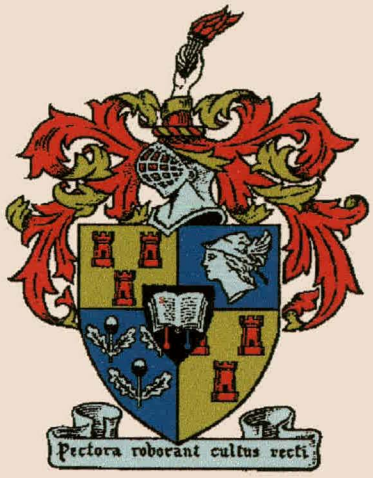


Superconducting *COSL* building blocks for ultra-high speed logic circuits

F.J.Rabie

Thesis presented in partial fulfillment of the requirements for the degree ***Master of Science in Engineering*** *in the Department of Electrical and Electronic Engineering at the University of Stellenbosch.*

Supervisor: Prof. W.J. Perold

December 1999

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

F.J.Rabie
8th February 2000

Abstract

Complementary Output Switching Logic, or *COSL*, is used to define building blocks for ultra-high speed ($\sim$ 20 GHz) logic circuits. The family consists of the standard logic functions OR/NOR, AND/NAND and XOR. A review of the principles of the Josephson junction, one-junction and two-junction *SQUID* and the operation of *COSL* is given as an introduction. The new building blocks include a new single gate inverter, alternatives to the existing NOR and NAND gates and a number of latching functions. Yield analysis and optimisation by a Monte Carlo method is discussed. Additionally, the physical layout of these circuits is considered. Changes in the layout of *COSL* gates to improve the uniformity of current distribution in the gates are briefly mentioned.

Opsomming

Complementary Output Switching Logic, of *COSL*, word gebruik om boublokke te definieer vir ultra-hoëspoed ($\sim$ 20 GHz) logikabane. Die familie bestaan uit die standaard OF/NOF, EN/NEN en eksklusiewe OF funksies. 'n Oorsig oor die beginsels van die Josephsonvlak, een-vlak en tweevlak *SQUID* en die werking van *COSL* word ter inleiding gegee. Die nuwe boublokke sluit 'n nuwe enkelhek omkeerder, alternatiewe vir die bestaande OF en NEN hekke en 'n aantal grendelfunksies in. Opbrengsanalise en -optimering deur middel van 'n Monte Carlo metode word bespreek. Verder word die fisiese uitleg van hierdie bane beskou. Veranderinge in die uitleg van *COSL* hekke om die uniformiteit van stroomverspreiding in die hekke te verbeter word kortliks genoem.

Acknowledgements

A great deal of thanks should go to Prof. W. J. Perold for his patience and invaluable assistance. Thank you for contagious enthusiasm when my own seemed non-existent. Also, thank you for the opportunity to visit the University of California at Berkeley to test my circuits.

Thank you to Dr. Stephen R. Whiteley of Whiteley Research Inc., whose Unix based simulation and layout packages allowed me to use my operating system of choice — Linux. Thank you for promptly fixing the bugs I managed to uncover.

I would also like to thank Dr. Masoud Radparvar at HYPRES, who, despite a barrage of E-mails, remained helpful.

Contents

List of Figures

List of Tables

Nomenclature

β_c Stewart-McCumber parameter, page 4.

β_L Basic parameter of a *SQUID*, page 6.

COSL Complementary Output Switching Logic.

I_c Josephson junction critical current.

I_{con} Two-junction *SQUID* control current.

I_g Two-junction *SQUID* gate current.

I_{min} Input current at which a two-junction *SQUID* switches back to the previous quantum state , page 6.

I_{th} Input current at which a two-junction *SQUID* switches to the next quantum state , page 6.

K Fringe field factor, page 31.

k Inductive coupling coefficient.

k_c Constant depending on the required confidence level of yield prediction.

k_s Scaling factor for the determination of K.

λ London penetration depth of a superconductor.

$\mathcal{L}$ Confidence interval of circuit yield prediction, page 25.

L_J Josephson inductance, page 46.

ϕ Gauge-invariant phase difference across a Josephson junction.

Φ_0 Magnetic flux quantum, page 3.

R_n Josephson junction normal state resistance.

R_{sg} Josephson junction subgap resistance.

SQUID Superconducting Quantum Interference Device.

θ Phase of the quantum mechanical wave function.

V_g Josephson junction gap voltage.

VLSI Very Large Scale Integration.

y True statistical yield of a circuit, page 25.

y' Observed yield of a circuit.

Chapter 1

Introduction

The speed at which electronic circuits operate is limited by the physical properties of the materials from which these circuits are manufactured. Semiconductor materials are fast approaching the limits of their operation. Superconducting materials can be considered a viable alternative for the manufacture of high speed circuits.

The first superconducting digital element was proposed in 1956 [1]. The cryotron was too slow, however, to provide any significant competition for semiconductor technology. The discovery of the Josephson effect in 1962 [2] and the subsequent development of the Josephson junction led to advent of high speed superconducting digital circuits. Many new logic families were developed, capable of operating at much higher speeds than their semiconductor counterparts.

Complementary Output Switching Logic, or *COSL* [3],[4], is currently the fastest superconducting voltage-state logic family [5]. This high speed return-to-zero family consists of OR/NOR, AND/NAND and XOR gates. *COSL* gates have been tested at frequencies as high as 18 GHz [6].

This thesis presents a number of additions to the *COSL* family, introduced in Chapter 3. Alternatives to the existing NOR and NAND circuits are proposed, as well as a number of latches. As background, the basic operation of *COSL* is reviewed in Chapter 2.

Process variations during manufacture have a detrimental effect on the correct operation of digital circuits [7]. In order to increase the reliability of the *COSL* family, the circuits are optimised by using a Monte Carlo yield prediction technique. Yield analysis and optimisation of the proposed circuits are discussed in Chapter 4.

The physical layout of the circuits mentioned above is discussed in Chapter 5. Changes in the layout geometry are considered in an effort to limit the effects of field concentrations in the *COSL* circuits.

Chapter 2

Review of *COSL*

2.1 Introduction

Before the operation of *COSL* can be discussed, the basic superconducting digital element, the Josephson junction, has to be considered. This device forms the basis of the so-called Superconducting Quantum Interference Device, or *SQUID*.

The *SQUID* is widely used as a very sensitive magnetometer capable of detecting magnetic fields in the order of 10^{-14} T [8, p. 1313]. It is formed by a superconducting ring containing one or two Josephson junctions. The heart of *COSL* is formed by a one-junction *SQUID* which is inductively coupled to a two-junction *SQUID*.

In this chapter, the basic properties of the Josephson junction are presented, followed by a review of the operation of the one-junction and two-junction *SQUID*. Finally, the basic operation of a *COSL* gate will be discussed.

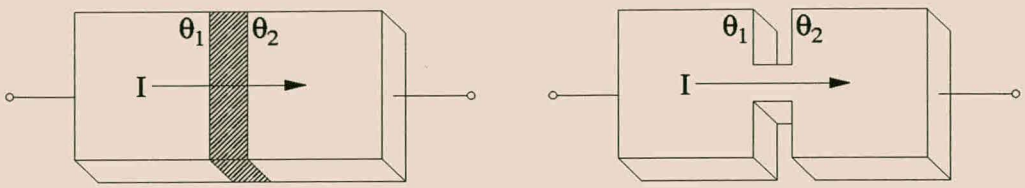
2.2 Josephson junction

2.2.1 Basic Josephson junction

The Josephson junction is formed by weak coupling between two superconductors [9], either by an insulator or a connection with a small cross-sectional area. The weak coupling causes the junction to switch to the normal state when the current through the junction exceeds a certain critical value. Figure 2.1 (a) and (b) show the structure of the so-called SIS, or superconductor-insulator-superconductor, and weak link junction respectively. The weak link junction is also known as a microbridge [10, p. 507].

The behaviour of the Josephson junction is governed [10, pp. 405-406] by a current-phase relation

$$i = I_c \sin\phi(t), \tag{2.1}$$



(a) (b)

Figure 2.1: Structure of the (a) SIS and (b) weak link Josephson junction.

gauge-invariant phase relation

$$\phi(t) = \theta_1(t) - \theta_2(t) - \frac{2\pi}{\Phi_0}\int_1^2 \mathbf{A}(\mathbf{r},t)\cdot \mathrm{d}\mathbf{l}, \tag{2.2}$$

and voltage-phase relation

$$v = \frac{\Phi_0}{2\pi}\frac{\mathrm{d}\phi}{\mathrm{d}t}, \tag{2.3}$$

where I_c is the critical current of the junction, θ is the phase of the quantum mechanical wave function [10, p. 236] and Φ_0 is the magnetic flux quantum, which is equal to $\frac{h}{2e}$.

The vector potential $\mathbf{A}$ in (2.2) is the curl of the magnetic flux density. Because the number of possible vector potentials is infinite, it is desirable to make the phase relation *gauge invariant* [10, p. 237]. ϕ is therefore known as the gauge-invariant phase difference across the junction. The path of integration is from θ_1 to θ_2, as indicated in Figure 2.1.

With no applied voltage across the junction the phase difference is zero and a constant current is allowed to flow through the junction. This phenomenon is known as the DC Josephson effect.

An applied (constant) voltage results in an oscillating current with a constant frequency, which has a voltage dependency of 483.598 MHz/μV. This effect is referred to as the AC Josephson effect.

2.2.2 Generalised Josephson junction

Practical Josephson junctions are influenced by resistive and capacitive effects. The so-called resistively shunted junction model takes these effects into account. An ideal junction is shunted [10, p. 459] by a resistance, R, and a capacitance, C, as shown in Figure 2.2.

The current through this generalised junction is given by

$$i = I_c \sin\phi + C\frac{\mathrm{d}v}{\mathrm{d}t} + \frac{v}{R}, \tag{2.4}$$

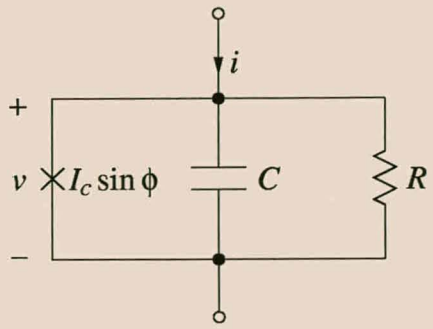


Figure 2.2: Equivalent circuit of the generalised Josephson junction.

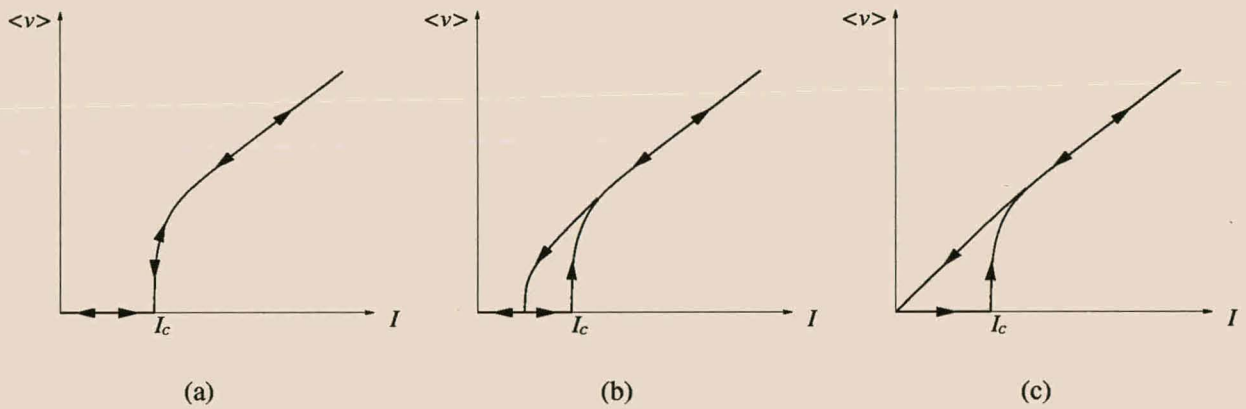


Figure 2.3: Voltage-current relationship of the generalised Josephson junction for (a) $\beta_c \ll 1$, (b) $\beta_c \approx 1$ and (c) $\beta_c \gg 1$. I is the DC current through the junction and $< v >$ is the average voltage accross the junction.

which, with substitution of the voltage-phase relation in (2.3), becomes

$$\frac{i}{I_c} = \sin\phi + \frac{d\phi}{d\tilde{\tau}} + \beta_c \frac{d^2\phi}{d\tilde{\tau}^2}, \tag{2.5}$$

where $\tilde{\tau} = \frac{t}{\tau_J}$, $\beta_c = \frac{\tau_{RC}}{\tau_J}$ is the Stewart-McCumber parameter, $\tau_J = \frac{\Phi_0}{2\pi R I_c}$ and $\tau_{RC} = RC$ [10, p. 459].

The Stewart-McCumber parameter is an indication of the influence of the junction capacitance. The amount of hysteresis exhibited by the voltage-current relationship of the junction increases proportionally to its capacitance. This is illustrated in Figure 2.3, for a constant resistance, in order of increasing capacitance.

The superconducting and normal states of a Josephson junction are separated by a so-called energy gap [10, p. 450]. This gap can be expressed as a voltage, which, in the case of niobium, is approximately equal to 2.5mV. The quasi-static [11] current-voltage curve of a Josephson junction, which is shown as the dark curve in Figure 2.4 (a) clearly shows the gap voltage V_g. The slope of the curve is given by the so-called subgap resistance, R_{sg}, and the resistance of

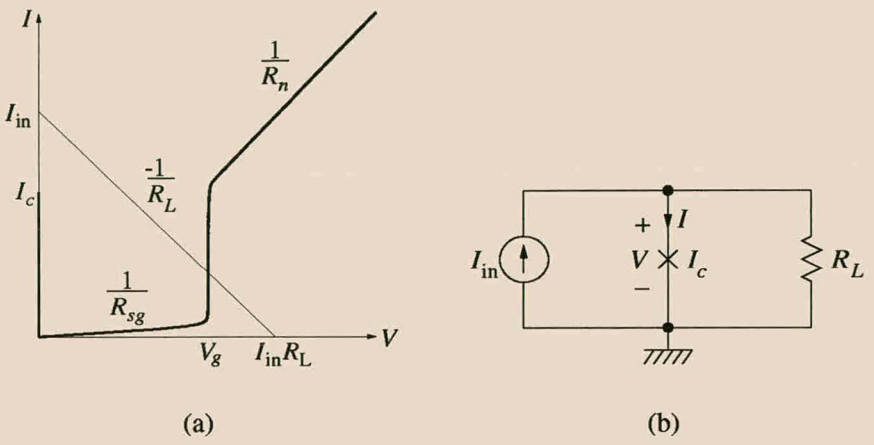


Figure 2.4: (a) Quasi-static current-voltage characteristic with loadline and (b) equivalent circuit of a resistively shunted Josephson junction.

the junction in the normal state, R_n. By resistively shunting the Josephson junction as shown in Figure 2.4 (b) the junction can be forced to switch to a particular point on the current-voltage curve. This point is determined by the positioning of the loadline.

2.3 One-junction *SQUID*

As mentioned in Section 2.1, a one-junction *SQUID* is formed by a superconducting loop containing a single Josephson junction. An equivalent circuit for the one-junction *SQUID* can be obtained by modelling the loop as an inductor, as shown in Figure 2.5. From the circuit, it can be seen that:

$$i_{\text{in}} = I_c \sin\phi + i_L, \tag{2.6}$$

where the current through the Josephson junction is given by the current-phase relation, (2.1). Substitution of an expression for ϕ from the voltage-phase relation, (2.3), allows the relationship between inductor current and input current to be written as

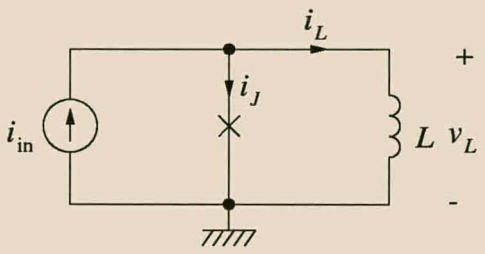


Figure 2.5: Equivalent circuit of the one-junction *SQUID*.

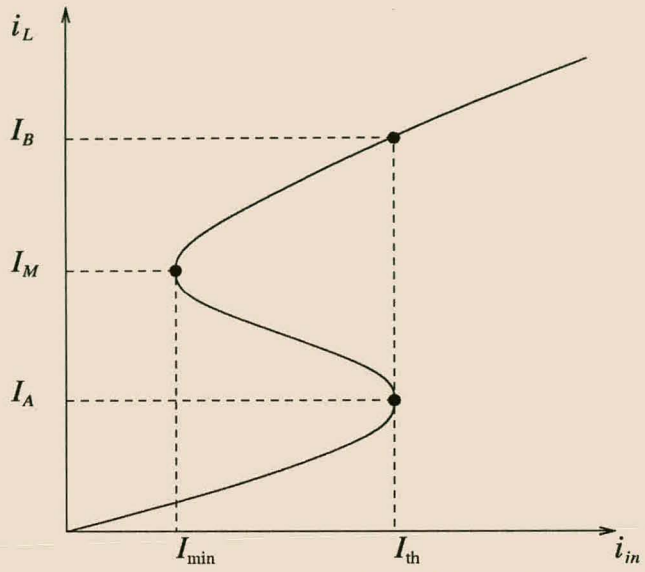


Figure 2.6: Relationship between inductor current and input current of the one-junction *SQUID*.

$$i_{\text{in}} = I_c \sin\left(\frac{2\pi L i_L}{\Phi_0}\right) + i_L. \tag{2.7}$$

This relationship is illustrated in Figure 2.6.

