Hardware Description Language Modelling and Synthesis of Superconducting Digital Circuits

Nicasio Maguu Muchuka

Dissertation presented for the Degree of Doctor of Philosophy in the Faculty of Engineering,
at Stellenbosch University

Supervisor:
Prof. Coenrad J. Fourie

March 2017

Declaration

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

Signature: N. M. Muchuka

Date: March 2017

Acknowledgements

I would like to thank all the people who gave me assistance of any kind during the period of my study.

Abstract

The energy demands and computational speed in high performance computers threatens the smooth transition from petascale to exascale systems. Miniaturization has been the art used in CMOS (complementary metal oxide semiconductor) for decades, to handle these energy and speed issues, however the technology is facing physical constraints. Superconducting circuit technology is a suitable candidate for beyond CMOS technologies, but faces challenges on its design tools and design automation. In this study, methods to model single flux quantum (SFQ) based circuit using hardware description languages (HDL) are implemented and thereafter a synthesis method for Rapid SFQ circuits is carried out. This enhances the ability to design at the logic and behavior level. To begin with, a survey is carried out to identify the available free software tools and a choice is made on a suitable set of tools for use in every stage of the superconducting electronics (SCE) design process. Secondly, each of the cells in rapid single flux quantum (RSFQ) and adiabatic quantum flux parametron (AQFP) cell libraries is simulated at circuit level to extract the circuit behavior. For the RSFQ cells, timing parameters are extracted whereby the propagation delay is expressed as a mathematical function dependent on bias voltage. Thirdly, to build HDL models, a proposed method is used for the AQFP logic circuits whereas, for the RSFQ circuits the existing methods are modified to handle cell delay in a mathematical expression. The resulting HDL models constitute the corresponding AQFP and RSFQ HDL cell libraries. Finally, a proposed synthesis method utilizing the RSFQ cell library is successfully implemented and tested using adders of different data widths (one bit to 64 bits). From the proposed toolchain, it is evident that superconducting circuits can be modelled and designed using free software tools. In addition, the mathematical function used in delay representation, allows the circuit to be simulated at logic level for any value of bias voltage within the circuit's bias margin. Moreover, the synthesis method implemented can allow logic circuit designers with little or no SCE knowledge to design SCE logic circuits.

Uittreksel

Die energie verbruik en verwerkingspoed van hoë-spoed rekenaars bedreig die veranderings gemak waarmee sisteme vanaf peta- na exa-skaal opgeskaal kan word. Die verkleinings tegnieke wat tot dus ver deur die Complimentary Metal Oxide Semiconductor (CMOS) gemeenskap gebruik word om die tegnologie af te skaal het die fisiese perke van die tegnologie bereik. Supergeleidende geïntegreerde stroombaan tegnologie is ‘n gepaste kandidaat vir die vervang van CMOS-tegnologieë, maar het nog talle probleme as dit kom by die ontwerp sagteware en outomatisering. In die studie word metodes ondersoek en beskryf om Single Flux Quantum (SFQ) stroombane te modelleer deur middel van Hardware Description Language (HDL); waarna ‘n sintese metode gebruik word om ‘n Rapid Single Flux Quantum (RSFQ) stroombaan te sinteseer. Dié verhoog die gehalte van ontwerp wat op ‘n logika en gedragsvlak gedoen kan word. ‘n Steekproef word eers gedoen om die relevante sagteware pakkette te identifiseer wat vrylik beskikbaar is, waarna ‘n besluit geneem word oor die stel sagteware wat gepas is vir elke stadium van die ontwerp proses. Tweedens word elke logiese sel in die RSFQ en Adiabatic Quantum Flux Parametron (AQFP) sel-biblioteek gesimuleer op stroombaan vlak op die stroombaan gedrag te bepaal. Vir die RSFQ selle word die tydsverloop parameters bepaal waarmee die voortplantings vertraging beskryf kan word as ‘n funksie van die voorspanning. Verder, om HDL modelle te bou, word ‘n metode voorgestel en gebruik vir AQFP stroombane. Vir RSFQ stroombane word klaar bestaande metodes aangepas om wiskundig die selvertraging in ag te neem. Die resulterende HDL modelle bevat dan beide die AQFP en RSFQ HDL selbiblioteke. Laastens word ‘n voorgestelde sintese metode, gerig op RSFQ selle, suksesvol geïmplementeer waarna dit getoets word met sommeerders van verskillende data wydtes (1 tot 64 bis). Die voorgestelde sagteware pakket beloof dus om dit moontlik te maak vir ontwerpers om supergeleidende geïntegreerde stroombane te kan ontwerp en modelleer met slegs vrylik beskikbare sagteware. Daarbenewens laat die wiskundige vergelykings wat gebruik word vir die verteenwoordiging van logikavertraaging toe dat ‘n stoombaan gesimuleer kan word op logika vlak vir enige voorspanning solank dit binne die perke van die stroombaan bly. Verder kan die sintese metode wat geïmplementeer word, logika ontwerpers toelaat om stroombane te ontwerp, al het hul geen kennis van supergeleidende stroombaan ontwerp nie.

List of publications and conference proceedings

Maguu N. Muchuka, Johannes A. Delport, and Coenrad J. Fourie, "Superconducting Digital Circuit Design with an Open Source and Freeware Tool Chain" IEEE Transactions on Applied Superconductivity, vol. 26, no. 8 December 2016.

N. Maguu and C. J. Fourie, "Modeling of AQFP logic gates with HDL using multivalued logic approach," in Proc. Int. Supercond. Electon. Conf., Nagoya, Japan, Jun. 2015, DS-P14..

Contents

List of Figures

Listings

Nomenclature

AQFP Adiabatic Quantum Flux Parametron

ASIC Application Specific Integrated Circuits

AST Abstract Syntax Tree

BCS Bardeen, Cooper and Schrieffer

CAT clock arrival time

CDFG Control and Data Flow Diagram

CMOS Complementary Metal Oxide Semiconductor

COSL Complementary output switching logic

EDA Electronic Design Automation

eSFQ energy-efficient Single-Flux-Quantum

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPL General Public License

HDL hardware description language

HLS High-Level Synthesis

IARPA Intelligence Advanced Research Projects Activity

IC integrated circuits

JJ Josephson Junction

JSIM Josephson Simulator

JTL josephson transmission line

LLVM Low-Level Virtual Machine

MVTL Modified variable threshold logic

PNG portable network graphics

PTL passive transmission line

RAT clock required arrival time

RMSE root mean square error

RQL Reciprocal Quantum Logic

RTL register-transfer level

RTLIL RTL intermediate language

SBU Stony Brook Univesity

SCE Superconducting Electronics

SFQ Single-Flux-Quantum

SSE summed square of residuals

SU Stellenbosch university

UHSC Ultra high-speed computing

VAUL VHDL Analyzer and Utility Library

VHDL VHSIC Hardware Description Language

VLSI Very Large Scale Integration

YNU Yokohama National University

Chapter 1

Introduction

1.1 Motivation

The demand for data processing and storage in the information communication technology is growing day by day. To keep up with the demand more powerful computing systems are necessary. A breakthrough to exascale systems is predicted to take place in the 2018 -2020 time frame [1]. Currently the top ranked high performance supercomputer, TaihuLight, is capable of performing 93 petaflops consuming 15.37 MW of power [2]. With an approximate running cost of US$ 1 million per year for a megawatt system, TaihuLight has an annual power budget of US$15.37 million. This implies that incremental evolution from petascale to exascale systems with a reasonable power budget is a challenge.

The current computing systems are based on silicon-Complementary Metal Oxide Semiconductor (CMOS) technology. To combat issues with regard to power consumption and increase speed in CMOS technology, miniaturization has been pushed for decades. However, according to [3] the technology may reach its physical limits in the next decade and it is unclear if this technology will provide sustainable support during the advancement of high performance computing and telecommunication systems. A search for new more effective technology as an alternative to CMOS technology has been under way for more than two decades. Superconducting Single-Flux-Quantum (SFQ) integrated circuit technology is an attractive candidate. It encapsulates both speed and energy efficiency which are crucial for the realization of future supercomputers.

Rapid SFQ (RSFQ) [4], energy-efficient SFQ (eSFQ) [5], Reciprocal Quantum Logic (RQL) [6], and Adiabatic Quantum Flux Parametron (AQFP) [7] are among the contemporary logic families in Superconducting niobium SFQ integrated circuits. Among these, RSFQ technology is currently well established, reliable and reproducible, with reported junctions on a chip up to 400,000 junctions [8] and a gate level switching speed of 770 GHz for a T-Flipflop [9] and a system clock speed of about 120 GHz. Circuit level design dominates in RSFQ circuits, with very little at logic level and behaviour level design. Moving to higher levels of design abstraction requires a standard design flow and reliable design automation

tools.

The aim with this research is to demonstrate that, based entirely on free software tools, SFQ circuits can be modelled at behaviour level using HDLs, and with slight modification of the semiconductor synthesis tools, behavioural synthesis can be performed. This is achieved by first formulating a design flow which is used as the guide in the formation for free software tool-chain for SFQ circuits. Second, cell characterization is carried out to obtain the timing parameters. Third, hardware description language (HDL) models for each cell are developed resulting to an HDL cell library. Fourth, a method to synthesize behavioural description of large combination logic circuits is demonstrated.

1.2 Superconductivity and Superconducting Circuits

1.2.1 Superconductivity

Perfect conductivity discovered by Kamerlingh Onnes (1911) and perfect diamagnetism discovered by Meissner and Ochsenfield [10], are the hallmarks of superconductivity. The phenomenological analysis done by Fritz and Heinz London, revealed that superconductivity is a quantum phenomenon manifesting itself in macroscale [11], and cannot be completely understood through classical concepts. J. Bardeen, L.N. Cooper, J.R. Schrieffer (BCS) theory (1957) revolutionized the understanding of superconductivity. Since then, diverse research in superconductivity has emerged.

1.2.2 Superconducting Circuits

Although the application of superconductivity can be traced back to 1936, (Grayson Smith et al.) when it was used in a galvanometer to detect 10^{-12} V., a glimpse of superconductor-based logic circuit was witnessed in 1956 with cryotron-based logic circuits [12, 13]. Although these circuits had a switching speed in the order of nanoseconds, this was surpassed by the semiconductor circuits.

The modern superconducting logic circuit is based on Josephson tunneling junction discovered in 1962 by B.D Josephson [14]. Josephson junction-based technologies have four common features when used in digital applications: fast switching speed (a few picoseconds/bit), low dynamic power consumption ($\sim 10^{-19}$ J/bit), ballistic intra-chip signal transmission with superconducting microstrip lines (lossless), and simpler fabrication technology than the semiconductor technology for equivalent physical design rules [4].

The two broad classes of superconducting logic families that have received much attention in research are, one, voltage-state logic circuits also known as latching logic family, and two, single flux quantum logic circuits. Complementary output switching logic (COSL) and Modified variable threshold logic (MVTL) [15] are voltage state logic families, while RSFQ, RQL, eSFQ and AQFP are single flux quantum logic families. The need to reset the gates in the latching logic has been its major drawback since this limits the operation speed of

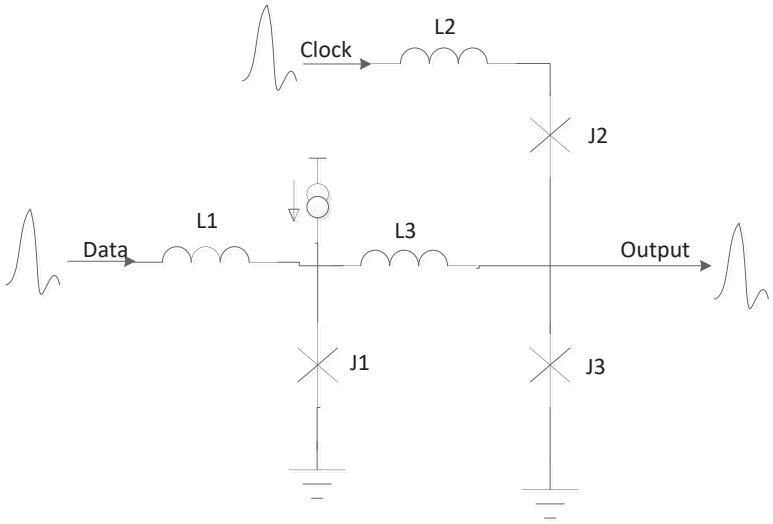
the circuits. Single flux quantum technology is a promising future technology with much research being done around the globe [16]. Various fabrication processes such as HYPRES , FLUXONICS and the AIST ADP2, STP2 & HSTP are in existence and logic circuits with a junction density higher than 400,000 per chip have been fabricated and tested for full operation [8].

Most of the SCE research today is on SFQ circuits and can further be sub divided into two categories. The first category is motivated by the need to acquire high speed circuits i.e logic circuits clocked at frequencies that semiconductor-based circuits cannot achieve. An example is the RSFQ family. The second category is driven by the need to reduce the power consumption of the circuits. Depending on the approach used in the power reduction, different logic families have emerged, such as RQL, eSFQ, and AQFP. While the SFQ circuits may differ in the way they are biased and how the logic is implemented at circuit level, the software tools used in their design process are the same. Thus, in the design automation perspective, challenges faced by the designers are the same. In this research, most of the work was done with RSFQ circuits and part of work was extended to AQFP circuits.

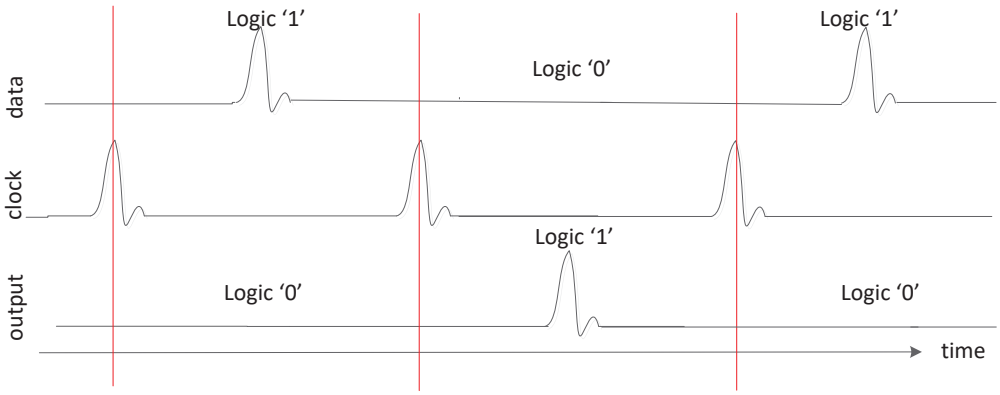
1.2.3 Rapid Single Flux Quantum Circuits (RSFQ)

RSFQ is a well established, reliable and reproducible, SFQ-based technology. It was developed in 1985 as a step to cross the limits of operation speed foreseen in latching logic Josephson Junctions digital circuits [17]. The devices are DC powered and utilize overdamped Josephson junctions. In RSFQ circuits, the logic information is presented in the form of single flux quantum pulses which are very short voltage pulses [4] unlike the semiconductor transistor logic technology and superconducting latching logic where the logic information is represented by a DC voltage level. The SFQ pulses are naturally generated by Josephson junctions, and can be reproduced, amplified, memorized and processed with Josephson junction-based circuits. SFQ pulses are transferred from one logic gate to another either with the help of an active Josephson transmission line (JTL) or a passive transmission line (PTL). Generally RSFQ is a positive-type logic technology, where a logic '1' is determined by the arrival of an SFQ data pulse, and logic '0' by the absence of an SFQ pulse. Working with logic '1' is easy, however, it can be hard to work with logic '0'. The absence of the quantum pulse must be taken with reference to some clock pulse. This requires most RSFQ logic gates to be clocked and especially the gates that must perform some form of inversion. The data is therefore sent to the inputs of the gates before the clock pulse arrives.

An illustration of the working of an RSFQ clocked cell can be seen in the Figure 1.1. In Figure 1.1a, is a typical RSFQ DFF schematic circuit [4] consisting of the input junction J1, a storage inductance L3 and decision pair junctions J2 and J3. The input SFQ pulse switches J1 transferring a quantum flux to the storage loop, consequently raising the amount of current through J3. If the clock pulse arrives when the quantum flux is stored, the quantum flux is transferred to the output by switching J3 and as a result clearing the storage. When



(a) Schematic diagram



(b) Logic representation with respect to clock pulses

Figure 1.1: RSFQ DFF cell and logic representation illustrated with timing diagram

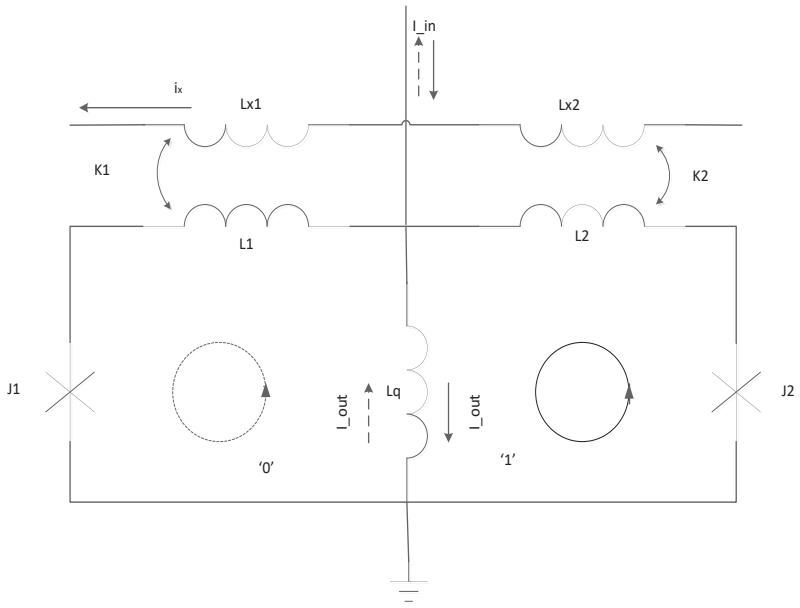


Figure 1.2: AQFP gate schematic diagram.

no quantum flux is stored, which implies that there was no data input since the previous clock, J3 does not switch, hence no output is observed. Figure 1.1b illustrates that a pulse appearing with the clock interval is treated as logic '1' and if no pulse appears it is a logic '0'.

1.2.4 Adiabatic Quantum Flux Parametron Circuits

AQFP is an ultra-low power superconducting logic family. The AQFP circuits are based on the AQFP gate [7, 18] shown in Figure 1.2. The simulations have shown that energy per logic operation can be 5.5 times smaller than that of RQL [19], and a switching energy of 10 zJ at 5 GHz has been reported [20].

The circuit operation begins by supplying a small input current (I_in) shown in Figure 1.2, then the excitation current (i_x) is raised. This results in the circuit developing a double well potential energy. The final state is either the SFQ to the left loop or to the right, representing state '0' and state '1' respectively. It is the direction of the input signal that influences the final state and thus the direction of the output current (I_out).

With regard to the inputs and the outputs of the circuit, the binary information is represented by negative current for logic '0' and positive current for logic '1'.

1.3 Research Goals

The goal with this research is to develop HDL models for SFQ circuits using Verilog HDL and VHDL which will build an HDL cell library and to develop a method to synthesize behavioural HDL design description using free CAD software tools.

In order to achieve this goal:

1. A design flow for SCE circuits is proposed and the available free software tools are analysed, which yield to a proposed free software toolchain.

2. Cell characterization is performed to acquire the circuit behaviour and timing parameters of the RSFQ cells available in Stellenbosch University (SU) cell library.

3. To develop a Verilog HDL and VHDL models of each cell used to create a HDL cell library.

4. To propose a method for behavioural synthesis of Verilog HDL design for combination logic circuit description to RSFQ gate level netlist.

1.4 Research Contributions

The review and analysis of free software tools for superconducting circuit resulted to a guideline for selecting free tools and a free CAD tool chain. This mitigates the challenge of selecting tools especially for novice researchers and budget-constrained research groups.

The research was entirely based on the free CAD tools, which for the first time demonstrated that the entire design process - as per current state of technology - can be accomplished using free tools, thereby making the design process accessible to any research group in the world.

The propagation delay for each cell was represented as mathematical function of bias voltage. These mathematical functions are implemented in HDLs as "functions" allowing the designer to simulate the circuit operation at an arbitrary bias voltage within the bias margin.

A multilevel logic approach to AQFP logic circuit modelling presented in this work was the first attempt towards modelling AQFP circuits using HDLs.

A method to achieve Verilog HDL behavioural synthesis of combination logic circuits is also presented in this work for the first time in the SCE literature.

1.5 Dissertation Outline

This document is organized into six chapters of which the introduction is the first. Chapters 2-5 contain the review of the related task, the task being addressed, the methodology, results and conclusion.

In chapter 2, the design flow and the free CAD tools useful in the superconducting technology are discussed. Chapter 3 contains the discussion on how the cells were characterized to obtain the timing parameters necessary to build an HDL model. The HDL model development procedure is presented in chapter 4. In chapter 5, a synthesis method for superconducting logic circuits is discussed. The 6^{th} chapter contains the overall conclusion and recommendation for future work.

Chapter 2

Free Software tools for SCE

2.1 Introduction

Electronic Design Automation(EDA) has been one of the driving forces for the advancement in semiconductor design industry. EDA tools allow the circuit designer to explore the design from its basic formulation such as the mathematical model to the lowest level of abstraction such as the physical representation the circuit. This brings better understanding of the circuit being designed and increases the probability of the circuit working after fabrication. It is evident that for superconducting electronic technology to grow, existence of capable EDA tools are paramount.

There are two major approaches used in the development of SCE CAD tools. The first is using the semiconductor technology CAD tools as they are, where possible, or by modifying them to cater for SCE concepts missing. This method is relatively easier and many successful attempts have been made. The second method involves creating new SCE tools . Despite the method being time-consuming and requiring an in-depth understanding of Josephson junction (JJ) dynamics and superconductivity in general, various tools have been developed to service the circuit simulation stage of SCE design flow.

The existing tools that can be used in SCE technology can be broadly classified as commercial tools and free software tools. The commercial tools are considered powerful, flexible, accurate, well documented, regularly updated, with customer support among other attractive qualities. On the other hand, most of these tools are expensive and licensed on per seat basis. The cost of licenses and accessibility of such tools keeps them out of reach for budget-constrained research groups/organizations and individual researchers, especially in the developing nations. Moreover, some organizations buy only a few licenses which limits the number of the developers in the team who can make use of the tool at a time. Furthermore, the type of license acquired in some tools may hinder the users from exploring all the artifacts of these tools.

The alternative to commercial tools is the free software tools. The ‘free’ in the ‘free software tools’ is commonly mistaken for ‘no price’, however, the emphases here is on

the user's freedom [21]. As opposed to having a license that restricts the users, they have a license that gives freedom to the user. These tools are generally available on digital repositories such as http://opencircuitdesign.com/, https://sourceforge.net/, and http://wrcad.com/freestuff.html. The interested user downloads these tools without any restriction imposed by the developer. Thus the cost of ownership is minimal. The free software tools are further classified into the open source tools and the freeware tools.

The open source tools are based on open software development methodology. These tools give the designer the freedom to use, copy, study, make changes, improve and distribute. Many users face a challenge when they have to compile the open source tools in different platforms especially if they do not have expertise in software development tools. This has been mitigated by the use of build-automation software that can automatically build an executable program for the specified target platform.

Freeware provides the user with the freedom to use and share the tool but the source code is not provided. These should not be confused with shareware, which limits the user to a few artifacts of the tool and the ability to extend these by buying a license.

The main drawbacks of free software are lack of good documentation, limited features, and lack of regular updates. Despite all that, the open source approach to CAD tools development has a good future, as demonstrated by well reputed EDA companies such as Synopsys participating in OpenMAST [22]. In this chapter, the open source tools that are useful in SCE circuit design flow are described. The tools are described with reference to the design flow proposed herein. A set of free software tools for small-scale design is first discussed followed by the medium and large-scale free software tools. A free software tool chain is then formulated. The tools used in the development of the HDL cell library and the demonstration of a synthesis method are part of the presented free software toolchain.

2.2 SCE design flow

With no existing standard design flow in SCE digital circuits, the designers formulate one depending on the steps they consider to be important in the design process. A SCE design flow for digital circuits is proposed that is similar to [23]and [24]. This design flow is developed such that in each major step an idea of the tools required by the designer is given. Based on the circuit complexity, the design flows are formulated and classified as small-scale circuit design flow and medium & large-scale circuits design flow. A clear borderline on the actual size or complexity of the two categories is an approximation and not a standard. In this work, a small scale circuit is considered as a circuit that implements a basic logic primitive operation such as AND, OR, Bit storage etc., otherwise it is considered a medium-scale or a large-scale circuit.

