Cryogenic Digital-Analog Converter for Two-Qubit Gates in Spin Qubits

M.Sc. Thesis

Ilker Polat

Delft University of Technology

TU Delft

CRYOGENIC DIGITAL-ANALOG CONVERTER FOR TWO-QUBIT GATES IN SPIN QUBITS

M.SC. THESIS

by

Ilker Polat

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday August 30, 2023 at 15:00 AM.

Student number:	5375312	
Project duration:	Sep 1, 2023 – August 30, 2023	
Thesis committee:	Dr. Fabio Sebastiano,	TU Delft, supervisor
	Dr. ir. Muhammed Bolatkale,	TU Delft
	Dr. Maximilian Rimbach-Russ,	TU Delft

This thesis is confidential and cannot be made public until August 30, 2028.

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Prof. Fabio Sebastiano for giving me this opportunity to work on this project. For me, this project was like crossing the jungle with an incomplete map and your full support and trust in me gave me the encouragement to explore more. When I look back, your reply to my internship email when I was a bachelor's student was one of the milestones in my life. Thank you so much for giving me that chance. It is always exciting to work with you.

Secondly, I would like to thank Ramon Overwater for being my daily supervisor. The discussions we had with you during this process were always fun and helpful. Many times it has helped me solve critical problems and even understand my own thoughts.

I would also like to thank Prof. Maximilian Rimbach-Russ for helping me with the theoretical parts of the project. Your support has opened up new approaches to this project.

I would also like to thank Prof. Muhammed Bolatkale for being on the committee for my thesis defense.

I would also like to thank my fellow master's students in Coolgroup Zenghui, Deb, Ivor, Maoran, and Andrea. You were always my first option to escape from work.

I would also like to thank other members of the Coolgroup. Bagas, thank you for your layout support. Luc, thank you for always answering my quick questions. Job, thank you for sharing your experiences on Intel PDK. And all others, Sadık, Rob, Neils, Ali thank you so much for all that fun.

I would also like to thank the people in QuTech. This work couldn't achieve these outcomes without such an environment. Thank you to all the people answering my questions with more than I ask every time; Stephan, Brennan, Xioa, and Pablo.

I would like to thank Lokum (cat) for playing with me whenever I stood up from the chair. I always admire your sleeping schedule.

I would like to thank my parents and brother for supporting my decision all these years.

Finally, I would like to thank Sevgi for supporting me through this process. You always make me realize what is really important in our life. Even in my most tense times, you pulled me out of the bubble and made me breathe without even realizing it. I will be here always for your thoughts, surprises, and questions.

CONTENTS

ABSTRACT

As the number of qubits increases, controlling qubits at cryogenic temperatures with electronics at room temperature becomes infeasible due to the vast number of cabling. This necessitates controlling quantum operations on the qubits with a cryogenic interface, which is low power, scalable and maintains qubit fidelity.

In pursuit of this interface, this thesis investigates the two-qubit DAC specifications of both the adiabatic and nonadiabatic CPHASE gate. The DAC specifications are identified using simulation models of spin qubits. The adiabatic signal requirements are found to be lower than the nonadiabatic signals in both sampling frequency and quantization resolution. A new method using spectral analysis on the unitary of the adiabatic operation is used to infer the minimum sampling frequency by applying the Nyquist Criterion and shows why they lose their fidelity as gate time decreases.

A novel two-stage current-switching DAC architecture is proposed, which maintains fidelity operating at 140MHz sampling rate with less than 75 μW of power consumption. The first 6-bit stage maintains CPHASE fidelity for different qubit pairs, while the second 5-bit stage creates the adiabatic pulse. Finally, the DAC is implemented in the Intel 16-nm finFET process.

1

INTRODUCTION

QUANTUM COMPUTERS

Quantum computers utilize quantum mechanics to solve complex computational problems. They have the potential to outperform classical computers for many problems in a variety of fields, including cybersecurity, drug development, and financial modeling [1], [2]. Quantum computers consist of qubits, short for "quantum bits", which are the quantum dual of the classic bit. Qubits can exist in multiple states simultaneously, due to a phenomenon known as superposition. Together with the entanglement of the qubits, superposition allows quantum computers to process an enormous amount of information in parallel.

Qubits are implemented on various technological platforms, including nitrogen vacancies in diamond lattices, trapped ions, electron/hole spin in semiconductors, and superconducting circuits. Most qubit implementations require operating at very low temperatures (below 1 K) to extend the qubits' coherence time [3].

CRYO-ELECTRONICS

Besides the quantum processor described in the previous section, a quantum computer also contains a classical electronic system. The classical electronics accommodate the classical processor, read-out electronics, as well as a control block, capable of changing the state of qubits [3], [4]

In semiconductors spin qubits, the target of this work, current systems manage the qubits with classical electronics at room temperature. However, as the number of qubits increases to thousands or even millions, this approach becomes less feasible. To overcome this bottleneck, cryogenic electronic interfaces have been proposed [5].

TWO-QUBIT GATES

To control the qubit states, so-called quantum gates are needed. Quantum gates perform operations like flipping the state of a qubit, entangling qubits, etc. When these gates are combined, quantum algorithms can be constructed to solve problems. To achieve

1

a functional quantum computer, a universal gate set is required [6]. Such a gate set not only requires single-qubit gates but at least one two-qubit gate.

In several spin qubit demonstrations, the CPHASE gate is chosen as the two-qubit gate [7]–[9]. A CPHASE gate can be achieved either by adiabatic or non-adiabatic control signals. Although both adiabatic and non-adiabatic signals implement the same CPHASE gate, their pulse shapes and thus qubit evolution differ. Important for this work is that these differences in pulse shape affect both the time and amplitude resolution required by the electronics

1.1. OBJECTIVE

In the literature, CPHASE-gate cryogenic control electronics focused only on nonadiabatic signals [10]. However, experimental works [8], [9] proved that adiabatic control signals can also achieve high-fidelities. As stated before, these two control signals create different evolutions on the qubit system and they may have different requirements from the electronics to maintain their evolution characteristics. Also, their evolutions depend on the qubit-pair parameters that may vary for different pairs. Therefore, it is essential to investigate the evolution of various control signals with different qubit parameters to determine the specifications of the cryogenic control system for a scalable system.

To address these issues, this thesis covers the implementation of a Digital-Analog Converter (DAC) that can control CPHASE gates in spin-qubits. During the design, three main points will be taken into consideration. These are: the fidelity of the CPHASE gate should be maintained, the DAC should be able to work below 1K, and lastly, this system should be scalable for controlling different qubit pairs.

1.2. THESIS OUTLINE

This work starts with a necessary background on the spin qubits, quantum logic gate, and Digital-Analog Converters in Chapter 2.

In Chapter 3, the influence of the control signals on CPHASE fidelity and the DAC requirements is investigated theoretically and with spin-qubit DAC simulation models. In this chapter there are three main outcomes for the literature; the Nyquist Criterion is applied to find the minimum sampling rate of a qubit system evolution; the limiting factor of the minimum adiabatic pulse duration is shown in the frequency domain; and an DAC architecture proposed to cover different qubit fidelities.

Chapter 4 covers the design of low-power current switching DAC with Intel 16nm technology.

In the end, the thesis concludes with discussions and conclusions in Chapter 5.

2

BACKGROUND INFORMATION

In this chapter, the background information required to understand the following chapters will be given. First, implementations of spin qubits in semiconductors will be examined with explanations of quantum computer basics. Later, the sampling rate, signal reconstruction, and resolution of the DAC will be explained.

2.1. QUANTUM LOGIC OF SPIN QUBITS

The basic structure of semiconductor spin qubits is trapping electrons in tiny regions known as quantum dots. Electrons in the quantum dots can be controlled by sending microwave pulses or changing the tunnel coupling between electrons. An example of two semiconductor spin qubits can be found in Fig. 2.1. Quantum dots are trapped under the left and right plungers (LP and RP). By controlling the voltage on the barrier (B), the coupling strength between qubits can be manipulated. Microwave pulses can be sent to the MW plunger to rotate spin qubits individually. Lastly, the readout of a spin-qubit can be achieved by measuring the change in the biasing current of a single quantum dot (SQD).

2.1.1. LARMOR FREQUENCY AND ZEEMAN SPLITTING

The Larmor frequency E_z represents the rotational speed of the qubits in the presence of a magnetic field. Microwave pulses need to be fine-tuned to match this Larmor frequency in order to achieve precise control for single-qubit spins. The mismatch of the pulse frequency and the Larmor frequency can lead to decreasing the fidelity of the operation.

The gradual magnetic field variation across multiple spin qubits leads to unique Larmor frequencies for each qubit. This alteration in frequency is crucial, as it creates a distinction in the qubits' frequency, facilitating the independent manipulation of qubits using microwave signals. The difference in Larmor frequency between pairs of qubits is called Zeeman Splitting (δE_z).

In Fig. 2.2, the Larmor frequencies of six qubits are depicted. The Larmor frequency

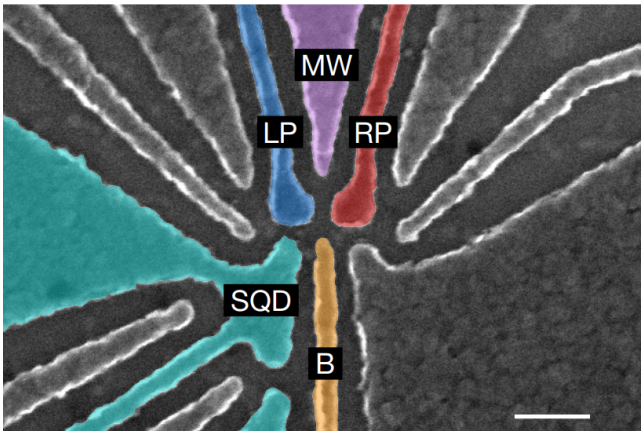


Figure 2.1: Scanning electron microscopy of the two-qubit device (Scale bar indicates 100nm) [8]

distribution is nonlinear. Consequently, this nonlinearity will result in distinct variations in the value of δE_z between different pairs of qubits. Nevertheless, given that this is one of the initial demonstrations of a multiqubit apparatus, a better degree of uniformity in δE_z values is anticipated.

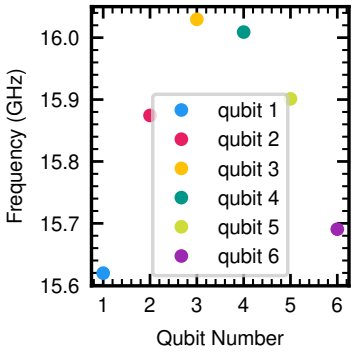


Figure 2.2: Larmor frequencies of six-qubits placed in a row [9]

2.1.2. COUPLING STRENGTH

The coupling strength (J) is the degree of interaction between individual qubits in a quantum system. The relationship between J and the barrier voltage can be written as $J = exp(\alpha_{Barrier} \cdot V_{Barrier}) \cdot J_{offset}$, where $V_{Barrier}$ is the voltage on the barrier plunger and J_{offset} is the offset coupling when $V_{Barrier}$ is zero. Lastly, $\alpha_{Barrier}$ is a coefficient that characterizes the sensitivity of the voltage to coupling.

In Fig. 2.4, the coupling strengths of five different pairs of qubits are shown versus the normalized barrier voltage. J_{offset} values can be found where the barrier voltage is

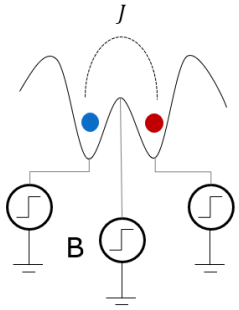


Figure 2.3: Control of the coupling strength between qubits can be controlled by barrier voltage

0. The slopes of the lines also show the values of $\alpha_{Barrier}$ for the pairs. As can be seen from here, for different qubit pairs, the barrier voltage relation might be quite different.

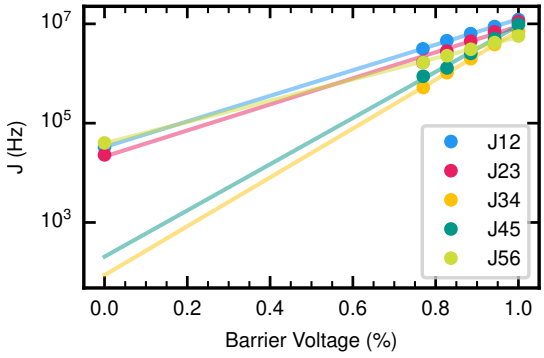


Figure 2.4: $J-V$ relation between different qubit pairs [9]

2.1.3. HAMILTONIAN OF TWO-QUBIT

The Hamiltonian is used in quantum mechanics to describe how the system evolves, which is essential for understanding quantum systems.

Below, an exchange Hamiltonian of a two-qubit system can be found as an example. E_z is the average of the Larmor frequencies $\left(\frac{f_1+f_2}{2}\right)$. The difference between Larmor frequencies$\delta E_z = (f_1 - f_2)$ is the Zeeman splitting.. J is the coupling strength between qubits, which can be controlled.

$$H_{exc} = \begin{bmatrix} E_z & 0 & 0 & 0 \\ 0 & (-\delta E_z - J)/2 & J/2 & 0 \\ 0 & J/2 & (\delta E_z - J)/2 & 0 \\ 0 & 0 & 0 & -E_z \end{bmatrix}$$

The calculation of the Hamiltonian time evolution will be explained in Section 3.2.1.

2.1.4. QUANTUM LOGIC GATES

Classical logic gates process bits to perform logical operations such as AND, OR, NOT, and XOR. In quantum computers, there are also quantum logic gates which are responsible for logic operating on qubits. The operation of quantum logic gates is described as unitary matrices.

Quantum logic gates can operate on single or multiple qubits at the same time. Some of the single quantum logic gates can be found below.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Here, the X, Y, and Z gates rotate the qubit around the related axis. Because they manipulate individual qubits, they are referred to as single-qubit gates. These gates can be expressed by combining two other single-qubit gates, for example, $iY = ZX = -XZ$. This implies that only two types of single-qubit gates are necessary for a functional quantum computer.

The Controlled-Z (CZ) quantum gate is one of the two-qubit gates that operate on a pair of qubits. Depending on the state of one qubit, a Z gate is applied to the other. Its matrix representation is as follows.

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

In addition to single-qubit gates, at least one two-qubit gate is also necessary for a functional quantum computer to create an interaction between different qubits.

2.2. DIGITAL TO ANALOG CONVERTER

Digital-to-analog converters (DACs) are devices that transform discrete digital signal information into continuous physical analog signals. The conversion process involves three main aspects.

The first aspect is the conversion sampling rate. It ensures that the digital representation is accurately translated into analog form. The low sampling rate may disturb the signal.

The second aspect is the reconstruction of the sampled data. This step involves transitioning from discrete time intervals to continuous-time ones.

The third aspect is the process of converting discrete digital values into quantized analog voltage or current levels. The quantization resolution determines how accurately the analog signal can represent the original digital values. Higher resolution results in a more accurate conversion, but may require more complex hardware.

These aspects, the sampling rate, signal reconstruction and the resolution of the output levels, play crucial roles in determining the quality of the final analog signal produced by the DAC.

2.2.1. SAMPLING FREQUENCY

The sampling frequency refers to the conversion rate of the analog output signal. Between specific time intervals, the DAC should generate the sampled signal at its output.

Sampling the signal on the time domain generates replicas of the signal around the sampling frequency on the frequency domain. This can be seen in Fig. 2.5.d.

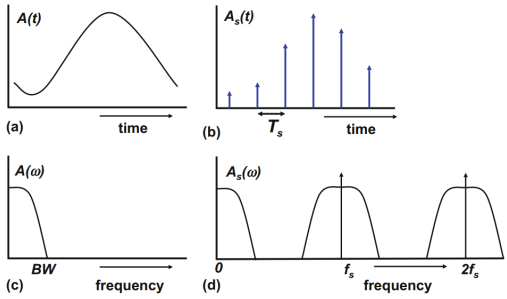


Figure 2.5: Sampling a continuous signal in the time domain (a) results in discrete time samples (b). Also, the frequency spectrum of the continuous-time signal (c) gets replicated around the sampling frequency (f_s) and its integer multiples (d). [11]

To ensure that the information carried by the signal is preserved accurately enough, the Nyquist Criterion must be followed. It means that the sampling frequency should be higher than two times the signal bandwidth. Otherwise, replicas of the signal may overlap with each other. For example, in Fig. 2.6, the main signal and its replica overlap with each other. After this, it cannot be possible to get the carried information.

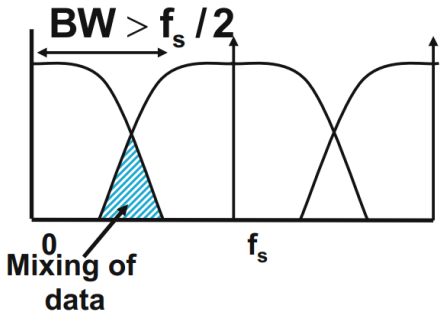
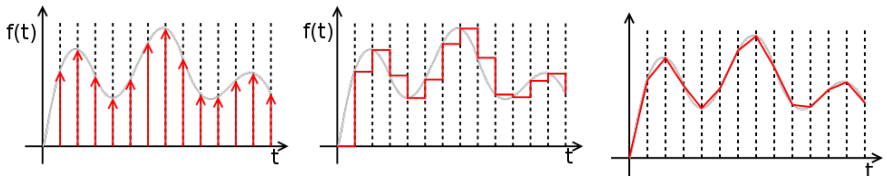


Figure 2.6: An example of aliasing of two replicas on frequency spectrum[11]

2.2.2. SIGNAL RECONSTRUCTION

After the signals are sampled, they need to be converted to continuous signals. To do that, generally zero or first-order hold mechanisms are used for DACs. An example of zero-order and first-order hold is shown in Fig. 2.7. The zero-order hold mechanism keeps the sampled value over a sampling period. However, the first-order hold mecha-

nism creates linear interpolations between samples.



(a) Sampled Signal **(b)** Zero-Order Hold **(c)** First-Order Hold

Figure 2.7: Different Reconstructions of Sampled Singal

2.2.3. QUANTIZATION

DAC converts the sampled signal values to the nearest limited number of output levels of a DAC. This fundamental process is known as quantization. When the output resolution is increased, signals can be generated with greater precision. In Fig. 2.8, the sinusoidal signal is quantized with different resolutions. The N-bit resolution refers to 2^N number of output levels, therefore, each additional bit doubles the output levels of the DAC.

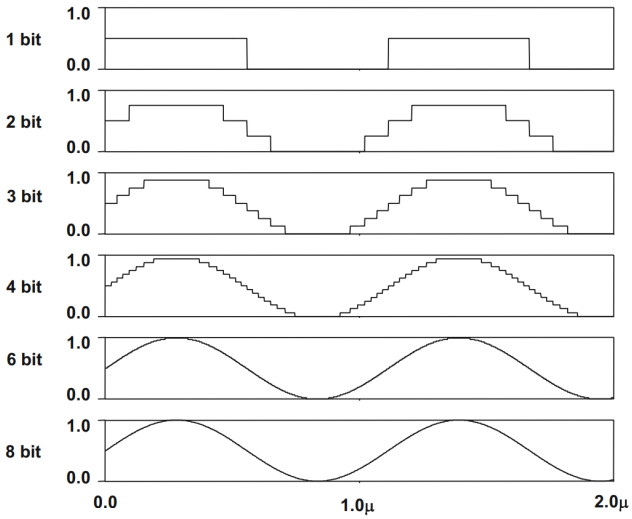


Figure 2.8: Quantizated sinusoidal Signal with Different Resolutions [11]

The output voltage of the DAC with the given digital input data can be found as follows.

$$V_{OUT} = \frac{D_{IN}}{2^N} \cdot V_{REF}$$

In this equation D_{IN} is the sampled digital input data that can have an integer value

between 0 and $2^N - 1$. V_{REF} is the reference voltage of the DAC, and it scales the output voltage by the given $\frac{D_{IN}}{2^N}$ proportion.

3

CPHASE Gate Implementation and Electronic Requirements for Silicon Spin Qubits

The development of a functional quantum computer requires an increase in the number of qubits while preserving high-accuracy operations. To create an electronic interface for quantum computers, the primary objective is to construct a scalable system that maintains the accuracy of the qubits. To create such a system, it is necessary to analyze and understand the control signals of the qubit system, where control signals are the bridge between electronics. The characteristics of the control signals can influence the quantum operations or electronics requirements. Therefore, determining the effects of control signals on both sides is important in designing a scalable electronic interface while keeping the qubits coherent.

In this chapter, we examine how different control signals for the CPHASE gate affect the operation of the qubit system and their requirements on the electronics. It will start with background information on how CPHASE gates work for spin qubits and the criteria for achieving high-fidelity CPHASE gates. Later, the CPHASE gate simulation model will be described. This model is used to investigate how different control pulse shapes change the CPHASE operation and how this impacts fidelity. Finally, different pulse shapes will be examined to see which one appears the most suitable for scalable and accurate control.

3.1. CPHASE Gates for Silicon Spin Qubits

A universal quantum gate set not only requires single qubit gates but also at least a two-qubit gate to create entanglement between qubits. For spin qubits, the CPHASE gate is implemented by changing the coupling between two qubits. When the coupling is increased, the energy levels of the qubits will shift, and this energy shift will cause an accumulation of phases over time. After the required phase is achieved, the coupling

can be lowered again and the qubit energy levels will return to their initial levels.

ACHIEVING CPHASE GATE FROM TWO SPIN-QUBIT HAMILTONIAN

The exchange interaction between two spin qubits can be described by the Hamiltonian below:

$$H_{exc} = \begin{bmatrix} E_z & 0 & 0 & 0 \\ 0 & (-\delta E_z - J)/2 & J/2 & 0 \\ 0 & J/2 & (\delta E_z - J)/2 & 0 \\ 0 & 0 & 0 & -E_z \end{bmatrix}$$

In this Hamiltonian, E_z is the average of the Larmor frequencies of the spin qubits in the system ($\frac{f_1+f_2}{2}$) and δE_z is the difference between the Larmor frequencies of the spin qubits ($f_1 - f_2$). J is the coupling strength of the exchange interaction between the qubits.

As J can be time-dependent or independent, for the rest of this thesis, J will be considered time independent and $J(t)$ as a time-dependent signal.

To explain the characteristics of this Hamiltonian and how we can achieve a CPHASE operation, the Hamiltonian is split into commuting matrices to explain different oscillations more easily. Since the definition of commute matrices A & B is that they commute if $AB = BA$, they can operate the system in any order. These commuting matrices H_{rot}, H_{cphase} and H_{swap} are listed below:

$$H_{rot} = \begin{bmatrix} E_z & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -E_z \end{bmatrix} \qquad H_{cphase} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -J/2 & 0 & 0 \\ 0 & 0 & -J/2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$H_{swap} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\delta E_z/2 & J/2 & 0 \\ 0 & J/2 & \delta E_z/2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

H_{rot} basically, describes the rotation of two qubits with average Larmor frequencies E_z.

H_{cphase} is the Hamiltonian that produces the CPHASE operation. The time-dependent operator of H_{cphase} is as follows:

$$U_{CPHASE} = \exp(-i\,2\pi \cdot H_{cphase}(t)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\pi J(t)} & 0 & 0 \\ 0 & 0 & e^{i\pi J(t)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

U_{CPHASE} adds a phase on the two-qubit system depending on their state. This can be turned to the commonly used Controlled-Z (CZ) gate by letting $\int J(t)dt = 1/2$ and applying virtual Z gates.

$$U_{CZ} = Z_1(-\pi/2) \cdot Z_2(-\pi/2) \cdot U_{CPHASE} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Although U_{CPHASE} unitary creates the required two-qubit gate, the last Hamiltonian H_{swap} creates a SWAP oscillation between qubits. Since SWAP and CHPASE gates operate at the same time, to achieve one of these gates with high fidelity, the other gate operation should be equalized to the I gate. To achieve a high-fidelity CPHASE gate, this SWAP oscillation should be canceled by letting $U_{SWAP} = I$. Otherwise, they will create unwanted state evolutions. In this thesis, methods will be examined to achieve high-fidelity U_{CPHASE} gate by canceling U_{SWAP} gate oscillations.