The threshold current, I_{th}, is the input current at which the *SQUID* switches to the next quantum state. It is determined by letting

$$\frac{\mathrm{d}i_{\text{in}}}{\mathrm{d}i_L} = 0, \tag{2.8}$$

which gives an expression for the inductor current I_A. Substitution of this expression into (2.7) leads to

$$I_{\text{th}} = I_c \sin\left[\arccos\left(-\frac{1}{\beta_L}\right)\right] + \frac{I_c}{\beta_L}\arccos\left(-\frac{1}{\beta_L}\right), \tag{2.9}$$

where $\beta_L = \frac{2\pi L I_c}{\Phi_0}$. β_L is a basic parameter of the *SQUID* [12, p. 155].

In order to determine the input current, I_{min}, at which the *SQUID* switches back to the original quantum state, (2.6) is written as [13, p. 259]

$$i_{\text{in}} = I_c \sin\left(2n\pi - 2\pi\frac{\Phi_{\text{int}}}{\Phi_0}\right) + i_L, \tag{2.10}$$

where Φ_{int} is the flux enclosed by the *SQUID* loop. The sinusoidal nature of this relationship facilitates the assumption that I_{th} and I_{min} lie equidistant above and below an enclosed flux of $\frac{\Phi_0}{2}$, which leads to

$$\frac{\Phi_0}{2L} - I_{\text{min}} = I_{\text{th}} - \frac{\Phi_0}{2L}. \tag{2.11}$$

(2.11), with β_L substituted, results in

$$I_{\min} = \frac{2\pi I_c}{\beta_L} - I_{\text{th}}. \tag{2.12}$$

2.4 Two-junction *SQUID*

The two-junction *SQUID* was introduced in Section 2.1 as a superconducting loop containing two Josephson junctions. An equivalent circuit for the general two-junction *SQUID* is shown in Figure 2.7. A damping resistor, R_d, can be added, as indicated by the dashed connection. Current can be directly injected, indicated by the gate current I_g, and/or inductively coupled, indicated by the control current I_{con}. [14, p. 50].

The behaviour of the two-junction *SQUID* can be described by relating the gate current to the control current. The relationship between $I_{g\max}$, the maximum gate current the *SQUID* can carry before switching to the normal state, and the control-current is called a threshold curve [14, p. 49].

$I_{g\max}$ is determined from the current-phase relation of the Josephson junction, (2.1). Summation of I_1 and I_2 lead to

$$I_{g\max} = I_c \sin\phi_1 + I_c \sin\phi_2, \tag{2.13}$$

The relationship between the control current and the junction phases is determined from the magnetic flux threading the loop. The phase difference of the loop, in terms of the total magnetic flux, Φ, is given by [13, p. 262]

$$\phi_2 - \phi_1 = 2n\pi - \frac{2\pi\Phi}{\Phi_0}. \tag{2.14}$$

Φ is divisible in two parts [15], namely externally generated flux,

$$\Phi_{\text{ext}} = MI_{\text{con}}, \tag{2.15}$$

and flux caused by circulating currents in the loop,

$$\Phi_s = \frac{L}{2}I_2 - \frac{L}{2}I_1. \tag{2.16}$$

Substitution of (2.15) and (2.16) into (2.14) allows the normalised control current to be written as

$$\frac{MI_{\text{con}}}{\Phi_0} = \frac{\phi_1 - \phi_2}{2\pi} + \frac{\beta_L}{4\pi}(\sin\phi_1 - \sin\phi_2) + n, \tag{2.17}$$

where n identifies the flux quantum state [16]. Finally, a relationship between the junction phases is needed to determine the threshold curve. An auxiliary equation [17],

$$\sec\phi_1 + \sec\phi_2 + \beta_L = 0, \tag{2.18}$$

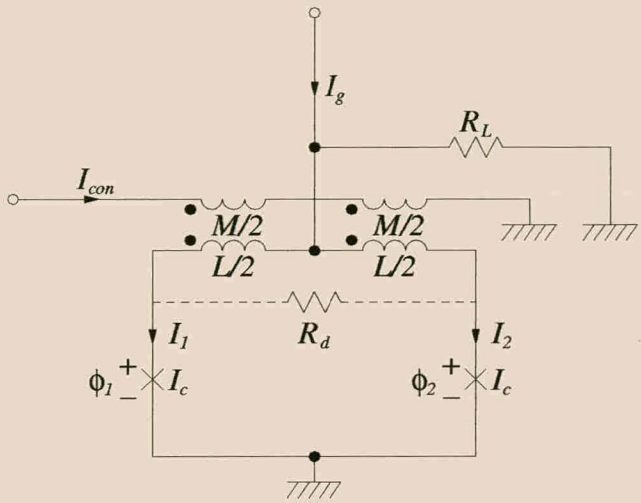


Figure 2.7: Equivalent circuit of a symmetrical two-junction *SQUID*.

serves this purpose. Figure 2.8 shows a typical threshold curve for the two-junction *SQUID*, where the modulatory effect of the control-current is apparent.

The curve is inversely symmetrical [15], and is only shown for positive gate currents. It is clear from the figure that the periodicity of the enclosed flux is one flux quantum. The red curve corresponds to the zero flux quantum state for $-\frac{\pi}{2} < \phi_2 < \frac{\pi}{2}$ and the green curve to the one flux quantum state for $-\frac{\pi}{2} < \phi_1 < \frac{\pi}{2}$.

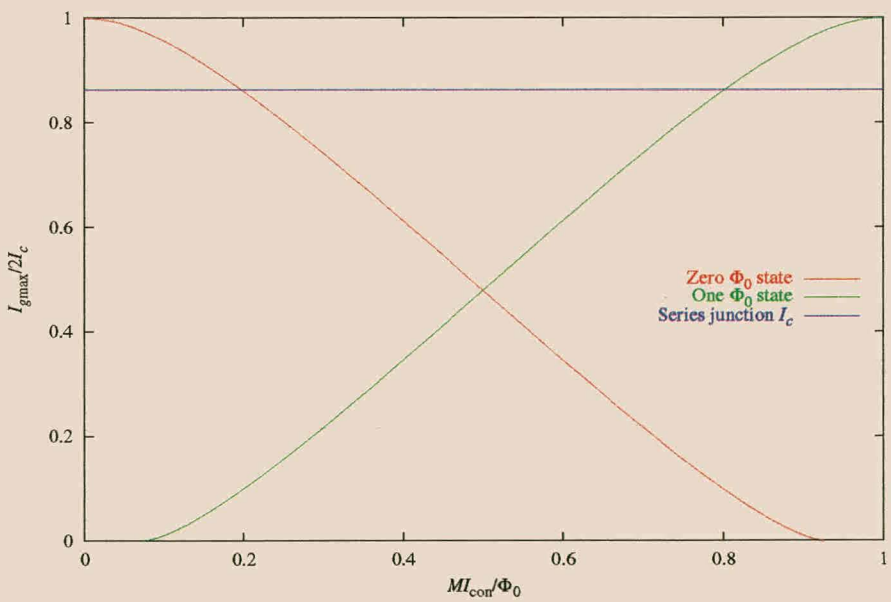


Figure 2.8: Threshold curve of the two-junction *SQUID*.

2.5 COSL

The two-junction *SQUID* forms the principle element of a *COSL* gate. A one-junction *SQUID* is used to supply the control current. The basic *COSL* gate is shown in Figure 2.9. The one-junction *SQUID* will hereafter be referred to as the input *SQUID*; and the two-junction *SQUID* as the output *SQUID*.

COSL gates are clocked with a three-phase clocking scheme. Each phase is separated by 120° or one third of one clock period. The input and output *SQUID*s are driven by separate clock phases. The input clock is in phase with the output clock of the previous gate, with the exception of the XOR gate. In this case, the input clock is delayed by one third of one clock period with respect to the output clock of the previous gate. The output clock usually lags one third of one clock period behind the input clock, to allow for pulse propagation from input to output.

The clock shapers indicated in Figure 2.9 ensure a constant clock amplitude of 2.5 mV. The shunt resistance forces the shaping junctions to switch to the gap voltage as soon as the clock current exceeds the critical current of these junctions, as discussed in Section 2.2. A nominal external clock amplitude of 10 mV is used.

All *COSL* gates except XOR are driven by 100 μA inputs. These currents, together with the current from the input clock, determine the state of the Josephson junction of the input *SQUID*. The input clock current is limited by the bias resistor, which effectively determines the logic function of the gate. A bias resistance of 8.8 Ω provides enough clock current to cause the

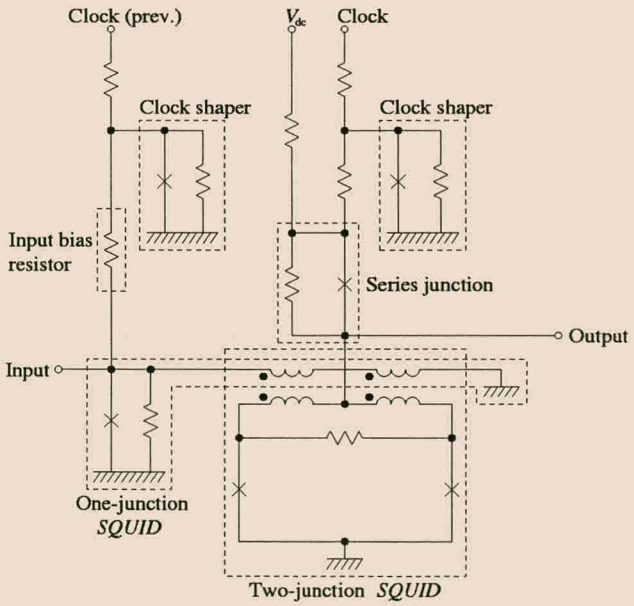


Figure 2.9: Basic *COSL* gate.

input *SQUID* to be switched for one 100 μA input, which provides the OR/NOR function. The 14.3 Ω bias resistance of the AND/NAND gates necessitates two 100 μA inputs to switch the input *SQUID*, which satisfies the required logic function.

An additional series Josephson junction, which is not shown, is added to the input of the basic gate to form the XOR function. The critical current of this junction is chosen at 300 μA so that it is not exceeded by one 200 μA input. This input will switch the input *SQUID* and cause a logical high output. Two 200 μA inputs are required to switch the series junction, which cuts off the input current and causes a zero output.

COSL gates are designed to deliver 1 mV into 5 Ω, which gives a fanout of two for all cases except XOR. The current requirements of the XOR gate necessitate the use of OR buffers for each input. Fanout-of-one gates require an additional 10 Ω resistor to ground to maintain the required 5 Ω output resistance. The 5mV DC voltage applied to the output prevents the output *SQUID* from switching for negative currents.

The complementary nature of the *COSL* family arises from the series junction located at the output of each gate. When the gate input is not suitable to switch the input *SQUID*, the threshold current of the output *SQUID* is higher than the critical current of the series junction. Hence, only the series junction switches, which corresponds to a zero output. Conversely, if the input *SQUID* is switched by the gate input, the control current of the output *SQUID* is raised. This higher control current modulates the threshold current to a value below the critical current of the series junction. As a result, the output *SQUID* switches before the series junction, producing a logical high output. The series junction critical current and the output *SQUID* threshold current are compared in Figure 2.8. Ideally, a noninverting gate (OR or AND) can be made to invert (NOR or NAND) by placing the series junction below the output *SQUID* and taking the output across this junction.

The reliability of the family is increased by the addition of a so-called trim voltage, which is applied to the input of each gate through a 50 Ω resistor. This is discussed in more detail in Section 4.2.

2.6 Conclusions

From the previous discussion it can be seen that the operation of *COSL* relies on a one-junction *SQUID*, which is inductively coupled to a two-junction *SQUID*. The one-junction and two-junction *SQUID* are superconducting loops, containing one and two Josephson junctions respectively. Attention is paid to the principles of the Josephson junction and one-junction and two-junction *SQUID*. The operation of *COSL* is described in some detail.

Chapter 3

Design of the Building Blocks

3.1 Introduction

The *COSL* family comprises the basic logic functions OR/NOR, AND/NAND and XOR. These gates can be used to produce other (more complex) logic circuits. However, because of the three-phase clocking scheme used, increased complexity could mean increased latency. Additionally, complex circuits could exhaust valuable integrated circuit area. Solutions requiring a minimum number of gates are therefore preferable.

Process variations limit the usefulness of superconducting circuits [7] by lowering the yield of these circuits. The yield of inverting *COSL* gates were found to be significantly lower than that of the non-inverting gates [3].

This chapter presents proposed building blocks which expand the versatility of *COSL* without unnecessary circuit complexity. Furthermore, in an effort to increase the overall yield of the family, alternatives to the existing inverting gates are investigated.

Circuits were simulated with WRspice [18]. A clock frequency of 10 GHz is used throughout.

3.2 Inverter

Because of the relatively low yield of *COSL* inverting gates, the inverting function is usually performed by an XOR gate, of which one input is kept high. This permanent logical high is provided by a so-called dummy OR, which has a bias resistance of 6 Ω [19]. This resistance is low enough to allow the clock current alone to switch the gate. Thus, together with the buffer needed to drive the other XOR input, three gates are needed to implement an inverter. Moreover, this combination causes a delay of one clock period.

In an attempt to implement a single gate inverter, the dummy OR was considered. The

continuous switching of this gate can be compared to the inversion of a logical low input. The addition of a Josephson junction with a critical current of 360 μA in the inductive branch of the input *SQUID* of the gate facilitates the inversion of a logical high input. This junction, as well as the modified bias resistance, is indicated by a dashed box in Figure 3.1.

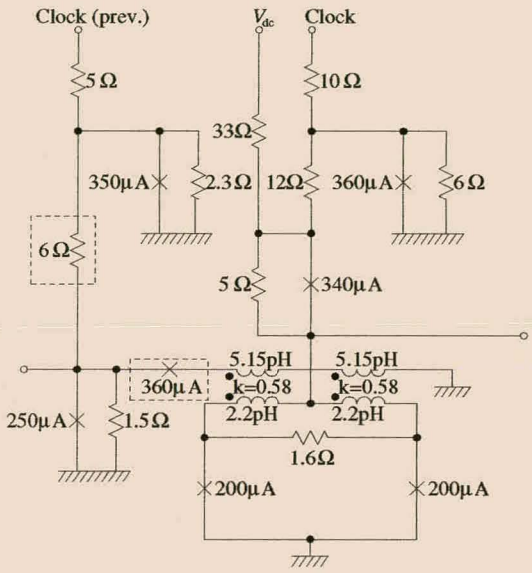


Figure 3.1: *COSL* inverter. The additional Josephson junction and modified bias resistance are indicated by dashed boxes.

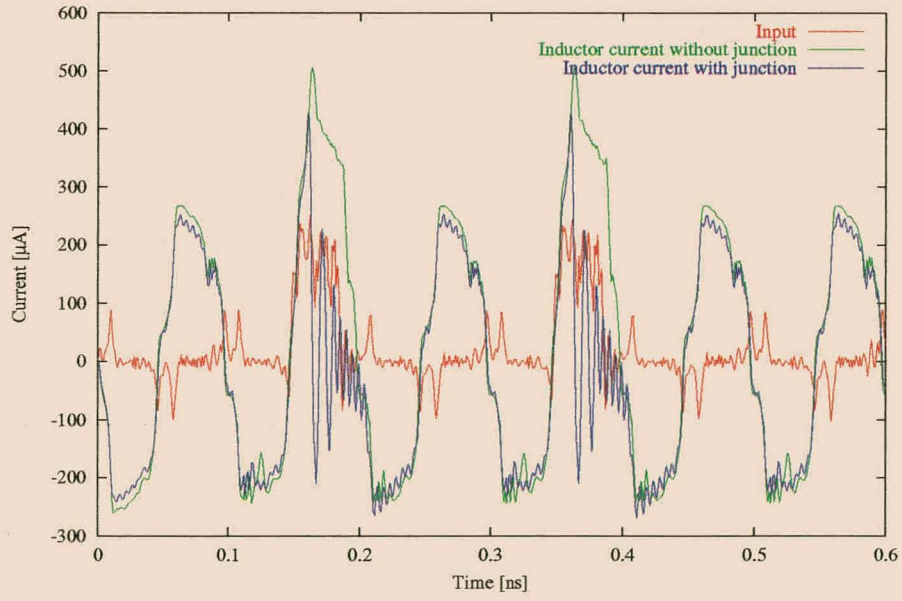


Figure 3.2: Simulated inductor current of the *COSL* inverter with and without the additional Josephson junction. Input pulses are shown for clarity.

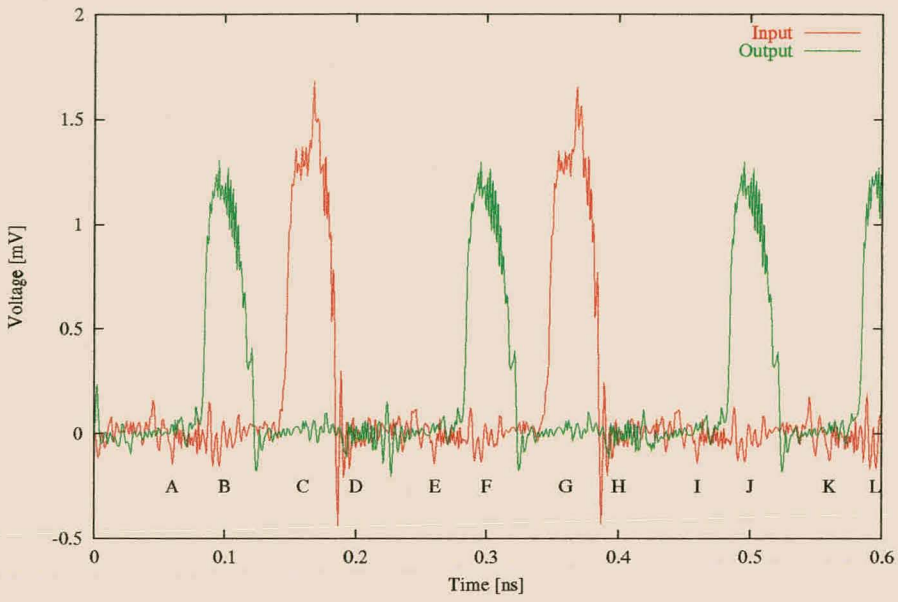


Figure 3.3: Simulated input/output behaviour of the *COSL* inverter. The delay between input and output is one third of one clock period. Table 3.1 gives the corresponding truth table.