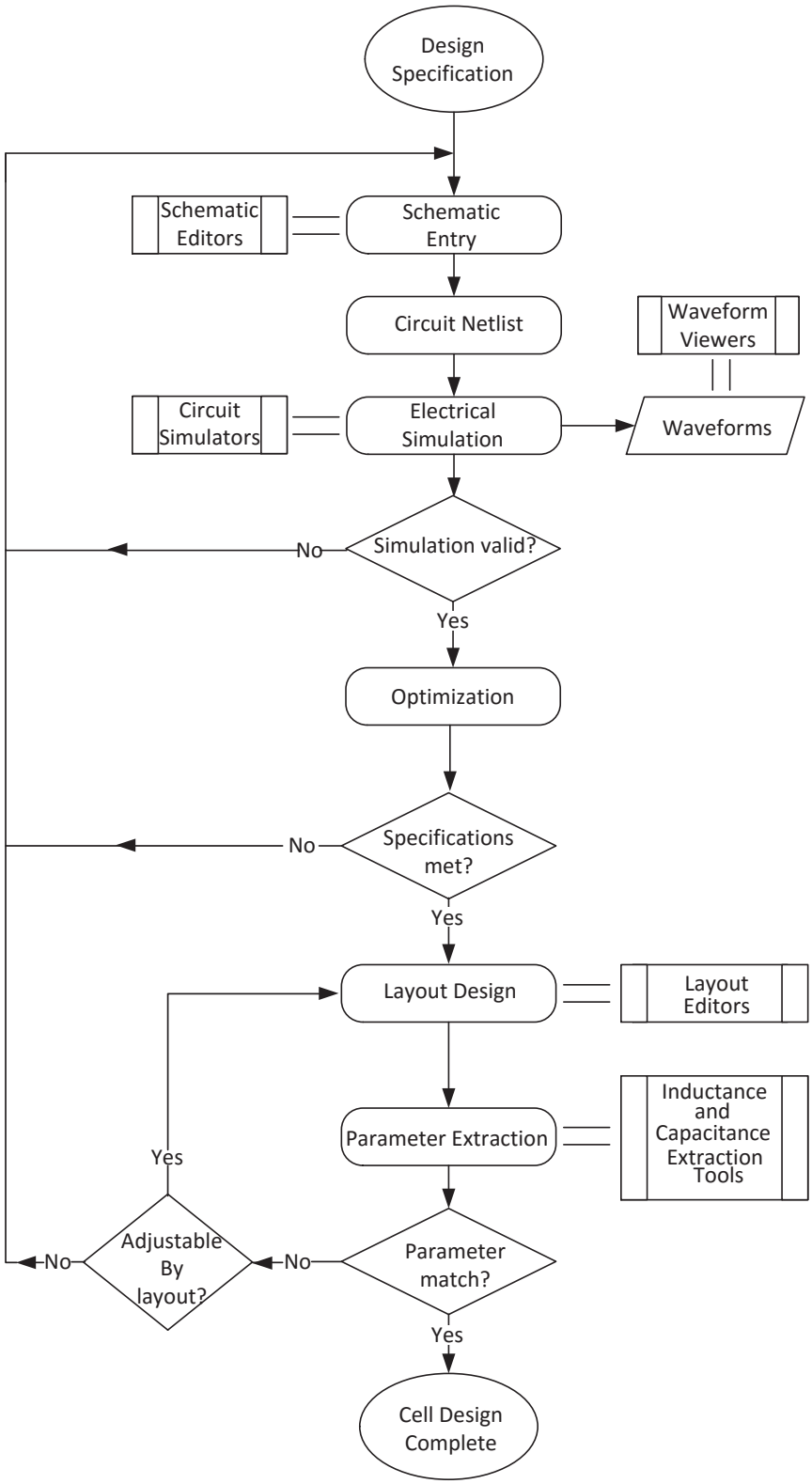


Figure 2.1: SCE small-scale design flow

2.2.1 Small-scale circuit design flow

The proposed small-scale circuit design flow is shown in Figure. 2.1. The designer ought to select the target logic family and fabrication process prior to commencement of the design process. The two will then aid the designer with the design rules they impose.

To design a small-scale circuit, the designer initially obtains the design specification. This could be in the human language, a mathematical model such as a Boolean expression, state table/diagram or a truth table. The aforementioned specifications are analysed and then a logical representation of the design is developed using basic circuit components which yield a suitable topology for use in circuit parameters approximation. The initial parameter values are computed, then the design is transferred to a digital canvas with the help of schematic capture tools. This is an entry to design simulation/automation process as well as a form of design documentation. The circuit schematic is then converted to a circuit netlist compatible with the available circuit simulator. The conversion can be done using a netlist generator which automatically converts the schematic to its corresponding netlist, or it can be hand coded using schematic as the guideline. The designer is required to prepare a testbench that generates the necessary test pattern for the designed circuit. To verify the operation of the circuit, electrical simulation is performed with the help of the circuit level simulator tool. The simulator runs the circuit netlist and its testbench. The test patterns are fed to the circuit and the output is evaluated at every time step. An output file is created whose content depend on the simulator directives specified in the testbench. To visualize the output, analog wave viewers such as Gwave [25] are used, or the output can be plotted using plotting and numerically based tools such as GNUplot [26], Octave [27], or SciLab [28]. If the expected operation is not observed, the design is modified and simulated iteratively until the desired operation is achieved.

During the circuit fabrication the design values are subject to change due to parameter spread. Moreover, a variation in bias current required for optimal operational margins may occur. To mitigate the effects of these shortcomings on the circuit operation after fabrication, a design optimization is performed with the help of the optimization tools. The obtained optimal circuit parameters are used to verify that the required specification are met. If optimal values that fulfill the requirements cannot be obtained, then the design is modified at schematic entry or netlist stage. Once the design optimal values are obtained, a geometrical representation of the circuit is designed. The layout editing tools are used at this stage. The design rules dictated by the target fabrication process guide the designer while laying out the circuit. The circuit layout file is then used as an input to parameter extraction tools, which computes the circuit parameters from their geometrical representation in the layout. The values extracted are compared with circuit optimal values and if they differ beyond the tolerable range, the layout is modified to correct the difference. In cases where the layout cannot be changed further without violating the design rules or design constraints, the design process is restarted. When the layout design is finalized, it is taped out as a test

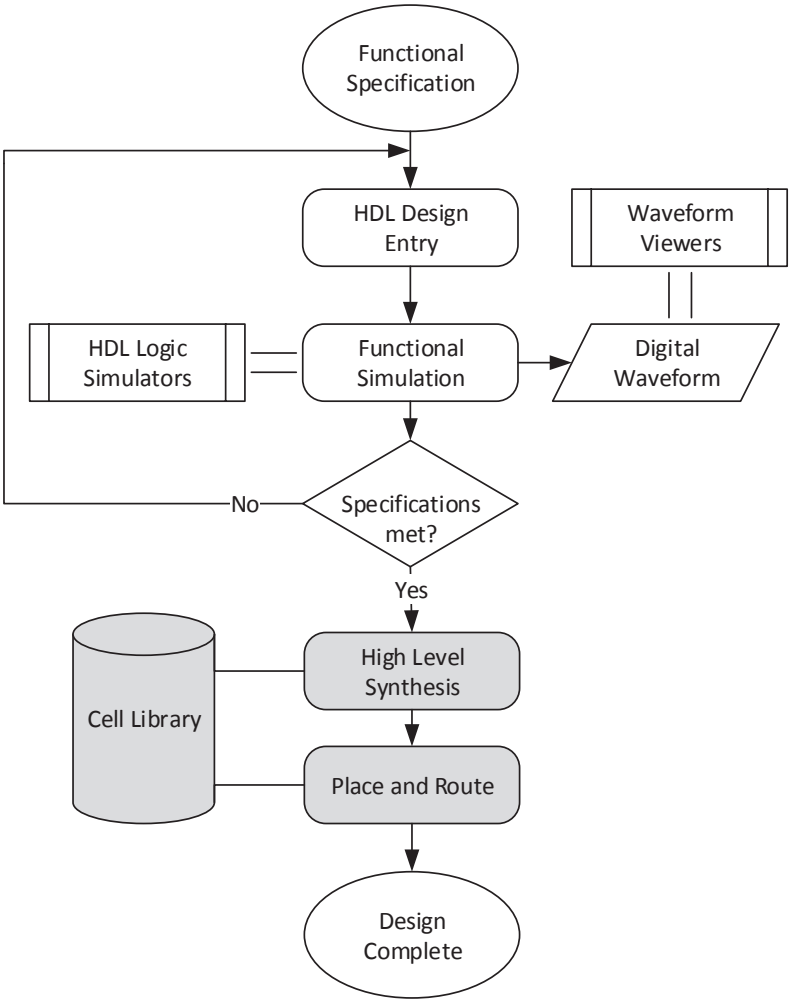


Figure 2.2: SCE Medium and Large -scale design flow. The grey components do not exist for SCE circuits yet

cell. If the test is successful the cell is added to a reliable cell library.

2.2.2 Medium- and Large-scale circuit design flow

The design of SCE medium-scale and large-scale circuits uses a semi-custom methodology based on a collection of reliable small-scale circuits called cell library. The method has been adopted from semiconductor technology, however in SCE it is not well automated. Using this method the designer can build a large circuit by interconnecting the standard cells from the cell library. Custom routing circuits are used when necessary. This method has been used successfully in [29] and [30].

A simplified design flow for medium-scale and large-scale is presented in Figure 2.2. The design specifications are analyzed, then the design captured at a high level abstraction

using hardware description languages (HDL) such as Verilog HDL or VHDL. The design description either behavioural or/and structural is simulated with a logic simulator. A testbench is used to feed the design with the necessary bit patterns capable of covering most of the possible faults. A verification section may also be included in the testbench to perform auto-verification. The simulation is then viewed from a digital waveform viewer such as GTKwave [31]. The output of the simulation is checked to ensure that the design meets the desired specification. In cases where the specifications are not met, the design description is modified at HDL design entry stage to capture the required behaviour of the circuit. The description is then simulated again. This process of modifying the design description and simulating it is repeated until the desired output is obtained.

The design stages that follow the function simulation step, are presented in the grey shaded background in Figure 2.2, to indicate that there are no open-source tools either developed or adopted from semiconductor technology to handle the design stage. In the next step, design synthesis, register transfer level or logic representation of the circuit is generated by a synthesis tool from the HDL description. Currently, the design synthesis stage is a weak link in general superconducting circuit technology. The synthesis tools depend on cell libraries to perform their tasks. While there are several cell libraries developed by different research groups for their own use, a complete open-source cell library has not yet been reported. Design place and route is another stage where the technology is still waiting for automation tools. Thus, when the circuit simulates correctly, the rest of the work is mostly completed by hand, especially when relying on open-source tools.

2.3 Selected free tools useful in SCE design

The free and open source community has free tools corresponding to each stage of the design flow except for the instances noted in the previous section 2.2.2. Most of these tools are calibrated from semiconductor technology tools, with just a few that have been developed exclusively for SCE application. In this section, free tools used in this work and the related useful tools are discussed as per the design stages discussed in SCE design flow in section 2.2.

2.3.1 Schematic Editors

Schematic capture tools are used in the design entry stage to lay out the circuit schematic on a digital canvas. These tools allow an easy transformation of the circuit topology when needed while providing the design visualization and providing design documentation. Exportation of the schematic diagram to a publication-quality graphics format such as portable network graphics (PNG) and enhanced postscript (eps) can be done by some tools.

Drawing a schematic using the tools can be quite easy, however most designers especially those new in the technology find it challenging when they have to deal with the non-standard

devices such as Josephson Junction (JJ). Most free schematic editors don't have a built-in JJ. In order to use these tools, a JJ model symbol needs to be created and its attributes defined. This is time-consuming and challenging for novice designers especially those who are not conversant with the tool. For this reason, schematic tools with built-in JJ are preferred.

Commonly used schematic editors in the SCE technology are modified from schematic editors used in the semiconductor technology. There are many tools in existence under this category, however only the those tried out at SU are pointed out. The schematic editors analyzed in order to select the one to use when developing the circuit level testbenches include gSchem [32], Sced, Fritzing [33], Eeschema [34], KTechLab [35], and Xcircuit [36].

gSchem is the schematic capture tool developed in the gEDA project, and it is part of gEDA suite available under the general public license (GPL). The gEDA suite is available in binary or source code for POSIX systems. Some of attractive features of gSchem are user-friendliness, a built-in JJ model in its technology library, and allows the user to enhance its functions using Guile Scheme scripting language or by extending the source code.

Sced is the schematic capture used with JSpice3. It is keyboard-driven and offers the basic features of a schematic editor which make it difficult to use. Despite the fact that it has a JJ model built-in, it is useful for circuits containing a few JJs making it less attractive to many users.

2.3.2 Netlist Generators

The conventional method of generating a netlist from a schematic circuit by hand coding is not only time consuming but also extremely difficult as the number of circuit elements increases. Therefore Netlist generator tools are used to convert the schematic diagram of a circuit to its equivalent netlist. gEDA suite, KTechlab, and Xcircuit have netlist generators. Among them, only gNetlist of gEDA has been used to generate a netlist for superconducting circuits. The target simulator is an important factor to consider when generating the netlist so that the resulting netlist is compatible with the simulator. In some situations the SPICE database file is modified to ensure compatibility. The disadvantage of the auto generated netlist is that it's not as compact as the hand coded one. Moreover, the circuit nodes are not ordered, which makes it difficult to debug at the time of simulation error.

2.3.3 Circuit simulators

The initial step towards superconducting circuit simulation was achieved through including JJ models in the SPICE program.

JSPICE [37] which consists of SPICE2G5 [38] with a JJ model was the first simulator. The JJ model gives the program the ability to simulate SCE circuits. The simulator takes a considerably longer time for a SCE circuit than it would take for a classical circuit of comparable complexity. This is because tracking the Josephson Oscillations require a very small time step.

The limitations of JSPICE motivated the development of JSIM, the Josephson SIMulator [39]. JSIM was developed purposely to handle superconducting circuits' simulation. The developers emphasized the reduction of the computational resources per time step. This resulted in JSIM being a fast simulator for SCE circuits compared to other simulators, and is currently the most commonly used circuit simulator in the SCE community. JSIM is purely a command line program. Its simulation outputs are dumped into a data file according to the simulator directives given in the circuit model. The data file is then used to plot the output for visualization using plotting tools like GNUplot or numerically-based tools such as Octave or Scilab.

PSCAN (personal superconductor circuit Analyzer) [40] is a SCE circuit simulator designed initially with special emphasis on the RSFQ circuit family. It has a built-in circuit verification mechanism, where the user specifies the correct circuit operation using hSFQHDL (hierarchical single flux quantum hardware description language), then the simulation output is compared with the user specifications for verification. PSCAN performs the simulations fast enough and it has been used a lot by various research groups, however its use is currently declining due to lack of support and difficult in accessing it.

Csim [41] is an open source SCE simulator that uses a scripting language that resembles C programming language. With this popular syntax it can be quite attractive to designers but Csim take a reasonably longer time to simulate the circuits compared to its counterparts.

Other semiconductor technology circuit simulators are currently not a choice for many designers in SCE, one of the reasons being that most SPICE derivatives under free software have no JJ model built-in. However the JJ model sub-circuit can be added by the user as a sub circuit. Second, with the JJ model sub-circuit added, the time taken to simulate the circuit can be discouraging especially when high accuracy is needed. Nevertheless, for novice researchers with experience in them, this could be a good starting point. In this work LTSpice [42] and ngspice were successfully used to simulate a JTL and a DFF. It was noted that Ngspice failed to simulate when the time step is less than 5 ps.

2.3.4 Optimization tools

The aim of any SCE circuit designer is to obtain a circuit that is least affected by the change in circuit parameters during fabrication and fluctuations in bias current. In addition, the number of working chips per wafer should be as high as possible. Circuit optimization tools improve the chances that these objectives will be met. The tools under this category have all been developed with SCE circuit in mind. The popular open source optimization tools are MALT [43] and COWBOY [44]. MALT is a yield optimization tool based on the inscribed hyperspheres algorithm. COWBOY was developed as an integral part of PSCAN. Other than using already developed tools many designers implement the optimization algorithms such those described in [44, 45, 46, 47]. This requires software development tools such compilers which are abundantly available in the open source community.

2.3.5 Circuit Layout and Verification

To generate the geometrical representation of the circuit elements, a layout editor is used. Most of the layout editors used in SCE have been calibrated from semiconductor technology tools. A number of free software tool projects address the issues of integrated circuit layout for semiconductor technology. The most popular are Magic [48], Graal , Toped [49] and LASI [50]. The use of Magic and LASI in designing SCE circuit layouts is demonstrated in [51] and [52] respectively. The main requirement for using these tools is the ability to change the default technology file in the layout editor. Therefore, a user-defined technology file is created defining all the layers that match the target fabrication process. This file is then set as the active layer definition file. The layout tools which allow change of the active layer definition file have the potential of being used in SCE.

Currently, neither design rule check (DRC) nor layout vs schematic (LVS) are available in these tools for superconducting technology. The only method applicable for layout verification is manually comparing the design optimal parameters and the values extracted from the layout. Inductance extraction tools are used to compute the circuit element values from the layout structures. The tools that are available for free are LMETER [53], FastHenry [54] and InductEx [55] model-size limited version.

2.3.6 HDL logic simulators

As the circuit complexity of SCE increases, designing at circuit level becomes increasingly onerous. HDLs such as Verilog HDL and VHDL can be used to design the circuits at a higher level of abstraction by either behavioural or/and structural description. The design can then be simulated using the HDL logic simulators. Use of these languages has been demonstrated in [56, 57, 58]. The authors utilized commercial simulators, the cost and availability of which may discourage many who want to reproduce their study. In this work, open source simulators were for the first time used to simulate HDL models for AQFP [59] and RSFQ logic circuits.

In this section the five HDL simulators evaluated are presented and the two simulators used in the implementation of HDL models are discussed.

It was noted that virtually any HDL logic simulator is capable of supporting the simulation of HDL models for SCE circuits. However, the choice of description style used is crucial. This is because the HDL description style depends on the HDL standard and most of simulators do not fully cover the recent language standards, as depicted in the table 2.1. While this limits the flexibility of the designer, it should be noted that the new modelling features included in the recent HDL standards can be achieved using the previous standards although not as succinctly. Moreover, the language standard supported by the simulator cannot be an obstacle for an accurate HDL model of a SCE logic circuit.

Three VHDL simulators and two Verilog HDL simulators were assessed for use in the simulation of the SCE circuit models.

Table 2.1: Open source HDL simulators evaluated

Simulator	Language	Standard supported	
		Full	Partial
GHDL	VHDL	2002	2008
FreeHDL	VHDL	93	-
NVC	VHDL	93	2008
iVerilog	Verilog	2001	2005
Verilator	Verilog	2001	2005

GHDL is an open source VHDL simulator that currently supports the VHDL IEEE standard 1076-87, 1076-93, 1076-2002, in full and 1076-2008 partially. GHDL can be downloaded as source code or in binary form. Currently, the available binary distributions are for Debian Linux, Mac OS X, and Windows. It uses LLVM (low-level virtual machine) [60], or GCC, or the built-in code generator to convert the VHDL design to machine code without using an intermediate C file. This makes this simulator faster than any simulator that involves the intermediate language. The machine code is then executed by the native machine. The output generated is then written in **.vcd** or **.fst** file, which can be opened with digital waveform viewers.

NVC is a GPLv3 VHDL compiler which is IEEE 1076-93 compliant, and capable of supporting a small subset of IEEE 1076-2008. NVC runs LLVM code generator to produce a bit code file which runs on the host machine. It also supports a subset of VHPI (VHDL procedural interfaces) allowing access to signals and events at runtime. The results of the simulation are then stored in a file that can be opened by a digital wave viewer for visualization.

FreeHDL is a VHDL simulator which currently supports IEEE 1076-93 standard. It utilizes VAUL (VHDL Analyzer and Utility Library) front-end to generate an intermediate C++ code from a VHDL design. The C++ code is then compiled to generate object code which is linked and executed on the host computer. To start the simulation, the designer uses a script **gvhdl** (part of the simulator) which automates the simulator's internal steps. The output of the simulation is stored in a **.vcd** file.

Varilator is a Verilog HDL compiler and event-based simulator. It compiles synthesizable Verilog HDL code into a well optimized C++ or System C or SystemPerl. The testbenches are usually written in C++ mainly because testbenches contain non-synthesizable code. The current version of Verilator 3.888 is capable of parsing SystemVerilog.

Icarus Verilog, commonly known as iVerilog, is a Verilog HDL simulator and also a synthesis tool, the current version of which provides full coverage of IEEE 1364-2001 and partially supports IEEE 1364-2005 standards. The tool can be utilized in two modes. One, as a simulator and second, as a synthesis tool. When used as a simulator, the Verilog HDL code is compiled into an intermediate form known as vvp assembly, which is then executed using the vvp command. The simulation output is either written to a .vcd file or to the

terminal depending on the directives in the testbench. In the synthesis mode, the tool compiles the Verilog HDL code into a netlist of the preferred format.

2.3.7 Synthesis tools

Icarus Verilog used as synthesis tool converts the design into one of supported netlist formats specified by the user. This can result in EDIF, XNF or VHDL among others. With its extensible code generator, this tool can be modified to handle the synthesis of superconducting circuits.

Yosys provides a framework for synthesizing Verilog HDL. It has the ability to synthesize Verilog HDL design description to various output formats popular in semiconductor technology such as BLIF, EDIF, BTOR, among others. The tool can also map the design to the ASCI standard cell library. Basically, Yosys achieves behavioural, RTL, and Logic synthesis. One of the most attractive features of Yosys to SCE tool developers is its ability to combine the existing algorithms with user-defined algorithms using synthesis script or by extending its base class.

2.3.8 SCE free toolchain

To accomplish the main objective of this work – SFQ HDL cell libraries and behaviour synthesis – a set of tools was selected from the analysed free tools discussed in previous sections. This subset of free tools was then used as a free SCE toolchain.

The schematic of the testbenches used to analyse the cell behaviour for characterization purposes were captured using the gEDA's gSchem. Gnetlist, also part of the gEDA was used to generate a circuit netlist from the schematic. The resulting netlist was JSIM compliant, therefore a JSIM circuit simulator was utilized in the electrical simulation stage. The circuit simulation waveforms were plotted using GNUplot. The selection of gSchem, gnetlist and JSIM was based on the capabilities, availability, documentation of the tool and also the support in the SU research group.

Although, this work did not involve the circuit level design of the cells, for cases where a cell exhibited a behaviour different from the expected, the circuit design team modified the cell and utilized in-house optimization tool to optimize the modified circuit and LASI for the layout design.

FreeHDL simulator and Icarus Verilog simulator were used to simulate the VHDL models and Verilog HDL models respectively. These simulators were selected based on the capabilities and language standard supported. Although, GHDL and NVC supports recent standard features, the research aimed to demonstrate that the HDL standard should not be an obstacle to modelling SCE digital circuits with HDLs. The synthesis of SCE large scale circuits was demonstrated only with Verilog HDL by modifying the Yosys synthesis tool.

Table 2.2: A summary of the free tools used in each design step of SCE circuit in this work

Category	Tools
Schematic editor	gEDA's gschem
Netlist generator	gEDA's gnetlist
Circuit simulator	JSIM
Analog waveform viewer	GNUplot
Optimization	custom using C++
Layout Editor	LASI
Parameter Extraction	InductEx (free version)
HDL description entry	Notepad ++
HDL simulators	Iverilog and FreeHDL
Digital waveform viewer	GTKwave
Synthesis	Modified Yosys

2.4 Conclusion

The prime question addressed here is whether there are free CAD tools available that can be adopted or adapted for SCE circuit design. Design flows for small scale and medium & large scale SCE circuits were formulated to bring out all the necessary steps in the SCE design process that require a form of automation. These design flows are useful as a guide in the design process and development of tool chains. From the review performed on the available free CAD tools in each step of the design flows, a free toolchain was formulated. Using this toolchain, a small scale SCE circuit can be designed using free tools only. However, for large circuits synthesis and placement & routing tools need to be developed.

The ambitious IARPA SuperTools project, announced in 2016, foresees the development of a complete SCE design toolchain based on open source tools, so the work presented here is a timely precursor to SuperTools.

Chapter 3

Cell Characterization

3.1 Introduction

The success of digital VLSI circuits depends on a well-characterized standard cell library as the backend for the synthesis tools. To build powerful automated synthesis tools for SCE VLSI design flow, a cell library containing cells which are characterized such that a model based on them closely describes the actual circuit is crucial. The two methods that can be used to characterize a cell are analogue simulation-based method and fabricated circuit-based. The first method relies on electrical simulations and circuit simulator such as JSIM is used to run simulations of the circuit netlist, while the objective parameters are varied in every simulation run. The second method involves measuring the actual waveforms on a fabricated cell using an oscilloscope. This method is neither efficient nor practical for a large cell library. The cell characterization process described here is based on analogue simulation. To speed up the process, a set of programs written in C++ and scripts that were written in Octave/Matlab have been developed.