As the H_{swap} matrix only has inner elements, we can write it as follows to simplify our calculations:

$$H'_{swap'} = \begin{bmatrix} -\delta E_z/2 & J/2 \\ J/2 & \delta E_z/2 \end{bmatrix}$$

$$U'_{SWAP} = \exp(-i\,2\pi \cdot H'_{swap}(t)) =$$

$$\sigma_I \cdot \cos\left(\pi t\sqrt{\delta E_z^2 + J(t)^2}\right) + i\left(\delta E_z \cdot \sigma_Z - J(t) \cdot \sigma_X\right) \cdot \frac{\sin\left(\pi t\sqrt{\delta E_z^2 + J(t)^2}\right)}{\sqrt{\delta E_z^2 + J(t)^2}}$$

U_{SWAP} gate can be found as follows.

$$U_{SWAP} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\left(\pi t\sqrt{\delta E_z^2 + J^2}\right) + \frac{i\delta E_z \sin\left(\pi t\sqrt{\delta E_z^2+J^2}\right)}{\sqrt{\delta E_z^2+J^2}} & -\frac{iJ\sin\left(\pi t\sqrt{\delta E_z^2+J^2}\right)}{\sqrt{\delta E_z^2+J^2}} & 0 \\ 0 & -\frac{iJ\sin\left(\pi t\sqrt{\delta E_z^2+J^2}\right)}{\sqrt{\delta E_z^2+J^2}} & \cos\left(\pi t\sqrt{\delta E_z^2 + J^2}\right) - \frac{i\delta E_z \sin\left(\pi t\sqrt{\delta E_z^2+J^2}\right)}{\sqrt{\delta E_z^2+J^2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

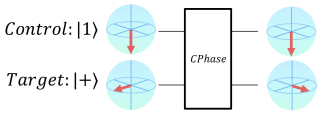
Even if it seems a bit complicated to understand how this gate operates on the qubits, it is clear that I can be obtained by removing the non-diagonal terms from U_{SWAP}. Since non-diagonal terms are equal to each other, removing a single term will be enough to achieve I. One way to remove them is to choose the duration of the operation as an integer multiple of $\frac{1}{\sqrt{\delta E_z^2+J^2}}$. That will make the non-diagonal terms 0. Another way is letting $J << \delta E_z$. This does not remove the oscillation but lowers the amplitude of the non-diagonal terms by $\frac{J}{\sqrt{\delta E_z^2+J^2}} \approx \frac{J}{\delta E_z}$. J can be chosen low enough to keep the fidelity high enough.

Another way to remove the sinusoidal oscillation is to change J adiabatically. However, this is a topic of another section (Section 3.3.2).

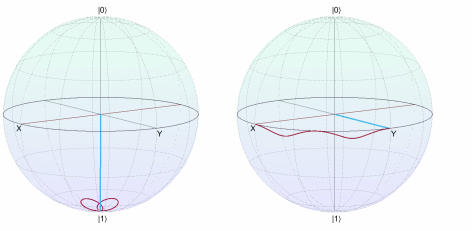
A SIMPLE EXAMPLE OF CPHASE

Now, let's look at how this Hamiltonian operates on the two-qubit system with an example of the quantum circuit in Fig. 3.1a. This qubit system is initialized to $|\phi_0\phi_1\rangle = |1+\rangle$. In this state, $|1\rangle$ will have the role of the control qubit since it determines the rotation direction of target qubit $|+\rangle$. The ideal CPHASE gate would rotate the target qubit around the Z-axis with a $\pi/2$ angle while keeping the control qubit in its initial position. In Fig. 3.1b, the time evolution of the qubits can be seen under the Hamiltonian H_{exc}. If there was only the U_{CPHASE} gate, the control qubit should stay in its initial position while the target qubit rotates around $Z-axis$ on the equator. However, due to the U_{SWAP} gate, both qubits oscillate during the operation. This example clearly shows how U_{CPHASE} and U_{SWAP} operate at the same time on both qubits.

Although the final position of the target qubit in Fig. 3.1b is on the equator, it is not exactly on the Y-Axis. An additional Z-rotation is required, indicating the need for calibration.



(a) Caption



(b) Caption

Figure 3.1: The same cup of coffee. Two times.

In this example, the ratio of $J/\delta E_z$ is chosen as $1/4$, for means of illustration. This is a bit higher than the usual ratio to show the nonidealities clearly. This ratio is generally closer to 0.1. As stated before, the amplitude of nondiagonal terms of U_{SWAP} is $\approx J/\delta E_z$, thus choosing this ratio smaller can directly reduce the oscillation amplitude.

The working principle of choosing the gate duration as integer multiples of the $\frac{2}{\sqrt{\delta E_z^2+J^2}}$ can also be seen in this example. With a period of $\frac{2}{\sqrt{\delta E_z^2+J^2}}$, the control qubit is at the south pole at the same time that the target qubit touches the equator. Therefore, finishing the gate operation when the qubits touch the pole and equator cancels the effect of the U_{SWAP} oscillation.

To conclude, a pure U_{CPHASE} gate does not exist for spin qubits. Due to their exchange Hamiltonian (H_{exc}), a U_{CPHASE} gate can only be obtained by removing the effect of the U_{SWAP} oscillation. To achieve this, we have two conditions:

- $\int J(t)dt = 0.5$ to make sure we have a high fidelity U_{CPHASE} gate
- $U_{SWAP} = I$ to cancel the unwanted oscillation at the end of the operation

Violation of these two conditions will create infidelity ($\bar{F} = 1 - F$) in the system. If we call, respectively, $\bar{F}_{cphase}$ and $\bar{F}_{swap}$ infidelities due to these two conditions, the total infidelity of the operation can be found below:

$$
\begin{aligned}
(1-F_{total}) &= 1-(1-\bar{F}_{cphase})(1-\bar{F}_{swap}) \\
\bar{F}_{total} &= 1-(1-\bar{F}_{cphase})(1-\bar{F}_{swap}) \\
\bar{F}_{total} &= 1-(1-\bar{F}_{cphase}-\bar{F}_{swap}+\bar{F}_{cphase}\cdot\bar{F}_{swap}) \\
\bar{F}_{total} &= \bar{F}_{cphase}+\bar{F}_{swap}-\bar{F}_{cphase}\cdot\bar{F}_{swap} \\
\bar{F}_{total} &\approx \bar{F}_{swap}+\bar{F}_{cphase}
\end{aligned}
$$

Here, $\bar{F}_{total}$ is calculated by the infidelity of other operations. Since in most cases these two infidelities will be smaller than 10^{-3}, the product $\bar{F}_{cphase}\cdot\bar{F}_{swap}$ can be neglected. This calculation shows that the fidelity of our operation depends on the equality of both the fidelity of U_{CPHASE} and the cancelation of U_{SWAP} .

3.2. MODELLING SPIN QUBITS AND DAC

In the previous section, the implementation of the CPHASE gate and its non-ideality were explained. Before looking into how it may affect the electronics requirements, the MATLAB simulation models of a two-qubit system together with the DAC will be explained.

3.2.1. HAMILTONIAN SOLVER

To simulate the evolution of the two-qubit system, the Hamiltonian of the system is converted to a unitary operator by the Schrödinger equation.

$$
\begin{aligned}
H\Psi &= i\hbar\frac{\delta\Psi}{\delta t} \\
U &= e^{\frac{-iHt}{\hbar}} \\
\Psi(t) &= U\Psi(0)
\end{aligned}
$$

If Hamiltonian is time-dependent, the calculation of unitary gate U_j is done for every time step Δt.

$$
\begin{aligned}
U_j &= e^{\frac{-iH(j)\Delta t}{\hbar}} \\
\Psi(j) &= U_j\Psi(j-1)
\end{aligned}
$$

3.2.2. PULSE GENERATOR

The simulator should be able to create different pulses to be tested. The Pulse Generator class is created to generate and manipulate different signals, windows, and noise models. For all, the offset and/or power of the signal can be defined. A list of signals in the class can be found below:

- Square
- Rising
- Falling
- Triangle
- Steps
- Gauss
- Sine
- Cosine
- Exponential
- Tukey
- Hamming
- Hann
- Blackman
- Slepian
- Chebyshev
- Hyperbolic
- Kaiser
- White (Thermal) Noise
- Pink (Flicker) Noise

3.2.3. QUBIT NOISE AND DISTORTION MODELS

Noise in qubits refers to unwanted disturbances or fluctuations that can affect the fidelity of the qubits. As the qubit fidelity is limited by noise sources, the contribution of the electronics to infidelity should be lower than the noise. Otherwise, the control electronics will be the bottleneck of the system. Therefore, modeling noise sources is important to choose the correct specs for the electronics.

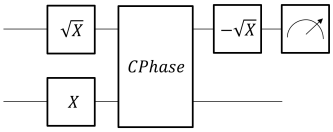
All noise and distortions are modeled in three parts. Their Hamiltonian describes how quantum states will evolve. The amplitude of these Hamiltonians has different sources depending on the model. Lastly, their frequency spectrum determines their noise characteristics.

Model	Type	Hamiltonian	Amplitude	Spectrum
Lever arm	Distortion	Z	$V_{barrier} \cdot \frac{\delta f_i}{v_{B,leverarm}}$	DC
Dephasing	Noise	Z	$\frac{1}{2\pi T_2^*}$	Gaussian
Charge Noise	Noise	Z	$V_{cn} \cdot \frac{\delta f_i}{v_{B,leverarm}} 4$	1/f
		Heisenberg	$\exp(\alpha \cdot V_{cn} + b)$	1/f

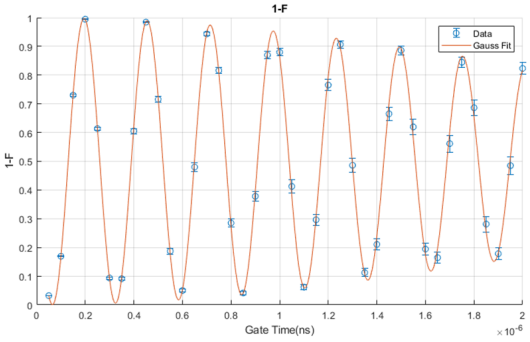
Table 3.1: Noise and Distortion Models

The **lever arm** is a parameter that refers to the sensitivity of the qubit's energy to the barrier voltage. A change in the barrier voltage can change the location of the qubits in the quantum dots, and that may lead to a shift of their Larmor frequencies. This should be considered for precise control of qubits. $v_{B,leverarm}$ is the ratio of frequency change for a given voltage and δF_i is the frequency shift. After these two parameters are measured with experiments, the amplitude of the signal can be tuned to compensate for this phenomenon.

The **dephasing** represents the phenomenon where the relative phase of the qubit spins becomes uncertain or randomized, due to environmental noise. With the Ramsey experiment, the relaxation time of the qubit (T_2^*) can be measured. In Fig. 3.2, Ramsey's experiment results are shown with our model. Increasing the time of the CPHASE gate leads to the accumulation of noise and eventually loss of information.



(a) Quantum Circuit



(b) Simulation Results

Figure 3.2: Ramsey Experiment with Dephasing

The **charge noise** models the charge noise on the barrier plunger. It has two different effects on the system. Firstly, like the Lever Arm, it shifts the frequency of the qubits. Secondly, it creates an exchange interaction between qubits.

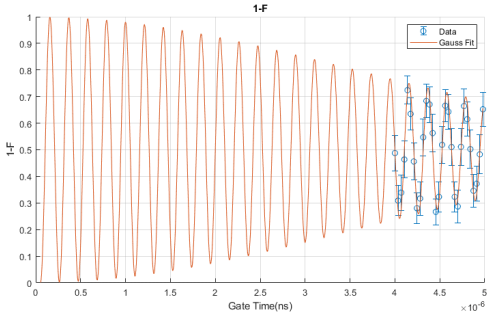


Figure 3.3: Ramsey Experiment on Charge Noise

3.2.4. DAC MODEL

The DAC model turns the continuous time and voltage signal into a discrete-time and voltage signal. This is done by repeatedly sampling the input signal at a specified time interval and mapping those values to discrete voltage levels. The given input signal is first down-sampled with a given sampling frequency and their values are quantized with given resolution and reference. After the quantized signal is up-sampled again with 0 and 1st order hold, it passes through an N^{th} order RC filter.

3.2.5. AMPLITUDE OPTIMIZATION

With or without DAC, generating a signal that gives high fidelity (> 99.99%) for all different combinations of pulse shapes and DAC parameters requires specific calculations. For example, to ensure the CPHASE fidelity is high enough, the area under the pulse

$2\pi \int J(t)dt$ should be close enough to π. To achieve this, amplitude optimization is implemented to find the optimal amplitude for the given signal and DAC. This is done by choosing the error function as $\cos(\pi \cdot \int J(t)dt)$.

3.2.6. QUANTUM PROCESS TOMOGRAPHY

Quantum Process Tomography (QPT) characterizes the process a quantum system undergoes [12]. To calculate the fidelity of our gate with different setups, QPT will be used to determine the process.

3

3.3. PULSE SHAPES AND THEIR EFFECTS ON CPHASE

Implementation of the CPHASE for spin qubits can be achieved when the coupling signal integral is 0.5 and the SWAP oscillation is canceled. In this section, it will be examined how different pulse shapes affect the performance of the CPHASE and later we will look at which one is more suitable for our DAC design.

For the following simulations, a set of parameters in Table 3.2 are used unless otherwise mentioned.

δE_z	$100MHz$
J	$10MHz$
$\alpha_{Barrier}$	24
J_{offset}	$60kHz$

Table 3.2: Noise and Distortion Models

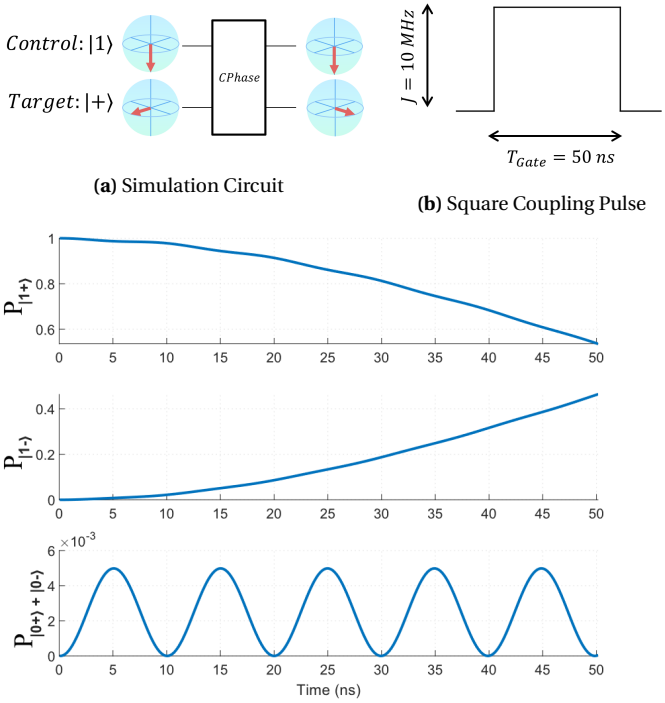
3.3.1. SQUARE PULSE

To see the example of how a square pulse evolves the system over time, the CPHASE gate in Fig. 3.4a quantum circuit is implemented with the square pulse. Initial states are chosen such that only the target qubit will rotate around the Z axis, while the control qubit's state shouldn't change.

The applied square signal pulse can be seen in Fig. 3.4b. Its duration is chosen as an integer multiple of the SWAP oscillation period ($\sim \frac{1}{\delta E_z} = 10ns$) to cancel oscillation effects. And the amplitude of the signal chosen such that $\int J(t)dt = 0.5$.

The probabilities of the states during the operation can be observed in Fig. 3.4c. The shown states are chosen as they clearly show the evolution of both the U_{CPHASE} and U_{SWAP} gates. The transition from $|1+\rangle$ to $|1-\rangle$ is performed by U_{CPHASE} and corresponds to the rotation of the target qubit.

Without the unwanted U_{SWAP} operation in the system, there would be only the transition between $|1+\rangle$ and $|1-\rangle$. However, U_{SWAP} creates an oscillation between other states. $|0+\rangle + |0-\rangle$ refers to the sum of all states other than $|1+\rangle$ and $|1-\rangle$, in other words residual state.



(a) Simulation Circuit

(b) Square Coupling Pulse

(c) Time evolution of the probabilities of the states $|1+\rangle$, $|1-\rangle$, and $|0+\rangle + |0-\rangle$.

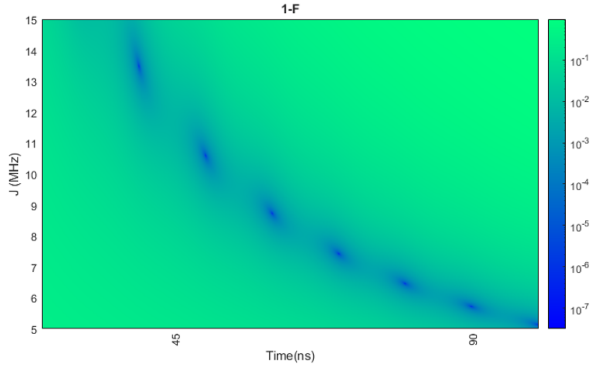
Figure 3.4: Simulation results of implementing CPHASE with a square pulse

Choosing the pulse duration as 5 times the SWAP oscillation period stopped the oscillation where residual state probability is almost 0. That removes the effect of U_{SWAP} , or in other words, it makes it equal to I. Ending the operation probability of residual states bigger than 0 will increase the infidelity of the CPHASE gate.

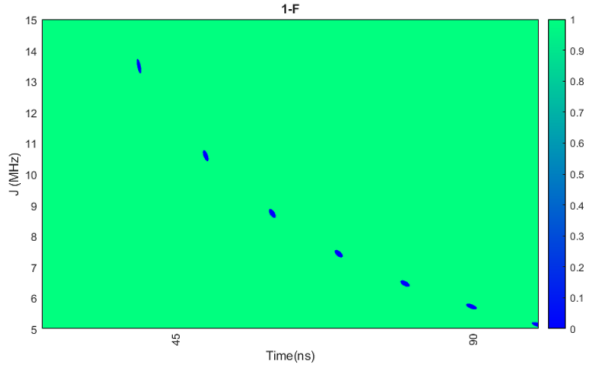
Different values for amplitude and gate time are swept in Fig. 3.5a. It is visible that high-fidelity operations can be achieved around specific islands. To see these islands even clearer, the plot is filtered with a binary filter that converts values below 10^{-4} to 0 and above 10^{-4} to 1. These islands occur at the intersections of two effects. The first is a hyperbolic curve where J and T_{gate} let $\int J(t)dt$ equal 0.5. The second effect is that the SWAP oscillation with roughly a period of $\frac{1}{\sqrt{\delta E_z^2 + J^2}} \approx 10$ns allows low infidelity only within certain periods over time. Since $\bar{F}_{total} = \bar{F}_{cphase} + \bar{F}_{swap}$, 10^{-4} infidelity is achieved only when the sum of infidelity of two operations is below 10^{-4}.

In Fig. 3.5b, it can be seen that the gate time can't be chosen arbitrarily since it has to be an integer multiple of the SWAP oscillation period ($\approx 1/\delta E_z = 10ns$). Also, due to the small size of the blue islands, the control needs to be very accurate on both the gate time and the amplitude of the signals. Otherwise, the performance can easily lie outside of the blue islands.

Targeting a single low-infidelity island in Fig. 3.5b with a square pulse can be imple-



(a) Infidelity of the square pulse with different J amplitude and gate duration



(b) Binary filtered heatmap with the cut-off at 10^{-4}

Figure 3.5: Simulation results of implementing CPHASE with a square pulse

mented with relatively simple electronics. However, the variance in the qubit characteristics requires more complexity in the electronics. The variance of δE_z shifts the islands to right-left, and different values of $\alpha_{barrier}$ and J_{offset} change the relationship between voltage and coupling. Therefore, the electronics should be able to handle any possible combination of δE_z, $\alpha_{barrier}$, and J_{offset}.

3.3.2. ADIABATIC PULSE

The Adiabatic Theorem claims that for a gradual change of the Hamiltonian from $H(0)$ to $H(T)$, the nth eigenstate of $H(0)$ will be carried to the nth eigenstate of $H(T)$ [13]. Let us consider a hypothetical situation in which a pendulum oscillates at the north pole. The pendulum is taken slowly from the north pole to the equator, traversing a quarter of the equator's circumference, and then returned to the north pole. After completing its journey, the swing will still have the same amplitude, but it will swing in a different direction as it has gained phase adiabatically during its movement.

To understand how qubits evolve under an adiabatic process, let us run a simulation.

The same simulation will be performed as before in Fig. 3.4a but this time with an adiabatic pulse. In this example, a pulse named **Tukey** will be used. It is preferable to use this pulse as the initial example, since it is simpler to explain than the others. We will examine other pulses in the following sections.

The Tukey pulse is created by modifying a square pulse with raised cosine edges. The ratio of the raised cosine edges and the square pulse varies depending on the value of the coefficient λ. The equation for the Tukey pulse is as follows:

$$w(t,\lambda) = \begin{cases} \frac{1}{2-\lambda}\left[1-\cos\left(\frac{2\pi t}{\lambda t_g}\right)\right] & 0 \le t \le \frac{\lambda t_g}{2} \\ \frac{2}{2-\lambda} & \frac{\lambda t_g}{2} < t < t_g - \frac{\lambda t_g}{2} \\ \frac{1}{2-\lambda}\left[1-\cos\left(\frac{2\pi(t_g-t)}{\lambda t_g}\right)\right] & t_g - \frac{\lambda t_g}{2} \le t \le t_g \end{cases}$$

The optimization of the amplitude, which was discussed in Section 3.2.5, is used to ensure that the integral of the pulse remains the same regardless of the λ value. A selection of Tukey pulses with varying λ values is shown in Fig. 3.6.

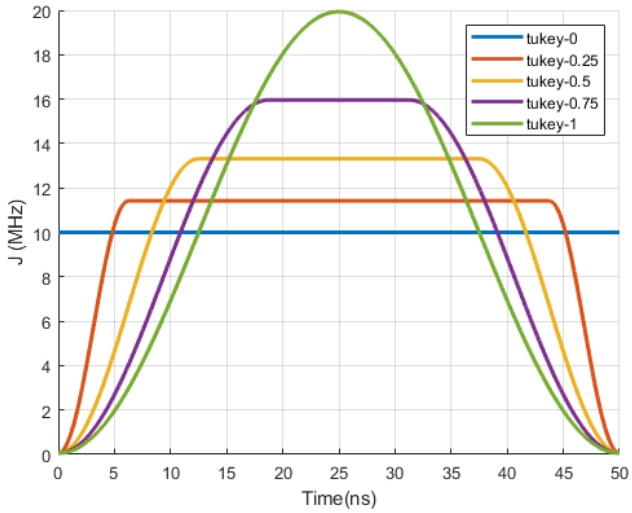


Figure 3.6: Amplitude optimized Tukey pulses with various $\lambda values$

To investigate the evolution of a quantum system with an adiabatic J pulse, a quantum circuit (see Fig. 3.4a) was constructed and a Tukey-1 pulse was used to generate J. The time evaluation of the states can be found in Fig. 3.7. The transition from $|1+\rangle$ to $|1-\rangle$ is similar to the square pulse in Fig. 3.4c. The $(0+\rangle + 0-\rangle)$ state undergoes accumulation and decreases instead of a SWAP oscillation. To gain a better understanding of how it works, we must analyze the non-diagonal elements of the U_{SWAP} , as they are responsible for the oscillation.