Input		Output	
0	(A)	1	(B)
1	(C)	0	(D)
0	(E)	1	(F)
1	(G)	0	(H)
0	(I)	1	(J)
0	(K)	1	(L)

Table 3.1: Truth table for the *COSL* inverter, corresponding to Figure 3.3.

When no input is applied to the gate the current through the inductor branch is approximately 250 μA, which is insufficient to switch the junction. This current is, however, enough to switch the output *SQUID*. With a logical high input the inductor current peaks at more than 500 μA, which forces the junction to switch to the normal state. The resulting increased resistance of the junction lowers the current in the inductor branch to such an extent that the output *SQUID* is not switched. Figure 3.2 clearly demonstrates the inductor current peaks caused by input pulses if the junction is omitted. For clarity, the input pulses are also given. When the junction is included the inductor current behaves as shown, where the current limiting effect of the junction is evident.

Finally, the simulated input/output behaviour of the inverter is presented in Figure 3.3. The delay between input and output is one third of one clock period. The corresponding truth table is given in Table 3.1.

3.3 T latch

The T, or toggle, latch is useful for the implementation of, for example, flags. The output of these latches is inverted with every input pulse. The three-phase clocking scheme makes the XOR logic function ideal for the realisation of this building block. The circuit diagram for the T latch is given in Figure 3.4. For simplicity, standard *COSL* gates have been replaced by their respective logic symbols. Clock phases and modifications are indicated. Input and output resistances are also included.

The operation of the T latch relies on the delay between the input and output of a *COSL* gate. Consider, for instance, the case where the output of the latch is zero. A single input pulse is applied to the XOR gate. An output pulse arrives after a delay of two thirds of one clock period. This pulse is fed back to the input by the OR gate, with an additional delay of one third of one period. Thus, one period after the external input pulse, the XOR gate receives another pulse from the OR gate. This pulse triggers the XOR gate and the process is repeated. So the latch is set to the logic high state. If an external input pulse is applied while this is true, the XOR gate would give a zero output. Nothing is fed back by the OR gate and the latch is reset to the zero state.

The feedback scheme causes a slight discrepancy to develop between the output phase of the OR gate and the input phase of the XOR gate. The output pulse of the OR gate shifts slightly with each feedback action. Eventually, the feedback pulse arrives too late to trigger the XOR gate and the logical high state is destroyed. An increase in the amount of input clock current of the XOR gate remedies this problem. The bias resistance of the gate is decreased to 8.8 Ω, which is the same as that of the standard OR gate.

The simulated response of the *COSL* T latch is shown in Figure 3.5, with the corresponding truth table in Table 3.2. The traces demonstrate how an input pulse propagates through the XOR gate and is fed back by the OR gate. The clocking scheme causes a delay of two thirds of one clock period between input and output. The comparatively large input pulse is caused by the switching of the XOR series junction.

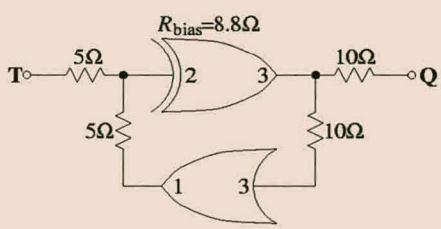


Figure 3.4: *COSL* T latch. Input and output resistances and clock phases are shown. The modified bias resistance of the XOR gate is also indicated.

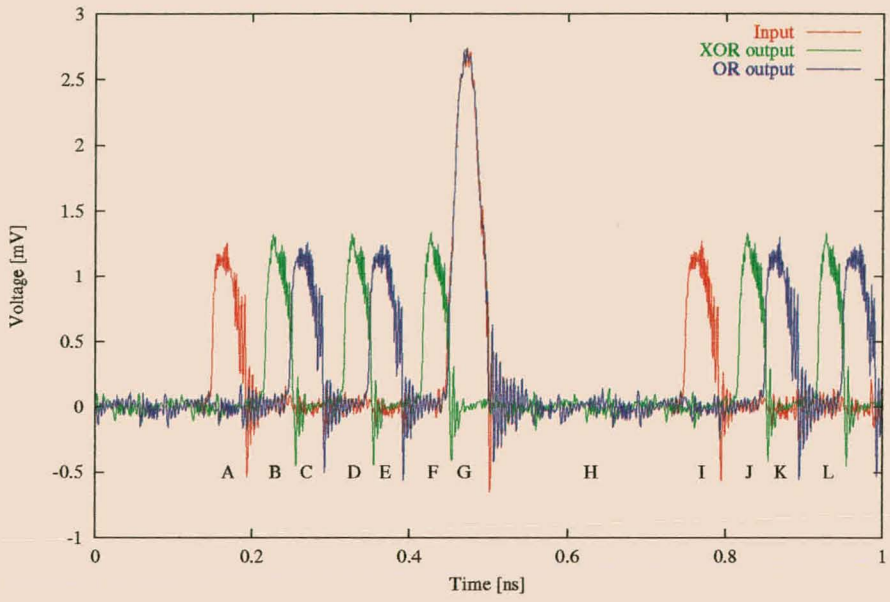


Figure 3.5: Simulated input/output behaviour of the *COSL* T latch. The delay between input and output is two thirds of one clock period. Table 3.2 gives the corresponding truth table.

Input		Output	
1	(A)	1	(B)
0	(C)	1	(D)
0	(E)	1	(F)
1	(G)	0	(H)
1	(I)	1	(J)
0	(K)	1	(L)

Table 3.2: Truth table for the *COSL* T latch, corresponding to Figure 3.5.

3.4 NOR and NAND gates

The symmetrical behaviour of the Josephson junction and one-junction *SQUID* is evident from Section 2.2 and 2.3. This property plays a fundamental role in the operation of the proposed alternative NOR and NAND gates.

As is the case with the inverter discussed in Section 3.2, the inversion of a logical low is accomplished by lowering the bias resistance of a basic *COSL* gate. Again, this makes the input clock current sufficient to switch the gate. In the case of the NOR gate the bias resistance is set to 6.5 Ω and in the case of the NAND gate to 5 Ω.

To implement the inversion of a nonzero input a different clocking scheme is needed. The input phase of these gates was chosen to lead the output phase of the previous gate by one third of one clock period. The output phase remains the same, as it would for a normal gate. The

consequence is that the negative inductor current corresponds to a positive output clock current, as shown in Figure 3.6. The symmetry of the output *SQUID* and the positive DC bias ensure a positive output pulse.

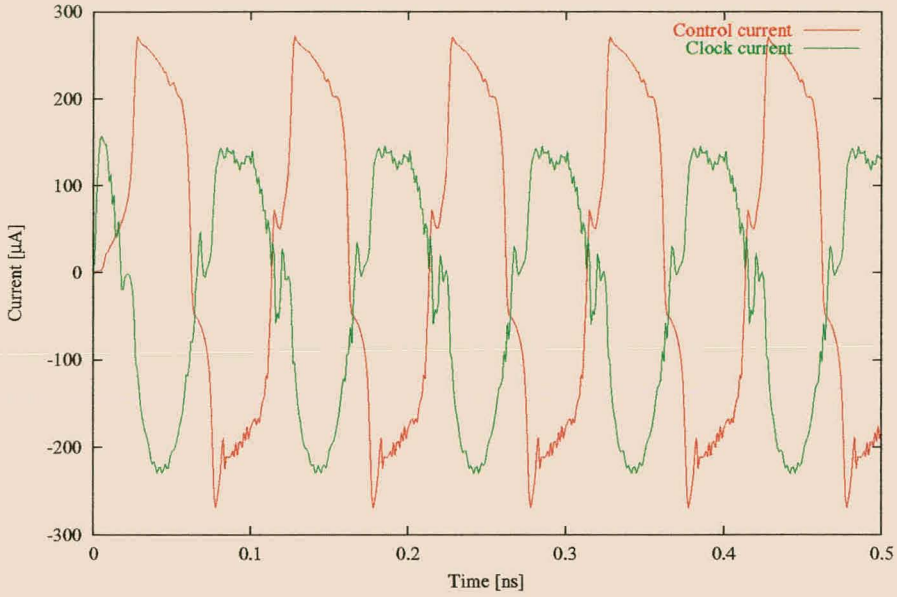


Figure 3.6: Simulated control current and clock current of the NOR gate output *SQUID*.

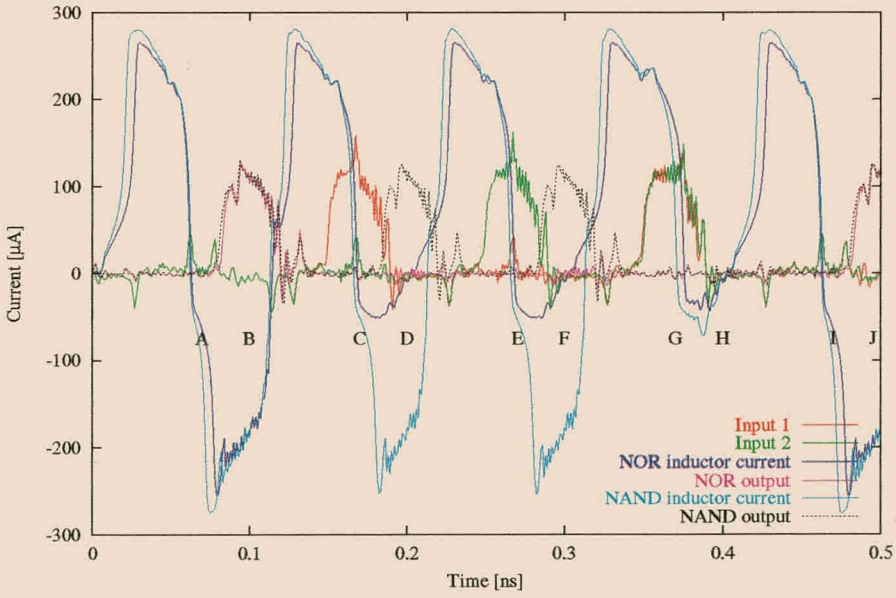


Figure 3.7: Simulated response of the *COSL* NOR and NAND gate, including inductor currents. Table 3.3 gives the corresponding truth table.

Inputs 1,2		Outputs NOR,NAND	
0,0	(A)	1,1	(B)
1,0	(C)	0,1	(D)
0,1	(E)	0,1	(F)
1,1	(G)	0,0	(H)
0,0	(I)	1,1	(J)

Table 3.3: Truth table for the *COSL* NOR and NAND gate, corresponding to Figure 3.7.

The input pulses to the gates are used to modify the inductor current. The positive current from these pulses serve to make the inductor current less negative. When the appropriate inputs are applied to the gates the inductor current is increased to such an extent that the output *SQUID* is not switched. In the case of the NOR gate this would mean that one input would increase the inductor current sufficiently to prevent the output *SQUID* from switching, while two inputs are needed for the NAND gate. This corresponds exactly to the NOR and NAND logic functions respectively. Figure 3.7 shows the input/output behaviour, as well as the inductor currents of the NOR and NAND gate. The corresponding truth table is given in Table 3.3.

3.5 Negative gate

As stated in Section 2.4, the threshold curve of the two-junction *SQUID* is inversely symmetrical. This characteristic is exploited to implement a *COSL* gate which generates negative output pulses from positive input pulses.

The direction of the bias current of the output is reversed and the clock phase chosen so that the control current peaks supplied by the input *SQUID* corresponds to a negative peak in the clock current, as shown in Figure 3.8. The output clock phase of the negative gate leads the input clock phase by one third of one period. The negative DC bias causes a negative output pulse to be generated, as illustrated by Figure 3.9. By using the normal clocking scheme, subsequent gates will propagate this negative pulse, provided that the DC bias of these gates is negative. The pulse can be reversed again by a standard gate, with the same clocking scheme as the negative gate. In other words, the output clock leads the input clock.

Any of the standard noninverting *COSL* gates can be used as negative gates.

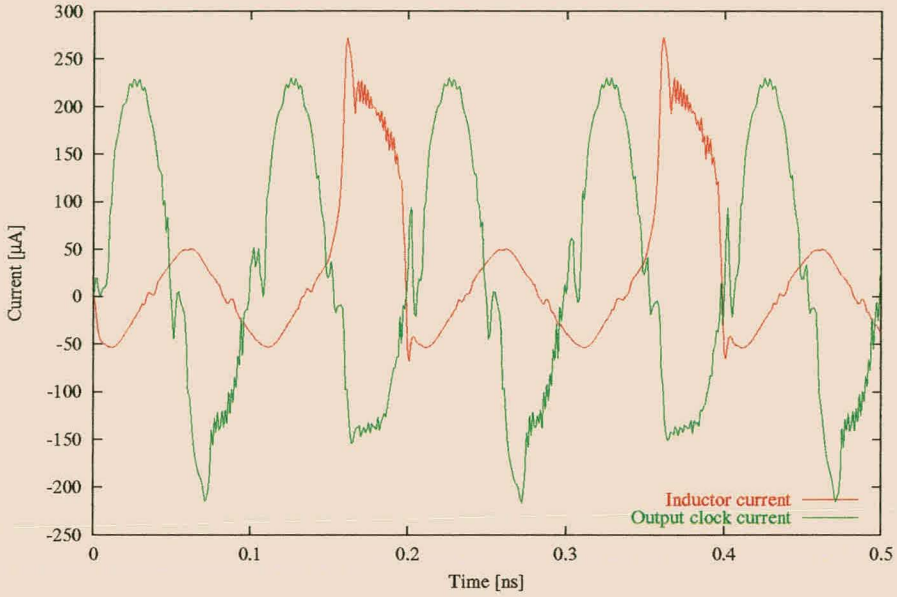


Figure 3.8: Simulated control and clock current of the negative gate output *SQUID*.

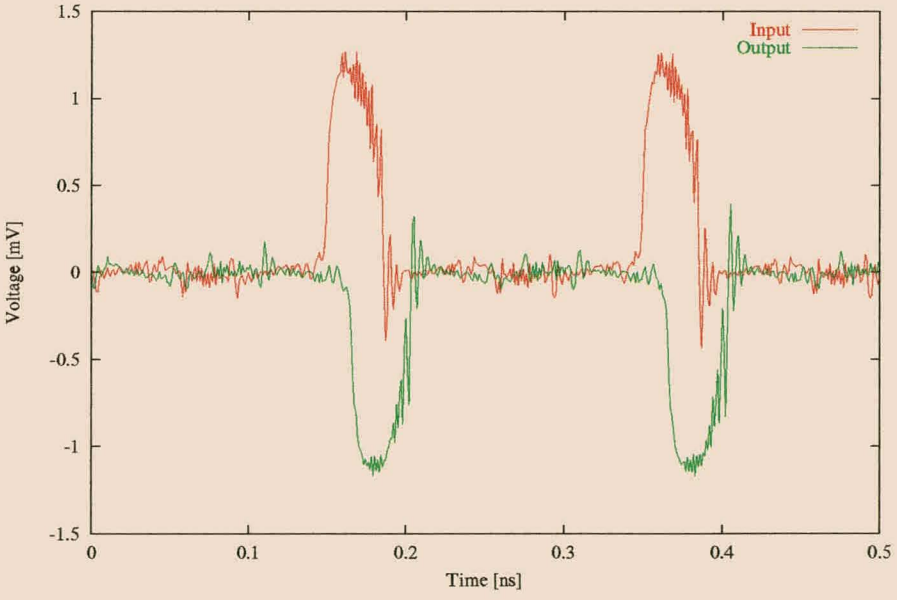


Figure 3.9: Simulated response of the *COSL* negative gate.

3.6 S-R latch

3.6.1 First attempt

For the first attempt at implementing a set-reset or S-R latch the inductive coupling between the input and output *SQUID* of a *COSL* gate is considered. Like the T latch, the latching property

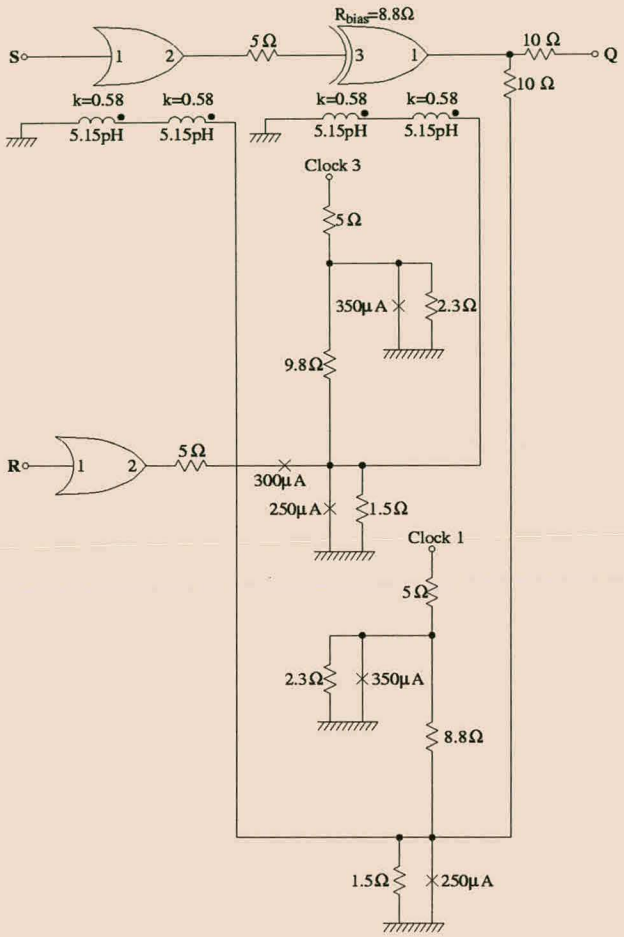


Figure 3.10: *COSL* S-R latch (first attempt). Input resistors have been omitted.

of this building block is implemented by a feedback scheme, shown in Figure 3.10. The output is fed back to the S input by an additional OR input *SQUID*. This *SQUID* is inductively coupled to the output *SQUID* of the OR gate located at this input. This allows a logical high output to be maintained if the latch is set.