3.2 Cell Characterization method

The main characteristics of the cells extracted in this work are pulse propagation timings to facilitate in the building of a high accuracy HDL model. The timing parameters extracted are propagation delay and pulse separation constraints of each cell in the SU RSFQ cell library. The testbed shown in the Figure 3.1 is used to logically organize the cell characterization process. It is used in both asynchronous and synchronous cells. The arrangement takes care of cells with one or more inputs and outputs. In the case of the synchronous cells, the clock is treated as one of the inputs while other input lines are for data. In the SU cell library, the only cell with more than one output is a signal splitter. Each input is generated by feeding piecewise linear triangular dc pulse to a DC-SFQ cell to obtain an SFQ pulse. Two JTLs are used in both inputs and outputs to the cell under analysis to ensure that the SFQ pulse is near perfect while isolating the two cells from influencing each other's functionality, either

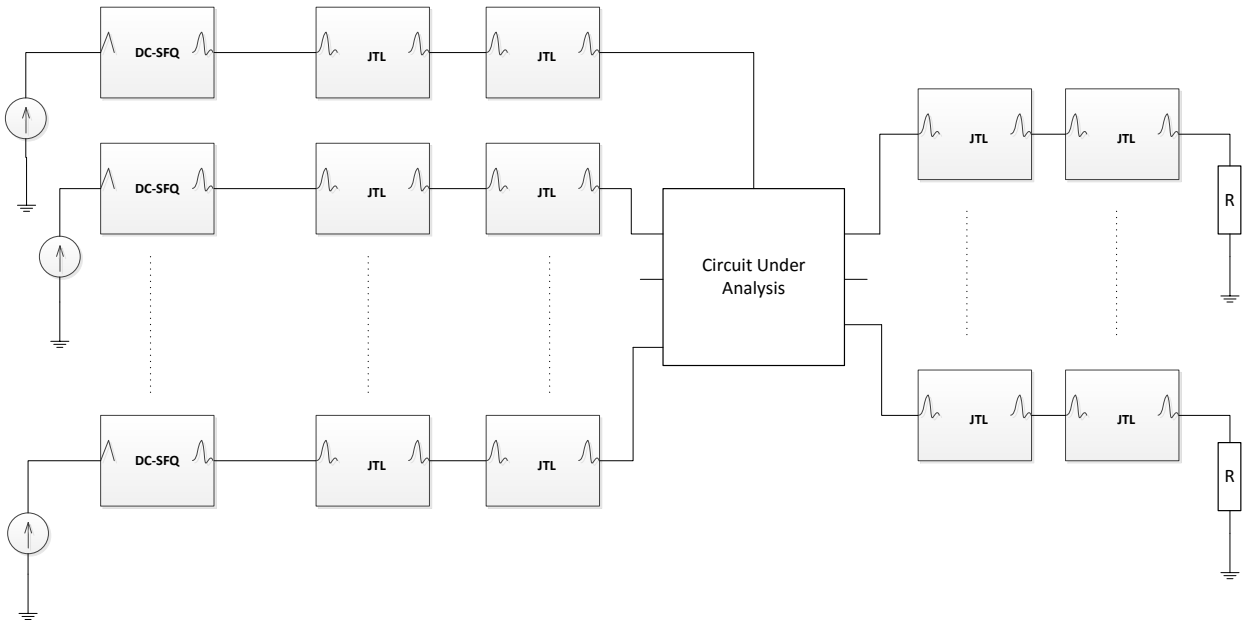


Figure 3.1: General testbed for RSFQ circuits

by the bias current or switching effect. The last JTL in the outputs is terminated with a resistor equivalent to the JTL's output impedance.

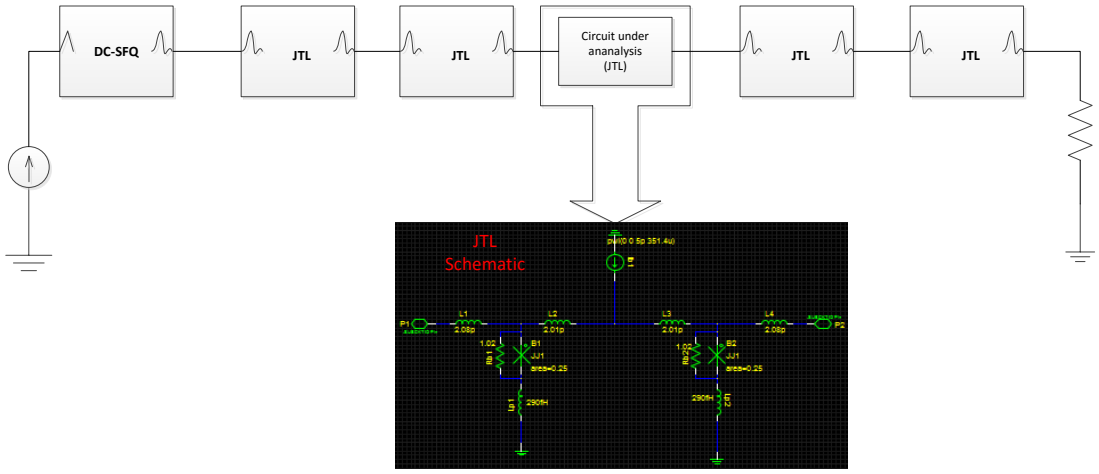


Figure 3.2: JTL analysis setup

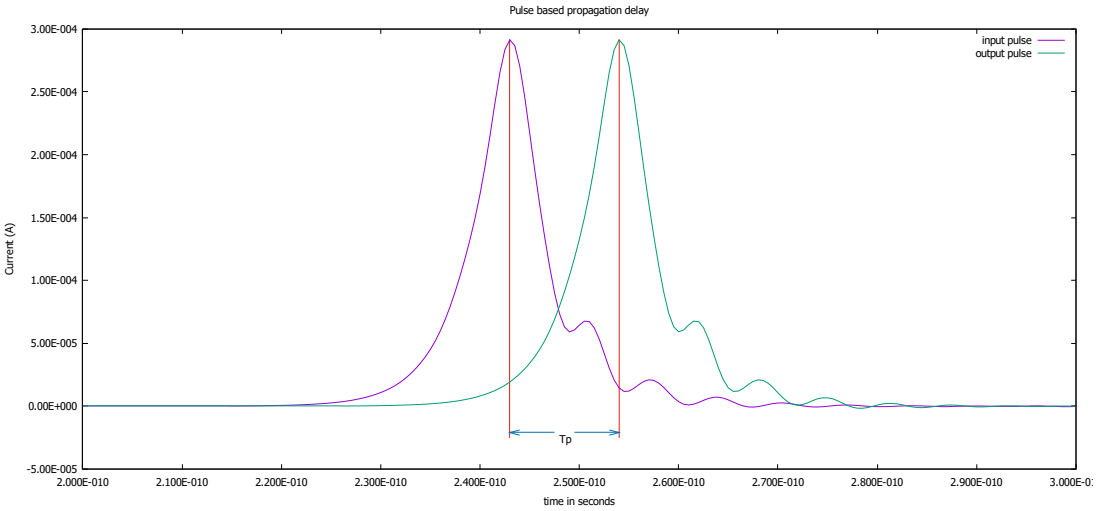


Figure 3.3: Time delay evaluation from pulse position; Tp represents the time delay

3.2.1 Propagation delay

Propagation delay is generally defined as the time taken by a circuit to generate an output signal as a response to an input signal. In asynchronous circuits the delay can be evaluated as a single entity, however, in synchronous circuits, the delay has two parts, the first being the minimum time separation between data and clock, and the second is the time taken to obtain the output after the clock pulse arrives.

3.2.1.1 Propagation delay in asynchronous circuits

To extract delay values, the test bench is set up as illustrated in figure 3.2, where JTL is the cell being analysed. The circuit is then simulated with the JSIM to obtain the waveforms such as the ones shown in figure 3.3. From the waveforms the peak position (in time) t_1 for input pulse and t_2 for output pulse are read. The delay is then computed by taking the difference between t_2 and t_1 i.e. $T_p = t_2 - t_1$. In the cases where a pulse is distorted, the highest peak position is taken for the delay calculation. Unlike the semiconductor logic gate where propagation delay is the only key timing parameter for unclocked circuits, unclocked cells restrict the input pulse interval in the same or related inputs [61]. In the case of confluence buffer, pulse separation with respect to each other was restricted such that the output pulses are distinct.

3.2.1.2 Propagation delay in synchronous circuits

Synchronous circuits also referred to as clocked circuits are characterized by the ability to store data for use when desired. In addition to data inputs they have a clock input that triggers the circuit to generate the outputs. Clocked cells in semiconductor technology are characterized mainly by setup time, hold time and propagation delay. The setup time is defined as the minimum amount of time that the inputs must remain stable before the clock arrives, while the hold time is defined as the minimum time the inputs must remain stable after the clock arrives [62]. And generally the propagation delay is taken to be the time taken to obtain the output after the cell is clocked. Due to the fact that the input signals are stored before the clock pulse in the RSFQ clocked cells unlike the case of semiconductor circuits, the setup time definition used in semiconductor circuits may not be used in superconducting RSFQ circuits without modification. Kransniewski [63] and Gaj et al [61] defined setup time (t_{su}) for RSFQ cells as the minimum time separation between the data pulses and the clock pulse. In this work, this is interpreted intuitively as the time required for the input quantum flux to propagate and get stored in the circuit's storage loop (t_{sl}) minus the time required by the clock pulse to reach the decision junction of the circuit (t_{cj}). $t_{su} = t_{sl} - t_{cj}$. The authors in [63] and [61] further defined hold time as the minimum allowable separation between the clock pulse and the following input pulse. In our analysis, this is interpreted as the time required to transfer the stored a quantum flux from the storage loop. The setup time and

hold time are crucial for proper operation of clocked cells and consequently, the validity of their HDL models. However, to visualize the circuit logic operation clock pulse to output pulse time i.e. the propagation delay, is used in this work. A DFF testbed is shown in figure 3.4 and its circuit's simulation depicted in figure 3.5. T_{d2c} is data pulse to clock pulse time which should be greater than the setup time of the DFF and T_{c2o} (clock pulse to output pulse) is the propagation delay. These two parameters are calculated using the pulses' peak position as shown in figure 3.5.

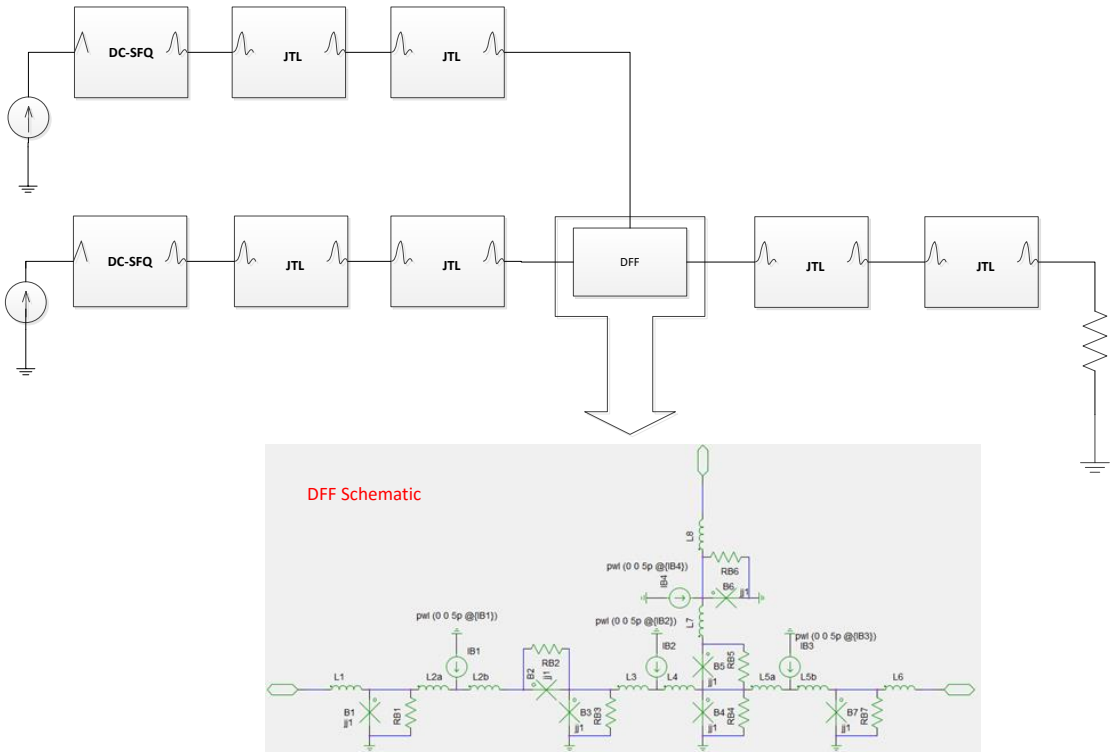


Figure 3.4: DFF test bed

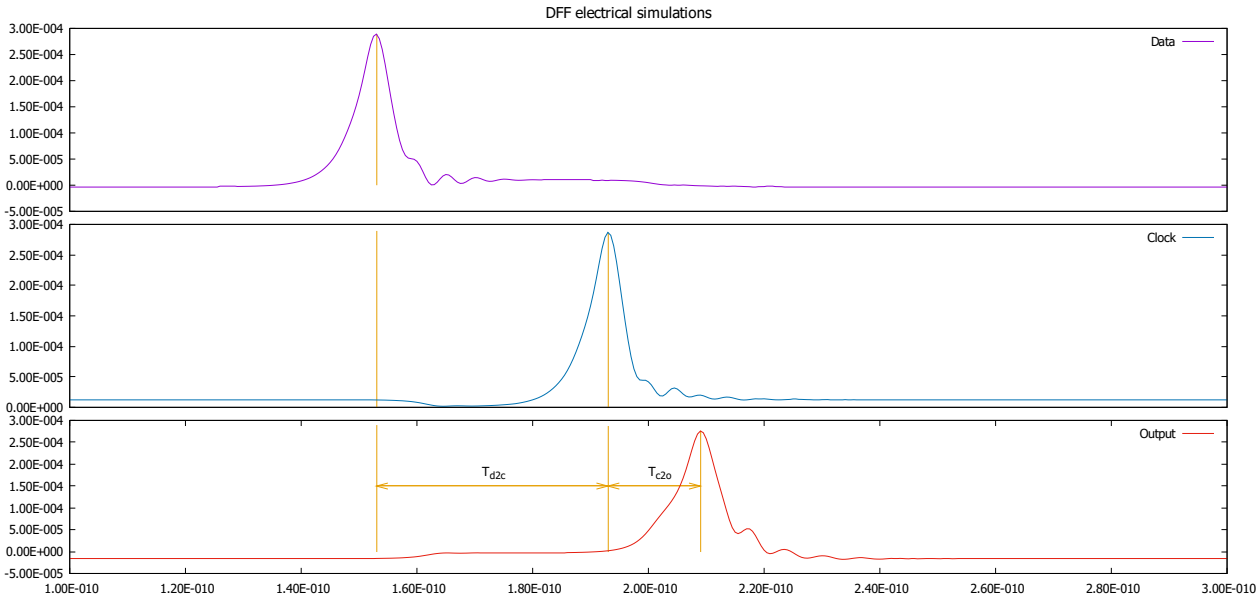


Figure 3.5: DFF circuit simulation waveforms

3.2.2 Time delay extraction

In this work, bias dependent circuit timings were evaluated. The effects of bias variation in a cell delay were first investigated and later the effect of a change in bias of the neighboring cells was analyzed. To speed up the timing extraction process, various scripts written in Octave were developed. The main aim of the scripts was to automatically vary the bias voltage in user-defined steps within the operation margin, invoke the JSIM simulator every time the values are updated, compute and record the delay at every simulation instance and plot the bias vs time delay. This resulting variation was then used with curve fitting tool to formulate a generic function to compute delay for an arbitrary bias voltage. In the case of asynchronous cells, this is straightforward. In clocked cells, the setup time and hold time in addition to propagation delay is evaluated in each instance. To handle this, a script for extracting setup and hold time was invoked at every bias step.

The pseudo code for the script used in extracting delay timing in asynchronous cells is shown in listing 3.1

The script starts by creating an output file that will be used to store the bias value and the corresponding delay computed at that simulation instance. Bias voltage is set to vary from lower margin value to upper margin value. The user specifies the margin according to the specifications from the cell designers. Most of the cells in the SU cell library have an operation margin of ±30%. In such cases the bias voltage varies from -30% of the design value to +30% of the design value. At every step of the bias voltage variation, the JSIM netlist file is opened for reading and a new file is created for writing. The former is then copied to the latter line by line. Before the line is written into the destination file, a variable label is searched for and if it is found the bias value is updated. This label is inserted in the design file before timing extraction starts. The new bias value is equal to the value of

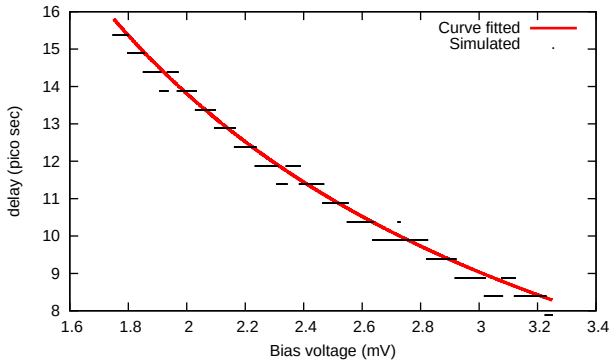
Listing 3.1: pseudo code for the script used in extracting delay of asynchronous cells

```
open file delay.dat for writing
for bias_value = lower_margin to upper_margin
  begin
    open file JSIM_netlist file F1 in read mode
    open file new JSIM file F2 in write mode
        while(not end of file F1)
          begin
                read a line from F1
                search for label: variable
                if(label found)
                  begin
                        locate the current bias value
                        update the bias value
                  end if
                copy the line to F2
          end while
        close F1 and F2
        run JSIM with F2
        compute delay
        write delay and the current bias value to delay.dat
  end for
close delay.dat
plot the bias values vs delay from delay.dat
curve fit
```

the iterator. Once all the lines are copied to the destination file in a particular step, JSIM starts to simulate the new file. The resulting data file is then used to compute the cell delay based on pulse position, as illustrated in the previous section. The current bias value and the cell delay are then written to the delay file created at the beginning of the script execution. Each iteration ends with closing the original JSIM netlist file and its copy. When all the iterations are exhausted, a graph of bias values versus cell delay is plotted from the delay file. A generic function is then formulated using curve fitting techniques. With this function, the cell delay can be calculated accurately at an arbitrary bias value within the operation margin

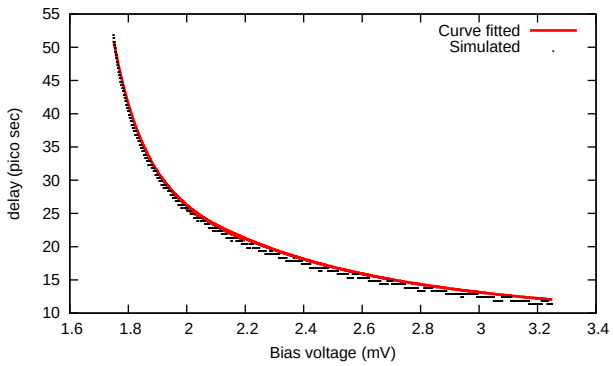
To evaluate the fitness of the model, graphical examination and a statistical report from the curve fit tools were used. Graphically, there are several models that seem to fit the delay variation, however the reported one was selected based on the best statistical values. The three statistical parameters considered were, summed square of residuals (SSE), the square of the correlation between simulation value and fitted value (R-square), and root mean square error (RMSE). SSE measures the total deviation of simulation value to the fit. The fit is considered to be more useful if this value is closer to zero. The R-square can take a value between 0 and 1. The closer the value is to 1 the better the fit. RMSE also known as a fit standard error, it indicates how close the simulation values are to the fit of the delay variation. Values of RMSE closer to zero indicate the most useful model.

The following figures [3.6 - 3.14] show the delay vs bias voltage and curve fit plots of



Curve fitting equation
$p_d(v) = av^b + c$
$a = 2.907e - 11$
$b = -0.9154$
$c = -1.605e - 12$
Goodness of fit
$SSE : 2.737e - 22$
$R - square : 0.99$
$RMSE : 2.136e - 13$

Figure 3.6: JTLbias dependent delay model



Curve fitting equation

$$p_d(v) = \begin{cases} a_1 exp(b_1 v) + c_1 exp(d_1 v) & 1.75 \leq v \leq 2.2 \\ a_2 exp(b_2 v) + c_2 exp(d_2 v) & 2.2 < v \leq 3.25 \end{cases}$$

$a_1 = 0.001139$ $a_2 = 7.312e - 10$
$b_1 = -10.19$ $b_2 = -2.063$
$c_1 = 1.355e - 10$ $c_2 = 1.985e - 11$
$d_1 = -0.8528$ $d_2 = -0.178$
Goodness of fit
$SSE : 1.407e - 22$ $SSE : 2.711e - 22$
$R - square : 0.9987$ $R - square : 0.9962$
$RMSE : 2.799e{-}13$ $RMSE : 2.329e{-}13$

Figure 3.7: Splitter bias dependent delay model

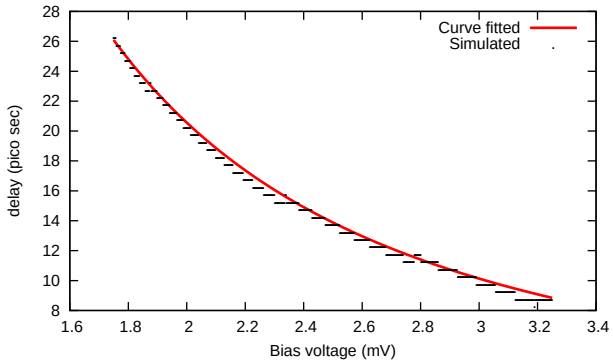
selected cells in the SU library alongside their curve fitting details. The fit equation is plotted in red while the simulation values are plotted in black. In the figures the dots appear as line segments because the number of points considered is quite large. All the cells were simulated at more than 4000 points within their bias margin. The curve fits account for more than 98% of variance as depicted by R-square values except for the PTL receiver cell which is 89%.

.

3.2.3 The extraction of other critical timing parameters

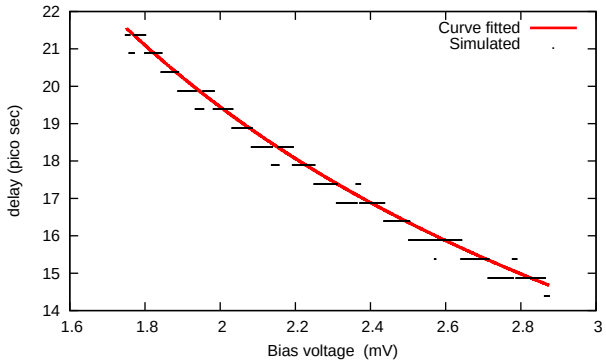
In this section, input-pulse-position related timing extraction methods are presented. These timings provide the forbidden window for input in SFQ circuit. First, the methods for unclocked cells are presented, second, the method for single input clocked cells and third, the method for two input clocked cells.