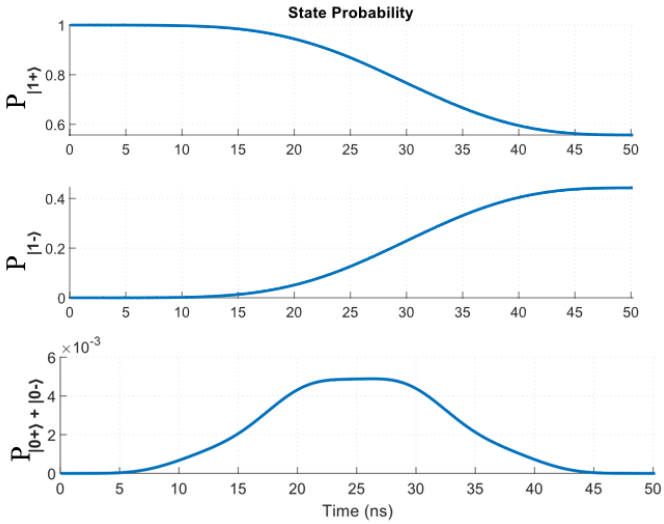
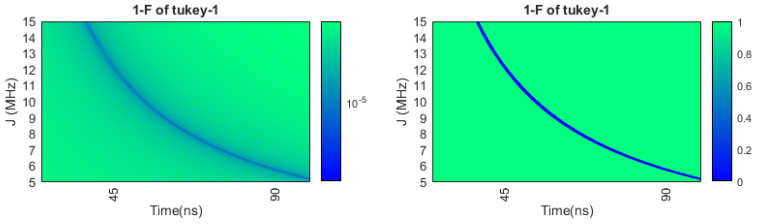


Figure 3.7: Time evolution of the probabilities of the states $|1+\rangle$, $|1-\rangle$, and $|0+\rangle + |0-\rangle$ with Tukey-1 pulse

$$-\frac{iJ(t)\cdot\sin\left(\pi t\sqrt{\delta E_z^2+J^2}\right)}{\sqrt{\delta E_z^2+J^2}}$$

If a square pulse is selected for $J(t)$, it will generate only a sine wave. However, when $J(t)$ is selected as a Tukey pulse, it modulates the sinusoidal function with a lower frequency cosine wave. This leads to a gradual alteration of the $(0+\rangle + 0-\rangle)$ state instead of a sudden oscillation.

Let us now consider what happens when we vary the amplitude and gate duration.



(a) Infidelity of the Tukey pulse with different J amplitude and gate duration

(b) Binary filtered heat-map with the cut-off at 10^{-4}

Figure 3.8: Simulation results of implementing CPHASE with a Tukey pulse

In Fig. 3.8a, both the amplitude and duration of the Tukey signal are varied to observe the effect on fidelity. To make the illustration more understandable, the results

are filtered with a value of 10^{-4} and shown in Fig. 3.8b. Compared to the square pulse results in Fig. 3.5b, the places with high fidelity are on the same hyperbolic curve as $\int J(t)dt = 0.5$, however, now without any islands. It eliminates any inaccuracies caused by the SWAP oscillation. The primary benefit of the adiabatic pulse is that T_{Gate} does not have to be an integer multiple of the SWAP oscillation. Thanks to the continuity, ensuring that the power of the J signal is equal to 0.5 would be enough to get high fidelity.

The lack of SWAP oscillations means that fidelity is no longer influenced by changes in δE_z. However, it still depends on $\alpha_{barrier}$ and J_{offset}.

OTHER ADIABATIC PULSES

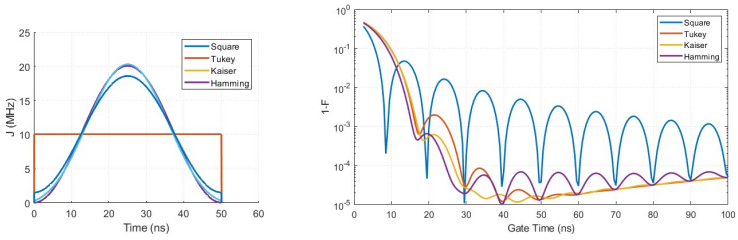
Let us take a look at how different adiabatic pulses will affect the qubit system. We will limit our examination to three mainly used adiabatic pulses in the literature [14]: Hamming, Kaiser, and Tukey.

In Fig. 3.9a, various adiabatic pulses can be observed. To compare, also a nonadiabatic square pulse is plotted. In the time domain, these operations appear familiar, yet their effect on the qubit system is quite distinct. In Fig. 3.9b, different pulse shapes are simulated while the gate time is swept and amplitude optimization is enabled.

The square pulse infidelity reveals the SWAP oscillation once more. When the gate time is increased, J has to be reduced to keep the power of the coupling signal ($J \cdot T_{Gate}$.) constant. This also decreases the $J/\delta E_z$ ratio, and thus the SWAP oscillation. Therefore, a longer gate time causes damping on the infidelity of the square pulse in Fig. 3.9b.

For adiabatic pulses in Fig. 3.9b, one common observation is that they cannot reach low infidelity like square pulse when T_{Gate} is smaller than two and half SWAP oscillations (25ns). Since the adiabatic theorem says that the system should evolve gradually, that can explain why adiabatic pulses cannot achieve fast gates, while nonadiabatic pulses can. However, it is not clear where this border is.

The infidelity rises slowly with gate time over 50 ns because over time the coupling offset creates too much coupling and increases the infidelity.



(a) Adiabatic Pulses over time

(b) CPHASE gate infidelity versus gate time for different pulse types

Figure 3.9: Adiabatic Pulses

In state-of-the-art implementations, the gate time is usually chosen around 50 ns [7], [8]. This is because it does not require high J. As explained in Section 2.1.2, the voltage-J relationship is exponential; when higher values of J are needed, the accuracy of the voltage increases rapidly. Therefore, we also decided to choose 50 ns gate time.

For a 50-ns gate time, the Tukey and Kaiser pulses yield similar results. We opt for the Tukey pulse because of its simpler formulation.

In this chapter, we discussed how to obtain a high-precision CPHASE gate through both adiabatic and nonadiabatic evolution. The main difference between these two evolutions is the cancelation of the SWAP oscillation. Nonadiabatic gates require a gate time that is an integer multiple of the SWAP oscillation period. On the other hand, adiabatic gates modulate the SWAP oscillation with low-band signals to create a gradual increase and decrease of residual states.

3.4. ELECTRONIC REQUIREMENTS FOR THE CPHASE GATE FIDELITY

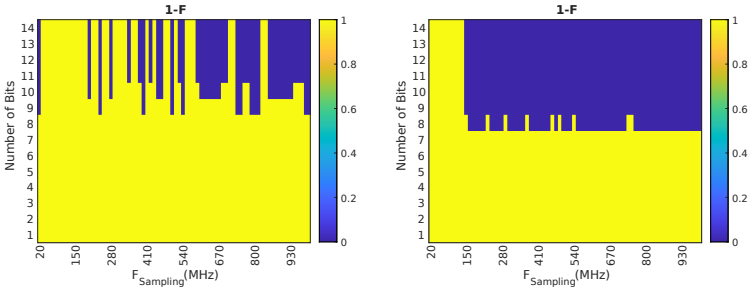
In this section, the influence of resolution and sampling rate will be separately assessed on the CPHASE and SWAP oscillations. When performing the calculations and simulations, two criteria will be taken into consideration.

- $\int J(t)dt = 0.5$
- $U_{SWAP} = I$

In order to quickly compare the DAC requirements for adiabatic and nonadiabatic pulses, Fig. 3.10 shows the sweeping of both the sampling frequency and the resolution of a zeroth-order hold DAC. In the following sections, we will go through the process of determining the required resolution and sampling rate. However, Figure 3.10 provides a general overview. For the demonstration, all graphs are filtered with a binary filter of 10^{-4}. In Fig. 3.10a, the blue high-fidelity points occur periodically until 860MHz. This is because the sampling interval is in sync with the SWAP oscillation cycle during the sweep. However, after 860MHz, its sampling period becomes low enough to keep T_{Gate} synced with SWAP oscillation. For the resolution, high fidelity is reached after the resolution is high enough to let $J \cdot T_{Gate}$ equal 0.5. It is worth noting that the reference of the DAC may influence the resolution of this DAC.

The Fig. 3.10b indicates that the requirements for the Tukey pulse are lower both in terms of resolution and sampling rate. Achieving accurate $\int J(t) \cdot dt$ still holds for resolution. However, during the adiabatic pulse, the duration of the pulse that has high J values is less than that of the non-adiabatic pulse. Therefore, error accumulation in that region is much lower with adiabatic gates. And for lower J values, the voltage-J relationship is much more linear. The sampling rate can also be lower for adiabatic than for non-adiabatic pulses, while still ensuring that fidelity is maintained. However, this is a topic of another section (Section 3.4.1).

We can draw a lot of conclusions from Fig. 3.10, but it is only applicable to a single set of qubit parameters. In the next sections, the DAC's resolution and sampling frequency will be found for a range of qubit parameters. During the calculations, targeted specs for qubit characteristics and gates can be found in Table 3.3.



(a) Square Pulse **(b)** Tukey Pulse

Figure 3.10: Infidelity of Square and Tukey pulses with different DAC setups

δE_z	90 – 110 MHz
α	25 – 50
J_{offset}	5 – 50 kHz
T_{Time}	50 ns

Table 3.3: Noise and Distortion Models

3.4.1. SAMPLING FREQUENCY

NON-ADIABATIC GATE

The precision of the U_{CPHASE} is contingent upon $\int J(t)dt = 0.5$. For a square pulse, this can simply be expressed as the product $J \cdot T_{Gate} = 0.5$, since it is time-independent during the pulse duration. In order to keep the infidelity below 10^{-4}, the rotation angle created by $J \cdot T_{Gate}$ should follow: $\cos((J \cdot T_{Gate} - 0.5)/2)^2 > 0.9999$ [15]. From here it can be found that $|J \cdot T_{Gate} - 0.5| < 0.00637$.

By selecting a $J/\delta E_z$ ratio of 10, the amplitude of the SWAP oscillation will remain constant regardless of the δE_z value. During the simulations, it is observed that at this ratio, infidelity of U_{CPHASE} is more dominant for sampling frequency. The equation can be rewritten as $0.49363 < 10 \cdot \delta E_z T_{Gate} < 0.50637$. For given specs in Table 3.3, T_{Gate} should have a 1.15 ns accuracy when δE_z is 110 MHz. That means the sampling frequency of the non-adiabatic gate should be bigger than **864 MHz**.

ADIABATIC GATE

Determining the minimum sampling rate for adiabatic pulses is more difficult. Decreasing the sampling rate can cause an adiabatic pulse to become a non-adiabatic pulse because of the rapid changes in the sampled signal. In Fig. 3.12, the Tukey signal is sampled at different rates. Preserving the adiabatic condition of the pulse should cancel the SWAP oscillation in the signal. However, it is not clear how the adiabatic characteristic of the

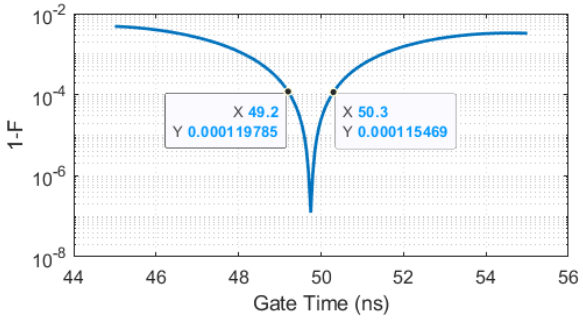


Figure 3.11: Caption

signal can be preserved. In Fig. 3.12, signals are simulated, and found that the adiabatic characteristic is preserved except for the last two sampled signals. However, the last signal can also reach high fidelity as it is two sequential non-adiabatic signals.

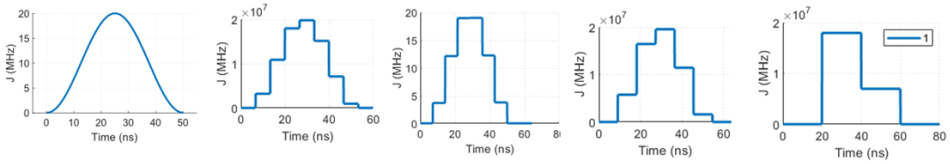


Figure 3.12: Tukey pulse sampled with various sampling frequencies

In Fig. 3.13, CPHASE fidelity versus sampling frequency of the Tukey pulse can be found. For each point, $\int J(t)dt$ is tuned with amplitude optimization. Therefore the primary reason for infidelity in the plot is SWAP oscillations. Below 140MHz, all low-infidelity notches are due to the sampling period syncing with the SWAP oscillation and creating sequences of non-adiabatic pulses. However, after 140MHz the infidelity of the qubit is kept under 10^{-4}. This plot shows that there exist criteria that can give the minimum sampling frequency.

As calculated in Section 3.1, the SWAP unitary can be written with a small simplification ($\sqrt{\delta E_z^2 + J(t)^2} \approx \delta E_z$) as below.

$$U'_{SWAP} = \exp(-i\,2\pi \cdot H'_{swap}(t))$$
$$U'_{SWAP} = \sigma_I \cdot \cos(\pi t \cdot \delta E_z) + i\left(\sigma_Z - \frac{J(t)}{\delta E_z} \cdot \sigma_X\right) \cdot \sin(\pi t \cdot \delta E_z)$$

The cancellation of the SWAP oscillation is possible when $U'_{SWAP} = I$. To do that, the non-diagonal elements of the matrix should be 0. σ_I and σ_Z can be ignored since they don't have any non-diagonal terms and σ_Z's evolution can be canceled with calibration. Therefore, the only term that has diagonal elements is also the only term that is time-dependent $\sigma_X \cdot \frac{J(t)}{\delta E_z} \cdot \sin(\pi t \cdot \delta E_z)$. Sampling $J(t)$ will only change the attribute of this

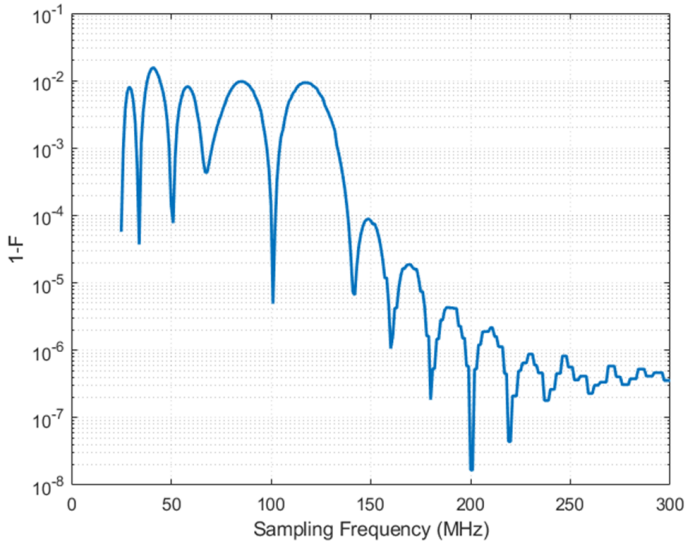
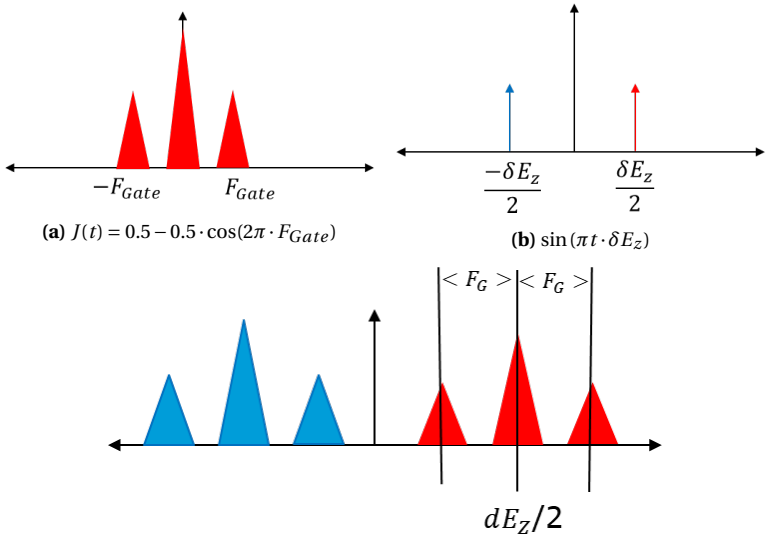


Figure 3.13: 1-F of adiabatic CPHASE operation versus sampling frequency

term. Therefore, this term can be used to find the minimum sampling frequency with the Nyquist Criterion.



(a) $J(t) = 0.5 - 0.5 \cdot \cos(2\pi \cdot F_{Gate})$

(b) $\sin(\pi t \cdot \delta E_z)$

(c) $J(t) \cdot \sin(\pi t \cdot \delta E_z)$

Figure 3.14: Frequency spectrum of signals

In Fig. 3.14a, the simplified frequency spectrum of the Tukey signal can be found. It consists of two tones on $F_{Gate} = 1/T_{Gate}$ from the cosine signal and a DC component. Similarly, Fig. 3.14b, shows the frequency spectrum of a sine signal with frequency $\delta E_z/2$. The phase difference of the sine signal is shown with different colors in the plot. Lastly, in Fig. 3.14c, the frequency spectrum of the non-diagonal unitary term can be found. Here, the Tukey signal is up-modulated with the sinusoidal signal to higher frequencies. On the different sides of the spectrum, tones have opposite phases. In Fig. 3.14c, two important outcomes can be found. Firstly, cancellation of the SWAP oscillation can be achieved only when the DC component of this spectrum is 0. Otherwise, the DC component in the spectrum will cause residual states in the evolution. Therefore F_{Gate} should be smaller than $\delta E_z/2$. It is evident from Fig. 3.9b that the highest fidelity operations are only achieved when the gate time is greater than $2/\delta E_z$.

Secondly, the Nyquist Criterion can be used to determine the minimum sampling frequency of this spectrum. Since the bandwidth of this signal is $\delta E_z/2 + F_{Gate}$, the sampling frequency should be bigger than the $\delta E_z + 2 \cdot F_{Gate}$. This explains the infidelity drop on the Fig. 3.13, where $\delta E_z + 2 \cdot F_{Gate} = 140MHz$.

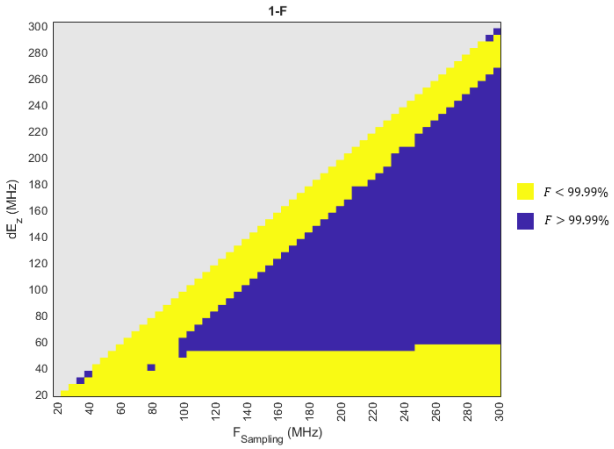


Figure 3.15: Infidelity of Adiabatic CPHASE for different δE_z and $F_{Sampling}$ with F_{Gate} = 20MHz

In Fig. 3.15, the relationship between different sampling frequencies and δE_z is examined using a $F_{Gate} = 20MHz$ Tukey pulse. The region where $F_{Sampling} < \delta E_z$ is grayed out since in that region high fidelity can only be reached by sequences of non-adiabatic pulses. In this plot, the two outcomes we have discussed can be seen clearly as the high-fidelity adiabatic gate is only possible when $F_{Sampling} > \delta E_z + 2 \cdot F_{Gate} = \delta E_z + 40$MHz and $F_{Gate} \cdot 2 = 40$MHz$< \delta E_z$.

In conclusion, two main criteria are found to evolve the system under an adiabatic operation. Firstly, an adiabatic gate should be two times slower than the oscillation. For the CPHASE gate, it means that $F_{Gate} < \delta E_z/2$. Secondly, the sampling period of the signal can be found with the Nyquist Criterion that $F_{Sampling} > \delta E_z + 2 \cdot F_{Gate}$. These two results are essential to find the specifications of the electronics for target qubit features.

As the target δE_z range is chosen as between 90-110 with F_{Gate} = 20MHz, the sam-

pling rate of the electronics should be bigger than **150MHz**.

ADIABATIC GATE WITH 1ST ORDER HOLD

In the preceding parts, signals were sampled using a zero-order hold. Nevertheless, a 1st order hold can be more advantageous since it reduces the abrupt changes of the signal, which is essential for adiabatic signals.

Once the signals are sampled in the time domain, the convolution of these signals with 0 or 1st-order hold windows produces the hold operation. These windows can be found in Fig. 3.16. Zeroth order hold has a square window while 1st order hold has a triangular window.

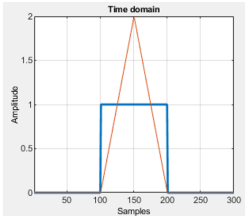


Figure 3.16: Zero and First Order Hold

However, since these windows are on the voltage domain, they will be exponential on the J domain. In Fig. 3.17, these windows are shown in the time and frequency domain. Convolution in the time domain is equivalent to multiplication in the frequency domain. That means sampled signals will be multiplied with 0 or 1st-order spectra. The Sinc function of zero-order hold can be seen clearly for the spectrum. 1st order hold spectrum does not have notches and it has lower power on high frequencies.

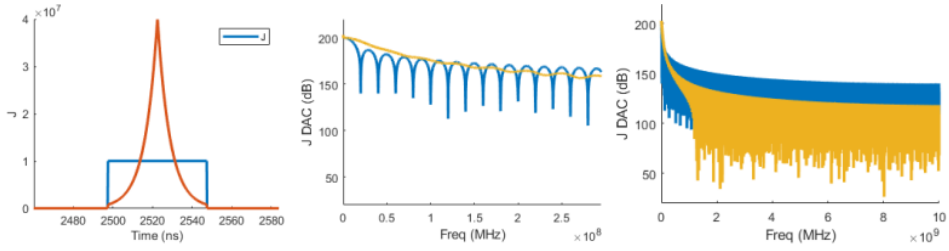
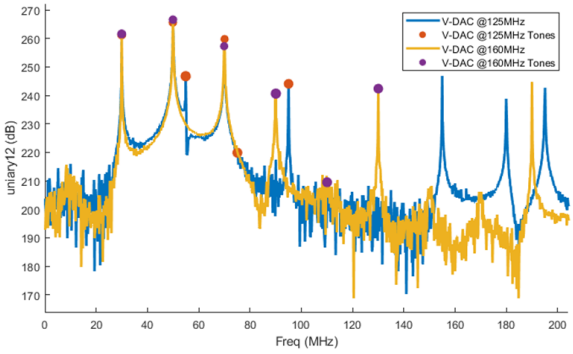


Figure 3.17: Zero and first-order hold on J domain

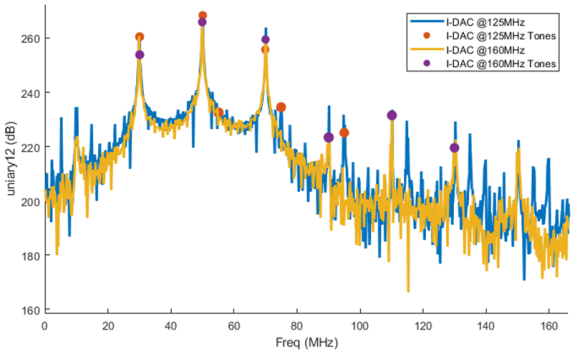
To see how different hold orders change the evolution, in Fig. 3.18a, the frequency spectrum of $\frac{J(t)}{\delta E_z} \cdot \sin(\pi t \cdot \delta E_z)$ can be found for different sampling rates. For a high-fidelity operation, this signal should be sampled with the Nyquist criterion. The yellow curve has a higher sampling rate than the minimum sampling frequency (140MHz). Tones of the first two harmonics of the sampled signal are marked with violet. The left three tones are the first harmonic of the Tukey pulse, with the remaining tones being the second harmonic of the sampled Tukey pulse. Due to the Sinc envelope of the zeroth-order hold, the middle tone in the second harmonic is suppressed until the noise level.

