r1
50
51    def create_jj1_jj2(self):
52      R1 = spira.Route(
53        port1=self.jj1.ports['e1'],
54        port2=self.jj2.ports['e3'],
55        player=RDD.PLAYER.M6
56      )
57      r1 = spira.SRef(R1)
58      return r1
59
60    def create_jj2_p2(self):
61      R1 = spira.Route(
62        port1=self.jj2.ports['e1'],
63        port2=self.p2,
64        player=RDD.PLAYER.M6
65      )
66      r1 = spira.SRef(R1)
67      return r1
68
69  class Jtl(__Routes__):
70    """ Parameterized Cell for JTL circuit. """
71
72    def create_structures(self, elems):
73      elems += self.jj1
74      elems += self.jj2
75      return elems
76
77    def create_routes(self, elems):
78      elems += self.p1_jj1
79      elems += self.jj1_jj2
80      elems += self.jj2_p2
81      return elems
82
83    def create_ports(self, ports):
84      ports += self.p1
85      ports += self.p2
```

```
86      return ports
```

The code above generated a parameterized circuit cell as shown in Fig. 3.8. This example uses basic Python inheritance to mimic the hierarchical representation (or layout design flow) of the created cell. In other words, the lowest level in this inheritance tree is the `__Devices__` class. The dunbar (double underscore) notation is used to indicate that this is not part of the standard SPiRA framework. Rather, it simply documents the code base. The designer can give any name to these dunbar classes. In the code example shown, the device cells are placed first, followed by the placement of ports (or input, output terminal), and finally by defining the routes that connect these device and port instances.



(a) JTL Circuit PCell showing all cell references used to construct the parameterized JTL circuit.



(b) JTL PCell showing all polygon elementals (flattened)

Figure 3.8: Results of the JTL Circuit cell.

The parent class, `Jtl`, which actually constructs the physical circuit, simply adds the parameterized methods to their respective class methods; `create_structures`, `create_routes`, and `create_ports`. The defined structure and route objects are automatically added to the instance elementals. This is automatically taken care of by the elementals method in the `spira.Circuit` class. It is important to understand that this hierarchical structure using inheritance is not required by the

framework, but is implemented for ease of reading. The example code below shows that these classes can be collapsed using an IDE to simplify navigation.

```
1  class __Devices__(spira.Circuit): ...
2
3  class __Ports__(__Devices__): ...
4
5  class __Routes__(__Ports__): ...
6
7  class Jtl(__Routes__):
8    """ Parameterized Cell for JTL circuit. """
9
10   def create_structures(self, elems):
11     elems += self.jj1
12     elems += self.jj2
13     return elems
14
15   def create_routes(self, elems):
16     elems += self.p1_jj1
17     elems += self.jj1_jj2
18     elems += self.jj2_p2
19     return elems
20
21   def create_ports(self, ports):
22     ports += self.p1
23     ports += self.p2
24     return ports
```

3.10 Conclusion

This chapter demonstrated how to design PCells using the proposed framework. Process and technology data are added to these designs, using parameters, in conjunction with the RDD. A PCell is created by inheriting from any of the three cell construction classes, defined as `spira.Cell`, `spira.Device` or `spira.Circuit`. Each of these classes add extra design patterns to help improve the flow of designing a layout.

The aim of this chapter is to introduce the successful development of a layout generator framework for an open superconducting industry, giving designers the freedom to create their own Rule Deck Database without compromising the rest of the framework, opportunity to work with any desired S-EDA tools, and to share their designs easily with others. These PCells are developed to be created independent from the fabrication process data. This allows designers to create complex designs, even if process data is not available.

Principle: Parameterized Templates *Implicit in this chapter is the ability of the parameterized methodology to implement first and second order implementations. First order: By showing how the SPiRA framework generates PCell instances and connects to process data using parameters. Second order: Due to it being possible to create parameterized layouts, it is also possible to implement patterns to manipulate more complex layouts. These patterns can be used as building blocks to add hierarchical levels of simplification, which results in the emergence of a templated framework.*

3.11 Future Work

Future work will involve updating the SPiRA core to cache parameters. The single-model approach used by the SPiRA core will simplify the implementation of caching parameters. Polygon elemental and cell stretching is also an important feature that has to be implemented. Stretching is especially important for automatically correcting a parsed layout when design rules are violated.

Chapter 4

Validate-by-Design

This chapter builds from the previous chapter, which gave a basic overview of the SPiRA design environment. In this chapter a novel method, called *validate-by-design*, is proposed that validates the design using parameter restrictions. These design restrictions are defined in the RDD, under the layer mapping category, explained in Section 2.4.2. When designing a PCell class, all defined parameters goes through the SPiRA core. This happens using a meta-configuration that will be explained in Chapter 5. Restrictions can be added to these parameters. When creating an instance the received attributes are first checked against these defined restrictions, before committing them to the instance. If restrictions are violated, the design environment will immediately throw errors on the violated parameters. This way, a parameterized cell can be designed without violating any design rules, and the instance will not have to undergo traditional design rule checks. These cell designs can then be added to the LVS database, discussed in Section 2.4.3, to be used as a template for devices detection, when extracting a layout netlist.

4.1 Objectives

With superconductor electronics still relatively immature, there are no standardized PDKs being provided by any process foundry. Instead, there are multiple foundries experimenting with different process methods. Therefore, developing a system that can easily adapt to different process data is paramount to develop a stable solution for physical design verification.

The objective of this chapter is to introduce the novel design verification method, and how the framework introduced in the previous chapter, is used to effectively define layout elementals, while applying design restrictions. This chapter discusses four different PCell examples, including their added restrictions. Each example builds on the previous one, which eventually converges to creating a simple SFQ circuit. The methodologies codified in this chapter are innate to the design flow of any circuit layout.

4.2 Via

This example illustrates designing a via device that can be used to vertically connect resistive layer, R_5, to inductive layer, M_6. The aim is to parameterize this cell using process data from the RDD to check for any parameter restrictions when an instance is created.

```
1  from spira.param.restrictions import RestrictRange
2
3  class ViaC5RA(spira.Device):
4    """ Alternative C5R via for the MITLL process. """
5
6    width = param.NumberField(
7      default=RDD.R5.MIN_SIZE,
8      restriction=RestrictRange(lower=RDD.R5.MIN_SIZE)
9    )
10
11   height = param.DataField(fdef_name='create_r5_height')
12   via_width = param.DataField(fdef_name='create_via_width')
13   via_height = param.DataField(fdef_name='create_via_height')
14
15   def create_via_width(self):
16     return self.width + 2*RDD.C5R.R5_MAX_SIDE_SURROUND
17
18   def create_via_height(self):
19     return RDD.C5R.MIN_SIZE
20
21   def create_r5_height(self):
22     return self.via_height + 2*RDD.R5.C5R_MIN_SURROUND
23
24   def create_contacts(self, elems):
25     elems += pc.Box(ps_layer=RDD.PLAYER.C5R, w=self.via_width, h=self.via_height)
26     return elems
27
28   def create_metals(self, elems):
29     elems += pc.Box(ps_layer=RDD.PLAYER.R5, w=self.width, h=self.height)
30     return elems
```

The constructed `ViaC5RA` class contains only one changeable parameter, the width of the resistor. Line $6-9$ creates a `width` parameter with the default value set to the minimum allowed size of the R_5 layer. A range restriction is applied, which checks that the value received from an instance is at least equal or larger than this value.

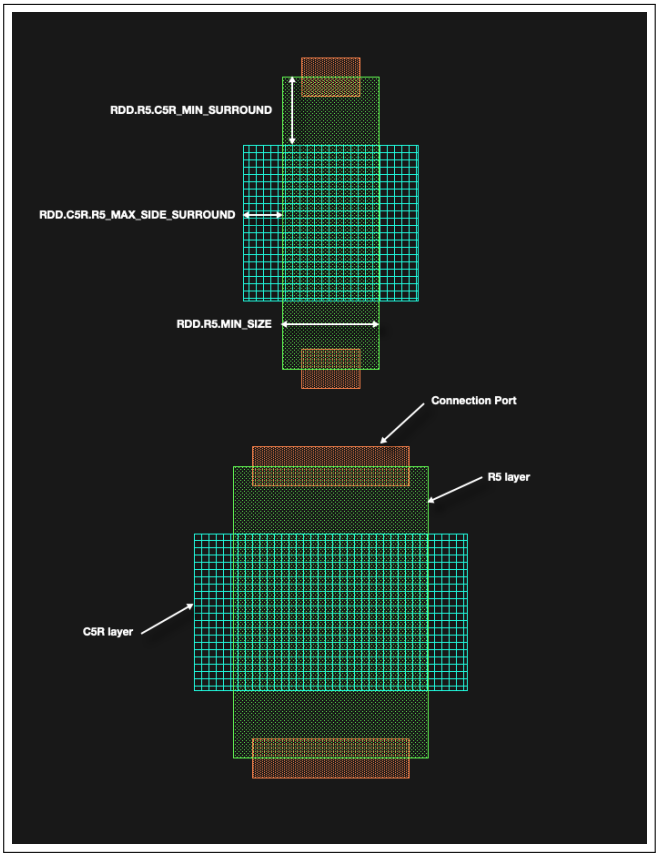


Figure 4.1: PCell for via C_5R that connects layer R_5, with layer M_6. The minimum design restrictions are shown.

The width of the via layer is calculated using the `RDD.C5R.R5_MAX_SIDE_SURROUND` constraint in line 16. This constraint only allows the C_5R layer to overlap the R_5 layer by a specific value on each side. Therefore, the width of the via layer, C_5R, is equal to the resistor width plus double this surrounding constraint. The minimum via height is set in line 19, which is equal to the minimum allowed size of the C_5R layer. Line 25 and 29, generates box process layers for layer, C_5R, and layer, R_5. This is a very basic example showing how a via PCell can be created using SPiRA without breaking any design rules. The following instance will generate the top layout shown in Fig. 4.1, with the default resistor width equal to `RDD.R5.MIN_SIZE`.

```
1 v1 = spira.ViaC5RA()
```

The following instance will create the layout in the bottom of Fig. 4.1, with a width of 1.0 μm.

```
1 v1 = spira.ViaC5RA(width=1.0)
```

The via width (parameter `self.via_width`) is automatically adjusted without violating any design rules, due to it being a function of the resistor width.

4.3 Resistor

The second example shows the design of generating a resistor layout. There are two possible resistor layouts for the MITLL SFQ5ee process: One using the alternative via, `ViaC5RA`, created in the previous section, and the other using the standard via, `ViaC5RS`. The code for via `ViaC5RS` is shown in Appendix E.1. This example illustrates how SPiRA can create a general PCell, and then swap out different cells. The following code has three changeable parameters, the width and length of the resistive layer, and the type of via connection that has to be used:

```
from spira.technologies.mit import devices as dev
from spira.param.restrictions import RestrictType

class Resistor(spira.Circuit):
  """ Resistor PCell of type circuit between
  two vias connecting to layer M6. """

  length = param.NumberField(default=10)
  width = param.NumberField(
    default=RDD.R5.MIN_SIZE,
    restriction=RestrictRange(lower=RDD.R5.MIN_SIZE),
    doc='Width of the shunt resistance.'
  )
  via = param.CellField(
    default=dev.ViaC5RS,
    restriction=RestrictType([dev.ViaC5RA, dev.ViaC5RS]),
    doc='Via component for connecting R5 to M6'
  )

  via_left = param.DataField(fdef_name='create_via_left')
  via_right = param.DataField(fdef_name='create_via_right')

  def validate_parameters(self):
    if self.length < self.width:
      raise ValueError('Length cannot be less than width.')
    return True

  def create_via_left(self):
    if isinstance(self.via(), dev.ViaC5RA):
      via = self.via(width=self.width)
    else:
      via = self.via()
    return spira.SRef(via, rotation=-90)
```

```
34
35    def create_via_right(self):
36      if isinstance(self.via(), dev.ViaC5RA):
37        via = self.via(width=self.width)
38      else:
39        via = self.via()
40      return spira.SRef(via, midpoint=(self.length, 0), rotation=-90)
41
42    def create_structures(self, elems):
43      elems += self.via_left
44      elems += self.via_right
45      return elems
46
47    def create_routes(self, elems):
48      res = spira.Route(
49        port1=self.via_left.ports['R5_e0'].modified_copy(width=self.width),
50        port2=self.via_right.ports['R5_e2'].modified_copy(width=self.width),
51        ps_layer=RDD.PLAYER.R5
52      )
53      elems += spira.SRef(res)
54      return elems
```

Line 8 defines the length of the resistive layer. This parameter contains no restrictions, since there are none to be applied from the RDD. However, by definition the length cannot be smaller than the width. The width of the instance must also be known, before a check can be applied. The `validate_parameters` method, line $23-26$, is unique to SPiRA and is automatically executed by the SPiRA core upon creating a cell instance. In this example, after having defined the `length` and `width` parameters, a pass/fail check is implemented in this method. This check will throw an error if the received length is smaller than the received width.

Line $9-13$ defines the resistor width in conjunctions with a design restriction. Line $14-18$ defines a cell parameter, with the default value referencing to the `ViaC5RS` device cell from Appendix E.1. A type restriction is added to this parameter, that only accepts one of two device types: a `ViaC5RA` device or a `ViaC5RS` device. This gives the designer the ability to create a *standard resistor* or an *alternative resistor* cell, depending on the `via` value given when creating a resistor instance. If the designer gives any other device to the instance, for example `ViaI5`, a restriction error will be thrown.

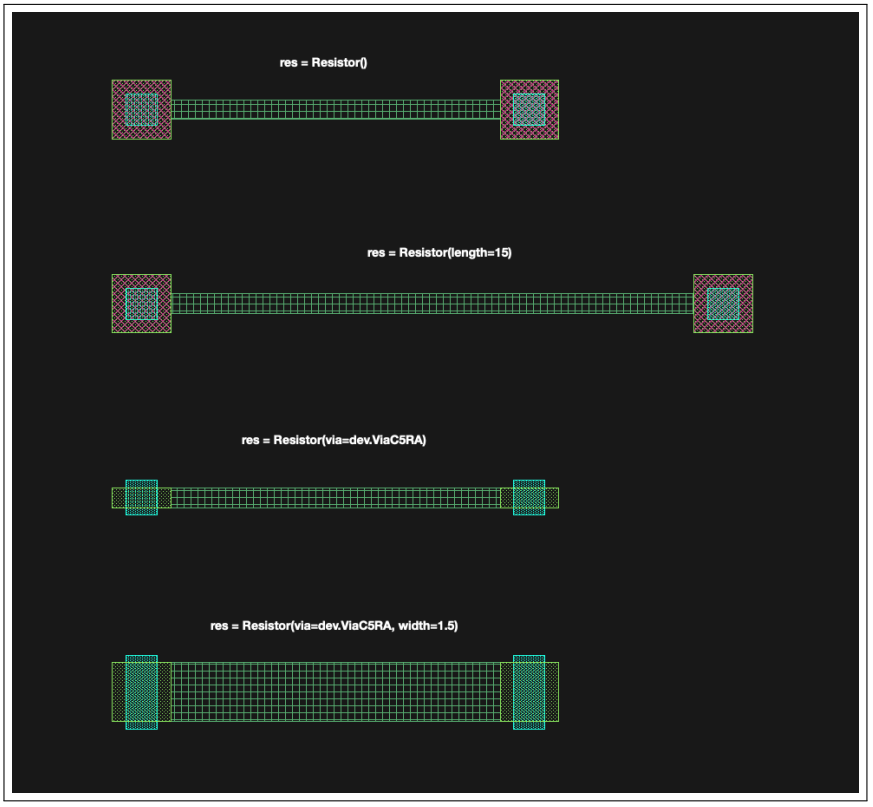


Figure 4.2: PCell for generating a resistor circuit. The via cells for this resistor cell can be swapped out to create either a *standard* or an *alternative* resistor

Lines $48 - 52$ are responsible for generating a process layer of type, R_5, between the two defined via cells. It does this using the `spira.Route` class, which takes the position and orientation of the two ports into account, and tries to automatically generate a Manhattan polygon structure between them. Line 49 and 50 defines the specific ports for each via used in constructing a resistive route. The width of this generated route is depended on the port widths. Therefore, a modified copy of these port instances must be made with the updated width defined as a parameter in the `Resistor` class. Line 51 sets the physical layer that must be used to construct the GDSII polygon of this route.

Fig. 4.2 shows four generated resistor instances. The first instance, is generated using the set default values. This resistor has a length of $10\,\mu m$, a width of $0.5\,\mu m$, and uses the standard via to connect to layer, M_6.

```
1 res = Resistor()
```

The second instance, contains the same parameter values as the first, but with the length increased to $15\,\mu m$.

```
1 res = Resistor(length=15)
```

The third instance, also uses the default values, but with the via cell swapped out. This instance uses the alternative via between layers R_5 and M_6, instead of the standard via.

```
res = Resistor(via=dev.ViaC5RA)
```

The fourth and final instance, uses the alternative via and increases the resistor width from $0.5\,\mu m$ to $1.5\,\mu m$.

```
res = Resistor(via=dev.ViaC5RAi, width=1.5)
```

For all four instances, the generated elementals and relations are prohibited from violating the defined design rules (restrictions). The cell type restriction added to the `via` parameter, constraints the designer from adding any cell instance to this parameter. It also limits the designer from making mistakes when creating a cell instance.

4.4 Josephson Junction

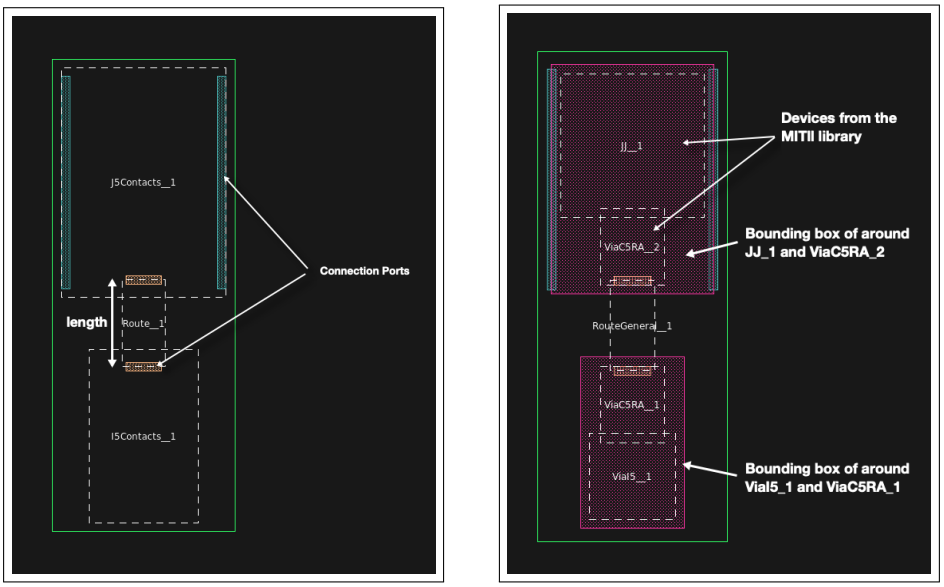
The next example illustrates the design of a Josephson junction device. It consists of multiple hierarchical cells. There are a few reasons for following this approach: First, it allows the device to be automatically updated if any of the sub-devices are updated. Second, using a hierarchical approach structures the code base, making the design more readable and maintainable for any future process changes. Third, due to the nature of creating the junction for the MITLL process, certain design implementation are best done by categorizing elementals, such as adding bounding box polygons around all elementals.

The junction device is dissected into three subcells; the upper junction part, `J5Contacts` cell, the lower junction part, `I5Contact` cell, and the shunt resistance route between these two subcells. Fig. 4.3a illustrates these cells including their unlocked ports used for connecting to metal layer M_6, shown in blue, and metal layer R_5, shown in orange.

```
class Junction(spira.Device):

  __name_prefix__ = 'Junction'

  color = param.ColorField(default=color.COLOR_PLUM)
  text_type = param.NumberField(default=91)

  length = param.NumberField(default=3, doc='Length of the shunt resistance.')
  width = param.NumberField(
    default=RDD.R5.MIN_SIZE,
    restriction=RestrictRange(lower=RDD.R5.MIN_SIZE, upper=RDD.R5.MAX_WIDTH)
  )
  radius = param.NumberField(
```

```
    default=RDD.J5.MIN_SIZE,
    restriction=RestrictRange(lower=RDD.J5.MIN_SIZE, upper=RDD.J5.MAX_SIZE)
  )

  i5 = param.DataField(fdef_name='create_i5_cell')
  j5 = param.DataField(fdef_name='create_j5_cell')

  gnd_via = param.BoolField(default=False)
  sky_via = param.BoolField(default=False)

  def create_i5_cell(self):
    D = I5Contacts(width=self.width, radius=self.radius, sky_via=self.sky_via)
    S = spira.SRef(D)
    S.move(midpoint=S.ports['R5_e2'], destination=(0, self.length))
    return S

  def create_j5_cell(self):
    D = J5Contacts(width=self.width, radius=self.radius)
    S = spira.SRef(D)
    S.move(midpoint=S.ports['R5_e2'], destination=(0,0))
    return S

  def create_elementals(self, elems):
    R = spira.Route(
      port1=self.i5.ports['R5_e2'],
      port2=self.j5.ports['R5_e2'],
      width=self.width,
      ps_layer=RDD.PLAYER.R5
    )
    elems += spira.SRef(R)
    elems += [self.i5, self.j5]
    elems += MetalBlock(ps_layer=RDD.PLAYER.M5).create_elementals(elems)
    return elems

  def create_ports(self, ports):
    for k, p in self.j5.ports.items():
      if p.name == 'M6_e1':
        el = p.edgelayer.modified_copy(datatype=199)
        ports += p.modified_copy(name='P2', text_type=self.text_type, edgelayer=el)
      if p.name == 'M6_e3':
        el = p.edgelayer.modified_copy(datatype=199)
        ports += p.modified_copy(name='P1', text_type=self.text_type, edgelayer=el)
    return ports
```

The `Junction` device takes three numerical parameters as input: the length and width of the shunt resistance, and the radius of the junction layer. Depending on the boolean value of the `gnd_via` and the `sky_via` parameters, a junction instance can be created to either contain these via connections, or not. Thus, similar to the `Resistor` PCell, by creating and parameterizing a single cell, multiple different instances can be generated by a designer. In essence, this prohibits the creation of multiple Josephson junction devices in the LVS database, since a single object can detect multiple deviations of the same device.



(a) Junction device consists of the three cell references illustrated by the dashed lines. Unlocked connection ports are also shown in this figure.

(b) The via devices including layer M_5 bounding boxes that coalesces the junction device are shown.

Figure 4.3: The constituent parts of the parameterized junction device cell are shown in these figures.

Fig. 4.3b shows the device cells that make up each of the two subcells in the junction. The reason for categorizing the junction into two subcells is to apply a bounding box to each, also shown in Fig. 4.3b by the *purple* polygons. From the `Junction` class, the `I5Contact` cell and `J5Contact` cell, are created in their respective class methods, line $24 - 34$. These cells are positioned depending on the length of the shunt resistance. This length is defined between the two *orange* ports shown in Fig. 4.3. These ports are labeled as `'R5_e2'` for each cell. Therefore, these references are placed relative to the distance between these ports, line 27 and 33. Line $37 - 42$ connects these two references with a shunt resistance by generating a route between these two ports. Lines 43 and 44 add the subcells to the junction class, and line 45 creates a bounding box around

all the elementals added to the `elems` object. The next two sections discusses the design of these subcells.

4.4.1 Lower Junction Cell

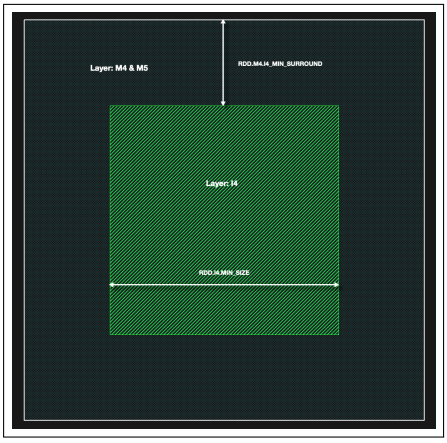
The following code shows the creation of the `I5Contact` subcell. This cell forms the bottom part of the Josephson junction. It is responsible for connecting metal layer, M_5, to metal layer, M_6, though the via layer I_5. If required, it also connects the junction to the ground plane or skyplane.

```
class I5Contacts(spira.Cell):
  """ Cell that contains all the vias of the bottom halve of the Junction. """

  i5 = param.DataField(fdef_name='create_i5')
  i6 = param.DataField(fdef_name='create_i6')

  radius = param.FloatField()
  width = param.FloatField(doc='Shunt resistance width')
  c5r = param.DataField(fdef_name='create_c5r')

  sky_via = param.BoolField(default=False)

  def create_i5(self):
    via = dev.ViaI5()
    V = spira.SRef(via, midpoint=(0,0))
    return V

  def create_i6(self):
    c = self.i5.midpoint
    w = (self.i5.ref.width + 4*RDD.I6.I5_MIN_SURROUND)
    via = dev.ViaI6(width=w, height=w)
    V = spira.SRef(via, midpoint=c)
    return V

  def create_c5r(self):
    via = dev.ViaC5RA(width=self.width)
    V = spira.SRef(via)
    V.connect(port=V.ports['R5_e0'], destination=self.i5.ports['M5_e2'])
    return V

  def create_elementals(self, elems):
    elems += self.i5
    elems += self.c5r
    if self.sky_via is True:
```

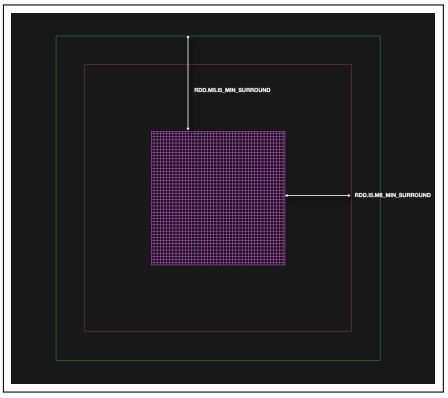
```
35        elems += self.i6
36      elems += MetalBlock(ps_layer=RDD.PLAYER.M6).create_elementals(elems)
37      return elems
38
39    def create_ports(self, ports):
40      ports += self.c5r.ports['R5_e2']
41      return ports
```

The `I5Contact` cell consists of two via devices, `ViaI5` and `ViaC5RA`, with an optional via, `ViaI6`, which is only set if the `sky_via` boolean parameter is set to true upon instance creation. This `ViaI6` device is responsible for connecting the junction to the sky plane.

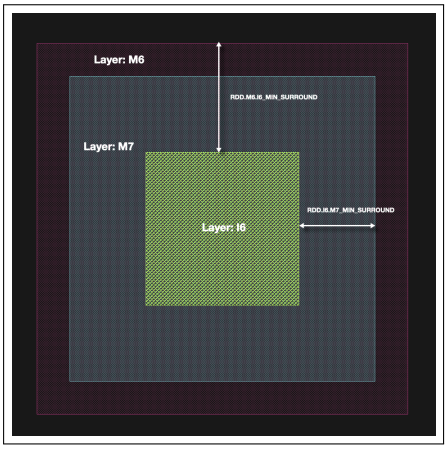
The `ViaI5` device is depicted in Fig. 4.4b, including the minimum restriction parameters between the different layers, see Appendix E.3 for the class code.



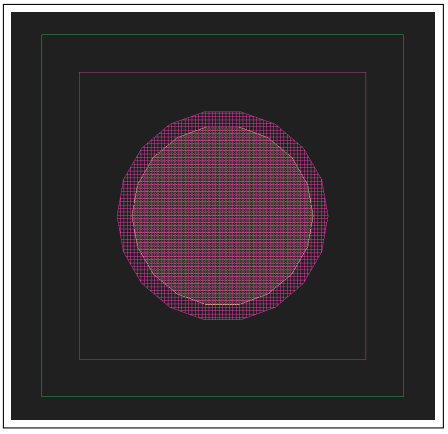
(a) PCell for via device I_4, including the minimum parameter restriction values.



(b) PCell for via device I_5, including the minimum parameter restriction values.



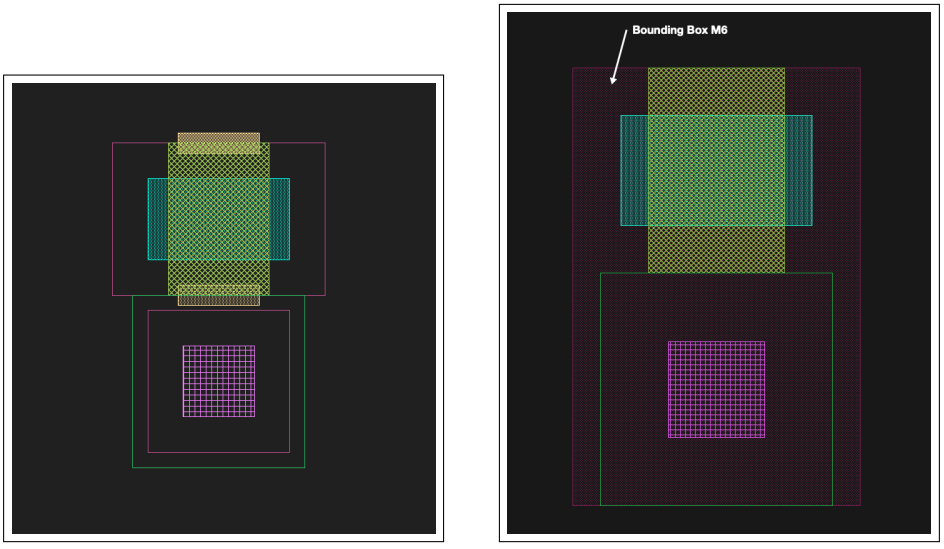
(c) PCell for via device I_6, including the minimum parameter restriction values.



(d) PCell for device JJ, including the minimum parameter restriction values. This device consists of C_5J and J_5

Figure 4.4: Views of the via devices used in the construction of the Josephson junction device.

The `I5Contact` cell in constructed by first placing the `ViaI5` cell, followed by placing the `ViaC5R` cell. The minimum spacing between layer R_5 and layer I_5 is the same as that of M_5 and I_5. From Fig. 4.4b, the `ViaI5` cell already contains a M_5 process layer, that will always satisfy the minimum resign constraints. Therefore, the port of layer R_5 in device `ViaC5R` can be directly connected to the port of layer M_5 in device `ViaI5`. The code for this is shown in the previous snippet, between lines $25 - 29$, using the `connect` method, and the resultant layout is shown in Fig. 4.5a.



(a) Connecting via I_5 with via C_5R, without violating any design rules.

(b) Connected via device (I_5 and C_5R) with a M_6 bounding box.

Figure 4.5: Results of the JTL Circuit cell.

Line 36 uses a SPiRA cell, called `MetalBlock`, to generate a bounding box around the list of elementals. This line of code shows one of the many useful "tricks" to manipulate layout data using the SPiRA framework. Line 36 does not create a new cell instance of type, `MetalBlock`, instead it only uses the bounding box algorithm inside the `create_elementals` method in the `MetalBlock` class to construct, and return, a process layer M_6 bounding box. In other words, elementals of different SPiRA cells can easily be swapped out. This technique is innate to parameterizing a received GDSII layout by swapping or interchanging elementals between different cell classes. To some extent, this means the `create_elementals` design pattern—and methods of similar kind, such as `create_routes` and `create_ports`—is a type of container that can take any list of elementals and apply a set of operations on them. These operations uses the defined class parameters as a way of connecting these elementals to process data. This process data can be defined in the RDD, but is not required to do so. The RDD is just a simple template for representing PDK data, but in realty the PDK can be parsed from any data source into class parameters. Fig. 4.5b shows the result after applying the bounding box. One property of the `spira.Device` class is that it automatically merges polygons of the same type. Therefore, the M_6 layers defined in the via devices are merged by the bounding box, resulting in a single M_6 polygon, seen in Fig. 4.5b.

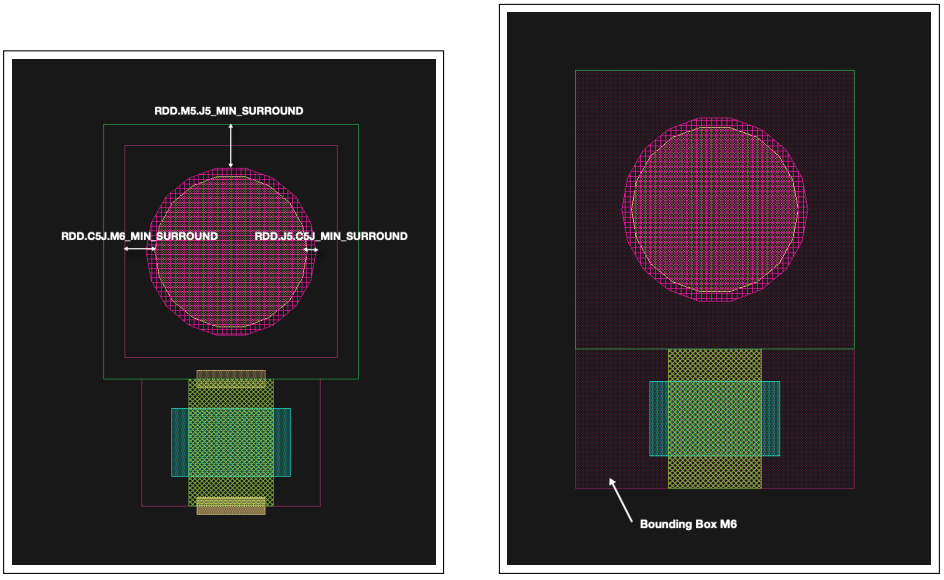
4.4.2 Upper Junction Cell

This cell is responsible for connecting layer, M_6, to layer M_5, through the junction layer J_5. The `J5Contact` cell is constructed in very much the same way

as the `I5Contact` cell. The only difference being that the `JJ` device is used instead of the `ViaI5` device.

```
1  class J5Contacts(spira.Cell):
2    """ Cell that contains all the vias of the top halve of the Junction. """
3
4    j5 = param.DataField(fdef_name='create_j5')
5
6    radius = param.FloatField()
7    width = param.FloatField(doc='Shunt resistance width')
8    c5r = param.DataField(fdef_name='create_c5r')
9
10   def create_j5(self):
11     jj = dev.JJ(width=2*self.radius)
12     D = spira.SRef(jj, midpoint=(0,0))
13     return D
14
15   def create_c5r(self):
16     via = dev.ViaC5RA(width=self.width)
17     V = spira.SRef(via)
18     V.connect(port=V.ports['R5_e0'], destination=self.j5.ports['M5_e0'])
19     return V
20
21   def create_elementals(self, elems):
22     elems += [self.j5, self.c5r]
23     elems += MetalBlock(ps_layer=RDD.PLAYER.M6).create_elementals(elems)
24     return elems
25
26   def create_ports(self, ports):
27     ports += self.c5r.ports['R5_e2']
28     return ports
```

The `J5Contact` cell uses the width of the junction parameter to create a `JJ` device (line 11), and the width of the shunt resistance to create a `ViaC5RA` device (line 16). Similar to the `I5Contact` cell, a bounding box layer is generated around the defined devices in the `J5Contact` cell, line 23.



(a) Connecting via Jj with via C_5R, without violating any design rules.

(b) Connected via device (Jj and C_5R) with a M_6 bounding box.

Figure 4.6: Results of the JTL Circuit cell.

The junctions depicted in Fig. 4.7, is the results generated from running the following three instances:

```
1 jj = Junction()
```

The first instance uses the default parameter values defined in the `Junction` class.

```
1 jj = Junction(sky_via=True)
```

The second instance includes a via connection to the sky plane, M_7.

```
1 jj = Junction(gnd_via=True, sky_via=True)
```

The second instance includes a via connection to the ground plane, M_4.

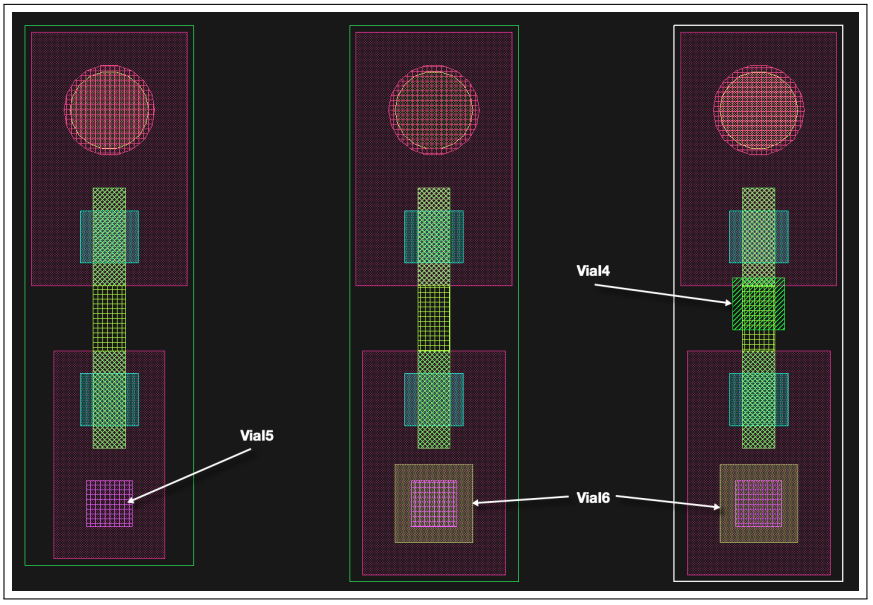


Figure 4.7: Three different Josephson junction device instances generated from a single PCell.

It is important to understand that the specific design of a Josephson junction discussed in this section is not the only way of defining such a device. Instead, what is important, is the design environment that SPiRA allows designers to effectively define layouts.

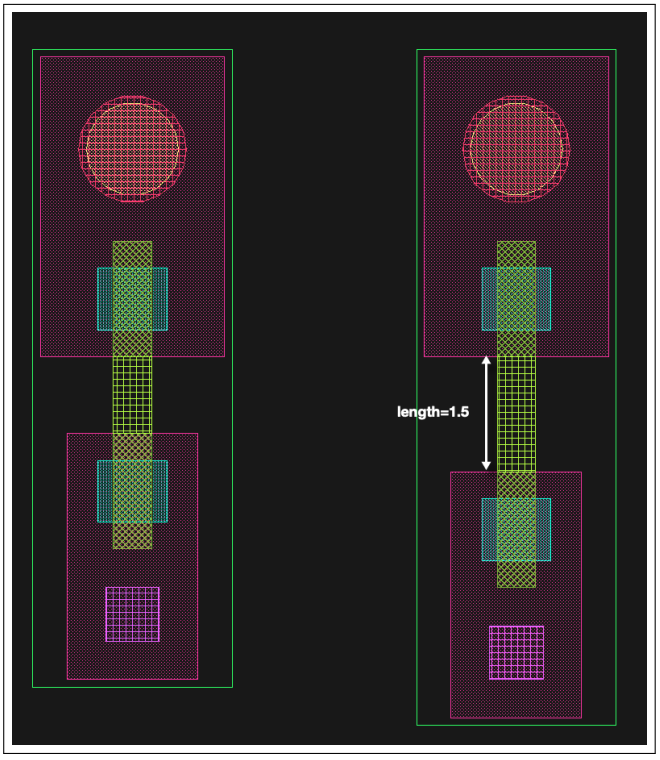


Figure 4.8: The length of the shunt resistor can be adjusted without breaking any design rules.

The two junction instances in Fig. 4.8 are generated using the following code:

```
1 jj_1 = Junction()
2 jj_2 = Junction(length=1.5)
```

The first junction has a default length of 1 μm, and the second one has a length of 1.5 μm.

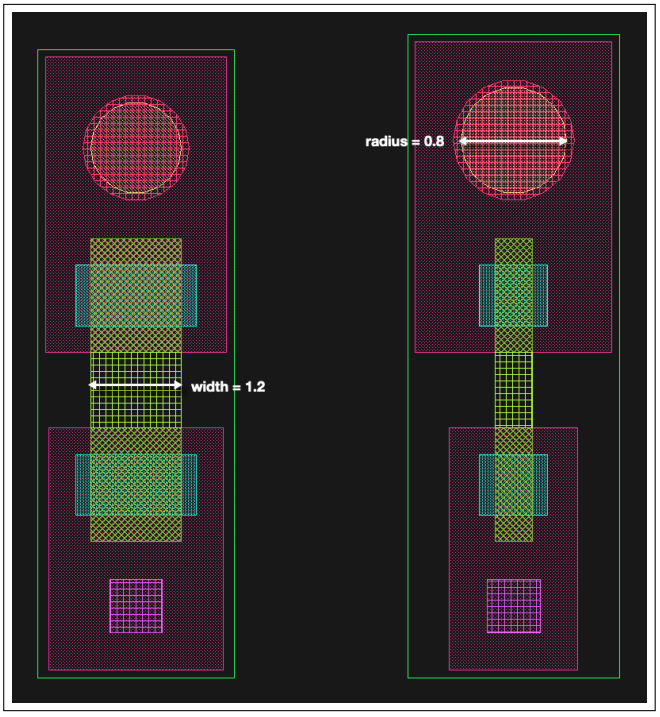


Figure 4.9: The width shunt resistor can be adjusted without breaking any design rules. Also, the radius of the J_5 layer can be adjusted and the entire PCell is automatically updated to compensate for design restrictions.

The two junction instances in Fig. 4.9 are generated using the following code:

```
1 jj_1 = Junction(width=1.2)
2 jj_2 = Junction(radius=0.8)
```

The first junction changes the shunt resistor width from 0.5 μm to 1.2 μm. The second junction changes the junction layer radius from 0.35 μm to 0.8 μm

4.5 Josephson Transmission Line

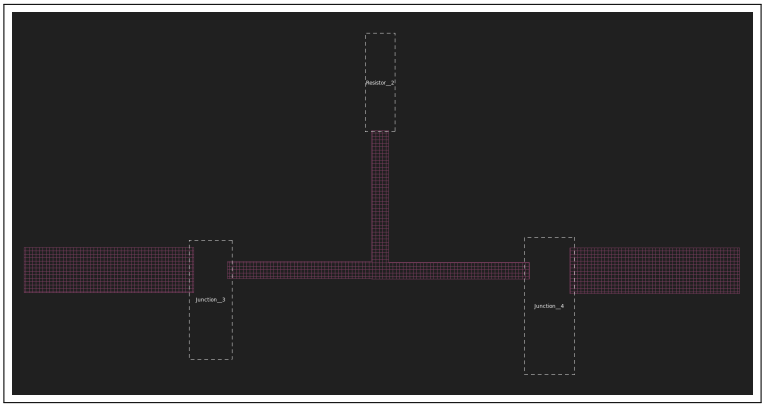
Having created a set of parameterized devices, creating parameterized SFQ circuit becomes a much easier task. This chapter explains the creation of a basic JTL circuit. More complex circuits can easily be created using these same steps. The parameters of each cell that coalesces a larger circuit can be changed programmatically. The parameters of the defined junctions in the JTL circuit, can be changed as explained in the previous section. This makes it possible for SPiRA to interface with other simulation software, such as InductEx and SPICE engines, to automatically adjust the layout after physical parameter extraction.

