# Quantum Compilation for Post-NISQ Architectures

Alan Robertson

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computer Science
University of Sydney

2024

**Acknowledgements**

This has been the hardest part of my thesis to write.

In no particular order, I would like to thank the Algorithms Group, for providing both a social tapestry along with technical advice and guidance throughout my PhD.

I would like to thank Yuval, for being the physics grounding in an otherwise abstract bunch of nonsense.

To collect a few more names: Anuj, Adam, Andrew, Lilian, Linsday, Chris, Haowen, Alan. F, Henny, Gertie, Jo, Julia and of course Sparkles Shimmershine, without whom this work would not have been possible.

*This research reported in this thesis was supported by the award of a Research Training Program scholarship to the PhD Candidate.*

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

- Chapter 3 is based on a manuscript that is currently being written. I designed the compiler and implemented it with the assistance of Haowen Gao. Yuval Sanders and I jointly drafted the manuscript that this chapter was based on.
- Chapter 4 is a greatly expanded form of a work published at the ASPLOS Young Architect Workshop 2021. I designed, implement and wrote the drafts for this manuscript. Shuaiwen Song provided guidance and jointly performed editing on the manuscript.
- Chapter 6 is published as 'Mitigating Coupling Map Constrained Correlated Measurement Errors on Quantum Devices' at Supercomputing 2023. I designed the work, Shuaiwen Song performed editing on the manuscript.
- Sections of this thesis have been proof-read and guidance on editing has been provided by Anuj Dhavalikar and Yuval Sanders.

# Contents

# Chapter 1

# Introduction

Advancements in techniques to accurately store and process quantum information have led towards the realisation of small scale prototype quantum devices [114, 1, 7, 113, 132, 122]. It is hoped that these near term, intermediate scale quantum devices (NISQ) may improve to the point that they are able to implement algorithms with asymptotic improvements over their classical counterparts [40, 119, 65]. These algorithms present more efficient methods for tackling problems such as integer factorisation and nano-scale simulation [79, 2].

A common approach to error mitigation is quantum error correction [58]. By introducing redundant ancillary qubits an encoded state is protected against errors by repeated rounds of measurement and correction. While small error correction protocols admit tailored decoding schemes [3, 36], recent advances have focused on scalable quantum error correction, in particular the surface code [49]. The surface code does not support a transversal, universal fault tolerant gate set. Instead it relies on expensive distillation protocols to produce a set of non-Clifford 'magic states'[49, 47].

Current NISQ devices admit the implementation of arbitrary rotation gates up to an architecturally determined precision [114]. The surface code cannot implement these arbitrary angles without either assuming non-planar topologies or introducing potential sources of error, and undermining the protocol [49, 58, 23, 97]. Magic states are required to approximate arbitrary rotation angles by sequences of non-commuting rotations[55, 88, 20, 19].

The computational overhead of programs implemented on the surface code is then commonly benchmarked as a function of the number of magic states as these often dominate the projected runtime costs of quantum algorithms [15, 94, 108]. This often, but not always presumes an arbitrarily large region for magic state distillation [73]. Without such a region, otherwise parallel operations that jointly depend on these magic states as a shared resource must act sequentially, increasing the runtime of the algorithm. Additionally implementing routes to bus data between different elements of a surface code is potentially a blocking operation that is not captured by a pure circuit model. In this thesis we provide a scheme for compiling programs for the surface code by specifying a physical layout over a defined surface code region. The compiler attempts to optimise the allocation of memory elements to provide as much support for parallelism as is possible for a given circuit, while ensuring that data may be bussed between registers correctly. It additionally attempts to allocate additional routing lanes as

auxillary busses to minimise the potential for operations blocking on busses.

In Chapter 2 I provide a brief overview of some aspects of quantum computation. In particular the construction of a set of common gates that underlie a range of quantum algorithms. Chapter 3 contains an overview of the surface code model of quantum computation, before describing the implementation of a compiler for the surface code. Our compiler implements closures as a unit of programmatic composition, and implement language level concurrency control to ensure the correctness of the executed circuit.

Chapter 4 details a schema for the implementation of a Harvard architecture inspired quantum CPU (QCPU). Current NISQ systems and envisaged surface code architectures perform pre-determined schedules of operations controlled by a classical computer. A natural question arises as to what advantages may be gained by implementing a quantum control unit. This is an architecture that describes a set of quantum operations that implement a universal computer. The computer admits the entanglement of the data registers and control registers as a form of constrained control flow. Operations on the quantum system are then expressed as a linear combination of unitary gates [32]. This QCPU design incorporates the reversibility of quantum operations by providing instruction level function invertibility, and bounded automatic uncomputation. This represents is the second proposal for a QCPU, and the first to provide an implementation [82, 135].

While these features seem appealing, they also come with large memory and computational overheads. Combining the compiler from Chapter 3 and this architecture design, we turn our attention to benchmarking the overheads for each cycle of this QCPU. Chapter 5 provides benchmarking of the surface code compiler for a range of common quantum circuits and algorithms. Additionally we benchmark the implementation of the quantum CPU architecture from and demonstrate the overheads required to begin considering developing general purpose quantum-programmable QCPUs on the surface code.

Chapter 6 provides a discussion of mitigating correlated error measurements on current NISQ devices. As surface code quantum computing requires a large number of measurement operations, reducing the error rate of these measurements is an ongoing pursuit in the development of fault tolerant quantum computation.

# Chapter 2

# A Brief Introduction to Quantum Computing

## 2.1 Quantum States and Operations

### 2.1.1 From Classical to Quantum Computing

Classical computers act as state transition functions between binary vectors of length $2^n$. These binary vectors express the state of all memory in the machine, while the state transition function is the action of computation.

$$\mathbb{Z}_2^{\otimes n} \to \mathbb{Z}_2^{\otimes n} \tag{2.1}$$

These state transition functions take the form of operations traditionally implemented within a central processing unit. Each physical component within the computer then acts as a continuous time evolution operator over an electromagnetic field. Here the state of the system may be expressed as a set of scalar quantities, such as the 'voltage' over a particular wire at a particular point in time. The binary elements of vector representing the state of the computer, is then a representation in the adiabatic limit of 'high' and 'low' values of those scalar quantities. By letting the system relax after the application a given time evolution operator the action of the computer over these scalar quantities are resolved as 'bit flips'. The time required for this relaxation then bounds the rate at which computational operations can be performed. By selecting a universal basis gate set of these bit flip operations an arbitrary state transition function may be constructed via the composition of a small number of repeated operations of the basis gates. The execution time of a given classical program may then be measured in the number of operations of the aggregate state transition function of all basis gates. This aggregate state transition function is expressed as a CPU cycle.

To move this from a classical to a quantum domain we swap out the underlying physical basis of our computation from an electromagnetic field to a quantum system. Rather than manipulating the projection of scalar values as binary states, we instead operate directly on a collection of quantum objects, and projectively measure back to binary states. With classical computation these scalar values are projected back to binary states at the end of each operation. When we transpose this to a quantum system the projection is performed as a 'measurement' opera-

tion, and typically terminates a quantum computation. The quantum time evolution operator is described by the Schrödinger equation. The key insight of the Schrödinger equation is that the time evolution operator of a quantum system may be expressed as a linear operator. That is to say that for a state expressed as $\left|\psi\right\rangle$ the evolution of the state is dictated by a linear operation $\mathscr{H}$ termed a Hamiltonian [101].

$$i\hbar\frac{\partial}{\partial t}\left|\psi(t)\right\rangle = \mathscr{H}\left|\psi(t)\right\rangle. \tag{2.2}$$

As this time evolution is expressed as a solution to a partial differential equation we are forced to contend with the idea that if there exists multiple independent solutions to a partial differential equation. As the differential operator is linear, we are left with a linear combination of solutions .

This state is then a linear combination, or superposition of a set of states. To re-apply some physical meaning to this system, the eigenvalues of the system represent physical observables, while the eigenvectors are the states corresponding with that observable. For example if this system was describing an electron between a high and low energy state, the measured energy would be the eigenvalue, while the state itself would be the eigenvector. Measuring a superposition state is then a projective operation that maps to one of the observables in the state, with a probability determined by the scalar coefficient.

The operations over this space can be conceived of as maps of probability amplitudes between sets of observable states. This is then extended by the admission of both negative, and complex valued probabilities. A common computational pattern within quantum systems is to map into a superposition of states, and then compose these negative and imaginary probabilities to amplify or annihilate selected states from the system. The qubit model of quantum computing selects two computational basis states, and assert that all other observables in the system are only reachable by errors. When applied to a physical system this includes higher and lower energy states that may exist, but are excluded from computation. These two-basis-state quantum systems are termed qubits.

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}, \qquad |+\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}.$$

Linearly separable multi-qubit states are constructed using the Kronecker product of the states. Similarly operations as linear maps over independent qubits can be joined using the Kronecker product. Entanglement describes a set of states that are not linearly separable, for example Bell states .

Operations acting over a single qubit are then $2\times 2$ unitary matrices. Linear combinations of single qubit operations are the tensor product of these matrices, and multi-qubit operations are $2^n\times 2^n$ unitary matrices that are not decomposable into a linear combination of single qubit rotations.

As these operations are all unitary, they each also admit an inverse, and by extension must be reversible. Any quantum circuit that consists solely of unitary operations must be reversible. By constructing a universal basis set of these basic operations (also termed 'gates') it is possible to decompose an arbitrary unitary into a basis set. By convention the notion of a quantum

circuit and a quantum program are used interchangeably. By scheduling (or constructing a programme) of these gates the cost of implementing a quantum program may be measured either in the gate count of the circuit (the total number of gates required) or the depth of the circuit (the maximum number of dependent gates along any path in the circuit).

Measurement operations are a special class of non-unitary projective operators that return a classical eigenvalue as an output. With our previous constraint of treating a two level quantum system as having a 'high' and 'low' state these measurement values indicate our $|0\rangle$ and $|1\rangle$ states. As measurement is projective, it discards phase information. For this reason it is often treated as a halting point in NISQ-style quantum computation, or used to project a state into the code-space for error correction style quantum computation [58, 1].

### 2.1.2 Quantum Gates

A simple set of single qubit gates are the rotation gates $R_x, R_y, R_z$. These are commonly presented as parameterised single qubit rotations. [101]. Some non-fault tolerant architectures decompose a $U_3$ gate into a sequence of rotation gates which describes the complete set of operations for single qubit gates.

$$U_3(\theta, \phi, \lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}. \tag{2.3}$$

Some prominent single qubit rotations include:

- $X$, $Y$ and $Z$ gates that act as $\pi$ rotations in their respective bases, the Hadamard gate that maps $X \leftrightarrow Z$ under conjugation.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

- The $S$ gate that maps $X \leftrightarrow Y$ under conjugation. Additionally $S^2 = Z$. Combined with the Hadamard gate the combination of $S$ and $H$ gates allows $Z \leftrightarrow Y$ under conjugation.

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

- $T$ gate, which is the square root of the $S$ gate. This gate features as a common choice of 'non-Clifford' gate for fault tolerant architectures.

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i}{4\pi}} \end{pmatrix}$$

The CNOT gate is a simple two qubit entangling gate that flips a target qubit conditioned on the state of the control qubit. In the computational basis this is equivalent to an XOR operation between the control and target, assigned to the target. As XOR is self inverse this gate is also self inverse. Single qubit rotations and any non-trivial two-qubit gate form a universal basis set [44].

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.4}$$

Extending from a single control $X$ gate we can implement a multi-controled $X$ gate, or a 'Toffoli' gate[101]. A construction of this gate using CNOT, Hadamard and $T$ gates can be seen in Figure 2.1. Of note is that this operation must be performed out of place and that rather than writing the result of the AND operation to the target qubit, instead the target qubit is flipped $|a\rangle|b\rangle|c\rangle \rightarrow |a\rangle|b\rangle|c \oplus (a \cdot b)\rangle$. By ensuring that the target state is in the $|0\rangle$ state prior to the Toffoli gate we can guarantee that the circuit performs an 'AND' gate.

## 2.2 Circuit Model of Quantum Computing

In this section we will detail the construction of some common useful multi-qubit operations as circuits. In particular scalable and reversible AND, OR, CSWAP and C-$U$ gates. The circuits in this section are not the most efficient (though efficiency in this context depends on the basis gate set of the device) but are illustrative. For this section we will assume a basis gate set of single qubit arbitrary rotations, and two qubit CZ and CX gates.

In quantum circuit notation horizontal single lines represent qubits or collections of associated qubits. Labelled boxes represent gates on single or multiple qubits. Vertical lines represent multi-qubit operations, where the symbols on the line indicate the operation: a $\oplus$ indicates the target, and a $\circ$ represents the control qubit.

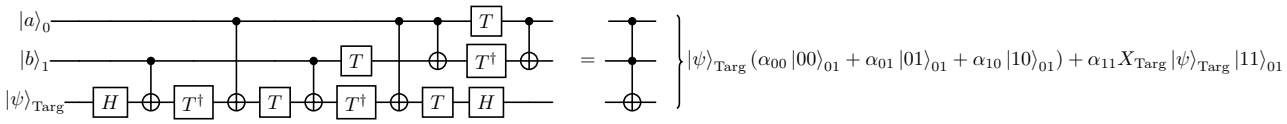



Figure 2.1: Toffoli gate acting as a reversible AND gate. The gate entangles the target qubit with the control qubits such that the target qubit is flipped if both control qubits are in the $|1\rangle$ state.

As with classical computing, the AND operation is associative, and so the operation scales linearly in the number of control bits. A four qubit multi-controlled AND gate can be seen in Figure 2.2. This construction uses $n-2$ ancillae qubits to achieve the shortest circuit depth possible given the basis gate set, however implementations without ancillae also exist.

It is worth noting that while the output of the multi-qubit AND gate has been written after the third Toffoli, this leaves both the ancillae and potentially the data qubits in a modified state. To ensure their re-usability and disentangle these ancillae qubits from the state of the rest of the system we apply gates to uncompute the any unwanted actions by the circuit. This nearly doubles the number of gates present in the circuit.

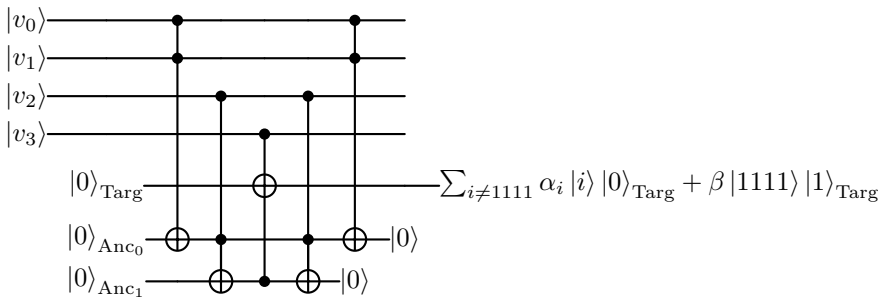



Figure 2.2: Four qubit reversible AND gate using $2n-1$ Toffoli gates and $n-2$ ancillae qubits. The target qubit is entangled with the control qubits such that it is flipped only if all control qubits are in the $|1\rangle$ state.

Similarly it is possible to construct out of place OR gates, as seen in Figure 2.3 and Figure 2.3. As the OR operation is also associative it can be similarly composed to create a multi-qubit OR gate seen in Figure 2.5.

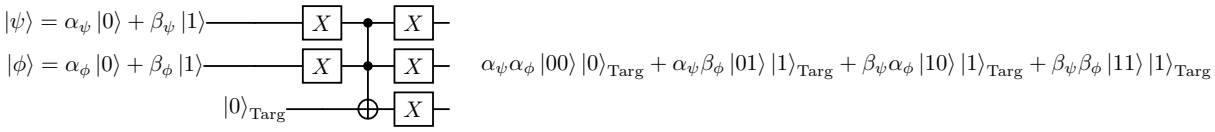



Figure 2.3: Toffoli gate acting as a reversible OR gate. The gate entangles the target qubit with the control qubits such that the target qubit is flipped if either of the control qubits are in the $|1\rangle$ state. As a Toffoli gate may be implemented as a sequence of CNOT, $T$ and Hadamard gates, gate cancellation may be performed to fold at least one of the CNOT operations into the Toffoli, leaving the cost identical to the AND gate.

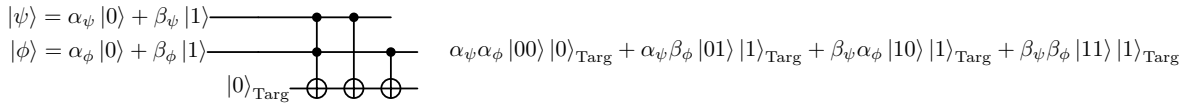



Figure 2.4: Toffoli gate acting as a reversible OR gate. The gate entangles the target qubit with the control qubits such that the target qubit is flipped if either of the control qubits are in the $|1\rangle$ state. As a Toffoli gate may be implemented as a sequence of CNOT, $T$ and Hadamard gates, gate cancellation may be performed to fold at least one of the CNOT operations into the Toffoli, leaving the cost identical to the AND gate.

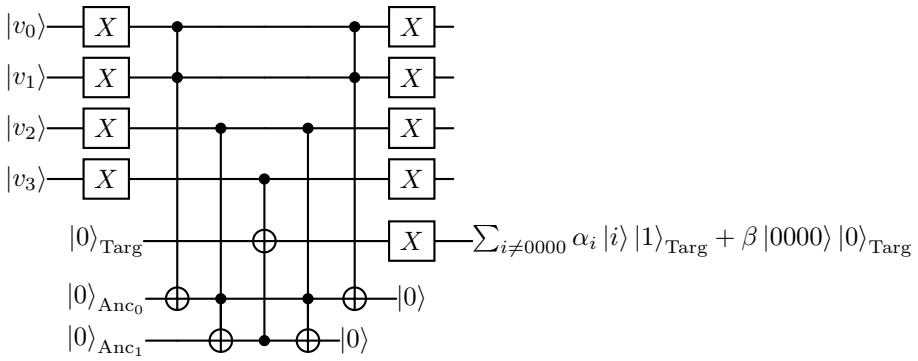



Figure 2.5: Four qubit reversible OR gate using $2n-1$ Toffoli gates and $n-2$ ancillae qubits. The target qubit is engangled with the control qubits such that it is flipped only if any of the control qubits are in the $|1\rangle$ state.

It is also useful to permute the order of qubits in a circuit by swapping their states. Swap operations are denoted with cross marks at the end of a wire and may be constructed using CNOT gates. An example of this construction may be seen in Figure 2.6.

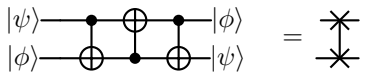



Figure 2.6: SWAP gate flipping the state of two qubits.

By replacing the CNOT gates with Toffoli gates the SWAP then becomes a controlled SWAP, or a CSWAP, as seen in Figure 2.7.

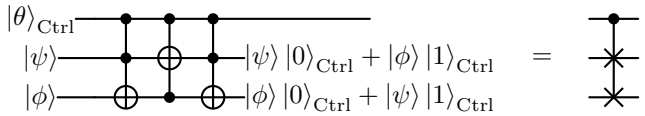



Figure 2.7: Four qubit reversible OR gate using $2n-1$ Toffoli gates and $n-2$ ancillae qubits. The target qubit is entangled with the control qubits such that it is flipped only if any of the control qubits are in the $|1\rangle$ state.

The idea of a controlled swap may be extended to network of controlled swap operations, conditioned on an 'address' register of qubits. This forms the primitives needed to implement a queryable quantum memory.

### 2.2.1 Queryable Quantum Memory

A common form of addressable quantum memory is QRAM [57]. This is a catch-all term for a range of variants of quantum memory models with both classical and quantum access, and quantum or classical data. Typically this operates with a address register the contents of which form the basis of the query, and a target readout register, which the output of the query is

placed in [110]. Unlike a purely classical addressable memory, we must also consider the ability to query the memory in superposition.

Quantum memories are commonly used as proxies for oracles. Given an oracle function $f$ such that $|i\rangle|0\rangle \rightarrow |i\rangle|f(i)\rangle$ the function may be implemented as a lookup over a queryable memory where $i$ is the address. This can be implemented by a CNOT-based memory that XORs the output of the query with a target register

$$\sum_i \alpha_i |i\rangle_{\text{address}} |x\rangle_{\text{readout}} \rightarrow \sum_i \alpha_i |i\rangle_{\text{address}} |x \oplus f(i)\rangle_{\text{readout}}. \tag{2.5}$$

By ensuring that the readout register is in the $|0\rangle$ state before the query then $f(i)$ can be obtained reversibly. The query is also self-inverse.

Conversely a swap-based memory stores and loads data by swapping a target register with its entry in memory

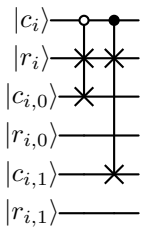
$$\sum_i \alpha_i |i\rangle_{\text{address}} |x\rangle_{\text{readout}} (|m_0\rangle_0 ... |m_i\rangle_i ...) \rightarrow \sum_i \alpha_i |i\rangle_{\text{address}} |m_i\rangle_{\text{readout}} (|m_0\rangle_0 ... |x\rangle_i ...). \tag{2.6}$$

As the overall action of this operation only permutes the order of registers it is reversible, and in this case is also self-inverse. Unlike the query only memory, this swap based memory implies the existence of a memory comprised of registers capable of storing quantum data, and remaining entangled with the state of the address and target readout registers.

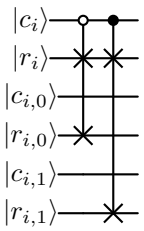
Continuing along these lines, QRAM models can be divided among[77]:

- 'Quantum Random Access' (QRA): the address register may be a quantum state and perform queries in superposition
- 'Classical Random Access' (CRA): the address must be a classical state.
- 'Quantum Memory' (QM): a quantum state is written from the memory to the target register
- 'Classical Memory' (CM): only classical states may be written to the target register.
- 'Read-Only' or 'CNOT' memory: the output is performed via a CNOT, the memory may hence not be written to (unless it is possible to perform in-QRAM operations). The QRAM will always output the same value from the same address.
- 'Read-Write' or 'SWAP' memory: the QRAM lookup swaps the target register into the QRAM in place of the queried register. This is a destructive operation and sequential calls to the QRAM with the same address will alternate between those two values.

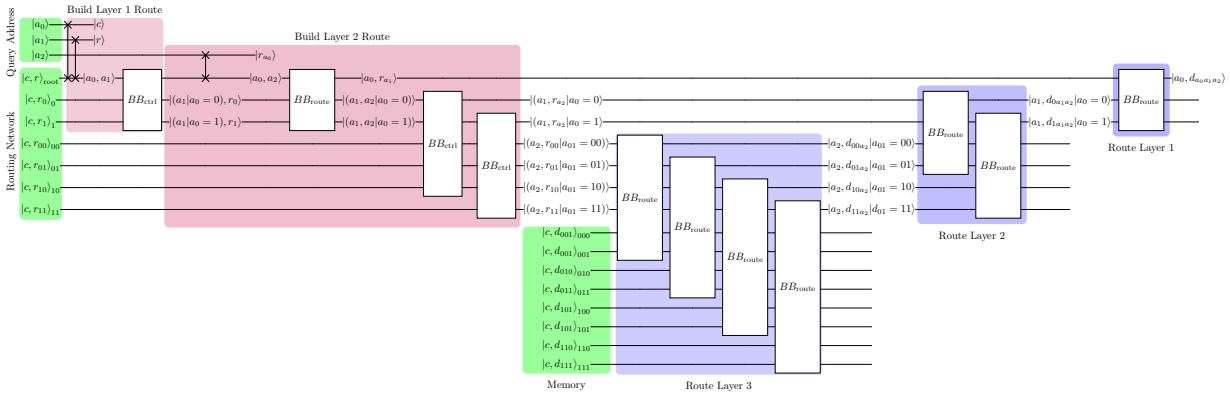
In this thesis when we refer to QRAM we assume a SWAP-QRAQM for storing and accessing quantum data in an addressable fashion. While we don't yet specify an implementation of this QRAQM, we may assume a standard bucket brigade or fanout-and-swap approach[57].




(a) Bucket Brigade QRAM Control routine. This loads the current layer's routing register to the subsequent layer's control register.




(b) Bucket Brigade QRAM Route routine. This swaps the current layer's routing register with a target in the next layer's routing register. The control bit will have been set by a previous control routine.




(c) Bucket Brigade QRAM Readout[57]. Given a query address $a_0...a_i$ a routing network comprising of pairs of control and routing registers is constructed as a binary tree of swap gates conditioned on the address bits. The control bit in each layer is set using the CTRL gadget, and control bits are propagated to lower layers using the Route gadget. The bottom layer of the binary tree contains the memory registers for the QRAM. The lookup is performed using a bottom up sweep of routing gadgets that perform controlled swaps up the binary tree until the addressed element is at the root node. This element may then be accessed via a CNOT to build a read-only CNOT-QRAM, or via a swap to build a read-write SWAP-QRAM. The circuit is then performed in reverse order to reset the registers to their initial values (and propagate the swapped value back to the memory in the case of the SWAP-QRAM).

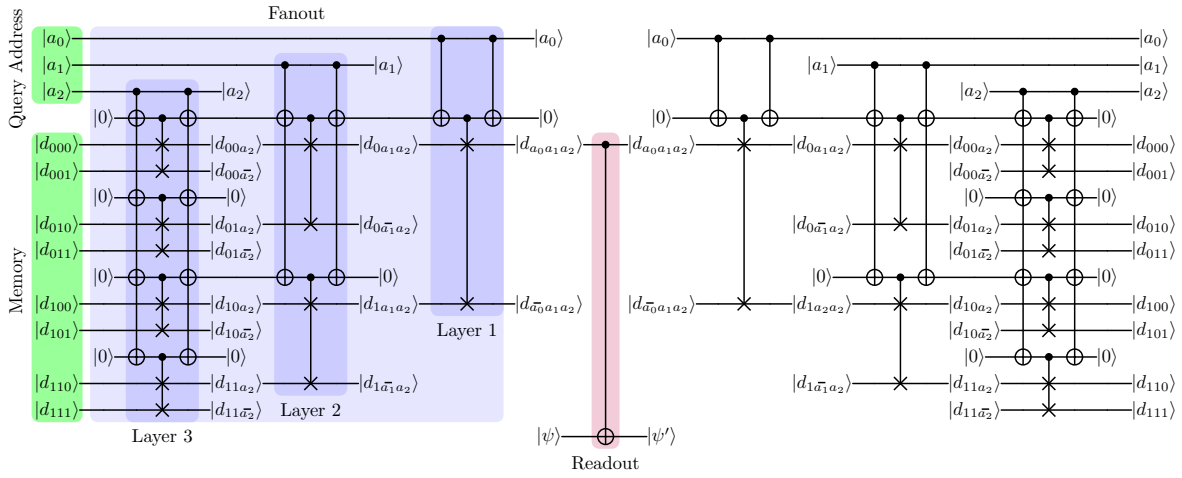



Figure 2.9: Fanout CNOT-QRAQM[77]. This approach implements a binary tree of swaps conditioned on the query address. This approach uses fewer ancillae qubits than the bucket brigade QRAM in Figure 2.8c. The readout operation may be changed to a SWAP to implement a SWAP-QRAQM instead.

When we refer to a QROM, we typically refer to any CNOT-QRACM. These objects are useful for storing fixed lookup tables. For the purposes of this a QROM may be implemented as any CNOT-QRAM where all states in the QROM are prepared in the computational basis states.

Figure 2.9 and Figure 2.8c describe fanout and swap, and bucket brigade QRAM implementations[57][77]. The bucket brigade approach constructs a routing network of ancillae using a set of controlled swaps conditioned on bits in the query address. Once the routing network is constructed the required element in memory is swapped along the routing network to an output register and a readout using either a sequence of CNOT operations or SWAPs is performed.

The fanout-and-swap approach eschews the separate routing network and instead directly performs a sequence of controlled swap operations, with each swap operation directly permuting the order of all elements in the network.

Both of these approaches act as binary search trees, with the fanout and swap performing simultaneous searches starting at the child nodes, and the bucket brigade starting at the root.

One final approach is to implement a QRACM style QROM as a series of direct multi-controlled $X$ gate as seen in Figure 2.10.

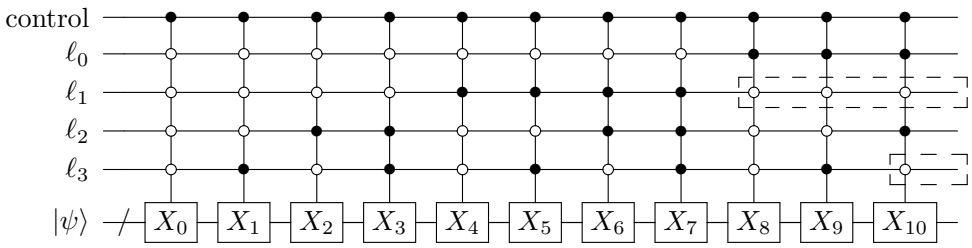



Figure 2.10: An intuitive but suboptimal implementation of controlled QROM as a uniform quantum circuit. The horizontal lines represent quantum registers; the slash on the left-hand side of the bottom register indicates that this register contains several qubits whereas the other registers contain exactly one qubit. Each register is given a label to indicate its role within the circuit, with the top-most being an overall control bit for the entire operation and the $\ell$ registers being used to index the array values $X_{0-10}$, which are encoded in binary as strings of identity (for 0) or NOT (for 1) operations. The vertical lines represent controls for the operations labelled $X_{0-10}$ with a filled dot $\bullet$ indicating activation on a value of 1 and an empty dot $\circ$ indicating activation on a value of 0. Thus $X_k$ is activated if and only if the control is 1 and $k = 2^3\ell_0 + 2^2\ell_1 + 2^1\ell_2 + 2^0\ell_3$. The dashed lines indicate controls that can be eliminated without altering the behaviour of the circuit given acceptable input [9].

### 2.2.2 Conditional Operations

Multi-controlled operations on quantum devices are generally expensive in most models of quantum computation. As these form the primitives for branching control we will be performing a non-trivial number of conditional operations.

While we can perform controlled $X$ operations, it is also useful to be able to perform controlled arbitrary rotations. This is a common gadget in a number of quantum algorithms such as the quantum Fourier Transform (QFT). Using Clifford gates to perform basis changes, a rotation gate in any basis can be mapped to any other basis. Figure 2.11 demonstrates an implementation of an arbitrary $Z(\theta)$ rotation using rotations of $\frac{\theta}{2}$.

Under most NISQ hardware cost models $Z$ gates are implemented in software and are hence considered 'free'[1]. Conversely in surface code cost models arbitrary angle rotations are expensive, and more generally the smaller the rotation angle the more expensive the operation [128].

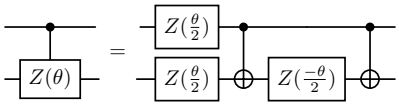



Figure 2.11: Controlled $Z(\theta)$ circuit. The pattern of constructing a gate $U$ can be constructed using a combination of $U^{\frac{1}{2}}$ and $U^{\frac{1}{2}\dagger}$ will be a recurring one.

While the efficiency of the construction of a $U_3$ gate, and by extension a $CU_3$ gate depends on the gateset, we can leverage our $CZ(\theta)$ gate and the Hadamard gate to demonstrate an implementation of a $CU_3$ gate. Using the circuit in Figure 2.11 this controlled unitary invokes a cost of six CNOT gates, and nine arbitrary $Z$ rotations.

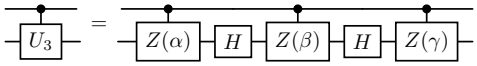



Figure 2.12: Controlled $U_3$ circuit. If the controlled $Z$ rotations are implemented using Figure 2.11 then this circuit requires six CNOT operations, plus the overhead of the $Z$ rotations. In NISQ computational models the $Z$ rotations are typically performed in software and are considered 'free', and two-qubit operations are considered expensive. In surface code computational models logical CNOT operations are performed in a small fault tolerant constant time operation, while arbitrary rotations require expensive magic state distillation protocols.

These gates may be extended to multi-controlled operations by replacing the control with the output of the multi-controlled CNOT based AND or OR gates in Figure 2.2 and Figure 2.5. An example of a multi-controlled $Z$ rotation may be seen in Figure 2.13.

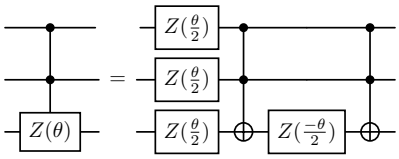



Figure 2.13: Multi-Controlled $Z(\theta)$ circuit.

# Chapter 3

# Surface Code Compilation

## 3.1 Introduction

There is a great need for the thorough analysis of quantum computer algorithms. Whereas researchers have claimed that quantum computers can reduce the complexity of important computational problems such as integer factoring [121] and ground state energy estimation for physically reasonable Hamiltonians [79, 2]. The description of these quantum algorithms often rely on assumptions about computational cost models thatmay not entirely align with architectural cost models [47].

Presently, quantum resource estimates are based on estimates of the number of uses of a magic state factory[27, 104], required to execute a given quantum circuit and then translate that number into estimates of the number of of physical (uncorrected) qubits and the number of error-correction cycles needed to perform the required number of magic state factory operations [52].

We seek to replace such methodologies with something more reliable. Specifically, we want to devise a rigorously-defined compilation process that would translate a given quantum circuit into a hardware-level executable, or to a form akin to a hardware executable.

If we assume, alongside much of the quantum computing community, that the eventual quantum computer would execute quantum computations using the surface code [49], we envisage that hardware executables would comprise a sequence of lattice surgery operations [87].

The problem of surface code allocation and routing draws strong parallels to a number of disciplines in classical computing. Each element of the surface code may be considered a resource, while gates require some subset of explicitly defined memory (i.e. registers) and computational resources. Classically computation resources are managed by the operating system, where memory resources are distributed by an allocator, and computational execution time is managed by the scheduler. Sharing memory between computational processes requires the judicious application of synchronisation primitives. Unlike classical parallelism, our computational resources cannot be loaded and unloaded by a scheduler without destroying entanglement information, or having a second copy of the computational unit. Each cycle of this device may then be described by what operation has ownership over each of the elements of

the surface code, in effect reducing the description of this problem to a series of mutual exclusions. For example a CNOT operation requires a control and a target register, each of which are patches on the surface code, in addition to a collection of intermediary 'routing' ancillae patches. Unlike classical operations, these routing ancillae imply that operations must also satisfy topological constraints such that a joint operation over multiple resources is connected. If two operations would share a resource then either the circuit describes the order in which that resource is to be accessed (as not all operations are commutative), or if the order of access would not change the operation of the process (for example accessing an ancillae) then the order of access must be determined by the compiler.

Continuing this analogy the order in which operations are implemented and routing ancillae are allocated to them mimics the operation of a scheduler. Unlike classical computation, as we are fundamentally acting over a circuit model of computation, in which all operational dependencies are acyclic our computational model is provably free of deadlocks. Instead the ordering of these gates and their associated mutual dependencies have implications for the runtime of the program.

In essence a qubit mapping algorithm is a basis for an allocator. By ensuring that the mapping is stateful it becomes possible to map, unmap and re-map qubits as they are initialised, and removed from computations. The core concern of such an allocator is then assessing and mitigating the associated computational overheads while maintaining the correctness of the underlying surface code computation. Given an understanding of the computational overheads associated with qubit allocation and routing we can then factor these constraints into the resource estimation of a quantum computation.

Previous approaches to qubit mapping and allocation on the surface code have focused on either static grids[74], checkerboards[83], and more recently a pattern of static lanes and registers[126]. Some have deferred routing and allocation to implementation in generic computation regions [70] These approaches do not deal with magic state factory placement, scoping the lifetimes of registers within a circuit or benchmarking the tradeoff between the avaliable area on the surface code and its runtime. Conversely when benchmarking the implementation of algorithms on the surface code these concerns are currently addressed by the hand placement of elements[53, 128, 56].

In this chapter we devise a compilation sequence that translates quantum circuits and surface code architectures (defined as a rectangular region of encoded qubits) into a sequence of surface code operations for each time step of the execution of the circuit. The compiler then additionally provides a number of surface code cycles required to implement the circuit.

This chapter is structured as follows.

- In section 3.2, we carefully formulate the computational problem our algorithm seeks to solve. Our primary concern is the definition of logical computational units. As part of this formulation we introduce a distinction between local registers, local ancillae, and externally defined patches in the surface code. Local registers store information that persists through the current computational scope. Local ancillae store information pertinent to a single surface code operation, such as providing support for routing and rotation gates. These ancillae are then freed. Externally defined patches are collections of surface code patches that are allocated, executed and may then be freed and reallocated. The

allocation and interaction with externally defined patches provides a basis for the composition of our underlying computational units. We implement magic state distillation and other locally scoped operations as external patches, and route over the allocation of these patches.

- In section 3.4, we describe our qubit mapping and routing algorithms. Our algorithm proceeds by ordering parallel qubit operations and identifies bottlenecks where otherwise independent operations have an implicit dependency on a shared resource. For example, a magic state factory, routing ancillae or other externally defined computational unit. These constraints prohibit the parallel execution of these components of the program. We then attempt to provide a static mapping between qubits and external computational units within the program and physical locations on the QCB. Finally we use a routing algorithm based on A* to implement non-local surface code operations.

## 3.2 Execution Model

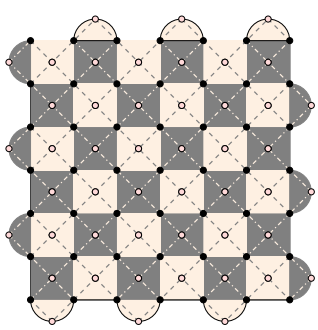
In this section, we provide a description of the lattice surgery execution model that serves as our compiler target. We present this description in four parts.

1. In section 3.2.1, we give a brief description of the surface code, a topological quantum-error-correcting code that has long been investigated by Google and IBM as the most promising avenue to implement fault-tolerant quantum computation.
2. In section 3.2.2, we describe rotations on surface code patches.
3. In section 3.2.3, we provide an over-view of lattice surgery based Clifford operations. We specifically explain how to execute the controlled-not (CNOT), Hadamard, and Phase gates, and we presume that other Clifford gates are executed by composing an appropriate list of these three gates. In the case of the CNOT gate, which is a two-qubit gate, we point out the need for a 'routing lane' to exist between the two qubits.
4. In section 3.2.4, we describe the concept of a magic state factory as the standard means by which to execute non-Clifford operations.

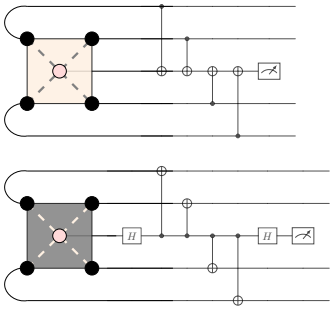
Our description of the surface code execution model indicates the need for two key concepts: routing lanes and externs. Both concepts imply certain constraints on the compilation process that we manage by introducing a data structure we call a 'quantum circuit board'. The quantum circuit board is an abstraction atop the surface code that is designed to facilitate the development of a core component of our compilation algorithm, a heuristic for what we call the 'allocation' problem.

### 3.2.1 The Surface Code

The problem of errors on quantum systems has been widely appreciated since the beginning of research into quantum computing [39]. Many early papers have considered the use of quantum error-correcting codes as an integral component to a strategy for managing logical errors in quantum computations that arise from hardware noise[58]. The surface code is a particular choice of quantum error-correcting code that envisions the quantum system as being composed of a square lattice of noisy qubits in which it is only possible to execute two-qubit gates that



(a) An example surface code patch.



(b) The $X$ (top) and $Z$ (bottom) stabiliser circuits.

Figure 3.1: An illustration of the (rotated) surface code. The dark data qubits are arranged in a lattice. Light regions indicate $X$ stabiliser operations, while dark regions indicate $Z$ stabiliser operations. Each stabiliser region is centered on an ancillae qubit. fig. 3.1b give detail on the correspondence between the stabiliser regions and the associated circuit.

affect pairs of qubits that are connected by an edge[74].

The square lattice of noisy qubits is to be viewed at a 45° angle as depicted in fig. 3.1, where the qubits are illustrated with alternating dark- and light-coloured dots to distinguish their separate roles in the surface code ('data' and 'ancillae', respectively) [49, 87]. In its idling state, the surface code is a short-depth quantum circuit that performs a set of joint measurements over all physical data qubits. These joint measurements are performed by a set of controlled rotations targeting neighbouring ancillae qubits [74]. The ancillae qubits are then measured in the computation basis. Only the data qubits store quantum information relevant to the current computation between clock cycles. This large quantum circuit is to be understood as a tensor product of 'stabiliser' measurements, in reference to the fact that the surface code is an example of a stabiliser code [59]. The output of these measurements are then passed to a classical decoder, whose task is to determine what error (if any) has occurred, and then to perform the necessary correction operations [71]. Assuming errors are sufficiently sparse and infrequent [49, 48], the effect of the detected error is mathematically equivalent to an inappropriate quantum gate having been applied to some physical qubits.

As the stabiliser operations commute with the state encoded by these stabiliser measurements, mapping an error to a product of stabilisers 'corrects' the error [49]. The stabilisers themselves are plaquettes on the surface code, any closed loop over the face of a surface code patch is a product of stabilisers. Hence any string of errors that starts and ends at the same positions will act to correct that initial error. This error correction procedure is equivalent to minimum weight perfect matching, and performing this decoding and correction procedure is a necessary component of the operation of the surface code [71]. Each round of error correction on the surface code is termed a *surface code cycle*[87, 52].

A string of errors that commutes with all stabilisers in the surface code without forming a closed loop would stretch between two boundaries of the same stabilisers. This string acts as a logical operator and hence as a logical error on the encoded state. The minimum number of physical errors required to implement a logical error (without any further inspection of the

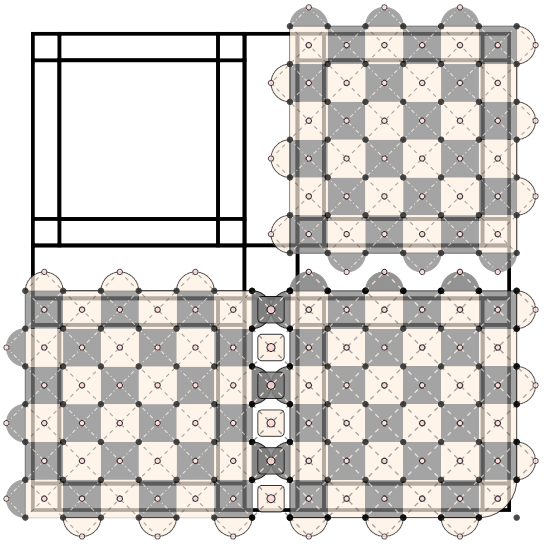


Figure 3.2: An illustration of the five different types of independently programmable regions. The illustration depicts three small surface code patches arranged in a $2 \times 2$ grid, with varying degrees of detail about the possible interactions with one another.

