The Design and Testing of a Superconducting Programmable Gate Array

by

Hein van Heerden

Thesis presented at the University of Stellenbosch in partial fulfilment of the requirements for the degree of

Master of Science in Electronic Engineering

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

Study leader: Dr. C.J. Fourie

December 2005

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: ....

H. van Heerden

Date: ....

Abstract

This thesis investigates to the design, analysis and testing of a Superconducting Programmable Gate Array (SPGA). The objective was to apply existing programmable logic concepts to RSFQ circuits and in the process develop a working prototype of a superconducting programmable logic device. Various programmable logic technologies and architectures were examined and compared to find the best solution. Using Rapid Single Flux Quantum (RSFQ) circuits as building blocks, a complete functional design was assembled incorporating a routing architecture and logic blocks. The Large-Scale Integrated circuit (LSI) layout of the final chip is presented and discussed followed by a discussion on testing. This thesis demonstrates the successful implementation of a fully functional reprogrammable logic device using RSFQ circuitry.

Opsomming

Hierdie tesis handel oor die ontwerp, analise en toets van ʼn SPGA (Superconducting Programmable Gate Array). Die doel is om huidige programeerbare logika konsepte aan te pas en in die proses ʼn werkende prototipe te ontwikkel wat beskryf kan word as ʼn supergeleier programeerbare logiese toestel wat se werking soortgelyk is aan dié van FPGAs. ʼn Verskeidenheid van programeerbare logika tegnologieë en argitekture is ondersoek en met mekaar vergelyk om die beste oplossing te vind. Met RSFQ (Rapid Single Flux Quantum) stroombane as boublokke is ʼn hele funksionele ontwerp aanmekaar gesit wat beide ʼn verspreidingsargitektuur en logieseblokke inkorporeer. Die grootskaalse geïntegreerde stroombaan uitleg van die finale vlokkie word bespreek en voorgelê. Daarna volg ʼn bespeking oor toetsing van die ontwerp. Hierdie tesis demonstreer die suksesvolle implementering van ʼn werkende herprogrameerbare logika toestel bestaande uit RSFQ stroombane.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations who have contributed to making this work possible:

- Steve R. Whiteley for his invaluable simulation tools, which made this project feasible.
- Hypres Inc. for providing the fabrication platform on which the physical implementation of this project is based. Especially Dr. S. Tolpygo, head of fabrication, for his help and direction.
- The National Research Foundation of South Africa (and Prof. D.B. Davidson as grantholder) for providing some financial assistance during my research.
- My fellow students and good friends, Hennie de Villiers and Wynand van Staden for providing inspiration and support during difficult times.
- Prof. Willem J. Perold for giving me the inspiration and opportunity to pursue my quest for further enrichment in the field of engineering. Also, for his eternal optimism and enthusiasm.
- Lastly, but most of all, Dr. Coenrad J. Fourie as my study leader for the countless hours of advice and guidance and also for providing me with a good background and basis on which to build this project.

Dedications

This thesis is dedicated to my mother
for her support, encouragement and eternal love.
Sadly, she passed away before the completion of this project.
She will be missed.

Contents

List of Figures

List of Tables

Nomenclature

2D	Two-Dimensional
3D	Three-Dimensional
AC	Alternating Current
ADC	Analogue to Digital Converter
CAD	Computer-Aided Design
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
COSL	Complementary Output Switching Logic
DAC	Digital to Analogue Converter
dc/DC	Direct Current
DCRL	DC-Resetable Latch
Dff	D-type Flip-Flop
DIV	Divider
DRC	Design Rule Checker
DRO	Destructive Read-Out register
DUT	Device Under Test
EEPROM	Electrically Erasable Programmable Read-Only Memory
EM	Electromagnetic
EPROM	Erasable Programmable Read-Only Memory
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HEMT	High-Electron-Mobility Transistors
HUFFLE	Hybrid Unlatching Flip-Flop Logic Element
GHz	Gigahertz = 10^9 [Hz]
IC	Intergrated Circuit
IO	Input-Output

JJ	Josephson Junction
JTL	Josephson Transmission Line
K	Kelvin
LAB	Logic Array Block
LASI	LAyout System for Individuals
LM	Logic Module
LUT	Lookup Table
MSL	Microstrip Line
MOS	Metal-Oxide Semiconductor
MUX	Multiplexer
μ	micro = 10^{-6}
n	nano = 10^{-9}
Ω	Ohm
p	pico = 10^{-12}
PGA	Programmable Gate Array
PLD	Programmable Logic Device
RAM	Random Access Memory
RSFQ	Rapid Single Flux Quantum
SFQ	Single Flux Quantum
SPICE	Simulation Program with Integrated Circuit Emphasis
SPGA	Superconducting Programmable Gate Array
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integration

Chapter 1

Introduction

' "Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop." '

[Lewis Carrol - Alice's Adventures in Wonderland (1865)]

1.1 Semiconductor programmable logic

Semiconductor 'full-custom' chips (such as microprocessors) are realized using specific tailoring of each part of a Very Large Scale Integration (VLSI) circuit to meet its requirements. This technique demands extensive manufacturing effort and may take several months to complete. Furthermore, the process has high initial setup costs and requires large production volumes to avoid unrealistic end-user prices.

The intense rivalry in the electronics industry has sparked the need for rapid prototyping and the shortest possible time-to-market solutions. Therefore, reduced development and production time is essential.

Programmable logic has become increasingly popular due to its low cost and prototyping time. More specifically, Field-Programmable Gate Arrays (FPGAs) have emerged as the ultimate solution to these time-to-market and financial risk problems because they provide a means where the end-user can directly configure any logic structure without the need for specialized Integrated Circuit (IC) fabrication.

1.2 Superconducting logic

Superconducting electronics, but more specifically Single Flux Quantum (SFQ) logic have demonstrated speeds up to 770GHz [2]. Furthermore, Rapid Single Flux Quantum (RSFQ) circuits consume very little operational power due to their low bias voltage (a mere 2.6mV). Additionally, superconducting microstrip lines allow ballistic transfer of pulses over arbitrary distances with negilible attenuation and dispersion. This means that integration and chip packaging can be dense and the total performance of a system using superconducting Large Scale Integration (LSI) can rival (and even surpass) that of a system using semiconductor LSI [3].

1.3 SPGA

With the recent evolution of superconducting electronics to medium- and large scale levels, it has become feasible to implement superconducting reprogrammable logic in the same manner as its semiconductor counterpart. The same cost and time related advantages can be gained in the superconducting industry, as was for the semiconductor industry, with added speed and power benefits related to superconducting logic.

The goal of this thesis is to design and test a prototype of the first superconducting programmable logic device (similar to semiconductor programmable gate arrays), namely the Superconducting Field-Programmable Gate Array (SPGA).

1.4 Summary of thesis

Chapter 2 gives a more detailed background and history of the SPGA and followed by a short description of RSFQ logic and its basic operation. In Chapter 3 a study is undertaken to compare existing types of programmable logic in an effort to find the best solution to a possible superconducting counterpart. Various topics relating to programmable logic are also discussed. Next, in Chapter 4, the RSFQ circuit design is discussed by describing most of the RSFQ circuits and blocks that constitute the SPGA. The simulations included in this chapter develop and illustrate concepts used during the design of this project. In Chapter 5 the concept of IC layout is discussed. The chapter includes a brief discussion on a few of the RSFQ circuit layouts and blocks, and also elaborates on the task of error checking and other tasks relating to the layout of the final chip. Finally, issues regarding the physical testing of the SPGA are mentioned and possible solutions proposed, followed by a thorough testing procedure including examples.

'Don't panic.'

[Douglas Adams - The Hitch Hiker's Guide to the Galaxy (1979)]

Chapter 2

SPGA background and RSFQ basics

'Crafty men contemn studies; simple men admire them; and wise men use them.'

[Francis Bacon - Of Studies (1625)]

THIS chapter gives some background information on the SPGA project and also provides a short primer on RSFQ electronics.

The section on the SPGA provides some history and precedings leading up to the original concept of superconducting programmable logic.

The section relating to RSFQ provides a short description on how basic RSFQ circuits work. It is not meant as a complete tutorial on all aspects of RSFQ but rather a short orientation in pulse-based logic. More detailed explanations can be found in the references given throughout this section.

2.1 SPGA background

The original SPGA concept was introduced by Fourie [4] and provided a platform from which a fully working prototype could be designed and built. The basic idea is to construct a superconducting electronic circuit that can be reused for multiple purposes, which in effect means that the circuit has to be reprogrammable. The semiconductor industry paved the way for reprogrammable circuits some time ago, and the technology is fairly common these days and is generally known as *programmable logic*. Superconducting programmable logic can be made by borrowing some concepts used in semiconductor programmble logic and adapting the circuitry.

Components needed to add reprogrammable functionality to RSFQ circuits such as, a DC-Resetable latch, a Current-Set switch and the HUFFLE were discussed by Fourie [4]. Further information is given on developing routing structures and switch blocks as well as conceptually developing a lookup table and address decoder.

The name Superconducting Programmable Gate Array (SPGA) derives from the fact that many of its concepts were taken from *Field-Programmable Gate Arrays* (FPGAs).

2.2 RSFQ basics

Much of the information given here originates from the seminal paper by Likharev and Semenov on RSFQ [5].

In RSFQ logic circuits, binary information is presented not by discrete voltage levels (as in all semiconductor transistor logic), but by very short (picosecond) voltage pulses *V(t)* of a quantitized area:

$$\int V(t)\,dt = \Phi_0 \equiv \frac{h}{2e} \simeq 2.07\ \text{mV} \times \text{ps} \tag{2.2.1}$$

An essence of this idea is that these *single flux quantum* (SFQ) pulses can be quite naturally generated, reproduced, amplified, stored, and processed by elementary circuits comprising of *overdamped* Josephson junctions (brought about by the Josephson effect) [6] [7] [8] [9].

Figure 2.1a shows the simplest logic gate which can be used to demonstrate the elementary circuit operation of a Josephson junction. The junction is biased with a constant current I_b which is slightly less that the junction's critical current I_c. The junction is damped with a resistor R_d so that the overdamped characteristics of the junction dominate. The I-V curve of an overdamped junction is presented in Figure 2.1b. The junction drives a load R_{load} and produces an output current I_0 and output voltage V_0.

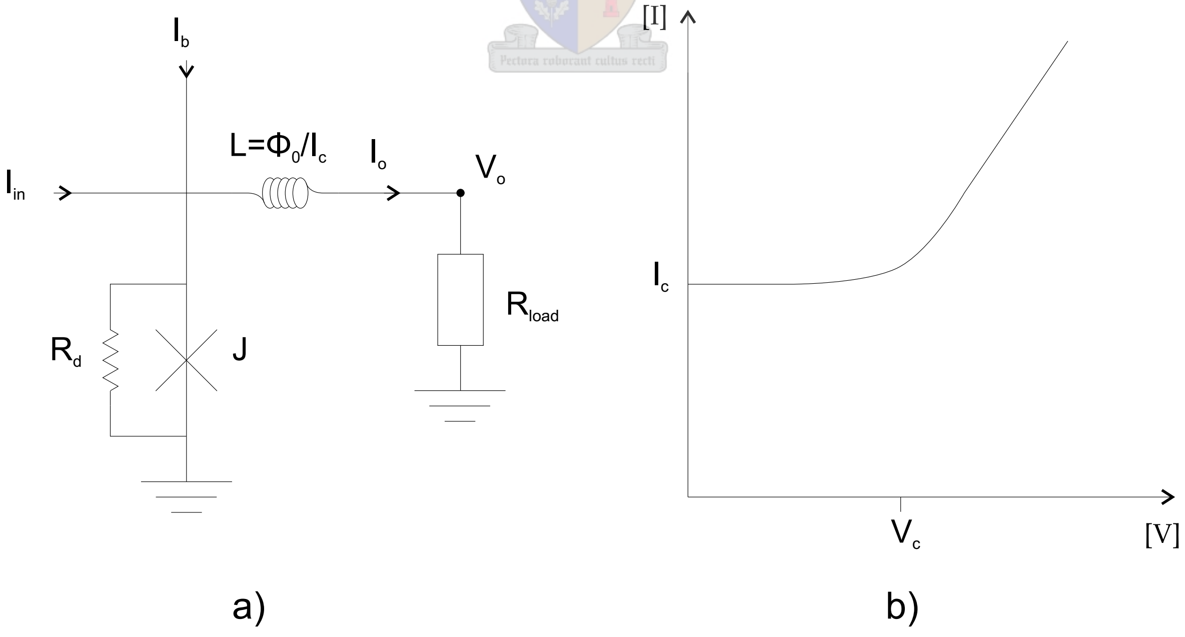

Figure 2.1: a) Equivalent circuit of a resistively shunted junction; b) I-V curve of overdamped junction

Initially the junction is in its superconducting state (V = 0). An arriving signal current I_{in} drives the total junction current beyond I_c, and induces switching to the resistive state '1' with $V \neq 0$, so that a considerable part, I_0, of the current is steered into the load, R_{load}. This is the '0' $\rightarrow$ '1' switching process, which can be very fast (in the order of a few picoseconds). For the overdamped case, the junction is reset to its superconducting state very quickly. The resulting output is presented in Figure 2.2, which shows the shape of the pulse.

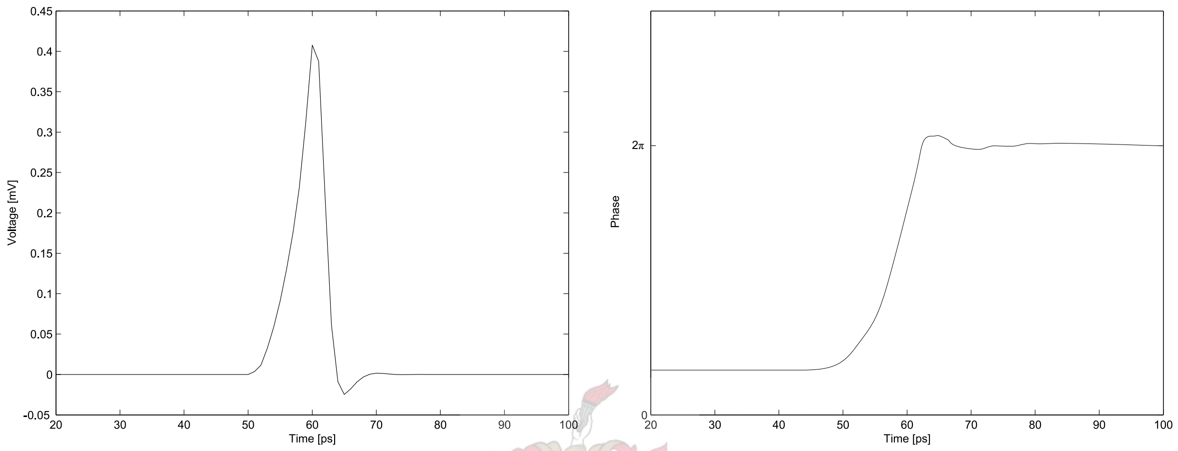

Figure 2.2: Resulting output of an excited junction: Pulse voltage shape and phase

Calculations [10] show that if the dc bias current I_b is close enough to the junction's critical current I_c, an SFQ pulse can be triggered by a similar pulse, with either the nominal or somewhat smaller amplitude. This means that the circuit in Figure 2.1a can *reproduce* SFQ pulses, bringing their areas to the nominal value specified by (2.2.1) and, if necessary, provide voltage gain. On the other hand, if the input signal is too weak (for instance, it represents 'noise' due to parasitic crosstalk between the signal transfer lines) it is not reproduced by the circuit, so that it also serves as a noise discriminator.

Two important remarks should be made here. Firstly, the load need not necessarily be an ohmic resistor; more typically, a similar junction serves as a load in RSFQ circuits. Secondly, these junctions need not be close to each other; they can be connected by an appropriate superconducting microstrip line.

Figure 2.3 shows another key circuit comprising several junctions connected in parallel with relatively low inductance $L \sim \Phi_0 / I_c$ between them. Let J1 be triggered by an incoming pulse from the left. Calculations show that the resulting SFQ pulse developed across J1 will trigger J2 to produce a pulse, and this process will continue until the pulse is reproduced at the right edge of the array. While the pulse propagates, a small time delay is introduced by each junction.

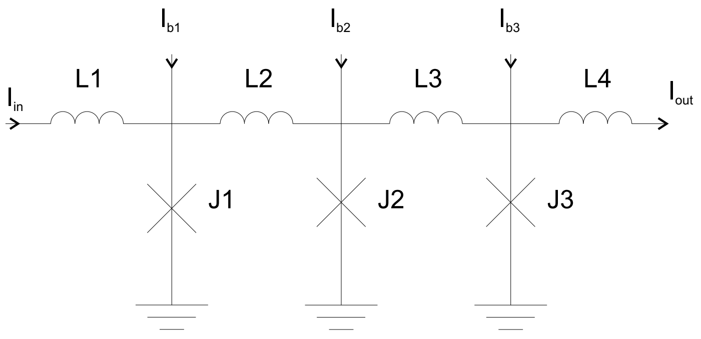

Figure 2.3: Circuit schematic of a junction array

One important concept of pulse-based logic is the definition of logic states. Logic states are represented by the presence or absence of a pulse during a given time period. If a pulse appears at a measurement point in a circuit during a defined time frame, it is regarded as a logical '1'. Whereas, if no pulse appears it is regarded as a logical '0'. Figure 2.4 illustrates the concept more clearly, showing a pulse between times t_0 and t_1 which represents a logical '1'. In contrast, between times t_1 and t_2 there is no pulse and represents a logical '0'. A pulse does not to be centered in the time period, as long as the area underneath the pulse (coinciding with one time frame) equates to one fluxon (as defined in (2.2.1)).

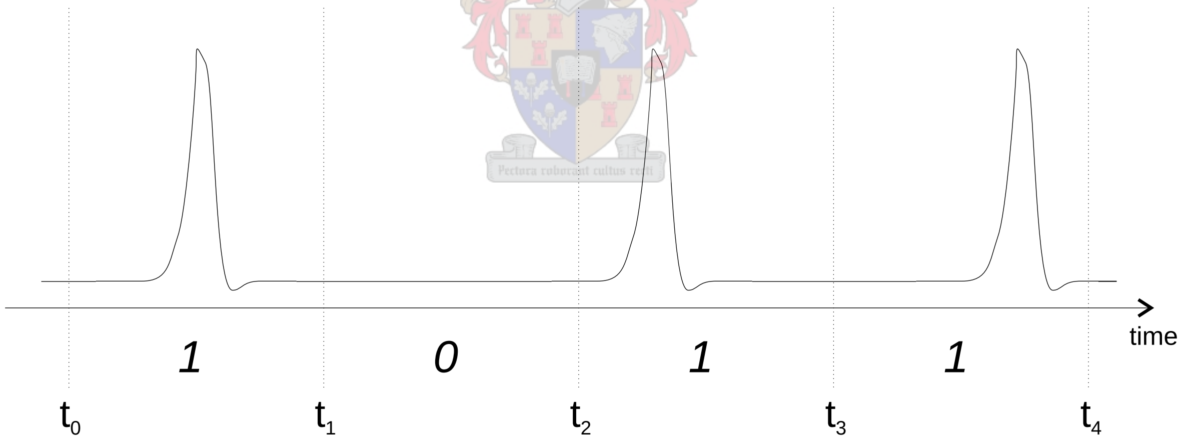

Figure 2.4: Illustration of pulse-based logic definitions

Chapter 3

Programmable logic

'Logic is a systematic method of coming to the wrong conclusion with confidence.'

[Anonymous]

PROGRAMMABLE logic generally pertains to any type of circuit that can be configured or changed by the user to implement a logic design [11]. Of special interest are programmable logic devices that refer to integrated circuits that can be programmed in the field and are so called, *field-programmable logic devices*.

This chapter covers the various types of programmable logic devices and architectures as well as the programming technology used. The aim is to compare the advantages and disadvantages of all the types and find a suitable candidate (or combination) that can be employed to best realize our Superconducting Programmable Gate Array.

3.1 Field-programmable gate array

A *Field-programmable gate array* (FPGA) consists of an array of uncommitted elements that can be interconnected in a general way where the interconnections between elements are user-programmable [12].

3.1.1 Architectures

Several companies have introduced a number of different types of FPGAs. Four basic categories can be identified [12] [11]:

1. symmetrical array
2. row-based
3. hierarchical PLD
4. sea-of-gates (or fine-grain)

Figure 3.1 shows a diagram of each architecture.

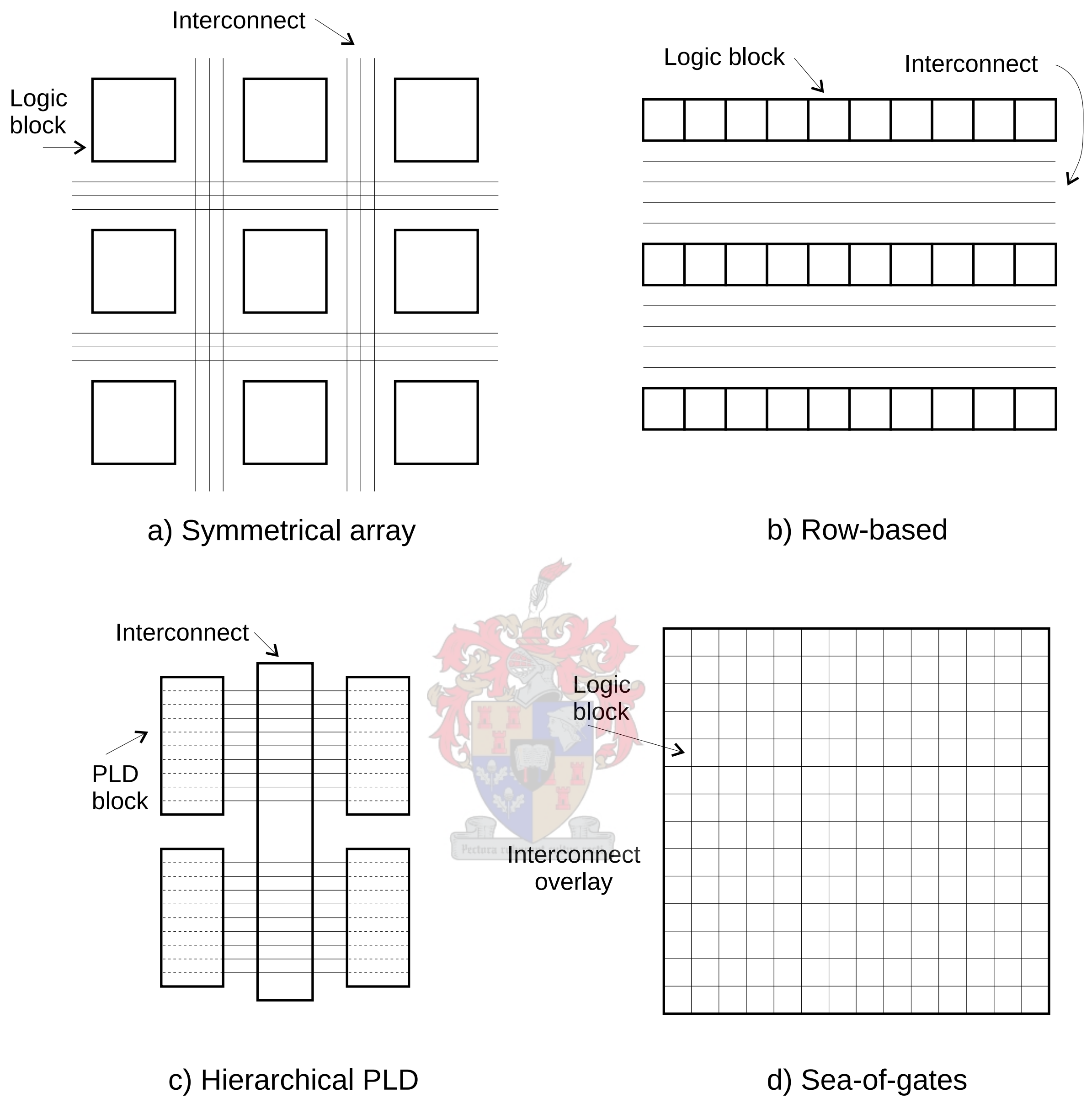

Figure 3.1: FPGA architectures

3.1.1.1 Symmetrical array

Symmetrical arrays contain memory cells that control the logic. These memory cells are more often than not SRAM-based. In most common architectures the RAM cells are used in the logic cells, and to control data flow in the routing channels. A logic cell usually contains a set of RAM cells in a look-up table (LUT) configuration has the advantage of high functionality. The draw-backs of using RAM cells are its size and volatility.

3.1.1.2 Row-based

Rows of logic cells are located parallel to the routing channels. This architecture inherits the advantages of mask programmable gate arrays (i.e. speed and ability to implement large circuits) and the flexibility of user programmability. The routing channels contain predefined wiring segments of various lengths that are connected using antifuses. The one-time programmability of antifuses is a major disadvantage. This architecture has to deal with the problem of estimating the number of tracks in the routing channel as well as the length that the wiring segments in the tracks should be.

3.1.1.3 Hierarchical PLD

Altera FPGAs use a 2-level hierarchical grouping of logic blocks. The first level of hierarchy is called a LAB (Logic Array Block). Each LAB contains two blocks, an Array of macrocells and an Expander product term array. Each macrocell comprises three wide AND gates that feed an OR gate connected to a XOR gate, and a flip-flop. Each expander product terms block consists of a number of p-terms that are inverted and fed back into the macrocells, and to itself. The second level of hierarchy provides connections among the LABs, which is accomplished through a number of long wire segments that pass adjacent to each LAB. This design permits the implementation of very wide logic functions but takes up a lot of space.

3.1.1.4 Sea-of-gates

This fine-grain architecture allows direct connections between the neighboring cells, enabling users to combine cells to form compact logic functions. The advantages are the short routing distances which result in high speed, and efficient resource allocation. The disadvantage is that the routing switches must be extremely small and it lends itself to antifuse switches, which are one-time programmable.

3.1.2 Programming technologies

The word '*switch*' refers to the entities that allow programmable connections between wire segments. A more precise term for such an entity is *programming element*. Since there are a number of different ways of implementing a programming element, it has become customary to speak about *programming technology* that is used to implement these elements.

Programming technologies that are used in commercial products include: static RAM cells, anti-fuses, EPROM transistors and EEPROM transistors [12] [11].

3.1.2.1 Static RAM

Programmable connections used in these devices are *multiplexers*, *transmission gates* or *pass-transistors* that are controlled by SRAM cells. Figure 3.2 contains diagrams of the different technologies.

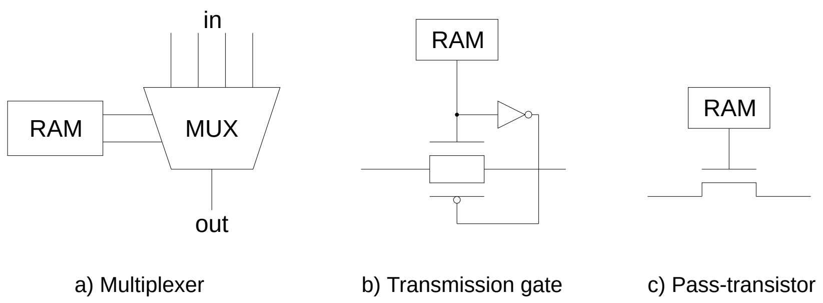

Figure 3.2: Programming technology in conjunction with SRAM

In the case of the **pass-transistor**, the RAM cell controls whether the pass-gates are on or off. When off, the pass-gates presents a very high resistance between the two wires, effectively disconnecting the two terminals. When the gate is on, it forms a relatively low resistance connection between the two wires.

For the **multiplexer** approach, the RAM cells control which one of the multiplexer's inputs is connected to its output.

In a FPGA that uses the SRAM programming technology, the logic blocks may be interconnected using a combination of pass-gates and multiplexers. Since static RAM is volatile, these FPGA's must be configured each time power is applied to the chip.

The RAM cell bits may be loaded into the FPGA either through a serial arrangement or each RAM cell may be addressed as an element in an array.

One disadvantage is that RAM cells require relatively large chip area while an advantage of their use is quick in-circuit programming

3.1.2.2 Antifuse

An Antifuse normally resides in a high-impedance state but can be fused (or melted) into a permanent low-impedance state on application of a high voltage. To program antifuses requires extra on-chip circuitry to deliver the high voltage. The main advantages of the antifuse is its extremely small footprint, but the drawback is the permanent nature of the connection. Antifuse circuits are not reprogrammable but one-time programmable.

3.1.2.3 EPROM and EEPROM

Erasable Programmable Read Only Memory (EPROM) elements are like MOS transistors but comprise of two gates. The floating gate (not connected to the select gate) governs the behavior of the transistor by the accumulated charge under the gate. When no charge is present the transistor operates in the normal fashion, but when a large programming current flows between the source and drain a charge is deposited under the floating gate, turning off the transistor. The charge can be removed by exposing the gate to ultraviolet light.

One disadvantage is that the pull-up resistor consumes static power while it has the advantage is that it is both re-programmable and non-volatile. However, unlike static RAM, EPROM transistors cannot be re-programmed in-circuit.

Electrically erasable programmable read only memory (EEPROM) transistors can be re-programmed in-circuit although they take up double the chip space and require multiple voltage sources for reprogramming.

3.1.3 Implementation

The afore mentioned technologies are used in commercial products. This section presents basic implementations by several companies.

3.1.3.1 Xilinx

The general architecture is shown in figure 3.3. It consists of a 2-dimensional array of programmable blocks, called Configurable Logic Blocks (CLB's), with horizontal routing channels between rows of blocks and vertical routing channels between columns. Programmable resources are controlled by static RAM cells.

XC2000 The XC2000 CLB, shown in Figure 3.4, consists of a four-input look-up table and a D flip-flop. The look-up table can generate any variable of up to four function or any two functions of three variables. Both of the CLB outputs can be combinatorial, or one output can be registered.

The routing architecture shown in Figure 3.5 utilizes three types of routing resources, namely direct interconnect, general purpose interconnect and long lines. *SM* is a Switch Matrix while *CLB* is a Configurable Logic Block.

The direct interconnect connects the output of a CLB to inputs of other CLBs to its right, top and bottom. For connections that span more than one CLB, the general purpose interconnect provides horizontal and vertical wiring segments. Each wiring segment spans only the length or width of one CLB, but longer wires can be formed because each switch matrix holds a number of routing switches that can interconnect the wiring segments on its four sides. Note that a connection routed with a general purpose interconnect will incur

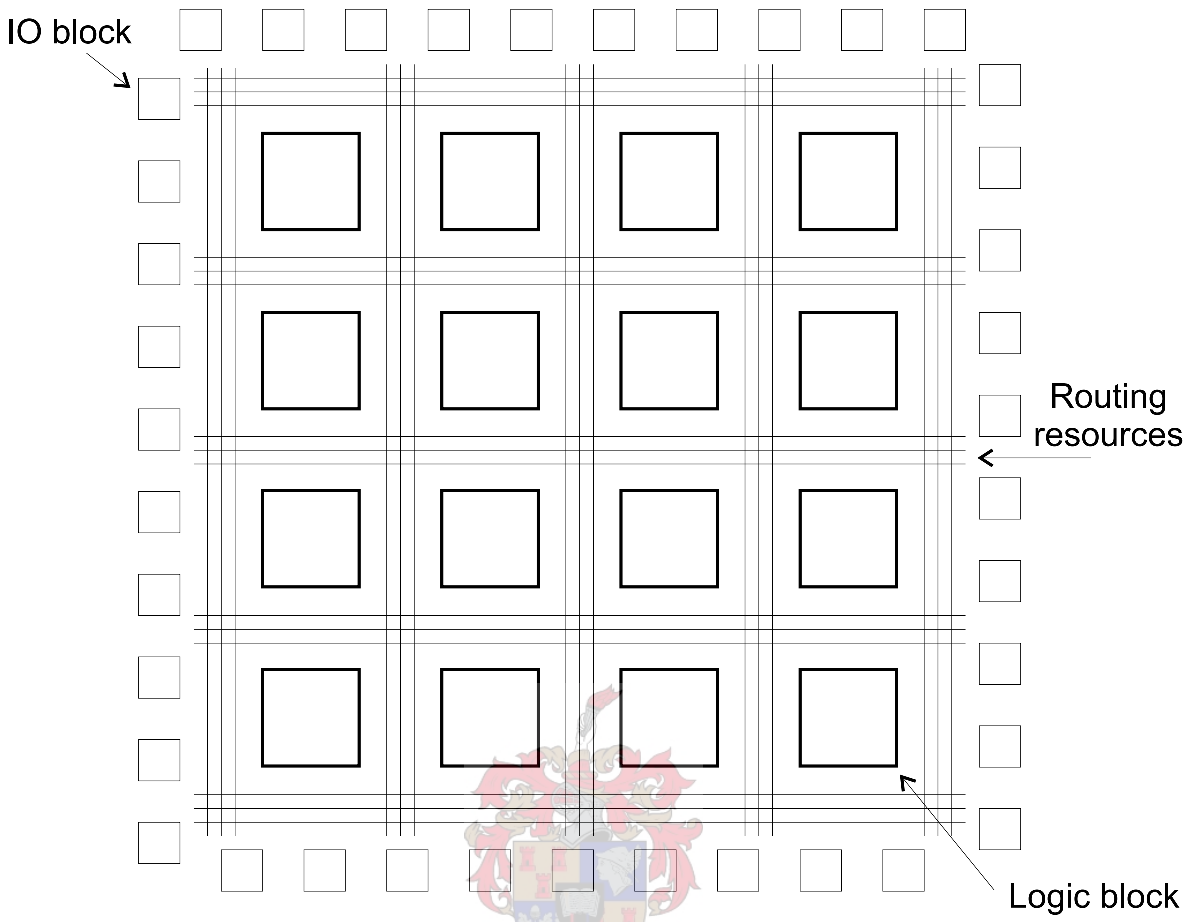

Figure 3.3: General architecture of Xilinx FPGAs

significant routing delays because it must pass through a routing switch at each switch matrix. Signals that are required to reach several CLBs with low skew can use long lines, which traverse at most one routing switch to span the entire length of the FPGA chip.

3.1.3.2 Actel

The basic architecture is similar to that of the row-based architecture shown in Figure 3.1 and consists of rows of programmable blocks, called Logic Modules (LMs), with horizontal routing channels between the rows. Each routing switch in these FPGAs is implemented by an anti-fuse.

Act-1 The Act-1 LM, shown in Figure 3.6, illustrates a very different approach. While Xilinx utilizes a large, complex logic block, Actel advocates a small, simple logic module. Research has shown [13] [14] that both these approaches have their merits, and the best choice for a programming block depends on the speed performance and area requirements of the routing architecture.

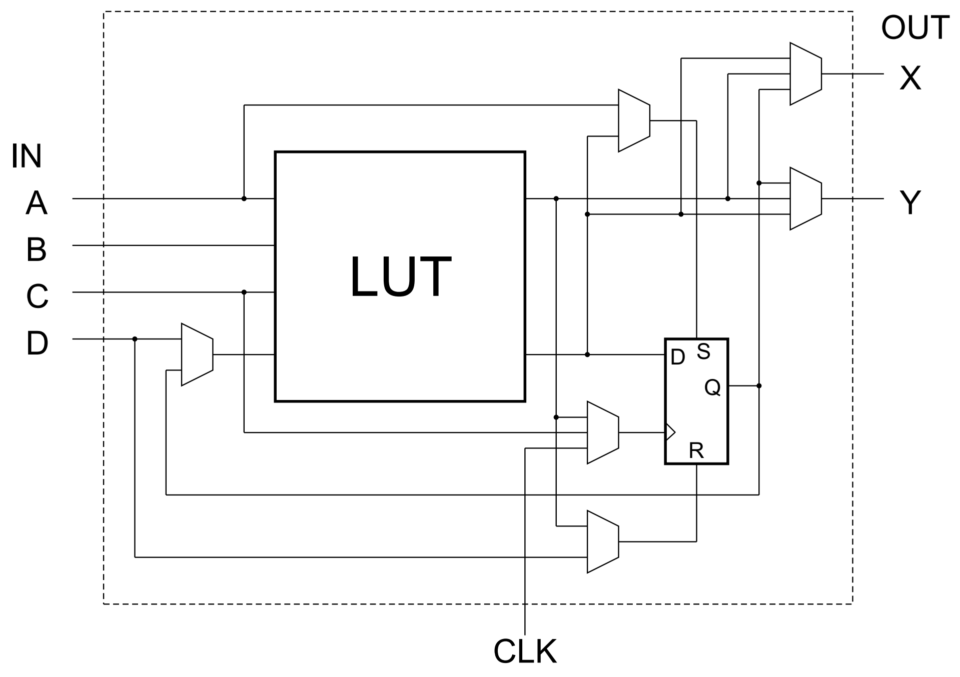

Figure 3.4: Logic block in XC2000

The Act-1 LM is based on a configuration of multiplexers, which can implement any function of two variables, most functions of three variables and some of four variable up to a total of 702 logic functions.

Illustrated in Figure 3.7 is the routing architecture used in the Act-1 [1]. The Act-1 employs four distinct types of routing resources, namely nput segments, output segments, clock tracks and wiring segments. Four input segments connect to the wiring segments above and below the LM. An output segment connects the LM output to several channels, both above and below the module. The wiring segmens consist of straight metal lines of various lengths that can be connected together through anti-fuses to form longer lines. Clock tracks are special low-delay lines that are used for signals that must reach many LM's with minimum skew.

3.1.3.3 Altera

Altera FPGAs utilize hierarchical grouping of programmable logic devices. The architecture is unnecessarily complex for the purpose of this project. This type of structure may be more suited in the future when the superconducting IC process can produce smaller elements and more space is available on a die.

[1]Only the routing resources of the middle LM are shown

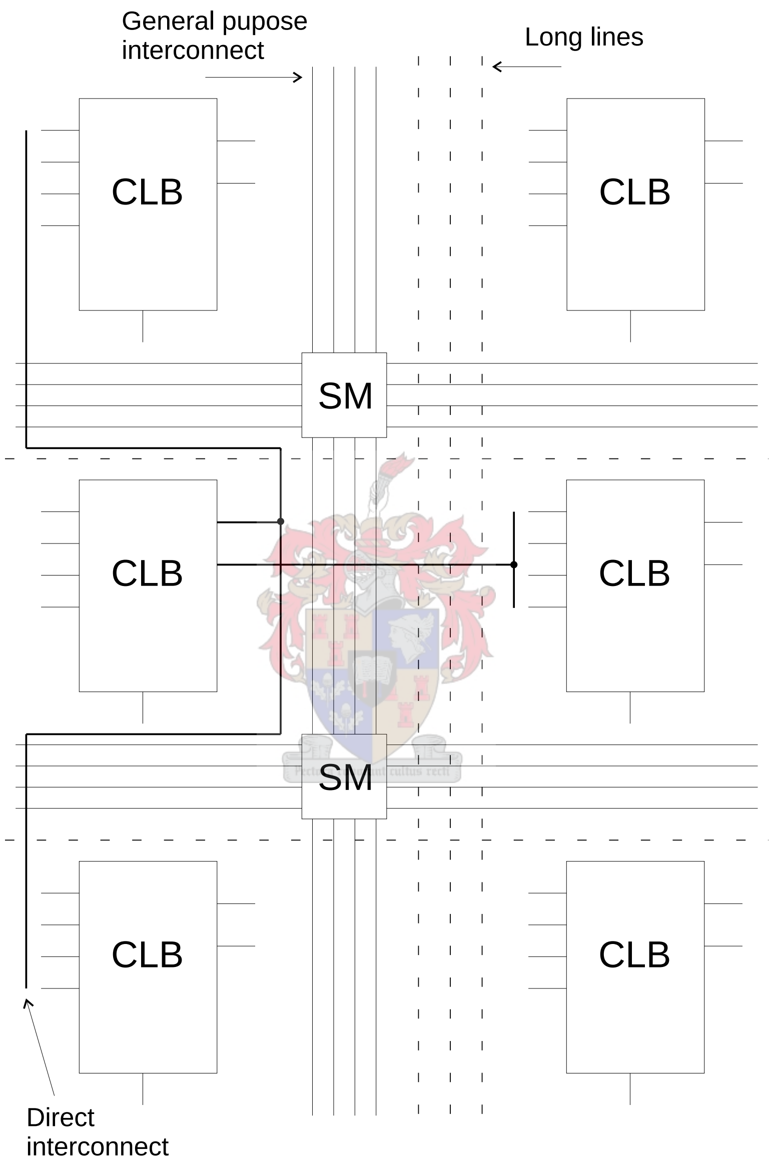

Figure 3.5: XC2000 routing architecture

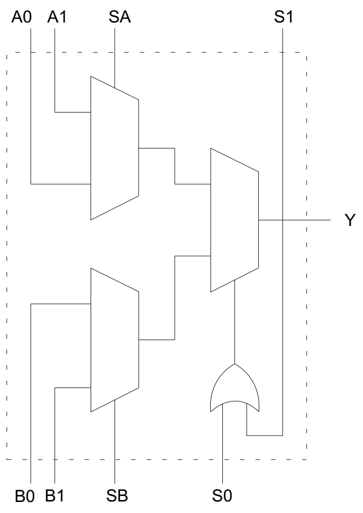

Figure 3.6: Logic Module in Act-1

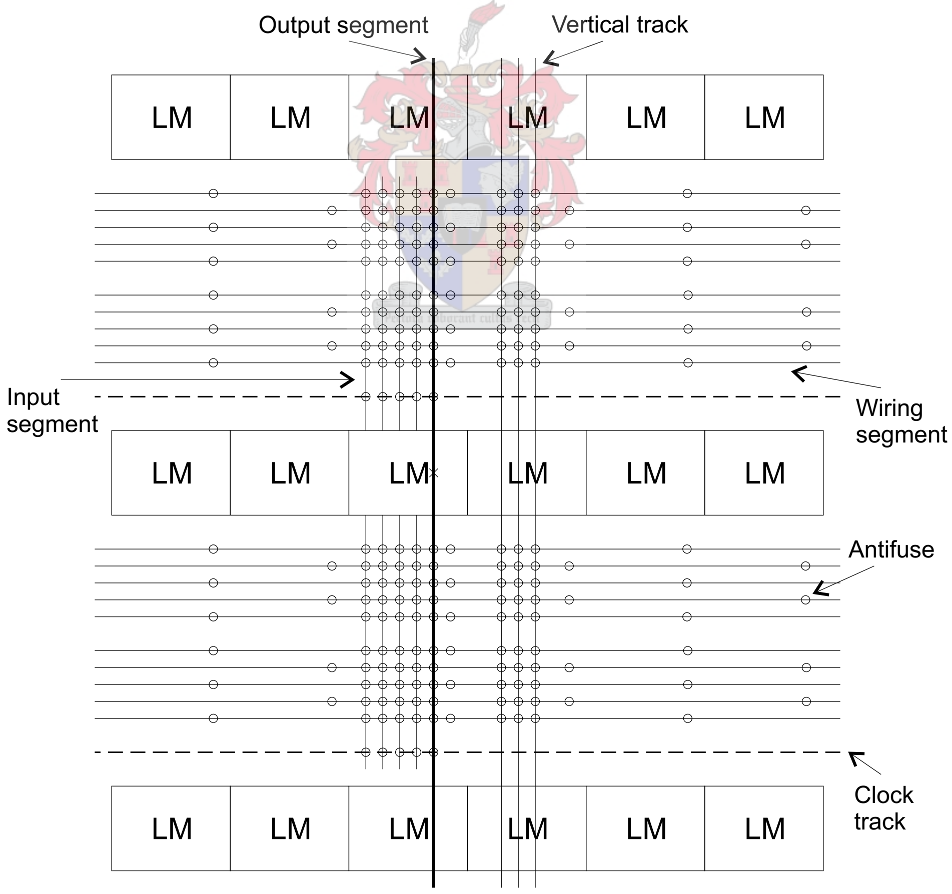

Figure 3.7: Interconnect architecture of the Act-1

3.1.3.4 Plessey

The Plessey FGPA architecture resembles the sea-of-gates structure in that a matrix of logic blocks is overlayed with dense interconnect resources. The logic block is relatively simple and is presented in Figure 3.8. The multiplexer is controlled by a static RAM block and is used to connect the logic block to the routing resources, which comprise wiring segments of varying lengths.

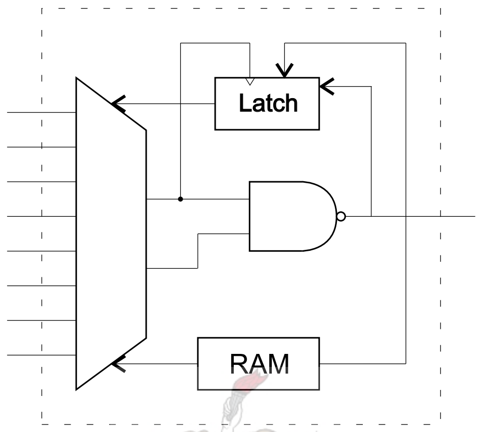

Figure 3.8: Logic block used by Plessey

3.2 Technology mapping

The technique of *technology mapping* transforms the required logic function to operate using the circuit elements of the programming technology for a given architecture [2]. This section looks at mapping into lookup tables and multiplexers.

3.2.1 Lookup table mapping

A K-input lookup table (LUT) is a digital memory that can implement any boolean function of K variables. The K inputs are used to address a memory array of 2^K 1-bit cells that stores the truth table of the boolean function. A K-input LUT can implement $2^{(2^K)}$ different boolean functions.

a number of LUT technology mappers exist, including *Chortle* [15] [16] [17], *mis-pga* [18] [19] [20], *Asyl* [21], *Hydra* [22], *Xmap* [23] and *VISMAP* [24]. All of these programs map a boolean network into a circuit of K-input LUTs, attempting to minimize either the total number of LUTs, or the number of levels of LUTs in the final circuit. Minimizing the total

[2]The process of *logic synthesis* actually consists of two separate phases, namely *logic optimization* and *technology mapping*, but logic optimization is beyond the scope of this project

number of LUTs allows the implementation of larger logic networks whilst minimizing the levels improves the speed-performance of the circuits.

The details of the algorithms used in the mappers are beyond the scope of this project but after consulting the comparative study by Brown *et al.* [12], it may be worth commenting on the implementation of the Chortle algorithms.

Chortle-crf [16] maps a boolean network into a circuit of K-input LUTs with the objective of minimizing the number of LUTs. The original network is first partitioned into a forest of trees and then each tree is separately mapped into a subcircuit of K-input LUTs. The final circuit is then assembled from the subcircuits implementing the trees. The major innovation of Chortle-crf is that it simultaneously addresses the decomposition and matching problems using a bin-packaging approximation algorithm and optimizes further by exploiting reconvergent paths.