As can be seen, two harmonics are separated with 20MHz. A lower sampling rate of 125Mhz creates the blue curve, with orange tones. As can be seen at 55MHz, one of the tones interferes with the main tone. That destroys the characteristics of the main signal.

Now, if we look at the same setup in Fig. 3.15 with 1st order hold, at a 160MHz sampling rate, the tones are separated again nicely. Even though the first harmonics are similar to zeroth-orders, the tones of the second harmonics (90-110-130MHz) are quite different. Rather than the middle tone, side tones are suppressed to the white noise limit. At the 125MHz sampling rate, the interfering tone lies under the quantization noise level.



(a) Zero-order Hold



(b) 1st Order Hold

Figure 3.18: Frequency spectrum of signals

The change of the hold order from 0 to 1st, filters the side tones of harmonic. That enables us to lower the sampling frequency until the main tone in the second harmonic interferes with the first harmonic. It can be formalized as $F_{Sampling} > \delta E_z + F_{Gate}$ from the frequency spectrum in Fig. 3.14c. For given targets, the minimum required sampling rate is **130MHz** for a 1st order hold.

3.4.2. RESOLUTION

Resolution of the DAC mostly influences the fidelity of U_{CPHASE} operation. To achieve high fidelity, the resolution of the DAC should be high enough to let $\int J(t)dt = 0.5$.

NON-ADIABATIC GATE

For the resolution requirements, the same equation will be used as in Section 3.4.1. To keep infidelity below 10^{-4}, $|J \cdot T_{Gate} - 0.5|$ should be below 0.00637.

With the targeted qubit parameters in Table 3.3, the output voltage range can be calculated as $ln(10e+6/J_{offset,min})/\alpha_{Barrier,min} = 304.5mV$, and the accuracy should be 0.39mV. From here minimum DAC resolution can be found as **10 bits**.

ADIABATIC GATE

For the adiabatic gate, formulating the resolution requirements is not easy for varying parameters, since it is a discrete-time problem. Its resolution requirements can be found in the simulation results. However, due to its discrete behavior, we could miss a combination of points where the DAC resolution is not enough. Therefore, an architectural solution to encompass different parameters is proposed.

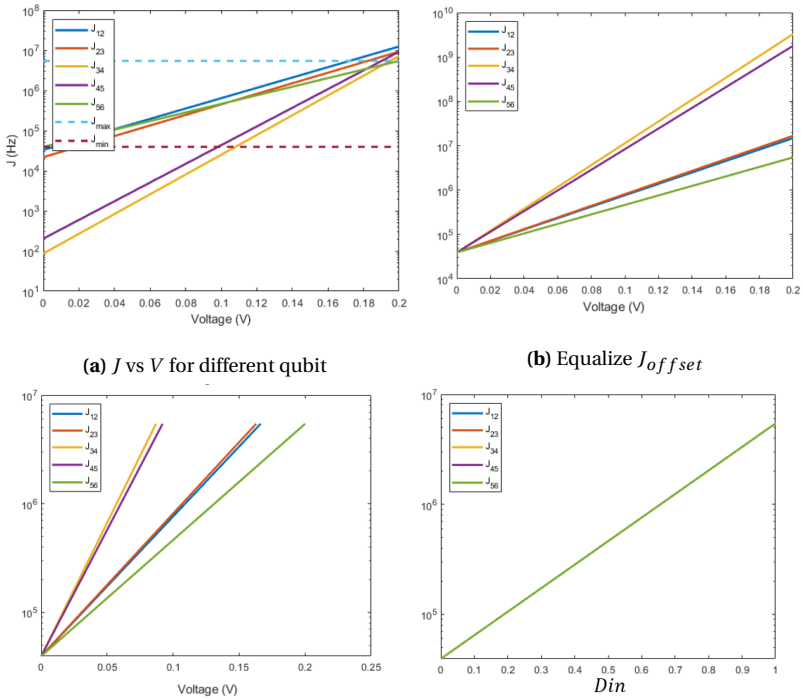
Proposed solution steps are explained in Fig. 3.19. Different $J-V$ relations are extracted from [9] as a case study. For different qubit pairs, $\alpha_{barrier}$ and J_{offset} create different slopes in Fig. 3.19a. To make sure the resolution of our DAC will cover all possible relations, these lines should be equalized. Firstly, all bias voltage levels are tuned to set the coupling offset of the different pairs to the worst coupling in the system, J_{min}. This can be done easily by a DC-biasing DAC. With that, the $J-V$ relations of different pairs become as shown in Fig. 3.19b. After that, the highest J value can be chosen as the lowest peak point, J_{max}, and lines become like in Fig. 3.19c. After this point, only the slopes of the lines are different. To equalize them for the digital input of the DAC, different reference values can be chosen. That will make the DAC free from the differences of slope as in Fig. 3.19d.

To achieve such a system, the DAC architecture is designed as a two-stage architecture as in Fig. 3.20. The first stage, the Reference-DAC (REF-DAC), will provide the required reference to equalize different slopes. The second stage, the Pulse-DAC, creates the pulse shape into qubits. In this system, for different qubit pairs, the REF-DAC should provide a different reference while the PULSE-DAC will use the same digital input for all qubits. The REF-DAC resolution should ensure a high-fidelity U_{CPHASE} operation, meanwhile, the PULSE-DAC will be responsible for creating an adiabatic pulse.

To find the resolution of the REF-DAC, the relation between the amplitude of the Tukey pulse and the fidelity should be observed. In Fig. 3.21, the normalized amplitude of Tukey swept and fidelity is calculated. To achieve higher than 0.9999 fidelity, amplitude accuracy should be below 3%.

The resolution of the PULSE-DAC can be calculated by the highest and the lowest slope for the 1st-order signal. The sampling frequency can be used to determine the number of sample points, and the slope between them can be calculated. With the maximum and the minimum slope, the number of bits can be found as $log_2(\frac{I_{max}-I_{min}}{I_{min}})$.

With the targeted qubit parameters in Table 3.3, the resolution of both the REF-DAC and the PULSE-DAC are found as **5-bit**. However, it should be noted that only half of



(a) J vs V for different qubit

(b) Equalize J_{offset}

(c) Limiting maximum J

(d) Equalization of line with reference

Figure 3.19: Equalization $J - V$ Relation

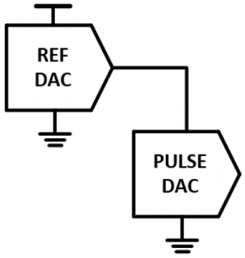


Figure 3.20: Two-stage DAC architecture

the REF-DAC output levels will be used. Its reason is for the specs target, the minimum reference needed for PULSE-DAC is half of the maximum value.

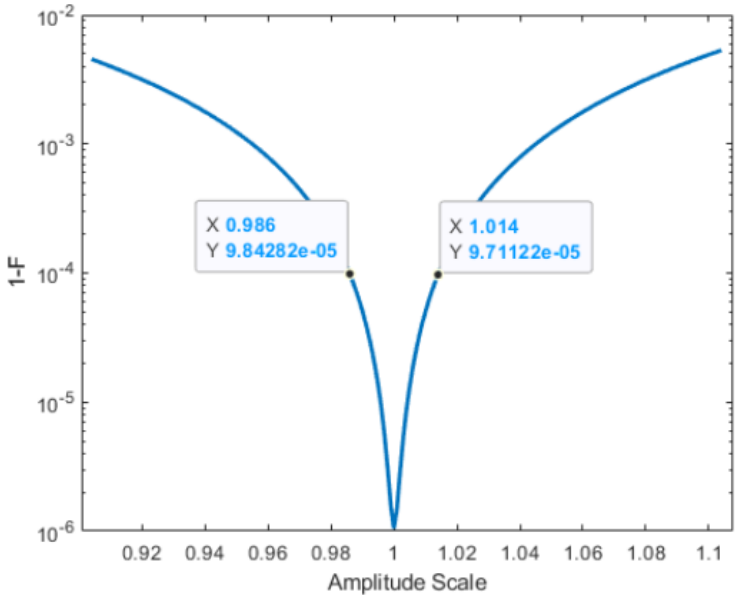


Figure 3.21: Fidelity vs Amplitude of Tukey Pulse

3.4.3. COMPARISON OF THE NON-ADIABATIC AND ADIABATIC SIGNAL REQUIREMENTS

In the Table 3.4, the obtained specification for the square and Tukey pulse is found, together with an example from the literature. As can be seen, the Tukey pulse requirements are much lower than the square pulse both in terms of resolution and the sampling rate. In the work [10], the square pulse requirements are a bit different than our calculations for a square pulse. The main reason for that could be the choice of different gate times and the J-voltage relation.

Specs	Square Pulse	Tukey Pulse Zero-Order Hold	Tukey Pulse First-Order Hold	[10](Square Pulse)
Number of Bits	10	8	6-5	11
Sampling Rate	864MHz	150 MHz	130 MHz	625MHz
Gate Time	50ns	50ns	50ns	100ns

Table 3.4: DAC Specs

3.5. CONCLUSION

In this chapter, firstly the resolution and the sampling requirements for adiabatic and nonadiabatic CPHASE gates are compared, and the adiabatic CPHASE gate is shown as a better option for the control electronics.

Later, the Nyquist Criterion is used to determine the minimum sampling frequency of adiabatic operation first time in the literature. Also, the reason for adiabatic gates losing their fidelity with lower gate time is explained with spectrum analysis of unitary operators.

Lastly, a two-stage DAC design is introduced to maintain high fidelity for different qubits. While the first stage preserves the integral of coupling strength, the second one is responsible for the sampling adiabatic pulse with high-fidelity.

4

DAC DESIGN

4.1. SYSTEM OVERVIEW

The results of Chapter 3 demonstrate that the adiabatic control of the CPHASE gate requires a much lower resolution and sampling rate than the nonadiabatic control. Also, the proposed two-stage architecture offers a scalable system while preserving the fidelity for varying qubit parameters. In this chapter, the design steps of the two-stage DAC will be examined. As shown in Section 3.4.3, the combination of a 6-bit reference DAC and a 5-bit pulse DAC can achieve high fidelity for the adiabatic CPHASE gate.

In Fig. 4.1, an example design of a spin-qubit processor is shown. The CPHASE gate will be controlled by the "fast DAC" in this design. Its output is AC-coupled with the "bias DAC" which is responsible for a precise DC biasing voltage control. Lastly, this design assumes that CMOS and qubit chips are connected by wire bonding. The other modules of the system are beyond the scope of this thesis.

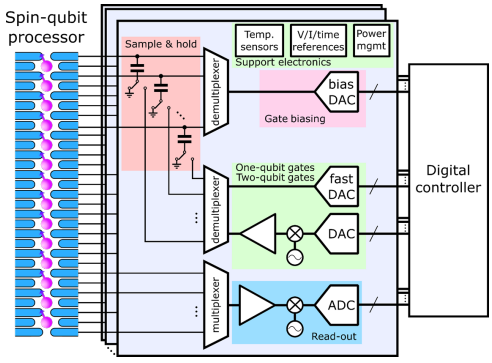


Figure 4.1: Spin Qubit Processor Interface [16]

The parasitics of the hold capacitor used for the DC biasing may influence the performance of the fast DAC. The total output capacitance of the fast-DAC is the sum of the

parasitic capacitances of the hold-cap, the wires, and the pads of the CMOS and qubit chip. It is estimated as 2.5 pF, however, due to mismatch and different signal paths, this value can be different for each pair of qubits.

The target working temperature of this two-stage DAC is below 1K. Taking into account the cooling power of a dilution refrigerator at 1K, the total power consumption of the chip must be below 100μW.

Since the power of the system is the main limitation for our circuit, first, different types of DACs will be compared by their power consumption. Later, the architecture of the DAC will be introduced. Finally, the peripherals to test the DAC will be presented.

4.2. POWER ESTIMATION OF DIFFERENT DAC TYPES

VOLTAGE DOMAIN

One simple example of a voltage domain digital-to-analog converter is resistor strings. It can be created by connecting the switching network to the taps of a resistor ladder. To ensure low output impedance, a buffer must be used to link the chosen tap to the load. However, below 1K, the resistors in the target technology become superconductive. Therefore, it is not feasible to employ a voltage-based design for frequencies lower than 1K. However, as we will see in Section 4.2, a buffer will already consume too much power, so this design is not feasible.

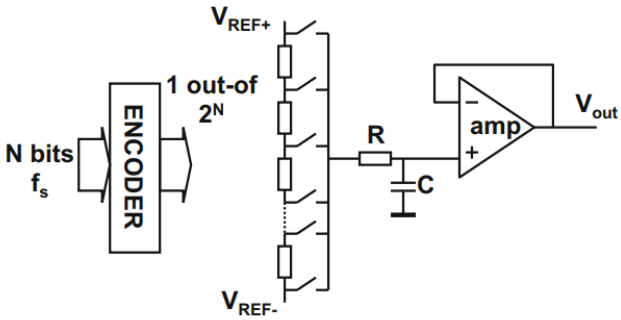


Figure 4.2: Voltage Domain DAC

CHARGE DOMAIN

The charge domain converter uses electric charge accumulation or discharge to provide an analog voltage. Charge-domain DACs use discrete charge packets instead of standard voltage-output DACs, which produce analog voltages directly, to create an analog signal.

The charge domain DAC uses zero-order hold. Therefore, it should have an 8-bit resolution and 160MHz sampling frequency (20MHz more than the minimum sampling frequency). The power of the capacitor bank and the amplifier should be calculated to get a general idea of the total power consumption of the charge DAC.

The unit capacitance size is chosen as 1 fF, which has a standard deviation lower than required (calculation of the required standard deviation will be discussed in Sec-

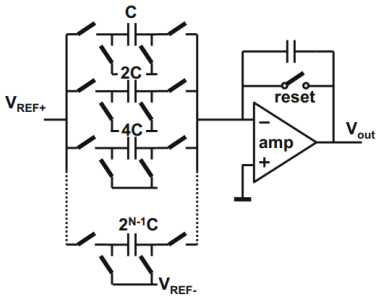


Figure 4.3: Charge Domain DAC

tion 4.3.1). In the worst case, all the capacitors in the bank will be charged/discharged. In that case, the power consumption of the capacitor bank can be calculated as $C_{Bank} \cdot V_{REF}^2 \cdot F_{Sample} = 2.816\mu$W.

After charges are disturbed in the capacitor bank, the amplify generates the required output voltage between sampling periods. The accuracy of this settling at the output of the amplifier can be determined as $\tau = \frac{C_{OUT}}{g_m \beta}$. Here C_{OUT} is the total output capacitance. β is the negative feedback coefficient and its value changes depending on the number of switching capacitors $\frac{M \cdot C}{2^N C}$. The required τ can be found by choosing the worst-case inaccuracy lower than 0.5 LSB. To calculate the power consumption of the amplifier, its bias current should be estimated. Ideally, an amplifier can drive all of its bias current to its output. With the assumption that folded cascode architecture will be used, there will be 4 branches in the amplifier, its current can be calculated as $I_{amplifier} = I_{out,max} \cdot 4$ where $I_{out,max}$ is the initial current that charges the output capacitance with the highest voltage step. It can be calculated as shown below:

$$
\begin{aligned}
V_{out}(t) &= V_{ref} \cdot (1 - \exp(-t/\tau)) \\
I_{out}(t) &= C_{out} \cdot \frac{d(V_{out}(t))}{dt} \\
&= C_{out} \cdot \frac{V_{ref}}{\tau} \cdot \exp\left(-\frac{t}{\tau}\right) \\
I_{out,max} &= I_{out}(0) \\
&= C_{out} \cdot \frac{V_{ref}}{\tau}
\end{aligned}
$$

From here the power of the amplifier can be calculated as follows.

$$
\begin{aligned}
P_{amp} &= V_{supply} \cdot I_{amplifier} \\
P_{amp} &= V_{supply} \cdot I_{out,max} \cdot 4 \\
P_{amp} &= 4 \cdot V_{supply} \cdot V_{ref} \cdot C_{out}/\tau
\end{aligned}
$$

From this equation, the power of the amplifier can be found as 1.88 mW. This is much higher than the cooling power of cryogenic fridges. That shows that a charge domain architecture cannot reach the DAC power requirements.

CURRENT DOMAIN

A current domain DAC feeds the load impedance directly from the current sources. These current sources must supply the load current, which results from the desired voltage output swing for a certain output load. To avoid a voltage drop at the output, a proper low-ohmic connection must be used to distribute the current over the array of current sources.

The output current will be used to charge and discharge the output load capacitance. This creates a first-order hold at the output and reduces the sampling rate by 20MHz as discussed in Section 3.4.3.

The power in the current DAC is spent on the currents used to charge and discharge capacitors. The charged or discharged energy of capacitance will be consumed by the current mirrors. Therefore, the total energy consumption of the current DAC is $C_{LOAD} \cdot V^2_{out,max}$. With a maximum output voltage of 0.4V, output capacitance of 2.5 pF, and a gate time of 50 ns, the power consumption can be found as 8 μW. This assumption of power consumption is much lower than the limitations of the refrigerator.

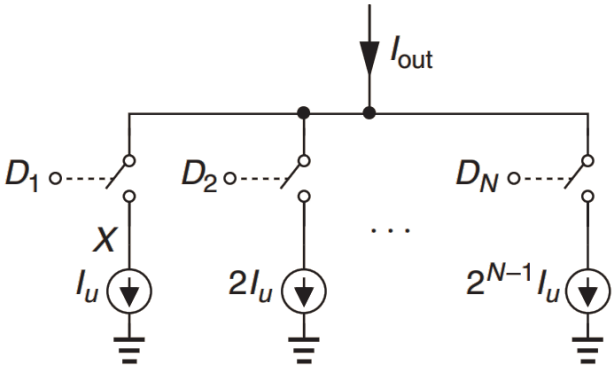


Figure 4.4: Binary-weighted current-switching DAC [17]

As explained in Section 4.1, the total capacitance of the signal path between the DAC and the pair of qubits can be different. This will result in the generation of different voltages with the same output current. However, this will be calibrated automatically while calibrating the reference DAC inputs for different qubits. Due to their different α_{Barier} values, each pair of qubits will have different REF-DAC inputs. Since the output capacitance and the α_{Barier} scale the voltage, these two can be calibrated together by reference DAC.

TIME DOMAIN

In addition to the division in the voltage, current, or charge domains, time division can also be used to achieve digital-to-analog conversion. The series of switching moments

in Fig. 4.5 changes the output signal. One-bit digital-to-analog conversion is frequently used for time-domain conversion.

The power consumption of the one-bit digital-to-analog conversion can be found similarly as in Section 4.2, 8 μW. However, a higher working rate increases power consumption. Another drawback of time-domain DAC is that it rapidly changes the signal slope. This may influence the cancellation of the SWAP oscillation.

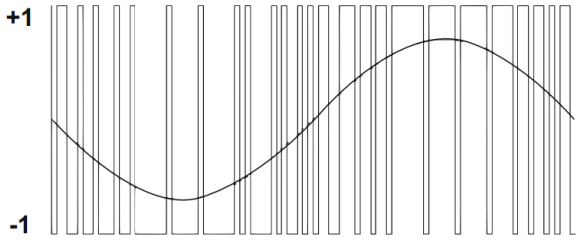


Figure 4.5: The signal is contained in the low-frequency portion of a pulse sequence

4.3. ARCHITECTURE

The current switching DAC is chosen because it has low power consumption, a first-order hold, and a simple design. The current switching DAC is also suitable to use in the two-stage architecture introduced in Section 3.4.2 to cover all possible variances in $\alpha_{Barrier}$ and J_{offset}. As it was explained in Section 3.4.3, the REF and PULSE-DAC will need 6-bit and 5-bit resolutions, respectively. The REF-DAC will operate at a 20 MHz sampling rate since it needs to update the reference before every gate. The PULSE-DAC sampling rate is chosen as 140 MHZ, which is 10 MHz higher than the required sampling frequency found in Table 3.4.

The voltage swing of the output ranges from 0.2 V to 0.6 V. Nevertheless as it can be seen in Fig. 4.6, due to the AC-coupled connection of the DAC output to the plunger through the DC-biasing network, it will result in a swing of 0.4V around the DC-biasing voltage on the plunger side. If all mirrors are inactive, the voltage output can be set back to 0.2V using a switch.

4.3.1. SEGMENTATION

Both pure unary DACs and pure binary DACs have certain issues. Unary DACs have a huge number of units, which makes it hard to control. However, binary DACs result in large DNL errors. To mitigate both problems, segmentation is generally employed. For example, the DAC is separate, by letting the lower N-bits of a converter be binary, whilst the upper M-bits are unary.

The segmentation ratio can be determined such that the unit mismatch does not cause more than 0.5 LSB DNL error. With an N-bit binary, the biggest DNL error appears when switching from $2^{N-1}-1$ to 2^{N-1}. During this transition, half of the units turn on, and the other half turn off. The DNL variation of this step can be found in the following [11].

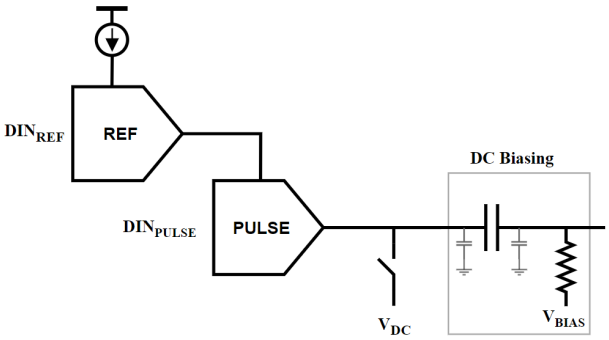


Figure 4.6: DAC Architecture

$$\sigma^2_{DNL,MAX} = \frac{(2^{N-1})\sigma^2_{LSB}}{A^2_{LSB}} + \frac{(2^{N-1}-1)\sigma^2_{LSB}}{A^2_{LSB}} = \frac{(2^N-1)\sigma^2_{LSB}}{A^2_{LSB}}$$

If M-bit unary units are added to these N-bit binary units, still the biggest DNL will be the same as only the N-bit binary. Unary units switch one by one, therefore their $\sigma^2_{DNL,MAX}$ is $\frac{(2^{N-1}-1)\sigma^2_{LSB}}{A^2_{LSB}}$. The DNL of an N-bit binary and M-bit unary DAC can be calculated as follows.

$$\sigma_{DNL,MAX} = \frac{\sqrt{(2^N-1)}\sigma_{LSB}}{A_{LSB}}$$

The mismatch of the ratio of the units for different segmentations of REF and PULSE-DAC can be found in Table 4.1. As can be seen, increasing the number of binary bits requires a better mismatch in the units. With Monte Carlo analysis, the standard deviation in the current of the unit current mirrors is found as 2.29% and 2.72% respectively for REF and PULSE-DAC. However, at cryogenic temperatures, the mismatch of current mirrors increases significantly [18]. To prevent this, segmentation should be chosen much bigger than required. Since full-unary increases the circuit complexity, 2-bit binary and 4-bit unary are chosen for REF-DAC, and 1-bit binary and 4-bit unary are chosen for REF-DAC.