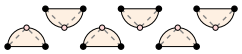
decoding operation) scales with the distance between topologically distinct boundaries of the same stabiliser. For a regularly oriented surface code patch this is a function of the height and width of the patch. As a result the distance of the code (or the minimum number of physical errors before a logical error occurs) is typically regarded as a function of the size of a surface code patch.

Each surface code patch (fig. 3.1a) is capable of stabilising a single logical qubit. This stabilisation involves a classical co-processor sequencing stabiliser measurements for the bulk and boundary of the surface code patch, and implementing a decoder over the measurement outcomes for that region. In order to perform multi-qubit operations using lattice surgery, we expect classical co-processors to synchronously control multiple adjacent surface code regions. This splits our programmable regions between the bulk and boundaries of each patch. This requires that our decoder tracks measurement outcomes between connected patches.

Further inspection shows that for generic lattice surgery operations that five dependently tunable regions of stabilisers may be identified. Programmable surface code operations may then be considered in terms of applying pre-determined control sequences to each of these regions to implement the appropriate quantum gates, and then passing the stabiliser measurement outcomes back to the decoder. These regions are illustrated in fig. 3.2.

Each surface code operation may be described as a combination of operations over each region. Surface code operations are only considered valid if the resulting combination of stabilisers commutes, that the boundaries of the stabilised region support the encoding of a sufficient number of logical qubits, and if the distance of the smallest error string between two topologically distinct boundaries of the same time is greater than or equal to the length of the patch. These constraints are necessary to ensure that a given collection of stabiliser operations constitutes a valid error correcting code, and that it does not decrease the distance of the surface

(a) Stabilisation of light-light border.

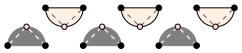


(b) Stabilisation of light-light border with offset.

(c) Stabilisation of dark-dark border.

(d) Stabilisation of dark-dark border with offset.

(e) Stabilisation of mixed border.



(f) Stabilisation of mixed border with offset.

Figure 3.3: Border stabilisation operations. These operations stabilise the borders of two adjacent patches without entangling the associated logical qubits. If the ancillae qubit is used by both patches then the ancillae will need to be reset between measurements. This stands in contrast to boundary operations where the ancillae is used to project two independent patches to perform a computational operation.

code, and undermine the basis of the assumptions of the error tolerance of the device.

In summary, the surface code quantum computer consists of a large square lattice of raw qubits to which a largely repeating sequence of short-depth quantum circuits are applied each clock cycle. This square lattice is partitioned into many equal-sized patches that are used to process individual logical qubits.

**Region Stabilisation**

Our first concern with the surface code is preparing and protecting a state against logical errors. A $|0\rangle$ or a $|+\rangle$ state may be prepared by setting all data qubits within a patch to the $|0\rangle$ state and then applying the stabiliser measurements. Each round of stabiliser measurements projects errors to local $X$ and $Z$ operations. Each ancillae then detects odd parity flips caused by errors on data qubits. Errors become logical errors when they form a 'string' between two boundaries of the surface code of the same stabiliser type, promoting themselves to a logical operation and commuting with all stabiliser measurements. Errors that do not commute with all the stabilisers are then detected and diagnosed by minimum weight perfect matching using the ancillae qubits as sentinels for the errors[71].

Each round of stabiliser measurements is termed a cycle. To correct any errors introduced by a surface code operation typically either $O(1)$ or $O(n)$ cycles are required. It is a convention to measure surface code operations in factors of $n$ cycles[87]. We adopt this convention and hence $\circlearrowright_1$ will imply $n$ rounds of stabiliser measurements and corrections.

This stabilisation acts as a set of unary operations over a single patch, including borders, edges, corners and bulk. We have three such operations: no-activation, and two possible stabiliser alignments, and a specialised sequence of stabiliser twists that implement an $S$ gate [51]. This sequence of stabiliser twists takes $\frac{n}{2}$ surface code cycles, and so we maintain an upper bound

of $n$ cycles for this operation. The instructions passed to the borders of the patch will jointly depend on neighbouring patches[87].

### Transversal Operations

Having stabilised a $|0\rangle$ state, we can prepare a $|1\rangle$ state transverally by applying a set of X operators along a 'string' between two boundaries of $|Z\rangle$ stabilisers. Similarly a logical $Z$ operation may be applied transversally by appling $Z$ operations along a string between two $X$ stabilised boundaries, [49, 48].

Another transversal operation is the Hadamard gate, which can be performed by flipping our $X$ and $Z$ stabilisers. This leaves the surface code patch in a rotated and flipped state[49, 48].

Lastly some operations that are not fault tolerant may be applied transversally. These operations may introduce logical errors and are typically subject to state distillation protocols to reduce this introduced error rate [56].

### Merging Patches

While we have some initial transversal operations, these do not constitute a universal basis set. Another architectural constraint that appears at this point is the qubit connectivity, with denser connectivities reducing the number of gates required per cycle. For this work we will assume a sparse grid connectivity where data qubits are only connected to local ancillae and vice versa. Some other architectural models assume a degree of limited non-local connectivity [48], connectivity that equivalent to a local 'folding' of the surface code [97], stacking of layers of surface codes [126] or other teleport supported grid arrangements [105, 41].

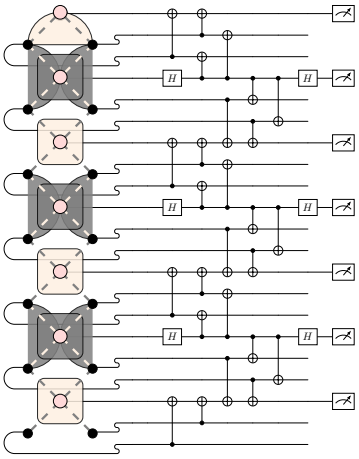
Transversal CNOT gates are also possible, but this implies a lattice connectivity that admits non-local physical CNOT operations between qubits [97, 78, 126]. Given this architectural constraint it is often assumed that CNOT operations cannot be performed without the use of ancillae qubits [49].

Instead multi-qubit operations are realised by 'merging' and 'splitting' adjacent patches [49, 109].

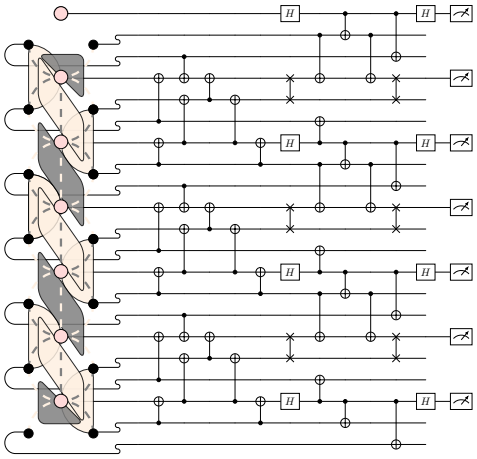
Lattice merging is a basic topological operation on the surface code. Two patches are merged by measuring the stabilisers along their boundary. As logical operations act as 'strings' spanning between boundaries, the joint measurement of the stabilisers at that boundary then acts as a joint measurement of the logical state [74]. A 'smooth' merge acts as a joint $ZZ$ measurement, while a rough merge acts as an $XX$ measurement. An example circuit for this operation can be seen in Figure 3.4, while a small beastiary of such operations can be seen in Figure 3.5.

While still acting as joint measurement operations, twisted merge operations provide support for offset stabilisers [89].

The merge operation is performed by preparing the ancillae qubits along the boundary in either the $|0\rangle$ or $|+\rangle$ state and then performing stabiliser measurements between the adjacent patches. After the correction round it is possible to determine the parity of the stabiliser measurements across the old boundary. This becomes a two logical qubit stabiliser measurement, either $X_L X_L$, $Z_L Z_L$ or some combination of X and Z measurements depending on the merged boundary.

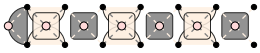


(a) Circuit for standard smooth merge.

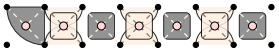


(b) Circuit for a twisted merge.

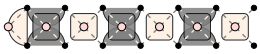
Figure 3.4: Example circuits for standard and twisted merges. The smooth merge acts as a $ZZ$ measurement between two adjacent surface code patches, while this particular twisted merge operation acts as an $XX$ measurement.



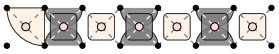
(a) Rough merge.



(b) Rough merge with corner.

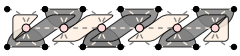


(c) Smooth merge.



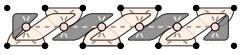
(d) Smooth merge with corner.

Figure 3.5: Aligned merge operations. These act as a joint $XX$ or $ZZ$ measurement over a pair of adjacent surface code patches. These merge operations cannot be used if the stabiliser boundaries of the patches are not aligned.



(a) Dark twisted merge.

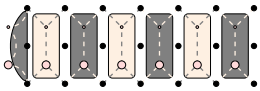
(b) Dark twisted merge with offset.



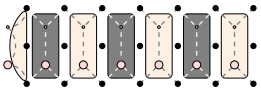
(c) Light twisted merge.

(d) Light twisted merge with offset.

Figure 3.6: Twisted merge operations. These operations act identically to the aligned merge operations at the logical level, but require either specialised architectural support beyond local two-qubit operations, or have a greater circuit depth as seen in Figure 3.4.



(a) Stretched rough merge.



(b) Stretched smooth merge.

Figure 3.7: Stretched merge operations. These operations are analogous to the twisted operations in that they require either additional local SWAP operations with increased circuit depth, or dedicated architecturally supported two qubit operations.

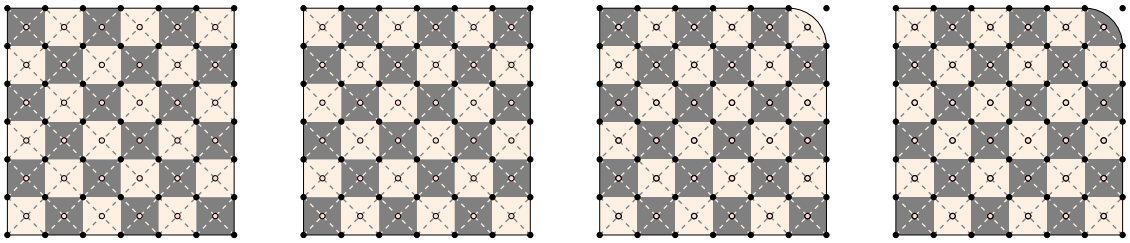


Figure 3.8: Joint Bulk, Margin, and Corner Operations. These operations act to stabilise a patch and act as constraints on the choice of legal boundary or merge operations.

Without twisted merge operations we must consider the local alignment of the surface code patches. Without twisted operations merges are only possible between aligned edges while some topological operations may break this alignment. For example a rotation with one ancillae rotates the orientation of the stabiliser, and also breaks alignment, while a rotation with two ancillae in an L shape can perform this rotation without breaking alignment.

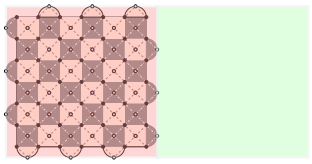
### 3.2.2 Rotations

While having no immediate computational effect the patch may be rotated to expose a different set of stabilisers along the boundary. This rotation is necessary for exposing a boundary that may be necessary for a given multi-qubit operation [89].

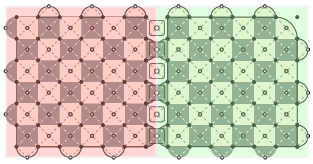
These rotations are performed by a sequence of merge operations that smoothly deforms the boundary without reducing the distance of the code. By combining a sequence of rotations it is possible to rotate to any boundary alignment in constant time. Our first rotation (Figure 3.9) is performed by merging two patches, then rotating the boundary stabilisers along the joint edge of both patches.

A rotation flips the interacting X and Z edges of a single register while leaving the overall state unchanged. This differs from a Hadamard gate that maps logical $X_L \rightarrow Z_L$ and $Y_L \rightarrow -Y_L$.

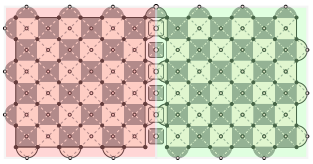
This rotation leaves the boundary in an 'unaligned' state compared to the default orientation of surface code patches [48], and so a secondary rotation is required to return it to the correct configuration. This secondary rotation only alters the alignment of the boundary stabilisers rather than flipping them, and requires a deformation of the grid architecture of the surface



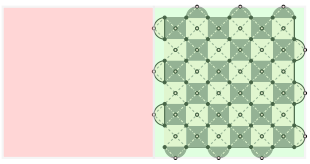
(a) $\circlearrowleft_0$ Initial State



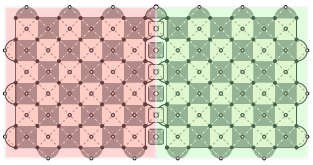
(b) $\circlearrowleft_1$ Merge right into cut corner.



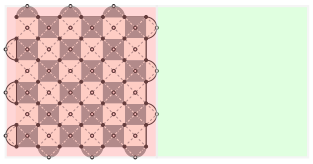
(c) $\circlearrowleft_2$ Rotate Boundary



(d) $\circlearrowleft_2$ Measure out left patch



(e) $\circlearrowleft_3$ Merge with original patch.



(f) $\circlearrowleft_3$ Measure out right patch.

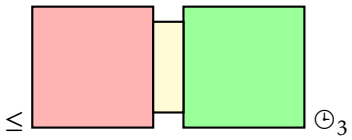
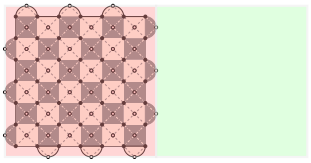
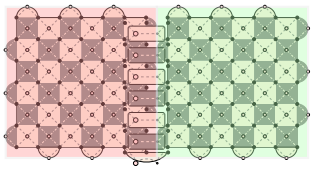


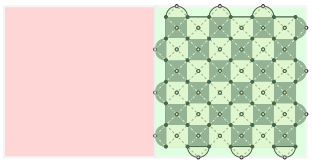

Figure 3.9: Rotating boundary stabilisers on a surface code patch. Note that this leaves the boundaries of the stabilisers in an unaligned state, and may require twisted join operations.



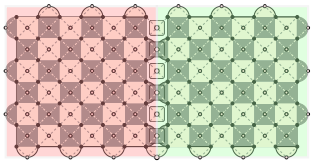
(a) $\circledcirc_0$ Initial State



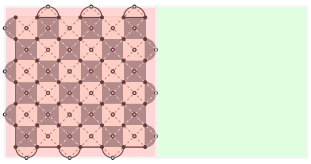
(b) $\circledcirc_1$ Stretched Merge



(c) $\circledcirc_1$ Measure out left patch.



(d) $\circledcirc_2$ Merge with original patch.



(e) $\circledcirc_2$ Measure out right patch.

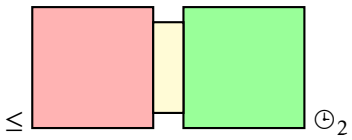



Figure 3.10: Flipping the alignment of the boundary using a stretched stabiliser [48].

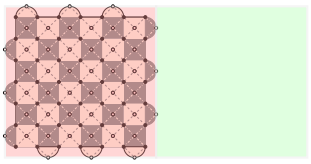
code. This deformation may be performed by a series of local SWAP operations, as seen in Figure 3.4. Rather than the twisted merge this operation may also be performed using a stretched merge [48] . The stretched merge requires the addition of another base operation on the border region.

The twisted merge rotation may be seen in Figure 3.11, while the stretched merge rotation may be seen in Figure 3.10.

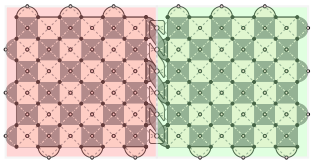
By combining both a boundary stabiliser rotation and a boundary alignment rotation into a single operation all rotations of stabiliser boundaries may be performed in $3n$ cycles of the surface code with a single additional ancillae. These rotations provide support for implementing an arbitrary choice of merge operations on a given target patch. A full sequence of operations implementing a rotation can be seen in Figure 3.12.

### 3.2.3 Clifford Operations

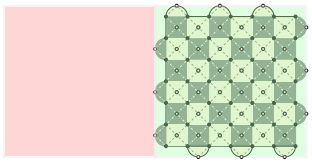
A common choice of gatesets for the surface code is CHP+T[87]. This represents the CNOT, Hadamard and Phase gate, and with the addition of a non-Clifford gate (commonly a T gate) forms a universal basis for approximating an arbitrary unitary operation. Compilers synthesise more complex rotations into sequences based on these gates.



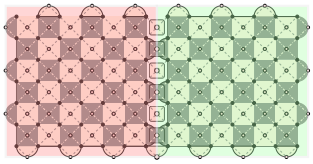
(a) $\circlearrowleft_0$ Initial State.



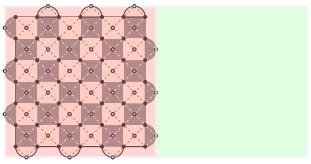
(b) $\circlearrowleft_1$ Twisted merge.



(c) $\circlearrowleft_1$ Measure out left patch.



(d) $\circlearrowleft_2$ Merge with original patch.



(e) $\circlearrowleft_2$ Measure out right patch.

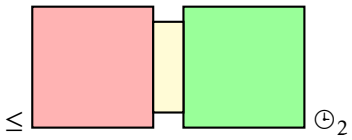



Figure 3.11: Flipping the alignment of the boundary using a twisted stabiliser in $2n$ surface code cycles[89]. This operation may be merged with the boundary rotation from Figure 3.9 to achieve any re-ordering of the boundary stabilisers for a patch in $3n$ operations.

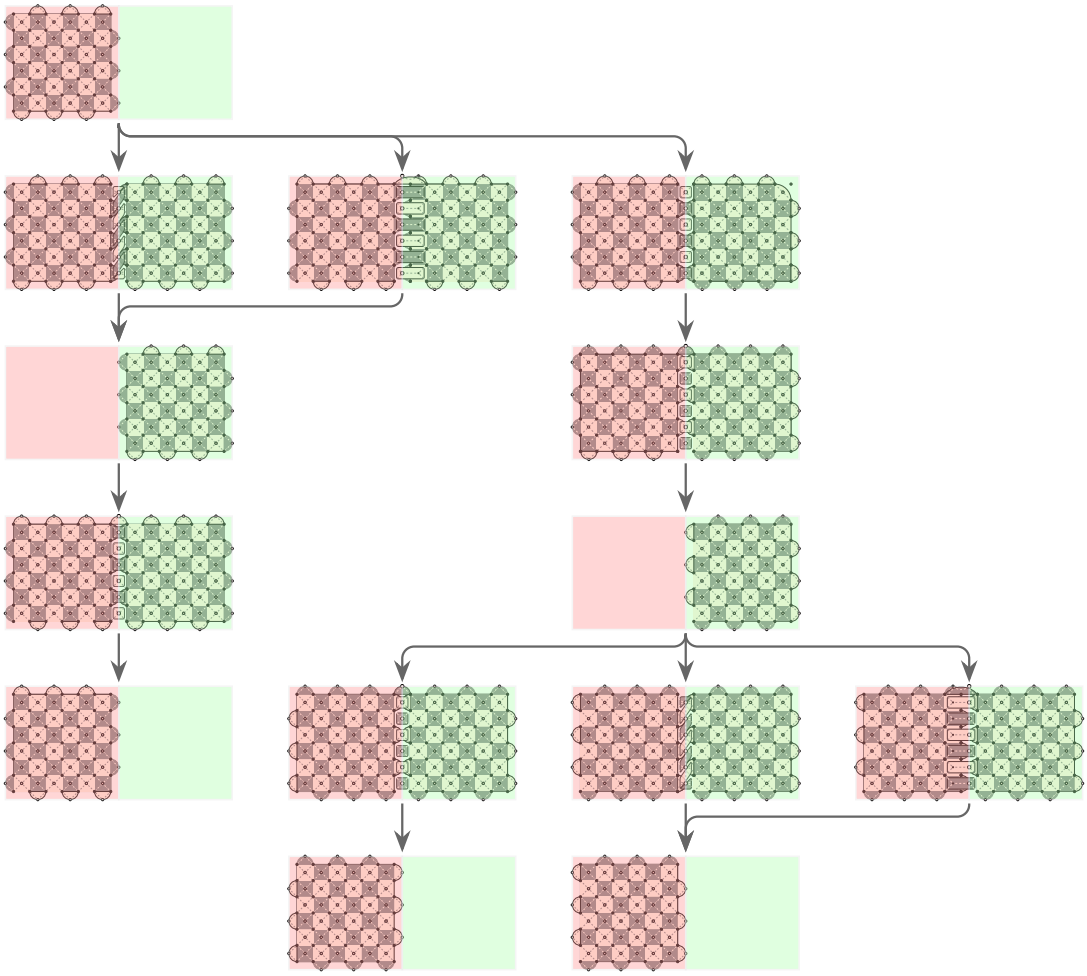


Figure 3.12: Flowchart of single patch rotations using edge stabiliser movement, twisted and stretched joins. A single patch may be rotated to any orientation in $\circlearrowleft_3$ cycles with the use of an ancillae patch.

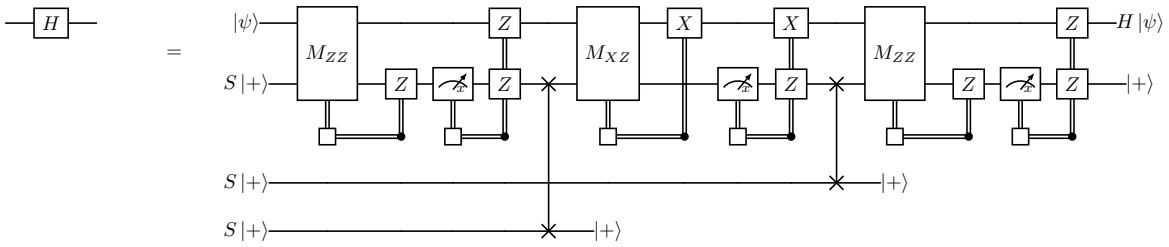



Figure 3.13: Implementation of a Hadamard using three $S|+\rangle = |Y\rangle$ states. This protocol acts in place without rotating the target state, but requires the preparation of three $|Y\rangle$ states.

### Hadamard

There are multiple implementations for the Hadamard gate, including transversal, topological, and compiler tagged.

A transversal Hadamard gate flips both the encoded X and Z logical states and the stabilisers, leaving the patch in a rotated orientation [48]. Interacting with a patch in this misaligned state may require a rotation, as shown in Figure 3.9 or a twisted or stretched merge operation. This operation takes $\leq \clubsuit_1$. With the addition of a rotation back to the original state it takes at most $\clubsuit_3$ and an ancillae qubit.

The Hadamard may also be implemented using a sequence of merge and split operations acting as joint measurement operators. This sequence can be seen in Figure 3.13 [87]. This method requires either access to both the $X$ and $Z$ boundaries of the patch containing the target state, or implies the need for rotations. Otherwise it has the advantage that it leaves the stabilisers of the surface code patch unchanged, whereas the other implementations leave it in a rotated state. As we will see when we encounter the phase gate, this sequence is equivalent to $Z(\frac{\pi}{4})X(\frac{\pi}{4})Z(\frac{\pi}{4})$.

The tradeoff with this approach is that it is bound by our ability to construct $S$ gates, as we can only prepare $|+\rangle$ and $|0\rangle$ states natively. The time cost then depends on the cost of constructing a $|Y\rangle$ state, and whether a rotation is required.

The default Hadamard gate for our compiler is to use the transversal gate, and track rotations as needed.However both forms of the Hadamard gates are supported, as are 'L' shape ancillae rotations[48].

### CNOT

The CNOT gate can be expressed as a ZX rotation about two qubits[101]. This rotation along with a set of correcting operation can be constructed from a sequence of rough and smooth merge operations. Circuits implementing a CNOT can be seen in Figure 3.14 and Figure 3.15.

These implementations use an ancillary $|+\rangle$ and $|0\rangle$ state respectively. As established in (SECTION) surface code patches may be prepared in these states in time (GOS). The implementation of the first of these circuits on the surface code may be seen in Figure 3.17.

It is important to note that this action takes place over an ancillae state, not necessarily an ancillae patch. Using a chain of ancillae this operation can be performed non-locally. As merge

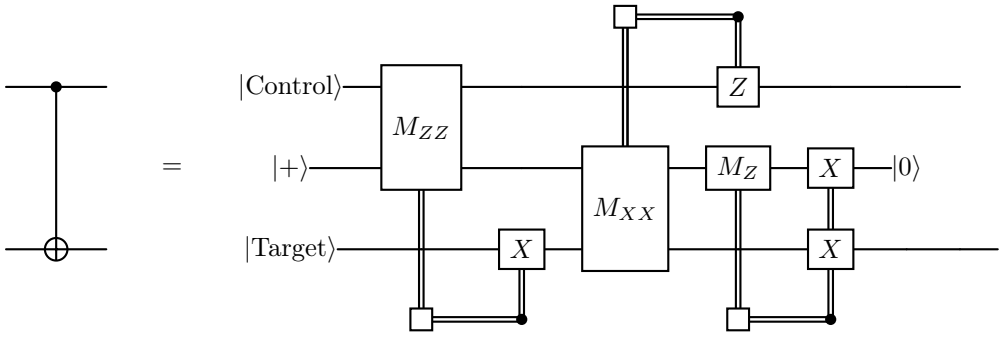



Figure 3.14: CNOT operation using an ancillae qubit prepared in the $|+\rangle$ state. The joint $Z$ measurement and the joint $X$ measurement are equivalent to rough and smooth merges and splits between surface code patches.

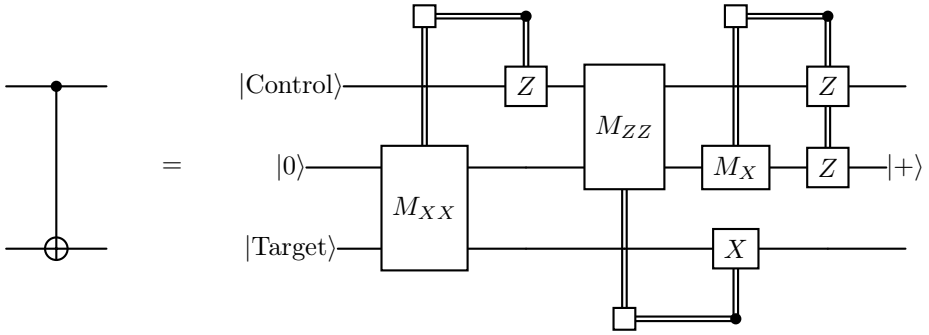
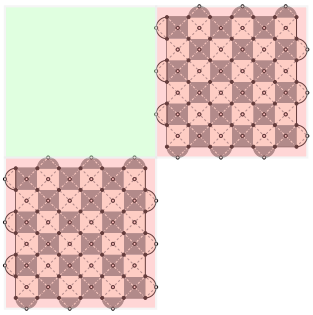
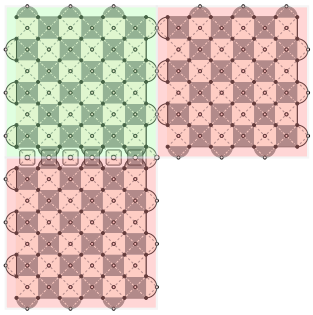


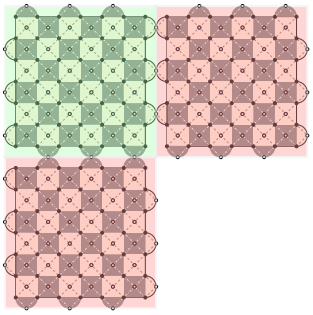

Figure 3.15: CNOT operation using an ancillae qubit prepared in the $|0\rangle$ state. The joint $Z$ measurement and the joint $X$ measurement are equivalent to rough and smooth merges and splits between surface code patches. This is a commuted version of the circuit in Figure 3.14



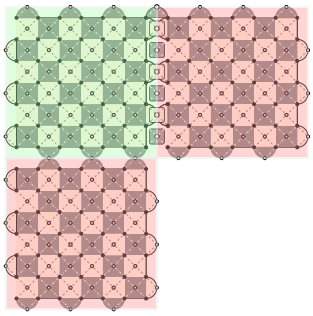
(a) $\circledcirc_0$ Initial State



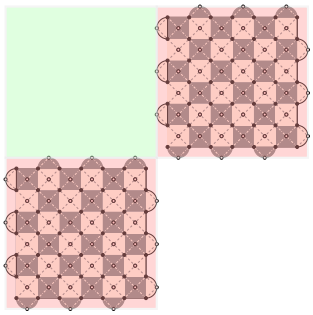
(b) $\circledcirc_1$ Merge control with ancillae.



(c) $\circledcirc_1$ Split control and ancillae

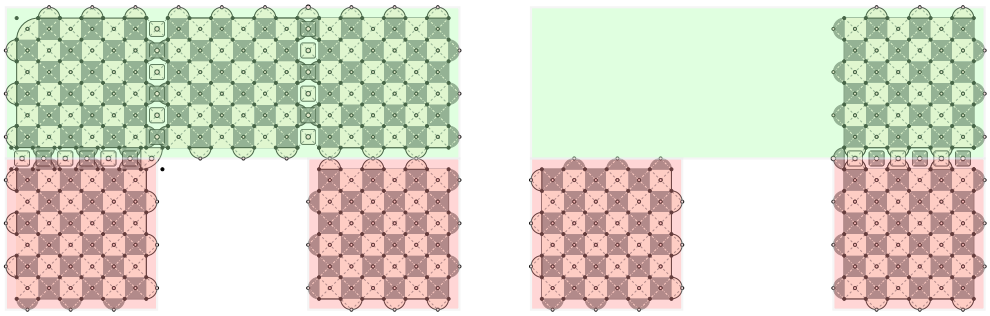


(d) $\circledcirc_2$ Merge target and ancillae



(e) $\circledcirc_2$ Split and measure out Ancillae

Figure 3.16: Implementation of a CNOT operation on the surface code. The ancillae is prepared in the $|+\rangle$ state, and merge operations are used to perform joint $ZZ$ and $XX$ measurements. These operations implement the circuit in Figure 3.14. Reversing the order of the merge operations and preparing a $|0\rangle$ state instead gives Figure 3.15



(a) $\circlearrowleft_1 M_{ZZ}$ (b) $\circlearrowleft_2 M_{XX}$.

Figure 3.17: Implementation of a non-local CNOT operation on the surface code. The ancillae is prepared in the $|+\rangle$ state, and merge operations are used to perform joint $ZZ$ and $XX$ measurements. These operations implement the circuit in Figure 3.14. Reversing the order of the merge operations and preparing a $|0\rangle$ state instead gives Figure 3.15.

operations may be performed to construct a multi-patch $|0\rangle$ or $|+\rangle$ state, the same circuits may be applied against this larger ancillary qubit.

This idea may be extended to construct a non-local but disconnected logical intermediary $|+\rangle$ or $|0\rangle$ state that may be consumed to support a CNOT operation [14]. An example of this on the surface code may be seen in Figure 3.22.

Additionally this extension may then support controlled mutli-qubit Pauli rotations [circuit]

So long as all boundaries of an ancillae patch are not used simultaneously, it is possible to construct an arbitrarily long ancillae patch such that the smooth boundary faces a control qubit, and all other registers border rough edges. The final smooth edge is unpaired. This condition is satisfied if not all connected ancillae are in the patch, as the boundary to an unincluded ancillae may then contain the smooth boundary. From this construction it is then possible to perform a multi-target CNOT gate using a single control.

The non-locality of CNOT operations may be a spatial or temporal non-locality. By constructing a multi-patch $|+\rangle$ state, and then measuring intermediary elements of the patch in the $X$ basis a disjoint set of entangled patches may be prepared. Applying the appropriate smooth and rough merges over this object then performs the same CNOT operation as if the chain of patches were connected[14].

**Phase**

While Hadamard and CNOT gates can be performed transversally and topologically, we are still missing a Phase gate. Depending on whether or not we have twist operations we can perform this either topologically, or using an ancillae in the $|0\rangle + i|1\rangle$ state (termed a $|Y\rangle$ state). As we may only prepare qubits in the $|0\rangle$ and $|+\rangle$ states (and may obtain $|1\rangle$ and $|-\rangle$ states from transversal Pauli operations) with the current gateset, we require a distillation protocol to construct these $|Y\rangle$ states. There are three main approaches for this: state distillation [49, 3, 47] twist braiding [23, 89], non-planar surface code folding [97] or twist fusion[51].

Once a single $|Y\rangle$ state has been prepared it can be used to perform phase gates on other qubits

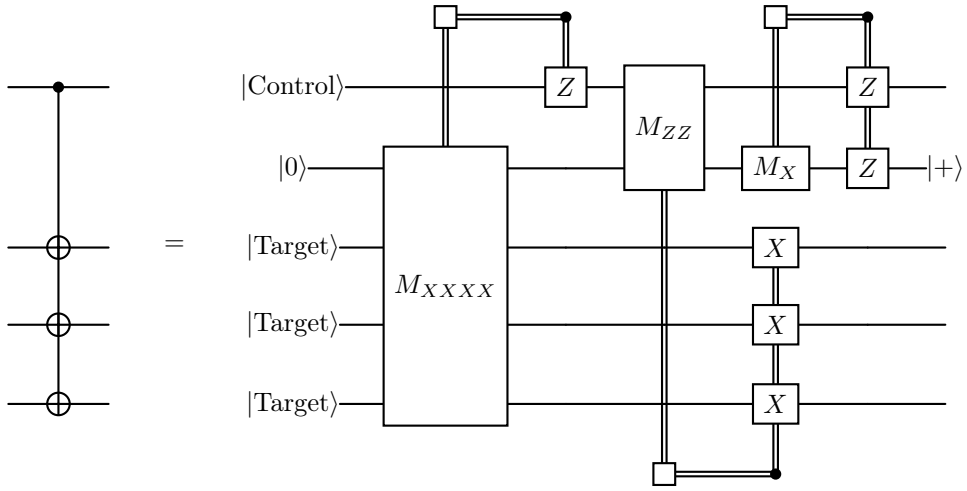



Figure 3.18: CNOT operation using an ancillae qubit prepared in the $|+\rangle$ state. The joint $Z$ measurement and the joint $X$ measurement are equivalent to rough and smooth merges and splits between surface code patches.

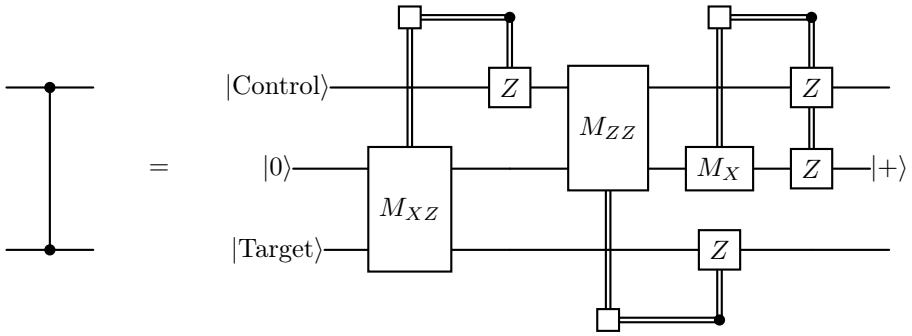



Figure 3.19: CZ operation using splits and merges.

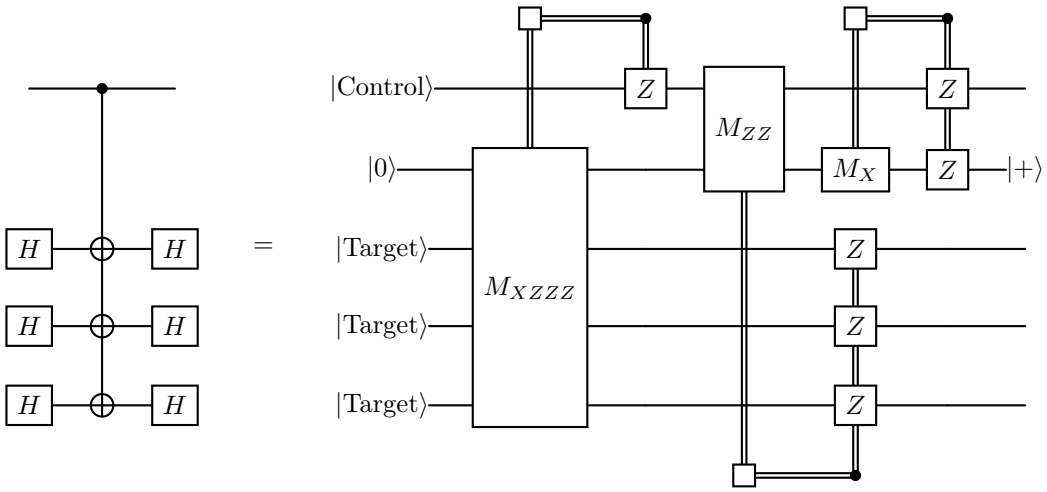



Figure 3.20: CZ$\cdots$Z operation.

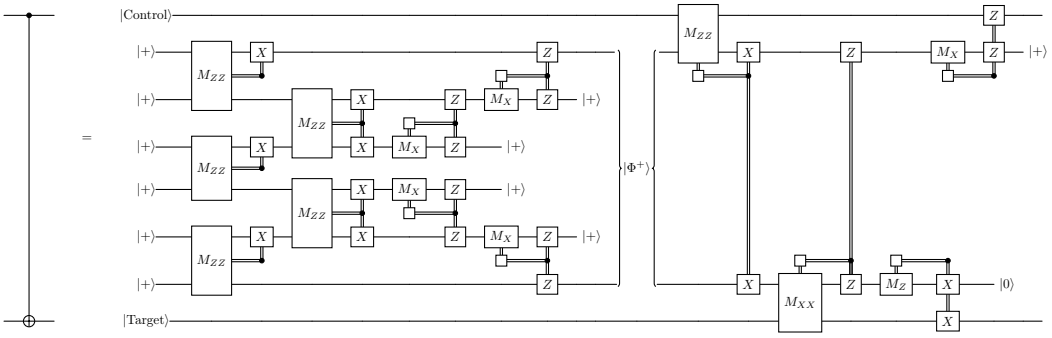
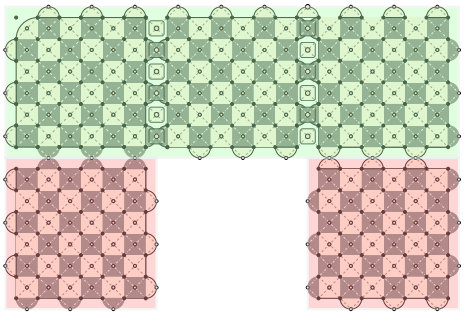
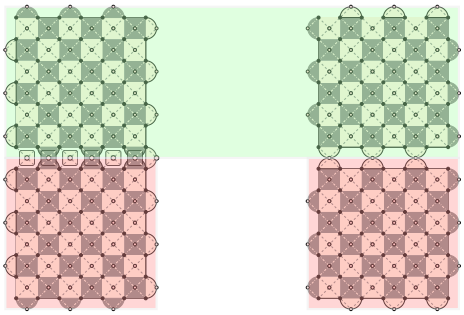


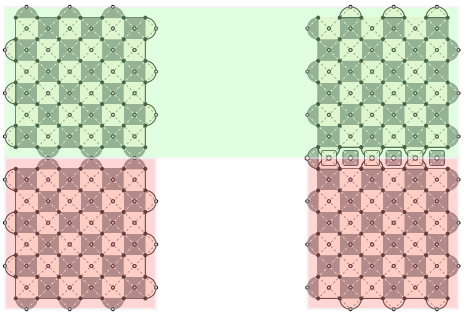

Figure 3.21: CNOT circuit using a chain of $|+\rangle$ state ancillae. The first part of the circuit constructs a Bell state consisting of the top-most and bottom-most ancillae qubit, other ancillae are recurited to construct the state before being measured out. The CNOT operation is then mediated only by operations between these terminal ancillae and the control and target qubits[14]. This implies that routing ancillae may be prepared ahead of time, and that only the terminal ancillae need to be preserved to implement a multi-qubit operation.



(a) $\circled{1}$ Prepare $|+\rangle$ state.

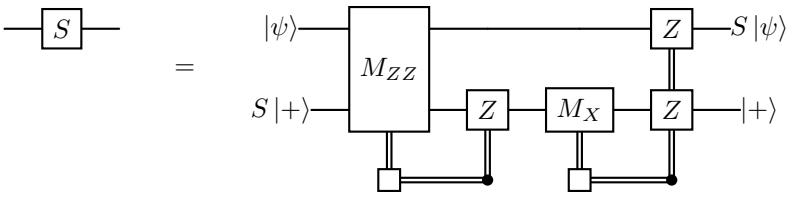


(b) $\circled{3}$ Measure intermediary qubit, rotate boundary and $M_{ZZ}$.



(c) $\circled{4}$ $M_{XX}$.

Figure 3.22: Implementation of a non-local CNOT operation on the surface code with a disconnected ancillae [14]. The ancillae is prepared in the $|+\rangle$ state, before patches are measured out in the $X$ basis. Rotations are applied to the boundaries to ensure appropriate alignments with the target and ancillae qubits before merge operations are used to perform joint $ZZ$ and $XX$ measurements. These operations represent a small example using the same logic as the circuit shown in Figure 3.21.




(a) Circuit representation of a phase gate consuming a $|Y\rangle = S|+\rangle$ ancillae state.


Figure 3.23: Phase operation where an ancillary $|Y\rangle$ state is consumed.

without depleting the original $|Y\rangle$ state [54], i.e.: $|\psi\rangle\otimes|Y\rangle \rightarrow S|\psi\rangle\otimes|Y\rangle$. However this approach is slower than directly consuming the $|Y\rangle$ state. An example of a scheme that consumes a $|Y\rangle$ state may be seen in Figure 3.23, while one that preserves the $|Y\rangle$ state may be seen in Figure 3.24.