```
1 class Jtl(spira.Circuit):
```

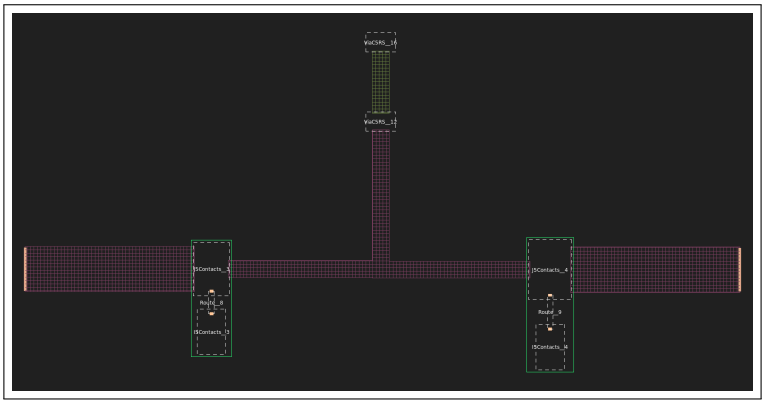
```
2
3   p1 = param.DataField(fdef_name='create_p1')
4   p2 = param.DataField(fdef_name='create_p2')
5
6   jj1 = param.DataField(fdef_name='create_jj_left')
7   jj2 = param.DataField(fdef_name='create_jj_right')
8
9   bias_res = param.DataField(fdef_name='create_bias_res')
10
11  w1 = param.NumberField(
12    default=RDD.M6.MIN_SIZE,
13    restriction=RestrictRange(lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH)
14  )
15  w2 = param.NumberField(
16    default=RDD.M6.MIN_SIZE,
17    restriction=RestrictRange(lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH)
18  )
19  w3 = param.NumberField(
20    default=RDD.M6.MIN_SIZE,
21    restriction=RestrictRange(lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH)
22  )
23  w4 = param.NumberField(
24    default=RDD.M6.MIN_SIZE,
25    restriction=RestrictRange(lower=RDD.M6.MIN_SIZE, upper=RDD.M6.MAX_WIDTH)
26  )
27
28  def create_p1(self):
29    midpoint = self.jj1.ports['P1'] + [-30, 0]
30    return spira.Term(name='P1', midpoint=midpoint, orientation=-90.0, width=4)
31
32  def create_p2(self):
33    midpoint = self.jj2.ports['P2'] + [30, 0]
34    return spira.Term(name='P2', midpoint=midpoint, orientation=90.0, width=4)
35
36  def create_bias_res(self):
37    shunt_res = cir.Resistor()
38    return spira.SRef(shunt_res, midpoint=(0, 15*1e6), rotation=90)
39
40  def create_jj_left(self):
41    jj = dev.Junction(length=4)
42    return spira.SRef(jj, midpoint=(-15, 0), rotation=180)
43
44  def create_jj_right(self):
45    jj = dev.Junction()
```

```
46      return spira.SRef(jj, midpoint=(15, 0), rotation=180)
47
48    def create_structures(self, elems):
49      elems += self.jj1
50      elems += self.jj2
51      elems += self.bias_res
52      return elems
53
54    def create_routes(self, elems):
55
56      p1 = self.jj1.ports['P1'].modified_copy(width=self.p1.width)
57      R1 = spira.Route(port1=self.p1, port2=p1, ps_layer=RDD.PLAYER.M6)
58      elems += spira.SRef(R1)
59
60      p2 = self.jj2.ports['P2'].modified_copy(width=self.p2.width)
61      R2 = spira.Route(port1=self.p2, port2=p2, ps_layer=RDD.PLAYER.M6)
62      elems += spira.SRef(R2)
63
64      p3 = self.bias_res.ports['M6_e2_left'].modified_copy(width=self.w2)
65
66      p4 = self.jj1.ports['P2'].modified_copy(width=p3.width)
67      R3 = spira.Route(port1=p3, port2=p4, ps_layer=RDD.PLAYER.M6)
68      elems += spira.SRef(R3)
69
70      p5 = self.jj2.ports['P1'].modified_copy(width=p3.width)
71      R4 = spira.Route(port1=p3, port2=p5, ps_layer=RDD.PLAYER.M6)
72      elems += spira.SRef(R4)
73
74      return elems
75
76    def create_ports(self, ports):
77      ports += self.p1
78      ports += self.p2
79      return ports
```

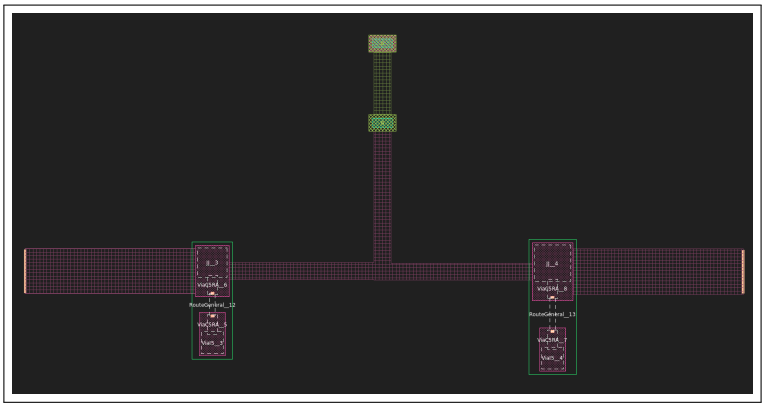
Recall from Section 3.9, that inheritance can be used to group larger designs. The JTL design in this section using the *validate-by-design* method, is not much more complex than the JTL design in Section 3.9. The overall structure of the design is similar, except for minor changes, such as different port connections, width parameters, and the addition of a bias resistor. Fig. 4.10 illustrates the hierarchical levels of the created JTL.



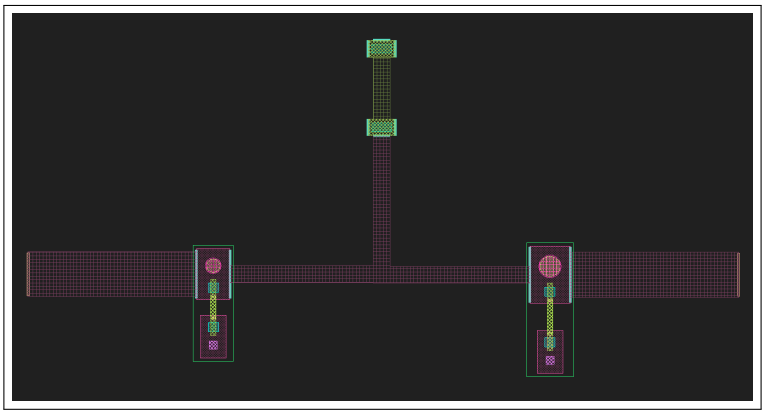
(a) Level 0: Shows the routing layers connecting the different devices.



(b) Level 1: Shows the subcells of each each device.



(c) Level 2: Shows the sub-devices of the junctions.



(d) Level 3: Shows the flattened JTL cell.

Figure 4.10: Hierarchical level of the JTL.

4.6 Conclusions

A new design verification methodology was added to the design of SDE circuits. Even though, parameterizing cells have been used to a limited extend, it has never been for full circuit design. Individual building blocks need to be designed according to a set specifications. This is typically a physical design step involving mask layout and parameter extractions. Device design is still very much a part of SDE 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.

The next chapter will have an in-depth explanation of how the SPiRA framework core was implemented, how it parameterizes class attributes, and how it utilizes metaclasses for type-checking and parameter restrictions.

4.7 Future Work

Even though design restrictions can be placed on creating single instances, checking for design rule violations between different hierarchical cells, is still to be implemented in the SPiRA framework. This, to some extent, requires traditional DRC implementations that are ran after the entire layout has been created. This is the purpose of the DRC database discussed in Section 2.5. However, these post-design DRC methods have not fully been implemented into the SPiRA framework.

Chapter 5

Framework Core

In Section 3.3, a *general* definition of a PCell is given. The SPiRA definition of a PCell in *programming* terms: "An object that receives data and describes how this data should be coalesced to generate a physical layout." From this definition it becomes apparent that a PCell is a metamodel. A PCell class acts as a data descriptor. It receives unstructured data and outputs generated structured data by modeling this data with predefined patterns. The SPiRA core is responsible to parsing this unstructured data to a single model. The previous two chapters explained the design environment for generating data-dependent cells. This chapter discusses the implementation of the framework core, how these parameters are automatically type-checked, and how the restrictions connect to these parameters before constructing the cell instance.

Primarily the core is responsible for object initialization (cells, polygons), and data binding (parameters). The fundamental structure of the core architecture is discussed, including reasons for why the vanilla Gdspy package is not sufficient to solve SDE physical verification problems, and why it is best used as a package in the SPiRA framework. The methodology of the core is to develop a templated-parameterized system that removes a lot of Python boiler plate code and makes it easier for non-programmers to design parameterized layouts. An in-depth explanation is given as to how parameters are linked to the layout elementals, and how this implementation evolved into a more general framework for advanced layout manipulation. The underlining idea is to use metaprogramming to bind metadata to a Python object before the object itself is created.

5.1 Objectives

The previous two chapter gave an introduction to the SPiRA design environment and how the framework generally works. This chapter discusses the implementation of developing a single meta-configuration system that can effectively apply the following attributes when creating a PCell or parsing a

GDSII layout:

1. Apply type-checking and restrictions on received parameters before connecting them to an instance.

2. Automatically connect the cell to a library. This library contains parameterized template cells used for device detection.

3. Set up a framework to easily create and define elementals and elemental relations. If a framework is developed that can easily define layout cells, then existing cells can be parsed and parameterized using the same design patterns.

5.2 Python Overview

IC designers need full control of their design framework to ensure that the fabricated layouts matches the design layouts. In addition, they need to be able to re-use and distribute their design layouts in a hierarchical framework that saves time and improves reliability. Previously reported LVS tools [20] was developed in C++. The GDSII parser implemented in this software toolchain for in-house use is not capable of fully extracting a layout cell. The Gdspy library written in Python is fully capable of parsing any GDSII layout. Other reasons for using Python over C++ is listed below:

- Most hardware designers have a background in Python. Therefore, developing a framework with an industry standard language will increase mass adoption.

- Python is the most used scripting language which makes is easier for defining PDK data.

- Metaprogramming methods are used to effectively couple metadata to layout elementals. Metaclasses will only be implemented in C++20.

This chapter articulates some of the core Python concepts to understand the engineering involved in developing the SPiRA framework. The most challenging area in developing SPiRA was on the software side, rather than the technical side. More precisely, coherently linking design requirements with software solutions. Python is a high-level programming language with dynamic typing and dynamic binding. Limited constraints are placed on writing classes and then instantiating objects from them. Everything in Python is an object. Functions are objects and even variables are objects. If no complex structures are needed, one can simply write functions, or a flat script can be written for executing a simple task, without structuring the code at all. At the same time Python is a fully object-oriented language. Acquiring a detailed

understanding of Python is fundamental to certain core concepts in this dissertation. Discussing these software methods is required to understand how certain artifacts of the SPiRA framework fits together and how they evolved.

5.3 Magic Methods

Magic methods and variables are methods that are automatically called by Python itself. These are special Python artifacts that are called behind the scenes. Magic artifacts provides a simple way to make objects behave like built-in types and are always surrounded by double underscores (e.g. `__init__` or `__add__`). SPiRA defines its own *magic method* design patterns which are abstracted away from higher-level methods.

5.4 Attributes

Before understanding the composition of the SPiRA framework, it is first necessary to get a basic overview of attributes in Python. Python can dynamically define new attribute fields to a class during runtime:

```
>>> class Layer(object):
...     def __init__(self):
...         self.name = 'M5'
...     def add_attr(self):
...         self.number = 50
...
>>> a = Layer()
>>> a.name
'M5'
>>> a.number
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Layer' object has no attribute 'number'
>>> a.add_attr()
>>> a.number
50
```

Attributes defined in the initialization method is available to the class instance and can be dynamically added to an instance:

```
>>> a.datatype = 1
>>> a.datatype
1
```

Class attributes are defined as the attributes of the class, rather than an attribute of an object, which is an instance of a class. The difference between a

class attribute and an instance attribute is depicted in the following example. By default, all attributes are added to the `__dict__` magic variable, which is connected to the specific class or instance.

```
class Polygon(object):
  # class attribute
  layer = 50

  def __init__(self):
    # instance attribute
    self.name = 'poly'
```

All instances of the class have access to the `layer` variable, but only the instance itself has access to the `name` variable. In Python a *namespace* is used to map names to objects, with the property that there is zero relation between names in different namespaces. They are abstracted away in Python by adding them to the *magic* dictionary variable of the class, `__dict__`. Recall that magic variables are always surrounded by two underscores and are abstract methods that adds extra *magical functions* to Python objects.

When trying to access an attribute from an instance of a class, it first tries to find the attribute in the instance namespace. If the attribute is not found, it looks in the class namespace and returns the attribute. For example,

```
pp = Polygon()

# Finds ‘name‘ in pp’s instance namespace
>>> pp.name
'poly'

# Does not find ‘layer‘ in instance namespace (pp.__dict__).
# So looks in class namespace (Polygon.__dict__)
>>> pp.layer
50
```

If a class attribute is set by accessing the class itself, using the dot operator, and will override the value for all instances:

```
>>> pp = Polygon()
>>> pp.layer
50
>>> Polygon.layer = 60
>>> pp.layer
60
```

This updates the class dictionary to `Polygon.__dict__['layer'] = 60`. Accessing the `layer` attribute, the new value (equal to 60) in the class namespace is

returned. Updating the attribute, by setting an instance, it will override the value only for the specific instance, as shown below:

```
>>> pp = Polygon()
>>> pp.layer
50
>>> pp.layer = 60
>>> pp.layer
60
>>> Polygon.layer
50
```

This only updates the `layer` attribute of the instance, `pp.__dict__['layer'] = 60`. Accessing the `layer` attribute, the new value is equal to 60, but other instances of `Polygon` will not have `layer` in their instance namespaces, so they continue to find `layer` in `Polygon.__dict__` and thus return 50. Following is a list of reasons why it is beneficial to use class attributes to define layout parameters:

- **Store metadata**: Can be used for storing Class-wide, Class-specific parameters.
- **Data constraints**: By using class attributes, it is possible to hook the attributes to custom classes that apply restrictions and type-checking.
- **Tracking metadata across all instances of a specific class**: The best example of this is incrementing the class ID after each instantiation.

5.4.1 Attribute Restriction

Applying software restrictions to a design framework can overcome potential design nuances, which otherwise may have resulted in unforeseen errors. For examples, a designer can be limited to adding a gds number to a polygon instance that exists in a given predefined set. The Polygon class has to be restricted to only accept specific attributes to its internal dictionary. Instead of having the `__dict__` variable adding attributes to objects dynamically, the `__slots__` variable provides a static structure, which prohibits additions after the creation of an instance.

```
>>> class Layer(object):
...     __slots__ = ['name']
...     def __init__(self):
...         self.name = 'one'
...     def add_attr(self):
...         self.number = 8
...

```

```
>>> layer = Layer()
>>> layer.add_attr()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in add_attr
AttributeError: 'Layer' object has no attribute 'number'
```

The `__slots__` variables is a list that contains all the attributes that can be used or set in the class. In the example code above the `number` variable cannot be set as a class attribute of instance attribute.

5.5 Mixins

Mixins in Python are used to *mix-in* extra properties and methods into a class. The delineation between using true inheritance and using *mixins* is nuanced, but it comes down to the fact that a *mixin* is independent enough that it does not act the same as a parent class. *Mixins* are not generally used on their own and are not abstract classes either, since they are not used as the base class of any concrete classes. *Mixins* are a special kind of multiple inheritance to architect code by abstracting complex functionality. A *mixin class* adds extra options to a class, whereas a *base class* defines the concrete class. This is best explained using a diagram, shown in Fig. 5.1.

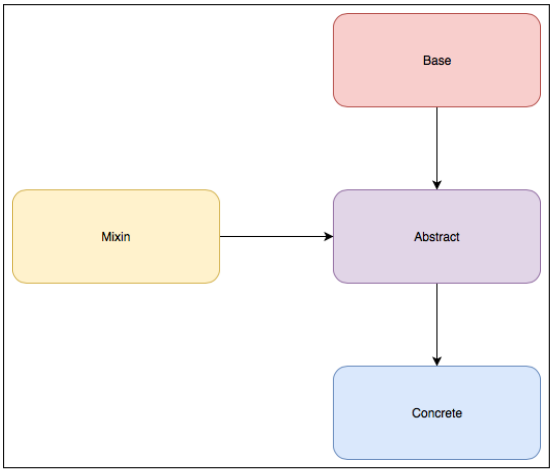


Figure 5.1: Structural implementation of base classes and mixin classes

Traditionally, *mixins* are used to expand the multi-inheritance tree; as shown in the code below:

```
class GeometryMixin(object):
  @property
  def area(self):
```

```
4       # Calculate the area of the cell.
5
6     @property
7     def center(self):
8       # Get the cell center.
9
10    @property
11    def bbox(self):
12      # Get the bounding box of the cell.
13
14  # Incorrect: Mixin added using normal
15  # inheritance, causing the inheritance
16  # tree to blow up.
17  class Cell(gdspy.Cell, __Cell__, GeometryMixin):
18    pass
19
20  # The mixin as added to the class base tree
21  # in a meta—configuration.
22  class Cell(gdspy.Cell, __Cell__)
23    __mixin__ = [GeometryMixin, InspectMixin]
```

One reason for using mixins in SPiRA is to overcome the problem of multi-inheritance. The purpose that these *mixins* serve is to provide additional features. Geometry operations such as calculating the bounding box, area, scaling, etc, can be applied on paths, cells, polygons and any shape elemental—such as rectangles and circles. A geometry *mixin* can add this functionality to all these different classes. The SPiRA framework added *mixins* in a unique fashion by automatically adding them to the class's base tree by hooking the `__mixin__` class attribute to a meta-configuration. In line 23 the `__Cell__` base-class is not shown, but is basically used to connect the created class to the SPiRA framework kernel.

Another reason for using mixins in SPiRA is for architecture documentation. *Mixins* typically add extra functionality, but are relatively stable implementations, so that future developing does not have to compensate for them.

5.6 Metaprogramming

Metadata is a set of data that describes and gives information about other data. Defining different data sets into different metadata sets constitutes the design flow of constructing the parameterized architecture:

- **Descriptive Metadata**: Data used to describe a structure using parameters that can link to a PDK.

- **Structural Metadata**: How the descriptive metadata are connected and organized. A cell class is inherently structured metadata, since the cell describes how the parameter are connected and organized.
- **Administrative Metadata**: The defined class attributes can automatically populate the `__slots__` variable using a meta-configuration. This will restrict the user to define certain keyword arguments when instantiating an object.

Just like metadata is data about data, metaprogramming is writing programs that manipulate programs, specifically class construction and object behaviours. Basic class construction in Python has to be understood, before understanding how metaclasses are constructed and the role they play in the proposed design framework.

5.6.1 Class Creation

Magic methods allow overriding the behaviour and operations of a class. Magic methods, in conjunction with metaclasses, give the ability to override the basic logic of a class. A class describes how to produce an object and all the functionalities linked to manipulating the data encapsulated in that class. Thus, classes in Python define how the instance of that class behaves. Creating a new class calls the `__call__` magic method which basically does the following:

```
def __call__(cls, *args, **kwargs):
  # Constructs the class.
  instance = cls.__new__(*args, **kwargs)
  # Initializes the class.
  instance.__init__(*args, **kwargs)
  return instance
```

Generally, classes are pieces of code that describes how to produce an object. In Python, classes themselves are objects. Since Python is a dynamically typed language, the moment the `class` line is defined, the compiler generates an object. The following code generates an object with the name Polygon:

```
class Polygon(object):
  pass
```

This class—which is an object— is capable of creating new objects—which is actually an instance. For example,

```
polygon_instance = Polygon()
```

Since the class is an object, it can be treated as such. This allows the assignments of variables, the addition of attributes and sending it as an argument to a function.

```
>>> Polygon.new_drc_rule = 'spacing' # you can add attributes to a class
>>> print(hasattr(Polygon, 'spacing'))
True
```

5.6.2 Metaclasses

Finally, metaclasses defines how classes behave, whereas a class defines how an instance behaves. In other words, a metaclass is the class of a class. A class is an instance of a metaclass. All classes in Python are created by a metaclass. The default metaclass in Python is `type`, which is itself a unique class in Python. This means that all classes are created by the `type` class. Replacing `type` is not possible, but this process can be intercepted by creating a custom metaclass, which inherits from `type`. So, a metaclass simply extends the `type` class.

```
1 class Polygon(metaclass=MetaBase):
2   pass
```

Line 1 generates the class Polygon, but before creating it in memory, Python will first checks for the `metaclass` keyword in the class arguments. If it is found, it will use it to create the `Polygon` *object*, otherwise it will use `type` to generate the object. Incidentally, using a metaclass it becomes possible to define how a class object should be constructed. This feature becomes useful when creating APIs or frameworks in order to create classes matching a specified context. A metaclass is not only useful for context mapping, but can instil extremely powerful design patterns.

In Python, the `__new__` is the constructor method and the `__init__` is the initializer. The `Cell.__new__` magic method creates an instance of the `Cell` class, but the `MetaBase.__new__` method creates the `Cell` class itself. This makes it possible to dynamically change the `Cell` class before physically creating it in memory.

5.7 Fields

In SPiRA parameters are defined as class attributes. These attributes are called *fields*, since they are custom Python objects that can be mutable, immutable or a class method. Fields are abstract classes that represents a parameter, were each parameter has its own field. SPiRA uses fields to connect class attributes to external data—such as the RDD. Consequently, they are used to define and bind PDK data to a specific class instance; that is, data retrieved from the RDD. For instance, electrical and material properties can be linked to primitives using parameters specified through a field. This makes for an extremely powerful framework that can apply boundary checks for each

parameter. In other words, errors by the designer can be checked at **run-time**. Therefore, fields are defined in the class attribute area as shown below.

```
1 from spira import param
2 class Junction(spira.Cell):
3   # Creates layer parameter by connecting
4   # to the LayerField interface.
5   layer = param.LayerField(name='J5', number=51)
6   # Creates a field that connects to a class method.
7   ports = param.DataField(fdef_name='create_ports')
```

In this code example, the `LayerField`, containing the essential parameter information, and creates a `Layer` parameter of the defined attributes in the field object. Generally, each field maps to a single piece of data of a layout. The process of binding parameter values through a field is called *dynamic binding*. Dynamic binding is done by connecting to a set of base metaclasses, which is divided into two segments; *initialization* and *description*. Fields can be summarized as follows:

- Each field is a class that connects to the base metaclass of the SPiRA core.
- Class attributes are intercepted by the `__new__` method in the metaclass before class construction.
- Fields can connect to class methods to create a set of complex parameters, or connect parameters to elementals.
- The keyword arguments of a new instance have to be defined as parameters in the class that instantiates the object.
- Field classes are responsible for type-checking parameters, and checking for any restriction violations.

PCell structures can be grouped into separate classes containing their own set of parameters and functions. Functions are dynamically linked to these structures using dynamic binding.

5.8 Initializer Module

The Initializer Module is responsible for class construction and maps the fields to the class as *class attributes*. It consists of an assembly of metaclasses and abstract base classes.

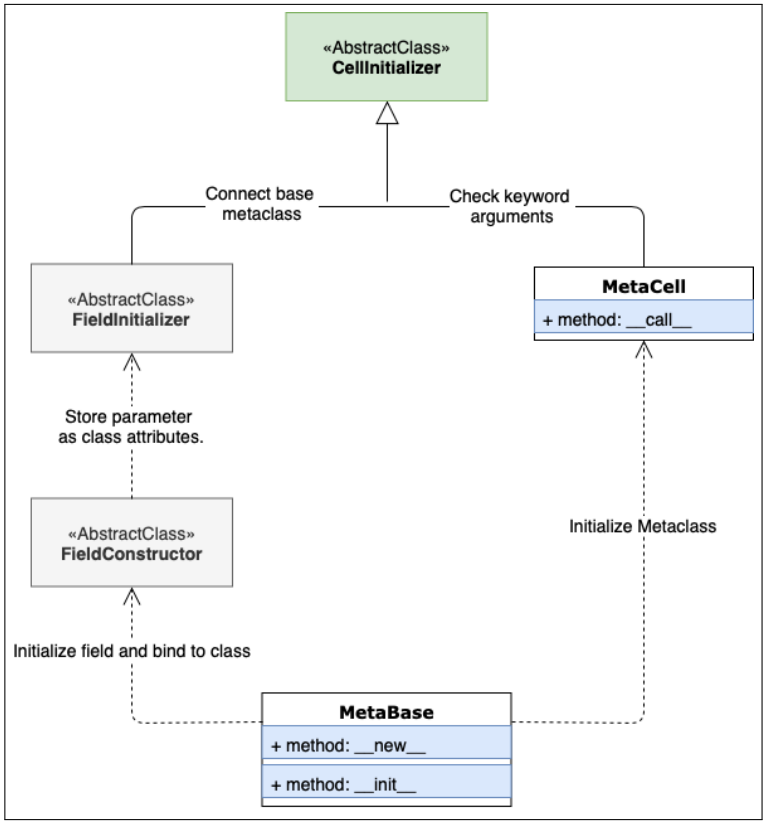


Figure 5.2: Architectural structure of Cell class initialization and construction.

This section discusses the backend of how the fields are initiated and bound to the class instance. Fig. 5.2 demonstrates the structure of a generic abstract `Field` class:

- **MetaBase**: The base metaclass to which **all** concrete SPiRA classes (Elementals) connect. This metaclass constructs and initializes the `Elemental` class. This means, all classes, that are created using the LGK, are an *instance* of the MetaBase *metaclass*.
- **MetaCell**: All defined PCells connect to this metaclass. It takes all the keyword arguments and maps them to the corresponding fields defined in the current Cell class.
- **FieldConstructor**: This class parses all fields, defined in the class, before constructing the class with MetaBase.
- **FieldInitializer**: Binds the fields to the class and sets the parameters mapped in MetaCell as the class attributes.
- **CellInitializer**: Class from which the `spira.Cell` class inherits. It connects `spira.Cell` to the Initializer Module.

5.8.1 MetaBase Class

The `__new__` class in MetaBase registers the constructed class to overcome class type duplication. There can only be one class of the same type connected to the same Process Library. A class type is defined by the name when initializing the class. Therefore, the following is not possible:

```
1  # Module: elementals.py
2  # Creates a new cell type called Junction
3  class Junction(spira.Cell):
4    print('')
5
6  # Module: mask.py
7  # Cannot create a new Junction class,
8  # since it is already registered to spira.Cell.
9  class Junction(spira.Cell):
10   print('')
```

Actually, when creating a new class that inherits from `spira.Cell`, a new device is created. This class duplication restriction was implemented so that a user cannot accidentally override a PCell already created. This restriction is also applicable to future development. Creating new `Elemental` classes, which has already been registered, will give an error before compile time. Recall that this API has been developed using inductive reasoning, so this restriction does not only make SPiRA less fragile to errors, but the system benefits from these restrictions.

5.8.2 FieldConstructor Class

The `FieldConstructor` class is best described as an intermediate step between the `MetaBase` and `FieldInitializer` classes. It acts as an interface class between `MetaBase` and `FieldInitializer`. It is responsible for parsing all data connected to the **class**, but not "act" on any of it. This class gets all the parameter names (keywords) and stores them in a magic list `__params__`. These attributes are grouped into restricted, locked, unlocked, external and internal categories. In other words, it not only parses parameters, but also makes it possible to copy attributes between different classes and different instances. This flexibility becomes invaluable when transferring metadata through the hierarchical GDSII tree, as will become apparent in later parts of this dissertation.

The `FieldConstructor` class has a custom `__get_fields__` magic method that gets all (and only) the parameter names of a defined class. It does not yet bind them to `Fields`. It uses the Python `dir` function to get a list of all the names in the current local scope. This list includes a lot of redundant and unorganised data, such as a mixture between class attributes, instance attributes and magic

method names. These attribute names are used to filter the defined field attributes, by only storing those that inherit from the `BaseDescriptorField`.

5.8.3 FieldInitializer Class

Elemental classes connect to the `FieldInitializer` base class. `FieldInitializer` is responsible for storing the parameters of an **instance**. It is then possible to do:

```
1 >>> class Junction(spira.Cell):
2       layer = param.LayerField(name='J5', number=51)
3
4 >>> jj = Junction(layer=RDD.METALS.M6.LAYER)
5 >>> jj.layer
6 [SPiRA: Layer] (‘M6‘, layer 60, datatype 0)
```

When this class is constructed the `layer` attribute in line 2 is stored by the `FieldConstructor` in the custom `__params__` magic variable. Recall from the previous discussion that `FieldConstructor` does not set the parameter to either the class or the instance. The `FieldInitializer` class is responsible for setting the parameter to the instance.

The user should not be allowed to define new keyword arguments when creating an instance. Recall from Section 5.4.1 that the `__slots__` magic variable can be used to restrict the user from setting arbitrary attributes. However, by defining class parameters the user is by definition describing the class restrictions. Inferring this statement, the `FieldInitializer` class only sets the parameters that are stored by the `FieldConstructor` class. In essence, this class sets the instance parameters while simultaneously restricting the instance to only accept specific parameters.

To bind these parameters to the instance requires replacing the `__dict__` variable of the class with a custom dictionary. The `__dict__` magic variable stores all the instance attributes, but not the class attributes. This means the instance parameters will not be detected when trying to access them with the dot operator:

```
1 >>> class C(object):
2       x = 4
3
4 >>> c = C()
5 >>> c.y = 5
6 >>> c.__dict__
7 {'y': 5}
```

Notice how `x` is not in `c.__dict__`. This is because `y` was defined for the instance `c`, while `x` was defined for its class `C`. Therefore, it will appear in the `__dict__` of `C`. The `__dict__` variable of `C` contains a clutter of other unwanted keys:

```
>>> c.__class__.__dict__['x']
4
>>> c.__class__.__dict__
dict_proxy({'__dict__': <attribute '__dict__'
of 'C' objects>, 'x': 4,
'__module__': '__main__',
'__weakref__': <attribute '__weakref__' of 'C' objects>,
'__doc__': None})
```

Therefore, the `FieldInitializer` class initializes the `__store__` magic dictionary and binds it to the created instance. The `__store__` dictionary contains all the parameters defined with the initialized values.

5.9 Descriptor Module

A field can be seen as a parameter descriptor. It describes which values should be set to a specific parameter. A *descriptor* is an object that has the following magic methods in its attributes `__get__`, `__set__` and `__delete__`. The `DataFieldDescriptor` is the descriptor class that all `Field` classes inherit from. This `DataFieldDescriptor` class populates the `__store__` dictionary, previously described. Creating the following elemental instance is not possible using default Python:

```
1  class StringField():
2    def __init__(self, value=''):
3      self.value = str(value)
4    def __get__(self, instance, owner):
5      return self.value
6    def __set__(self, instance, value):
7      self.value = str(value)
8
9  class Port():
10   name = StringField()
11
12 >>> port = Port(name='P1')
13 TypeError: Port() takes no arguments
```

This can be solved by linking the `DataFieldDescriptor` and the `FieldInitializer` classes. The following has to happen: When creating an elemental instance and setting a parameter, by passing a keyword argument (line 8 below), the parameter has to be stored in the `__store__` variable of the instance.

```
1 # Port Elemental class snippet.
2 # Class in the spira module.
```

```
3  class Port(__Port__):
4    name = param.StringField()
5
6  # Creating a port instance.
7  # Keyword argument: name='P1'
8  >>> port = spira.Port(name='P1')
9  >>> port.name
10 'P1'
```

Executing line 8 will call `FieldInitializer`, which will automatically call the setter function of `DataFieldDescriptor`. The `__set__` magic method of the field descriptor class will set the instance, `port.__store__['name'] = 'P1'`. This is possible due to the `FieldInitializer` intercepting the instantiation of `port`.

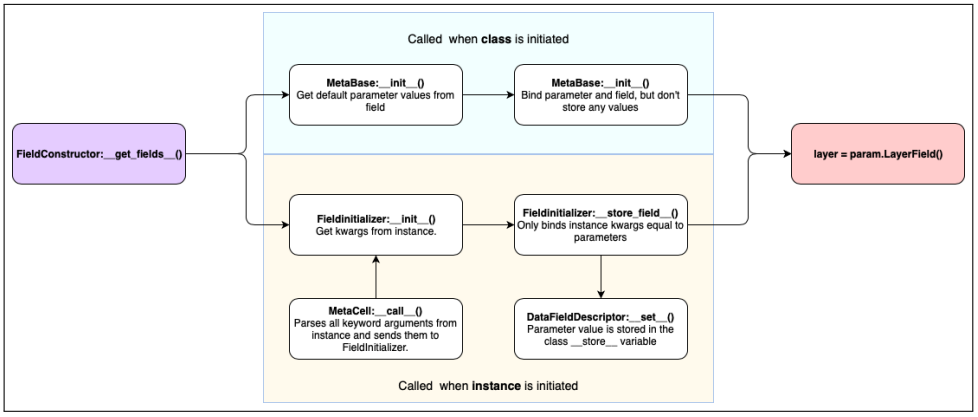


Figure 5.3: Flow diagram of how a field is initialized and bound to a class as a parameter.

Recall that a class is the instance of a metaclass, and an object is the instance of a class. Fig 5.3 shows how a parameter—which actually is a class attribute—is added to the class and the object. Now that the basic descriptor class has been discussed, a better description of a field is as follow: A parameter `Integer` will have a corresponding field `IntegerField` that wraps the `Integer` object with a descriptor class. Instead of using the cluttered `__dict__` or `__dir__` variables, which are default in Python, this descriptor class binds the parameter to the newly created `__store__` variable of the parent class. Fig. 5.3 shows how the `FieldInitializer` connects to the `DataFieldDescriptor` by setting the parameter equal to the argument specified by the instance.

```
1  class Via(spira.Cell):
2    layer = param.LayerField()
3
4  >>> Via.layer
5  <spira.kernel.DataFieldDescriptor at 0x7f217d8ef518>
6  >>> Via.layer.default
```

```
[SPiRA: Layer] ('', number 0, datatype 0)
```

The `layer` parameter is a DataFieldDescriptor class, connected as a class attribute to the `Via` class. The `LayerField` class is the `DataFieldDescriptor` that wraps around the `spira.Layer` class. The *default* attribute of the `layer` parameter is set to the `spira.Layer` class. Other properties that can be bound to this parameter are listed below:

- layer.required: The `layer` is required by the parent class.
- layer.locked: The `layer` is currently locked and cannot be used by the parent class.
- layer.fdef_name: The `layer` connects to a class method.

The field initialization is done in the `DataFieldDescriptor` base class. The following code shows the basic structure:

```
# Base class that binds the Field arguments to the
# parameter instance.
class __BaseField__(object):
  def __init__(self, **kwargs):
    # Initialize field parameter here

class DataFieldDescriptor(__BaseField__):
  def __get__(self, instance):
    pass

  def __set__(self, instance, value):
    pass
```

Presupposing that PCells will always adhere to adding GDSII elementals to a cell, a design model can be is derived: Analogous to a layout canvas, the framework can instill the adding of elementals to a cell in a similar fashion. Below follows an example:

```
class NewPCell(spira.Cell):

  def create_elementals(self, elems):
    # Add elementals to the NewPCell cell
    # by incrementing the ‘elems‘ list.
    elems += Polygon(points=[])
    return elems
```

A new SPiRA pattern is created: *Elementals must be added to a new cell class through the* `create_elementals` *method.* Implementing this functionality is simple, due to the `FieldInitializer` class.

```
class ElementMixin(object):

  elementals = param.ElementListField(fdef_name='create_elementals')

  def create_elementals(self, elems):
    result = ElementList()
    return result

class Cell(gdspy.Cell, __Cell__):
  __mixins__ = [ElementMixin, OutputMixin, InspectMixin]
```

The concrete `Cell` class extends `gdspy.Cell` and connects to the parameterized framework—through the `__Cell__` class—which in turn connects to the `MetaCell` metaclass. Because Python does not have (and does not need) a formal interface contract, like *C#* which distinguishes between abstract and interface classes, creating an interface can be done through inheritance. A mixin is used to extend the `spira.Cell` class to include this elemental functionality. The `__mixins__` parameter in the `spira.Cell` class adds these extra abstract 'interfaces' to the `spira.Cell` class.

The `ElementMixin` class is responsible for binding the `create_elementals` function to each newly created component that inherits from `spira.Cell`. The `OutputMixin` class connects the Cell (and all the GDSII primitive elements that it contains) to a set of read/write functions. The `InspectMixin` consists out of a set of logic debugging tools, which can be used to inspect the created cell, such as listing all the subcells in the layout or removing any specific subcell.

5.10 Conclusions

This chapter gave a basic overview of some core Python concepts: *Magic methods* that are largely used for data abstraction and class behaviour. *Attributes* which, by definition are the class "parameters". *Mixins* used for multi-inheritance between cohesively coupled classes. Finally, *Metaclasses* are introduced, which forms the basis of the proposed package. Understanding the constituents of classes and metaclasses serves to coalesce the cohesion between elementals and metadata.

The initial goal of the SPiRA core was to develop a module that can parse PDK data (plus any external data) and connect it to the objects created by the Gdspy library after parsing a GDSII layout. This data must be type-checked and restrictions must be applied, given by the process design rules. A solution was implemented that parameterizes layout elementals with metadata. This metadata can be connected to these objects through any format, since type-checking is implemented. However, the RDD is proposed as the standard for encapsulating process specific (PDK) data.

Chapter 6

Cell Conversion

For large-scale layout design, using a hierarchical system [27] is a promising approach, in which netlist partitioning is an important task [28]. Before the netlist of an entire layout is generated, it is possible to first detect individual devices and generate separate netlists from each [29]. Device netlists can then be included into the top-level layout netlist as a single graph node.

A netlist can only be extracted for a hand-designed layout after having converted native GDSII cell instances to `spira.Device` or `spira.Circuit` instances, through a process known as *cell conversion*. In other words, cell conversion is the process that converts a hand-designed layout into a parameterized layout.

After all sub-cells in the layout has been converted to parameterized cells, the metal polygons in the top-level cell is parameterized by converting them to GDSII paths, since they form routing structures between different devices. Finally, electrical rule checking (ERC) algorithms can be applied on the layout to create electrical connections between conducting layers and devices. Note, a PCell class already contains the necessary connection information, since the layout designer already defined routes between different device structures. Therefore, it is not always necessary to do an ERC run before netlist extraction for a designed PCell.

The previous chapters looked at creating a PCell class and defining parameter restrictions so as to ensure the instance of a created PCell object does not violate any design rules. This chapter looks at how the proposed framework can parse a layout that was designed by a layout engineer using an external layout editor into a new layout that contains device information, which includes:

- Defined contact ports that represents via connections between different metal layers inside the device cell.
- Defines which cells in the parsed layout (GDSII input file) are devices, such as Josephson junctions which is the active component in SDE.

This chapter is divided into three sections: First, how a cell is recognized as a device using pre-defined PCells. Second, once a cell is recognized as a device, how is it converted to be of type `spira.Device`. Third, how the electrical rule checking (ERC) algorithm works to detection connections between devices and connected route structures.

6.1 Objectives

The parameterized-test-pattern-methodology used to detect devices, are inherited from the PCell concept. The parametric design philosophy of the SPiRA framework describes the test cases in terms of parameterized cells. As an example: If a PCell is created then the parameters are set by the designer. If a hand-designed layout is parsed, the cell elementals are mapped to the current RDD, which extracts default parameters to corresponding elementals, and the layout is automatically parameterized. Once a cell and all its sub-cells have been parameterized, each individual cell is cross-referenced with the devices present in the LVS database. If the specific cell matches a device in the database, it is detected as a device. This chapter focuses on the updates that had to be implemented to automate this detection flow:

- Update SPiRA to parse and extend a layout cell, without creating multiple new instances.
- Update SPiRA to convert cell types. For examples, if a layout cell is detected to be a device, it must be converted from a native GDSII cell to a parameterized device cell.
- Define port connections (vertical connections) between contact-metals layers.
- Define terminal connections (horizontal connections) between metal-metal layers.

Chapter 3 introduced a templated design methodology to generate elementals, based on the received instance parameters. The macro goal is to leverage this framework, introduced for PCell creation, to automatically parameterize a hand-designed layout.

6.2 Device Detection

Device detection from a GDSII layout is done following a hierarchical approach. Each parsed cell is compared to a pre-defined device PCell included in the LVS database in the RDD.

The previous chapters, discussed designing cells using a process independent template flow. Process data is used in these designs, to describe layout

elementals and their relations, such as layer overlaps for contact detection between different metal layers.

The device detection algorithm does not need updating if the fabrication process changes, since the defined device cell is fully parameterized with data defined in the PDK. Therefore, the pre-defined device PCells act as a pattern matching template for a specific device, such as a junction.

6.2.1 Defining via contacts as ports

The via detection algorithm is based on a set of boolean operation (union, intersection, etc.) defined by die user in the LVS database. Each via requires a unique set of boolean operations that will be used to detect that specified device. The boolean operations are applied to all polygons in the GDS layout using the Clippers library C.4. The polygons that remain, after all the boolean operations of a specific via are applied, will represent the detected via. A contact port is added to the cell instance to represent the via connection.

6.2.2 Define a cell as a device

The extracted patterns from a layout cell (via ports, shapes, process layers) are compared with those defined by each device cell in the LVS database. If a match is found, the layout cell is updated to the corresponding device cell. It is important to understand that before a parsed cell is compared to a device cell, that the parsed cell undergoes a set of filtering operations to remove of any redundancies that might interfere with the comparison algorithm. A parsed cell undergoes the following filters:

1. Merge all polygons of the same process type.
2. Convert shape orientation to be counterclockwise.
3. Identical coordinate points of a shape are removed.
4. Shape points that have a turn angle of 0 or 180 degrees are removed. These are points that fall on a straight line.
5. Snap all points of a shape to grid value.
6. Apply the via detection algorithm to add ports for each contact.

Now that all shapes for both a parsed cell and the device cell have undergone the same set of filtering methods, a set of comparison tests can be executed. The first time a test fails, the algorithm will stop and the cell will not be detected as a device:

1. If the process layers in the cell and device do not match the test returns false.

2. If the amount of ports for each port-type does not match, the test returns false. For example, if a cell contains only one via port for via connection C_5R then it is not a `Junction` device, since a junction contains two C_5R ports.

3. The polygon count for each process layer must match. For example, the junction device only contains a single resistor polygon, if a parsed cell contains two resistor polygons the test will return false.

A cell netlist defines the relations between different ports in a cell instance. Therefore, the device detection algorithms can be updated to include an extra check, which compares the extracted netlist between the two cells.

6.3 Cell Swapping

Cell swapping is a technique for swapping the elementals of one cell with that of another. The main purpose of using this technique is to convert a `spira.Cell` object to a `spira.Device` object. For example, when a hand-designed layout is parsed and a cell matches a specific device defined in the LVS database (say a `Junction`) after running the device detection algorithm, then the elementals of the device cell (which is a pre-defined PCell) is swapped with that of the parsed cell. The parsed cell is then deleted from the original layout and replaced with the updated device PCell. This updated device now contains the elementals of the parsed cell, including the contact ports that defined via connection between different metal layers inside the device. This means that a device cell acts as a *container* when device detection is done on a hand-designed layout.

In order to do pattern comparisons between a test case in the LVS database—which is simply a set of device cells—and a layout cell, the layout cell has to be dissected into a set of process independent categories. As mentioned, a device PCell is created by inheriting form the `spira.Device` class and a circuit PCell is created by inheriting from the `spira.Circuit` class. The aim of this section is to discuss some of the basic changes that had to be made to these specific classes to automatically categorize a layout cell into a device or a circuit. The hierarchical approach used for extracting a layout netlist requires SPiRA to define the purpose of each cell in the layout. Similar to defining the purpose of each process layer in the RDD, the purpose of the cell can be detected as either a `spira.Device` type or a `spira.Circuit` type.

Chaining cell elementals is difficult and requires creating multiple *dummy cells*. These complications lead to multiplicative chains of unanticipated behaviours when dissecting medium complex layouts. This complexity is inherent in the kind of problem; trying to solve a hierarchical reconstruction problem following a logical sequential methodology, which consequently evolved into applying a hierarchical design pattern. The basic idea behind this simplified

implementation is that the inheritance tree constitutes the hierarchical structure of a layout. The heuristic is that there is symptomatic patterns between all layout hierarchies.

Container cells are used to overcome this design problem. This is done by dissecting an input cell into categories before creating a cell instance. More specifically, a *Container Cell* is a cell that extends another cell or receives a list of elementals which should be replaced or changed after having ran a set of tests. Subsequently, a container cell is a newly proposed design pattern rather than a new concept. The code below shows the base class of a container cell:

```
# Container Cell bass class.
class __ContainerCell__(spira.Cell):
  cell = param.CellField()

  def create_elementals(self, elems):
    elems += spira.SRef(self.cell)
    return elems
```

A Container Cell has two innate uses defined as follows: *To create a new cell instance that extends from an already created cell, or to temporarily contain created elementals, while operations are applied on them, before committing them to the new cell instance.*

6.3.1 Creating Devices

To recap from Section 3.8, device templates are composed of *metals* and *contacts*. *Metals* are GDSII polygons transformed to process layers, with a metal purpose layer. Edge ports are generated for each metal polygon, which are used for metal-to-metal connections. *Contacts* are GDSII polygons transformed to process layers, with the purpose equal to that of a contact layer. No edge ports are generated for contact layers, since they connect vertically to metal layers. Instead, ports are added in the center of each contact polygon. The default `spira.Device` class already contains a `create_metals` and `create_contacts` method, as shown below:

```
class Device(spira.Cell):
  def create_metals(self, elems):
    # Only used when a layout cell is received.
    return elems

  def create_contacts(self, elems):
    # Only used when a layout cell is received.
    return elems
```

When creating a device PCell, by inheriting from `spira.Device`, these methods are overridden on creation.