Chortle-d [17] has the objective of minimizing the number of levels of LUTs in the final circuit and thereby increasing the circuit performance. The difference from Chortle-crf is that instead of minimizing the number of LUTs in the decomposition tree, it minimizes its depth. Results show that compared to Chortle-crf, Chortle-d reduces the number of logic levels by 38 percent, but increases the number of LUTs by 79 percent [12].

3.2.2 Multiplexer mapping

A multiplexer-based lookup table contains a tree of multiplexers to decode and implement boolean functions in a variety of different ways. An uncommitted logic block is *personalized* to implement different functions by connecting its inputs either to variables or to constants 0 or 1. Examples of mappers include *mis-pga* [18] [25], *Proserpine* [26] [27], *Amap* [28] and *XAmap* [28]. They minimize either the number of logic blocks or the delay in the final circuit.

3.2.3 Examples

The purpose of the examples given in this section are to more clearly illustrate the working of technology mapping and to show that it can be a difficult task, but also some of these examples can be used later to test the SPGA.

3.2.3.1 Arbitrary boolean functions

LUTs with 3 inputs are used to implement a boolean function with 5 variables (A,B,C,D and E). The final outputs are given as $X_{\#}$ where the following boolean functions are mapped[3]:

$$\begin{aligned} X_1 &= (A+B+C+D+E) && (3.2.1)\\ X_2 &= (A+B+C)*(A+D+E) && (3.2.2)\\ X_3 &= ((A+B+C)*(D+E+A))+ \\ &\quad ((A+C+E)*(D*B))*(A*C) && (3.2.3) \end{aligned}$$

Using Chortle-crf, the functions can be mapped as follows, with each line corresponding to a LUT[4]:

$$\begin{aligned} X_1^{a1} &= C+D+E \\ X_1 &= A+B+X_1^{a1} \\ \\ X_2^{a1} &= A+B+C \\ X_2^{a2} &= A+D+E \\ X_2 &= X_2^{a1}*X_2^{a2} \\ \\ X_3^{a1} &= A+B+C \\ X_3^{a2} &= A+D+E \\ X_3^{a3} &= A+C+E \\ X_3^{a4} &= (B*D)*X_3^{a3} \\ X_3^{a5} &= (A*C)*X_3^{a4} \\ X_3 &= (X_3^{a1}*X_3^{a2})+X_3^{a5} \end{aligned}$$

[3]Note: '+' is OR ; '$*$' is AND

[4]Note: superscript a indicates additional LUTs that had to be incorporated

Using Chortle-d, the functions mapped as follows:

$$\begin{aligned}
X_1^{a1} &= A + B \\
X_1^{a2} &= C + D + E \\
X_1 &= X_1^{a1} + X_1^{a2}
\end{aligned}$$

$$\begin{aligned}
X_2^{a1} &= A + B + C \\
X_2^{a2} &= A + D + E \\
X_2 &= X_2^{a1} * X_2^{a2}
\end{aligned}$$

$$\begin{aligned}
X_3^{a1} &= A + B + C \\
X_3^{a2} &= A + D + E \\
X_3^{a3} &= X_3^{a1} * X_3^{a2} \\
X_3^{a4} &= B * D \\
X_3^{a5} &= A + C + E \\
X_3^{a6} &= C * A \\
X_3^{a7} &= (X_3^{a4} * X_3^{a5}) * X_3^{a6} \\
X_3 &= X_3^{a3} + X_3^{a7}
\end{aligned}$$

Highlights of the algorithm that produce the above stated results are as follows: a) The results for boolean function 3.2.1 from both Chortle-crf and Chortle-d, show that Chortle-crf tries to minimize the number of LUTs because Chortle-crf produced two LUTs whilst Chortle-d produced three LUTs. b) The second boolean function was originally in a forced form and thus does not produce different output from either algorithm. c) The third function is a longer, more complex function and once again highlights the fact that the crf algorithm tries to minimize the number of LUTs (6 vs. 8) but does not minimize the number of levels (4 vs. 3).

3.2.3.2 Adder

A more common circuit that could be implemented in a FPGA is an adder. An interesting experiment would be to compare the mapping capabilities of the Chortle algorithm versus that of a human.

The task is to add two numbers 'A' and 'B', each consisting of two bits and produce the answer in 'S' (sum) and 'C' (carry). Let us first consider the case for the Chortle-crf algorithm. The first step is to set up the truth table as in Table 3.1. The next step is to find simplified boolean functions for each output bit. By using QMC software [29], the process is eased and the resulting output functions are[5]:

$$\begin{aligned}
C &= (A_1 * A_0 * B_0) + (A_0 * B_1 * B_0) + (A_1 * B_1) && (3.2.4)\\
S1 &= (A_1 *!A_0 *!B_1) + (!A_1 *!A_0 * B_1) + \\
&\quad (A_1 *!B_1 *!B_0) + (!A_1 * B_1 *!B_0) + \\
&\quad (A_1 * A_0 * B_1 * B_0) + (!A_1 * A_0 *!B_1 * B_0) && (3.2.5)\\
S0 &= (A_1 * B_1 * B_0) + (A_0 *!B_0) + (!A_0 * B_0) && (3.2.6)
\end{aligned}$$

Table 3.1: Truth table for adding two 2-bit numbers

input				output		
A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	1

The last step is to use Chortle and map the boolean functions into 3-input LUTs. The

[5]Note: '!' indicates a NOT operation

results are as follows, with each line representing a LUT:

$$\begin{aligned}
C^{a1} &= A_1 * A_0 * B_0 \\
C^{a2} &= A_0 * B_1 * B_0 \\
C^{a3} &= C^{a2} + (A_1 * B_1) \\
C &= C^{a1} + C^{a3}
\end{aligned}$$

$$\begin{aligned}
S_1^{a1} &= B_1 * A_0 * A_1 \\
S_1^{a2} &= !B_1 * A_0 *!A_1 \\
S_1^{a3} &= (B_0 * S_1^{a1}) + (B_0 * S_1^{a2}) \\
S_1^{a4} &= (A_1 *!B_1 *!B_0) + (!A_1 * B_1 *!B_0) \\
S_1^{a5} &= (A_1 *!A_0 *!B_1) + (!A_1 *!A_0 * B_1) \\
S_1 &= S_1^{a3} + S_1^{a4} + S_1^{a5}
\end{aligned}$$

$$\begin{aligned}
S_0^{a1} &= A_1 * B_1 * B_0 \\
S_0 &= (!A_0 * B_0) + (A_0 *!B_0) + S_0^{a1}
\end{aligned}$$

Now let us consider the human approach for the same addition task. With a little intuition and clever thinking the LUTs can be setup directly to produce the desired output, as shown in Table 3.2. In the table, the letter 'd' is a *don't care* placeholder.

Program LUT1 to add bit 0 of both input numbers and produce bit 0 of the sum. Then program LUT2 to produce the carry bit of the sum operation. Bit 0 of both input numbers will never be used again. Program LUT3 to add bit 1 of both input numbers and the carry bit of bit 0 to produce bit 1 of the sum. Similarly, program LUT4 to produce the carry bit of the operation.

This process can be expanded to add numbers of arbitrary width by simply using two LUTs (sum LUT and carry LUT) for each bit of the input number.

Compared to the algorithm which produced 12 LUTs, a human can do the same with 4 LUTs. The conclusion here is that for small circuits a human is far better than any algorithm but when larger circuits incorporating a large number of LUTs there is no other way but to use the CAD tools to help.

Table 3.2: Lookup tables: Human approach to adding two 2-bit numbers

LUT1			
inputs			**output**
d	A_0	B_0	S_0
d	0	0	0
d	0	1	1
d	1	0	1
d	1	1	0
d	0	0	0
d	0	1	1
d	1	0	1
d	1	1	0

LUT2			
inputs			**output**
d	A_0	B_0	C_0
d	0	0	0
d	0	1	0
d	1	0	0
d	1	1	1
d	0	0	0
d	0	1	0
d	1	0	0
d	1	1	1

LUT3			
inputs			**output**
C_0	A_1	B_1	S_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

LUT4			
inputs			**output**
C_0	A_1	B_1	C_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3.2.3.3 Multiplier

The multiplication operation is another common function that would be amenable to a FPGA. Let us define a 3-bit multiplication procedure as a human would do it:

$$
\begin{array}{cccccccl}
 & & & A_2 & A_1 & A_0 & \\
 & & \times & B_2 & B_1 & B_0 & \\
\cline{3-6}
 & & & (A_2^{c_0} & A_1 & A_0) & \text{AND } (B_0) \\
+ & & (A_2^{c_1} & A_1 & A_0) & & \text{AND } (B_1) \\
+ & (A_2^{c_2} & A_1 & A_0) & & & \text{AND } (B_2) \\
\cline{1-6}
c_3 & O_4 & O_3 & O_2 & O_1 & O_0 & \\
\cline{1-6}
\end{array}
$$

The output can be defined as follows: (NOTE: henceforth, '+' is an add operation):

$$\begin{aligned}
O_0 &= (A_0 \text{ AND } B_0) \\
O_1 &= [(A_1 \text{ AND } B_0) + (A_0 \text{ AND } B_1)] \\
O_2 &= [c_0 + (A_2 \text{ AND } B_0) + (A_1 \text{ AND } B_1) + (A_0 \text{ AND } B_2)] \\
O_3 &= [c_1 + (A_2 \text{ AND } B_1) + (A_1 \text{ AND } B_2)] \\
O_4 &= [c_2 + (A_2 \text{ AND } B_2)]
\end{aligned}$$

Then, let us define the carry bits:

$$\begin{aligned}
c_0 &= [(A_1 \text{ AND } B_0) + (A_0 \text{ AND } B_1)] \\
c_1 &= [c_0 + (A_2 \text{ AND } B_0) + (A_1 \text{ AND } B_1) + (A_0 \text{ AND } B_2)] \\
c_2 &= [c_1 + (A_2 \text{ AND } B_1) + (A_1 \text{ AND } B_2)] \\
c_3 &= [c_2 + (A_2 \text{ AND } B_2)]
\end{aligned}$$

This whole 3-bit multiplication operation can be mapped to 3-input look-up tables as follows:

$$\begin{array}{ll}
L_1 = (A_0 \text{ AND } B_0)_s & [O_0] \\
L_2 = (A_1 \text{ AND } B_0) & \\
L_3 = (A_0 \text{ AND } B_1) & \\
L_4 = (L_2 + L_3)_s & [O_1] \\
L_5 = (L_2 + L_3)_c & [c_0] \\
L_6 = (A_2 \text{ AND } B_0) & \\
L_7 = (A_1 \text{ AND } B_1) & \\
L_8 = (A_0 \text{ AND } B_2) & \\
L_9 = (L_5 + L_6 + L_7)_s & \\
L_{10} = (L_5 + L_6 + L_7)_c & \\
L_{11} = (L_8 + L_9 + L_{10})_s & [O_2] \\
L_{12} = (L_8 + L_9 + L_{10})_c & [c_1] \\
L_{13} = (A_2 \text{ AND } B_1) & \\
L_{14} = (A_1 \text{ AND } B_2) & \\
L_{15} = (L_{12} + L_{13} + L_{14})_s & [O_3] \\
L_{16} = (L_{12} + L_{13} + L_{14})_c & [c_2] \\
L_{17} = [L_{16} + (A_2 \text{ AND } B_2)]_s & [O_4] \\
L_{18} = [L_{16} + (A_2 \text{ AND } B_2)]_c & [c_3]
\end{array}$$

The human approach resulted in 18 LUTs with a depth of 5 layers. In comparison, the Chortle-crf algorithm produced 53 LUTs with a depth of 4 layers. The same process was used as for the adder. Start with a truth table, followed by boolean minimization and finally the mapping by Chortle.

3.3 Area vs Functionality

A lookup table can implement a large number of functions and thus has a high degree of functionality associated with it. As the functionality increases the number of logic blocks needed to implement a circuit decreases, but the area per block increases. From this statement one can deduce that it may be possible to find a compromise between logic block area and functionality.

As the functionality of a block increases, it is most likely that the routing requirements for each block will also increase. With multiple blocks in a FPGA, the total area of the chip is related to the functionality of the logic block. Since routing typically takes up a large percentage of the total area, the effect of logic block functionality on the routing can be very important. This section undertakes to find the optimal solution for this trade-off.

3.3.1 Model

In a comparative study done by Brown *et al.* [12], where different logic block implementations were assessed, the conclusion was reached that between lookup tables, NAND gates, multiplexers and AND-OR gates, the lookup table was found to be superior. Thus our model for the logic block incorporates a lookup table.

The model[6] used for the logic block is presented in Figure 3.9. Therein, *Dff* is a D flip-flop. The inclusion of this D flip-flop caused considerable debate. As will be seen in Chapter 4 (Section 4.4.4), the LUT is clocked and thus, the logic block already has synchronous functionality. A memory element (for state machines) can be made by feeding back the output of the logic block to one of its input. Hence, inclusion of a D flip-flop would not yield extra functionality to the logic block. It is was decided not to include a D flip-flop in the logic block. For completeness it is shown in the model although not used during calculations.

3.3.2 Logic block area and routing model

The mathematical model for the logic block area may be defined in such a way that it can be scaled according to the number of inputs. The logic block has a certain fixed area (FA) associated with it, which contains components that are necessary for basic functionality but do not increase with multiple inputs. As more inputs are added to a logic block, more components are needed, and thus the area of the logic block increasesr each additional input, one HUFFLE [7] is needed and its area is defined as HA. The number of memory cells needed

[6] According to [12], single-output lookup tables are the best choice

[7] HUFFLE: RSFQ circuit; see chapter 4.1

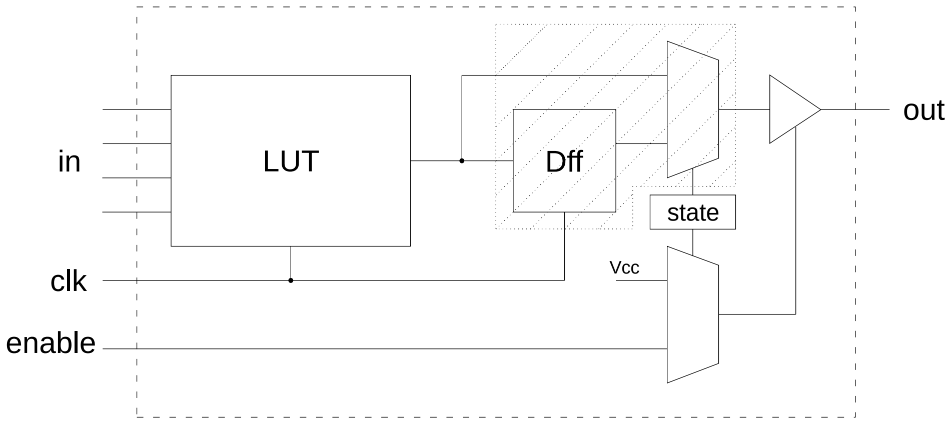

Figure 3.9: Model for logic block study

for a certain value of inputs is given by 2^K, where K is the number of inputs. Supporting components are needed to access each memory cell and thus the memory area (MA) is defined as the area taken by the memory cell including its supporting components. Now the area taken by a logic cell is given by:

$$LBA = FA + ((K \times HA) + (MA \times 2^K)) \tag{3.3.1}$$

A typical value[8] for FA is $45000\mu m^2$ while HA is $19500\mu m^2$. MA is typically in the order of $54000\mu m^2$.

The total area used by each logic block in the architecture of the whole chip can be accurately determined by taking into account the routing area around each logic block. In Figure 3.10 the Routing Area per Block (RApB) is given by:

$$RApB = 2 * (LBSL * W * RP) + (W * RP)^2 \tag{3.3.2}$$

where LBSL is the length of a side of the block (assuming the logic block is square) and is calculated from the logic block area: LBSL = $\sqrt{\text{LBA}}$. W is the number of routing wires and is usually equal to K+1. The routing pitch (RP) is determined by the size of the routing switches which are located inside the switch matrix (SM). The typical value of RP is $250\mu m$.

3.3.3 Area vs. functionality experiment

The goal of this experiment is to determine the optimal number of inputs to a logic block taking into account the trade-off between area and functionality. The experiment involves mapping typical circuits into lookup tables and then determine how much area each circuit requires with the help of our model.

[8]These typical values are calculated from RSFQ circuit layouts for the Hypres 3μm process [1]

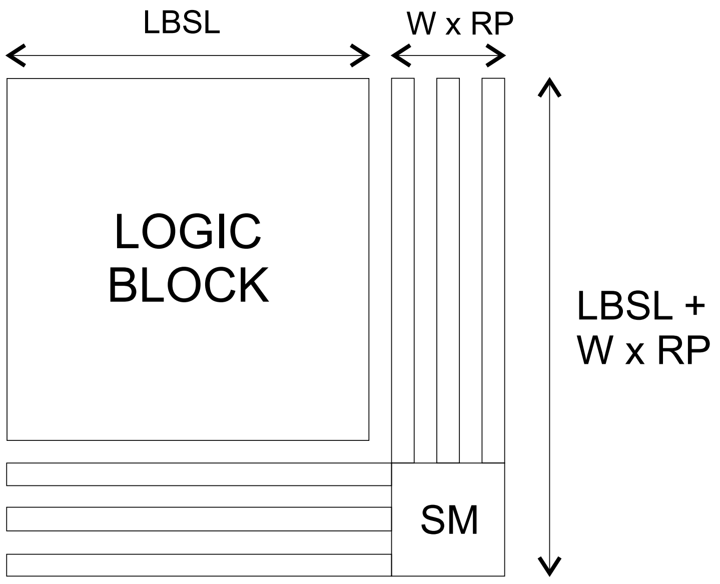

Figure 3.10: Routing area model

Table 3.3: Amount of LUTs required for each circuit vs. number of LUT inputs

	Number of LUT inputs					
Circuits	**2**	**3**	**4**	**5**	**6**	**7**
X_1	4	2	2	1	1	1
X_2	5	3	2	1	1	1
X_3	12	6	5	1	1	1
Adder	30	12	3	3	3	3
Multiplier	117	53	27	16	10	10

The typical circuits considered are those that were discussed in section 3.2.3 and they include the three arbitrary binary circuits, the adder and multiplier. For each case, the circuit is mapped into lookup tables with 2 and up to 7 inputs. The number of lookup tables that each circuit requires for each amount of inputs is shown in Table 3.3.

Figure 3.11 shows two graphs, the average number of blocks of the five circuits for each number of inputs, and the logic block area (including the routing area) for each number of inputs.

To better understand how the areas of the circuits compare to each other, the values were normalised. By normalising the values, the results are more readily comparable to other circuits that were not discussed in this study. The graph in Figure 3.12 shows the individual results for the normalised area for each circuit. The results show that for small circuits, it is better to use logic blocks with fewer inputs. For larger circuits an advantage is gained when more inputs are used.

From Table 3.3, the average number of LUTs over the five circuits can be used as a yard-stick to judge the total area that a typical circuit would require. Figure 3.13 shows the average

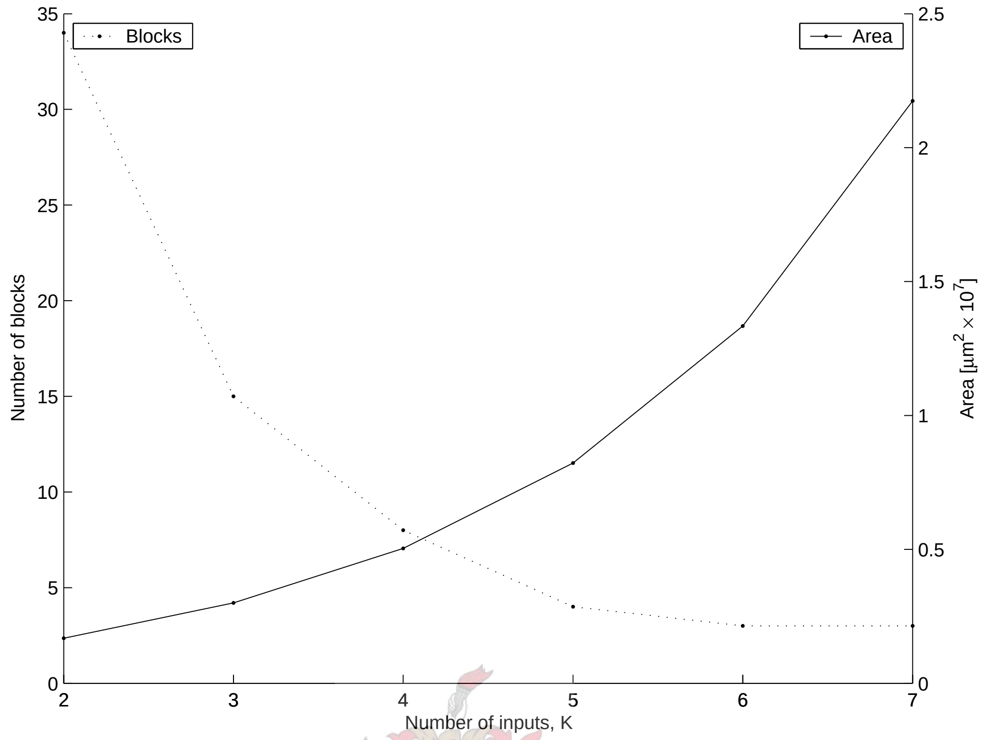

Figure 3.11: Number of blocks and Area (incl. routing) per block

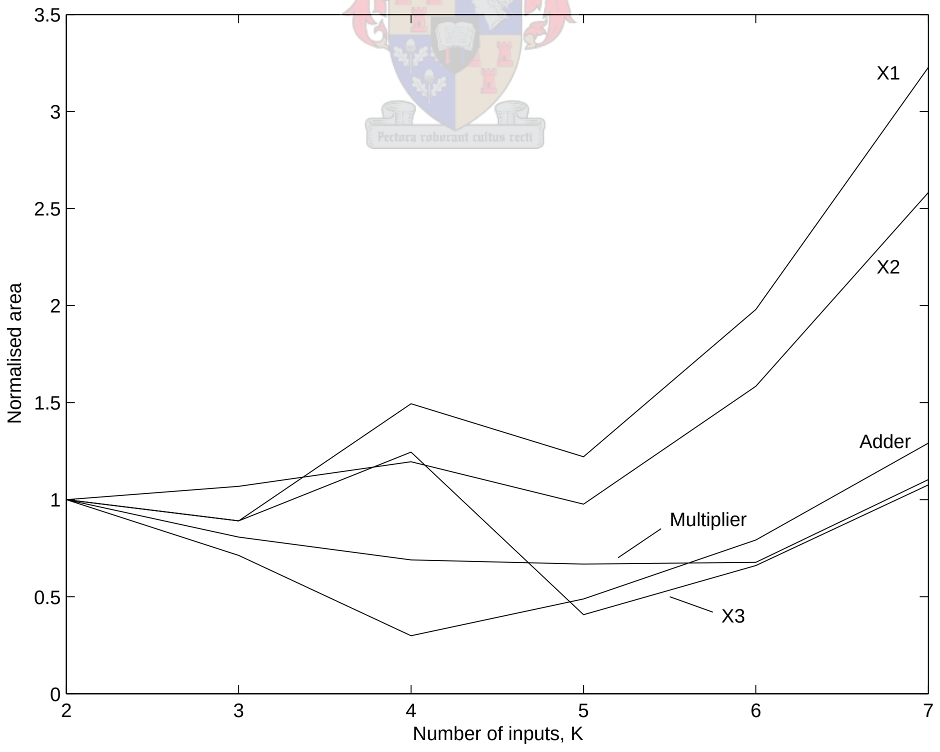

Figure 3.12: Number of blocks and Area (incl. routing) per block

normalised area that this 'average' circuit would require against the number of inputs. The total area[9] is calculated as: TA = (LBA + RApB) $\times$ [number of blocks]. The result shows that by using existing RSFQ technology a lookup table with 5 inputs is the best choice for the SPGA (although a lookup table with 3 inputs is a very good second choice).

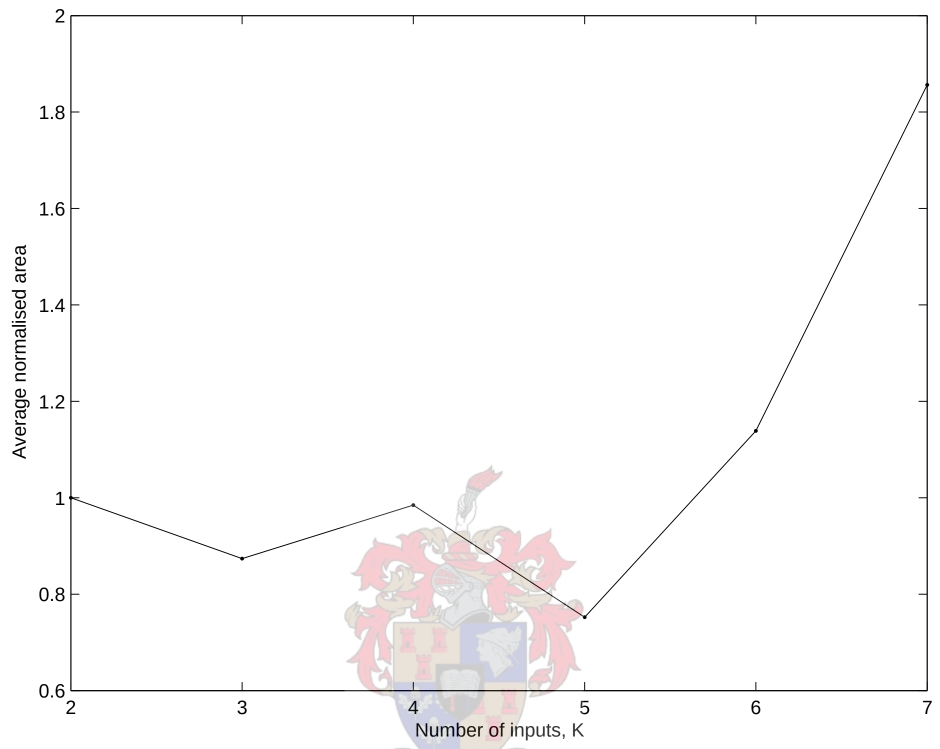

Figure 3.13: Average normalised area for the circuits in Table 3.3

3.4 SPGA

Information gathered and work done in this chapter culminated in the determination of the best solution or architecture for the Superconducting Programmable Gate Array (SPGA). The conclusion is that by using a symmetrical array of logic blocks containing RAM-based lookup tables the best results were achieved. The lookup table provides great flexibility and functionality while the symmetrical array architecture provides a natural routing structure to feed the logic blocks. Each logic block contains a lookup table with 5 inputs and one output without a D flip-flop. The symmetrical array routing architecture lends itself to the use of switch matrices. The same memory cells used in the lookup tables can be used to control the switches. Another advantage is that a wealth of information is available regarding mapping into lookup tables along with algorithms and programs to aid in the process.

[9]The total area is also represented by multiplying the two graphs in Figure 3.11

Unfortunately during the layout phase[10], it became clear that there is not enough space on the die to implement a 5-input lookup table. After some layout trials with 4- and 3-input LUTs, results showed that even 4-input LUTs are to big too use and 3-input LUTs do not leave enough space for the routing architecture and the on-chip programming circuits. The eventual decision was to use four 2-input LUTs and a 2-line routing architecture. Although this is not optimal it is sufficient to demonstrate the concept of an SPGA.

Now that the structure and architecture of the SPGA has been proposed, the next step is to discuss *placement of logic blocks* and *routing*. A circuit can be mapped into LUTs with relative ease, but to implement a boolean circuit, the LUTs must be connected with limited routing resources. This is the problem associated with *placement and routing*. Following the discussion in the previous paragraph, and the realization that only four 2-input LUTs will be availiable on-chip, it is not worth going into much detail about placement and routing algorithms because it has become a trivial task.

3.5 Chapter summary

During this chapter, research was presented on various types of programmable logic and different architectures that are employed in existing semiconductor programmable logic. A comparison was done to ascertain which combination will best suit our superconducting programmable logic. The resulting solution will now be used as a framework from which RSFQ circuits can be designed and implemented to realize an SPGA.

[10]Refer to Chapter 5

Chapter 4

SPGA design

'Any sufficiently advanced technology is indistinguishable from magic.'

[Arthur C. Clark - Technology and the Future]

FROM the research in Chapter 3 flows the ideal structure for a reprogrammable logic device. In this chapter those ideas and concepts are used as a framework to design and construct a superconducting reprogrammable device called the SPGA. Following in this chapter are discussions on RSFQ circuits and blocks that constitute the final design, including some simulations that provide insight into the operation of certain key circuit constructs.

Some existing RSFQ circuits are used and some newer circuits were developed specifically for this project. These gates are discussed and shown how they can be assembled to create larger circuit blocks that have functional properties that to in constructing the final SPGA.

4.1 Basic gates

This section discusses most of the basic gates used in this project. Basic gates may be defined as circuits that cannot be broken up into smaller functional circuits.

Circuits diagrams are presented and a short discussion follows on the overall working of the gates. In all gates a DC bias of 2.6mV is assumed unless stated otherwise. Only the most important gates, gates that need explanation or gates that have changed under investigation in this project are discussed in this chapter. All other gates are presented in Appendix C.

4.1.1 DCRL

DC-Resetable Latch (DCRL) [30] in Figure 4.1 is one of the more important gates in this project because it is the basis on which the reprogrammability concept will be built. It is the main memory element in the design with the added bonus that it is electrically resetable. More detail on the design of this gate was given previously by Fourie [4].

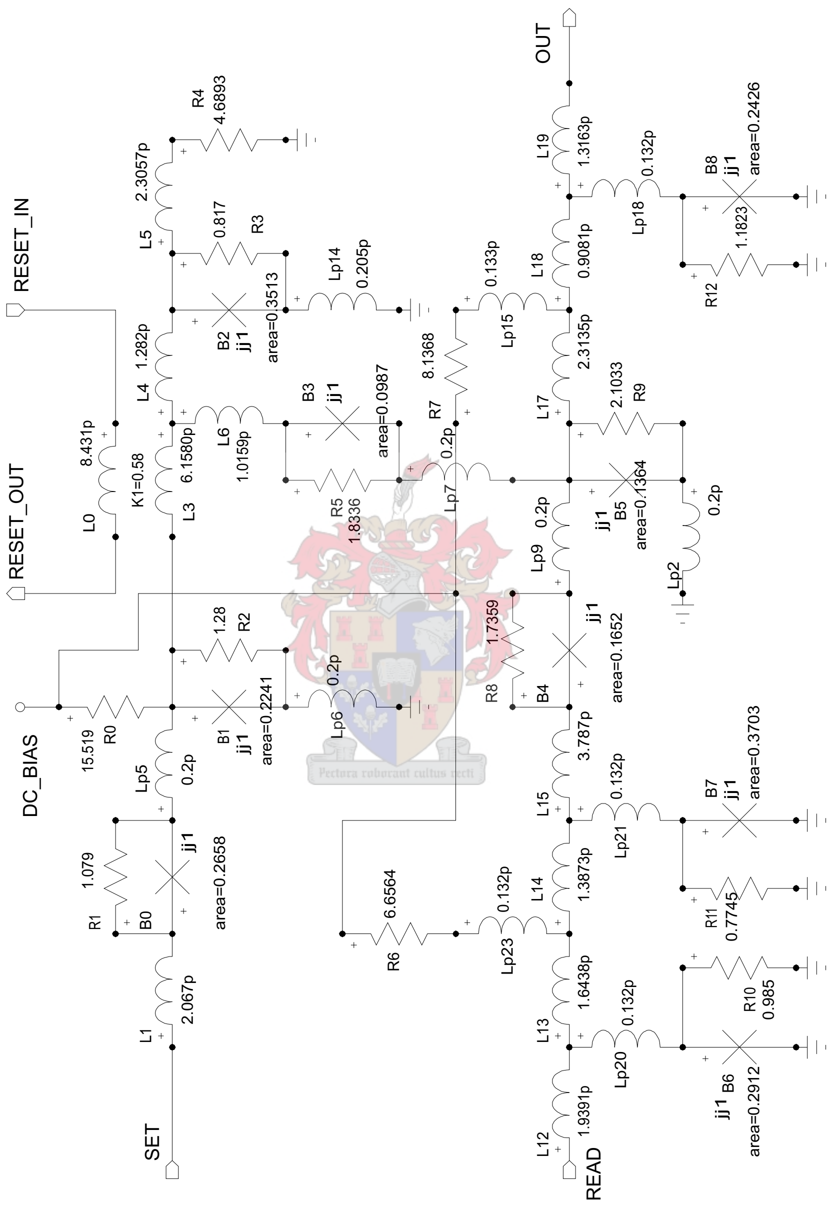

Figure 4.1: Circuit diagram: DCRL

The output is dependent on the state of the gate (*set* or *unset*), which can be changed by means of the *SET* or *RESET* ports. The latch is set by applying a pulse to the *SET* input, or reset (to the unset state) by applying a current through the *RESET* port. When the gate is in the set state, a pulse is produced at the *OUT* port, when a pulse is received at the *READ* port.

4.1.2 HUFFLE

The Hybrid Unlatching Flip-Flop Logic Element (HUFFLE) in Figure 4.2 is a versatile circuit. It was originally developed as a memory element but its primary task in this project is as a bipolar current driver. It is mainly used as a selection tool during the programming of the memory elements and as a decoder in the lookup table.

The HUFFLE's output is a current flowing between *I_OUT+* and *I_OUT-*. The direction of the current flow is determined by the state of the gate. By application of a pulse to the *SET* port the current will flow from + to -. The opposite flow is achieved by application of a pulse to the *RESET* port. The state can be predetermined with the aid of the prebias (PB) port. The HUFFLE is capable of driving large inductance values but the trade-off is slower response times.

The HUFFLE is a rather large circuit to simulate and it is used extensively in the design. For that reason it was decided to substitute a current source to simulate the output of the HUFFLE. Under ideal conditions the amplitude of output current should be around 340μA, but slight deviations can occur due to varying load conditions and changes in circuit parameters (manufacturing process variations). With the aid of a Monte Carlo simulation, where circuit parameters and loads were varied, a statistical model for the output of the HUFFLE was developed. The results are Gaussian distributions with a nominal amplitude and standard deviation for both positive and negative flowing currents. The positive mean is 339μA with a standard deviation of 12.27μA. The negative mean is 333μA with a standard deviation of 12.11μA.

This statistical model for a current source can now be used instead of the large HUFFLE circuit in simulations. The advantage is shorter simulation times and more control over timing. The only drawback is the absence of the transient response during state changes which should always be kept in mind. Obviously, this model is not a true replacement and a final simulation should always incorporate the whole HUFFLE circuit to ensure accurate results.

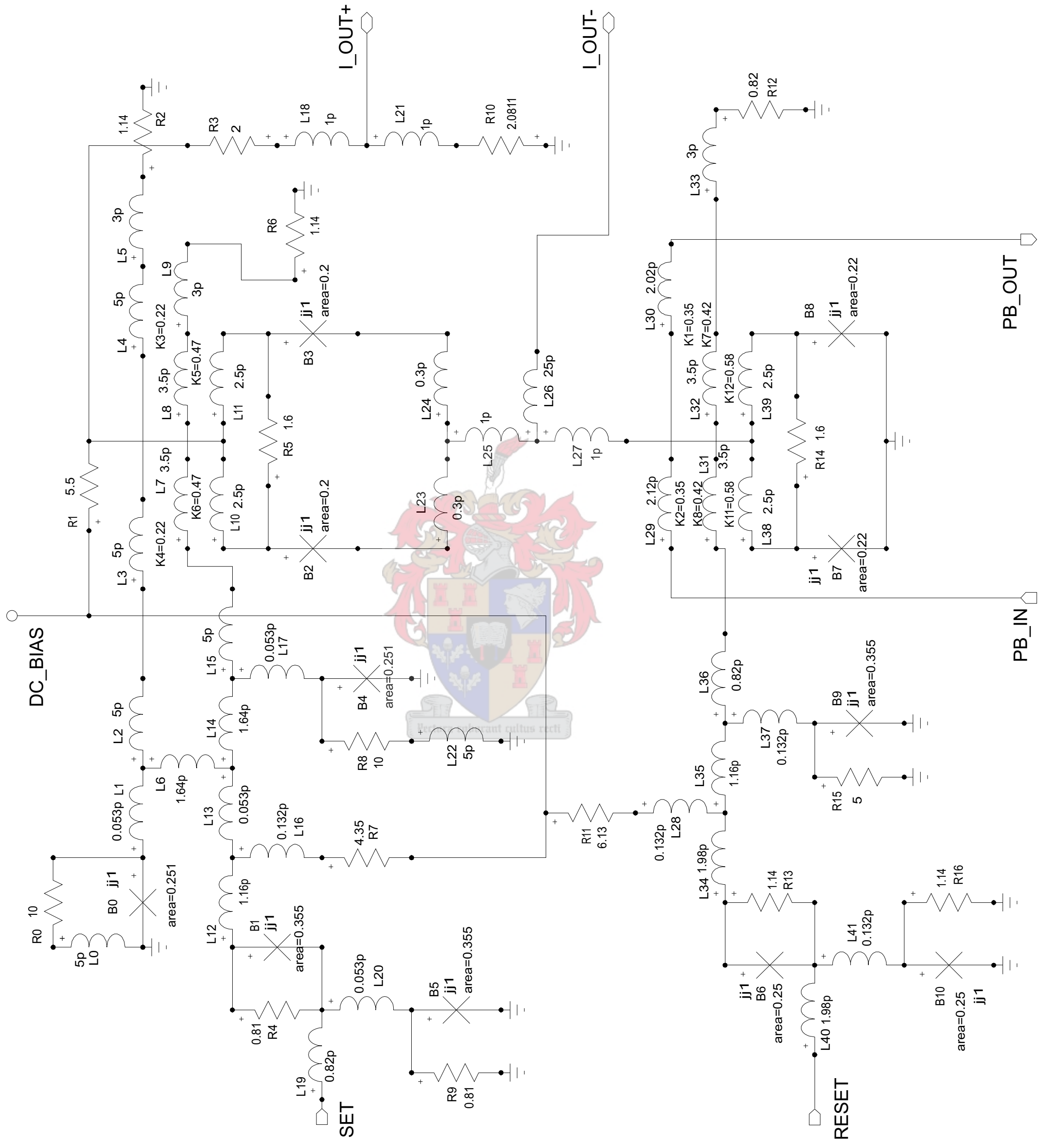

Figure 4.2: Circuit diagram: HUFFLE

4.1.3 I-Switch

A large number of uniquely accessable memory cells (as is the case in this project) would require some sort of access scheme. A matrix-type scheme has been proposed [4] where each memory element could be represented as a cell in a matrix (i.e. as intersections of row and column lines). The Current-Set Switch (I-Switch) has been purposefully designed for this task [4].

The *DC_SET* port in Figure 4.3 requires a dc current of the type that can be supplied by the HUFFLE. When a pulse arrives at the *READ* port while a positive (IN to OUT) dc current is applied to the *DC_SET* port, an output pulse is generated. Any other combination does not produce output.

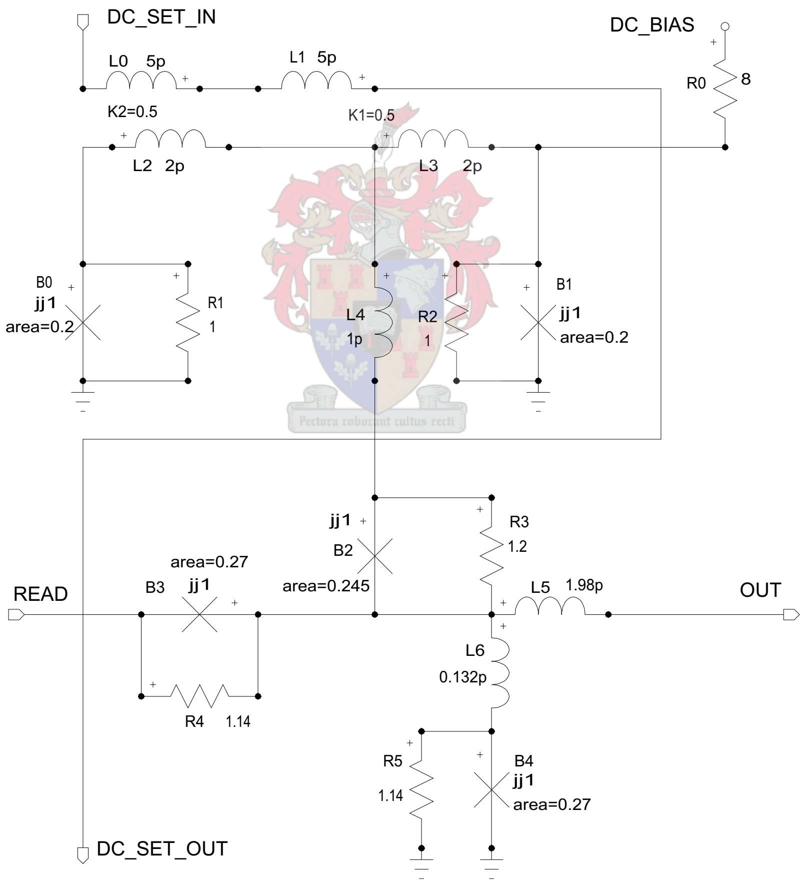

Figure 4.3: Circuit diagram: Current-Set switch

4.1.4 I2-Switch

The Two-line Current-Set Switch in Figure 4.4 works on the same principle as the Current-Set Switch but has been extended to two *DC_SET* lines. Now, a positive current (IN to OUT) must be injected on both *DC_SET* lines while an input pulse arrives at the *READ* port for an output pulse to be generated.

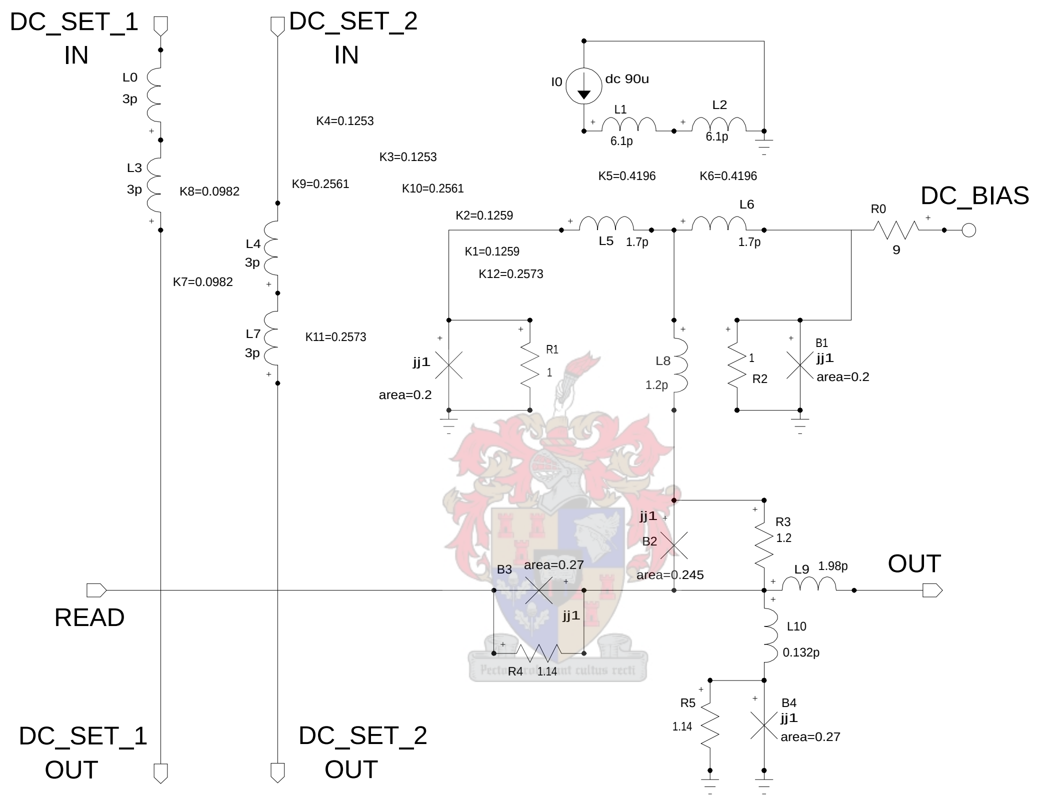

Figure 4.4: Circuit diagram: Two-line current-set switch

This gate is the basis for the decoder in the lookup table. The logic block circuit (including the lookup table) will be discussed in greater detail in section 4.4.4.

The SPICE model in Figure 4.4 is the nominal circuit, but an extended Monte Carlo model was derived (see Chapter 5.6.3) that also takes into account the variations in the coupling factors between the inductors of the select lines, bias control line and SQUID. Using this extended model in combination a HUFFLE[1], a yield of 100% was achieved.

[1]Note: the HUFFLE usually drives the select lines

4.1.5 MSL Driver and Receiver pair

The Microstrip transmission line (MSL) Driver and Receiver pair from [31] was used as a means to transmit SFQ pulses over relatively long distances. A small series resistor (not shown), which provides decoupling between the two gates, should be included in the layout when connecting the two gates together. The circuits for the driver and receiver are presented in Figure 4.5 and Figure 4.6 respectively.