The important factor when considering the reset action of the latch, is the direction of the induced current in the output *SQUID* of the XOR gate. An additional XOR input *SQUID* is used to couple the reset input to this output *SQUID*. The inductors are arranged in such a way, that the induced current from the reset input has the opposite direction to the induced current from the set input. Consequently, if the latch is set, the output is cancelled when a reset is received.

Figure 3.11 shows the response of the S-R latch to various inputs. It is evident that undefined inputs have no effect. The corresponding truth table is shown in Table 3.4.

The functionality of the S-R latch can be increased by adding an enable input. This is achieved by simply replacing the OR gates by AND gates. As a result, a further logical high input is needed for a set or reset to take effect.

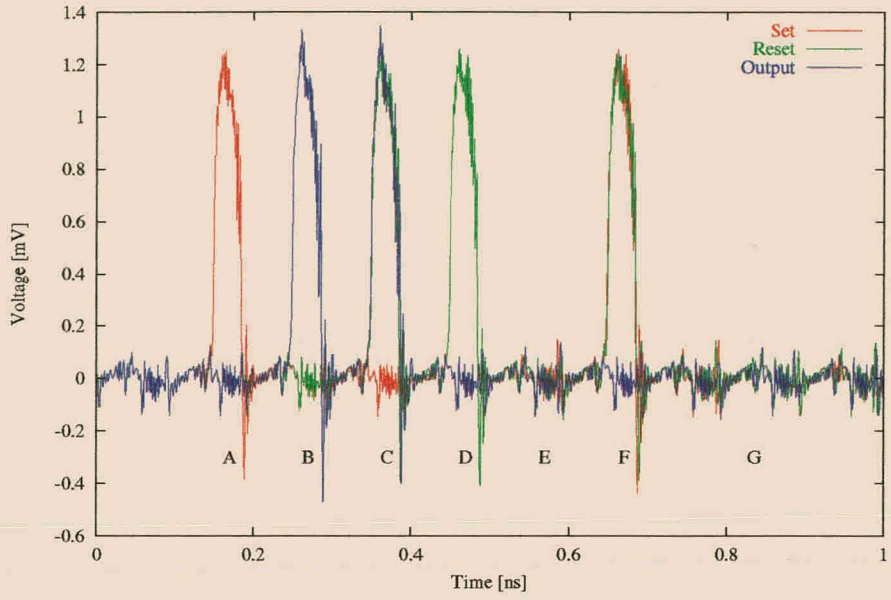


Figure 3.11: Simulated response of the *COSL* S-R latch (first attempt) to various inputs. The delay is one clock period. Table 3.4 gives the corresponding truth table.

Inputs Set,Reset		Output	
1,0	(A)	1	(B)
0,0	(B)	1	(C)
0,1	(C)	0	(D)
0,1	(D)	0	(E)
0,0	(E)	0	(F)
1,1	(F)	0	(G)

Table 3.4: Truth table for the *COSL* SR latch (first attempt), corresponding to Figure 3.11.

3.6.2 Second attempt

The physical realisation of the complicated layout of the inductive coupling used in the S-R latch in the previous section proved to be a problem. This led to a different approach for a subsequent implementation of the S-R latch. When the input *SQUID* of a *COSL* gate has the appropriate inputs, the Josephson junction switches to the normal state and the input current is forced through the inductor. When the sinusoidal clock current becomes less than the minimum current, $I_{\min}$, mentioned in Section 2.3, the junction returns to the superconducting state and the inductor current returns to its minimum value. If, on the other hand, the clock current does not reach $I_{\min}$, the junction remains normal and the current in the inductor is sustained. An output pulse will be generated in phase with the output clock as long as this current is present.

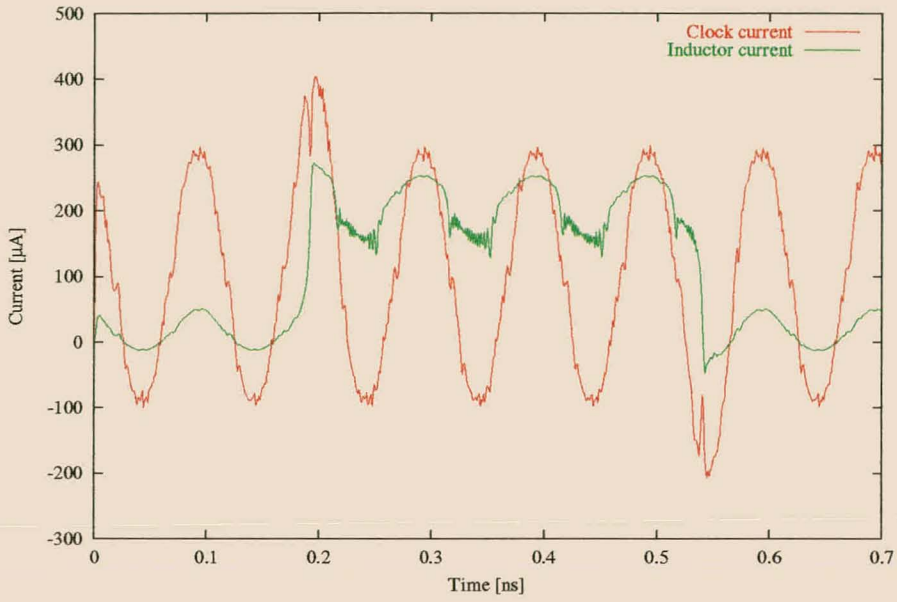


Figure 3.12: Simulated input clock current and inductor current of the *COSL* S-R latch (second attempt).

The *COSL* AND gate is used as the basis of the S-R latch presented in this section. A DC offset of 100 μA is added to the clock current by applying 5 mV to the 50 Ω trim resistor. This raises the negative peak of the clock above $I_{\min}$ and prevents the input *SQUID* from switching back to the superconductive state once the latch has been set. Due to the DC offset and the logic function of the AND gate, only one additional 100 μA input is needed to set the latch. Such an input causes a sustained inductor current as described above and the latch is set. This inductor current is compared to the input clock current in Figure 3.12.

The latch is reset by a negative input pulse. This pulse, added to the clock current, supplies enough negative current to exceed $I_{\min}$, which switches the input *SQUID* back to the superconductive state and resets the latch. The behaviour of the inductor current in this case is also shown in Figure 3.12.

The negative pulse needed to reset the latch is generated by a negative OR gate as described

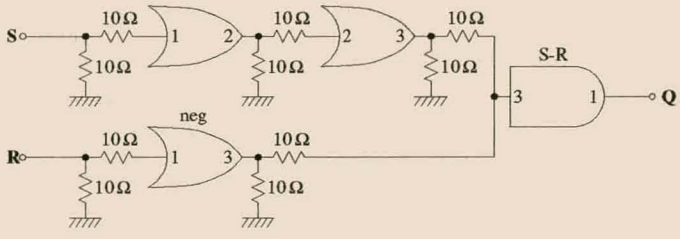


Figure 3.13: Input circuit of the *COSL* S-R latch (second attempt). Clock phases, as well as input and connecting resistors are shown.

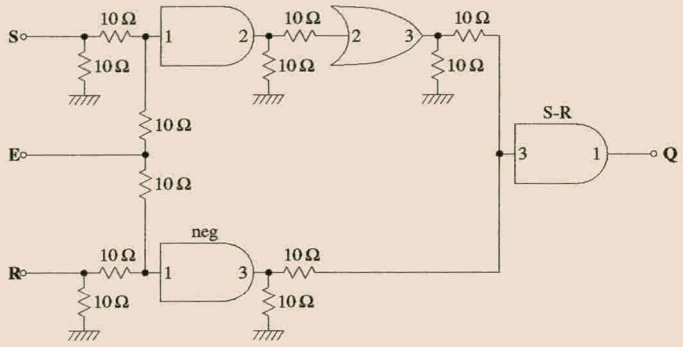


Figure 3.14: Input circuit of the *COSL* S-R latch (second attempt) with enable. Input and connecting resistors and clock phases are indicated.

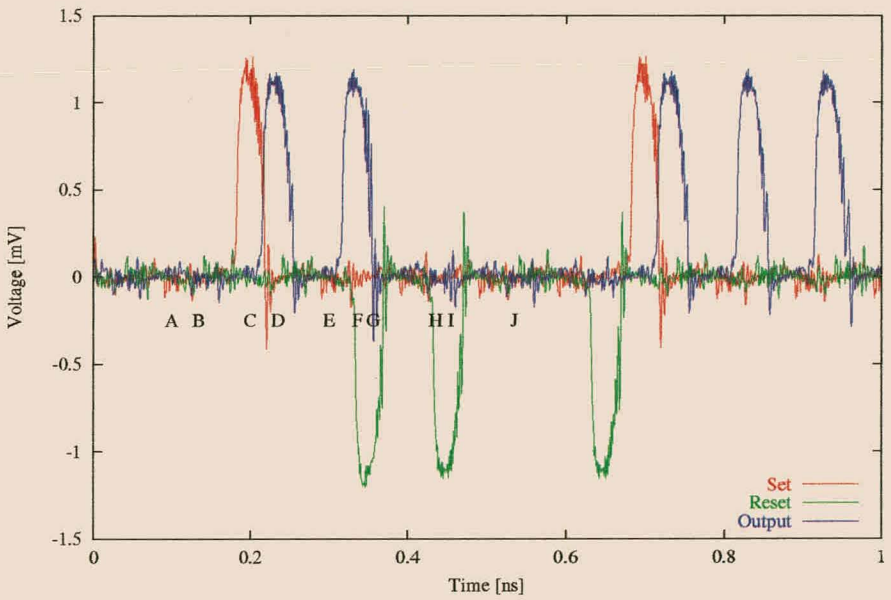


Figure 3.15: Simulated behaviour of the *COSL* S-R latch. The delay through the latch is one third of one clock period. The input circuit adds an additional delay of two thirds of a clock period. Table 3.5 gives the corresponding truth table.

Inputs Set,Reset		Output	
0,0	(A)	0	(B)
1,0	(C)	1	(D)
0,0	(E)	1	(F)
0,1	(G)	0	(H)
0,1	(I)	0	(J)

Table 3.5: Truth table for the *COSL* SR latch (second attempt), corresponding to Figure 3.15.

in Section 3.5. Due to the phase sequence of this gate, the set pulse for the latch is delayed by two positive OR gates. The input circuit for the S-R latch is shown in Figure 3.13. Input and connecting resistors, as well as clock phases, are shown.

Similarly to the S-R latch from Section 3.6.1, an enable function can be added by the replacement of the input OR gates with AND gates. This variation of the latch is depicted in Figure 3.14. Relevant clock phases and resistors are shown again.

The same inputs used for the S-R latch in the previous section are used to obtain the simulated response of this S-R latch. Figure 3.15 shows that the delay between the input and output is one third of one clock period. An additional delay of two thirds of one clock period is added by the input circuit. Table 3.5 gives the corresponding truth table.

3.7 D latch

The D, or data, latch is the most common latching component of clocked logic circuits. A clock input controls the latching of the device. The presence of a logical high clock signal makes the latch transparent. If the clock signal is removed, the current input value is latched.

The D latch can simply be implemented with the S-R latch with enable described in the previous section [20, p. 359]. The set and reset inputs are generated by the single data input by buffering and inverting respectively, as illustrated by Figure 3.16. Clock phases are shown, but input and output resistors are omitted. The clock input, which drives the enable input of the S-R latch, is also buffered to ensure synchronisation between clock and data.

The simulated response of the D latch is given in Figure 3.17. The total delay of the latch is one and one third of one clock period. The corresponding truth table is shown in Table 3.6.

3.8 Conclusions

Design considerations are presented for several new *COSL* building blocks. Slight modifications to the basic *COSL* gate, feedback and changes to the standard sequence of the three-phase clocking scheme facilitate the implementation of an inverter, NOR and NAND gate and T, S-R and D latch. Simulation results are provided, which verify the correct operation of these circuits.

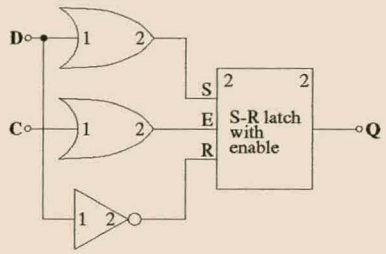


Figure 3.16: *COSL* D latch. Clock phases are shown, but input and output resistors are omitted.

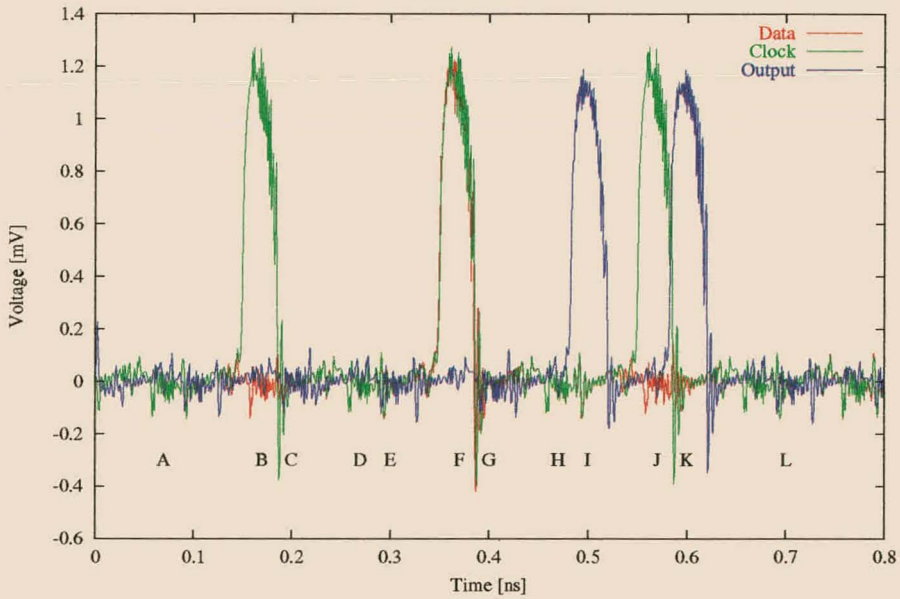


Figure 3.17: Simulated response of the *COSL* D latch. The delay between input and output is one and one third of a clock period. Table 3.6 gives the corresponding truth table.

Inputs Clock,Data		Output	
0,0	(A)	0	(C)
1,0	(B)	0	(E)
0,0	(D)	0	(G)
1,1	(F)	1	(I)
0,0	(H)	1	(K)
1,0	(J)	0	(L)

Table 3.6: Truth table for the *COSL* D latch, corresponding to Figure 3.17.

Chapter 4

Monte Carlo Yield Analysis and Optimisation

4.1 Introduction

In order to increase the probability that the circuits proposed in the previous chapter function correctly, a Monte Carlo method is used to predict their theoretical yield. Process variations are simulated by random variation of circuit parameters. Differences in parameters between chips, as well as local variations between components on the same chip, are taken into account. The circuits are simulated with these parameters and the process repeated a number of times. The success rate determines the yield of the circuits.

This chapter presents the Monte Carlo analysis results for the proposed circuits.

4.2 Analysis procedure

The parameters used in the Monte Carlo analysis are resistance, inductance and critical current. The effect of process variations are simulated by multiplying the nominal values of these parameters by Gaussian random variables. The global deviation is determined before each Monte Carlo cycle, while a different local deviation is generated for each component. The Monte Carlo analysis determines the observed yield y' of the circuit. The true statistical yield y is given by [3]

$$y = y' \pm \mathcal{L}, \tag{4.1}$$

where the confidence interval $\mathcal{L}$ is defined as

$$\mathcal{L} = k_c \sqrt{\frac{y'(1-y')}{N}}. \tag{4.2}$$

The constant k_c depends on the required confidence level of prediction and N is the number of Monte Carlo cycles. The value of k_c varies between 2 for a 95 % confidence level and 2.6 for a 99 % confidence level.

The circuits are manufactured with the HYPRES all niobium process. The 1 kA/cm^2 and 2.5 kA/cm^2 processes are used, but the circuits are optimised for the 2.5 kA/cm^2 process. The global variations in critical current density and resistance are specified by the HYPRES design rules as ± 15 % and ± 20 % respectively [21]. The global variation in inductance values was estimated as ± 8.5 %, but a worst case value of ± 15 % is used [19]. Local variations of ± 10 % are used throughout [3]. In all cases these variations are used as the 3σ parameter spreads for the Gaussian random variables.

At the beginning of each Monte Carlo cycle, the global variations in the relevant parameters are calculated. A separate Josephson junction model is used for the junctions that are varied during the analysis. The global tolerance in the critical current density is applied to the maximum critical current of this model.

The nominal input *SQUID* threshold current is determined from (2.9) with $\beta_L = 2\pi$ [22], as well as the nominal bias current. The ratio between these currents reflects the optimal bias for switching the particular gate. The effects of the global variations are taken into account by recalculating the bias current and input *SQUID* threshold current. The latter depends on the value of β_L, which is determined with the effective inductance. The effective inductance is given by (A.7) in Appendix A. The trim voltage needed to maintain the ratio is calculated for the given global variation. The reason for the trim is to include the possibility that a non-functional circuit can be made to work by a slight adjustment of the input current.

Different local variations are applied to each circuit element subject to the yield analysis. The local tolerance in critical current is reflected by variation of the area of the Josephson junction in question.

The input to the circuit being analysed is supplied by nominal gates. The output is measured for all possible input combinations. A circuit fails if, for any input combination, the expected output is not generated. A Monte Carlo run is considered successful if the output of the circuit is able to switch a nominal OR gate. The yield is calculated with (4.1) and (4.2) for both the trimmed and untrimmed cases. Appendix A contains a sample of the WRspice code used for the Monte Carlo analysis.