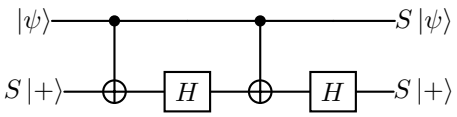
Once a $Y$ state is constructed there are multiple methods for performing a Phase gate. The first and fastest consumes the $|Y\rangle$ state, which would require a repetition of the preparation routine. One slower method preserves the state and does not entangle it with the target qubit. Between these methods it is possible to replicate a producer-consumer pattern. After one round of the $|Y\rangle$ preparation routine as a seed for the $|Y\rangle$ factories further states may be produced arbitrarily. Phase gates may then be constructed by routing the $|Y\rangle$ states to a target qubit and performing the faster $ZZ$ measurement procedure. The consumed $|Y\rangle$ states may then be replenished by the seeded factories.

Another method of performing phase gates is using a $Y$ basis measurement rather than a $|Y\rangle$ state[87, 23]. These twist-based $Y$ measurements can be performed with two ancillae qubits in a $|0\rangle$ state, but risk reducing the code distance[51]. Implementing the twists also also requires architectural support for weight five stabilisers on the corners of surface code patches.

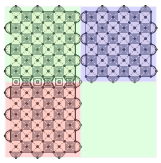
Lastly fusion twists provide a $\circledcirc_2$ method of preparing a $|Y\rangle$ state in a single surface code patch which may then be consumed by an adjacent target qubit [51]. This reduces phase gates to only requiring a single local ancillae. This twist fusion approach would require architectural support for twisted stabilisers within the bulk of some subset of surface code patches.

### 3.2.4 T Gates and Magic State Factories

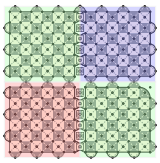
As discussed in the context of $|Y\rangle$ states and phase gates, some operations on the surface code are only possible using prepared ancillae states using non-fault tolerant operations. These are generally referred to as magic states and support a family of operations related to the $T$ gate [49].



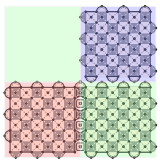

(a) Circuit representation of a Phase gate using a catalysing $|Y\rangle = S|+\rangle$ ancillae state. This implementation preserves the ancillae state without entangling it with the target.



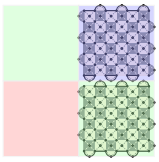
(b) Merge control and ancillae. The state $|\psi\rangle$ is represented in the bottom left corner, while $|Y\rangle$ is in the top right.



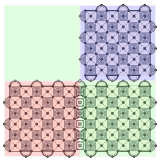
(c) Merge $|Y\rangle$ state and ancillae, completing first CNOT. Start rotation of boundary stabilisers of $|\psi\rangle$



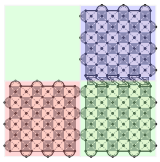
(d) Hadamard $|Y\rangle$ state register. Rotate boundary stabilisers of $|\psi\rangle$



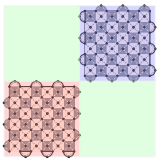
(e) Continue rotation of boundary stabilsiers.



(f) Merge $|\psi\rangle$ with ancillae $|+\rangle$ state.



(g) Twisted merge due to offset boundary stabilisers, completing the CNOT operation.



(h) Hadamard on $|Y\rangle$ state.

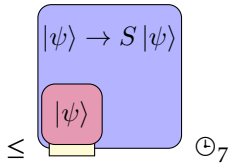



Figure 3.24: Phase operation where an ancillary $|Y\rangle$ state is not consumed and is not entangled with the target state allowing for its re-use.

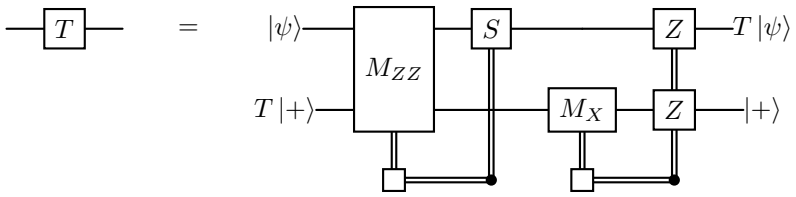



Figure 3.25: Circuit representation of a T gate consuming a $|T\rangle = T|+\rangle$ ancillae state. The correction operation involving an $S$ gate conditionally requires a $|Y\rangle$ state.

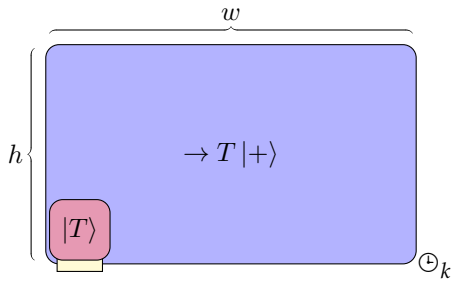



Figure 3.26: $T$ state distillation QCB template. An example implementation circuit for the 15-1 $T$ state factory using bare noisy $T\dagger$ gates can be seen in Figure 3.27.

In general a magic state factory can be concieved of as implementing an error correcting code on top of the surface code[56, 55] . The error correcting code is chosen such that the desired gate is a transversal gate in that code [19, 26]. The target gate is applied, the state is decoded, and syndromes measured. If the syndrome bits report no error then computation continues, otherwise the distillation process is repeated until the syndrome bits report no error [88].

As these gates are themselves part of an error correcting code, the distillation protocol may be nested, such that the output of one round is the input to another.

We are fortunate that there exists a defect twist protocol for a $\circlearrowright_2$ $S$ gate. This operation notes that a $S$ gate may be constructed using a $M_Y$ operation, and provides a set of stabiliser operations that implement this sequence on a patch [51].

$T$ gate implementations are often performed by the noisy application of independent 'bare' $T$ gates to individual patches of the surface code, before performing a distillation protocol. This distillation process ends with a measurement that either projects to a flagged and failed $T$ gate preparation, or a successful $T$ gate preparation with some probability.

An example 15-1 $T$ state distillation protocol may be seen in Figure 3.27. This implementation is based on a tri-orthogonal code with a transversal $T$ gate[48].

As discussed above this protocol may additionally be nested. Rather than performing bare $T$ gates each of those $T$ gates are replaced with the output of a previous distillation protocol [88, 48].

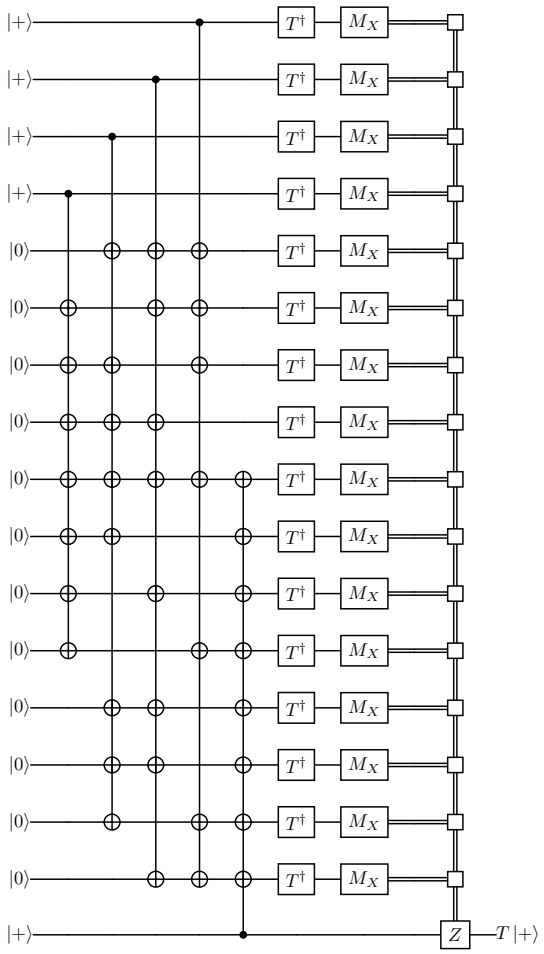



Figure 3.27: T state distillation circuit using noisy $T^\dagger$ gates [48]. These can be constructed by flipping the parity of the classical control on the $S$ gate in Figure 3.25 An example QCB template from compiling this circuit can be seen in Figure 3.26.

### 3.2.5 Cost Model

In all of the above example we may consider a computational model where stabiliser operations are centred on the ancillae qubits. By enforcing the same local ordering of two qubit operations (for example clockwise) the majority of these operations are independent and can be performed in parallel. These assumptions form the basis of a hardware cost model which combined with the runtime for a decoder provides an upper bound on the wall time for each cycle of the surface code. Previous estimates assume a surface code cycle time on the order of 1 microsecond [52, 78, 49]. Additionally as many surface code operations must be repeated to ensure fault tolerance, the minimal computational unit of $n$ surface code cycles per operation adds a factor of $n$ to the time required to perform each computational operation. This estimate depends on the runtime of the decoder, the gate operations for each cycle (which then depends on the physical device connectivity) and the measurement time. Current decoders are bounded to $n \leq 17$ for the measurement time of current superconducting quantum computers [71].

## 3.3 Quantum Circuit Boards

We term a collection of surface code patches a 'Quantum Circuit Board' or QCB. Our model of the QCB assumes that each patch has the same distance and we will default to a level of granularity that discusses individual patches, while the implementation of operations requiring the manipulation of components of patches will be abstracted. We assert that there exists a set of primitive stabiliser operations that provides support for each of the operations described, and provide examples of each in subsequent sections. Gates are treated as operations that lock collections of patches for a number of cycles. Typically numbers of cycles will be reported as multiples of the distance of the underlying patches.

### 3.3.1 Surface Code Patches

Elements of the surface code have an X-Z orientation depending on what stabilisers are exposed on the boundary of a given patch. Each gate tags which of its registers requires either an X or a Z boundary exposed [89].

We associate a type with each surface code patch in the QCB. These types are twinned with gate elements in the QCB which forms the basis of our binding from the gate description to the physical memory.

- Register patches are local memory elements. Each is associated with a particular name and may be the target of operations acting over the surface code.
- Routes are ancillae qubits. These are used for non-local operations, and as ancillae for local operations such as rotation gates. Some route patches are assigned greedily at the end of compilation and are not guaranteed to be usable as routes. These are 'Local' patches and may be used as ancillae qubits for local gates, but not for routing.
- Externs represent externally defined QCB operations. A compiled QCB may be treated as an extern by aligning its IO and locking it for a requisite number of cycles. We treat externs as a set of both resources and dependencies, the management of which is crucial for reducing the runtime of a wide range of quantum algorithms.

- IO elements acts as input and output 'pins' on a QCB. Within an internal scope they are treated as registers along the bottom edge of the QCB, while from an external scope they are treated as the bottom edge of an extern. By routing from an external scope to an IO element, data may be passed to the internal scope of an extern.

Gates then act over the names of memory types; registers, externs and IO elements. Compilation is the act of mapping these named elements to particular surface code patches, and treating a gate as a map from a set of labels to those patches any additional ancillae patches.

To better conceptualise extern and IO elements, in Figure 3.24 once a specification for the phase gate is provided we can containerise the operation as a seven cycle lock over a collection of surface code patches. Here the $|\psi\rangle$ patch is the internal IO element, while the locked region is the extern. Generalising this notion in Figure 3.26 the extern symbol that generates a $|T\rangle$state may apply to a family of distinct implementations. Each with varying collections of extern and IO patches and symbols. We note at this point that the term 'symbol' is highly overloaded, and in this context we may consider register and gate symbols as 'idents', while DAG and Extern symbols require further explanation. For the purposes of resolving gates any of these implementation are treated equivalently.

### 3.3.2 Quantum Logic Gates at the Level of Patches

We define our gates as an integer number of cycles (each representing $n$ surface code cycles), a set of $X$ and $Z$ symbols (corresponding to which surface code boundary is required for the operation) and an optional flag for a single ancillae or an elbow ancillae. Gates may additionally be expressed as an 'Extern', which implies that the implementation of the gate will be externally defined during compilation with a specification for a smaller QCB and the instructions that it implements for a fixed number of cycles. This extends the idea from previous works [107] of dedicated magic state distillation regions and includes a notion of a local scope, along with the potential for dynamic reassignment of extern regions.

For example of this XZ argument typing a CNOT operation between two qubits might be contructed as:

CNOT:`Z{CTRL} X{TARG}`

and last for 2 cycles, while a multi-target CNOT would be similarly defined as:

Multi-Target CNOT: `Z{CTRL} X{TARG_0, TARG_1, ... TARG_n}`.

Unary gates do not have an edge associated typing and so these fields are treated as equivalent. Externally defined gates use the Z field to indicate inputs and the X field to indicate outputs, these are resolved using move operations. Some gates may additionally permit an argument in both the X and Z fields such as can be found with Litinski slice operations [87].

We extend the idea such that labels for registers are themselves symbols. A sequence of gates may also be treated as a symbol. The Toffoli gate can be decomposed into a sequence of CNOT, Hadamard and $T$ gates. The Toffoli gate may itself be expressed as a symbol

Toffoli: `Z{CTRL_0, CTRL_1} X{TARG}`,

a 'scope' then acts to remap labels from the internal definition of the Toffoli gate to the external symbols. In this case based on the ordering of the arguments within the X and Z blocks.

Now that both gates, collections of gates and memory are all represented with the same notion of a 'symbol', we extend the idea to a QCB itself. Each compiled QCB acts as a closure over a collection of surface code elements. A QCB is defined as a symbol, where the $X$ and $Z$ elements of that symbol provide the IO labels for that patch. Unlike a gate sequence, calling a QCB does not trigger a remapping of the internal labels of the QCB, but instead requires the allocation of a block of memory to be treated as an extern. Only the IO elements defined in the symbol are exposed, while the internal variables are not. An extern may be allocated and freed indicating when that region of memory is required, and if and when it has been safely uncomputed and may be released to the system.

Our QCB compiler then establishes an association between each gate and a collection of memory elements and acts as a lock over those elements of memory during the operation of the gate. For unary gates without ancillae this applies to register and IO elements, for multi-qubit gates or unary gates with ancillae this extends to routing patches, and for externs it requires locking an extern block. While an element is locked it cannot be accessed by another gate. Once compilation of a QCB is complete the final routing provides an integer number

may be determined how many cycles are required to implement each gate while satisfying the locking dependencies. This specification of a number of cycles, a set of $X$ and $Z$ symbols as IO elements and a height and width in turn allows a compiled QCB to act as an extern in another QCB.

### 3.3.3 The Qubit Routing and Mapping Problems

The routing and allocation problems are subsets of the larger qubit mapping problem[86, 14]. Given a quantum device with a cost function (such as circuit depth, runtime or error rate) find an allocation of qubit labels from a circuit to physical qubits on the device that minimises this cost function. On NISQ devices the qubit mapping and routing problems arise from the limited device connectivity and relatively high error rates for two-qubit operations. Non-local two qubit gates require networks of SWAP or CNOT gates [86] to move the target qubits into proximity with each other before performing the gate. Strategies for placing qubits to minimise the number of these routes is termed the 'qubit mapping problem' while minimising the routing is termed the 'qubit routing problem'.

This model differs significantly for surface codes. Large ancillae routing states can be constructed in a number of cycles equal to the distance of the code. These non-local states can then be interacted with by neighbouring patches and measured out to implement a non-local gate. The primary concern with mapping and routing on the surface code is the construction of these ancillae states.

The qubit mapping problem for surface codes is two fold. The first is the need that all qubits that are involved in a multi-qubit gate have an appropriate route of ancillae qubits that supports them. As each route depends on the intermediary ancillary qubits, they are constrained to operating on a single route at a given time. The secondary concern is then to minimise the number of simultaneous joint dependencies of each route by minimising the number of intersections between simultaneous operations.

The mapping itself then should assign each register to a surface code patch, and demark other

patches for ancillae operations. In full generality these allocations may be dynamic [31], and change during the course of execution of the program. The routing then requires path finding between surface code patches such that simultaneous paths of ancillaes do not intersect. This protocol may be extended by preparing intermediary routing ancillae states as collections of edge disjoint paths [14].

### 3.3.4 Definition of a Quantum Circuit Board

A program, or circuit that acts over the QCB consists of a sequence of gates. Each gate acts over a set of labels corresponding to registers. Some subset of gates may be flagged as external dependencies. The sequencing of the gates and the labels over which they act expand to form a directed acyclic graph (DAG) indicating the dependencies between gates. The program itself additionally indicates a subset of labels which are treated as input and output pins.

To construct a QCB that can implement this program we require that

- The QCB contains enough register patches to match each non-IO, non-exten label.
- It must contain at least one copy of each external dependency.
- It must be sufficiently wide to accomedate all IO elements.
- All registers, IO channels and externs must be connected by routing ancillae, such that any multi-patch operation in the circuit can be implemented.

In the next section we will discuss the implementation of a compiler that constructs QCB specifications that satisfy these requirements, while tackling aspects of the qubit mapping and routing problems.

## 3.4 Compilation

This section gives an overview of our compiler. For our purposes, the task of compilation is as follows: (1) choose a mapping of virtual memory to surface code patches, (2) identify routing lanes between surface code patches (including edge disjoint paths), (3) nesting this mapping as sequence of pre-compiled elements, and (4) trying to find reductions in runtime. Our input is a quantum circuit expressed as Clifford + $R+z(\theta)$ gates, and the output is a QCB plus a sequence of surface code instructions defined relative to that QCB.

This section is organised as follows. In section 3.4.1, we describe how the quantum circuit, which may be presented in a gate set we are not prepared to manage, is to be reduced Clifford+T. In section 3.4.2, we describe how we transform the Clifford+T quantum circuit into a DAG. In section 3.4.3, we explain how we construct an initial memory allocation on the QCB, which is minimal in the sense that it is guaranteed to support the execution of the quantum circuit but is not optimised. In section 3.4.4, we explain how the previous 'minimal' QCB allocation may be improved in order to achieve better runtime. In section 3.4.5, we explain how the virtual qubits are assigned to different 'QCB registers'. In section 3.4.6, we explain how to allocate memory for the externs. In section 3.4.7, we detect and inject rotation gates where routing is not possible. In section 3.4.8, we perform routing, replace routes with edge disjoint paths and decide on a mapping strategy for externs.

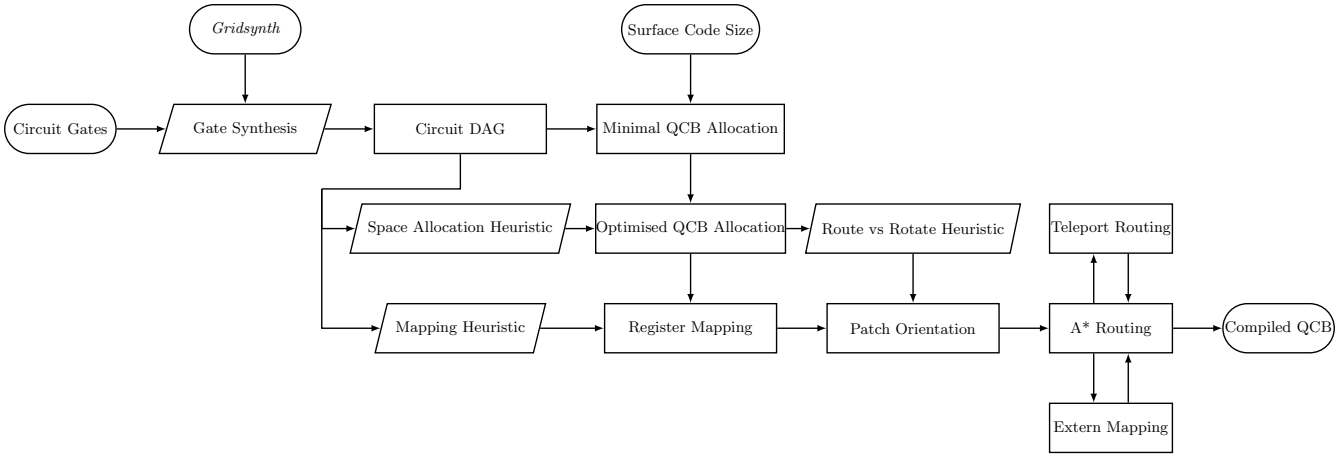



Figure 3.28: An overview of the compilation pipeline

### 3.4.1 Gate Synthesis

While quantum operations may consist of an arbitrary unitary operation, only a subset of operations are supported natively by the surface code. Other gates must be constructed or approximated by a sequence of gates in some basis set. We utilise *GridSynth* as an efficient gate synthesis stage[116] to reduce the input unitaries into a set of legal gates for the surface code architecture.

We decompose larger programs for the surface code into the operations defined in Section 3.2 along with a sequence of $R_Z(\theta)$ gates. These Z rotation gates are passed to GridSynth with an appropriate approximation precision. The output of 'GridSynth' is expressed as a sequence of Pauli operations, Clifford operations and $T$ gates, which in turn we map to surface code operations.

Angular rotations that are smaller than the precision provided are a programmer concern. If the identity is within $\epsilon$ of the requested gate Gridsynth will report an empty gate sequence.

### 3.4.2 Circuit to DAG and Resource Queuing

A quantum circuit may be expressed as a directed acyclic graph. Verticies in the graph represent operations while directed edges in the graph represent dependencies. With no additional constraints the runtime of a quantum circuit (the circuit depth) may then be approximated as the maximum path length through this DAG.

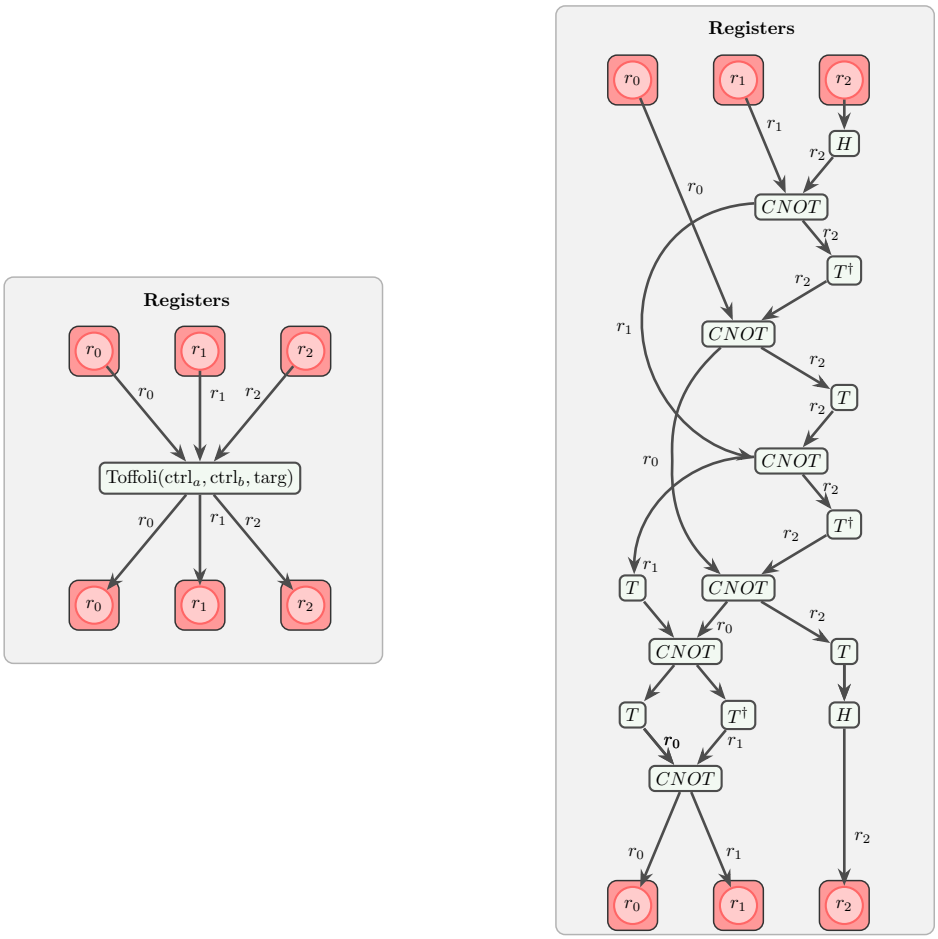



Figure 3.29: Toffoli Instruction and Expanded DAG.

Some gates are themselves expressible as a smaller DAG as seen in Figure 3.29. These gates are expanded and symbolic substitution is performed to map registers from the scope of the outer DAG to the inner DAG. To continue the example from the previous section a Toffoli might similarly be defined as `Toff: Z{CTRL_0, CTRL_1} X{TARG}`. As a Toffoli is not a native gate on the surface code, this dag expands to a sequence of gates as seen in Figure 2.1. Each $T$ and $T^\dagger$ gate in that sequence is an 'extern' gate and introduces a dependency on a magic state factory. The symbolic substitution remaps the arguments of these gates to the scope of the current DAG. If a register is not defined in the current scope it has been declared in the subscope and is flattened into the current scope. It is promoting registers from subscopes, as may lead to aliasing of those registers in the higher scope. As a result this is not advised for non-ancillae registers.

To better represent the topological operations on the surface code we specify a type system for different gates in the DAG.

A unary gate acts locally on a single register and may or may not require ancillae, examples of surface code unary gates include rotation gates and transversal Pauli operations. A non-unary gate necessarily acts non-locally and requires a set of external routing resources that are not

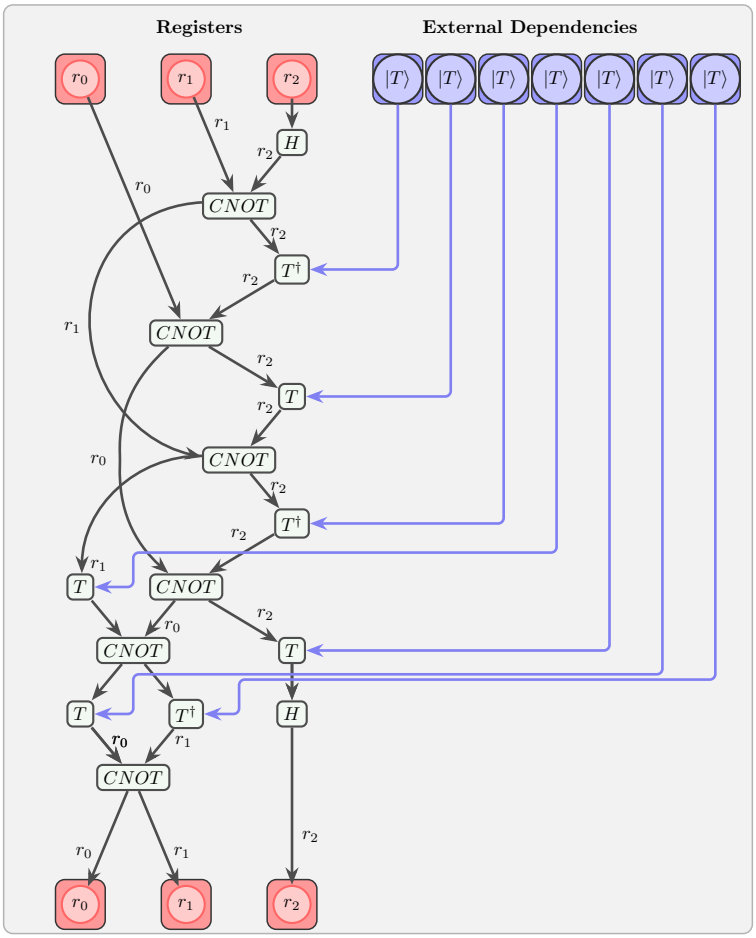



Figure 3.30: DAG for the Toffoli gate decomposed into Clifford + T operations. The $T$ gates depend on external magic state factories.

captured in the DAG's dependency model.

An externally defined gate is an operation that makes use of a pre-compiled set of surface code operations. At the level of the DAG, an extern exposes only a name, and a set of IO symbols. The external dependency may be satisfied by any QCB that matches this naming scheme. Matching a DAG node with a QCB instance provides the extern with a specification for a memory footprint, and a number of surface code cycles which are required to execute that externally defined operation. The symbols and number of cycles may additionally be overloaded for externs with multiple functions acting over a persisent unit of memory.

A 'factory' is an extern with no inputs, only outputs. This definition leads to complexities, as each factory will exist at the top level of the DAG, which gives no indication as to the order in which they are to be resolved. A DAG decomposed into registers, gates and external dependencies can be seen in Figure 3.30. The order of resolution of the magic state factories is indeterminate as

For these external gates we add a special RESET flagging operation that indicates that the memory occupied by a given extern is now free to be measured and re-used. Other uses of a factory

can be manipulated in the DAG by not triggering a RESET. For example seeding an then accessing a re-usable phase factory.

As multiple instances of externs may share the same name, but evoke distinct contexts and scopes we mangle their variable name resolution. Non-extern registser symbols reference one object within the scope uniquely. Extern symbols reflect both a register and a function. The register element is dynamically bound and re-bound as the extern is reset and the memory re-used. To avoid confusion variable name resolution on externs is bound to the identity of the instance of the symbol, rather than the value of the symbol. Extern scope is then tracked by the scope of its associated symbol object, and multiple identical externs with unique symbol objects may be utilised simultaneously. Identical symbol values that are of different instances fail to match allowing for the selective aliasing to maintain the appropriate scope for each extern.

To this DAG we add two constraints, an integer bound on the number of simultaneous non-local operations (representing an approximate bound on the number of routing ancillae that are avaliable), and a bound on the number of simultaneous gates that require an external dependency (for example a T gate requiring a T factory). The DAG then describes the minimum number of registers and a minimumal set of external dependencies that must be defined to execute a given circuit. Given this resource constraints the DAG may also be used as a heuristic evaluation of the circuit depth given a particular bound on routing channels and on the number of external dependencies that have been allocated. Lastly the circuit provides a bound on the number of registers that are required to store the variables required execute the circuit.

An example of a DAG constrained by these bounds can be seen in Figure 3.31.

To calculate this circuit depth we allocate an integer number of nodes for each type of extern, and a semaphore for the number of simultaneous non-local gates. A priority queue of waiting gates is associated with these resources. The priority queue begins as the set of non-extern gates with no dependencies. When a gate is resolved any gates that depend on it are added to the queue if all of their dependencies have been resolved. Each cycle the queue is iterated over in priority order to determine if any element may acquire the resources to execute its associated gate and be dequeued. Once an element is dequeued it locks the associated resources for the number of cycles required to implement that gate before being flagged as resolved. This acts as a heuristic for the circuit depth, which will be replaced once the memory layout has been established and routing costs can be determined exactly.

Given this setup we can now parse a DAG and establish a relative ordering of gates. Of particular interest is determining gate priority when contending for a shared resource (be it the number of channels, or an external dependency). Each gate is assigned a 'slack' value based on the minimum number gates (independent of the number of cycles that each gate requires) that must be resolved before one of its dependencies may be resolved. This represents a minimum bound on the time required before a dependent gate is blocking on the current gate. Gates are ordered in the priority queue in terms of this slack value.

Despite this ordering, factories still present an issue. As factories have no dependencies and no relative ordering, but share a resource without determining an ordering a deadlock may occur if factory states are produced to satisfy dependencies later in the DAG, that are now unreachable due to earlier gates depending on the same locked resource states. To fix an ordering we

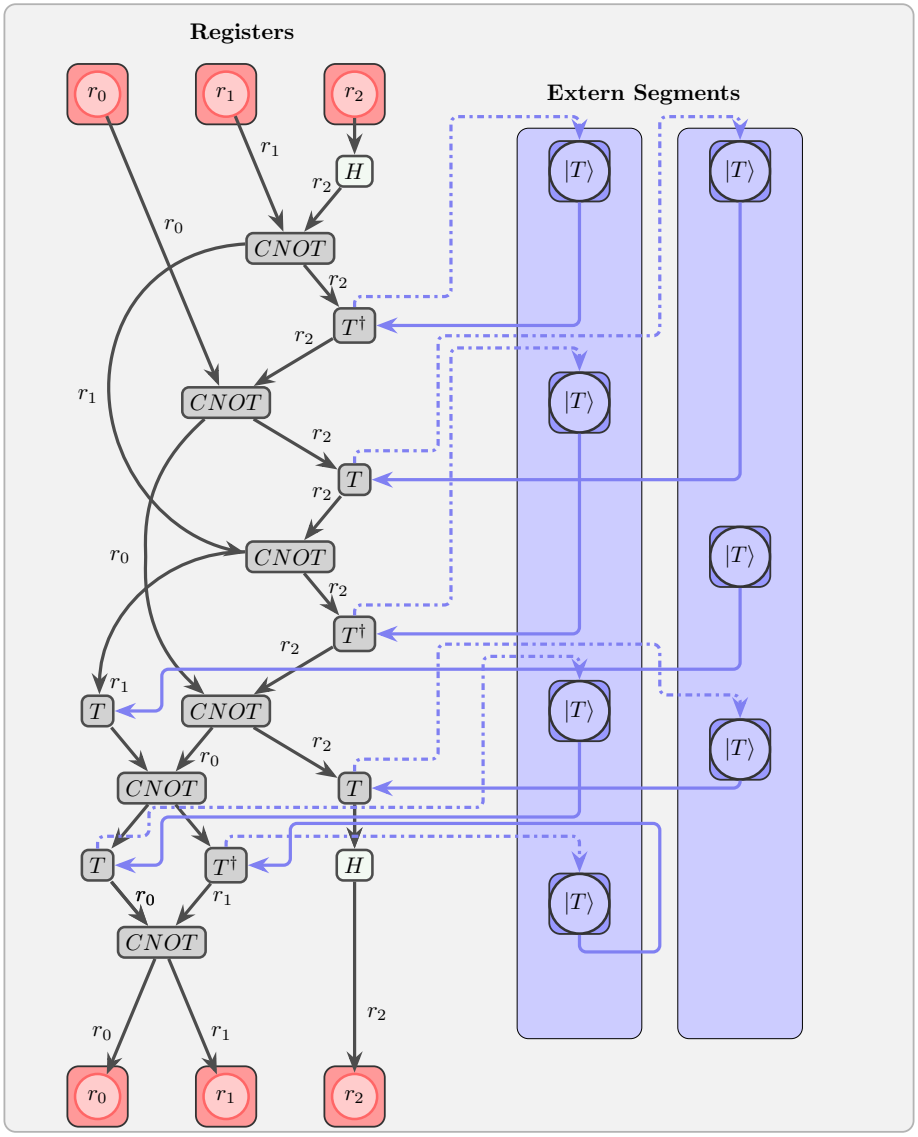



Figure 3.31: DAG for the Toffoli gate decomposed into Clifford + T operations. Two extern segments are allocated, and act as $|T\rangle$ state factories. Shaded gates act non-locally and are constrained by the routing semaphore. The depth of this circuit including queuing for the semaphor acts as a heuristic approximation of the execution time of the final QCB. This heuristic is used during the allocation phase to determine how many externs to allocate, and when to focus on placing additional routing ancillae. Dashed lines indicate tagged dependencies. These dependencies are placed into the DAG by allocating them to a legal and idle extern segment, where legality implies that the size of the extern matches or exceeds the height and width of the required QCB. The circuit depth of this model varies with the number and size of the externs, and the maximum value of the semaphore.

'tag' gates with dependent factories and only enqueue factories when a tag is encountered.

1. Gates that depend on factories are tagged.
2. Unary gates that depend on another tagged unary or factory gate copy those tags.
3. Multi-qubit gates must depend on at least one non-tagged gate.
4. External gates are flanked by multi-qubit operations between target registers and the extern's IO bank.

These constraints ensure that tags are triggered by the move operations to a target register which ensures that factory states are requested in the order in which they are depended.

The constraint that multi-qubit gates depend on at least one non-tagged gate is to ensure that the relative ordering of factory resolution from any given tag does not introduce deadlocks.

As this would normally result in factories only being queued when they are required this would delay the execution of the gate until that factory completed. To speed this process up each extern resource tracks the last cycle which it was required and factories are back-dated to this cycle. The extern resources are themselves then ordered by their last cycle, as a first in first out (FIFO) queue for each type of extern.

Lastly, the DAG may be intended to be compiled as an extern with its own IO elements. When the DAG is defined is is associated with its own set of IO symbols that are otherwise treated as regular registers.

When combined this process translates a circuit into a DAG, and given a set of extern resources and a bound on the number of channels approximates the circuit depth. Lastly it also provides a lower bound on the number of registers required to implement a that circuit.

### 3.4.3 Minimal QCB Allocation

With the three constraints provided by the DAG (the externs, the number of IO elements and the number of registers) we can construct a minimal layout of a surface code (termed a QCB) that may implement the quantum circuit. The QCB must contain at least one copy of each dependency, be wide enough to support each of the IO registers, and contain enough registers to meet memory requirements. Lastly to ensure that any non-local operations are satisfiable we require that all pairs of registers and extern IO elements are reachable via routing ancillae.

We structure the 2D plane of surface code patches into rectangular segments using a set of doubly connected edge lists and associated vertices. Each segment is described by four verticies and four edge lists of paired references to neighbouring segments. These segments may then be split into up to nine new segments, or an edge may be partially or fully merged with a neighbouring edge to create a larger segment. Some splitting may be required to ensure that the newly merged segments are still rectangular. With this object our split and merge operations now scale with the number of neighbouring segments, rather than the size of the segment. These splits and merges form the primitive operations for the allocation of our QCB.

By creating a segment and only updating the forward edges we can create a depth one rollback on the allocation. An allocation is not 'confirmed' until the back references on the neighbouring elements are updated.

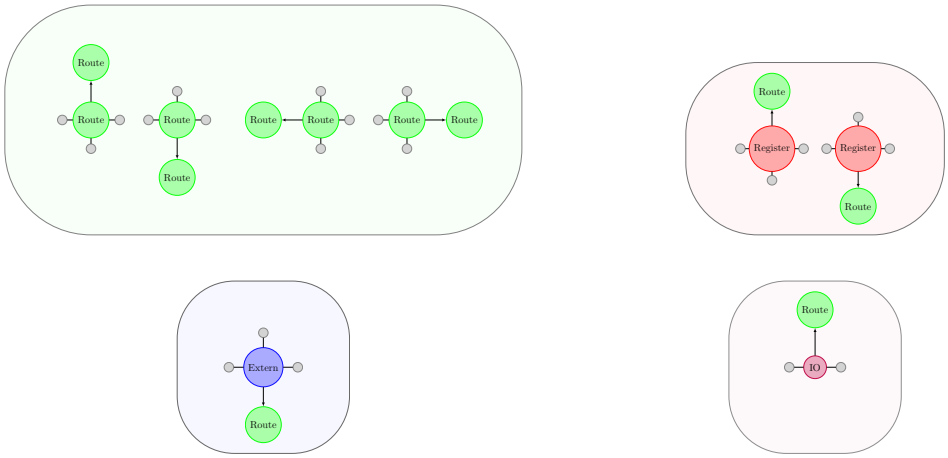



Figure 3.32: Allocation constraints for Routes, Registers, IO and Extern patches. The placement of each surface code patch on the QCB (as the larger circle) must satisfy at least one of these constraints

With these split, merge and rollback primitives we may now consider assigning each segment as either for routing, IO, registers or external use.

Our IO concerns are the simplest; we place each IO element from the bottom left of the QCB. If the width of the QCB is less than the number of IO elements compilation fails.

Next we can constrain the routing concerns. To place any routing element on the QCB (except the first) it must be in contact with an existing routing element. By induction all routing elements are connected. To ensure that all registers are similarly connected we set the constraint that a register may only be allocated if a route is either above or below it. That an IO channel must have a route above it, and that an extern must have a route below it (hence the IO elements of the extern are all reachable via that route).With these constraints we ensure that all elements are reachable by all other elements and hence (with the addition of rotations) all non-local operations are satisfiable.

By treating each external dependency as another QCB specification we can sequence these compilation steps. To resolve an external dependency we need only allocate a segment of our QCB that satisfies the size and shape requirements of the pre-compiled QCB that implements it, and must ensure that it is possible to route between the output of the dependency and our QCB. Using the same arguments for routing we can assert that all IO through an external segment of the QCB must occur on its bottom edge, and that all external segments must have routing ancillae along their bottom edges. Mirroring this argument by placing routing ancillae above the IO registers they may access the internal routing of the extern.

With these constraints we may now consider how to perform the allocation of segments to the QCB to satisfy the DAG. Our QCB is defined as a doubly connected set of edge lists of rectangles (in effect four DAGs, one for each cardinal direction). Elements of the QCB may be allocated (and are hence immutable), split or merged.

We begin with one copy of each external dependency that is required. So long as there exists a QCB specification for each external dependency we place them in decreasing order of width.

After performing a global merge maximising the length of each remaining segment we start from the top left and enumerate each unallocated segment. An external dependency may be placed only if it fits within the unallocated segment, if it is possible to place a route below that segment, and if it is possible to join that route to an already allocated route. If it is not possible to satisfy any of the external dependencies then the compiler halts.

After this first pass for the external dependencies a similar approach is taken with the register placements. Registers are allocated in blocks of height 1 and of arbitrary width, must be routable from either above or below, and that route must be able to be joined with an already allocated route segment to ensure connectivity. If it is not possible to route a given segment then it may be marked as either routing ancillae (if it is connected) or demarked purely for local topological surface code operations (such as rotations).

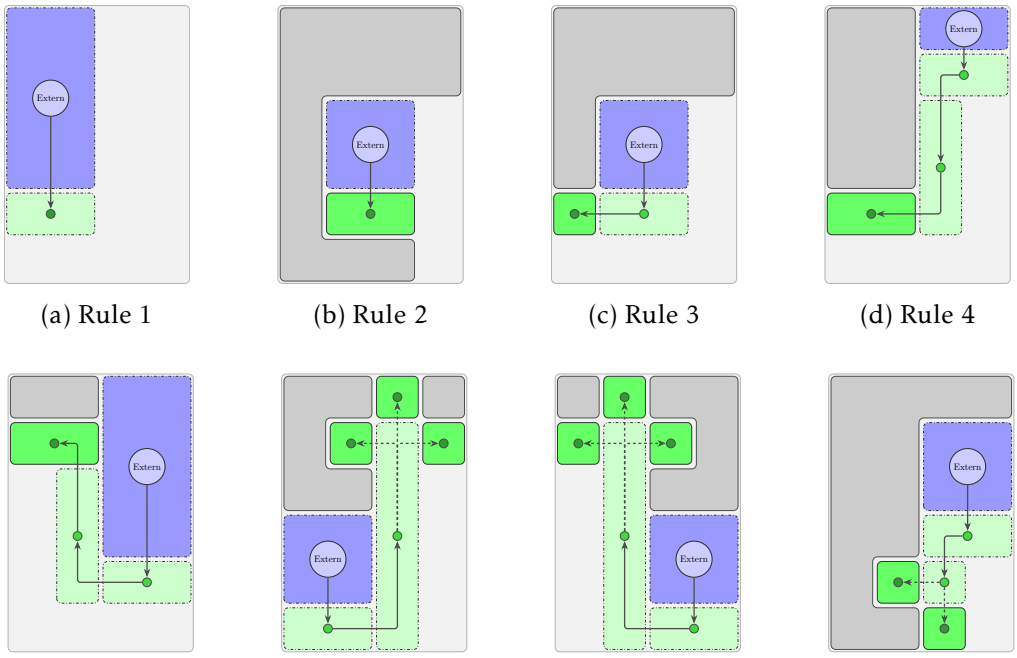
If each external dependency has been placed and enough registers have been allocated to satisfy memory requirements then the initial allocation phase ends.