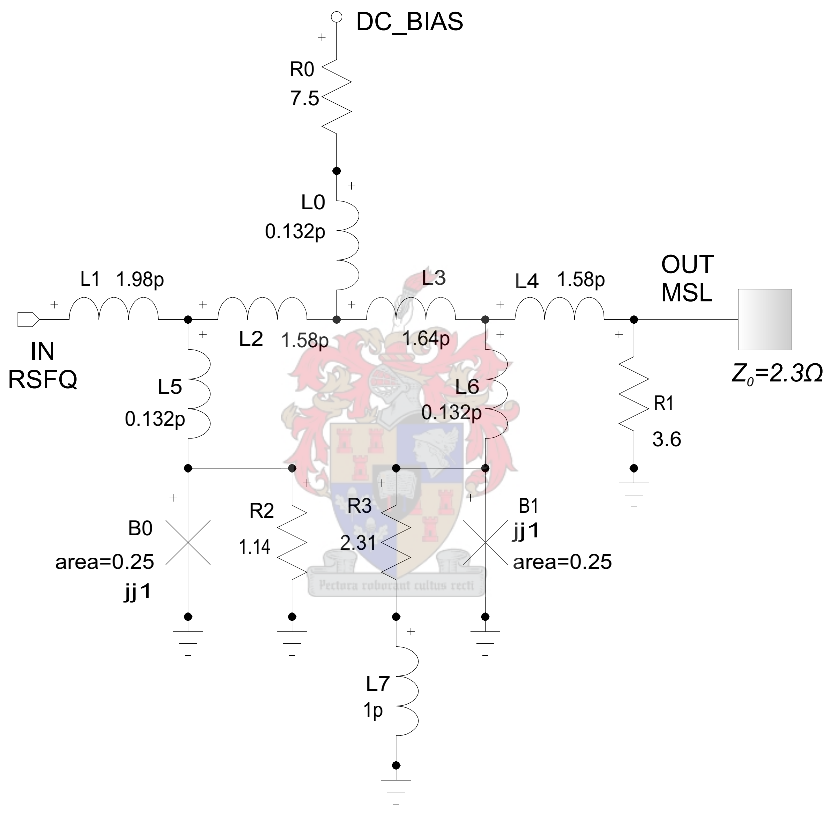

Figure 4.5: Circuit diagram: Microstrip Transmission Line Driver

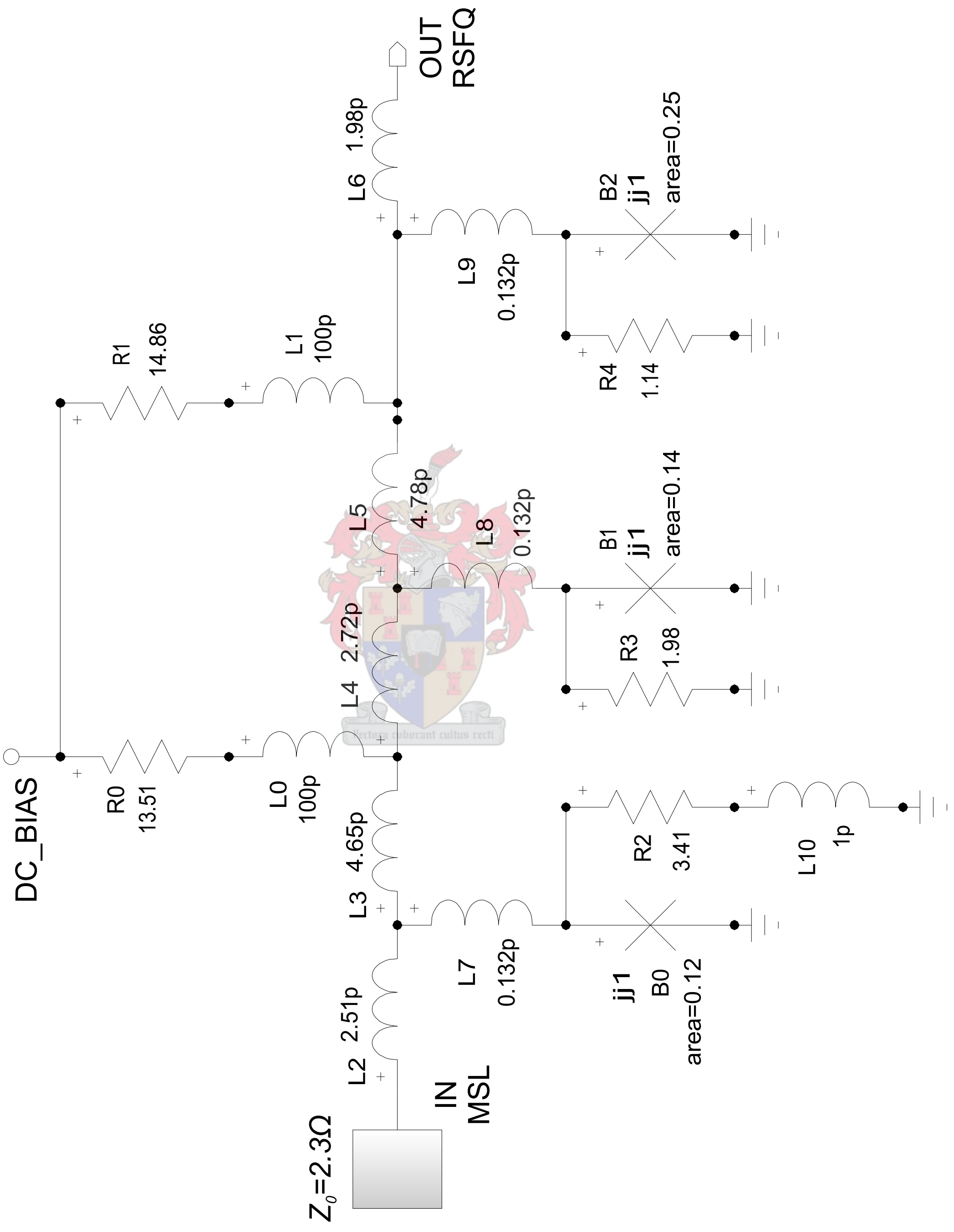

Figure 4.6: Circuit diagram: Microstrip Transmission Line Receiver

4.1.6 RSFQ-to-COSL Converter

An RSFQ-to-COSL converter in Figure 4.7 was implemented as discussed by Fourie *et al.* [32]. This gate is mainly used as a reliable interface between RSFQ and room-temperature circuits. The gate accepts RSFQ signals at the IN port which are stored in a modified DRO. The only limitation is that an input signal may not arrive during most of the positive cycle of the sinusoidal clock (CLK) signal. The sinusoidal clock signal should have a amplitude of 10mV. The output signal shape is similar (but slightly longer) than RSFQ pulses and can reach an amplitude of around 1mV. The output port has been optimized for a 5Ω load.

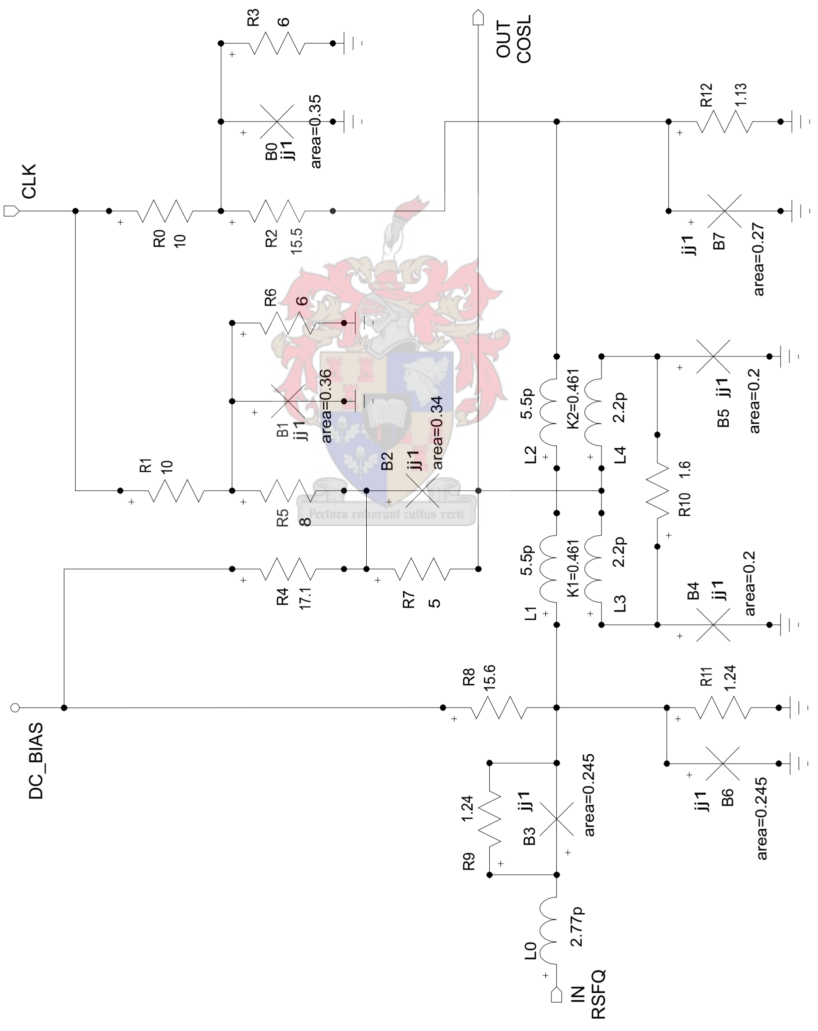

Figure 4.7: Circuit diagram: RSFQ-to-COSL converter

4.2 Monte Carlo analysis and simulations

Monte Carlo simulations have proved to be a useful tool in predicting circuit yield of RSFQ circuits [33] and have evolved from margin analysis [34] to encompass all circuit elements and take into account integrated circuit (IC) process parameter variations as well as bias trimming. A complete Monte Carlo model [35] encompasses tolerances expected for a standard layout, and is derived from process specifications such as uncertainties in etch widths and layer thickness.

Monte Carlo analysis in itself can be used to optimize circuits [36] but in combination with a genetic algorithm [4] [37] provided a very powerful technique which was used to optimize nearly all of our basic gates.

4.3 Inductance restrictions between gates

SPICE models of our RSFQ gates have been optimized with a specific value of input- and output inductance values. These inductances (IO inductances) represent the physical interconnections between gates. During the layout phase, it was not always possible to keep the IO inductances to the exact value specified by the SPICE circuits. Due to the tight space restrictions on the die, some gate- and block configurations result in either too short or too long distances between gates to restrict the inductance values.

Therefore, a study was undertaken to find the range of IO inductances that would be allowed between gates and still produce a working circuit with good yield. The investigation involves setting up a SPICE simulation with two gates connected together. The IO inductances of the separate gates are combined into a single inductance. A Monte Carlo simulation is used to calculate the yields of the combined circuit while the new IO inductance is varied with greater range until undesirable results (yield less than 98%) are produced. The outcome of this process is a minimum- and maximum value of tolerable IO inductances. The process is laborious because the IO inductance has to be varied by hand in each run. The ideal situation would be to test all possible combinations of available gates and compose a matrix so that any layout engineer can see what his/her range of freedom is in terms of intergate inductances. Unfortunately due to time restrictions, only the gates that caused considerable trouble during layout were tested. The results are given in Table 4.1 where the '?' indicates an intractable value.

More work has to be done in this regard and it may be worth while investing time to find an automated process to acquire intergate IO inductance ranges. This information is invaluable to a layout engineer (or CAD package). Also, a more reliable method to acquire the maximum inductance value needs to be found. Considerable difficulty was encountered because pulses get trapped in large inductances or get shifted in time. The technique in-

Table 4.1: Intergate IO inductance ranges

gate -> gate	range [pH]
DCRL → DCRL(reset)	1.5 - 7.2
DCRL → JTL	0.5 - ?
DIV → JTL	0.5 - 7
DRO → DIV	2.3 - ?
DRO → JTL	1.5 - 7.8
I-switch → JTL	3.5 - ?
I2-switch → JTL	2 - ?
JTL → DIV	0.5 - 5.2
JTL → DRO(set)	0.5 - 7
JTL → I2-switch	0.5 - 4.7
most gates	0.5 - 5

volves three consecutive SFQ pulses, but testing for missing- or shifted pulses is difficult to implement in Monte Carlo simulations and yields inconsistent results.

4.4 Composite blocks

Composite blocks are combinations of basic gates and/or other blocks. These blocks have been composed to serve a specified purpose in the operation of the SPGA. A description of the basic working (including diagrams) of all the blocks is presented in this section.

4.4.1 Inline Switch

The inline switch in Figure 4.8 can be used to stop pulses from propagating along a line. When the inline switch is active, the DCRL has been engaged to the set state (with the aid of the I-Switch). Hereafter, when a pulse arrives at the IN port, it triggers the DCRL to produce a pulse at the OUT port. When the switch is inactive, the DCRL has not been engaged, and acts as a barricade.

4.4.2 Junction Switch

The junction switch in Figure 4.9 can be used to connect two lines. It is mainly constructed from a divider, DCRL and a merger. The divider duplicates the signal in the original line (which in Figure 4.9 is the vertical line), while the DCRL arbitrates propagation and finally the merger combines the two lines. When active, it allows pulses to propagate from the original to the second line. When inactive, pulses cannot propagate from the original line to the second line. It does not hinder pulses already travelling on either line.

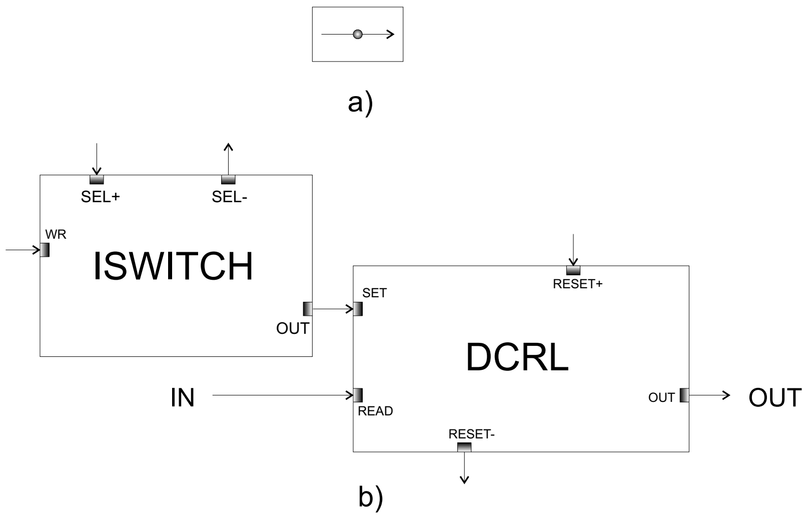

Figure 4.8: Inline switch: a) Symbol; b) Schematic block diagram

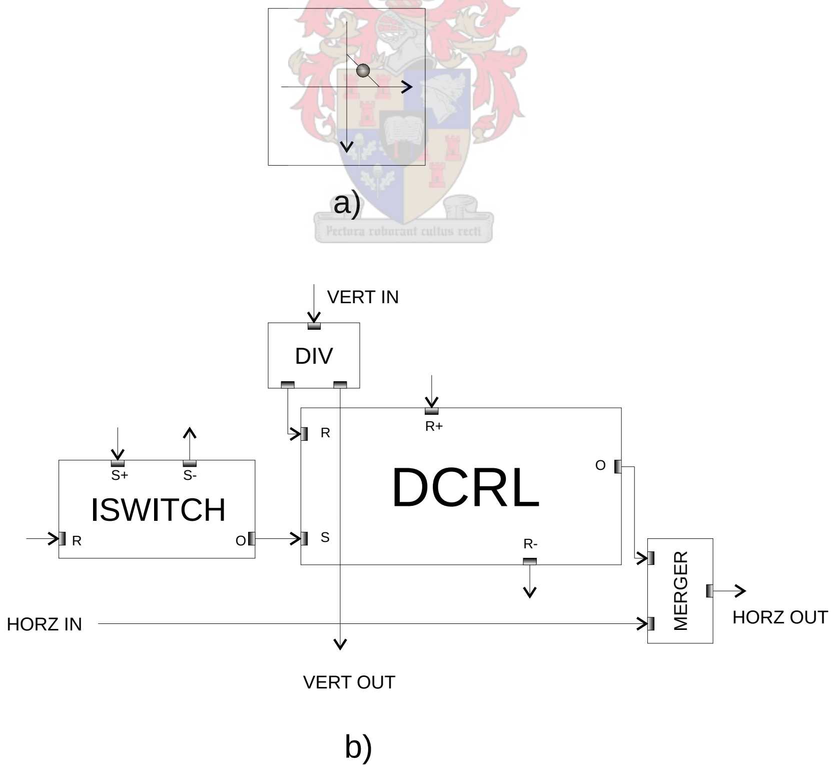

Figure 4.9: Junction switch: a) Symbol; b) Schematic block diagram

4.4.3 Crossbar Switch

The crossbar switch [4] in Figure 4.10 is an extension of the junction switch idea. In Figure 4.10, it connects the horizontal- and vertical lines and allow pulses to flow from one to the other. It is constructed of two DCRLs, but both are programmed with the same data. The result is that signals from both lines can either cross-flow or not. There is no case where one side is connected and the other not. It does not hinder pulses already travelling on either lines.

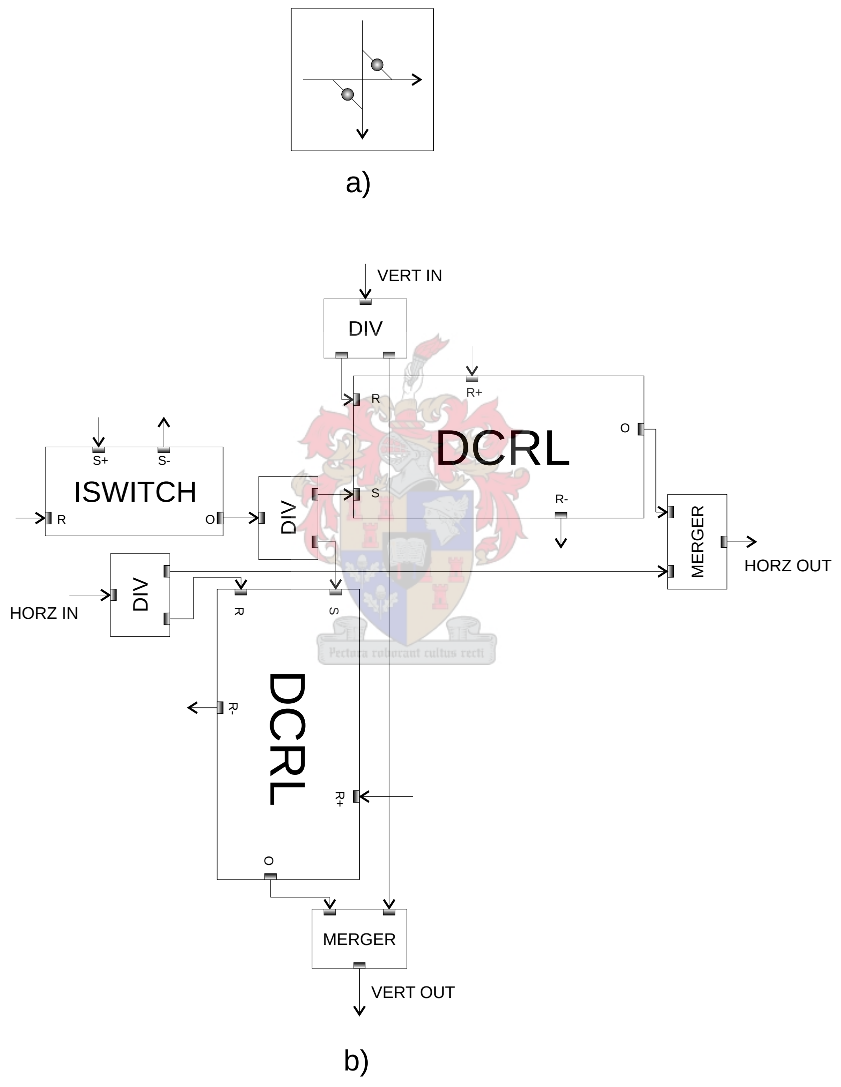

Figure 4.10: Crossbar switch: a) Symbol; b) Schematic block diagram

4.4.4 Logic Block

A diagram of the logic block is presented in Figure 4.11. The logic block is based on a lookup table and is composed of a decoder, memory cells (including support for data loading) and merging sections.

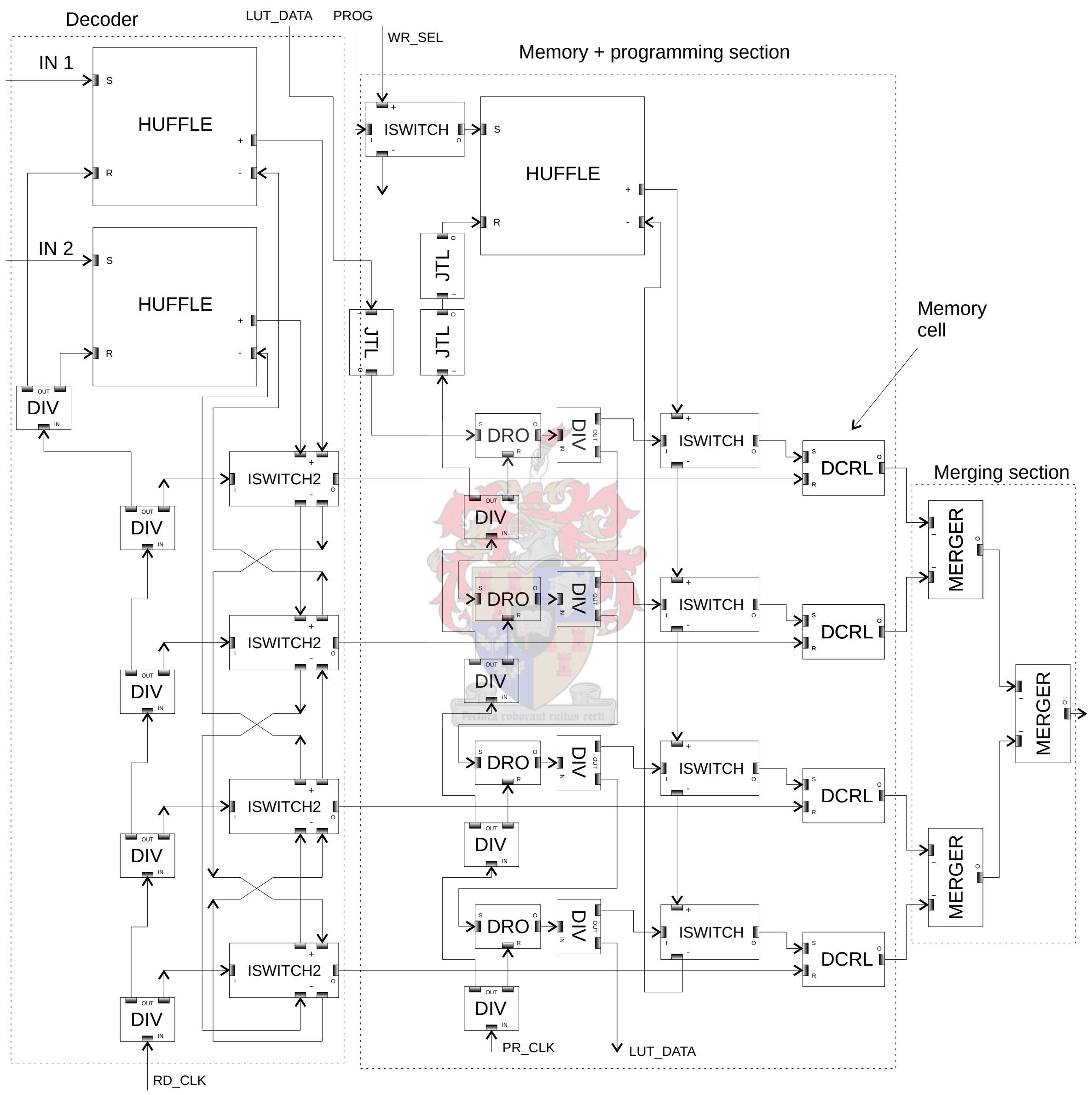

Figure 4.11: Schematic diagram: Logic block

The decoder section is composed mainly of the two HUFFLEs to the left and the four I2SWITCH blocks. The RSFQ inputs (*IN 1* and *IN 2*) are translated to quasi-DC signals by the HUFFLEs. By connecting the HUFFLE output[2] lines to the I2-switches in the manner shown[3] it is possible for only one of the I2-switches (specified by the input) to generate an output pulse. The two select lines of the I2-switch represent boolean bits: a current flowing from top ('+') to bottom ('-') is regarded as positive or a logical '1'. Table 4.2 shows how the memory cells (DCRLs) are mapped to the logic block inputs.

Table 4.2: Mapping: Logic block inputs to memory cells

cell	IN 2	IN 1
0	1	1
1	1	0
2	0	0
3	0	1

The rest of the circuitry in the logic block is related to programming, clock distribution and merging the output.

The memory cells of the logic block are programmed by serial shifting. The bits are shifted in from the *LUT_DATA* port, one bit at a time as the *pr_clk* port is pulsed. Just before the last clock pulse, both the *WR_SEL* and *PROG* are asserted to activate the HUFFLE, which in turn will allow the program data to set the respective memory cells (DCRLs).

4.4.4.1 Logic block decoder simulation

SPICE is the preferred choice to investigate the behaviour of RSFQ circuits and gates. In this project, WRspice [38] was used extensively to explore the working of various gates, blocks and configurations of gates.

For reference, Figure 4.12a shows a diagram of the decoder section found in the logic block. The simulation tests all the possible input combinations and verifies that the correct I2-switch produces an output pulse. Refer to Table 4.2 for the mapping of input to I2-switch.

The simulation results are presented in Figure 4.13, where a clock with 300ps period is applied to the *RD_CLK* port. The clock serves as read input for the I2-switches and as reset for the HUFFLEs. The two select lines are the HUFFLE output currents, which is a direct translation of the *IN 1* and *IN 2* ports, but shows more clearly the timing of the various signals. The *IN 1* and *IN 2* pulses precede the clock pulses by 100ps.

The simulation shows that for all the possible inputs, the correct I2-switch produces a pulse [4].

[2]The HUFFLE produces a current flowing either from '+' to '-' or the other way depending on its state

[3]The sequence is known as a Gray code, which minimizes the simultaneous bit changes, and in our configuration minimizes the number of wire-crossings from one I2-switch block to the next [4]

[4]NOTE: The Gray code wiring implementation of the I2-switches changes the natural order that one would expect the I2-switches would produce output

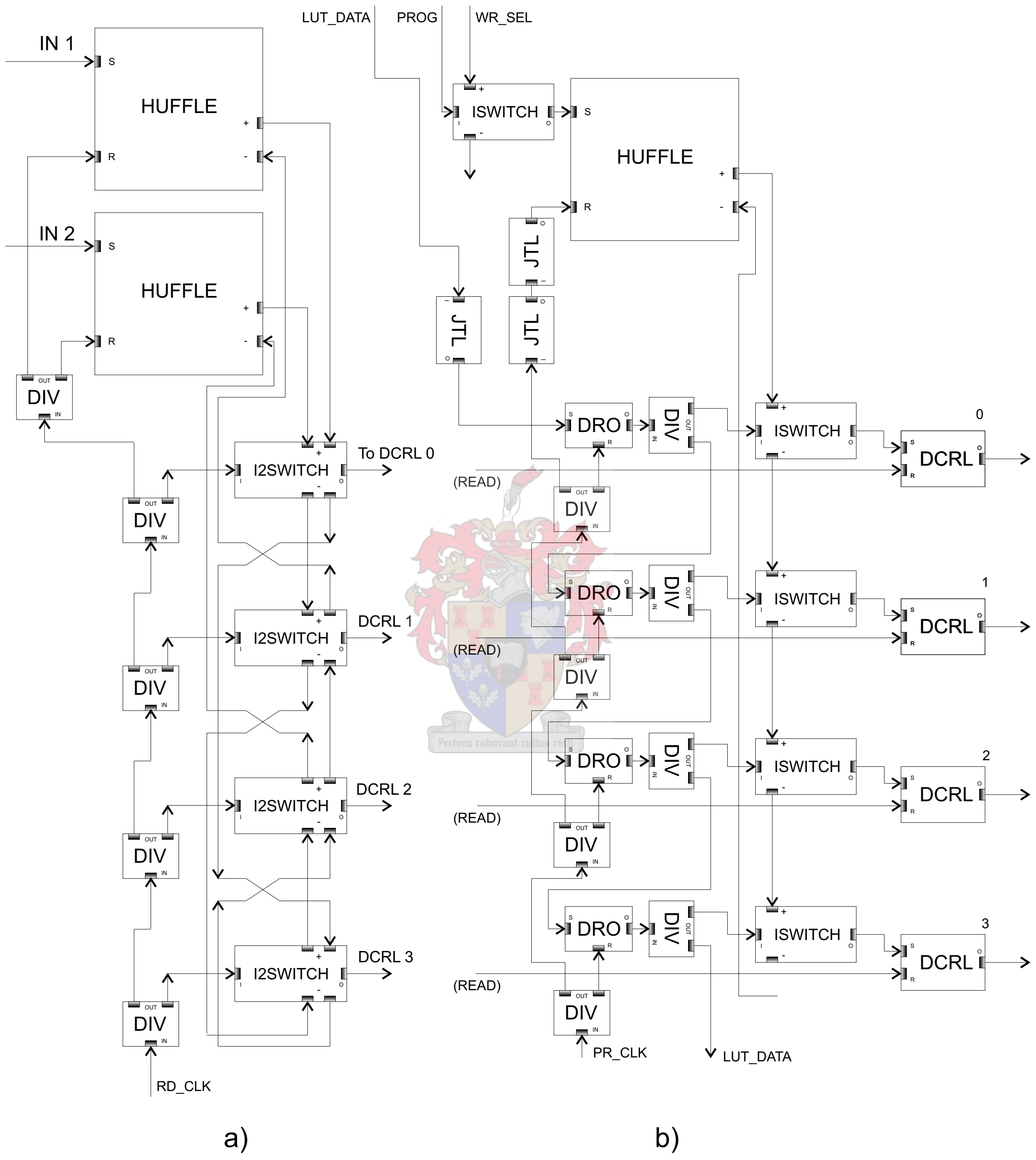

Figure 4.12: Schematic diagram of the Logic block: a) Decoder; b) Programming and memory cells

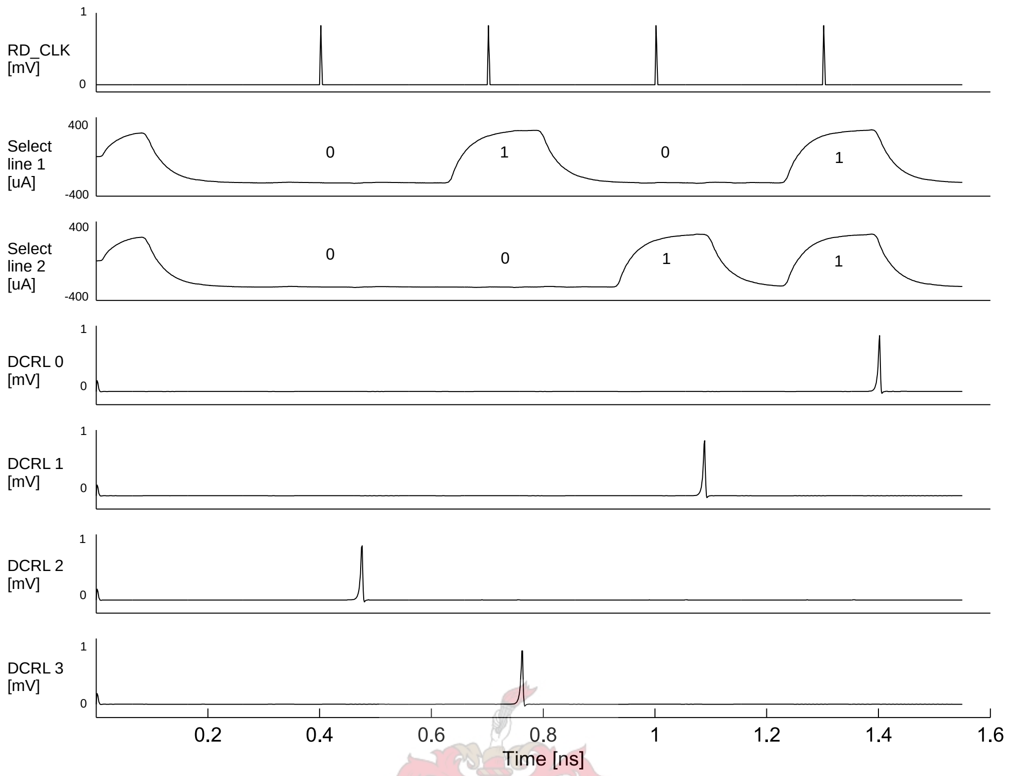

Figure 4.13: Simulation: Logic block decoder

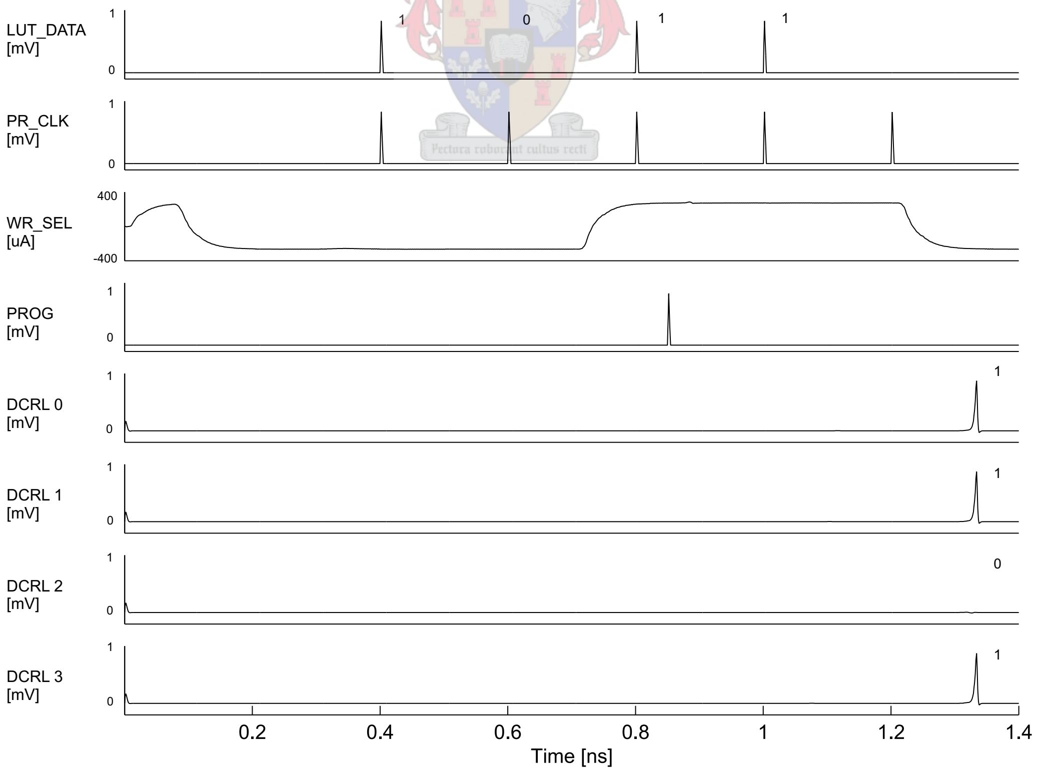

Figure 4.14: Simulation: Logic block programming sequence

4.4.4.2 Logic block program sequence simulation

For reference, Figure 4.12b shows a diagram of the programming- and memory cells sections found in the logic block. The simulation aims to to show that the logic block can be programmed correctly also indicating the signals (including timing) that are required.

The data for the memory cells (DCRLs) are shifted in serially by means of the *LUT_DATA* port, and shifting takes place with the aid of clock pulses at the *PR_CLK* port. During the last clock period (i.e. between the third and fourth clock pulses), the *PROG* port needs to be pulsed to activate the HUFFLE, which in turn allows the data that has been shifted in, to propagate to the DCRLs' *set* inputs. The *WR_SEL* signal has to be activated even before the *PROG* signal, but the timing between theses two signals is handled by the Programming Frame[5].

At 1.3ns *(READ)* signals were manually injected to trigger the DCRLs. The DCRL signals show that the output is produced correctly, with the data bits inversely corresponding to the order of the memory cells (i.e. first data bit to DCRL 3 and last data bit to DCRL 0).

4.4.5 Programming Frame

A matrix-type access scheme has been employed to gain access (and program) all the memory cells of the SPGA [**?**]. Each memory cell (DCRL) has been assigned to a cell in the matrix.

The on-chip circuitry used to implement this matrix-type access scheme is collectively known as a Programming Frame. The frame has two main sections, namely the column- and row driver sections. The column driver section is located at the top, while the row driver section is located to the left. A corner section connects the two sections in the top-left corner.

All the switches in the routing architecture can be programmed with the Switch Programming Frame, while the lookup tables have their own LUT Programming Frame. The reason is that the data in the lookup tables are loaded in with serial shifting and requires a slightly different timing procedure that can be used with the normal switch memory cells.

The columns in the matrix are labelled with letters, while the rows are labelled with numbers. The LUT programming frame (located in the top-right corner) continues the same labelling scheme.

During the cell assignment the amount of rows were minimized because the row access lines are implemented with RSFQ lines (which are wide and difficult to route).

The diagram of the setup is shown in Figure 4.15. The shaded triangles represent row drivers, while the shaded rectangles represent the column drivers of the programming frame. Each matrix cell is indicated by a small shaded circle, and its access combination written nearby.

[5]Refer to Chapter 4.4.5 for more detail

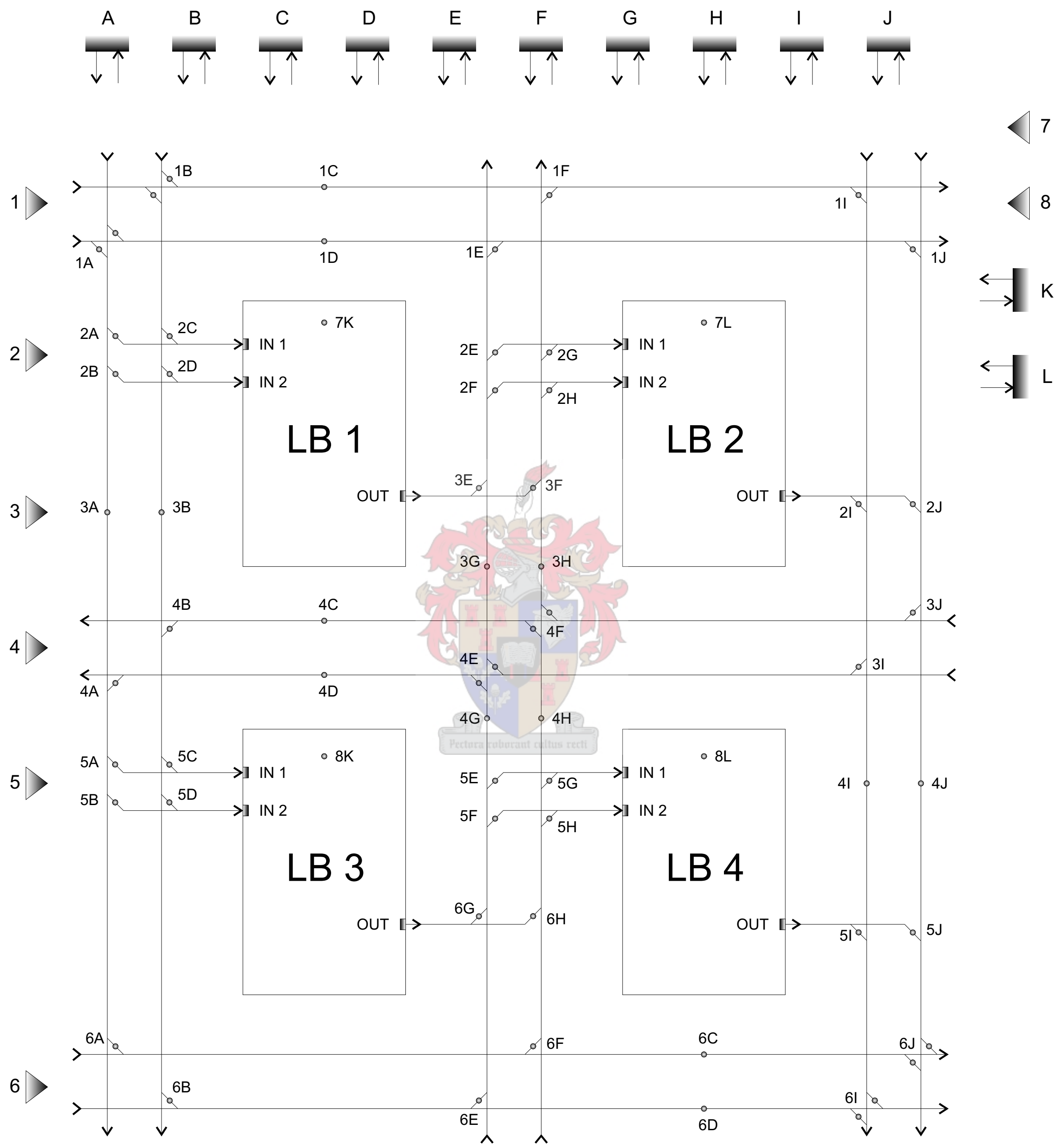

Figure 4.15: Diagram illustrating the matrix-cell assignment of the programming frames

The diagram in Figure 4.16 shows a more detailed but smaller implementation to illustrate how the programming frame works. In the diagram, the upper dotted frame encapsulates the circuitry that implements a single column driver, while the lower dotted frame that of a single row driver. These drivers can respectively be duplicated side by side to increase the capacity of the programming frame.

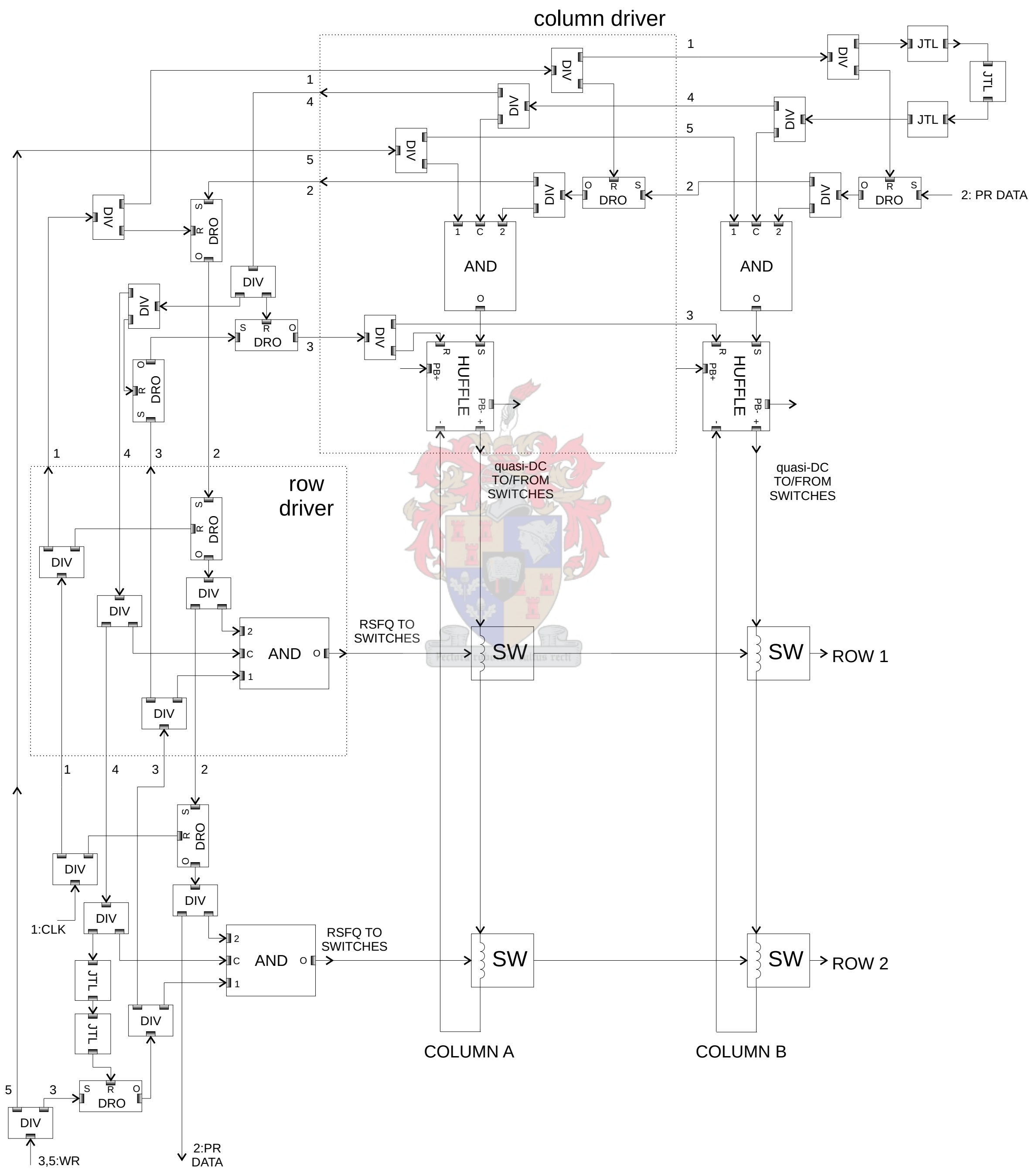

Figure 4.16: Schematic diagram: Simplified 2x2 Programming Frame

The smaller implementation can be used as an example to illustrate how the programming process works. The matrix is programmed one column at a time. The size of the programming data word is the sum of columns and rows (in this case, four). The data bits are entered in at *PR_DATA* port; row data first followed by column data. Both in order from highest to lowest (2 to 1). By asserting a respective bit in the column data, a specific column is selected. All the rows in that column are then programmed with the data in the row data bits. The programming data are shifted through a shift register with the aid of the *CLK* pin. In our case the first four clock pulses correspond to the shifting of data through the registers and will be named accordingly, *shift clock pulses* (SCPs). *WR* must be pulsed in the same clock cycle as the last data bit [6]. The last SCP shifts the last data bit in, and also moves the column bits into AND-gates that have one input set by *WR*. After a short internal delay, this clock reads the column driver AND-gates into the *set* inputs of HUFFLEs. The last SCP also releases an internally delayed *WR* pulse to propagate to the inputs of the row driver AND-gates. After the last SPC, the next clock pulse ensures that the row bits and the delayed *WR* signal gets processed by the AND-gates. This in turn releases the row SFQ pulses into the switch matrix. The next clock pulse resets any HUFFLEs.

Figure 4.17 shows a diagram of how the bits should be ordered. Row bits first (R2 and R1) followed by column bits (CA and CB). One has to remember that only one column can be programmed at a time, so the process has to be repeated as many times as there are columns.

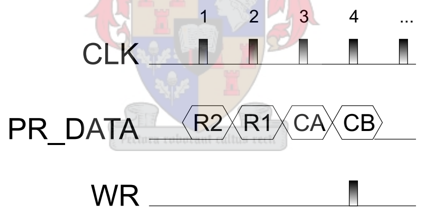

Figure 4.17: Programming frame bit sequence

This serial loading of data has the added advantage that it minimizes the amount of pins needed on the die.

4.5 SPGA

By combining the circuits as illustrated in Figure 5.14, the SPGA is finally realized. The structure in the figure represents the architecture of the final design. The reasons for certain design choices have been discussed previously and the others will become apparent later.

[6]The last SCP corresponds to the last data bit

One may identify some familiar blocks (that were discussed in this chapter) in the figure and some new blocks (such as the switch matrices at the input and output of the logic blocks) which are obvious extensions of previous ideas.

In the final design there is a total of 4253 junctions and the total bias current is 560.48 mA. These numbers were computed after consideration for added blocks during the physical layout phase (such as JTL- and MSL gates).