4.3 Results

The yield is predicted for a 99 % confidence level, 399 Monte Carlo cycles and a 10 GHz clock frequency. In order to minimise the analysis time, only the circuit parameters of the

Circuit	Yield [%]	
	Untrimmed	Trimmed
Inverter	66.9±6.1	70.7±5.9
NOR	82.5±5.0	97.0±2.2
NAND	65.4±6.2	82.5±5.0
T latch	46.9±6.5	75.2±5.6
Initial S-R latch	42.9±6.4	69.2±6.0
S-R latch	92.2±3.5	80.2±5.2
D latch	70.9±5.9	90.0±3.9

Table 4.1: Theoretical yield of the proposed circuits.

particular gates performing a function are varied. In the case of the D latch, therefore, the yield is determined by quite a complex circuit. Input and output circuit parameters remain at their nominal values. Circuit parameters are modified for optimal yield. The results of the yield prediction is presented in Table 4.1.

4.4 Conclusions

From the results presented above, it is evident that trimming significantly improves yield. An exception is the second S-R latch, which shows quite a high yield without trimming. The automatic calculation of the trim voltage lowered the yield of the circuit considerably. This means that this S-R latch is not sensitive to the ratio between the input current and the input *SQUID* threshold current. Manual trimming should increase the yield further. Most gratifying is the yield of the D latch, which, despite the complexity of the circuit, is quite high. The yield of the inverter is relatively low, because of its sensitivity to variation of the critical current of the additional Josephson junction. The complex inductive coupling used in the first S-R latch also proved to be sensitive to variations.

Chapter 5

Physical Layout

5.1 Introduction

A very substantial part of the development process of a superconducting digital circuit is the Very Large Scale Integration, or *VLSI*, layout of the circuits. The accuracy of this process ultimately determines the success or failure of the circuit.

The circuits are laid out with Xic [23], a companion package to the WRspice simulator. This chapter describes the layout process of *COSL* circuits.

5.2 HYPRES process

The basic layers of the HYPRES process are shown in Figure 5.1, together with the layers needed to define a Josephson junction. M0 to M3 are superconducting niobium layers, separated by a silicon dioxide dielectric. M0 is the ground plane. M0 definitions in a layout indicate areas where the groundplane is to be etched away. Vias between the niobium layers are defined

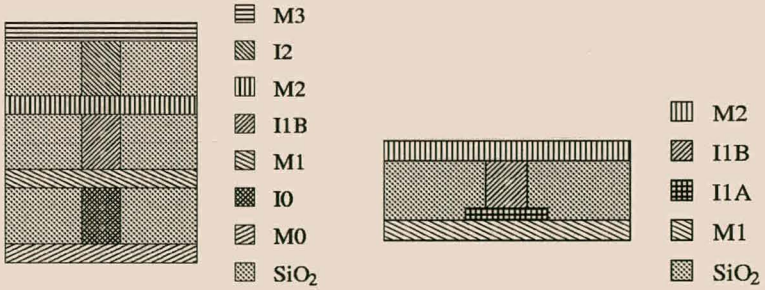


Figure 5.1: Cross sections of (a) the basic layers and (b) the Josephson junction definition of the HYPRES process.

Layer	Bias [μm]
M0	-0.5±0.25
I0	0.0±0.25
M1	-0.9±0.25
I1A	0.0±0.25†
R2	0.0±0.5
I1B	0.0±0.25
M2	-0.5±0.25
I2	0.0±0.25
M3	-0.75±0.25

†For dimensions $\geq 5.75\mu$m. HYPRES specifies specific values for dimensions $< 5.75\mu$m [21].

Table 5.1: Wafer to mask bias specified by the HYPRES process.

with I0, I1B and I2. The area of I1A definitions determines the critical current of the Josephson junction.

A resistive molybdenum layer, R2, which lies between M1 and M2, is used to form resistors. The sheet resistance of this layer is 1 $\Omega/\square$. This implies that the resistance is given by the ratio between the length and width of the R2 line. I1B vias are used to make contact from M2 to R2.

All layers except I1A and I1B use a grid size of 0.5 μm. The grid size for these remaining layers is specified as 0.25 μm. The bias from mask to wafer is detailed in Table 5.1. Complete specifications can be found in the design rules supplied by HYPRES [21].

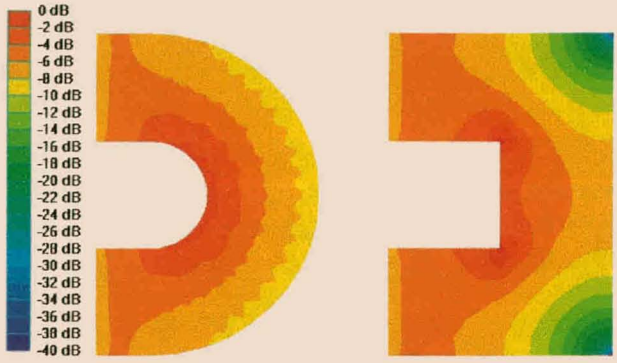
5.3 Current distribution

The *COSL* gates are laid out to maximise the uniformity of current distribution in the transmission lines. Current concentrations could cause undesirable field concentrations in the gates, as well as heating effects. All right angles are replaced with curves and square entities like vias and Josephson junctions are made circular. Examples can be seen in Figure 5.3, 5.4 and 5.5.

The current distribution in a section of a resistor was obtained by simulation with IE3D [24], an electromagnetic field simulation package. The results for the two geometries are compared in Figure 5.2. It is clear that the curved geometry provides a better current uniformity than the angular configuration.

5.4 Resistors

All resistors in the *COSL* circuits are realised with 6 μm wide R2 lines. Straight and semicircular sections are used to approximate the desired resistance values as closely as possible. The



(a) (b)

Figure 5.2: Current distribution in different resistor geometries.

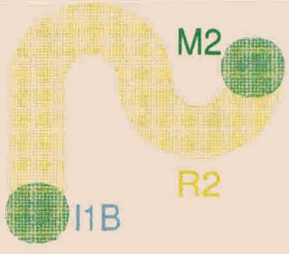


Figure 5.3: 8.8 Ω *COSL* OR gate bias resistor.

resistance of the semicircular line sections are approximated in terms of their length, measured on the center of the line. A semicircular section with a center radius of 6 μm has a length of 6π μm, which corresponds to a resistance of π Ω. M2 transmission lines are used as inputs and outputs.

The accuracy of a resistor value for a given geometry is checked with a parameter extraction tool included in the WRspice package. As an example of the resistor geometry, the 8.8 Ω *COSL* OR gate bias resistor is, with M2 contacts, in Figure 5.3. Wrspice gives the value of this resistor as 8.729 Ω.

5.5 Inductance calculation

The inductive coupling between the input and output *SQUID*s of *COSL* gates is accomplished by the use of an M3 transmission line which is placed above an M2 transmission line. The physical layout of the inductor pair not only depends on the desired mutual inductance, but on the respective self-inductances of the transmission lines as well.

The inductance per unit length of a superconducting strip transmission line is approximately

given by [25]

$$L \approx \frac{\mu_0}{wK}\left[h + \lambda_l \coth\left(\frac{t_l}{\lambda_l}\right) + \lambda_g \coth\left(\frac{t_g}{\lambda_g}\right)\right], \tag{5.1}$$

where w is the width, h the height above the ground plane, t the thickness and λ the London penetration depth of the transmission line. The London penetration depth of niobium is 900 Å or 0.09 μm. The transmission line and ground plane are denoted by the subscripts l and g respectively. K is the fringe field factor, which can be approximated by

$$K \approx 1 + \frac{k_s h}{w}, \tag{5.2}$$

where k_s is a scaling factor. For most practical cases the value of k_s can be taken as 4 [26], but better accuracy has been obtained by using different values of k_s for each of the layers of the HYPRES process [27].

The second and third term in (5.1) can be considered as modified London penetration depths of the form

$$\lambda^* = \lambda \coth\left(\frac{t}{\lambda}\right). \tag{5.3}$$

With this substitution (5.1) reduces to

$$L \approx \frac{\mu_0(h + \lambda_1^* + \lambda_2^*)}{wK}. \tag{5.4}$$

Substitution of (5.2) into (5.4) allows the mutual inductance per unit length between two transmission lines to be written as

$$M = k\mu_0 \sqrt{\frac{h_1 + \lambda_{l1}^* + \lambda_g^*}{w_1 + k_{s1}h_1} \frac{h_2 + \lambda_{l2}^* + \lambda_g^*}{w_2 + k_{s2}h_2}}, \tag{5.5}$$

where the subscripts 1 and 2 denote the respective transmission lines and k is the coupling coefficient.

(5.4) and (5.5) are solved for the desired values of L_1, L_2 and M, where L_1 and L_2 represent the self-inductance per unit length of the two transmission lines and M is the mutual inductance per unit length between them. The respective widths of the transmission lines are then written as

$$w_1 = \mu_0 k^2 \frac{L_2}{M^2}\left(h_1 + \lambda_1^* + \lambda_g^*\right) - k_{s1}h_1 \tag{5.6}$$

and

$$w_2 = \mu_0 k^2 \frac{L_1}{M^2}\left(h_2 + \lambda_2^* + \lambda_g^*\right) - k_{s2}h_2. \tag{5.7}$$

The lengths of the transmission lines are calculated with a numerical algorithm [28] implemented in C^{++}. This algorithm is already used in an existing inductance calculation program

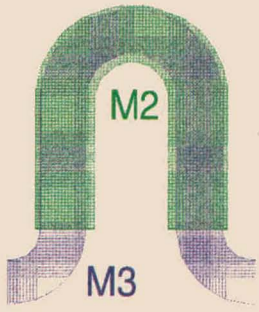


Figure 5.4: *COSL* mutual inductor.

[29], written in *C*. This code was rewritten, mostly for the enlightenment of the author, to make use of a C^{++} matrix manipulation library [30]. Appendix B contains the complete C^{++} code. In addition to the standard matrix input, a more user friendly graphical input method was added. Examples of both methods of input are also given in Appendix B. The functionality of the program was also expanded to include (5.6) and (5.7).

Comparison between the output of both programs for the same structure verified the correctness of the new code. Unfortunately, this also lead to an error in subsequent calculations. A penetration depth of 850 Å is used as the default value in the original program and was also used in the new code. This incorrect value was accidentally left as the default in the new program.

For *COSL* the inductance values L_1 and L_2 are 4.4 pH and 10.3 pH respectively and the mutual inductance M between them 3.905 pH [3]. With these values, a nominal length of 47.8 μm and the incorrect penetration depth, (5.6) and (5.7) give $w_1 = 6.113$ μm and $w_2 = 4.203$ μm. With the grid size taken into account, these values become $w_1 = 6$ μm and $w_2 = 4.25$ μm.

With the correct penetration depth, a length of 53 μm and the snapped values the inductances values are calculated as 4.709 pH, 10.527 pH and 3.899 pH for L_1, L_2 and M respectively. The deviation from the nominal values resulting from the incorrect penetration depth, as well as the grid size, is 7.026 %, 2.205 % and 0.135 % for L_1, L_2 and M respectively. The final layout of the inductor pair used in the *COSL* gates is shown in Figure 5.4. As in Section 5.4, the radii used to calculate the lengths of the semicircular sections are measured to the center of the respective transmission lines.

5.6 Flux trapping

If stray magnetic flux is present when a superconducting circuit is cooled through the critical temperature this flux may become trapped in the groundplane of the circuit. When this trapped

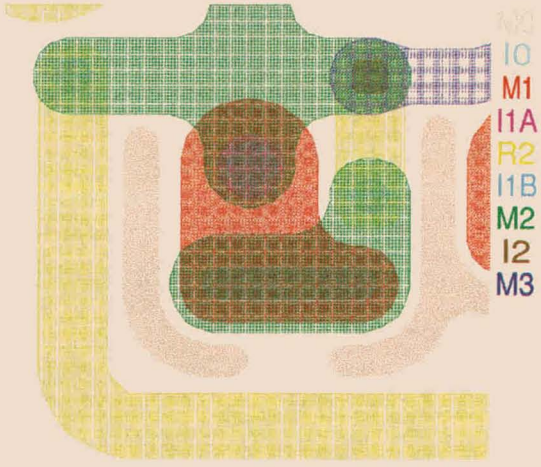


Figure 5.5: Moats surrounding a section of the input *SQUID* of a *COSL* OR gate.

flux is close to a Josephson junction, or it couples into a *SQUID* loop, it can significantly reduce circuit performance. So-called moats have been proposed to alleviate the problem [31]. These moats, which are narrow channels in the groundplane, are placed around sensitive areas in a circuit. They provide a low energy site where flux can be trapped, away from sensitive areas.

It has been found that moats surrounding a small area provide more effective protection at higher magnetic fields [32]. In the layout of the proposed circuits, the moats are, therefore, placed as close as possible to the sensitive Josephson junctions and the enclosed area kept as small as possible. As an example, the moats surrounding the Josephson junction of a *COSL* OR gate input *SQUID* is shown in Figure 5.5.

5.7 Clocking

The three phase clocking scheme should be considered during layout. Unbalanced clock phases can cause non-zero currents flowing in the groundplane, which, in turn, can cause the ground reference to bounce. Balancing is done by ensuring that the input impedance of all the clock phases is equal. Transmission line impedance matching and parallel resistors to ground are used to equalize the respective input impedances. The line width needed for a specific impedance is calculated with SLINE [33]. An extra measure of protection against ground bounce is the use of an additional clock phase, which is connected to the groundplane through a resistor. This clock phase can be tuned to minimise bounce.

The input impedance of the input and output clocks of each gate is calculated from the voltage at and current into the respective clock terminals. One such terminal is shown in Figure 5.6. The input impedance of the input clock is found to be $6\frac{2}{3}\ \Omega$ and that of the output clock $13\frac{1}{3}\ \Omega$.

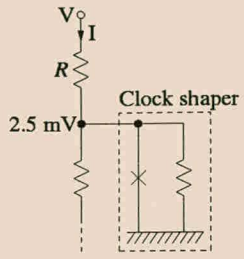


Figure 5.6: Clock terminal of a *COSL* gate.

The length of the lines supplying the clocks are made approximately the same to ensure that each clock phase arrives at the correct time, together with the input. Input lines are laid out to have the same approximate length.

5.8 Input and output matching

All the circuits are matched to 50 Ω inputs and outputs. The minimum width constraint imposed by the HYPRES design rules prevent the use of "normal" transmission lines for 50 Ω lines. Coplanar waveguide is therefore used [22]. Input matching is done by the resistive network in Figure 5.7. Realising that

$$R_i = R_1 + \frac{R_2 R_o}{R_2 + R_o} \tag{5.8}$$

and

$$R_o = \frac{R_2(R_1 + R_i)}{R_2 + R_1 + R_i} \tag{5.9}$$

and solving for R_1 and R_2 lead to

$$R_1 = \sqrt{R_i(R_i - R_o)} \tag{5.10}$$

and

$$R_2 = R_o\sqrt{\frac{R_i}{R_i - R_o}}. \tag{5.11}$$

R_i and R_o are 50 Ω and 5 Ω respectively.

An amplifier circuit [22], shown in Figure 5.8, is used to convert 1 mV into 5 Ω output signals to 2.5 mV into 50 Ω. Because of the 10 Ω output resistance of the T latch, OR buffers were added in the layout to ensure 5 Ω. The clock phase is the same as the output phase of the gate driving the amplifier.

5.9 Final layout

The final chip layout of a selection of the proposed circuits is shown in Figure 5.9. The T, S-R

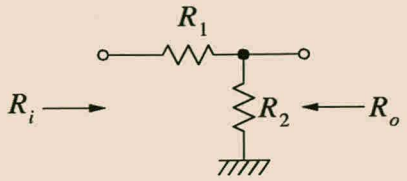


Figure 5.7: Resistive network used to match 50 Ω to 5 Ω.

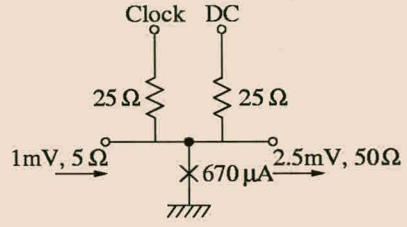


Figure 5.8: Amplifier circuit to convert 1 mV into 5 Ω signals to 2.5 mV into 50 Ω.

and D latch were laid out for the 1 kA/cm^2 and 2.5 kA/cm^2 HYPRES processes. Because of the large number of inputs needed for trimming, the extra clock phase mentioned in Section 5.7 was omitted. Figure 5.10 shows an enlarged view of the D latch used in the chip layout.

5.10 Conclusions

A number of considerations are important where the *VLSI* layout of *COSL* is concerned. A set of design rules to be obeyed is specified by HYPRES. Current uniformity in the gates, calculation of the dimensions of the *COSL* inductor pair, protection against magnetic flux trapping, clock balancing and input and output impedance matching are considered during the layout of a selection of the new building blocks.