This placement is performed over a set of bi-directional doubly linked list segments representing non-overlapping rectangular regions of the QCB. Each placement is performed by sequencing split and merge operations between these segments. Each of these segements may be marked as allocated or as unallocated, and if allocated may additionally be marked with what type of QCB segment it has been allocated as: either a register, route, extern or 'local route'. The general approach taken is to allocate from the top-left corner down, and to maintain access to routing elements along this edge. Placement is then concerned with first ensuring that the element fits in the segment, and then in finding an appropriate set of routes to ensure connectivity with the remainder of the QCB.

Allocation is repeated for each extern until all free segments have been attempted. If it is not possible to allocate an extern at this stage then compilation fails.

If the extern does not fit within the height or width of the current segment attempt to merge it with any unallocated neighbours to the right or below until it does fit. If it does not fit then the allocation fails. If the segment fits then we proceed to routing concerns. If it is not possible to place all externs then the compilation fails.

1. If the placement is the top left corner of the QCB: then place an extern segment along with a routing row directly below it. This is the first placement on the QCB.
2. If the QCB is already routed from below, simply add it.
3. If placing a route below would already connect with another routing element, add the extern and the route
4. If the placement is on the top row of the QCB: attempt to place the extern and attempt to place a routing row below it, if the segment to the left of the routing row is free then attempt to place a route up the left hand side of the extern and rejoin with a route.
5. If the placement is on the top row of the QCB: attempt to place the extern and attempt to place a routing row below it, if the segment to the left of the routing row is not free then attempt to place a route down the left hand side of the allocated segment to rejoin with a route.
6. Attempt to route up the right hand side of the extern and connect with any existing route




(a) Rule 1 (b) Rule 2 (c) Rule 3 (d) Rule 4

(e) Rule 5 (f) Rule 6 (g) Rule 7 (h) Rule 8

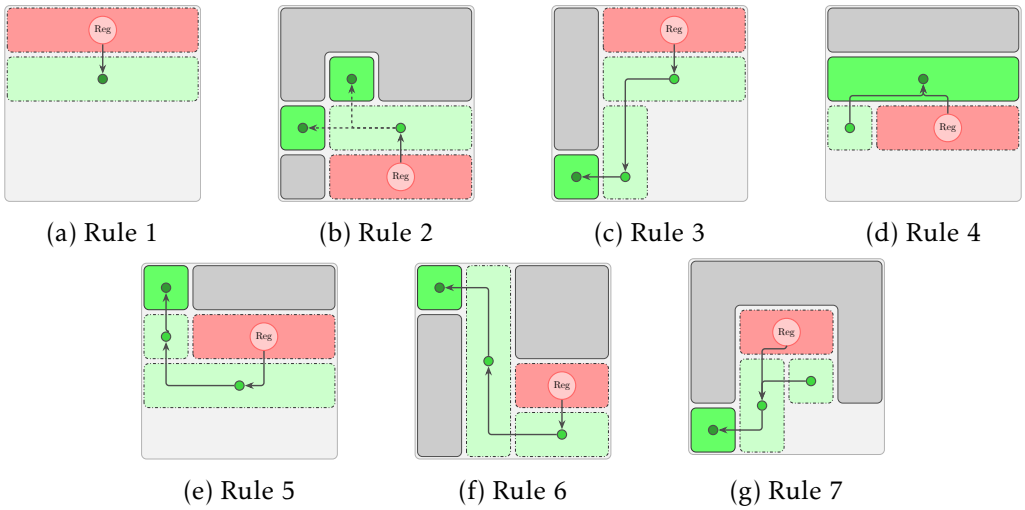
Figure 3.33: Extern placement examples. Elements with a solid border represent the initial state of the QCB, while elements with a dotted border are placed along with the register segment while preserving the constraints in Figure 3.32. Gray regions indicate an arbitrary allocated or unallocated state for the QCB.

7. Attempt to route up the left hand side of the extern and connect with any existing route
8. Attempt to route down from the left of the extern and connect with any existing route
9. Route placement has failed

For a valid route allocation the region must either be free (unallocated) space, or have already been allocated as a route.

Once the external dependencies have been satisfied we next consider our memory constraints. The register allocation process allocates a block of registers on each iteration, and is repeated until enough memory has been allocated to meet the requirements specified by the DAG. Each register block is one high, and unlike the extern may be routed from either above or below.

1. If the placement is the top left corner of the QCB: then place a register segment and place a routing row directly below it. This is the first placement on the QCB.
2. If the placement is on the final row of the QCB, attempt to place a route above it, and enusure that the route connects to an existing route.
3. If the placement is on the top row of the QCB: place the segment and attempt to place a routing row below, and drop down the left hand side of the segment.
4. If all segments above the target are routing segments: split the current segment such that the leftmost element is a route and the remainder are allocated as registers.

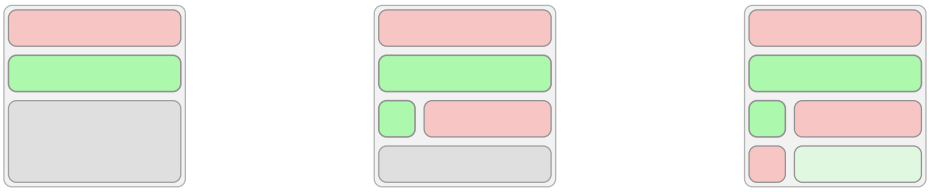



(a) Rule 1 (b) Rule 2 (c) Rule 3 (d) Rule 4

(e) Rule 5 (f) Rule 6 (g) Rule 7

Figure 3.34: Register placement examples. Elements with a solid border represent the initial state of the QCB, while elements with a dotted border are placed along with the register segment while preserving the constraints in Figure 3.32.

5. If the row above is not a routing row, but the top left corner is then place a route on the left hand side, fill the rest as the segment, and place a routing row below.
6. If none of the above are true attempt to drop a route vertically up from the left, place the segment and route below
7. Try to drop a route vertically down from the left, place the segment and route below
8. Route placement has failed



(a) Initial register placement, following register allocation rule 1.

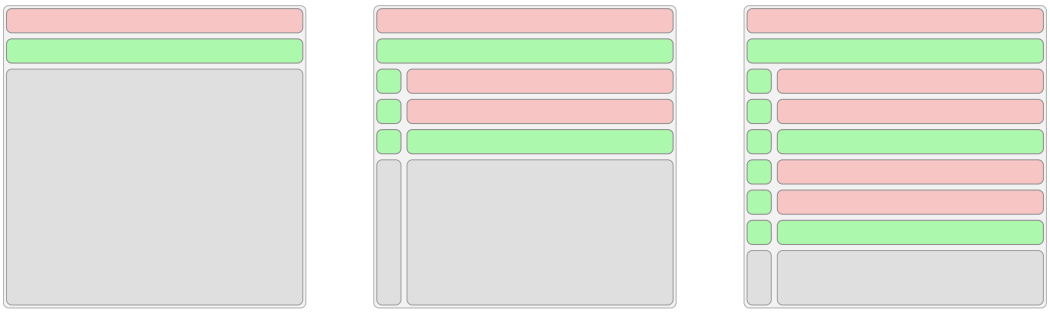
(b) Second register placement, following register allocation rule 4.

(c) Final register placement, rule 4, followed by the placement of a local route in the unallocated space.

Figure 3.35: Register placement for a circuit containing no externs or IO elements. Of particular interest in this case is the light row on the bottom right of the third allocation. As routing between all qubits has already been achieved these 'local' ancillae may only be used for non-routing ancillae operations.

If the QCB requires any IO elements they are placed after the first register or route.

1. Place an IO segment with one register per IO element from the bottom left corner of the QCB.
2. If possible place a route segment above the IO



(a) Initial register placement, following register allocation rule 1.

(b) Second and third register placement, following register allocation rules 4 and 5.

(c) Continued allocation using register allocation rules 4 and 5. Allocation halts at this point as the number of registers allocated to the QCB satisfies the number of registers required by the DAG.

Figure 3.36: Register placement for a circuit containing no externs or IO elements. The initial allocation halts in fig. 3.41c as the required number of registers have been allocated. This placement will be improved upon during the optimisation pass.

3. If possible place a route segment to the right of the IO and join it to the route above the IO

These operations ensure that the IO will eventually be joined to the routing network. Until the join occurs, the routing IO element is not considered a legal anchorpoint for routing. If the placement anywhere above the IO is a register then:

1. Register Rule 1 is not applicable, unless the QCB is of height 3, in which case it places the route that would be used by the IO.
2. Register Rule 2 would trigger a join between the IO and the rest of the routing network.
3. Register Rule 3 through 6 would place a route in connection with the IO's route
4. Extern Rule 1 follows a similar logic to Register Rule 1. If the extern's height is two less than that of the QCB then it lays the route for the IO.
5. Extern Rule 2 is a special case, and may not be used unless the IO is already connected with the rest of the routing network. This can be tracked via a flag.
6. Extern rules 4 through eight would similarly connect the IO route to the remainder of the routing network.

Examples of possible IO placements can be seen in Figure 3.37.

By considering the final placement on the left edge of the QCB and directly above the IO route the set of legal IO connections can be demonstrated.

At the end of this process the QCB should specify enough elements to implement each of the gates in the DAG, but may still have unallocated free patches. We will next attempt to allocate additional routing and extern segments to reduce the runtime of the circuit.

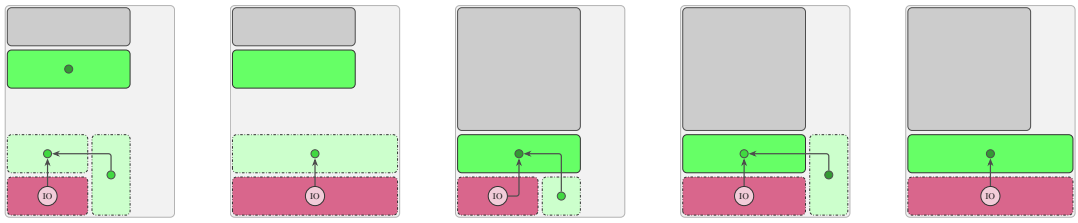
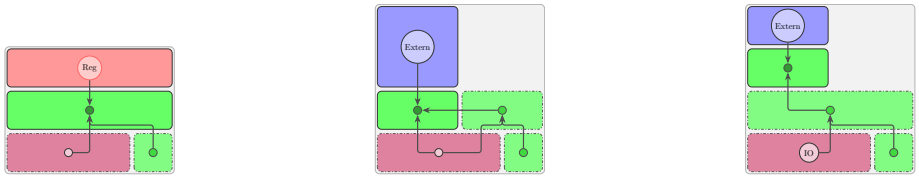


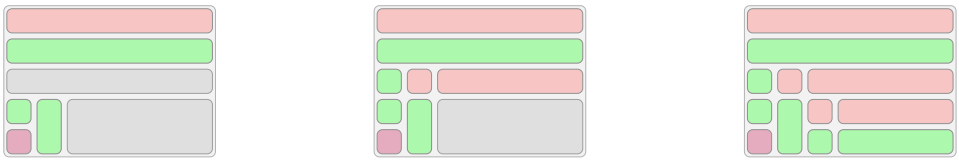

Figure 3.37: IO placement examples. Elements with a solid border represent the initial state of the QCB, while elements with a dotted border are placed along with the IO segment while preserving the constraints in Figure 3.32. The IO must be in contact with the bottom element of the QCB to ensure connectivity with the corresponding route at the bottom of each extern. The IO placement occurs after the first register or extern placement within the current QCB, represented by the grey region.




(a) IO connection on a height 3 QCB

(b) IO from extern rule 1 where the height of the qcb is two more than the height of the extern.

(c) IO from extern rule 1 where the height of the qcb is three more than the height of the extern.



(a) Initial register and IO placement, following register allocation rule 1.

(b) Second register placement, following register allocation rule 4, and connecting the IO to the routing network.

(c) Continued allocation using register allocation rule 5. Allocation halts at this point as the number of registers allocated to the QCB satisfies the number of registers required by the DAG.

Figure 3.39: Register placement for the T factory circuit shown in Figure 3.27 containing a single IO output for the $|T\rangle$ state. This demonstrates the join between the routing network from the allocations and the routing ancillae placed around the IO pins. This allocation halts as a sufficient number of registers have been allocated to satisfy the DAG, and as all segments have been allocated. If this condition was not met and all segments had been allocated, compilation would fail.

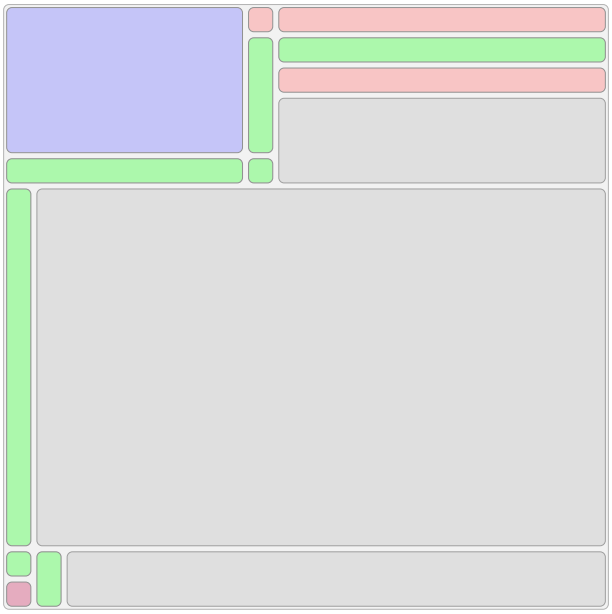
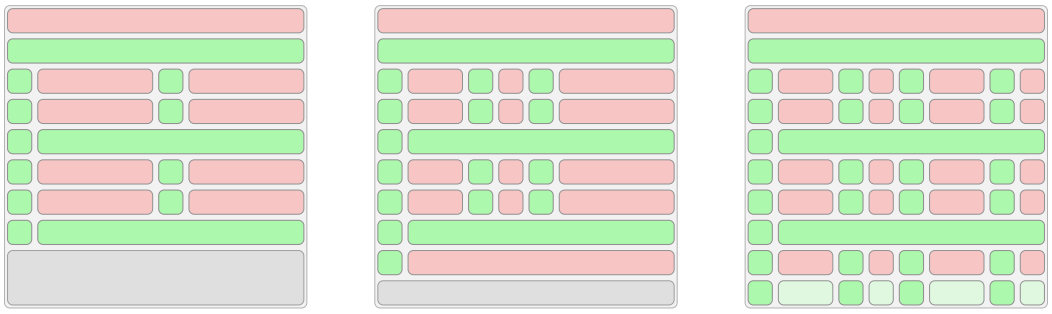


Figure 3.40: Initial QCB allocation for a nested T factory circuit. The circuit being nested shown in Figure 3.27 where each $T^\dagger$ gate injects a dependency onto an instance of the T factory that is being allocated in Figure 3.39. This particular instance of a QCB contains both an external dependency, and an IO output. Of interest in this figure is the demonstration that if the IO was not connected during the initial allocation that routing channel is dropped up the left hand side of the QCB, ensuring that the initial allocation represents a valid QCB description.



(a) Split existing register blocks to add more ancillae

(b) Insufficient registers after a second split, add another register block.

(c) The final allocation, with local routes speculative placed on the bottom.

Figure 3.41: Optimising the placement of additional registers for a circuit containing no externs or IO elements. This is the same circuit as Figure 3.36, and follows on from that initial allocation. The bottom row contains a number of 'route local' segments that cannot satisfy any of the register allocation rules. A subsequent pass will determine that they are connected to existing routing segments, and will promote them to routes.

### 3.4.4 Optimising the QCB Allocation

From this initial allocation we now wish to try to allocate any remaining resources to improve the runtime of the circuit. This may be achieved either through allocating more external segments to parallelise any operations that rely on independent instances of these dependencies, or by introducing redundant paths into the network of routing ancillae to alleviate any bottlenecks where independent operations depend on the same subset of routing ancillae.

Using the DAG depth as a heuristic approximation of the runtime of the circuit we may order our attempts by the hypothetical reduction in runtime achieved by placing an additional externally defined segment, or an additional redundant path. If the greatest reduction in runtime would be achieved by placing an externally defined segment then that may be attempted using the same process described above. Instead if another routing channel would improve the runtime then we may attempt a vertical split across all register elements in a column, replacing one of their elements with a route segment. If this would reduce the number of registers below the memory requirements of the circuit then more register blocks may be allocated using the same process decribed above. If it is not possible to allocate enough memory to perform the split then the operation is rolled back. If none of these placements are legal, or if no possible placement would lead to an improvement in the runtime then this stage of optimisation ends.

The last step is an attempt at flood filling the remaining unallocated segments. Using the register placement algorithm register segments are allocated across the remainder of the QCB. For each section that is unallocated:

1. If no neighbours are routes then flag the region as a 'Local Route', this region may only be used for local ancillae operations.
2. If the segment above or below is a routing segment then fill it with alternating route and register patches

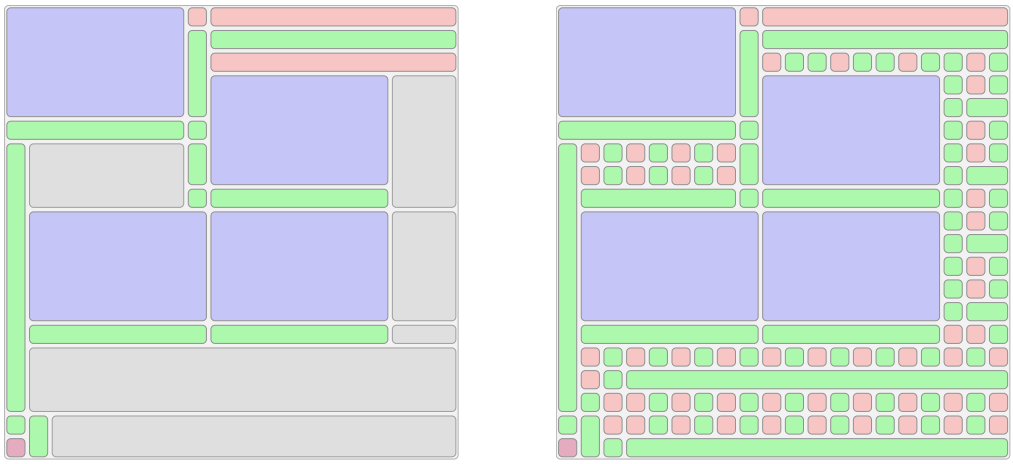


Figure 3.42: Optimising remaining unallocated space for a nested T factory circuit. The depth of the DAG conditioned on the number of extern regions and the non-local gate semaphore. For each step in the optimisation phase the DAG depth determines what the next placement is, if a legal placement exists. In this case this case after placing four extern regions with a size determined by the T factory QCB in Figure 3.39

3. Otherwise set it as a set of routing patches (routability is guaranteed by not meeting the first condition).

Lastly a final pass is performed on all of the local routing segments to determine if any have been connected by the placement of subsequent routes.If they have been connected to the routing network then they are re-typed as routes. At this point all patches on the QCB have been allocated.

Our next consideration is mapping named registers from the circuit to patches on the QCB that have been marked as registers.

### 3.4.5 Register Mapping

As optimal mapping algorithms are NP hard [86] we will have to opt for another heuristic. Our goal in this instance is to attempt to maximise the distance between subsequent allocations on the QCB. To do this we begin by turning our QCB into a connected graph. All registers and externally defined segments are connected to a single routing element, all routing elements are connected to a maximum of four neighbours.

From this graph representation we construct a spanning tree where each of the leaves of the tree is a register. To construct the tree we begin with the set of leaves. For each leaf we merge all adjacent routing verticies with the current leaf and increase its score by the number of routing verticies that were merged. If two leaves would merge the same routing vertex we instead create a parent node in the tree and evenly distribute scores to its children. As our graph is completely connected this process produces a single tree. The scores on each leaf now represent the number of local routing paths around that node, while the distance between two leaves in the tree approximates the physical distance between those nodes on the graph.

Allocating registers from the circuit is then a matter of ordering the nodes by their score (par-

ents take the maximum score of their child nodes) and allocating it to the child with the highest score recursively until it reaches a leaf node. As one register segment may contain multiple physical registers this process is performed as a round robin; a child may not have $n+1$ elements allocated to it until all other children have had at least $n$ elements allocated or only represent a collection of $< n$ register elements. As subsequent allocations are now guaranteed to be allocated to different branches of the tree, this represents a physical separation. Using the DAG we may calculate which pairs of registers have the most simultaneous non-local operations and assign each pair a 'conjestion' score. Alloction is then performed in order of the element with the highest conjestion score, followed by each node it is conjested with. This heuristic attempts to then minimise the maximum conjestion between elements in the circuit, and by extension reduce the number of shared dependent routing ancillae.

As each register segment in the QCB is connected to the routing network, we now replace any unallocated registers with route segments.

Each of the register segments of our QCB may contain multiple surface code patches. Given the allocation of a named register to this register segment we still encounter a decision problem with regards to which of the physical patches the register should be assigned to. To reduce the number of rotation operations if possible we wish to expose as many edges of each register as possible. Using a similar argument to the round robin allocation, we also wish to maximise the distance between sequential allocations.

For this purpose our register mapper allocates physical patches in a manner that would split the largest free space in the register as shown in Figure 3.44.

By register allocation rules 3 and 4 (Figure 3.41) register blocks may be allocated in matching pairs. The deterministic ordering of the register allocations within these matched pairs create additional vertical routing paths.

This is achieved by iterating over the odd coefficients of $\alpha 2^{-i}$ up to $2^i - 1$ with a modular offset between each coefficient equal to $2^{-i+1}+2-i$ between each choice of $\alpha$. When all odd coefficients are exhausted $n$ is increased. As the register's length may not be a power of two, each value is multiplied by the length of the register, and floored to an integer. This operation gives rise the probability of a collision, in which case the process is repeated. This achieves a worst case of $O(\frac{n}{2})$ on an insertion with collisions only occurring on the final $n - 2^{\lfloor \log_2(n) \rfloor}$ allocations. Given the resolution of the function the the total number of collisions is upper bounded by $n$ over all $n$ allocations. This produces a final bound of $O(2n)$ function evaluations to place $n$ elements in a register, and a lower bound of $O(n)$ evaluations, while maintaining free elements adjacent to each placement, and a distance of at least $\frac{n}{4}$ between subsequent placements for the first $2^{\lfloor \log_2(n) \rfloor}$ allocations.

### 3.4.6 Extern Mapping

Unlike our registers externs may be reset and re-used, we provide three different extern mapping algorithms.

1. A static allocation using the DAG ordering that simply waits for the physical memory location to be free before continuing the program. The static allocator acts as a lookup table that binds physical externs to gates in the DAG and is defined when parsing the

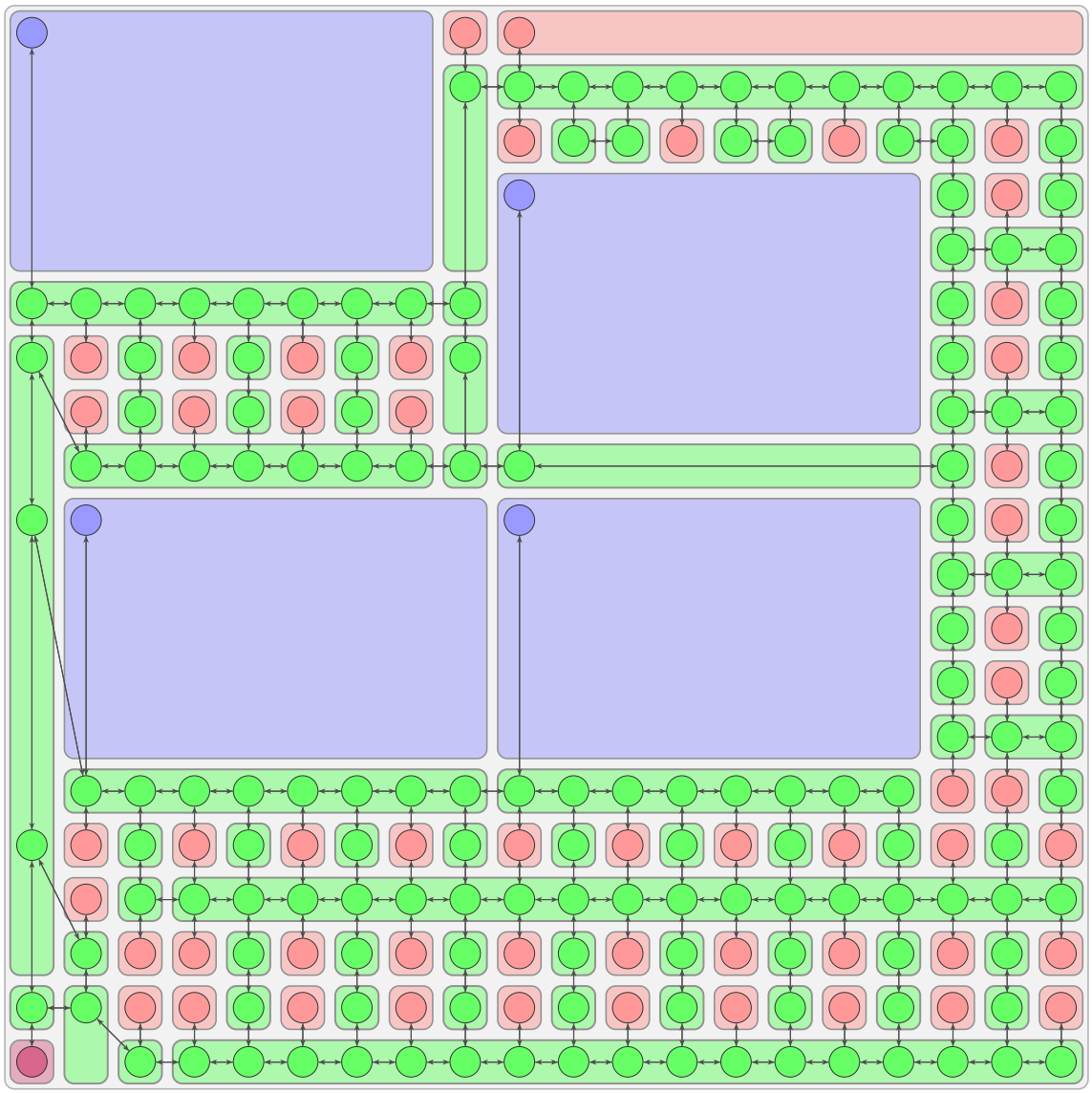
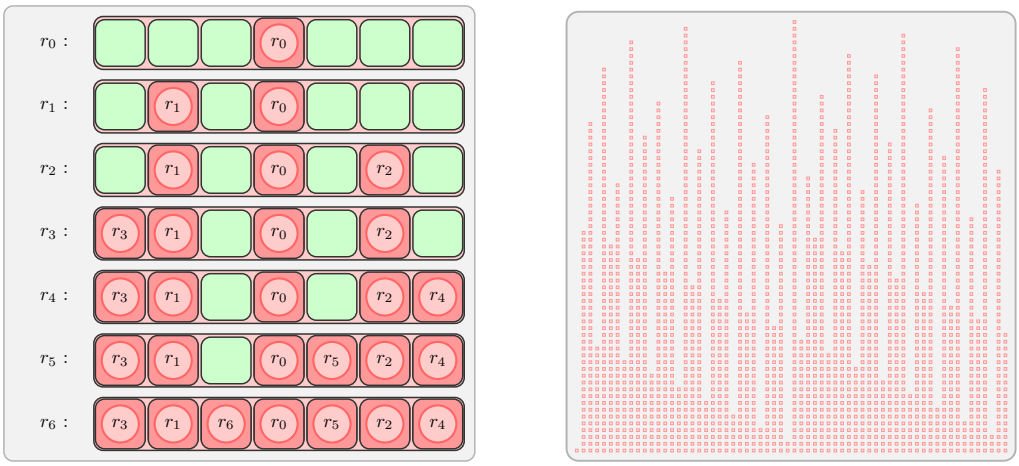


Figure 3.43: Graph of routing connectivities with for a nested T factory. This graph will be remapped to a tree where externs, IO and reg nodes form the leaves. Allocation will then take place as a round robin over the branches of the tree starting from the root.




(a) 7 qubit register allocation ordering (b) 64 qubit register allocation ordering

Figure 3.44: Example mappings for a register block. Registers are allocated to maximise the number of registers with a horizontal adjacent routing node for vertical routing and ancillae operations, while also seeking to maximise the distance between each successive allocation.

DAG as described in Section 3.4.3.

2. A dynamic allocation that binds each extern to a typed set of memory slots, such that each such slot my only implement externs of that type. This allocation is controlled by a priority queue with membership determined during routing.
3. Lastly we relax the typing assumption and instead assert that any extern dependency may be satisfied by any extern segment on the QCB that has a sufficient height and width. These can be ordered to prioritise the smallest avaliable extern that meets the criteria, or the extern that has been idle for the longest (which we will see later provides speedups to factories).

With the ability to assign QCB patches to registers, externs and IO elements from the DAG we can now consider operations in terms of the borders of these segments and in terms of shared ancillae resources for routing and other gates.

### 3.4.7 Patch Orientation

As the previous register, IO and extern allocation methods does not guarantee that all edges of each patch are always avaliable for routing we must consider injecting rotations into our circuit to satisfy this dependency.

As routing to another exposed edge incurs no time penalty when compared to injecting a rotation, we will always prefer routing where possible. Depending on the previous allocation and operations acting on the gate, it may be possible to join the rotation into a previous operation. For example during initialisation the register may be created in an alternative orientation. If the above two methods fail then we are forced to inject a rotation requiring a number of local routing ancillae. If there are any adjacent ancillae that are not connected to the routing network then these are selected preferentially.

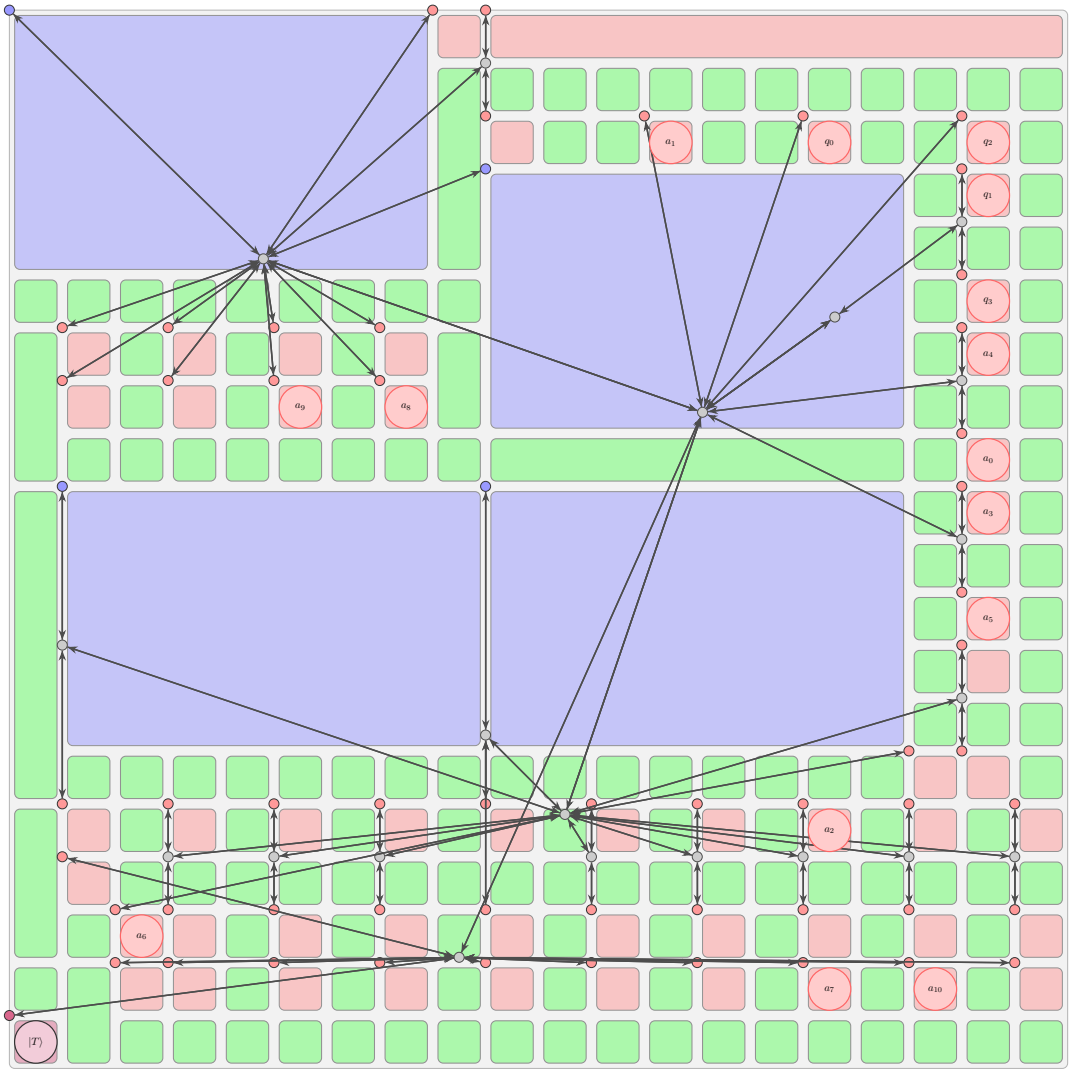



Figure 3.45: Mapping tree for a nested T factory. This follows from the segment allocation show in figure Figure 3.43. The unused register segments will be remapped to routing ancillae.

Rotations are unary operations, and as a result it is not possible for a rotation gate to trigger the resolution of a tagged extern. Either the rotation is dependent on a tagged unary gate, which has already propagated its tag to its sole dependent gate, or the rotation depends on a non-unary gate which will resolve the tag. As a result it is not necessary to propagate extern tags to injected rotation gates.

As externs have not yet been mapped to physical memory locations it is not possible to track the orientation of IO elements. Instead the orientation of the IO elements of the extern may be defined by the gates interacting with it, and rotations may be performed using the internal route directly above each IO element. Rotations on IO elements of externs may then be treated as a unary operation without the need for any ancillae in the current scope, are guaranteed to be parallelisable if using Litinski rotations (as each element of the IO has its own correponding routing ancillae), and may be performed in ø3. The default orientation of IO elements of an extern may be set by the $X$ and $Z$ components of the gate of the extern, though this behaviour may be overridden by the implementation of the extern.

### 3.4.8 Routing

Lastly we may consider routing each of the gates. The core of the routing algorithm is an A* search between the qubits in each non-local gate. To reduce the search space, and as we attempted to improve our routing by increasing the number of vertical channels, we bias the A* search by slightly prioritising vertical paths over horizontal ones.

Each gate specifies a number of surface code cycles over which it locks its dependent registers and mutually excludes its routing ancillae. The runtime of the circuit may then be determined by sequencing these route operations and leaving any mutually excluded ancillae by other active gates in a locked state. Gates are scheduled for routing based on how urgently their dependencies need to be satisfied, and on how long they have been waiting to be scheduled. Gates additionally specify which boundary of a register is required and routes are anchored from that boundary.

Using our previous tags on DAG nodes we can encounter and inject factory dependencies. Similar to the edge disjoint path tracking each surface code patch tracks which cycle it was last used in, such that factories may be back-dated to the first free cycle. This is a greedy approach that only requires tracking the last cycle in which a given surface code patch was used.

A similar constraint to our eager factory allocation problem is the eager allocation of external dependencies. If an externally defined gate depends on both an operation deep in the DAG, and a shallow operation (for example in a QRAM circuit), then there is a chance that it will be prematurely allocated a region of memory. If there are insufficient memory regions this could lead to a deadlock situation. The approach we have taken is to associate each extern gate with a barrier. The barrier blocks until all operations preceding that extern in the tree that do not directly depend on the extern have completed.

Unlike externally defined factories, externs with inputs are not back weighted. A possible extension of this approach is detecting how many cycles of an extern may be pre-computed and including that in the routing. For example if an extern contains factories, then the extern may be operated for $n$ cycles to 'pre-warm' those factories. However this would involve recalculat-

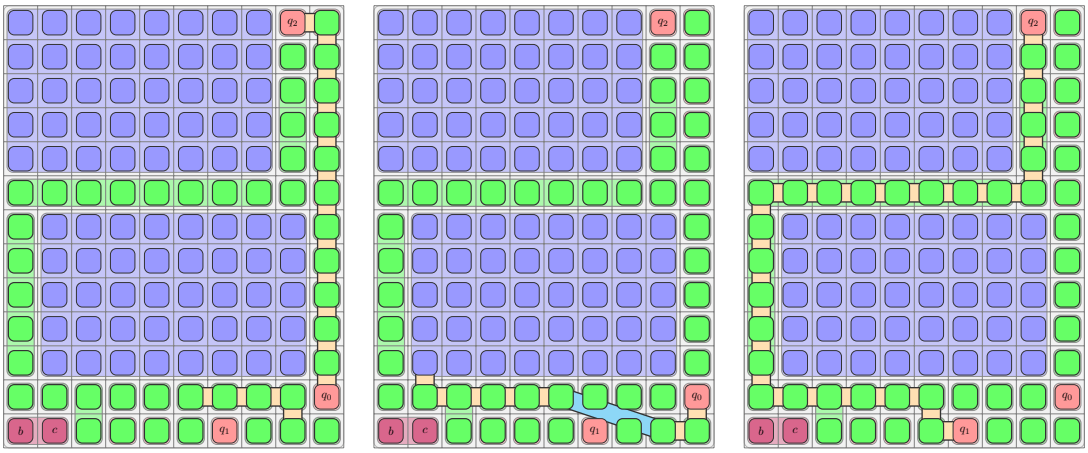



Figure 3.46: Sample Routes from a circuit performing a sequence of Toffoli gates. The blue regions indicate the magic state factories implemented as externs. An set of IO pins exists on the bottom left of the QCB. The routes are indicated by the 'bridges' between adjacent patches. (Left) a two qubit CNOT operation between $q_2$ and $q_0$ while a edge disjoint route is prepared between two routing ancillae. (Middle) A route making use of the split $|+\rangle$ state.to perform a $T$ gate on $q_0$. (Right) another two qubit operation between $q_1$ and $q_2$.

ing the runtime of a pre-compiled QCB.

Once a legal route has been found we attempt to greedily implement edge disjoint routing [14]. The routing is performed by preparing a Bell state over the switch using a sequence of merges, the intermediate Patches may then be measured without impacting the state, and a non-local operation may then be performed while treating these endpoints as if the disjoint route was still in tact. To decide where to implement disjoint routes we treat route patches that are adjacent to at least two other route patches as a set of potential switches. If a route would pass through a switch and exactly two of its adjacent routing patches then it is a candidate for replacement with a disjoint path in an earlier cycle, freeing up the switch for later operations. Teleports are implemented with a greedy ASAP scheduler that checks the last cycle in which all the route patches over the switch were last used. If two adjacent switches implement disjoint routes for the same route in the same cycle then they are merged into a single longer distance disjoint route.

An example of this routing protocol can be seen in Figure 3.46.

# Chapter 4

# Reversible QCPU

In the usual model of quantum computation a quantum device plays a subordinate role to a non-quantum ('classical') processor. This classical processor controls each operation on the quantum device. Depending on the architecture these classical control devices may manipulate sequences of pulses to implement single qubit rotations, initialise and readout states, and couple qubits for two qubit gates.

The subordination of the quantum computer to the classical control unit is emphasised by the computationally limited nature of quantum circuits. As quantum circuits are fixed-size directed acyclic graphs of quantum operations. This fixed maximum depth constraint arises in part from the long readout times for modern NISQ devices, making it infeasible to condition operations on the outcomes of other quantum states. This leads to NISQ resembing a system of 'classical control' on 'quantum data'.

One naturally wonders what would happen if the control mechanism were itself a quantum device. In this chapter we demonstrate a fully quantum controlled architecture with analogous programmatic primitives to classical computing. Of particular interest is the ability to recreate classical control flow via entangling the state of a quantum control unit with the state of the registers on which it acts.

A major limiting factor on this approach is that quantum systems are reversible, and hence jump based control must be implemented reversibly. One approach to this is to ensure that each jump is bijectively associated with a reverse jump [135] at the landing point, however the total size of the number of instructions in the program is now bound to the depth of the program. Instead we make use of a history buffer, which tracks the execution of instructions. As the computational device is reversible, it then becomes possible to periodically flush the history buffer via uncomputation, after saving a selection of data to a protected section of memory.

By implementing quantum control flow we recover a number of compositional features of quantum algorithms. For example, the linear combination of unitaries [69] quantum programming construct corresponds to a quantum 'switch' expression with the choice of executed unitary operator depending on the condition of the switch. As another example, because non-relative jumps enable function calls, we can use the reversibility of quantum operations to execute the code of the function in reverse order. Such reverse function calls are often used

to safely (i.e. noiselessly) erase quantum registers that are no longer needed for a computation; the technique is called *uncomputation*. We even combine these ideas to perform a linear combination of function computation and uncomputation, which extends into a 'repeat until success' gadget that is implicit to a number of quantum algorithms [106, 17].

We emphasise that our proposed architecture does not have increased computational power or performance over the current paradigm of hand-assembled quantum circuits. Instead, we believe the technical value of our architecture arises from the ease with which one can express quantum algorithmic techniques like those given above. Such a layer of abstraction atop more realistic models of quantum hardware could prove useful in the development of high-level quantum programming languages.

In the remainder of this chapter, we outline the contents of this QCPU architecture, and demonstrate a range of computational operations.

This work relates to prior efforts in both quantum compilation and quantum algorithm design, along with the development of QRAM[57] and QROM[9]. The focus of this work is for near term fault tolerant or noise suppressed devices and the costing of the implementation of quantum algorithms in this regime[11].