4.6 Functional Verilog simulation of the SPGA

SPICE simulations of large circuits are slow and memory intensive leading to problems with functional simulation of medium- to large scale circuits. The *Verilog Hardware Description Language* (Verilog HDL) can be employed to simulate large and complex RSFQ circuits [39]. By using Verilog to describe the functional (and/or timing) behaviour of RSFQ gates or RSFQ blocks, a high-level functional abstraction is achieved.

Most of the research has been done with regard to timing and delays [39]. Although timing and delays are very important in large scale circuits, especially with regard to clock skew and asymmetrical delays, the emphasis of the Verilog models in this project was on functional behaviour only. The reasons being that we were only interested in the behavioural logic operation of the circuit as a whole, and did not have enough time to go through the exhaustive process of extracting the timing parameters from the RSFQ gates and blocks.

4.6.1 Functional models

The diagram in Figure 4.18 provides the framework from which the functional models of the blocks can be extracted. In total, 6 functional blocks can be identified and are indicated by dotted frames named above the right-hand corner. The whole circuit is composed of rotated and/or mirrored copies of these blocks.

4.6.1.1 Switch Matrix 1 (SM1)

The switch matrix in the upper-left corner of the design is also known as a crossbar matrix. Looking at the block more closely in Figure 4.19, the left-hand side denotes the horizontal input (h_in), and the right-hand side the horizontal output (h_out). The same holds for the top- and bottom sides, v_in and v_out which represent the vertical flow. The four small circles indicate switch elements that control the flow of data. There are two clusters, one to the left and one to the right, each with two elements. Each cluster (encircled) is controlled by one memory element. When a memory cell is active and contains a boolean ‘1’, the switches are closed, connecting the vertical and horizontal lines in the manner shown in the diagram[7].

[7]Note: lines crossing perpendicularly are not connected

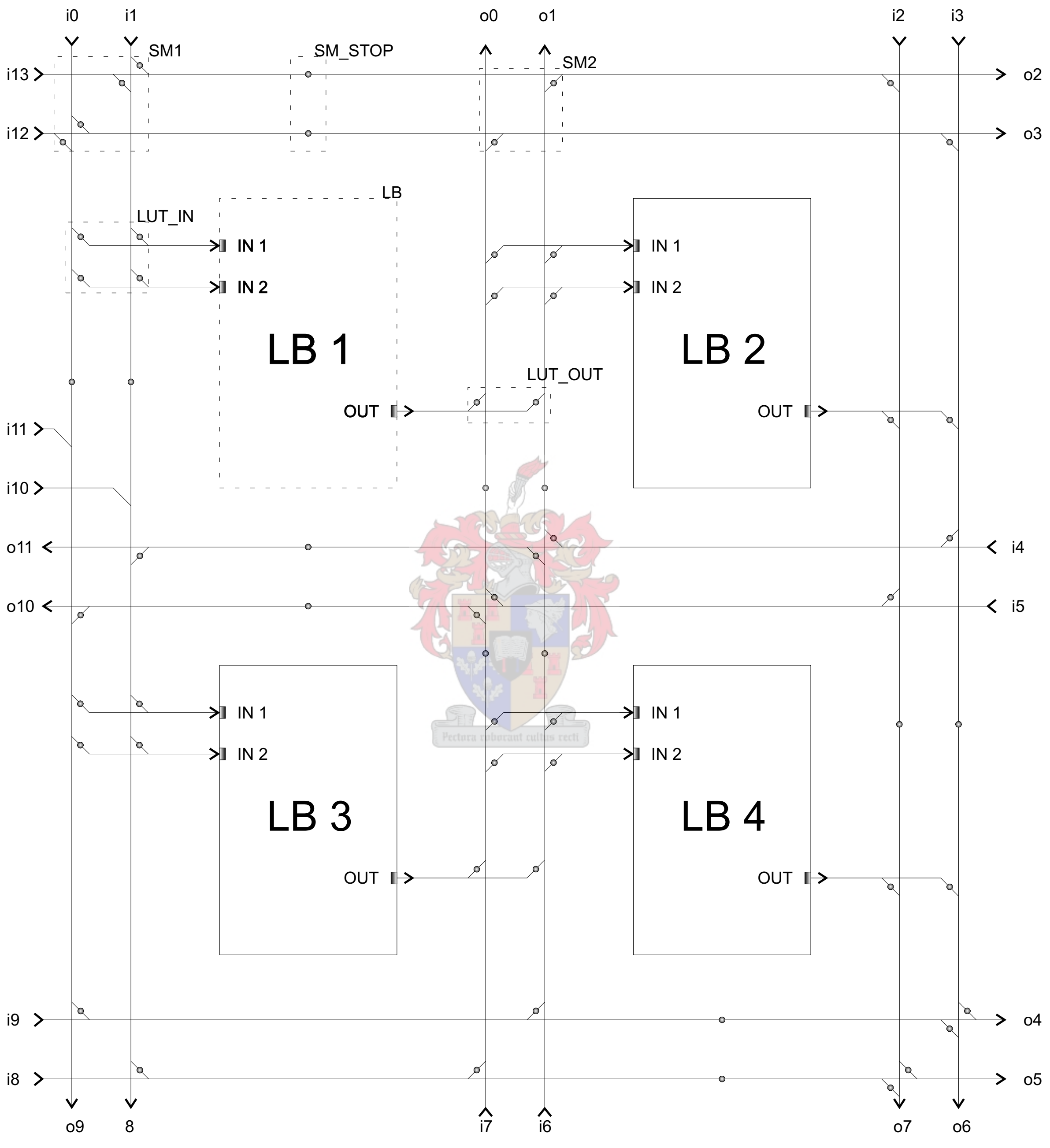

Figure 4.18: Symbolic diagram used as framework for Verilog simulations

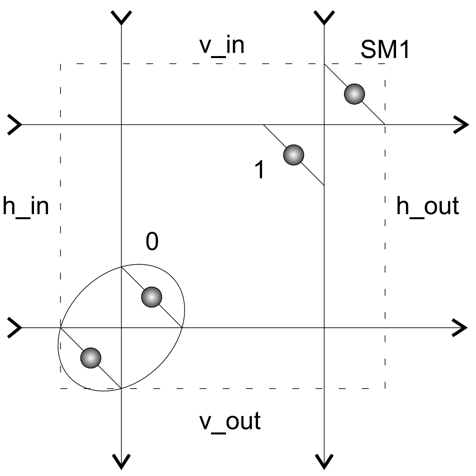

Figure 4.19: Symbolic diagram: Switch Matrix 1 (Crossbar matrix) (SM1)

Using this description, it is possible to compose a Verilog module that emulates this behaviour:

```
module sm1 ( pr_h_set, pr_v_set, pr_d, reset, h_in, v_in, h_out, v_out);

input pr_h_set, pr_v_set;
input [1:0] pr_d;
input reset;
input [1:0] h_in, v_in;
output [1:0] h_out, v_out;
reg [1:0] h_out, v_out;
reg [1:0] SRAM;
wire pr_clk;

assign pr_clk = pr_h_set && pr_v_set;

always @(posedge reset or posedge pr_clk)
begin
  if (reset)
    SRAM = 2'b00;
  else
    SRAM = pr_d;
end

always @(h_in or v_in)
begin
  h_out[0] = h_in[0] || (v_in[1] && SRAM[1]);
  h_out[1] = h_in[1] || (v_in[0] && SRAM[0]);
  v_out[0] = v_in[0] || (h_in[1] && SRAM[0]);
  v_out[1] = v_in[1] || (h_in[0] && SRAM[1]);
end

endmodule
```

The input parameters *pr_h_set* and *pr_v_set* are used during the programming stage[8] to allow indivudual access to each block. The *pr_d* parameter is used to inject the memory cell data into the *SRAM* register (or variable). The *SRAM* register represents the memory cells in each block. The *reset* signal is used as an asynchonous reset to all the blocks.

4.6.1.2 Switch Matrix 2 (SM2)

The switch matrix in the upper-middle part of the design basically works on the same principal as the crossbar matrix, but not all the lines are connected. Figure 4.20 gives a closer view and more detail. The Verilog code is given in Appendix A.

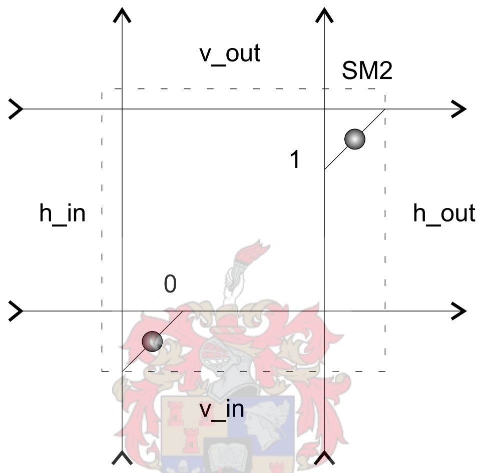

Figure 4.20: Symbolic diagram: Switch Matrix 2 (SM2)

4.6.1.3 Stop Switch Matrix (SM_STOP)

The switches in the *Stop Switch Matrix* can prevent throughput. If the switch is active the lines are connected. Figure 4.21 give more detail and the code is in Appendix A.

4.6.1.4 LUT_IN

The switch matrix to the left of any logic block provides access from routing lines to the logic block inputs. The matrix contains 4 switches to allow any input combination of the available data lines[9]. Figure 4.22 provides more detail while the code is in Appendix A.

[8]Refer to Chapter 4.6.2 for more infomation

[9]This routing configuration allows the implementation of a virtual OR function in the routing architecure itself. By enabling switches 0 and 1, the resulting output signal from the top *h_out* port is then the OR of both *v_in* signals

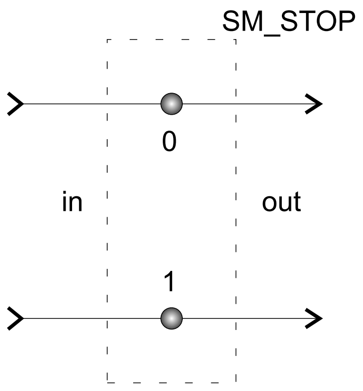

Figure 4.21: Symbolic diagram: Stop Switch matrix (SM_STOP)

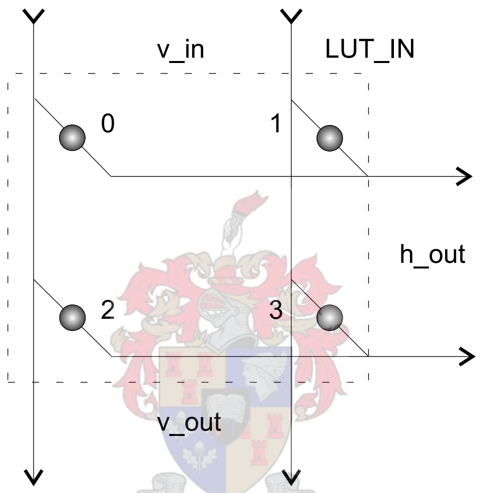

Figure 4.22: Symbolic diagram of the LUT_IN switch matrix

4.6.1.5 LUT_OUT

The logic block output is re-integrated into the routing architecture by means of the LUT_OUT switch matrix. See Figure 4.23 and Appendix A for more detail about the implementation.

4.6.1.6 Logic Block (LB)

The logic block architecture has been described in Chapter 3, and in essence contains a clocked lookup table. The logic block contains four memory cells. The contents of the memory cells correspond to the output of the truth table that describes the behaviour of the logic block. For example, if both inputs are '0', the content of the first memory cell is output. If input 1 is '1' and input 2 is '0' then the second memory cell's contents is output, and so forth. Output is produced when the data clock (d_clk) is asserted. The Verilog code is given in Appendix A.

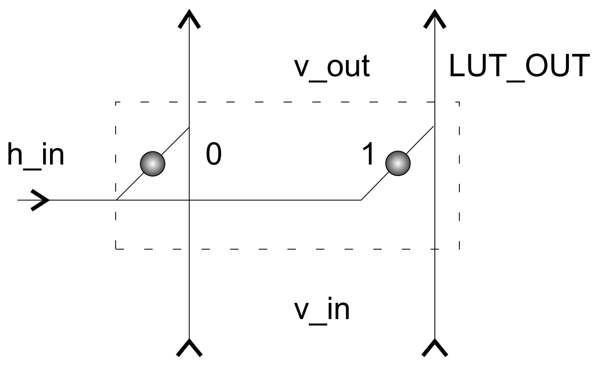

Figure 4.23: Symbolic diagram of the LUT_OUT switch matrix

4.6.1.7 Verilog SPGA model

By structuring the blocks of Verilog code (modules) so that it implements the architecture shown in Figure 4.18, a functional abstraction of the SPGA[10] was created. This model can now be used to demonstrate the overall working of the circuit.

4.6.2 Programming and simulation

A matrix-type access scheme has been employed to program all the memory cells (both in the switch matrices and logic blocks). Two vectors *pr_h_set* and *pr_v_set* have 9 and 5 bits respectively. Each bit represents a row or column in the horizontal- or vertical direction respectively. Vertical columns are marked *v1* to *v4* while horizontal rows are marked *h0* to *h8* in Figure 4.24. When one bit in each vector is asserted to access a defined module, the data in the *pr_d* vector is read into that specific module's memory cells. The logic block is the only exception, where *pr_clk* must also be asserted to load the data into the memory. Parallel (vs. serial) data loading from the *pr_d* vector is used to avoid unnecessarily long simulation setup and programming times. The relation of the bits in the *pr_d* vector to memory cells in each block is indicated by the small numbers in each block in Figure 4.24.

As an example, the boolean logic functions in (4.6.1)-(4.6.2) were mapped into the SPGA Verilog model and simulated.

$$O_1 = (A \text{ XOR } B) \text{ OR } (C \text{ AND } D) \qquad (4.6.1)$$

$$O_2 = (C \text{ AND } D) \text{ NAND } E \qquad (4.6.2)$$

[10]Refer to Appendix A for the Verilog code

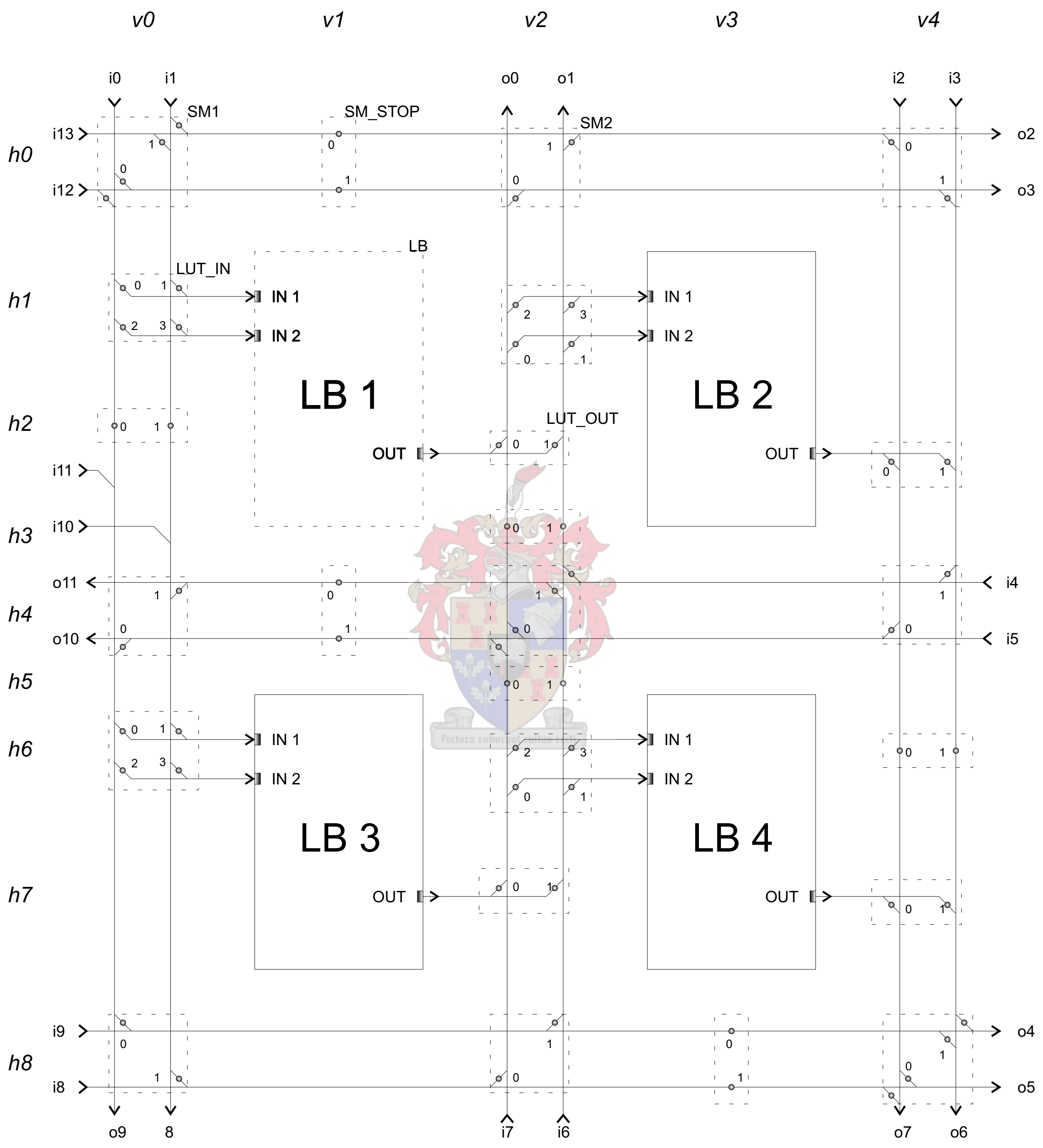

Figure 4.24: Symbolic diagram illustrating Verilog programming matrix

The lookup tables are as follows[11]:

$$\begin{aligned} L_1 &= A \text{ XOR } B \\ L_2 &= L_1 \text{ OR } L_3 \\ L_3 &= C \text{ AND } D \\ L_4 &= L_3 \text{ NAND } E \end{aligned}$$

The truth table for three test cases is given in Table 4.3.

Table 4.3: Partial truth table for (4.6.1)-(4.6.2)

	input							**O_1**	**O_2**
	A	B	C	D	E	A XOR B	C AND D	L_1 OR L_3	L_3 NAND E
a	0	1	0	1	0	1	0	1	1
b	1	0	0	1	1	1	0	1	1
c	1	1	1	0	0	0	0	0	0

An easy way to program the memory cells is to step from the *h0* to *h8* rows while stepping from *v0* to *v5* for each row. It this way all the possible cases are covered. Thus, Table 4.4 shows how to set up all the switches and lookup tables for the example. Input A is mapped to i0, input B to i1, input C to i11 input D to i10, output O_1 to o6 and output O_2 to o7.

Figure 4.25 shows part of the program loading sequence as done in QuartusII [40]. The figure contains 5 signals used during the loading sequence, namely *reset*, *pr_clk*, *pr_h_set*, *pr_v_set* and *pr_d*. The reset is pulsed at the beginning to initialize all the blocks in a known state. The programming clock (*pr_clk*) is used to load the data from the programming data bus (*pr_d*) into the memory of the logic blocks. The block-selection variables (*pr_h_set* and *pr_v_set*) are used to allow access to all the blocks in the virtual matrix configuration.

Figure 4.26 shows the results of the simulations for the three input cases discussed previously. The figure contains one clock signal (*d_clk*) and two data buses (*d_in* and *d_out*). The Verilog model was designed to process only at positive-edge clock changes. The data buses have been broken into bit-pairs for clarity and correspond to port assignments illustrated in Figure 4.24. In the simulation figure (Figure 4.26) the input data bits, *d_in[0]* and *d_in[1]*, correspond to the afore mentioned example inputs, A and B. The rest of the example input and output assignments are shown just to the right of the respective data bit-pairs. The simulation shows the resulting operational output bits (O_1 and O_2) for three cases of input stimulation and correspond the truth table (Table 4.3) of the function that was programmed into the Verilog SPGA model.

[11] L_1-L_4 refers to logic blocks LB1 - LB4

Table 4.4: Programming sequence to implement functions (4.6.1)-(4.6.2)

pr_h_set	pr_v_set	pr_d	comment
000000001 (h0)	00001 (v0)	0000	SM1
	00010 (v1)	0000	SM_STOP
	00100 (v2)	0000	SM2
	01000 (v3)	0000	
	10000 (v4)	0000	SM2
000000010 (h1)	00001 (v0)	1001	LUT_IN
	00010 (v1)	0110	LB1
	00100 (v2)	0110	LUT_IN
	01000 (v3)	1110	LB2
	10000 (v4)	0000	
000000100 (h2)	00001 (v0)	0000	SM_STOP
	00010 (v1)	0000	
	00100 (v2)	0001	LUT_OUT
	01000 (v3)	0000	
	10000 (v4)	0010	LUT_OUT
000001000 (h3)	00001 (v0)	0000	
	00010 (v1)	0000	
	00100 (v2)	0010	SM_STOP
	01000 (v3)	0000	
	10000 (v4)	0000	
000010000 (h4)	00001 (v0)	0000	SM2
	00010 (v1)	0000	SM_STOP
	00100 (v2)	0000	SM1
	01000 (v3)	0000	
	10000 (v4)	0000	SM2
000100000 (h5)	00001 (v0)	0000	
	00010 (v1)	0000	
	00100 (v2)	0010	SM_STOP
	01000 (v3)	0000	
	10000 (v4)	0000	
001000000 (h6)	00001 (v0)	1001	LUT_IN
	00010 (v1)	1000	LB3
	00100 (v2)	0110	LUT_IN
	01000 (v3)	0111	LB4
	10000 (v4)	0010	SM_STOP
010000000 (h7)	00001 (v0)	0000	
	00010 (v1)	0000	
	00100 (v2)	0010	LUT_OUT
	01000 (v3)	0000	
	10000 (v4)	0001	LUT_OUT
100000000 (h8)	00001 (v0)	0000	SM2
	00010 (v1)	0000	
	00100 (v2)	0000	SM2
	01000 (v3)	0000	SM_STOP
	10000 (v4)	0000	SM1

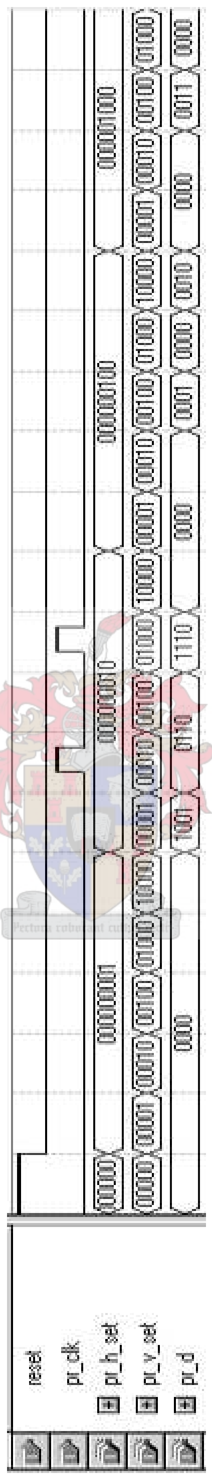

Figure 4.25: Verilog simulation: Partial program loading sequence

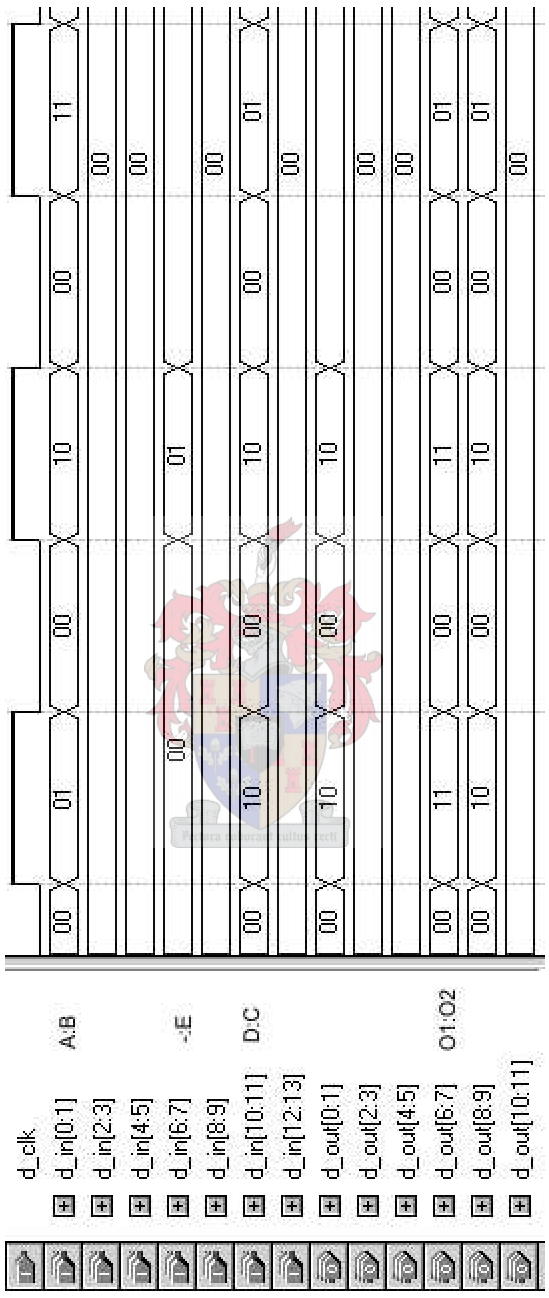

Figure 4.26: Verilog simulation: Operational input and output results

4.7 Chapter summary

This chapter encapsulates the schematic circuit design of the SPGA, which includes descriptions of gates used to compose larger functional blocks and ultimately construct the first prototype. SPICE simulation results were included in this chapter to substantiate the operation of gates and smaller blocks. Ultimately, a functional Verilog simulation was introduced to showcase the overall operation of the SPGA.

Chapter 5

Physical layout

'The six stages of production:
- Wild enthusiasm
- Total confusion
- Utter despair
- The search for the guilty
- The persecution of the innocent
- The promotion of the incompetent.

No project was ever completed on time and within budget.'

[Anonymous - Cheops law]

LAYOUT design may be defined as the process of drawing an accurate physical representation of an engineering circuit schematic (netlist) that conforms to constraints imposed by the manufacturing process, the design flow, and the performance requirements shown to be feasible by simulation [41].

ICs are made using an extremely complicated process that results in tiny elements and wires constructed and connected on a substrate. Layout design is the art of drawing these elements and wires and can be thought of as a depiction of the physical circuit.

The layout CAD tool used during this project is LASI (LAyout System for Individuals) [42] and is a 'general purpose' IC layout and design system.

The fabrication process from Hypres Inc. [1] was used.
Niobium is used as the superconducting material due to its high critical temperature, electrical and thermal stability, and ability to be thermally cycled many times without degradation. Josephson junctions are trilayer Niobium/Aluminum-Oxide/Niobium tunnel junctions which are defined with a photolithography and etching process. This method yields good uniformity and reproducibility of junction parameters. One resistive layer of molybdenum has a resistivity of 1 Ohm per square at 4.2K and the other layer of palladium gold alloy is used for contact pad metallization and can be used to provide low value resistors of

0.02 Ohm per square. Silicon dioxide is deposited between junctions, resistors, ground plane and different wiring layers to provide isolation.

The physical process specifications are provided in Table 5.1 and is listed in process flow sequence (M0 deposited first and R3 last).

Table 5.1: Physical layer specifications of the Hypres process

Layer	**Bias** $[\mu]$	**Comments**	**Thickness [nm]**	**Deviation [nm]**
M0	0.2 ± 0.25	Nb. Penetration depth λ_L = 100nm ±5 %	100	±10
I0	0.2 ± 0.25	SiO_2. Contact (via) between M1 and gnd plane	150	±15
M1	-0.3 ± 0.25	Trilayer base electrode. $\lambda_L \sim$ 100nm	135	±10
I1A	-0.27 ± 0.05	Counter electrode (junction area) definition	45	±5
SiO_2		SiO_2, insulator	100	±10
R2	0.0 ± 0.25	Mo Resistor - 1.0Ω/sqr ±20%	100	±20
SiO_2		SiO_2, insulator	100	±10
I1B	-0.1 ± 0.05	Contact hole (M2 to I1A/R2/M1)		
M2	-0.2 ± 0.25	Nb. λ_L = 90nm ±5 %	300	±20
SiO_2		SiO_2, insulator	500	±40
I2	0.1 ± 0.25	Contact (via) between M2 and M3		
M3	-0.3 ± 0.25	Nb. λ_L = 90nm ±5 %	600	±50
R3	0.0 ± 1.00	Ti/PdAu Resistor - 0.02Ω/sqr	350	±60

The layout representations in this thesis are given in top-view format (as typically used in layout programs). A side-on-view diagram is shown in Figure 5.1 which illustrates how layers fit onto each other.

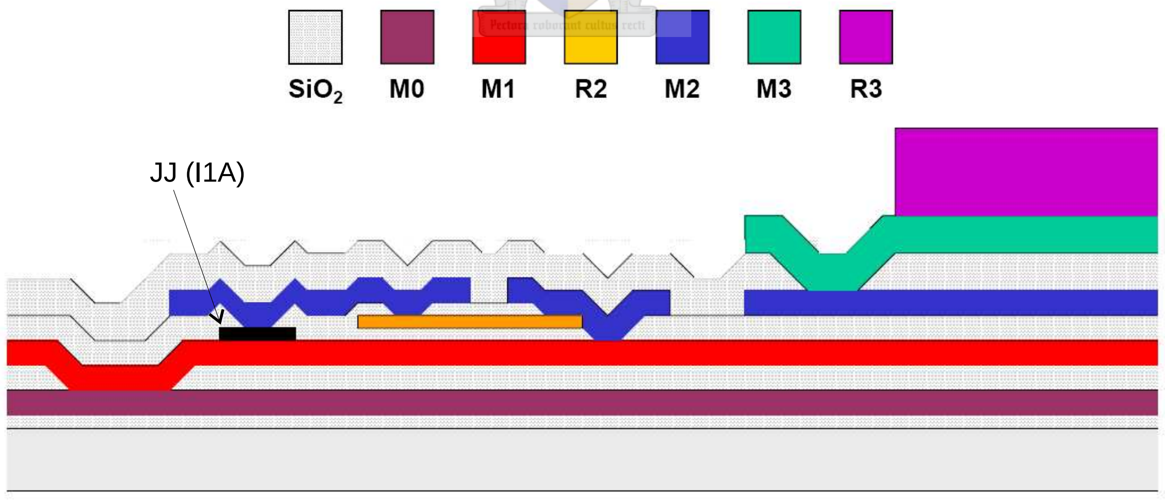

Figure 5.1: Side-on representation of Hypres' process flow [1]

The layout of a typical Josephson junction is presented in Figure 5.2. The figure shows a) the square shape of a 250μA junction and b) a 250μA resistively-shunted junction with a 1.21Ω resistor. The area of the I1A definition specifies the critical current of the junction[1].

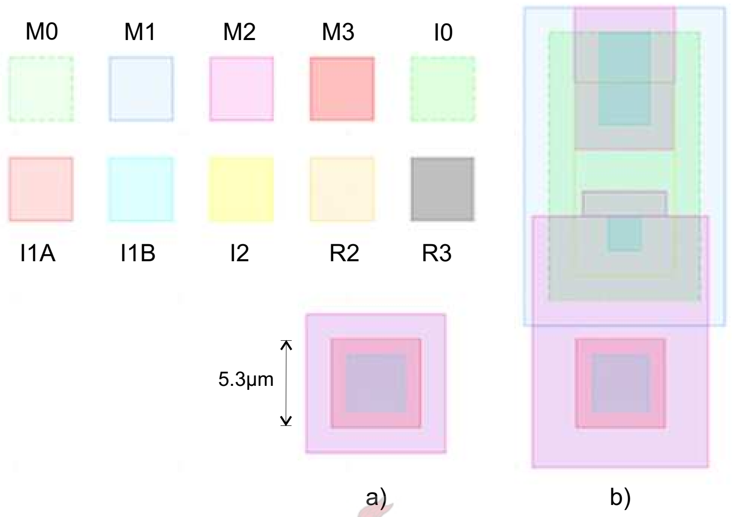

Figure 5.2: Layouts of a) 250μA Josephson junction and b) 250μA Josephson junction with grounded resistor

5.1 Hierarchical layout design approach

In LASI, a layout is drawn in a *cell*. A cell may contain the following objects: boxes, paths, polygons, text and other cells. Associated with each object, is a rank number. Boxes, paths, polygons and text objects always have the lowest possible rank, namely zero. Cells may only contain objects of lower rank. An important consequence is that cells may contain cells of lower rank. This mechanism enables a designer to create large and complex (high ranking) cells by using less complex (lower ranking) cells and objects as building blocks. It also eases the process of making repetitive changes. With regards to circuits, basic RSFQ gates are drawn in low rank cells and then easily used to draw more complex circuits.

5.2 Basic gates

Basic gates usually contain only a few Josephson junctions, resistors and inductors. These gates cannot be broken into smaller functional circuits. In this section a discussion regarding the layout of some of these gates follows while the other gates are presented in Appendix D.

[1]The calculation must take into account the photolithography effect that results in a so called 'missing area' specified by the process data sheet [1]

5.2.1 AND gate

The AND gate is an amalgamation of adapted gates which include two DROs, a divider and a merger. The DROs and the divider were used with little change, while the merger had to be extensively adapted from its original form. The optimized circuit in Figure C.1 was used as the blueprint for the adaptation. All the necessary junctions were changed and the inductor values of the whole circuit were extensively checked and modified to reflect the behaviour of the circuit.

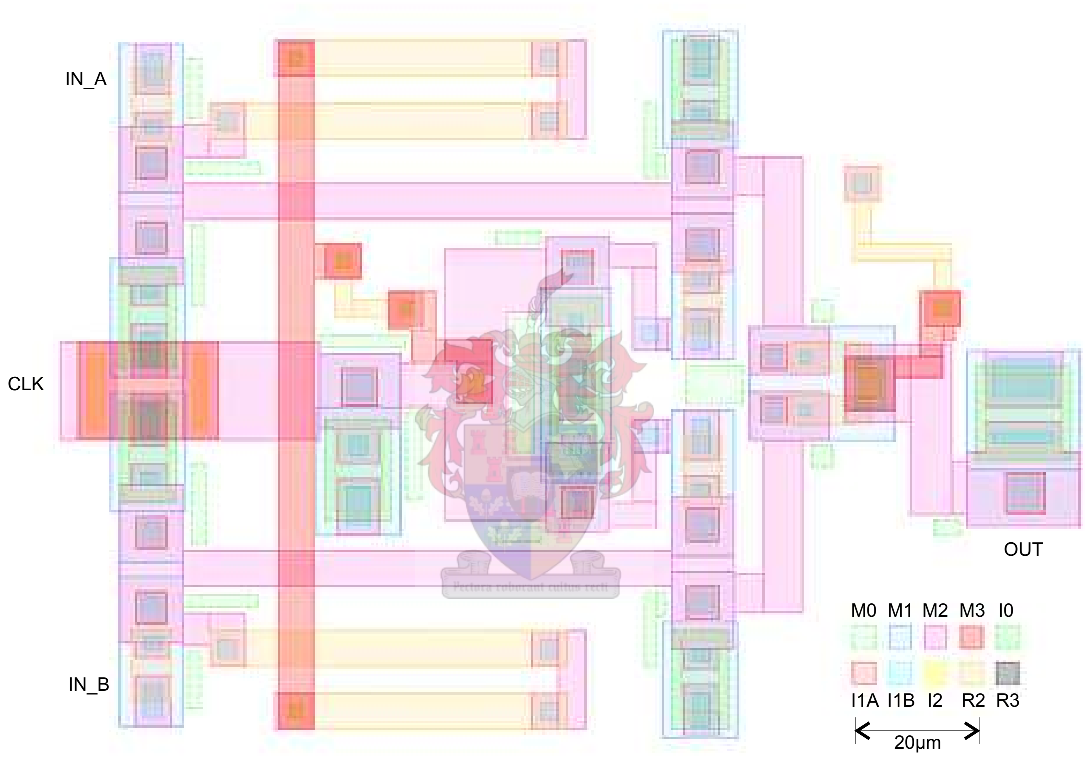

Figure 5.3: Layout of the AND gate

5.2.2 DCRL

Although the DC-Resetable Latch (DCRL) gate was discussed by Fourie [4] it may be worth noting a few changes and complications regarding the layout. There is a total of 78 DCRLs in a large variety of locations and configurations throughout the final SPGA layout. Regard the layout presented in Figure 5.4. In order to achieve a compact layout, the 15.5Ω resistor was broken up in parts *Ra* and *Rb* and connected together with an M2 line (because the *set* input is M1). Resistor *Rb* had to be swung around the *set* input to achieve a compact layout. In some other locations it was not possible to use that configuration and one other variation was to let *Rb* lie alongside the *READ* JJ in a downward facing position, as shown in Figure 5.5b. Also shown in Figure 5.5 are a few other variations.

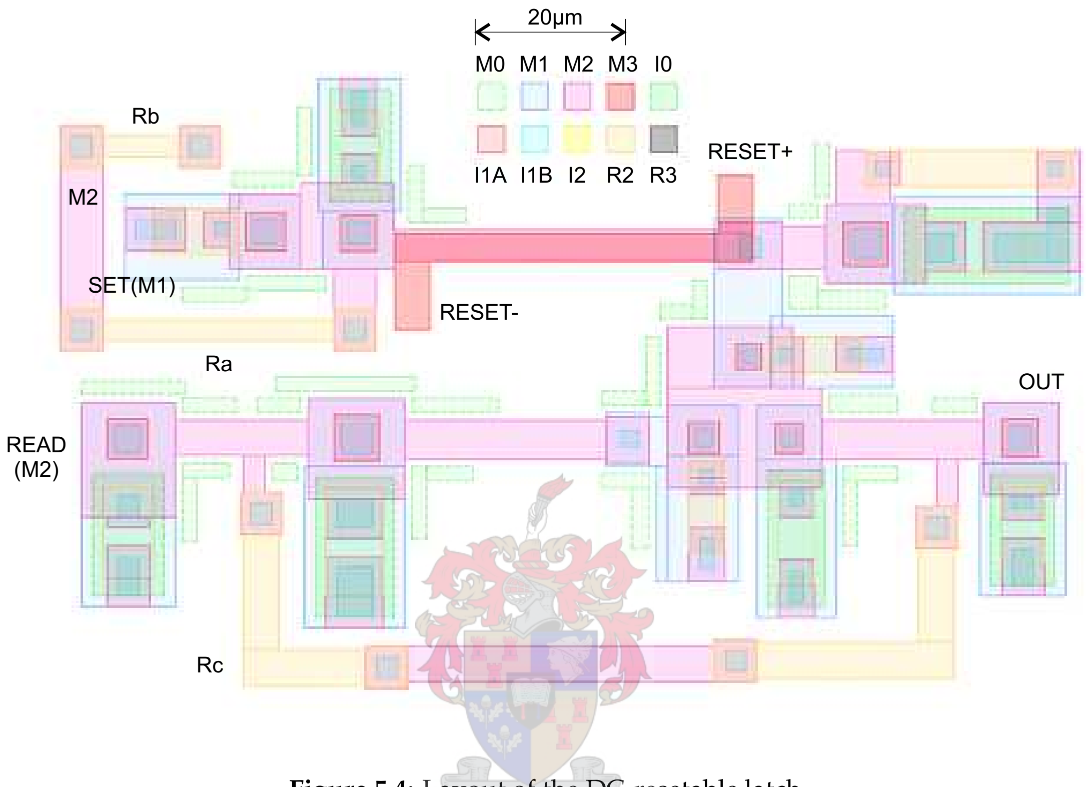

Figure 5.4: Layout of the DC-resetable latch

Other minor problems with the layout of the DCRL were, for instance, that the resistor *Rc* had to be flipped so that the dogleg lies to the left and not the right. Various configurations on how the *reset* signal was connected also posed a few problems.

5.2.3 I2-Switch

For the 2-Line Current-Set Switch (I2-Switch), basically the same layout shape was kept from the 4-line current-set switch by Fourie [4] but was adapted to use 2 lines (see Figure 5.6). The new parameters (inductance and mutual coupling) were extracted[2] and were used to re-optimize the circuit. The full circuit layout is shown in Figure 5.7. Lines *a* and *b* are the primary control lines while *bias ctrl* is the bias control line.

[2]Refer to chapter 5.6 for more detail

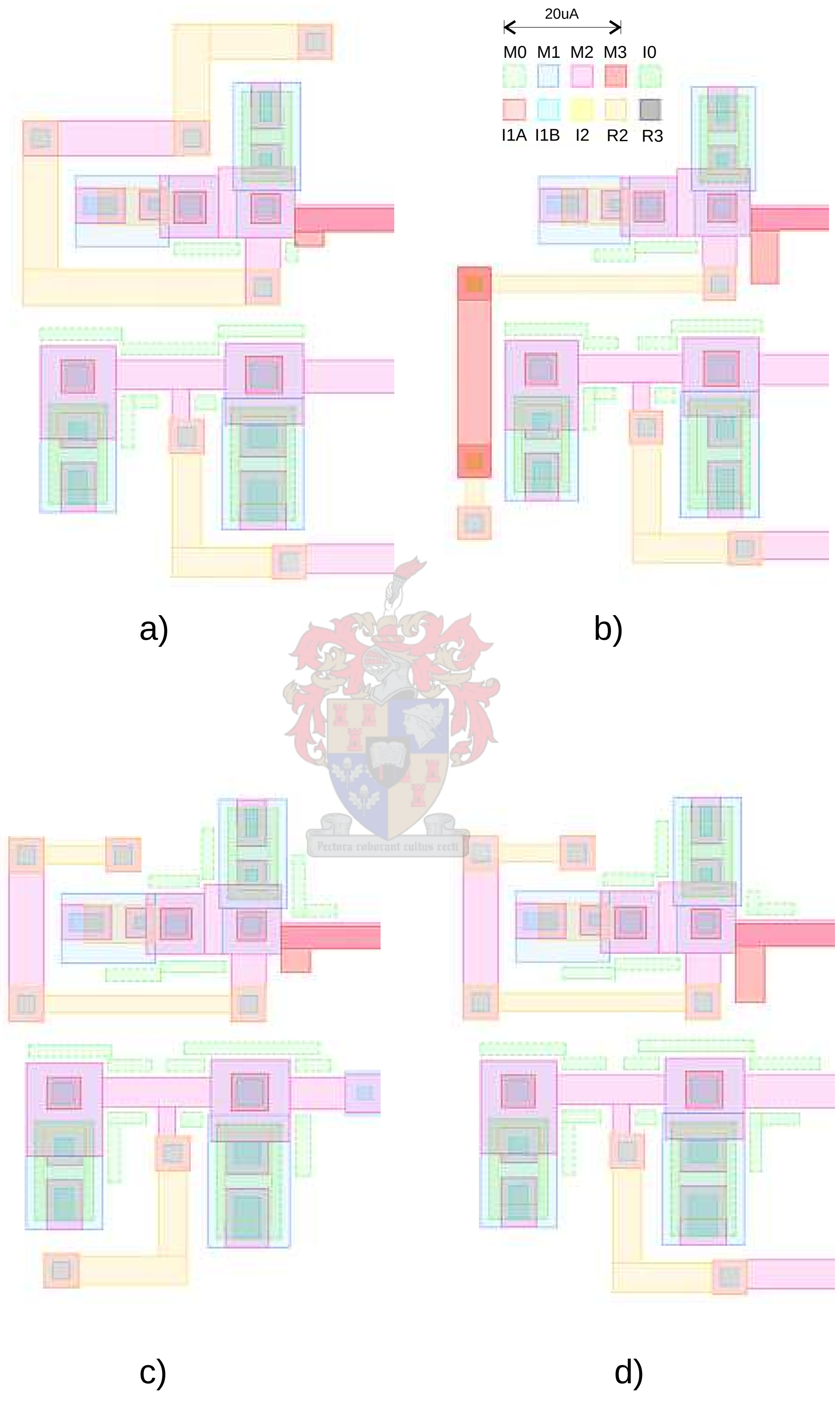

Figure 5.5: Some variations on the layout of the DC-resetable latch

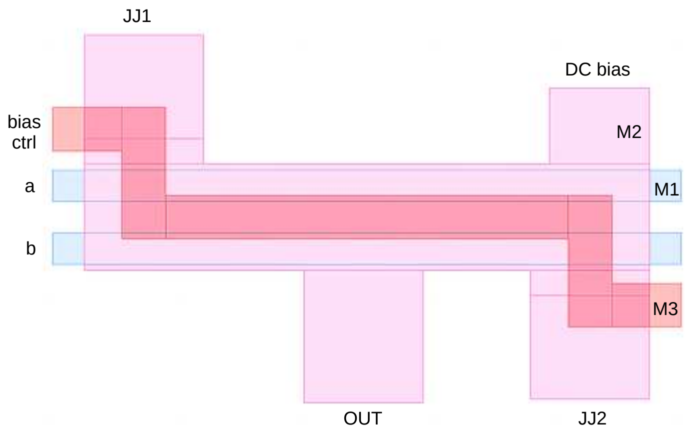

Figure 5.6: Layout of the 2-Line Current-Set Switch: bare essential layout showing control lines

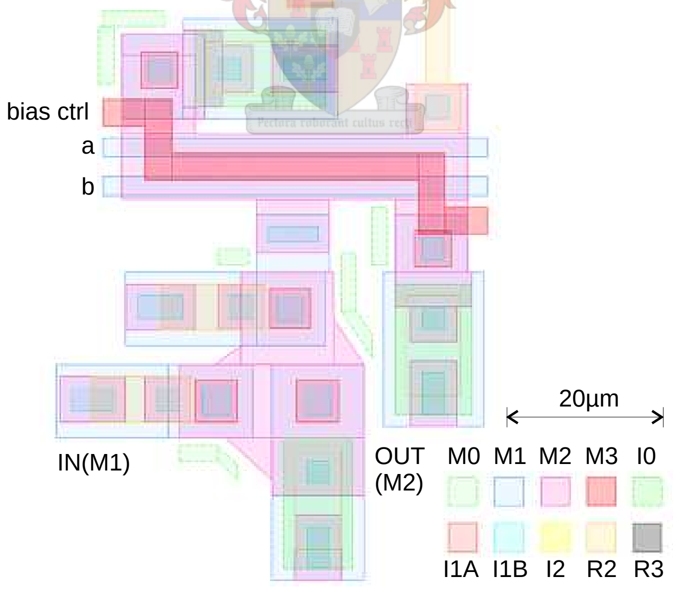

Figure 5.7: Layout of the 2-Line Current-Set Switch gate

5.3 Microstrip transmission lines

Josephson transmission lines (JTLs) are usually used for transmitting SFQ pulses (such as data- and clock signals) between circuits. However, JTLs have some disadvantages and restrictions in some cases. For example, the propagation delay becomes very large when connecting distant SFQ circuits. Furthermore, the value of the interconnect inductance is restricted in JTL circuits, which reduces the flexibility of the circuit layout [43]. One of the key features of the RSFQ family is the possibility to using superconducting passive microstrip lines for ballistic transfer of the pulses along the integrated circuits with very low power dissipation [44].