```
class Junction(spira.Device):
  def create_metals(self, elems):
    # Define junction metal polygon elementals.
    return elems

  def create_contacts(self, elems):
    # Define junction contact polygon elementals.
    return elems
```

When parsing a layout cell, the `create_metals` and `create_contacts` methods in the `spira.Device` class uses an algorithm to automatically categorize elementals into each category (metal or contact). This section focuses on explaining this implementation, as well as some of the modifications that had to be made, in order to sync device detection for both PCells and layout cells. Therefore, `spira.Device` class has two functionalities:

1. **PCell Creation:** When the `create_metals` and `create_contacts` classes are populated by creating a new PCell, the `spira.Device` class connects these elementals to a set of pattern tests to automatically add ports between overlapping contact layers (vias) and metal layers.

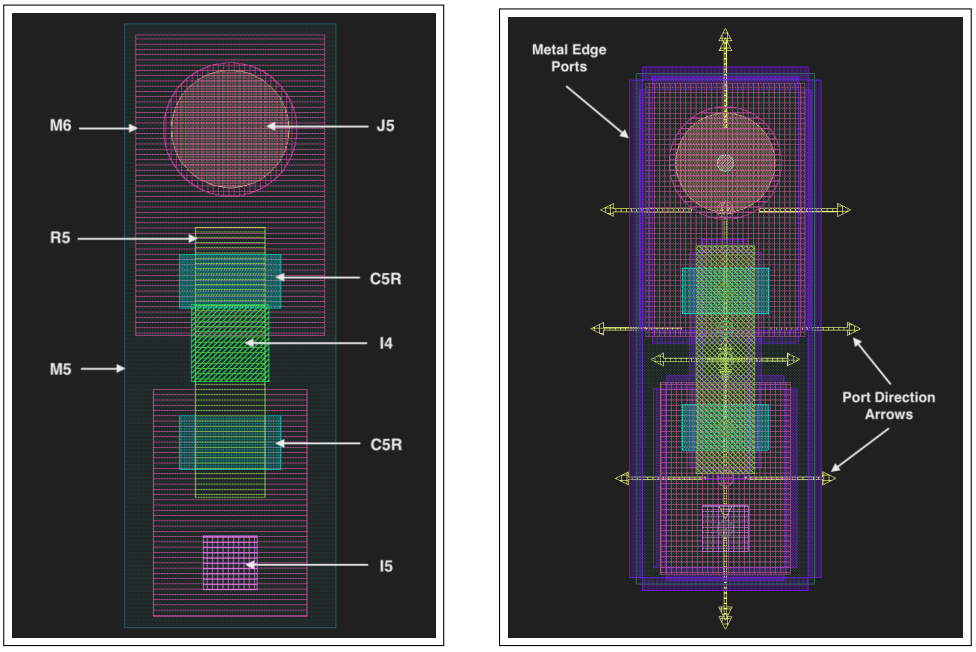
2. **Layout Cell Received:** When a parsed cell is received, it must first populate the `create_metals` and `create_contacts` classes before applying these same set of tests.

To automatically categorize the elementals of a received cell, a container is required to contain the elementals which has to be categorized, before adding them to the cell instance. The updated version of the `spira.Device` class inherits from the container cell, rather than the `spira.Cell` class.

```
# Concrete Container Cell
class Device(__ContainerCell__):

  def create_elementals(self, elems):
    # Add cell to container
    super().create_elementals(self, elems):

    # Parse the metal and contact elementals
    # according to a set of specific pattern
    # matching algorithms defined by a template
    # in the RDD.

    return elems

  def create_metals(self, elems):
```

```
16      # Extracts all metal layers from the received cell.
17      for e in self.cell.elementals:
18        for pp in RDD.PLAYER.get(['METAL', 'GND']):
19          if e.player.purpose == pp:
20            elems += e
21      return elems
22
23    def create_contacts(self, elems):
24      # Extracts all contact layers from the received cell.
25      for e in self.cell.elementals:
26        for pp in RDD.PLAYER.get(['VIA', 'JJ']):
27          if e.player.purpose == pp:
28            elems += e
29      return elems
```

Container cells shows the flexibility that SPiRA adds to manipulating GDSII cells. The default metal and contact methods in the updated `spira.Device` class, iterates over all elementals in the container cell, `self.cell`. These elementals are already converted to process layers, which contains their respective purposes. Therefore, the metal layers can be separated from the contact layers as shown between lines 15 – 29.

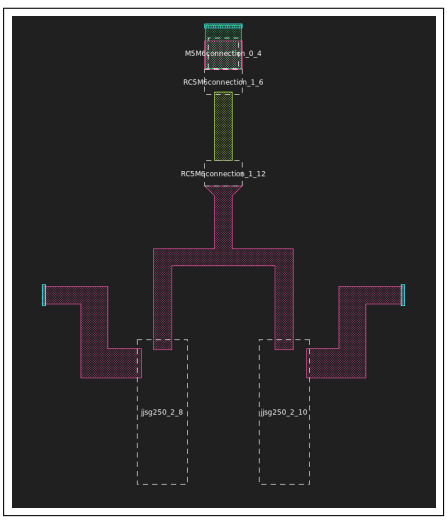


(a) Josephson junction cell layout

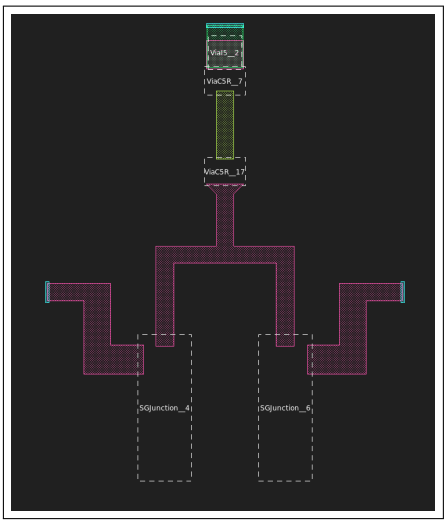
(b) Extracted device cell from layout in Fig. 6.1a.

Figure 6.1: Converted JJ layout cell to device cell, including edge ports and direction arrows.

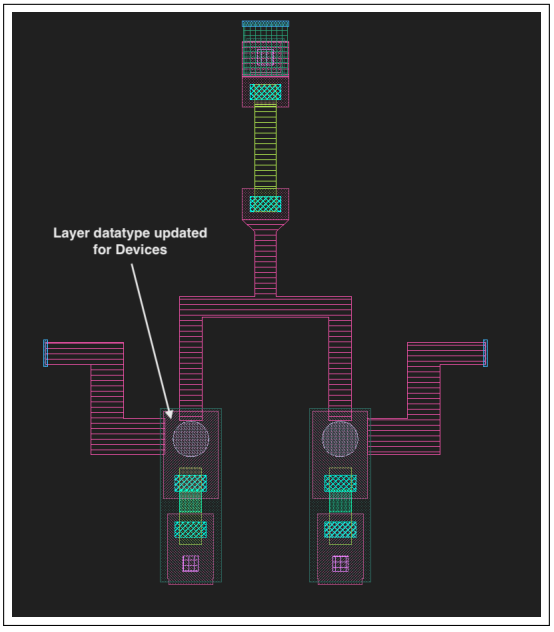
Fig. 6.1 shows a basic JJ layout for the MITLL SFQ5ee process, with the resulting device cell, after applying the conversion algorithm. The layout in Fig. 6.1b shows the automatic generation of edge ports for each metal layer in the device, along with vertical ports that represents contact connections between these metals though vias.



(a) Josephson Transmission Line (JTL) containing two junctions and three vias.



(b) Native GDSII cells are swapped out with parameterized device cells.



(c) Circuit JTL flattened showing the updated layers in each device cell.

Figure 6.2: JTL cell extraction example with subcells detected as devices.

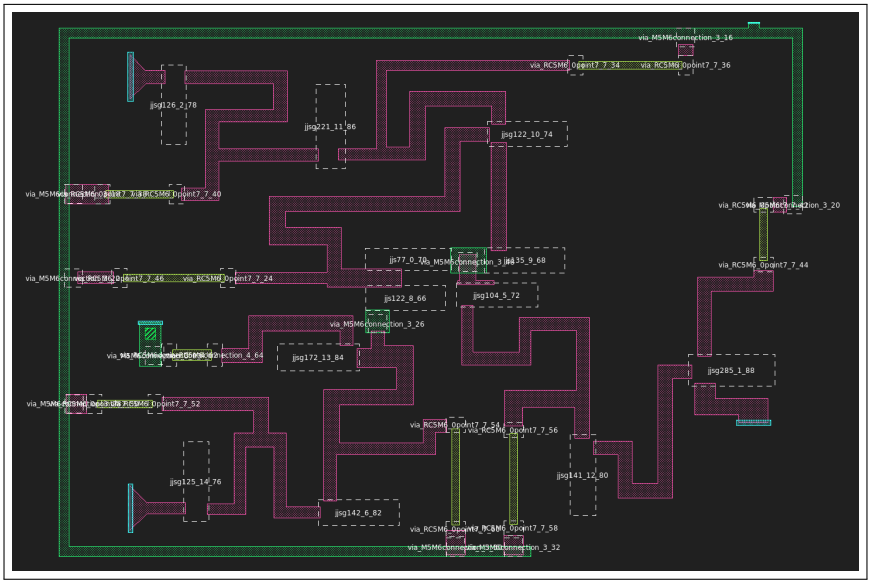
Fig. 6.2a shows a JTL layout, followed by Fig. 6.2b that shows the updated

circuit cell containing devices as subcells after device detection. Fig. 6.2c simply shows the metal layers in the device have a unique datatype values for debugging purposes.

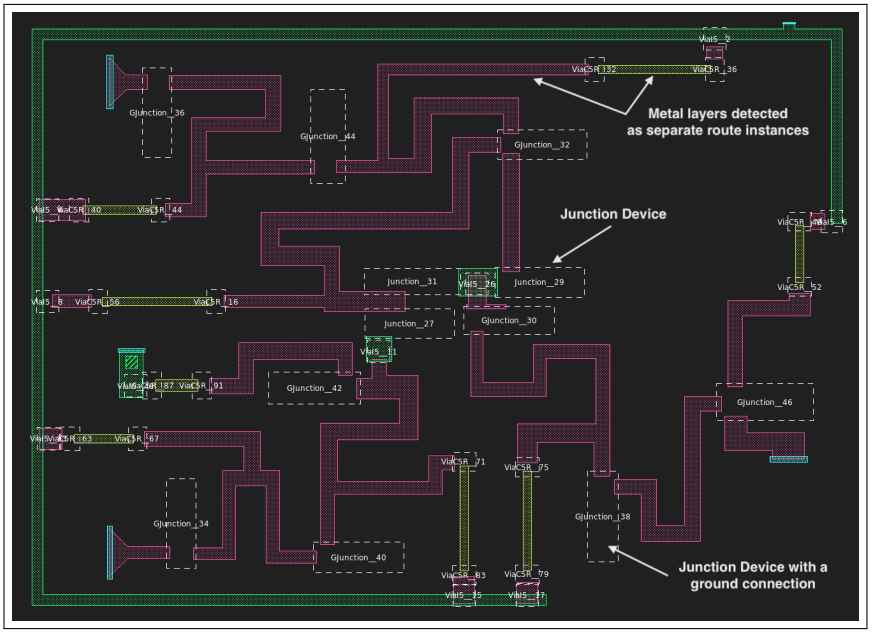
6.3.2 Creating Circuits

Structures are defined as cell references that have a parent cell of only type `spira.Device` or type `spira.Circuit`. The way SPiRA parameterises cells is by looking at the elementals present in the received cell. These elementals are compared with the elementals of each cell defined in the LVS database, using a set of pattern matching algorithms. If they match after having ran these algorithms, the cell is converted to a `spira.Device` type cell. If a cell is not detected to be a device and all of its subcells are of type `spira.Device` or type `spira.Circuit`, it is assumed to be of type `spira.Circuit`. If a subcell is of type `spira.Cell`, it tries to convert it to either a circuit or a device. If it fails, the current cell cannot be parsed, and an error is thrown.

Routes are defined as metal layers that interconnect different structures in a circuit cell. Typically, routes are metals layers in the highest hierarchical level of the current cell instance. Routes are detected by mapping all metal layers, which are in the highest hierarchical level, to a purpose defined in the RDD. Once the mapping is successful, a shape object is created from the metal polygon. Then, from this shape a new process layer is created using the corresponding purpose data structure, such as `RDD.PURPOSE.M6`. A parameterized route cell is created, and this process layer is added as an elemental. This route cell now consists of a single process layers, that has a defined purpose layer parameter and a list of edge ports that constitutes the connection points of this shape.



(a) RSFQ Not gate layout, showing all the included cell references.



(b) The resultant Not gate circuit after device detection.

Figure 6.3: The Not gate (a) before and (b) after converting it to a circuit cell.

A circuit is very similar to a device, but instead of categorising its elementals into metals or contacts, they are categorised into *structures* or *routes*. Structures are device or sub-circuit cells and routes are polygons that represents path connections between different structures. Fig. 6.3 illustrates the resultant layout of the Not gate that was converted to a circuit cell. Fig. 6.3a

shows the original gate layout, and Fig. 6.3b shows the circuit cell, including the device cells, which will be used for netlist extraction.

```
1 class Circuit(__ContainerCell__):
2
3   def create_structures(self, elems):
4     # Extract devices and sub—circuits from the received cell.
5     return elems
6
7   def create_routes(self, elems):
8     # Extract routes from the received cell.
9     return elems
```

Similar to the devices, when creating a circuit PCell, the structures and routes are manually defined. When a layout cell is passed as a parameter to a `Circuit` instance, the structures and routes are automatically extracted. In summary, a circuit cell now comprises of dependencies containing device cells that are parameterized, verified, consists of process layers, and contains port connections. Consequently, this hierarchical reconstruction makes it easier to debug all subsequent artifacts.

6.4 Electrical Rule Checking

Having detected and categorised cells, the next step is detecting electrical connections inside and between different structures. These connections are established by adding port objects in their respective positions. Therefore, the role ports play in the SPiRA framework is to create electrical connections. Input/output ports simply define the connection points of a specific cell. Port objects are categorised into two sections: Vertical ports, `spira.Port`, that connects overlapping metals through a via or a junction layer, and horizontal ports, `spira.Term`, that connects metal layers to metal layers.

6.4.1 Connect Process Data to GDSII Polygons

Recall from Section 3.7 which gave an introduction to the `ProcessLayer` class that extends a single GDSII polygon to contain information about a specific fabrication process defined in the RDD. A *process layer* is a type of multi-purpose layer and is unique to SPiRA. Fundamentally, it is a cell data structure that revolves around a single polygon, instead of working with native polygon data structures, `spira.Polygon`, to manipulate individual polygons. There are a few crucial benefits to rather work with a cell:

Depending on the purpose of the process layer, it will automatically generate a set of default ports. These ports are locked and cannot be changed by any other code artefact or designer. This is to prevent designers from accidentally

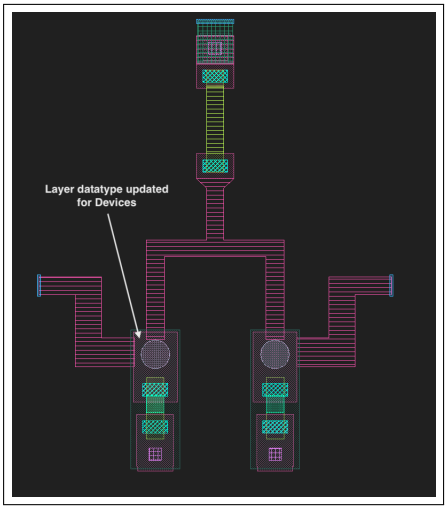
redefining or changes these ports when creating a PCell. The purpose of these ports is to represent electrical connections. When a connection is detected, the respective port is automatically unlocked, updated and made visible for debugging or verification.