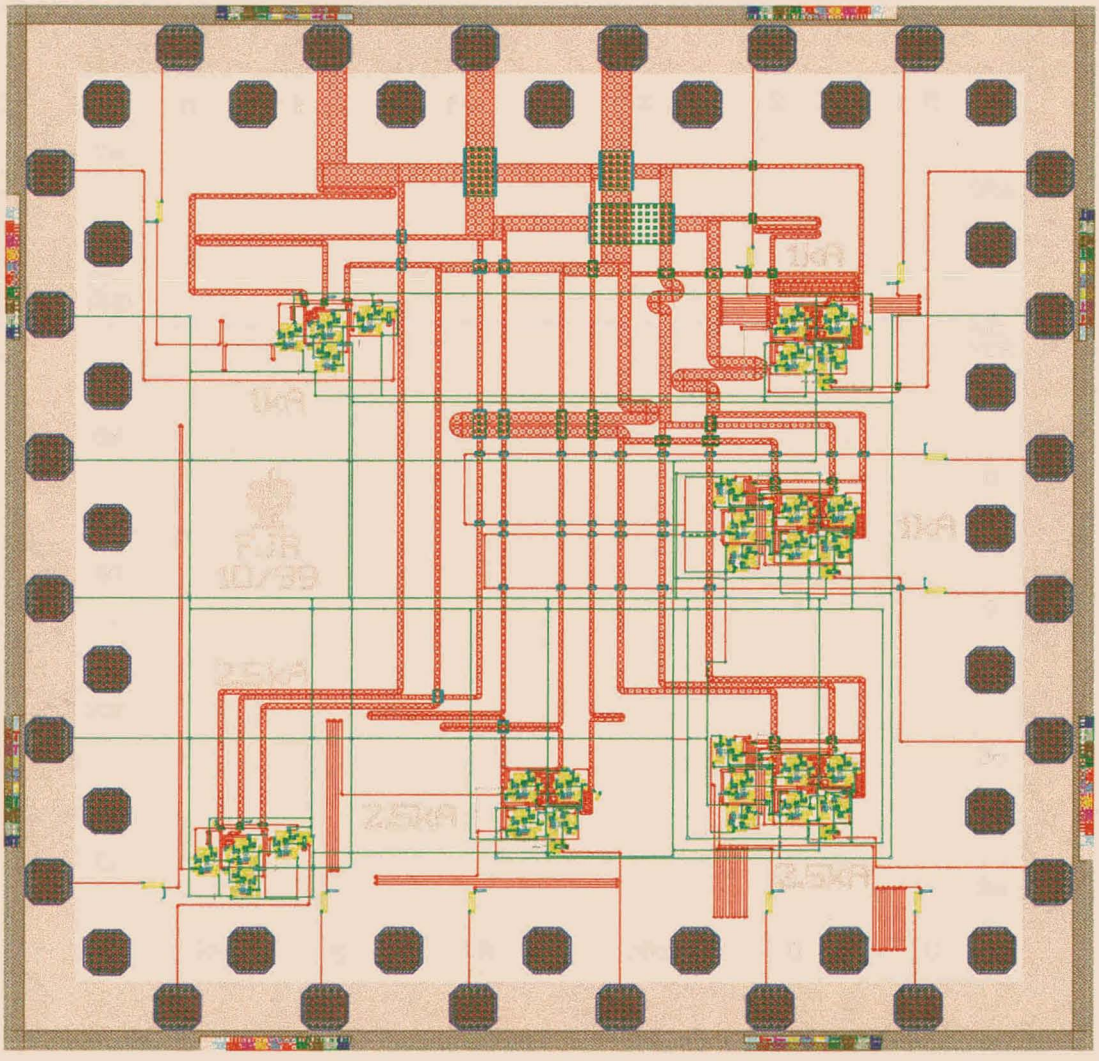


Figure 5.9: Final layout of the chip containing 1 kA/cm^2 and 2.5 kA/cm^2 versions of the T, S-R and D latch.

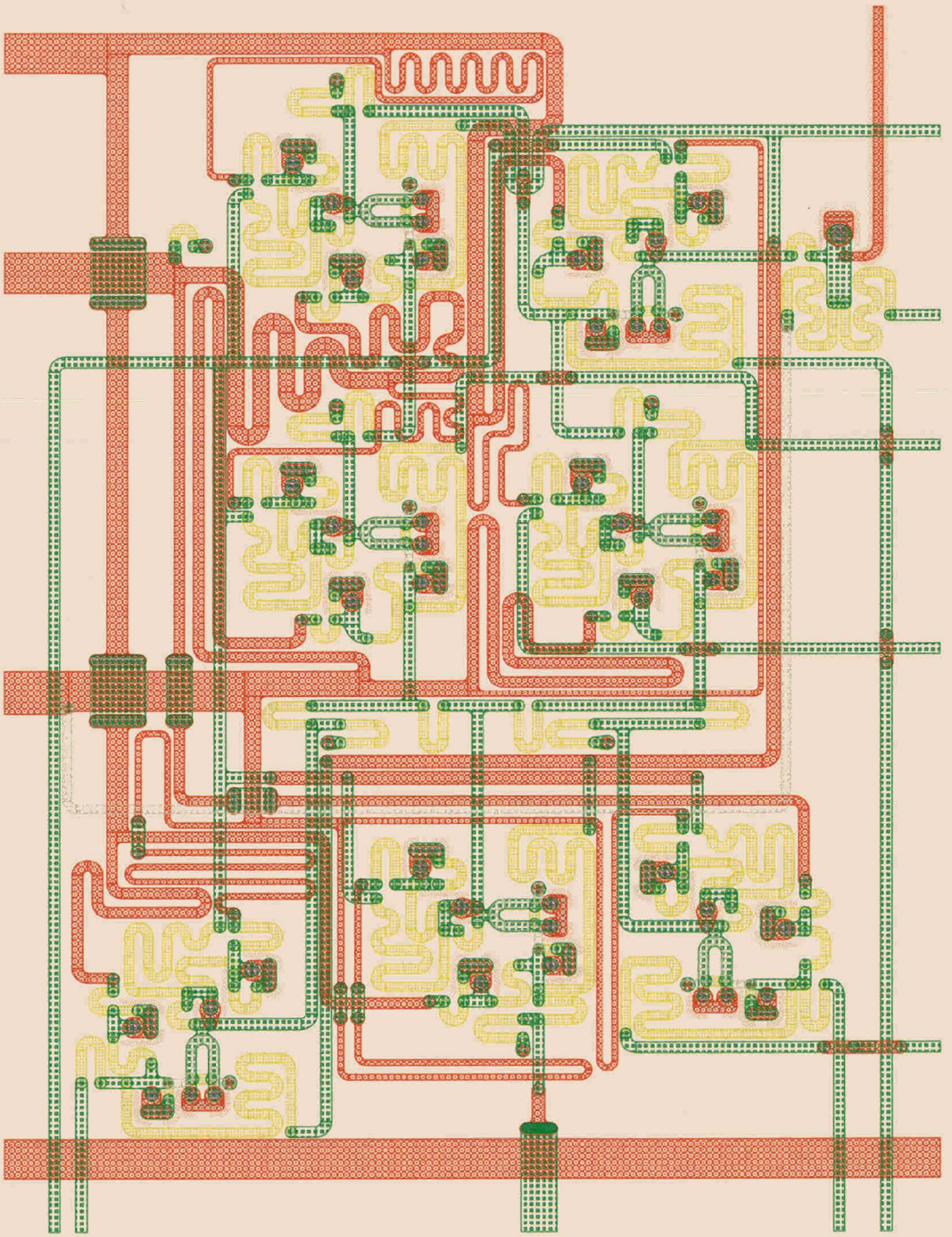


Figure 5.10: Enlarged view of the *COSL* D latch used in the chip layout.

Chapter 6

Conclusions

As an introduction, an overview of the principles of the Josephson junction, one-junction and two-junction *SQUID* and the basic operation of *COSL* gates were given. *COSL* was used to define new and alternative building blocks for use in ultra-high speed logic circuits. A new single gate inverter was proposed, as well as alternatives to the existing NOR and NAND gates. A number of latching functions were also described, namely a T, S-R and D latch. Two attempts at the realisation of the S-R latch were discussed. Correct operation of the circuits was verified by simulation and results presented. Circuit yield was analysed and optimised by a Monte Carlo technique. The *VLSI* layout of *COSL* circuits were discussed.

The untrimmed yield of the circuits were, in most cases, quite low. The S-R latch exhibited the highest untrimmed yield of 92.2 ± 3.5 %. This gate could prove useful for future applications. The highest trimmed yield, 97.0 ± 2.2 %, was obtained for the NOR gate. The D latch showed the most promising results, despite the complexity of the circuit. As mentioned in Section 4.3, the input circuit of ths latch was included in the Monte Carlo analysis, which makes the trimmed yield of 90.0 ± 3.9 % quite satisfactory.

The most noteworthy aspect of the NOR and NAND gate and S-R latch is that they were all realised by basic non-inverting *COSL* gates. By simply modifying the bias resistance and changing the clock sequence, a range of different functions could be obtained.

A selection of the circuits were laid out for manufacture. The T, S-R and D latch were laid out for the 1 kA/cm^2 and 2.5 kA/cm^2 HYPRES process. Testing of these circuits will take place at the University of California at Berkeley in the near future.

This project provided much needed insight into the operation of the Josephson junction and one-junction and two-junction *SQUID*. This knowledge will prove very useful for future research. Valuable experience in *VLSI* layout was also gained.

Bibliography

[1] D. A. Buck, "The cryotron — a superconductive computer component," *Proceedings of the Institute of Radio Engineers*, vol. 44, pp. 482–493, 1956.

[2] B. D. Josephson, "Possible new effects in superconducting tunneling," *Physical Review Letters*, vol. 1, p. 251, 1962.

[3] W. J. Perold, M. Jeffery, Z. Wang, and T. Van Duzer, "Complementary output switching logic—a new superconducting voltage-state logic family," *IEEE Transactions on Applied Superconductivity*, vol. 6, pp. 125–131, Sept. 1996.

[4] M. Jeffery, W. J. Perold, and T. Van Duzer, "Superconducting complementary output switching logic operating at 5-10Gb/s," *Applied Physics Letters*, vol. 69, pp. 2746–2748, Oct. 1996.

[5] W. J. Perold, "Complementary output switching logic: a superconducting voltage-state logic family operating at microwave frequencies," in *Proceedings of the 1998 South African Symposium on Communications and Signal Processing: COMSIG '98*, pp. 435–440, 1998.

[6] M. Jeffery, T. Van Duzer, and W. J. Perold, "Superconducting complementary output switching logic operating at 10-18GHz." 1998 March Meeting of the American Physical Society, Los Angeles, CA, 16-20 March 1998.

[7] A. Smith, S. Thomasson, and C. Dang, "Reproducibility of niobium junction critical currents: statistical analysis and data," *IEEE Transactions on Applied Superconductivity*, vol. 3, pp. 2174–2177, Mar. 1993.

[8] R. A. Serway, *Physics for scientists and engineers*. Saunders College Publishing, third ed., 1992.

[9] T. Van Duzer, "Superconductor electronic device aplications," *IEEE Journal of Quantum Electronics*, vol. 25, pp. 2365–2377, Nov. 1989.

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

[11] E. S. Fang and T. Van Duzer, "An efficient method for finding dc solutions for Josephson circuits," *IEEE Transactions on Applied Superconductivity*, vol. 1, pp. 127–133, Sept. 1991.

[12] K. K. Likharev, *Dynamics of Josephson junctions and circuits*. Gordon and Breach Science Publishers, 1986.

[13] T. Van Duzer and C. W. Turner, *Principles of superconductive devices and circuits*. Upper Saddle River, New Jersey 07458: Prentice Hall PTR, second ed., 1999.

[14] E. S. Fang, *A Josephson flash-type analog-to-digital converter and related topics in superconductive circuits*. Ph.D. dissertation, University of California at Berkeley, Nov. 1991.

[15] W.-T. Tsang and T. Van Duzer, "Dc analysis of parallel arrays of two and three Josephson junctions," *Journal of Applied Physics*, vol. 46, pp. 4573–4580, Oct. 1975.

[16] R. L. Peterson and C. A. Hamilton, "Analysis of threshold curves for superconducting interferometers," *Journal of Applied Physics*, vol. 50, pp. 8135–8142, Dec. 1979.

[17] G. S. Lee, "A simple physical principle for determining mode boundaries in superconducting loop circuits," *Journal of Applied Physics*, vol. 66, pp. 2732–2734, Sept. 1989.

[18] Whiteley Research Inc., 456 Flora Vista Avenue, Sunnyvale, CA 94086, *WRspice Circuit Simulation System*. Homepage: `http://www.srware.com`.

[19] M. Jeffery, W. J. Perold, Z. Wang, and T. Van Duzer, "Monte Carlo optimization of superconducting complementary output switching logic," *IEEE Transactions on Applied Superconductivity*, vol. 8, pp. 104–119, Sept. 1998.

[20] J. F. Wakerly, *Digital design: principles and practices*. Englewood Cliffs, New Jersey 07632: Prentice-Hall Inc., first ed., 1990.

[21] HYPRES, 175 Clearbrook Road, Elmsford, New York 10523, 1997. Design rules available via the HYPRES homepage: `http://www.hypres.com`.

[22] W. J. Perold. Private communication.

[23] Whiteley Research Inc., 456 Flora Vista Avenue, Sunnyvale, CA 94086, *Xic Integrated Circuit Design System*. Homepage: `http://www.srware.com`.

[24] Zeland Software Inc., 39120 Argonaut Way, Suite 499, Fremont, CA, *IE3D Version 3.0*, Jan. 1996.

[25] W.H.Chang, "The inductance of a superconducting strip transmission line," *Journal of Applied Physics*, vol. 50, pp. 8192–8134, Dec. 1979.

[26] N. Fujimaki, S. Kotani, T. Imamura, and S. Hasuo, "Josephson modified variable threshold logic gates for use in ultra-high-speed LSI," *IEEE Transactions on Electron Devices*, vol. 36, pp. 433–446, Feb. 1989.

[27] W. J. Perold, "Modeling superconducting components based on the fabrication process and layout dimensions." Unpublished.

[28] W.H.Chang, "Numerical calculation of the inductance of a multi-superconductor transmission line system," *IEEE Transactions on Magnetics*, vol. MAG-17, pp. 764–766, Jan. 1981.

[29] J. Fleischman, *Induct — inductance calculation program.* Available via the UC Berkeley cryoelectronics group homepage: `http://swordfish.eecs.berkeley.edu`.

[30] D. Weber, M. Sipe, R. Shenoy, *et al.*, *uMatrix — C++ matrix manipulation library.* Available via `http://espresso.ee.sun.ac.za/uMatrix`.

[31] S. Bermon and T. Gheewala, "Moat-guarded Josephson *SQUID*s," *IEEE Transactions on Magnetics*, vol. MAG-19, pp. 1160–1164, May 1983.

[32] M. Jeffery, T. Van Duzer, J.R.Kirtley, and M.B.Ketchen, "Magnetic imaging of moat-guarded superconducting electronic circuits," *Applied Physics Letters*, vol. 67, pp. 1769–1771, Sept. 1995.

[33] S. R. Whiteley, *SLINE Version 1.0*, June 1996. Available via the Whiteley Research homepage: `http://www.srware.com`.

Appendix A

Detail of Monte Carlo Analysis

A.1 WRspice code

```
COSL nor gate, Monte Carlo analysis
.monte
.exec
     checkSTP1=5
     checkSTP2=4
     let Jtol = gauss(0.15/3,1)
     let Rtol = gauss(0.1/3,1)
     let Ltol = gauss(0.1/3,1)
     let Ic = 250u
     let B = 2*pi
     let Arg = -1/B
     let Acos = -j(ln(Arg+j(sqrt(1-Arg^2))))
     let Ith_nom = -(Ic*sin(Acos)+Ic/B*Acos)
     let Rbias_nom = 6.5
     let Iin_nom = -2.5m/Rbias_nom
     let Ratio = Ith_nom/Iin_nom
     let L1_nom = 10.3p
     let L2_nom = 4.4p
     let k = 0.58
     let Phi0 = 2.06783461f
     let Lone = L1_nom*Ltol
     let Ltwo = L2_nom*Ltol
     let M = k*sqrt(Lone*Ltwo)*Ltol
```

```
      let Lj = Phi0/2/pi/0.2/Jtol/1m
      let Leff = Lone - M*M/(Ltwo+2*Lj)
      let Ic = 0.25*Jtol*1m
      let B = 2*pi*Leff*Ic/Phi0
      let Arg = -1/B
      let Acos = -j(ln(Arg+j(sqrt(1-Arg^2))))
      let Ith = -(Ic*sin(Acos)+Ic/B*Acos)
      let Rbias = Rbias_nom*Rtol
      let Iin = -2.5m/Rbias
      let Vtrim = 50*Rtol*(Ith/Ratio-Iin)
.endc
.control
      if t1 < 0.6m or t2 > 0.4m or t3 > 0.4m or t4 > 0.4m or t5 < 0.6m
            let checkFAIL=1
      end
.endc
.tran 1p 600p uic
.param Avar = gauss(0.1/3,1)
.param Rvar = Rtol*gauss(0.1/3,1)
.param Lvar = Ltol*gauss(0.1/3,1)
.measure tran t1 from=90p to=110p avg v(2)
.measure tran t2 from=190p to=210p avg v(2)
.measure tran t3 from=290p to=310p avg v(2)
.measure tran t4 from=390p to=410p avg v(2)
.measure tran t5 from=490p to=510p avg v(2)
.measure tran t1or from=120p to=140p avg v(20)
.measure tran t2or from=220p to=240p avg v(20)
.measure tran t3or from=320p to=340p avg v(20)
.measure tran t4or from=420p to=440p avg v(20)
.measure tran t5or from=520p to=540p avg v(20)
B0 8 2 27 jjvar area=$&(0.34*Avar)
B1 7 0 28 jjvar area=$&(0.35*Avar)
B2 9 0 29 jjvar area=$&(0.36*Avar)
B3 6 0 30 jjvar area=$&(0.25*Avar)
B4 4 0 31 jjvar area=$&(0.2*Avar)
```

```
B5 3 0 32 jjvar area=$&(0.2*Avar)
K1 L0 L2 0.58
K2 L1 L3 0.58
L0 6 5 $&(5.15p*Lvar)
L1 5 0 $&(5.15p*Lvar)
L2 3 2 $&(2.2p*Lvar)
L3 2 4 $&(2.2p*Lvar)
R0 7 0 $&(2.3*Rvar)
R1 8 2 $&(5*Rvar)
R10 26 19 10
R11 2 0 10
R12 17 6 10
R13 6 0 $&(1.5*Rvar)
R14 1 6 10
R15 17 0 10
R16 3 4 $&(1.6*Rvar)
R17 1 0 10
R18 2 21 10
R19 20 0 5
R2 7 6 $&(Rbias_nom*Rvar)
R3 23 6 $&(50*Rvar)
R4 10 9 $&(10*Rvar)
R5 11 7 $&(5*Rvar)
R6 9 0 $&(6*Rvar)
R7 9 8 $&(12*Rvar)
R8 24 8 $&(33*Rvar)
R9 25 16 10
V0 24 0 5m
V1 23 0 $&(Vtrim)
V2 25 0 pulse(0 1m 110p 10p 33p 11p 0 310p)
V3 26 0 pulse(0 1m 210p 10p 33p 11p 0 310p)
X0 15 clk2
X1 18 clk2
X10 14 clk1
X2 10 clk3
```

```
X3 22 clk3
X4 12 15 16 17 or
X5 13 18 19 1 or
X6 22 14 21 20 or
X7 12 clk1
X8 13 clk1
X9 11 clk1
.subckt or 8 7 4 1
     B0 6 1 12 jj2 area=0.34
     B1 5 0 13 jj2 area=0.35
     B2 10 0 14 jj2 area=0.36
     B3 4 0 15 jj2 area=0.25
     B4 2 0 16 jj2 area=0.2
     B5 9 0 17 jj2 area=0.2
     K1 L0 L1 0.58
     K2 L2 L3 0.58
     L0 3 0 5.15p
     L1 1 9 2.2p
     L2 4 3 5.15p
     L3 2 1 2.2p
     R0 5 0 2.3
     R1 6 1 5
     R2 5 4 8.8
     R3 7 10 10
     R4 8 5 5
     R5 10 6 12
     R6 11 6 33
     R7 10 0 6
     R8 4 0 1.5
     R9 2 9 1.6
     V0 11 0 5m
.ends or
.subckt clk1 1
     V0 1 0 sin (0 10m 10g)
.ends clk1
```

```
.subckt clk2 1
     V0 1 0 sin (0 10m 10g -66.6666666667p)
.ends clk2
.subckt clk3 1
     V0 1 0 sin (0 10m 10g -33.3333333333p)
.ends clk3
.model jj2 jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=1m, r0=30, rn=1.64706, cap=1.54894p)
*Nb 2500 A/cm2   area = 40 square microns (generated by JJMODEL)
.model jjvar jj(rtype=1, cct=1, icon=10m, vg=2.8m, delv=0.08m,
+ icrit=$&(Jtol*1m), r0=30, rn=1.64706, cap=1.54894p)
.end
```

A.2 Effective inductance

The equivalent circuit for the calculation of the effective inductance is shown in Figure A.1.