An RSFQ microstrip transmission line (MSL) driver and receiver from Stony Brook University [31], presented in Figures D.8 and D.9 (in Appendix D) respectively, were used to facilitate the transfer of RSFQ pulses between gates that are too far apart to feasibly use JTLs.

In a project of this scale and taking into account the nature of FPGA-type circuits with many routing lines, it is inevitable that lines will have to cross at some point or another. For this reason it was crucial to investigate the influence that crossing lines have on one another. The MSL driver and receiver circuits were designed to work with an M2 line. To accommodate crossings, it was decided to implement microstrip lines in both M2- and M1-layers and use a rectangular via to connect the two layers (see Figure 5.8). With the aid of *SLine* [45] it was found that a 2.3Ω M2 MSL should be 34μm wide while a 2.3Ω M1 MSL should be 19μm wide.

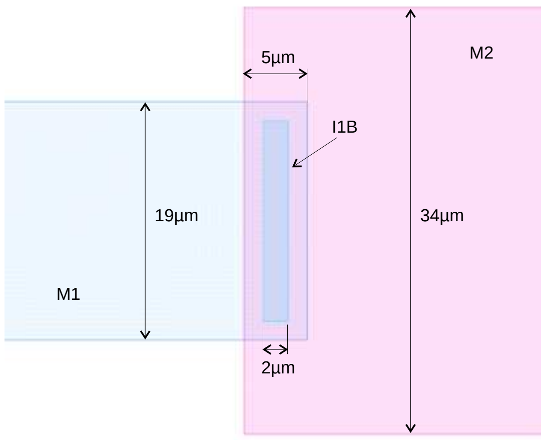

Figure 5.8: Layout: MSL via

The influence that densely packaged microstrip lines have on one another has been investigated [46] and found that microstrip lines running parallel to each other, have a negligible effect on the functioning of the circuit. On the other hand, for lines crossing perpendicularly, only the driven line's upper bias current margin was diminished.

Geometric discontinuities, such as steps, bends and T-junctions, in planar transmission lines influence the performance of a circuit by affecting the effective reactance of the line. The line can be physically compensated to reduce the effect of these discontinuities. A 90° bend can be compensated by chamfering the corner [47]. It was found that the percentage of chamfer required remains constant, independent of dielectric constant, width and the like. The chamfering involves removing a right-angled triangle corner and the optimal value of the smaller sides of the triangle equals 0.828 times the width, W, of the line. Figure 5.9 shows typical MSL structures used in this project, including a bend, two vias and an M1 line going underneath a M2 line.

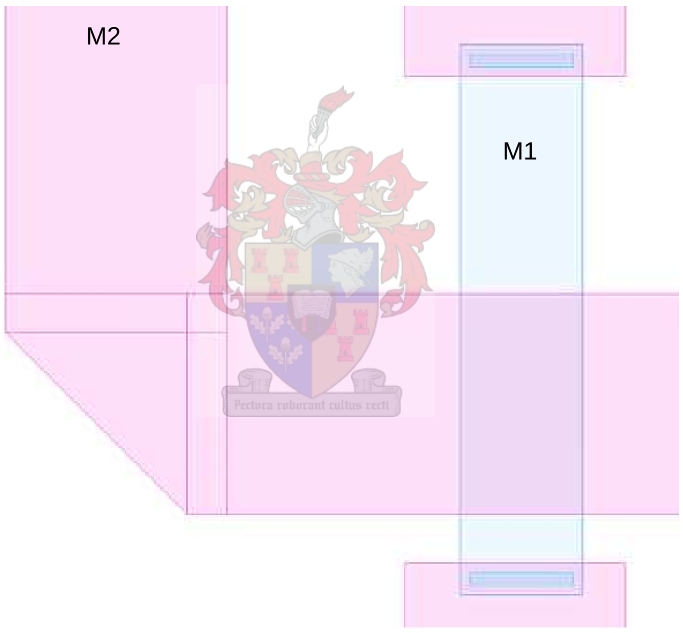

Figure 5.9: Layout: MSL structures

5.4 Composite blocks

Composite blocks are composed of a combination of basic gates and other blocks. The structure of most of the composite blocks can be seen in the diagrams from section 4.4 and the layout is a direct implementation of those diagrams. A few of the more important or complex blocks will be discussed in this chapter while the rest are presented in Appendix D.

5.4.1 Logic Block

The layout of the Logic Block is presented in Figure 5.10 [3]. The schematic diagram is shown in Figure 4.11. The Logic Block is the largest structure [4] in the design and was used as a starting point for the floorplan. In Figure 5.10, *RD CLK* indicates the input for the read clock signal while *PR CLK* indicates the input for the programming clock signal. The *LUT DATA* input is used to program the memory cells in the lookup table while *WR SEL* and *PROG* indicate the signals used to individually access one of the four logic blocks during the programming stage.

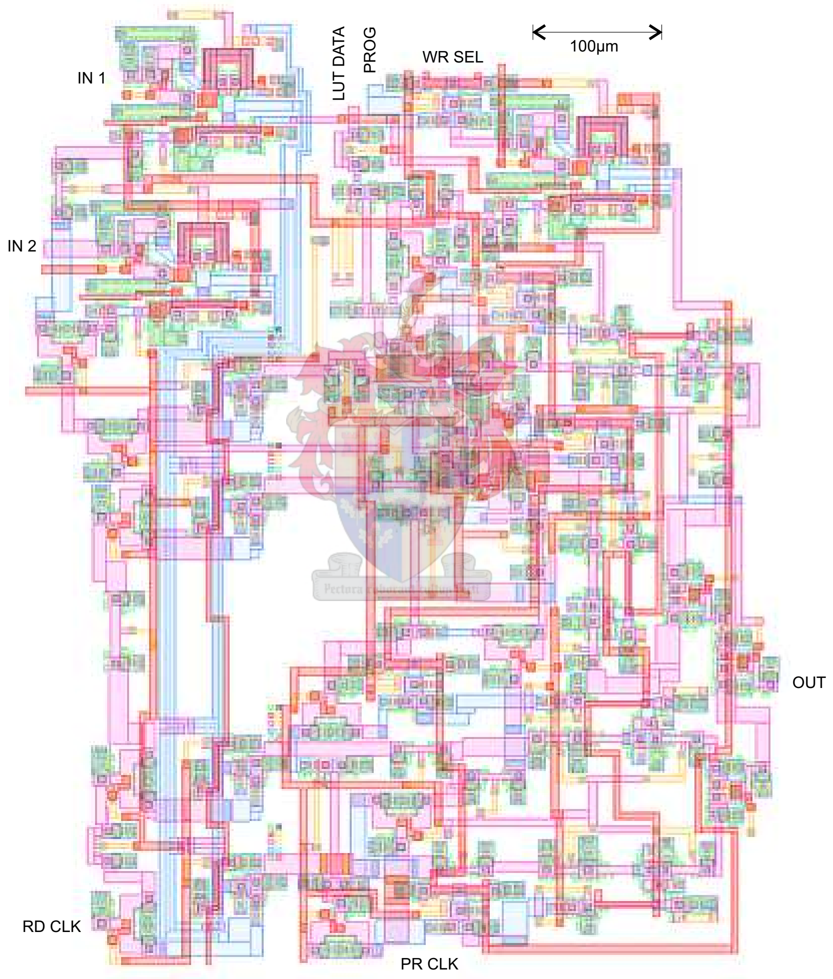

Figure 5.10: Layout of the Logic Block

[3]The layout is tightly laid out and the various sections intermix between each other and are difficult to clearly define with frames

[4]During the preliminary design iterations, 4-bit and 3-bit Logic Blocks were laid out but found to be too large to fit into the 5mm square wafer without sacrificing routing space and functionality.

5.4.2 Programming Logic Column driver

The layout of the Programming Logic Column driver block in Figure 5.11 is almost a direct implementation of the upper dotted box in the diagram of Figure 4.16 except for the fact that it was optimized to be as narrow as possible to accommodate many drivers next to each other.

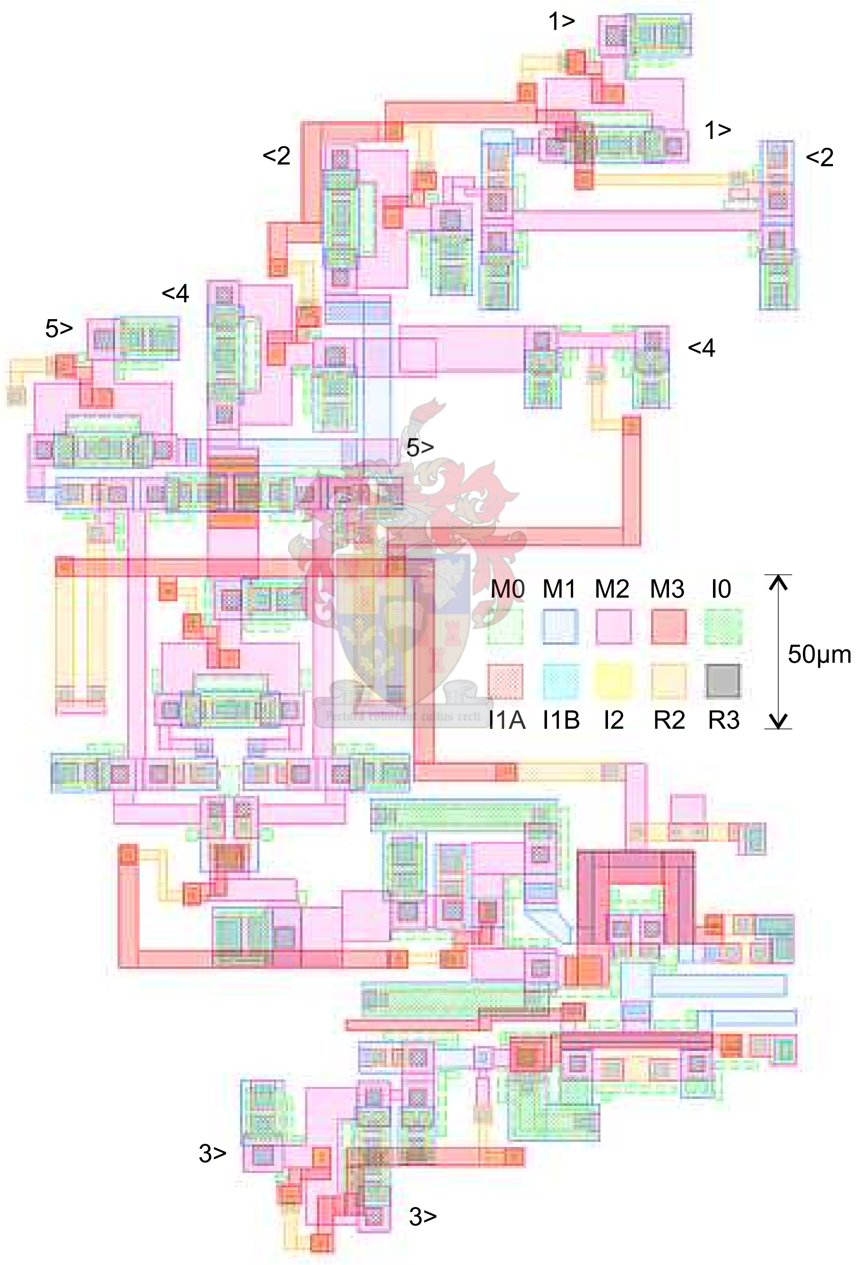

Figure 5.11: Layout of the Programming Logic Column driver

5.4.3 Programming Logic Row driver

The layout of the Programming logic (Row driver) block in Figure 5.12 posed significant problems, especially the encircled section where three potentially long lines overlap.

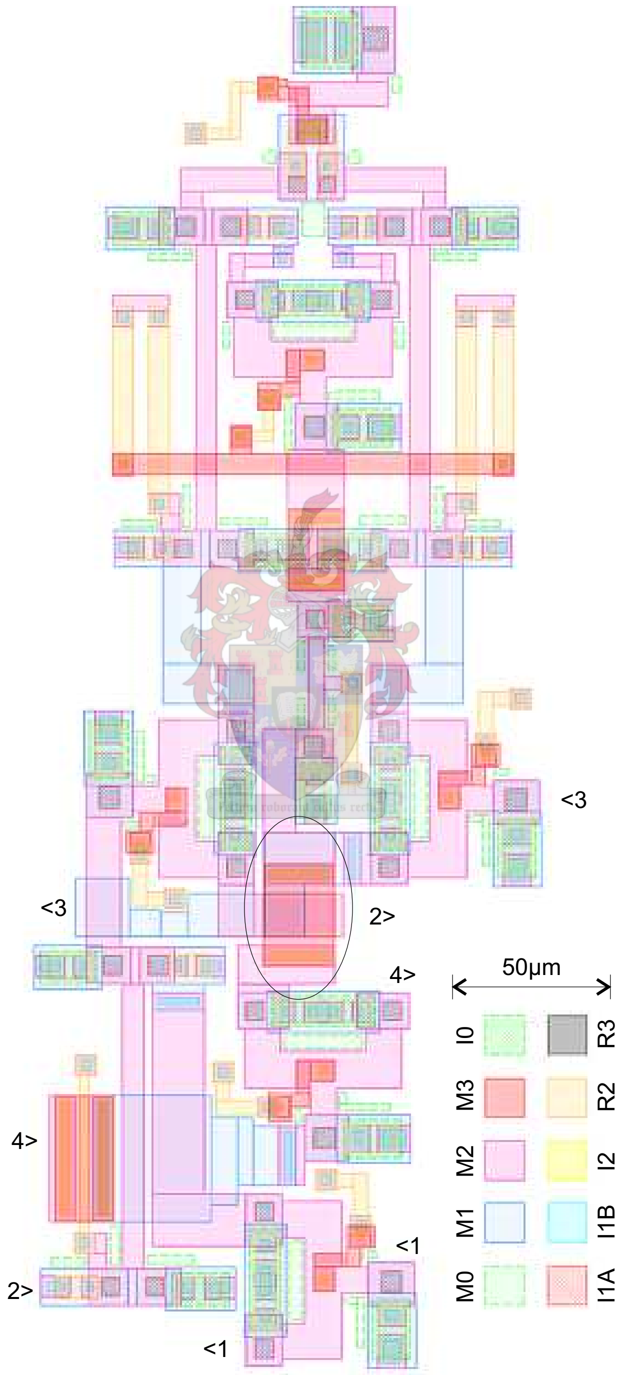

Figure 5.12: Layout of the Programming Logic Row driver

5.4.4 Full chip layout

The layout of the SPGA chip is presented in Figure 5.13 and includes connection pads. The layout is on a 5mm $\times$ 5mm die.

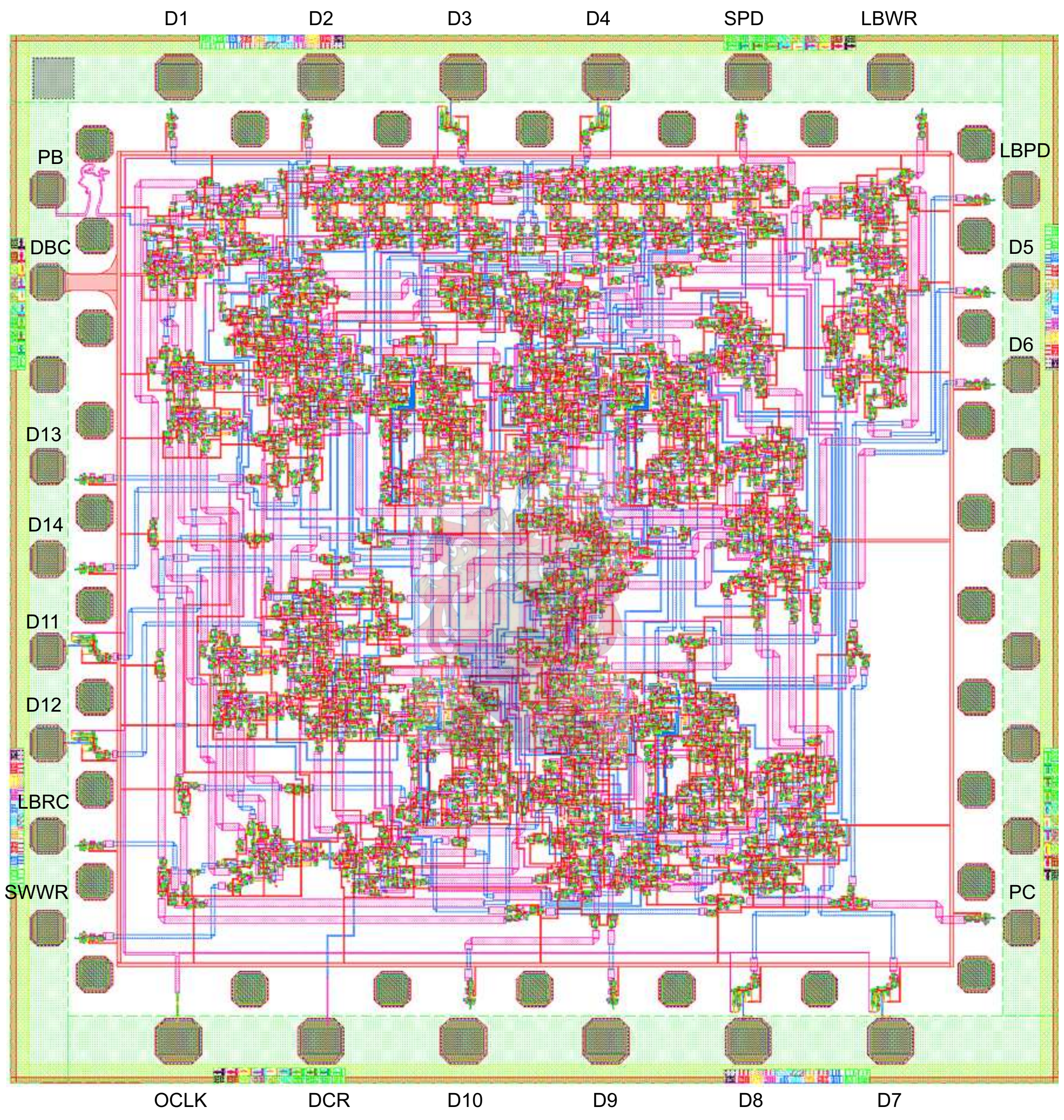

Figure 5.13: Layout: Full chip

5.5 Error checking and verification

„Es irrt der Mensch, solang'er strebt." :: 'Man errs as long as he strives.'

[Goethe - Faust (1808)]

Error checking is crucial to ensure a perfectly working design. Two factors in this design make it a tremendously difficult task. Firstly, this design is rather large and complex. Secondly, parameter extraction (for layout vs. schematic verification) is not supported by CAD tools at our disposal and thus is not an automated process. Parameter extraction by hand is difficult and time consuming which means that 100% circuit verification from the physical layout is virtually impossible. Thus an extensive set of tests and checks were used to ensure that the layout is correct.

5.5.1 Gate verification

The basic gates are the building blocks of the whole design. It is absolutely crucial that the basic gates are flawless. The basic gates are small and can be thoroughly checked and verified. The layout must accurately reflect the functionality of the circuit. All the physical sizes of the junctions and resistors were checked and verified by hand. Critical inductor values were verified with *InductEx*[5]. Overlapping and distances between layers were checked to ensure that they do not break any design rules.

5.5.2 Signal route checking

A signal route is the specified path that a signal must take to complete the circuit. The blueprint for signal routing is the SPICE circuit. The idea behind signal route checking is to ensure that a signal follows the correct route as specified by the SPICE circuit.

Simplified diagrams of the SPICE circuit were used as an aid during signal route checking. The simplified diagrams do not contain any unnecessary SPICE circuits or fine detail. The diagrams mostly consist of blocks with input and output ports connected together with lines. SPICE circuits contain detail not needed when checking signal routes, and the diagrams strip away all the detail to leave only the relevant information. Paper printouts of these diagrams with appropriate signal routes highlighted were used to keep track of on-screen signal paths.

Different diagrams were used to track signal routing:

Data routes contain the digital signals that are used for computation after the chip has been programmed. Figure 5.14 shows the diagram used to check the data routes. The small circles indicate switches used for diverting data flows. Lines crossing at 90° angles are not

[5]Refer to chapter 5.6 for more detail

connected but lines touching at 45° angles are connected. The abbreviation LB stands for *Logic Block*. Places where two or more data routes cross are called switch matrices. The

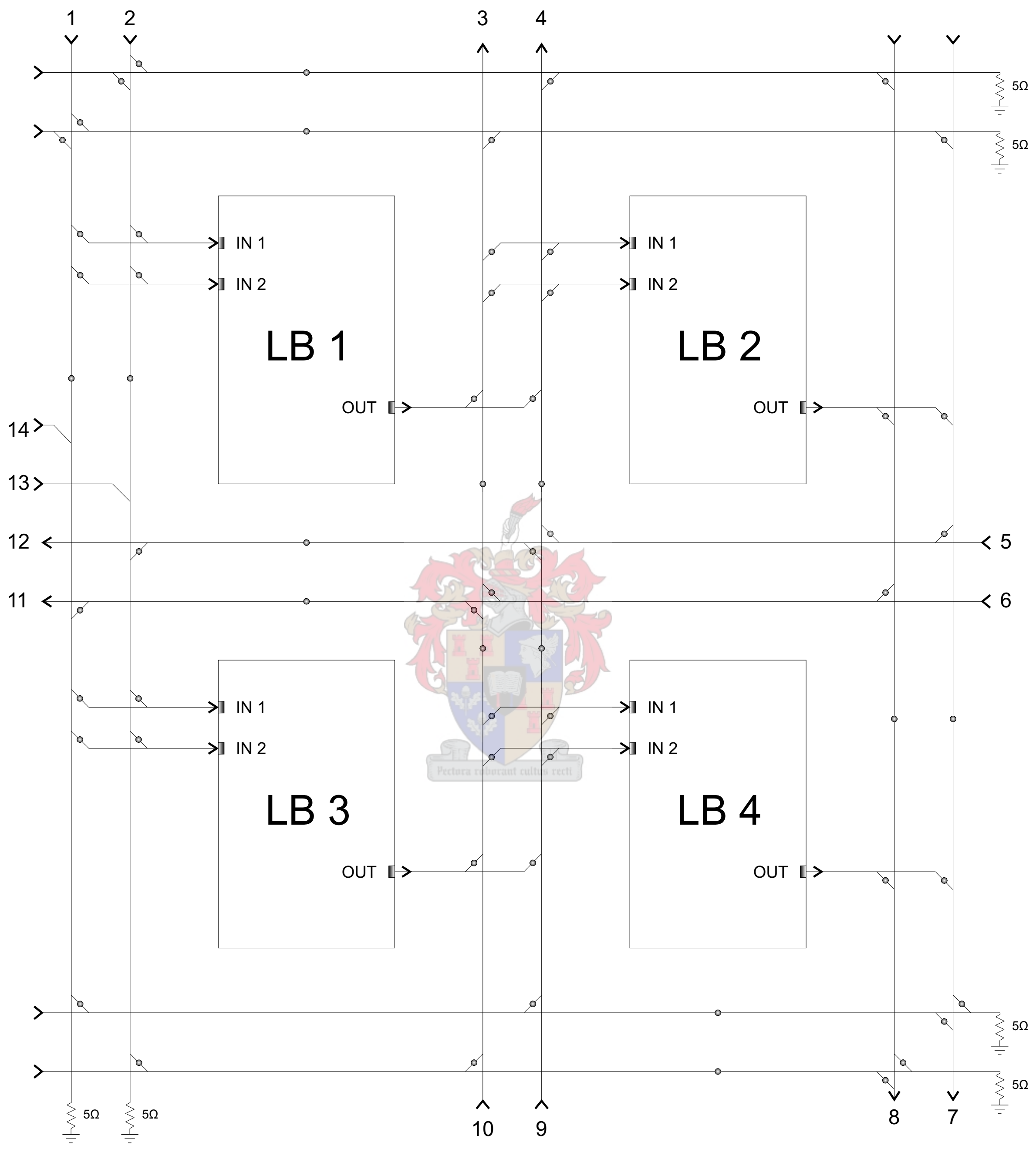

Figure 5.14: Diagram for data route checking

numbers at the edge of the diagram indicate the active input and output (IO) data ports of the SPGA. Some ports are inactive because of limited pads on the chip. Input ports 1 and 2 can provide input for LB1. Ports 3 and 4 output data from LB1 and/or LB3. Input ports 5 and 6 will most likely provide data for LB2 while ports 7 and 8 may output data from LB2 and LB4. Ports 9 and 10 can provide input data for LB4. Data from LB3 can be output either to ports 3, 4, 11 or 12. Finally ports 13 and 14 was inserted as an afterthought to provide easy access to the input of LB3.

Programming routes contain the signals that program the switches. Figure 5.15 shows the diagram used to check the routes to all the switches. It is more or less the same as the data route diagram in Figure 5.14 but with a few additions. The switches are divided into what may be called a matrix to gain access to all the switches individually. The columns are labeled using letters of the alphabet and the rows using numerals. In combination the crossing of a column and a row provides individual access to a determined switch.

Columns A through J in combination with rows 1 through 6 provide access to all the data route switches. Columns K and L in combination with rows 7 and 8 provide access to the logic blocks.

The lines that originate from column programming blocks are not RSFQ lines but are DC lines (HUFFLE outputs). These form current loops and must be traced back to the originating HUFFLE.

The lines that originate from row programming blocks are RSFQ lines and terminate easily into a 5 ohm load and are not required to return to the programming block. Microstrip transmission lines are used to connect switches that are far apart.

The diagram in Figure 5.15 was used to ensure that all the programming lines (RSFQ and DC) go where they are supposed to, and terminate at the correct location.

DC reset route carries the DC current signal that resets the DCRLs (DC resetable latches). This line is driven by an off-chip source to produce a predetermined current which resets the DCRLs. Since it is a DC line, the line may be as long as needed to connect all the latches. There are 78 latches and all of them need to be connected to the same line. The line may not split, divert or break at any point along the route. Figure 5.16 indicate the connection sequence to the latches.

HUFFLE prebias routes contain the signals that initialize the HUFFLE with a predetermined state. This line effectively carries a DC current and is driven by an external source. It was decided to split the signal into three parallel routes to accomplish the routing. The three lines terminate to ground through three 3Ω resistors. The three resistors in parallel give 1Ω and should be taken into account when calculating the external driving signal. The external signal that is now required to bias the HUFFLES will have to be three times as large as it would be for a single serial HUFFLE.

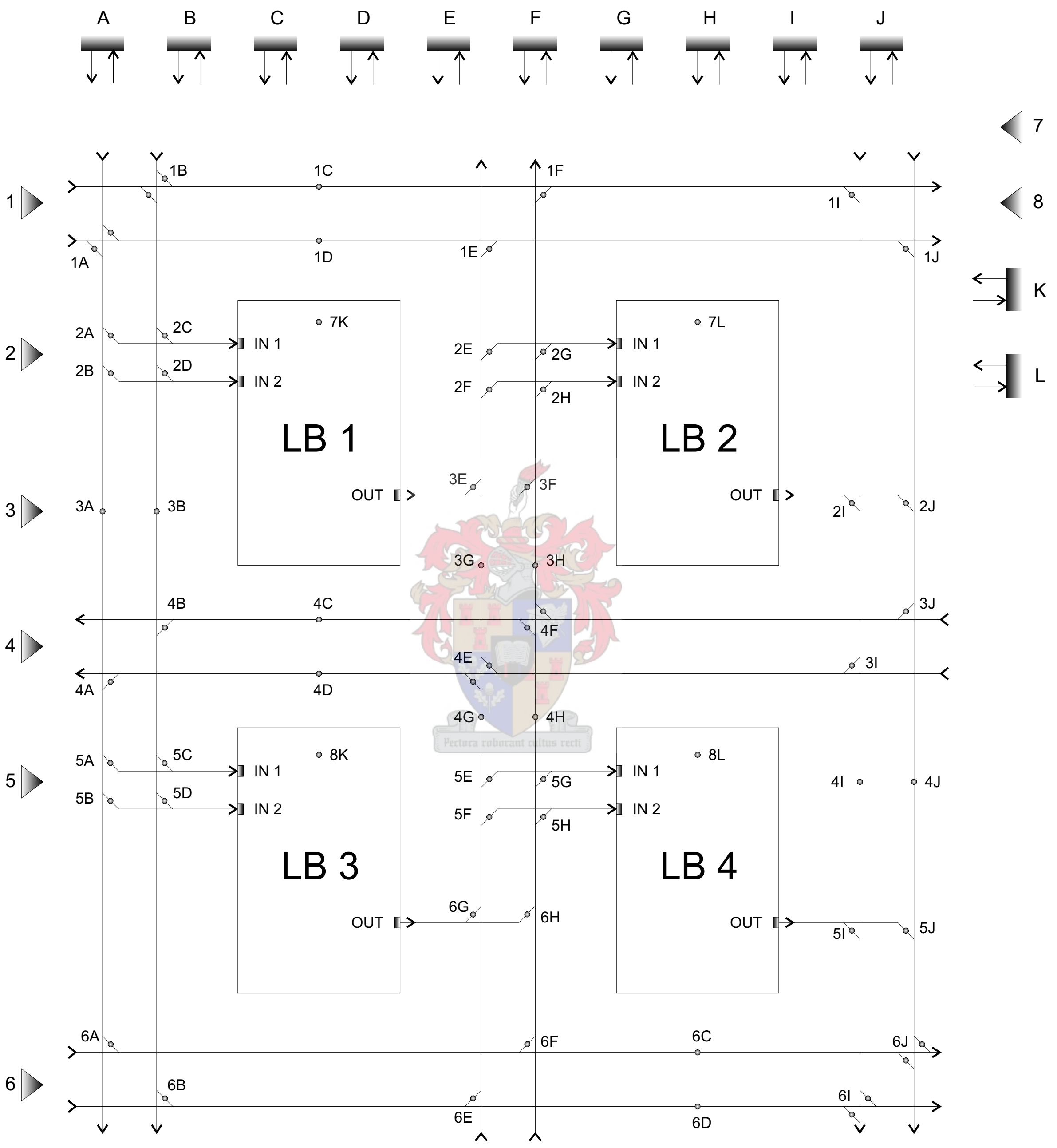
A
B
C
D
E
F
G
H
I
J
1
2
3
4
5
6
7
8
K
L
LB 1
LB 2
LB 3
LB 4
IN 1
IN 2
OUT
1A 1B 1C 1D 1E 1F 1I 1J
2A 2B 2C 2D 2E 2F 2G 2H 2I 2J
3A 3B 3E 3F 3G 3H 3I 3J
4A 4B 4C 4D 4E 4F 4G 4H 4I 4J
5A 5B 5C 5D 5E 5F 5G 5H 5I 5J
6A 6B 6C 6D 6E 6F 6G 6H 6I 6J
7K 7L 8K 8L

Figure 5.15: Diagram for checking programming routes to switches

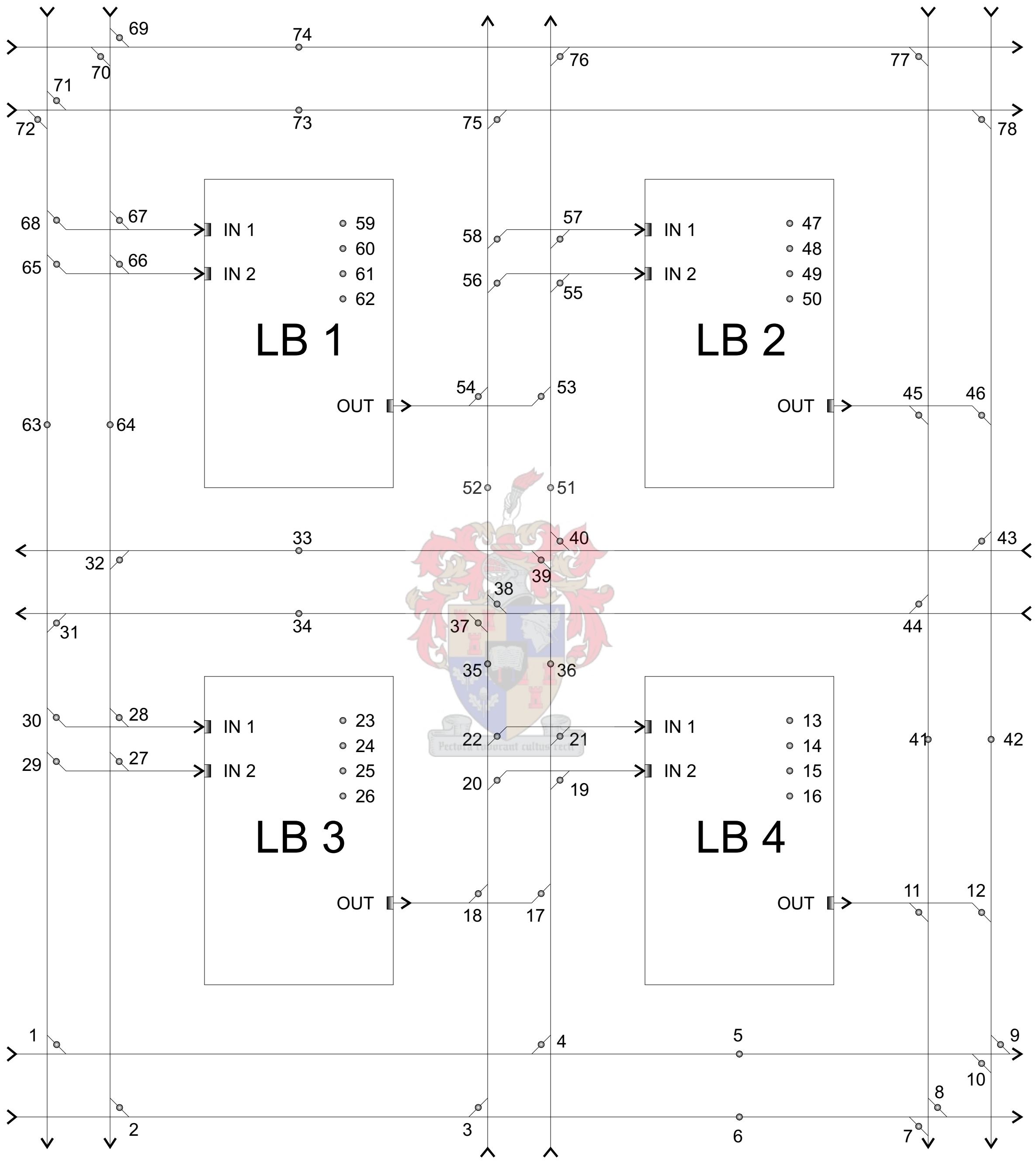

Figure 5.16: Diagram for checking DC reset routes

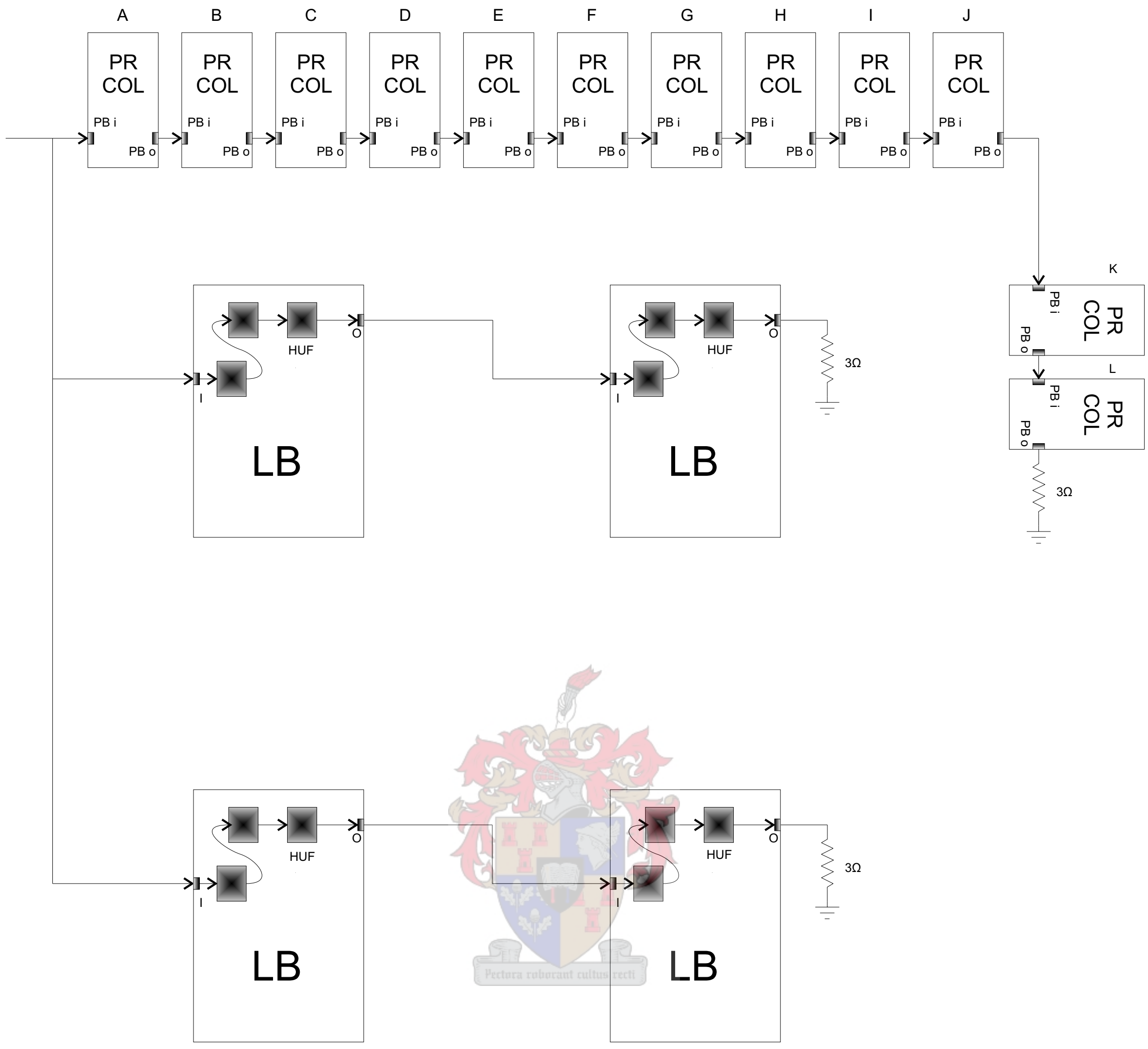

Figure 5.17: Diagram for checking HUFFLE prebias routes

DC bias routes are too many and complex to sketch in a diagram. It was decided not to use a diagram but to check them on-screen on a per gate basis instead. Each gate was checked several times to make sure that it has a DC bias connection.

Output clock routes are few and simple and it was decided to them check directly in the layout. The output clock routes are on the outskirts of the chip and they do not entangle in many other lines making it very easy to check in the layout.

5.5.3 Connection checking

Signal route checking ensured that the signals are sent where they are supposed to, while connection checking makes sure that the many connections in a route are solid and fast.

Connection checking makes sure that an input signal reaches its intented destination without broken links or serious disruption. Thus, during connection checking, all the routes are followed with the focus on connections between blocks and interconnects.

5.5.4 Design rule checking

'That's not a regular rule: you invented it just now.'

[Lewis Carrol - Alice's adventures in Wonderland (1865)]

In essence, design rules represent the physical limits of the manufacturing process. Overall, design rules are there to help layout designers understand and account for the physical three-dimensional limitations and manufacturing tolerances within the CAD environment.

After initially checking for design rule violations by hand and finding that too many errors could be missed in that way it was decided to use a CAD-based design rule checker (DRC). Up until this point, an automated DRC was not used because the design rules are broken by grounded Josephson junctions.

It was possible to implement Hypres' design rules [1] bar one in LASI's DRC. The result is that a much more effective method to find design rule violations can be used. The complete implementation of Hypres' design rules for LASI is listed in appendix E.

It may be worth noting a few oddities: With regards to spacing from one layer to another, for instance, rule 1.4 - M0 spacing to M1 >= $1\mu m$. There are two cases allowed by the rule:

1. M0 block and M1 block separate from each other not touching or overlapping in any way.
2. M0 block totally overlapping M1 block

The first case is the obvious case that the rule allows. But the second is also allowed because the *edge* of M0 may not cross or overlap the *edge* of M1. This second allowable case poses a slight problem for the implementation of this rule in the DRC. The DRC can check either for spacing or overlapping but not both at the same time. Thus this specific rule (and a few others like it) had to be divided into two DRC implementations.

Another oddity worth noting is the issue of the grounded Josephson Junction (JJ) layout, where I1B overlaps the R2 layer. One could not completely leave out rule 5.3 because it would then miss other possible critical violations. It was decided to allow I1B to overlap R2 but only over M1. In this way, the grounded JJ would be exempt from that rule but, for instance, a via between R2 and M2 would be checked.

Rule 5.4 was not implemented in the DRC because almost all the resistors connected to M2 break this rule in some way.

5.5.5 Full-chip scan

During a full-chip scan the whole layout is looked at gate by gate. Each gate is checked to see if all the connections are made and also checked for any gross errors that may have been missed during any of the previous checks. The full-chip scan was the last check in an effort to eliminate all possible errors and was done three times.

5.6 Parameter extraction

Physical layout of RSFQ ICs is a difficult task for a number of reasons, one being that circuit parameters such as inductance are complicated to estimate and/or extract from the layout. During the layout phase, a layout engineer has to be certain that what is being done is in accordance with the circuit requirements. To do so, the engineer has to be able to estimate or extract values for inductance, resistance and junction sizes, taking into account the manufacturing process variations and tolerances.

With that in mind, a few tools are available to help the engineer accomplish this task.

5.6.1 SLine

SLine [45] uses Chang's 2D analytical equation to estimate inductance values [48]. It uses process parameters such as dielectric thickness, line thickness and penetration depth (to name a few) to produce an analytical estimate of a superconducting microstrip line's inductance. It is not able to analyze complex structures, such as lines crossing over each other or mutual inductance or even corners, but it quickly produces results for simple straight structures.

5.6.2 FastHenry and InductEx

FastHenry [49] with superconductor support [50] is a numerical 3D program that uses a magnetoquasistatic formulation of Maxwell's equations, from which a mesh analysis is created and solved with a multipole-accelerated algorithm. A drawback of numerical 3D analysis is that time- and memory resource requirements are substantial. Nevertheless, it is very powerful and versatile in handling complex structures.

InductEx [51] [52] is a front-end for FastHenry or an intermediary step between CAD layout tools and FastHenry. It uses segmentation routines developed by Fourie [4] to discretize complex 3D layout structures and pass the results to FastHenry. FastHenry is then used to analyze the structures and return results which are interpreted by InductEx to produce values for inductance and mutual coupling. InductEx has been invaluable during this project, especially to analyze the highly complex structures in the multi-line current-select switch.

5.6.3 Example: I2-Switch

As an example to showcase the power of the tools, the Two-line Current-Set Switch will be analyzed to extract values for the coupling factors between the inductors in the select lines, bias control line and SQUID.

Figure 5.18 shows part of the layout, highlighting the various select- and control lines. The labels *JJ1* and *JJ2* represent where the two junctions of the SQUID should be[6]. The lines *a* and *b* are the *select* lines and are defined in M1, while the bias control line is defined in M3. The SQUID line is defined in M2, thus sandwiching it between the select and control lines.

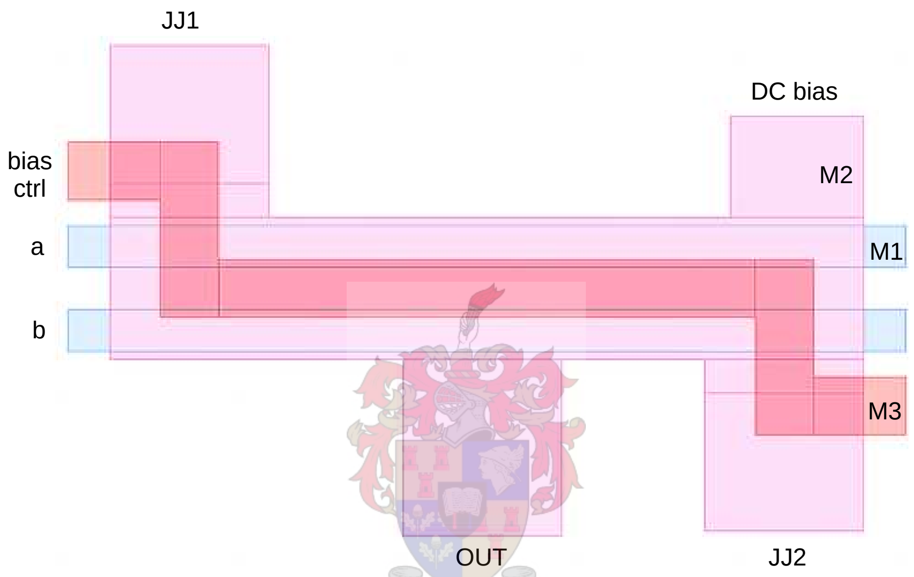

Figure 5.18: Layout of the 2-Line Current-Set Switch: bare essential layout showing control lines

This layout is passed to InductEx[7] to extract the inductor and coupling values. An extra feature of InductEx incorporates random variations by which the program can vary all geometries according to predefined distributions. These random variations can be repeated automatically to obtain inductance and coupling spreads. Table 5.2 shows these results, which can be incorporated in the Monte Carlo model for this gate.

The bias control line is used to bias the SQUID loop with a certain amount of current and in effect deform the SQUID switching curve. The reason of this is that the SQUID switching curve demands that the total current must go negative (or close to zero) for the SQUID to reset [4]. Without the bias current in the I2-switch circuit, the SQUID does not reset. Two factors determine the behaviour of the SQUID current: the dc-bias current (determined by the gate's bias resistor), and the (newly introduced) bias control current. These two factors can be varied to find an optimal yield solution. A manual process was used where both the

[6]The junctions have been removed for clarity
[7]InductEx port definitions are not shown

Table 5.2: Gaussian distribution parameters for inductances and coupling factors of the I2-switch

Inductance	**Mean [pH]**	**Std. dev. (σ) [pH]**
SQUID	3.4146	0.0433
DC bias ctrl.	12.4339	0.2433
Line A	6.1730	0.2118
Line B	6.1675	0.2329
Coupling factor	[dimensionless]	
S - dc	0.4184	0.0047
S - A	0.2556	0.0050
S - B	0.2575	0.0060
A - B	0.0991	0.0037
dc - A	0.1244	0.0027
dc - B	0.1254	0.0032

dc-bias current and bias control current were varied in both positive and negative directions until unsatisfactory results were encountered. The end result can be viewed (in Figure 5.19) as a 2-dimensional graph (dc-bias resistor vs. bias control current) with the shaded area defining the region of acceptable values. The optimal is the mid-point of the area, which was found to be a dc-bias resistor value of 9Ω and a bias control current of 85μA.