With regard to JTL cell, the minimum separation time estimation process is explained with the aid of Figure 3.15. The circuit is fed with two input pulses per simulation session. Initially, these pulses are with sufficient separation such that neither pulse influences the propagation of the other. Then in the consecutive simulation sessions, the separation time



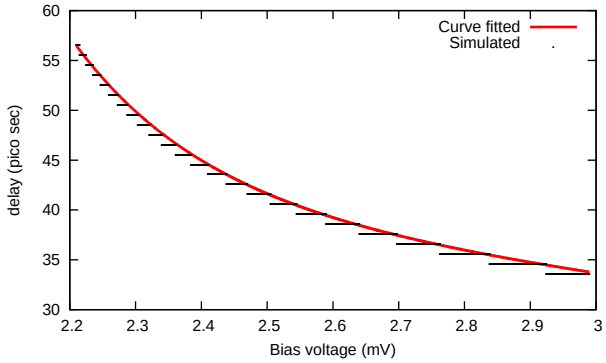
Curve fitting equation
$p_d(v) = av^b + c$
$a = 7.108e - 11$
$b = -1.842$
$c = 7.382e - 13$
Goodness of fit
$SSE : 6.226e - 23$
$R - square : 0.9977$
$RMSE : 2.28e - 13$

Figure 3.8: Confluence Buffer bias dependent delay model



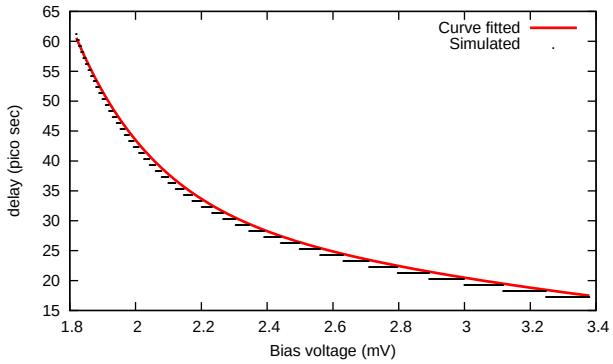
Curve fitting equation
$p_d(v) = av^b + c$
$a = 3.363e - 11$
$b = -0.7535$
$c = -4.99e - 13$
Goodness of fit
$SSE : 1.837e - 22$
$R - square : 0.9895$
$RMSE : 2.021e - 13$

Figure 3.9: D Flip-Flop bias dependent delay model



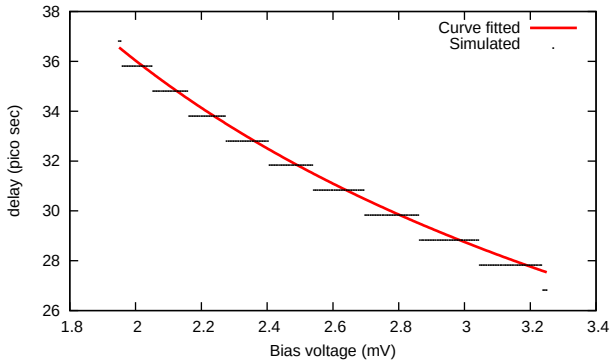
Curve fitting equation
$p_d(v) = a.exp(bv) + c.exp(dv)$
$a = 1.149e - 6$
$b = -5.08$
$c = 7.396e - 11$
$d = -0.265$
Goodness of fit
$SSE : 2.749e - 22$
$R - square : 0.9974$
$RMSE : 3.029e - 13$

Figure 3.10: AND gate bias dependent delay model



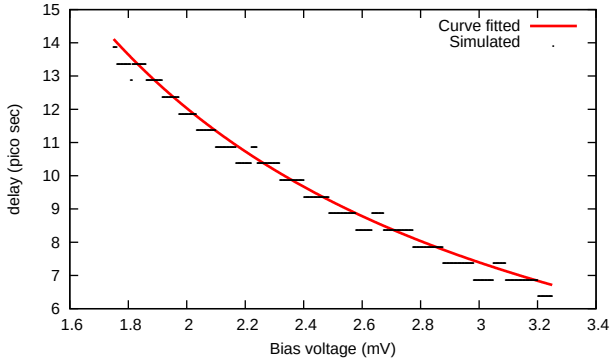
Curve fitting equation
$p_d(v) = a.exp(bv) + c.exp(dv)$
$a = 5.572e - 8$
$b = -4.171$
$c = 6.703e - 11$
$d = -0.3985$
Goodness of fit
$SSE : 1.387e - 22$
$R - square : 0.9989$
$RMSE : 3.404e - 13$

Figure 3.11: XOR gate bias dependent delay model



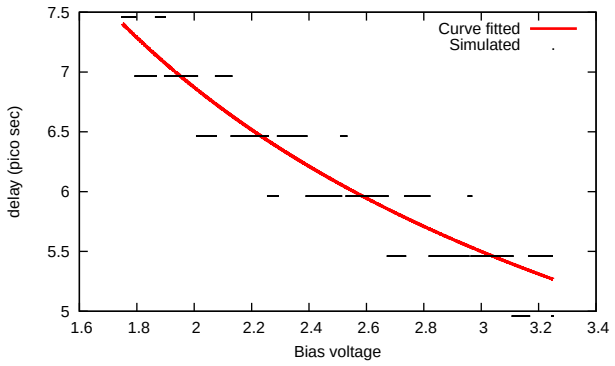
Curve fitting equation
$p_d(v) = av^b + c$
$a = 4.842e - 11$
$b = -0.6795$
$c = 5.799e - 12$
Goodness of fit
$SSE : 2.136e - 23$
$R - square : 0.9874$
$RMSE : 2.935e - 13$

Figure 3.12: NOR gate bias dependent delay model



Curve fitting equation
$p_d(v) = av^b + c$
$a = 2.764e - 11$
$b = -1.181$
$c = -1.589$e-13
Goodness of fit
$SSE : 2.844e - 23$
$R - square : 0.9892$
$RMSE : 2.181e - 13$

Figure 3.13: PTL driver bias dependent delay model



Curve fitting equation
$p_d(v) = p_1 v + p_2 v + p_3$
$p_1 = 4.761e - 13$
$p_2 = -3.766e - 12$
$p_3 = 1.25e - 11$
Goodness of fit
$SSE : 2.673e - 22$
$R - square : 0.8917$
$RMSE : 2.111e - 13$

Figure 3.14: PTL Receiver bias dependent delay model

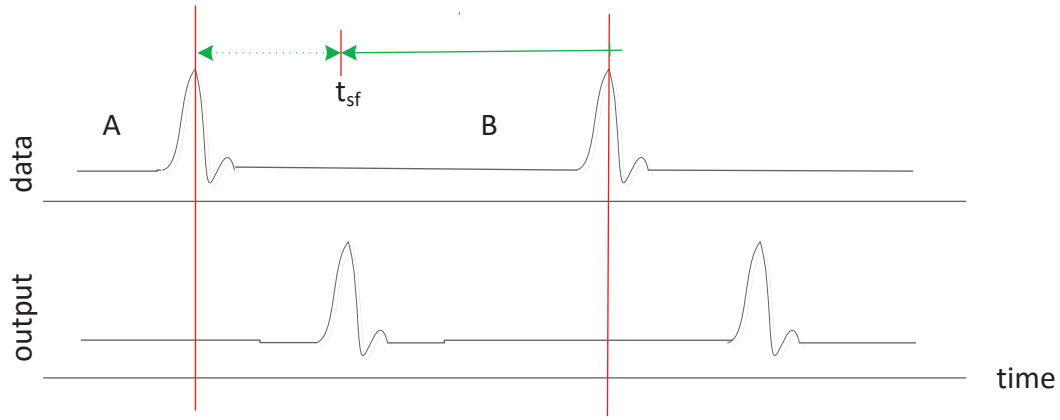


Figure 3.15: Minimum input pulse separation for JTL

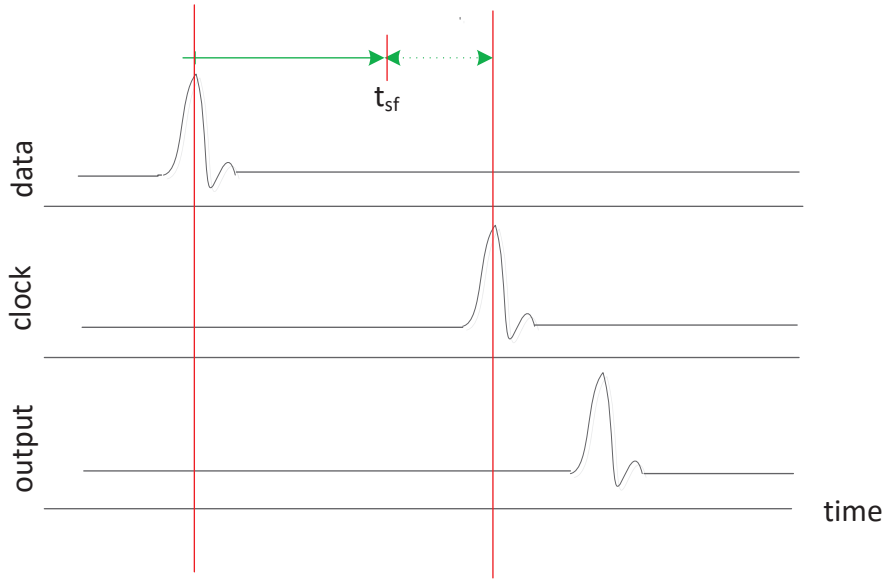


Figure 3.16: Bits pattern for evaluating input to clock (I2C) minimum time

is reduced in steps of 1 ps or suitable values. The green line in the figure indicates that the second input pulse is moved closer to the first input pulse. The process stops when the second output pulse is not produced. Then the previous time position of the second pulse (t_{sf}) is taken and the difference between t_{sf} and the first pulse position defines the forbidden window for the input. This is indicated by the arrow-ended dashed line in the figure. A similar approach was used with a splitter, however, in this case if any of the outputs fail to be generated the process is stopped.

In the case of a confluence buffer, the position of input pulse in one terminal with respect to the other influences the delay of the circuit as their separation reduces and generates only one pulse if the separation is very small. In this work, the model is restricted to constant delay for a certain bias voltage. Thus, the minimum separation of pulses in input 1 and input 2 is estimated followed by the minimum pulse separation in one input line. The former is obtained by fixing the position of a pulse in one of the inputs and then varying the other such that the separation reduces until the second output pulse experiences a delay greater than the first output pulse by 2 ps. The latter is achieved in a similar approach to JTL with one input not supplied with input pulses.

Clocked circuits with one input and one output require the extraction of the following timing parameters: minimum time the input should precede the clock (I2C) analogous to setup time, minimum time that the circuit takes to allow another input after the clock (C2I) analogous to hold time and the minimum clock to clock separation. To evaluate I2C

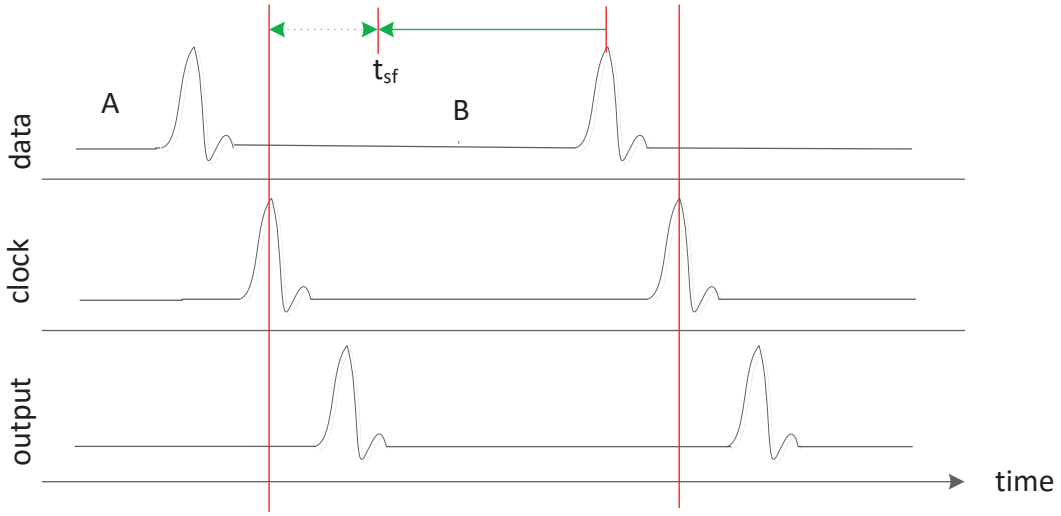
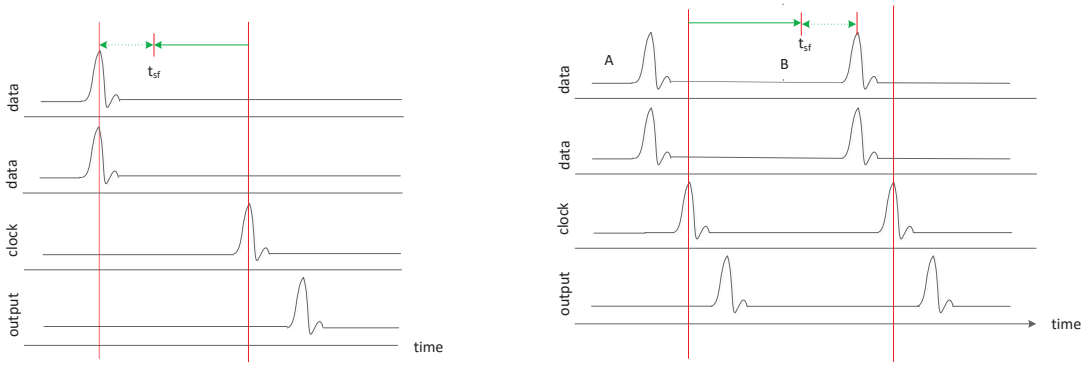


Figure 3.17: Bits pattern for evaluating clock to input (C2I) minimum time

minimum, the test bits pattern is set as shown in the Figure 3.16. Initially, enough time is provided between the data and the clock pulse for the circuit to operate correctly. The position of the data pulse is then varied iteratively simulating the circuit until the circuit fails. Then the previous time position of the data pulse is subtracted from clock pulse time position to obtain the minimum I2C. To obtain C2I minimum time, the test bits pattern are set as shown in Figure 3.17. In this figure, the circuit operates normally in both clock cycles A and B. The position of the input pulse in clock cycle B is then move closer to the clock pulse in the cycle A in steps. In each step, the circuit is simulated. If the circuit fails in cycle B, then the previous position of data pulse in cycle B t_{sf} delineates the input's forbidden region and the allowed region. C2I is then computed by subtracting the position time of clock in cycle A from t_{sf}

In the case of two input clocked cells such as AND gate, the input combination that results in a logic 1 is used in timing estimation. To find the minimum I2C, the input pulses are held at a fixed position while the clock pulse position is adjusted in steps towards the data pulses. Simulation is carried out at every step to verify the operation. If the circuit malfunctions, the previous step marks the separation between input forbidden region and the safe region. Figure 3.18a shows the initial bit pattern for a 2-input AND gate. C2I minimum time, in this case, is found by using the initial bits pattern setup shown in figure 3.18b, then varying the position of the clock pulse in cycle A towards the inputs of cycle B. Simulation is performed in each step. The process stops when the circuit operation fails. The time position of the second data set minus the time position of the clock pulse in cycle A is the C2I.



(a) **(b)**

Figure 3.18: Bits pattern for evaluating (a) input to clock (I2C) minimum time and (b) clock to input (C2I) minimum time

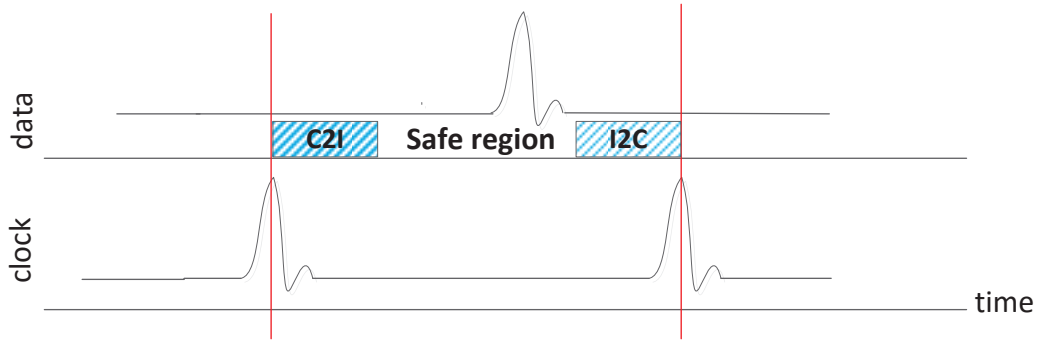


Figure 3.19: clock pulses separation

Table 3.1: Timing parameters of clocked cells expressed in terms of propagation delay

Cell	propagation delay (clock to output)	Input to clock separation (min) (I2C)	Clock to input separation (min) (C2I)
NOT	$p_d(v) = -1.716e-11 * v^3 + 1.458e-10 * v^2 - 4.353e-10 * v + 4.877e-10$	$0.33p_d$	$0.31p_d$
OR	$p_d(v)6.438e-11 * v^{-1.545} + 1.228e-12$	$1.53p_d$	$0.87p_d$
NOR	$p_d(v) = 4.842e{-}11{*}v^{0.6795}{+}5.799e{-}12$	$2p_d$	$1.0p_d$
DFF	$p_d(v) = 3.363e{-}11{*}v^{-0.7536}{-}4.99e{-}13$	0	$0.5p_d$
AND	$p_d(v) = 1.149e-6 * exp(-5.08v) + 7.396e-11 * exp(-0.265v)$	0	$0.63p_d$
NAND	$p_d(v) = -3.066e-17 * exp(3.626v) + 4.831e-11 * exp(-0.108v)$	0	$0.65p_d$
XOR	$p_d(v) = 5.572e-8 * exp(-4.171v) + 6.703e-11 * exp(-0.3985)$	0	$1.0p_d$

Upon obtaining the forbidden and the safe input regions of the each cell the minimum separation between clock pulses can be computed as the sum of C2I, safe region, and I2C. This can be visualized as shown in Figure. 3.19. In this figure, the shaded region represents the forbidden regions while the unshaded is the safe region within the clock period. The safe region can be adjusted to fit the designer requirements whereas the forbidden region is cell dependent. The values of C2I and I2C have been expressed in terms of the cell's propagation delay. The timings for the analysed cells are presented in Table 3.1. In this study, the input pulse separation for the asynchronous cells is taken to be its propagation delay.

3.3 Conclusion

The purpose with this section of the study was to perform the extraction of the timing parameters of RSFQ cells. The methods used to extract the propagation delay in both clocked and unclocked cells, and extraction of input to clock and clock to input timings are presented. The propagation delay is expressed in a mathematical model to cater for variation in bias voltage. I2C an C2I timings are expressed in terms of the propagation delay of the cell. These timing parameters are useful in developing accurate HDL models of RSFQ circuits.

Chapter 4

HDL Cell Library

4.1 Introduction

The two methodologies used in integrated circuit (IC) designs are full custom and semi-custom. In full custom, a detailed manipulation of the electrical and physical characteristics of the IC through circuit simulation tools and layout design tools is performed. As the circuit size and complexity increases, this method becomes extremely time consuming and results to high non-recurring engineering cost. The semi-custom method on the other hand, is based on pre-designed logical or functional level circuits as standard cell libraries or cell arrays. The designer builds a large circuit by placing and interconnecting the standard cells from a standard cell library or functional cells in a cell array. This method is fully automated in the semiconductor technology and it is extensively used to reduce time-to-market of an IC.

In superconducting technology ICs, full custom method dominates. Nevertheless, the semi-custom design has been demonstrated. The cell-based design has been deployed in various superconducting logic families such as RSFQ [64], RQL[65] and AQFP[66]. Cell array, on the other hand, has only been reported on RSFQ [67]. In order to reap the benefits of semi-custom design in superconducting logic circuits, automation from high-level design abstraction to a low level is necessary. This has led to cell libraries containing behavior view of each cell, presented in industrial standard hardware description language (HDL).

At the beginning of this chapter there is a brief discussion of cell library attributes, followed by a review of HDL modeling methods. An example of the non-parameterized method for modeling SFQ circuits is shown using AQFP logic family. Thereafter a method for modeling RSFQ cell behavior utilizing only the mathematical delay model is demonstrated. Finally, a RSFQ HDL cell library is discussed.

4.2 Cell Library

A cell library comprises of a comprehensive set of primitive logic or functional elements (cells) such as gates, latches, etc., that are characterized by fixed physical and electrical characteristics. The cell characteristics are generally specific to the developer and fabrication process. A cell library is a crucial part of advanced circuit design flow as it forms the backbone of design automation and therefore the accuracy of cell information is paramount. Different developers may provide varying cell information depending on the target application. Typical information found in superconducting circuits cell libraries is timing parameters, layout view, cell area, netlist symbolic view and behavior view. A collection of behavior views of all the cells in the library presented in a hardware description language (HDL) is herein referred to as HDL cell library.

The process of creating a cell library which contains the behavior model of the cells can be divided into three parts. Part one is the small scale circuit design flow, part two is cell characterization and part three is the HDL modeling of the cell.

Figure 4.1 shows a typical cell library development flow chart. A cell begins with the function specification such as a logical expression which relates to the input and outputs. A design schematic is then developed and used to generate the circuit netlist. The functionality of the circuit is verified by performing electrical simulation on the netlist. The circuit is then optimized for the required operation margins. Once the optimal design parameters are obtained, the circuit layout is built. To ensure that the layout accurately represents the designed circuit, the component parameters are extracted from the layout and compared with the design values. In case the values do not match, the layout is modified to obtain the required accuracy. The circuit is then taken through a characterization process where parameters such as time-related, area, power etc., are extracted and stored in a cell library as attributes of the cell. Using the timing parameters of the cell, a behavior description is developed using HDLs such as VHDL or Verilog HDL. High-level logic simulators are then used to simulate the circuit operation from the description. The high-level and low-level simulation results are further compared to validate the HDL model. The resulting HDL model is then added to the cell's attributes in the cell library. A brief review of popular existing cell libraries is presented next.

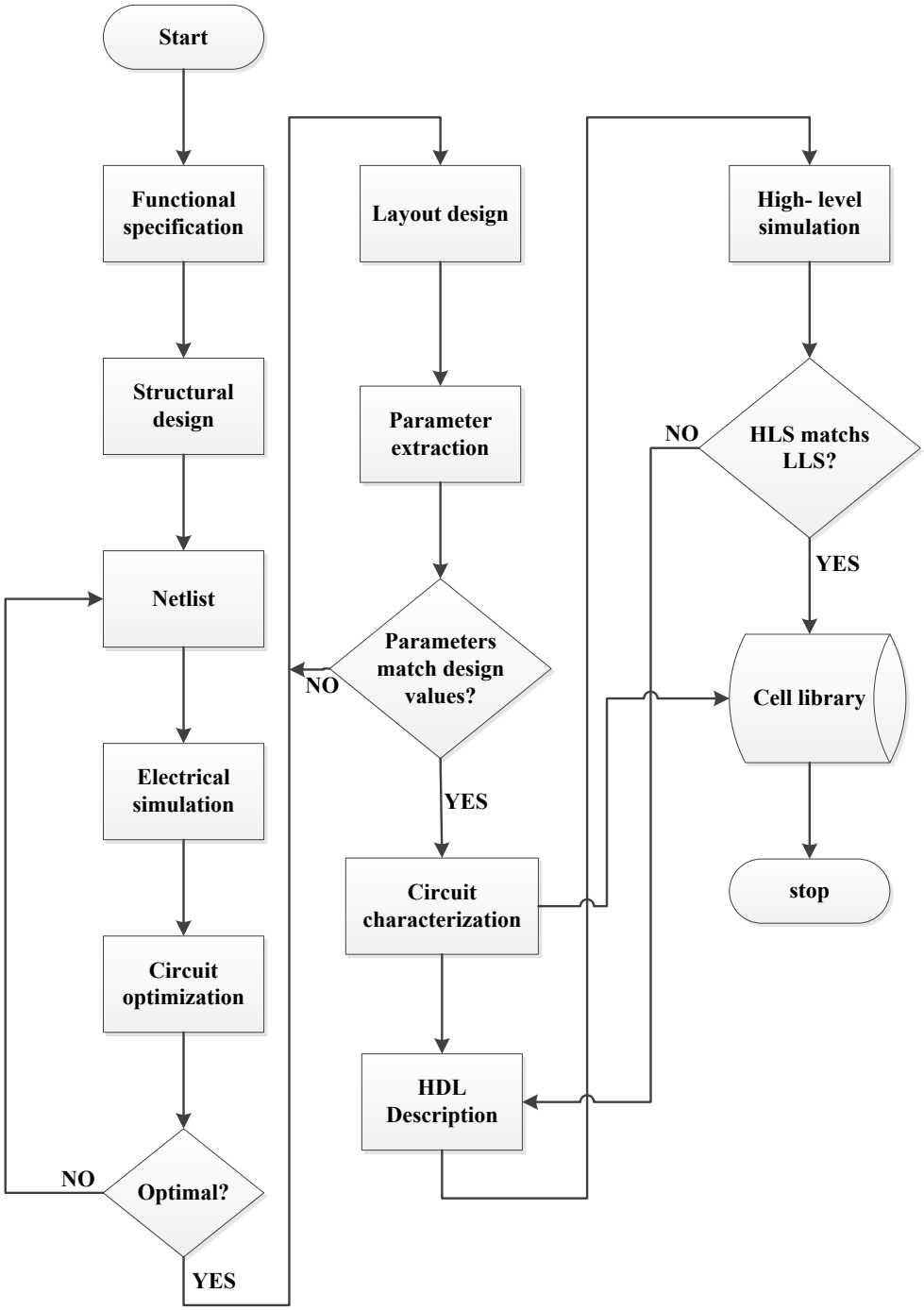


Figure 4.1: A flow chart for the creation of a cell library. HLS-High level simulation, HLL - Low level simulation

4.2.1 CONNECT Library

CONNECT library was developed by a team of Japanese researchers from Yokohama National University and Nagoya University. It was designed into a unified simulation environment in Cadence CAD tools. Initially, it was reported to have around 100 cells [64],

however, it was later reported to have about 230 cells [13]. The cells are designed such that when interconnected, their static interaction is negligible. Most of the cells in this library are wiring cells some of which are the same type duplicated with different sizes, or layout orientation, or delay timings. This provides design flexibility.

Each cell has the following attributes:

- Analog model simulatable with JSIM
- Schematic view; used in the design capture phase of the design flow.
- Cell layout; also known as the layout view, developed based on the NEC standard fabrication process. The cell layouts are designed such that the height and width are multiples of 30 micrometers.
- A logic model developed on Verilog HDL to facilitate logic simulation which is faster than analog simulation. The models use bias-voltage-dependent timing delays and constraints given in a lookup table.

This cell library allows the designer to design a circuit with the final physical layout in mind. The design is captured in schematic view with the placed cells being oriented to provide logical connection while achieving direct correlation with physical layout view [68]. The modular approach can be used with the CONNECT library to design large RSFQ logic circuit.