The input and output voltages are given by

$$v_1 = L_1 \frac{di_1}{dt} + M \frac{di_2}{dt} \tag{A.1}$$

and

$$v_2 = M \frac{di_1}{dt} + L_2 \frac{di_2}{dt} \tag{A.2}$$

$$= -2L_J \frac{di_2}{dt}, \tag{A.3}$$

where

$$L_J = \frac{\Phi_0}{2\pi I_c \cos\phi} \tag{A.4}$$

is known as the Josephson inductance [10, p. 470] and reflects the contribution of the Josephson

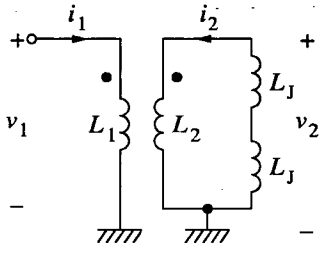


Figure A.1: Equivalent circuit for the calculation of the effective inductance of a *COSL* input *SQUID*.

junctions in the two-junction *SQUID*. (A.2) and (A.3) lead to

$$\frac{di_2}{dt} = -\frac{M}{L_2 + 2L_J}\frac{di_1}{dt}, \tag{A.5}$$

which, upon substitution into (A.1), give the effective inductance as

$$L_{\text{eff}} = L_1 - \frac{M^2}{L_2 + 2L_J}. \tag{A.6}$$

The effective inductance is calculated at the beginning of a Monte Carlo cycle, before any of the dynamics of the circuit is known. Hence, the phase dependence of the Josephson inductance is indeterminable. As an approximation, the phase difference of the junction is considered small enough to make the $\cos\phi$ term in (A.4) approximately 1. This leads to

$$L_{\text{eff}} = L_1 - \frac{M^2}{L_2 + \frac{\Phi_0}{\pi I_c}}. \tag{A.7}$$

Appendix B

C^{++} Inductance Calculation Program

B.1 C^{++} source code

B.1.1 Induct.h

```
/* This program implements an algorithm to calculate the inductance matrix of
 a multi-superconductor transmission line system. The algorithm was proposed
 by Chang ( IEEE Trans. on Magnetics, vol. MAG-17, no. 1, p. 764). */

#include <fstream>
#include <iomanip>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <string>
#include <uLapack.h>
#include <uMatrix.h>
/***************************/
/* Miscellaneous constants */
/***************************/
/* Permeability of free space ( scaled to get pH/um) */
#define Uo 1.2566370614
/* Permittivity of free space ( scaled to get pF/um) */
#define Eo 8.854187817e-6
/* Relative permittivity of SiO2 */
#define Er 3.9
#define PI 3.14159265358979
/* Half pi */
#define hPI 1.57079632679490
#define Kp  Uo/96/PI
/* Inductances in pH */
#define L1 4.4
#define L2 10.3
/* Coupling constant between L1 and L2 */
#define K 0.58
/* Mutual inductance between L1 and L2 */
#define Mind K*sqrt(L1*L2) /* pH */
/* Constant defined for use with atan() */
#define SMALL 8.8817841971e-16
/* Hypres layer thicknesses in um */
#define M0M0 0.1
#define M1M0 0.15
#define M1M1 0.2
#define M2M0 0.35
#define M2M2 0.3
#define M3M0 0.85
```

```
#define M3M3 0.6
/* Penetration depth of niobium */
#define PD 0.09
/* Number of x and y segments per (part of) conductor */
#define X 10
#define Y 5
/* Layer constants for width calculation */
#define kM1 4
#define kM2 3.52
#define kM3 3.13
/* Return the minimum and maximum of X and Y */
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
/**********************/
/* Type declarations */
/**********************/
/* Scale factors for Magic and Xic dimensions */
enum ciffiletype {none=1, magic=400, xic=1000};
/* Sort direction for Sort function */
enum sortdirtype {up, down};
/* Layers used in LayerWidth function */
enum inductortype {m2m3, m1m2, m1m3};
/**************************/
/* Procedure declarations */
/**************************/
void AddRow(uMatrix<double> &,
            const double []);

uMatrix<double> LineWidth(const inductortype);

uMatrix<double> Parse(const int,
                      char *);

double Pint(const double,
            const double,
            const double,
            const double);

uMatrix<double> Pmn(const int,
                    const uMatrix<double> &,
                    const uMatrix<double> &,
                    const uMatrix<double> &);

uMatrix<double> Qmk(const int,
                    const uMatrix<double> &,
                    const uMatrix<double> &,
                    const uMatrix<double> &);

uMatrix<double> Reduce(int,
                       uMatrix<int>,
                       uMatrix<double>,
                       uMatrix<int>,
                       int);

void Sort(uMatrix<double> &,
          const int,
          const sortdirtype sortdir);

void Subdivide(const uMatrix<double> &,
               int &,
               int &,
               uMatrix<int> &,
               uMatrix<double> &,
               uMatrix<double> &,
               uMatrix<int> &,
               uMatrix<double> &,
               uMatrix<double> &);
```

```
double Sum(const uMatrix<double> &);

void Swop(uMatrix<double> &,
          const int,
          const int);

/* End of Induct.h */
```

B.1.2 Parse.cc

```
#include "Induct.h"

uMatrix<double> Parse(const int mainargc,
                      char *mainargv[])
  /* Parses the commandline and prepares the input matrix to be used in
     calculation. */
{
  uMatrix<double> pinput(0,0);
  switch (mainargc)
    {
    case 3:
      if (string(mainargv[1]) == "-u")
        {
          ifstream infile(mainargv[2], ios::in);
          if (!infile)
            {
              cerr << "Cannot open file "
                   << mainargv[2]
                   << " for input" << endl;
              exit(-1);
            }
          pinput.readAscii(mainargv[2]);
          if (pinput.rows() < 2)
            {
              cerr << "Cannot calculate inductance matrix for only one conductor\n";
              exit(-1);
            }
          else
            {
              pinput.markAsTemporary();
              return pinput;
            }
        }
      else if (string(mainargv[1]) == "-c")
        {
          ifstream infile(mainargv[2], ios::in);
          if (!infile)
            {
              cerr << "Cannot open file "
                   << mainargv[2]
                   << " for input" << endl;
              exit(-1);
            }
          string buf;
          ciffiletype cifscale = none;
          int layer = 0, cifcn = 0;
          double cifcoor, cifheight;
          while (infile >> buf)
            {
              if (strstr(buf.c_str(), "xic"))
                cifscale = xic;
              else if (strstr(buf.c_str(), "Magic"))
                cifscale = magic;
              if (buf == "L")
                {
                  infile >> buf;
```

```
            if ( buf == "M0;")
              layer = 0;
            else if ( buf == "M1;")
              layer = 1;
            else if ( buf == "M2;")
              layer = 2;
            else if ( buf == "M3;")
              layer = 4;
          }
        if ( buf == "B")
          {
            cifcn ++;
            infile >> cifheight ; /* dummy read */
            infile >> cifheight ;
            cifheight /= cifscale ;
            infile >> cifcoor ; /* dummy read */
            infile >> cifcoor ;
            cifcoor /= cifscale ;
            if ( pinput.rows() > 0)
              {
                uMatrix<double> tempinput = pinput ;
                pinput . resize ( pinput . rows ()+2,3);
                pinput . insert ( uIndex (0,1, tempinput . rows ()-1), uIndex (0,1,2), tempinput );
              }
            else
              pinput . resize ( pinput . rows ()+2,3);
            pinput ( pinput . rows ()-2,0) = double( cifcn );
            pinput ( pinput . rows ()-2,1) = double( layer );
            pinput ( pinput . rows ()-2,2) = cifcoor - cifheight /2;
            pinput ( pinput . rows ()-1,0) = double( cifcn );
            pinput ( pinput . rows ()-1,1) = double( layer );
            pinput ( pinput . rows ()-1,2) = cifcoor + cifheight /2;
          }
      }
infile . close ();
if ( pinput . rows () == 0)
  {
    cerr << "Input file is probably not a CIF file\n";
    exit (-1);
  }
Sort ( pinput ,2, up); /* sort on coordinates */
bool M0 = false , M1 = false , M2 = false , M3 = false , multileft = false ;
int M0cn = 0, M1cn = 0, M2cn = 0, M3cn = 0;
double leftcoor = 0, rightcoor = pinput (0,2);
uMatrix<double> modinput (0,8);
for ( int i=0; i<pinput . rows ()-1; i++)
  {
    int s = 1;
    while (( rightcoor == pinput (i,2)) && ( i+s < pinput . rows ()))
       {
        rightcoor = pinput (i+s,2);
        s++;
      }
    if (( pinput (i,2) != leftcoor ) || ( pinput (0,2) == 0))
      {
        leftcoor = pinput (i,2);
      }
    if ( s > 2)
      multileft = true;
    else if ( pinput (i+1,2) != leftcoor )
      multileft = false;
    if ( leftcoor == rightcoor )
      multileft = true;
    double newblock [8];
    switch ( int( pinput (i,1))) /* switch on layer */
      {
      case 0:
```

```
      M0cn = int(pinput(i,0));
      M0 = !M0;
      break;
    case 1:
      M1cn = int(pinput(i,0));
      M1 = !M1;
      break;
    case 2:
      M2cn = int(pinput(i,0));
      M2 = !M2;
      break;
    case 4:
      M3cn = int(pinput(i,0));
      M3 = !M3;
    }
  if (!multileft && M0)
    {
      newblock[0] = M0cn;
      newblock[1] = leftcoor;
      newblock[2] = 0;
      newblock[3] = rightcoor;
      newblock[4] = M0M0;
      newblock[5] = PD;
      newblock[6] = X;
      newblock[7] = Y;
      AddRow(modinput, newblock);
    }
  if (!multileft && M1)
    {
      newblock[0] = M1cn;
      newblock[1] = leftcoor;
      newblock[2] = M1M0;
      newblock[3] = rightcoor;
      newblock[4] = M1M0 + M1M1;
      newblock[5] = PD;
      newblock[6] = X;
      newblock[7] = Y;
      if (M0)
        {
          newblock[2] += M0M0;
          newblock[4] += M0M0;
        }
      AddRow(modinput, newblock);
    }
  if (!multileft && M2)
    {
      newblock[0] = M2cn;
      newblock[1] = leftcoor;
      newblock[2] = M2M0;
      newblock[3] = rightcoor;
      newblock[4] = M2M0 + M2M2;
      newblock[5] = PD;
      newblock[6] = X;
      newblock[7] = Y;
      if (M0)
        {
          newblock[2] += M0M0;
          newblock[4] += M0M0;
        }
      if (M1)
        {
          newblock[2] += M1M1;
          newblock[4] += M1M1;
        }
      AddRow(modinput, newblock);
    }
  if (!multileft && M3)
```

```
            {
              newblock [0] = M3cn;
              newblock [1] = leftcoor ;
              newblock [2] = M3M0;
              newblock [3] = rightcoor ;
              newblock [4] = M3M0 + M3M3;
              newblock [5] = PD;
              newblock [6] = X;
              newblock [7] = Y;
              if (M0)
                {
                  newblock [2] += M0M0;
                  newblock [4] += M0M0;
                }
              if (M1)
                {
                  newblock [2] += M1M1;
                  newblock [4] += M1M1;
                }
              if (M2)
                {
                  newblock [2] += M2M2;
                  newblock [4] += M2M2;
                }
              AddRow( modinput , newblock );
            }
        }
      Sort (modinput ,0, down); /* sort on conductor number */
      modinput . markAsTemporary ();
      return modinput ;
    }
  else
    {
      cerr << "Unknown␣parameter:␣" << mainargv [1] << endl ;
      exit (-1);
    }
  break ;
case 2:
  if (( string (mainargv [1]) == "-u") || ( string (mainargv [1]) == "-c"))
    {
      cerr << "No␣input␣file␣specified\n";
      exit (-1);
    }
  else if ( string (mainargv [1]) == "m2m3")
    {
      pinput = LineWidth (m2m3);
      pinput . markAsTemporary ();
      return pinput ;
    }
  else if ( string (mainargv [1]) == "m1m2")
    {
      pinput = LineWidth (m1m2);
      pinput . markAsTemporary ();
      return pinput ;
    }
  else if ( string (mainargv [1]) == "m1m3")
    {
      pinput = LineWidth (m1m3);
      pinput . markAsTemporary ();
      return pinput ;
    }
  else
    {
      cerr << "Unknown␣parameter:␣" << mainargv [1] << endl ;
      exit (-1);
    }
  break ;
```

```
    case 1:
      cout << "Syntax: Induct -u matrix-filename\n";
      cout << "        Induct -c CIF-filename\n";
      cout << "        Induct m1m2\n";
      cout << "        Induct m2m3\n";
      cout << "        Induct m1m3\n";
      exit(-1);
    default:
      cerr << "Invalid number of arguments: " << mainargc-1 << endl;
      exit(-1);
    }
}; /* Parse */

int main(int argc, char *argv[])
{
  uMatrix<double> input = Parse(argc, argv);
  cout << "Input matrix\n" << input << endl;
  int ncond, M = 0;
  uMatrix<double> segcoor, segl, area, lambda2;
  uMatrix<int> cnum, nsegpc;
  Subdivide(input, ncond, M, cnum, segcoor, segl, nsegpc, area, lambda2);
  uMatrix<double> Lmatrix = uInv(Reduce(M,
                                        cnum,
                                        uInv(Qmk(M,
                                                 Pmn(M,
                                                     segcoor,
                                                     segl,
                                                     area),
                                                 area,
                                                 lambda2)),
                                        nsegpc,
                                        ncond));
  uMatrix<double> modL(2,2);
  modL(0,0) = L1; modL(0,1) = Mind; modL(1,0) = Mind; modL(1,1) = L2;
  cout << "Inductance matrix [pH/um]\n" << Lmatrix << endl;
  if (ncond == 3)
    {
      cout << "Inductances\n" << modL << endl;
      cout << "Length matrix [um]\n" << modL.pointDiv(Lmatrix) << endl;
    }
  return 0;
}; /* main */

/* Induct.cc */
```