The SPICE listing used to finally determine the yield of the I2-switch is given in Appendix B.

5.7 Input and output impedance matching

Although the emphasis during design was on functionality, performance was not totally ignored. The simulations show that the chip could be tested to frequencies in the GHz range. Therefore, those input and output ports on the chip that may contain high frequency signals must be impedance matched. It is assumed that the external connection line to the chip has a 50Ω characteristic impedance and that the impedance of the port (pad) is negligible. On-chip, however, it is not possible to facilitate such a high impedance microstrip transmission line and thus a resistive matching network was used to match the external line to the on-chip circuit impedance.

5.8 Signal-to-Pad assignments

Figure 5.20 in combination with Table 5.3 provide information about pad assignments of the various IO and control signals.

Operational data ports provide computational data (after the whole chip has been programmed). The programming clock is shared between the two programming frames and all the logic blocks. The DC bias should be 2.6mV, but for reasons discussed in Chapter

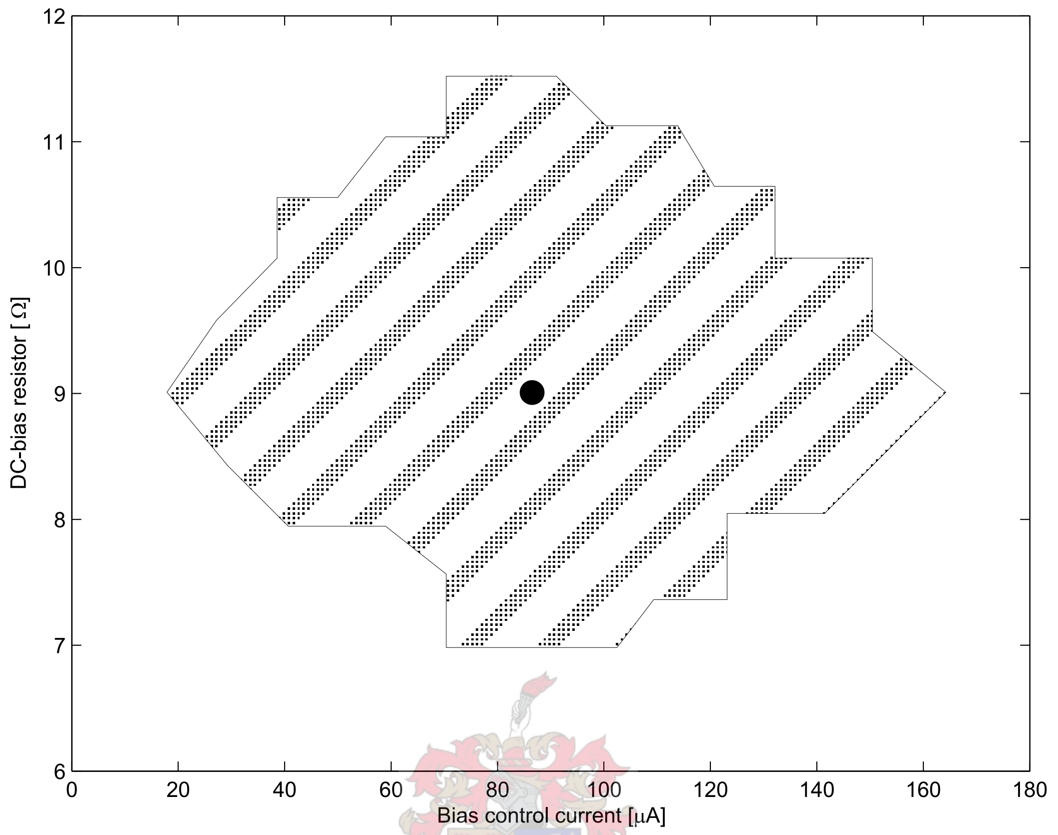

Figure 5.19: Area of nominal working values for DC-bias resistor vs. Bias control current, also showing the optimal value indicated by the black dot

6.1.2.1, the total bias current (560.48 mA) should be used as reference instead. The gray box in the upper left-hand corner has been placed there as a visual guide on the physical chip to indicate where the first pad is located.

The DC bias current creates other problems relating to layout. According to Terai *et al.* [53] and Kadin *et al.* [54], to avoid disturbances caused by currents flowing in the ground plane, the ground bonding (or ground pad) should be close to the point of current injection (i.e. DC-bias pad). Bias lines should not be close to critical structures, such as junctions. Ground currents return immediately under the bias line, which is one of the good qualities of microstrip lines. Thus to avoid diffusion of bias current in the ground plane, one should have one ground pad close to each current injection point (preferably only one injection point).

One should avoid running bias lines parallel with any inductance and preferably use a 'sky plane'[8] [55]. However, this is not practical in the 3 metal layer process available from Hypres.

[8]A sky plane is an extra metal layer, which in effect is another ground plane

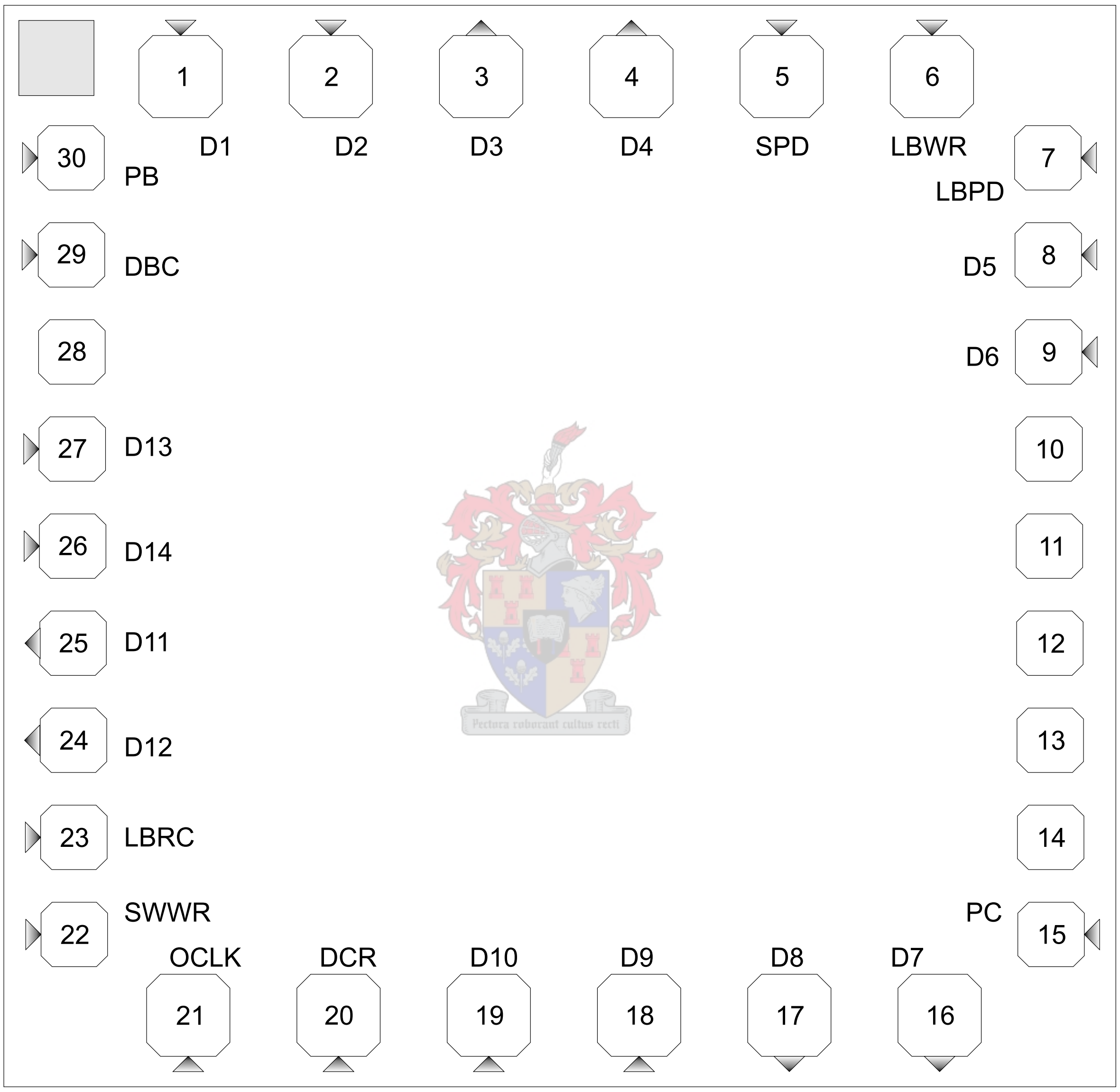

Figure 5.20: Diagram: Signal-to-Pad assignments

Table 5.3: Table: Signal-to-pad assignments

Pad #	Name	Comment
1	D1	Operational data port 1 [in]
2	D2	Operational data port 2 [in]
3	D3	Operational data port 3 [out]
4	D4	Operational data port 4 [out]
5	SPD	Switch programming data [in]
6	LBWR	Logic block write (WR) [in]
7	LBPD	Logic block programming data [in]
8	D5	Operational data port 5 [in]
9	D6	Operational data port 6 [in]
10	-	-
11	-	-
12	-	-
13	-	-
14	-	-
15	PC	Programming clock [in]
16	D7	Operational data port 7 [out]
17	D8	Operational data port 8 [out]
18	D9	Operational data port 9 [in]
19	D10	Operational data port 10 [in]
20	DCR	DC-reset [in]
21	OCLK	Output clock (for RSFQ-to-DC conv.) [in]
22	SWWR	Switch write (WR) signal [in]
23	LBRC	Logic block read clock [in]
24	D12	Operational data port 12 [out]
25	D11	Operational data port 11 [out]
26	D14	Operational data port 14 [in]
27	D13	Operational data port 13 [in]
28	-	-
29	DCB	DC bias (power supply) [in]
30	PB	HUFFLE prebias [in]

5.9 Moats

A moat is a narrow rectangular cavity in the ground plane which provides a low energy path for magnetic flux [56]. The effect is to divert flux through the moats instead of nearby sensitive elements. By creating moats near critical structures such as Josephson junctions, it protects the devices from performance degradation caused by flux trapping during the cooling process. It was prefered to use short narrow rectangular shaped moats around Josephson junctions in contrast to long moats surrounding an entire circuit block. Figure 5.21 shows typical use of moats.

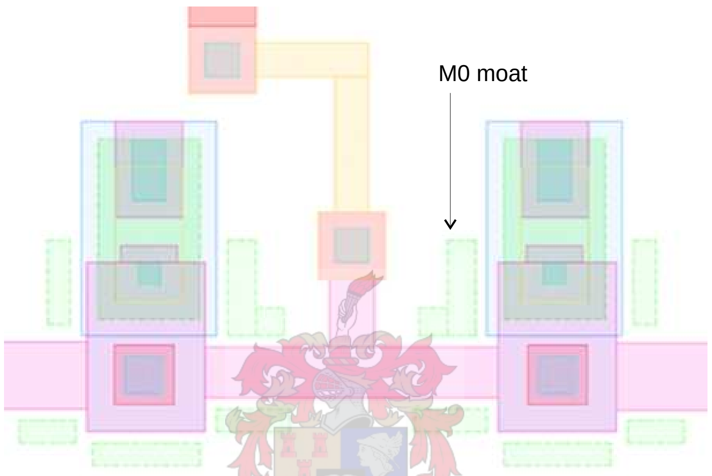

Figure 5.21: Typical use of moats in a layout

5.10 Chapter summary and conclusions

In summary this chapter considered the layout process and most related aspects. Various layouts of gates and blocks were discussed and it was shown how they fit into the bigger picture of the SPGA framework. Problems relating to layout such as error checking and verification were minimized or solved where possible. Finally, the input- and output signals were mapped to chip pads.

The major accomplishment in this chapter was the completion of the full chip layout considering the size of the project and great manual effort required to render a layout design without automated CAD tools. Various verification techniques were developed to minimize human error, including the design rule checker.

Limitations in fabrication technology restrict the scale of the design, which in turn constrain the overall functionality of the SPGA. The eventual layout design include four 2-bit logic blocks and a 2-channel routing architecture that feed the logic blocks.

The subject of layout automation demands attention, especially if larger projects are to be undertaken. Automation from circuit schematic to layout and feedback from layout (which include parameter extraction) will profoundly ease and speed up development of projects as well as minimize human error during layout. It may be worth investigating the use of standardized layout block shapes and sizes to ease the placement of gates during the layout phase.

As fabrication technology improves, future SPGA development can extend the functionality and broaden the capabilities of this concept.

'Information necessitating a change of design will be conveyed to the designer after and only after the design is complete.'

[Anonymous - 'Now they tell us' law]

Chapter 6

Testing

'There are very few problems that cannot be solved through a suitable application of high explosives.'

[Scott Adams - The Dilbert principle]

TESTING any electrical or electronic equipment is critical and requires proper planning and preparation. This chapter gives a description of the test equipment and configuration that will most likely be used to test the SPGA.

Some problems relating to the test setup are discussed and possible solutions are presented.

Further, typical examples of test functions are given and the expected results are discussed.

6.1 Testbed

The testbed consists of three main parts, namely the excitation stage, the Device Under Test (DUT) and the measurement stage. The excitation stage encompasses the electronics that generate the input signals for the DUT. In other words, the signals that drive the DUT. The DUT, in our case, is the SPGA. The measurement stage comprise of acquisition and measurement equipment.

For superconducting electronics to function correctly, they need to be cooled to cryogenic temperatures (<60K). For low-temperature electronics (used in Hypres' 3μm Nb process) to function correctly, the circuit needs to be cooled to below 4.2K. Two known methods are in use today to accomplish such low temperatures, namely the use of liquid Helium (with a boiling point of 4.2K) or with the aid of a cryogenic cooler (or cryocooler). We have had little success with liquid Helium testing because of the high boil-off rate in an open cryostat. Other difficulties related to liquid Helium are its cost, availability and transportation problems. For these reasons and have recently acquired a mechanical cryocooler. The SPGA chip must be inserted into this complicated cooling system. Aspects of the cooler may aid us in accomplishing our measurement task and thus a cryocooler's operation will be discussed shortly in section 6.1.1.

The room temperature electronics of the excitation stage will be discussed in section 6.1.2.

The measurement equipment will include a high-quality oscilloscope. Although, the output signals are small, they are visible (or can be amplified if needed).

Lastly, a number of functions to be used to test the operation of the SPGA are discussed.

6.1.1 Cryocooler

Small mechanical cryocooler development has been strongly stimulated over the past years by the emergence of specific applications requiring low-temperature operation with relatively low cooling power.

Today the main applications of cryocoolers are [57]:

- Cryopumping for high and clean vacuum (semiconductor industry, space simulation chambers, particle accelerators)
- Cooling of detectors (for example, infrared detectors for Earth observation, night vision, and missile guidance as well as gamma ray detectors and bolometers for astrophysics).
- Cooling of electronic components (cold amplifiers) or of devices including superconducting materials (SQUIDs, Josephson junctions, high-field magnets).
- Cooling of samples for physics.
- Cooling of radiation thermal shields and recondensation of boil-off in cryogenic liquid storage tanks or large superconducting magnet cryostats for magnetic resonance imaging.

The development of small mechanical cryocoolers has been based mainly on the technology of regenerative heat exchangers. These regenerators are generally constituted by a porous matrix (metal wire mesh or spheres) that acts like a thermal sponge by alternately storing or rejecting heat.

A number of different cryocooler designs exist, of which Grifford-McMahon, Stirling and Joule-Thomson are the most popular. For the purposes of this project, only the Grifford-McMahon type will be discussed because the SPGA will be tested in a cryocooler based on this design.

Figure 6.1 shows a elementary diagram of a Grifford-McMahon cooler and the various phases are presented in Figure 6.2.

The process can be described as follows:

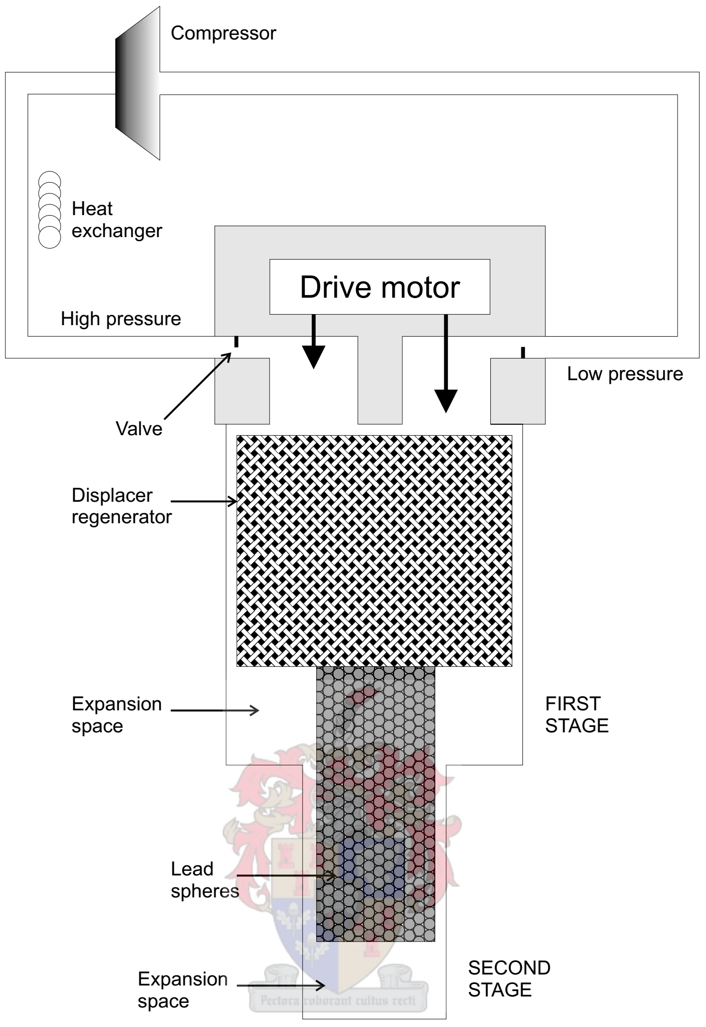

Figure 6.1: Elementary diagram of a Grifford-McMahon cryocooler

- Phase 1: The displacer is at its lowest position, the outlet valve is closed, and the inlet valve is opened. The high-pressure gas fills the regenerator and the space above the displacer is at room temperature.
- Phase 2: The inlet valve is still open, and the displacer moves to its upper position. The high-pressure gas passes through the regenerator, is cooled down isobarically[1] by the matrix, and fills the space below the displacer at low temperature.
- Phase 3: The displacer is at its upper position, the inlet valve is closed, and the outlet valve is opened. The gas in the regenerator and in the cold space undergoes expansion; the cooling effect achieved can be used for refrigeration.
- Phase 4: The outlet valve is still open, and the displacer moves to its lowest position.

[1]Isobar: at constant pressure

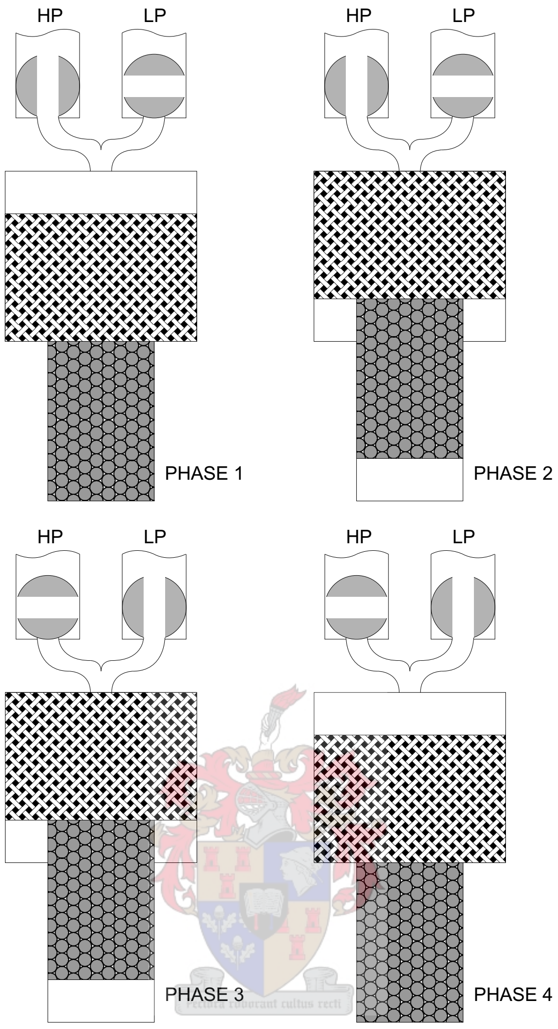

Figure 6.2: Phases of Grifford-McMahon cryocooler

The low-pressure gas passes through the regenerator, is warmed up isobarically by the matrix, and fills the space above the displacer at room temperature.

A heat exchanger at the exhaust of the compressor is used to reject heat at the ambient temperature and theoretically to achieve an isothermal compression[2].

We have recently acquired a ST405 cryocooler from Cryomech. Specifications state that the second stage can cool down to 2.8K (which was verified through testing) with zero watt cooling power and 4.5K with half-watt cooling power. The first stage has a typical temperature of 60K. The cooler takes about one hour to cool from room temperature to our operating temperature. The vacuum is of the order 10^{-5} atm.

[2]Isothermal compression: compression at constant temperature

6.1.2 Room temperature electronics

Testing RSFQ electronics is complex. Problems such as noise is a major concern to any test engineer because RSFQ signals are so small (on the order of only a few millivolts). Without sufficiently suppressing noise, input pins can be falsely triggered and output signals could get lost in a sea of noise.

6.1.2.1 Power supply

Another problem is that of a clean and constant power supply to the chip. Firstly, the supply needs to be able to provide about 0.6A of clean constant current without any noise or disturbances. After initial trials, it became clear that ordinary electronics-based supplies (such as AC-DC or DC-DC converters) produce unacceptable levels of noise. A possible solution is to use a battery supply.

Secondly, complications arise when large currents are transported in copper wires from an external supply to the chip on the inside of the cryocooler. A copper wire has a small resistivity per length associated with it. Thus, when large currents flow in the wire, a potential difference develops across the wire, which means that the input voltage is not the same as the output voltage. The same scenario is evident for the grounding wire which may cause a discrepancy between the on-chip ground and the outside reference ground. A solution is to minimize the length of low resistance wire from supply to chip. Also, measure current instead of voltage, making sure the right amount of bias current is flowing in the supply line irrespective of input voltage.

6.1.2.2 Input- and output signals

It is standard practice in RSFQ testing laboratories to start with low-frequency tests. This significantly lowers the requirements on fast driving- and measurement electronics.

Compounding the issue of noise, is the fact that the SPGA has numerous IO ports which have to be excited and measured. Generating 17 synchronized input signals posed quite a challenge. We are in the process of experimenting with various configurations for testing. Two options seem feasible:

1. Signal generation with the aid of a computer

2. Signal generation using microcontrollers

The first option would, for instance, generate signals using numerous digital-to-analogue converters (DACs) and measure signals with analogue-to-digital converters (ADCs) all with the help of tailored software that can easily change signals in real time. Unfortunately, the first prototype yielded poor results [58], the reason being that to address so many DACs took too long; and the computer generated noisy signals.

The second option would enlist the help of dedicated microcontrollers to control the DACs and generate the needed signals. The advantage is faster response but the downside is that input signals cannot be changed in real time. Once again, the first prototype yielded unsatisfactory results [58], but this option showed the most potential.

6.1.2.3 Noise

Various noise sources have been determined, some of which could be easily minimized whilst others pose significant problems.

As mentioned earlier, the *power supply* is a noise source but a battery solution seems to have worked.

The *vacuum- and compression pumps* of the cryocooler were a major source of noise. The best way to eliminate this was to run the cooler to the required temperature (around 4K) and momentarily switch off the vacuum pump. The compression pump must remain running for the cryocooler to function correctly. The cooler kept its temperature for a short time (at least a few seconds) before it increased above the critical temperature of the superconductor. This provides us with a short noiseless (from the vacuum pump at least) time-window in which to test, after which the pump could be switched on again.

One suggestion has been made to move some of the electronics to the first stage of the cryocooler. The first stage has a temperature of about 60-70 Kelvin and provides a substantially less noisy environment. CMOS chips have been demonstrated to operate at 80K and we are in the process of testing at lower temperatures [59].

Various options are available[3]:

1. The most elaborate setup would have all the DACs and ADCs including a dedicated microcontroller at the first stage. The electronics would then be controlled by a RS232 or USB data stream from outside the cooler.

2. Another option would be to generate large (5V) input signals outside the cooler and filter the noise immediately before it goes into the cooler. At the first stage these signals would then be attenuated to the correct amplitude with low-noise resistive divider circuits. The output signals also get amplified at the first stage with High-Electron-Mobility Transistor(HEMT) [60] amplifier circuits which have been shown to operate at such low temperature.

6.2 Test cases

As is the case when testing all types of logic, one has to verify that all the possible input variations produce the expected output. Even more so with RSFQ logic because of its pulse-

[3]These options have not been tested at time of publication

based nature. A missing pulse can represent a logical '0' or a malfunction.

For reference Figure 6.3 is presented and indicates all the inputs, outputs and switch assignments. Input and output ports are indicated by numbers in italics.

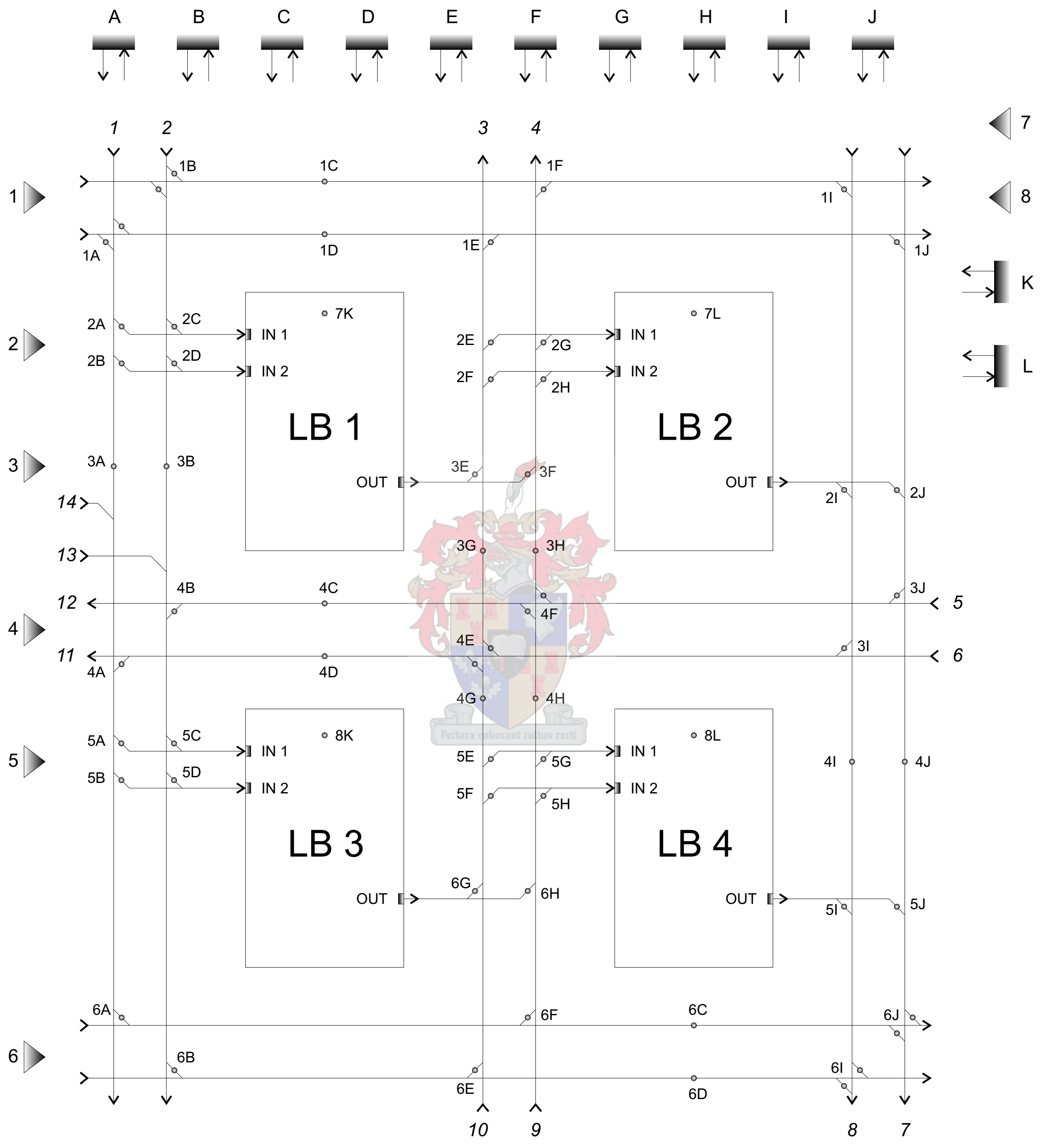

Figure 6.3: Diagram of SPGA architecture indicating inputs, outputs and switch assignments

As a starting point, the most basic operation will be tested first, then progressing to more comprehensive circuits with the aid of the following three examples:

1. Direct translation of input to output: allocate the two (input) ports 5 and 6, enable the switches 4C and 4D and output should be presented at ports 11 and 12.

2. One logic block: allocate (input) ports 1 and 2, use logic block LB1 and assign output to port 3. Thus, switches 2A, 2D and 3E should be enabled.

3. Implement a more comprehensive logic function to use all four logic blocks and in the process, route logic block outputs to other logic block inputs and chip output ports.

The examples given in this section serve as guidelines for a simple testing session. A more thorough session extends these same ideas to individual tests of all the logic blocks and switches systematically.

6.2.1 Example 1: Direct input to output

This is a typical example to test if the routing architecture is working correctly. The goal is to see if the inputs at ports 5 and 6 can translate (propagate) to ports 12 and 11 respectively.

For this example only two switches need to be programmed, namely 4C and 4D. The switch programming data words entered at port SWPD should be as given in Table 6.1.

Table 6.1: Switch programming data words for Example 1

	6	5	4	3	2	1	J	I	H	G	F	E	D	C	B	A
i	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
ii	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0

The expected results for all the possible input variations are presented in Table 6.2.

Table 6.2: Expected results for Example 1

Input		Output	
5	6	12	11
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

6.2.2 Example 2: One logic block

By testing a logic block in the way prescribed in this example, one is able to verify if the logic block itself is working (i.e. decoder, memory cells, etc). The routing architecture should already have been tested with extensions of the concept in the previous example.

This example programs logic block LB1 while input is provided from ports 1 and 2 and the output should present itself at port 3. The programming has two stages, one for switches and another for the logic block. The switches that are to be activated are 2A, 2D (for input) and 3E (for output). The switch programming data words are presented in Table 6.3:

Table 6.3: Switch programming data words for Example 2

	6	5	4	3	2	1	J	I	H	G	F	E	D	C	B	A
i	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
ii	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
iii	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0

The function that is to be implemented in the logic block for this example is an OR operation[4]. The programming data word for an OR operation is entered in the sequence: 1011.

The correct logic block still needs to selected (in this case, LB1 assigned to 7K). This is done with the aid of the logic block programming frame but the setup is a bit complicated. The logic blocks and the logic block programming frame share the same data line as well as the same programming clock. The configuration is more clearly illustrated in Figure 6.4.

In Figure 6.4 two lines are visible (LBPD - Logic Block Programming Data; and PR_CLK - Programming Clock). The data line first enters the programming frame where it cascades through five DROs and one additional DRO which serves to correctly synchronize the data clocking[5]. After exiting the programming frame, it splits into two lines which flow through LB1 and LB2, and continue on to LB3 and LB4 as well. The clock line splits in various directions to serve all the DRO's of the logic block and those of the programming frame.

For instance, for data to arrive at LB3, it must first pass through the programming frame as well LB1 before it can be processed by LB3. One has to keep in mind that data can only exit a DRO when it gets clocked. Thus, it would take 6+4=10 clock pulses for the first data bit to arrive at LB3.

At the same time, one has to keep in mind that the last 4 data bits in the LBPD data stream signifies the 'row' and 'column' data bits of the programming frame.

With regards to this section's example, logic block LB1 has to be programmed with the data bits (1011) and getting the programming frame to process the correct logic block would require the data sequence in Table 6.4. The sequence is entered at port LBPD as given in the table from left to right.

Simulation results are presented in Figure 6.5 to more clearly illustrate the operation and timing. The inputs *PR_CLK*, *LBPD* and *WR* are the clock, logic block programming data and

[4]The function could easily be anything else, but the OR operation tends to produce more logical '1's and thus is more easily verified. Just to be 100% certain, another function such as the AND operation should also be tested

[5]Note: the extra DRO is not present in the switch programming frame

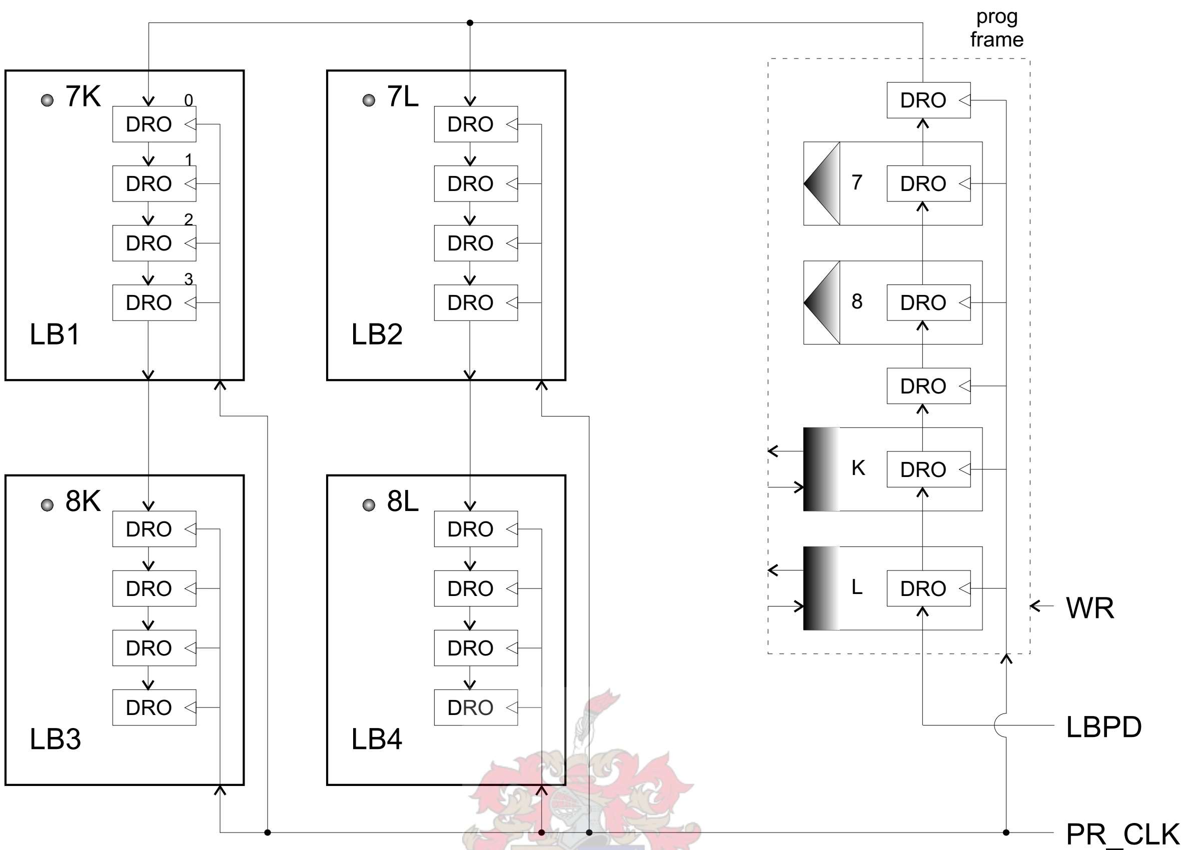

Figure 6.4: Wiring of data- and clock lines for the logic block programming frame

Table 6.4: Logic block programming data word for Example 2

LB1(3)	LB1(2)	LB1(1)	LB1(0)	7	8	L	K
1	0	1	1	1	0	0	1

the programming write signal respectively[6]. The simulation results also show the signals in the relevant selection lines (*HUF K* and *ROW 7*). Inside the logic block, the programming HUFFLE's current is also presented, and finally the data that gets transfered to the DCRLs are also shown in the *LB mem* dotted frame[7].

6.2.3 Example 3: Comprehensive logic function

An attempt to test the SPGA in its totality would ideally test all four logic blocks and at least a number of switches scattered throughout the chip. A combination of two logic functions were been selected to fulfill these criteria, and are listed below[8]. The single-bit inputs are A, B, C, D and E while the outputs are O_1 and O_2.

[6]Refer to Chapter 4.4.5 for more detail on the programming frame
[7]Refer to Chapters 4.4.4 and 4.4.4.2 for more detail on the logic block and the programming thereof
[8]These functions are the same as in Chapter 4.6.2 used during the Verilog simulations

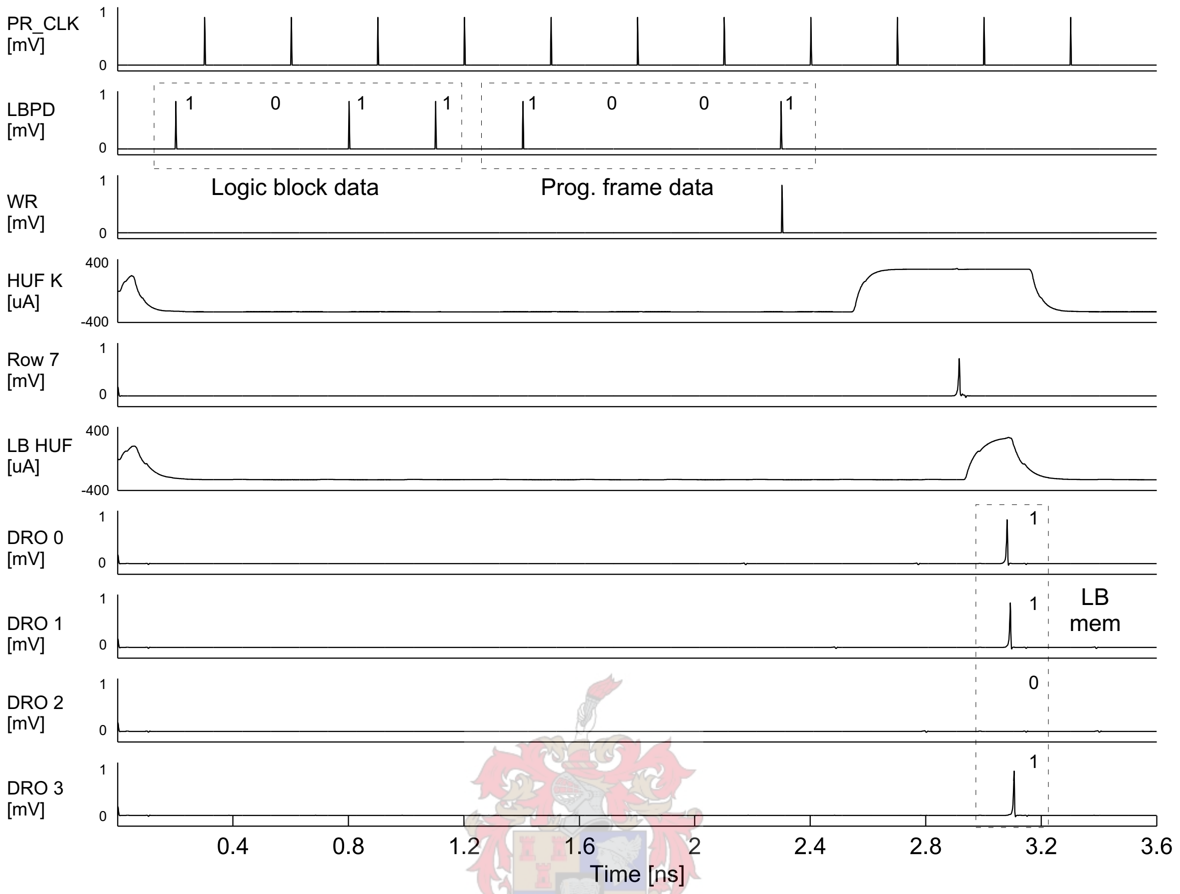

Figure 6.5: Simulation showing programming of logic block

$$O_1 = (A \text{ XOR } B) \text{ OR } (C \text{ AND } D) \quad (6.2.1)$$

$$O_2 = (C \text{ AND } D) \text{ NAND } E \quad (6.2.2)$$

A simplistic mapping of the logic functions into logic blocks was done and the results are given below, where LB# refers to the output produced by the corresponding logic block:

$$\begin{aligned} LB1 &= A \text{ XOR } B \\ LB2 &= LB1 \text{ OR } LB3 \\ LB3 &= C \text{ AND } D \\ LB4 &= LB3 \text{ NAND } E \end{aligned}$$

For the above logic functions to find its way into the SPGA logic blocks requires the aid of the logic block programming frame. As discussed in the previous example, the data for the logic blocks must first flow through the programming frame followed by the programming data for the frame itself. The last four bits of this programming data sequence represent logic block selection bits used by the programming frame. As illustrated by Figure 6.4 for programming data to arrive at LB3 and LB4, it must first pass through LB1 or LB2 respectively. For this reason extra 'spacer' bits are inserted, but have no other functional significance. The resulting data words are presented in Table 6.5 where '-' represent the *spacer* bits and LB(#) denote the respective memory bits (DCRLs) inside the logic block. The data words are processed from left to right, thus the logic block bits would be clocked in first, followed by the programming frame bits.

Table 6.5: Logic block programming data words for Example 3

-	-	-	-	LB1(3)	LB1(2)	LB1(1)	LB1(0)	7	8	L	K
0	0	0	0	1	0	1	0	1	0	0	1

-	-	-	-	LB2(3)	LB2(2)	LB2(1)	LB2(0)	7	8	L	K
0	0	0	0	1	0	1	1	1	0	1	0

LB3(3)	LB3(2)	LB3(1)	LB3(0)	-	-	-	-	7	8	L	K
0	0	0	1	0	0	0	0	0	1	0	1

LB4(3)	LB4(2)	LB4(1)	LB4(0)	-	-	-	-	7	8	L	K
1	1	1	0	0	0	0	0	0	1	1	0

With the aid of Figure 6.6 the flow of data through the SPGA can be clearly seen in the shaded areas. From this diagram one can identify the switches (also gray-shaded) that are involved and thus setup Table 6.6 which contain the switch programming data used by the switch programming frame.

For testing purposes, the complete truth table is presented in Table 6.7. The table shows all the possible input combinations and the expected output results.

6.3 Test results

'I love deadlines. I especially love the swooshing sound they make as they go flying by.'

[Scott Adams - The Dilbert principle]

Immediately following the manufacture of the SPGA chip (incomplete at the submission date of this thesis), physical test results should become available. The intention is to present these at a conference and to publish a paper at the same time.

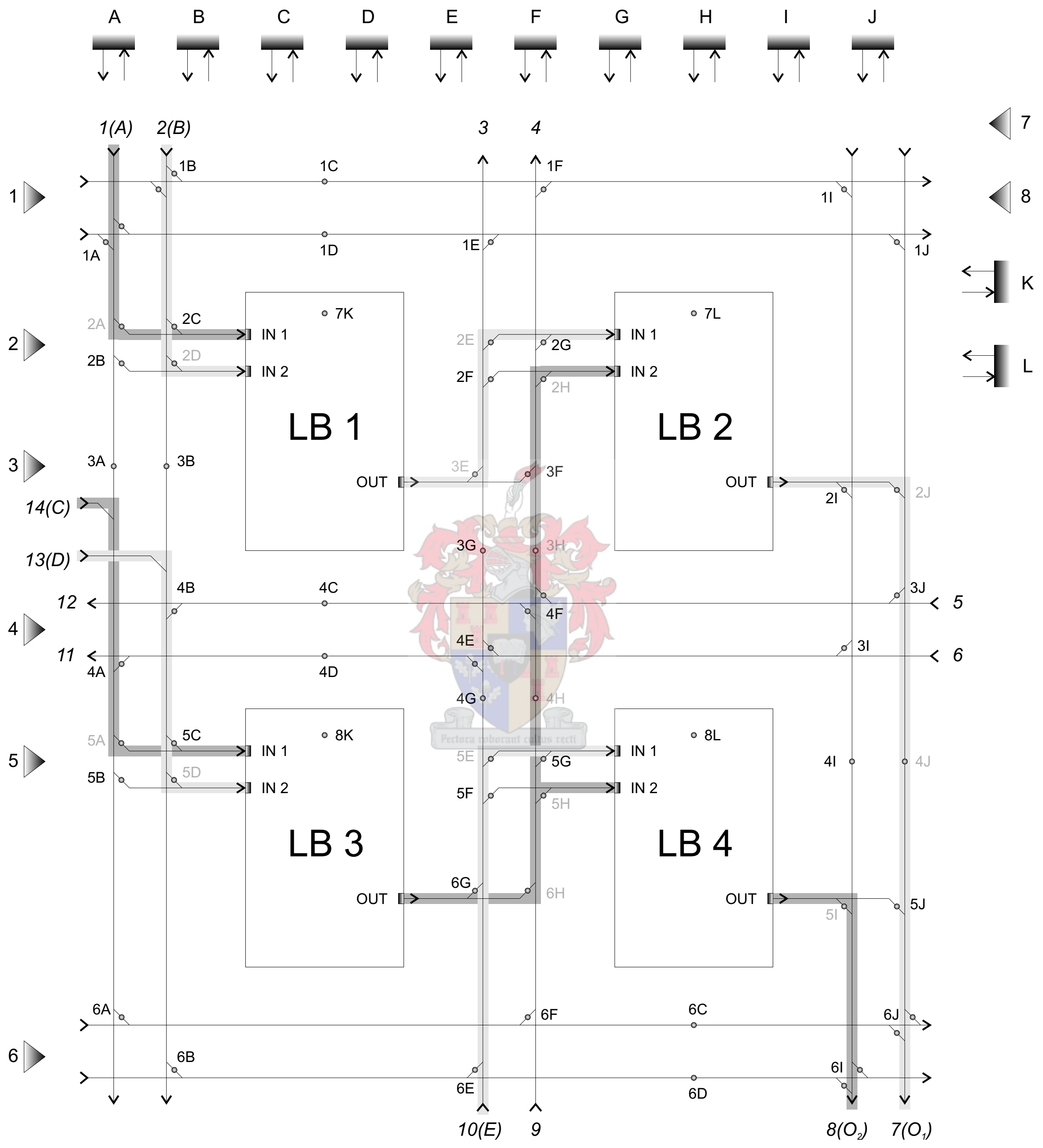

Figure 6.6: Wiring diagram showing data flow for Example 3

Table 6.6: Switch programming data words for Example 3

	6	5	4	3	2	1	J	I	H	G	F	E	D	C	B	A
a	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
c	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
d	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0
e	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
h	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0
i	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
j	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0

Table 6.7: Truth table for functions 6.2.1-6.2.2

Input							**O_1**	**O_2**
A	B	C	D	E	A XOR B	C AND D	LB1 OR LB3	LB3 NAND E
0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1
0	0	1	0	0	0	0	0	1
0	0	1	0	1	0	0	0	1
0	0	1	1	0	0	1	1	1
0	0	1	1	1	0	1	1	0
0	1	0	0	0	1	0	1	1
0	1	0	0	1	1	0	1	0
0	1	0	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0
0	1	1	0	0	1	0	1	1
0	1	1	0	1	1	0	1	0
0	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	0	1	1
1	0	0	0	1	1	0	1	0
1	0	0	1	0	1	0	1	1
1	0	0	1	1	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	0	1	1	0	1	0
1	0	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	0
1	1	0	0	0	0	0	0	1
1	1	0	0	1	0	0	0	1
1	1	0	1	0	0	0	0	1
1	1	0	1	1	0	0	0	1
1	1	1	0	0	0	0	0	1
1	1	1	0	1	0	0	0	1
1	1	1	1	0	0	1	1	1
1	1	1	1	1	0	1	1	1

Chapter 7

Conclusions and recommendations

THIS thesis considered various aspects of existing programmable logic to find an optimal solution for designing a Superconducting Programmable Gate Array. Related concepts such as technology mapping (logic function to lookup table mapping) were investigated and implemented with specific consideration for the SPGA architecture.