4.3.2. REFERENCE DAC

The reference DAC is responsible for providing the different reference currents to the PULSE-DAC. Its segmentation is chosen as 4-bit binary and 2-bit unary. To target varying J_{offset} values, bias current changes from $1.77\mu A$ to $2.71\mu A$. The equivalent W / L ratio of the M_1 FinFET transistor is (5/40)(180n/40n) where (5/40) is the multiplier to stack ratio. And M_2 has (1/40)(180n/40n). With this ratio, the unit current of M_2 changes between 0.35 and 0.54 μA. Lastly, $M_S W, N$ switching transistors has the aspect ratio of (1/1)(45n/40n)

Binary	Unary	$\frac{\sigma_{LSB}}{A_{LSB}}$
6	0	1.48%
5	1	2.10%
4	2	2.99%
3	3	4.30%
2	**4**	**6.30%**
1	5	9.62%
0	6	16.67%

(a) REF-DAC

Binary	Unary	$\frac{\sigma_{LSB}}{A_{LSB}}$
5	0	2.10%
4	1	2.99%
3	2	4.30%
2	3	6.30%
1	**4**	**9.62%**
0	5	16.67%

(b) PULSE-DAC

Table 4.1: Required unit mismatch for different segmentations

4

As explained in Section 3.4.2, half of the REF-DAC output levels will not be used. Therefore, the output current of the REF-DAC changes between 11.3 and 34.7μA.With $V_{DS1,max}$=528 mV, that results in a 1.43μW power consumption.

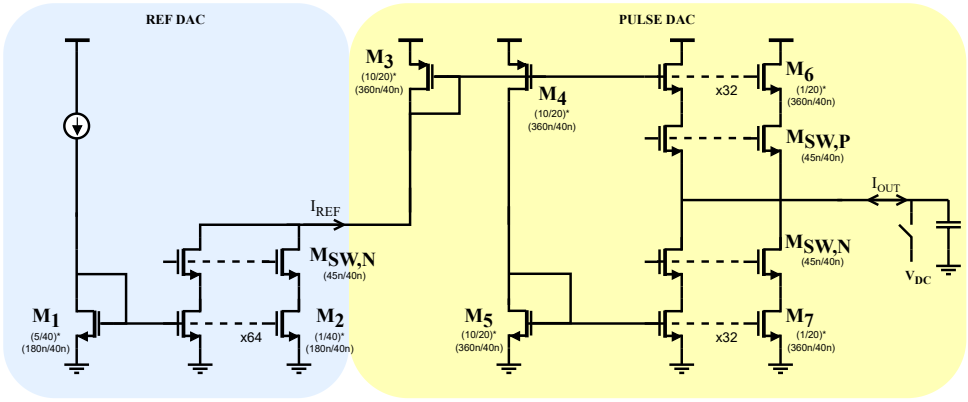


Figure 4.7: 2 Stage Current DAC Schematic

In Fig. 4.8, the DNL(LSB) versus output current can be found. The main reason for the inaccuracy is due to the settling time of the signal.

4.3.3. PULSE DAC

The PULSE-DAC in Fig. 4.7 is responsible for generating the adiabatic pulse. After J_{offset} is equalized for the target-qubit pairs, the same PULSE-DAC input data can be used for all pairs.

The PULSE-DAC consists of three main parts: biasing, pull-up, and pull-down networks. The Pull-up/down networks charge and discharge the output capacitance with the required current. The equivalent W / L ratio of the $M_{3,4,5}$ FinFET transistor is (10/20)·(360n/40n) where (10/20) is the multiplier to stack ratio. And $M_{6,7}$ has the (1/40)·(360n/40n)

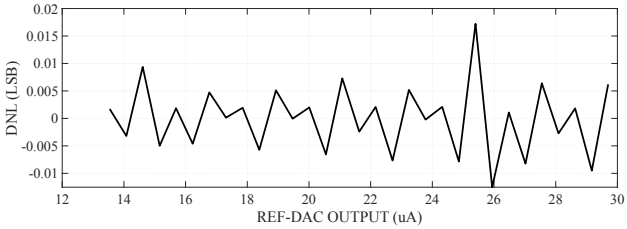


Figure 4.8: REF-DAC DNL(LSB)

ratio. Lastly, M_SW,N and M_SW,P switching transistors have the aspect ratio of $(1/1)\cdot$ (45n/40n). With this ratio, the output current changes between 1.13 and 111.04μA. The power of pull-up/down networks during the operation can be calculated as $C\cdot V_{out,max}^2\cdot F_{sampling} = 8\mu$W.

The biasing network creates the biasing voltages needed for the current mirrors. The biasing network branches have the same current and range as REF-DAC outputs 11.3-34.7μA. This results in a 27.76μW power consumption for each branch. Biasing currents are kept high to recover the system from kick-back that will be explained in Section 4.3.3.

The total power consumption of the REF and PULSE-DAC is 64.95 μW.

One drawback of this architecture is the wake-up time of the unit cells. When the unit cell switch, $M_{SW,N}$ is closed, $V_{DS6,7}$ in Fig. 4.7 should rapidly charge from 0 to the required voltage. During this charging, the current mirror will not operate accurately. To avoid this, generally current steering architectures are used to keep V_{DS} constant. That will increase the power consumption of the pull-up/pull-down network from 8μW to $V\cdot I\cdot 2 = 176\mu$W. However, since wake-up time did not create any problem for fidelity, current steering is not necessary.

COUNTER KICK-BACK

When a switch is open, the V_{DS} of a transistor is at 0V as in Fig. 4.9. After the switch closes, the charge sharing between the output node and V_D increases the V_{DS} rapidly. This sudden increase also increases the biasing node voltage V_A in Fig. 4.9 due to the AC coupling via C_{GD}. The biasing node slowly corrects the jump. However, due to the low current level of the biasing transistor, V_A slows down and creates an inaccuracy of the output current.

In Fig. 4.10, the simulation results of this phenomenon can be found for both pull-up and pull-down. First, the pull-up unit is activated. As can be seen, V_D has a rapid drop and drops V_A as well. After that, V_A cannot quickly return to its initial position, and that causes an error in the I_D current. The same effect can also be seen for the NMOS (pull-down).

In Fig. 4.11, the proposed solution schematic can be found. To remove the effect of kickback, we have applied an artificial counter kickback. When the current mirror switch toggles, a pulse will be sent through C_{GD} and C_{GS} of the M_3. That will generate a similar signal in the opposite direction. Although it will not cancel the kickback completely for different output levels, it helps the biasing network recover the biasing voltage V_A.

In Fig. 4.12, simulation results can be found. Compared to Fig. 4.10, the biasing node

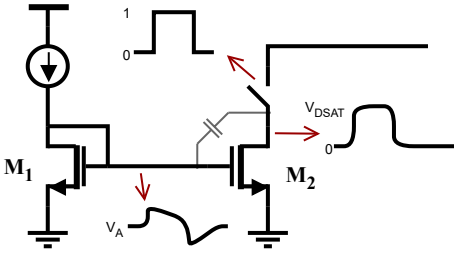


Figure 4.9: Kick-back to biasing node

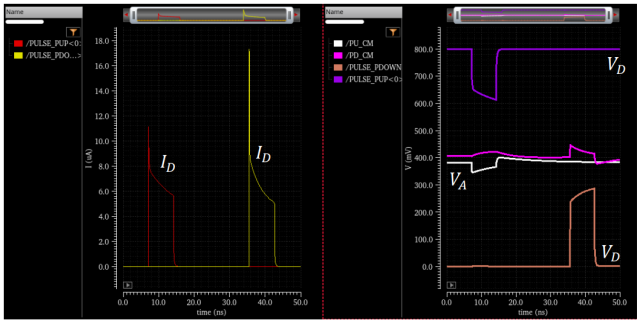


Figure 4.10: Simulation of kickback to biasing network

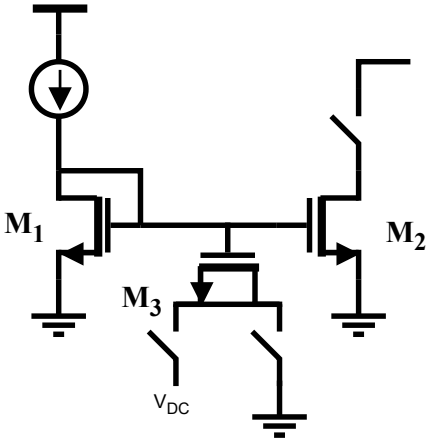


Figure 4.11: Counter Kickback

stays more constant over time. In Fig. 4.13, a comparison of the output currents can be found with and without counter-kickback. As can be seen, counter-kickback stabilizes the output current much earlier.

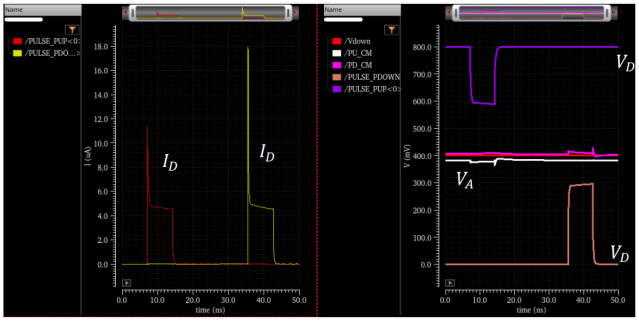


Figure 4.12: Simulation of kickback to biasing network with counter kickback

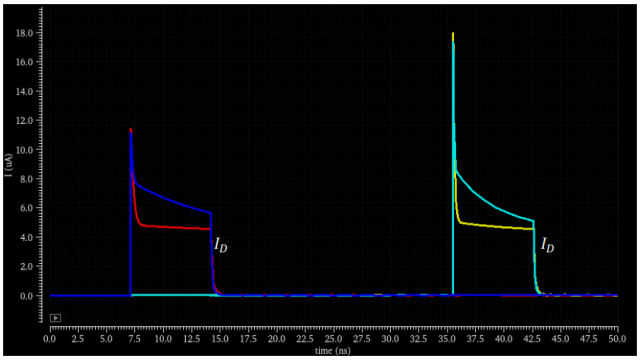


Figure 4.13: Comparison of the current of the current sources with and without kick-back

In figure Fig. 4.14, the DNL(LSB) versus different output currents can be found. The output currents are sampled at 140 MHz. Since output voltage will change the current linearity of current mirrors, DNL is generated on three different output voltages.

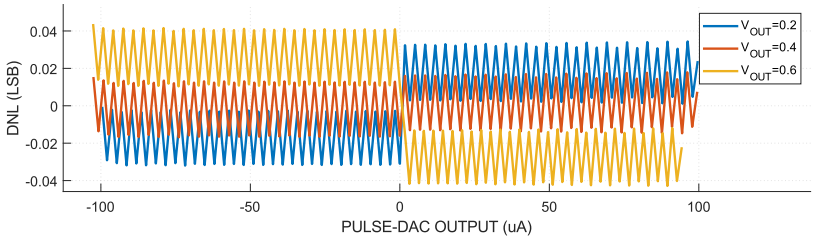
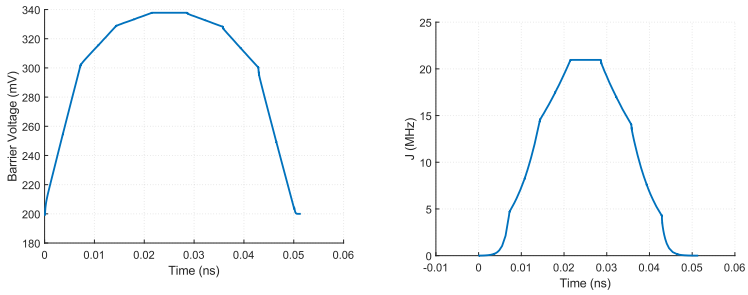


Figure 4.14: PULSE-DAC DNL (LSB)

In Fig. 4.15, generated signal on voltage domain and J domain can be found. This pulse is tested with the qubit model in Matlab and achieves more than 99.99% fidelity.



(a) Output matching of driver versus input voltage

(b) Output vs input voltage

Figure 4.15: A coupling strength signal generated with DAC

4.4. FLOORPLAN

The 2-stage DAC floorplan can be found in Fig. 4.16. The system is composed of three primary segments: digital front-end, analog core, and analog back-end. These three parts have different supply voltages and a common ground. The digital front end and analog core use low-power thin-oxide transistors, while the analog back-end uses thick-oxide transistors.

The digital front end is primarily responsible for managing the data that will be used for DAC control. While loading the data, the shift registers will be used to parallelize the serial input from DIN. After all serial data is sent to the shift registers, the LOAD signal will store this data in memories. The memories are controlled with counters that have only two simple control signals. RST can set the counter to its initial value, and START/STOP can control the counting. After that, BIN2THERM converts the binary code into thermometer code for the unary part of the DACs.

The analog core section consists of the two-stage DAC architecture explained in Section 4.3.

The analog back-end consists of mainly testing structures. A variable capacitor bank will be used to control the output capacitance. Multiplexers and demultiplexers are used to change the input and output of DAC and the amplifier. With these multiplexers/demultiplexers, the circuit can be used in three different modes. Firstly, the output of the DAC can be connected to the amplifier. Second, by connecting the TAKE_OFF and LANDING pads with wire bonding, the output of the DAC can reach the amplifier after going through two pads. This setup mimics the chip-to-chip connection. And lastly, the DAC's output can directly connect to the output PAD and the input of the amplifier can be connected to the input pad for testing purposes.

4.5. DIGITAL FRONT END

4.5.1. COUNTER

The counter is used to go through the memory addresses. Therefore, a simple up-counter with stop/stop and reset signal would be enough.

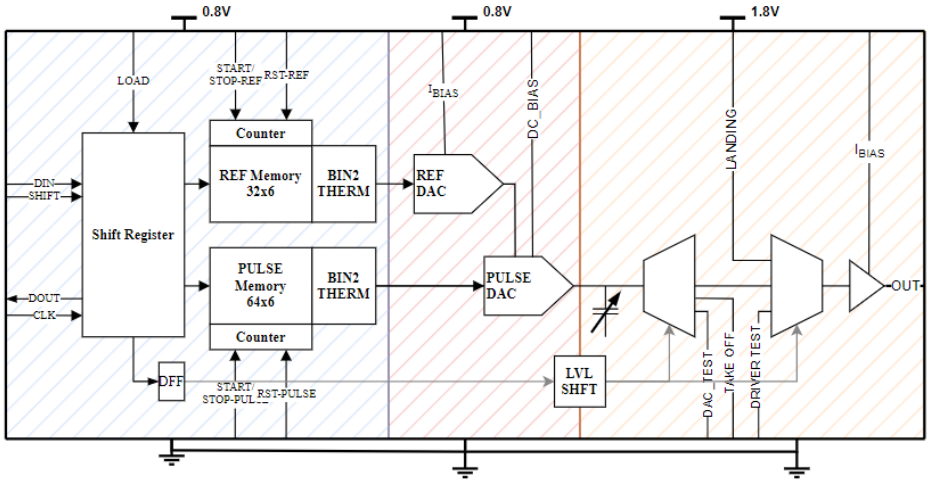


Figure 4.16: Floorplan of the DAC

The schematic of the up-counter with D-flip-flops is shown in Fig. 4.17. By connecting the inverted outputs of the previous flip-flop, the input clock frequency is divided by two for each flip-flop. This creates a binary counter on the stored data. START_STOP can stop the counting by stopping the input clock. RST resets all flip-flops to 0. After each flip-flop, buffers are used to drive signals to high-capacitive loads.

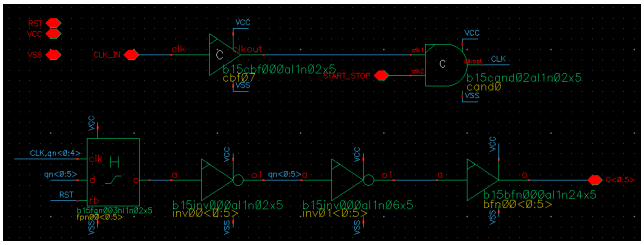


Figure 4.17: Counter

4.5.2. SHIFT AND MEMORY REGISTER

Shift and memory registers are designed together to make the layout design easier.

Its schematic diagram can be found in Fig. 4.18. SHIFT_CLK and LOAD_CLK are buffered in each register and pass through to the next register. The direction of the SHIFT_CLK and LOAD_CLK is in the opposite direction of Q to prevent any data loss due to clock skew.

After the serial input data shifts through the first flip-flops, the LOAD signal stores the data to the second flip-flop.

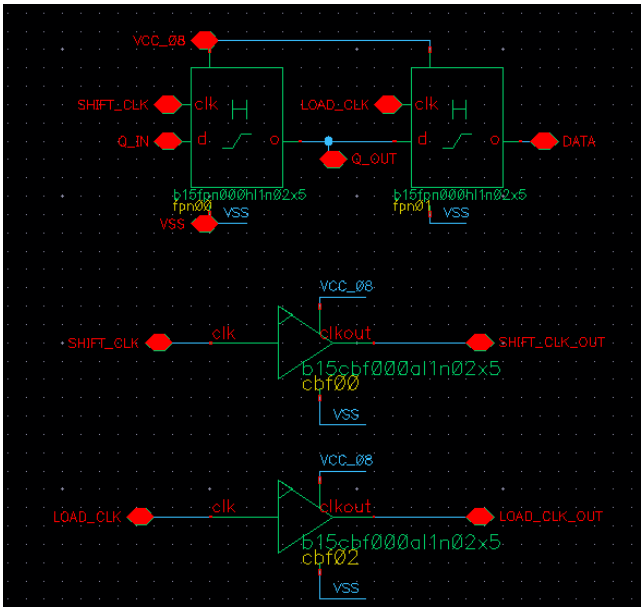


Figure 4.18: 1-bit shift register and memory unit cell

4.5.3. MEMORY

The memory consists of the shift-memory register and the multiplexers. After registers are stacked together, the multiplexer is connected to choose a single bit for a given time. This can be repeated for the number of bits that will be stored in a row. ADDR input will choose which row to be selected.

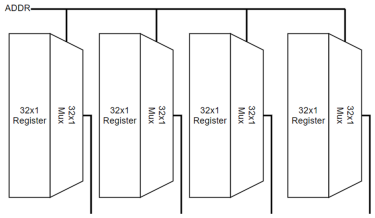


Figure 4.19: Memory

4.5.4. BINARY TO THERMOMETER CONVERTER

The binary-to-thermometer converter converts a 4-bit binary code into a 15-bit thermometer code. For the input value n, n logic-1s and $15-n$ logic-0s should be generated at the output. This conversion is made with logical gates in Fig. 4.20.

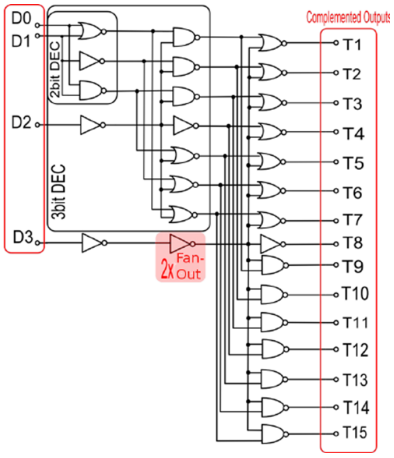


Figure 4.20: 4-bit Binary to thermometer code converter [19]

4.6. ANALOG FRONT END

4.6.1. ANALOG SWITCH

The bi-directional analog switch is one of the main units of the analog back-end. It consists of two thick-oxide PMOS and NMOS devices. It controls the connection of two nodes and it has a 34 Ω on-resistance.

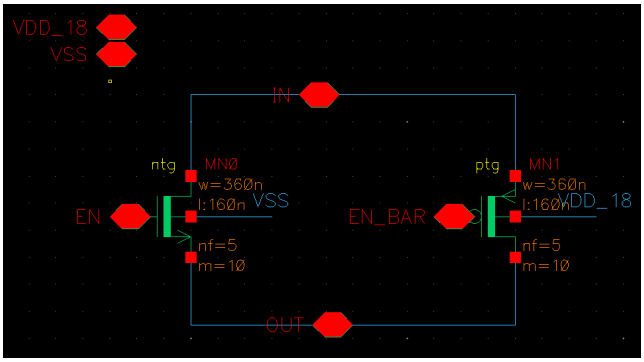


Figure 4.21: Analog Switch

4.6.2. LEVEL SHIFTER

The level shifter changes the voltage level of the signals. Control signals that come from the digital front-end have a 0.8 V voltage level while the analog back-end needs 1.8 V. Therefore, the signals are shifted up with a level-shifter before being used in the analog back-end.

The device features a latching comparator along with two inverter buffers. To ensure that the node between NMOS and PMOS is pulled down when the input signal changes,

NMOS transistors have a larger aspect ratio than PMOS transistors. After the comparator latches, the inverters buffer the output signal.

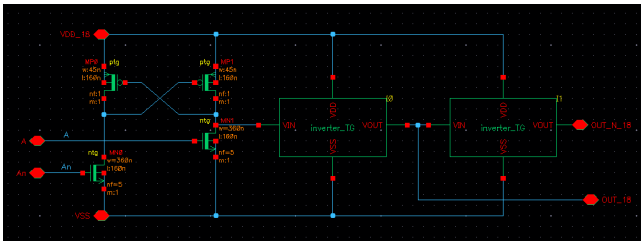


Figure 4.22: Level Shifter

4.6.3. MULTIPLEXER/DEMULTIPLEXER

The multiplexer/demultiplexer is built by a combination of level shifters and switches. It allows the user to select which analog signals are sent to and from the circuit.

4.6.4. CAPACITOR BANK

A capacitor bank can change the output capacitance of the DAC. It can be used to tune the output capacitance of the DAC to around 2.5 pF. To do that, 16 unit capacitances are connected to the output through switches. Each unit's capacitance is 49.20 fF and they are controlled with a binary code.

The voltage difference between switches of the capacitance bank should be small enough to prevent any infidelity. The ratio between a single capacitor unit and the rest of the output capacitance is C_u/C_{out} = 44.20fF/2500fF=0.017. The current that flows through the unit capacitor is $I_{switch} = I_{out} \cdot C_u/C_{out}$. From here, the voltage difference between the switch nodes will be $V_{switch} = I_{switch} \cdot R_{on} = I_{out} \cdot R_{on} \cdot 0.017$. When the DAC stops charging/discharging the output capacitance, the voltage drop at the output will be $V_{drop} = V_{switch} \cdot C_u/C_{out} = I_{out} \cdot R_{on} \cdot 0.017^2$. With a maximum DAC output current of 111μA, V_{drop} can be found as 1 μV. This provides a fidelity of over 99.99% for the worst-case scenario on J control.

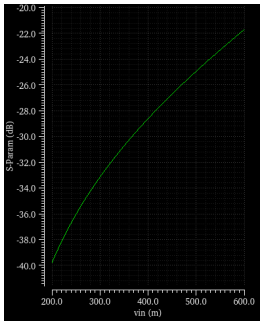
4.6.5. OUTPUT AMPLIFIER

To test the circuit at 4K temperatures, an output amplifier will be used to drive the signal to room temperature with 50 Ω cables. Therefore we need a 50 Ω driver to test our circuit at cryogenic temperatures.

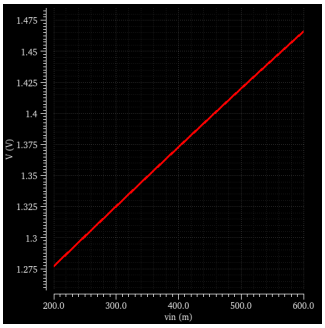
This driver is designed as a PMOS source follower and it has more than 20dB matching for inputs between 0.2-0.6V. Its simulation results can be found in Fig. 4.23a. Its power consumption is 48.675 mW.

4.7. LAYOUT

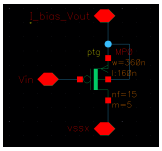
The layout of the reference and the current DAC is necessary to prevent non-linearities. To achieve this, all designs are split into unit cells as shown in Fig. 4.24. C and U_REF



(a) Output matching of driver versus input voltage



(b) Output vs input voltage



(c) Output Driver Schematic

Figure 4.23: Output driver simulation results and schematic

have the same circuit layout, while only the connection of high metal layers is different. In that way, the mismatch of the transistors is minimized. In the same way, A, B, and U_PULSE have the same structure until high-metals.