4.2.2 Stony Brook University VHDL cell library

The ultra high-speed computing (UHSC) laboratory at Stony Brook Univesity (SBU) has two VHDL cell libraries for SFQ logic circuits, the SBU tunable VHDL cell library for RSFQ and the SBU VHDL RQL cell library .

1. SBU tunable VHDL RSFQ cell library

This library was developed to handle the high-level design verification of RSFQ logic circuits [29, 65] . It was tuned to Hypres 1.5 micrometre 4.5 kA/cm2 standard niobium process but provides flexibility with the aid of a customizable parameter file, through which cell parameters can be edited to tune the model to a different fabrication process [68].

2. SBU VHDL RQL cell library

This library was basically developed for a similar purpose as the previously discussed library but on RQL logic circuits. The cell library has been successfully tested with Mentor Graphics design and verification tools [69]

4.2.3 Stellenbosch Univerity cell library

Stellenbosch university (SU) cell library consists of two sets of 14 RSFQ primitive cells, one based on the IPHT's 1 kA/cm^2 process and the other based on the Hypres 4.5 kA/cm^2 process. The cells designed with the IPHT process have layout dimensions which are multiples

of 150 micrometres and those built with Hypres process have layout dimensions in multiples of 100 micrometres [52]. The library consists of JTLs, Splitters, Confluence buffers, DFFs, TFFs, NOTs, ANDs, ORs, NANDs, NORs, XORs, XNORs, DC-SFQs, SFQ-DCs, PTL Drivers, and PTL receivers. At the beginning of this work the attributes available for the cells were analog schematic, JSIM netlist, layouts, and propagation delay time under design static parameters. To build the HDL models, which constitute to HDL cell library, a subset of the cell library which excludes DC-SFQ and SFQ-DC . Each of these cells was then characterized to extract the required timing model.

4.2.4 Other cell libraries

One of oldest RSFQ cell libraries in the literature was reported in Krasniewski's work 1993 [63]. The library of 15 cells was built at the University of Rochester on DesignWorks^{TM} to utilize the user-friendly nature of the tool and perform the logic simulation. Later, in 1997, a Verilog HDL behavior model library was developed at the same university [61] for use with the Cadence Verilog –XL logic simulator. The authors claimed that the Verilog models were not limited to this particular simulator, however, no further work provided the evidence.

There are other cell libraries mentioned in the literature but their documentation available in the public domain is limited. Some of these are FLUXONIC cell library [70], and RSFQ64 cell library [71][72].

4.3 SFQ information encoding for HDL models

The challenge faced when using conventional HDLs to model SFQ circuits is that these languages are created entirely to handle voltage-level coded information. In order to create a cell library with HDL functional models of the cells, or that can be used with standard logic simulators in the design of a large circuit, SFQ pulses need to be suitably encoded. Different encoding methods have been used in literature. In this work, these methods are classified into three: pulse-encoded, event-based and level-based methods.

4.3.1 Pulse-based encoding

In this method the cells are modeled as if they were pulse driven CMOS circuits. An SFQ pulse is encoded as a standard logic pulse of fixed width as shown in Figure 4.2. This method was first used to create a cell library for use with schematic capture tool with standard logic simulator [63] but was later used in building several other RSFQ cell libraries using HDLs. The method requires that every output is returned to zero after a duration equal to the pulse width. This results in an increase in circuit simulation time which becomes significant in very large circuits.

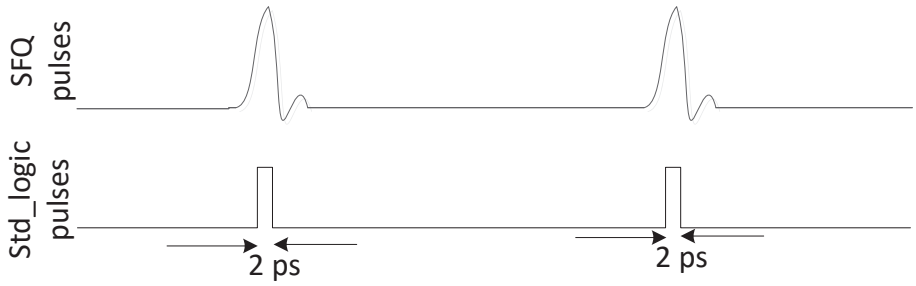


Figure 4.2: SFQ pulse encoded as std_logic pulse

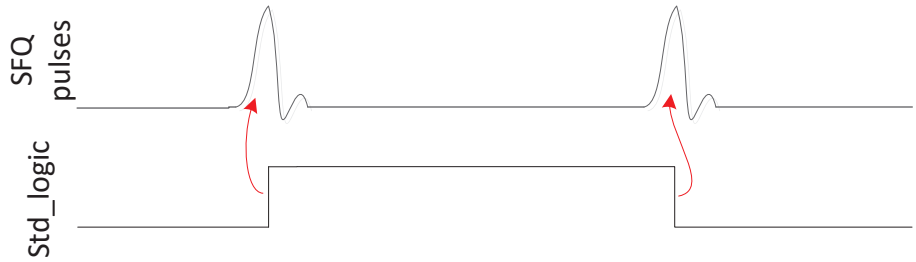


Figure 4.3: SFQ pulse encoded as signal event

4.3.2 Event-based encoding

Pioneered in SBU, the event-based method has been used in developing the SBU tunable cell library [68] discussed in section 4.2.2. It uses a change of logic level (event) to represent the occurrence of a SFQ pulse. Thus, a change in signal value from '0' to '1' or from '1' to '0' in the HDL functional model of the cell is used to represent a SFQ pulse. It is assumed that the SFQ pulse width is so small that it can be modeled as a level transition. Fig. 4.3 shows the signal encoding. Circuits modeled using this method take less time to simulate than the previously discussed method. The main drawback of this method is that, the waveform analysis is not straightforward.

4.3.3 Level-based encoding

Similar to conventional CMOS logic, in level–based method, the superconducting logic '1' is represented as high voltage/current level and '0' as low voltage/current level. This method can be used in superconducting logic families such as MVTL, COSL, RQL, and AQFP. While it may be easy to apply this method to MVTL and COSL, it is not straightforward in the case of RQL and AQFP. This is because they use AC biased and clock. The AC wave is divided into three regions as shown in Figure 4.4. This is done by approximating a

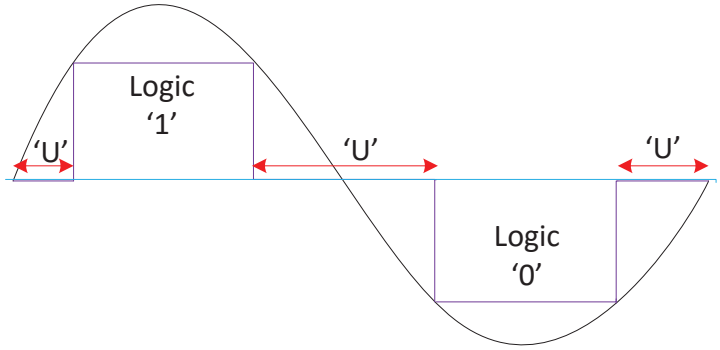


Figure 4.4: Level-based encoding of signals in SCE

pulse in both the positive and the negative cycle. The lower level in the figure is encoded as logic '0' and the high-level as the logic '1'. The off region marked 'U' does not carry information and the developer may choose to map it to one of the standard logics such as 'x' or 'z'. This encoding method was used to develop the RQL HDL models in [73] and the non-parameterized AQFP HDL cell library in this work.

4.4 HDL modeling of SCE

HDL modeling involves the use of hardware description languages to capture a circuit design such that the design is simulated with logic simulators. The model is usually behavioral and/or structural description of the circuit. In HDL cell library development, small-scale circuit implementing basic logic primitives are described with HDL. Generally, the circuit behavior and timings extracted from circuit level simulations are used to develop the HDL model. The two common methods used in literature are the hand coded description, and automated description. The hand coding method requires the designer to write the behavior circuit description while specifying the timing constraints. This method results in a compact and easy to debug HDL model, however it is relatively time-consuming. The automated method utilizes a program or/and scripts to automatically generate the HDL model of a circuit from its JSIM/SPICE netlist. This method is fast and easy to use, but debugging the code is not easy as the code is not usually optimal. The automation program may incorporate the timing extraction [74] or utilize pre-extracted timing parameters in a cell library. HDL models used to simulate just the logic operation of the circuits can be built without timing constraints. In this section, two methods for building HDL cell libraries are discussed. First, A method to model AQFP logic cells by the multi-valued approach and second a method to model RSFQ cells with bias dependent delay.

4.4.1 The AQFP HDL cell Library

At the time of creating this cell library, design of the AQFP logic circuits was entirely based on circuit-level simulators. This cell library was developed to demonstrate that AQFP circuits can be modeled using HDLs and thus simulated at the logic level. The cell library is based on the first generation of the AQFP described in [18]. The logic gates were designed and optimized by the YNU research team from whom the JSIM netlist were acquired. The main objective being to simulate the behavior of logic circuit design under normal operation, the cell library does not perform critical time check. The method used is presented in three parts:

1. Simulation of AQFP gates using the JSIM circuit simulator to observe their behaviour
2. Encoding the AQFP signals and mapping them to IEEE 9-state logic system
3. Developing HDL models

The method is hereby presented with the help of a majority gate. The model was developed using VHDL and Verilog HDL. Open source simulators, FreeHDL and Icarus Verilog were used to simulate VDHL and Verilog models respectively.

4.4.1.1 Circuit level gates simulation

To obtain the circuit behavior at the circuit level, JSIM simulations were carried out on the netlist. The circuit is excited with a symmetrical trapezoidal current wave of 1.6 mV and 100 ps rise and fall time . Inputs are supplied with a current of -5 μA or +5 μA depending on the binary information being supplied to the inputs. Figure 4.5 shows the circuit simulation output of AQFP majority gate. The inputs shown in the figure have been obtained from the output of the AQFP buffer. The inputs and outputs of the circuit can be positive, negative or zero current.

4.4.1.2 AQFP signal encoding

In order to develop an HDL model of an AQFP gate, the AQFP signal obtained from the circuit simulation needs to be encoded in a subset of IEEE standard logic. The trapezoidal excitation current is encoded as a square wave of standard level logic, as shown in Figure 4.6. The wave is divided into 'on' and 'off' regions which are represented as logic 1 and logic 0 respectively. Since cell characterization for timing extraction was not part of the study, the AQFP cell is considered fully excited when the excitation current reaches its maximum level. As shown in the Figure 4.5, the inputs and output of the AQFP gates can acquire three states. The negative current state is encoded as logic '0' and the positive current as logic '1'. In this work the 'no current' state is represented as logic 'z' equivalent to the high impendence of standard logic in HDLs. Moreover, if the AQFP buffer circuit receives

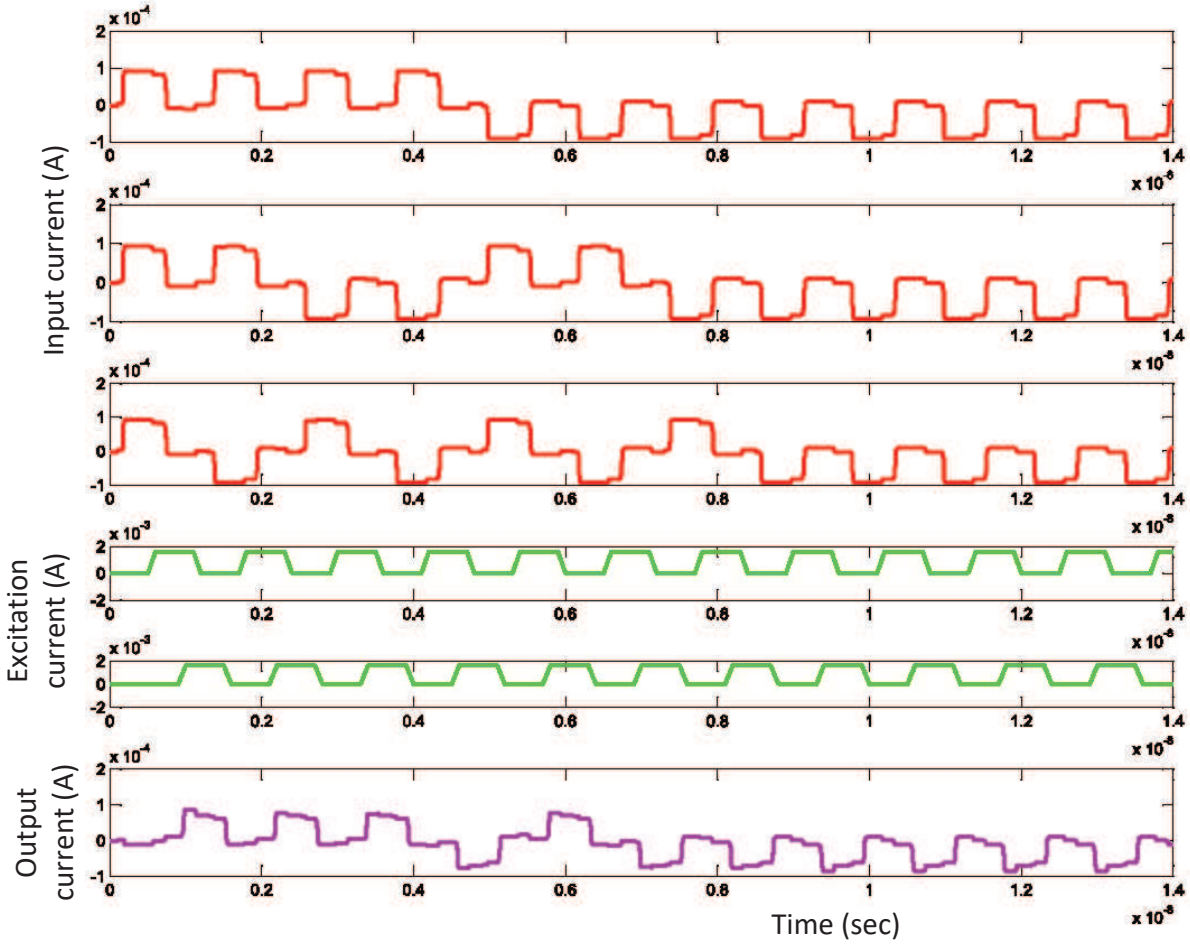


Figure 4.5: AQFP majority gate circuit simulation

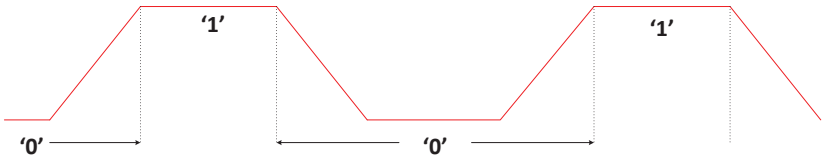


Figure 4.6: Encoding trapezoidal excitation current in standard logic

excitation current when the input is in the no-current state, the output is non-deterministic. Such a logic state is encoded as the 'x' equivalent of an undefined state in standard logic.

4.4.1.3 Developing HDL model

Table 4.1: Basic logic operators AND (&), OR (|) and NOT (~)

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

~	
0	1
1	0
x	x
z	x

Table 4.2: Basic logic operators AND, OR and NOT for AQFP logic modeling

andq	0	1	x	z
0	0	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

orq	0	1	x	z
0	0	1	x	x
1	1	1	x	x
x	x	x	x	x
z	x	x	x	x

notq	
0	1
1	0
x	x
z	x

To model AQFP logic gates, the HDL primitives are first compared with the AQFP equivalent gates based on the encoded logic. Table 4.1 shows the truth table of the three basic primitives AND, NOT and OR. The outputs in red indicate the ones that do not match the AQFP gate operation for the same input combination. Thus, AND and OR primitives cannot be used in AQFP circuit modeling without modification. Since an operation involving 'x' and 'z' either with '0' or '1' cannot result in a definite output, the red marked outputs are replaced with an 'x' for AQFP. This results in table 4.2 which is used to write user defined primitives as functions in Verilog HDL and VHDL. In Verilog HDL the functions are written into a file which is included in every design using the **'include** directive. In VHDL, the functions are written in a package that is used in all AQFP HDL description files by placing the **'use'** directive in the program.

Listing 4.1: User defined primitives in Verilog HDL as functions

```
1  //user defined functions;
2  //operators.v
3  function  qand;
4    input a,b;
5    begin
6      case ({a,b})
7        2'b00: qand = 1'b0;
8        2'b01: qand = 1'b0;
9        2'b10: qand = 1'b0;
10       2'b11: qand = 1'b1;
```

```
11        default:qand = 1'bx;
12       endcase
13     end
14   endfunction
15
16   function  qor;
17     input a,b;
18     begin
19       case ({a,b})
20        2'b00: qor = 1'b0;
21        2'b01: qor = 1'b1;
22        2'b10: qor = 1'b1;
23        2'b11: qor = 1'b1;
24        default:qor = 1'bx;
25       endcase
26     end
27   endfunction
```

Listing 4.2: User defined primitives in VHDL as functions in a user defined package

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  package AQFP is
4   function qor(L,R: std_logic) return std_logic;
5   function qand(L, R: std_logic) return std_logic;
6  end AQFP;
7  package body AQFP is
8   function qand(L, R : std_logic) return std_logic is
9               variable sel: std_logic_vector(1 downto 0);
10  begin
11     sel:= L & R;
12     case sel is
13      when "00"=> return '0';
14      when "01"=> return '0';
15      when "10"=> return '0';
16      when "11"=> return '1';
17      when others=>return 'X';
18     end case;
19  end function;
20
21  function qor(L, R : std_logic) return std_logic is
22   variable sel: std_logic_vector(1 downto 0);
23   begin
24    sel:= L & R;
25    case sel is
26     when "00"=> return '0';
27     when "01"=> return '1';
28     when "10"=> return '1';
29     when "11"=> return '1';
```

```
30      when others=>return 'X';
31    end case;
32  end function;
33 end package body AQFP;
```

The user-defined primitive functions are named 'qand' and 'qor'. Their VHDL and Verilog HDL codes are shown in Listings 4.1 and 4.2 respectively. The functions in both languages utilize **case** statement control structure to implement the the truth of AND and OR shown in Table 4.2. This is represented by lines 6 - 12 for qand and lines 19 - 25 for qor in Verilog code and lines 12 - 18 for qand and 25 - 31 for qor in VHDL code.

Listing 4.3: Verilog HDL model for AQFP buffer

```
1  'timescale 1ps/100fs
2  module AQFP_buffer(data_in, I_ex, data_out );
3  input data_in;
4  input I_ex;
5  output data_out;
6  reg data_out ;
7  always@(posedge I_ex) begin
8    data_out <= data_in;
9  end
10 always@(negedge I_ex) begin
11   data_out <= 'bz;
12 end
13 endmodule
```

Listing 4.4: VHDL model for AQFP buffer

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity AQFP_buffer is
4   port(data_in, I_ex : in std_logic;
5        data_out : out std_logic);
6  end AQFP_buffer;
7  architecture t0 of AQFP_buffer is
8   begin
9    P1: process(I_ex)
10   begin
11    if(I_ex'event and I_ex ='1') then
12      data_out <= data_in;
13    else
14      data_out <= 'Z';
15    end if;
16  end process;
17 end t0;
```

The Verilog HDL and VHDL models of AQFP gate are shown in Listing 4.3 and Listing 4.4 respectively. The excitation current of the gate is modeled as the clock. The behavior of the buffer is captured in lines 7 - 12 in Verilog code and 11 - 15 in VHDL code.

Listing 4.5: Verilog HDL model for AQFP NOT

```
1  `timescale 1ps/100fs
2  module AQFP_NOT(data_in, I_ex1, I_ex2, data_out );
3   input data_in;
4   input I_ex1,I_ex2;
5   output data_out;
6   reg data_out ;
7   reg buffer_out;
8   always@(posedge I_ex1) begin
9     buffer_out <= data_in;
10  end
11  always@(negedge I_ex1) begin
12    buffer_out <= 'bz;
13  end
14  always@(posedge I_ex2) begin
15    data_out <= ~buffer_out;
16  end
17  always@(negedge I_ex2) begin
18    data_out <= 'bz;
19  end
20 endmodule
```

Listing 4.6: VHDL model for AQFP NOT

```
1  library ieee;
2  use  ieee.std_logic_1164.all;
3  use WORK.AQFP.all;
4  entity AQFP_NOT is
5   port(data_in_n, I_ex1, I_ex2 : in std_logic;
6                     data_out_n : out std_logic);
7   end AQFP_NOT;
8   architecture not_t0 of AQFP_NOT is
9    signal buffer_out : std_logic := 'Z';
10   begin
11   P1: process(I_ex1)
12   begin
13    if (I_ex1'event and I_ex1 ='1') then
14        buffer_out <= data_in_n;
15    else
16        data_out_n <= 'Z';
17    end if;
18   end process;
19  P2: process(I_ex2)
20  begin
21   if (I_ex2'event and I_ex2 ='1') then
22     data_out_n <= qnot(buffer_out);
23   else
24     data_out_n <= 'Z';
25   end if;
```

```
26   end process;
27 end not_t0;
```

The NOT gate has two excitation current signals which are considered as clock for the gate. The Listings 4.5 and 4.6 present the Verilog HDL and VHDL models respectively. The behavior of NOT gate is similar to a buffer followed by an inverting buffer, each of them having an independent clock. This behavior has been captured using "always blocks" in Verilog HDL as shown in lines 8 - 19, and processes in VHDL as shown in lines 11 - 26.

Listing 4.7: Verilog HDL model for AQFP Majority gate

```
1  `timescale 1ps/100fs
2  module aqfp_maj(d1, d2, d3, I_ex1, I_ex2, d_out );
3   input d1, d2,d3;
4   input I_ex1,I_ex2;
5   output d_out;
6   reg d_out ;
7   reg bf1, bf2, bf3;
8  `include "aqfp_udp.v"
9   always@(posedge I_ex1) begin
10    bf1 <= d1;
11    bf2 <= d2;
12    bf3 <= d3;
13  end
14  always@(negedge I_ex1) begin
15    bf1 <= 'bz;
16    bf2 <= 'bz;
17    bf3 <= 'bz;
18  end
19  always@(posedge I_ex2) begin
20    d_out<=q_or(q_or(q_and(bf1,bf2),q_and(bf2, bf3)),q_and(bf3,bf1));
21  end
22  always@(negedge I_ex2) begin
23    d_out <= 'bz;
24  end
25 endmodule
```

Listing 4.8: VHDL model for AQFP Majority gate

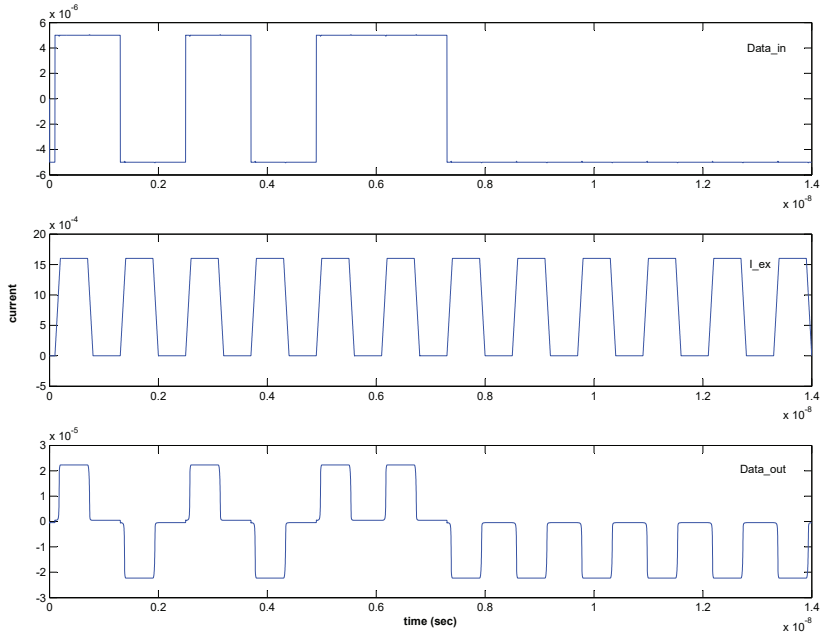
```
1  library ieee;
2  use  ieee.std_logic_1164.all;
3  use WORK.AQFP.all;
4
5  entity AQFP_MAJ is
6   port(d1, d2, d3, I_ex1, I_ex2 : in std_logic;
7   d_out : out std_logic);
8  end AQFP_MAJ;
9  architecture maj_t0 of AQFP_MAJ is
10  signal input_1, input_2, input_3 : std_logic := 'Z';
```

```
11 begin
12  P1: process(I_ex1)
13   begin
14   if (I_ex1'event and I_ex1 ='1') then
15    input_1 <= d1;
16    input_2 <= d2;
17    input_3 <= d3;
18   else
19    input_1 <= 'Z';
20    input_2 <= 'Z';
21    input_3 <= 'Z';
22   end if;
23  end process;
24  P2: process(I_ex2)
25   variable tmp : std_logic;
26  begin
27   if (I_ex2'event and I_ex2 ='1') then
28   tmp := qor(qand(input_2, input_3), qand(input_3, input_1));
29   d_out <= qor(qand(input_1,input_2),tmp);
30  else
31   d_out <= 'Z';
32  end if;
33  end process;
34 end maj_t0;
```

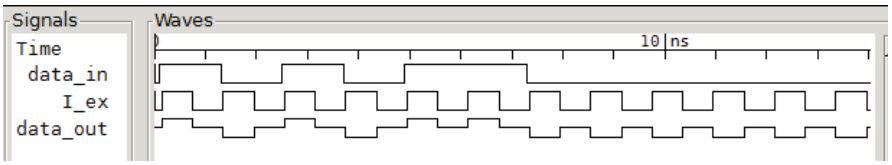
The Majority gate models in Verilog HDL and VHDL are shown in Listings 4.7 and 4.8 respectively. The HDL models utilizes the user define primitive. Line 8 in Verilog code ensures that the primitives are included in the module. A similar task is performed by line 3 in the VHDL code. The two excitation current signals of the gate are used as clock signals. One of the clocks is used at the input stage where the input signals are transferred to the majority logic section which utilizes the second clock. This is shown in lines 9 - 18 in the Verilog code and lines 12 - 23 in VHDL code. The majority logic is implemented by lines 19 - 24 in the Verilog code and 24 - 33 in the VHDL code.