Associated with the design solution are RSFQ circuits that were developed and used to construct a programmable architecture. This thesis provided a detailed description of the design process involved to fully assemble the first superconducting reprogrammable logic device (with similar functionality to semiconductor FPGAs). The thesis included discussions on basic circuit gates and larger circuit structures that provide functionality needed to implement reprogrammable logic. Strategic diagrams and illustrations provide insight into the framework that constitute the overall architecture of the SPGA. Consideration for in-circuit programming lead to the development of a programming frame to allow access to the numerous switches and logic blocks.

SPICE simulations provided insight into the fundamental operation of gates while Verilog simulations provided a method to evaluate the overall behaviour of the device.

The design process included a large-scale integrated circuit layout that will be manufactured using Hypres' 3μm Nb fabrication process. The full layout included four 2-bit logic blocks (using a lookup table implementation which include a decoder, memory cell array, support for programming and a merging section) and a 2-channel routing architecture (which include various types of switch matrices). A computer aided design rule checker was implemented to minimize layout errors.

Lastly, a comprehensive testing procedure was given that should be followed to test all the functional aspects of the device.

Fabrication limitations in size and layout capacity resulted in a prototype with little functionality but is still sufficient to demonstrate the overall concept of an SPGA. Small-scale SPICE simulations and larger functional simulations showed positive results.

This project could be classified as a medium- to large scale design with more than 4000 junctions. The final stage of design, physical layout, required great manual effort and presented tremendous challenges regarding space optimization and re-usability of certain blocks. In this regard, especially in larger designs in the future, a more automated layout process is needed. By automating the process from circuit schematic to final layout including feedback from layout to schematic, a much faster turnararound time can be achieved. Not only can the project be finished faster but human error (during layout) will be minimized.

With regard to functional simulations, better models for the gates and blocks are needed that include timing parameters so that a more accurate representation can be developed. This may help identify issues regarding clock- and signal distribution.

The next step would be to complete the manufacturing process and test the prototype.

Future SPGA development can extend the functionality and broaden the capabilities of this concept as fabrication technology improves. With the proper motivation and development drive, the SPGA concept can rival semiconductor programmable logic (especially in terms of performance and power requirements).

'I never think of the future. It comes soon enough.'

[Albert Einstein - Interview (1930)]

List of References

[1] Hypres Inc., "Niobium integrated circuit fabrication process 100-1000-2500-1 revision 21c," [Online] Available: http://www.hypres.com, 2005.

[2] W. Chen, A. V. Rylyakov, J. E. Lukens, and K. K. Likharev, "Rapid Single Flux Quantum T-Flip Flop Operating up to 770 GHz," *IEEE Transactions on Applied Superconductivity*, vol. 9, no. 2, pp. 3212–3215, 1999.

[3] S. Tahara, S. Yorozu, Y. Kameda, Y. Hashimoto, H. Numata, T. Satoh, W. Hattori, and M. Hidaka, "Superconducting Digital Electronics," *IEEE Transactions on Applied Superconductivity*, vol. 11, no. 1, pp. 463–468, 2001.

[4] C. J. Fourie, "A Tool Kit for the Design of Superconducting Programmable Gate Arrays," Ph.D. dissertation, University of Stellenbosch, 2003.

[5] K. K. Likharev and V. K. Semenov, "RSFQ Logic/Memory Family: A New Josephson-Junction Technology for Sub-Terahertz-Clock-Frequency Digital Systems," *IEEE Transactions on Applied Superconductivity*, vol. 1, no. 1, pp. 3–28, 1991.

[6] K. K. Likharev, *Dynamics of Josephson Junctions and Circuits*. New York: Gordon and Breach, 1986.

[7] T. van Duzer and C. W. Turner, *Superconductive Devices and Circuits*. Prentice Hall PTR, 1999.

[8] A. M. Kadin, *Introduction to Superconducting Circuits*. John Wiley & Sons, 1999.

[9] T. Orlando and K. Delin, *Foundations of Applied Superconductivity*. New York: Addison-Wesley, 1991.

[10] K. K. Likharev, O. A. Mukhanov, and V. K. Semenov, "Quantum pulse reproduction in Josephson junction system," *Mikroelektronika (Soviet Microelectronics)*, vol. 17, 1988.

[11] A. K. Sharma, *Programmable Logic Handbook: PLDs, CPLDs and FPGAs*. McGraw-Hill Handbooks, 1998.

[12] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[13] S. Singh, J. Rose, D. Lewis, K. Chung, and P. Chow, "Optimisation of Field-Programmable Gate Array Logic Block Architecture for Speed," *Custom Integrated Circuits Conference*, pp. 6.1.1 – 6.1.6, 1991.

[14] S. Singh, "The Effect of Logic Block Architecture on FPGA Performance," Master's thesis, University of Toronto, 1991.

[15] R. J. Francis, J. Rose, and K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-based Field-Programmable Gate Array," *Proc. 27th Design Automation Conference*, pp. 613–619, 1990.

[16] R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-crt: Fast Technology Mapping for Lookup Table-based FGPAs," *Proc. 28th DAC*, pp. 227–223, 1991.

[17] ——, "Technology Mapping of Look-up Table-based FPGAs for Performance," *Proc. IDCAD-91*, 1991.

[18] R. Murgai, Y. Nishizaki, N. Shenay, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proc. 27th DAC*, pp. 620–625, 1990.

[19] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," *ICCAD*, 1991.

[20] R. Murgai, N. Shenoy, and R. K. Brayton, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," *ICCAD*, 1991.

[21] P. Abouzeid, L. Bouchet, K. Sakouti, G. Saucier, and P. Sicard, "Lexigraphical Expression of Boolean Function for Multilevel Synthesis of High Speed Circuits," *Proc. SASHIMI 90*, pp. 31–39, 1990.

[22] D. Filo, J. C. Yang, F. Mailhot, and G. de Micheli, "Technology Mapping for a Two-Output RAM-based Field Programmable Gate Array," *Proc. EDAC 91*, pp. 534–538, 1991.

[23] K. Karplus, "Xmap: a Technology Mapper for Table-lookup Field Programmable Gate Arrays," *Proc. 28th DAC*, pp. 240–243, 1991.

[24] N. Woo, "A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility," *Proc. 28th DAC*, 1991.

[25] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "An Improved Synthesis Algorithm for Multiplexor-based PGAs," *ACM/SIGDA First International Workshop on Field-Programmable Gate Arrays*, pp. 97–102, 1992.

[26] S. Ercolani and G. de Micheli, "Technology mapping for electrically programmable gate arrays," *Proc. 28th DAC*, pp. 234–239, 1991.

[27] A. Bedarida, S. Ercolani, and G. de Micheli, "A New Technology Mapping Algorithm for the Design and Evaluation of Fuse/Antifuse-based Field-Programmable Gate Arrays," *ACM/SIGDA First International Workshop on Field-Programmable Gate Arrays*, pp. 103–108, 1992.

[28] K. Karplus, "Amap: a Technology Mapper for Selector-based Field Programmable Gate Arrays," *Proc. 28th DAC*, pp. 244–247, 1991.

[29] M. Wieser, "Quine Mc Cluskey 3.10," [Online] Available: http://www.iapetus.ch/wieser/software, 2002.

[30] C. J. Fourie and W. J. Perold, "An RSFQ DC-Resettable Latch for Building Memory and Reprogrammable Circuits," *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 348–351, 2005.

[31] Stony Brook University, [Online] Available: http://pavel.physics.sunysb.edu/RSFQ/RSFQ.html.

[32] C. J. Fourie and W. J. Perold, "A Single-Clock Asynchronous Input COSL Set-Reset Flip-Flop and SFQ to Voltage State Interface," *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 263–266, 2005.

[33] M. W. Johnson, Q. P. Herr, and J. W. Spargo, "Monte-Carlo Yield Analysis," *IEEE Transactions on Applied Superconductivity*, vol. 9, no. 2, pp. 3322–3325, 1999.

[34] C. A. Hamilton and K. C. Gilbert, "Margins and Yield in Single Flux Quantum Logic," *IEEE Transactions on Applied Superconductivity*, vol. 1, no. 4, pp. 157–163, 1991.

[35] C. J. Fourie, W. J. Perold, and H. R. Gerber, "Complete Monte Carlo Model Description of Lumped-Element RSFQ Logic Circuits," *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 384–387, 2005.

[36] M. Jeffery, W. J. Perold, Z. Wang, and T. van Duzer, "Monte Carlo Optimization of Superconducting Complementary Output Switching Logic Circuits," *IEEE Transactions on Applied Superconductivity*, vol. 8, no. 3, pp. 104–119, 1998.

[37] C. J. Fourie and W. J. Perold, "Comparison of Genetic Algorithims to Other Optimization Techniques for Raising Circuit Yield in Superconducting Digital Circuits," *IEEE Transactions on Applied Superconductivity*, vol. 13, no. 2, pp. 551–514, 2003.

[38] Whiteley Research Inc., "WRspice 2.2.54," [Online] Available: http://www.srware.com, January 2004.

[39] K. Gaj, C. Cheah, E. G. Friedman, and M. J. Feldman, "Functional Modeling of RSFQ Circuits using Verilog HDL," *IEEE Transactions on Applied Superconductivity*, vol. 7, no. 2, pp. 3151–3154, 1997.

[40] Altera Corporation, "QuartusII 4.2," [Online] Available: http://www.altera.com/, 2005.

[41] D. Clein, *CMOS IC Layout: Concepts, Methodologies and Tools*. Newnes, 2000.

[42] D. E. Boyce, "LASI 7," [Online] Available: http://members.aol.com/lasicad, 2005.

[43] H. Suzuki, S. Nagasawa, K. Miyahara, and Y. Enomoto, "Characteristics of Driver and Receiver Circuits with a Passive Transmission Line in RSFQ Circuits," *IEEE Transactions on Applied Superconductivity*, vol. 10, no. 3, pp. 1637–1641, 2000.

[44] Q. P. Herr, M. S. Wire, and A. D. Smith, "Ballistic SFQ signal propagation on-chip and chip-to-chip," *IEEE Transactions on Applied Superconductivity*, vol. 13, no. 2, pp. 463–466, 2003.

[45] S. R. Whiteley, "Sline version 1.0," [Online] Available: http://www.srware.com, June 1996.

[46] B. Dimov, T. Ortlepp, H. Toepfer, and H. F. Uhlmann, "Design issues for interconnects in densely packaged RSFQ structures," *IEEE Transactions on Applied Superconductivity*, vol. 13, no. 2, pp. 498–501, 2003.

[47] R. Chadha and K. C. Gupta, "Compensation of discontinuities in planar transmission lines," *IEEE Transactions on Microwave Theory and Techniques*, vol. MTT-30, no. 12, pp. 2151–2156, December 1982.

[48] W. H. Chang, "The inductance of a superconducting strip transmission line," *Journal of Applied Physics*, vol. 50, no. 12, pp. 8129–8134, 1979.

[49] M. Kamon, M. J. Tsuk, and J. K. White, "FastHenry: A multipole-accelerated 3-D inductance extraction program," *IEEE Transactions on Microwave Theory and Techniques*, vol. 42, no. 9, pp. 1750–1758, 1994.

[50] S. R. Whiteley, "FastHenry ver. 3.0wr," [Online] Available: http://www.srware.com, February 2001.

[51] C. J. Fourie, "InductEx," [Online] Available: http://staff.ee.sun.ac.za/cjfourie/rsfq, 2004.

[52] C. J. Fourie and W. J. Perold, "Simulated Inductance Variations in RSFQ Circuit Structures," *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 300–303, 2005.

[53] H. Terai, Y. Kameda, S. Yorozu, A. Fijimaki, and Z. Wang, “The Effects of DC Bias Currents in Large-Scale SFQ Circuits,” *IEEE Transactions on Applied Superconductivity*, vol. 13, no. 2, pp. 502–506, 2003.

[54] A. M. Kadin, R. J. Webber, and S. Sarwana, “Effects of Superconducting Return Currents on RSFQ Circuit Performance,” *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 280–283, 2005.

[55] E. Tolkacheva, H. Engseth, I. Kataeva, and A. Kidiyarova-Shevchenko, “Influence of the Bias Supply Lines on the Performance of RSFQ Circuits,” *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 276–279, 2005.

[56] S. Bermon and T. Gheewala, “Moat-guarded Josephson SQUIDS,” *IEEE Transactions on Magnetics*, vol. MAG-19, no. 3, pp. 1160–1164, 1983.

[57] J.C. Weisend II, Ed., *Handbook of Cryogenic Engineering*. Taylor and Francis, 1998.

[58] Private communications with C. van Niekerk.

[59] Private communications with Dr. C.J. Fourie.

[60] K. H. G. Duh, M. W. Pospieszalski, W. F. Kopp, P. Ho, A. A. Jabra, P. Chao, P. M. Smith, L. F. Lester, J. M. Ballingall, and S. Weinreb, “Ultra-Low-Noise Cryogenic High-Electron-Mobility Transistors,” *IEEE Transactions on Electron Devices*, vol. 35, no. 3, 1998.

Appendices

Appendix A

Verilog modules

A.1 SM1

```
module sm2 ( pr_h_set, pr_v_set, pr_d, reset, h_in, v_in, h_out, v_out);

input pr_h_set, pr_v_set;
input [1:0] pr_d;
input reset;
input [1:0] h_in, v_in;
output [1:0] h_out, v_out;
reg [1:0] h_out, v_out;
reg [1:0] SRAM;
wire pr_clk;

assign pr_clk = pr_h_set && pr_v_set;

always @(posedge reset or posedge pr_clk)
begin
if (reset)
SRAM = 2'b00;
else
SRAM = pr_d;
end

always @(h_in or v_in)
begin
h_out[0] = h_in[0] || (v_in[1] && SRAM[1]);
h_out[1] = h_in[1] || (v_in[0] && SRAM[0]);
v_out[0] = v_in[0];
v_out[1] = v_in[1];
end

endmodule
```

A.2 SM_STOP

```
module sm_stop ( pr_h_set, pr_v_set, pr_d, reset, d_in, d_out);
```

```
input pr_h_set, pr_v_set;
input [1:0] pr_d;
input reset;
input [1:0] d_in;
output [1:0] d_out;
reg [1:0] d_out;
reg [1:0] SRAM;
wire pr_clk;

assign pr_clk = pr_h_set && pr_v_set;

always @(posedge reset or posedge pr_clk)
begin
if (reset)
SRAM = 2'b00;
else
SRAM = pr_d;
end

always @(d_in)
begin
d_out[0] = d_in[0] && SRAM[0];
d_out[1] = d_in[1] && SRAM[1];
end

endmodule
```

A.3 LUT_IN

```
module lut_in ( pr_h_set, pr_v_set, pr_d, reset, v_in, h_out, v_out);

input pr_h_set, pr_v_set;
input [3:0] pr_d;
input reset;
input [1:0] v_in;
output [1:0] h_out, v_out;
reg [1:0] h_out, v_out;
reg [3:0] SRAM;
wire pr_clk;

assign pr_clk = pr_h_set && pr_v_set;

always @(posedge reset or posedge pr_clk)
begin
if (reset)
SRAM = 4'b0000;
else
SRAM = pr_d;
end

always @(v_in)
begin
h_out[0] = (v_in[0] && SRAM[0]) || (v_in[1] && SRAM[1]);
```

```
h_out[1] = (v_in[0] && SRAM[2]) || (v_in[1] && SRAM[3]);
v_out[0] = v_in[0];
v_out[1] = v_in[1];
end

endmodule
```

A.4 LUT_OUT

```
module lut_out ( pr_h_set, pr_v_set, pr_d, reset, h_in, v_in, v_out);

input pr_h_set, pr_v_set;
input [1:0] pr_d;
input reset;
input h_in;
input [1:0] v_in;
output [1:0] v_out;
reg [1:0] v_out;
reg [1:0] SRAM;
wire pr_clk;

assign pr_clk = pr_h_set && pr_v_set;

always @(posedge reset or posedge pr_clk)
begin
if (reset)
SRAM = 2'b00;
else
SRAM = pr_d;
end

always @(v_in or h_in)
begin
v_out[0] = v_in[0] || (h_in && SRAM[0]);
v_out[1] = v_in[1] || (h_in && SRAM[1]);
end

endmodule
```

A.5 LB

```
module lut ( pr_clk, pr_h_set, pr_v_set, pr_d, reset, d_clk, d_in, d_out);

parameter size=4;

input [1:0] d_in;
input pr_clk, pr_h_set, pr_v_set;
input [3:0] pr_d;
input reset, d_clk;
output d_out;
reg d_out;
reg [3:0] SRAM;
```

```
always @(posedge pr_clk or posedge reset)
begin
if (reset)
SRAM = 4'b0000;
else if (pr_clk && pr_h_set && pr_v_set)
SRAM = pr_d;
end

always @(d_clk)
begin
if (d_clk)
case (d_in)
2'b00 : d_out = SRAM[0];
2'b01 : d_out = SRAM[1];
2'b10 : d_out = SRAM[2];
2'b11 : d_out = SRAM[3];
default : d_out = 1'bx;
endcase
else
d_out = 1'b0;
end

endmodule
```

A.6 SPGA

```
module spga ( pr_clk, pr_h_set, pr_v_set, pr_d, reset, d_clk, d_in, d_out);

input pr_clk;
input [8:0] pr_h_set;
input [4:0] pr_v_set;
input [3:0] pr_d;
input reset, d_clk;
input  [13:0] d_in;
output [11:0] d_out;
reg [11:0] d_out;

wire [1:0] w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w14,w15,w16,w17,w18,w19;
wire [1:0] w20,w21,w22,w23,w24,w25,w26,w27,w28,w29,w32,w33,w34,w35,w36;
wire w12,w13,w30,w31;

sm1 sm1_ul ( pr_h_set[0], pr_v_set[0], pr_d[1:0], reset,
{d_in[12], d_in[13]}, {d_in[1], d_in[0]},
{w1[1], w1[0]}, {w4[1], w4[0]} );

sm_stop sm_stop_ul ( pr_h_set[0], pr_v_set[1], pr_d[1:0], reset,
{w1[1], w1[0]}, {w2[1], w2[0]} );

sm2 sm2_um ( pr_h_set[0], pr_v_set[2], pr_d[1:0], reset,
{w2[1], w2[0]}, {w5[1], w5[0]},
{w3[1], w3[0]}, {d_out[1], d_out[0]} );

sm2 sm2_ur ( pr_h_set[0], pr_v_set[4], pr_d[1:0], reset,
{d_in[2], d_in[3]}, {w3[1], w3[0]},
{w6[0], w6[1]}, {d_out[3], d_out[2]} );

lut_in lut_in_ul ( pr_h_set[1], pr_v_set[0], pr_d[3:0], reset,
{w4[1], w4[0]},
{w7[1], w7[0]}, {w9[1], w9[0]} );

lut lut_ul ( pr_clk, pr_h_set[1], pr_v_set[1], pr_d, reset, d_clk,
{w7[1], w7[0]}, {w12} );

lut_in lut_in_ur ( pr_h_set[1], pr_v_set[2], pr_d[3:0], reset,
{w10[1], w10[0]},
{w8[0], w8[1]}, {w5[1], w5[0]} );

lut lut_ur ( pr_clk, pr_h_set[1], pr_v_set[3], pr_d, reset, d_clk,
{w8[1], w8[0]}, {w13} );

sm_stop sm_stop_lum ( pr_h_set[2], pr_v_set[0], pr_d[1:0], reset,
{w9[1], w9[0]}, {w11[1], w11[0]} );

lut_out lut_out_ul ( pr_h_set[2], pr_v_set[2], pr_d[1:0], reset,
{w12}, {w14[1], w14[0]},
{w10[1], w10[0]} );

lut_out lut_out_ur ( pr_h_set[2], pr_v_set[4], pr_d[1:0], reset,
{w13}, {w6[1], w6[0]},
{w15[1], w15[0]} );

sm_stop sm_stop_mum ( pr_h_set[3], pr_v_set[2], pr_d[1:0], reset,
{w17[1], w17[0]}, {w14[1], w14[0]} );

sm2 sm2_ml ( pr_h_set[4], pr_v_set[0], {pr_d[0], pr_d[1]}, reset,
{w16[1], w16[0]}, {w18[1], w18[0]},
{w21[1], w21[0]}, {d_out[10], d_out[11]} );

sm_stop sm_stop_ml ( pr_h_set[4], pr_v_set[1], {pr_d[1], pr_d[0]}, reset,
{w19[1], w19[0]}, {w18[1], w18[0]} );

sm1 sm1_mm ( pr_h_set[4], pr_v_set[2], {pr_d[1], pr_d[0]}, reset,
{w22[0], w22[1]}, {w20[0], w20[1]},
{w17[0], w17[1]}, {w19[0], w19[1]} );

sm2 sm2_mr ( pr_h_set[4], pr_v_set[4], {pr_d[0], pr_d[1]}, reset,
{d_in[4], d_in[5]}, {w15[0], w15[1]},
{w20[0], w20[1]}, {w23[0], w23[1]} );

sm_stop sm_stop_mlm ( pr_h_set[5], pr_v_set[2], {pr_d[1], pr_d[0]}, reset,
{w24[1], w24[0]}, {w22[1], w22[0]} );

lut_in lut_in_ll ( pr_h_set[6], pr_v_set[0], pr_d[3:0], reset,
{w21[1], w21[0]},
{w25[1], w25[0]}, {w27[1], w27[0]} );

lut lut_ll ( pr_clk, pr_h_set[6], pr_v_set[1], pr_d, reset, d_clk,
{w25[1], w25[0]}, {w30} );
```

```
lut_in lut_in_lr ( pr_h_set[6], pr_v_set[2], pr_d[3:0], reset,
{w28[1], w28[0]},
{w26[0], w26[1]}, {w24[1], w24[0]} );

lut lut_lr ( pr_clk, pr_h_set[6], pr_v_set[3], pr_d, reset, d_clk,
{w26[1], w26[0]}, {w31} );

sm_stop sm_stop_mlr ( pr_h_set[6], pr_v_set[4], {pr_d[1], pr_d[0]}, reset,
{w23[1], w23[0]}, {w29[1], w29[0]} );

lut_out lut_out_ll ( pr_h_set[7], pr_v_set[2], pr_d[1:0], reset,
{w30}, {w32[1], w32[0]},
{w28[1], w28[0]} );

lut_out lut_out_lr ( pr_h_set[7], pr_v_set[4], pr_d[1:0], reset,
{w31}, {w29[1], w29[0]},
{w33[1], w33[0]} );

sm2 sm2_lr ( pr_h_set[8], pr_v_set[0], {pr_d[1], pr_d[0]}, reset,
{d_in[9], d_in[8]}, {w27[1], w27[0]},
{w34[0], w34[1]}, {d_out[8], d_out[9]} );

sm2 sm2_lm ( pr_h_set[8], pr_v_set[2], {pr_d[1], pr_d[0]}, reset,
{d_in[7], d_in[6]}, {w34[0], w34[1]},
{w32[0], w32[1]}, {w35[0], w35[1]} );

sm_stop sm_stop_lr ( pr_h_set[8], pr_v_set[3], {pr_d[1], pr_d[0]}, reset,
{w35[1], w35[0]}, {w36[1], w36[0]} );

sm1 sm1_lr ( pr_h_set[8], pr_v_set[4], {pr_d[1], pr_d[0]}, reset,
{w36[1], w36[0]}, {w33[1], w33[0]},
{d_out[5], d_out[4]}, {d_out[6], d_out[7]} );

assign {w16[1], w16[0]} = {(w11[1] || d_in[10]), (w11[0] || d_in[11])};

endmodule
```

Appendix B

Spice code

B.1 I2-switch

```
.monte
.exec
checkSTP1=15
checkSTP2=15
* global variations
let Jtol = gauss(0.1/3,1)
let Ctol = gauss(0.05/3,1)
let Rtol = gauss(0.2/3,1)
let Ltol = gauss(0.1/3,1)

let Cmax = gauss(0.0362,1)
let Cmin = gauss(0.0363,1)

let Kvar1 = gauss(0.019432,1)
let Kvar2 = gauss(0.023397,1)
let Kvar3 = gauss(0.037535,1)
let Kvar4 = gauss(0.011211,1)
let Kvar5 = gauss(0.025501,1)
let Kvar6 = gauss(0.021321,1)

.endc

.control
if (tg1*504e-12) > 0.5f
  let checkFAIL=1
end
if (tp1*40e-12) < 1.5f or (tp1*40e-12) > 2.5f
  let checkFAIL=1
end
if (tg2*225e-12) > 0.5f
  let checkFAIL=1
end
.endc

* local variations
.param Jvar = Jtol*gauss(0.05/3,1)
```

```
.param Avar = gauss(0.05/3,1)
.param Rvar = Rtol*gauss(0.05/3,1)
.param Lvar = Ltol*gauss(0.15/3,1)
.measure tran tg1 from=20p to=524p avg v(2)
.measure tran tp1 from=529p to=569p avg v(2)
.measure tran tg2 from=574p to=799p avg v(2)

.save v(7)
.save V5#branch
.save V6#branch
.save V(2)

.tran 1p 800p 0 0.5p UIC

B0 13 0 29 jjmc1 area=$&(0.2*Avar)
B1 16 0 28 jjmc2 area=$&(0.2*Avar)
B2 8 4 27 jjmc3 area=$&(0.245*Avar)
B3 4 5 26 jjmc4 area=$&(0.27*Avar)
B4 1 0 25 jjmc5 area=$&(0.27*Avar)

I0 0 18 pwl(0 0 10p $&(-333u*Cmin) 390p $&(-333u*Cmin) 410p $&(339u*Cmax))
I1 0 17 pwl(0 0 10p $&(-333u*Cmin) 190p $&(-333u*Cmin) 210p $&(339u*Cmax) 590p $&(339u*Cmax) 610p $&(-333u*Cmin))

I2 0 24 dc 85u

K1 L4 L6 $&(0.2556*Kvar1)
K2 L7 L5 $&(0.2556*Kvar1)
K3 L0 L6 $&(0.2575*Kvar2)
K4 L3 L5 $&(0.2575*Kvar2)
K5 L0 L4 $&(0.0991*Kvar3)
K6 L3 L7 $&(0.0991*Kvar3)
K7 L2 L6 $&(0.4181*Kvar4)
K8 L1 L5 $&(0.4181*Kvar4)
K9 L0 L1 $&(0.1254*Kvar5)
K10 L3 L2 $&(0.1254*Kvar5)
K11 L4 L1 $&(0.1244*Kvar6)
K12 L7 L2 $&(0.1244*Kvar6)

L0 22 18 $&(3p*Lvar)
L1 24 23 $&(6.1p*Lvar)
L2 23 0 $&(6.1p*Lvar)
L3 15 22 $&(3p*Lvar)
L4 21 17 $&(3p*Lvar)
L5 13 12 $&(1.7p*Lvar)
L6 12 11 $&(1.7p*Lvar)
L7 9 21 $&(3p*Lvar)
L8 12 20 $&(1.2p*Lvar)
L9 4 19 $&(1.98p*Lvar)
L10 4 1 $&(0.132p*Lvar)

R0 14 16 $&(9*Rvar)
R1 13 0 $&(1*Rvar)
R2 16 0 $&(1*Rvar)
R3 8 4 $&(1.2*Rvar)
R4 2 0 $&(5*Rvar)
```

```
R5 5 4 $&(1.14*Rvar)
R6 1 0 $&(1.14*Rvar)

V0 11 16 dc 0
V1 14 0 $&(2.6m*Rtol*Jtol)
V2 10 0 $&(2.6m*Rtol*Jtol)
V3 20 8 dc 0
V4 7 0 pulse(0 824u 50p 2p 3p 0 150p)
V5 15 0
V6 9 0

X0 10 7 6 jtl_250uA
X1 14 6 5 jtl_250uA
X2 14 19 3 jtl_250uA
X3 14 3 2 jtl_250uA

.subckt jtl_250uA 11 10 9
B0 2 0 8 jj1 area=0.25
B1 1 0 7 jj1 area=0.25
L0 6 5 0.132p
L1 10 4 1.98p
L2 4 5 1.98p
L3 5 3 1.98p
L4 3 9 1.98p
L5 4 2 0.132p
L6 3 1 0.132p
R0 11 6 7.4
R1 2 0 1.14
R2 1 0 1.14
.ends jtl_250uA

.model jj1 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=1m, r0=30, rn=1.64706, cap=3.8065p)
*Nb 1000 A/cm2   area = 100 square microns (generated by JJMODEL)
.model jjmc1 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(1m*Jvar), r0=30, rn=1.64706, cap=$&(5.0p*Ctol))
*Nb 1000 A/cm2 (Hypres 3u-process) area = 100 square microns : variations by MConvert
.model jjmc2 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(1m*Jvar), r0=30, rn=1.64706, cap=$&(5.0p*Ctol))
*Nb 1000 A/cm2 (Hypres 3u-process) area = 100 square microns : variations by MConvert
.model jjmc3 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(1m*Jvar), r0=30, rn=1.64706, cap=$&(5.0p*Ctol))
*Nb 1000 A/cm2 (Hypres 3u-process) area = 100 square microns : variations by MConvert
.model jjmc4 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(1m*Jvar), r0=30, rn=1.64706, cap=$&(5.0p*Ctol))
*Nb 1000 A/cm2 (Hypres 3u-process) area = 100 square microns : variations by MConvert
.model jjmc5 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(1m*Jvar), r0=30, rn=1.64706, cap=$&(5.0p*Ctol))
*Nb 1000 A/cm2 (Hypres 3u-process) area = 100 square microns : variations by MConvert
```

Appendix C

Circuit schematics

C.1 Basic gates

C.1.1 AND gate

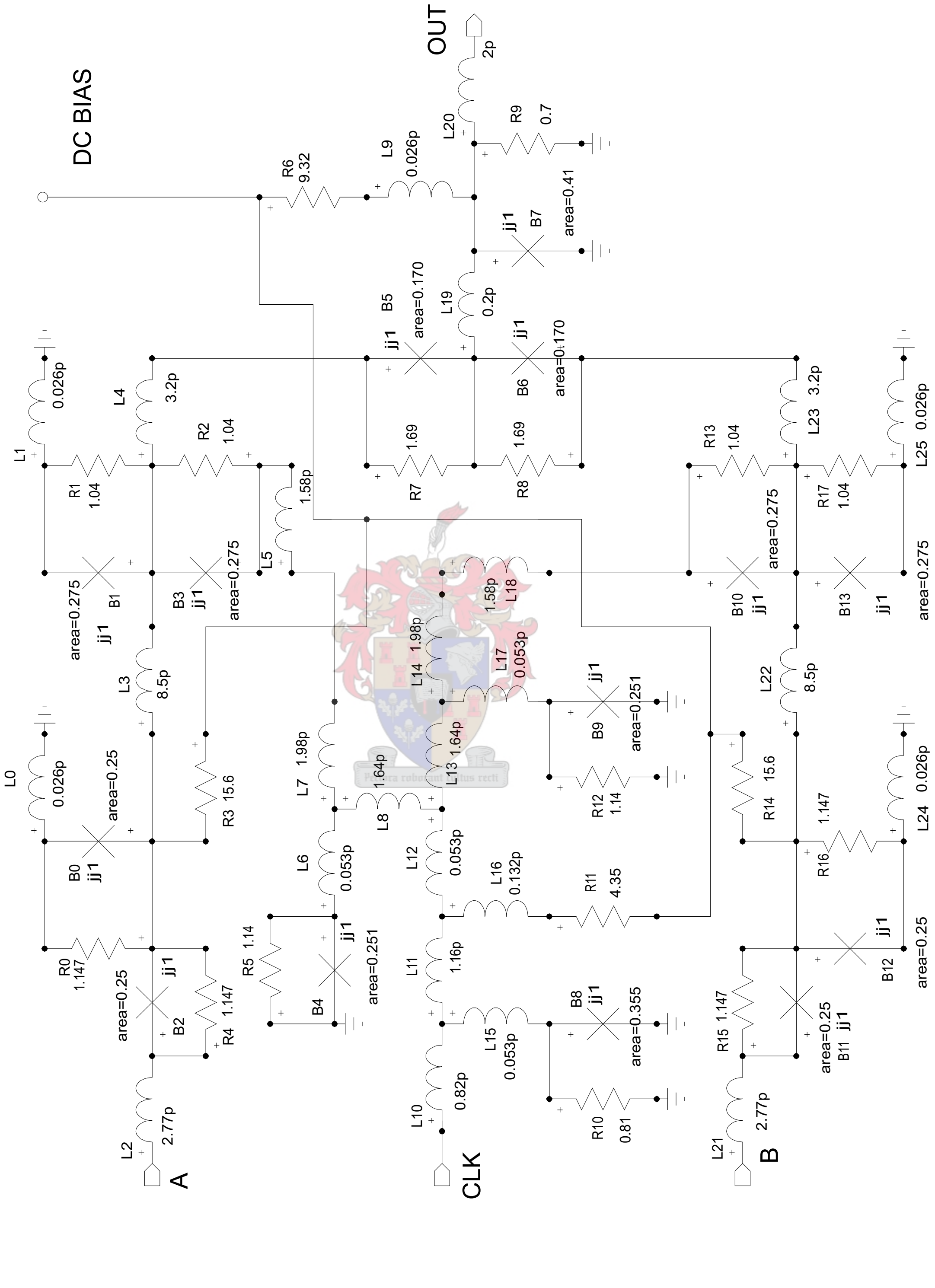

Figure C.1: Circuit diagram: AND gate

C.1.2 DC-to-SFQ Converter

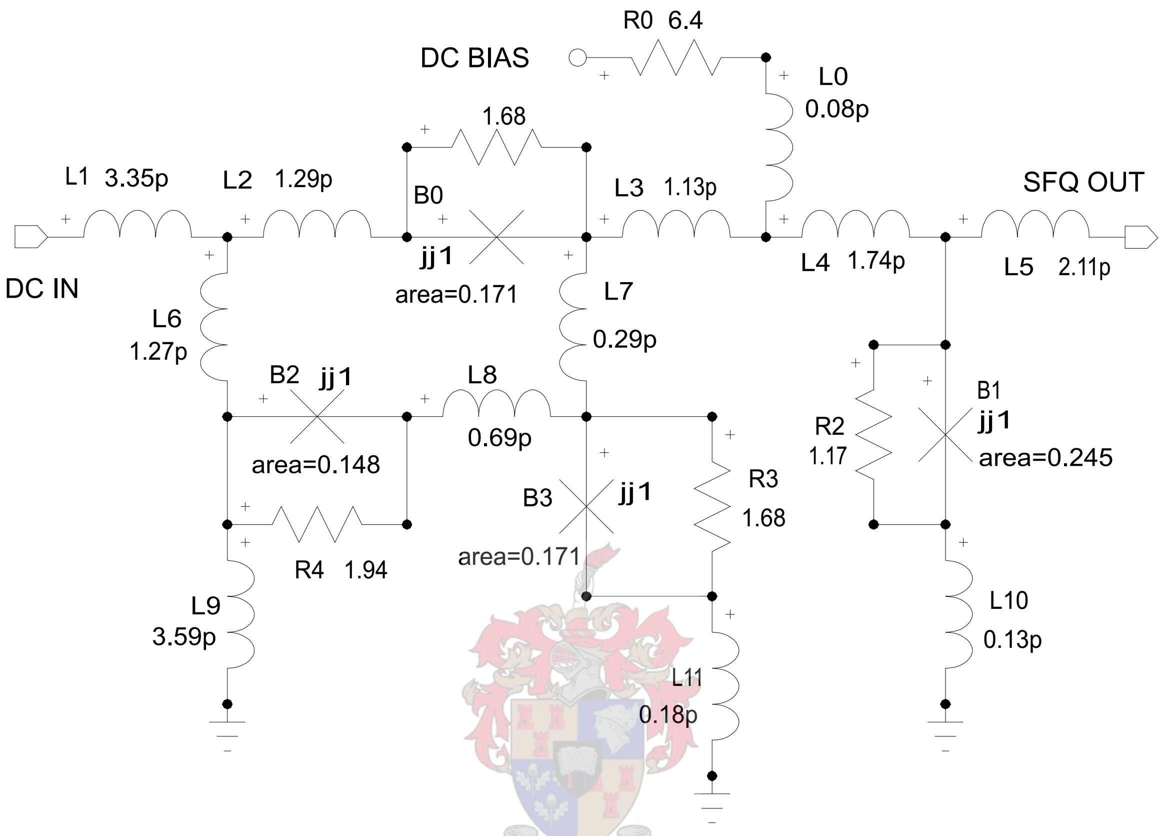

Figure C.2: Circuit diagram: DC-to-SFQ converter

C.1.3 Divider

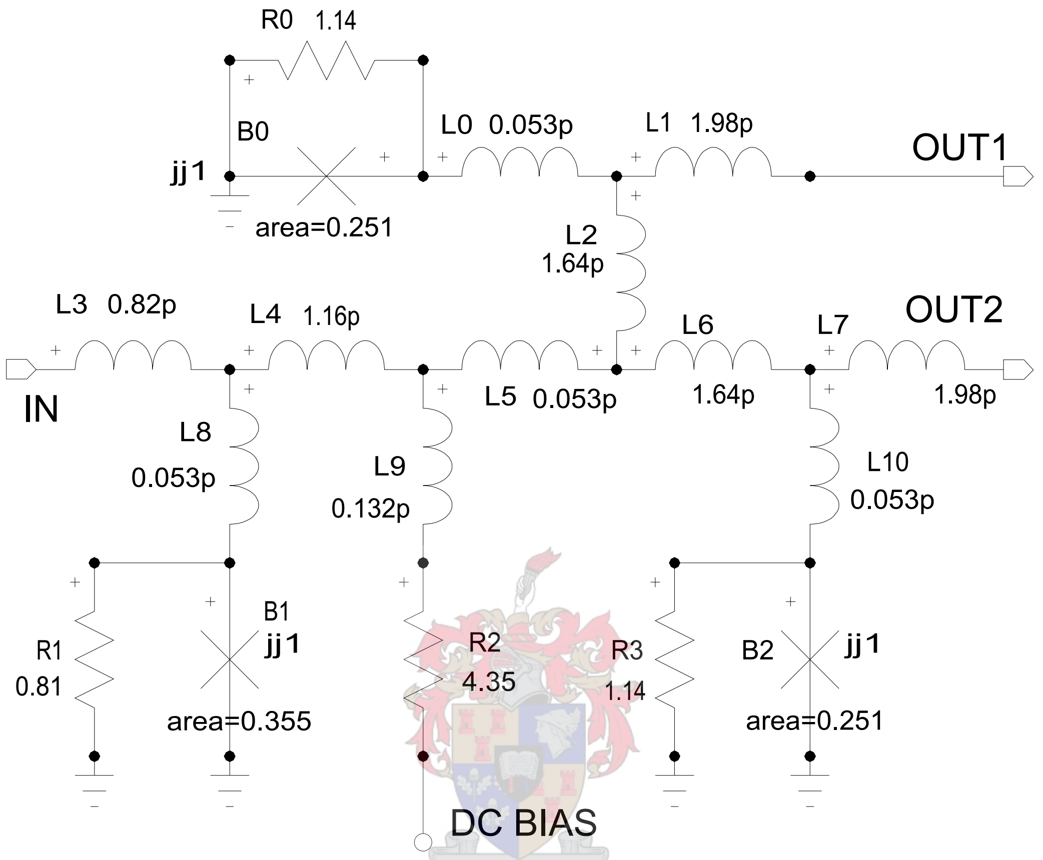

Figure C.3: Circuit diagram: Divider

C.1.4 DRO

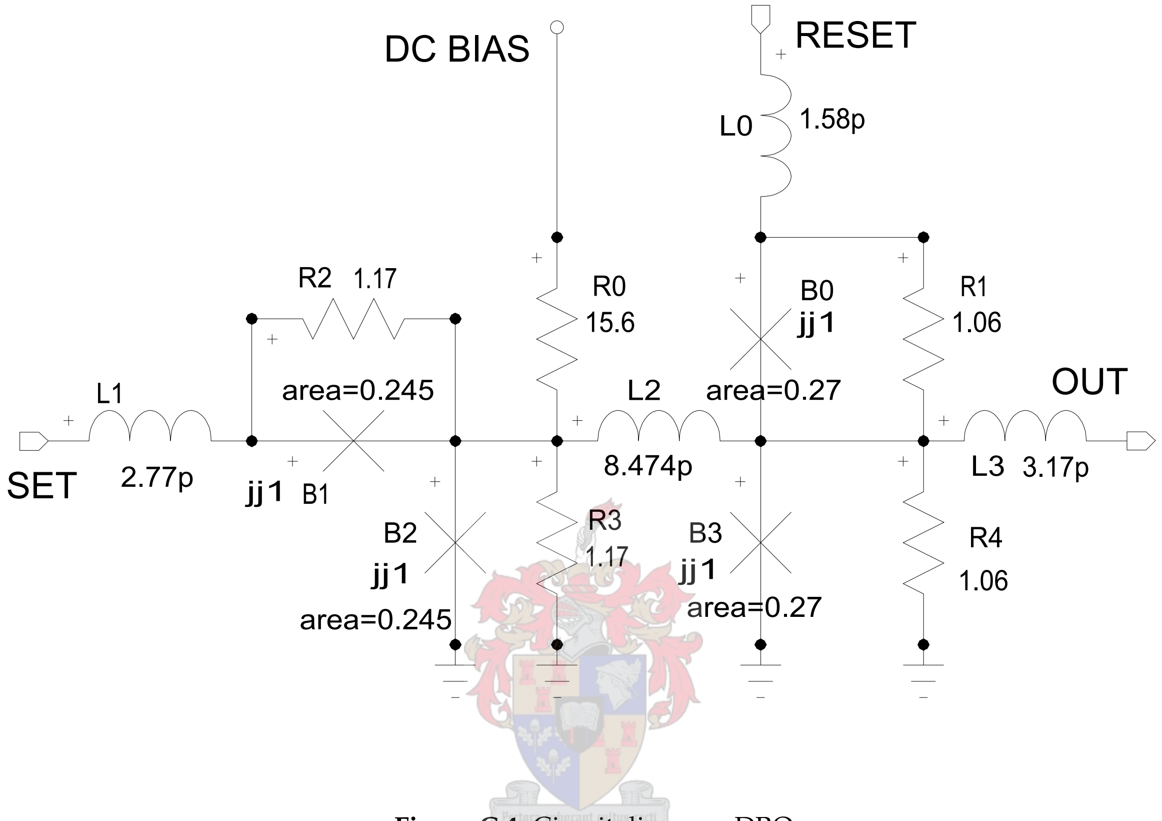

Figure C.4: Circuit diagram: DRO

C.1.5 JTL (250μA)