The general structure of the layout can be found in Fig. 4.25. The upper-row is the REF-DAC and the lower-row is the PULSE-DAC. To keep the output capacitance of the PULSE-DAC low, the U_{PULSE} units are kept in the middle, and A and B units are placed on both sides. Connections from the output of the REF-DAC are made from two sides, directly to the A, and B units. The reason is to prevent the voltage drop over the A, and B units. For the REF-DAC, the C units are placed in the middle and the U_{REF} units are at both sides. Finally, dummy cells are closing both ends of two DACs.

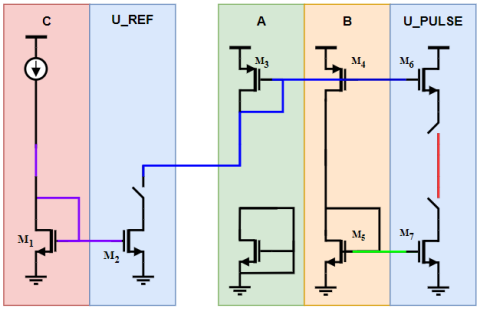


Figure 4.24: Layout Circuit Units

In the Fig. 4.26, the layout of the all-chip design can be found.

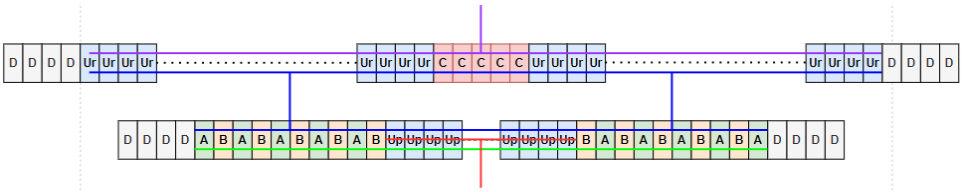


Figure 4.25: Abstract representation of the REF and PULSE-DAC Layout

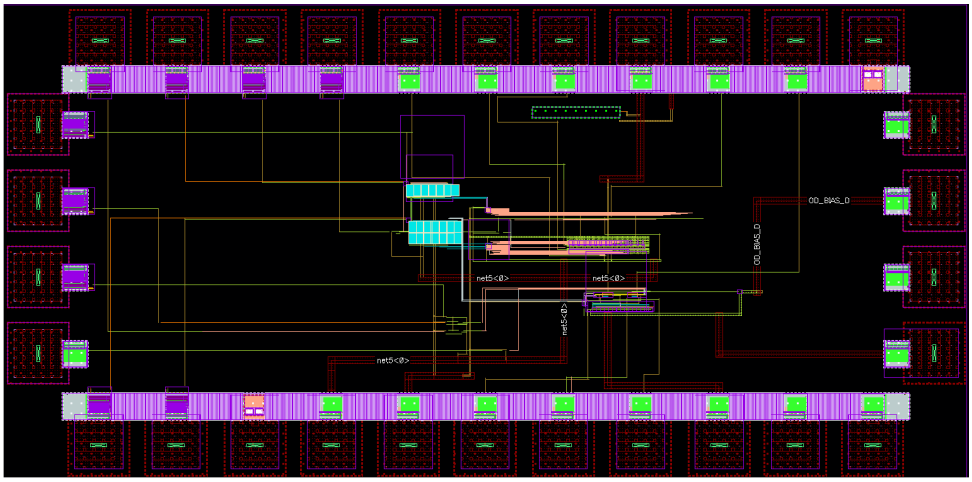


Figure 4.26: Layout of the full chip

5

CONCLUSION

5.0.1. CONCLUSION AND CONTRIBUTION

This thesis aimed to design a scalable cryogenic Digital-to-Analog Converter tailored to control the CPHASE quantum gate in semiconductor spin qubits. The two-qubit quantum system and DAC models were developed to simulate the influence of the control signal on the CPHASE gate and DAC requirements. It is shown that the adiabatic signal resolution and sampling requirements are much lower than those for nonadiabatic pulses. For this reason, it was decided to control the CPHASE gate with an adiabatic signal.

During the analysis to find the DAC requirements for the adiabatic gate, two phenomena were discovered for the first time in the literature. First, the minimum sampling rate of the DAC to control the adiabatic pulse was found by applying the Nyquist criterion to the unitary operator of the system evolution. Also, spectrum analysis of the time-dependent terms explained why adiabatic signals cannot achieve high-fidelity operations with lower gate times.

A two-stage DAC architecture is proposed to maintain the CPHASE fidelity by equalizing the J-Voltage characteristics for different qubits. The first stage, REF-DAC, has enough resolution to preserve the fidelity of the CPHASE for different values of $\alpha_{Barrier}$. The second stage, PULSE-DAC, is responsible for creating the adiabatic signal to cancel SWAP oscillations.

The DAC specifications are found to control the CPHASE gate for qubits with the following parameters: The frequency range of δE_z is between 90-110MHz, $\alpha_{Barrier}$ ranges from 25 to 50, J_{offset} ranges from 5kHz to 50kHz, and finally the gate time of the operation is 50ns.

The proposed two-stage DAC is implemented as a current-switching DAC with Intel's 16nm technology. The initial stage of the DAC, REF-DAC, has a 6-bit resolution and it only consists of a pull-down network. The second stage, PULSE-DAC, has a 5-bit resolution with pull-up and pull-down networks. DAC preserves the fidelity of the CPHASE with 74μW of power consumption.

5.0.2. FUTURE WORK AND DISCUSSION

In this work, an adiabatic process and its electronic requirements are examined for the first time in the literature. Therefore, a lot of different research is possible to extend the results.

To start, in this work, only predefined functions are used as an adiabatic pulse. However, the frequency spectrum analysis in this work showed how adiabaticity can be preserved by looking at the frequency spectrum. This enables the design of tailored signals for adiabatic pulses, which can even lower the DAC requirements.

The PULSE-DAC's resolution could also be lowered by using some non-regular segmentation such as hyperbolic or logarithmic. This would lower the resolution even more, since the voltage and the coupling strength have an exponential relation. Although this idea was a concern during the thesis, due to time limitations, it was not investigated.

When sending the pulse to the barrier plunger of a qubit pair, due to capacitive coupling, the voltage on the other plungers may be influenced. To overcome this problem, in the literature [9], the same signal is also sent to other plungers with a much smaller and negative amplitude. For this operation, the feasibility of this DAC should be investigated or an architectural solution should be proposed.

BIBLIOGRAPHY

[1] A. Montanaro, "Quantum algorithms: An overview," *npj Quantum Information*, vol. 2, no. 1, p. 15 023, Jan. 2016, ISSN: 2056-6387. DOI: 10.1038/npjqi.2015.23. [Online]. Available: https://doi.org/10.1038/npjqi.2015.23.

[2] L. Vandersypen and A. Van Leeuwenhoek, "Quantum computing - The next challenge in circuit and system design," *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 60, pp. 24–29, Mar. 2017, ISSN: 01936530. DOI: 10.1109/ISSCC.2017.7870244.

[3] C. G. Almudever, L. Lao, X. Fu, *et al.*, "The engineering challenges in quantum computing," *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pp. 836–845, May 2017. DOI: 10.23919/DATE.2017.7927104.

[4] H. Homulle, S. Visser, B. Patra, *et al.*, "Cryocmos hardware technology," pp. 282–287, 16 May 2016 through 19 May 2016 2016, Conference contribution. DOI: 10.1145/2903150.2906828. [Online]. Available: https://doi.org/10.1145/2903150.2906828.

[5] E. Charbon, F. Sebastiano, A. Vladimirescu, *et al.*, "Cryo-cmos for quantum computing," in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 13.5.1–13.5.4. DOI: 10.1109/IEDM.2016.7838410.

[6] D. P. DiVincenzo, "The physical implementation of quantum computation," *Fortschritte der Physik*, vol. 48, no. 9-11, pp. 771–783, Sep. 2000. DOI: 10.1002/1521-3978(200009)48:9/11<771::aid-prop771>3.0.co;2-e. [Online]. Available: https://doi.org/10.1002%2F1521-3978%28200009%2948%3A9%2F11%3C771%3A%3Aaid-prop771%3E3.0.co%3B2-e.

[7] T. F. Watson, S. G. J. Philips, E. Kawakami, *et al.*, "A programmable two-qubit quantum processor in silicon," *Nature*, vol. 555, no. 7698, pp. 633–637, Mar. 2018, ISSN: 1476-4687. DOI: 10.1038/nature25766. [Online]. Available: https://doi.org/10.1038/nature25766.

[8] X. Xue, M. Russ, N. Samkharadze, *et al.*, "Quantum logic with spin qubits crossing the surface code threshold," *Nature*, vol. 601, no. 7893, pp. 343–347, Jan. 2022, ISSN: 1476-4687. DOI: 10.1038/s41586-021-04273-w. [Online]. Available: https://doi.org/10.1038/s41586-021-04273-w.

[9] S. G. J. Philips, M. T. Mądzik, S. V. Amitonov, *et al.*, "Universal control of a six-qubit quantum processor in silicon," *Nature*, vol. 609, no. 7929, pp. 919–924, Sep. 2022. DOI: 10.1038/s41586-022-05117-x. [Online]. Available: https://doi.org/10.1038%2Fs41586-022-05117-x.

[10] J.-S. Park, S. Subramanian, L. Lampert, *et al.*, "13.1 a fully integrated cryo-cmos soc for qubit control in quantum computers capable of state manipulation, readout and high-speed gate pulsing of spin qubits in intel 22nm ffl finfet technology," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 208–210. DOI: 10.1109/ISSCC42613.2021.9365762.

[11] M. Pelgrom, *Analog-to-Digital Conversion*, en, 3rd ed. Cham, Switzerland: Springer International Publishing, Sep. 2016.

[12] M. Mohseni, A. T. Rezakhani, and D. A. Lidar, "Quantum-process tomography: Resource analysis of different strategies," *Physical Review A*, vol. 77, no. 3, Mar. 2008. DOI: 10.1103/physreva.77.032322. [Online]. Available: https://doi.org/10.1103%2Fphysreva.77.032322.

[13] D. J. Griffiths and D. F. Schroeter, *Introduction to quantum mechanics*, Third edition. Cambridge ; New York, NY: Cambridge University Press, 2018, ISBN: 978-1-107-18963-8.

[14] M. Rimbach-Russ, S. G. J. Philips, X. Xue, and L. M. K. Vandersypen, *Simple framework for systematic high-fidelity gate operations*, 2022. arXiv: 2211.16241 [quant-ph].

[15] J. van Dijk, E. Kawakami, R. Schouten, *et al.*, "Impact of classical control electronics on qubit fidelity," *Physical Review Applied*, vol. 12, no. 4, Oct. 2019. DOI: 10.1103/physrevapplied.12.044054. [Online]. Available: https://doi.org/10.1103%2Fphysrevapplied.12.044054.

[16] F. Sebastiano, J. van Dijk, P. t. Hart, *et al.*, "Cryo-cmos interfaces for large-scale quantum computers," in *2020 IEEE International Electron Devices Meeting (IEDM)*, 2020, pp. 25.2.1–25.2.4. DOI: 10.1109/IEDM13553.2020.9372075.

[17] B. Razavi, "The current-steering dac [a circuit for all seasons]," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 1, pp. 11–15, 2018. DOI: 10.1109/MSSC.2017.2771102.

[18] P. A. 'T Hart, M. Babaie, E. Charbon, A. Vladimirescu, and F. Sebastiano, "Characterization and modeling of mismatch in cryo-cmos," *IEEE Journal of the Electron Devices Society*, vol. 8, pp. 263–273, 2020. DOI: 10.1109/JEDS.2020.2976546.

[19] G. Bertotti, A. Laifi, E. Di Gioia, *et al.*, "An 8 bit current steering dac for offset compensation purposes in sensor arrays," *Advances in Radio Science*, vol. 10, pp. 201–206, 2012. DOI: 10.5194/ars-10-201-2012. [Online]. Available: https://ars.copernicus.org/articles/10/201/2012/.

A

Simulation Codes

A.0.1. CPHASE.M

```
% Cleanup
clear all;
close all;
% clc;
% system('~/higher_order.bash &');

set(0,'defaultfigurecolor',[1 1 1 ])
set(0, 'DefaultLineLineWidth', 2);
datetime.setDefaultFormats('default','yyyy:MM:dd:hh:mm:ss
    ')
format long

% SIGNAL PARAMETERS
signal_table=[
        "window",     "alpha";
        "square",     1;
         "tukey",     1;
        ];

% time_alpha  = [ 5:0.01:7 ]/50e-3;
% time_alpha           = 0.97:0.001:1.03;
% amp_alpha            = 0.97:0.001:1.03;

time_alpha           = 1;
amp_alpha            = -1;
```

A

```
window                = signal_table(2:end,1);
alpha                 = signal_table(2:end,2);
window_table          = table(window,alpha);
utils.myFigureConfigs();
%DAC PARAMETERS
% dac_type = ["bin_V" "bin_J"  "CF_I" "bin_J_I" "
    unary_thermo_J" "function_weighted_unary_DAC"  "
    output_level_defined_DAC" "RefDAC_&
    _weighted_unary_I_DAC"];
dac_type              = ["bin_V" ];
dac_fs                = [1000]*1e+6;
N_bits                = 16;
coarse_dac_N_bits                 = 6;
dac_phase             = 0;
dac_filter_degree     = 0;
dac_filter_freq       = 800e+6;
% dac_vref              = [40]*1e+06;
dac_vref              = -1;
% dac_vref            = [0.23 0.26];
unit_variance            = [ 0 ];
% unit_variance            = repelem([ 0 logspace(-3,1,25)
    ],100);
% kt_noise            = [0 logspace(-5,-2,6) ];
kt_noise              = 0;

% SPIN QUBIT PARAMETERS
dEz = [ 110 ]*1e+6;
barrier_alpha = (50);
barrier_Joff = 1e+3 * ([5]);

switch 04
    case 0  %see the run time figures
        noise = [ 0 ];
        sweeping = 0;
        QPT = 0;
    case 1  %paralel run
        noise = [ 0 ];
        sweeping = 1;
        QPT = 1;
    case 2  %paralel run noisy
        noise = [ 1 ];
        sweeping = 1;
        QPT = 1;
    case 3  %noise wo qpt
        noise = [ 1 ];
```

```
        sweeping = 1;
        QPT = 0;
    case 4  %single thread closed plots
        noise = [ 0 ];
        sweeping = 0;
        QPT = 1;
end

par = param_manager();
par.add(window_table);
par.add(noise);
par.add(dac_fs);
par.add(N_bits);
par.add(time_alpha);
par.add(amp_alpha);
par.add(QPT);
par.add(dac_vref);
par.add(dac_filter_degree);
par.add(dac_phase);
par.add(dac_filter_freq);
par.add(dEz);
par.add(dac_type);
par.add(kt_noise);
par.add(sweeping);
par.add(coarse_dac_N_bits);

par.add(barrier_Joff);
par.add(barrier_alpha);
par.add(unit_variance);

% barrier_table=table(barrier_alpha',    barrier_Joff');
% par.add(barrier_table,"barrier_alpha,    barrier_Joff")
    ;

L1 = par.Nrows();
% L1 = 16380;

log1 = NaN(1,L1);
log2 = log1;
log3 = log2;
log4 = log2;
log5 = log2;
% log6 = log2;
log6 ={};
```

A

```
[a, host_name]=system('hostname');
disp(['--' host_name(1:end-1)])
% disp(['--Workload: ' num2str(total_sim_time) ])
% disp(['--Estimated Time: ' num2str(0.75/8015.5*
    total_sim_time) ' mins'])
disp(['--Simulation started at: ' datestr(datetime('now')
    ) ])

workID=round((datenum(datetime('now'))*1e+6));

disp(['--WorkID: ' num2str(workID) ])
disp(['--Work size: ' num2str(L1) ])

general_tools = utils();

%%

D = parallel.pool.DataQueue;
afterEach(D, @general_tools.nUpdateWaitbar);

% parpool(30);delete(gcp('nocreate'))

mnct = maxNumCompThreads;
disp(['--maxNumCompThreads: ' num2str(mnct) ])

% strcmp(host_name(1:end-1) , 'qce-cn001.cluster');

if (sweeping && L1>1) || strcmp(host_name(1:end-1) , '
    qce-cn001.cluster')
    sweeping = 1;% if running in server
    p = gcp('nocreate');

    utils.progress(workID,L1)
    if isempty(p)
        p = parpool(mnct,'IdleTimeout',60*16);
    end

    disp(['--Pool size: ' num2str(p.NumWorkers) ])
end

tic
if (sweeping && L1>1) || isequal(host_name(1:end-1) , '
    qce-qn001.cluster') || (noise == 1)
    sweeping = 1;% if running in server
    disp(['PARALLEL RUN'])
```

```
    parfor j = 1:L1
        try
            [ log3(j), log4(j), log5(j), log6{j}] =
               parametric_sweep(par.get(j),sweeping);
        catch
%             warning('!!!DEAD!!! Assigning a value of
    NaN.');
            warning(num2str(L1));
        end

        utils.progress(workID)
    end
else
    disp(['Single run'])
    if(L1 > 10)
        x = input("Simulation number is high. Are you
           sure to continue with single thread?");
    end
    for j = 1:L1
        [log3(j), log4(j), log5(j), log6{j}] =
           parametric_sweep(par.get(j), sweeping);
        disp(['...' num2str(j/L1)]);
    end
end
t_t=toc;
disp(['--Elapsed time is : ' num2str(t_t) ])
disp(['--Simulation finished at: ' datestr(datetime('now'
    )) ])

%% Save the result for sweeping parameters
if(sweeping==1)
        file_name = datestr(datetime('now'),'yyyy-mm-dd-
           hh-MM-ss');
        yourFolder=['logs/'   file_name];
        if ~exist(yourFolder, 'dir')
           mkdir(yourFolder)
           cd(yourFolder)
           save(file_name)
           cd ../..
        else
            disp("!!!!!!! Folder exist !!!!!!")
        end
end

%% Functions
```

A

```
function [log3, log4, log5,log6] = parametric_sweep(param
    ,par3)
    %% SIGNAL PARAMETERS

    dt                  = 50e-12;
    J_seq               = [param.window param.alpha];
    amp_alpha           = param.amp_alpha;
    sweeping            = param.sweeping;

    %% DAC PARAMETERS
    dac_type            = param.dac_type;
    dac_fs              = param.dac_fs;
    N_bits              = param.N_bits;
    dac_phase           = param.dac_phase;
    dac_vref            = param.dac_vref;
    dac_filter_degree   = param.dac_filter_degree;
    dac_filter_freq     = param.dac_filter_freq;
    kt_noise            = param.kt_noise;
    unit_variance       = param.unit_variance;
    %% QUBIT PARAMETERS
    QPT                 = param.QPT;
    noise               = param.noise;
    dEz                 = param.dEz;
    barrier_alpha       = param.barrier_alpha;
    barrier_Joff        = param.barrier_Joff;
    barrier.leverarm    = barrier_alpha;
    barrier.Joffset     = barrier_Joff;

    log3 = NaN;
    log4 = NaN;
    log5 = NaN;
    log6 = NaN;


        % Signal parameters
        J_offset = barrier_Joff;
        time_norm = 5/dEz;
        J_amp_norm = 0.5 / time_norm;

        if (amp_alpha ~= -1 && param.time_alpha == -1)
%             Time  optimization for tukey
            amp_alpha = amp_alpha/(2/(2-str2double(J_seq
                (2))));
            param.time_alpha = 1/amp_alpha;
            J_amp = J_amp_norm * amp_alpha + J_offset;
```

```
        elseif(amp_alpha == -1 && param.time_alpha ~= -1)
%             Amplitude optimization
            J_amp = J_amp_norm / param.time_alpha +
                J_offset;

        elseif(amp_alpha == -1 && param.time_alpha == -1)
%             Time and Amplitude optimization
            amp_alpha = 1;
            param.time_alpha = 1;
            J_amp = J_amp_norm * amp_alpha + J_offset;

        else
            J_amp = J_amp_norm * amp_alpha + J_offset;
        end

        Nsample = round(time_norm/dt*param.time_alpha);

        if(dac_filter_freq==-1)
            f_filt = (dac_fs/2);
        else
            f_filt = dac_filter_freq;
        end

    %% SIGNAL DEFINITION
        function J_signal = create_J(J_amp, Nsample)
            J_pulse = pulse_generator(dt);
           J_pulse.append('square',   J_offset,  2,
              str2double(J_seq(2)));
            J_pulse.append(J_seq(1),   J_amp,  Nsample,
               str2double(J_seq(2)), J_offset);
           J_pulse.append('square',   J_offset,  1,
              str2double(J_seq(2)));
            J_signal = J_pulse.get_signal();
        end

        j2v = @(j)spine.systems.system_spin.j2v_static( (
           j),barrier );
        v2j = @(v)spine.systems.system_spin.v2j_static( (
           v),barrier );

        J_signal = create_J(J_amp, Nsample);


    %% DAC
```

A

```
if(N_bits ~= -1)

    ref_alpha = 1;

    %Create DAC model
    if( dac_type ~= "CF_I" )
        DAC = d2a(dac_fs, N_bits,  f_filt,
            dac_filter_degree, dac_vref*ref_alpha,
             dac_type,0);
    else
        DAC = d2a(dac_fs, [param.
            coarse_dac_N_bits param.N_bits],
            f_filt, dac_filter_degree, dac_vref*
            ref_alpha, "CF_I",unit_variance );
    end

    %Have a symetrical samples
    if(dac_phase == -1)
        per_sample = round((1/dac_fs)/dt);
        if(DAC.type == "bin_V")
            dac_phase = (mod(Nsample/2,per_sample
                )/per_sample)+0.5;
        else
            dac_phase = (mod(Nsample/2,per_sample
                )/per_sample)+0.5;
        end
    end

    %Create cost function for amplitude
       optimization
    opt_fund = @(v) AreaER(v2j(v),dt);

    %Define J -> V -> V_DAC -> J_DAC conversion
    if(amp_alpha == -1)
        J2JDAC = @(J_in, phase) v2j( DAC.
            signal2output( j2v(J_in), dt, phase,
            opt_fund) );
    else
        J2JDAC = @(J_in, phase) v2j( DAC.
            signal2output( j2v(J_in), dt, phase) )
            ;
    end

    %Convert J -> V -> V_DAC -> J_DAC
    J_DAC = J2JDAC(J_signal, dac_phase);
```

```
        %Thermal noise
        if(kt_noise~=0)
            V_DAC = j2v(J_DAC);
            kt_signal = randn(1,length(V_DAC))*
               kt_noise;
            V_DAC = (V_DAC + kt_signal);
            J_DAC = v2j(V_DAC);
        end
    else
        J_DAC=J_signal;
    end


%% RUN SIM

        [F, Px, Py, Pz] = simulation_handler(J_DAC, ~
           sweeping,dt,noise, QPT, dEz,barrier);
        log3 = 1.-F;
        log4 = 1-( sin(  sum( J_DAC )*dt *pi)  )^2 ;
        log5 = Pz(end,1);
        log6 = Px;

%% PLOTS
if(sweeping==0 && QPT==0)


        f=utils.myfig(1);
        hold on
        utils.simple_plot(j2v(J_signal), dt,1e-9)
        utils.simple_plot(j2v(J_DAC), dt,1e-9)
        xlabel("Time (ns)")
        ylabel("V")
        legend(["J"])

        f=utils.myfig(2);
        hold on
        utils.simple_plot(J_signal/1e+6, dt,1e-9)
        utils.simple_plot(J_DAC/1e+6, dt,1e-9)
        xlabel("Time (ns)")
        ylabel("J (MHz)")

        f=utils.myfig(3);
        hold on
        plot(Py(1:1:end,1)+Px(1:1:end,1))
```