Previous work in developing such an architecture has either attempted its implementation on a small number of qubits[93], and has been concerned with implementing QRAM and runtime zeroing of qubits, or has simplified the scheme using hybridized classical control units directly managing the quantum system at runtime[18]. We differ in our proposal of a fully quantum control unit, and our focus on acting as a compilation target, rather than explicit hardware implementation details. With that said, in Chapter 5 we provide a costing for an implementation of this QCPU on the surface code. From the description of this history buffer element alone, this implies the operation of QRAM in each cycle, and hence is more of an exercise in the operation of the compiler, rather than in producing a viable implementation of a quantum architecture.

## 4.1 Background

In this section we will provide a brief background relating to some useful operations, quantum memories and quantum control flow.

### 4.1.1 Reversible Quantum Programs

A constraint on quantum operations is that they are invertible. It then follows that the inverse of a sequence of quantum operations (indeed a schedule of such operations, or a programme) may be found by taking the inverse of each operation, and reversing the order. Previous attempts to build quantum Turing machines have encountered difficulties with reversible jump operations, and the implications for invertible control flow and by extension functions and programs.

The first attempt at constructing a quantum Turing machine envisaged a simple tape mechanism with MOV, read and write operations. Most interestingly it proposed a sequence of HALT operations with a persistent quantum memory[82]. Programs would be performed, a HALT

would then projectively determine the success of a given program, and then the next program would begin acting over the same memory.

This architecture dedicates a negative portion of the tape that tracks the updates to the program header between operations, and hence may be used to read back through and uncompute the program if the operations are self-inverse. The authors do not discuss the implementation of uncomputation and leave it to the programmer to manage. This leaves an interesting divergence between uncomputation and the computational inverse.

**Algorithm 1** Decrement Loop

1: Prepare register $A$ with value $i$
2: **while** $\text{Reg}_A \neq 0$ **do**
3: Decrement $\text{Reg}_A$

The description raises an interesting problem. In Algorithm 1 we can construct a sequence where each operation is reversible, however the program itself is not. The algorithm takes an arbitrary non-zero integer, and decrements the value until it reaches zero before continuing. As the number of instructions executed depends on the initial state we reach Equation (4.1).

$$\begin{aligned} \text{Decrement Loop}(x > 1) &\to 0 \\ \text{Decrement Loop}^{-1}(0) &\to 1 \\ \text{Decrement Loop}^{-1}(\text{Decrement Loop}(x > 1)) &\neq x \end{aligned} \tag{4.1}$$

While the sequence of operations is invertible and reversible, this program is demonstrably not. The program dictates a sequence of $n$ iterations of the decrement operator, resulting in a non-injective program, that specifies a set of individually invertible bijective sequences of operations. A core component of this chapter will be designing a quantum system that admits invertible and failing that uncomputable functions as part of regular control flow.

The circuit model of quantum computing avoids this problem by requiring that all circuits contain a constant number of instructions determined at compile time. Any quantum circuit may be composed into a single quantum operation. The inverse of a program in the circuit model is then found by taking the complex conjugate of each gate, and then reversing the order of the gates. This is notably distinct from the program Algorithm 1, where the length of the schedule of operations depends on the input state.

More recent extensions to OpenQASM have dabbled in this problem by asserting that all iterations are conditioned on integers, and conditional operations are conditioned on measurement outcomes. This first constraint ensures that the circuit model cannot implement an arbitrary depth program, while the measurement conditioned operations do not yet have an associated implementation on current NISQ devices.

Yuan [135], looks instead at constraining all jumps such that each jump has a singlular source. The concern arises again that if any sequence of quantum operations can be reduced to a map that performs $|a\rangle \to |\psi\rangle$ and $|b\rangle \to |\psi\rangle$ then a non-unitary operation has occurred. In this case the problem is resolved by asserting that all jump operations are bijectively associated with an 'unjump' operation. This bijective association reconstructs local structure, but discards the

ability to call a block of code from multiple points in the program, effectively discarding the possibility of implementing a 'function' call and return structure.

**Algorithm 2** Decrement Loop

```
1: Prepare Reg_A with value i
2: Reg_B : 0 ↔ L2
3: L1
4: Reg_B : 0 ↔ L2                                   ▷ Reset Reg_B
5: Decrement Reg_A
6: if Reg_A ≠ 0 then
7:     Prepare Reg_B with value L1
8:     L2 Swap Reg_B with the instruction pointer   ▷ Jump to L1, Reg_B contains L2
9: Continue
```

In our approach we incorporate a history buffer as a form of block encoding [28], such that local non-unitary operations are encoded in a larger unitary operator. To extend the earlier analogy, by block encoding $|a\rangle|0\rangle \rightarrow |\psi\rangle|a\rangle$ and $|b\rangle|0\rangle \rightarrow |\psi\rangle|b\rangle$ are admissibile unitary and reversible operations.

One complexity with this approach is that the contents of the history buffer are entangled with the working registers. We attempt to limit this entanglement between the history buffer and our working memory via uncomputation. The history buffer is treated as a source of instructions rather than a sink for performed instructions. Instructions are then inverted and the history of the program is consumed. A set of 'forwards computation' only operations are implemented to preserve the output of functions from this uncomputation. When the program halts the history buffer should contain a set of initial state preparations, and uncomputation traps. While the working memory should contain the output of the computation.

In this chapter we will construct a quantum CPU design, such that an arbitrary program may be implemented by loading it into a quantum memory, and letting the system run until it reaches a halt condition.We will provide additional specifications relevant to implementing an eight bit quantum logic unit, and in later chapters will benchmark this system against a surface code architecture using the compiler from Chapter 3.

### 4.1.2 QRAM Interrupts and Parallel QRAM

A useful feature for our QRAM is the ability for particular address values to serve as 'interrupts'. That is that if a particular address is loaded the QRAM lookup returns the input. This effectively turns the QRAM into a controlled QRAM conditioned on the input address.

By conditioning a SWAP operation between the readout and a targeted address on that address we first swap the target into QRAM, then query QRAM at that address and swap the target back out. The result is a QRAM which operates normally for all addresses except the 'poisoned' address. An implementation of this interrupt can be seen in Figure 4.1.

Modification for the bucket brigade QRAM is permitting parallel non-intersecting queries by enforcing a final 1 bit in the address. As the control bit for the bucket brigade must be prop-

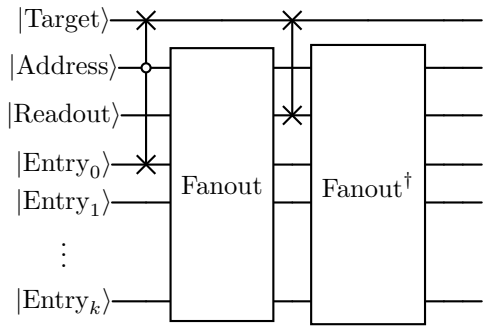



Figure 4.1: Poisoned QRAM entry. If the query address matches the entry then no operation occurs. This is a useful gadget for idling the history buffer for a cycle when swapping between computation and uncomputation, while turning the QRAM into a controlled-QRAM, with the loss of one entry

agated through the routing network, for a given query string ending in a 1 only a single final control bit will be flipped. The first round of the readout may then skip the 0 conditioned swap. This adds an additional round of routing, but now ensures that if two queries do not contain any overlapping addresses that they may be performed simultaneously.

Without this additional round all non-queried addresses ending in 0 would be swapped into the routing network on the first round of lookups. With the modification only marked states are swapped out of the memory region. As this is a fixed modification to the address it neither expands the address space and can be stored as an ancillae in the routing network.

## 4.2 The QCPU Cycle

In this section we will specify the components and operation of the QCPU cycle.

We define the size of an instruction opcode as a fixed width $r$ qubits, and a register as $n$ qubits, in the QCPU there are $k$ processor registers.

### 4.2.1 Memory Architecture

To implement our QCPU architecture we make use of four different QRAM elements of various types.

#### Main Memory

The simplest element is the working memory, which we implement as a $2^n$ element SWAP-QRAQM QRAM, where each element contains an $n$ qubit register. This main memory implements two stack structures, one from address 0x0 up, which stores the working memory, and one from address `0xff...` down, that stores garbage ancillae. The top of each of these stacks is stored at a reserved address.

These two stacks then necessitate separate LOAD and STORE operations, which load the current top of the stack and may access elements via offset from this location. For our working

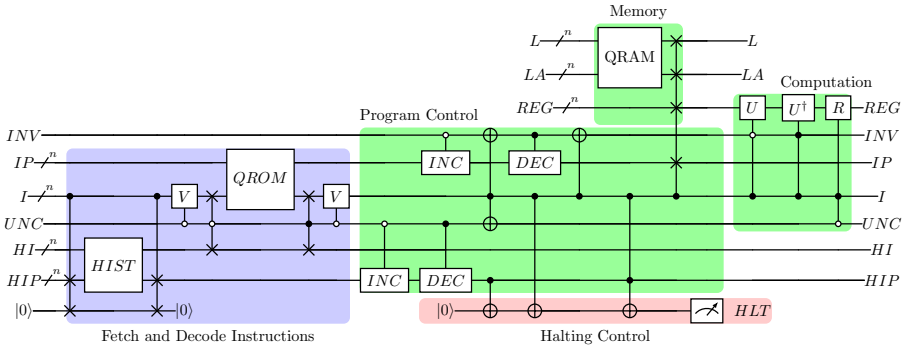



Figure 4.2: Sample Quantum CPU cycle architecture. $REG$ is a collection of general purpose registers. $L$ and $LA$ predicate QRAM queries. $IP$ controls QROM queries to the instruction register $I$, which acts in a controlled fashion over a series of pre-defined operation codes to conditionally implement the other operations within the cycle. $INV$ controls whether an operation or its inverse is applied, $HI$ and $HIP$ load registers into the QROM query, while $UNC$ controls whether operations are being computed or uncomputed.

memory this gives rise to the ability to create stack frames.

This main memory is accessed via an address register $LA$ and a target loading register $L$.

### Program Code

The next element is a memory that stores the current set of operation codes for the given program. These codes are prepared at compile time and the region may be implemented as either a CNOT-QRACM or a CNOT-QRAQM, to implement a read-only data structure. As with the main memory, the program code may be queries with an instruction pointer register $IP$, which writes out to the instruction register $I$. The value in the $IP$ register is then incremented. This program code may contain up to $2^n$ elements, where each element is an $r$ bit or $r$ qubit operation code.

By storing the instructions as part of a quantum addressable memory, so long as we can independently manipulate the instruction pointer as part of the operations of the program we have broken the linear execution of the classical co-processor quantum circuit model.

To demonstrate the distinction clearly: as instruction pointer and QRAM stored instructions permit lookups in superposition, we may execute instructions on the device in superposition. Or rather we have so far demonstrated the ability to load instructions in superposition, next we must map each of these instruction codes to a unitary operation.

### Logic Unit

The logic unit is a CNOT-QRAQM that stores the translation of each of the operation codes into the action over the registers. The address register $I$ is a $r-1$ qubit register, while the output $kn$ qubits act over the processor registers.

The logic unit does not perform program-control operations.

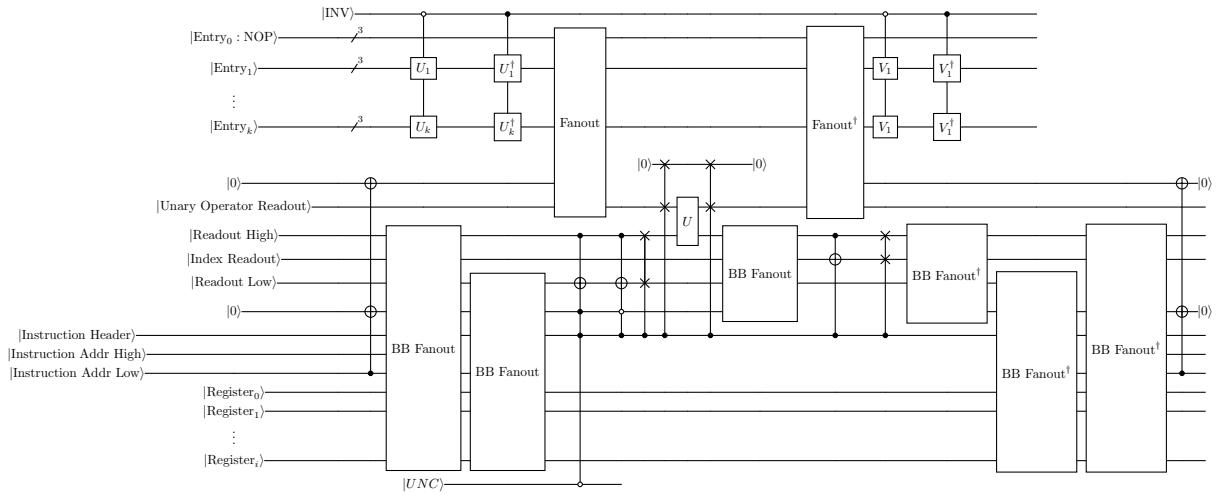



Figure 4.3: Example logic unit architecture. The instruction is separated into a header and two sets of addresses. The header dictates what the addresses are to be used for. The high address acts as a query over the registers to load a target register for the operation. The low address then acts either to load a second register for a binary operation, a unitary operator for a unary operation, or an address for a qubit within the first register for an intra-register operation. To achieve simultaneous lookups on the same set of memory banks two independent bucket brigade QRAM routing networks are used, and the instructions constrained such that the high and low addresses may not query the same element for a inter-register operation. This constraint may be applied at compile time when writing instructions to the program code. The action of the unitary operation $U$ is shown in Figure 4.4, while a minimal replacement for the instruction loading can be seen in Figure 4.5.

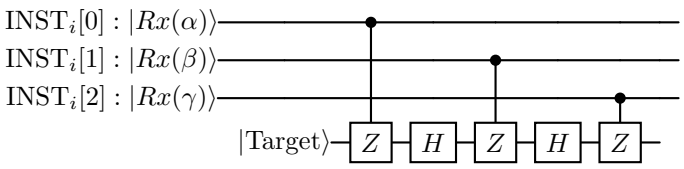



Figure 4.4: Example unitary operation on a target register in the logic unit. The instruction qubits are loaded from QRAM as seen in Figure 4.3. By setting $\alpha, \beta$ and $\gamma$ appropriately an arbitrary unitary operation may be applied to the target.

As a sample implementation for a logic unit we separate the each instruction into three components, a header, a high address and a low address. Depending on the operation these high and low addresses act to index either an instruction, a register or a qubit within a register.

If the header specifies an instruction we query the registers using the high address qubits, while the low address qubits are used as the query condition for which instruction should be executed. By performing a sequence of controlled rotation operations an arbitrary $U$ can be constructed on the target qubit. One construction for this arbitrary unitary can be seen in Figure 4.4. The sequence of phase operations combined with Hadamard operations can construct an arbitrary unitary up to a global phase [101]. To map each potential instruction to an arbitrary unitary operation we may construct a sequence of controlled operations that builds each of the phase gates. Simplifying this approach each set of controls may be set in a queryable table where the addresses are the instructions. Querying the instruction table produces three qubits, each of which controls the angle of rotation for one of the phases.

The control qubits for each of the sequence of control operations are set by the QRAQM lookup. The readout of this QRAM lookup is then the $CU$ operation. After the operation is performed the fanout is inverted and any cleanup that is necessary is performed.

While the unitary operations provide a basis for arbitrary single qubit rotations, support is still needed for multi-qubit operations. These multi-qubit operations may be between two qubits within the same register, or between two distinct registers. Conditioned on the instruction header we can use the low address as the query condition in another QRAM lookup over the addressable registers. As a fanout and swap QRAQM would permute the order of each of the registers, we instead use two bucket brigade QRAMs with independent routing networks. By guaranteeing that no multi-register instruction can target the same register twice we can ensure that these two QRAM queries over the same data may be performed in parallel. Once both registers are loaded we can perform either a CNOT between the first qubit of each register, or we may swap the contents of both registers, the operation performed is determined by the instruction header. The bucket brigade fanouts are then inverted and both registers are returned to their respective locations.

Finally for intra-register operations we can again use the instruction header and the low address bit. This time the low address bit is used to query the address of a qubit within the target register. A CNOT is then performed between the zeroth qubit in the register and the target of the query.

As will be discussed in more detail in Section 4.2.2, the $|INV\rangle$ register controls whether an

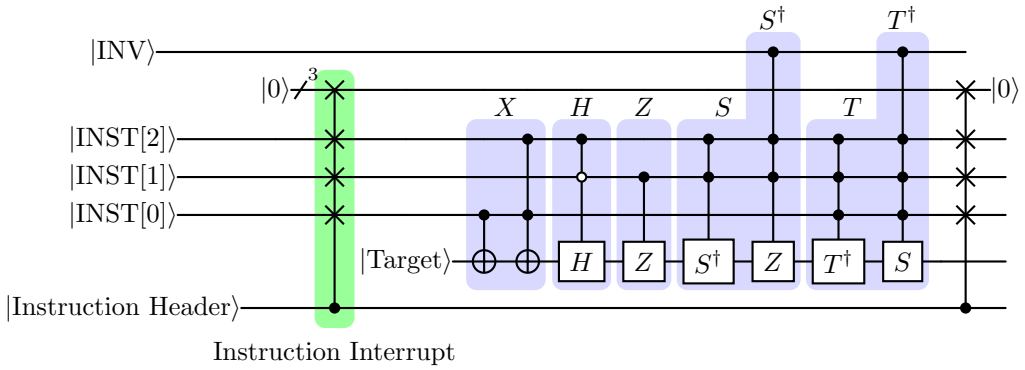



Figure 4.5: A more direct minimal unitary logic unit. This one eschews the QRAM lookups for arbitrary operations in favour of a fixed set of instructions: Hadamard, Phase, T, X and Z. The X and Z gates may be composed to produce an XZ in a single instruction.

| Instruction | 0 | 1 | 2 | INV | Operations | Result |
|---|---|---|---|---|---|---|
| NOP | 0 | 0 | 0 | * | $I$ | $I$ |
| $X$ | 1 | 0 | 0 | * | $X$ | $X$ |
| $Z$ | 0 | 1 | 0 | * | $Z$ | $Z$ |
| $XZ$ | 1 | 1 | 0 | * | $XZ$ | $XZ$ |
| Hadamard | 1 | 0 | 1 | * | $XXH$ | $H$ |
| Hadamard | 1 | 0 | 0 | * | $H$ | $H$ |
| Phase | 0 | 1 | 1 | 0 | $ZS^\dagger$ | $S$ |
| Inv Phase | 0 | 1 | 1 | 1 | $ZS^\dagger Z$ | $S^\dagger$ |
| $T$ | 1 | 1 | 1 | 0 | $ZS^\dagger T^\dagger$ | $T$ |
| $T^\dagger$ | 1 | 1 | 1 | 1 | $ZS^\dagger ZT^\dagger S$ | $T^\dagger$ |

Table 4.1: Example table of gates and associated instruction codes for the circuit in Figure 4.5. Each cycle three qubits from the instruction register are loaded to the circuit, the resulting gate depends on the instruction. The target is loaded by querying over the space of registers. Instructions may be loaded in superposition. For example if the instruction register was in the state $\alpha|000\rangle + \beta|100\rangle$ and passed to the logic unit, then $LU(\alpha|000\rangle + \beta|100\rangle)|0\rangle_{\text{Target}} \rightarrow \alpha|000\rangle|0\rangle_{\text{Target}} + \beta|100\rangle|1\rangle_{\text{Target}}$.

operation or an inverse operation is performed.

**History Buffer**

Out last memory region is the history buffer. Registers in the history buffer begin in the $|0\rangle$ state. When an instruction $U_i$ has been executed by the logic unit the state of the $I$ register is swapped into a holding $HI$ register that is obtained from the history buffer. On the subsequent cycle $U_i$ is swapped into the history buffer in exchange for a $|0\rangle$ state. The history buffer queried using a $HIP$ register which is incremented each cycle.

While we can operate in a 'forwards' direction by writing to the history buffer, we may also reverse the computation by instead decrementing $HIP$ and $IP$ and reading previously executed opcodes from the history buffer. To ensure that the history buffer is queried correctly we require a one cycle 'skip' during which the state of the history buffer is unchanged. To achieve this we can poison our QRAM lookup as shown in Figure 4.1.

To skip a history buffer lookup all that needs to be done is replace the query address with the poisoned one as can be seen in Figure 4.2.

### 4.2.2 Computation and Inverse Computation

With these memory units the operation of the cycle may then be described by setting up the program code memory unit with the appropriate instructions. A high level overview of the operation of the cycle may be given as:

1. Swap a fresh $|0\rangle$ state from the history buffer with the $I$ register.
2. Load the current instruction $I$ by querying the instruction pointer $IP$ for an instruction stored in the program code CNOT-QRAQM. As $I$ was is in a $|0\rangle$ state it now contains only the instruction from the program code.
3. Increment $IP$ to the next instruction
4. Query the logic unit ($LU$) to perform a sequence of multi-controlled operations conditioned on the state of $I$ over a set of target registers. Such that:

$$LU\left(\left|\psi\right\rangle\right) = \sum_{i \in IP} \alpha_i U_i \left|\psi\right\rangle.$$

5. Repeat.

We may extend this model of computation to include inverse computation by adding a control wire termed INV.

1. Swap a fresh $|0\rangle$ state from the history buffer with the $I$ register.
2. Load the current instruction $I$ by querying the instruction pointer $IP$ for an instruction stored in the program code CNOT-QRAQM. As $I$ was is in a $|0\rangle$ state it now contains only the instruction from the program code.
3. Conditionally increment $IP$ to the next instruction if $|INV\rangle = 0$, else decrement $IP$ if $|INV\rangle = 1$.
4. If $I$ contains a control instruction, flip $INV$. The control instruction acts as a NOP for the logic unit.

5. Query the logic unit to perform a sequence of multi-controlled operations conditioned on the state of $I$ and $INV$, such that the program code acts as:

$$LU\left(\left|\psi\right\rangle|INV\rangle\right) = \sum_{i\in IP} \alpha_i \left(\langle 0|INV\rangle\, U_i \left|\psi\right\rangle + \langle 1|INV\rangle\, U_i^\dagger \left|\psi\right\rangle\right)|INV\rangle.$$

6. Repeat.

The action of a sequence of operations between instructions $n$ and $m$: $U_m...U_{i+1}U_iU_{i-1}...U_n$ may be inverted by setting the instruction pointer to the end of the sequence $m$ and flipping the $INV$ control wire. This inverts each operation in the sequence and reverses their order, giving: $U_n^\dagger...U_{i-1}^\dagger U_i^\dagger U_{i+1}^\dagger...U_m^\dagger$.

However while this may produce the inverse of a function, it does not uncompute it. In each cycle a fresh instruction is still fetched from the history buffer and our total depth of computation is still limited.

### 4.2.3 Uncomputation

We may extend inverse computation to uncomputation, where rather than writing executed instructions to the history buffer, we will read instructions from the history buffer and execute their inverse. As with the $INV$ register, we introduce a $UNC$ register that controls whether we are incrementing or decrementing the history buffer's instruction pointer.

1. Conditionally swap the top of the history buffer with the I register, controlled on the $|UNC\rangle$ register in the $|0\rangle$ state. If $I$ contains an uncompute control instruction then instead do nothing. This is to ensure that if the last instruction was to start and uncomputation that the control instruction is not stored, but is instead uncomputed immediately.
2. Load the current instruction $I$ by querying the instruction pointer $IP$ for an instruction stored in the program code CNOT-QRAQM. If no swap occurred then the instruction in $I$ will be the same instruction queried by $IP$, which will reset $I$ to $|0\rangle$. Conversely if the swap occurred then we are in the usual forwards computation mode.
3. Conditionally swap the top of the history buffer with the I register, controlled on the $|UNC\rangle$ register in the $|1\rangle$ state. If we are currently uncomputing then this will swap the uncomputed value in the $I$ register back to the history buffer, while swapping in the top value of the history buffer as the next operation to perform.
4. Conditionally increment $IP$ to the next instruction if $|INV\rangle = 0$
5. Conditionally increment $HIP$ to the next instruction if $|UNC\rangle = 0$, else decrement $HIP$ if $|UNC\rangle = 1$.
6. Conditionally decrement $IP$ if $|INV\rangle = 1$. This ensures that if $UNC$ is flipped that the state of $IP$ is left unchanged by that cycle. This is essential in ensuring that the subsequent cycle uncomputes this one.
7. If $I$ contains an $INV$ control instruction, flip $INV$. The control instruction acts as a NOP for the logic unit.
8. If $I$ contains an $UNC$ control instruction, flip $UNC$ and $INV$. The control instruction acts as a NOP for the logic unit.

| Instruction | UNC | REV | Header | Address$_i$ | Address$_j$ |
|---|---|---|---|---|---|
| **Control Operations** | | | | | |
| NOP | * | * | 00 | 000 | 000 |
| INV | * | * | 00 | 001 | 000 |
| UNC$_b$ | * | * | 00 | 100 | 000 |
| UNC$_f$ | * | * | 00 | 110 | 000 |
| UNC$_{\text{NOP}}$ | * | * | 00 | 010 | 000 |
| HLT | * | * | 00 | 111 | 000 |
| **Register Swap Operations** | | | | | |
| Reg$_i[0] \leftrightarrow$ Reg$_i[j]$ | * | * | 10 | $i_0 i_1 j_2$ | $j_0 j_1 j_2$ |
| Reg$_i \leftrightarrow$ Reg$_j$ | * | * | 01 | $i_0 i_1 i_2$ | $j_0 j_1 j_2$ |
| **Logic Unit Operations** | | | | | |
| $U_j \circ$ Reg$_i$ | * | 0 | 00 | $i_0 i_1 i_2$ | $j_0 j_1 j_2 \neq 000$ |
| $U_j^\dagger \circ$ Reg$_i$ | * | 1 | 00 | $i_0 i_1 i_2$ | $j_0 j_1 j_2 \neq 000$ |
| CNOT(Reg$_i$, Reg$_j$) | * | * | 11 | $i_0 i_1 i_2$ | $0 j_1 j_2$ |
| CNOT$_f$(Reg$_i$, Reg$_j$) | 0 | * | 11 | $i_0 i_1 i_2$ | $1 j_1 j_2$ |
| NOP$_b$(Reg$_i$, Reg$_j$) | 1 | * | 11 | $i_0 i_1 i_2$ | $1 j_1 j_2$ |
| **Memory Unit Operations** | | | | | |
| Main Memory Bank Switch | * | * | 00 | 101 | 000 |
| Program Code Bank Switch | * | * | 00 | 011 | 000 |

Table 4.2: Example table of quantum operation codes for an eight bit set of operations. Each of these is associated with a multi-controlled operator over $I$, $UNC$ and $INV$ with various targets. Each cycle the $IP$ register is incremented and a new operation code is fetched from QROM. As this space of operations contains a universal gateset [101, §4.5] it is possible to perform any quantum operation with a sufficiently long chain of these operation codes.

9. Query the logic unit to perform a sequence of multi-controlled operations conditioned on the state of $I$ and $INV$, such that the program code acts as:

$$LU\left(\left|\psi\right\rangle\left|INV\right\rangle\right) = \sum_{i \in IP} \alpha_i \left(\left\langle 0|INV\right\rangle U_i \left|\psi\right\rangle + \left\langle 1|INV\right\rangle U_i^\dagger \left|\psi\right\rangle\right)\left|INV\right\rangle.$$

10. Repeat.

While this demonstrates that uncomputation is possible, we will discuss how this is handled in Section 4.3.1 in more detail.

## 4.3 Operations

In this section we detail a set of operations for the QCPU. The main bottleneck on the operations for the QCPU is the number of operations in the logic unit.

An overview of a sample set of operations can be found in Table 4.2.

### 4.3.1 Control Operations

We specify five 'control' operations that modify the action of the QCPU cycle in situ, along with one adjunct operation.

- The $|INV\rangle$ instruction causes the QCPU to invert operations
- The $|UNC_b\rangle$ instruction causes the QCPU to read instructions from the history buffer and invert them.
- The $\left|UNC_f\right\rangle$ instruction flags a point in the history buffer. If this point is reached while uncomputing then uncomputation is reversed and the program returns to regular computation. This instruction is associated with a $|UNC_{\text{NOP}}\rangle$ instruction that it is replaced with when computing in a 'forwards' direction.
- The $|HALT\rangle$ instruction causes the QCPU to halt operations.
- The $|NOP\rangle$ instruction, that performs no operation

Each of these operations leaves the active registers unchanged.

#### INV Control

The $|INV\rangle$ operation is a simple multi-controlled CNOT from the $I$ register to the $INV$ register. This operation takes place *after* the $IP$ register has been updated for this cycle. This post-update is required as otherwise the $INV$ register would immediately return to the previous instruction, whereas a post-update allows it to perform one additional instruction in its current direction of computation. As shown later this may be a jump instruction, which then allows for functions to be inverted.

#### UNC Control

Uncomputation requires two distinct operations, one that converts from 'forward' computation to uncomputation, and another that reverses this procedure.

The forward to backwards computation immediately uncomputes itself. This can be seen in Figure 4.6.

On the first cycle:

1. The instruction from two cycles ago is swapped into the top of the history buffer.
2. A fresh $|0\rangle$ state is swapped from the history buffer, and the previously executed instruction is placed in the $HI$ register.
3. $|UNC_b\rangle$ is loaded as the current instruction
4. As we are computing in a forwards direction, the history instruction pointer is incremented.
5. If $INV$ is $|0\rangle$ then $IP$ is incremented, else $IP$ does not change
6. Conditioned on the $|UNC_b\rangle$ instruction, both the $UNC$ and the $INV$ registers are flipped. As $|UNC_b\rangle$ can only be performed in a forwards operating mode of computation (as it immediately consumes itself), this will always involve setting $UNC$ to $|1\rangle$

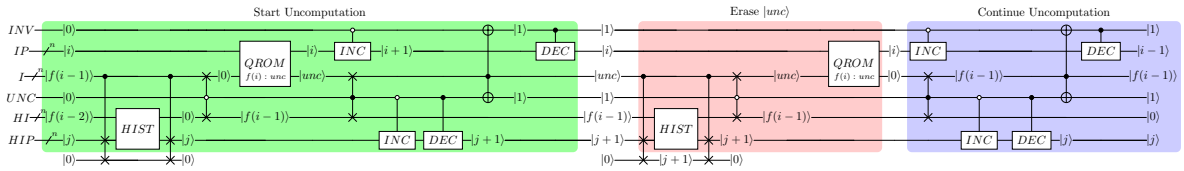



Figure 4.6: Compute to Uncompute Operation, after two cycles the program will continue uncomputing. The swap gadget on the history buffer ensures that the history buffer fetch is 'skipped' for one cycle. Interlacing the INV control between the INV increments and decrements similarly ensures that $IP$ is not changed over the first two cycles, which results in the uncomputation instruction being erased. After this has been completed in the third cycle the history instruction pointer is decremented each cycle, operations are fetch from the history buffer rather than being written to it and the inverse of these operations is being performed prior to the instruction being uncomputed. The operation holds if the INV register begins in either the $|0\rangle$ or $|1\rangle$ state, with the only difference being whether the device is operating in a regular computation mode and incrementing the IP, or an inverse computation mode, in both cases the state of the $IP$ register remains unchanged over the first cycle of this operation.

7. If $INV$ is $|1\rangle$ then $IP$ is incremented, else $IP$ does not change. As $INV$ was just flipped, in combination with the previous update to $IP$ either both updates have resulted in no change to $IP$, or both have applied and $IP$ has been incremented and decremented. In either case $IP$ is now unchanged and still points to the $|UNC_b\rangle$ operation in program code.

On the second cycle:

1. As the history buffer query is conditioned on the $I$ register, the $|UNC_b\rangle$ operation causes this to be skipped for this cycle, leaving the $HI$ register unchanged and containing the instruction from before $|UNC_b\rangle$
2. As the swap from $HI$ is conditioned on the state of $UNC$ this now occurs after program code has been queried
3. $|UNC_b\rangle$ is loaded as the current instruction to the $I$ register, which already contains $|UNC_b\rangle$. This therefore erases that register and leaves $I$ in a $|0\rangle$ state.
4. Now the swap from $HI$ to $I$ is triggered, passing a clean $|0\rangle$ state to $HI$, and retrieving the operation from before $|UNC_b\rangle$ from $HI$.
5. As we are computing in a backwards direction, the history instruction pointer is decremented. In combination with the increment in the previous round $HIP$ now points to the location in the history buffer that stores the instruction that was performed two cycles before $|UNC_b\rangle$.
6. $IP$ is updated based on its flipped state compared to before $|UNC_b\rangle$ was performed.
7. The instruction before $|UNC_b\rangle$ is passed to the logic unit, with the $INV$ register flipped, which performs the inverse of that operation. (Or if the inverse was performed previously, then instead the inverse of the inverse is performed. In either case the operation is inverted).

Once we have begun an uncomputation, we require another operation to restore computation to a forwards direction. Here the $\left|unc_f\right\rangle$ operation is treated as a 'trap' that is placed in the history of the program. When the trap is encountered while uncomputing, the QCPU resumes regular computation.

While at a high level this appears simple, we have an additional constraint: that while in a forwards computing direction the trap acts as a NOP. This leads to issues with constructing an operation that is conditioned on the state of the $UNC$ register and modifies the $UNC$ register.

A forward operation would be expected to simultaneously satisfy

$$|0\rangle_{UNC}\left|unc_f\right\rangle \rightarrow |0\rangle_{UNC}\left|unc_f\right\rangle,$$

and

$$|1\rangle_{UNC}\left|unc_f\right\rangle \rightarrow |0\rangle_{UNC}\left|unc_f\right\rangle.$$

This requirement clearly violates the reversibility of this operation. We will briefly explore a few different approaches to resolving this problem.

A flag or set of flag qubits may delay the issue. Any system of updates to the set of flags is necessarily cyclic, and will eventually result in a state where the uncomputation is reached and a $unc_{\text{NOP}}$ instruction is encountered instead of a $unc_f$ instruction.

Formalising the set of flags we may implement an accumulator such that

$$|0\rangle_{UNC}\left|unc_f\right\rangle|0\rangle \rightarrow |0\rangle_{UNC}\left|unc_f\right\rangle|1\rangle$$

and

$$|1\rangle_{UNC}\left|unc_f\right\rangle|i>0\rangle \rightarrow |0\rangle_{UNC}\left|unc_f\right\rangle\left|i+1 \mod 2^k\right\rangle$$

However this limits the number of uncomputations to $2^k$, and requires storing the state of the counter between independent uncomputations. One solution is to then trigger a halt if a maximum number of uncomputations is reached. Alternatively, a memory that associates $IP$ values with $unc_f$ instructions is effectively a duplicate of the existing program code lookup, offset by one. The overhead of which would be excessive.

Instead we implement an operation

$$V : unc_f \leftrightarrow unc_{\text{NOP}} \tag{4.2}$$

conditioned on $UNC$ computing in a forwards direction. When $unc_f$ is first loaded from the program code the first $V$ occurs before the control operations are performed. This prevents a $unc_f$ immediately reversing computation and sets up our uncompute trap. A second $V$ operation is performed before the state is swapped to the $HI$ register, such that the history buffer stores a $unc_f$ operation rather than a $unc_{\text{NOP}}$.

Once a subsequent $unc_b$ has been encountered (without another $unc_f$ being set first) and uncomputation reaches the trap the first $V$ does not occur as the program is still operating in an uncomputation mode. The $unc_f$ flips the $UNC$ register back to the $|0\rangle$ state, and the second $V$ is triggered, which flips it back to the $unc_{\text{NOP}}$ state. While this operation successfully reverses one uncomputation, it is not re-usable. We will extend this operation in Section 4.4.5 to create a resetting trap.

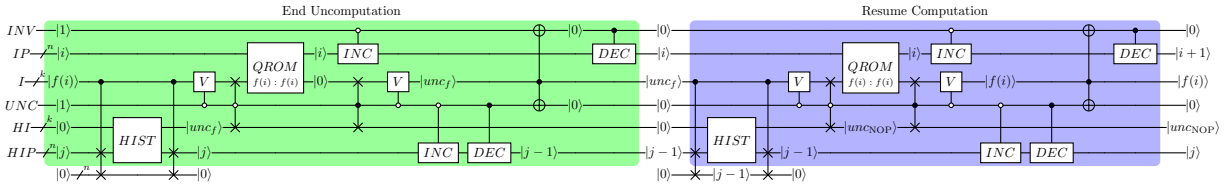



Figure 4.7: Uncompute to compute Operation, after one cycle the program will continue computing. The swap gadget on the history buffer ensures that the history buffer fetch is 'skipped' for one cycle. Interlacing the INV control between the INV increments and decrements similarly ensures that $IP$ is not changed over the first two cycles, which results in the uncomputation instruction being erased. After this has been completed in the third cycle the history instruction pointer is decremented each cycle, operations are fetch from the history buffer rather than being written to it and the inverse of these operations is being performed prior to the instruction being uncomputed.

#### Halt

The halt instruction writes to a halting register each cycle. The register is measured and if a $|1\rangle$ state is observed the QCPU halts.

Another instance that may trigger a halt is if this history buffer is full and the $HIP$ register has reached a maximum address. In this case a halt is required as it is no longer possible to guarantee that the fetched register is in a $|0\rangle$ state, and hence that querying program code will produce the intended instruction.

### 4.3.2 Routing Operations

In addition to the program control we require the ability to move states between registers. These should consist of the $I$, $IP$, $LA$, $L$ and any general purpose registers.

This may be implemented as a set of CSWAP operations between one of the general purpose registers, and each of the other active registers, such that any two registers may be swapped within three cycles. Alternatively using a subset of the instruction as a pair of addresses two SWAP-QRAQM lookups may be performed, and the resulting states swapped before the queries are uncomputed.

For these purposes the bucket brigade QRAQM is parrallelisable so long as different addresses are passed to each element of the swap.

Similarly we must be able to perform swaps between qubits within a register. As before addressing may be performed by either a set of controlled operations, or a SWAP-QRAQM, this time over a single active register and elements $i$ and $j$ are swapped.

### 4.3.3 Logic Unit Operations

To ensure universality, the logic unit must implement a universal basis set of gates over the active registers. As with the routing operations, the logic unit operations may be addressed to an active register using either a set of controlled operations or a SWAP-QRAQM. Operations

are then targeted on the addressed register before it is returned to its original position.

The simplest operations are unary, self inverse operations. The target register is loaded and the operation is applied. An example of this is the Hadamard gate.

For unary, non-self inverse operations we condition both the operation and its inverse on the $INV$ register. When operating in an 'inverse' computation mode the inverse operation is applied in the place of the regular operation. These operations may include Phase gates and T gates.

For non-unary, self-inverse operations, depending on the choice of architecture these may be addressed with a range of non-unary operations. As a minimal example we demonstrate a CNOT operation as a proxy for a self-inverse binary operator. At this point our minimal example with CNOT, Hadamard, Phase and T gates is universal.

For non-unary, non-self inverse operations the appropriate active registers are addressed and the operation is again conditioned on the $INV$ register, and the inverse operation is applied.

Finally, a separate sub-set of operations that are only performed in the forwards computation mode. These are essential to ensuring that elements of the history buffer may be unconsumed without destroying a resulting state. An example of this would be copying the output of a computation to a pre-allocated region of the main memory and then flipping a flag qubit and uncomputing a function, both in a 'forwards' computation direction only. When the program attempts to uncompute these operations their instructions are successfully uncomputed, but they do not modify the state of the active registers. When the program catches a $\left|UNC_f\right\rangle$ instruction and continues in the forwards direction it detects the flag and performs a jump, avoiding being trapped in a cycle of uncomputations.

As an example of these 'forwards only' operations, we may consider a CNOT operation to a target register. In forwards operating mode a $|0\rangle$ value may be swapped from main memory, targeted with a CNOT and then returned to main memory. In an uncomputation operating mode the swaps occur leaving no overall action on the state.

With these primitive operations set out we can now begin to consider implementing control flow primitives on the device.

### 4.3.4 Bank Switching

Typically the size of the address space of the main memory is dictated by the size of the addressing register. For the Harvard architecture this applies independently to the size of the program code memory.

One approach that has been applied to this problem is buffered addressing. The address buffer reads $n$ bits per cycle over $k$ cycles before addressing the QRAM. This gives a $kn$-qubit address space. The buffer is then swapped out over another $k$ cycles. The buffer address ordering may be cyclic or symmetric about the query. To try to mitigate the synchronisation issues, the buffer may instead be bound to the low order bits of the $IP$ register. This would ensure that the buffer's access is determinable at compile time. However this is infeasible for addressing instructions (as opposed to memory), as it implies that an instruction would be processed every $kth$ cycle.

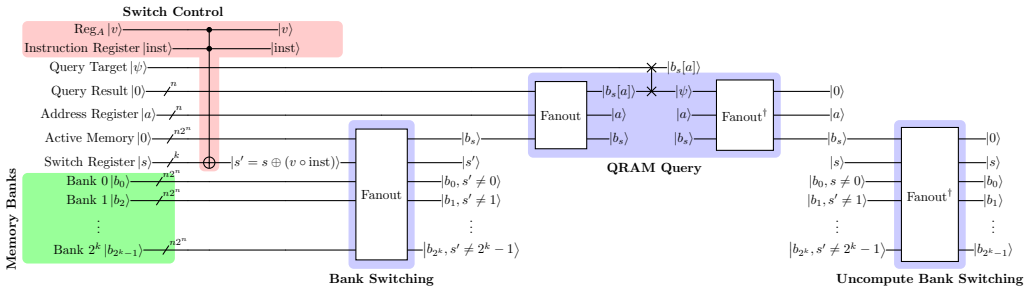



Figure 4.8: A QRAM bank switch involving one QRAM fanout with each bank as a register acting as a bank switch and addressed on a switch register. The output of the bank switch is then used as the basis of another fanout conditioned on the address register. A CNOT or SWAP readout is performed and the fanouts are uncomputed. The state of the switch register is modified via a bitmask prior to bank switching. As the fanout is performed 'bottom up' this would normally imply a fanout-and-swap QRAM, or would require re-indexing for banks of size $n2^k$ for a bucket-brigade QRAM.

An alternative approach is bank switching[22, 33]. A collection of $k$ different memory banks each of size $2^n$ exist. A separate routing network conditioned on a 'switch' register controls which of the $k$ banks is accessed. By modifying the switch register using a dedicated instruction a different memory bank is loaded. This decouples the size of the memory from the maximum addressable unit.

The switch unit itself may be implemented in a number of different ways.

- A literal switch network that physically swaps all memory entries between a 'loaded' memory unit and a set of 'unloaded' memory units. The bank switch instruction would swap the current memory back to its assigned bank, then swap in the new bank. For a large $k$ this switch would itself be a QRAM.
- Each cycle each bank operates a QRAM in parallel on the query address. The switch then acts as a QRAM over the output of each of the banks addressed using the switch register. The bank switch instruction then swaps an assigned register with the switch register. In effect this treats memory as one larger QRAM, addressed over the concatenation of the switch register and the query address.