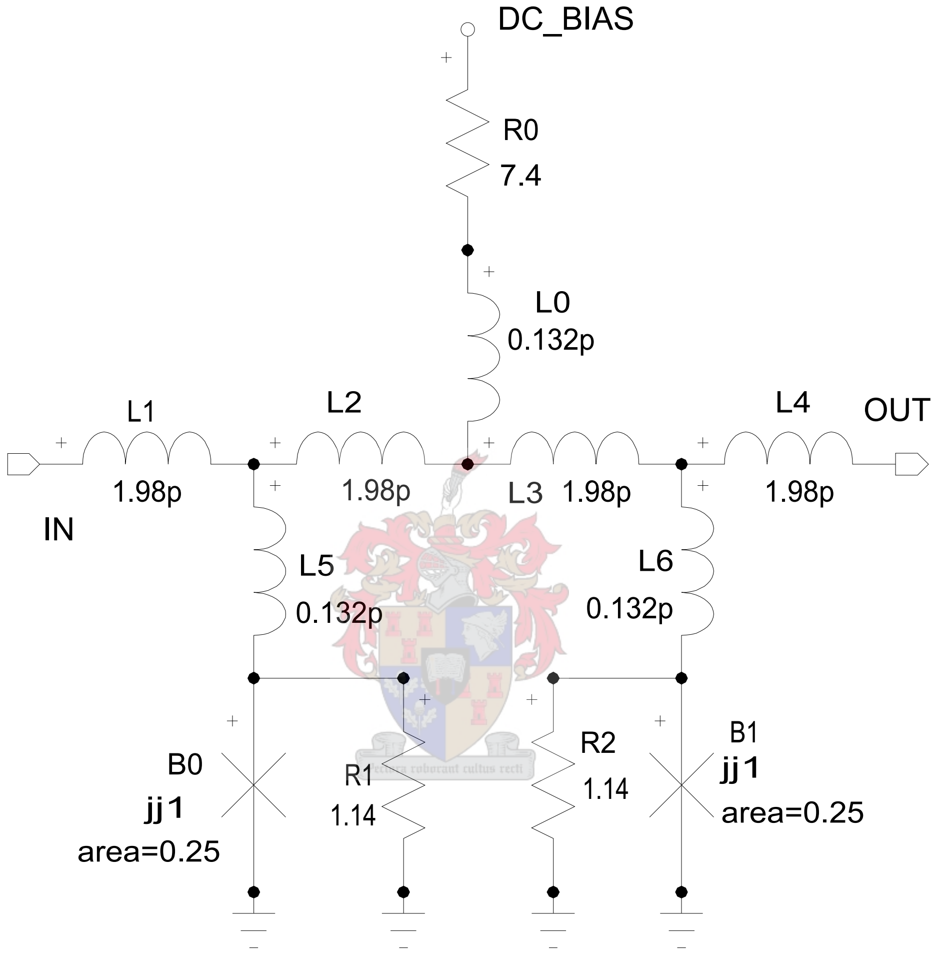

Figure C.5: Circuit diagram: 250μA Josephson Transmission Line

C.1.6 Merger

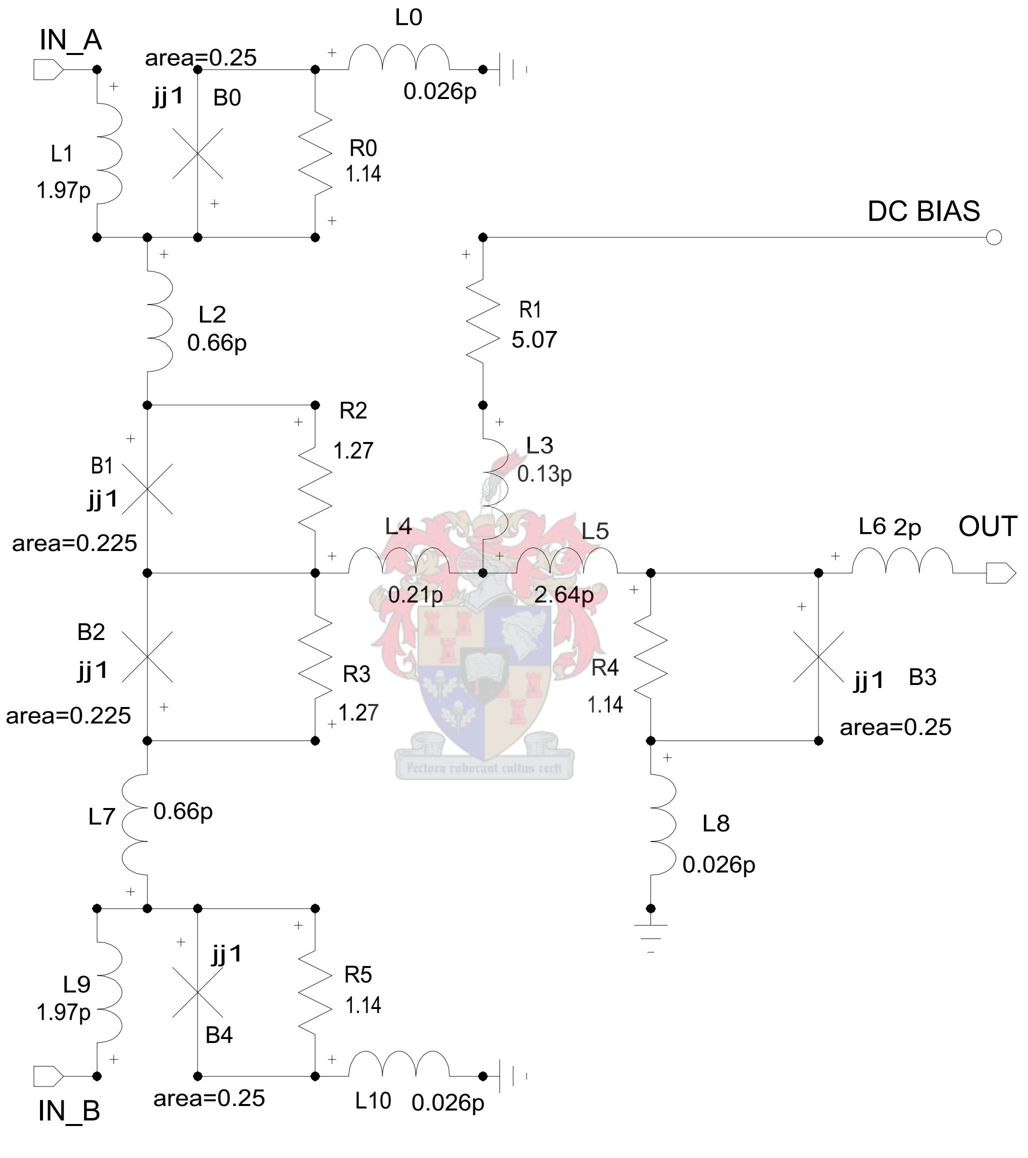

Figure C.6: Circuit diagram: Merger

Appendix D

Layouts

D.1 Basic gates

D.1.1 DC-to-SFQ converter

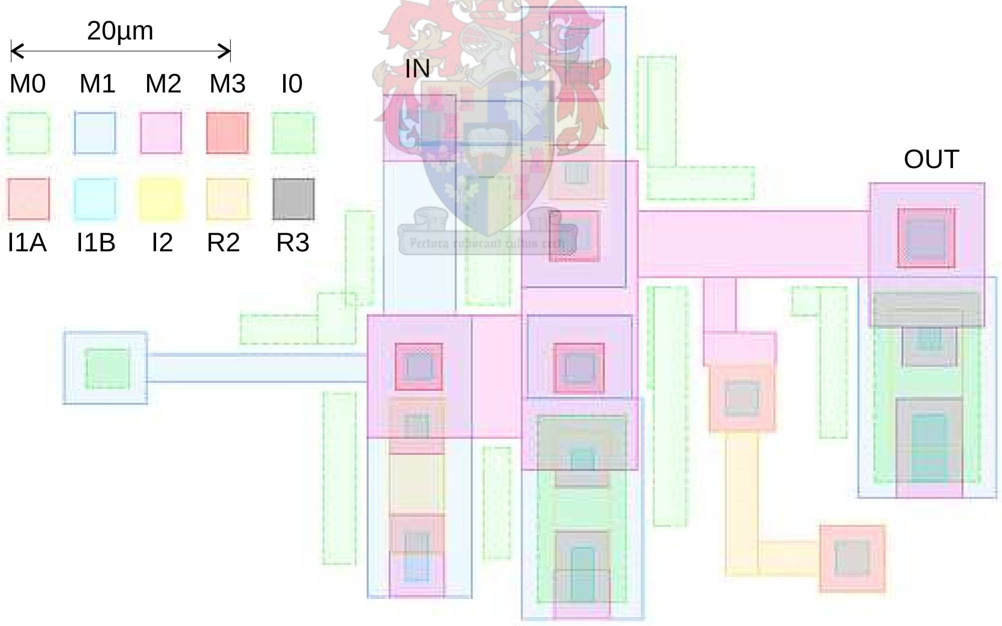

Figure D.1: Layout: DC-to-SFQ converter

D.1.2 Divider

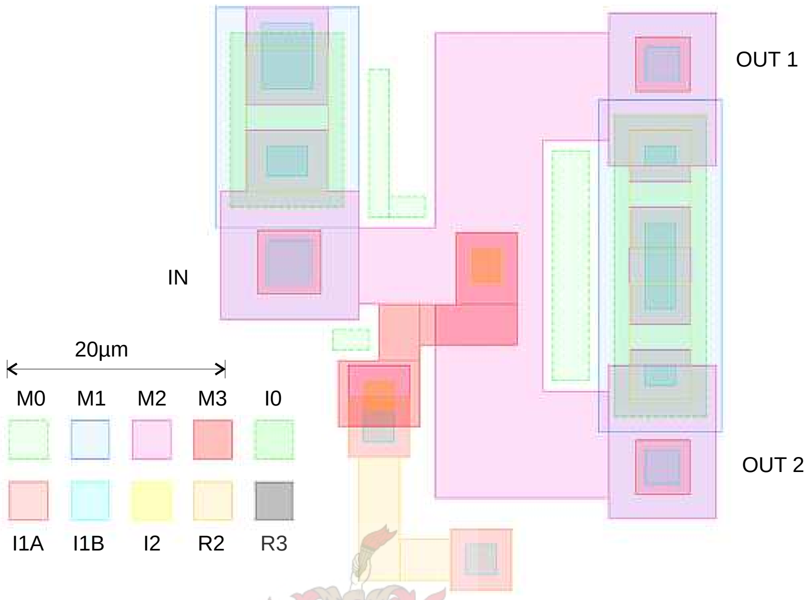

Figure D.2: Layout: Divider gate

D.1.3 DRO

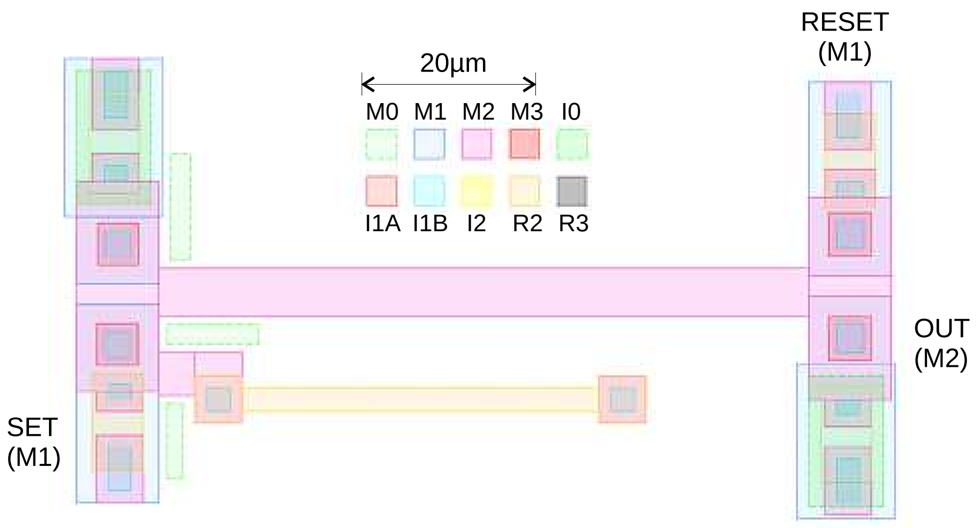

Figure D.3: Layout: DRO gate

D.1.4 HUFFLE

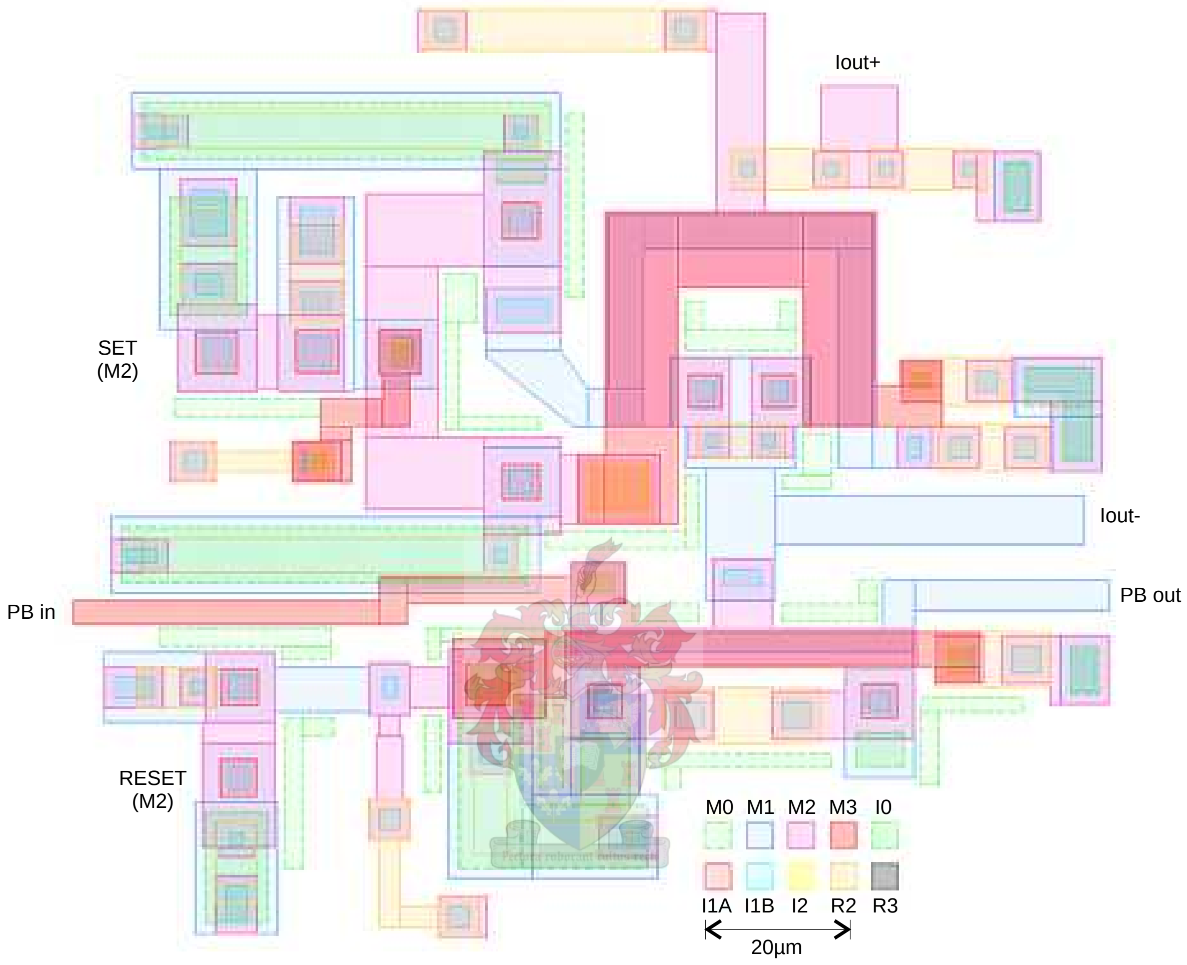

Figure D.4: Layout: HUFFLE gate

D.1.5 I-Switch

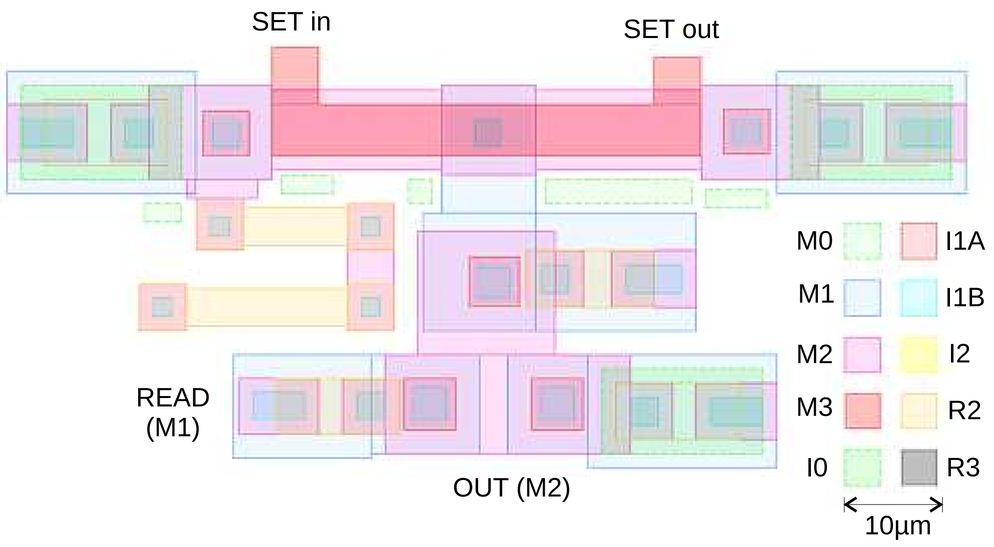

Figure D.5: Layout: Current-Select switch

D.1.6 JTL250

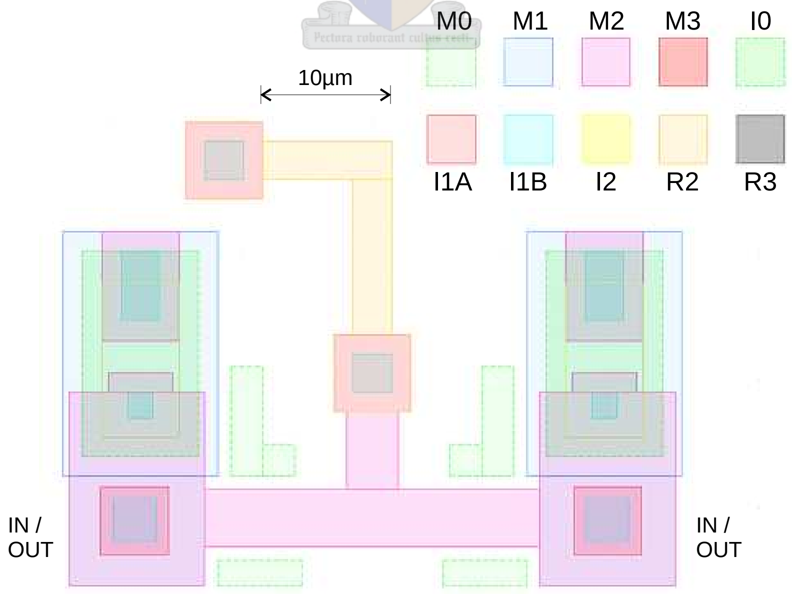

Figure D.6: Layout: 250μA Josephson transmission line gate

D.1.7 Merger

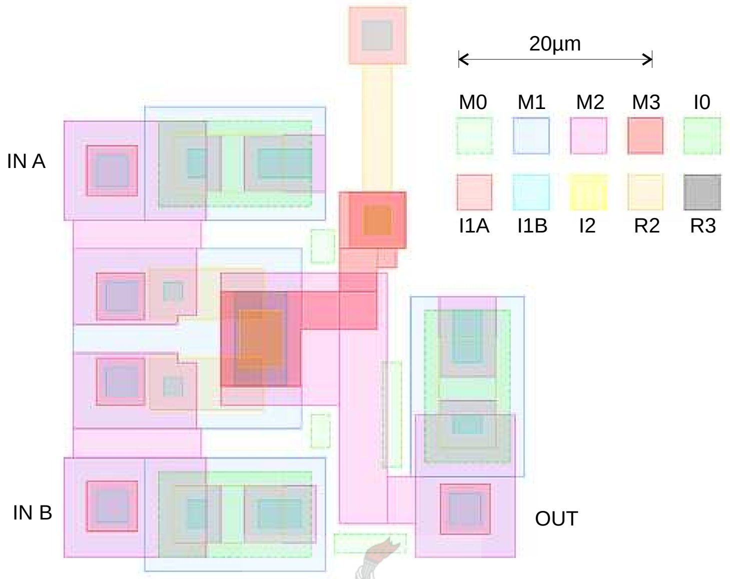

Figure D.7: Layout: Merger

D.1.8 MSL driver

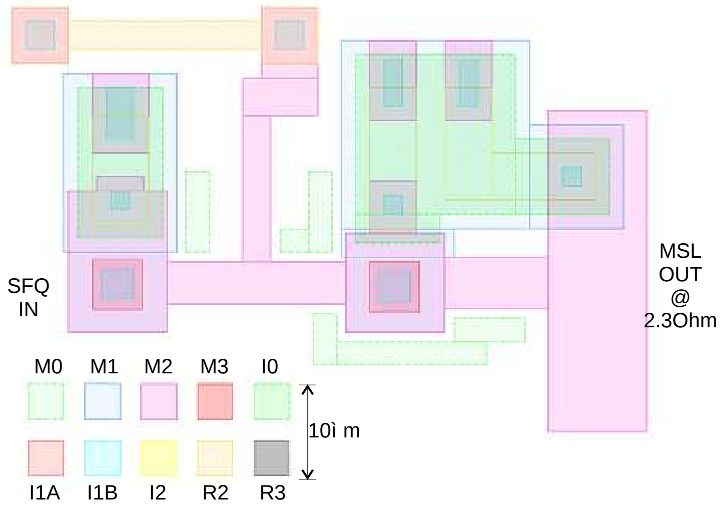

Figure D.8: Layout: Microstrip transmission line driver gate

D.1.9 MSL receiver

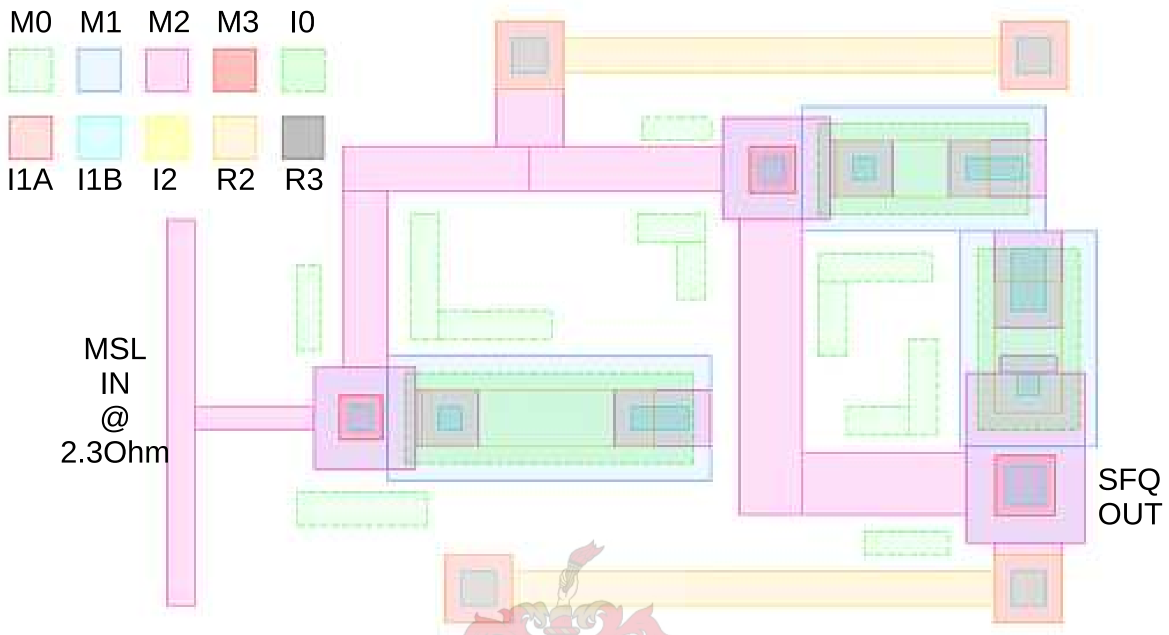

Figure D.9: Layout: Microstrip transmission line receiver gate

D.1.10 RSFQ-to-COSL converter

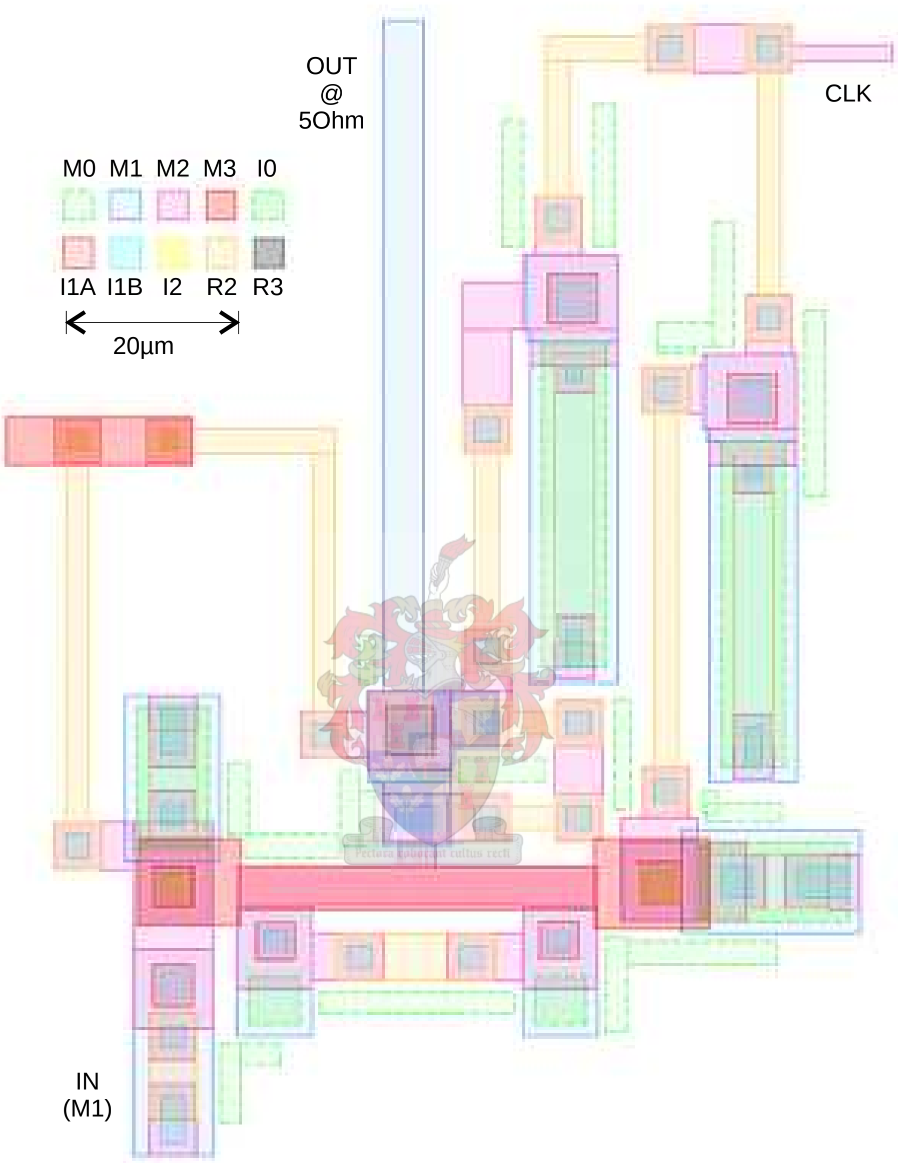

Figure D.10: Layout: RSFQ-to-COSL converter

D.2 Composite blocks

D.2.1 Inline Switch

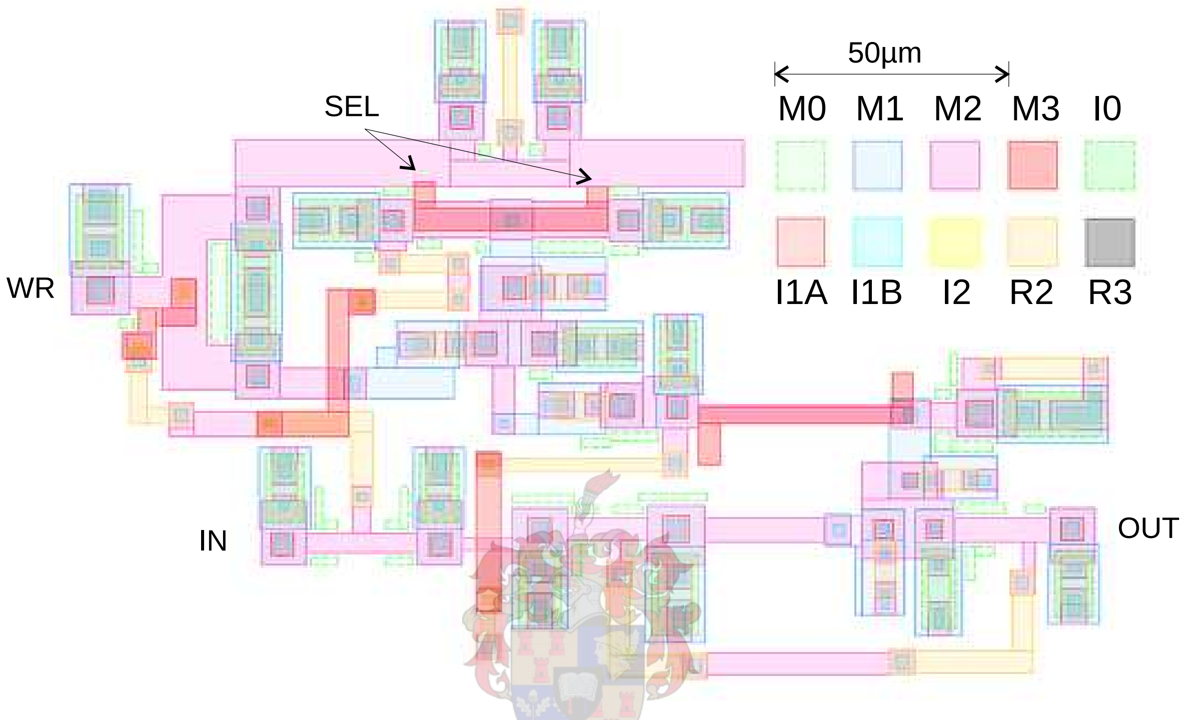

Figure D.11: Layout: Inline Switch

D.2.2 Crossbar Switch

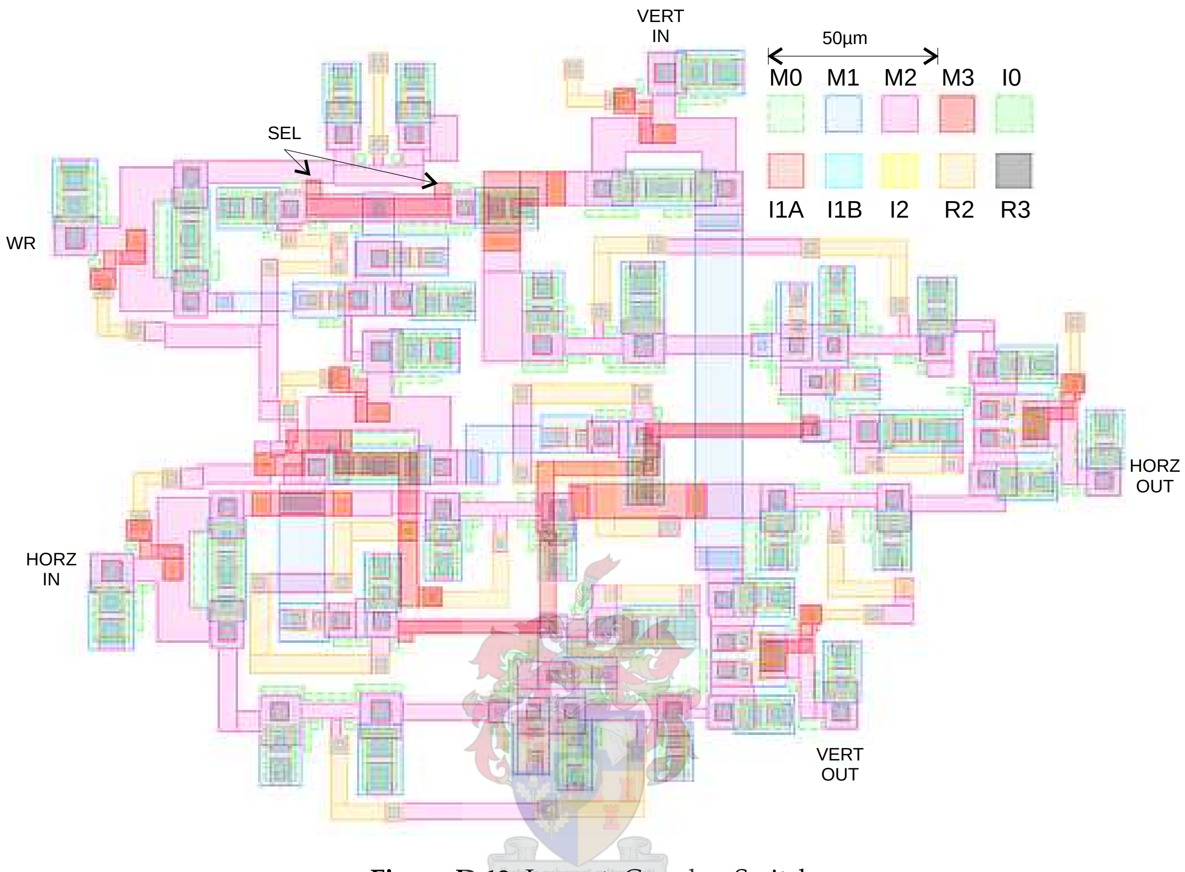

Figure D.12: Layout: Crossbar Switch

Appendix E

Design Rule Checker listing

```
title=M0 layer spacing >= 2 um (Rule 1.1)
resolution=0.1
distance=2
operators=
{
map,30       ;map M0 layer
push,30      ;push it
jrc         ;nothing to do
expr        ;expand M0 layer
jrn         ;test results
dsp,30,w,2   ;display M0 layer in grey
dspr,Y,1    ;display result in yellow
scpy
}
title=M0 layer width >= 2 um (Rule 1.2)
resolution=0.1
distance=2
operators=
{
map,30       ;map M0 layer
push,30      ;push it
jrc         ;nothing to do
notr        ;invert M0 layer
expr        ;expand M0 layer
jrn         ;test results
dsp,30,w,2
dspr,Y,1
scpy
}
title=M0 spacing to I0 >= 1.5 um (Rule 1.3a)
resolution=0.2
distance=1.5
operators=
{
map,30 ;map M0
map,31 ;map I0
push,30 ;M0
jrc
```

```
push,31 ;I0
jrc
andrs
push,31 ;I0
xorrs
push,30 ;M0
exprs
jrn
dsp,30,g,2 ;M0
dsp,31,w,2 ;I0
dspr,y,1
scpy
}
title=M0 surround I0 >= 1.5 um (Rule 1.3b)
resolution=0.2
distance=1.5
operators=
{
map,30 ;map M0
map,31 ;map I0
push,30  ;M0
jrc
push,31 ;I0
jrc
andrs
push,30 ;M0
notr
exprs
jrn
dsp,30,g,2
dsp,31,w,2
dspr,y,1
scpy
}
title=M0 spacing to M1 >= 1 um (Rule 1.4a)
resolution=0.1
distance=1
operators=
{
map,30 ;map M0
map,1 ;map M1
push,30 ;M0
jrc
push,1 ;M1
jrc
andrs
push,1 ;M1
xorrs
push,30 :M0
exprs
jrn
dsp,30,g,2 ;M0
dsp,01,b,2 ;M1
dspr,y,1
scpy
```

```
}
title=M0 surround M1 >= 1 um (Rule 1.4b)
resolution=0.1
distance=1
operators=
{
map,30 ;map M0
map,1 ;map M1
push,1   ;M0
jrc
push,30 ;M0
jrc
andrs
push,30 ;M0
notr
exprs
jrn
dsp,30,g,2 ;M0
dsp,1,b,2 ;M1
dspr,y,1
scpy
}
title=M0 spacing to R2 >= 1.5 um (Rule 1.5a)
resolution=0.2
distance=1.5
operators=
{
map,30 ;map M0
map,9 ;map R2
push,9 ;R2
jrc
push,30 ;M0
jrc
andrs
push,9 ;R2
xorrs
push,30
exprs
jrn
dsp,30,g,2 ;M0
dsp,09,w,2 ;R2
dspr,y,1
scpy
}
title=M0 surround R2 >= 1.5 um (Rule 1.5b)
resolution=0.2
distance=1.5
operators=
{
map,30 ;map M0
map,9 ;map R2
push,9   ;R2
jrc
push,30 ;M0
jrc
```

```
andrs
push,30 ;M0
notr
exprs
jrn
dsp,30,g,2 ;M0
dsp,9,w,2 ;R2
dspr,y,1
scpy
}
title=I0 width >= 2.5 um (Rule 2.1)
resolution=0.3
distance=2.5
operators=
{
map,31       ;map I0 layer
push,31      ;push it
jrc         ;nothing to do
notr        ;invert I0 layer
expr        ;expand I0 layer
jrn         ;test results
dsp,31,w,2
dspr,Y,1
scpy
}
title=I0 spacing to I1A >= 2.0 (Rule 2.2)
resolution=0.2
distance=2
operators=
{
map,31 ;map I0
map,2 ;map I1A
push,31
jrc
push,2
jrc
andrs
push,31
xorrs
push,2
exprs
jrn
dsp,31,g,2
dsp,2,y,2
dspr,y,1
scpy
}
title=I0 surrounded by M1 >= 1.5 (Rule 2.3)
resolution=0.2
distance=1.5
operators=
{
map,31 ;map I0
map,1 ;map M1
push,1
```

```
jrc
push,31
jrc
andrs
push,1
notr
exprs
jrn
dsp,1,b,2
dsp,31,g,2
dspr,y,1
scpy
}
title=I0 spacing to R2 >= 1.5 (Rule 2.4)
resolution=0.2
distance=1.5
operators=
{
map,31 ;map I0
map,9 ;map R2
push,9
jrc
push,31
jrc
andrs
push,9
xorrs
push,31
exprs
jrn
dsp,31,g,2
dsp,9,w,2
dspr,y,1
scpy
}
title=I1A layer spacing >= 2 um (Rule 3.1)
resolution=0.1
distance=2
operators=
{
map,2       ;map I1A layer
push,2      ;push it
jrc         ;nothing to do
expr        ;expand M0 layer
jrn         ;test results
dsp,2,y,2   ;display M0 layer in grey
dspr,Y,1    ;display result in yellow
pause
scpy
}
title=Minimum I1A size >= 3.6 (Rule 3.2)
resolution=0.3
distance=3.6
operators=
{
```

```
map,2       ;map I1A layer
push,2      ;push it
jrc         ;nothing to do
notr        ;invert I1A layer
expr        ;expand I1A layer
jrn         ;test results
dsp,2,y,2
dspr,Y,1
pause
scpy
}
title=I1A spacing to M1 >= 2.0 (Rule 3.3a)
resolution=0.2
distance=2.0
operators=
{
map,2 ;map I1A
map,1 ;map M1
push,2
jrc
push,1
jrc
andrs
push,2
xorrs
push,1
exprs
jrn
dsp,1,b,2
dsp,2,y,2
dspr,y,1
scpy
}
title=I1A surrounded by M1 >= 2.0 um (Rule 3.3b)
resolution=0.2
distance=2.0
operators=
{
map,2 ;map I1A
map,1 ;map M1
push,2   ;
jrc
push,1 ;
jrc
andrs
push,1 ;
notr
exprs
jrn
dsp,1,b,2 ;
dsp,2,y,2 ;
dspr,y,1
scpy
}
title=I1A spacing to R2 >= 0.5 (Rule 3.4)
```

```
resolution=0.1
distance=0.5
operators=
{
map,2 ;map I1A
map,9 ;map R2
push,2
jrc
push,9
jrc
andrs
push,2
xorrs
push,9
exprs
jrn
dsp,2,y,2
dsp,9,w,2
dspr,y,1
scpy
}
title=I1A surround I1B >= 0.8 um (Rule 3.5)
resolution=0.1
distance=0.8
operators=
{
map,2 ;map I1A
map,3 ;map I1B
push,2   ;
jrc
push,3 ;
jrc
andrs
push,2 ;
notr
exprs
jrn
dsp,2,y,2 ;
dsp,3,c,1 ;
dspr,y,1
scpy
}
title=M1 layer spacing >= 2.5 um (Rule 4.1)
resolution=0.2
distance=2.5
operators=
{
map,1       ;map M1 layer
push,1      ;push it
jrc         ;nothing to do
expr        ;expand M1 layer
jrn         ;test results
dsp,1,b,2   ;display M1 layer in grey
dspr,Y,1    ;display result in yellow
scpy
```

```
}
title=M1 layer width >= 2.5 um (Rule 4.2)
resolution=0.2
distance=2.5
operators=
{
map,1       ;map M1 layer
push,1      ;push it
jrc         ;nothing to do
notr        ;invert M1 layer
expr        ;expand M1 layer
jrn         ;test results
dsp,1,b,2
dspr,Y,1
scpy
}
title=M1 spacing to R2 >= 1 (Rule 4.3a)
resolution=0.2
distance=1
operators=
{
map,1 ;map M1
map,9 ;map R2
push,9
jrc
push,1
jrc
andrs
push,9
xorrs
push,1
exprs
jrn
dsp,1,b,2
dsp,9,w,2
dspr,y,1
scpy
}
title=M1 surround R2 >= 1.0 (Rule 4.3b)
resolution=0.2
distance=1.0
operators=
{
map,9 ;map R2
map,1 ;map M1
push,9   ;
jrc
push,1 ;
jrc
andrs
push,1 ;
notr
exprs
jrn
dsp,1,b,2 ;
```

```
dsp,9,w,2 ;
dspr,y,1
scpy
}
title=M1 surround I1B >= 1.5 (Rule 4.4)
resolution=0.2
distance=1.5
operators=
{
map,1 ;map M1
map,3 ;map I1B
push,1   ;
jrc
push,3 ;
jrc
andrs ;M1 + I1B
push,1 ;
notr ;
exprs
jrn
dsp,1,b,2 ;
dsp,3,c,1 ;
dspr,y,1
scpy
}
title=R2 layer spacing >= 2.0 um (Rule 5.1)
resolution=0.2
distance=2.0
operators=
{
map,9       ;map R2 layer
push,9      ;push it
jrc         ;nothing to do
expr        ;expand R2 layer
jrn         ;test results
dsp,9,w,2   ;display R2 layer in grey
dspr,Y,1    ;display result in yellow
scpy
}
title=R2 layer width >= 3.0 um (Rule 5.2)
resolution=0.2
distance=3.0
operators=
{
map,9       ;map R2 layer
push,9      ;push it
jrc         ;nothing to do
notr        ;invert R2 layer
expr        ;expand R2 layer
jrn         ;test results
dsp,9,w,2
dspr,Y,1
scpy
}
title=R2 surround I1B excp over M1 >= 1.5 (Rule 5.3excp)
```

```
resolution=0.2
distance=1.5
operators=
{
map,9 ;map R2
map,3 ;map I1B
map,1 ;map M1

push,1
push,3
andrs ;M1 + I1B

notr

push,3 ;I1B
xorrs
notr ;I1B except over M1

push,9
andrs

push,9
notr
exprs ;R2 + (I1B except over M1)

jrn
dsp,9,w,2
dsp,3,c,1
dspr,y,1
scpy
}
title=I1B layer spacing >= 2.0 um (Rule 6.1)
resolution=0.2
distance=2.0
operators=
{
map,3       ;map I1B layer
push,3      ;push it
jrc         ;nothing to do
expr        ;expand I1B layer
jrn         ;test results
dsp,3,c,1   ;display I1B layer in grey
dspr,Y,1    ;display result in yellow
scpy
}
title=I1B layer width >= 2.0 um (Rule 6.2)
resolution=0.1
distance=2.0
operators=
{
map,3       ;map I1B layer
push,3      ;push it
jrc         ;nothing to do
notr        ;invert I1B layer
expr        ;expand I1B layer
jrn         ;test results
dsp,3,c,2
dspr,Y,1
scpy
}
title=I1B surrounded by M2 >= 1.5 (Rule 6.3)
resolution=0.2
distance=1.5
operators=
{
map,3 ;map I1B
map,6 ;map M2
push,6
jrc
push,3
jrc
andrs
push,6
notr
exprs
jrn
dsp,6,m,2
dsp,3,c,1
dspr,y,1
scpy
}
title=M2 layer spacing >= 2.5 um (Rule 7.1)
resolution=0.2
distance=2.5
operators=
{
map,6       ;map M2 layer
push,6      ;push it
jrc         ;nothing to do
expr        ;expand M2 layer
jrn         ;test results
dsp,6,m,2   ;display M2 layer in grey
dspr,Y,1    ;display result in yellow
scpy
}
title=M2 layer width >= 2.0 um (Rule 7.2)
resolution=0.2
distance=2.0
operators=
{
map,6       ;map M2 layer
push,6      ;push it
jrc         ;nothing to do
notr        ;invert M2 layer
expr        ;expand M2 layer
jrn         ;test results
dsp,6,m,2
dspr,Y,1
scpy
}
```

```
title=M2 surround I2 >= 1.5 (Rule 7.3)
resolution=0.2
distance=1.5
operators=
{
map,6 ;map M2
map,8 ;map I2
push,6   ;
jrc
push,8 ;
jrc
andrs
push,6 ;
notr
exprs
jrn
dsp,6,m,2 ;
dsp,8,w,1 ;
dspr,y,1
scpy
}
title=I2 layer width >= 3.0 um (Rule 8.1)
resolution=0.2
distance=3.0
operators=
{
map,8       ;map I2 layer
push,8      ;push it
jrc         ;nothing to do
notr        ;invert I2 layer
expr        ;expand I2 layer
jrn         ;test results
dsp,8,w,1
dspr,Y,1
scpy
}
title=I2 surrounded by M3 >= 1.5 (Rule 8.2)
resolution=0.2
distance=1.5
operators=
{
map,8 ;map I2
map,10 ;map M3
push,8
jrc
push,10
jrc
andrs
push,10
notr
exprs
jrn
dsp,10,r,2
dsp,8,w,1
dspr,y,1
scpy
}
title=M3 layer spacing >= 2.5 um (Rule 9.1)
resolution=0.2
distance=2.5
operators=
{
map,10 ;map M3 layer
push,10 ;push it
jrc ;nothing to do
expr ;expand M3 layer
jrn  ;test results
dsp,10,r,2 ;display M3 layer in grey
dspr,Y,1 ;display result in yellow
scpy
}
title=M3 layer width >= 2.0 um (Rule 9.2)
resolution=0.2
distance=2.0
operators=
{
map,10       ;map M3 layer
push,10      ;push it
jrc         ;nothing to do
notr        ;invert M3 layer
expr        ;expand M3 layer
jrn         ;test results
dsp,10,r,2
dspr,Y,1
scpy
}
title=Minimum M3 contact size with R3 >= 3x3 um (Rule 9.3)
resolution=0.2
distance=3.0
operators=
{
map,10 ;map M3 layer
map,11 ;map R3
push,10 ;push it
jrc ;nothing to do
push,11
jrc
andrs
notr
expr
jrn
dsp,10,r,2
dsp,11,w,2
dspr,Y,1
scpy
}
title=R3 layer spacing >= 5 um (Rule 10.1)
resolution=0.2
distance=5
operators=
{
map,11 ;map R3 layer
push,11 ;push it
jrc ;nothing to do
expr ;expand R3 layer
jrn  ;test results
dsp,11,w,2 ;display R3 layer in grey
dspr,Y,1 ;display result in yellow
scpy
}
title=R3 layer width >= 3.0 um (Rule 10.2)
resolution=0.2
distance=3.0
operators=
{
map,11       ;map R3 layer
push,11      ;push it
jrc         ;nothing to do
notr        ;invert R3 layer
expr        ;expand R3 layer
jrn         ;test results
dsp,11,w,2
dspr,Y,1
scpy
}
```