Figures 4.7, 4.8 and 4.9 show the HDL simulation waveforms alongside the circuit simulation waveform for the three AQFP gates; Buffer, NOT and Majority respectively. To handle signal splitting, an AQFP signal splitter was modeled as AQFP buffer with multiple outputs as proposed in [75]

A collection of the AQFP buffer, NOT, Majority and Splitter HDL descriptions formed the cell library. Using this cell library, a single bit full adder structural model was successfully simulated. To further demonstrate the ability to use the cell library in structural modeling of AQFP LSI and VLSI circuits, a 4 bit carry look ahead adder was also modeled and simulated successfully.



(a)



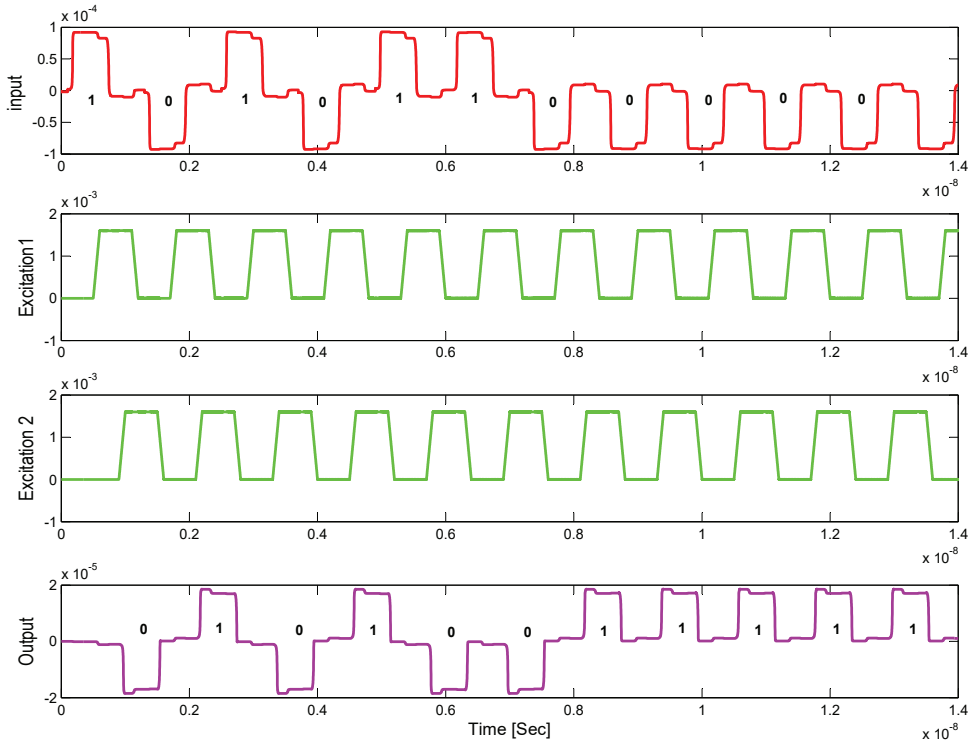
(b)

Figure 4.7: AQFP buffer (a) circuit simulation waveforms (b) logic simulation waveforms

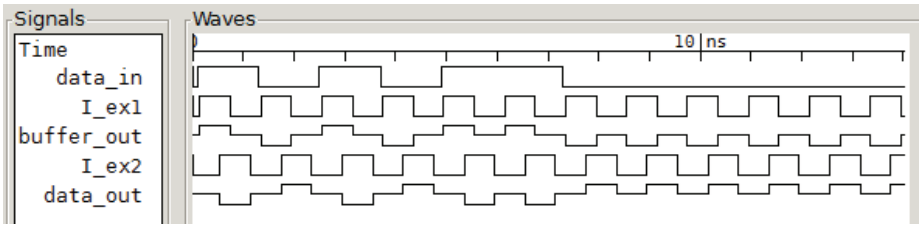
4.4.2 RSFQ HDL cell Library

The RSFQ cell libraries discussed previously in sections 4.2.1,4.2.2 and 4.2.4 of this chapter have HDL models of each cell. Most of them are developed using either Verilog or VHDL with the SFQ pulse encoded as either a standard logic pulse or signal event. The fact that one HDL language is used, forces the designers to use it and those unfamiliar with it to learn it. The model's time parameters are either in a lookup table or pre-calculated fixed value for each cell. While look up table have been used for bias dependent delay modeling, a lookup table has a limited number of bias points. Moreover, these libraries have been built with commercial simulators. Commercial simulators are usually expensive and out of the reach of most beginners and small research groups especially in developing countries.

This work presents a RSFQ HDL cell library with models developed using the two most popular HDLs, VHDL and Verilog HDL, to increase flexibility. The cell libraries are based on open source simulators to ensure that the entire superconducting electronic community

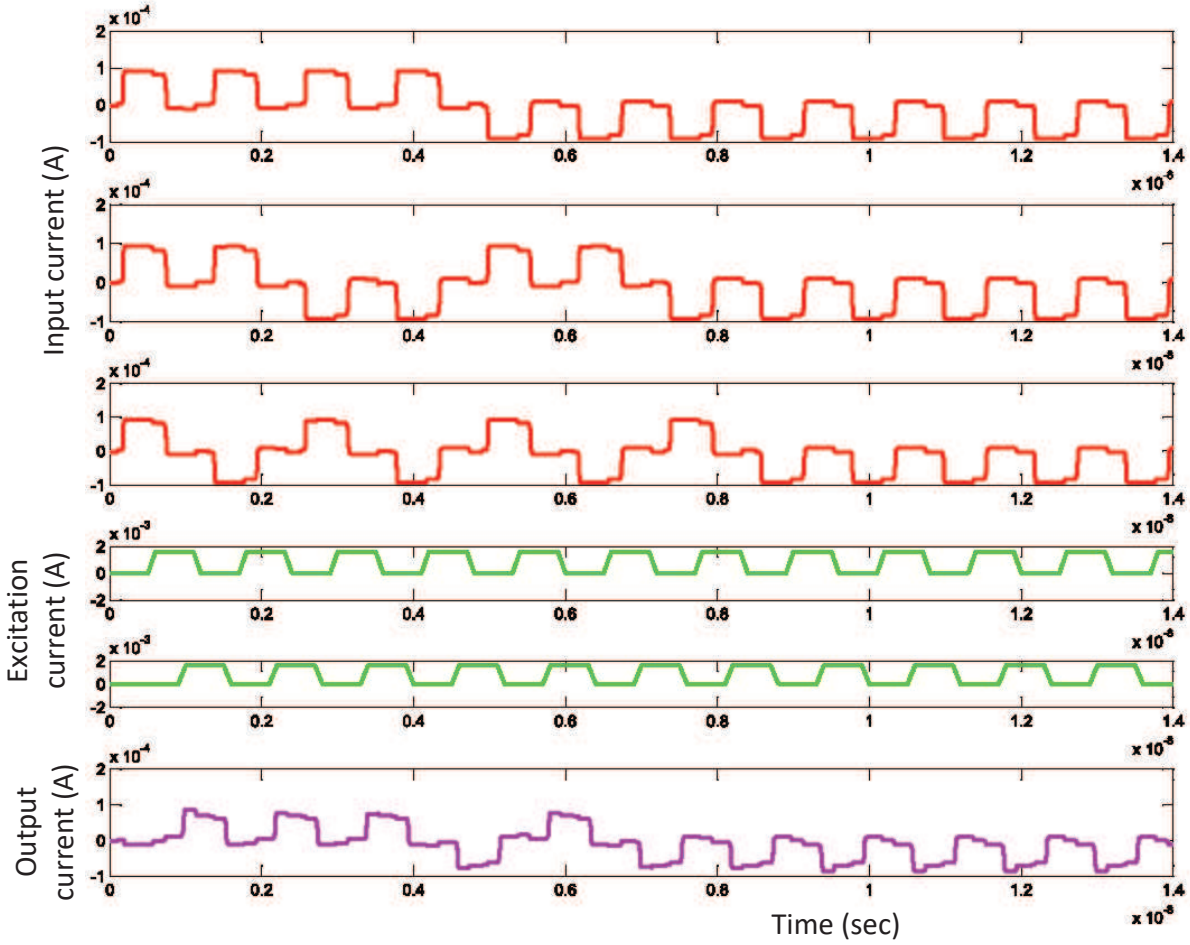


(a)

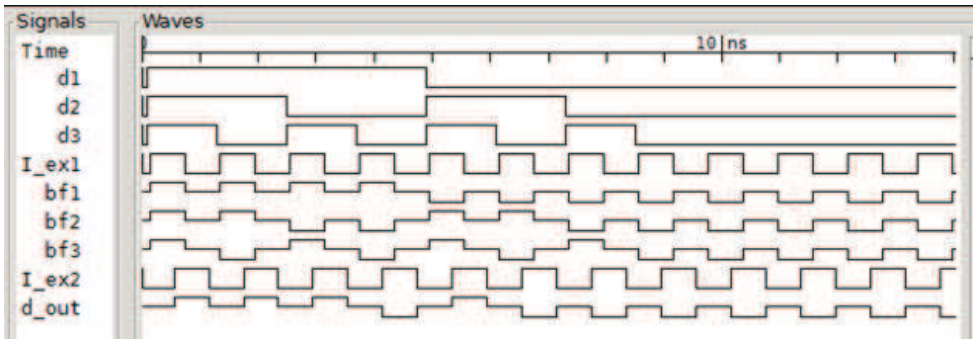


(b)

Figure 4.8: AQFP NOT gate (a) Circuit simulation waveforms (b) Logic simulation waveforms



(a)



(b)

Figure 4.9: AQFP Majority gate (a) Circuit simulation waveforms (b) Logic simulation waveforms

can benefit from this approach. To capture the timing parameters in the HDL model, a mathematical representation of propagation delay dependent on bias voltage is used. All the other timing constraints are then expressed in terms of propagation delay. The timing parameters described in the cell characterization chapter are used in HDL modeling.

4.4.2.1 HDL modeling of RSFQ cells

Two sets of the HDL models for the SU RSFQ cell library were built. The first set of HDL models were built using the standard logic pulse representation of SFQ pulse while in the second set, event-based SFQ pulse encoding was used. To develop the first set of HDL models, a SFQ pulse was encoded as a standard logic pulse, the pulse width of which is two picoseconds. The actual pulse on a waveform viewer may spread more than two picoseconds, however the standard logic pulse is positioned in the middle region of the actual pulse with its peak at mid point. The two picoseconds pulse width was used as the switching time of a typical junction. The timing parameters are computed using a function that receives the bias voltage of the cell and returns its propagation delay. Other parameters are then calculated using the propagation delay. Thus, a function for each cell was developed. In VHDL, these functions were placed in a user-defined package which is invoked with '**use**' directive in each design description file. The Verilog HDL functions were placed in a Verilog-type file which is invoked in each design using the **'include** directive.

The mathematical model for delay can be in the form of either linear, nth order polynomial, power, exponential or any other complex function. The models used in this work are polynomial, power and exponential functions. These were selected based on the goodness of fit in the delay versus bias voltage curve as discussed in chapter 3 section 3.2.2. The mathematical functions were implemented in VHDL and Verilog HDL by suitable language operators. While equivalent operators for all the necessary operations were found in VHDL, Verilog HDL (2001) lacks the exponential operator. The constant e rounded to 12 decimal places and the power operator were used. The models have been developed such that the logic level designer can simulate the circuit operation for an arbitrary value of bias voltage within the bias margins of the cell with the narrowest margins.

Listing 4.9: A function to compute delay for a DFF Verilog HDL model

```
1 function real rsfq_dff_delay(input real vbias);
2  begin     //delay in picoseconds
3    rsfq_dff_delay=(3.363e-11*vbias**(-0.7535)-4.99e-13)*1.0e12;
4  end
5 endfunction
```

Listing 4.10: A function to compute delay for DFF VHDL model

```
1  library ieee;
2  use IEEE.math_real.all;
3  package rsfq_pkg is
4     function compute_dff_delay (v :real) return real;
5  end;
6  package body rsfq_pkg is
7    function compute_dff_delay (v : real ) return real is
8      variable delay:real;
9      begin
10        delay := 3.363e-11*v**(-0.7535) - 4.99e-13;
11        return delay;
12     end function;
13 end package body;
```

A D-Flip-Flop delay functions is used as an example. Listing 4.9 is the Verilog HDL implementation of the function. As shown in line 1, a function named rsfq_dff_delay receives the value of the bias voltage and returns the delay of the cell which is computed by the expression in line 3. The VHDL version of the code is shown in Listing 4.10.

The use of the functions in building the HDL model is demonstrated by the DFF models described in Listing 4.11 and 4.12 for Verilog HDL and VHDL respectively.

Listing 4.11: Verilog HDL DFF model using std_logic pulse encoding method

```
1  `timescale 1ps/100fs
2  module dff(din,clk,outp);
3  input din,clk;
4  output outp;
5  reg outp;
6  reg din_int = 1'b0;
7  real vbias = 2.5;
8  real dl,i2c,c2i;
9  real din_last_change;
10 `include "celldelay.v"
11 always@(posedge din) begin
12   if(din_int == 0)
13     din_last_change = $realtime;
14   din_int = (din_int | din);
15 end
16 always@(posedge clk) begin
17   dl = rsfq_dff_delay(vbias);
18   c2i=0.5*dl;
19   i2c=0.0;
20   if(($realtime - din_last_change)>=i2c)
21     begin
22       outp<=#(dl)din_int;
23       din_int <= #(c2i) 1'b0;
```

```
24          outp<=#(dl+2) 1'b0;
25        end
26      else
27        outp<=#(dl)1'b0;
28  end
29  endmodule
```

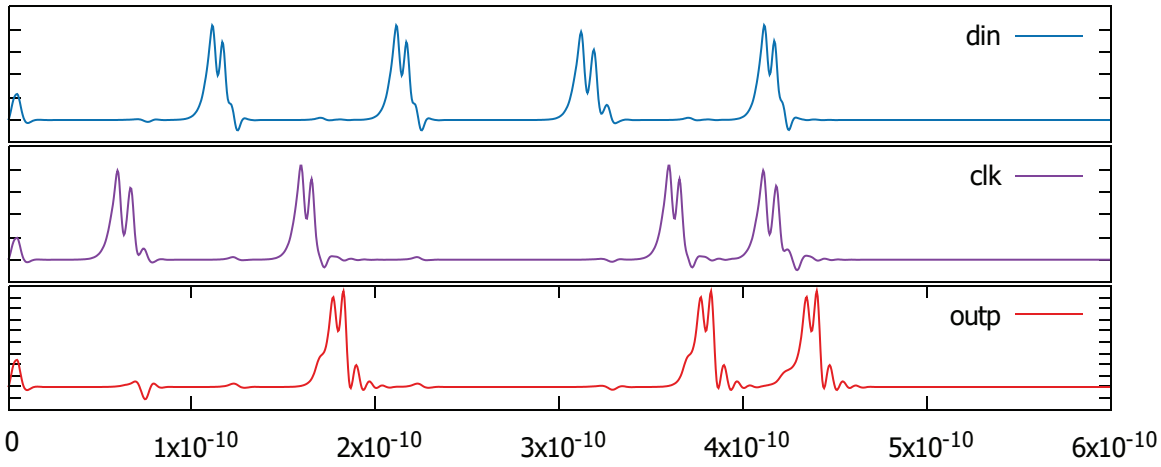
In line 10, the file containing delay function is specified using the include directive. The input pulse is latched in lines 11 - 15, and the delay of the cell is computed by calling the delay function in line 17. The delay value returned is then used in line 18 and 19 to compute the clock-to-input and input-to-clock timings. If the i2c time is not violated the latched value is transferred to the output, as shown in line 22. The output is set to logic '0' after 2 ps and the internal state is then cleared after the c2i time implemented by lines 24 and 23 respectively.

Listing 4.12: VHDL DFF model using std_logic pulse encoding method

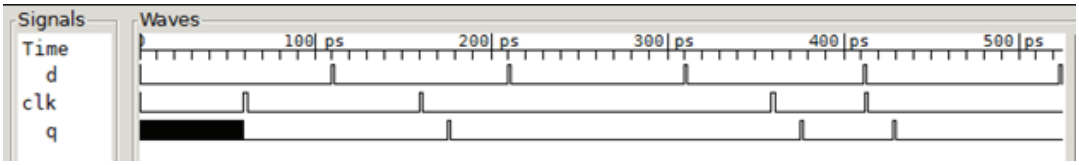
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use WORK.rsfq_pkg.all;
4  entity dffpul is
5   generic (vbias: real := 2.5);
6   port(din, clk:in std_logic;
7        dout:out std_logic);
8  end dffpul;
9  architecture bhv of dffpul is
10  signal din_int : std_logic:='0';
11  begin
12   P1: process(din,clk) is
13    variable delay: time;
14    variable i2c : time := 0 ps;
15    variable c2i : time := 0 ps;
16    begin
17     delay := compute_dff_delay(vbias)*1 sec;
18     i2c := 0 ps;
19     c2i := 0.5*compute_dff_delay(vbias)*1 sec;
20     if((din'event) and (din='1') )then
21      din_int <= din_int or din;
22     end if;
23     if (clk'event and clk='1') then
24      if(now - din_int'last_event >= i2c) then
25        dout <= transport din_int after delay;
26        din_int <= transport '0' after c2i;
27        dout <= transport '0' after delay + 2 ps;
28      else
29        dout <= '0' ;
30      end if;
31     end if;
```

```
32      end process;
33  end bhv;
```

Listing 4.12 is the VHDL equivalent of Listing 4.11



(a)



(b)

Figure 4.10: Simulation waveforms for D-Flip-Flop biased at 2.5mV, with SFQ pulse encoded as std_logic pulse (a) Circuit simulation with JSIM (b) HDL simulation with FreeHDL and iVerilog

Figure 4.10 shows the circuit simulation waveforms obtained using JSIM in (a), and the simulation waveforms of the HDL DFF models obtained using FreeHDL and Iverilog in (b). From the results, it can be observed that the HDL model accurately reproduces the behavior of the modeled circuit.

To build the second set of the HDL models of RSFQ cells, the SFQ pulse was encoded as a signal event that is described in 4.3.2. The timing constraints and the functions to

compute the cell propagation delay remain the same as the ones used in the pulse encoded method. For uniformity, the method is demonstrated using DFF HDL models as examples.

Listing 4.13: A DFF Verilog HDL model using event encoding method

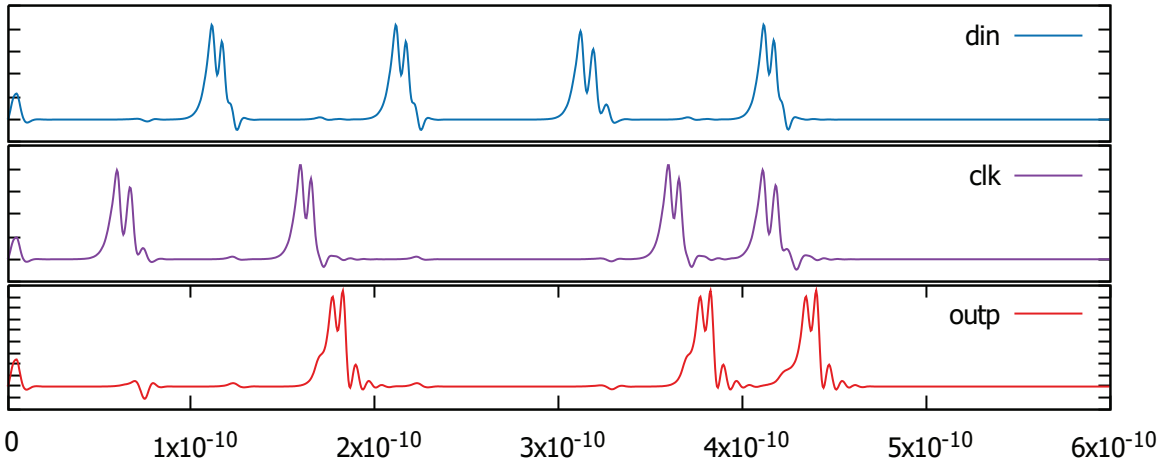
```
1  `timescale 1ps/100fs
2  module dff(din,clk,outp);
3  input din,clk;
4  output outp;
5  reg outp;
6  reg m= 1'b0,din_int = 1'b0;
7  real vbias = 2.5;
8  real dl,i2c,c2i,t;
9  `include "celldelay.v"
10
11 always@(din) begin
12   if(din_int == 0) begin
13      t = $realtime;
14      din_int = 1'b1;
15   end
16 end
17 always@(clk) begin
18   dl = rsfq_dff_delay(vbias);
19   c2i=0.5*dl;
20   i2c=0.0;
21   if(($realtime - t)>=i2c && (din_int == 1'b1))
22     begin
23        outp<=#(dl)~m;
24        din_int <= #(c2i) 1'b0;
25        m<=~m;
26     end
27 end
28 endmodule
```

Listing 4.14: Verilog HDL DFF model using event encoding method

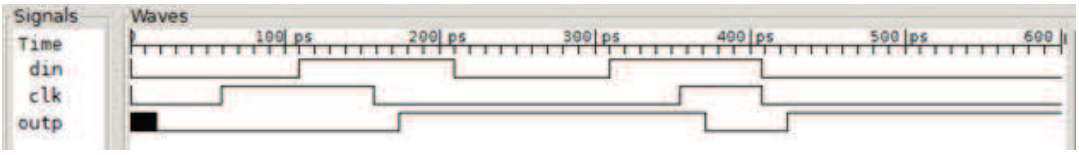
```
1  library ieee;
2  use  ieee.std_logic_1164.all;
3  use WORK.rsfq_pkg.all;
4
5  entity dffpul is
6    generic (vbias: real := 2.5);
7    port(din, clk:in std_logic;
8              dout:out std_logic);
9  end dffpul;
10 architecture bhv of dffpul is
11    signal din_int : std_logic:='0';
12 begin
13    P1: process(din,clk) is
14           variable t,i2c,c2i : time := 0 ps;
```