```
1 >>> import spira
2 >>> from spira import pc
3 >>> from lib.mitll.pdk.process.database import RDD
4 >>> via = pc.Box(player=RDD.PLAYER.I5, width=1, height=1)
5 >>> via.ports
6 [
7   [SPiRA: Port] (name 'P1', midpoint (0,0), locked True),
8   [SPiRA: Port] (name 'P2', midpoint (0,0), locked True)
9 ]
```

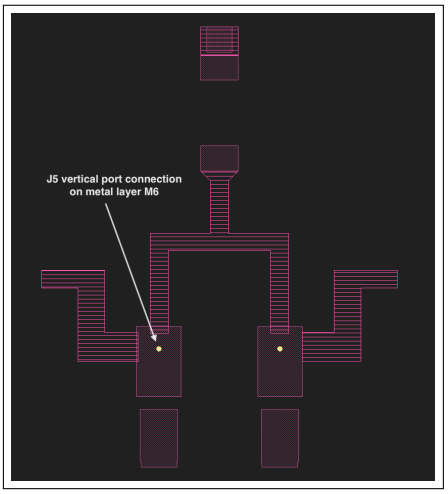
For example, when a contact layer (of type `ProcessLayer`) is created with a via purpose, two locked ports are automatically added to the center of the polygon shape, since a via layer can only vertically connect two metal layers. From the code above, line 4 creates a via process layer with physical layer I_5, width of $1\,\mu m$, and height of $1\,\mu m$. Lines $7-8$ shows the two ports generated for this process layer.

6.4.2 Contact Connections in a Device

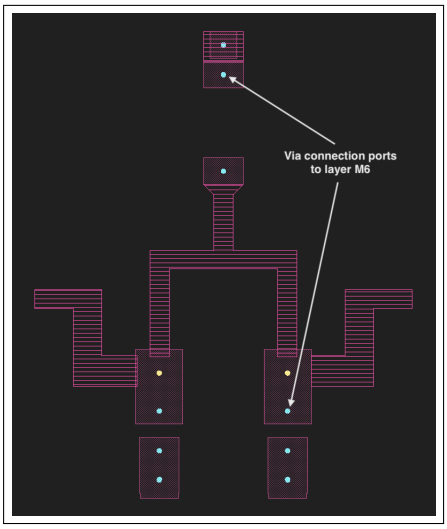
In SDE, devices typically represents Josephson junction or vias. These devices are composed of a set of contact layers, which connect different metal layers. These contacts are represented in the device as a set of ports, that vertically connects metal layers. Defined patterns are used to detect contacts belonging to a specific category. As explained in Section 6.3.1 a container cell is used to apply a set of boolean operations on a received cell, before adding a port to represent a metal contact.



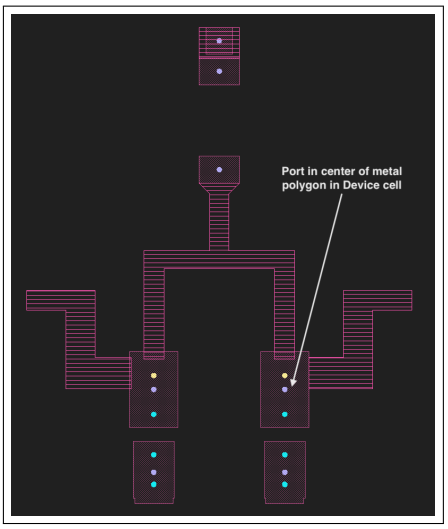
(a) Circuit JTL flattened showing the updated layers in each device cell.



(b) JTL showing the port junction port connections in each junction device



(c) JTL showing the via port connections in each junction device



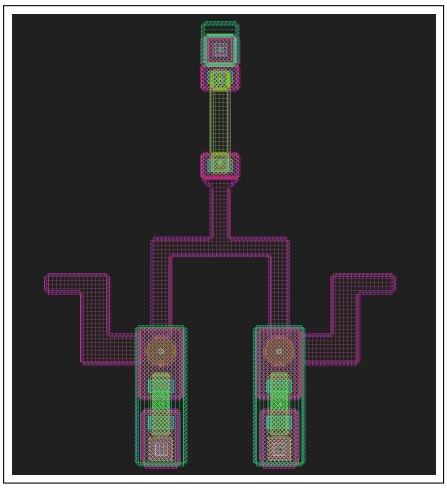
(d) JTL showing ports placed in the center of each metal layer.

Figure 6.4: This figure shows the vertical port connections placed on the metal layer, M_6.

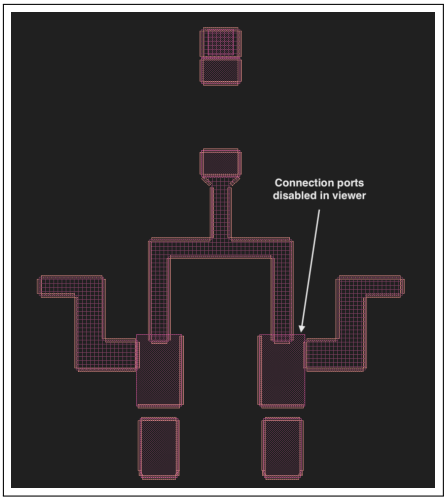
The detected port connection between each contact and metal layer for a JTL circuit can be inspected using the SPiRA viewer, as shown in Fig. 6.4. With each layer in the circuit cell being a process layer, these ports are automatically added when constructing a device.

6.4.3 Device Connections in a Circuit

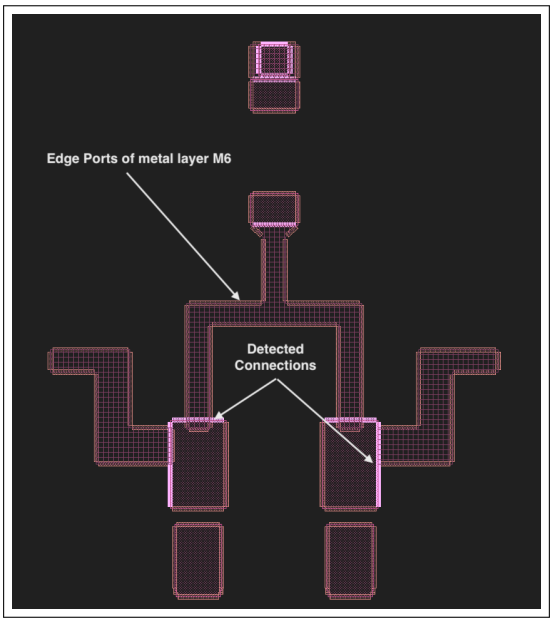
From Section 3.7 each metal layer is wrapped in a `ProcessLayer` cell, which is responsible for adding extra methods to the specific set of polygon coordinates.



(a) The JTL circuit cell including the edge ports of all metal layers.



(b) The locked edge ports of metal layer, M_6.



(c) The unlocked edge ports of metal layer, M_6, that represents route-to-structure connections.

Figure 6.5: This figure shows how the SPiRA GDSII Viewer can be used to display layer connections between different hierarchical cells.

Fig. 6.5 shows the result, when all polygon edges are activated as terminal ports, which leads to detecting horizontal electrical connections between different metal layers, either in the same cell or in lower subcells. Fig. 6.5b depicts the edge ports in the circuit cell that is locked and has no other connections. Typically, these edge ports will not be shown when debugging a SPiRA design, but are shown here for illustrative purposes. The layout if Fig. 6.5c

includes the unlocked, connected edges, with an updated `gds_layer` parameter that contains a different datatype value; hence, the different coloring scheme.

6.5 Conclusion

From this chapter, it becomes apparent that the SPiRA framework automatically detects layout components by taking an input cell and dissecting it into two different cell types, `spira.Device` or `spira.Circuit`. More importantly, a templated-parameterized methodology was used in designing these base classes, to fluently support elemental categorisation for both PCells and already designed GDSII layouts. Once correctly categorised, pattern matching algorithms are applied to effectively add ports between overlapping layers.

In this chapter the heuristic is developed that, by semantic transformation of a designed circuit layout into logical building blocks, using the fundamental structure of the GDSII file format is a valid solution for device detection. First, the parameterized functionality of the SPiRA framework is used to wrap polygon elementals into more descriptive cells. Then, these cells are used, in conjunction with templates, to describe component connections.

Principle: Hierarchical Inheritance *Using a hierarchical architecture, it becomes possible to extend each level without compromising the entire system. These hierarchical levels are connected via the parameterized nature of the system. Using inheritance as a design pattern, analogous to the hierarchical structure of a physical layout, simplifies the code implementation, serves as documentation, increases error detection, while adding to the hierarchical structure of the system to increase robustness.*

6.6 Future Work

Device detection can only be as accurate as the parameterized setup of subject devices. Therefore, updates made to designed PCells will have a direct influence on device detection. Currently, the focus of this dissertation is to accurately detect Josephson junction devices and single via connections. However, the same methodology can be used for detecting multi-stacked vias and circuit-to-circuit electrical connections.

Chapter 7

Netlist Extraction

A pragmatic approach is taken to detect the paths between structures in superconductor electronics. Using attributed graph networks for pattern analysis was introduced by Tsai and Fu [30]. It gives a straightforward representation of structural patterns. The vertices of the graph represent layout components, while the edges are the relations between these components. The proposed graph extraction methodology uses mesh elements to generate a graph, by first meshing the necessary geometries into two-dimensional triangles. This chapter describes the process of generating a graph from a set of meshed structures and connecting these subgraphs to generate a single netlist, which will be analogous to the circuit topology [31].

In the semiconductor electronics industry, component connection is done by connecting the three transistor terminals to one graph node [16]. Edge connection is easier, due to the natural topology of semiconductor circuits: Components are placed in a Manhattan-like fashion, which makes devices easier to connect, since they are connected with simple one-dimensional polygon wires [32]. Paradoxically, superconductor circuit components are interconnected through a complex polygon path network [33], [20].

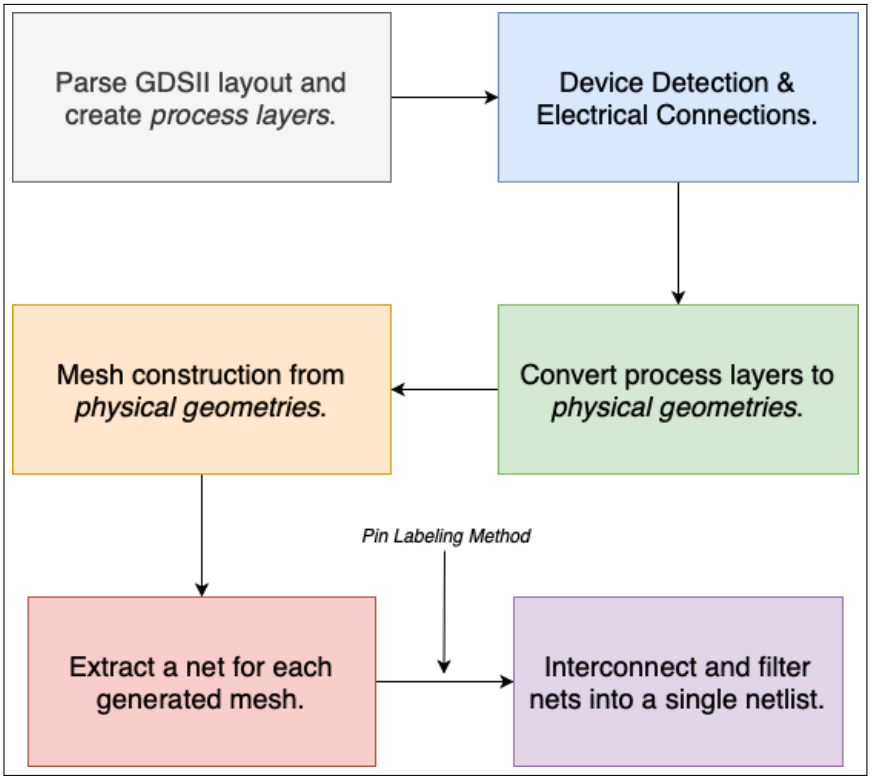


Figure 7.1: Proposed Netlist Extraction Flow.

Fig. 7.1 gives a birds-eye view of the steps implemented to generate a netlist, from layout parsing to full netlist extraction. The newly proposed method for determining component connectivity does not require dummy layers for accurate inductance extraction, as is explained in [19], [33]. Component interconnections are calculated as explained in Section 6.4; after having done device detection, and consequently having created the process layers. A graph network can be extracted by meshing the metal layers into large two-dimensional triangles. Each triangle represents a vertex in the graph. This graph is called a *net*. A device net represents the extracted graph of a device cell, and a circuit net is defined as the net that represents the toplevel layout cell (the whole circuit). In this chapter, the netlist extraction algorithm, along with the implementation thereof with the automatic detection of dummy nodes, is discussed.

7.1 Objectives

Electric rule checking was presented in the previous chapter, which connects different elementals using port objects. However, the problem with extracting a full netlist representation for superconducting circuits, requires understanding how these different structure interconnect. This chapter presents a path-

dependent solution for extracting a graph between different connected cells. This is done by discretising the generated geometry of a cell. Appendix D, gives a more integrate explanation of the geometry modelling.

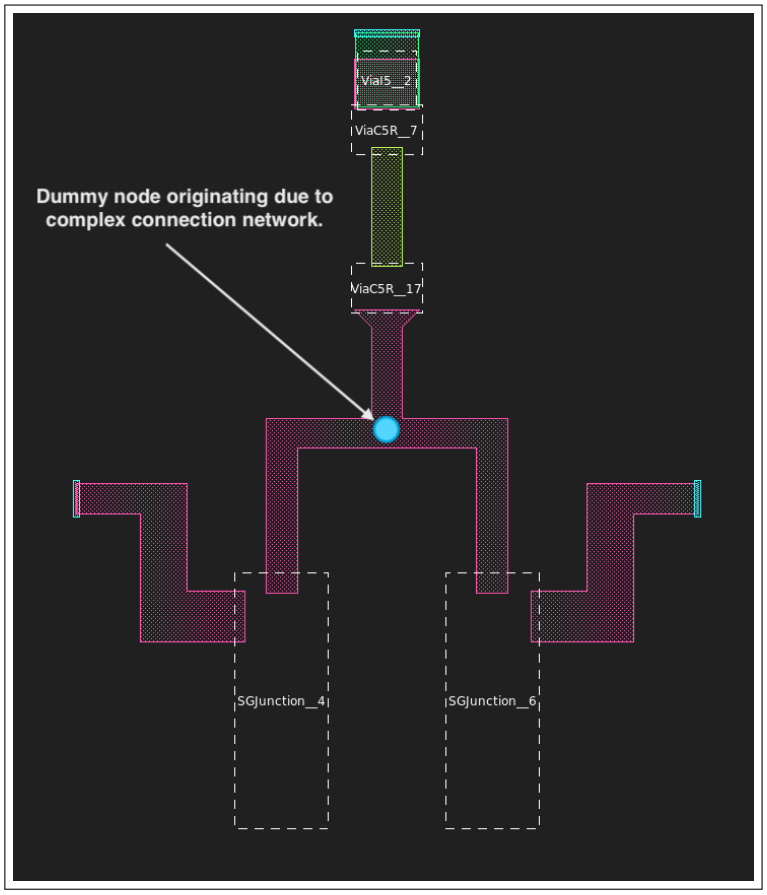


Figure 7.2: JTL RSFQ cell after device detection.

Net simplification for a generated circuit net is more problematic than that of a device net, which simply compresses all metal nodes into a single representative node. The reason being, that circuit cells interconnect structures through complex inductive paths. These paths can contain multiple layer crossovers, or consist of curved structures, that results in generating many small mesh elements, see Appendix C.3.

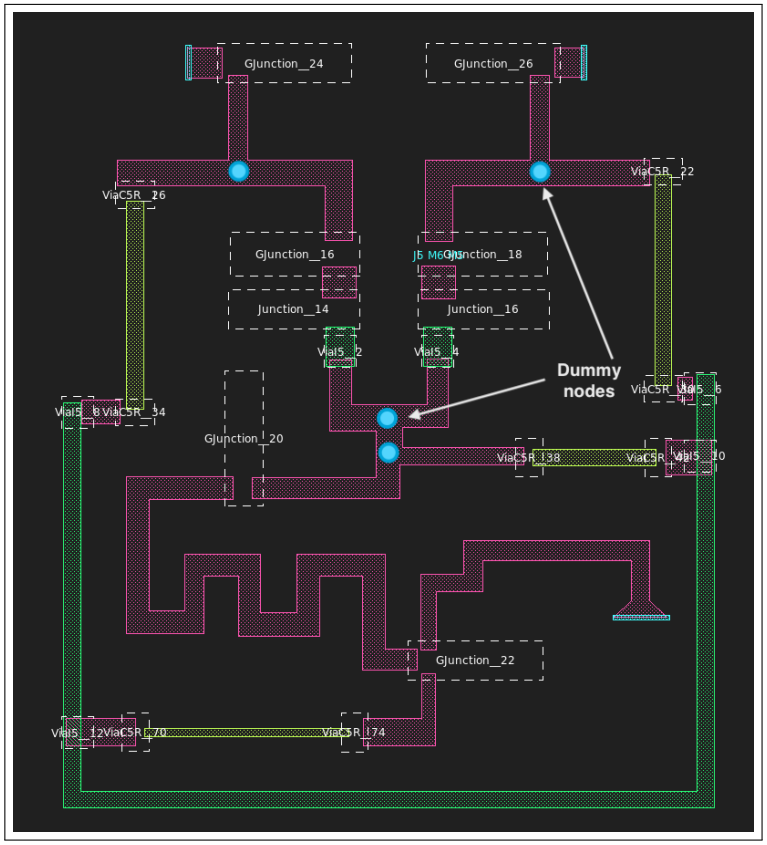


Figure 7.3: Merger RSFQ cell after device detection.

This problem does not just emerge with complex SDE circuits, but are already present in the most basic SFQ gates, as shown in Fig. 7.2, that already requires the addition of a dummy node. To illustrate the problem with a larger circuit, the Merger RSFQ circuit is used as an example, Fig. 7.3. In previous reported versions of the LVS tool [33], [20], the user had to manually label the position of different wiring layer crossings. This was done by adding a layer with a unique GDSII number on a metal polygon, shown by the dummy nodes in Fig. 7.3. The LVS solution reported in [34], also requires manually adding dummy layers to represent inductive branches. Adding these nodes manually is problematic, since the user has to to have a clear understanding of the layout and the desired netlist.

This chapter is categorized into two section. The first explains the basic netlist extraction of a Josephson junction device layout. The second explains the methods implemented to extract a netlist from a circuit layout, building from the methods used for device-netlist-extraction. The *branch-detection* method is introduced, followed by the implemented solution for automatic *dummy node* detection.

7.2 Mesh Construction using the Gmsh Library

The amalgamation of generating a graph, by geometrically following the polygonal paths, requires merging metal layers between structures. Polygon operations are done using the Clippers library, see Appendix C.4, and the implemented algorithm is as follow:

1. Coordinate orientation is checked to be in the clockwise direction. Polygons with clockwise coordinates represent positive polygons and counterclockwise coordinates represent negative polygons.

2. To overcome floating point artifacts a simplification algorithm is run on each individual polygon.

3. Polygons of the same metal layer type are merged using the union boolean operator.

It is paramount that polygons are merged, since connectivity between points within a layer is dependent on whether the points coincide the same polygon. The netlist extraction module introduces two new *elementals* to the SPiRA framework: `Net` and `Mesh`. Appendix D.2 discusses the construction of physical geometries, which is required in order to ensure the coherent connection of mesh triangles. The Pygmsh package is used as an interface between the *physical geometry* object and Gmsh library, Appendix C.1. Once the correct models have been sent to Gmsh, a two-dimensional triangular mesh is extracted. The result is a dictionary that encapsulates the generated triangle segments. Each segment is connected to a set of attributes containing node positions, elemental data and other information. A new mesh elemental is generated, using this dictionary, that inherits from the `meshio.Mesh` class.

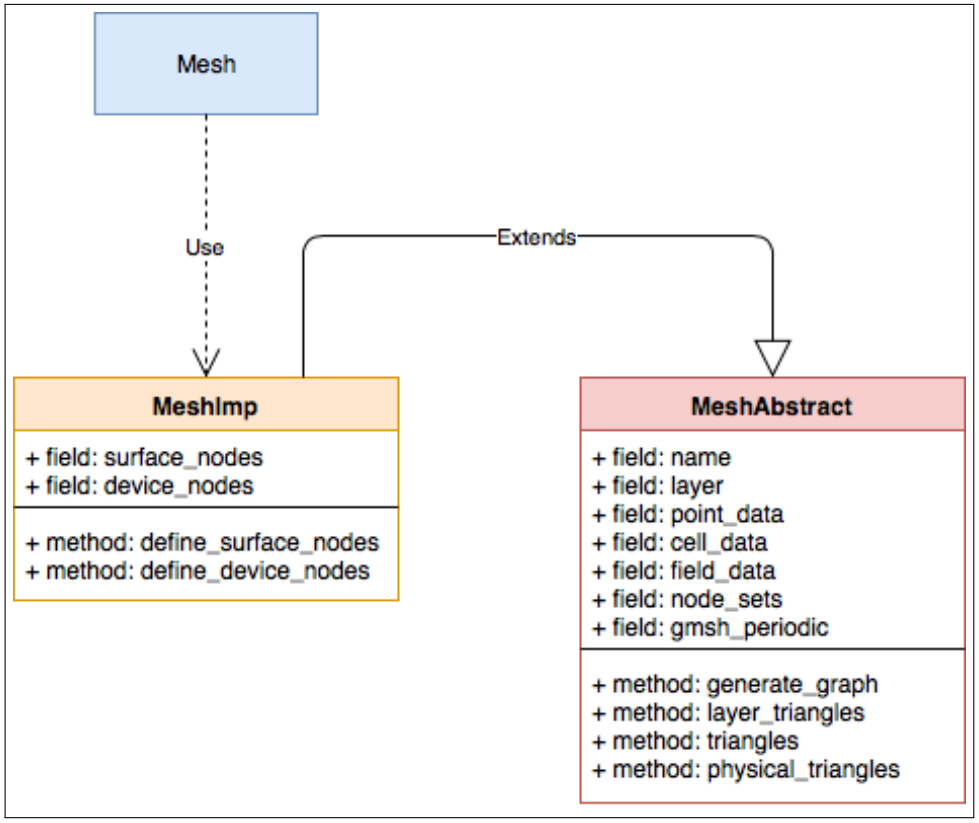


Figure 7.4: Basic functional flow of the Mesh elemental class.

`MeshImp` maps the generated graph with the process data by connecting to the RDD, Fig. 7.4. The `MeshAbstract` class extends the `meshio.Mesh` class, and also checks that the mesh data format is consistent with the current package version. Any changes in the Meshio library that breaks the system will be logged by the `MeshAbstract` class.

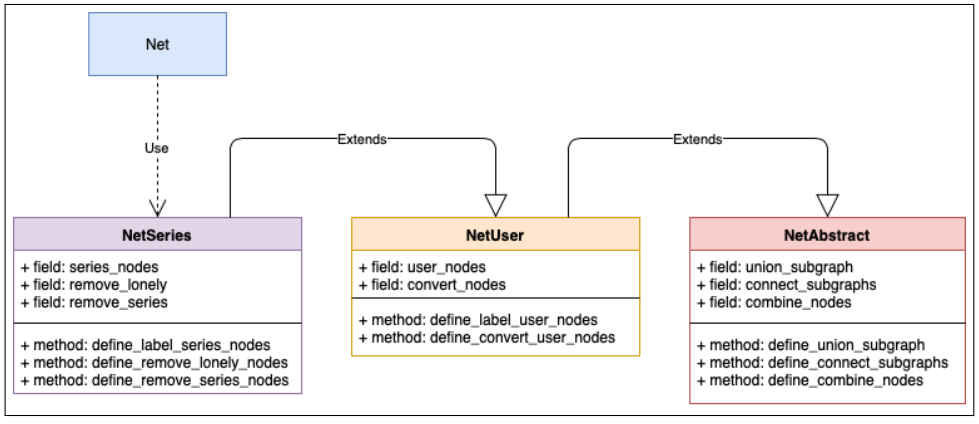
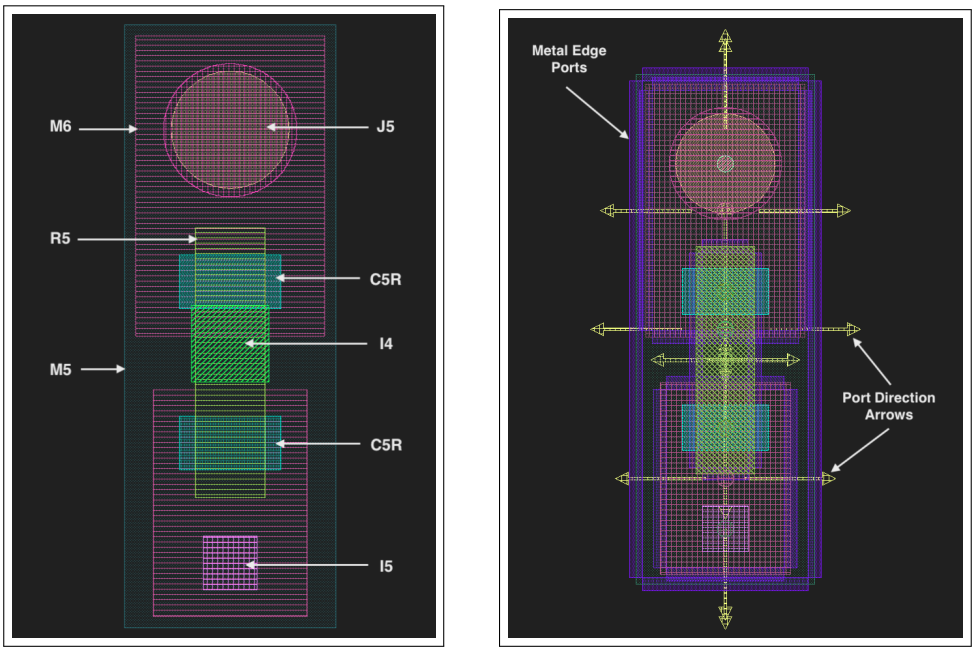


Figure 7.5: Basic functional flow of the Net elemental class.

The *Net* class is shown in Fig. 7.5, were the `NetAbstract` class is responsible for generating the elemental. The extended classes: `NetSeries` and `NetUser` are used for net simplification algorithms—explained later in this chapter.

7.3 Generating a Device Netlist

From Chapter 6, when a cell is detected as a device, it gets automatically parameterized and updated as a `spira.Device` object. Fig. 7.6 shows the layout of the junction device that will be used as an example in this section. A device contains a list of ports, each representing a contact between two or more metal layers that was detected inside the device cell, see Section 6.4.2.

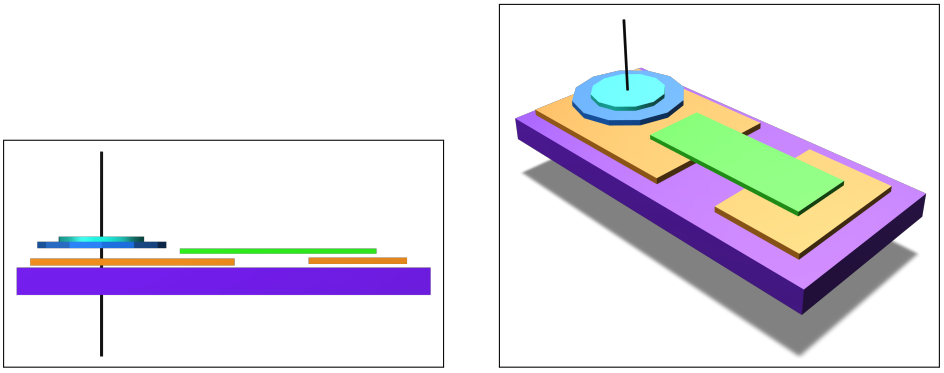


(a) Josephson junction cell layout

(b) Extracted device cell from layout in Fig. 6.1a.

Figure 7.6: Converted JJ layout cell to device cell, including edge ports and direction arrows.

A method, called *pin labeling*, is used to map these ports to the respective metal graphs, also called metal nets. Fig. 7.7 shows the basic idea behind this method. Each port position is used to map the port to a specific triangle in the mesh network, from which the metal net was constructed, using an enclosure algorithm between the triangle polygon and the port position.

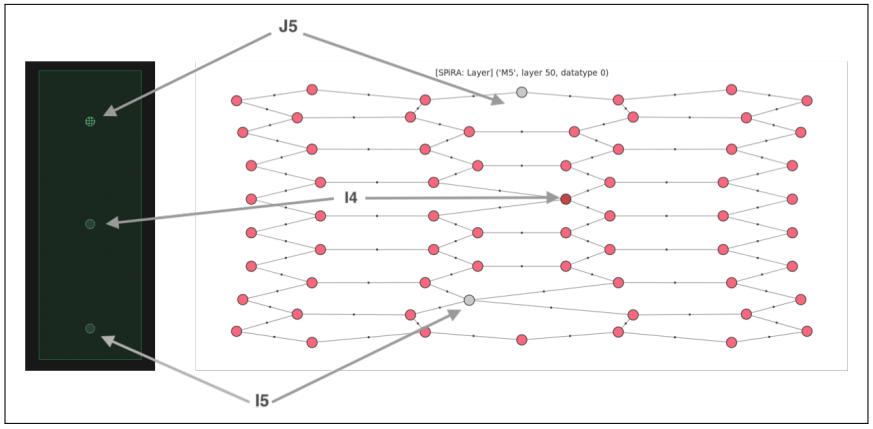


(a) Sideview of port piercing all device layers.

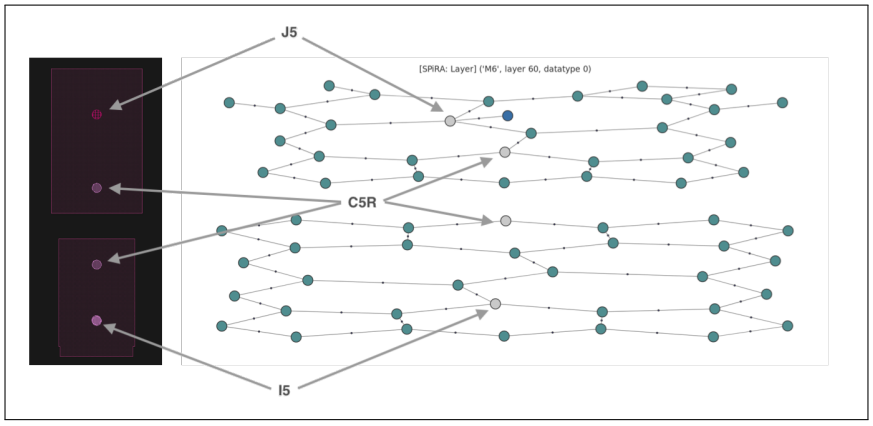
(b) Port labeling the J_5 via in a junction device.

Figure 7.7: Pin labeling method marks all passing layers.

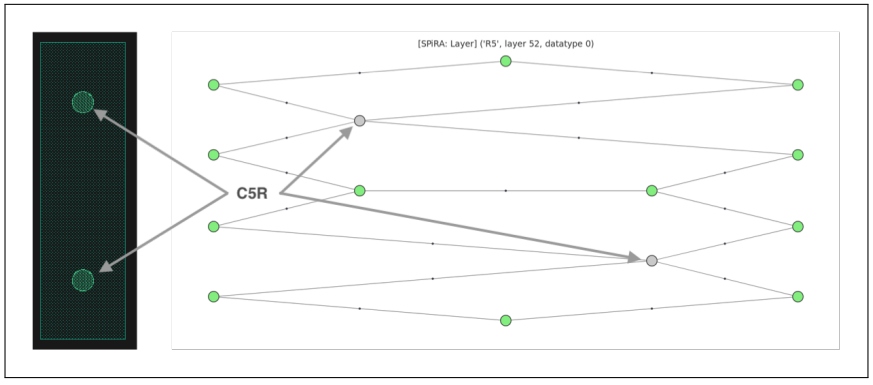
For instance, via layer I_5 (which is of type `ProcessLayer`) inside the junction device, connecting metal layers M_5 and M_6, will contain two ports. Each port has an elemental ID parameter, equal to the respective metal layer (M_5 or M_6) to which they connect. This parameter is automatically set when the device is detected. Recall from Section 6.4.2, that ports are automatically added following a pattern matching algorithm. Fig 7.8 shows each metal layer of the junction device, including port positions. The positions of these ports are used as a *pin* to pierce all triangle segments which represents the contact node in the metal net.



(a) Layer M_5 showing the positions of the detected ports of contacts I_4, I_5 and J_5.



(b) Layer M_6 showing the positions of the detected ports of contacts C_5R, I_5 and J_5.



(c) Layer R_5 showing the positions of the detected ports of contact layer C_5R.

Figure 7.8: Metal layers of the junction device including the ports that connects to each layer.

Once all metal nets coalescing the device has been labeled, they are used as subgraphs in a *graph union algorithm* [35] to be merged into a single net—

resulting in the emergence of a device net, see Fig 7.9.

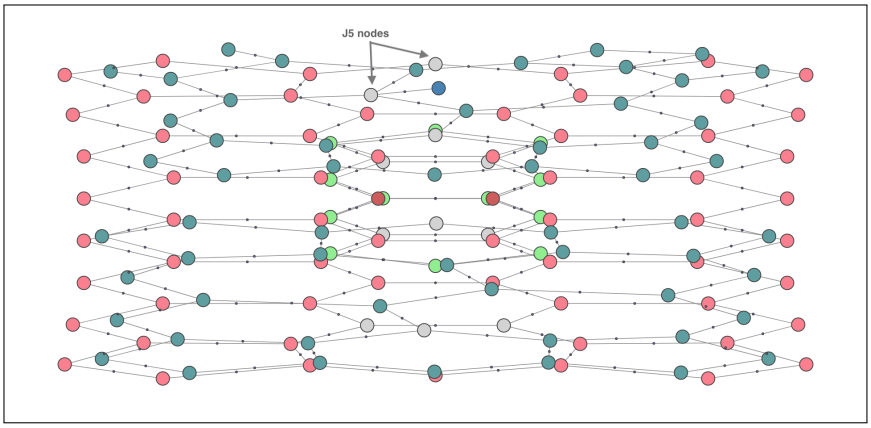


Figure 7.9: Metal nets added to the same domain.

After having constructed the device net, nodes sharing the same unique IDs are collapsed into a single node. This is done by applying a *disjoint graph algorithm* [35] to the device net, which produces the result seen in Fig. 7.10.

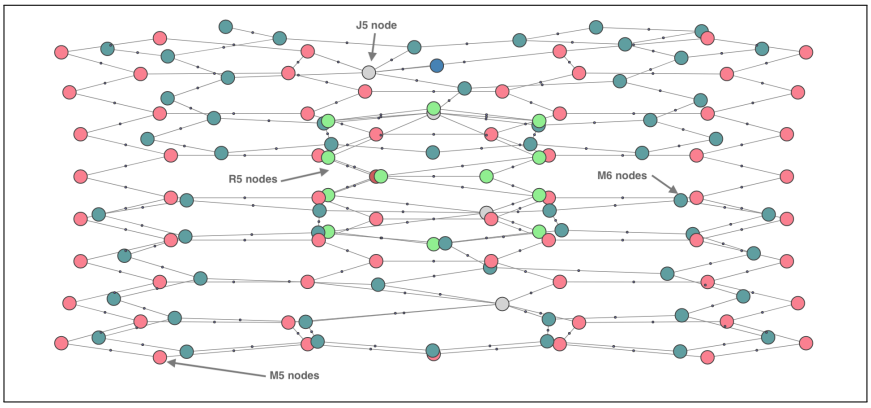


Figure 7.10: Metal nets are connected by connecting their shared via nodes.

Finally, all metal nodes are collapsed into a single node to generate the final device netlist, shown in Fig. 7.11. The full Josephson junction device netlist in this example, consists of three metal-to-metal vias (one I_5, and two C_5R), one sky-plane via (I_6), one ground-via (I_4), and one junction via, (J_5), between layers M_5 and M_6.

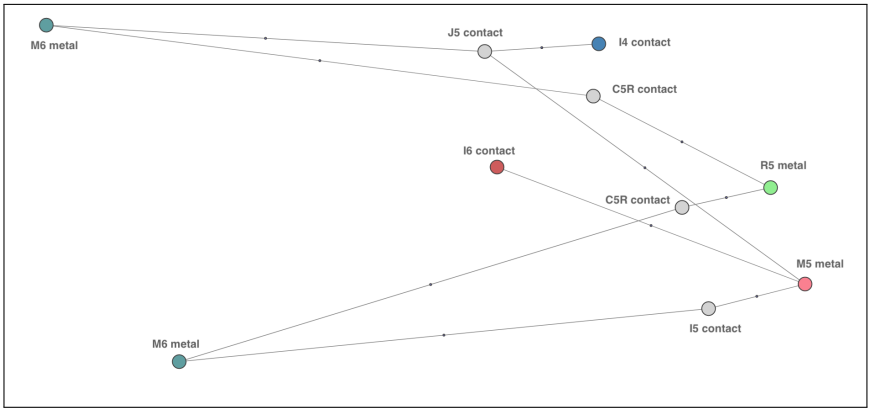


Figure 7.11: All metal nodes of the same type are collapsed into a single node.

7.4 Generating Circuit Netlist

A device net describes how the contact layers inside a device connects different metal layers, resulting in a netlist representation for the device topology. A circuit net primarily describes how devices are connected in a higher hierarchical level. In a device net the majority of nodes represents contact layers, in a circuit net the majority of nodes represents structures (devices and circuits). Recall from Chapter 6, that a circuit cell is similar to a device cell, with the major exception being that metal layers in a circuit can create complex interconnections, while that of a device is planar. Therefore, a device is analogous to a circuit with minor changes; that is, a circuit net can be delineated using the same methods for that of constructing a device net. Consequently, a circuit net is constructed using the following steps: First, meshing the metal layers and generating metal nets, which in this case connect devices instead of contact layers. Second, using the *pin labeling* method to detect nodes that represents devices. Third, applying the union- and disjoint graph algorithm to connect all metal nets into a single circuit net. Fig. 7.12 illustrates the sequential steps of filtering a netlist representation of a circuit, after having constructed a netlist from the generated mesh network.

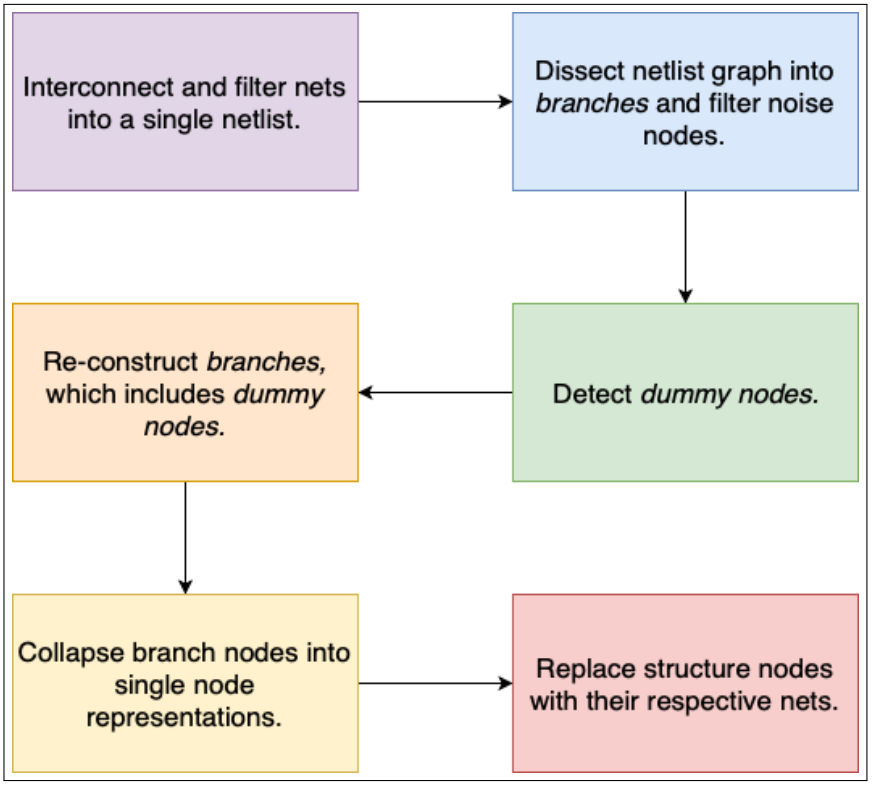


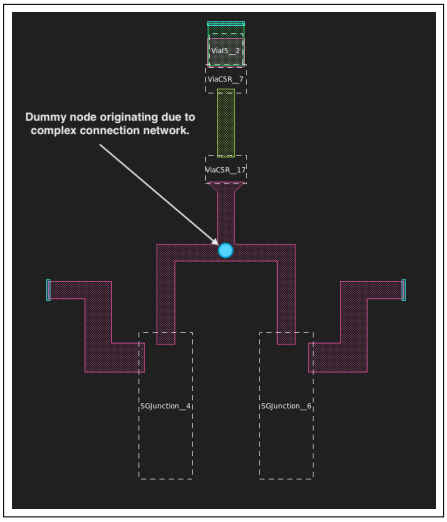
Figure 7.12: Netlist filtering flow for circuit layouts, following from figure Fig. 7.1.

However, because of the complex interconnections found between devices in a circuit, extra graph algorithms have to be applied. These algorithms include categorizing all metal nodes in the circuit net into different branches, filtering noise nodes, and automatically adding dummy nodes.

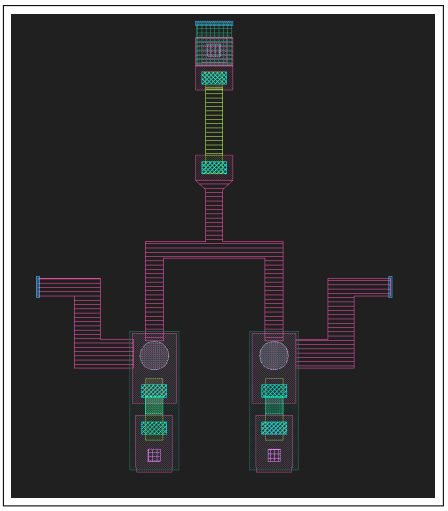
7.4.1 Circuit Geometry Simplification

Using a hierarchical netlist extraction algorithm requires the circuit geometry to take into account the placement of different devices, before generating a mesh for each metal layer. A new type of cell is introduced, called a `Block`, that consists of bounding box polygons of the device cell. The number of polygons in a block cell are equivalent to the number of different metals present in the device. A copy of the circuit cell is made, with its device cells swapped out for block cells.

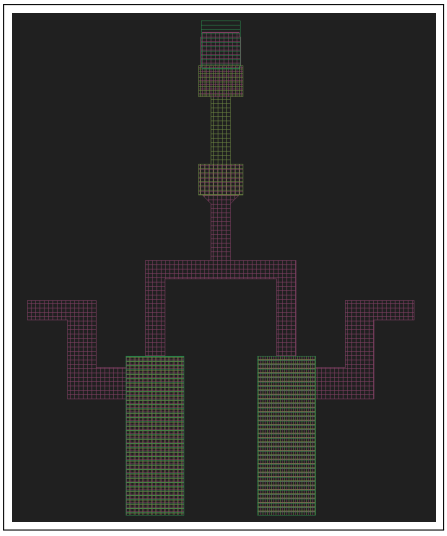
Fig. 7.13b illustrates a basic JTL containing two junction devices, and three via devices. Fig. 7.13c shows the result of swapping each device with a block cell. From this cell all metal layers of the same type are merged, as shown in Fig. 7.13d. This merged metal polygon is then converted to a physical geometry, from which a two-dimensional mesh is generated.



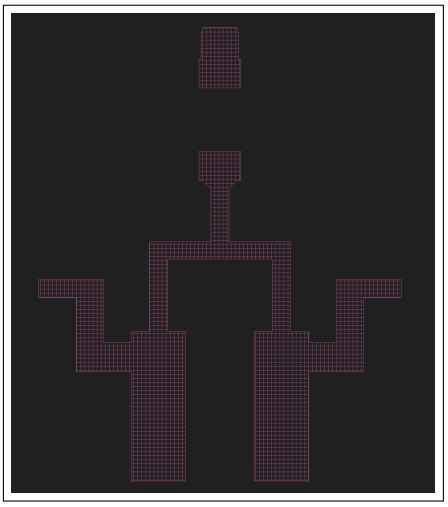
(a) JTL circuit with device cells shown.



(b) Flattened layout of JTL circuit after device detection.



(c) Device cells swapped out with block cells.

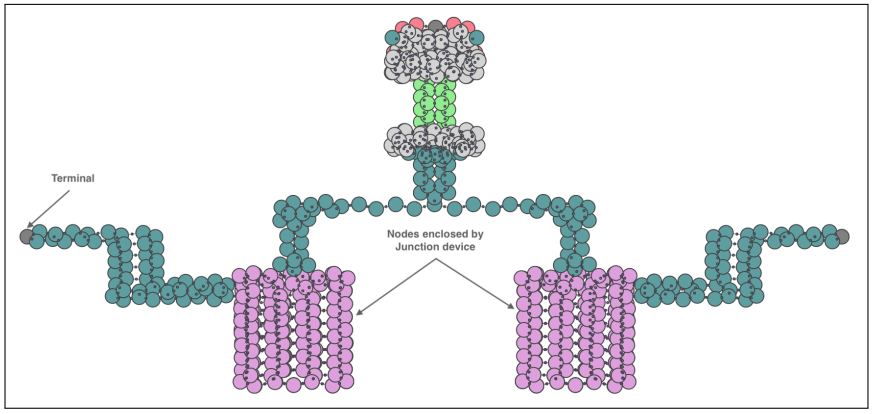


(d) Metal layer, M_6, merged using blocks as depicted in Fig 7.13c.

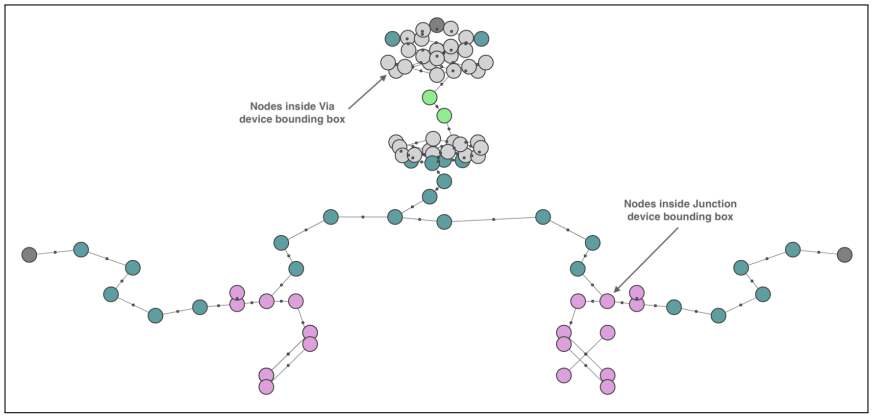
Figure 7.13: Example of merging the JTL circuit metal layer, M_6, using blocks to represent device positions.

7.4.2 Changing Mesh Size

The proposed mesh-to-graph methodology causes scaling problems when working with larger structures. Devices contain multiple contact layers, which requires generating a mesh that tessellates the metal layers into small triangles to accurately describe contact interconnections. However, generating a mesh for a circuit, using the equivalent algorithm, results in an over sophisticated graph representation, shown in Fig. 7.14a.



(a) Generating a mesh for a circuit using the same algorithm as that for devices, results in a complex graph network.



(b) Increasing the mesh element size, results in a simpler graph network

Figure 7.14: Extracting a circuit net requires the use of larger mesh elements.

A solution is found, shown in Fig. 7.14b, by adjusting the mesh size depending on the hierarchical level of the subject cell. This method is based on the presupposition that the higher the hierarchical level, the less spatially confined the underpinning subcells. The nodes belonging to a device is labeled (shown in pink in Fig. 7.14) by shadowing the bounding box of the device cell onto the extracted circuit net.

7.4.3 Net Simplification

After having applied the *disjoint graph algorithm* to construct a single netlist, the resultant graph consists of redundant vertices that have to be filtered. Fig. 7.15 depicts such vertices that emanates from asymmetric mesh structures, which was extracted from the JTL example in Fig. 7.14b. Obtaining an accurate layout netlist that equates to a schematic netlist requires getting rid of unwanted graph vertices. By delineating the net through filtering algorithms, the final result can be representative of the designed netlist schematic.

7.4.3.1 Branch-Detection Algorithm

The biggest problem in extracting a netlist for SDE circuits is to extract the layout into a model that can define individual inductive branches. The raw net shown in Fig. 7.15 does not contain information about different branches, since all the nodes generated from the same polygon object shares the same unique ID.

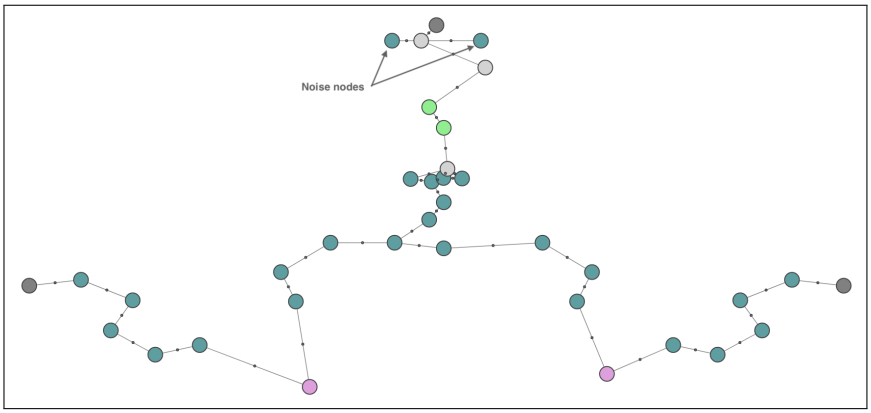


Figure 7.15: JTL circuit net containing noise nodes.

Conducting branches can be detected from inspecting paths between different *branch nodes*. Branch nodes represent layout structures and ports, such as terminals, vias and junctions. A conducting branch is defined as follow: *A graph path between two branch nodes, which only comprises metal layer nodes. None of the nodes in this detected path are allowed to be shared between other branches.*

A *branch-detection* algorithm is used to take the raw extracted graph from a circuit layout, as shown in Fig. 7.15, and update it to contain branch information, as presented in Fig. 7.16. This information is added to the `Net` object from Section 7.2 as a parameter. The updated net, consists of nodes containing information about the specific branches to which they belong.

After having updated the net to contain nodes divided into separate branches, nodes that are not part of any *branch*, gets filtered as *noise nodes*. After applying the branch detection algorithm for the net, shown in Fig. 7.15, the resultant net equates to that shown in Fig. 7.16.

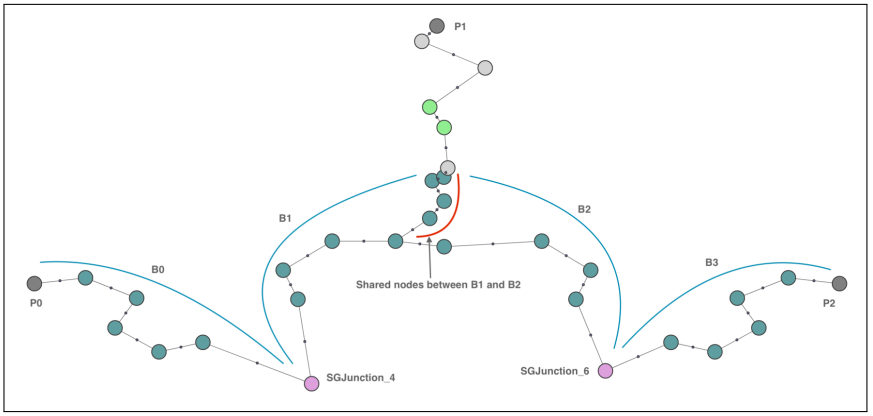


Figure 7.16: JTL circuit net after defining branch nodes, and filtering noise nodes.

7.4.3.2 Defining Dummy Nodes

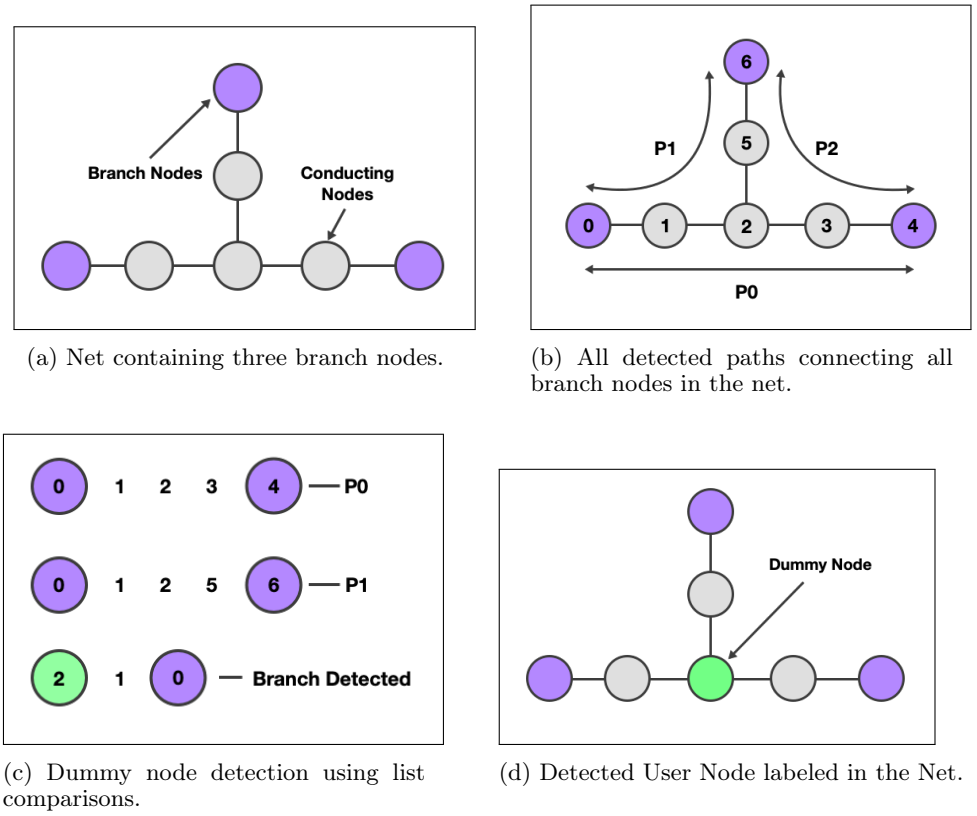
The second part of the branch definition, states that nodes are not allowed to be shared between different branches. However, from Fig. 7.16 nodes are shared between branches B_1 and B_2. Solving this problem, fundamentally solves the problem of detecting dummy nodes. The following definition of a dummy node is proposed: *A dummy node is defined as a crossing between different inductive branches. Therefore, they can be detected by identifying overlapping branches.* Following inductive reasoning, the algorithmic flow of automatically detecting dummy nodes is shown in Fig. 7.17:

- First, the netlist is updated to contain branches. This means, each *conducting node* has a set of unique branch IDs.
- Second, an intersection algorithm is applied to all branches connected to a single *branch node*.
- Third, since the node order are kept unchanged, the last node in the resultant intersection algorithm, is considered to be a *dummy node*.

From Fig. 7.17b, the branches connected to node, 0, is P_0 and P_1. The nodes present in each of these branches are listed in Fig. 7.17c. An intersection algorithm is applied to these branches, if a resultant branch is found, the last node is considered to be a crossover node between the listed branches, and is labeled as a dummy node.

This algorithm emerges from the fact that, technically, nodes in the generated graph cannot be shared between different branches. Nodes can only belong to a single branch. Fig. 7.16 illustrates the shared nodes between branches, B_1 and B_2. From this example, these shared nodes will be the result produced by the intersection algorithm, and the detected dummy node is labeled, as presented in Fig. 7.18.

Once a dummy node is found, the corresponding node is updated, and the branch-detection algorithm is executed again to take into account this dummy node. The result is a graph netlist that consist of nodes that are; either labeled as a *structure node* or a *branch node*. Fig. 7.18 shows the detection of a dummy node for the JTL example, and the updated branches.



(a) Net containing three branch nodes.

(b) All detected paths connecting all branch nodes in the net.

(c) Dummy node detection using list comparisons.

(d) Detected User Node labeled in the Net.

Figure 7.17: Basic steps for detecting a dummy node. Purples nodes represents detected branch nodes.

Each metal layer has a unique label which corresponds to the branch they are in. Having detected branch nodes, the graph can be simplified by grouping all conducting path nodes in the same branch, into a single node representation, while preserving the edge connections, see Fig 7.19.

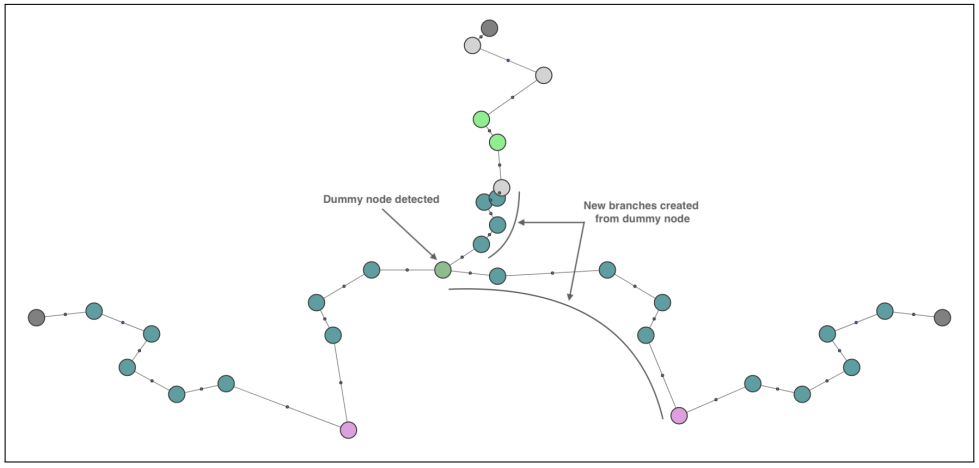


Figure 7.18: Branch reconstruction after dummy node detection.

Fig. 7.19 represents the grouped nodes that connect branch nodes after filtering. The resultant graph now represents lumped elements, such as inductors, with graph edges.

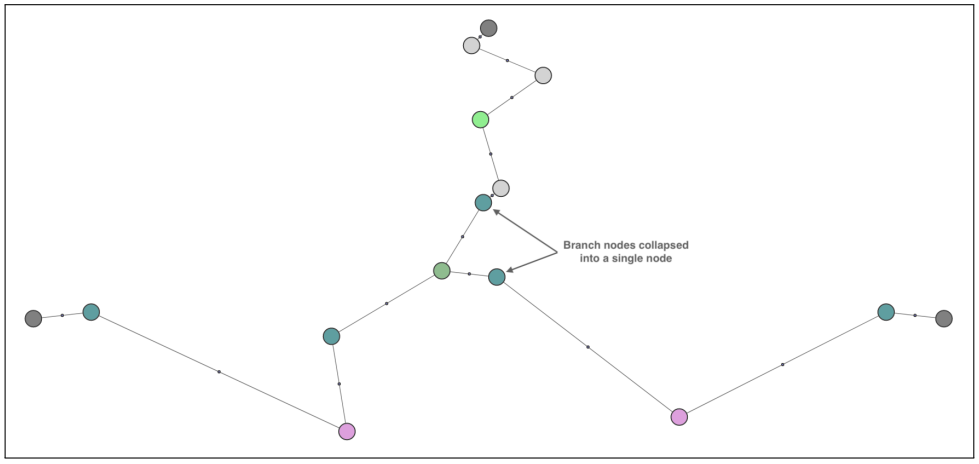


Figure 7.19: Branch nodes grouped into a single node.

Once the circuit has been fully constructed, dissected into branches, noise nodes filters, and dummy nodes detected, each device node—which is a reference to the device cell containing its representative net—can be replaced by its extracted net, as shown in Fig. 7.20.

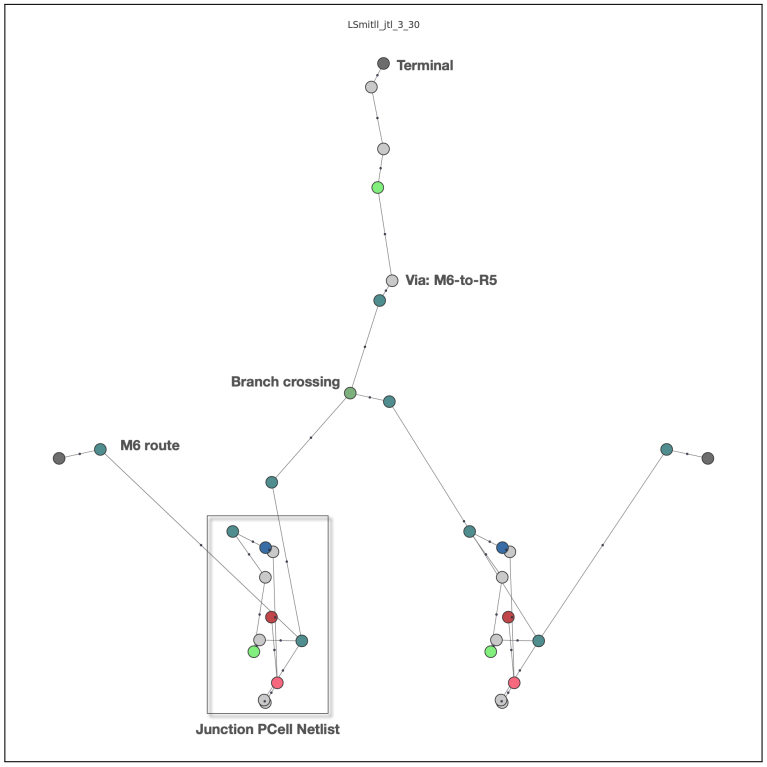


Figure 7.20: Device nodes replaced with their respective device nets.

7.5 Conclusion

The netlist extraction module was developed as a package, building from the SPiRA framework, using a hierarchical parameterized approach for full netlist extraction at the gate level. First, a netlist is constructed for each device reference in the circuit, which includes applying all the necessary net simplification algorithms. Second, a netlist is constructed for the circuit, while representing each device instance as a single node. Third, the device nodes in the circuit netlist can be swapped out with their complete extracted netlists, after having simplified the circuit net. This hierarchical method aids in understanding the constituents of more complex layouts and improves debugging and extending the proposed module.

Implicit in this module code is the parameterized design patterns, such as data connection routines and elemental construction, allowing abstracted hierarchical solutions. For example, in developing a netlist extraction solution it was concluded that the SPiRA framework has to be extended to easily categorise different layer purposes. This leads to adding purpose properties in the RDD, see Chapter 2, and introducing process layers required for device detection, discussed in Chapter 6.

This chapter shows how following a bottom-up hierarchical approach to extracting a layout netlist does not just solve the problem of layout netlist

extraction, but elicits adding artifacts unique to each module that compounds through the hierarchical codebase of the framework.

7.6 Future Work

Once a netlist has been extracted from either a device or circuit layout, it can be compared to the SPICE schematic using graph isomorphism [36], [37]. The netlist extraction module can be extended to show wiring disconnects, by directly referring to the generated netlist, rather than showing highlighted connects as explained in Chapter. 6. The package is still unstable in automatically extracting a netlist for layouts that consists of nested circuit cells, however the same methodologies covered in this chapter can be used to extend this module. Similar methods can be used for extracting a netlist for flattened cells, but has not been fully tested and implemented.

Chapter 8

Extraction Results

This chapter discusses a set of RSFQ circuit layout extractions preformed using the SPiRA framework. Each section contains a short description of the extracted circuit, along with figures produced by the framework for validation. The current version of SPiRA still requires a designer to manually check the correctness of the generated graph. However, since netlist extraction is one of the last steps in PDV these generated graphs validates the following aspects of the framework:

- *Valid PCell instances*: A device PCell is used as a template for device detection. If the generated graph contains the specific device node, that means the PCell-device-mapping was successful. If the device PCell was incorrectly defined this may cause the framework to not correctly detect a device. However, this is also a problem that may occur in traditional semiconductor LVS tools if a device is incorrectly defined in the LVS database.

- *Device Detection*: Similar to the previous aspect, having a specific node in the graph represent that specific device in the layout, proves that the parsed cells was successfully detected. However, it might be the case that a device is detected that should not be detected since there is rule violations in the parsed cell. Checking for this type of negative validation methods are beyond the scope of this framework.

- *Netlist Extraction*: By definition the extracted graph structures, which includes the labeling of inductive branch cross-overs (dummy nodes), port objects, device nodes (or device netlists if device nodes expanded), validates the correctness of the netlist extraction algorithm. Similar, to the device detection aspect, negative validation methods are beyond the scope of this framework. The correctness of the extracted netlist was checked using a basic graph isomorphic algorithm from the NetworkX library C.6, but this check only returns a boolean value. More sophisti-

cated graph matching algorithms, such as a *fuzzy attributed graphs approach* [2] has to be implemented in future versions.

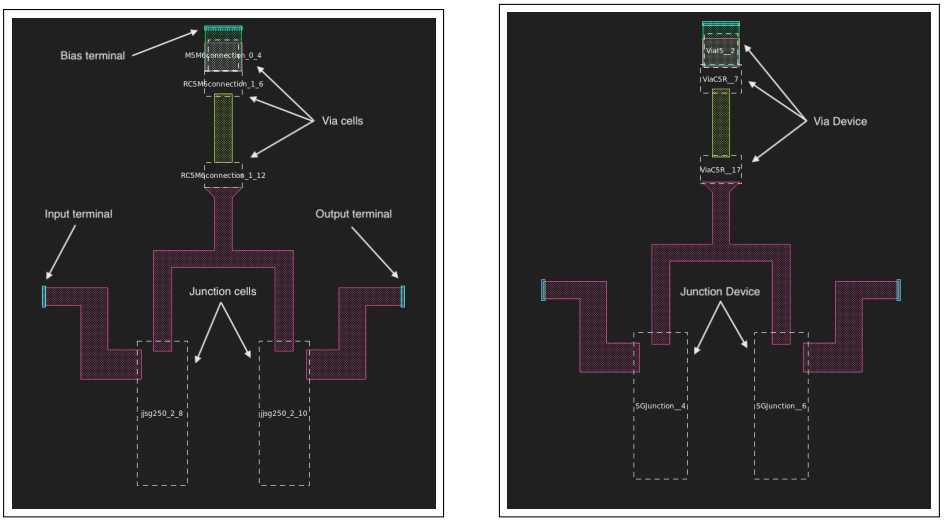
Seven layout extraction tests are discussed:

- **JTL**: Introduces the basic procedures used in debugging the different extraction steps.
- **PTLRX**: Similar to the JTL example, but with an extra junction and the bias input shifted. The JTL and PTLRX test layouts are used to verify that the PCell-device-mapping methodology was correct.
- **JTLT**: This test shows that multiple *dummy nodes* can be detected, given that they are separated by different structures. The JTLT test therefore illustrates the the netlist extraction algorithm can automatically extract dummy nodes for simple layouts.
- **Splitter**: This test shows that multiple *dummy nodes* can be detected without being separated by different structures. The Splitter test therefore illustrates the the netlist extraction algorithm can automatically extract dummy nodes for medium-complex layouts.
- **Merger**: Tests for connections between the different metal layers present in the junction device. The Merger test therefore shows that the electrical rule checking algorithm effectively detects metallic connections.
- **SQFDC**: This example introduces the lack of detecting via connections that are not included as cell reference in the layout.
- **DFF**: This example illustrates an issue with the current electrical connection algorithm.

The extraction times for both device detection and netlist extraction, for each of the layouts presented in this chapter, are tabulated and conclusions are drawn.

8.1 JTL

The Josephson Transmission Line (JTL) is responsible for interconnecting more complex cells over short distances while insulating the cells from each other.

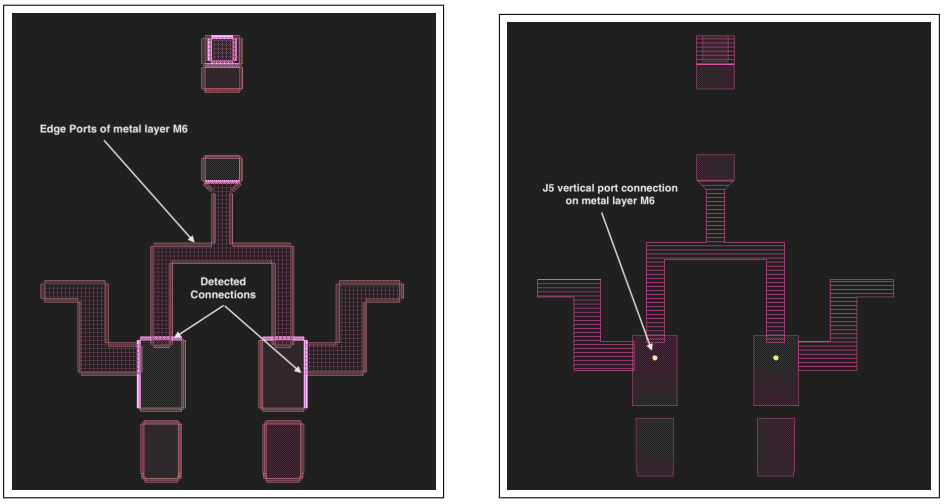


(a) Josephson Transmission Line (JTL) layout.

(b) The parameterized JTL layout.

Figure 8.1: A JTL layout is parsed and parameterized, consisting of device subcells.

Fig. 8.1 shows the original JTL layout with the converted, parameterized layout. This layout is of type `spira.Circuit` and has sub-cells of type `spira.Device`. The Josephson junction cells are equal to the `Junction` class constructed in Section 4.4, but with the elementals swapped out and parameterized.

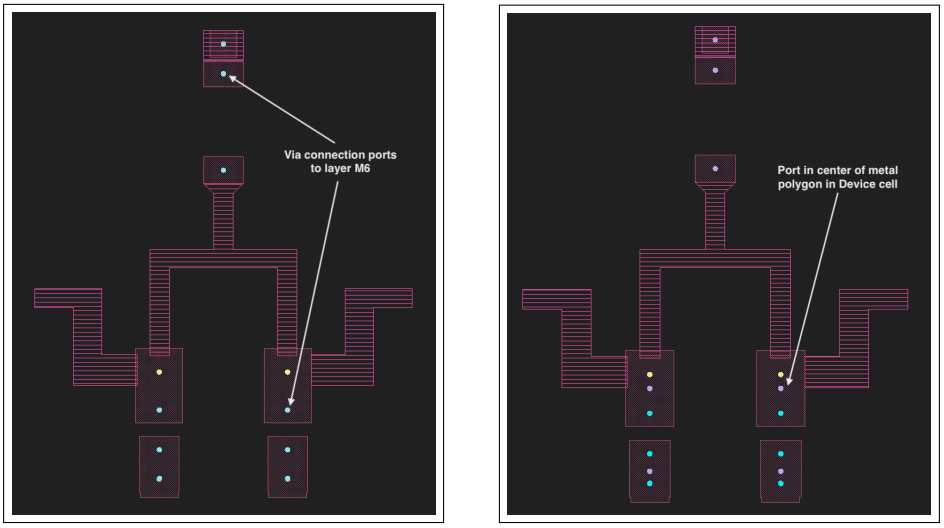


(a) Active edge ports between device and circuit routes are highlighted.

(b) Vertical ports representing the JJ layer connection to metal layer M_6.

Figure 8.2: Vertical and horizontal layer connections detected in the JTL layout.

Fig. 8.2a highlights the device edges that connects to the circuit routes, and Fig. 8.2b shows the ports representative of the detected junction layers, J_5.



(a) Horizontal via connections to metal layer.

(b) Ports representing individual metal polygons.

Figure 8.3: Horizontal via connections and metal ports.

Fig. 8.3a shows the vertical port connections to the M_6 metal layer, which represents via connections. Fig. 8.3b includes the center ports for each detected metal layer in the junction device.

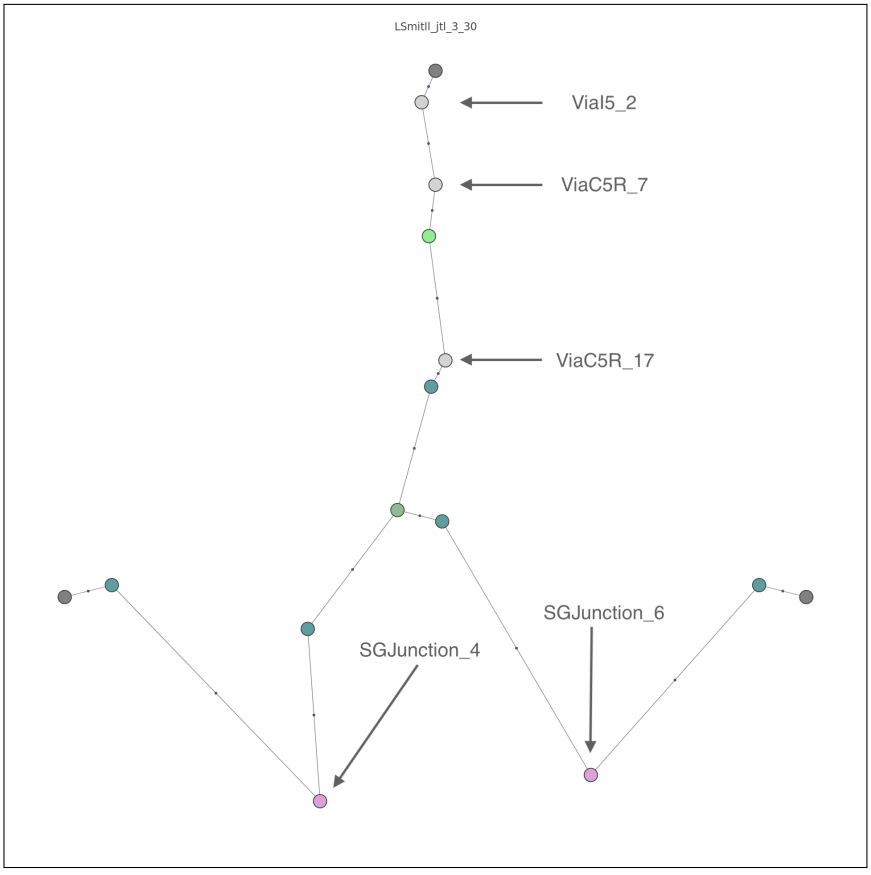


Figure 8.4: Extracted circuit net of JTL in the highest hierarchical level..

Fig. 8.4 shows the extracted netlist with each device represented as a single node in the circuit netlist, and Fig. 8.5 shows the complete netlist including the extracted nets for each device. The bias resistor branch is shown between vias `ViaC5R_7` and `ViaC5R_17`.

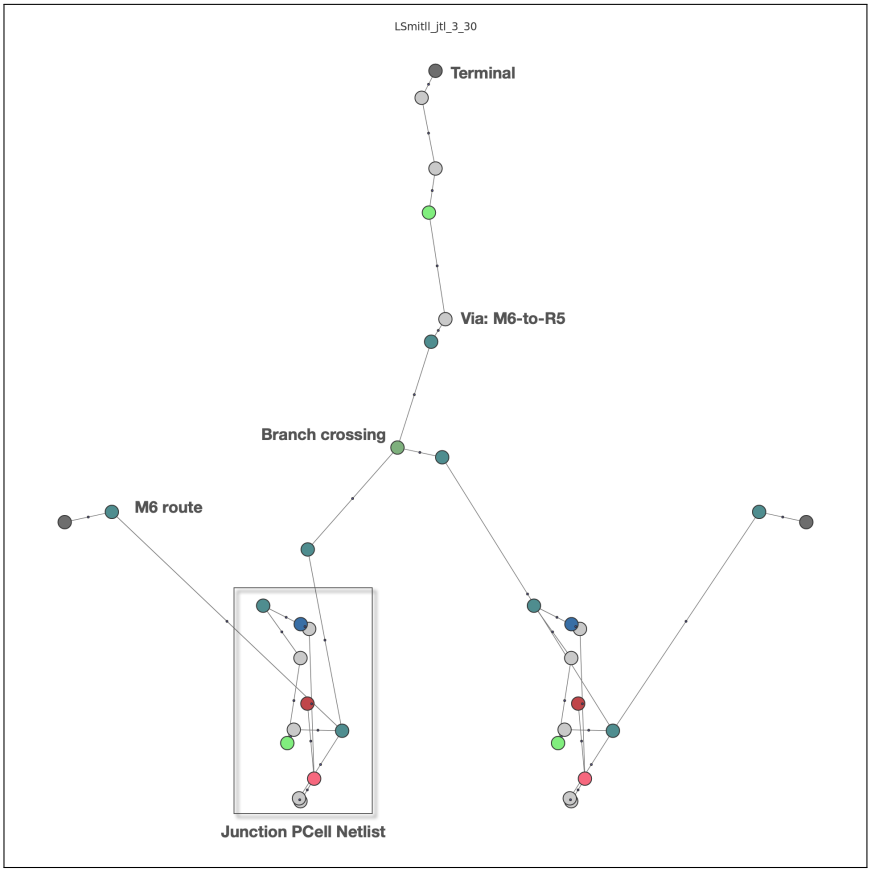


Figure 8.5: Extracted circuit net of JTL in the lowest hierarchical level.

The branch crossing node, that is also the *dummy node*, that defines the bias inductor is detected and labeled. The junction nodes, `SGJunction_4` and `SGJunction_6` are connected to their respective inductor nodes using the active edge ports illustrated in Fig. 8.2a.

8.2 PTLRX

The Passive Transmission Line Receiver (PTLRX) is a circuit that connects to a PTL, which is used to transferring SFQ pulses over long distances between different cells.

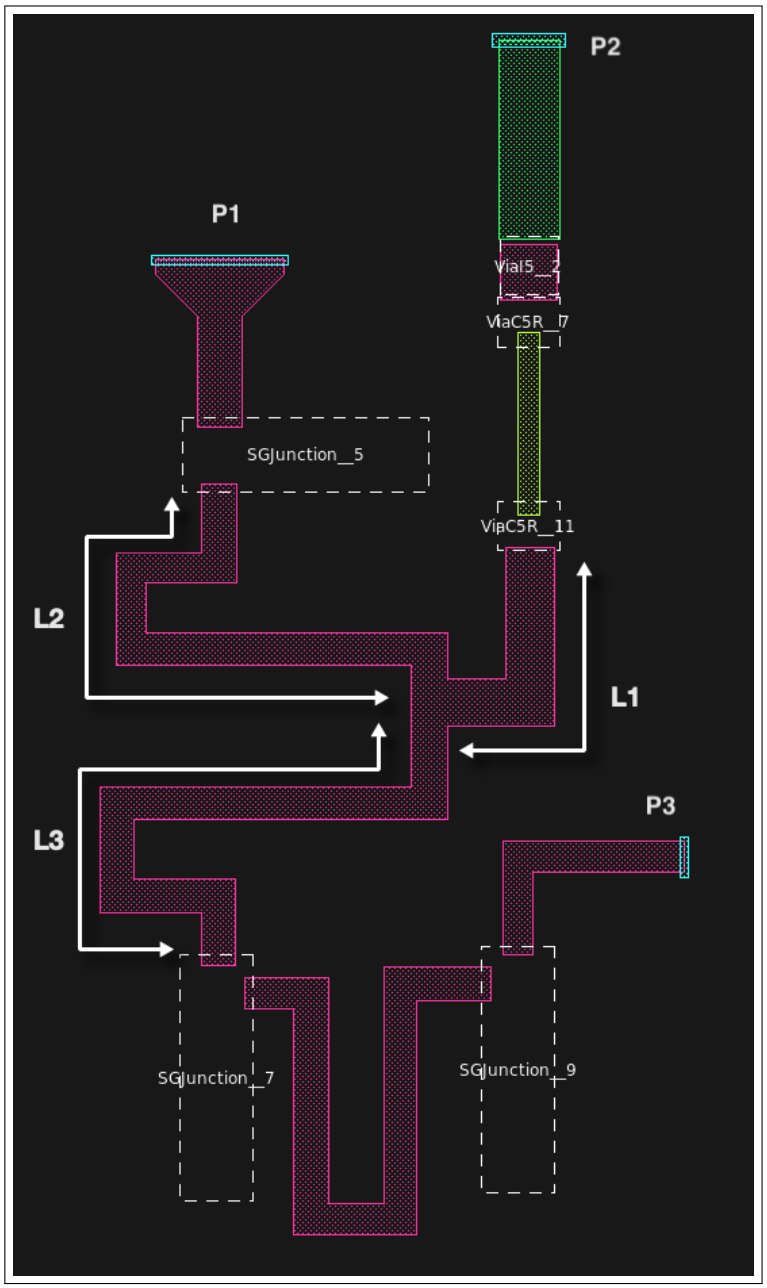


Figure 8.6: PTLRX layout consisting of detected device cells.

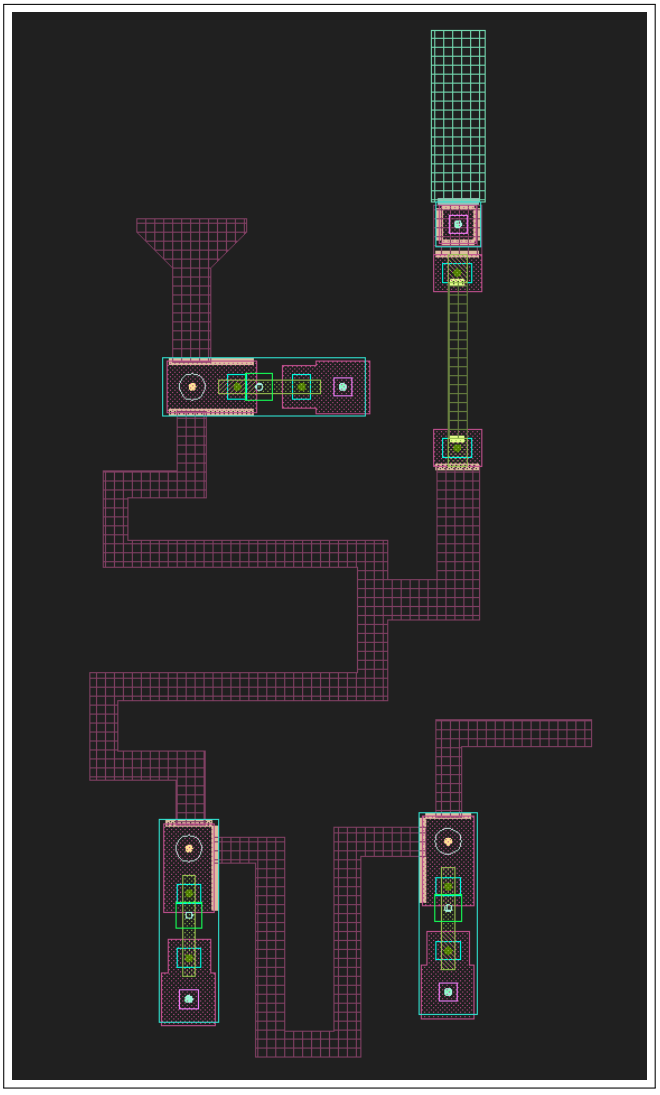


Figure 8.7: Highlighted edges represents connections between different cell layers.

The PTLRX circuit is very similar to the JTL circuit. This examples is used as a basic test case in conjunction with the JTL. This test only shows that the automatic detection of a single *dummy node* is possible using a different circuit topology. The detected dummy node dissects the inductor network into three separate inductor branches, namely L_1, L_2, and L_3, as can be seen in Fig. 8.8 below.

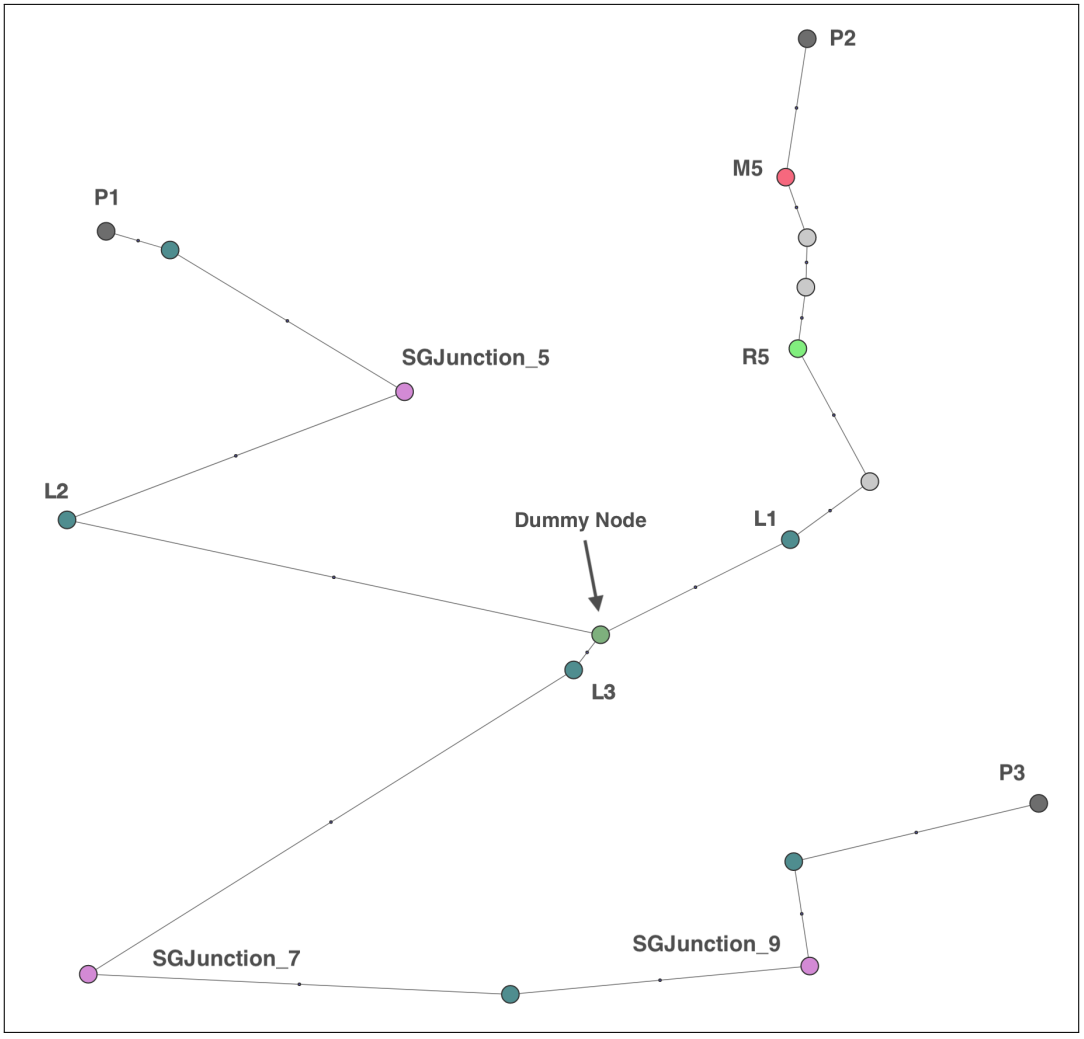


Figure 8.8: Extracted circuit net of PTLRX in the highest hierarchical level.

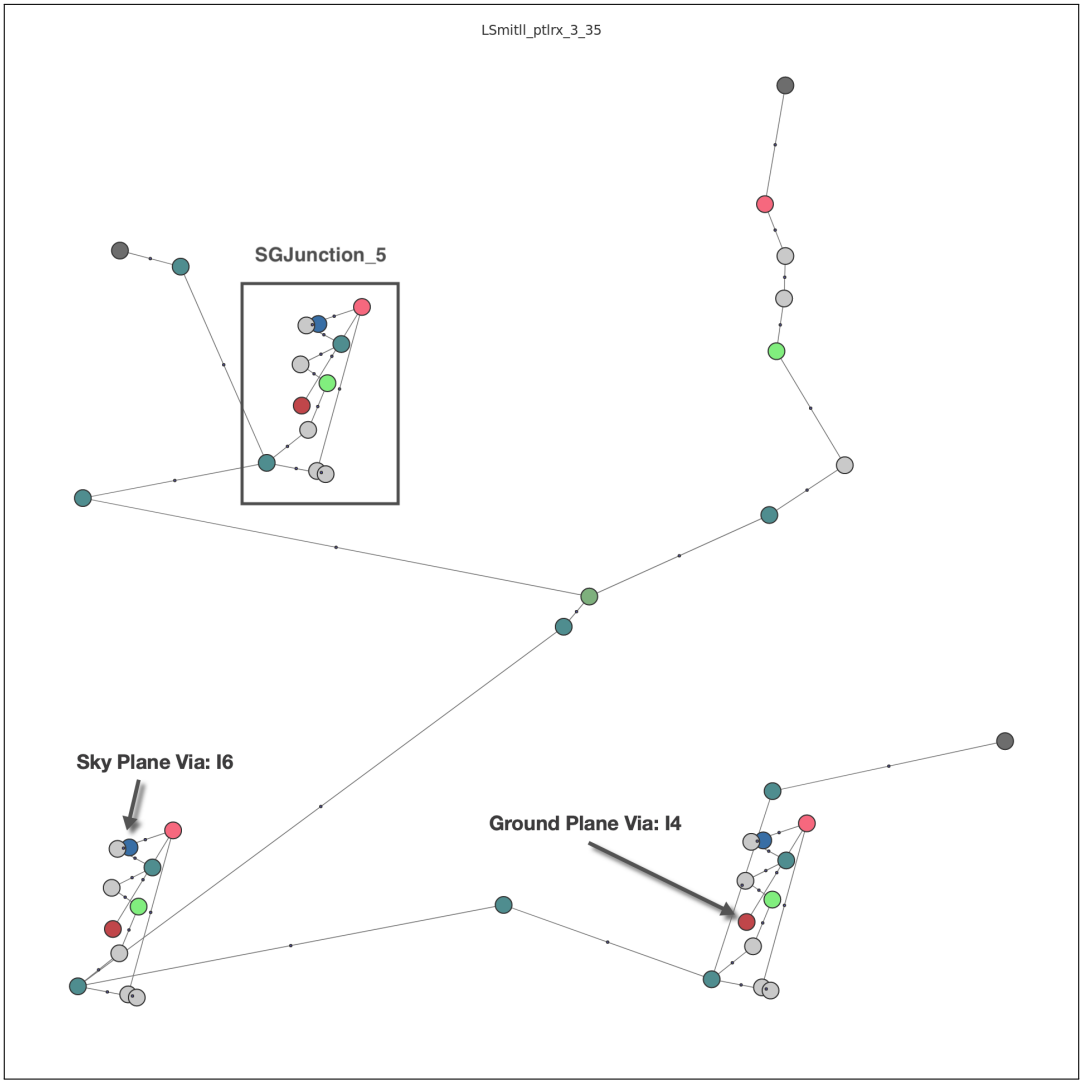


Figure 8.9: Extracted circuit net of PTLRX in the highest lowest level.

Fig. 8.9 illustrates the full netlist, including the junction device nets. Each junction in this circuit connects to both the ground plane and the sky plane, as observed by the respective via nodes.

8.3 JTLT

The JTLT cell is commonly used to re-establish and propagate SFQ pulses when long PTL connections are required.

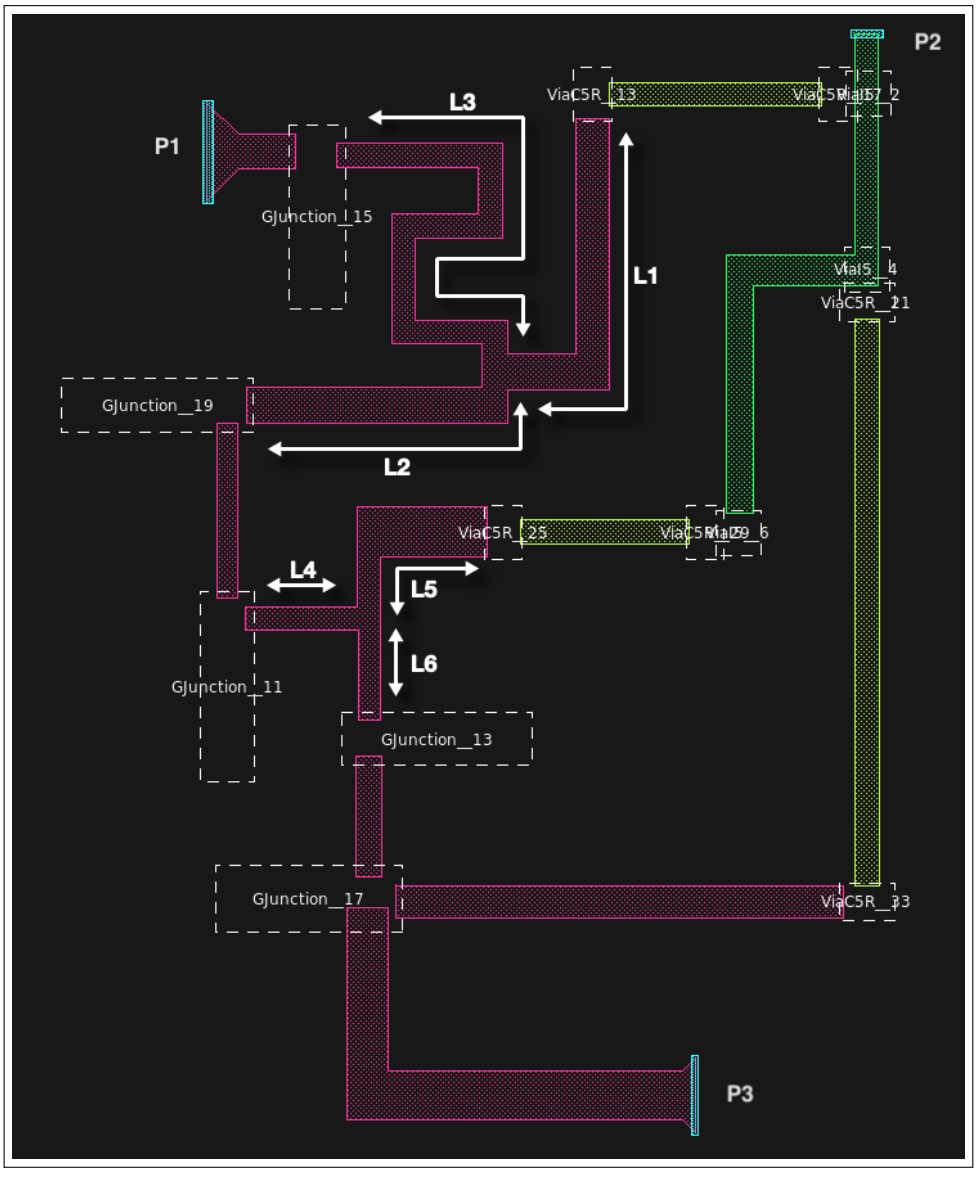


Figure 8.10: JTLT parameterized layout containing detected devices.

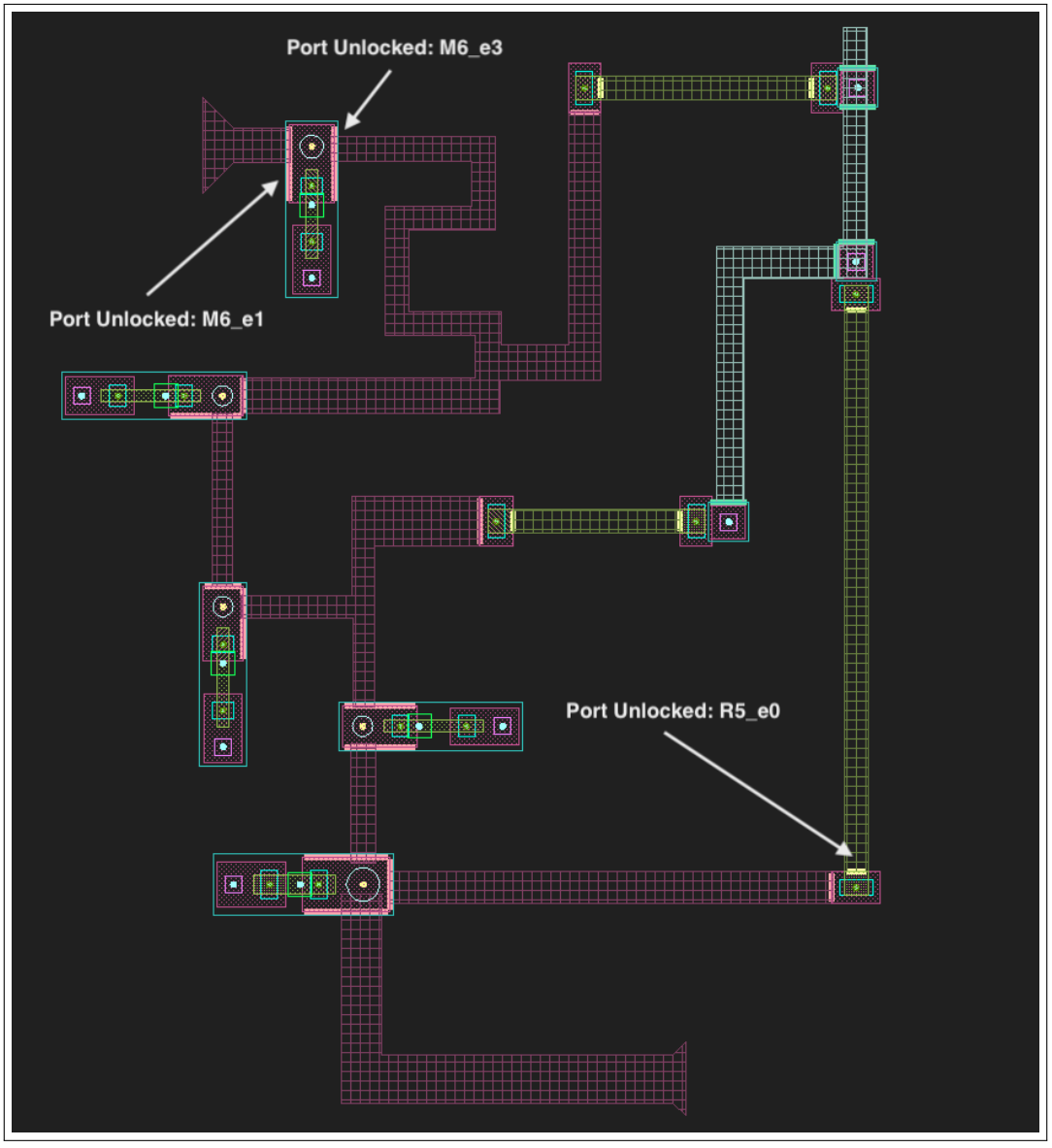


Figure 8.11: JTLT circuit showing connectivity ports.

The JTLT circuit is a more complex layout than the JTL or PTLRX. It contains two *dummy nodes*, five Josephson junctions and three resistive layers. This circuit layouts test for junctions that are only connected to ground (M_4 layer), and that multiple *dummy nodes* can be automatically detected. The resistive layers are also correctly detected, along with their connecting via devices.