A

```
            utils.simple_plot(J_DAC_fft, dt,1e-9)
            xlabel("Time (ns)")
            xlim([-5 55])
            legend(["J"])

            f=utils.myfig(23);
            hold on
            utils.plot_fft(J_signal_fft, dt, 1e+6);
            xlabel("Freq (MHz)")
            ylabel("J (dB)")
            legend(fft_legends)

            f=utils.myfig(3);
            hold on
            utils.plot_fft(J_DAC_fft, dt, 1e+6,dac_tones)
                ;
            xlabel("Freq (MHz)")
            ylabel("J DAC (dB)")
            legend(fft_legends)

            f=utils.myfig(4);
            hold on
            [y,ff]=utils.plot_fft(uniary12, dt, 1e+6,
                unitary_tones);
            xlabel("Freq (MHz)")
            ylabel("uniary12 (dB)")
            legend(fft_legends)

            f=utils.myfig(5);
            hold on
            [y,ff]=utils.plot_fft(uniary1212, dt, 1e+6,
                unitary_tones);
            xlabel("Freq (MHz)")
            ylabel("uniary1212 (dB)")
            legend(fft_legends)

%% STATE PROBABILITY
        t = (1:length(Pz(:,1)))*dt*1e+9;

        utils.myfig(7);
        subplot(3,1,1)
        title("State Probability")
        hold on
        plot(t, (1-Pz(:,1)).*Px(:,2))
        ylabel(append("P_{|1+",char(10217) ,"}"),'
```

```
            fontweight','bold')
        xlim([0 max(t)])
        legend(["1" "2" "3" "4"])

        subplot(3,1,2)
        hold on
        plot(t, (1-Pz(:,1)).*(1-Px(:,2)))
        ylabel( append("P_{|1-",char(10217) ,"}"),'
            fontweight','bold')
        xlim([0 max(t)])

        subplot(3,1,3)
        hold on
        plot(t, Pz(:,1).*Px(:,2) + Pz(:,1).*(1-Px(:,2)))
        ylabel(append("P_{|0+",char(10217),"} + P_{|0-",
            char(10217) ,"}"),'fontweight','bold')
        xlim([0 max(t)])
        xlabel("Time (ns)")



    end
end

function ER = AreaER(J,dt)

    total_area =  sum( J )*dt;
    if total_area > 1
        ER = 1-total_area/0.5;
    elseif total_area <= 0.05
        ER = 0.5;
    else
        ER = cos(  total_area*pi);
    end

end
```

A.0.2. D2A.M

```
classdef d2a < handle


    properties
        % Common parameters have same effects for each
```

A

```
            DAC type
        % Private parameters are special for each DAC
            type

        % Common parameters
        type,fsample, N_bits, f_filt, d_filt, ref;
        %Private parameters
        LSB, units, output_levels;

        %
        last_samples;
        domain;
        unit_var;
    end

    methods

        function obj = d2a(fsample, N_bits, f_filt,
            d_filt, ref, type, unit_var)

            %  =
            if(type~="custom")

                switch type
                    case "bin_V"
%                       obj = d2a.binaryDAC(fsample,
    N_bits,  f_filt, d_filt, ref );
                        obj.domain = "V";
                    obj.fsample=fsample;
                    obj.N_bits=N_bits;
                    obj.f_filt=f_filt;
                    obj.d_filt=d_filt;
                    obj.ref=ref;
                    obj.unit_var=unit_var;

                        obj.LSB=1/(2^(N_bits));
            %             DAC.LSB=1/(2^(N_bits-1));

                        obj.units = 2.^(0:(N_bits-1));

            %             DAC.output_levels = DAC.
                digital2analog(0:2^N_bits);
                        obj.output_levels = (0:2^N_bits)*
                            obj.LSB;
                    case "bin_I"
```

```
                    obj =  d2a.binaryDAC_current(
                        fsample, N_bits,  f_filt,
                        d_filt, ref );
                    obj.domain = "I";
                case "CF_I"
                    obj = d2a.binaryCF_DAC_current(
                        fsample, N_bits,  f_filt,
                        d_filt, ref , unit_var);
                    obj.domain = "I_CF";
%                 case "CF_I"

%                     Ibias = dac_vref;
%                     DAC =  d2a.binaryCF_DAC_current
    (dac_fs, [param.coarse_dac_N_bits param.N_bits],
    f_filt, dac_filter_degree, Ibias );

                case "bin_J"
                    obj =  d2a.binaryDAC(fsample,
                        N_bits,  f_filt, d_filt, ref )
                        ;
                    obj.domain = "V";
                case "bin_J_I"
                    obj =  d2a.binaryDAC_current(
                        fsample, N_bits,  f_filt,
                        d_filt, ref );
                    obj.domain = "I";
                case"unary_thermo_J"
                    obj =  d2a.thermoDAC(fsample,
                        N_bits,  f_filt, d_filt, ref )
                        ;
                    obj.domain = "V";
                case"function_weighted_unary_DAC"
                    func = @(x) -0.5*cos( x*pi ) +
                        0.5;
                    obj =  d2a.
                        function_weighted_unary_DAC(
                        fsample, N_bits,  f_filt,
                        d_filt, ref, func );
                    obj.domain = "V";

                case "RefDAC_&_weighted_unary_I_DAC"
                    func = @(x) -0.5*cos( x*pi ) +
                        0.5;
                    obj =  d2a.
                        RefDAC_weighted_unary_I_DAC(
```

A

```
                    fsample, N_bits,  f_filt,
                    d_filt, ref, func );
                obj.domain = "V";
            end
            obj.type=type;
        else
            obj.type="custom";
            obj.fsample=fsample;
            obj.N_bits=N_bits;
            obj.f_filt=f_filt;
            obj.d_filt=d_filt;
            obj.ref=ref;
            obj.unit_var=unit_var;
        end

    end

    function out = signal2output(obj, signal, dt,
       phase, opt_func)
        if(nargin>4)
            switch obj.domain
                case "V"
                    out = obj.signal2voltage(signal,
                        dt, phase, opt_func );
                case "I"
                    out = obj.signal2current(signal,
                        dt, phase , opt_func);
                case "I_CF"
                    out = obj.signal2current_CF(
                        signal, dt, phase , opt_func);
            end
        else
            switch obj.domain
                case "V"
                    out = obj.signal2voltage(signal,
                        dt, phase );
                case "I"
                    out = obj.signal2current(signal,
                        dt, phase );
                case "I_CF"
                    out = obj.signal2current_CF(
                        signal, dt, phase );
            end

        end
```

```
        end


        function Vout = signal2voltage(obj, signal, dt,
            phase, opt_func)

            N_per = round(1/(obj.fsample*dt));

            [~,holded]  = d2a.sample(signal,N_per, phase)
                ;

            if obj.ref ~= -1
% This is not the best way but I couldn't find how to
    choose
%                 max signal for every case
                dac_max_output = obj.ref;
            else
                dac_max_output = max(holded);
            end

%             [~,a]=size(obj.output_levels);

            Din = sum(holded >= obj.output_levels'.*
                dac_max_output);

            if ( find(Din<=0,1)~=-1 )
                warning("Din is wrong")
            end

            Dout = obj.output_levels( Din );

            if(nargin>4 && obj.ref == -1)
                dac_max_output = obj.ref_opt( Dout,
                    dac_max_output, opt_func);
            end

            Vout = Dout .* dac_max_output;

            Vout = d2a.filter(Vout, dt, obj.f_filt, obj.
                d_filt, N_per);

            obj.last_samples = Dout;
        end
```

A

```
function ref = ref_opt(obj,Dout, initial_ref,
    opt_func,ref_list)
        A = 1;

        %Error rate
        Vout = Dout.* initial_ref;
        ER = opt_func(Vout);
        scale = initial_ref;

        ER_history = zeros(1,101);
        ER_history(1) = ER;
        scale_history = zeros(1,101);
        scale_history(1) = scale;

        for i = 2:101
            scale = scale/(1-ER*A);
            if(nargin>5)
                Vout = Dout .* interp1(ref_list ,
                    ref_list ,scale,'nearest','
                    extrap');
            else
                Vout = Dout .* scale;
            end

            ER = opt_func(Vout);
            ER_history(i) = ER;
            scale_history(i) = scale;

            if abs(ER_history(i-1)) <  abs(ER)
                A = A * 0.5;
            end
        end
        [~,I]=min(abs(ER_history));
        ref = scale_history( I );

        if(obj.d_filt > 0)
            warning('ref opt doesnt work with
                filter')
        end
end

function Vout = signal2current_CF(obj, signal, dt
    , phase, opt_func)
```

```
N_per = round(1/(obj.fsample*dt));

[samples,~] = d2a.sample(signal,N_per, phase)
    ;

slopes = diff(samples) * obj.fsample;

Vout=[];
Dout=[];

obj.ref;

IoverC_ideal = max(abs(slopes));

coarse_output_levels = obj.output_levels{1};
fine_output_levels = obj.output_levels{2};
IoverC_DAC = interp1(coarse_output_levels*obj
    .ref ,coarse_output_levels*obj.ref ,
    IoverC_ideal,'next','extrap');

if(isnan(IoverC_DAC))
    warning('LOW REF!!!!!')
end
for s = 1:(length(samples)-1)
    % Determine the slope for the period
    if(s == 1)
        slope = (samples(s+1)-samples(s))   *
            obj.fsample;
        slope = [   repelem(0,round(phase*
           N_per))   repelem(slope,N_per)  ];
    else
        slope = (samples(s+1)-Vout(end))    *
            obj.fsample;
        slope =

           repelem(slope,N_per);
    end

    Vout_sample = interp1(fine_output_levels*
       IoverC_DAC ,fine_output_levels*
       IoverC_DAC ,slope,'nearest','extrap');
    % Save chosen output levels
    Dout = [Dout Vout_sample/IoverC_DAC];

    if(isempty(Vout))
```

```
                Vout = [Vout +cumsum(Vout_sample)* dt
                    ];
            else
                Vout = [Vout (Vout(end)+cumsum(
                    Vout_sample)* dt)];
            end

        end

        current_opt_func = @(vout) opt_func(cumsum(
            vout)*dt);

        if(nargin>4)
            IoverC_DAC = obj.ref_opt( Dout, IoverC_DAC
                , current_opt_func,
                coarse_output_levels*obj.ref );
        end


        Vout = interp1(fine_output_levels ,
            fine_output_levels ,Dout,'nearest','extrap
            ');
        obj.last_samples = samples;
        Vout = cumsum(Vout*IoverC_DAC*dt);

        Vout(Vout<0) = 0;
        Vout = d2a.filter(Vout, dt, obj.f_filt, obj.
            d_filt, N_per);

    end

    function Vout = signal2current(obj, signal, dt,
        phase, opt_func)

        N_per = round(1/(obj.fsample*dt));

        [samples,~] = d2a.sample(signal,N_per, phase)
            ;

        slopes = diff(samples) * obj.fsample;

        Vout=[];
        Dout=[];

        if obj.ref ~= -1
```

```
    IoverC = obj.ref;
else
    IoverC = max(abs(slopes));
end

for s = 1:(length(samples)-1)
    % Determine the slope for the period
    if(s == 1)
        slope = (samples(s+1)-samples(s))   *
            obj.fsample;
        slope = [   repelem(0,round(phase*
            N_per))   repelem(slope,N_per)  ];
    else
        slope = (samples(s+1)-Vout(end))    *
            obj.fsample;
        slope =

            repelem(slope,N_per);
    end

    Vout_sample = interp1(obj.output_levels*
        IoverC ,obj.output_levels*IoverC ,
        slope,'nearest','extrap');

    % Save chosen output levels
    Dout = [Dout Vout_sample/IoverC];

    if(isempty(Vout))
        Vout = [Vout +cumsum(Vout_sample)* dt
            ];
    else
        Vout = [Vout (Vout(end)+cumsum(
            Vout_sample)* dt)];
    end

end

current_opt_func = @(vout) opt_func(cumsum(
    vout)*dt);

if(nargin>4 && obj.ref == -1)
    IoverC = obj.ref_opt( Dout, max(abs(Dout))
        , current_opt_func);
end
```

A

```
            Vout = interp1(obj.output_levels ,obj.
                output_levels ,Dout,'nearest','extrap');
            obj.last_samples = samples;
            Vout = cumsum(Vout*IoverC*dt);

            Vout(Vout<0) = 0;
            Vout = d2a.filter(Vout, dt, obj.f_filt, obj.
                d_filt, N_per);

        end

    end

    methods(Static)

        function out = settling_curve(signal, dt, N_per,
            tau)

            tx=1: (N_per*10);
            impulse_response=1-exp(-tx*dt/tau);
            integ_filter=diff([0, impulse_response])';

            out = conv(integ_filter,signal);
            out = out( 1 : ( end - length(integ_filter)
                *9/10+2 ) );

        end

        function out = filter(signal, dt, fc, order,
            N_per)

            out = signal;
            for i = 1:order
                out = d2a.settling_curve(out, dt, N_per,
                    1/(fc*2*pi));
            end

        end

        function [sampled,holded] = sample(signal, N_per,
            phase)

            first = signal(1);
            last = signal(end);
            N_skip = round( N_per*phase );
```

```
    signal = [ signal repelem(last, N_per+N_skip)
        ];
    % down sample
    sampled = signal( (1+N_skip) : N_per : end );

    % up sample
    holded = [repelem(first,N_skip) repelem(
       sampled,N_per)];

end


function DAC = binaryDAC_current(fsampleDAC,
   N_bits, f_filt, d_filt,Ibias)

    DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
        Ibias,"custom",0);

    DAC.LSB=1/(2^(N_bits - 1));

    DAC.units = 2.^(0:(N_bits-2));

    DAC.units = [ -flip(DAC.units) 0 DAC.units ];

    DAC.output_levels = ( (0:2^(N_bits)) - 2^(
       N_bits-1) )*DAC.LSB;

end

function DAC = binaryCF_DAC_current(fsampleDAC,
   N_bits, f_filt, d_filt, Ibias, unit_var)

    DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
        Ibias,"custom", unit_var);

    DAC.LSB=1/(2^(sum(N_bits) - 1));

    DAC.units = { 2.^(0:N_bits(1)-1) ;  [-2.^(
       N_bits(2)-2:-1:0)  2.^(0:N_bits(2)-2)] };

    C_units = DAC.units{1};
    F_units = DAC.units{2};

    %Create random mismatch ratios
```

A

```
mismatch_seeds = normrnd(0,unit_var,[length(
    C_units), 1]);
%Bigger units will have smaller mismatch
    ratio
variance_scale = 1./sqrt(C_units);
%Change units
C_units = (1 + mismatch_seeds .*
    variance_scale').* C_units' ;

%Create random mismatch ratios
mismatch_seeds = normrnd(0,unit_var,[length(
    F_units), 1]);
%Bigger units will have smaller mismatch
    ratio
variance_scale = 1./sqrt(abs(F_units));
%Change units
F_units = (1 + mismatch_seeds .*
    variance_scale').* F_units' ;


function output_levels=units2outputlevels(
    units)
     units    = sort(units);
     N_bits = length(units);
     output_levels = zeros(1,2^(N_bits));
     N_level = length(output_levels);
     units_rounded    =    2.^(0:N_level-1);
     for( i =1:length(units) )
         output_levels = output_levels + units
             (i)*repmat([zeros(1,units_rounded(
             i)) ones(1,units_rounded(i))],1,
             N_level/units_rounded(i)/2);
     end
     output_levels = output_levels +
         output_levels(2);
     output_levels = output_levels ./ max(
         output_levels);

end

output_levels1 = units2outputlevels(C_units);
F_units_neg = F_units(1:length(F_units)/2);
F_units_pos = F_units(length(F_units)/2+1:end
    );
```

```
        output_levels2_neg = -flip(units2outputlevels
            (abs(F_units_neg)))';
        output_levels2_pos = units2outputlevels(
            F_units_pos)';
        output_levels2 = [output_levels2_neg; 0;
            output_levels2_pos]';


        DAC.output_levels = {output_levels1,
            output_levels2};

    end

    function DAC = thermoDAC(fsampleDAC, N_bits,
        f_filt, d_filt,dac_vref)

        DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
             dac_vref);

        DAC.LSB= 1/(N_bits);

        DAC.units = ones(1,N_bits);

        DAC.output_levels = [0 cumsum(DAC.units)*DAC.
            LSB];

    end

    function DAC = function_weighted_unary_DAC(
        fsampleDAC, N_bits, f_filt, d_filt, dac_vref,
        func)

        DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
             dac_vref);

        %input output range of func is from 0 to 1
        DAC.units = func((1:N_bits)/N_bits);

        DAC.output_levels = [0 DAC.units];

        DAC.LSB = min(diff(DAC.output_levels));

    end

    function DAC = function_weighted_unary_I_DAC(
```

A

```
            fsampleDAC, N_bits, f_filt, d_filt, dac_vref,
            func)

             DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
                 dac_vref);

             %input output range of func is from 0 to 1
             % DAC.units = func((1:N_bits)/N_bits) - func
                ((1:N_bits)/N_bits);

             DAC.output_levels = [0 DAC.units];
             DAC.LSB = min(diff(DAC.output_levels));

        end

        function DAC = output_level_defined_DAC(
            fsampleDAC, N_bits, f_filt, d_filt, dac_vref,
            output_levels)
             %   output_levels should be integer
             output_levels = output_levels./max(
                output_levels);

             DAC = d2a(fsampleDAC, N_bits, f_filt, d_filt,
                 dac_vref);

             %input output range of func is from 0 to 1
             DAC.units = length(output_levels)-1;

             DAC.output_levels = output_levels;

             DAC.LSB = min(diff(DAC.output_levels));

        end
    end
end
```

A.0.3. SYSTEM_SPIN.M

```
classdef system_spin < handle

        properties

            % Number of dots
            dots;
```

```
        dt;
        Nsim;

        % Hamiltonian Properties
        larmor_frequency;
        rabi_frequency;
        charging_energy;
        singlet_triplet_energy;
        hamiltonian;

        % Hamiltonian Control
        microwave_control;
        detuning_control;
        tunnel_control;

        % Plot handles
        p;

        %Noise parameters
        noise_enable;
        t2_star;
        charge_noise_var;

        %Barrier properties
        barriers;

    end

    properties (Constant)

        % Supported spin states:
        STATE_N = 0                 % 0 electrons
        STATE_0 = 1                 % 1 electron in
           ground state
        STATE_1 = 2                 % 1 electron in
           excited state
        STATE_S = 3                 % 2 electrons in
           singlet configuration (or in 00 spin state
           , depending on the Hamiltonian)
        STATE_T0 = 4                % 2 electrons in
           triplet (0) configuration (or in 01 spin
           state, depending on the Hamiltonian)
        STATE_TP = 5                % 2 electrons in
           triplet (+) configuration (or in 10 spin
           state, depending on the Hamiltonian)
```

A

```
        STATE_TM = 6                % 2 electrons in
            triplet (-) configuration (or in 11 spin
            state, depending on the Hamiltonian)

    end

    methods (Abstract)
        getIndex(obj, states)
        getIndexMeasurement(obj, dot, state)
        getDimension(obj)
    end

    methods

        function obj = system_spin(dots, dt, Nsim)
            obj.dots = dots;
            obj.larmor_frequency = zeros(1, dots);
            obj.rabi_frequency = zeros(1, dots);
            obj.charging_energy = zeros(1, dots);
            obj.singlet_triplet_energy = zeros(1,
                dots);
            obj.microwave_control = zeros(1, dots);
            obj.detuning_control = zeros(1, dots);
            obj.tunnel_control = zeros(dots, dots);
            obj.p = gobjects(1, 2 * dots);
            obj.dt = dt;
            obj.Nsim = Nsim;
            obj.hamiltonian = zeros(obj.getDimension
                (), obj.getDimension(),Nsim);
            obj.t2_star=9.41e-3;
            obj.charge_noise_var=1.668295e-7;
            obj.noise_enable=0;
            obj.barriers.leverarm=12.76;
            obj.barriers.Joffset=60e+3;
        end

        function state = initialize(obj)
            state = zeros(obj.getDimension(), 1);
            states = obj.STATE_0 * ones(1, obj.dots);
            state(obj.getIndex(states)) = 1;
        end

        function reset_hamiltonian(obj)
            obj.hamiltonian = zeros(obj.getDimension
                (), obj.getDimension(),obj.Nsim);
```

```
end

function add_hamiltonian(obj, H, amplitude)
    if(length(amplitude)==1)
        for i=1:obj.Nsim
            obj.hamiltonian(:,:,i)=H.*
                amplitude + obj.hamiltonian
                (:,:,i);
        end
    else
        for i=1:obj.Nsim
            obj.hamiltonian(:,:,i)=H.*
                amplitude(i) + obj.hamiltonian
                (:,:,i);
        end

    end
end

function H = get_hamiltonian(obj, n)

    H = obj.hamiltonian(:,:,n);

end


function [energies] = plotEigenEnergies(obj)

    obj.updateHamiltonian();
    energies = zeros(obj.getDimension, obj.
       Nsim);
    for nsim = 1:obj.Nsim

        %Update the Hamiltonian accordingly
        H=obj.get_hamiltonian(nsim);
        energies(:,nsim)=eig(H);
    end

    plot(energies');
    legend(["00","01","10","11"])
    xlabel('Time');
    ylabel('Energy');

end
```

```
function U = get_unitary( obj )
    U = eye(obj.getDimension);

    for i = 1:obj.Nsim
        H = obj.get_hamiltonian(i);
        U = expm(-1i *H*obj.dt) * U;
    end

end

function [Px, Py, Pz] = measure(obj, state,
   varargin)

    % Measure in rotating frame at time t if
       requested
    if (nargin == 3)
        t = varargin{1};
    end

    % For every dot
    Px = zeros(1, obj.dots);
    Py = zeros(1, obj.dots);
    Pz = zeros(1, obj.dots);
    for dot=1:obj.dots
        if (nargin == 3)
            rot_x = cos(obj.larmor_frequency(
               dot) * t) + 1i * sin(obj.
               larmor_frequency(dot) * t);
            rot_y = cos(obj.larmor_frequency(
               dot) * t + pi/2) + 1i * sin(
               obj.larmor_frequency(dot) * t
               + pi/2);
        else
            rot_x = 1;
            rot_y = 1i;
        end

        % Sum over all indices where this dot
            in the requested state
        Px(dot) = sum(abs((state(obj.
           getIndexMeasurement(dot, obj.
           STATE_0)) + rot_x * state(obj.
           getIndexMeasurement(dot, obj.
           STATE_1)))/sqrt(2)).^2);
        Py(dot) = sum(abs((state(obj.
```

```
                        getIndexMeasurement(dot, obj.
                        STATE_0)) + rot_y * state(obj.
                        getIndexMeasurement(dot, obj.
                        STATE_1)))/sqrt(2)).^2);
                    Pz(dot) = sum(abs(state(obj.
                        getIndexMeasurement(dot, obj.
                        STATE_0))).^2);
                end
            end

            function [Px, Py, Pz] = measure_arr(obj,
                state, varargin)

                % Measure in rotating frame at time t if
                    requested
                if (nargin == 3)
                    t = varargin{1};
                end

                % For every dot
                Px = zeros(length(state), obj.dots);
                Py = zeros(length(state), obj.dots);
                Pz = zeros(length(state), obj.dots);
                for dot=1:obj.dots
                    if (nargin == 3)
                        rot_x = cos(obj.larmor_frequency(
                            dot) * t) + 1i * sin(obj.
                            larmor_frequency(dot) * t);
                        rot_y = cos(obj.larmor_frequency(
                            dot) * t + pi/2) + 1i * sin(
                            obj.larmor_frequency(dot) * t
                            + pi/2);
                    else
                        rot_x = 1;
                        rot_y = 1i;
                    end

                    % Sum over all indices where this dot
                        in the requested state
%                     abs((state(obj.getIndexMeasurement(
    dot, obj.STATE_0),:) + rot_x .* state(obj.
    getIndexMeasurement(dot, obj.STATE_1),:))./sqrt(2));
                    Px(:,dot) = sum(abs((state(obj.
                        getIndexMeasurement(dot, obj.
                        STATE_0),:) + rot_x .* state(obj.
```