```
15              variable delay: time;
16              variable m: std_logic:='0';
17          begin
18                delay := compute_dff_delay(vbias)*1 sec;
19                i2c := 0 ps;
20                c2i := 0.5*compute_dff_delay(vbias)*1 sec;
21                if((din'event) and (din_int ='0') )then
22                        din_int <= '1';
23                        t:=now;
24                end if;
25              if (clk'event and din_int='1' and (now-t)>= i2c) then
26                         dout <= transport (not m) after delay;
27                         m:=not m;
28                         din_int <= transport '0' after c2i;
29                    end if;
30          end process;
31  end bhv;
```

Listings 4.13 and 4.14 represent the verilog HDL and VHDL models respectively. The major difference between these models and the pulse-based models can be seen in lines 23 and 25 of the Verilog code and lines 26 and 27 of the VHDL code. These lines indicate that the output toggles every time the output is logic '1'.



(a)

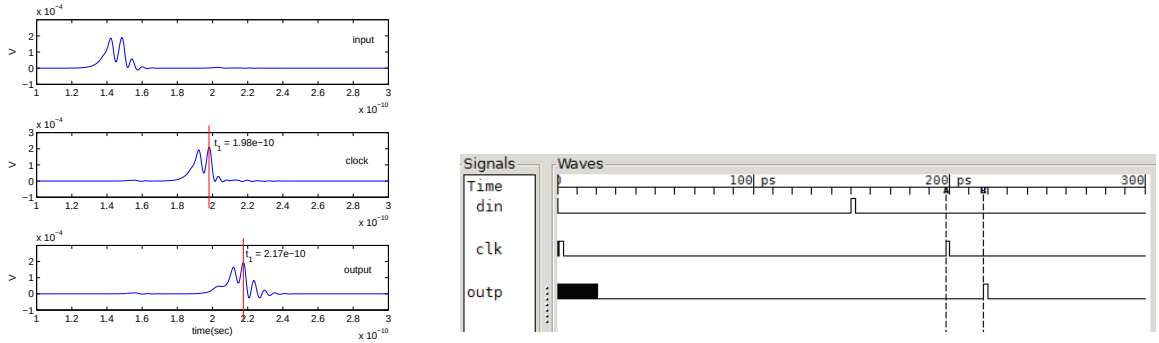


(b)

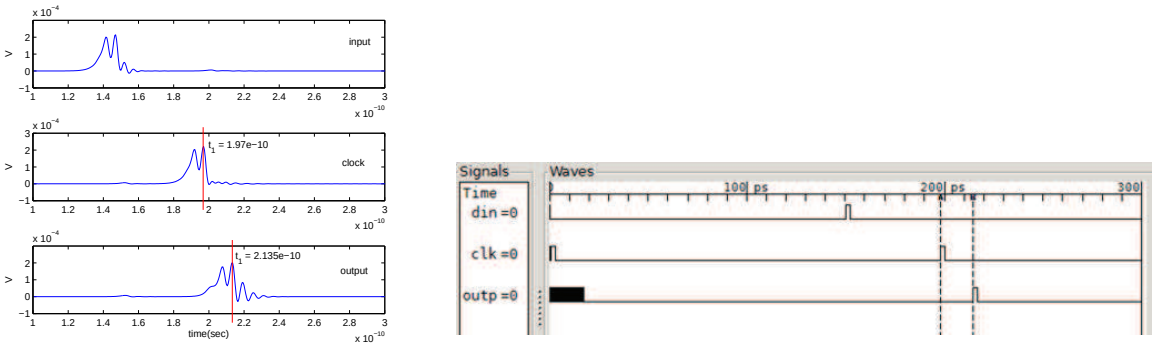
Figure 4.11: Simulation waveforms for D-Flop-Flop biased at 2.5mV, with SFQ encoded as a an event (a) Circuit simulation with JSIM (b) HDL simullation with FreeHDL and iVerilog

The visualization of the model simulation is given in figure 4.11 (b) alongside the circuit simulation waveform for easier comparison. This method eliminates an event for every data pulse and clock pulse in the previous method, which cuts down the simulation time. However, the analysis of the model's input/output relation is not straight forward as the SFQ pulse is represented by either a transition from zero to one or vice versa.

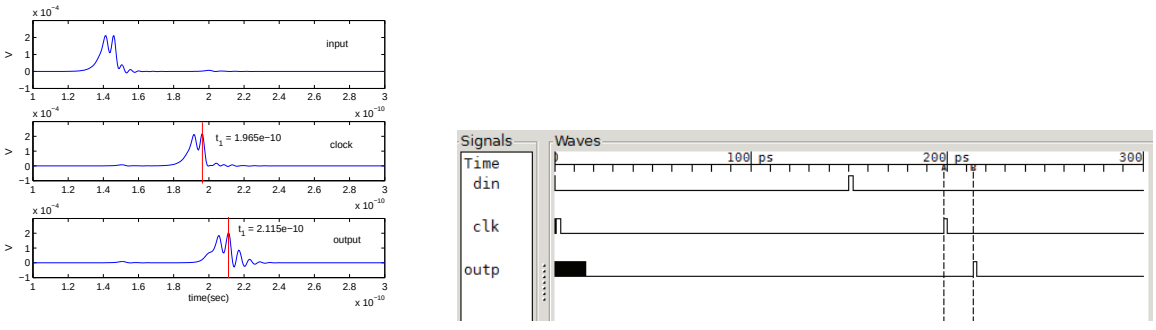
To demonstrate the ability of the HDL simulation to track the JSIM simulation in terms of bias-depended timing, a DFF cell was simulated at various bias voltages. In this section three of them are reported (2 mV, 2.5 mV and 2.8 mV) which were arbitrarily selected. The simulation results of the three cases are shown in Figure 4.12, where JSIM waveforms are



(a) JSIM and HDL simulation waveforms for a DFF biased at 2 mV, pulse delay from JSIM simulation $t_2 - t_1 = 2.17e-10 - 1.98e-10 = 19$ ps , the delay observed from HDL simulation is $B - A = 217.4 - 198 = 19.4$ ps



(b) JSIM and HDL simulation waveforms for a DFF biased at 2.5 mV, pulse delay from JSIM simulation $t_2 - t_1 = 2.135e-10 - 1.97e-10 = 16.5$ ps , the delay observed from HDL simulation is $B - A = 214.5 - 198 = 16.5$ ps



(c) JSIM and HDL simulation waveforms for a DFF biased at 2.8 mV, pulse delay from JSIM simulation $t_2 - t_1 = 2.115e-10 - 1.965e-10 = 15$ ps , the delay observed from HDL simulation is $B - A = 213 - 198 = 15$ ps

Figure 4.12: JSIM and HDL simulation waveforms for a DFF at bias voltages 2 mV, 2.5 mV and 2.8 mV.

placed alongside HDL simulation for each bias case. The pulse delay of the cell is computed from the JSIM waveforms by taking the time difference between t_2 and t_1 and in HDL waveforms by taking the time difference between rising edges at B and A. The observed delay at 2.5 mV and 2.8 mV are equal but at 2.0 mV the delay in the HDL simulation exceeds its counterpart by 0.4 ps. The results indicated that HDL simulations track the JSIM simulations in terms of bias depended timings with high accuracy.

4.5 Conclusion

The main goal of performing the task presented in this chapter is to build HDL cell libraries for SFQ circuits. From the review carried out on the popular cell libraries, three SFQ information encoding methods were compiled. Using these methods AQFP and RSFQ HDL cell libraries were developed with Verilog HDL and VHDL. From the findings of this study, the behavior of a large AQFP circuit can now be simulated using HDL logic simulator. SU cell library can now be used to design logic circuits using HDLs. The findings not only adds to the growing body of literature on HDL modeling of SFQ circuits, but also serves as the basis for future studies on modeling AQFP circuits with HDLs.

Chapter 5

Superconducting Digital Circuit Synthesis

5.1 Introduction

Synthesis is the heart of the automated design process of digital VLSI circuits. It transforms a behavioral-domain design model into a structural domain model, and structural-domain model into a physical-domain model revealing more circuit details. Thus a design can be modeled as behavior description with HDLs and synthesized to its structural equivalent and further to its physical equivalent as required by the designer if tools permit. While synthesis can be performed manually, interactively or automatically, the term synthesis is used here to refer to the automated synthesis process.

Synthesis in semiconductor technology has been thoroughly researched and practiced, and tremendous improvement has been made in productivity, correctness, re-targetability and design time-to-market. However, synthesis is the weak link in the superconducting digital circuit design process. Little has been done in logic synthesis of superconducting circuits [76, 57, 56] at the time of writing, but there are no reports on high-level behavior synthesis. In this chapter a behavior synthesis method for SCE digital circuits is presented. The major question addressed here is if a combinational logic circuit behavioral description can be synthesized to a superconducting logic circuit netlist with the help of the existing free synthesis tools in the semiconductor technology. The method targets synthesis of Verilog HDL models of combination logic circuits utilizing the Yosys synthesis framework. The method was implemented and successfully tested with circuits of different complexity.

5.2 General high-level synthesis flow

The three domains used to describe hardware are physical, structural and behavioral [77]. In each domain there are levels of abstraction. In the behavior domain, the circuit behavior

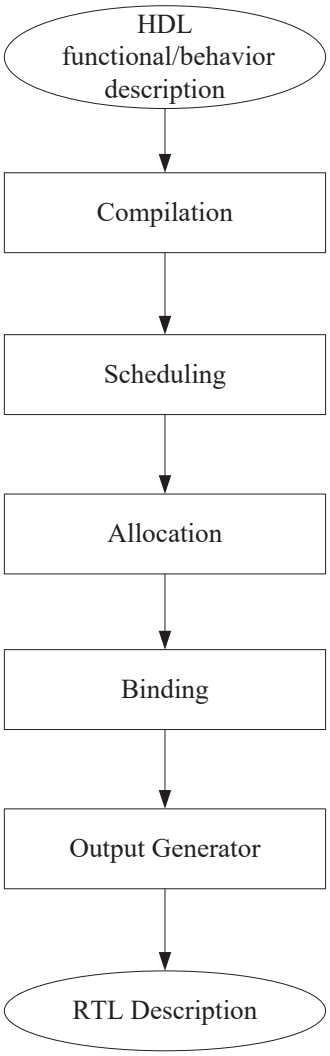


Figure 5.1: General high-level behavior synthesis flow

is described in Boolean expressions, finite state machines (FSM) or algorithmic levels of abstraction. FSM and algorithm description are generally referred to as high-level behavior description. High-level synthesis (HLS) is the transformation of the high-level behavior description of a design to its equivalent model in structural domain while satisfying the functionally and the specified constraints. A typical high-level synthesis process is shown in figure 5.1.

HLS starts with a behavior description of the circuit using a HDL. The description, containing operations, assignment statements and control constructs, is taken to the HLS process after it has been simulated and verified to operate as required.

In the compilation step, the compiler-based transformations are carried out. Other than syntax checking and semantic checking, the code is also optimized to reduce the resulting circuit modules, thus reducing circuit area and consequently reducing power consumption while maximizing the operation frequency. The HDL design description is compiled into an internal form of representation, typically a control and data flow graph (CDFG).

In the scheduling step, the execution sequence of the operations is determined and a control step schedule is produced. This schedule indicates the operations per control step. The scheduling problem could be unconstrained, time constrained, resource constrained, or time-resource constrained.

The allocation task starts once scheduling is complete. In this step, the operator modules and storage elements are selected from the design library to be used in operator and variable implementations respectively. A design library may contain alternatives for a certain operator or function units, however the allocation algorithm selects the best module based on the design constraints and optimization objectives.

The previous steps provide the information regarding the actual circuit modules in the design library which can be used. In order to generate the register-transfer level (RTL) netlist, the operators and variables are assigned to the hardware modules in the binding step. The functional units are assigned to operator modules; the input, output and variables are assigned to register units, and the data transfers are assigned to interconnection units.

The RTL netlist of the circuit is generated with the modules and the interconnections defined. A typical high level synthesis process ends with a RTL netlist but it can be refined to the lower level of abstraction within the structural domain such as gate level.

5.3 The Yosys synthesis tool

Yosys is a synthesis framework [78] for verilog HDL design descriptions. It supports a large subset of verilog 2005 and has been verified to support large circuits such as the Open-MSP430 microcntroller. Unlike Odin II [79] and vtr (verilog to routing) [80], Yosys is made suitable for both FPGA-based designs and ASIC designs. A simplified data flow for synthesis in Yosys [81] is shown in the Figure 5.2. Currently, Yosys has Verilog frontend and inter-

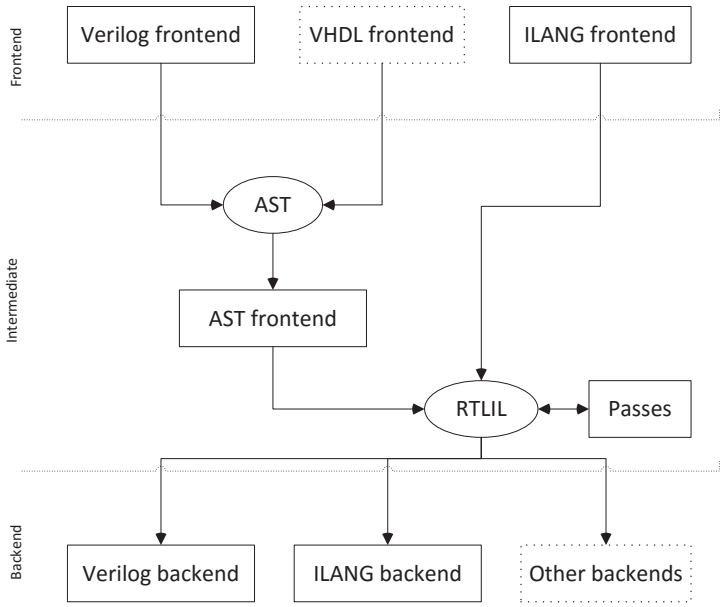


Figure 5.2: Simplified Yosys synthesis data flow

mediate language (ILANG) frontend implemented. The Verilog frontend receives the design description and generates an abstract syntax tree (AST). The same AST representation will also be used by VHDL frontend once it is implemented. The AST frontend then compiles the AST into the RTL intermediate language (RTLIL) format. With a series of passes on the RTLIL design representation, a final representation is generated, which becomes the input of the backends. The backends then convert the RTLIL data to text as Verilog netlist, ILANG compatible RTLIL data or other supported formats. In this thesis, the most attractive part of this tool is that except for the AST frontend, all the other program modules are invoked by the user. This gives us the freedom to selectively call the modules that satisfy our needs and in case the existing implementation is not sufficient for the required task, it is easy to locate the section of the program that can be modified by extending the Yosys base class. In this work, Yosys modules were selectively called to generate a structural design netlist which was used as the input to our program. Our tool extension then transforms this netlist to its SCE equivalent.

5.4 The Synthesis method

In order to start the synthesis, a custom liberty file is prepared containing the definition of the cells in the RSFQ cell library with inputs, output and the function of cells using CMOS notation. This file is used by the ABC tool within the Yosys environment when performing the mapping to ensure that the resulting circuit modules have their corresponding

Listing 5.1: Yosys synthesis script for a full adder design with module name FA

```
read_verilog work/fullAdder.v
hierarchy -check -top FA
proc; opt
techmap; opt
flatten FA
abc -liberty techlibs/maguu/mycells.lib
techmap; opt
clean
write_verilog fullAddernet.v
```

representation in the RSFQ cell library. Another necessary file is the delay file extracted from the cell library. This file is used in the computation of clock required arrival time (RAT) and clock arrival time (CAT). The synthesis is performed in two phases, a CMOS-like synthesis phase and the transformation phase where the netlist is converted to the RSFQ equivalent. The CMOS-like phase utilizes the built-in Yosys features to generate an intermediate design netlist while in the transformation phase, a custom program was written that takes the intermediate design netlist as its input and generates its RSFQ equivalent. The following steps summarize the synthesis process using the Yosys framework.

1. Read the HDL description into the synthesis tool
2. Synthesize to internal Yosys cell library
3. Invoke ABC from Yosys to map the design to the custom cell library
4. Generate the CMOS-like structural netlist in Verilog HDL
5. Insert data splitters to handle the fan-out of the RSFQ cells
6. Compute the RAT in each logic level
7. Generate the clock tree for the logic level such that CAT $>$ RAT
8. Insert latches at primary output
9. Generate the final Verilog HDL netlist
10. Post synthesis simulation

The first four steps are performed by Yosys with a synthesis script or by typing Yosys commands at the prompt. A script example is shown in listing 5.1. The **read_verilog** command loads the specified file into the Yosys environment. In order to identify the inter-modules communication the design hierarchy is extracted by specifying the top module to **hierarchy** command. The **proc** command then translate all the processes in the design. It is recommended that after every command that performs any sort of transformation in Yosys, an optimization is carried out. The **opt** command is used to perform design optimization.

Listing 5.2: Verilog HDL behaviral description of a single bit full adder

```
module FA(s,cout,cin,x,y);
  input y,x,cin;
  output s,cout;
  assign {cout,s}=x+y+cin;
endmodule
```

The design is then mapped to the internal Yosys logic library with the **techmap** command. The **flatten** command is then called to elaborate the module instances so that the design is viewed as a single module. The **abc** command is then used to map the design to the custom cell library. To ensure that any dangling connections are eliminated, the **clean** command is executed. The Yosys Verilog backend writes a Verilog HDL netlist when the **write_verilog** command is executed. The first phase ends with a verilog netlist consisting of instances of the cells defined in the custom cell library. As an example, the full adder design shown in Listing 5.2 passed through the first phase of results in the code in Listing 5.3. A schematic

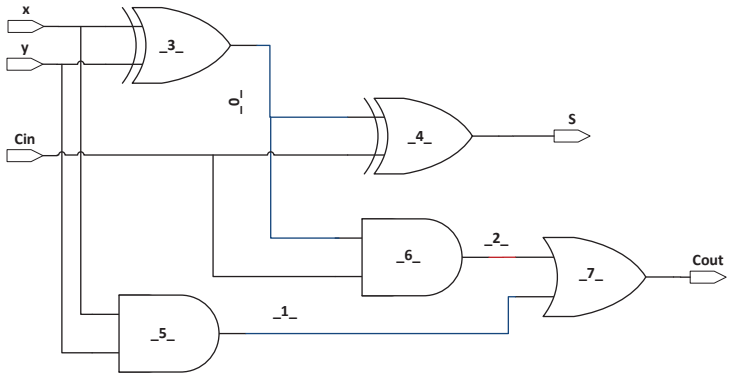


Figure 5.3: Full adder schematic equivalent to the generated nelist in listing 5.3

equivalent to the generated netlist is shown in Figure 5.3. The resulting circuit is a multilevel logic with the primary inputs x, y, and Cin and primary output S and Cout as defined in the HDL module. This is the CMOS representation of the design optimized to a custom cell library. While the gates in the resulting circuit are logically equivalent to those in the RSFQ cell library, the circuit needs to be transformed to accommodate the clock and handle the fan out issue.

In the second phase, the netlist generated by Yosys is taken through a series of transformations in order to obtain the RSFQ equivalent netlist. First, the netlist is cleaned so that it contains only the structural code for the design. The names of inputs and wires are extracted and their number of occurrences as gate's input is computed for each. This gives the number of times that each signal should be split. The splitting results in new signals whose names are then added as wire declarations in the netlist. The new signals are then

Listing 5.3: Yosys synthesis output as a Verilog netlist

```
/* Generated by Yosys 0.5 (git sha1 UNKNOWN, clang 3.6.0-2ubuntu1 -fPIC -Os)
    */
(* top = 1 *)
(* src = "work/fullAdder.v:1" *)
module FA(s, cout, cin, x, y);
  wire _0_;
  wire _1_;
  wire _2_;
(* src = "work/fullAdder.v:2" *)
  input cin;
(* src = "work/fullAdder.v:3" *)
  output cout;
(* src = "work/fullAdder.v:3" *)
  output s;
(* src = "work/fullAdder.v:2" *)
  input x;
(* src = "work/fullAdder.v:2" *)
  input y;
  XOR _3_ (
    .A(x),
    .B(y),
    .Y(_0_)
  );
  XOR _4_ (
    .A(_0_),
    .B(cin),
    .Y(s)
  );
  AND _5_ (
    .A(x),
    .B(y),
    .Y(_1_)
  );
  AND _6_ (
    .A(_0_),
    .B(cin),
    .Y(_2_)
  );
  OR _7_ (
    .A(_2_),
    .B(_1_),
    .Y(cout)
  );
endmodule
```

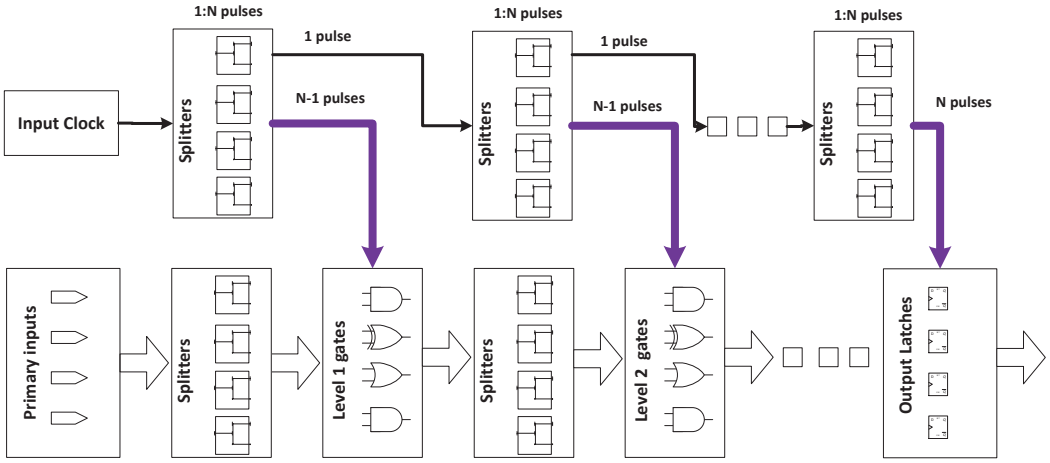


Figure 5.4: Data and Clock splitting illustration

used to replace the original signal's occurrences at the inputs of the gates. In cases where the signal occurs once, a JTL is inserted before the input terminal of a gate.

Once the signal is split, the clock signal for each gate is generated. A concurrent flow of the clock and data signal method [4] was deployed in synthesizing the clock in this work. Figure 5.4 illustrates the implementation of clock and data in the synthesis process. The number of clock signals required (N) in each logic level is equal to the number of the gates in that logic level plus one which propagates to the following level. If it is the last logic level then N is equal to the number of primary outputs as the clock pulses will be used to trigger the output latches. The primary clock input signal is split into N where N-1 pulses are for the gates and one pulse for the next level as shown in the figure. For the logic level to operate accurately, a timing analysis is performed. The time taken by each data pulse to arrive at the gate input is computed. The longest path defines the clock required arrival time (RAT). At the first logic level, RAT is the sum of propagation delays of splitters and JTL in the signal path (T_{psj}) and minimum input-to-clock ($i2c$) time for that gate receiving the signal ($RAT = T_{psj} + i2c$). In the other levels, the propagation delay (P_{dg}) of the gate generating the signal in the previous level is added ($RAT = T_{psj} + i2c + P_{dg}$). The time taken by a clock pulse to arrive at the gate i.e the clock arrival time (CAT) is also computed for each line. CAT is the pulse delay introduced by the clock splitting circuits. The CAT for each pulse is then compared with the RAT. If the CAT is less than the RAT, a delay element is added in the clock path, in this case a JTL was used. The new CAT is computed and again compared with the RAT. This is repeated until the CAT is greater than or equal to the RAT .

Once the timing for the clock signals is adjusted such that the RAT $<$ CAT in the logic levels, the clock signals are added to the gates' instances in the netlist and the name of the gate is changed to the equivalent RSFQ gate's name in the cell library. The resulting netlist

contains only instances of RSFQ cell descriptions.

Listing 5.4: The netlist obtained from the synthesis process

```
module FA(k, s, cout, cin, x, y);
input k;
  wire _0_;
  wire _1_;
  wire _2_;
  input cin;
  output cout;
  output s;
  input x;
  input y;
wire _0_0;
wire _0_1;
wire _0_00;
wire _0_10;
wire _1_0;
wire _2_0;
wire cin0;
wire cin1;
wire cin00;
wire cin10;
wire x0;
wire x1;
wire x00;
wire x10;
wire y0;
wire y1;
wire y00;
wire y10;
wire k0;
wire k1;
wire k00;
wire k10;
wire k000;
wire k001;
wire k0000;
wire k0010;
wire k100;
wire k1000;
wire k10000_;
wire k10000_0;
wire k10000_1;
wire k10000_00;
wire k10000_10;
wire k10000_000;
wire k10000_001;
wire k10000_0000;
wire k10000_0010;
wire k10000_100;
wire k10000_1000;
wire k10000_10000_;
wire k10000_10000_0;
wire k10000_10000_1;
wire k10000_10000_00;
```