## 4.4 Control Flow

One of the possible swap operations is to swap the state of $IP$ with one of the active registers. In this section we will manipulate this gadget to construct a reversible jump operation. We will extend this to demonstrate that these jumps are both reversible and uncomputable, and hence that this primitive allows for construction of function calls, which are themselves both invertible and uncomputable.

To select a particular instruction to perform within each cycle each $U$ is controlled on a distinct state of an instruction register $I$. To select the next instruction to perform an instruction pointer register $IP$ may be used to query program memory. With this simple model we can construct

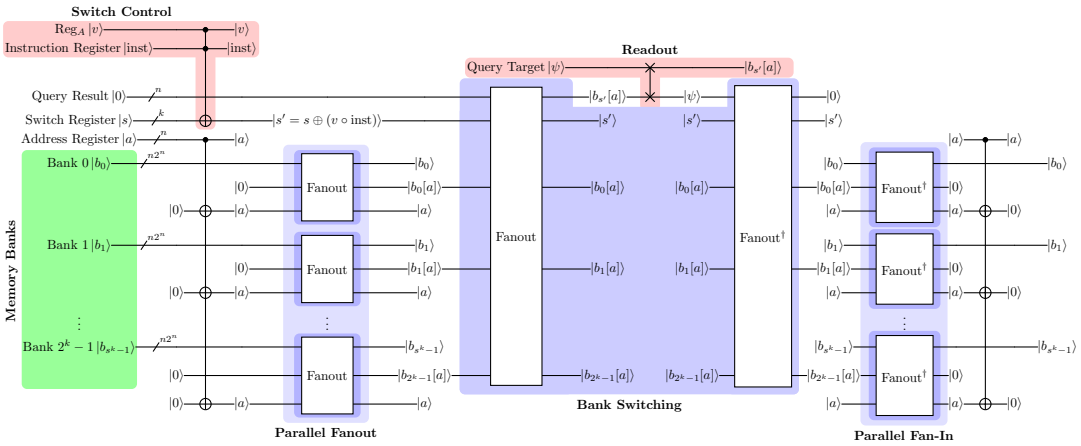



Figure 4.9: A second bank switching design. Rather than switching whole memory banks this approach selects the queried element from each bank, then performs a second query over the outputs of the banks.

linear programs where $IP$ is used to set $I$, an instruction is performed, $I$ is uncomputed and $IP$ is updated to the next instruction. In order to uncompute each $I_j$ such a machine would be limited where each $IP_{j+1}$ must be preceded by a singular $IP_j$. This constraint restricts jump operations as no two jumps may land at the same address and uncompute the source of the jump without tracking the history of the program[135]. In our architecture we have instead opted to track the history of our operations, which allows us greater freedom in the structure of the control flow of our programs.

### 4.4.1 Simple JMP Operations

We can perform a simple jump operation by modifying the contents of the IP register. To perform this reversibly we either swap in a new value from another active register, or we may CNOT another value with the IP.

Preparing and swapping a value into the IP register produces an absolute jump operation.

**Algorithm 3** A Simple Absolute Jump

1: Entrypoint: Prepare L1 in register $A$
2: SWAP Register $A$ with $IP$
3: L1: Continue Execution

Where the prepare operation is predetermined set of up to $n$ bit flips that correspond to the required address in program code.

As we have not defined addition as a primitive operation, we may implement an addition routine in a fixed location in memory and promote our absolute jump into a relative jump.

**Algorithm 4** A Simple Relative Jump

Prepare OFFSET in register $B$
Prepare ADD in register $A$
SWAP Register $A$ with $IP$ ▷ Jumps to ADD
L1: Unprepare $\text{ADD}^\dagger$ in register $A$ ▷ Uncomputes the exit from ADD
Unprepare OFFSET in register $B$
SWAP Register $C$ with $IP$ ▷ Jumps to L1 + Offset
ADD:
Out of Place Addition $C = A + B$ ▷ This may also be done in place
Swap Register $A$ with $IP$ ▷ Jumps to L1

Where the prepare operation is predetermined set of up to $n$ bit flips that correspond to the required address in program code.

**Algorithm 5** A Conditional SWAP Jump

L0: Prepare L1 in register $A$
CNOT CONDITION with A[i] ▷ Conditionally flip one bit of the address
SWAP Register $A$ with $IP$
L1 $\oplus|1\rangle_i$: ▷ If the bit was flipped this branch is taken
Prepare $\text{L}_{\text{True}}$ in register $B$
SWAP Register $B$ with $IP$
L1: ▷ If the bit was not flipped this branch is taken
Prepare $\text{L}_{\text{False}}$ in register $B$
SWAP Register $B$ with $IP$
$\text{L}_{\text{True}}$ or $\text{L}_{\text{False}}$
Perform some operations
CNOT(L3 $\oplus$ L0 + 2, Register $A$) ▷ XOR Register $A$ to L3
SWAP Register $A$ with $IP$
L3:
Control flow is rejoined by both branches

Another approach would be to perform the CNOT on the instruction pointer directly.

**Algorithm 6** A Conditional CNOT Jump

L1 CNOT(cond, $IP[m]$) ▷ Conditionally flip one bit of the $IP$
L1 + 1: ▷ If the bit was not flipped this branch is taken
Perform $k_{\text{False}} - 1$ operations
L1 + $k_f$: CNOT($((L1 + k + 1) \oplus L2), IP$) ▷ terminate this branch by XORing to L2
$(\text{L1} + 1) \oplus 2^m$ : ▷ If the bit was flipped this branch is taken
Perform $k_{\text{True}} - 1$ operations
$(\text{L1} + 1) \oplus 2^m + k_{\text{True}}$: CNOT($((L1 + 1) \oplus 2^m + k_{\text{True}}) \oplus L2, IP$) ▷ XOR $IP$ to L2
L2:
Control flow is rejoined by both branches

While the above examples all present a situation where the condition is a classical state, we may consider the following Bell State preparation. The program is conditioned to branch on the state of one of the qubits in register $A$ which is prepared in the state $|0\rangle + |1\rangle$. A CNOT is performed along the 'True' branch while a NOP is performed on the False branch, giving the state $|00\rangle|\text{False}\rangle + |11\rangle|\text{True}\rangle$. Finally the two branches are rejoined, and temporarily ignoring the history buffer the resulting state is then $\frac{1}{\sqrt{2}}|00\rangle + |11\rangle$. The history buffer may be resolved by performing a forward CNOT from $A$ to another register, setting a flag as a forward operation on another register, then calling the uncompute operation to erase the data from the history buffer and triggering a HALT (or another jump) on the flag that was set.

**Algorithm 7** Bell State Preparation

| | | |
|---|---|---|
| 1: | Hadamard Register $A[0]$ | ▷ $\|0\rangle(|0\rangle + |1\rangle)$ |
| 2: | COND Jump(A[0], $L_{\text{True}}$, $L_{\text{False}}$) | |
| 3: | **$L_{\text{True}}$** | |
| 4: | X A[1] | ▷ $|11\rangle|\text{True}\rangle$ |
| 5: | JUMP L2 | |
| 6: | **$L_{\text{False}}$** | |
| 7: | NOP | ▷ $|00\rangle|\text{False}\rangle$ |
| 8: | JUMP L2 | |
| 9: | **L2:** | ▷ $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ |
| 10: | Control flow is rejoined by both branches | |

### 4.4.2 Switches and Loops

With a relative jump instruction we can construct switches and loops. For these structures we require a scalable reversible comparison gadget: CMP.

We can consider two distinct CMP gadgets, one between a fixed value, and another between two registers.

A fixed value comparison on a single register may be performed by unpreparing that value on the target register than performing and OR operation over that register, outputting to another register in the $|0\rangle$ state. The circuit for the AND gate may be seen in Figure 2.5. If $\big|\psi\rangle == |v\rangle$ then the unpreparation operation will set the target register to $|0\rangle$. Subsequently the OR operation will report a $|0\rangle$ on the target. Flipping the target then outputs $|1\rangle$ if both states are equal, and $|0\rangle$ otherwise. The preparation operation is then performed on the target register to return it to its original state.

$$CMP(v, \big|\psi\rangle)|0\rangle_{\text{Targ}} \rightarrow \langle v\big|\psi\rangle|v\rangle|1\rangle_{\text{Targ}} + \sum_{i \neq v} \langle i\big|\psi\rangle|i\rangle|0\rangle_{\text{Targ}} \tag{4.3}$$

For a comparison between registers the state preparation is omitted and the OR operation is performed onto a third register. The output is once again flipped.

With the CMP gadget and relative jumps we can generalise our control flow. A target jump may be constructed by performing a 'controlled' prepare gadget. For a classical index in the

program code this can be achieved by simply replacing each of the $X$ gates with a CNOT gate conditioned on the output of the CMP gadget.

**Algorithm 8 CMP**$(\text{Reg}_B, v) \rightarrow \text{Reg}_C$

| | |
|---|---|
| 1: Prepare$^\dagger$ $v$ on $\text{Reg}_B$ | ▷ Unprepare $v$ on register $B$. |
| 2: OR $\text{Reg}_B \rightarrow \text{Reg}_C$ | ▷ Output should be in the low qubit |
| 3: X $\text{Reg}_C[0]$ | ▷ Flip the output |
| 4: Prepare $v$ on $\text{Reg}_B$ | ▷ Reset register $B$. |

**Algorithm 9 CMP**$(\text{Reg}_A, \text{Reg}_B) \rightarrow \text{Reg}_C$

| | |
|---|---|
| 1: OR $\text{Reg}_B \rightarrow \text{Reg}_C$ | ▷ Output should be in the low qubit |
| 2: X $\text{Reg}_C[0]$ | ▷ Flip the output |

As before we can distinguish between relative jumps achieved by setting a bitmask and performing a CNOT to the $IP$ register, and absolute jumps achieved by swapping the $IP$ register out.

To make this process easier address $|0\rangle$ of the program code can be set to swap $IP$ and a target register, and a convention established such that the jump is performed by swapping with that register. For the example below we will assume that this is register $A$. This fixed address may be set to any address in program code by replacing the controlled preparation with a bitmark of this base address.

**Algorithm 10** Relative Jump Switch Gadget: CMP-JMP

| | |
|---|---|
| 1: CMP($\text{Reg}_B$, v) $\rightarrow \text{Reg}_C$ | ▷ Compare Register $B$ and $v$, output to Register $C$ |
| 2: C-Prep $\text{Reg}_C$ $j \rightarrow \text{Reg}_A$ | ▷ Prepare a bitmask $j$ using CNOTs instead of $X$s |
| 3: CNOT $\text{Reg}_A$ $IP$ | ▷ $IP \rightarrow (IP+1) \oplus ((\text{Reg}_A == v) \cdot j)$ |
| 4: **IP + 1 :** | ▷ Condition failed, continue computation |
| 5: C-Prep $\text{Reg}_B$ $j \rightarrow \text{Reg}_C$ | ▷ Uncompute $j$ |
| 6: CMP($\text{Reg}_A$, $v$) $\rightarrow \text{Reg}_B$ | ▷ Reset $\text{Reg}_B$ |
| 7: Continue Computation | |
| 8: **(IP + 1)** $\oplus j$**:** | ▷ Condition met, continue computation |
| 9: C-Prep $\text{Reg}_B$ $j \rightarrow \text{Reg}_C$ | ▷ Uncompute $j$ |
| 10: CMP($\text{Reg}_B$, $v$) $\rightarrow \text{Reg}_C$ | ▷ Reset $\text{Reg}_B$ and $\text{Reg}_C$ |
| 11: Continue Computation | |

The uncomputation of $j$ presumes that only a single jump leads to that instruction. We will remove this assumption when we look at CALL and RET operations.

We may also build an absolute CMP-JMP gadget. By placing the following gadget at a fixed location in memory all switch operations may be performed by absolute jumps to this location. This gadget assumes that register $C$ contains the previous value of $IP$ before the absolute jump, registers $A$ and $B$ are being compared, and that register $D$ contains the jump bitmask. It also

assumes that the landing will uncompute or store the jump. Lastly one empty register $E$ is required.

**Algorithm 11** Absolute Jump Switch Gadget: CMP-JMP

| | | |
|---|---|---|
| 1: | **Address** $\|0\rangle$ : SWAP IP A | ▷ Fallback for a failed comparison. |
| 2: | **SWITCH:** | ▷ Entrypoint |
| 3: | CMP($\text{Reg}_B$, v) $\rightarrow \text{Reg}_C$ | ▷ Compare Register $B$ and $v$, output to Register $C$ |
| 4: | C-Prep $\text{Reg}_C$ $\text{L}_2$ $\rightarrow \text{Reg}_A$ | ▷ Conditionally prepare $\text{L}_2$ in register $A$ |
| 5: | CMP($\text{Reg}_B$, v) $\rightarrow \text{Reg}_C$ | ▷ Uncompute Register $C$ |
| 6: | SWAP $\text{Reg}_A$ $IP$ | ▷ Swap register $A$ with the IP. This jumps to either $\|0\rangle$ or $L2$ |
| 7: | **IP + 1 :** | ▷ Condition failed, $\text{Reg}_A = \|1\rangle$. |
| 8: | Continue Computation | ▷ |
| 9: | **L2** | ▷ Condition met |
| 10: | Continue Computation from | ▷ Can't uncompute $A$ as we can't guarantee where the jump was |

In both of these jump gadgets either the bitmask or the swapped $IP$ value remain in the registers after the jump and cannot be uncomputed without a guarantee of what the source of the jump was. In the case of $j$ this would imply that the bitmask between these two addresses is a compile time constant and hence can be uncomputed, while for the swapped $IP$ the state of the address prior to the swap would be known, and could be uncomputed directly.:

Even with this constraint, we can perform local loops and switches, reconstructing the primitives of structured programming [103].

**Algorithm 12** Post-Condition Loop

| | | |
|---|---|---|
| 1: | **Address** $\|0\rangle$ : SWAP IP A | ▷ Fallback for a failed comparison. |
| 2: | Prep L1 $\rightarrow \text{Reg}_A$ | ▷ Set $\text{Reg}_A$ |
| 3: | **L1:** | ▷ Entrypoint |
| 4: | Prep L1 $\rightarrow \text{Reg}_A$ | ▷ Reset $\text{Reg}_A$ |
| 5: | Perform loop operations... | |
| 6: | CMP($\text{Reg}_B$, v) $\rightarrow \text{Reg}_C$ | ▷ Compare Register $B$ and $v$, output to Register $C$ |
| 7: | C-Prep $\text{Reg}_C$ L1 $\rightarrow \text{Reg}_A$ | ▷ Conditionally prepare $\text{L}_2$ in register $A$ |
| 8: | CMP($\text{Reg}_B$, v) $\rightarrow \text{Reg}_C$ | ▷ Uncompute Register $C$ |
| 9: | SWAP $\text{Reg}_A$ $IP$ | ▷ Swap register $A$ with the IP. This jumps to either $\|0\rangle$ or $L1$ |
| 10: | **IP + 1 :** | ▷ Condition failed, $\text{Reg}_A = \|1\rangle$. |
| 11: | X $\text{Reg}_A[0]$ | ▷ Uncompute failed jump |
| 12: | Continue Computation | ▷ |

While we have a CMP gadget, it is also useful to have comparators. With the Cuccero adder and an overflow register this can be trivially performed by performing a subtraction, reading out the state of the overflow qubit as the comparator and then uncomputing the subtraction. However this requires that we have an additional qubit in the operation. Without this we can still do the operation in place by performing a modulo addition rather than having an overflow

bit. Algorithm 13 demonstrates an implementation by taking the parity of the high order bits before and after the subtraction, then uncomputing the subtraction.

**Algorithm 13 Greater Than**$(\text{Reg}_A < \text{Reg}_B) \rightarrow \text{Reg}_C$

| | |
|---|---|
| 1: $\text{Reg}_A[\text{high}] \rightarrow \text{Reg}_C[0]$ | ▷ Set the first qubit in $C$ to the high bit in $A$ |
| 2: $\text{Reg}_A$ - $\text{Reg}_B \rightarrow \text{Reg}_A$ | ▷ In place subtraction of register B from A |
| 3: $\text{Reg}_A[\text{high}] \rightarrow \text{Reg}_C[0]$ | ▷ Conditionally flip the first qubit in $C$ using the high bit in $B$ |
| 4: $\text{Reg}_A + \text{Reg}_B \rightarrow \text{Reg}_A$ | ▷ Restore $A$ |

Other comparators can be constructed by reversing the order of the arguments, taking the OR of the output with the CMP gadget and flipping the output.

### 4.4.3 Turning Memory into a Stack

In the previous operations we copiously avoided the question of what do do with the origin address of a jump operation after it has been swapped to a general purpose register. If the origin is known then it can be uncomputed, which implies a bijective association of origin to destination addresses. As we wish to avoid this constraint we instead build and store addresses on the stack [134].

Our collection of ancillae may be conceptually divided between active uncomputable ancillae, and 'garbage' ancillae. Similarly we ascribe a convention of dividing our stack into three regions:

- A set of active ancillae that grow up through the address space from near the bottom of the QRAQM.
- A set of garbage ancillae that grows down through the address space from the top of the QRAQM.
- A set of metadata containing information pertaining to the control of the stacks, along with any other elements of memory that require a static assignment.

An example of this layout can be seen in Figure 4.10.

While we have ascribed these conventions to the active and garbage ancillae stacks, there's no particular reason that within a local scope both stacks cannot be used for computation.

While we have stated that these are stacks, we still need to construct operational primitives that can fetch and load data from the stacks. The Push operation for the active ancillae pool can be seen in Figure 4.11.

1. Load address $|0\rangle$ or $|1\rangle$ from the main memory (denoted below as $\alpha$) for the active ancillae stack and the garbage ancillae stack respectively by swapping the respective value to the $LA$ register and waiting a cycle. This will swap the current contents of the $L$ register (denoted as $g$) with the respective stack pointer

$$|g\rangle_L |SP\rangle_{\text{Memory:}\alpha} |\alpha\rangle_{LA} \leftrightarrow |SP\rangle_L |g\rangle_{\text{Memory:}\alpha} |\alpha\rangle_{LA}.$$

2. Swap the stack pointer to the $LA$ register, which pushes $\alpha$ to the $L$ register. The QRAM

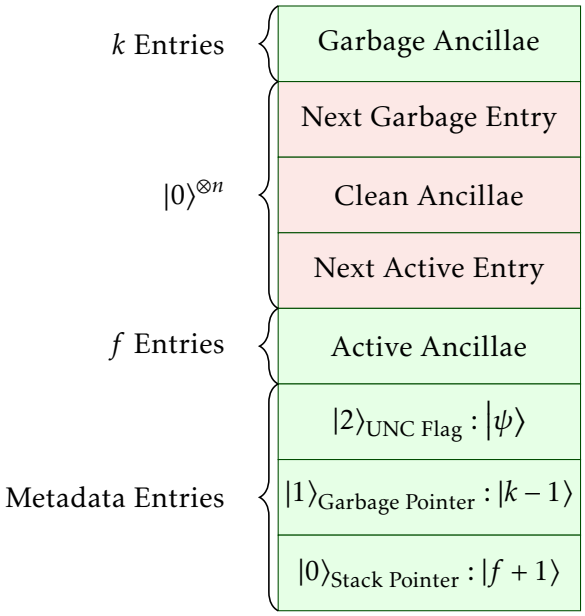



Figure 4.10: The QRAM layout. The first two entries contain the addresses of the first non-active and first non-garbage entry. Active entries grow from the bottom up, garbage entries grow from the top down. By saving states of $IP$ to the active ancillae stack it is possible to construct stack frames, which provides support for calling and returning from functions where the maximum depth of nested calls is only constrained by the amount of memory required.

lookup within the next cycle will now load a fresh $|0\rangle$ from the top of the stack, while $\alpha$ will be stored in QRAM

$$|\alpha\rangle_L |0\rangle_{\text{Memory:}SP} |SP\rangle_{LA} \leftrightarrow |0\rangle_L |\alpha\rangle_{\text{Memory:}SP} |SP\rangle_{LA}.$$

3. Swap the target data (denoted here as $\beta$ to the $L$ register. On the next cycle $\alpha$ will be swapped back out of memory for the target.

$$|\beta\rangle_{\text{Target}} |0\rangle_L |\alpha\rangle_{\text{Memory:}SP} |SP\rangle_{LA} \leftrightarrow |0\rangle_{\text{Target}} |\alpha\rangle_L |\beta\rangle_{\text{Memory:}SP} |SP\rangle_{LA}.$$

4. Swap the stack pointer to a general purpose register, and increment it for the active ancillae stack, or decrement it for the garbage ancillae stack to a new value $SP'$. For the next few cycles the RAM will harmlessly swap the contents of the $L$ register with an arbitrary location in memory, depending on what was in the register that the stack pointer was swapped with.
5. After an odd number of cycles $\alpha$ will be back in the L register. Swap $LA$ with the general purpose register containing the updated SP. This will return $g$ to the L register.
6. After the next cycle's QRAM query the system will be in the state:

$$|0\rangle_{\text{Target}} |g\rangle_L |SP'\rangle_{\text{Memory:}\alpha} |\beta\rangle_{\text{Memory:}SP} |\alpha\rangle_{LA}$$

Pop operations for the active ancillae pool are the inverse of the push operation. The stack

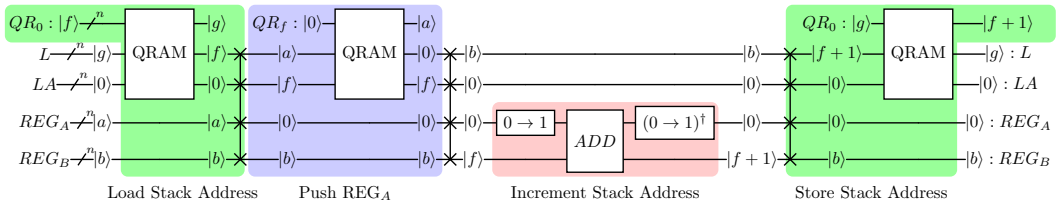



Figure 4.11: QRAM Push Circuit; swaps the value of $A$ to the top of the active ancilla pool.

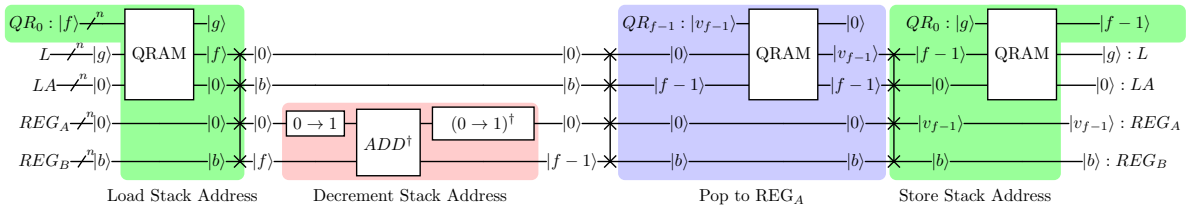



Figure 4.12: QRAM Pop Circuit; loads the last stored value of the active ancilla pool.

address is loaded and decremented, as can be seen in Figure 4.12.

By setting the initial state of $LA$ to $|1\rangle$ and either using a different $|0\rangle$ state for the stack increment operation, or skipping the $0 \to 1$ gates the circuit to push a value to the active ancillae pool is now identical to one that pops a value from the garbage ancillae pool. With the same modifications the pop circuit to the active ancillae pool is now a push circuit to the garbage ancillae pool.

While two stacks allow arbitrary depth searches on either stack (up to space constraints), we may modify the circuits to implement a relative load operations seen in Figure 4.13.

Our relative load operation now allows us to construct stack frames. These are regions of memory relative to a fixed location on the stack. Normally this is either the 'top' of the frame termed the frame pointer, or the bottom of the stack, or the stack pointer. Each stack frame is of a fixed size determined at compile time, and the member variables of a function may then be loaded from the stack frame via a compile time determined offset.

If two relative load operations occur in sequence, and the programmer can provide a guarantee that $IP$ cannot re-enter between those two load operations from an arbitrary program state we can skip a few instructions. Rather than uncomputing the offset each time the increment and decrement stack address operations instead map from one offset to the next.

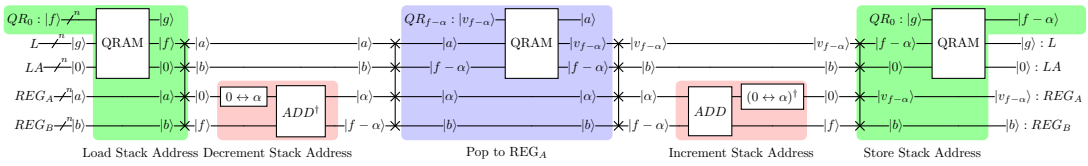



Figure 4.13: QRAM Relative Load Circuit; loads an address from an arbitrary offset from the top of the active ancillae pool. The operation is self inverse as reversing both the order of the gates and taking the inverse of the add gate produces the same circuit.

### 4.4.4 Calling and Returning from Functions

With our stack and the ability to access elements of a stack frame from an offset using the relative load operation specified in Section 4.4.3 and the SWAP-JMP and relative JMP operations from Section 4.4.1 we can now construct function call and return operations. A sequence of operations that implements a function call may be seen in Algorithm 14.

Our goal is to ensure that functions may be called reversibily from multiple points in the program code. Using the SWAP jump for this ensures that the return address may be stored on the stack, whereas a JMP and JMP$^\dagger$ approach requires that each function can only be called from one position in the code.

**Algorithm 14** Call

| | | |
|---|---|---|
| 1: | Increment $SP$ by $i+j+n_{\text{Reg}}$ | ▷ Make space for $i$ arguments, $j$ return values and $n_{\text{Reg}}$ registers |
| 2: | Prepare Arguments | ▷ Whatever operations are needed for $i$ arguments |
| 3: | Relative Swap Registers | ▷ Store the general purpose registers from $SP-i-j-n_{\text{Reg}}$ |
| 4: | Prepare L1, $\text{Reg}_A$ | ▷ Prepares the jump destination L1 in register A |
| 5: | Swap $\text{Reg}_A$ IP | ▷ Performs the Jump, $IP+1$ is in $\text{Reg}_A$ |
| 6: | Prepare L1$^\dagger$, $\text{Reg}_A$ | ▷ Return point of the function L1$^\dagger$ is the exit point of the called function |
| 7: | Relative Swap Registers | ▷ Store the general purpose registers from $SP-i-j-n_{\text{Reg}}$ |
| 8: | Resolve Return Values | ▷ Clear the argument region of the stack |
| 9: | Resolve Arguments | ▷ Clear the return value region of the stack |
| 10: | Decrement $SP$ by $i+j+n_{\text{Reg}}$ | ▷ Free allocated stack space |

As $IP$ is incremented before the swap operation (see Figure 4.2) the value saved to the stack is $IP+1$. When the function returns the next instruction executed is then the one located at $IP+1$.

Preparing and resolving arguments can be performed by swapping them back into the stack frame. In this style the total size of a stack frame should be determined at compile time to maintain the relative offset from the stack pointer. Dynamic allocation can be performed using bank switching, or the garbage ancillae pool. The stack itself now contains of a set of interleaved stack frames, and regions containing saved register states, arguments and buffers for return values.

Returning from the call is simply a matter of returning the system to the state before the call operation was made. The active ancilale stack should be cleared back to the arguments and return values, all registers should be reset to $|0\rangle$ excepting the register used for the call operation, which should contain the return address popped from the stack. This can be seen in Algorithm 15.

**Algorithm 15** Return

| | | |
|---|---|---|
| 1: | Clear current stack frame | ▷ Reset the active ancillae stack to the height it was called in |
| 2: | Pop Stack to $\text{Reg}_A$ | ▷ Pops the Calling Address + 1 from the stack |
| 3: | Swap $\text{Reg}_A$ IP | ▷ Returns to one instruction after the JMP that called the function |

### 4.4.5 Reusable Uncomputation Traps

In the previous section we demonstrated a $unc_b$ operation that uncomputed itself for re-use, and a one time use $unc_f$ trap that returned computation to its normal state, but was flipped to a NOP in the process. In this section we will make use of the relative jump operation to build a trap that sets another $unc_f$ after one is expended. The process for this is given in Algorithm 16.

**Algorithm 16** Reusable Uncomputation Gadget

```
1: Load Uncompute Flag          ▷ a fixed value in main memory, or a dedicated control wire
2: X Flag                                                          ▷ flag is |1⟩
3: L1
4: unc_f
5: Forward: X Flag                          ▷ flag is |0⟩, or |1⟩ if the trap has been flipped
6: if Flag == |1⟩: (Relative Jump L1)       ▷ can be implemented as a CNOT to IP directly
7: Store Uncompute Flag
8: Return
```

Here we set a flag to when $unc_f$ has become $unc_{\text{NOP}}$. The flag is initially in the $|0\rangle$ state. Understanding the process of the method is then a matter of tracking the state of the flag.

1. The flag is flipped to the $|1\rangle$ state before the trap is set
2. The uncompute trap is set by placing a $unc_f$ in the history. During its forward operation $unc_f$ is flipped to $unc_{\text{NOP}}$ before passing through the rest of the cycle, and is reverted to $unc_f$ prior to swapping it to the $HI$ register.
3. The flag is flipped back to the $|0\rangle$ state with a forward only operation
4. During uncomputation the program history will not flip the flag and will reach the $unc_f$ trap, flipping the trap to a $unc_{\text{NOP}}$ state
5. The program will continue in a forwards direction, flipping the flag back to the $|1\rangle$ state.
6. This meets the condition required for the relative jump back to L1, which then places a fresh $unc_f$ trap in the history buffer above the now expended $unc_{\text{NOP}}$ trap.
7. The flag is flipped back to the $|0\rangle$ state by the forward $X$ gate.
8. The gadget returns.

### 4.4.6 Invertable and Uncomputable Functions

With call and return gadgets, the question naturally arises as to whether we can perform inverse functions. This depends on the control flow within a function.

We can demonstrate this with a programmatic gadget that can transform another function $f$ into its inverse.

**Algorithm 17** Call Inverse $f$

```
1:  Jump L1
2:  Entrypoint: Prepare L1-4                ▷ Instruction 1
3:  Jump L1                                 ▷ Instruction 2
4:  Jump L2
5:  L1: Reverse                             ▷ Instruction 3
6:  Jump L2                                 ▷ Instruction 4
7:  Jump L3                                 ▷ Instruction 6
8:  L2: Call f                              ▷ Instruction 5
9:  Jump L3
10: Jump L4                                 ▷ Instruction 8
11: L3: Reverse                             ▷ Instruction 7
12: Jump L4
13: Return
14: L4: Unprepare L1-4                      ▷ Instruction 9
15: Return                                  ▷ Instruction 10
```

Notably, if the function is itself called in reverse then an alternative path of execution is taken through the function, which as $f^{\dagger\dagger} = f$ simply performs the operation.

If $f$ is called reversibly then the instruction pointer jumps to the entry point and decrements. The reverse instruction flips the $INV$ register and reverses the direction that the instruction pointer is travelling. As the instruction pointer is updated before this register is flipped, one subsequent instruction is processed before the direction is reversed.

This function may also be uncomputed, which combined with uncomputing its inverse gives four different paths of execution through this function. For simplicity we note that all operations within this function are self-inverse. When uncomputing this the return statement below L4 is first loaded to $IP$ and the instruction pointer decrements by default. It then inverse-unprepares L1-L4 before loading the jump above L3 from the history buffer, the reverse instruction is loaded, followed by the jump to L3, the exit-point of $f$ is then loaded and $f$ is uncomputed before the program returns (or un-enters from) the Call $f$ instruction. The remainder of the jump and preparation operations are uncomputed in the same fashion.

This quickly gives rise to the basis of a type system for a higher level language. Functions may only be called invertibly if all non-returned memory is uncomputed. As L1-4 are implementable as absolute jumps and hence un-computable the call inverse function is itself reversible if $f$ is reversible. Hence while all functions may be uncomputed, some functions may be typed as being reversible or irreversible.

One problem that swiftly arises with inverse functions is ensuring the correctness of execution paths in both the forwards and reverse directions as $IP$ is either incremented or decremented.

When returning from the function normally, the return operation is loaded, and instruction pointer incremented before being swapped out. The instruction pointer then contains the address of the return address plus one. When entering the function in reverse we wish to skip the initial return operation. We mark four addresses: our regular entry and exit points, and a

**Algorithm 18** Linear Invertible Function

1: $\text{Reg}_a : 0 \leftrightarrow f - k$
2: $\mathbf{L}\ f_r^\dagger$
3: $\text{Reg}_a : f - k \leftrightarrow f$
4: Swap $\text{Reg}_a \leftrightarrow$ Instruction Pointer ▷ Function entrypoint
5: $\text{Reg}_a : f^\dagger + k \leftrightarrow f^\dagger$
6: $\mathbf{L}\ f_r^\dagger$
7: $\text{Reg}_a : f^\dagger + k \leftrightarrow f$
8: ...
9: Swap $\text{Reg}_a \leftrightarrow$ Instruction Pointer ▷ Inverse return
10: Decrement $\text{Reg}_a$ ▷ Skips one instruction upon returning
11: $\mathbf{L}f$:
12: Set-up stack
13: Perform operations $U_0 ... U_k$
14: Teardown stack
15: $\mathbf{L}f^\dagger$
16: Increment $\text{Reg}_a$ ▷ Skips one instruction upon returning
17: Swap $\text{Reg}_a \leftrightarrow$ Instruction Pointer ▷ Forward return

second set of inverse entry and exit points, such that the inverse exit is $k$ instructions before the entry point, and the reversed entry is $k$ instructions before the exit point. The stack set-up and teardown operations are self inverse, and the operations are inverted and inverse in order. We split the preparation of the calling address of the function into the preparation of the inverse exit point, then a map from the inverse exit point to the entry point. This split is replicated with the uncomputation of the return addresses and the exit point and inverse entry point. In fowards operating mode the function is entered at $f$ and exited at $f^\dagger + k$, while in inverse mode it is entered at $f^\dagger$ and exited at $f - k$. By aligning the entry and exit addresses such that $f$ and $f \pm k$ only differ by one bit flip. By incrementing or decrementing the return address prior to returning, one of the preparation steps is skipped in either direction.

When uncomputing this function, the operations are simply read from the history buffer and control flow is handled by reading out the previous history, rather than by the instruction pointer.

## 4.5 Scholia

In this section we will detail the implementation of a few quantum algorithms as programs for the QCPU.

### 4.5.1 Reversibly Repeat Until Success

We present a small example a minimal repeat until success circuit of this process using only the $A$ and ring buffered $IP$ as an analogy for the operation of the QCPU. We limit this example to two operation codes: $|0\rangle$ is the NOP operation, while $|1\rangle$ attempts to prepare the state. In this

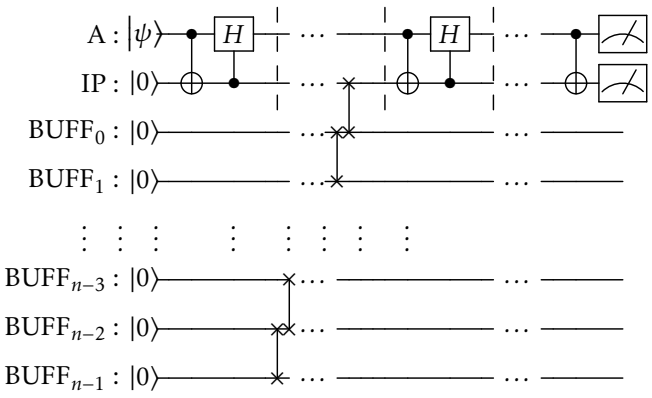



Figure 4.14: Sample repeat until success circuit. This model only requires the $A$ and ring buffered $IP$ register and may be implemented on current quantum devices. The final state of the $IP$ register may be retained as a flag register with state $\alpha|0\rangle_A|0\rangle_{IP} + \beta|1\rangle_A|1\rangle_{IP}$.

example we will be attempting to prepare the $|0\rangle$ state from an arbitrary initial state reversibly using a controlled Hadamard operation. We then consider two entangled lines of execution; in the instance that the target is in the $|0\rangle$ state the entangled $IP$ register performs a NOP operation, while in the instance that the target is in the $|1\rangle$ state the entangled $IP$ register performs a Hadamard operation. Control flow along each line of execution is coherently joined after each iteration. With a full QROM lookup with increment as described by the cycle model we would be forced to dump entangled IP in order to join our entangled execution paths.

Using the underlying uniform circuit model we have an inherent depth constraint in the number of attempts we may make in our repeat until success circuit, and as our attempts each require the consumption of an $IP$ register, we also limit the memory required for our program by the number of attempts. The circuit for $n$ attempts is presented in fig. 4.14 while the associated pseudo code is presented in algorithm 19

**Algorithm 19** Depth Constrained Reversibly Repeat Until Success

1: $i \leftarrow 0$
2: **while** $i < n$ **do**:
3: **if** $A \neq |\phi\rangle$ **then**
4: $\quad A \leftarrow \text{Prepare}(A)$
5: $i \leftarrow i + 1$
6: **done**

For the particular choice of $|\phi\rangle = |0\rangle$ and using the Hadamard operation as an approximate method of preparation we can evaluate the probability with which the state is correctly prepared when starting from an arbitrary state $|\psi\rangle$

$$\left|\psi\right\rangle_A^0 |0\rangle_{IP} |0\rangle_0 |0\rangle_1 \dots |0\rangle_{n-1}$$

$$(\alpha |0\rangle_A + \beta |1\rangle_A) |0\rangle_{IP} |0\rangle_0 |0\rangle_1 \dots |0\rangle_{n-1}$$

Apply CNOT and CH

$$\left(\alpha |0\rangle_A |0\rangle_{IP} + \frac{\beta}{\sqrt{2}} |0\rangle_A |1\rangle_{IP} - \frac{\beta}{\sqrt{2}} |1\rangle_A |1\rangle_{IP}\right) |0\rangle_0 |0\rangle_1 \dots |0\rangle_{n-1}$$

Ring Buffer Swap

$$\left(\alpha |0\rangle_A |0\rangle_{IP} |0\rangle_0 + \frac{\beta}{\sqrt{2}} |0\rangle_A |0\rangle_{IP} |1\rangle_0 - \frac{\beta}{\sqrt{2}} |1\rangle_A |0\rangle_{IP} |1\rangle_0\right) |0\rangle_1 |0\rangle_2 \dots |0\rangle_{n-1}$$

Store zeroth entry of the ring buffer

$$\left(\left(1 - \frac{\beta^2}{2}\right)^{\frac{1}{2}} |0\rangle_A - \frac{\beta}{\sqrt{2}} |1\rangle_A\right) |0\rangle_{IP} |0\rangle_1 |0\rangle_2 \dots |0\rangle_{n-1}$$

$$\left|\psi\right\rangle_A^1 |0\rangle_{IP} |0\rangle_1 |0\rangle_2 \dots |0\rangle_{n-1}$$

Repeat $n$ times.

$$\left(\left(1 - \frac{\beta^2}{2^n}\right)^{\frac{1}{2}} |0\rangle_A + (-1)^n \frac{\beta}{2^{\frac{n}{2}}} |1\rangle_A\right) |\text{flag}\rangle_{IP}$$

$$\left|\psi\right\rangle_A^1 |\text{flag}\rangle_{IP}$$

Figure 4.15: Evaluation of the depth constrained repeat until success circuit. This calculation demonstrates that the probability with which the incorrect state $|0\rangle$ is measured shrinks exponentially in the number of rounds.

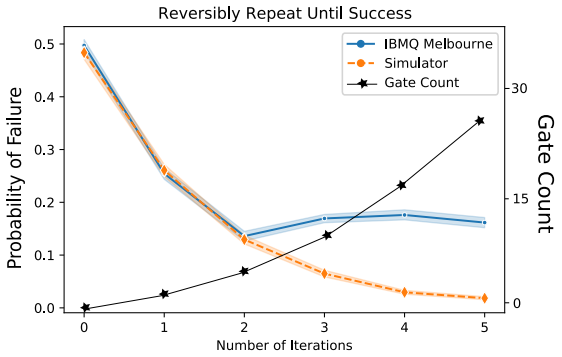



Figure 4.16: Performance of the reversibly repeat until success circuit on the IBMQ Melbourne and on the QASM simulator. Error bars presented at a 90% confidence interval. The failure probability remains approximately constant in spite of the additional associated gate error rate as the gate count increases quadratically. The circuit executed can be seen in fig. 4.14.

This particular example also demonstrates 'cleaning up' a particular register; albeit at the cost of a greater number of already clean registers. This approach can be generalised to any $\left|\psi\right\rangle_A \rightarrow |0\rangle_A \rightarrow \left|\phi\right\rangle_A$ with the success probability scaling with $n$.

### 4.5.2 Grover

To give an example of the operation of our model we can consider Grover's Algorithm. We will consider Grover's algorithm as a set of diffusion operators. Each diffusion operator maps from

non-flagged states to the flagged state with some probability, and maps from the flagged states to the rest of the program. These flagged states then 'accumulate' while non-flagged states 'disperse'.

The Grover search algorithm is one of the earliest discovered speedups that quantum computers have over classical. We are presented with an $n$-qubit register ('loc') that records as an unsigned binary integer a possible location $\ell = 0, 1, \ldots, N-1$ ($N = 2^n$) of a marked item. We are further given a black-box oracle $Q$ that reports whether or not the location $\ell$ is marked as follows:

$$Q|\ell\rangle_{\mathtt{loc}}|b\rangle_{\mathtt{flag}} = \begin{cases} |\ell\rangle_{\mathtt{loc}}|\neg b\rangle_{\mathtt{flag}} & \ell \text{ is marked,} \\ |\ell\rangle_{\mathtt{loc}}|b\rangle_{\mathtt{flag}} & \text{otherwise,} \end{cases} \tag{4.4}$$

where the register 'flag' consists of a single qubit that is prepared initially in the state $b = 0, 1$. In Grover search, the idea is to perform amplitude amplification on a basic procedure we refer to as the *Grover iterate*. The Grover iterate is defined by a two-step process on the two registers loc and flag, all of whose qubits are initialised to zero, as follows:

1. apply a Hadamard gate to each qubit in loc, and
2. apply the oracle $Q$ to (loc, flag).