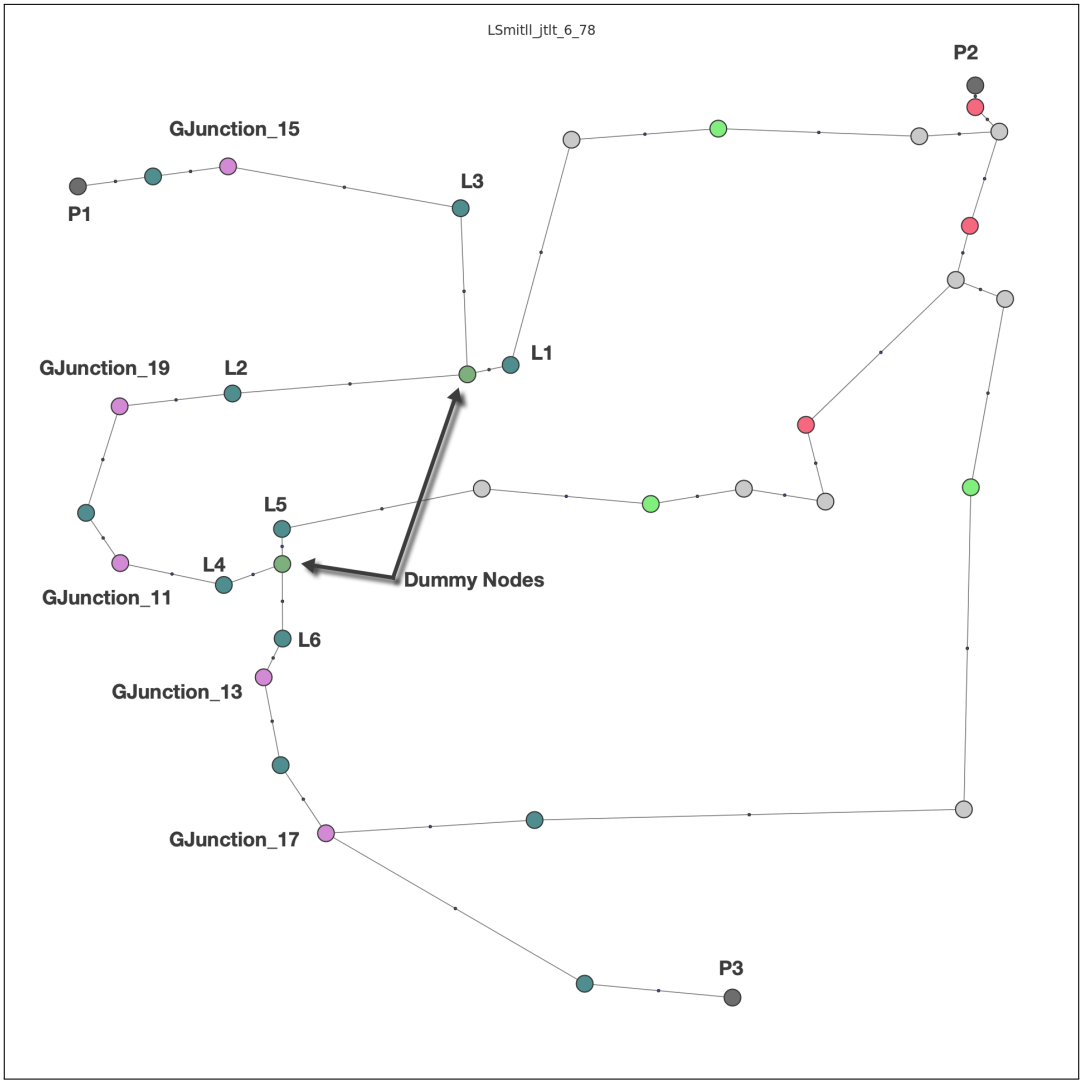


Figure 8.12: Extracted circuit net of JTLT in the highest hierarchical level.

Fig. 8.10 depicts the six interconnected inductor branches (L_1-L_6). Fig. 8.12 shows the representative netlist nodes for each of these inductors, and the detected *dummy nodes* to which they connect.

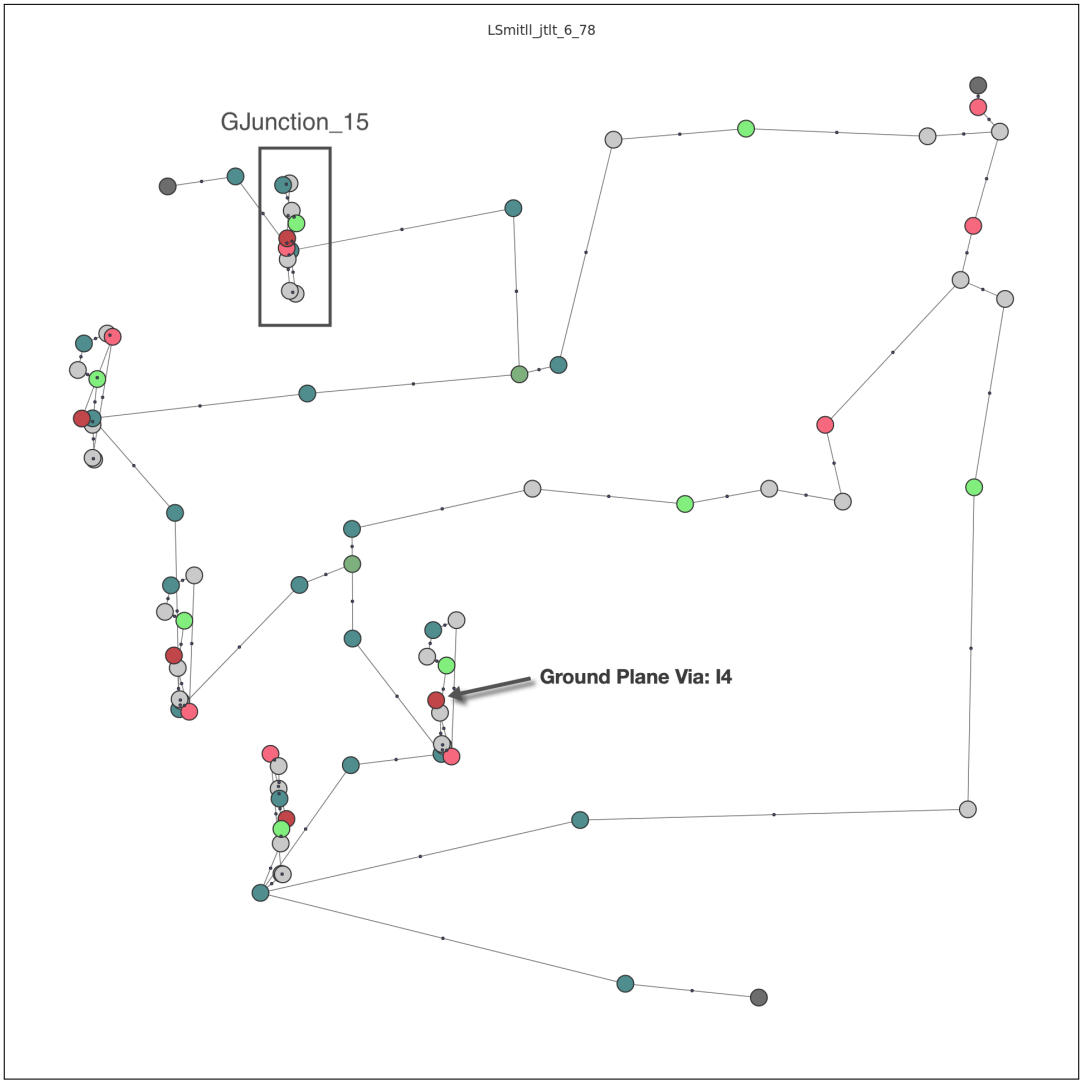


Figure 8.13: Extracted circuit net of JTLT in the highest hierarchical level.

Fig. 8.13 contains the Josephson junction netlist substituted. None of the junctions used in the JTLT layout connects to the sky plane layer (with via I_6), but each junction does connect to the ground plane, through via I_4.

8.4 Splitter

The splitter cell, as the name suggests, is used to divide a single pulse signal line into two pulse signal lines.

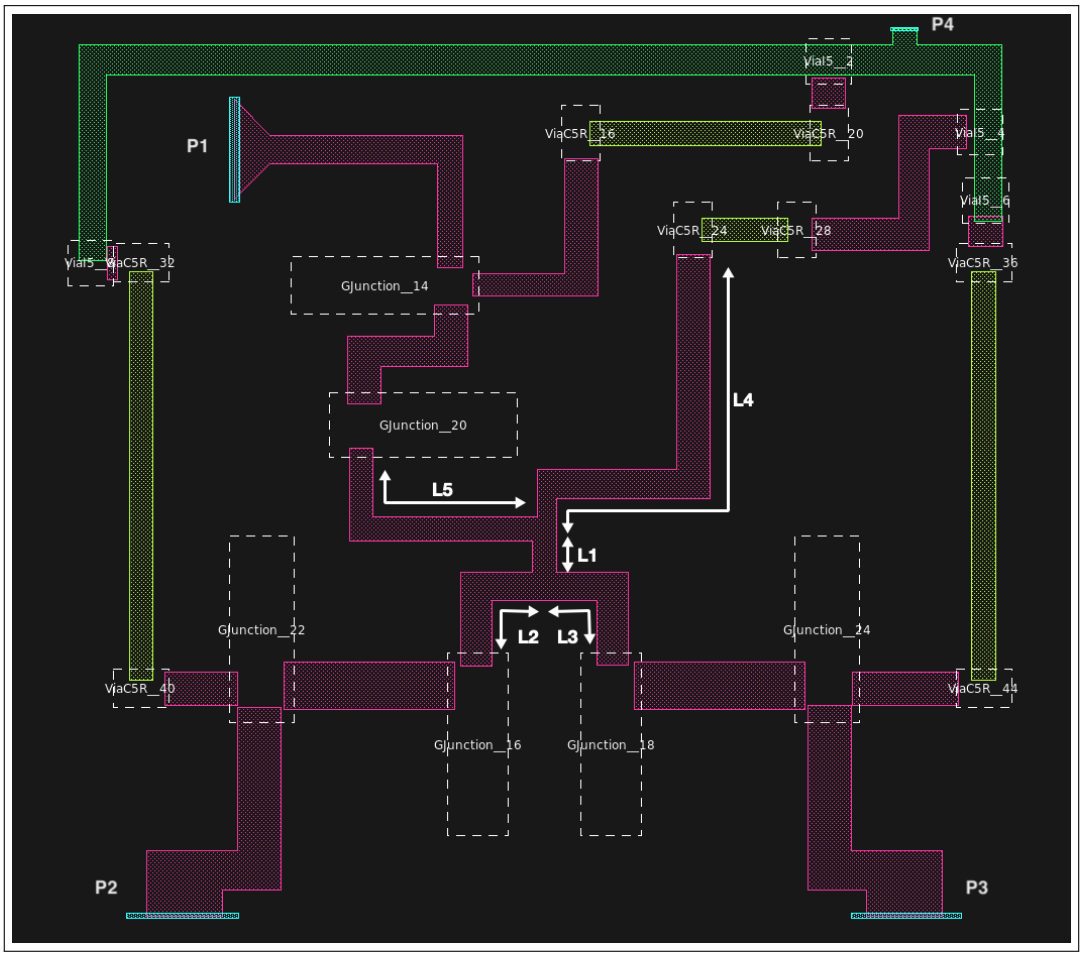


Figure 8.14: Splitter parameterized layout containing detected devices.

The Splitter consists of six grounded junctions, and four resistive layers. The most important part of this layout is the two *dummy nodes* that are directly connected. The dummy node detection algorithm detects paths between different structures. Therefore, it is important to test the accuracy of *dummy node* detection when multiple dummy nodes are directly connected.

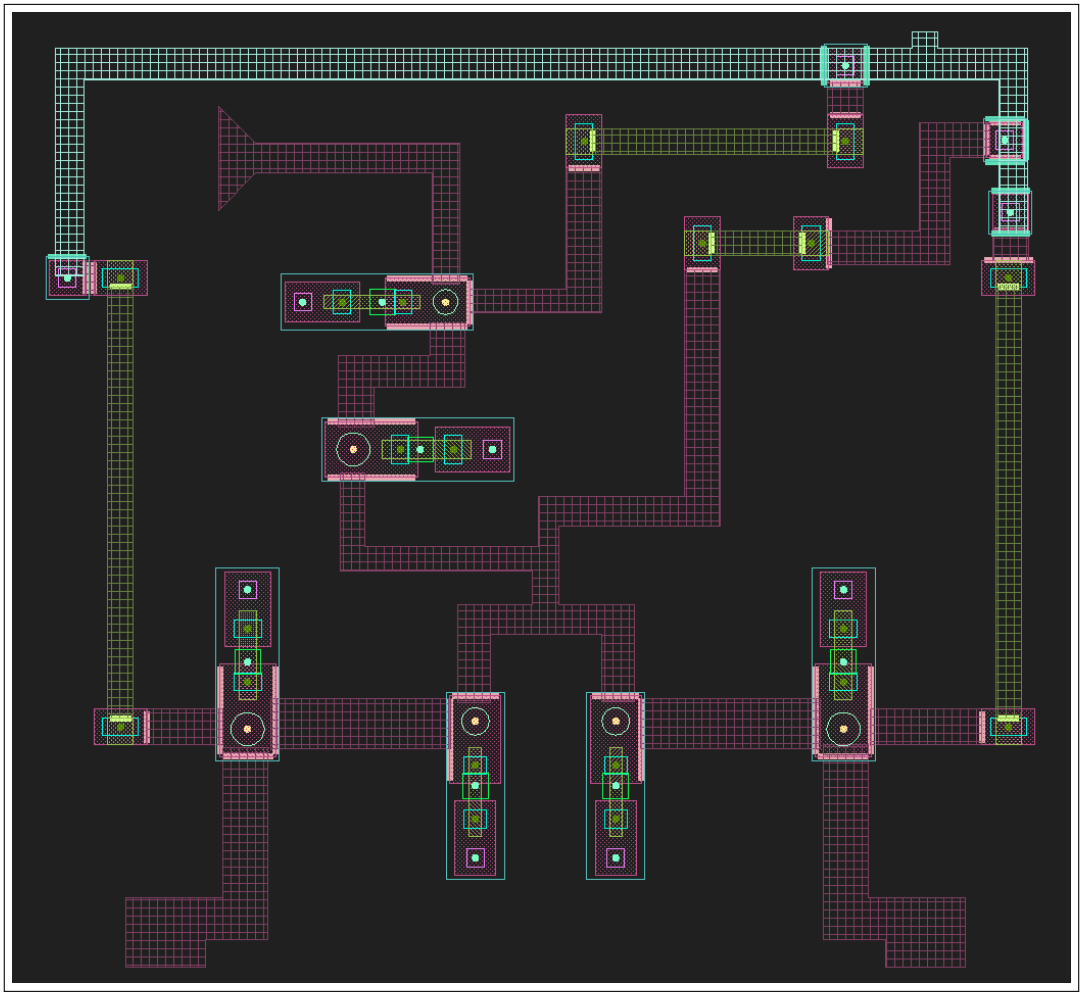


Figure 8.15: Splitter circuit with connected edges.

The layout shown in Fig. 8.15 is used for debugging purposes. It shows all the edge ports that are activated due to detected metal-to-metal connections. It also shows the vertical port connections for each contact layer.

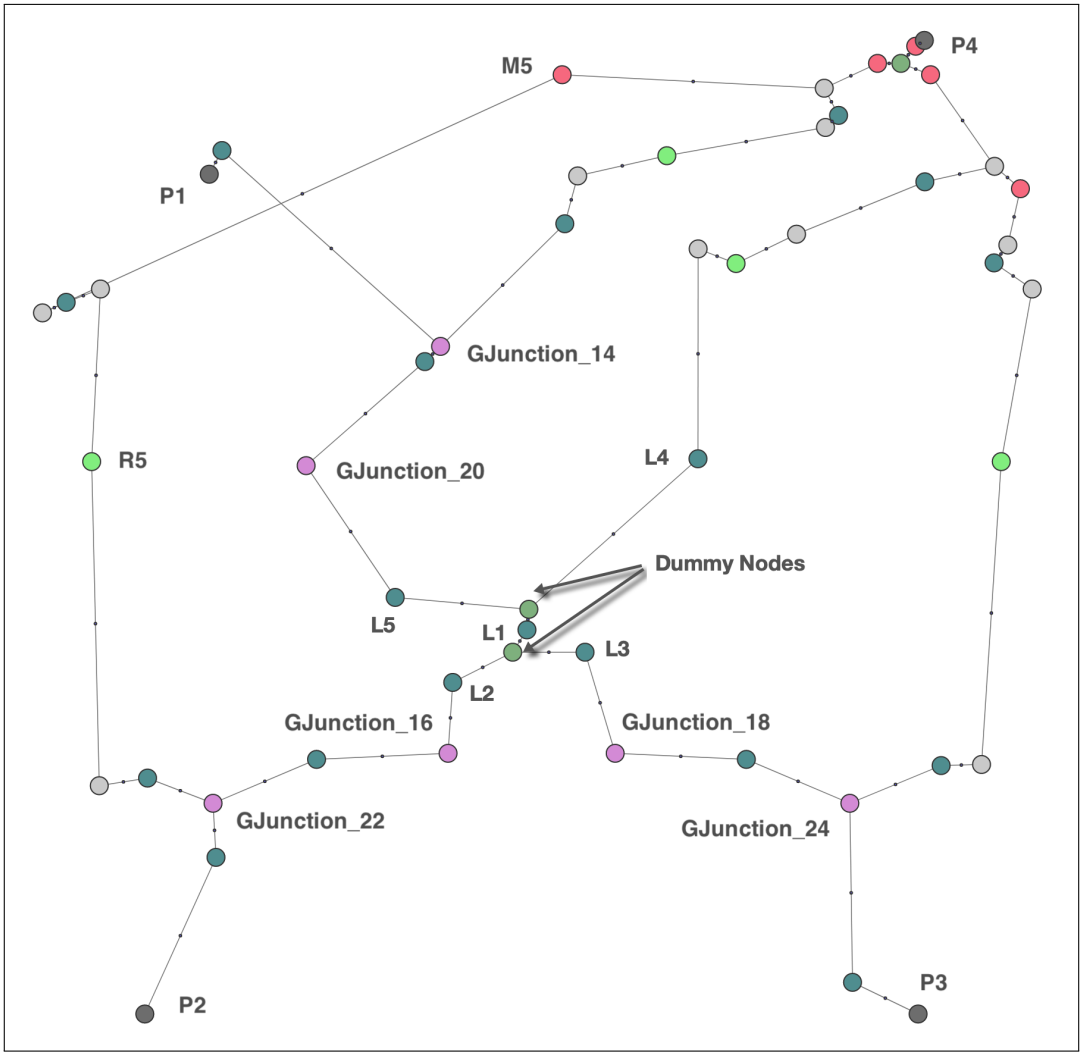


Figure 8.16: Extracted circuit net of Splitter in the highest hierarchical level.

The netlist shown in Fig. 8.16 contains two *dummy nodes* directly connection with an inductor, L_1. The bottom dummy node connects inductive branches L_1, L_2, and L_3. The top dummy node connects inductive branches L_1, L_4, and L_5.

8.5 Merger

The Merger joins two input pulse signal lines and provides a single output pulse signal line. If there is a pulse on either input lines, the merger will generate a pulse on the output signal line.

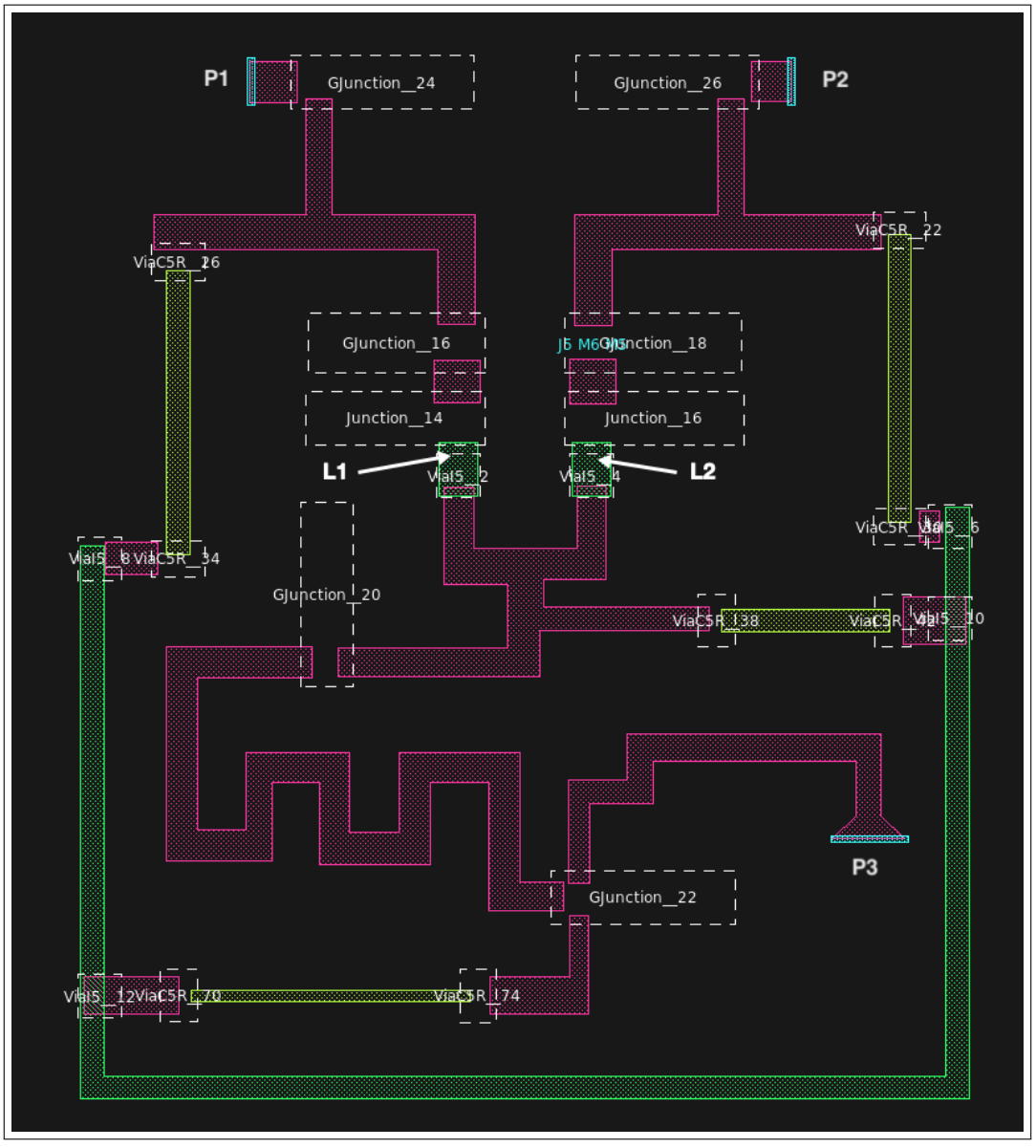


Figure 8.17: Merger parameterized layout containing detected devices.

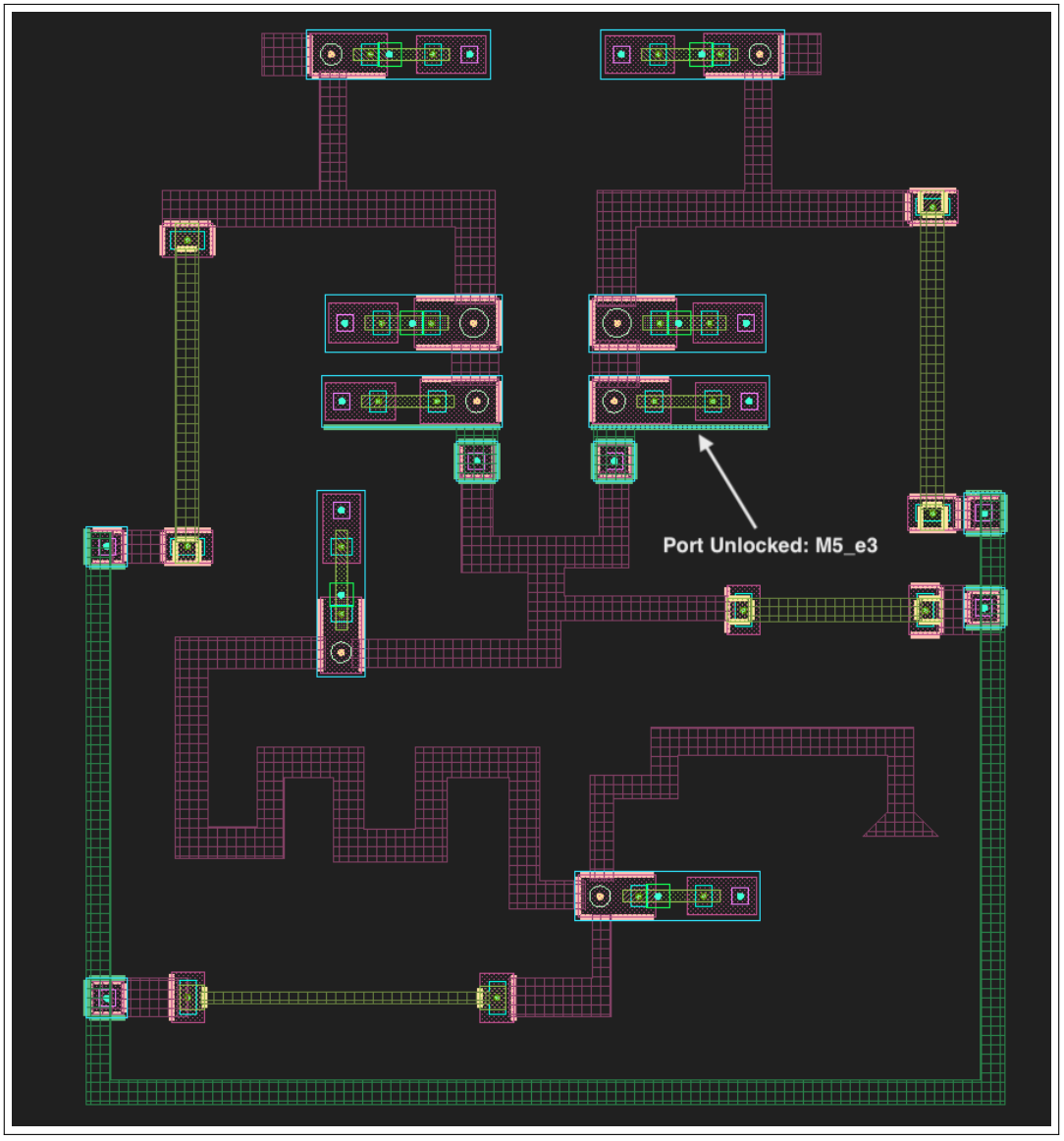


Figure 8.18: Merger showing device connections, with highlighted edges.

This Merger circuit contains two junctions that are neither connected to the ground plane, nor the sky plane (`Junction_14` and `Junction_15`). This layout also tests for junction connects between metal layer M_5. The generated netlist also shows that it is possible to detect *dummy nodes* that are directly connected with an inductor branch.

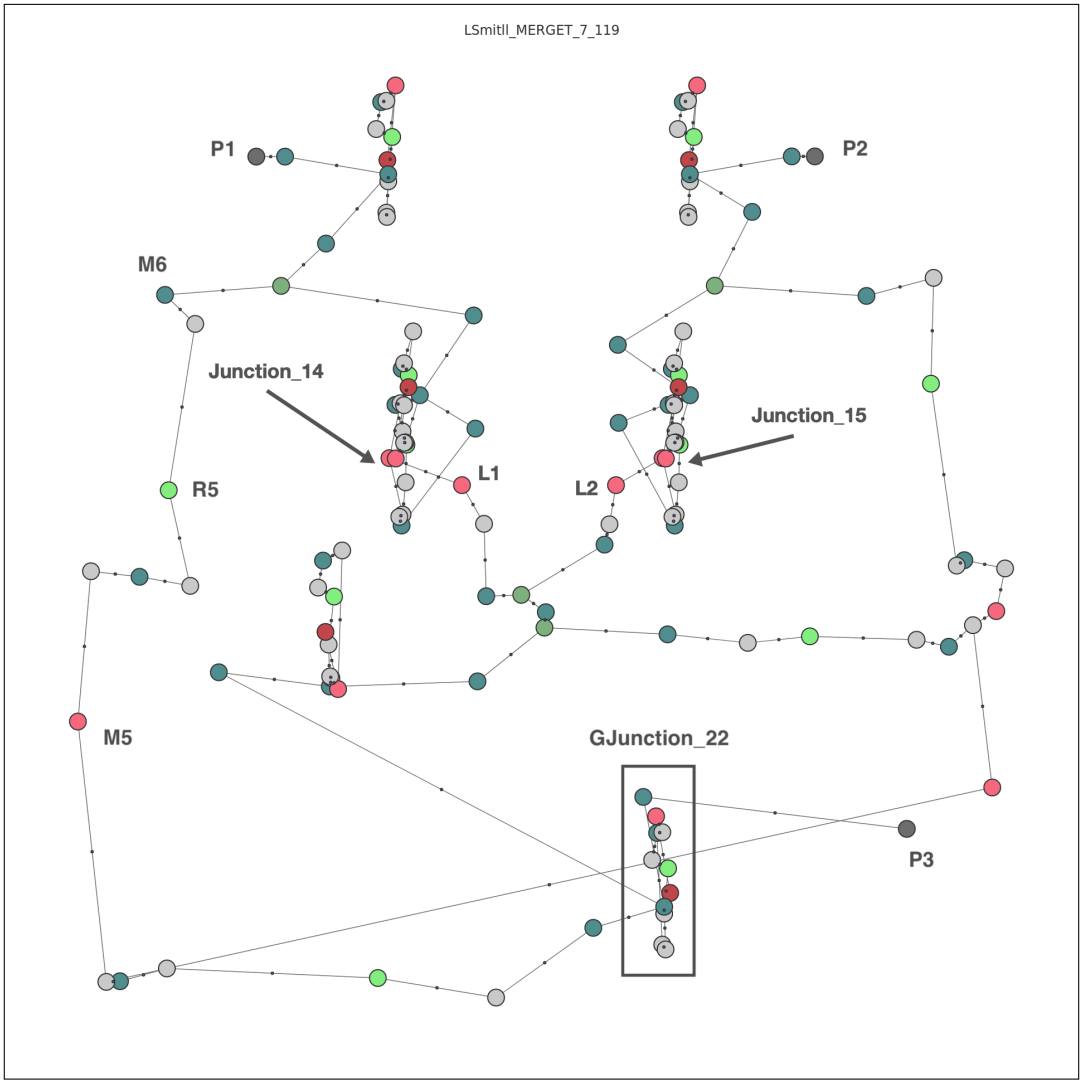


Figure 8.19: Extracted circuit net of Merger in the lowest hierarchical level.

Fig. 8.19 illustrates two junction devices (`Junction_14` and `Junction_15`) that connects to both layers M_6 and M_5, through inductors L_1 and L_2. Also, there are no via nodes I_4 or I_6 in these junction device nets, since these junctions does not connect to the ground plane or sky plane.

8.6 SFQDC

The SFQ-to-DC (SFQDC) converter circuit is used for converting a SFQ pulse to a DC signal. The SFQDC layout that has been tested is shown in Fig. 8.20.

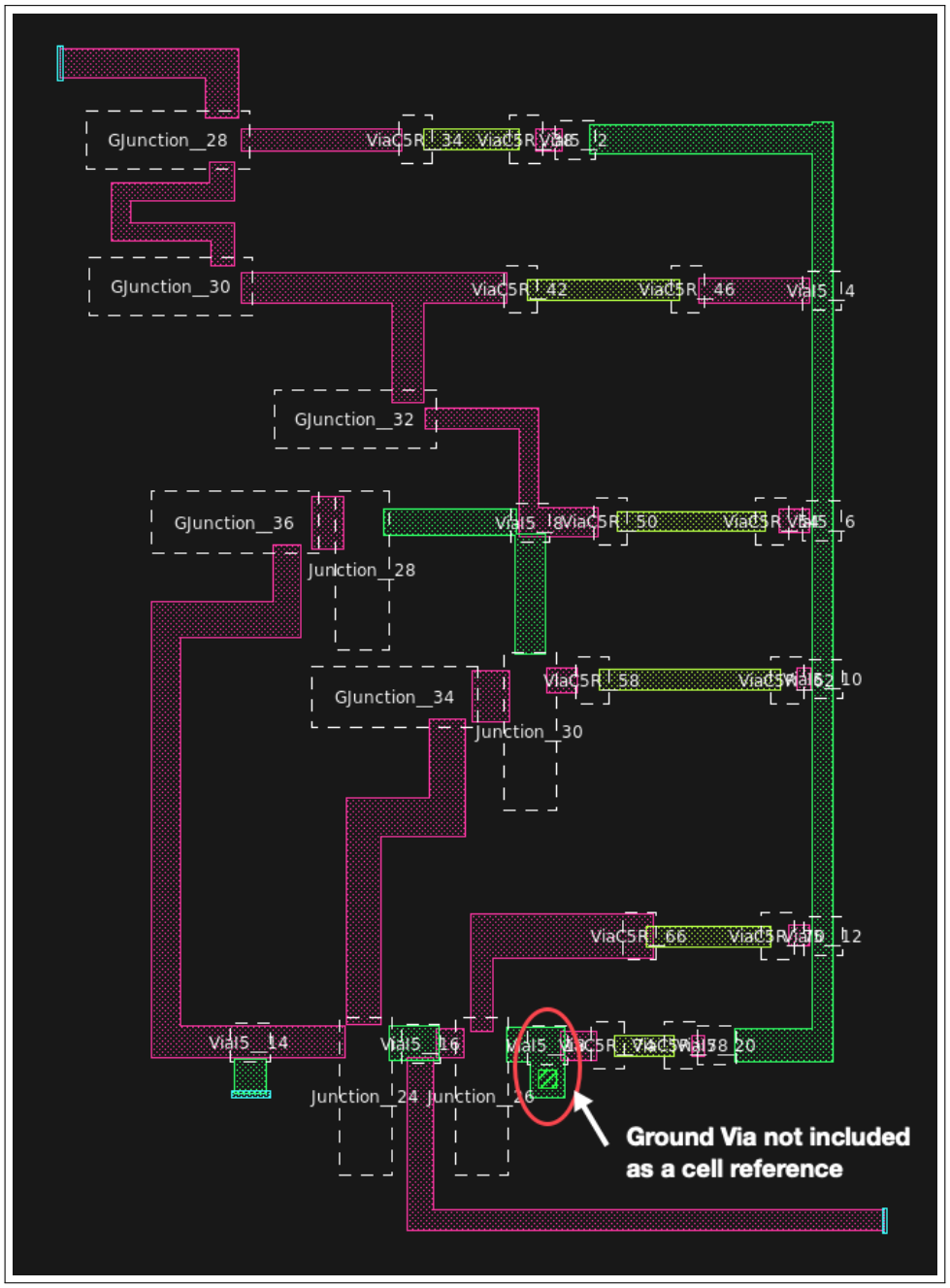


Figure 8.20: SFQ-DC parameterized layout containing detected devices.

The ground via, I_4, that is not detected is circled in this figure. This via is not detected, since it is not included as a cell reference in the layout. This

problem can be solved by future versions once the flattened device detection methods has been implemented.

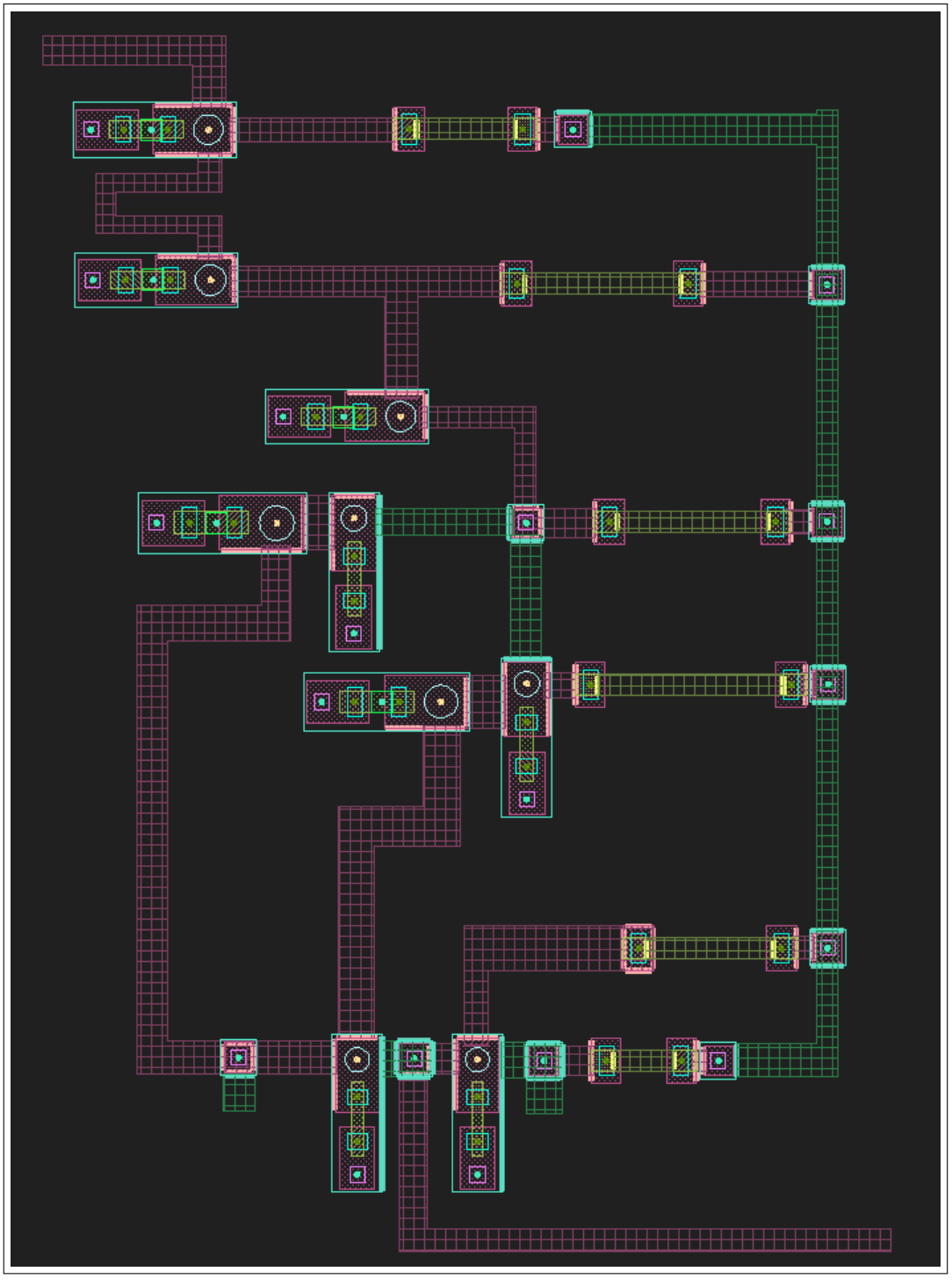


Figure 8.21: SFQ-DC layout showing connection edges.

Fig. 8.21 shows the lack of detecting the edges of the ground via, as well as the lack of adding vertical port connections between the via and the different metal layers.

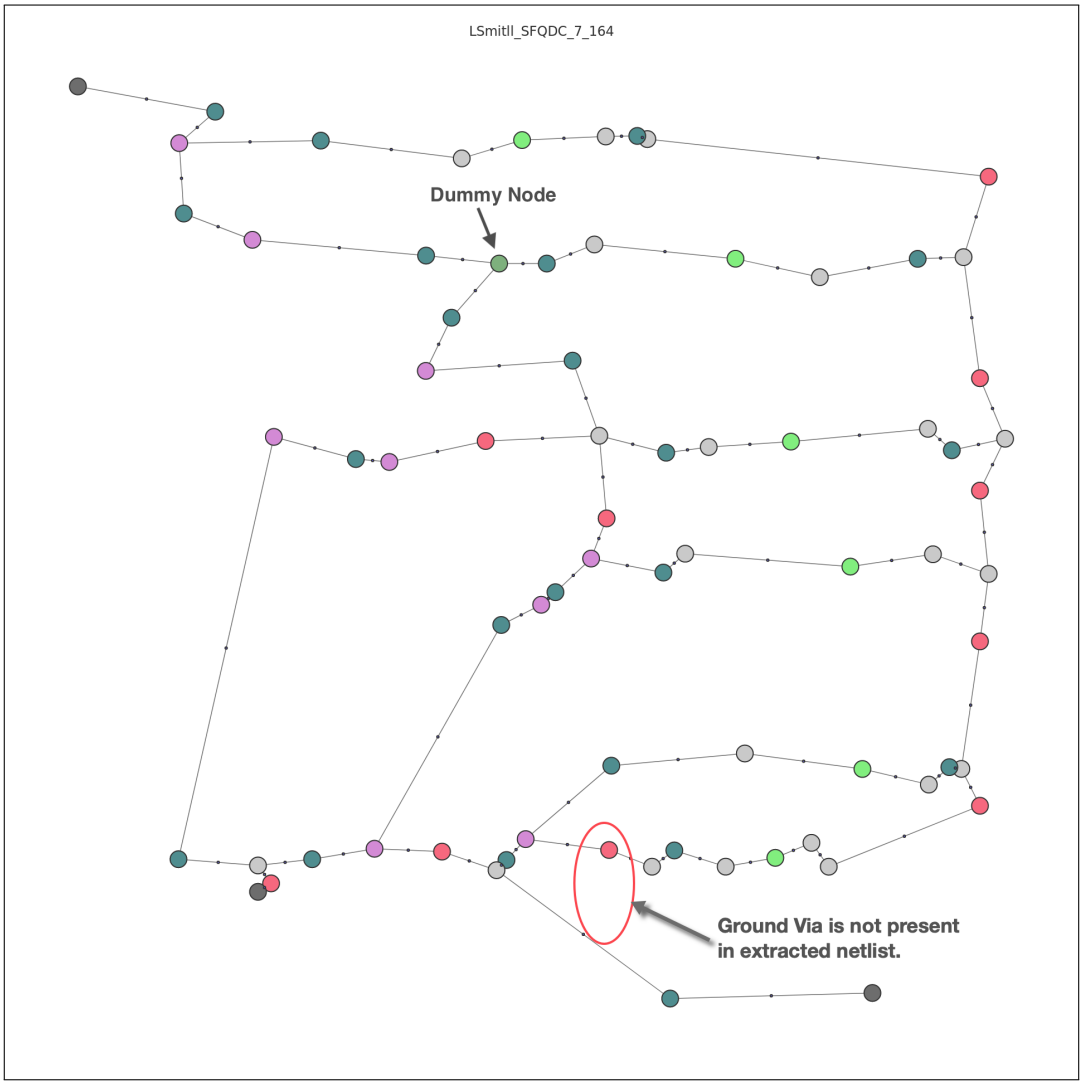


Figure 8.22: Extracted circuit net of SFQ-DC in the highest hierarchical level.

Even though the one ground via is not detected, the rest of the extracted netlist is not effected. The resultant netlist shown in Fig. 8.22 depicts the full layout extraction, without the ground via and the inductor branches directly connected to it.

8.7 DFF

The D-Flip-Flop is a basic memory cell which stores the input pulse and only generates an output pulse after the clock signal. If no input signal was received, before the clock signal, no output will be generated. After having detected all device cells the resultant DFF cell is shown in Fig. 8.23. The vertical ports (contact-metal connections) and horizontal ports (metal-metal connections) are shown in the Fig. 8.24, for each of the detected devices.

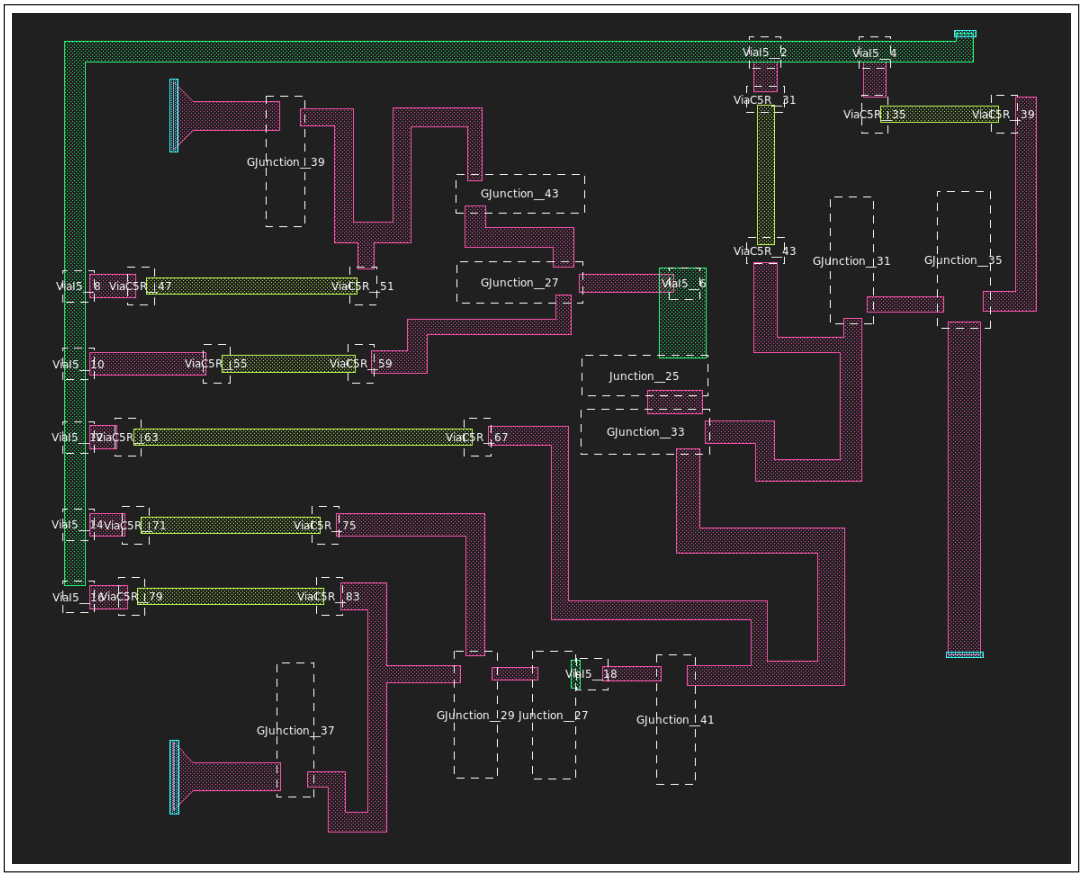


Figure 8.23: The parameterized DFF layout.

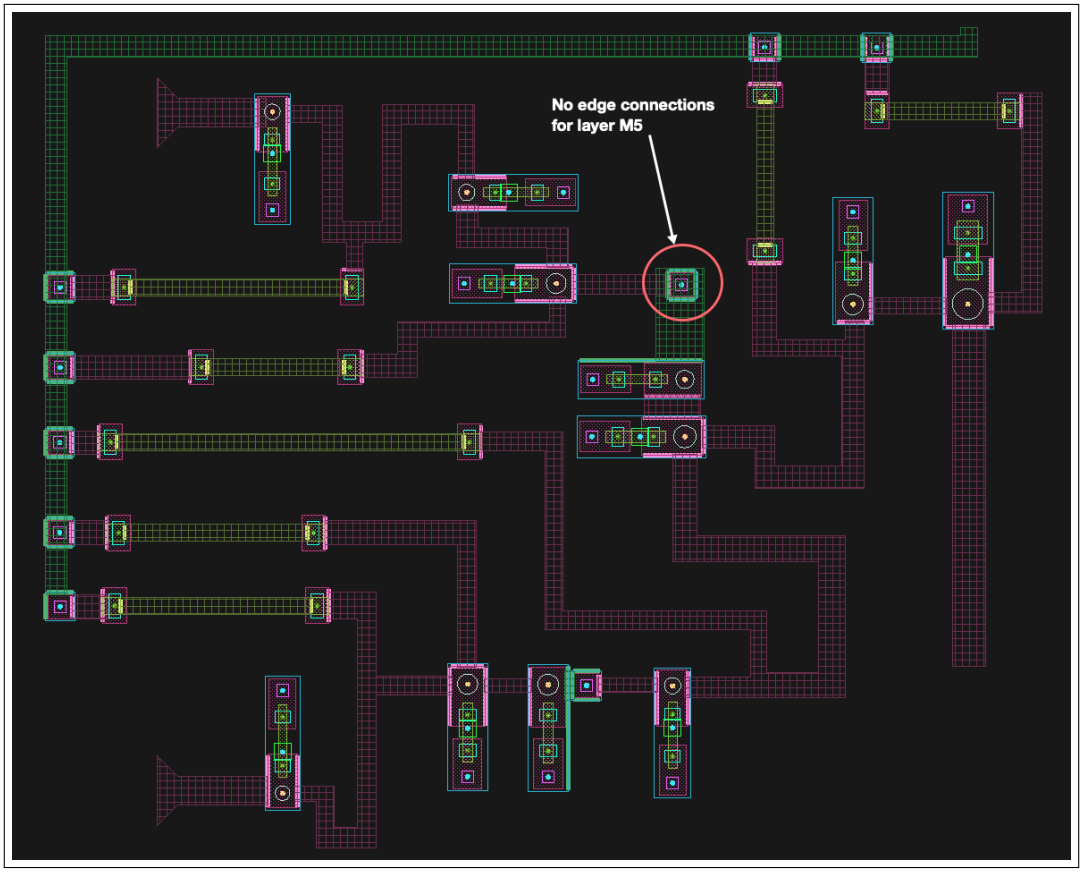


Figure 8.24: DFF circuit showing connectivity ports.

The red circled area in Fig. 8.24 shows a disconnection between metal layers M_6 and M_5. This happens because the current electrical connection algorithms checks for overlapping edges. If a polygon is connected to a device without overlapping edges, no connection is detected.

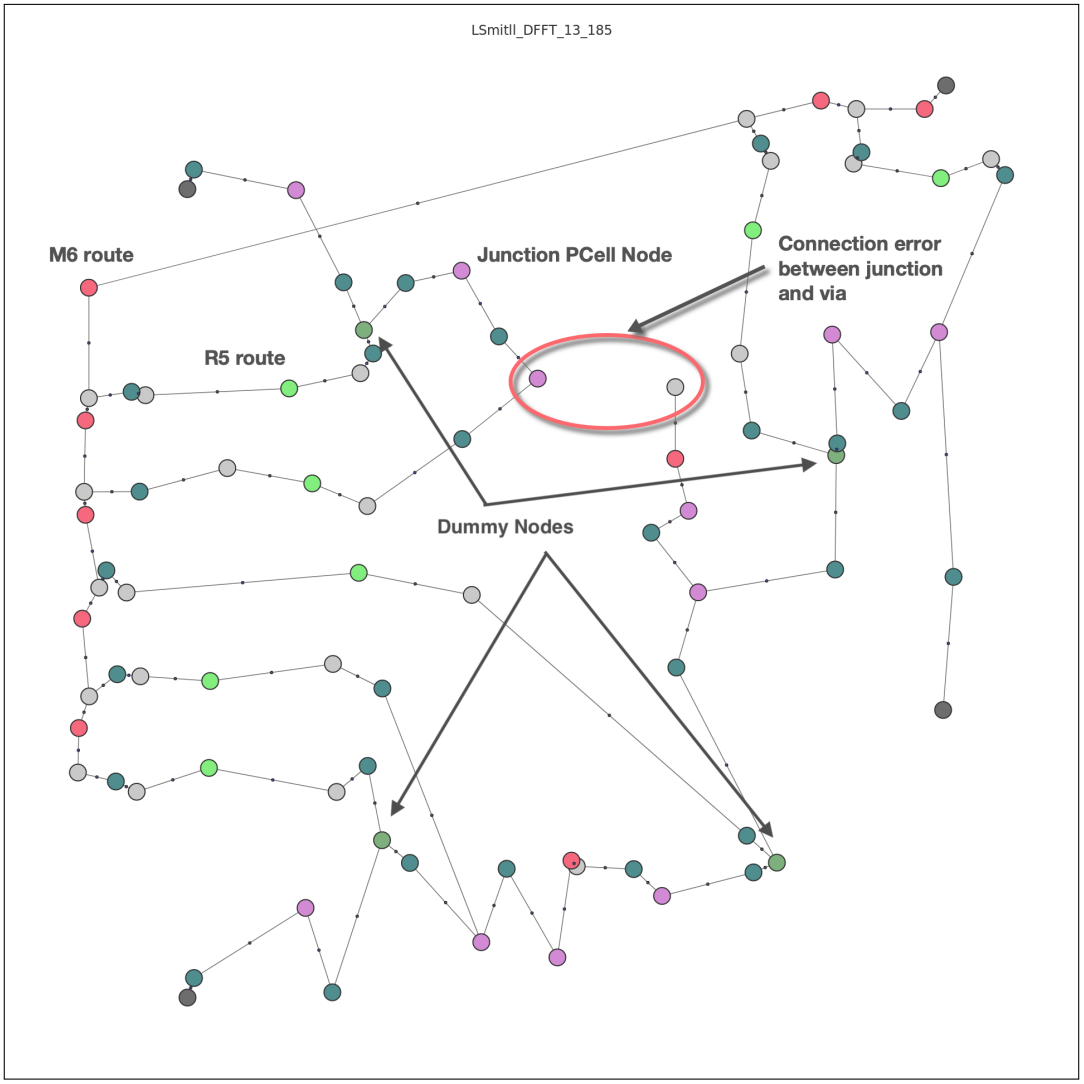


Figure 8.25: Extracted circuit net of DFF in the highest hierarchical level.

Similar to the SFQDC example, even though a disconnection is detected, the rest of the extracted netlist can still be analysed for further error inspections.

8.8 Extraction Timing Results

Table 8.1 below shows the extraction times for detecting devices and generating a netlist, for each of the listed logic gates.

Table 8.1: Extraction times for netlist extraction

Logic Gate	Device Detection (s)	Netlist Extraction (s)	Junction Count	Via Count
JTL	2.13	4.20	2	3
PTLRX	3.52	5.48	3	3
JTLT	7.80	16.35	5	9
SPLITTER	13.22	35.58	6	12
MERGER	18.41	55.21	8	14
SFQDC	39.22	117.02	9	22
DFF	42.65	180.5	11	23

From Table 8.1 it can be seen that the netlist extraction times increases as the number of detected devices increases. This is due to the fact that as the number of devices increases, the number of graph loops in the netlist increases. This creates more paths between the different devices, which leads to an increase in the extraction time of the *branch-detection* algorithm.

8.9 Conclusions and Future Work

This chapter showed the extraction of a set of designed circuit layouts at the gate level. A hierarchical approach was used to: First, extract a netlist for each individual device, and then the entire circuit. Second, the extracted nets for each cell reference are added to a single graph representation. Third, the interconnections between each device and the higher hierarchical circuit netlists are connected by adding graph edges to the corresponding overlapping nodes.

The first example, the JTL, introduced a micro-overview of how a designer can debug all the different extraction levels of a layout. The PTLRX example, showed the connections of each Josephson junction, by replacing the device node with its respective device net. The JTLT and Splitter layouts, showed that multiple *dummy nodes* can be automatically detected. The merger test shows that the junction devices connects to the correct metal layers.

The last two tests presents two errors that the current LVS extraction package has to solve in future versions: The first, shown by the SFQDC test, requires all devices to be included as a cell reference. By updating the LVS

package to solve flattened layouts, will automatically solve this problem. Second, the DFF circuit contains an electrical disconnection between a device and a metal layer. This problem can be solved by updating the electrical connection algorithm to check for *polygon-area-overlaps* along with the current *polygon-edge-overlaps* algorithm.

Future work, will involve more accurate testing with larger circuits. Including circuits that contains multiple nested circuit cells. Speed improvements can be made by caching each device net, instead of re-calculating each net every time a cell instance is called.

Chapter 9

Conclusions

The dissertation describes the development of a physical verification framework for superconductor and quantum electronics. In general, the purpose of the SPiRA framework is to check that the circuit layout corresponds to the actual designed circuit schematic, and that no process rules are violated. However, by following a parameterized approach to effectively connect process data to layout elementals, the SPiRA package has grown into a multi-purpose framework. The framework allows users to create parameterized cells, creating complex layout elementals, defining elemental relations, geometric modeling, and netlist extraction.

This dissertation introduced a novel parameterized framework that can be used for layout creation using the industry standard Python programming language. This is a replacement for designing layouts by hand, while still maintaining the option of designing layouts by hand. Building from the proposed parameterized framework it proves possible to develop a netlist extraction package using the design principles innate to the framework. Limited DRC support was also implemented using a hierarchical coding method, since it is implicit in a parameterized network. The proposed framework is the first of its kind, solving both layout circuits and parameterized circuits by connecting to a single model.

Having developed a system, rather than a single solution, makes the proposed codebase an attractive *framework* for further development of physical design verification solutions for SDE. With the current version of the framework that can effectively detect devices, implement limited rule checking, and extract layout netlists, it is indicative of having solved to proposed problems presented in this dissertation.

9.1 Connecting Process Data

A single meta-model (SPiRA Core) was developed to generate layout elements by first connecting to an external database. An object relational mapping

technique is used to convert data between process data and the defined parameters of a cell instance. PCells can be chained into complex hierarchical layouts using previously defined cells using software inheritance. Subsequently, by implementing a parameterized methodology, the SPiRA codebase evolved into a structured architecture with design patterns that can be used to build other modules, such as netlist extraction and design rule checking.

9.2 Parameterized Layout Generator

The layouts of circuit devices are defined by creating a parameterized model of a device cell. This device PCell is then added to the LVS database and later used in the device detection algorithm. The dichotomy between hand-crafted layout and parameterizing a layout via a scripting language was bridged by designing a framework that converges to a single implemented model—the SPiRA core that coalesces a set of metaclasses.

9.3 Device Detection

Device detection is done following a parameterized hierarchical approach. This method wraps all polygons into process layers, that allows dynamically extending data connected to the specific polygon instance. Via layers that connect metal layers are updated with port objects that describes these connections. Support for Design Rule Checking was implemented by defining parameter restrictions to which all elementals of an instance must apply.

9.4 Netlist Extraction

The newly proposed netlist extraction methodology is capable of automatically segregating inductive branches in a physical layout. This method uses a *mesh-to-graph* algorithm to generate a representative network using each triangle as a graph node. The edges between nodes are derived from shared polygon edges between two different triangles in the extracted mesh. Fundamentally, dummy nodes are intersections between different conducting branches. By creating a solution to effectively dissect a netlist into branches, automatically compensates for dummy node detections. This novel method is called *branch-detection*.

9.5 External Library Support

Support for external libraries was implemented using general software design patterns. The geometry module, for example, takes simple layout polygons and

converts them to physical geometries that can be used by other libraries, such as Gmsh. This module also consists of boolean methods and shape generators by connecting to the Clippers and Shapely libraries. These layout elementals are extracted using the Gdspy library, after which they are parameterized using the proposed framework.

9.6 Future Work

Future work will include extending the framework as follows:

1. Adding support for elemental stretching operations.
2. Update the SPiRA core to cache parameters and elemental instances.
3. Add DRC support for hand-crafted layout cells.
4. Parameterized elemental transformations (rotation, translation, etc) which will implicitly improve the electrical connection algorithms over different hierarchical levels.
5. Integrate with InductEx to automatically adjust inductive polygons depending on the extracted values. Before interfacing with InductEx, polygon stretching and full DRC support has to be implemented.
6. Extend the RDD to include InductEx specific information, elemental display resources, material stacking, and newly available PDK information.
7. More testing is required for the netlist extraction and connection algorithms to be considered as stable.
8. Implement comparison algorithms to compare the extracted layout netlist with that of the design schematic netlist, using graph isomorphic methods.
9. Improve speed performance for device detection and netlist extraction.

Appendices

Appendix A

Journal Paper - Influence of the Superconducting Ground Plane on the Performance of RSFQ Cells

Influence of the Superconducting Ground Plane on the Performance of RSFQ Cells

Ruben van Staden, Kyle Jackman, *Member, IEEE*, Coenrad J. Fourie, *Member, IEEE*, and Pascal Febvre, *Member, IEEE*

***Abstract*—Single-flux-quantum (SFQ) digital circuits are mostly based on cells that rely on reliable foundry processes that make use of a superconducting ground plane as a reference for the active elements and the microstrip line interconnects. The quantum of magnetic flux $h/2e$, associated with the binary information, corresponds to a magnetic field energy density that needs to be localized in space to limit interactions between adjacent cells. In other words, mutual inductances can harm the proper behaviour of circuits unless they are taken carefully into account during the design phase. We studied extensively the Josephson transmission line cell with different geometrical configurations of the ground plane and bias pads. We found with the use of InductEx that the return current sometimes follows paths that are far from what intuition tells, which can lead to nonoptimized designs. In this paper, we emphasize the limitations due to the presence of external or internal magnetic fields. Then, we compare obtained performances with the ones with optimized geometries for which the presence of the magnetic field is taken into account from the design phase.**

***Index Terms*—Magnetic fields, digital circuits, flux trapping, moats, ground plane, return current.**

I. Introduction

DEVELOPING a reliable tool chain to solve ground return current problems in SFQ circuits plays an important role for further large scale circuit integration. Unwanted magnetic fields originate from either the return current in the ground plane or an external field, such as the Earth magnetic field. In [1] the effect of the return current at different ground contact points was analysed. We analyse the effect that the magnetic field induced by the return current in the ground plane has on the circuit elements. This is done by representing the magnetic field as an inductance that couples to the circuit inductances.

Modeling the effect of return current has little significance in the design of individual cells, but does however become significant as the bias current increases to around the milliampere range [2], [3]. This normally happens when a couple of hundred thousand cells are connected to form a complete integrated circuit. We therefore investigate the effect a change in return current has on the circuit parameters when the ground contacts are placed at different locations around the chip. Since the JTL is a very stable circuit, we increased the return current exponentially to observe the effect the return current has on the circuit margins.

The return current in the ground plane can be modelled using linear equations to calculate the mutual coupling the ground plane has with the circuit elements. A new implementation on analysing external magnetic fields was implemented into Fast Fast Henry (FFH) [4], which will be used in this paper. However, in [5] a method was used to analyse the effect the external magnetic field has on the circuit parameters. In this paper the D-Flip Flop (DFF) was analysed and results are given for cases where no shield, a grid shield and a completed shield was used. This method requires a coil to be implemented around the circuit through simulation. In this paper we show that the circuit margins change significantly with a change in the ground contact location, and we derive a few general rules to improve the design with respect to the ground plane topology.

II. Ground Plane Drawbacks

The presence of the ground plane is a natural solution to limit the spread of magnetic flux lines through the shielding induced by the Meissner-Ochsenfeld effect. However, the presence of the ground plane brings the following drawbacks: magnetic vortices can be trapped in presence of a too high magnetic field energy density; the operation of the circuits can lead to a movement of vortices, causing magnetic noise and possible malfunctions of circuits in some cases. For dc bias lines the return current paths are not properly defined and the associated magnetic field can unexpectedly shift the point of operation of some cells and lead to a reduction of bias margins or even malfunctions as well. Consequently the specific position of the dc bias pads on the superconducting chips can lead to different operation margins of the digital circuits [1]. The use of moats in the ground plane and some on-chip shielding techniques have been studied already and are effective to improve the operations of the most complex circuits [1], [6]. This is done at the price of a higher complexity and of design rules based on experience rather than on quantitative established facts.

Manuscript received September 7, 2016; accepted December 15, 2016. Date of publication January 17, 2017; date of current version February 7, 2017. This work was supported in part by the South African National Research Foundation under Grant 93586 and in part by the French-South African Partenariat Hubert Curien (PHC) PROTEA No. 33944VG.

R. van Staden, K. Jackman, and C. J. Fourie are with the Department of Electrical and Electronic Engineering, Stellenbosch University, Stellenbosch 7600, South Africa (e-mail: rubenvanstaden@gmail.com; 16192044@sun.ac.za; coenrad@sun.ac.za).

P. Febvre is with the Universite Savoie Mont Blanc, IMEP-LAHC (CNRS UMR5130), 73376 Le Bourget du Lac, France (e-mail: pascal.febvre@univ-smb.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TASC.2017.2654345

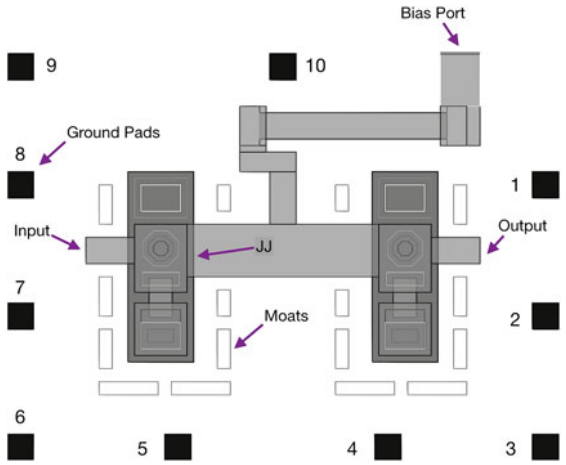


Fig. 1. FLUXONICS Foundry JTL layout [8] with the ground contact positions shown and numbered.

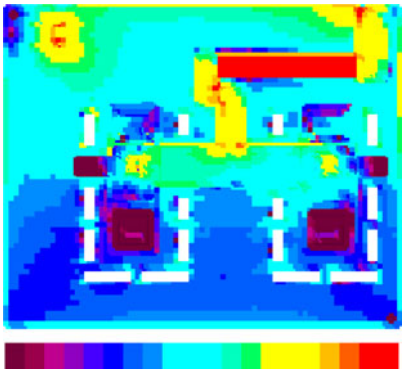


Fig. 2. Current distribution in JTL for the bias port excited and ground contact at position 9, as shown in Fig. 1. The color scale is relative to the maximum current value, with a 6 dB drop per color tick.

III. Ground Contact Positioning Effect

Rapid-Single-Flux-Quantum (RSFQ) cells typically have a superconducting ground plane that is general to all cells in the circuit. In traditional RSFQ circuits, all injected bias currents must leave through the ground plane [7]. The return current through the ground plane is not limited to a specific path in the ground plane and can spread out depending on the location of the ground contact. The JTL is used as a reference cell: see Fig. 1.

The return current in the ground plane follows the path directly under the circuit back to the bias port and then spreads to the ground contact [9]. The current in the ground plane tends to spread out to the sides of the plane, as shown in Figs. 2 and 3. The coupling factor is largest when the current direction from the bias port to the ground contact is parallel with the current flow in the inductor and their magnetic field couple (see contact 9 in Table I). The coupling is insignificant when the ground contact pad is almost directly below the bias port or close by. This validates our previous assumption of zero mutual coupling when the ground contact is directly below the bias port.

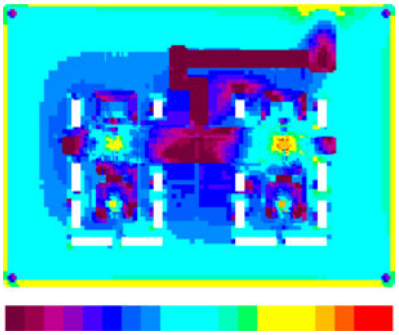


Fig. 3. Current distribution in JTL for an uniform external magnetic field applied perpendicular to the ground plane in the z-direction. The bias port is excited with the ground contact placed directly below the bias port.

TABLE I
Coupling Factors Between Ground Plane and L_2, L_3

Ground Contact (GC)	L_g (pH)	K_{g1}	K_{g2}
1	24	0 .008	−0 .008
2	45	0 .003	−0 .003
3	59	0 .001	−0 .001
4	53	0 .008	−0 .008
5	68	0 .029	−0 .028
6	84	0 .033	−0 .033
7	71	0 .040	−0 .040
8	66	0 .040	−0 .040
9	70	0 .044	−0 .044
10	30	0 .021	−0 .021

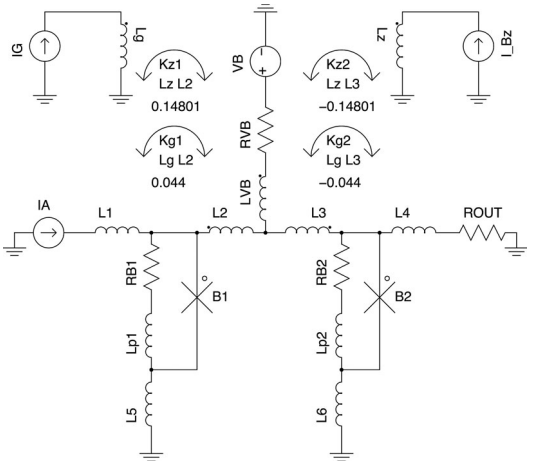


Fig. 4. JTL schematic with the added loop that represents the effect of the change in ground contact in Josephson Simulator (JSIM). Inductance L_g represents the magnetic field generated by the ground plane return current and L_z represents the inductance of the external magnetic field in the z-direction.

IV. Algorithm

We define the *magnetic inductance* as an extra inductance loop added to the system, see Fig. 4, to schematically analyse the effect that the coupling of the magnetic field, due to the return current, has on the circuit elements. This is possible since we can represent a magnetic field using an inductance with a current source. We can then change the current, I_G, to analyze the

effects that the return current has on the circuit margins, while still keeping the bias current of the JTL at its optimal value. The modeling of the circuit is done by taking multiple superconducting loops and then solving the set of linear equations simultaneously. Using Kirchhoff's voltage law, linear expressions can be derived for these superconducting loops. Each bias port in the circuit is excited using a 1 V source. To ensure that our newly derived equation gives the correct answers, the self inductance values using our new method was compared to that calculated with InductEx [10]. The linear system of equations for solving the self-inductance in Fig. 4, including the mutual coupling between the ground return current and the inductances, L_2 and L_3, leads to a rank deficient matrix. We solve self-inductance of L_2 and L_3 by placing the ground contact point directly below the structure and assume the mutual coupling is negligible. Using these fixed values for L_2 and L_3, the matrix is no longer rank deficient and the mutual coupling with the ground plane can be calculated for the various ground contact pads shown in Fig. 1. The linear system of equations that are derived from the JTL circuit are:

$$\begin{aligned}
&(L_2 I_2 + M_{g2} I_1) - (L_3 I_3 + M_{g3} I_1) = 0 \\
&(L_{gvb} I_1 + M_{g2} I_2) - (L_2 I_2 + M_{g2} I_1) + \frac{V_b}{j\omega} = 0 \\
&(L_{gvb} I_1 + M_{g3} I_3) - (L_3 I_3 + M_{g3} I_1) + \frac{V_b}{j\omega} = 0 \qquad (1)
\end{aligned}$$

Where V_b is the bias voltage and M_{g2}, M_{g3} are defined as the mutual inductances between the ground inductance L_{gvb} and L_2, L_3, respectively. The values for I_1, I_2 and I_3 are defined as the current through L_{gvb}, L_2 and L_3 for each ground contact as calculated using InductEx. Since InductEx cannot handle inductances in series, we have to combine the ground inductance with the bias line inductance, therefore $L_{gvb} = L_g + L_{vb}$. To represent this as a $Ax = b$ matrix, we simplify equation (1) to

$$\begin{aligned}
&M_{g2} I_1 - M_{g3} I_1 = L_3 I_3 - L_2 I_2 \\
&M_{g2} I_2 - M_{g2} I_1 = L_2 I_2 - L_{gvb} I_1 - \frac{V_b}{j\omega} \\
&M_{g3} I_3 - M_{g3} I_1 = L_3 I_3 - L_{gvb} I_1 - \frac{V_b}{j\omega} \qquad (2)
\end{aligned}$$

which can be written in matrix form as

$$\begin{bmatrix} I_1 & -I_1 & 0 \\ I_2 - I_1 & 0 & I_1 \\ 0 & I_1 - I_3 & -I_1 \end{bmatrix} \begin{bmatrix} M_{g2} \\ M_{g3} \\ L_{gvb} \end{bmatrix} = \begin{bmatrix} L_3 I_3 - L_2 I_2 \\ L_2 I_2 - \frac{V_b}{j\omega} \\ -L_3 I_3 + \frac{V_b}{j\omega} \end{bmatrix} \qquad (3)$$

These current values change as the ground contact position changes.

V. External Magnetic Field Effect

To analyse the effect of the external magnetic field coupling has on L_2 and L_3, we excite the circuit with a constant magnetic field in the x-, y- and z-direction. From our analysis the field in the y-direction has the strongest coupling with inductors L_2 and L_3. Using the excited current in each branch we model the coupling of L_2 and L_3 with the field as an inductor L_z, shown in Fig. 4, connected directly to a fictitious current source. The magnitude of the current represents the magnitude of the magnetic field in Tesla.

TABLE II
Coupling Factors Between External Magnetic Field and L_2, L_3

B_{ext}	K_{g1}	K_{g2}
x	0.00001	−0.00004
y	0.843	−0.842
z	0.148	−0.148

VI. Results

By keeping the ground contact stable at a specific position on the ground plane periphery while changing the return current, we can analyse at which ground contact positions the circuit margins are most sensitive to the change in the return current.

It was also observed that the closer the current path between the two points becomes to being parallel to the inductance structure, the larger the coupling factor becomes. The critical Superconducting-Quantum-Interference-Device (SQUID) loops in the circuit is that containing the bias inductance; since changing the current distribution in the ground plane drastically affects the bias inductance, which in turn changes the mutual coupling between bias inductance and the circuit inductance. In Table I the coupling factor between the ground plane inductance and L_2, L_3 is given at each position of the ground contact and in Table II the coupling factor of the external magnetic field in each direction is given, where the magnetic field magnitude is represented as an inductance, L_z. We note that the difference in coupling between GC_1 and GC_9 is one order of magnitude. In Fig. 5(b) we see that for ground contacts $GC_4 - GC_1$ the *magnetic inductance* does not have a significant change as the return current changes, even with a return current of 64 mA. This is due to the fact that when a line is drawn from the bias port to the ground contact this line is almost perpendicular to the orientation of the circuit inductance. Placing a ground pad in the top-right corner of the ground plane periphery has almost no effect on the circuit margins. The reason for this is because the length of the inductance (between the bias port and the ground pad) is very small. Even though this inductance lies parallel with L_2, L_3, its magnetic field does not couple with any of the circuit inductances.

Fig. 6 shows that the y-directed magnetic field has the strongest influence on the x-directed inductances due to a higher number of flux lines that penetrate the superconducting loop of L_2 and L_3. The x-directed external magnetic field has almost no influence on circuit operations, while the y-directed field causes the circuit to fail at around 9 μT, and the z-directed field causes circuit failure around 58 μT. In [11] measured results show that the magnetic fields are stronger when the ground pads are placed on the circuit corners, away from the bias port, than when a ground pad is placed close to the bias port.

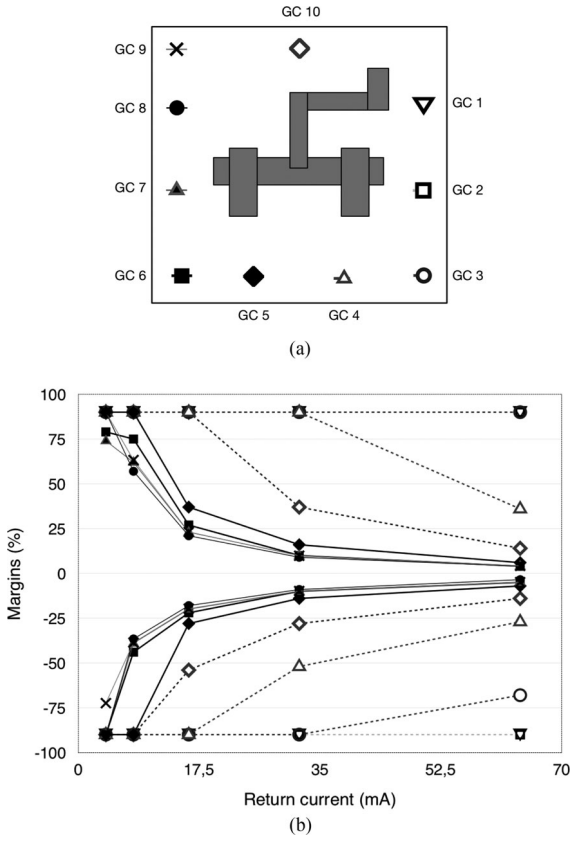


Fig. 5. Effect the ground plane has on circuit margins depending on the position of the ground contact, (a) shows the legend of (b), and (b) shows the critical margins for each ground contact.

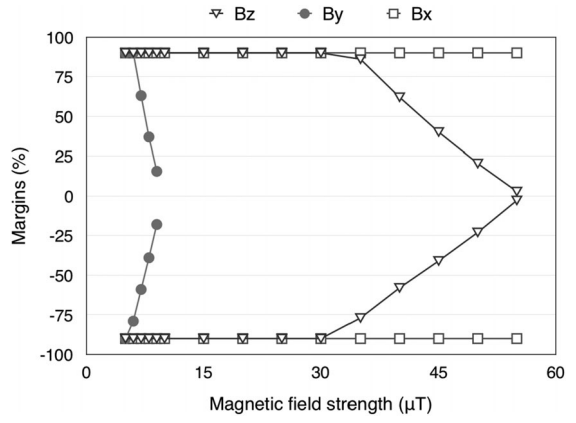


Fig. 6. Critical margins of circuit for different external magnetic field values in each directions.

VII. Conclusion

Layout designers can imagine the return current in the ground plane as a straight line connected from the current bias port to the ground contact. This *magnetic inductance* will then couple with the circuit elements depending on its orientation and distance relative to the specific circuit inductance. A good rule of thumb is to try and keep this *magnetic inductance* as perpendicular and far away as possible from the dominating (largest or most sensitive) inductances in the circuit. For instance in the JTL one wants the *magnetic inductance* to be as far away and perpendicular to the critical inductance (L_2 and L_3) as possible.

We have demonstrated that the ground contact placement around the circuit periphery does indeed affect the operating margins of the circuit elements. Numerical solutions were derived to calculate the mutual coupling between the ground plane inductance and the circuit elements. We also found that we are able to represent the effect the external magnetic field has on the circuit parameters by using mutual coupling with a inductance connected to a current source. Using our methods it is possible to test the maximum magnetic field that can be applied in any direction to the circuit before failure. Future work will involve expanding our current linear equations to include multiple ground contacts and multiple ground planes.

Acknowledgment

The authors would like to thank J. Delport for his input in margins analysis.

References

[1] A. M. Kadin, R. J. Webber, and S. Sarwana, "Effects of superconducting return currents on RSFQ circuit performance," *IEEE Trans. Appl. Supercond.*, vol. 15, no. 2, pp. 280–283, Jun. 2005.

[2] O. A. Mukhanov, D. Gupta, A. M. Kadin, and V. K. Semenov, "Superconductor analog-to-digital converters," *Proc. IEEE*, vol. 92, no. 10, pp. 1564–1584, Oct. 2004.

[3] K. L. Shepard and Z. Tian, "Return-limited inductances: A practical approach to on-chip inductance extraction," in *Proc. IEEECustom Integr. Circuits*, San Diego, CA, 1999, pp. 453–456.

[4] K. Jackman and C. J. Fourie, "Tetrahedral modeling method for inductance extraction of complex 3-D superconducting structures," *IEEE Trans. Appl. Supercond.*, vol. 26, no. 3, Apr. 2016, Art. no. 0602305.

[5] R. S. Bakolo, R. van Staden, P. Febvre, C. J. Fourie, "Modelling Magnetic Fields and Shielding Efficiency in Superconductive Integrated Circuits," *J. Supercond. Nov. Magn.*, pp. 1–5, 2016. doi:10.1007/s10948-016-3806-6

[6] V. K. Semenov and M. M. Khapaev, "How moats protect superconductor films from flux trapping," *IEEE Trans. Appl. Supercond.*, vol. 26, no. 3, Apr. 2016, Art. no. 1300710.

[7] H. Terai, Y. Kameda, S. Yorozu, A. Fujimaki, and Z. Wang, "The effects of DC bias current in large-scale SFQ circuits," *IEEE Trans. Appl. Supercond.*, vol. 13, no. 2, pp. 502–506, Jun. 2003.

[8] J. Kunert *et al.*, "Recent developments in superconductor digital electronics technology at FLUXONICS Foundry," *IEEE Trans. Appl. Supercond.*, vol. 23, no. 5, Oct. 2013, Art. no. 1101707.

[9] C. J. Fourie, S. Miyanishi, and N. Yoshikawa, "Grounding methods to reduce stray coupling in multi-layer layouts," *Proc. 15th Int. Superconductive Electron. Conf.*, Nagoya, 2015, pp. DS–P15.

[10] C. J. Fourie, "Full-Gate verification of superconducting integrated circuit layouts with InductEx," *IEEE Trans. Appl. Supercond.*, vol. 25, no. 1, Feb. 2015, Art. no. 1300209.

[11] H. Terai, S. Yorozu, A. Fujimaki, N. Yoshikawa, and Z. Wang, "Signal integrity in large-scale single-flux-quantum circuit," *Phys. C: Supercond. Appl.*, vol. 445–448, p. 1003–1007, Oct. 2006.

Appendix B

Journal Paper - Modelling Magnetic Fields and Shielding Efficiency in Superconductive Integrated Circuits

J Supercond Nov Magn
DOI 10.1007/s10948-016-3806-6

ORIGINAL PAPER

Modelling Magnetic Fields and Shielding Efficiency in Superconductive Integrated Circuits

R. S. Bakolo[1] · R. van Staden[1] · P. Febvre[2] · C. J. Fourie[1]

Received: 22 April 2016 / Accepted: 14 September 2016

Abstract We present a magnetic field model that shows how single flux quantum (SFQ)-based electronics can be affected by nearby current carrying strip lines and other external magnetic field sources. This work is an enabling step towards the design of SFQ circuits that can operate without the need for ferromagnetic shielding. Firstly, a specific 3-D coil system was optimized to apply external homogeneous magnetic fields during a magneto-quasistatic numerical analysis of a SFQ Delay-Flip-Flop (DFF) circuit. We used magnetic field and current density visualization tools to identify the most affected areas in the circuit layout. Secondly, grid patterned and solid on-chip shielding techniques were verified through simulations to design magnetic field tolerant SFQ circuits. Without any form of shielding, the DFF operated up to maximum external magnetic field of 38 or 46 μT, whereas with magnetic shields, the DFF failed at 50 or 98 μT and 50 or 86 μT for the grid and solid shields, respectively. Both shields have comparable failure points for the DFF. However, the grid-based shield has a lower influence on SFQ circuit inductance, requiring a faster re-optimization without the need to redesign the circuit.

This work has been partly funded by the French-South African Partenariat Hubert Curien (PHC) PROTEA No. 33944VG, the South African National Research Foundation, grant number 95237 and the Malawi Government.

✉ R. S. Bakolo
rbakolo@gmail.com

[1] Department of Electrical and Electronic Engineering, Stellenbosch University, Stellenbosch 7600, South Africa

[2] IMEP-LAHC (CNRS UMR5130), Université Savoie Mont Blanc, 73376 Le Bourget du Lac, France

Keywords SFQ · Magnetic fields · Modelling · Magnetic shielding · InductEx · Superconducting electronics

1 Introduction

Even though single flux quantum (SFQ)-based electronics presents a huge opportunity in high speed and low power circuit design, there are still challenges that need to be addressed. Of great concern are challenges related to magnetic fields. These include susceptibility to minute magnetic field interferences [1], need for large currents required for biasing [2–4], which also generate unwanted fields on-chip, and flux trapping [5]. All of which have the ability to cause malfunction in SFQ circuits, thereby hindering further attempts for large scale integration.

Magnetic fields can emanate from two sources: those that are external and the inherent ones from the circuit's bias lines and ground return currents. Suggested solutions have ranged from implementation of on-chip shields [6, 7] to keep magnetic fields away from the target cell to the reduction of bias currents. On-chip shielding techniques are still work-in-progress. Current recycling [8, 9] was suggested with the main focus of reducing the overall required bias current for a fairly large circuit. In this approach, only circuits of equal bias requirements are serially biased. Flux quanta can be trapped in critical circuit elements during cooling from room to cryogenic temperatures due, in particular, to thermal gradients. At the time of writing, the perfect cooling rate to prevent flux trapping is still unknown, although a figure of 3 mK/s has been reported [10] to be particularly efficient. Currently, flux trapping can be reduced, but not eliminated, by careful placement of moats [5, 11] at strategic positions on the circuits especially close to junctions and critical inductors.