A

```
                getIndexMeasurement(dot, obj.
                STATE_1),:))./sqrt(2)).^2,1);
            Py(:,dot) = sum(abs((state(obj.
                getIndexMeasurement(dot, obj.
                STATE_0),:) + rot_y .* state(obj.
                getIndexMeasurement(dot, obj.
                STATE_1),:))./sqrt(2)).^2,1);
            Pz(:,dot) = sum(abs(state(obj.
                getIndexMeasurement(dot, obj.
                STATE_0),:)).^2,1);
        end
    end

    function [Pn, Ps, Pt] = measureST(obj, state)

        % For every dot
        Pn = zeros(1, obj.dots);
        Ps = zeros(1, obj.dots);
        Pt = zeros(1, obj.dots);
        for dot=1:obj.dots

            % Sum over all indices where this dot
                 in the requested state
            Pn(dot) = sum(abs(state(obj.
                getIndexMeasurement(dot, obj.
                STATE_N))).^2);
            Ps(dot) = sum(abs(state(obj.
                getIndexMeasurement(dot, obj.
                STATE_S))).^2);
            Pt(dot) = sum(abs(state(obj.
                getIndexMeasurement(dot, obj.
                STATE_T0))).^2) + ...
                       sum(abs(state(obj.
                           getIndexMeasurement(dot,
                            obj.STATE_TP))).^2) +
                           ...
                       sum(abs(state(obj.
                           getIndexMeasurement(dot,
                            obj.STATE_TM))).^2);
        end
    end

    function plot(obj, state_or_U, t, varargin)

        % Plot style lab
```

```
% 0: No lab frame
% 1: Arrow lab frame
% 2: Trace lab frame
plot_style_lab = 0;
if (nargin == 4)
    plot_style_lab = varargin{1};
end

% Plot singlet-triplet
% 0: No additional plot
% 1: S occupancy
% 2: 1+S+T-N occupancy (expected number
   of electrons)
plot_st = 0;
if (nargin == 5)
    plot_st = varargin{2};
end

% Init, rotate, measure
if (size(state_or_U, 2) == 1)
    state = state_or_U;
else
    state = state_or_U * obj.initialize()
       ;
end
if (plot_style_lab ~= 0)
    [Px, Py, Pz] = obj.measure(state);
end
[Pxr, Pyr, Pzr] = obj.measure(state, t);
if (plot_st ~= 0)
    [Pn, Ps, Pt] = obj.measureST(state);
end

% Plot
ttt=get(obj.p);
if (isempty(fieldnames(ttt)))
    f=figure();
    f.Position = [000 0 1600 700];
    for dot=1:obj.dots
        subplot(floor(sqrt(obj.dots)),
           ceil(sqrt(obj.dots)), dot);
        spine.plotBlochSphere(dot);
        if (plot_style_lab == 1)
            obj.p(2*(dot-1)+1) = plot3
               ([0, 2*Px(dot)-1], [0, 2*
```

A

```
                    Py(dot)-1], [0, 2*Pz(dot)
                    -1]);
                elseif (plot_style_lab == 2)
                    obj.p(2*(dot-1)+1) = plot3(2*
                    Px(dot)-1, 2*Py(dot)-1, 2*
                    Pz(dot)-1);
                else
                    obj.p(2*(dot-1)+1) = plot3(0,
                     0, 0);
                end
                obj.p(2*(dot-1)+2) = plot3(2*Pxr(
                dot)-1, 2*Pyr(dot)-1, 2*Pzr(
                dot)-1);

                obj.p(2*(dot-1)+2).LineWidth = 2;
                obj.p(2*(dot-1)+1).LineWidth = 3;
            end
            if (plot_st ~= 0)
                for dot=1:obj.dots
                    subplot(floor(sqrt(obj.dots))
                    , ceil(sqrt(obj.dots)),
                    dot);
                    if (plot_st == 1)
                     obj.p(2*obj.dots+(dot-1)
                     +1) = plot(t, Ps(dot))
                     ;
                     ylim([0, 1]);
                    elseif (plot_st == 2)
                     obj.p(2*obj.dots+(dot-1)
                     +1) = plot(t, 1-Pn(dot
                     )+Ps(dot)+Pt(dot));
                     ylim([0, 2]);
                    end
                end
            end
        else
            for dot=1:obj.dots
                if (plot_style_lab == 1)
                    obj.p(2*(dot-1)+1).XData =
                    [0, 2*Px(dot)-1];
                    obj.p(2*(dot-1)+1).YData =
                    [0, 2*Py(dot)-1];
                    obj.p(2*(dot-1)+1).ZData =
                    [0, 2*Pz(dot)-1];
                elseif (plot_style_lab == 2)
```

```
        obj.p(2*(dot-1)+1).XData = [
            obj.p(2*(dot-1)+1).XData,
            2*Px(dot)-1];
        obj.p(2*(dot-1)+1).YData = [
            obj.p(2*(dot-1)+1).YData,
            2*Py(dot)-1];
        obj.p(2*(dot-1)+1).ZData = [
            obj.p(2*(dot-1)+1).ZData,
            2*Pz(dot)-1];
    end

    obj.p(2*(dot-1)+2).XData = [obj.p
        (2*(dot-1)+2).XData, 2*Pxr(dot
        )-1];
    obj.p(2*(dot-1)+2).YData = [obj.p
        (2*(dot-1)+2).YData, 2*Pyr(dot
        )-1];
    obj.p(2*(dot-1)+2).ZData = [obj.p
        (2*(dot-1)+2).ZData, 2*Pzr(dot
        )-1];

    obj.p(2*(dot-1)+1).XData = [0, 2*
        Pxr(dot)-1];
    obj.p(2*(dot-1)+1).YData = [0, 2*
        Pyr(dot)-1];
    obj.p(2*(dot-1)+1).ZData = [0, 2*
        Pzr(dot)-1];


    if (plot_st == 1)
        obj.p(2*obj.dots+(dot-1)+1).
            XData = [obj.p(2*obj.dots
            +(dot-1)+1).XData, t];
        obj.p(2*obj.dots+(dot-1)+1).
            YData = [obj.p(2*obj.dots
            +(dot-1)+1).YData, Ps(dot)
            ];
    elseif (plot_st == 2)
        obj.p(2*obj.dots+(dot-1)+1).
            XData = [obj.p(2*obj.dots
            +(dot-1)+1).XData, t];
        obj.p(2*obj.dots+(dot-1)+1).
            YData = [obj.p(2*obj.dots
            +(dot-1)+1).YData, 1-Pn(
            dot)+Ps(dot)+Pt(dot)];
```

A

```
                    end
                end
            end

%           f.Position = [680 0 680 690];
        end

        % Hamiltonian Property Getters/Setters
        % set(value)        : set to all dots (could
            be 1 dot)
        % set(dot, value)   : set to single dot
        % get()             : get value (for 1 dot,
            so dot 1)
        % get(dot)          : get value of single dot
        function setLarmorFrequency(obj, varargin)
            if (nargin == 2)
                obj.larmor_frequency = varargin{1} *
                    ones(1, obj.dots);
            else
                dot = varargin{1};
                obj.larmor_frequency(dot) = varargin
                    {2};
            end
        end

        function setNoise(obj, varargin)
            obj.noise_enable = varargin{1};
        end

        function larmor_frequency =
            getLarmorFrequency(obj, varargin)
            if (nargin == 1)
                larmor_frequency = obj.
                    larmor_frequency(1);
            else
                dot = varargin{1};
                larmor_frequency = obj.
                    larmor_frequency(dot);
            end
        end

        function setRabiFrequency(obj, varargin)
            if (nargin == 2)
                obj.rabi_frequency = varargin{1} *
                    ones(1, obj.dots);
```

```
        else
            dot = varargin{1};
            obj.rabi_frequency(dot) = varargin
                {2};
        end
    end

    function rabi_frequency = getRabiFrequency(
        obj, varargin)
        if (nargin == 1)
            rabi_frequency = obj.rabi_frequency
                (1);
        else
            dot = varargin{1};
            rabi_frequency = obj.rabi_frequency(
                dot);
        end
    end

    function setChargingEnergy(obj, varargin)
        if (nargin == 2)
            obj.charging_energy = varargin{1} *
                ones(1, obj.dots);
        else
            dot = varargin{1};
            obj.charging_energy(dot) = varargin
                {2};
        end
    end

    function charging_energy = getChargingEnergy(
        obj, varargin)
        if (nargin == 1)
            charging_energy = obj.charging_energy
                (1);
        else
            dot = varargin{1};
            charging_energy = obj.charging_energy
                (dot);
        end
    end

    function setSingletTripletEnergy(obj,
        varargin)
        if (nargin == 2)
```

```
            obj.singlet_triplet_energy = varargin
                {1} * ones(1, obj.dots);
        else
            dot = varargin{1};
            obj.singlet_triplet_energy(dot) =
                varargin{2};
        end
    end

    function singlet_triplet_energy =
        getSingletTripletEnergy(obj, varargin)
        if (nargin == 1)
            singlet_triplet_energy = obj.
                singlet_triplet_energy(1);
        else
            dot = varargin{1};
            singlet_triplet_energy = obj.
                singlet_triplet_energy(dot);
        end
    end

    % Hamiltonian Control Setters
    function setMicrowaveControl(obj, varargin)
        if (nargin == 2)
            obj.microwave_control = varargin{1} *
                ones(1, obj.dots);
        else
            dot = varargin{1};
            obj.microwave_control(dot) = varargin
                {2};
        end
    end

    function setDetuningControl(obj, varargin)
        if (nargin == 2)
            obj.detuning_control = varargin{1} *
                ones(1, obj.dots);
        else
            dot = varargin{1};
            obj.detuning_control(dot) = varargin
                {2};
        end
    end

    function setTunnelControl(obj, varargin)
```

```
            obj.tunnel_control = varargin{1};
        end

        function setBarrier(obj, barrier)
            obj.barriers=barrier;
        end

        function [v]=j2v(obj,j)
            v=obj.j2v_static(j,obj.barriers);
        end

        function [j]=v2j(obj,v)
            j=obj.v2j_static(v,obj.barriers);
        end

    end

    methods (Static)

        function [v]=j2v_static(j,barrier)
            v = log(j/barrier.Joffset);
            v = v/barrier.leverarm;
        end

        function [j]=v2j_static(v,barrier)
            v_offset = log( barrier.Joffset );
            j=exp(v * barrier.leverarm + v_offset );
        end

    end

end
```

A.0.4. SIMULATION_HANDLER.M

```
function [F, Px, Py, Pz]=simulation_handler(gate_pulse,
    plot_con,dt,noise, QPT, dEz, barrier)
    %% Simulation setup
    Nsim = length(gate_pulse);
    t = (1:Nsim) .* dt;

    %% Spin-qubit setup
    zeeman_spliting = dEz;
    LarmorFrequency = 1000e+6;
```

A

```
    system = spine.systems.system_2_spin_2_singlet_noise(
        dt,Nsim);
    system.setChargingEnergy(2 * pi * 100e9);
    system.setRabiFrequency(2 * pi * 100e6);
    system.setLarmorFrequency(1, 2 * pi * (
        LarmorFrequency + zeeman_spliting/2));
    system.setLarmorFrequency(2, 2 * pi * (
        LarmorFrequency - zeeman_spliting/2));
    tunneling_control =  spine.systems.system_spin.
        j2v_static(gate_pulse,barrier);
    system.setTunnelControl(tunneling_control)
    system.setNoise(noise);
    system.setBarrier(barrier);

    %% Initial state
    if( QPT )
    state = [   1, 0, 0, 0;
                0, 1, 0, 0;
                0, 0, 1, 0;
                0, 0, 0, 1 ];
    plot_con = 0;
    save_states=0;
    else
    state = [0; 0; 1; 1] ;
%     state = [0; 1; 1; 0] ;
%     state = [1; 1; 0; 0] ;
%     state = [0; 1; 0; 1] ;
    plot_con = 0;
    save_states=1;
    end

    % Normalization
    N_in = length(state(1,:));

    for(i = 1:N_in)
        state(:,i) = state(:,i)/sqrt(sum(abs(state(:,i)))
            );
    end

    %% Prepare post-simulation steps

    %   Rotating frame
    left_rot_undo   =   [ 1, 0; 0, exp( 1i * system.
        getLarmorFrequency(1) * t(end) )];
```

```
right_rot_undo  =   [ 1, 0; 0, exp( 1i * system.
    getLarmorFrequency(2) * t(end) )];
Rot_Frame = kron(left_rot_undo, right_rot_undo);

%   Cphase needs a phase correction
theta_correction = sum(zeeman_spliting - sqrt ( (
    gate_pulse).^2 + zeeman_spliting.^2 ))*dt/0.5/2;

U_calibration = [   1, 0, 0, 0;
                    0, exp(1i*pi*(+theta_correction))
                        , 0, 0;
                    0, 0, exp(1i*pi*(-
                        theta_correction)), 0;
                    0, 0, 0, 1 ];
Uideal = [  1, 0, 0, 0;
            0, 1i, 0, 0;
            0, 0, 1i, 0;
            0, 0, 0, 1];
post_calibration = U_calibration * Rot_Frame;

G = zeros(N_in^2);

if (QPT)
    rho_in = states_to_vec(state);
    G_ideal = kron(Uideal',Uideal);
    d = 4;
end

%%  Run simulation
% If noise is activated repeat the simulation

N_measurement = 1 + (noise)*99;

F = 0;

% Measurements
Px=[];Py=[];Pz=[];

for i = 1:N_measurement

    State_out = spine.simulate(system, @spine.solvers
        .solver_expm, state, plot_con, save_states);

    if save_states
        [x, y, z] = system.measure_arr(State_out,t);
```

A

```
        Last_S = State_out(1:4,end);
        if(isempty(Px))
            Px = [x./N_measurement];
            Py = [y./N_measurement];
            Pz = [z./N_measurement];
        else
            Px = [Px+x./N_measurement];
            Py = [Py+y./N_measurement];
            Pz = [Pz+z./N_measurement];
        end
    else
        for j = 1:length(state(1,:))
            [x, y, z] = system.measure(State_out(:,j)
                ,t(end));
            Px = [Px; x];
            Py = [Py; y];
            Pz = [Pz; z];
        end
        Last_S = State_out;
    end

    atan((2*Py(2)-1)./(2*Px(2)-1))/pi;

    S_final = post_calibration * Last_S;

    if(QPT)
        rho_out = states_to_vec(S_final);
        G_i = rho_out/rho_in;
        G = G+G_i;
        % Assuming rest of the gates will be perfect
        X_best = ( G + G_ideal*(N_measurement-i) ) /
            N_measurement;

        f_best = (trace(G_ideal'*X_best) + d) / (d *
            (d+1));
        % If average lower then 99%, drop the run
        if( f_best < 0.995 && i>1)
            F = NaN;
                    break
        end
    else
        initial_state_observed = Uideal'*S_final;
        G_i = abs(state'*(initial_state_observed*
            initial_state_observed')*state);
        G = G+G_i;
```

```
        end

    end

    G = G/N_measurement;
    if(~isnan(F))
        if(QPT)
            A = G_ideal'*G;
            tr = trace(A);
            F = real( (tr + d) / (d * (d+1)) );
        else
            F = G;
        end
    end

    clear system
end

function rho=states_to_vec(states)

    Nstates = length(states(1,:));
    rho = zeros(Nstates^2);

    for( i=1:Nstates)
        state_i = states(:,i);
        for( j=1:Nstates)
            state_j = states(:,j);

            rho_temp = state_j*state_i';
            rho( :, j + (i-1)*Nstates) = rho_temp(:);

        end
    end

end
```

A.0.5. SYSTEM_2_SPIN_2_SINGLET_NOISE.M

```
classdef system_2_spin_2_singlet_noise < spine.systems.
   system_spin

    methods

        function obj = system_2_spin_2_singlet_noise(dt,
```

A

```
    Nsim)
     obj = obj@spine.systems.system_spin(2, dt,
        Nsim);
end

function index = getIndex(~, states)
    if ((states(1) == spine.systems.system_spin.
        STATE_0) && (states(2) == spine.systems.
        system_spin.STATE_0))
         index = 1;
    elseif ((states(1) == spine.systems.
        system_spin.STATE_0) && (states(2) ==
        spine.systems.system_spin.STATE_1))
         index = 2;
    elseif ((states(1) == spine.systems.
        system_spin.STATE_1) && (states(2) ==
        spine.systems.system_spin.STATE_0))
         index = 3;
    elseif ((states(1) == spine.systems.
        system_spin.STATE_1) && (states(2) ==
        spine.systems.system_spin.STATE_1))
         index = 4;
    elseif ((states(1) == spine.systems.
        system_spin.STATE_N) && (states(2) ==
        spine.systems.system_spin.STATE_S))
         index = 5;
    elseif ((states(1) == spine.systems.
        system_spin.STATE_S) && (states(2) ==
        spine.systems.system_spin.STATE_N))
         index = 6;
    end
end

function index = getIndexMeasurement(~, dot,
    state)
     index = [];
     if (dot == 1)
         if (state == spine.systems.system_spin.
             STATE_0)
              index = [1, 2];
         elseif (state == spine.systems.
             system_spin.STATE_1)
              index = [3, 4];
         elseif (state == spine.systems.
             system_spin.STATE_N)
```

```
                    index = 5;
                elseif (state == spine.systems.
                    system_spin.STATE_S)
                    index = 6;
                end
            elseif (dot == 2)
                if (state == spine.systems.system_spin.
                    STATE_0)
                    index = [1, 3];
                elseif (state == spine.systems.
                    system_spin.STATE_1)
                    index = [2, 4];
                elseif (state == spine.systems.
                    system_spin.STATE_S)
                    index = 5;
                elseif (state == spine.systems.
                    system_spin.STATE_N)
                    index = 6;
                end
            end
        end

        function dimension = getDimension(obj)
            dimension = 4;
        end

        function H = updateHamiltonian(obj)

            obj.reset_hamiltonian();

      %% Microwave control
%             Single microwave driveline
%             microwave_control = obj.microwave_control
    (1) + obj.microwave_control(2);

            % Common Rabi frequency and charging energy
                in simple Hamiltonian
%             rabi_frequency = obj.rabi_frequency(1);

%             H_rot = [   0,
                                    microwave_control*
    rabi_frequency,   microwave_control*rabi_frequency,
    0;
%                         microwave_control*
    rabi_frequency,   0,
```

A

```
    0,                                      microwave_control*
    rabi_frequency;
%                           microwave_control*
    rabi_frequency,   0,
    0,                                      microwave_control*
    rabi_frequency;
%                           0,
                                        microwave_control*
    rabi_frequency, microwave_control*rabi_frequency,
    0;  ];
%
%             obj.add_hamiltonian(H_rot,1);
              %% Resonate rotations
              Ez=(obj.larmor_frequency(1) + obj.
                 larmor_frequency(2)) / 2.0;

              dEz=(obj.larmor_frequency(1) - obj.
                 larmor_frequency(2));
              H_Ez =      [   -1,     0,      0,       0;
                              0,      0,      0,       0;
                              0,      0,      0,       0;
                              0,      0,      0,       1;  ];

              obj.add_hamiltonian(H_Ez,Ez);

              H_dEz=[      0,      0,              0,
                           0;
                           0,      -1/2,           0,
                                      0;
                           0,      0,              1/2,
                                    0;
                           0,      0,              0,
                                          0; ];
              obj.add_hamiltonian(H_dEz,dEz);

              %% J coupling

              H_J =      [      0,      0,              0,
                           0;
                               0,      -1/2,           1/2
                                          0;
                               0,      1/2,            -1/2,
                                        0;
                               0,      0,              0,
                                            0; ] * 2*pi;
```

```
tunnel_control = obj.tunnel_control();
J=obj.v2j(tunnel_control);

if(obj.noise_enable == 1)
    offset = randn(1)*sqrt(obj.
       charge_noise_var);
    pink_noise = pulse_generator.
       generate_pulse("pink_noise", sqrt(obj.
       charge_noise_var), obj.Nsim, offset=
       offset);
    J=obj.v2j(tunnel_control+pink_noise);
end
obj.add_hamiltonian(H_J,J);

if(obj.noise_enable == 1)
    %% + Dephasing

    H_dephasing1 =    [   1,      0,      0,
          0;
                        0,      -1,     0,
                              0;
                        0,      0,      1,
                              0;
                        0,      0,      0,
                              -1;  ]* 1/2 *
                           2*pi;

    H_dephasing2 =    [   1,      0,      0,
          0;
                        0,      1,      0,
                              0;
                        0,      0,      -1,
                              0;
                        0,      0,      0,
                              -1;  ]* 1/2 *
                           2*pi;

    dephasing_amp = normrnd(0, sqrt(2) / (obj
       .t2_star) ,[2,1]);

    obj.add_hamiltonian(H_dephasing1,
       dephasing_amp(1));
    obj.add_hamiltonian(H_dephasing2,
```

A

```
    dephasing_amp(2));

 %% Charge noise on qubits

H_heisenberg =    [   0,      0,      0,
      0;
                      0,      -1,     1,
                            0;
                      0,      1,      -1,
                            0;
                      0,      0,      0,
                            0;  ]*1/2 * 2*
                         pi;

offset = randn(1)*sqrt(obj.
    charge_noise_var);
pink_noise = pulse_generator.
    generate_pulse("pink_noise", sqrt(obj.
    charge_noise_var), obj.Nsim, offset=
    offset);
pink_noise = exp( 2*( pink_noise +
    tunnel_control ) );

obj.add_hamiltonian( H_heisenberg,
    pink_noise);

%% + Leverarm / Frequency shift due to
    voltage on the gate
% Voltage on the gate causes frequency
    shift on qubits resonance frequency
%Creates arund 300KHz phase

lever_arm = 12.08;        %Hz/V
delta_freq_0 = 2.91e+6;     %Hz/V
delta_freq_1 = 2.91e+6;    %Hz/V

lever_arm0=delta_freq_0/lever_arm;
lever_arm1=delta_freq_1/lever_arm;

H_lever_arm =    diag([     +lever_arm0+
    lever_arm1, ...
                            -lever_arm0+
                               lever_arm1
                               , ...
                            +lever_arm0-
```

```
                                                    lever_arm1
                                                    , ...
                                                -lever_arm0-
                                                    lever_arm1
                                                      ] ) *
                                                    1/2 * 2*pi
                                                    ;

            obj.add_hamiltonian(H_lever_arm,
                tunnel_control);

            %% + Leverarm / Charge noise on gate
            %Creates arund 20KHz std phase

            offset = randn(1)*sqrt(obj.
                charge_noise_var);
            pink_noise = pulse_generator.
                generate_pulse("pink_noise", sqrt(obj.
                charge_noise_var), obj.Nsim, offset=
                offset);

            obj.add_hamiltonian(H_lever_arm,
                pink_noise);


        end
        %% Pass hamiltonian

        H = obj.hamiltonian;

    end

    function J=exchange_int(obj,tunneling_control,
        detuning_control,chargingEnergy,
        zeeman_spliting)
        % Efficient controlled-phase gate for single-
            spin qubits in quantum dots
        J=(obj.alpha(zeeman_spliting,detuning_control
            ,chargingEnergy,tunneling_control)+ ...
            obj.alpha(zeeman_spliting,
                detuning_control,chargingEnergy,
                tunneling_control));
    end

    function a=alpha(obj,dEz,epsilon,U,t)
```

```
            a=  t.^2.*(1./(U-epsilon-dEz/2) ...
                    +1./(U+epsilon-dEz./2));
        end


    end

end
```