```
wire k10000_10000_10;
wire k10000_10000_000;
wire cl;
wire cl0;
wire cl1;
wire cl00;
wire cl10;
wire cl000;
wire cl0000;
wire cl100;
wire cl1000;
wire ot0;
wire ot1;
SPL P_0_(.in_pulse(_0_),.out_pulse1(_0_0),.out_pulse2(_0_1));
JTL J_0_0(.in_pulse(_0_0),.out_pulse(_0_00));
JTL J_0_1(.in_pulse(_0_1),.out_pulse(_0_10));
JTL J_1_(.in_pulse(_1_),.out_pulse(_1_0));
JTL J_2_(.in_pulse(_2_),.out_pulse(_2_0));
SPL Pcin(.in_pulse(cin),.out_pulse1(cin0),.out_pulse2(cin1));
JTL Jcin0(.in_pulse(cin0),.out_pulse(cin00));
JTL Jcin1(.in_pulse(cin1),.out_pulse(cin10));
SPL Px(.in_pulse(x),.out_pulse1(x0),.out_pulse2(x1));
JTL Jx0(.in_pulse(x0),.out_pulse(x00));
JTL Jx1(.in_pulse(x1),.out_pulse(x10));
SPL Py(.in_pulse(y),.out_pulse1(y0),.out_pulse2(y1));
JTL Jy0(.in_pulse(y0),.out_pulse(y00));
JTL Jy1(.in_pulse(y1),.out_pulse(y10));
SPL Pk(.in_pulse(k),.out_pulse1(k0),.out_pulse2(k1));
JTL Jk0(.in_pulse(k0),.out_pulse(k00));
JTL Jk1(.in_pulse(k1),.out_pulse(k10));
SPL Pk00(.in_pulse(k00),.out_pulse1(k000),.out_pulse2(k001));
JTL Jk000(.in_pulse(k000),.out_pulse(k0000));
JTL Jk001(.in_pulse(k001),.out_pulse(k0010));
JTL Jk10(.in_pulse(k10),.out_pulse(k100));
JTL Jk100(.in_pulse(k100),.out_pulse(k1000));
JTL Jk1000(.in_pulse(k1000),.out_pulse(k10000_));
SPL Pk10000_(.in_pulse(k10000_),.out_pulse1(k10000_0),.out_pulse2(k10000_1));
JTL Jk10000_0(.in_pulse(k10000_0),.out_pulse(k10000_00));
JTL Jk10000_1(.in_pulse(k10000_1),.out_pulse(k10000_10));
SPL Pk10000_00(.in_pulse(k10000_00),.out_pulse1(k10000_000),.out_pulse2(k10000_001));
JTL Jk10000_000(.in_pulse(k10000_000),.out_pulse(k10000_0000));
JTL Jk10000_001(.in_pulse(k10000_001),.out_pulse(k10000_0010));
JTL Jk10000_10(.in_pulse(k10000_10),.out_pulse(k10000_100));
JTL Jk10000_100(.in_pulse(k10000_100),.out_pulse(k10000_1000));
JTL Jk10000_1000(.in_pulse(k10000_1000),.out_pulse(k10000_10000_));
SPL Pk10000_10000_(.in_pulse(k10000_10000_),.out_pulse1(k10000_10000_0),.out_pulse2(
    k10000_10000_1));
JTL Jk10000_10000_0(.in_pulse(k10000_10000_0),.out_pulse(k10000_10000_00));
JTL Jk10000_10000_1(.in_pulse(k10000_10000_1),.out_pulse(k10000_10000_10));
JTL Jk10000_10000_00(.in_pulse(k10000_10000_00),.out_pulse(k10000_10000_000));
JTL Jk10000_10000_10(.in_pulse(k10000_10000_10),.out_pulse(cl));
SPL Pcl(.in_pulse(cl),.out_pulse1(cl0),.out_pulse2(cl1));
JTL Jcl0(.in_pulse(cl0),.out_pulse(cl00));
JTL Jcl1(.in_pulse(cl1),.out_pulse(cl10));
JTL Jcl00(.in_pulse(cl00),.out_pulse(cl000));
JTL Jcl000(.in_pulse(cl000),.out_pulse(cl0000));
```

```
JTL Jcl10(.in_pulse(cl10),.out_pulse(cl100));
JTL Jcl100(.in_pulse(cl100),.out_pulse(cl1000));
  RSFQXOR _3_ (
    .clk(k0000),
    .A(x00),
    .B(y00),
    .Y(_0_)
  );
  RSFQXOR _4_ (
    .clk(k10000_0000),
    .A(_0_00),
    .B(cin00),
    .Y(ot1)
  );
  RSFQAND _5_ (
    .clk(k0010),
    .A(x10),
    .B(y10),
    .Y(_1_)
  );
  RSFQAND _6_ (
    .clk(k10000_0010),
    .A(_0_10),
    .B(cin10),
    .Y(_2_)
  );
  RSFQOR _7_ (
    .clk(k10000_10000_000),
    .A(_2_0),
    .B(_1_0),
    .Y(ot0)
  );
RSFQDFF DFF0(.in_pulse(ot0), .clk(cl0000), .out_pulse(cout));
RSFQDFF DFF1(.in_pulse(ot1), .clk(cl1000), .out_pulse(s));
endmodule
```

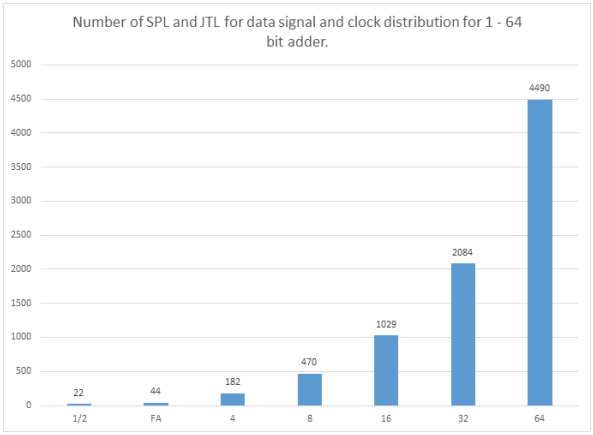


Figure 5.5: Number of Splitters and JTLs inserted for data splitting and clock distribution and adder data path increases

Listing 5.4 shows the resulting netlist after synthesis of the full adder described in Listing 5.2. A post synthesis simulation using Icarus Verilog was performed to verify the netlist.

It is noted that most of the code generated results from data splitting and clock distribution. The graph in the figure 5.5 shows the growth in total number of splitters and JTL starting from single bit half adder to 64 bit adder.

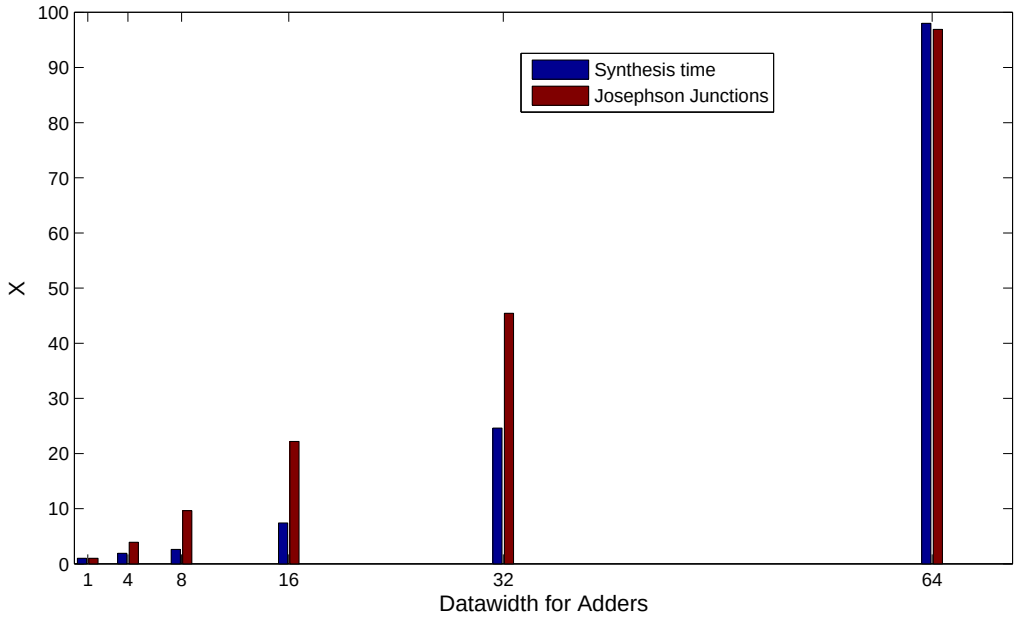


Figure 5.6: The growth of synthesis time and number of Josephson Junctions with respect to adder's datawidth. X is the number of times that of 1 bit adder.

As the data width for adder increased, there was a significant increase in synthesis time. Figure 5.6 shows the growth of synthesis time and the number of Josephson Junctions as the data width of the adder increases from one bit to 64 bits. 1x for synthesis time is 0.232 seconds (time taken to synthesize one bit adder) and 1x for JJ is 174 (the number of JJs in one bit adder). The increase in the number of Josephson Junctions is approximately linear while that of synthesis time is approximately quadratic. While this suggest that more computational time is required for wide adders, it is not a major challenge because the average synthesis time of a 64 bit adder was 23 sec.

It was also found out that the clock pulses do not arrive at the gates in one level at the same time. However, the difference is less than JTL's delay.

5.5 Conclusion

In this chapter, a synthesis method has been presented. The method was developed for combination logic Verilog HDL design. It was implemented using Yosys synthesis framework

partially and C++ programs. Using this method a combinational logic design described in Verilog HDL can be synthesized to RSFQ equivalent structural netlist. Various designs, among them adders of different data width, were tested and found to give valid results. Most of the circuit netlist was found to contain instances of splitter and JTL, however the sizes of these circuits is considerably smaller than the other logic cells. It was also noted that clock pulses do not arrive in all the gates in a logic level at the same time. The time difference can be reduced by having multiple JTLs in the cell library to provide different delay values. With well characterized cell library, a designer with little or no superconducting electronic knowledge can design logic circuits for superconducting circuit technology.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

The design automation of superconducting logic circuit is an important step towards the maturity of the technology. This dissertation - HDL modeling and synthesis of superconducting logic circuits - describes the methods used to build HDL models of SCE logic circuits and a synthesis method. The implementation and testing of the methods were carried out with free software tools either by adopting or adapting them.

To obtain suitable tools for this work, available free software tools in the the semiconducting technology and superconducting technology were evaluated. Based on a design flow proposed herein, a complete toolchain was formulated for superconducting circuits design. While this toolchain is complete for a small scale circuit, there are missing tools for medium- and large-scale circuits.

In order to facilitate medium- and large-scale SCE circuit design, HDL behavior models of the SFQ circuits were developed as a HDL cell library. The two types of HDL cell libraries developed were the AQFP cell library and the RSFQ cell library. The AQFP models were based on the level encoding method and did not include the timing parameters. These models, when used in large circuits demonstrated the ability to simulate large AQFP circuits at logic level for the first time. To create the RSFQ models, timing parameters were extracted. The propagation delay of cells was presented as a mathematical function of bias voltage. Thus, the models can be simulated at different bias voltages within the cell margins. The input-to-clock and clock-to-input time was then presented in terms of propagation delay. Two sets of the RSFQ HDL models were built, one based on the pulse encoding technique and the other based on event encoding technique. The models were developed such that the HDL description is not technology dependent, only the cell timing need to be changed for a different technology.

A synthesis method was proposed and demonstrated for Verilog HDL behavior description of combination logic circuits, based on the RSFQ HDL cell library. This method partially utilizes the Yosys synthesis framework and partially the custom code. The testing

was carried out with several combination logic descriptions, however attention was paid to adder circuits. Due to concern in the increase of interconnecting circuits during synthesis, a growth in the total number of Splitter and JTL cells added was observed. As the data width for the adder circuit increases from one bit to 64 bits there is an exponential growth of interconnection cells.

This research serves as the base for the future of the top-down design of SFQ circuits using free software tools. Moreover, the findings from this study show that designers with little or no knowledge of circuit level SFQ design can now design SFQ circuits using HDLs.

6.2 Benefit of the research

This research centred on HDL modeling and synthesis of SFQ circuits using free software tools is a valuable resource to novice researchers and budget-constrained research groups in SCE. It demonstrates that the cost of commercial tools, which are generally expensive, is not an obstacle towards exploring the SCE technology.

The use of a bias voltage-based mathematical function for the delay in RSFQ cells, allows the logic simulation of the circuit at arbitrarily bias voltage within the circuit bias margins. The AQFP HDL modeling approach provided the base line for moving to the HDL based design of AQFP circuits.

The synthesis method demonstrated in this work is the first to be documented for behavior synthesis application in SCE. It is an intuitive method that lays the foundation of behavior synthesis research.

6.3 Future work

The AQFP models should include the timing parameters so that accuracy in timing during simulation can be verified. The synthesis method demonstrated employed concurrent clock flow and other clocking methods such as counter flow should be implemented and the size of splitting circuits compared. Moreover synthesis of synchronous circuits behavior description needs to be addressed.

Bibliography

[1] "Power a major hurdle on road to exascale." [Online]. Available: http://www.scientificcomputing.com/articles/2011/10/power-major-hurdle-road-exascale

[2] Sunway taihulight. [Online]. Available: https://www.top500.org/lists/2016/06/

[3] A. Abdollahi and M. Pedram, "Efficient synthesis of quantum logic circuits by rotation-based quantum operators and unitary functional bi-decomposition," *IWSL*, 2005.

[4] 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.

[5] O. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Transactions on Applied Superconductivity*, vol. 21, no. 3, pp. 760–769, 2011.

[6] Q. P. Herr, A. Y. Herr, O. T. Oberg, and A. G. Ioannidis, "Ultra-low-power superconductor logic," *Journal of Applied Physics*, vol. 109, no. 10, p. 103903, 2011.

[7] N. Takeuchi, D. Ozawa, Y. Yamanashi, and N. Yoshikawa, "An adiabatic quantum flux parametron as an ultra-low-power logic device," *Superconductor Science and Technology*, vol. 26, no. 3, p. 035010, 2013.

[8] Y. Kameda, S. Yorozu, and Y. Hashimoto, "A new design methodology for single-flux-quantum (sfq) logic circuits using passive-transmission-line (PTL) wiring," vol. 17, no. 2, pp. 508–511, 2007.

[9] W. Chen, A. V. Rylyakov, V. Patel, 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.

[10] M. Tinkam, "Introduction to superconductivity," *McGraw-Hill, Inc., New York*, vol. 93, p. 94, 1996.

[11] T. Orlando and K. Delin, *Foundations of Applied Superconductivity*. Addison-Wesley Publishing Company, Massachusetts, 1991.

[12] E. A. Lynton, *Superconductivity.* Chapman and Hall Ltd., 1969.

[13] H. Hayakawa, N. Yoshikawa, S. Yorozu, and A. Fujimaki, "Superconducting digital electronics," *Proceedings of the IEEE*, vol. 92, no. 10, pp. 1549–1563, 2004.

[14] B. D. Josephson, "The discovery of tunnelling supercurrents," *Reviews of Modern Physics*, vol. 46, no. 2, pp. 251–254, 1974.

[15] W. Perold, "Complementary output switching logic: a superconducting voltage-state logic family operating at microwave frequencies," in *Communications and Signal Processing, 1998. COMSIG '98. Proceedings of the 1998 South African Symposium on*, 1998, pp. 435–440. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=736999

[16] H. ter Brake et. al., "Scenet roadmap for superconductor digital electronics," *Physica C: Superconductivity*, vol. 439, no. 1, pp. 1–41, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921453405007586

[17] V. Kaplunenko, M. Khabipov, V. Koshelets, K. Likharev, O. Mukhanov, V. Semenov, I. Serpuchenko, and A. Vystavkin, "Experimental study of the rsfq logic elements," *Magnetics, IEEE Transactions on*, vol. 25, no. 2, pp. 861–864, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=92422

[18] K. Inoue, N. Takeuchi, K. Ehara, Y. Yamanashi, and N. Yoshikawa, "Simulation and experimental demonstration of logic circuits using an ultra-low-power adiabatic quantum-flux-parametron," *IEEE Transactions on Applied Superconductivity*, vol. 23, no. 3, 2013.

[19] J. X. Przybysz and D. L.Miller, *Applied Superconductivity- Handbook on Devices and Applications*, P. Seidel, Ed. Wiley-VCH Verlag GmbH & Co. KGaA, Boschstr. 12, 69469 Weinheim, Germany, 2015.

[20] Y. Y. N. Takeuchi and N. Yoshikawa, "Measurement of 10 zj energy dissipation of adiabatic quantum-flux-parametron logic using a superconducting resonator," *Applied Physics Letters*, 2013.

[21] R. Stallman, *Free software, free society: Selected essays of Richard M. Stallman.* Lulu.com, 2002.

[22] Synopsys, "OpenMAST," [Accessed November, 2015]. [Online]. Available: http://news.synopsys.com/index.php?item=122764

[23] K. Gaj, Q. P. Herr, V. Adler, A. Krasniewski, E. G. Friedman, and M. J. Feldman, "Tools for the computer-aided design of multigigahertz superconducting digital circuits," *IEEE Transactions on Applied Superconductivity*, vol. 9, no. 1, pp. 18–38, 1999.

[24] B. Dimov, V. Mladenov, V. Todorov, T. Ortlepp, and F. Uhlmann, "Design aspects of complex asynchronous rsfq digital circuits," in *this conference*, 2006.

[25] S. Tell, "Gwave," [Accessed July, 2016]. [Online]. Available: https://sourceforge.net/projects/gwave/

[26] GNUplot. [Online]. Available: http://www.gnuplot.info/

[27] GNU, "GNU octave," [Accessed July, 2016]. [Online]. Available: https://www.gnu.org/software/octave/

[28] S. Enterprises, "Scilab," [Accessed July, 2016]. [Online]. Available: http://www.scilab.org/

[29] M. Dorojevets, C. Ayala, and A. Kasperek, "Data-flow microarchitecture for wide datapath RSFQ processors: Design study," *IEEE Transactions on Applied Superconductivity*, vol. 21, no. 3, pp. 787–791, 2011.

[30] F. Matsuzaki, N. Yoshikawa, M. Tanaka, A. Fujimaki, and Y. Takai, "A behavioral-level HDL description of SFQ logic circuits for quantitative performance analysis of large-scale SFQ digital systems," *Physica C: Superconductivity*, vol. 392, pp. 1495–1500, 2003.

[31] "GTKwave," [Accessed July, 2016]. [Online]. Available: http://gtkwave.sourceforge.net/

[32] gEDA Project, "http://www.geda-project.org," Accessed Nov, 2015.

[33] Fritzing, "http://www.fritzing.org/home." Accessed Nov, 2015.

[34] Kcad, "http://www.kicad-pcb.org." Accessed Nov, 2015.

[35] KtechLab, "https://github.com/ktechlab/ktechlab/wiki." Accessed Nov, 2015.

[36] Xcircuit, "http://opencircuitdesign.com/xcircuit." Accessed Nov, 2015.

[37] S. Whiteley, "Josephson junctions in spice3," *IEEE Transactions on Magnetics*, vol. 27, no. 2, pp. 2902–2905, 1991.

[38] R. Jewett, "Josephson junctions in spice 2g5," *Electronics Research Lab internal memoranda, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA*, p. 94720, 1982.

[39] J. E. Fang and T. V. Duzer, "A josephson integrated circuit simulator (jsim) for superconductive electronics application," *Extended Abstracts of 2nd International Superconductive Electronics Conference*, pp. 407–410, 1989.

[40] S. Polonsky, V. Semenov, and P. Shevchenko, "Pscan: personal superconductor circuit analyser," *Superconductor Science and Technology*, vol. 4, no. 11, p. 667, 1991.

[41] A. Dewes, "A tool to simulate superconducting circuits, comparable to spice," Accessed July 2016. [Online]. Available: https://github.com/adewes/superconductor

[42] LTSpice (linear technology spice). [Online]. Available: http://www.linear.com/designtools/software

[43] Q. Herr and M. Feldman, "Multiparameter optimization of RSFQ circuits using the method of inscribed hyperspheres," *IEEE Transactions on Applied Superconductivity*, vol. 5, no. 2, pp. 3337–3340, 1995.

[44] S. Polonsky, P. Shevchenko, A. Kirichenko, D. Zinoviev, and A. Rylyakov, "Pscan'96: new software for simulation and optimization of complex rsfq circuits," *IEEE Transactions on Applied Superconductivity*, vol. 7, no. 2, pp. 2685–2689, 1997.

[45] T. Harnisch, J. Kunert, H. Toepfer, and H. Uhlmann, "Design centering methods for yield optimization of cryoelectronic circuits," *IEEE Transactions on Applied Superconductivity*, vol. 7, no. 2, pp. 3434–3437, 1997.

[46] 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.

[47] Y. Tukel, A. Bozbey, and C. Tunc, "Development of an optimization tool for RSfq digital cell library using particle swarm," vol. 23, no. 3, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6380590

[48] Magic VLSI layout tool. [Online]. Available: http://opencircuitdesign.com/magic/

[49] Toped IC layout editor. [Online]. Available: http://www.toped.org.uk/

[50] LASI (layout system for individuals). [Online]. Available: http://www.lasihomesite.com/

[51] P. Xiao, E. Charbon, A. Sangiovanni-Vincentelli, T. Van Duzer, and S. Whiteley, "INDEX: an inductance extractor for superconducting circuits," *IEEE Transactions on Applied Superconductivity*, vol. 3, no. 1, pp. 2629–2632, 1993.

[52] R. S. Bakolo and C. J. Fourie, "Development of a RSFQ cell library for the university of stellenbosch," in *Proc. AFRICON*, 2011, pp. 1–5.

[53] P. Bunyk and S. Rylov, "Automated calculation of mutual inductance matrices of multilayer superconductor integrated circuits," *Ext. Abs. ISEC*, vol. 93, p. 62, 1993.

[54] 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.

[55] Inductex. [Online]. Available: http://www0.sun.ac.za/ix/?q=home

[56] N. Yoshikawa, H. Tago, and K. Yoneyama, "A new design approach for rsfq logic circuits based on the binary decision diagram," *IEEE Transactions on Applied Superconductivity*, vol. 9, no. 2, pp. 3161–3164, 1999.

[57] N. Yoshikawa and J. Koshiyama, "Top-down RSFQ logic design based on a binary decision diagram," *IEEE Transactions on Applied Superconductivity*, vol. 11, no. 1, pp. 1098–1101, 2001.

[58] J. Koshiyama and N. Yoshikawa, "A cell-based design approach for rsfq circuits based on binary decision diagram," *Applied Superconductivity, IEEE Transactions on*, vol. 11, no. 1, pp. 263–266, 2001.

[59] N. Maguu and C. J. Fourie, "Modeling of AQFP logic gates with HDL using multivalued logic approach," in *ISEC*, June, 2015.

[60] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on.* IEEE, 2004, pp. 75–86.

[61] K. Gaj, C.-H. Cheah, E. Friedman, and M. Feldman, "Functional modeling of RSFQ circuits using Verilog HDL," *IEEE Transactions on Applied Superconductivity*, vol. 7, no. 2, pp. 3151–3154, 1997.

[62] J. Charles H. Roth and L. L. Kinney, *Fundamentals of Logic Design.* Cengage Learning, 2014.

[63] A. Krasniewski, "Logic simulation of rsfq circuits," *IEEE Transactions on Applied Superconductivity*, vol. 3, no. 1, pp. 33–38, 1993.

[64] S. Yorozu, Y. Kameda, H. Terai, A. Fujimaki, T. Yamada, and S. Tahara, "A single flux quantum standard logic cell library," *Physica C: Superconductivity*, vol. 378, pp. 1471–1474, 2002.

[65] M. Dorojevets, Z. Chen, C. L. Ayala, and A. K. Kasperek, "Towards 32-bit energy-efficient superconductor RQL processors: The cell-level design and analysis of key processing and on-chip storage units," *IEEE Transactions on Applied Superconductivity*, vol. 25, no. 3, pp. 1–8, 2015.

[66] N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library adopting minimalist design," *Journal of Applied Physics*, vol. 117, no. 17, p. 173912, 2015.

[67] C. J. Fourie, "A tool kit for the design of superconducting programmable gate arrays," Ph.D. dissertation, University of Stellenbosch, 2003.

[68] C. L. Ayala, "Energy-efficient wide datapath integer arithmetic logic units using superconductor logic," Ph.D. dissertation, Stony Brook University, December 2012.

[69] M. Dorojevets and Z. Chen, "Fast pipelined storage for high-performance energy-efficient computing with superconductor technology," in *Emerging Technologies for a Smarter World (CEWIT), 2015 12th International Conference & Expo on*. IEEE, 2015, pp. 1–6.

[70] A. S. et al., "European roadmap on superconductive electronics status and perspectives," *Physica C*, vol. vol. 470,, pp. pp. 2079–2126,, 2010.

[71] S. Intiso, I. Kataeva, E. Tolkacheva, H. Engseth, K. Platov, and A. Kidiyarova-Shevchenko, "Time-delay optimization of RSFQ cells," *IEEE Transactions on Applied Superconductivity*, vol. 15, no. 2, pp. 328–331, 2005.

[72] I. KATAEVA, "Superconducting digital signal processor," Master's thesis, Chalmers University of Technology, 2005.

[73] O. T. Oberg, "Superconducting logic circuit operating with reciprocal magnetic flux quanta," Ph.D. dissertation, University of Maryland, 2011.

[74] L. Muller and C. Fourie, "Automated state machine and timing characteristic extraction for RSfq circuits," vol. 24, no. 1, 2014. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6651715

[75] K. Inoue, N. Takeuchi, T. Narama, Y. Yamanashi, and N. Yoshikawa, "Design and demonstration of adiabatic quantum-flux-parametron logic circuits with superconductor magnetic shields," *Superconductor Science and Technology*, vol. 28, no. 4, p. 045020, 2015.

[76] S. Yamashita, K. Tanaka, H. Takada, K. Obata, and K. Takagi, "A transduction-based framework to synthesize RSFQ circuits," in *Design Automation, 2006. Asia and South Pacific Conference on*, 2006.

[77] A. C.-H. W. Daniel D. Gajski, Nikil D. Dutt and S. Y.-L. Lin, *High-level synthesis, Introduction to Chip and System Design*. Springer Springer+Business Media, LLC, 1992.

[78] W. Clifford and G. Johann, "Yosys a free verilog synthesis suite," in proceedings of Autochip, 2013.

[79] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II-an open-source verilog HDL synthesis tool for cad research," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 149–156.

[80] *Verilog to routing*. [Online]. Available: http://docs.verilogtorouting.org/en/latest/vtr/

[81] *Yosys Manual*. [Online]. Available: http://www.clifford.at/yosys/documentation.html