Published online: 22 September 2016

So much has been done but not much progress has been made, as inferred in [12]. Suggested solutions to deal with magnetic fields have been isolated and most reported work has concentrated on small scale circuits. Stray magnetic fields continue to negatively affect SFQ circuits. In addition, issues of large bias currents and flux trapping are yet to be dealt with to the full. To properly address the problems of magnetic fields in SFQ circuits, a better understanding of these fields in SFQ circuits is required. We have been working towards modeling of arbitrary magnetic fields using coil constructs and bias lines in the vicinity of the circuits [13]. Shielding is normally achieved by using the topmost layer to envelop almost the entire circuit. The challenge is that shielding causes circuit inductors to drop in value and the circuit needs to be redesigned and re-optimized. In addition, with a solid shield in place, moats designed to trap flux become irrelevant and yet the flux trapping problem still persists. We present magnetic field simulation models using multidimensional coil constructs, magnetic and current density visualization models, and a grid shielding technique that has less effect on the circuit inductors.

2 Magnetic Field Simulations

The effect of running a current carrying conductor over or around a SFQ circuit was investigated in [13]. In this paper, we present how magnetic fields can be modeled using a 3-D coil system that roughly represents magnetic field orientation in all three orthogonal directions. In the calculation models reported here, the ground plane extends five times the height between ground plane and the second wiring layer beyond the furthest extents of the circuit, and simulation results for circuit operation range agree much better with measurements [1] than when the ground plane is cropped for complexity reduction as done earlier [13].

In this work, the following tools were used gEDA [14] for schematic editing, Josephson Simulator (JSIM) [15] for circuit simulations, Layout System for Individuals (LASI) [16] for making layouts, and InductEx [17] for inductance extraction and modelling of coil constructs.

2.1 A DFF Cell as Test Circuit

A 5-Josephson junction DFF has been used as a testbed for this work due to its simplicity and high occurrence in most SFQ circuits. The layout was made in accordance with the *Hypres'* 4.5 kA/cm^2 process design rules [18]. The process has 3 wiring layers : $M1$, $M2$, and $M3$. In this work, it is the topmost layer, $M3$, that has been used for making on-chip shields. The DFF circuit, shown in Fig. 1, was first simulated without coils to determine the best working

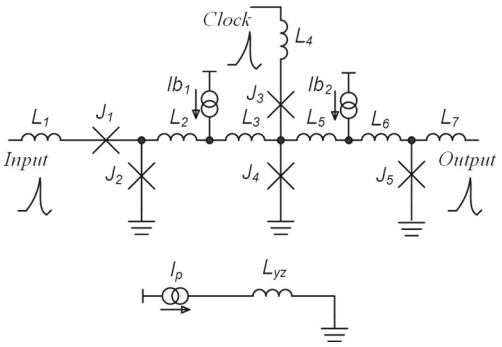


Fig. 1 A 5-Josephson junction DFF shown with one coil L_{yz}. Coil coupling to DFF inductances not shown. Parameters : $L_1 = 1.86\ pH$, $L_2 = 1.59\ pH$, $L_3 = 7.73\ pH$, $L_4 = 1.5\ pH$, $L_5 = 2.13\ pH$, $L_6 = 1.3\ pH$, $L_7 = 1.91\ pH$, $L_{yz} = 920.8\ pH$ at radius $R = 125\ \mu m$, $J_1 = J_4 = 200\ \mu A$, $J_2 = J_5 = 250\ \mu A$, $J_3 = 150\ \mu A$, $Ib_1 = 230\ \mu A$ and $Ib_2 = 135\ \mu A$

parameters and margins. It was used throughout the simulations. Initial optimized parameters are indicated in the absence of magnetic field and correspond to bias margins of $-53 \sim 42\ \%$.

2.2 A 3-D Coil System for Generating Magnetic Fields

A system of three orthogonal coils, that do not appear on the physical layout nor on the final fabrication plots, has been used to generate a magnetic field on the DFF cell. In *InductEx*, the coils can be modeled with current injection ports, finite radius, and segment size. A current flowing in the coil produces a magnetic field that couples to and hence affects the DFF circuit (refer to Fig. 2). The flux density B_0

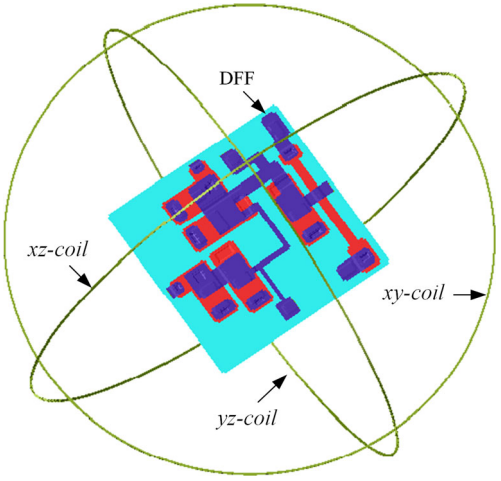


Fig. 2 Coils around the DFF laid out for *Hypres'* 4.5 kA/cm^2 process, DFF size is 100 μm by 100 μm. Radii of R = 125, 130, and 135 μm were chosen for each coil to prevent them from touching each other

at the center of a coil can be solved from the *Biot Savart law* as:

$$B_0 = \frac{\mu_0 I}{2R}, \tag{1}$$

where R is the coil radius and I is the amount of current through the coil. R and I determine the amount of flux density experienced by the circuit. To ensure that the magnetic field is fairly constant on all the elements on the circuit, a detailed analysis was carried out. In practice, the magnetic field varies along the coil axis and also radially from its value at the centre of the coil. The axial field, created by the $yz-$coil for instance, varies with the distance x from the centre according to

$$B(x) = B_0\left(1 + \left(\frac{x}{R}\right)^2\right)^{-3/2} \approx B_0\left(1 - \frac{3}{2}\left(\frac{x}{R}\right)^2\right) \tag{2}$$

for $x << R$. Though several (I, R) couples of constant ratio I/R generate the same flux density at the center of the coil according to (1), they do not lead all to a satisfactory flux density uniformity over the size of the chip. Only (I, R) couples for which R are sufficiently high with respect to x allow to neglect the spatial variations of the flux density (see (2)). Ideally, the coils should have a very large radius in comparison with the chip size. A compromise to optimize computing power and speed of the 3-D simulations was found by gradually increasing the current I through the coil until the DFF test circuit failed, for several radii R ranging from 75 to 150 μm. The failure point was determined by monitoring margins of the DFF cell until they reach 0 %. For each coil radius, the corresponding maximum acceptable current is $I_{\max}(R)$. The simulation results for $yz-$coil are shown in Fig. 3a. It can be observed that $I_{\max}(R)/R$ begins to stabilize above R = 120 μm. This means that the DFF circuit under test experiences mostly the same magnetic flux density conditions over its area when the coil radius is higher than R = 120 μm, corresponding to the same homogeneous field. The results are confirmed with a magnetic field plot in Fig. 3b. It can be observed that the color and hence strength of the fields is uniform, in the center, for a coil of R=125 μm. The other $xy-$ and $xz-$coils showed similar results.

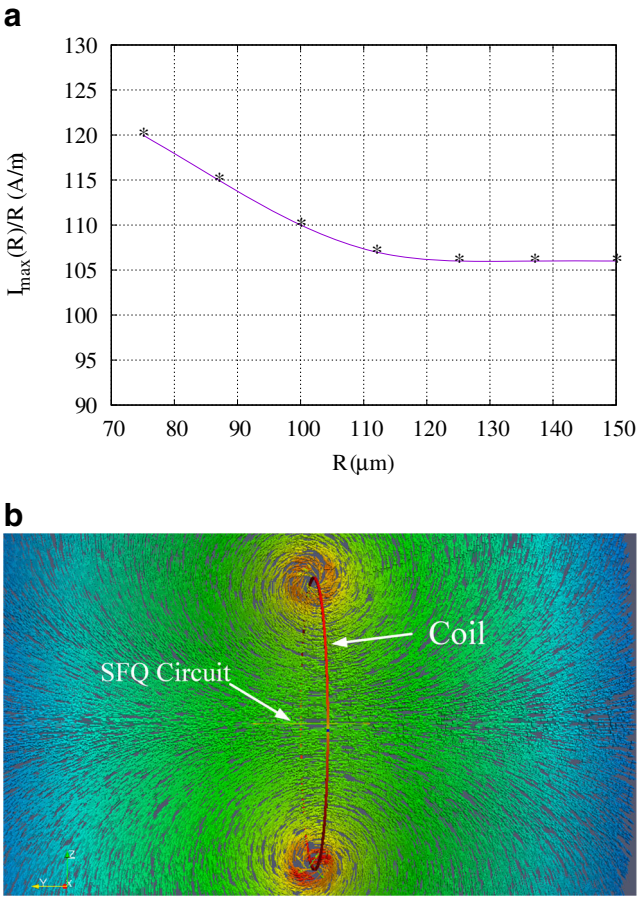


Fig. 3 (**a**) A plot of $I_{\max}(R)/R$ against R for the $yz-$coil and (**b**) cross-sectional view of the calculated magnetic field confirming the homogeneity of field at the center - color uniformity

3 Results

3.1 DFF Affected by Magnetic Field from One Coil

With a current $I_{yz} \approx$ 12.9 mA in the $yz-$coil of radius R = 125 μm and inductance L_{yz} = 920.8 pH, used to couple the magnetic field to the inductors in the JSIM transient simulation model, the magnetic field at the centre of the coil is 65 μT, to simulate the maximum magnetic field of the Earth. The bias margins dropped to $-48.5 \sim 14.1$ %. For the $yz-$coil orientation, the produced magnetic fields are parallel to the DFF gate. Parallel fields are more detrimental to the operation of SFQ circuits than perpendicular ones [1]. The DFF completely malfunctions at $I_{yz} \approx$ 13.5 mA (68 μT).

Nevertheless, the DFF can be re-optimized to work in an ambient field of 68 μT by reducing bias current I_{b1} to 100 μA and increasing the value of inductances such as L_3. This approach works because it is understood that the external magnetic field inductively contributes to an additional biasing, which makes the DFF circuit fail, among other factors. By making the inherent parameters, such as L_3, stiffer to this additional biasing, circuit operation can be restored with reasonable working margins. However, this only works if the magnetic field is oriented along the x-axis in the same direction as that used in simulation. Therefore, margin optimization at a known field requires that circuits be placed in a specific orientation.

Table 1 Values of DFF circuit inductances in presence of grid and solid shields

L	$L(Original) - pH$	$L(Grid) - pH$	$L(Solid) - pH$
L_1	1.86	1.65	1.46
L_2	1.59	1.42	1.06
L_3	7.73	7.19	5.69
L_4	1.50	1.34	1.25
L_5	2.13	1.79	1.44
L_6	1.30	1.11	0.88
L_7	1.91	1.72	1.48

3.2 Shielding Solutions

Grid and solid shields were first investigated in the absence of external magnetic field. The solid shield, a continuous superconducting layer, is the only current effective way to keep out external magnetic fields on-chip [7, 19]. However, such an approach results in the circuit inductors being reduced since the magnetic field lines are constrained by the presence of the shield. It also makes input/output and bias lines difficult to place. This requires that all circuit inductors be redesigned and possibly some circuit elements moved around. Consequently, a grid-patterned shield with minimal effect on the values of inductances was also investigated. The grid shield was made with 2.5 μm strips with a spacing of 5 μm. Table 1 shows the values of DFF inductances when grid and solid shields are implemented. Both shields were grounded at identified positions on the cell for more effective shielding [7]. It can be observed that a solid shield leads to a higher reduction of inductances than a grid shield.

The effectiveness of these shielding approaches can be viewed through our in-house magnetic and current density visualization tool as shown in Fig. 4. The DFF is much smaller compared to the coil diameter, and therefore, it was easier to visualize the fields within the DFF. Furthermore, if a shield keeps an externally applied field out, it would keep an internally generated field in, due to reciprocity. Hence, a port on the DFF (circled in Fig. 4) was excited with a sinusoidal voltage of $1v$ at a frequency of 10 GHz. The magnetic patterns reveal that the solid shield is marginally more effective than the grid one. In Fig. 4a, the DFF is unshielded and magnetic fields freely spread out from the injection point within the DFF. The DFF with a solid shield is shown in Fig. 4b, and it can be observed that most of the magnetic field is spread within the DFF because the solid shield provides a continuous flow of surface currents, which in turn generates other fields. However, the grid shield in Fig. 4c is able to interrupt the magnetic field lines without providing a continuous flow of surface currents. From the simulations shown in Fig. 4, a field of 46 μT is found at points marked X for the unshielded DFF, 0.387 μT for the solid shield, and 0.0178 μT for the grid shield. This shows that the grid shield is better at containing fields than the solid shield. However, more points are required to ascertain the overall effectiveness of a shield when using the visualization approach.

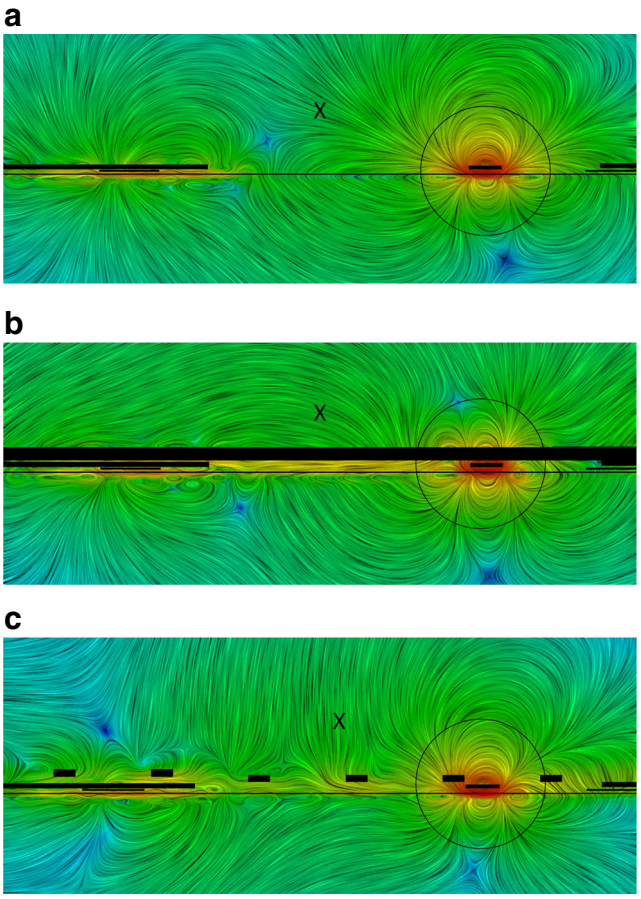


Fig. 4 Magnetic field images with cross-section of the DFF and a test current injected at the encircled points - no external fields. (**a**) unshielded DFF - cross-section, (**b**) with solid shielding, (**c**) with grid shielding. Point X was chosen arbitrarily

3.3 Shielded DFF in the Presence of External Magnetic Field

Due to the asymmetrical nature of the DFF layout, failure points were dependent on the direction of magnetic field. Therefore, both positive and negative coil currents were used to reverse the produced magnetic fields. Consequently, two failure points were realized, of which the lower one was deemed critical. A grid shielded DFF failed at a current of -34 mA (171 μT) or 25 mA (126 μT) through the yz-coil, while for the solid shield, it was at -24 mA (121 μT) or 29 mA (146 μT). In a practical setup, external magnetic fields can take any orientation. To accommodate this, we simulated the DFF surrounded by the 3-D coils as shown

in Fig. 2, first the unshielded DFF and then the ones covered with grid and solid shields. Without shielding, the DFF failed at −4.5 mA (38 μT) or 5.5 mA (46 μT) injected into each of the coils. A grid shielded DFF failed at −6 mA (50 μT) or 11.7 mA (98 μT), while the solid shielded one failed at −6 mA (50 μT) or 10.3 mA (86 μT). In this case, the magnetic field densities in parenthesis are vector sums of fields produced by the three coils at the quoted current.

The threshold magnetic fields, for both yz and 3-D coils cases, differ by small margins, and this confirms that both grid and solid shields have comparable efficiency at shielding SFQ circuits. However, the grid shield has less effect on circuit inductance, which is an advantage as re-optimization may not be required to obtain acceptable operating margins. For optimum shielding, SFQ chips need to be aligned correctly against specific field vectors.

4 Conclusion

We have presented magnetic field models that show how external magnetic fields can be modelled by using coils around a SFQ gate. The results obtained agree well with previous work. The coils in yz and xz orientations produce parallel fields which affect the circuit more, unlike the xy coil which produces perpendicular fields. By using magnetic and current density visualization tools, we are able to pin-point magnetic field trouble areas on the SFQ circuit. Apart from the solid shield already proposed and available, we have proposed a grid shield, which is equally efficient. The advantage with the grid-shield is that circuit inductance values are less affected compared to the solid shield. Only coupling coefficients between coils and circuit inductance were considered to determine failure points of the DFF. We are now investigating Josephson junction's critical current (I_c) drifts due to external fields as this may have an effect on the maximum magnetic field a SFQ circuit can withstand.

References

1. Collot, R., Febvre, P., Kunert, J., Meyer, H.G.: Operation of low-Tc circuits in a magnetic environment. IEEE Trans. Appl. Supercond. **3**(1700404), 23 (2013)
2. Suzuki, M., Maezawa, M., Hirayama, F.: Effects of magnetic fields induced by bias currents on operation of RSFQ circuits. Phys. C **412-414**, 1576–1579 (2004)
3. Terai, H., Kameda, Y., Yorozu, S., Fujimaki, A., Zhen, W.: The effects of DC bias current in large-scale SFQ circuits. IEEE Trans. Appl. Supercond. **13**(2), 502–506 (2003)
4. Tolkacheva, E., Engseth, H., Kataeva, I., Kidiyarova-Shevchenko, A.: Influence of the bias supply lines on the performance of RSFQ, circuits. IEEE Trans. Appl. Supercond. **15**(2), 276–279 (2005)
5. Narayana, S., Polyakov, Y.A., Semenov, V.K.: Evaluation of flux trapping in superconducting circuits. IEEE Trans. Appl. Supercond. **19**(3), 640–643 (2009)
6. Mizugaki, Y., Kashiwa, R., Moriya, M., Usami, K., Kobayashi, T.: Grounding positions of superconducting layer for effective magnetic isolation in Josephson integrated circuits. J. Appl. Phys. **11**(114509), 101 (2007)
7. Mizugaki, Y., Kashiwa, R.: Magnetic shielding effect of grounded superconducting niobium layers. J. Phys. Conf. Ser. **1**(012056), 97 (2008)
8. Kadin, A.M., Webber, R.J., Sarwana, S.: Effects of superconducting return currents on RSFQ circuit performance. IEEE Trans. Appl. Supercond. **15**(2), 280–283 (2005)
9. Kang, J.H., Kaplan, S.B.: Current recycling and SFQ signal transfer in large scale RSFQ circuits. IEEE Trans. Appl. Supercond. **13**(2), 547–550 (2003)
10. Herr, Q.P., Osborne, J., Micah, S.J.A., Harold, H.: Reproducible operating margins on a 72,800-Device digital superconducting chip. Supercond. Sci. Technol. **12**(124003), 28 (2015)
11. Fujiwara, K., Nagasawa, S., Hashimoto, Y., Hidaka, M., Yoshikawa, N., Tanaka, M., Akaike, H., Fujimaki, A., Takagi, K., Takagi, N.: Research on effective moat configuration for nb Multi-Layer device structure. IEEE Trans. Appl. Supercond. **19**(3), 603–606 (2009)
12. Tolpygo, S.K.: Superconductor digital electronics: scalability and energy efficiency issues. Low Temp. Phys. **42**, 361–379 (2016)
13. Bakolo, R.S., Fourie, C.J.: Modelling of the Influence of Magnetic Fields on the Operation of Digital Superconductive Circuits. 15th ISEC 2015, 1–3 (2015)
14. GNU's General Public Licence—Electronic Design Automation: gEDA. http://www.geda-project.org (2014)
15. Fang, E.S., van Duzer, T.: A Josephson integrated circuit simulator (JSIM) for superconductive electronic applications. 2nd ISEC 1989, 407–410 (1989)
16. Layout System for Individuals. http://www.lasihomesite.com (2014)
17. Fourie, C.J., Wetzstein, O., Ortlepp, T., Kunert, J.: Three-dimensional multi-terminal superconductive integrated circuit inductance extraction. Supercond. Sci. Technol. **125015**, 24 (2011)
18. Hypres: Niobium Integrated Circuit Fabrication: Design Rules Rev 24. http://www.hypres.com (2014)
19. Yamanashi, Y., Yoshikawa, N.: Design and evaluation of magnetic field tolerant single flux quantum circuits for superconductive sensing systems. IEICE Trans. Electron. **97**(3), 178–181 (2014)

Appendix C

External Software Libraries

This Appendix covers the external libraries used by the SPiRA framework, ranging from geometry construction, to netlist generation.

C.1 Gmsh Library

Gmsh [38] is a three-dimensional (3D) finite element mesh solver. It uses a boundary representation to describe geometries. Creating a geometry consists of the following sequential steps; first the boundary points of the polygons are defined. Next, the lines connecting these points are defined, followed by the surfaces they create and finally a volume can be created by extruding these surfaces. This LGM creates these volumes by either connecting to the built-in Gmsh 3D moduler or the openCASCADE 3D moduler. The openCASCADE library is a software development kit (SDK) intended for development of applications dealing with 3D CAD data. It includes a set of C++ class libraries, which provides services for 3D surface and volume modeling. The openCASCADE-backend is a C++ wrapper around the openCASCADE geometry modeling library built into the Gmsh library.

C.2 Pygmsh library

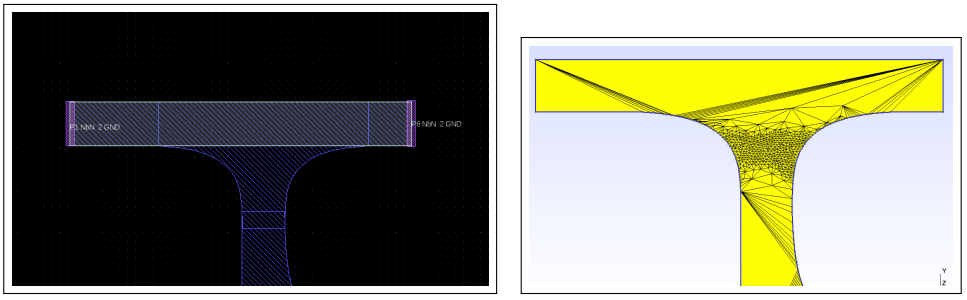
A finite element mesh is a tessellation of a given subset of the three-dimensional space by elementary geometrical elements of various shapes (in Gmsh's case: lines, triangles, quadrangles, tetrahedra, prisms, hexahedra and pyramids), arranged in such a way that if two of them intersect, they do so along a face, an edge or a node, and never otherwise. This class provides a Python interface for the Gmsh scripting language. It aims at working around some of Gmsh's inconveniences (e.g., having to manually assign an ID for every entity created) and providing access to Python's features.

These geometrical entities are assigned identification numbers when they are created. Elementary geometrical entities can then be manipulated in var-

ious ways, for example using the Translate, Rotate, Scale or Symmetry commands. Groups of elementary geometrical entities can also be defined and are called 'physical' entities. These physical entities cannot be modified by geometry commands: their only purpose is to assemble elementary entities into larger groups so that they can be referred to by the mesh module as single entities. As is the case with elementary entities, each physical point, physical line, physical surface or physical volume must be assigned a unique identification number [39].

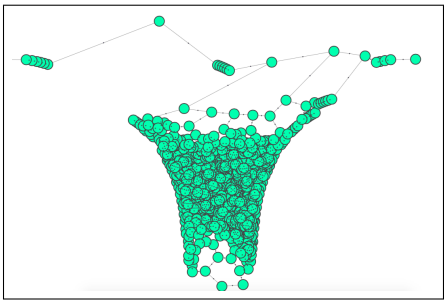
C.3 Shapely Library

Shapely is a library for manipulation and analysis of planar geometric objects [40]. Shapes are higher-level polygon objects that convert to lower-level polygons, such as circles, rectangles, etc.



(a) Original curved layout.

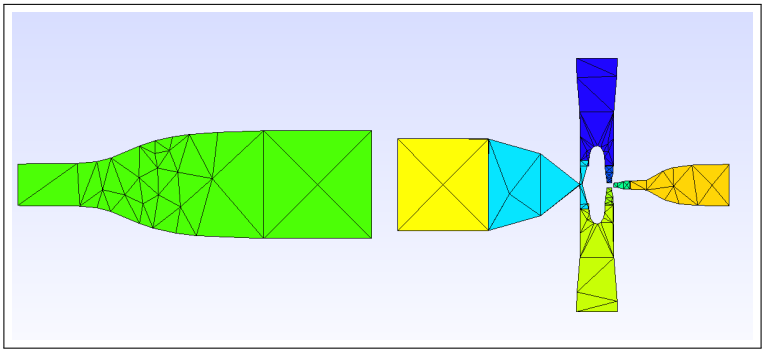
(b) Meshed structure of merged polygons.



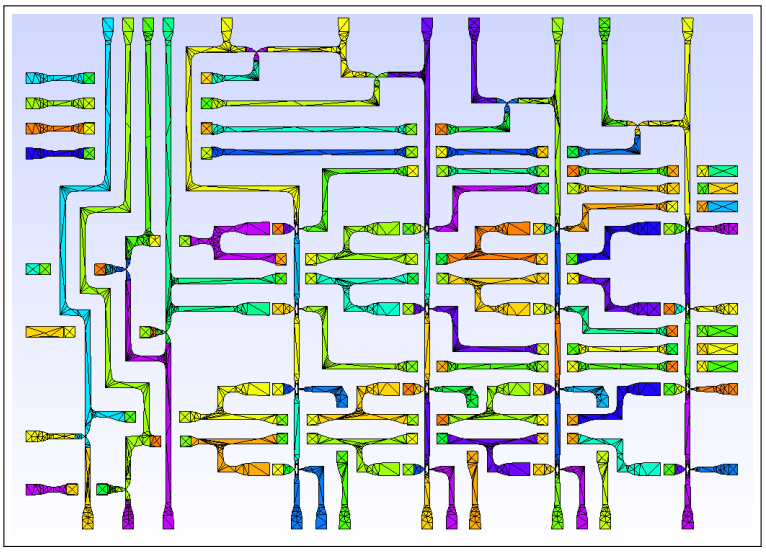
(c) Graph generated from mesh.

Figure C.1: Mesh problems occurring due to curved polygons.

Meshing curved shapes results in generating a mesh consisting of many small triangle segments. This is caused by the large mesh sampling necessary to accurately model the boundaries. Converting a polygon with curves to a more discrete form is required for netlist generation for two reasons: Processing time of extracting a netlist increases, and the implemented graph filtering algorithms causes errors.



(a) Generated mesh after applying the Douglas-Peucker smoothing algorithm.



(b) Mesh result from a larger circuit example. Each color represents a uniquely labeled polygon.

Figure C.2: Mesh results after applying the anti-smoothing algorithm.

After having tested for a curved polygon, an anti-smoothing algorithm is applied, Fig. C.1. The algorithm used is called the Douglas-Peucker algorithm [41] which takes a curve composed of line segments and finds a similar curve with fewer points. An example of applying this smoothing algorithm is depicted in Fig. C.2.

C.4 Clippers Library

The *Clippers* library performs line and polygon clipping intersection, union, difference and exclusive-or, and line and polygon offsetting. The library is based on Vatti's clipping algorithm [42]. A Python implementation is used, *Pyclipper*, which is a Cython wrapper around the C++ translation of the Clippers library [43]. Even thought the Gdspy library has some of the inherit

futures of the clippers library implemented, it is not fully sufficient for the purposes of this project. Therefore, a native geometry moduler is implemented that has its own Clippers interface.

C.5 Meshio Library

The *Meshio* library can create multiple mesh structures for different software systems, using the geometry file (.geo) generated by Gmsh. The mesh data is generated by the Meshio library that connects to the Pygmsh library. Connecting the Physical Geometries to the Meshio library enables SPiRA to generate mesh structures in a variety of different formats used by commercial libraries, such as ANSYS. The output mesh can also be written to a VTK or VTU file, enabling the use of Paraview for more accurate inspection.

C.6 NetworkX Library

The *NetworkX* library is used for graph construction. All vertex attributes are presented into a dictionary data structure. These data structures can be extended with Python objects. In SPiRA each graph vertex has two attributes key entries; *surface* and *device*. Every vertex in the graph network as a *surface* attribute, since it contains data unique to the polygon in which the vertex (triangle) falls. On the other hand, only vertices that represent a device have a *device* attribute. Edges are connected by shared triangle edges and the generated graph represents the basic topology of a merged metal polygon structure.

C.7 Visualization

Visualizing the generated graphs plays an integrate role in the debugging of a complex system like a LVS tool. Several visualization libraries has been implemented and tested. Here we discuss the different implementations and why the final result was chosen.

First versions of SPiRA used the *matplotlib* library. Matplotlib a vanilla simple to use two-dimensional plotting tool that allows for fast data visualization. It is fast and easy to implement, but can slow down when viewing large networks.

The second library implemented was, *Plotly*, which is an online data analytics and visualization tool. Plotly allows graphing in either Python, Ruby, Julia and more. This library generates an interactive graph networking plot that can be run online of offline. It integrates with Jupyter Notebook to making debugging easier.

The third and final library is called, Bokeh. It was developed by Anaconda's development team in 2012 with funding from Defence Advanced Re-

search Projects Agency. Bokeh is an open source and free to use for any type of project. The major concept of Bokeh is that graphs are built up one layer at a time. Its goal is to provide elegant, concise construction of versatile graphics, and to extend this capability with high-performance interactivity over very large or streaming datasets. It also generates interactive plot, like Plotly and also integrates seamlessly with Jupyter. Bokeh's ease of implementation and extendibility makes it more attractive than Plotly. Bokeh also seems to faster and more light-weight than Plotly. Bokeh has better support with the graphing library in use, NetworkX and seems to be faster in a Jupyter Notebook than Plotly.

Appendix D

Geometry Modeling

The Layout Geometry Moduler (LGM) is responsible for retrieving the geometric information related to cell polygons [44]. Device structures can be extracted by recognizing geometric structure patterns using boolean operations after parsing layout elementals. These pattern recognition algorithms are highly dependent on the accuracy of the LGM package [45]. Operations include merging (for layer-to-layer connections), intersection (for device detection) and difference (for layer-to-device connections).

The resultant polygons are converted to *physical geometries*, which are required by the Gmsh library [46], [38]. These physical geometries are used for mesh generation of the metal layers in order to extract a netlist. Polygons can either be positive (clockwise point orientation) or negative (counter-clockwise point orientation). Positive polygons represent surfaces that can be extruded and negative polygons represent polygon holes.

D.1 Objectives

Physical design verification is a geometry related problem. Therefore, a separate module is created inside the SPiRA framework responsible to creating and manipulating geometric objects. The LGK module solves the following problems:

- Manipulating and extracting routes between different ports. When designing a PCell routes are defined between different ports, but when parsing a layout the polygons with metal purpose layer have to be converted to route objects.
- Easily create and defined new geometric shapes, which can be used for device creation when defining a PCell. An example of this is creating a yTron or nTron device that contains arbitrary curves.

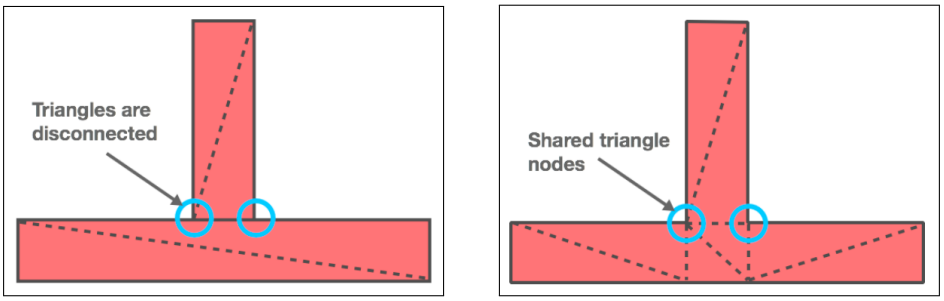
- All metal polygon elemental have to be converted to a physical geometry that can be parsed by the Gmsh library. This is necessary for both PCells and hand-designed layouts.
- In order to detect devices, such as vias and junction, it is necessary to apply boolean operations between different layers.

D.2 Physical Geometry

Identifying metal polygons that connect layout structures plays an important role in extracting a netlist. All polygon elementals are assimilated into a coherently connected network, called a physical geometry. The result of which can be used for generating edge connections between layout elementals. By converting a polygon to a physical geometry a set of new data attributes are created that is required by the Pygmsh library to effectively interface with the Gmsh library.

D.2.0.1 Built-in Engine

The SPiRA project started in 2016. The idea of using mesh elements for netlist extraction originated in the beginning of 2017. The first versions of SPiRA implemented geometry construction using the Gmsh built-in engine. The built-in engine does not support any boolean operations. At the time a native Gmsh parser was written in Python for basic Gmsh element support. Extracting a mesh using the built-in engine resulted in nuanced behaviours as the package grew in complexity. The only way to coherently generate a mesh is to send the entire polygon set to Gmsh at the same time, making it difficult to contain geometric information belonging to each specific polygon instance.



(a) Meshed triangles do not connect if two separate polygons are meshed.

(b) Mesh connects if a merged polygon is send to Gmsh.

Figure D.1: Built-in engine mesh discontinuity

Meshing a polygon set requires that all polygons in the set be merged prior to meshing, otherwise mesh discontinuity arises as shown in Fig. D.1. In

some cases it is necessary to bind information from a generated triangle t_i to a specific polygon P_i, but this polygon information gets lost when merging.

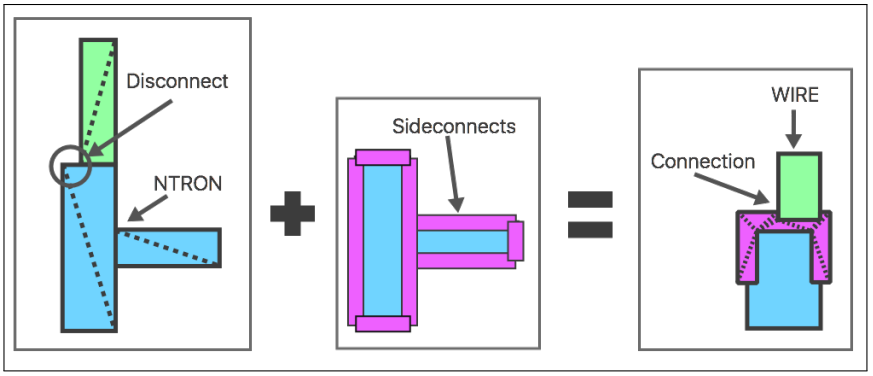
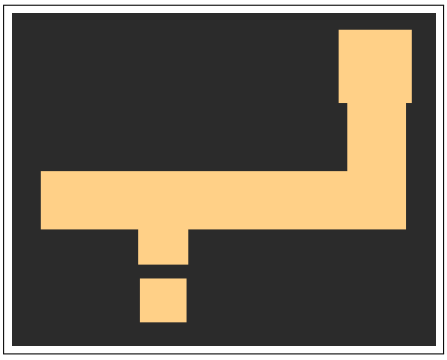


Figure D.2: Connection created by EdgePolygons

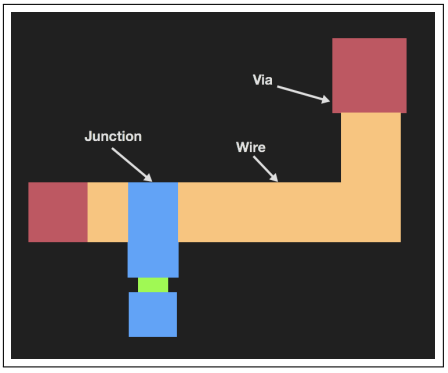
The initial solution to contain polygon attributes used Edge Polygons to detect intersection points. An Edge Polygon is created by extruding a single polygon edge to a two-dimensional polygon. Extruding the edges e_i of a polygon P_i creates a set of edge polygons $E_{Pi} = \{e|e \in P_i\}$. Intersection operations are applied to each e_i in E_{Pi} with a different polygon P_j. If this edge e_i and polygon P_j overlaps then difference operations are applied. The result is a new merged polygon G_i (polygon compatible with Gmsh) that has matching points, shown in Fig. D.2.

D.2.0.2 OpenCASCADE Engine

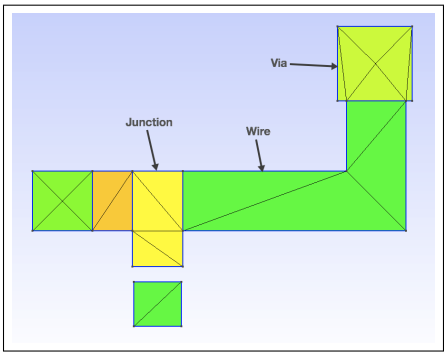
In April 2017 Gmsh version 3.0.0 implemented new constructive solid geometry features and boolean operations using OpenCASCADE. Before rewriting the native Python parser for Gmsh to support the OpenCASCADE features, it was decided to use the Pygmsh library, which at the time had already been updated to include OpenCASCADE. The OpenCASCADE library allows a list of separate polygons to be send to Gmsh as input. Merging these polygons overcomes the issue of discontinuous meshes, but at the same time keeps the original polygon data and shapes.



(a) Polygon information is lost after merging.



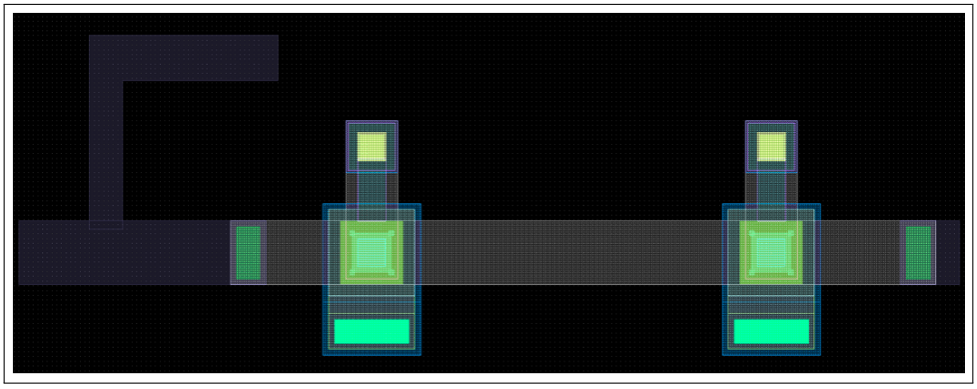
(b) Polygon information is kept when merging using the OpenCASCADE-backend.



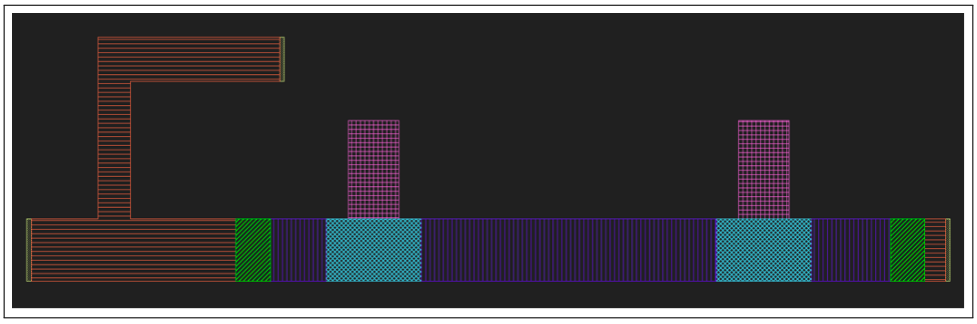
(c) Surface colors are analogous to unique surface attributes.

Figure D.3: Resultant mesh using OpenCASCADE maintains polygon instance attributes.

This allows the LGM to bind data to polygon elements which are then also connected to all triangles that tessellates a specific polygon. Fig. D.3 shows a basic Josephson junction connected to a via, using the built-in engine and after having implemented the necessary features using the OpenCASCADE engine. Note the final mesh discontinuity snaps to the connecting edges of two different polygon objects. Fig. D.3a shows the original merged polygon where the geometric information is lost. Fig. D.3b shows the new merging method where the polygon instance data is contained. Sending this polygon set for meshing results in the output shown in Fig D.3c. The different surface colors indicate the unique labeling of the corresponding polygons.



(a) Circuit layout containing two junctions.



(b) Resultant geometry with metadata connected to each polygon.

Figure D.4: Geometric data filtering using parameters

Gmsh version 4.0.0 implemented a new MSH4 format, added a new mesh partitioning code based on Metis 5 and much more, which has already been implemented by the Pygmsh and Meshio libraries. These changes were detected by the LGM interface which connects to these libraries. Also, a Python API has been added to the native Gmsh library. This API is extremely low-level and still requires a wrapper to be implemented into SPiRA and adds no new functionality to the system that Pygmsh does not already handle. Creating a native wrapper in SPiRA, even with the native Python API, can cause systemic degradation due the wrapper having to be updated when the Python API is updated. For the time being the reasoning between using the native Python API or staying with the Pygmsh wrapper can be left to *Lindy*.

Adding metadata to each polygon object can simplify the geometric construction. This simplification process will be left to the netlist generator, to add extra graph nodes and edges from the data inherit to the specific polygon. As an example the layout in Fig. D.4b contains the polygons of two junction, detected from the layout in Fig. D.4a. The ground connections can be disregarded as the junction polygons contain the necessary data to create ground nodes in the netlist. This metadata is connected to each Geometry object which are added to the cell elementals.

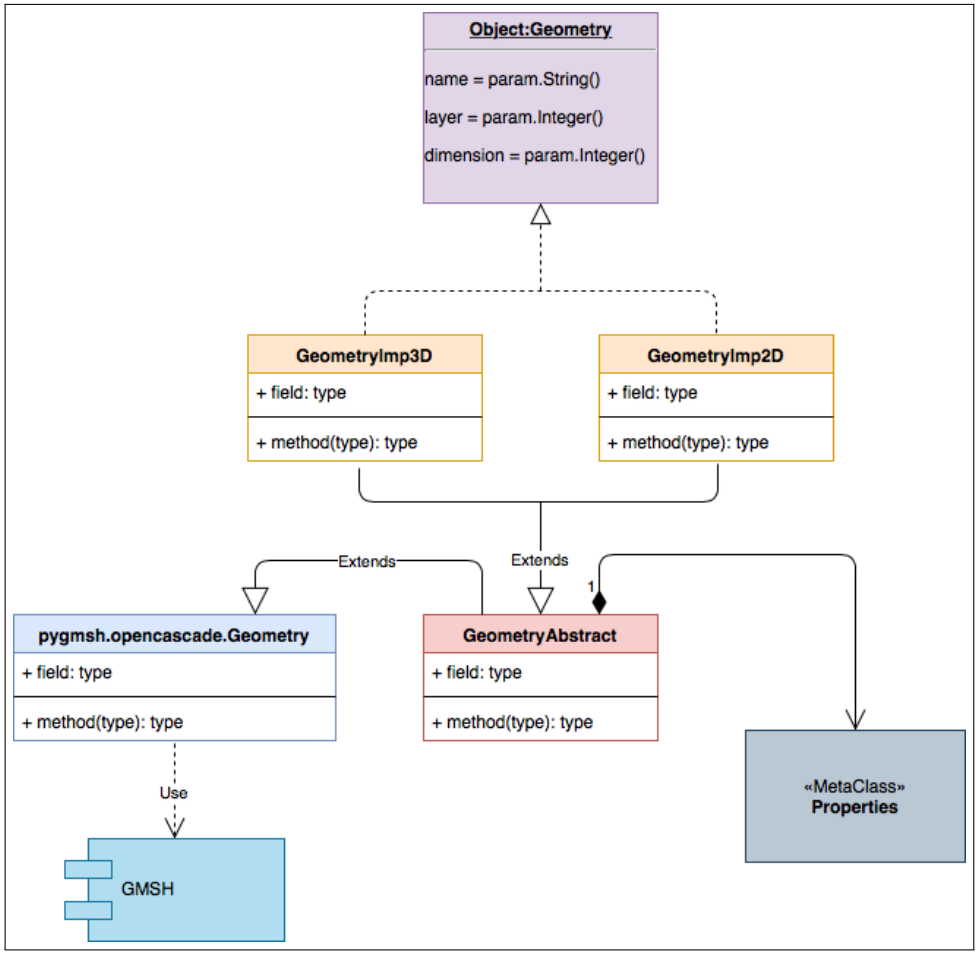


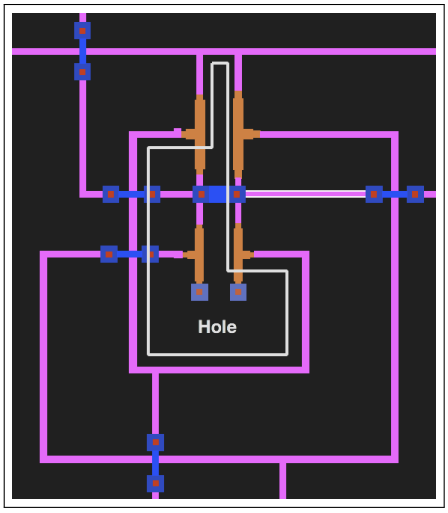
Figure D.5: Logical flow diagram of the Geometry class

Fig. D.5 shows a flow diagram of the `Geometry` class responsible for converting a polygon to a physical geometry. Inherit is both: the two-dimensional, and three-dimensional classes. Depending on the user input a 2D and/or 3D model can be constructed. The `Geometry` class receives polygon elementals from which it creates a Physical Geometry that consists of Gmsh *surfaces*.

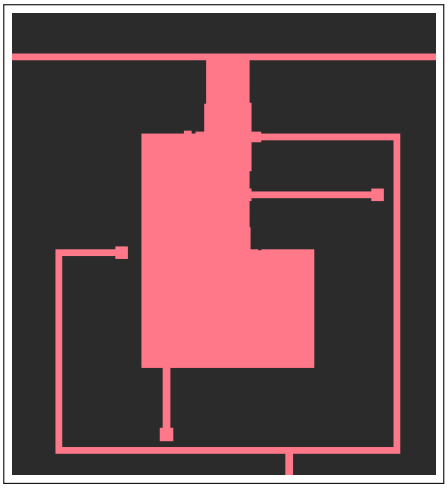
D.3 Negative Polygons Operations

Negative polygons define areas on the layout where isolated layers or metal layers are removed. In superconducting circuits these polygons typically represent moat structures [47]. Negative polygons are used by the Clippers library to represent polygon holes. In some cases when merging a set of polygons with the same layer number the Clippers library generates a hole when the merged result creates a closed loop. Fig. D.6 depicts such a case, where the merging of the conducing layers, Fig. D.6a, results in a closed loop and generates a

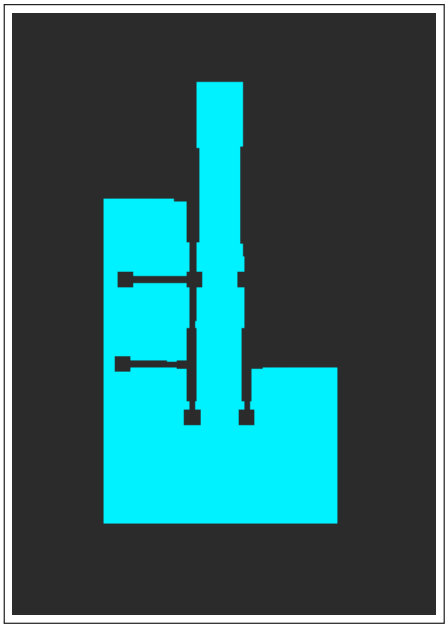
positive polygon, shown in Fig. D.6b, and a corresponding negative polygon, shown in Fig. D.6c.



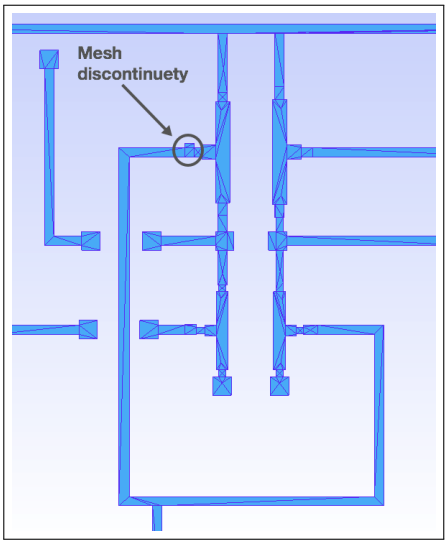
(a) Original layout with the white line showing the hole that has to be detected.



(b) The resultant polygon once all metal layers have been merged.



(c) The hole polygon detected from the original layout.



(d) The result of the meshed structure of the difference between (b) and (c).

Figure D.6: Detecting holes in polygon structures once all layer polygons has been merged.

The LGM takes these two polygons generated by the Clippers library and subtracts them, depicted in Fig. D.6d. The initial implementation to solve this problem was done using the *fast_ boolean* function in the Gdspy library. This

algorithm breaks the polygons by slicing one of its edges as shown in Fig. D.7. However, because physical geometries is a prerequisite for generating a layout netlist, the polygon information regarding the hole structure cannot be filtered out.

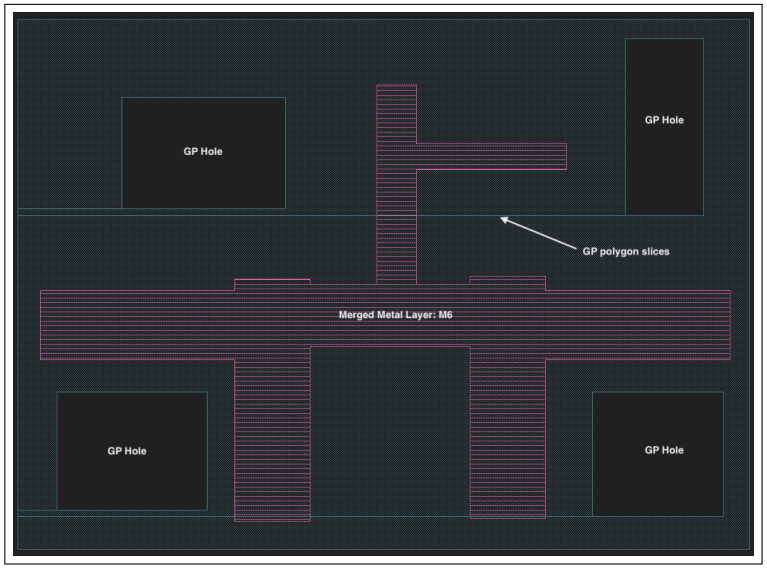


Figure D.7: Ground plane merging with holes/moats using the Gdspy Library.

Subsequently, the implemented method catogorizes polygons into two different types: One containing all the clockwise-oriented (positive) polygons, and the other all the counterclockwise-oriented (negative) polygons. This allows the LGM package to create *hole* objects using the Pygmsh library. These *holes* are then subtracted from the overlapping positive polygons by the Gmsh library.

D.4 Conclusion

Converting simple polygon objects to Physical Geometries without having to radically change the LGK shows the robustness of abstraction in the SPiRA framework. The Geometry class can be extended to convert layout polygons to other geometry formats required by other software packages, such as commercial once like ANSYS. Any one of the libraries used by the LGM can be swapped out due to its hierarchical implementation. Currently, the Clippers library is used for its speed advantages, even though Shapely has more implemented features. However, the Clippers library can be replaced by the Shapely library, and similarly the Gmsh library can be swapped out by the Netgen library [48]. The heuristic that an external library, such as Pygmsh will push updates faster than the development of SPiRA seems to have played an intricate role in making the LGM more robust.

Appendix E

Parameterized Via Devices

This chapter contains parameterized code examples of some of the via devices created for the MITLL fabrication process.

E.1 Standard Contact M6 to R5

The code for a standard C_5R via is shown.