The above refers to the *forward* version of the Grover iterate. If we run the above steps in reverse order, it is the *reverse* version of the Grover iterate. The forward and reverse versions of the Grover iterate form a pair of inverse operators on (loc, flag). Hence we refer to the forward version as $G$ and the reverse as $G^\dagger$. Amplitude amplification then refers to the following procedure given $G$: execute $G$, then $Z = HXH$ (where $H$ is Hadamard and $X$ is a NOT operation) on flag, then $G^\dagger$, then $Z$, then $G$, and so on for $\mathcal{O}(\sqrt{N})$ total steps.

Following amplitude amplification of the Grover iterate, we then measure flag. If the outcome is 0 (guaranteed to occur with less than, say, 1/3 probability), we discard the result as a failure. Otherwise, we read the value encoded in loc. That value is the location of the marked item.

In order to perform a Grover search we require three components; a preparation of the search space, an oracle and a Grover iterate[65]. We will begin with the assumption that the data we wish to query is contained within QRAM.

The preparation of the search space then requires that we construct a register containing an equal superposition of the addresses in QRAM that we wish to query over. We can then pass this to the *LA* register which will load the contents of those addresses in superposition. Our Grover oracle is then constructed as an evaluation of whether or not an entry in this range matches the search condition. Lastly the Grover iterate must flip the phases of all elements of the register.

We modify Grover's algorithm, such that after a search is completed, the program enters a superposition of the state where the search was successful, and where it was unsuccessful. For a function $f$ acting over $x$ After an application of Grover's algorithm we are left with:

$$|0\rangle|0\rangle \rightarrow G(\sum_x \alpha\,|x\rangle|0\rangle) \rightarrow \sum_{x|f(x)=1} \alpha'_x|1\rangle|x\rangle + \sum_{x|f(x)!=1} \alpha'_x|0\rangle|x\rangle \tag{4.5}$$

The branch where the search criterion is met is then 'skimmed' off, and the state containing the output registers is copied to a holding register and a latch set.

$$\sum_{x|f(x)=1} \alpha'_x |1\rangle|x\rangle|x\rangle|1\rangle_{\text{Latch}} + \sum_{x|f(x)!=1} \alpha'_x |0\rangle|x\rangle|0\rangle|0\rangle_{\text{Latch}} \tag{4.6}$$

The Grover sequence is then uncomputed, leaving the states in their initial configuration.

$$|0\rangle|0\rangle\left(\sum_{x|f(x)=1} \alpha'_x |x\rangle|1\rangle_{\text{Latch}} + \sum_{x|f(x)!=1} \alpha'_x |0\rangle|0\rangle_{\text{Latch}}\right) \tag{4.7}$$

Our full repeat until success Grover operation is then shown in Algorithm 20. By uncomputing the state preparation before returning we ensure that the next cycle of the algorithm is not entangled with the un-marked states. Each failed Grover cycle increases the history buffer by a constant number of instructions via the uncompute trap gadget. The program halts when either the correct output has been found, or the history buffer memory has been exhausted from the accumulated instructions from each round of uncomputations.

**Algorithm 20** Grover

1: Set Uncompute Trap ▷ See Section 4.4.5
2: FORWARD { if Latch == 1 : $X$ Latch; Return $\text{Reg}_{\text{Holding}}$ }
3: Prepare Uniform Superposition
4: Call $g$ ▷ Call Diffusion Operator
5: FORWARD { if $f(x) = 1$ : $X$ Latch; $x \rightarrow \text{Reg}_{\text{Holding}}$ }
6: $\text{UNC}_b$ ▷ Trip the uncompute trap

# Chapter 5

# Surface Code Benchmarks

## 5.1 Benchmarking Magic State Factories

In this section we will benchmark the operation of the 15-1 magic state factory using the circuit shown in Figure 3.27. This circuit is relatively simple in that all gates are either strictly dependent on each other, or trivially parallelisable. A $4 \times 8$ QCB implementing the magic state factory may be seen in Figure 5.1. This is strictly larger than more bespoke magic state factories [56, 88] but due to the IO channel limitations and the distinction between computational and topological ancillae, this is the smallest $T$ factory that the compiler can currently produce.

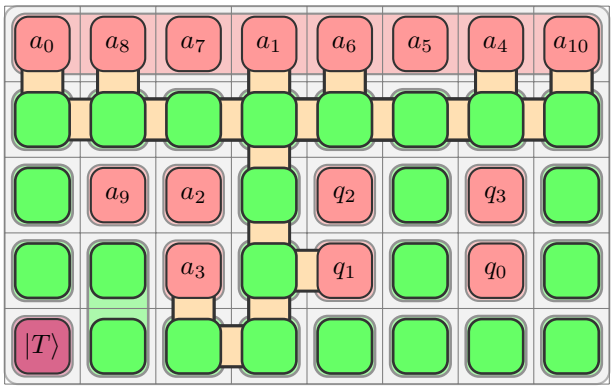
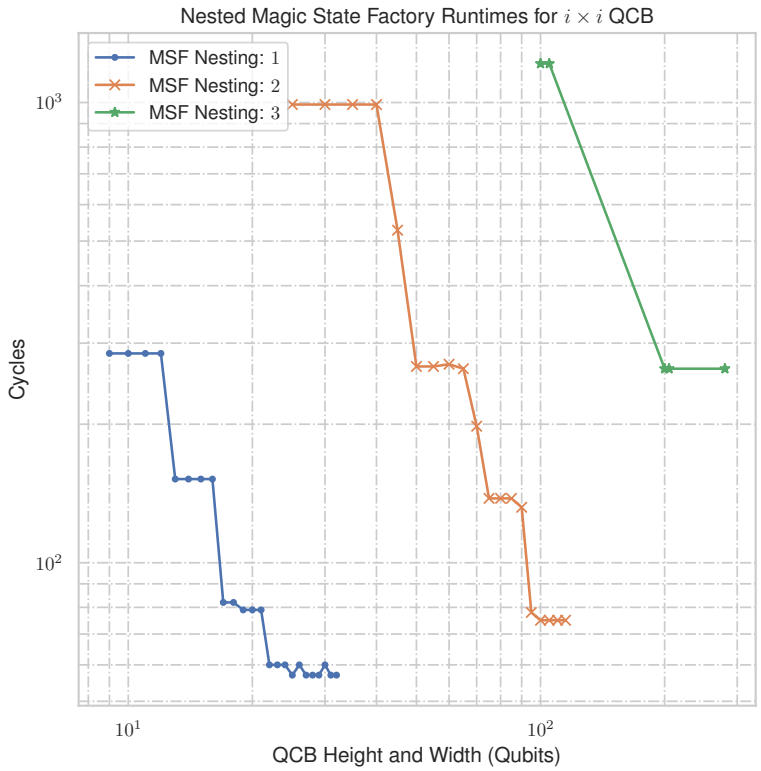



Figure 5.1: Compiler output showing a $4 \times 8$ $T$ factory extern qcb containing six externs implementing Figure 3.27. In all subsequent benchmarks unless stated otherwise this QCB will be used to satisfy any external dependencies on $T$ gates.

We may nest this initial magic state factory in a second round of distillation where each of the $T$ gates in the circuit are satisfied by the output of a previous round of distillation[43, 81]. An example of the nested layout can be seen in Figure 5.3. The runtime of the nested magic state factories is dominated by the magic state factory runtimes. The total execution time of the QCB then depends on the number of externs, which in turn depends on the size of the QCB. The requirement for fifteen $T$ gates at the end of the circuit creates a bottleneck, such that if the QCB is only sufficiently large enough for a single extern then this portion of the program.




;

Figure 5.2: Runtime for nested magic state factories. The QCBs for these magic state factories was constrained to be square. The stochastic drops in execution time correspond to larger QCBs providing support for additional nested externs. Quadrupling the dimensions of the nested QCB invariably admitted up to 16 externs in the larger magic state factory, which provided the maximum number of $T$ states needed to implement the circuit.

Conversely if the QCB is large enough to contain a full fifteen externs then the program returns to being trivially parallelisable. The runtime for a set of square (height and width are equal) QCBs can be seen in Figure 5.2.

While throughout the remainder of this chapter we will continue to use the $4 \times 8$ QCB magic state factory, as all $|T\rangle$ state magic state factories share the same symbol they are all directly interchangeable within the compiler.

## 5.2 Benchmarking Quantum Logic Gates

We will begin with benchmarking the output of the compiler for a set of common quantum logic gates. These each make use of the output of the magic state factory from the previous section as an external factory.

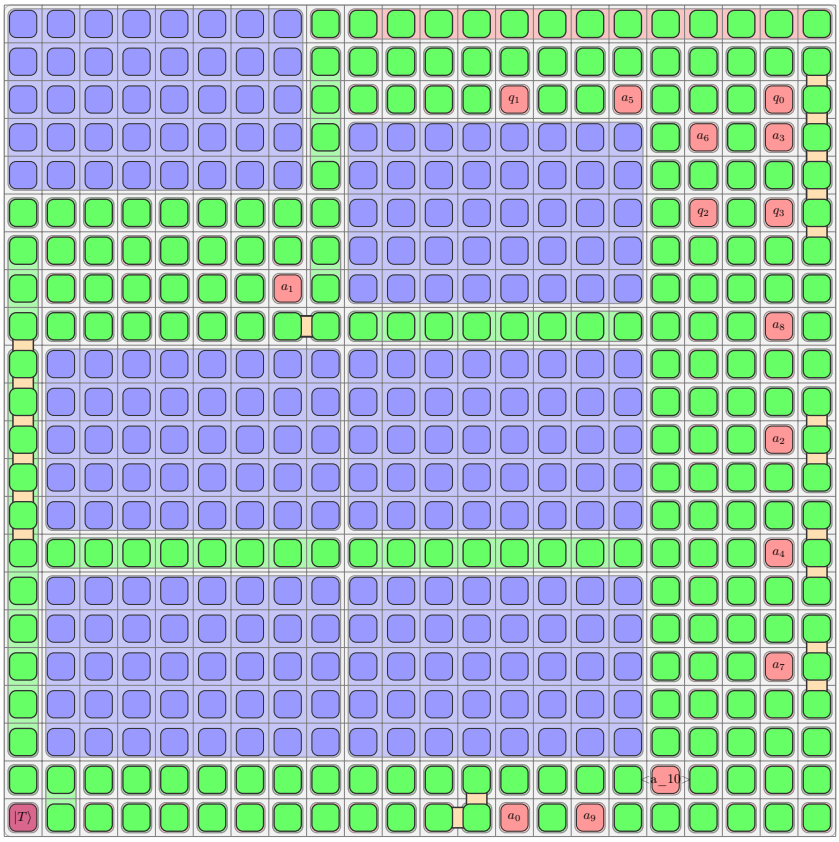
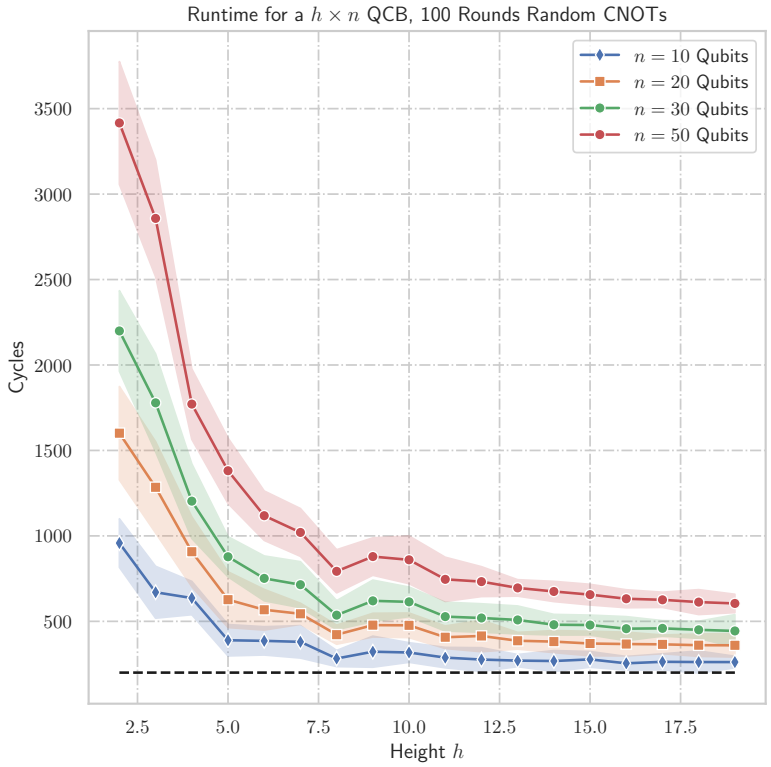



Figure 5.3: Compiler output showing a nested $T$ factory containing six externs implementing Figure 3.27. Also shown are a set of ancillae operations preparing a non-local routing ancillae.




;

Figure 5.4: Two rounds of 100 independent CNOT gates as the height of the QCB varies. At $h = 2$ all registers share a single routing lane and experience a factor of 5 slowdown compared to the expected runtime. Even with a square topology a factor of 1.5 slowdown is observed.

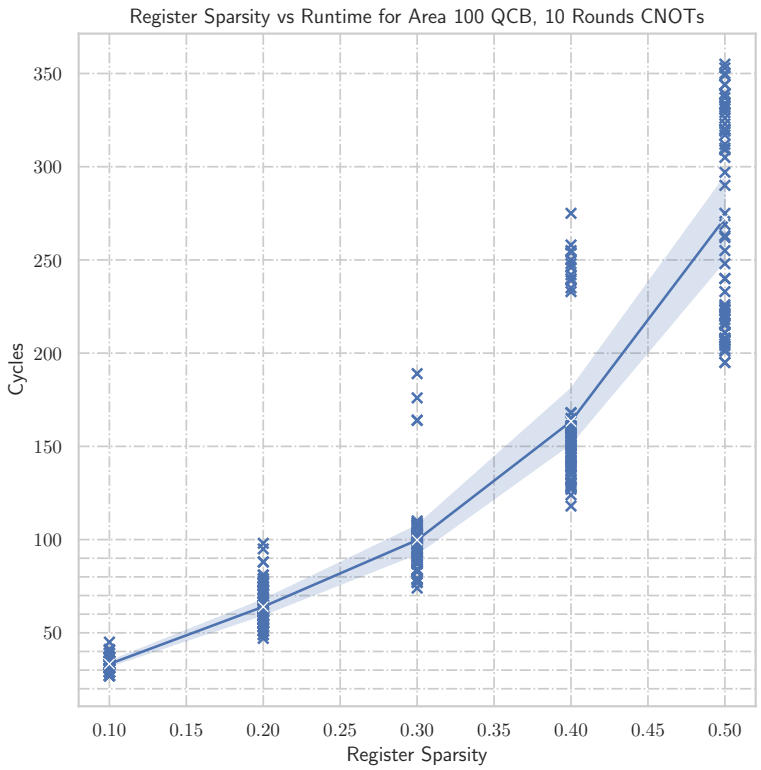
### 5.2.1 Random CNOT Network

Our first benchmark performs a series of 100 random CNOT gates between unique pairs of qubits on a $10 \times i$ QCB such that each qubit participates in at most two CNOT gates. Each CNOT depends only on CNOT operations in the previous layer. With no joint dependence on routing ancillae, these operations would take 200 cycles.

Instead we see that depending on the height of the QCB, and hence the number of routing ancillae avaliable that the runtime of the circuit varies from 1000 cycles to 300 cycles, representing between a 50% to a 500% slowdown as seen in Figure 5.4.

This may also be expressed as a function of the 'sparsity' of registers relative to routing ancillae. This may be seen in Figure 5.5.

These figures demonstrate the clear dependence of otherwise independent surface code operations on intermediary routing ancillae.




;

Figure 5.5: Runtime of random CNOT circuits expressed as a function of the sparsity of the QCB. The sparsity is the faction of the QCB dedicated to registers divided by that assigned for routing ancillae.

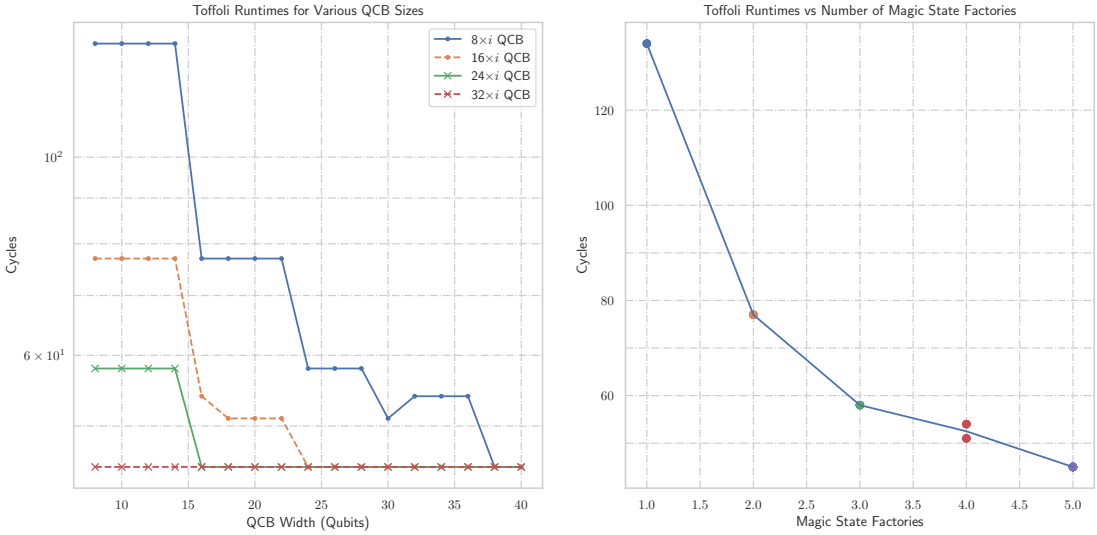



Figure 5.6: Toffoli runtime as QCB size increases. The Toffoli gate is implemented using the circuit in Figure 2.1. As the QCB size increases the allocator places an increasing number of magic state factories on the QCB, which reduces the waiting time. This reduction caps at five magic state factories for our 15-1 factories, after which the time required to perform the intermediary operations admits the first magic state factory a sufficient number of cycles to emit another magic state.

### 5.2.2 Toffoli Benchmarks

A simple proof of concept of compiling with externally defined factories is the Toffoli gate. Using the circuit in Figure 2.1 our Toffoli gate requires six $T$ states. The DAG representation of this circuit may be seen in Figure 3.31.

As the size of the QCB increases, more $T$ state factories are allocated reducing the waiting time. This can be seen in Figure 5.6. The allocator does not allocate the full six magic state factories as the number of intermediary cycles required to perform the intervening $T$ gates provides enough time for the emission of another $T$ state from the first factory.

This belies what will be a running theme in this section, the primary bottleneck on the runtime of the circuits is the rate at which magic states are required, rather than the number of magic states that are required as compared to the rate at which they are produced.

The relation between this dependency and the size of the QCB is based on the number of magic state factories that may be run in parallel to satisfy the rate of magic state consumption. In essence this replicates Amdahl's law.

### 5.2.3 Multi-Controlled X

Using the Toffoli gate as a primitive, we may extend our benchmarking to general multi-controlled $X$ gates as seen in Figure 2.2. These circuits are comprised of collections of Toffoli

gates. These QCBs may be compiled either using Toffoli gates as an extern, and hence preventing the sharing of magic states between extern regions, or by using magic state factories as externs, thereby increasing the number of surface code allocations, and the number of gates in the circuit.

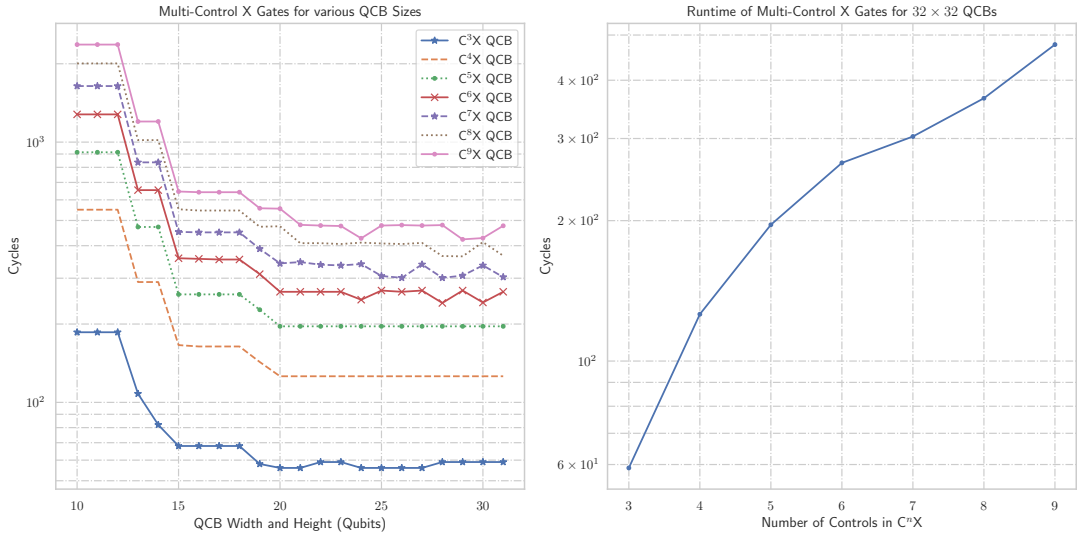



Figure 5.7: Runtime of Multi-controlled-X gates (CCCX, CCCCX ...) as a function of the number of controls, and the size of the QCB. These are implemented using the pattern demonstrated in Figure 2.2

### 5.2.4 Unary Rotations

A non-Toffoli magic state operation is approximating an arbitrary $Z(\theta)$ gate to a given precision. This is performed using a sequence of Clifford, Pauli and $T$ gates, with the sequence itself depending upon the choice of $\theta$ and the precision of the approximation. We generate these sequences using 'gridsynth' [116].

For a single $Z(\theta)$ rotation with the 15-1 $T$ state factory we saturate the magic state production rate at three magic state factories. The number of magic state factories corresponds to increasing size of QCB in Figure 5.8.

## 5.3 Benchmarking Quantum Arithmetic

Quantum arithmetic circuits underpin a collection of quantum algorithms. Shor's algorithm [121] assumes a quantum modulo power operation, while a number of quantum simulation algorithms depend on the implementation of Taylor series calculation on quantum hardware.

To construct these arithmetic circuits we have built a quantum multi-precision arithmetic library. The library performs classical simulation of CNOT, Toffoli and $X$ gates on binary vectors, and implements a linear allocator for ancillae tracking and re-use. The classical verification of

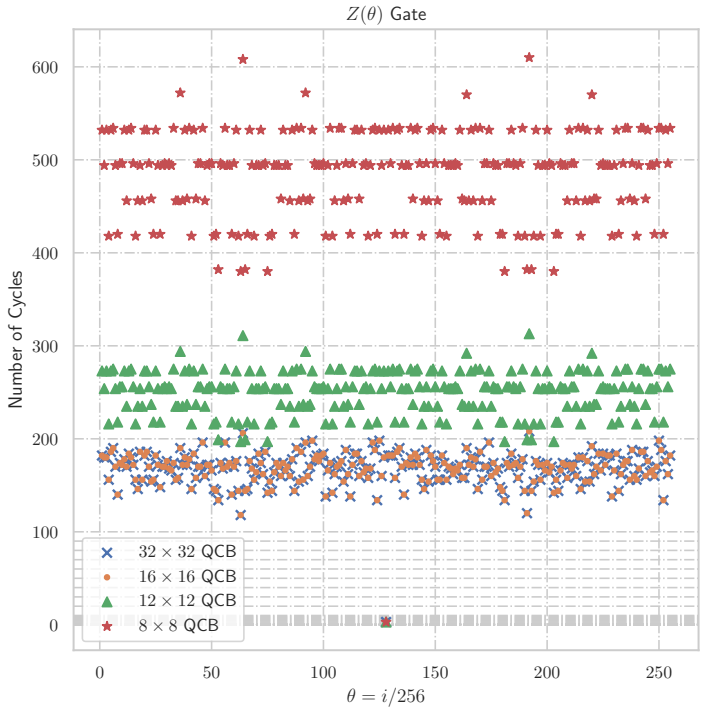



Figure 5.8: Cost of $Z(\theta)$ rotation where $\theta = \frac{i}{256}$. The drop in the center of the graph is the $S$ gate at three cycles and with no magic state dependencies. The magic state production rate is saturated with three 15-1 magic state factories. With slower factories this number would increase.

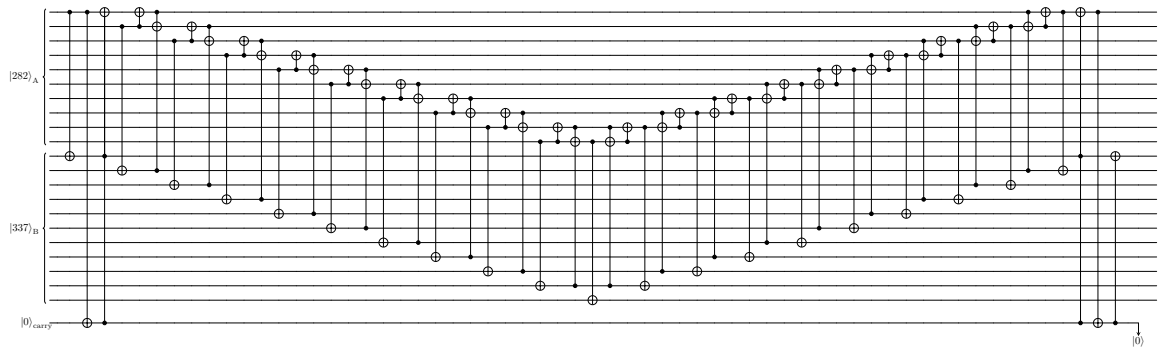



Figure 5.9: Example Addition Circuit demonstrating the linear sequence of dependent Toffoli and CNOT gates. This version of the adder takes two registers of size $n$ and $n+1$ qubits, along with an ancillae carry qubit that is reset to the $|0\rangle$ state after the operation of the circuit.

these operations is comparatively computationally cheap. Once verified each of the gates of these arithmetic circuits has then been mapped to a sequence of surface code operations and passes to the compiler.

This section details the costing of a range of quantum circuits for varying QCB sizes.

### 5.3.1 Addition and Subtraction

Addition and subtraction circuits are inverses of each other, and each are constructed with a sequence of 'MAJ' and 'UMAJ' gates [35]. These circuits may implement either modulo arithmetic or increase the size of the output register by a qubit to track an overflow bit. An example circuit may be seen in Figure 5.9.

As this implementation of an adder is not particularly parallelisable, the rate of magic state consumption by the circuit is approximately constant, and depends on the Toffoli gates. While the sequence length increases with the size of the register the rate of $T$ gates required by the circuit remains constant. As a result a constant number of magic state factories are required to implement this circuit within optimal time bounds for any size of adder.

Unfortunately, for a fixed size QCB, as the register size increases the memory available for dedicating to externs decreases. Eventually this results in a sub-optimal number of magic state factories and reciprocally impacts the performance of the QCB. This can be seen in Section 5.3.1. The $10\times10$ QCB begins with only a single magic state factory, and eventually runs out of space to place all the required registers, routes. Similarly as the register size increases the $12\times12$ and $16\times16$ QCBs shed magic state factories and suffer performance degradation.

### 5.3.2 Multiplication

A shift and add multiplication circuit is an out of place operation that takes two registers of size $a$ and $b$ and writes the output to a register of size $a+b+1$. This is performed by the sequential application of conditional addition circuits with shifted outputs. The conditional additions are performed by using Toffoli gates as a set of 'controlled copy' operations, such that a register may be conditionally copied, before addition is performed. By manipulating the

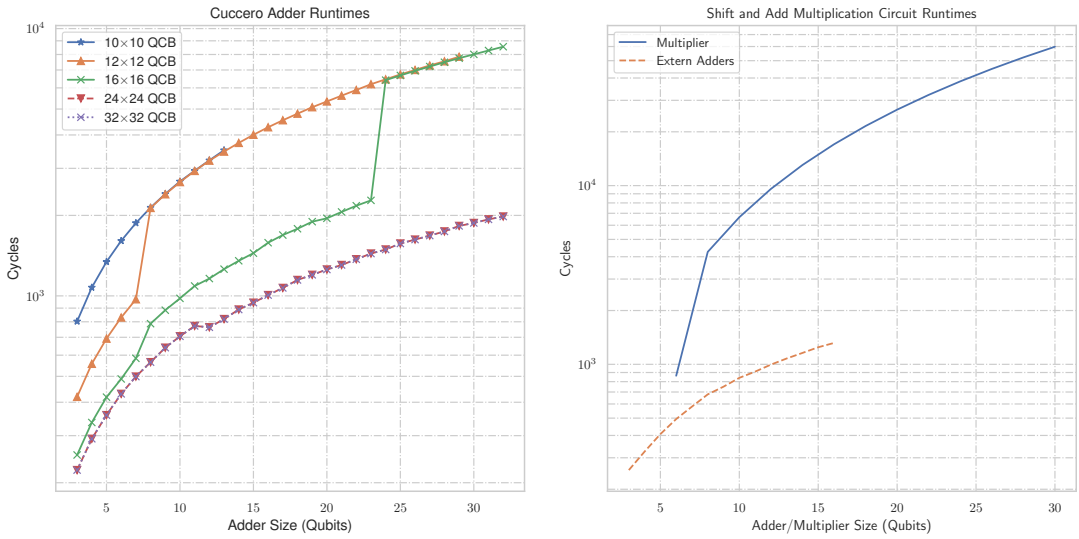



Figure 5.10: Cycle cost of addition and the multiplication circuits. (Left) The adder circuit's runtime depends on the number of magic state factories that are allocated. As this particular implementation of an adder is not particularly parallelisable the main constraint on the runtime is the number of register qubits. Once the number of register qubits grows to the point that it displaces a magic state factory the performance of the QCB degrades. (Right) The multiplication circuit is implemented using adder circuits as an external dependency. As the adders are performed linearly only a single addition extern is required. The program then comprises of a sequence of Toffoli gates implementing a controlled copy, followed by the addition extern. The Toffoli gates are supported by the inclusion of $T$ factory externs. The adder provides support for a multiplier circuit with an output register twice the size of the adder.

scope of the target register the output of the addition is raised by a integer power of two. The intermediary copy register may need to contain additional ancillae depending on the state of the target register.

If the output register is guaranteed to start in the $|0\rangle$ state, then each round of addition uses an adder of size $a$ or $b$. If this condition is not met then the round adder for a round $n$ should be of size $a+b+1-n$. This requires additional ancillae qubits for the intermediary copy register. If the output register starts in a state $\left|i > 2^{\max(a,b)}\right\rangle$, then the size of the output register needs to be increased by one.

Unsurprisingly, as a large part of the cost of the integer multiplication circuit is sequences of additions, the cost of the compiled multiplication circuit scales proportionally with the underlying adder. This can be seen in Section 5.3.1.

### 5.3.3 Integer Division

Our division circuit implements integer division via trial division. We make use of the same 'less than' gadget described in Section 4.4.2 . Our inputs to this circuit are a dividend of length $a$, a divisor of length $b$, and our outputs are the two inputs, along with a quotient of length $a-b+1$ and a remainder of length $a+1$. This circuit is similar to the implementation in [124].

Division begins by copying the top $a-b$ qubits from the dividend register to the remainder. Each round of the division circuit then compares whether the dividend or the remainder is larger, by subtracting one from the other. If the dividend is larger and the high bit is not flipped then the appropriate bit in the quotient is flipped, otherwise the subtraction is reversed. At the start of each round after the first another bit is added from the dividend to the remainder.

The performance of the division circuit can be seen in Figure 5.11. As the division circuit depends on repeated applications of the addition circuit, the same pattern of reduced performance is observed when the rate at which magic states are required exceeds the rate at which they are produced for a given QCB size.

Overall this continues the trend in which the runtime of a system is bound by the scale of the circuit against the available memory for magic state production.

## 5.4 Benchmarking QFT

The QFT circuit follows a similar pattern to the $Z(\theta)$ circuits. Each gate in the QFT circuit is a controlled $Z$ rotation. As $T^\dagger$ and $S^\dagger$ may be constructed from $T$ and $S$ by flipping the parity bit during their correction operation, these gates require the same number of cycles as their inverse. As $Z(\theta)$ is synthesised of self inverse gates, $T$ and $S$, $Z(\theta)$ requires the same number of cycles as $Z(-\theta)$.

As shown in Figure 2.11, each controlled $Z$ rotation is comprised of three $Z(\pm\theta/2)$ gates. A QFT is then a combination of synthesised $Z$ rotations, and a network of CNOT operations. The bottleneck on execution is then the rate at which magic states are required to support a collection of simultaneous $Z$ rotations.

The performance of the quantum fourier transform circuit on the QCB is shown in Figure 5.12. The register memory requirements an $n$ qubit QFT circuit increases linearly in $n$. Conversely

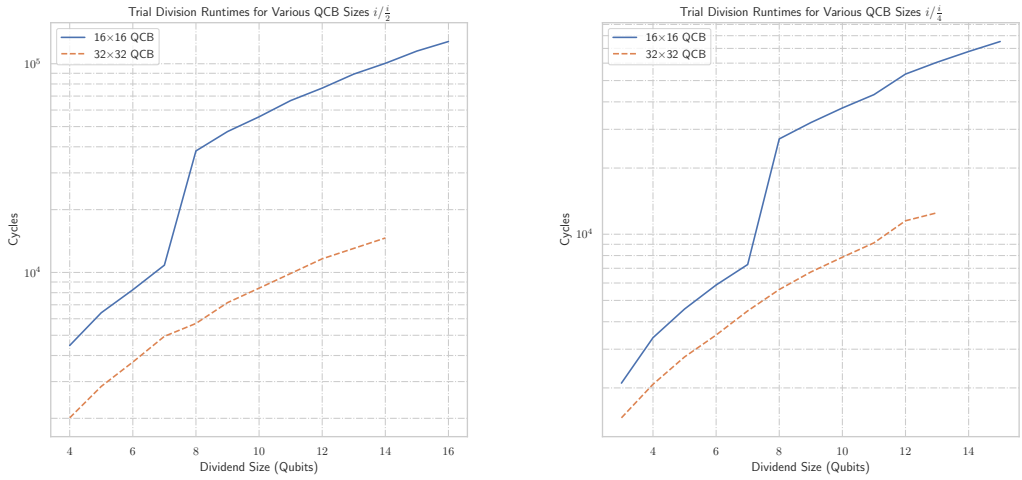



Figure 5.11: Performance of $i$ qubit division circuits for relative divisor register sizes of $i/2$ and $i/4$. As the trial division circuit depends on rounds of repeated addition and un-addition the same features from the performance of the addition circuit make a re-appearance.

the $T$ count increases with both the precision (where it depends on the angle) and the size of the QFT. As the precision of the QFT also depends on the size (with the smallest angle scaling as $2^{-n}$ ). Conversely the rate at which $T$ factories are required depends on the parallelisability of these gates.

## 5.5 QRAM

### 5.5.1 Bucket Brigade

The bucket brigade QRAM depends on a series of pairs of controlled swaps as seen in Figure 2.8c. Each of these control and routing circuits are implemented as an extern (the same extern, with different arguments passed to it). This bucket brigade 'gadget' does not scale with the size of the register or the address of the QRAM. However, as the controlled swap operations require a sequence of Toffoli gates as shown in Figure 2.7, the runtime of the gadget depends on the number of magic state factories that are available. This dependency can be seen in Figure 2.8c, where the performance of the QCB jumps stochastically when there is enough room to admit another magic state factory.

The performance of the bucket brigade qram itself also depends on the number of extern gadgets that are available. The construction of the routing network and the readout are highly parallelisable, as subsequent rounds of the routing network may begin while the current round is still being propagated. The readout is similarly parallelisable as it comprises of layers of $2^i$ independent routing blocks. The best case performance for the bucket brigade would then be $2nk + 2n$ gadgets for and address size of $n$ and a register size of $k$. Unfortunately the number of gadgets required to satisfy this performance scales exponentially in $n$. The runtime of the computation is then constrained by the size of the QCB, and the number of gadgets that can

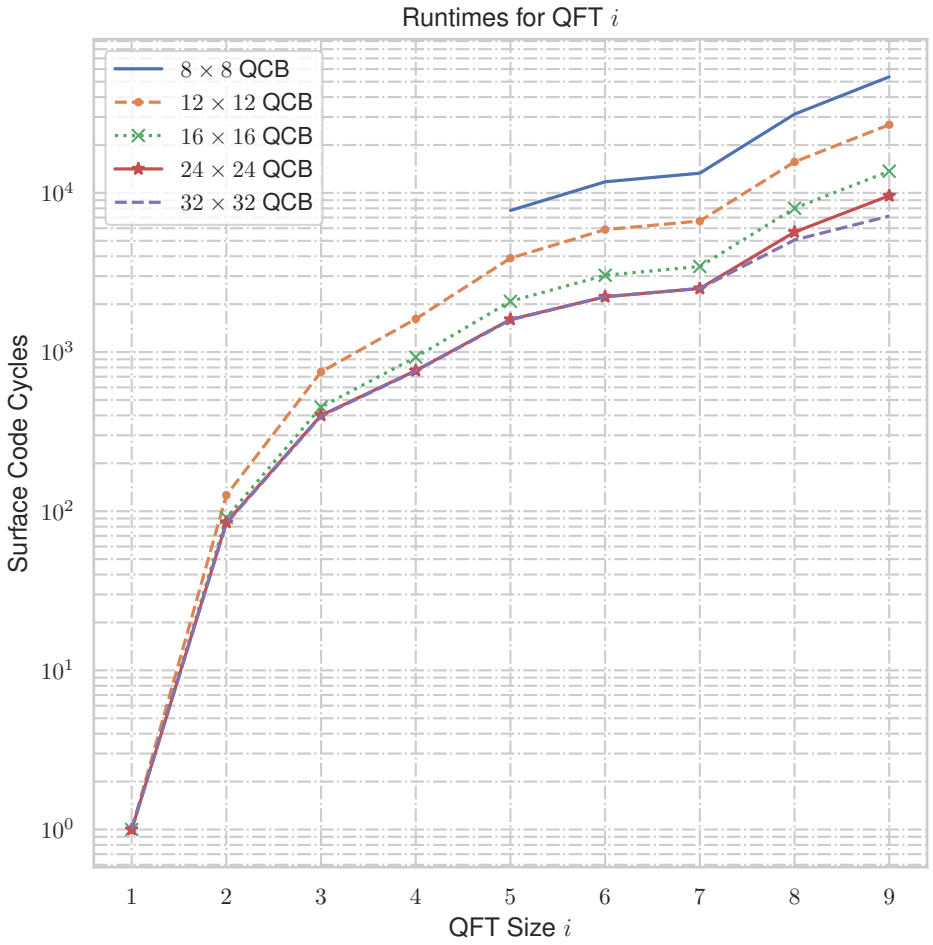



Figure 5.12: Surface code cycles required to implement a $i$ qubit QFT to precision $2^{-i}$ over a range of QCB sizes. The memory requirements of the QCB scale linearly, while the $T$ factory count increases in both the precision, and the size of the QFT.

be allocated. The scalability is further stymied by the exponentially increasing register size requirements; for the memory block $k2^n$ registers are needed, with another $k2^n$ for the routing network. These memory requirements may be reduced by performing the readout one qubit at a time, but this would increase the runtime by a factor of $k$.

Despite these large memory constraints, the advantage of the near duplication of the memory block is that the bucket brigade may be queried in parallel as described in Section 4.3.2.

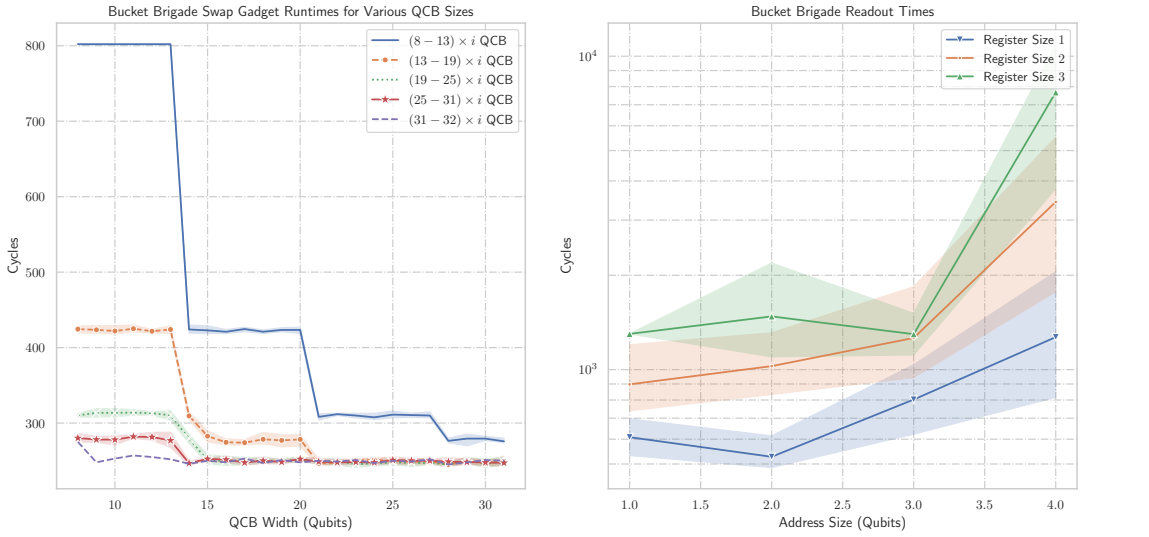



Figure 5.13: Bucket Brigade QRAM Gadget and Readout performance. The bucket brigade gadget is the set of controlled swaps in Figure 2.8c

### 5.5.2 Fanout and Swap

The fanout and swap QRAM circuit may be seen in Figure 2.9.

Similarly to the bucket bridgade, we construct a fanout and swap gadget, comprised of a set of $k$ controlled swaps. Each gadget takes in two banks of memory of size $k$ registers and a control ancillae, and performs the swap. With sufficient gadgets the runtime of the fanout and swap QRAM is then $2n$ gadgets. If there is not sufficient space to allocate $2^n$ gadgets, then similarly to the bucket brigade our performance is constrained by the rate at which externs are required compared to their avaliability.

Unlike the bucket brigade, the fanout and swap query is performed in place on the memory block rather than on a separate routing network. This reduced memory footprint admits additional room for externs, which in turn improves the runtime.