B.1.3 Evaluate.cc

```
#include "Induct.h"

uMatrix<double> Pmn(const int ts,
                    const uMatrix<double> &sc,
                    const uMatrix<double> &sl,
                    const uMatrix<double> &a)
  /* Evaluates the equation for Pmn (Chang eqn (17)). */
{
  uMatrix<double>
    xll = sc(uIndex(0,1,sc.rows()-1),0),
    xur = xll + sl(uIndex(0,1,sl.rows()-1),0),
    yll = sc(uIndex(0,1,sc.rows()-1),1),
    yur = yll + sl(uIndex(0,1,sl.rows()-1),1),
    P(ts,ts);
  for (int m=0; m<ts; m++)
    for (int n=0; n<=m; n++)
      {
        P(m,n) = Kp/a(m)/a(n)*\
          (Pint(xur(m), yur(m), xur(n), yur(n))-
           Pint(xll(m), yur(m), xur(n), yur(n))-
```

```
            ( Pint ( xur (m), yll (m), xur (n), yur (n))-
              Pint ( xll (m), yll (m), xur (n), yur (n)))-
            ( Pint ( xur (m), yur (m), xll (n), yur (n))-
              Pint ( xll (m), yur (m), xll (n), yur (n))-
              ( Pint ( xur (m), yll (m), xll (n), yur (n))-
                Pint ( xll (m), yll (m), xll (n), yur (n))))-
            ( Pint ( xur (m), yur (m), xur (n), yll (n))-
              Pint ( xll (m), yur (m), xur (n), yll (n))-
              ( Pint ( xur (m), yll (m), xur (n), yll (n))-
                Pint ( xll (m), yll (m), xur (n), yll (n)))-
              ( Pint ( xur (m), yur (m), xll (n), yll (n))-
                Pint ( xll (m), yur (m), xll (n), yll (n))-
                ( Pint ( xur (m), yll (m), xll (n), yll (n))-
                  Pint ( xll (m), yll (m), xll (n), yll (n))))));
        P(n,m) = P(m,n);
      }
  P.markAsTemporary ();
  return P;
}; /* Pmn */

double Pint(const double x,
            const double y,
            const double x_,
            const double y_)
  /* Evaluates the integral part of the equation for Pmn ( Chang eqn (17)) at
     the specified coordinates. */
{
  double
    E = x - x_,
    E2 = E*E,
    E4 = E2*E2,
    N = y - y_,
    N2 = N*N,
    N4 = N2*N2,
    f,
    a,
    b;
  if (( fabs (E) <= SMALL) && ( fabs (N) <= SMALL))
    {
      f = 0;
      a = hPI;
      b = hPI;
    }
  else
    {
      f = log (E2 + N2);
      if ( fabs (E) <= SMALL)
        a = hPI;
      else
        a = atan (N/E);
      if ( fabs (N) <= SMALL)
        b = hPI;
      else
        b = atan (E/N);
    }
  return ( E4 - 6*E2*N2 + N4)*f - 8*E*N*(E2*a + N2*b) + 25*E2*N2;
}; /* Pint */

uMatrix<double> Qmk(const int ts,
                    const uMatrix<double> &P,
                    const uMatrix<double> &a,
                    const uMatrix<double> &l2)
  /* Evaluates the equation for Qmk ( Chang eqn (23)). This equation could be
     somewhat simplified by taking the symmetry of P into account. */
{
  uMatrix<double> Q(ts-1,ts-1);
  int Dmk;
```

```
  for ( int m=0; m<ts-1; m++)
    for ( int k=0; k<=m; k++)
      {
        if (m == k)
          Dmk = 1;
        else
          Dmk = 0;
        Q(m,k) = P(m,k) + P(ts-1,ts-1) - P(m,ts-1) - P(k,ts-1) +
          Uo*(l2(ts-1)/a(ts-1) + l2(k)/a(k)*Dmk);
        Q(k,m) = Q(m,k);
      }
  Q.markAsTemporary();
  return Q;
}; /* Qmk */

uMatrix<double> Reduce(int ts,
                       uMatrix<int> cn,
                       uMatrix<double> R,
                       uMatrix<int> nsp,
                       int nc)
  /* Reduces R to S. Sij is formed by summing Rij over the columns involving
     the ith conductor and the rows involving the jth conductor. */
{
  int ref = cn(ts-1,0);
  while (cn(ts-1,0) == ref)
    {
      int decnsp = nsp(nc-1);
      uMatrix<int> tempcn = cn;
      cn.resize(0,0);
      cn = tempcn(uIndex(0,1,ts-decnsp-1),0);
      uMatrix<double> tempR = R;
      R.resize(0,0);
      R = tempR(uIndex(0,1,ts-decnsp-1),uIndex(0,1,ts-decnsp-1));
      uMatrix<int> tempnsp = nsp;
      nsp.resize(0,0);
      nsp = tempnsp(uIndex(0,1,nc-2),0);
      nc--;
      ts -= decnsp;
    }
  uMatrix<double> S(nc,nc);
  int cumi = 0;
  for (int i=0; i<nc; i++)
    {
      int cumj = cumi;
      for (int j=i; j<nc; j++)
        {
          uMatrix<double> Sij = R(uIndex(cumi,1,cumi+nsp(i)-1),
                                  uIndex(cumj,1,cumj+nsp(j)-1));
          cumj += nsp(j);
          S(nc-i-1,nc-j-1) = Sum(Sij);
          S(nc-j-1,nc-i-1) = Sum(Sij);
        }
      cumi += nsp(i);
    }
  S.markAsTemporary();
  return S;
}; /* Reduce */
```

B.1.4 Subdivide.cc

```
#include "Induct.h"

void Subdivide(const uMatrix<double>&ip,
               int &nc,
               int &ts,
               uMatrix<int>&cn,
               uMatrix<double>&sc,
```

```
                uMatrix<double> &sl,
                uMatrix<int> &nsp,
                uMatrix<double> &a,
                uMatrix<double> &l2)
  /* Splits the input matrix into the required components (conductor number,
     area, number of segments, segment lengths and penetration depths) and
     builds the vector containing the coordinates of the subsegments of the
     conductors. */
{
  nc = ip.rows();
  uMatrix<int>
    shortcn = intCast(ip(uIndex(0,1,nc-1),0)),
    ns = intCast(ip(uIndex(0,1,nc-1),uIndex(6,1,7)));
  uMatrix<double>
    ll = ip(uIndex(0,1,nc-1),uIndex(1,1,2)),
    ur = ip(uIndex(0,1,nc-1),uIndex(3,1,4)),
    pd = ip(uIndex(0,1,nc-1),5),
    shortsl = ur.pointDiv(doubleCast(ns)) - ll.pointDiv(doubleCast(ns));
  nsp = ns(uIndex(0,1,nc-1),0).pointMul(ns(uIndex(0,1,nc-1),1));
  for (int c=0; c<nc; c++)
    ts += ns(c,0)*ns(c,1);
  cn.resize(ts,1);
  sc.resize(ts,2);
  sl.resize(ts,2);
  l2.resize(ts,1);
  double cumcoor = 0;
  for (int c=0; c<nc; c++)
    {
      double shortnsp = ns(c,0)*ns(c,1);
      uMatrix<double>
        xcoor(ns(c,0),1),
        ycoor(ns(c,1),1);
      for (int i=0; i<ns(c,0); i++)
        xcoor(i,0) = ll(c,0) + i*shortsl(c,0);
      for (int j=0; j<ns(c,1); j++)
        ycoor(j,0) = ll(c,1) + j*shortsl(c,1);
      for (int i=0; i<ns(c,1); i++)
        sc.insert(uIndex(cumcoor+i*ns(c,0),1, cumcoor+(i+1)*ns(c,0)-1),0, xcoor);
      for (int j=0; j<ns(c,0); j++)
        sc.insert(uIndex(cumcoor+j, ns(c,0), cumcoor+shortnsp-ns(c,0)+j),1, ycoor);
      for (int k=0; k<shortnsp; k++)
        {
          cn(cumcoor+k) = shortcn(c);
          sl.insert(cumcoor+k, uIndex(0,1,1), shortsl(c, uIndex(0,1,1)));
          l2(cumcoor+k) = pd(c)*pd(c);
        }
      cumcoor += shortnsp;
    }
  a = sl(uIndex(0,1,sl.rows()-1),0).pointMul(sl(uIndex(0,1,sl.rows()-1),1));
  int c = 0;
  while (c < nc-1)
    {
      if (shortcn(c) == shortcn(c+1))
        {
          nsp(c+1) += nsp(c);
          uMatrix<int>
            tempnsp(nc-1,1),
            tempcn(nc-1,1);
          tempnsp.insert(uIndex(0,1,c),0, nsp(uIndex(0,1,c),0));
          tempnsp.insert(uIndex(c,1,nc-2),0, nsp(uIndex(c+1,1,nc-1),0));
          tempcn.insert(uIndex(0,1,c),0, shortcn(uIndex(0,1,c),0));
          tempcn.insert(uIndex(c,1,nc-2),0, shortcn(uIndex(c+1,1,nc-1),0));
          nsp.resize(0,0);
          nsp = tempnsp;
          shortcn.resize(0,0);
          shortcn = tempcn;
          nc--;
```

```
        }
      else
        c++;
    }
}; /* Subdivide */
```

B.1.5 LineWidth.cc

```
#include "Induct.h"

uMatrix<double> LineWidth(const inductortype i)
  /* Calculate the transmission line widths needed to obtain a desired mutual
     inductance and generate an input matrix. */
{
  double l = 0;
  cout << "Desired inductor length: "; cin >> l;
  if (l <= 0)
    {
      cout << "Cannot use negative or zero length\n";
      exit(-1);
    };
  double h1, h2, t1, t2, ks1, ks2;
  if (i == m2m3)
    {
      h1 = M2M0; //+M1M1;
      h2 = M3M0+M2M2; //+M1M1;
      t1 = M2M2;
      t2 = M3M3;
      ks1 = kM2;
      ks2 = kM3;
    }
  else if (i == m1m2)
    {
      h1 = M1M0;
      h2 = M2M0+M1M1;
      t1 = M1M1;
      t2 = M2M2;
      ks1 = kM1;
      ks2 = kM2;
    }
  else
    {
      h1 = M1M0;
      h2 = M3M0+M1M1;
      t1 = M1M1;
      t2 = M3M3;
      ks1 = kM1;
      ks2 = kM3;
    };
  cout << "h1 " << h1
       << " h2 " << h2
       << " t1 " << t1
       << " t2 " << t2
       << " ks1 " << ks1
       << " ks2 " << ks2 << endl;
  double
    Lpu1 = L1*1e-6/l,
    Lpu2 = L2*1e-6/l,
    Mpu = Mind*1e-6/l,
    w1 = Uo*1e-6*K*K*Lpu2/Mpu/Mpu*(h1 + PD/tanh(t1/PD) + PD/tanh(M0M0/PD)) - ks1*h1,
    w2 = Uo*1e-6*K*K*Lpu1/Mpu/Mpu*(h2 + PD/tanh(t2/PD) + PD/tanh(M0M0/PD)) - ks2*h2;
  cout << "Widths" << endl
       << "w1 (" << L1 << ") = " << w1 << " " << "w2 (" << L2 << ") = " << w2 << endl << endl;
  double
    w = min(w1/2, w2/2),
    W = max(w1/2, w2/2);
  uMatrix<double> lwinput(0,0);
```

```
lwinput = (double) 0;
uMatrix<double> lwelement (0,0);
int
  l1start = 4,
  l1end = 8,
  l2start = 1;
if (w1 == w2)
  {
    lwinput.resize (5,8);
    lwelement.resize (5,1);
    l1start = 2;
    l1end = 4;
  }
else
  {
    lwinput.resize (9,8);
    lwelement.resize (9,1);
    if (w1 < w2)
      l2start = 3;
  };
lwelement = PD;
lwinput.insert (uIndex (0,1, l1end), uIndex (5), lwelement);
lwelement = X;
lwinput.insert (uIndex (0,1, l1end), uIndex (6), lwelement);
lwelement = Y;
lwinput.insert (uIndex (0,1, l1end), uIndex (7), lwelement);
lwelement.resize (l1end-l1start+1,1);
lwelement = 1;
lwinput.insert (uIndex (l1start,1, l1end), uIndex (0), lwelement);
lwelement = M0M0;
lwinput.insert (uIndex (l1start,1, l1end), uIndex (4), lwelement);
lwelement.resize (l1start-l2start,1);
lwelement = 2;
lwinput.insert (uIndex (l2start,1, l1start-1), uIndex (0), lwelement);
lwelement = h1+M0M0;
lwinput.insert (uIndex (l2start,1, l1start-1), uIndex (2), lwelement);
lwelement = h1+t1+M0M0;
lwinput.insert (uIndex (l2start,1, l1start-1), uIndex (4), lwelement);
lwelement.resize (l2start,1);
lwelement = 3;
lwinput.insert (uIndex (0,1, l2start-1), uIndex (0), lwelement);
lwelement = h2+M0M0;
lwinput.insert (uIndex (0,1, l2start-1), uIndex (2), lwelement);
lwelement = h2+t2+M0M0;
lwinput.insert (uIndex (0,1, l2start-1), uIndex (4), lwelement);
lwinput (0,2) -= w1 < w2 ? t1 : 0;
lwinput (0,4) -= w1 < w2 ? t1 : 0;
lwinput (2,2) -= w1 < w2 ? t1 : 0;
lwinput (2,4) -= w1 < w2 ? t1 : 0;
lwinput (l1end,1) = -10*W;
lwinput (l1end,3) = -W;
lwinput (l1end-1,1) = -W;
if (w != W)
  {
    lwinput (l1end-1,3) = -w;
    lwinput (l1start+2,1) = -w;
    lwinput (l1start+2,3) = w;
    lwinput (l1start+1,1) = w;
  }
lwinput (l1start+1,3) = W;
lwinput (l1start,1) = W;
lwinput (l1start,3) = 10*W;
lwinput (l1start-1,1) = -w1/2;
if (w1 > w2)
  {
    lwinput (l1start-1,3) = -w2/2;
    lwinput (l1start-2,1) = -w2/2;
```

```
      lwinput(12start+1,3) = w2/2;
      lwinput(12start,1) = w2/2;
    };
  lwinput(12start,3) = w1/2;
  lwinput(12start-1,1) = -w2/2;
  if (w1 < w2)
    {
      lwinput(12start-1,3) = -w1/2;
      lwinput(12start-2,1) = -w1/2;
      lwinput(1,3) = w1/2;
      lwinput(0,1) = w1/2;
    };
  lwinput(0,3) = w2/2;
  lwinput.markAsTemporary();
  return lwinput;
}; /* LineWidth */
```

B.1.6 Misc.cc

```
#include "Induct.h"

void Sort(uMatrix<double>&sortin,
          const int sortcol,
          const sortdirtype sortdir)
  /* Sort the rows of the specified matrix in the specified direction,
     according to the specified column, using bubble sort. */
{
  if (sortdir == down)
    {
      for (int i=0; i<sortin.rows(); i++)
        for (int j=i+1; j<sortin.rows(); j++)
          if (sortin(i,sortcol) <= sortin(j,sortcol))
            Swop(sortin,i,j);
    }
  else
    {
      for (int i=0; i<sortin.rows(); i++)
        for (int j=i+1; j<sortin.rows(); j++)
          if (sortin(i,sortcol) >= sortin(j,sortcol))
            Swop(sortin,i,j);
    };
}; /* Sort */

void Swop(uMatrix<double>&sm,
          const int i,
          const int j)
  /* Swop rows i and j of the specified matrix. */
{
  uMatrix<double> tempsm = sm(i,uIndex(0,1,sm.columns()-1));
  sm.insert(i,uIndex(0,1,sm.columns()-1),sm(j,uIndex(0,1,sm.columns()-1)));
  sm.insert(j,uIndex(0,1,sm.columns()-1),tempsm);
}; /* Swop */

void AddRow(uMatrix<double>&inmatrix,
            const double newrow[])
  /* Append the specified row to the bottom of the specified matrix. */
{
  uMatrix<double> tempinmatrix = inmatrix;
  inmatrix.resize(tempinmatrix.rows()+1,tempinmatrix.columns());
  if (tempinmatrix.rows() > 0)
    inmatrix.insert(uIndex(0,1,tempinmatrix.rows()-1),
                    uIndex(0,1,tempinmatrix.columns()-1),
                    tempinmatrix);
  for (int i=0; i<inmatrix.columns(); i++)
    inmatrix(inmatrix.rows()-1,i) = newrow[i];
}; /* AddRow */
```

```
double Sum( const uMatrix<double> &m)
  /* Calculates the sum of the elements of the specified matrix. */
{
  double total = 0;
  for ( int i=0; i<m.rows()*m.columns(); i++)
    total += m.address()[ i ];
  return total;
}; /* Sum */
```

B.2 Input File Formats

B.2.1 Matrix

The conductors of the system in question are numbered, starting at 1 from the conductor being used as the groundplane. The transmission lines are then divided into sections, as demonstrated by the dashed lines in Figure B.1. Each section of a conductor has the same number, as shown. The lower left and upper right coordinates are determined, in accordance with the x and y axes shown. The absolute coordinates are unimportant.

Table B.1 lists an example of the matrix input. The first column contains the conductor number of each line section, while the coordinates occupy columns 2 through 5. The penetration depth of each section is listed in column 6. The algorithm used to calculate the inductance divides each line section into segments. Accuracy and calculation time are proportional to the number of segments. Column 7 and 8 contain the number of horizontal and vertical segments.

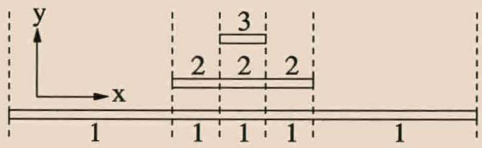


Figure B.1: Cross section of an example transmission line system.

Conductor number	Lower left x	Lower left y	Upper right x	Upper right y	λ	Segments x	Segments y
3	-2.125	1.25	2.125	1.85	0.09	10	5
2	2.125	0.45	2.75	0.75	0.09	10	5
2	-2.125	0.45	2.125	0.75	0.09	10	5
2	-3.25	0.45	-2.125	0.75	0.09	10	5
1	2.75	0	20	0.1	0.09	10	5
1	2.125	0	2.75	0.1	0.09	10	5
1	-2.125	0	2.125	0.1	0.09	10	5
1	-3.25	0	-2.125	0.1	0.09	10	5
1	-20	0	-3.25	0.1	0.09	10	5

Table B.1: Matrix input format for the Induct program

B.2.2 CIF

A transmission line system can also be represented graphically, with, for example, the Xic layout package. Currently, only the Caltech Intermediate Form, or CIF, format is supported. Because the Induct program calculates the inductance per unit length, only a short section of the system is needed. Figure B.2 demonstrates such a layout. Induct takes the cross section of the layout and determines the input matrix. The cross section is taken vertically, but the layout in Figure B.2 is rotated by 90° for typesetting purposes. Transmission line sections have to be represented by boxes, because wires are ignored by the program.

Currently only the HYPRES process is supported. M0 definitions represent the dimensions of the groundplane.

Figure B.2: Graphical representation of a section of the *COSL* inductor pair.