```
1  class ViaC5RS(spira.Device):
2
3    width = param.NumberField(default=RDD.C5R.MIN_SIZE,
4      restriction=RestrictRange(lower=RDD.C5R.MIN_SIZE))
5    height = param.NumberField(default=RDD.C5R.MIN_SIZE,
6      restriction=RestrictRange(lower=RDD.C5R.MIN_SIZE))
7
8    m6_width = param.DataField(fdef_name='create_m6_width')
9    m6_height = param.DataField(fdef_name='create_m6_height')
10
11   r5_width = param.DataField(fdef_name='create_r5_width')
12   r5_height = param.DataField(fdef_name='create_r5_height')
13
14   def create_m6_width(self):
15     return (self.width + 2*RDD.C5R.M6_MIN_SURROUND)
16
17   def create_r5_width(self):
18     return (self.width + 2*RDD.R5.C5R_MIN_SURROUND)
19
20   def create_m6_height(self):
21     return (self.height + 2*RDD.C5R.M6_MIN_SURROUND)
22
23   def create_r5_height(self):
24     return (self.height + 2*RDD.R5.C5R_MIN_SURROUND)
```

```
25
26    def create_contacts(self, elems):
27      elems += pc.Box(ps_layer=RDD.PLAYER.C5R, w=self.width, h=self.height)
28      return elems
29
30    def create_metals(self, elems):
31      elems += pc.Box(alias='M6', ps_layer=RDD.PLAYER.M6,
32        w=self.m6_width, h=self.m6_height)
33      elems += pc.Box(alias='R5', ps_layer=RDD.PLAYER.R5,
34        w=self.r5_width, h=self.r5_height)
35      return elems
```

E.2 Contact M5 to M4

The code for a I_4 via is shown.

```
1  class ViaI4(Via):
2
3    width = param.NumberField(default=RDD.I4.MIN_SIZE,
4      restriction=RestrictRange(lower=RDD.I4.MIN_SIZE))
5    height = param.NumberField(default=RDD.I4.MIN_SIZE,
6      restriction=RestrictRange(lower=RDD.I4.MIN_SIZE))
7
8    m4_width = param.DataField(fdef_name='create_m4_width')
9    m4_height = param.DataField(fdef_name='create_m4_height')
10
11   m5_width = param.DataField(fdef_name='create_m5_width')
12   m5_height = param.DataField(fdef_name='create_m5_height')
13
14   def create_m4_width(self):
15     return (self.width + 2*RDD.M4.I4_MIN_SURROUND)
16
17   def create_m5_width(self):
18     return (self.width + 2*RDD.I4.M5_MIN_SURROUND)
19
20   def create_m4_height(self):
21     return (self.height + 2*RDD.M4.I4_MIN_SURROUND)
22
23   def create_m5_height(self):
24     return (self.height + 2*RDD.I4.M5_MIN_SURROUND)
25
26   def create_structures(self, elems):
27     elems += pc.Box(ps_layer=RDD.PLAYER.I4, w=self.width, h=self.height)
```

```
28      return elems
29
30    def create_routes(self, elems):
31      elems += pc.Box(alias='M4', ps_layer=RDD.PLAYER.M4,
32        w=self.m4_width, h=self.m4_height)
33      elems += pc.Box(alias='M5', ps_layer=RDD.PLAYER.M5,
34        w=self.m5_width, h=self.m5_height)
35      return elems
36
37    def create_ports(self, ports):
38      for p in self.routes['M4'].ports:
39        ports += p.modified_copy(name=p.name, width=self.width)
40      for p in self.routes['M5'].ports:
41        ports += p.modified_copy(name=p.name, width=self.width)
42      return ports
```

E.3 Contact M5 to M6

The code for a I_5 via is shown.

```
1  class ViaI5(Via):
2
3    width = param.NumberField(default=RDD.I5.MIN_SIZE,
4      restriction=RestrictRange(lower=RDD.I5.MIN_SIZE))
5    height = param.NumberField(default=RDD.I5.MIN_SIZE,
6      restriction=RestrictRange(lower=RDD.I5.MIN_SIZE))
7
8    m6_width = param.DataField(fdef_name='create_m6_width')
9    m6_height = param.DataField(fdef_name='create_m6_height')
10
11   m5_width = param.DataField(fdef_name='create_m5_width')
12   m5_height = param.DataField(fdef_name='create_m5_height')
13
14   def create_m6_width(self):
15     return (self.width + 2*RDD.M5.I5_MIN_SURROUND)
16
17   def create_m5_width(self):
18     return (self.width + 2*RDD.I5.M6_MIN_SURROUND)
19
20   def create_m6_height(self):
21     return (self.height + 2*RDD.M5.I5_MIN_SURROUND)
22
23   def create_m5_height(self):
```

```
24      return (self.height + 2*RDD.I5.M6_MIN_SURROUND)
25
26    def create_structures(self, elems):
27      elems += pc.Box(ps_layer=RDD.PLAYER.I5, w=self.width, h=self.height)
28      return elems
29
30    def create_routes(self, elems):
31      elems += pc.Box(alias='M5', ps_layer=RDD.PLAYER.M5,
32        w=self.m6_width, h=self.m6_height)
33      elems += pc.Box(alias='M6', ps_layer=RDD.PLAYER.M6,
34        w=self.m5_width, h=self.m5_height)
35      return elems
36
37    def create_ports(self, ports):
38      for p in self.routes['M5'].ports:
39        ports += p.modified_copy(name=p.name, width=self.width)
40      for p in self.routes['M6'].ports:
41        ports += p.modified_copy(name=p.name, width=self.width)
42      return ports
```

E.4 Contact M6 to M7

The code for a I_6 via is shown.

```
1
2   class ViaI6(Via):
3
4     width = param.NumberField(default=RDD.I6.MIN_SIZE,
5       restriction=RestrictRange(lower=RDD.I6.MIN_SIZE))
6     height = param.NumberField(default=RDD.I6.MIN_SIZE,
7       restriction=RestrictRange(lower=RDD.I6.MIN_SIZE))
8
9     m6_width = param.DataField(fdef_name='create_m6_width')
10    m6_height = param.DataField(fdef_name='create_m6_height')
11
12    m7_width = param.DataField(fdef_name='create_m7_width')
13    m7_height = param.DataField(fdef_name='create_m7_height')
14
15    def create_m6_width(self):
16      return (self.width + 2*RDD.M6.I6_MIN_SURROUND)
17
18    def create_m7_width(self):
19      return (self.width + 2*RDD.I6.M7_MIN_SURROUND)
```

```
20
21    def create_m6_height(self):
22      return (self.height + 2*RDD.M6.I6_MIN_SURROUND)
23
24    def create_m7_height(self):
25      return (self.height + 2*RDD.I6.M7_MIN_SURROUND)
26
27    def create_structures(self, elems):
28      elems += pc.Box(ps_layer=RDD.PLAYER.I6, w=self.width, h=self.height)
29      return elems
30
31    def create_routes(self, elems):
32      elems += pc.Box(alias='M6', ps_layer=RDD.PLAYER.M6,
33        w=self.m6_width, h=self.m6_height)
34      elems += pc.Box(alias='M7', ps_layer=RDD.PLAYER.M7,
35        w=self.m7_width, h=self.m7_height)
36      return elems
37
38    def create_ports(self, ports):
39      for p in self.routes['M6'].ports:
40        ports += p.modified_copy(name=p.name, width=self.width)
41      for p in self.routes['M7'].ports:
42        ports += p.modified_copy(name=p.name, width=self.width)
43      return ports
```

E.5 Junction contact between M6 and M5

The PCell code for a junction connecting metal layers M_5 and M_6 are shown. The junction layer, J_5, in conjunction with layer, C_5J, are used in creating a Josephson junction.

```
1  class JJ(Via):
2
3    width = param.NumberField(
4      default=RDD.J5.MIN_SIZE+2*RDD.J5.C5J_MIN_SURROUND,
5      restriction=RestrictRange(
6        lower=RDD.C5J.MIN_SIZE+2*RDD.J5.C5J_MIN_SURROUND,
7        upper=RDD.J5.MAX_SIZE)
8    )
9
10   jj_radius = param.DataField(fdef_name='create_jj_radius')
11   c5j_radius = param.DataField(fdef_name='create_c5j_radius')
12
```

```
13    m5_width = param.DataField(fdef_name='create_m5_width')
14    m6_width = param.DataField(fdef_name='create_m6_width')
15
16    def create_jj_radius(self):
17      return self.width/2
18
19    def create_c5j_radius(self):
20      return self.jj_radius - RDD.J5.C5J_MIN_SURROUND
21
22    def create_m5_width(self):
23      return 2*(self.jj_radius + RDD.M5.J5_MIN_SURROUND)
24
25    def create_m6_width(self):
26      return 2*(self.c5j_radius + RDD.C5J.M6_MIN_SURROUND)
27
28    def create_structures(self, elems):
29      elems += pc.Circle(
30        ps_layer=RDD.PLAYER.J5,
31        box_size=[2*self.jj_radius, 2*self.jj_radius]
32      )
33      elems += pc.Circle(
34        ps_layer=RDD.PLAYER.C5J,
35        box_size=[2*self.c5j_radius, 2*self.c5j_radius]
36      )
37      return elems
38
39    def create_routes(self, elems):
40      elems += pc.Box(alias='M5',
41        ps_layer=RDD.PLAYER.M5,
42        w=self.m5_width, h=self.m5_width
43      )
44      elems += pc.Box(alias='M6',
45        ps_layer=RDD.PLAYER.M6,
46        w=self.m6_width, h=self.m6_width
47      )
48      return elems
49
50    def create_ports(self, ports):
51      for p in self.routes['M5'].ports:
52        ports += p.modified_copy(name=p.name, width=self.width)
53      for p in self.routes['M6'].ports:
54        ports += p.modified_copy(name=p.name, width=self.width)
55      return ports
```

List of References

[1] 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 Transactions on Applied Superconductivity*, vol. 25, 08 2014.

[2] N. Zhang and D. C. Wunsch II, "Speeding up vlsi layout verification using fuzzy attributed graphs approach," *IEEE Transactions on Fuzzy Systems*, vol. 14, pp. 728–737, Dec 2006.

[3] O. A. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Transactions on Applied Superconductivity*, vol. 21, pp. 760–769, June 2011.

[4] T. V. Filippov, D. Amparo, M. Y. Kamkar, J. Walter, A. F. Kirichenko, O. A. Mukhanov, and I. V. Vernik, "Experimental investigation of ersfq circuit for parallel multibit data transmission," in *2017 16th International Superconductive Electronics Conference (ISEC)*, pp. 1–4, June 2017.

[5] N. Tsuji, Y. Yamanashi, N. Takeuchi, C. Ayala, and N. Yoshikawa, "Design and implementation of scalable register files using adiabatic quantum flux parametron logic," in *2017 16th International Superconductive Electronics Conference (ISEC)*, pp. 1–3, June 2017.

[6] "Iarpa supertools program." `https://www.iarpa.gov/index.php/research-programs/supertools`.

[7] C. J. Fourie, "Digital superconducting electronics design tools-status and roadmap," *IEEE Transactions on Applied Superconductivity*, vol. 28, pp. 1–12, Aug 2018.

[8] C. J. Fourie, C. Shawawreh, I. V. Vernik, and T. V. Filippov, "High-accuracy inductex calibration sets for mit-ll sfq4ee and sfq5ee processes," *IEEE Transactions on Applied Superconductivity*, vol. 27, pp. 1–5, March 2017.

[9] J. A. Delport and C. J. Fourie, "A static timing analysis tool for rsfq and ersfq superconducting digital circuit applications," *IEEE Transactions on Applied Superconductivity*, vol. 28, pp. 1–5, Aug 2018.

[10] K. Jackman and C. J. Fourie, "Flux trapping analysis in superconducting circuits," *IEEE Transactions on Applied Superconductivity*, vol. 27, pp. 1–5, June 2017.

[11] S. K. Tolpygo, "Superconductor digital electronics: Scalability and energy efficiency issues," vol. 42, pp. 463–485, 05 2016.

[12] S. Alassi and B. Winter, "Pycells for an open semiconductor industry," *arXiv preprint arXiv:1607.00859*, 2016.

[13] C. J. Fourie and M. H. Volkmann, "Status of superconductor electronic circuit design software," *IEEE Transactions on Applied Superconductivity*, vol. 23, pp. 1300205–1300205, June 2013.

[14] L. M. Rosenberg, "The evolution of design automation to meet the challanges of vlsi," in *17th Design Automation Conference*, pp. 3–11, June 1980.

[15] E. Lyons, V. Ganti, R. Goldman, V. Melikyan, and H. Mahmoodi, "Full-custom design project for digital vlsi and ic design courses using synopsys generic 90nm cmos library," in *2009 IEEE International Conference on Microelectronic Systems Education*, pp. 45–48, July 2009.

[16] O. A. Marvik, "A method for ic layout verification," in *21st Design Automation Conference Proceedings*, pp. 708–709, June 1984.

[17] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm," in *30th ACM/IEEE Design Automation Conference*, pp. 31–37, June 1993.

[18] R. van Staden, K. Jackman, C. J. Fourie, and P. Febvre, "Influence of the superconducting ground plane on the performance of rsfq cells," *IEEE Transactions on Applied Superconductivity*, vol. 27, pp. 1–4, June 2017.

[19] K. Gaj, Q. P. Herr, V. Adler, A. Krasniewski, E. G. Friedman, and M. J. Feldman, "Tools for the computer-aided design of multigigahertz superconducting digital circuits," *IEEE Transactions on Applied Superconductivity*, vol. 9, pp. 18–38, March 1999.

[20] R. M. C. Roberts and C. J. Fourie, "Layout-versus-schematic verification for superconductive integrated circuits," *IEEE Transactions on Applied Superconductivity*, vol. 25, pp. 1–5, June 2015.

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

[22] "Xictools: Xic graphical editor, wrspice circuit simulator, and accessories for electronic design.." `https://github.com/wrcad/xictools`.

[23] K. Langner and J. Scheible, "Formal verification of a transistor pcell," in *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pp. 205–208, June 2017.

[24] G. Bouteloup, O. Franiatte, M. Gracietti, Y. Imbs, J. Jaklic, B. Lindberg, J. Lopez, and D. Signoret, "Stmicroelectronics package design rules check coverage enhancement with cadence ravel," in *2013 Eurpoean Microelectronics Packaging Conference (EMPC)*, pp. 1–7, Sept 2013.

[25] Gdsii, "Gdsii manual." `http://bitsavers.informatik.uni-stuttgart.de/pdf/calma/GDS_II_Stream_Format_Manual_6.0_Feb87.pdf`, Feb 1987.

[26] N. Taleb, *The Black Swan: The Impact of the Highly Improbable*. Incerto, Random House Publishing Group, 2007.

[27] W. W. . Dai, "Hierarchical placement and floorplanning in bear," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 1335–1349, Dec 1989.

[28] P. B. Wu, R. J. Mack, and R. E. Massara, "A multi-level netlist partitioning approach to hierarchical layout design of analog ics," in *Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems (Cat.No.CH37144)*, vol. 1, pp. 124–127 vol.1, Aug 2000.

[29] F. Luellau, T. Hoepken, and E. Barke, "A technology independent block extraction algorithm," in *21st Design Automation Conference Proceedings*, pp. 610–615, June 1984.

[30] W.-H. Tsai and K.-S. Fu, "Error-correcting isomorphisms of attributed relational graphs for pattern analysis," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 9(12), pp. 757 – 768, 01 1980.

[31] R. Razdan, "Hcnc: High capacity netlist compare," in *Proceedings of IEEE Custom Integrated Circuits Conference - CICC '93*, pp. 17.6.1–17.6.5, May 1993.

[32] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration*, vol. 19, no. 1, pp. 1 – 81, 1995.

[33] R. M. C. Roberts and C. J. Fourie, "Layout-to-schematic as a step towards layout-versus-schematic verification of sfq integrated circuit layouts," in *2013 Africon*, pp. 1–5, Sept 2013.

[34] K. Gaj, Q. P. Herr, V. Adler, D. K. Brock, E. G. Friedman, and M. J. Feldman, "Toward a systematic design methodology for large multigigahertz rapid single flux quantum circuits," *IEEE Transactions on Applied Superconductivity*, vol. 9, pp. 4591–4606, Sept 1999.

[35] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, Apr. 1975.

[36] X. Li, H. Qin, and R. Lang, "An algorithm for identifying the recurring subcircuits," 2006.

[37] M. S. Abadir and J. Ferguson, "An improved layout verification algorithm (lava)," in *Proceedings of the European Design Automation Conference, 1990., EDAC.*, pp. 391–395, March 1990.

[38] C. Geuzaine and J.-F. Remacle, *Gmsh Reference Manual*. http://www.geuz.org/gmsh, 1.12 ed., Aug. 2003.

[39] "Python interface for gmsh.." `https://github.com/nschloe/pygmsh`.

[40] "Manipulation and analysis of geometric objects.." `https://github.com/Toblerity/Shapely`.

[41] S. . Wu and M. R. G. Marquez, "A non-self-intersection douglas-peucker algorithm," in *16th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2003)*, pp. 60–66, Oct 2003.

[42] B. R. Vatti, "A generic solution to polygon clipping," *Commun. ACM*, vol. 35, pp. 56–63, July 1992.

[43] "Cython wrapper for the c++ translation of the angus johnson's clipper library.." `https://github.com/fonttools/pyclipper`.

[44] J. Ghosh, S. Mukhopadhyay, A. Patra, B. Culpepper, and T. Mei, "Estimation of dc performance of a lateral power mosfet using distributed cell model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 1452–1456, Sept 2012.

[45] S. Tadjiky and R. H. Jansen, "From layout to schematic using pattern recognition in microwave computer aided design," in *1997 IEEE MTT-S International Microwave Symposium Digest*, vol. 2, pp. 1029–1033 vol.2, June 1997.

[46] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities," *Int. J. Numer. Meth. Engng.*, vol. 79, pp. 1309–1331, Sept. 2009.

[47] V. K. Semenov and M. M. Khapaev, "How moats protect superconductor films from flux trapping," *IEEE Transactions on Applied Superconductivity*, vol. 26, pp. 1–10, April 2016.

[48] "Netgen/ngsolve is a high performance multi-physics finite element software.." `https://github.com/NGSolve/netgen`.