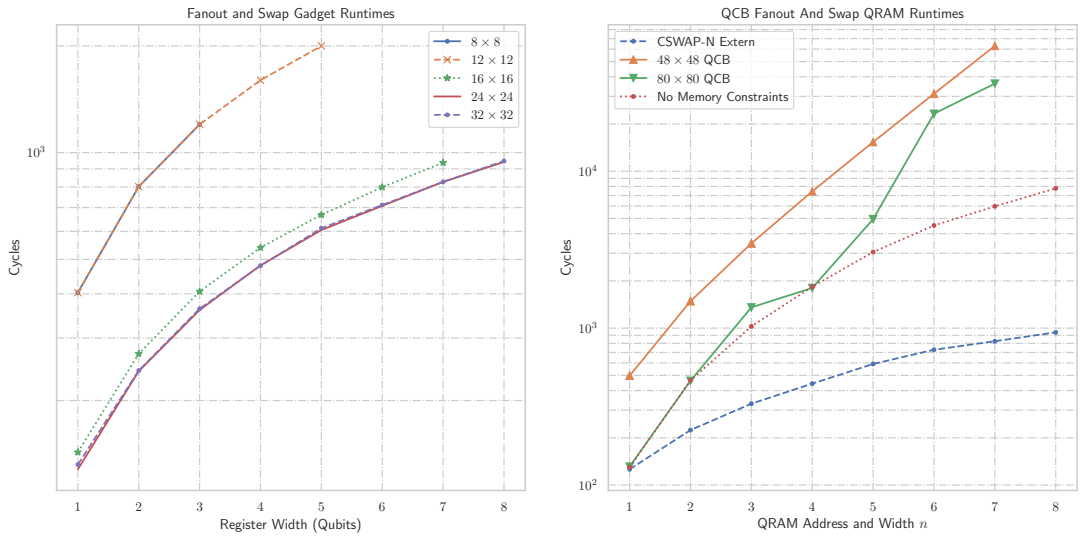



Figure 5.14: Runtime for fanout and swap QRAM gadgets and readout. Speedups from parallelism are constrained by the size of the QCB, and the number of gadgets that may be allocated. The gadget's size and runtime scales with the size of the registers in the QRAM, and the number of gadgets required scales with the size of the address.

This interaction may be seen in Figure 5.14. The $48x48$ QCB represents a highly constrained QCB. The $80x80$ QCB closely follows the limit with no memory constraints, until it is no longer possible to allocate enough gadgets. At which point it trends towards the much smaller constrained QCB.

## 5.6 Benchmarking The QCPU

In the previous sections we have benchmarked all components of the QCPU, excepting the logic unit. We will use the direct logic unit from Figure 4.5. The benchmark results from the compiler for the logic unit can be seen in Figure 5.15. The main bottleneck on the operation of the unit is the synthesis of the controlled Hadamard gate, constructed as a $Z(\pi/16)$ flanked by Cliffords. As the multi-controlled $Z$ rotation requires phase rotations on all qubits in the gate, this required the greatest rate of magic state factory production over the circuit. The structure of this gate may be seen in Figure 2.13. Where each of the rotation gates requires a set of independent magic states.

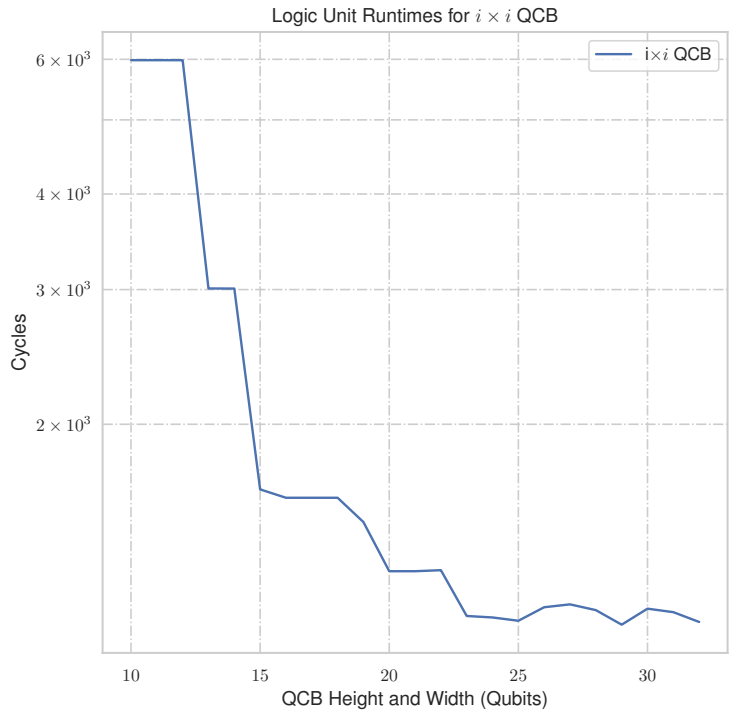



Figure 5.15

With all these components complete we may construct our QCPU benchmark. We will be implementing an eight qubit variant of the QCPU, such that all addresses and instructions are stored in eight qubit registers.

We will assume the availability of a $100 \times 100$ QCB for each independent memory region (the history buffer, QROM, QRAM) and a $200 \times 100$ QCB for the logic unit. This provides an eight qubit fanout and swap QRAM readout within $\mathcal{O}_{3857}$ cycles using a $32 \times 32$ gadget. This will need to be repeated to perform the fan-in.

The history region comprises of two controlled swaps conditioned on an eight qubit $I$ register. And one poisoned QRAM lookup. The program code lookup similarly consists of two $V$ gates each requiring an eight qubit multi-controlled $X$ gate, two sets of eight controlled swaps, and a QRAM lookup with an eight qubit address, and an eight qubit register width. From the benchmarks of these gates presented throughout this chapter the program code unit may be implemented in $\mathcal{O}_{9026}$.

The program control region requires four adders, that may be implemented in two parallel pairs, and three multi-controlled X gates with a large overlap that may be composed. This invokes a total cost of $\mathcal{O}_{1105}$ cycles.

The main memory is a QRAM lookup, that blocks only on the application of the logic unit. As it may be performed in parallel with the logic unit and the subsequent cycles' memory unit, its cost does not contribute to the overall runtime of the QCPU cycle.

The logic unit requires two eight qubit QRAM lookups that may be performed in parallel. A three address one qubit QRAM lookup, the logic unit benchmarked above, two three-control, one-qubit swaps and five eight control $X$ gates for inter-register operations. Using the benchmarks from above, this comes out to $\clock_{9392}$ cycles.

The history unit may be operated simultaneously with the logic unit from the previous cycle. Indeed the runtime of the logic unit exceeds that of the history unit when both have the same address size. As the history unit's addressing scheme is independent of the rest of the QCPU this is not universally true.

This gives a total runtime for one QCPU cycle of $\clock_{19523}$. The main takeaway is that despite the interesting concepts afforded by the QCPU architecture, the need for a constant overheads on the order of $\clock_{20000}$ to implement a programmatic Hadamard gate are likely to render any implementation of the design infeasible.

## 5.7 Discussion and Improvements to the Compiler

The primary result of the benchmarks was effectively a restatement of Amdah's law. Rather than speedups being due to an increase in the number of cores, intead the underlying model of costing an algorithm assumes an arbitrarily scaling parrallelism, which is undermined by the shared dependency on routing ancillae and magic state factories.

In a number of cases the compiler missed opportunities for further optimisation. The first was an occasional tendency to underestimate the number of magic state factories required to implement a circuit. This indicates that improvements to the allocator heuristic may be required.

A more concrete concern is that the composability of DAG objects and externs may introduce dependencies that impede instances where gate commutivity would admit parallel operations. For example if a qubit acts as a control on two multi-controlled $X$ gates these operations could be interleaved in any order. Instead the DAG enforces a strict ordering that may result in the register blocking execution of the remainder of the program.

One solution would be to introduce the concept of 'immutable borrows' to the compiler. Extern and DAG operations that guarantee that the state of the qubit is unchanged throughout their entire operation (effectively acting as 'read only operations') would not be bound by strict DAG ordering. Similarly performance improvements may be gained by implementing gate cancellation, and commutation passes [87, 25]. Another approach is to more directly map ZX calculus operations into the compiler. Or provide a direct decomposition to Litinski slices[87, 10, 109].

More recent magic state factory models [55] produce multiple $T$ states per operation. Implementing such a factory in the current compiler would require that the programmer specify which factory outputs are assigned to which gates. While defeating the purpose of automatic compilation, this would also introduce the possibility of deadlocks as resources would be unable to be released while holding conflicting locks on multiple externs. One solution would be to implement a more general producer-consumer pattern for factories, splitting them more explicitly from externs.

Lastly the implementation in Python leads to constraints on the maximum QCB and circuit depths that may be processed. The construction of a sufficiently large mapping tree trips Python's hard coded maximum stack size, while a large enough QCB runs it out of RAM altogether. Typically this occurs on the order of 10000 to 250000 surface code patches, while the maximum stack size limit depends on the allocation of routes within the QCB.

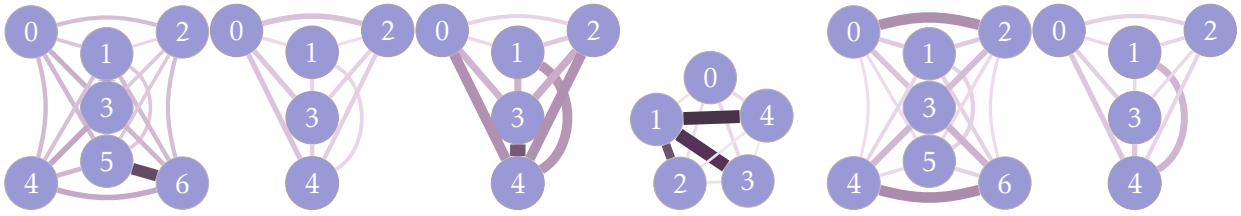
A more time and memory efficient implementation may provide dividends, however this particular model of QCPU will almost certainly remain impractical.

# Chapter 6

# Coupling Map Calibration

A common feature of quantum error correction is levying measurements to project errors back to the code space [58]. This prospectively involves the repeated operation of simultaneous measurements over the same set of physical qubits every cycle. In this chapter we will discuss the mitigation of correlated measurement errors on current NISQ devices.

Several approaches have been taken to the problems of characterising, suppressing[92, 91, 4, 61], mitigating [67, 75, 98], and correcting errors[58] in quantum systems. As quantum devices require regular recalibration, long-term characterisation is not always possible as the exhibition of errors on these systems drifts over time [68]. This characterisation is essential for mitigating errors in quantum systems. Once characterised, a variety of techniques may be deployed to attempt to suppress the associated errors, and in the long term approach a fault tolerant threshold.




(a) IBMQ Oslo (b) IBMQ Lima (c) IBMQ Quito (d) IBMQ Manila (e) IBMQ Nairobi (f) IBMQ Belem

Figure 6.1: Frobenius norm between calibration matrices $|C_{ij} - C_i \otimes C_j|_2$ for all pairs of qubits over a range of IBM quantum devices averaged over three weeks. Thicker edges indicate a greater correlation of measurement errors between those qubits. Values range [0:1].

Current state-of-the-art characterisation techniques include randomised benchmarking [92], tomography [13, 75, 4], heuristic methods [61], and particle filtering techniques [66, 62]. In this chapter, we focus on suppressing two categories of significant errors widely observed on today's superconducting NISQ devices: *state-dependent measurement errors* and *correlated measurement errors*. Prior approaches [123, 1] to suppress these errors have been hemmed in by the competing goals of scalability and complexity of characterising correlated errors. More recent

Bayesian approaches [37] are constrained by sampling requirements on devices that exhibit inherently correlated, non-uniform and non-Markovian noise [16]. An example of the complexity of this error landscape can be seen in Fig. 6.1. Here the physical layout of the qubits is represented by their proximity in the figure, while the correlation of measurement errors is represented by the thickness of the edges.

To address this challenge, we propose a *Coupling Map Calibration (CMC)* scheme, which represents a middle ground approach that efficiently stitches together small measurement error calibration patches and preserves the information about local correlations. To extend this approach, we also propose a coupling map profiling method *ERR* which constructs maps of the highest locally-correlated qubits measurements which may then be passed to CMC. In summary, this chapter makes the following contributions:

- We demonstrate a method for efficiently constructing and joining sparse measurement calibration matrices to form a full-device measurement calibration matrix. From this we construct CMC: a scheme for performing joint sparse calibrations using the coupling maps of NISQ devices;
- We extend this method to account for local but non-coupling map aligned error models with a patching technique termed *ERR*. We demonstrate that ERR maps are stable on the order of several weeks;
- Our experimental results in implementing CMC and CMC-ERR on a range of IBMQ devices reduce errors by an average of 35% and up to 41%, equalling or outperforming existing scalable methods and are consistent with expectations of measurement errors from simulation. We additionally demonstrate that the choice of measurement error mitigation technique depends upon the noise-profile and architectural topologies of the device.

## 6.1 A Brief introduction to Quantum Noise

### 6.1.1 Types of Quantum Noise

Given the probabilistic nature of quantum systems, even a small amount of noise results in a non-zero probability of obtaining an incorrect measurement outcome. To mitigate this quantum circuits are executed multiple times in order to collect statistics and attempt to reconstruct an 'error-free' output. Each iteration may be referred to as a 'shot' or a 'trial'. This repetition adds a multiplicative factor to the complexity of the execution of the quantum circuit. If the error rate is sufficiently high, it may not be possible to distinguish the output distribution from the noise. Errors may be characterised as being due to interactions with the environment [125], imperfect gate operations or, as we focus on in this chapter, due to imperfect state preparation and measurement (SPAM).

**Gate Errors** describe errors due to imperfect gate operations. As measurement is probabilistic any imperfect gate will result in a non-zero probability of producing an incorrect output. This noise will accumulate as the circuit depth grows.

**SPAM Errors** relate to the incorrect preparation and measurement of the system. Unlike the environmental and gate errors, they only occur during the preparation and measurement stages of the circuit and do not scale with circuit depth, though they do scale with the size of the

quantum register, and the number of measurements.

### 6.1.2 State-Dependent, Biased and Correlated Errors

In addition to the causes of errors on quantum devices we may also consider how errors act on quantum states. An example error is an independent and identically distributed (IID) uniform random Pauli error over all qubits. This is the same as applying the operation $E(p) = (1-p)I + \frac{p}{3}(X+Y+Z)$ independently to each qubit in the system for some error rate $p$. Other types of errors are not as impartial. In this chapter we will focus on two significant categories of NISQ errors: state-dependent errors and correlated errors, both of which have been observed on NISQ devices [123, 67, 37].

**State-Dependent** errors exist where the error rate associated with an operation depends on the state of the qubit [123] with a particular focus on state-dependent errors that occur during measurement. During the relatively long period of time required for measurement on NISQ superconducting devices, the rate of decay from the $|1\rangle$ state is greater than the rate at which qubits in the $|0\rangle$ state are spontaneously excited, producing a state-dependent error.

**Correlated Errors** occur when the probability of an error acting on two qubits is greater than the product of the probabilities with which the error would act on either qubit individually [67, 37]. As most modern quantum devices only admit two-qubit gates between a subset of the qubits on the device, this network of admissible two-qubit relations forms a *coupling map*. It has been observed that correlated errors tend to occur in close proximity on the coupling map[67].

### 6.1.3 Measurement Errors on IBMQ Devices

We focus on state-dependent and correlated measurement errors as currently exhibited on IBMQ devices [67, 123, 37]. State-dependent measurement errors are still present in the most current versions of IBMQ systems as demonstrated in Fig. 6.2. Here a single qubit has been prepared in the $|0\rangle$ state before a sequence of $X$ gates have been applied. As $X$ is a unitary operation, if an odd number of $X$ gates are applied the expected state is $|1\rangle$, while an even number should result in the $|0\rangle$ state. Transpiler optimisations have been disabled to prevent the compiler reducing these gate sequences to either a single $X$ gate or no gate at all.

If measurement errors were state independent we would expect that the error rate would increase exponentially with the circuit depth. Instead we observe that the error rate of the $|1\rangle$ states is significantly higher than that of the $|0\rangle$ states even as circuit depth increases, indicating the presence of state-dependent measurement errors. These results suggest that state-dependent measurement errors dominate both single-qubit gate errors and environmental noise up to the quantum volume of the device.

Additionally correlated measurement errors are present on current IBMQ devices as seen in Fig. 6.1. Here we compare the difference between two single-qubit calibrations $C_i \otimes C_j$ and two-qubit calibrations $C_{ij}$. If the measurement errors were entirely independent then these two calibrations would be equal. This demonstrates that not only do correlated measurement errors exist on these devices, but that some appear to persist between calibration cycles. Previous work has demonstrated that these correlated errors tend to occur in physical proximity on the

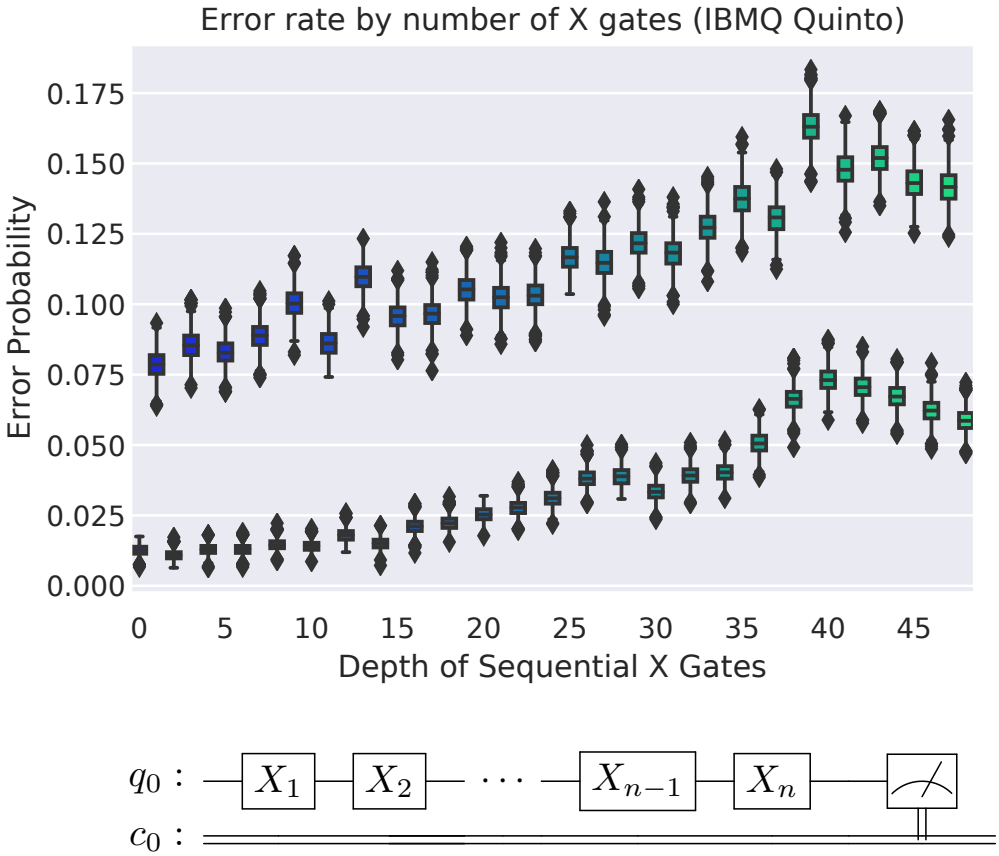



Figure 6.2: Error probability over 4000 shots of and Quito device following the application of a sequential number of X gates. Measuring a $|1\rangle$ state exhibits a significantly higher error rate than the $|0\rangle$ state indicating a state-dependent measurement error.

| Method | Computational Cost | Output |
|---|---|---|
| Process Tomography [75, 4, 95] | $r4^n$ | SPAM + Gate Errors |
| Complete Calibration [1] | $r2^n$ | SPAM Errors |
| Tensored Calibrations [1] | $2nr$ | Non-correlated SPAM Errors |
| Randomised Benchmarking [91, 92] | Poly($n$) | Average SPAM and Gate |
| Pauli / Clifford Twirling [67] | Poly($n$) | SPAM-Free Errors |
| AIM [123] | $4r$ | Average Biased SPAM |
| SIM [123] | $2nr + kr$ | Top $k$ least biased SPAM |
| JIGSAW [37] | $\frac{nk}{2}$ + k | Bayesian Error Distribution |
| MThree [99] | $2nk$ | Sparse Tensored Errors |
| *CMC (This Paper)* | $\frac{4}{k}er$ | Local SPAM Errors |

Table 6.1: Computational cost for a range of error characterisation methods. $n$ is the number of qubits and $r$ is the number of repetitions required of the circuit. The $k$ for AIM is a pre-chosen constant, typically 4. For the coupling map method (CMC), $e$ is the number of edges in the graph while $k$ is the speed-up from performing non-local patches simultaneously.

device [67].

Previous approaches in error mitigation have involved two often distinct lines of inquiry: the characterisation of the noise of the system [66, 92, 61, 68], and the design of the system to reduce noise [123, 111]. Often this may be an iterative approach as errors must be characterised before they can be suppressed or mitigated.

The output from an execution of a quantum system is a single sample from a larger probability distribution. In order to obtain any meaningful statistics to characterise the device, an ensemble of measurements must be taken. Measurement perturbs the state of a quantum system, so a diminishing amount of information is obtained by performing multiple measurements on the same qubit. Cloning a quantum state is also not permitted [130] which prevents "copying" a state and performing the same operations on the copies. As a result the most common approach to quantum characterisation is to repeatedly prepare a state and then perform measurements on it [13]. The primary figure of merit in comparing approaches to characterising quantum systems is the requisite number of circuits that must be executed. In this section we briefly sketch the details of a range of schemes that seek to mitigate measurement errors on quantum devices. These schemes trade between quantum computational overheads and the scope and quality of the data.

### 6.1.4 Tomography

One of the first and most accurate methods for characterising quantum states and processes, including errors, is *tomography* [61, 4, 75]. As the number of measurements required to perform tomography scales exponentially with the number of qubits, these approaches have become increasingly infeasible on recent devices. Tomography is performed by repeating an experiment over a set of orthogonal measurement bases to reconstruct the density state of the system prior to measurement [64, 4]. As a consequence of this exponential scaling, tomography provides the most accurate reconstruction of the state, and hence the most accurate profile of the noise [75,

95].

The insight that ties process tomography to error characterisation is that errors are themselves quantum processes that act on the device. By performing process tomography over a portion of the system, environmental, gate and SPAM errors can be diagnosed. i.e. an error is simultaneously an error and an operation that evolves the state and can hence be characterised. Process tomography provides a description of the error channel that incorporates information about state-dependent and correlated errors. Once an error $\mathcal{E}$ is characterised and if the matrix describing $\mathcal{E}$ is invertible, then we can apply $\mathcal{E}^{-1}$ to mitigate the error. The downside is the cost: as the number of qubits increases, repetitions of $4^n$ circuits become computationally unfeasible for modern quantum devices, let alone future devices with hundreds to thousands of qubits.

The reconstruction of a quantum state from a set of measurements is termed *quantum state tomography* [75, 4]. By taking a histogram of the measurement results over a complete basis of $2^n$ measurement operators, the resulting probability distribution can be used to estimate the quantum state [64]. To perform state tomography, we begin with a set of measurement operators $M$ and a density matrix $\rho$. Born's rule gives $\vec{p}_i = \mathrm{Tr}(M_i^\dagger M_i \rho)$. Where $p_i$ is the probability of obtaining a measurement result $M_i$ from the state $\rho$. This value can be determined experimentally by running the circuit and calculating the frequency with which $M_i$ is observed. The number of times the circuit is executed is termed the number of *shots*. A greater number of shots typically corresponds to a better estimate of $p_i$. By constructing a matrix $A = M_0^\dagger M_0 \otimes M_1^\dagger M_1 \ldots \otimes M_m^\dagger M_m$, we can derive $A(\rho^{\otimes m}) = \vec{p}$. Inverting $A$ then reconstructs the density matrix $\rho$ given the measurement outcomes $\vec{p}$.

The Choi-Jamiołkowski isomorphism may be used in open quantum systems to derive a map from states to processes [129, 61]. This forms the basis of *quantum process tomography*. By selecting $2^{2n}$ linearly independent inputs $\rho_j$ we can characterise the action of a process $V$ on the system. The approach is much the same as state tomography, except we now perform the gate we are characterising prior to measurement $\vec{p}_{ij} = \mathrm{Tr}(M_i^\dagger M_i V \rho_j V^\dagger)$.

### 6.1.5 Measurement Error Calibration

A more limited form of tomography can be used to characterise and invert measurement errors for devices with a fixed measurement basis (such as the IBM quantum devices) [1, 100]. For $n$ qubits, we construct a set of circuits that prepares and measures each of the $2^n$ possible states in the measurement basis. The columns of this *calibration matrix* $C$ (a subset of $\mathcal{E}$) contain the probabilities with which each measurement outcome was observed, while the rows correspond to the state that was prepared. This calibration matrix now describes the map from expected measurement states to observed measurement states, and hence describes the effect of measurement errors in the measurement basis. This calibration matrix can then be inverted and applied to mitigate noise on the device. This technique provides the most accurate characterisation of measurement errors, but comes with an exponential overhead in the number of calibration circuits.

If we assume that measurement errors are independent then our calibration matrix is linearly separable; $C_{1,2,\ldots n} = C_1 \otimes C_2 \ldots \otimes C_n$ and can be constructed with only $2n$ calibration circuits. This method will characterise state-dependent measurement errors, however no information will be gained about multi-qubit correlated errors. Taking the assumptions of linear indepen-

dence further, we can perform all of our calibrations using only two circuits; $I^{\otimes n}$ and $X^{\otimes n}$. The individual calibration matrices $C_i$ can be recovered from the joint calibration matrix by marginalising the contributions of the other qubits to the probability distribution; which takes the form of the normalised partial trace $|\mathrm{Tr}_j(C_{ij})| \approx C_i$. This technique is limited as NISQ devices exhibit measurement crosstalk [37] and as a result measurement errors are not independent.

**MThree** [99] is an approximate matrix inverse method, similar to the Qiskit full and linear approaches. Mthree sidesteps the scalability issues with these methods by heavily truncating the calibration matrix using $2n$ calibration circuits, and by assuming that non-correlated errors are dominant. Further speedups are gained by skipping the explicit construction of the calibration matrices. MThree performs strongly in an uncorrelated regime but is known to overcorrect in the presence of correlated errors.

### 6.1.6 Randomised Benchmarking (RB)

RB is a fundamentally different technique to tomography based calibration. A set of random circuits with the overall action $I$ of varying lengths are constructed. Each circuit is executed, and the probability of measuring $|0\rangle^{\otimes n}$ state dictates the average error rate of that circuit. The error rate is a function of the circuit depth, and by fitting error rates from random circuits of varying lengths we can estimate the average gate and SPAM errors on the device. Although it requires far fewer operations than tomographic methods, randomised benchmarking cannot distinguish correlated and state-dependent errors. This average error rate is useful for device profiling, but is not as useful for mitigating errors.

**Pauli / Clifford twirling** [67] is an extension of Randomised benchmarking that approximates the error channel of a quantum device as a *Gibbs random field* (GRF) [67]. It uses the difference between the fit to the GRF and the observed distribution to determine the incidence of correlated errors on the device. This method scales polynomially in the number of qubits $n$. Pauli twirling provides an efficient approximation of the SPAM free errors, but as a result does not include biased or correlated measurement errors.

### 6.1.7 Circuit Specific Strategies

Another class of approaches involves attempting to profile and minimise the noise at the whole circuit level. These methods depend on the choice of circuit and must be re-run for each new circuit, in contrast with schemes that directly characterise measurement errors on a device.

**Static Invert and Measure** (SIM) [123] targets state-dependent measurement errors on a particular circuit and tackles scalability by restricting itself to exactly four characterisation circuits. The circuit that is being characterised is executed, and prior to measurement one of the following four operators are applied: $I^{\otimes n}$, $X^{\otimes n}$, $(I \otimes X)^{\otimes \frac{n}{2}}$ and $(X \otimes I)^{\otimes \frac{n}{2}}$. The results of these circuits are then averaged to attempt to mitigate the action of any state-dependent error on the output. SIM may average over correlated measurement errors that align with its characterisation circuits.

**Adaptive Invert and Measure** (AIM) [123] follows a similar strategy to SIM in applying characterisation operators after the action of a target circuit. It increases the pool of characterisation

circuits and attempts to determine a set of $k$ 'correct' strings to use for averaging. It begins with $\frac{r_1 n}{2}$ characterisation circuits of the form $I^{\otimes 2i} \otimes X^{\otimes 4} \otimes I^{\otimes n-2i}$ for even $i$ in the range 0 to $\frac{n}{2}$ and applies them prior to measurement on a target circuit. These 'patch' circuits are then effectively acting on sets of four qubits at a time with an overlap of two qubits between different patches. The outputs of these circuits are then averaged and the top $k$ characterisation circuits are selected and used in a further $r_2 k$ executions which are in turn averaged to produce the final output. This selection mechanism assumes that some elements of those top $k$ bit strings are improving the success probability of the circuit.

**JIGSAW** [37] is another circuit specific strategy. Unlike AIM and SIM, it targets correlated measurement errors by constructing Bayesian filters. Each filter is constructed by randomly dividing the set of measured qubits into 'patches'. The measurement distribution of each patch then forms a sub-table which acts as a local Bayes filter on a global measurement distribution. These filters are applied by splitting the global measurement distribution and updating elements corresponding to the patch. This subset is then normalised and rejoined with the global distribution.

While JIGSAW is currently a state of the art method in scalable measurement error suppression, it suffers from inconsistent results: should one of the sub-tables contain a single value, the renormalisation of that subset will promote it to a state that is measured with probability 1. When convolved with the global measurement distribution, this can result in JIGSAW erroneously over-reporting states that occur with low probability. There are two avenues by which this can occur: (1) if a measurement outcome in the 'global' table is sufficiently distant from all other outcomes, or (2) if not all results are measured in the subset circuit. The impact of this inconsistency then depends on the choice of subset circuits, the distribution of measurement results, the associated noise of the device and the order in which sub-tables are applied.

## 6.2 model

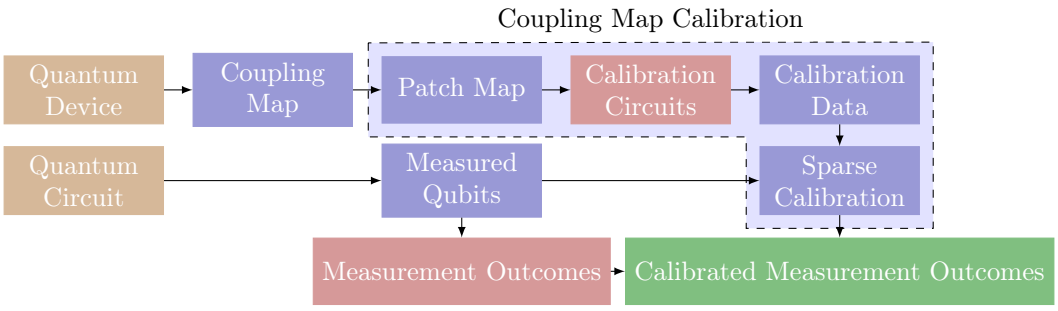



Figure 6.3: Coupling Map Calibration scheme for measurement error mitigation using patched calibrations. The scheme converts the coupling map of the device into a series of calibration patch circuits which are executed to collect data on two qubit calibrations. These calibrations can then be joined or traced over as required to mitigate measurement errors for any other circuit that is executed on the device. Red boxes indicate executions on a quantum device, blue elements are classical operations, orange inputs and green outputs.

From the limitations of the current state of the art techniques discussed in the previous section, we derive three design objectives:

1. Characterising both correlated and state-dependent measurement errors on NISQ devices;
2. Maintaining a polynomial number of characterisation shots in the number of qubits; and
3. Demonstrating an improvement in the error rate reduction compared to the other polynomial methods, ideally achieving similar rates of error mitigation as calibration techniques.

To achieve this, we propose a novel coupling map calibration (CMC) scheme, as outlined in Figure 6.3. Previous work [67] has established that the majority of correlated errors on modern quantum devices occur within physical proximity on the device. Leverage this insight we construct a set of sparse calibration matrices for each edge on the coupling map with each of these calibration edges termed a 'patch'. By joining these sparse matrices we can then reconstruct an approximate global calibration matrix, which can then be used for measurement error mitigation. While the number of measurement circuits required to construct a full calibration matrix scales exponentially with the number of qubits, CMC only requires four such circuits for each edge and hence scales linearly in the number of edges on the coupling map. By the same argument we can perform non-local calibration circuits simultaneously and trace out the individual results. The number of calibration circuits is then reduced as a function of the sparsity of the coupling map.

### 6.2.1 Coupling Map Calibration Patches

Under the same assertion that correlated errors are physically correlated [67], physically distant qubits should not exhibit correlated errors. In other words, $C_{ij} = C_i \otimes C_j$ if $i$ and $j$ are sufficiently distant qubits. By the same argument, for pairs of adjacent qubits $ab$ and $cd$ that are sufficiently distant $C_{abcd} = C_{ab} \otimes C_{cd}$ if $ab$ and $cd$ are sufficiently distant pairs of qubits. From this, $C_{ab} = |\mathrm{Tr}_{cd}(C_{abcd})|$ and $C_{cd} = |\mathrm{Tr}_{ab}(C_{abcd})|$. If $ab$ and $cd$ are distant edges on the coupling map, then we can calculate $C_{ab}$ and $C_{cd}$ simultaneously.

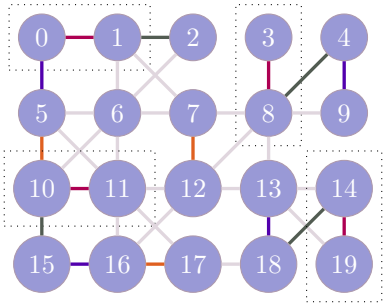



Figure 6.4: Example measurement patches on the IBM Tokyo device with two qubits per patch and a distance between patches of at least one qubit and where each patch is represented by a colour. This pattern could be continued until all edges are incorporated in at least one patch.

For this construction, we assume that all correlations are local on the device up to some dis-

**Algorithm 21** Greedy Distance $k$ Patch Construction

```
patch_construct(E, k)
Copy E to E'
Initialise an empty list C
while E' is not empty do
    Initialise empty lists C_i, B
    Pop e from E' and append it to C_i and B
    Mark all elements of E as unvisited
    while Not all vertices in E are visited AND B is not empty do
        Mark all elements of B as visited
        Perform a depth k BFS from each element in B, mark all vertices of up to distance k as 'visited'
        Replace B with the depth k boundary of the BFS
        for each edge b in B do
            if If b is not in E' AND b has not yet been visited then
                Add b to C_i
                Mark b as visited
                Perform a depth k BFS from each element in B, mark all vertices of up to distance k as
'visited'
    Append C_i to C
return C
```

tance $k$. Given the connectivity graph of the device, we can construct a set of 'calibration patches' containing $n$ qubits such that all edges on the graph are included in at least one patch. We can then calibrate these two patches simultaneously without an increase in the number of shots (i.e., number of circuits executed on the quantum devices).

From this strategy, the total number of shots per calibration matrix is $4r$, and the total number of calibration matrices is approximately a constant fraction of the number of edges in the coupling map, which in turn is typically on the order of the number of qubits on the device. For example, in the case of the IBM Tokyo device, the number of edges is 3-4 times the number of qubits. A $k = 1$ patching strategy reduces the number of circuits by approximately the same factor. This can be seen in Fig. 6.4. We present a greedy construction for these patches in Algorithm 21. This approach iteratively finds a set of patches separated by at least distance $k$ until all edges of the graph are contained in a set. Each patch within a set may then be constructed simultaneously. The total cost in the number of measurements required is then only four times the number of such sets. For the Tokyo device, this corresponds to 40 calibration circuits for all qubits individually, 140 calibration circuits to characterise each edge individually, 54 circuits for coupling map patching, 760 circuits for all pairs of qubits, and $2^{20}$ circuits for the full calibration.

When tested on large random coupling maps (> 100 qubits) with an average of four edges per qubit, this greedy approach reduced the number of calibration circuits by between a factor of 3 to 10. This style of coupling map (albeit not randomly constructed) is representative of the coupling maps of current IBMQ, Google, Rigetti, IonQ, and Honeywell devices [7, 122, 113, 1].

### 6.2.2 Joining Calibration Matrices

The core of our model is finding a joint approximation of two overlapping local calibrations. Two disjoint calibrations may be joined via the tensor product:

$$C_{ij} = C_i \otimes C_j \tag{6.1}$$

However, this necessarily cannot capture any information regarding correlated errors between those two qubits. In order to capture correlated errors we must construct calibration circuits from the combination of all measurement operators across both qubits. Expanding this problem to $n$ qubits on a device scales as $2^n$. Ideally we would wish to perform two qubit calibrations with four calibration circuits each and stitch together these individual 'patches' as seen in Fig. 6.5. This raises the problem of how to join two-qubit calibration matrices with overlapping qubits.

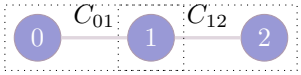



Figure 6.5: Overlapping patches of two qubit calibrations; the goal of the method is to join these calibrations to approximate $C_{012}$.

As calibration matrices are normalised maps between probability distributions, we can split a calibration matrix formed by the outer product of two other calibration matrices using the partial trace operation and normalising such that the sum of each column in the resulting calibration matrix is 1 as shown by

$$C_i = |\mathrm{Tr}_j(C_i \otimes C_j)|. \tag{6.2}$$

For a calibration matrix acting on multiple qubits, we can construct an approximate single-qubit calibration matrix using the same method

$$C_i \approx |\mathrm{Tr}_j(C_{ij})|. \tag{6.3}$$

If we have multiple multi-qubit calibration matrices all acting on the same qubit then we need to construct a technique by which we can apply each calibration jointly. For $v$ calibration matrices overlapping on some qubit labelled $j$, we can construct an approximation of the calibration matrix between qubits $i$ and $j$ (with some order parameter $v_a$) by

$$C'_{ij}(v_a) = \left(I \otimes C_j^{\frac{v-1-v_a}{v}}\right)^{-1} C_{ij} \left(I \otimes C_j^{\frac{v_a}{v}}\right)^{-1} \tag{6.4}$$

such that $\mathrm{Tr}_i(C'_{ij}) \approx C_j^{\frac{1}{v}}$ and $\mathrm{Tr}_j(C'_{ij}) \approx \mathrm{Tr}_j(C_{ij})$. This construction can be seen in Fig 6.6.

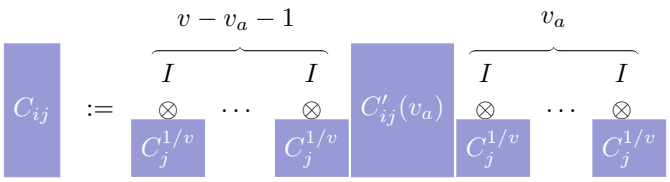



Figure 6.6: Construction of $C'ij(v_a)$ given $C_{ij}$ and a choice of order parameter $v_a$ and a total number of calibration matrices to join over this index such that $|\mathrm{Tr}_i(C'_{ij})| \approx C_j^{\frac{1}{n}}$, and $\mathrm{Tr}_j(C'_{ij}) \approx \mathrm{Tr}_j(C_{ij})$. Other pairwise calibrations acting on $j$ with their own order parameters $v_{0\ldots v-1}$ will fill out the other $v-1$ terms.

Similarly, for some other qubit $k$ with order parameter $v_b$ we can construct

$$C'_{jk}(v_b) = \left(C_j^{\frac{v-1-v_b}{v}} \otimes I\right)^{-1} C_{jk} \left(C_j^{\frac{v_b}{v}} \otimes I\right)^{-1} \tag{6.5}$$

such that if $v_1 > v_0$ and $v = 2$

$$C'_{ijk} = (I \otimes C'_{jk})(C'_{ij} \otimes I). \tag{6.6}$$

This allows us to patch together a joint calibration matrix from the overlapping individual calibrations given an explicit order.

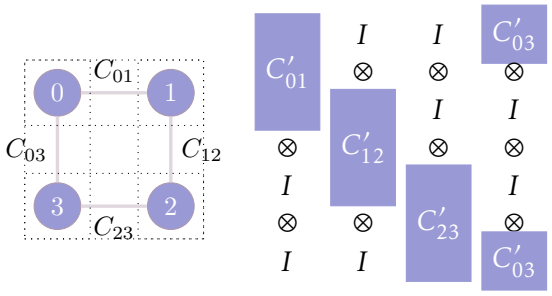



Figure 6.7: A spanning set of two qubit patches incorporating all edges in the graph and the form of the associated calibration matrix. Columns represent tensor product while rows represent matrix products. Each individual column is itself a sparse matrix. For large systems (with matrices scaling with the number of qubits as $2^n \times 2^n$) it is faster to do a series of sparse matrix vector multiplications than it is to perform a single dense matrix multiplication.

This method can be extended to joining calibration matrices of arbitrary sizes and with overlaps by selecting the appropriate order parameter for each contributing term in $C'$. An example of this for a square plaquette of connected qubits can be seen in Fig. 6.7.

Armed with the ability to approximate larger calibration matrices from smaller ones, we can now build a calibration matrix for the system. To construct this, we need a spanning set of calibration matrices of $n$ qubits across the system, before using the method above to construct a set of calibrations that may be applied linearly.

As correlated errors tend to be spatially close on the coupling map [67], the question then becomes one of deciding which spanning set of calibration matrices should be used. For a base

application of CMC we have opted to use edges of the coupling map as the spanning set of calibration matrices.

### 6.2.3 CMC: Mitigating Measurement Errors

Using this ability to construct and join sets of calibration matrices we may now consider how to wrangle this into a measurement error mitigation scheme. We start with the coupling map of the device and the set of qubits that we are measuring. If the measured qubit $i$ has no neighbours, then we can construct the effective calibration matrix from the patches $C_{ij}$ by tracing out the non-measured qubits $j$ such that $C'_i = |\prod_j \mathrm{Tr}_j(C_{ij})^{\frac{1}{v}}|$. If instead qubit $i$ shares a non-zero number of edges we follow the method discussed in section 6.2.2 and trace out over edges shared with non-measured qubits while multiplying the shared edges. In both cases, the order in which these matrices are applied must be identical to the order in which the calibration matrices were constructed as seen in Fig. 6.6.

Once this set of sparse calibration matrices have been constructed, we can then invert each matrix then take the tensor product with the identity for each non-participating qubit. By reversing the order of these inverted matrices we have constructed the inverse of the calibration matrix. This set of inverted calibration matrices can now be applied to a measurement distribution in order to perform the desired measurement error mitigation. We term this scheme CMC.

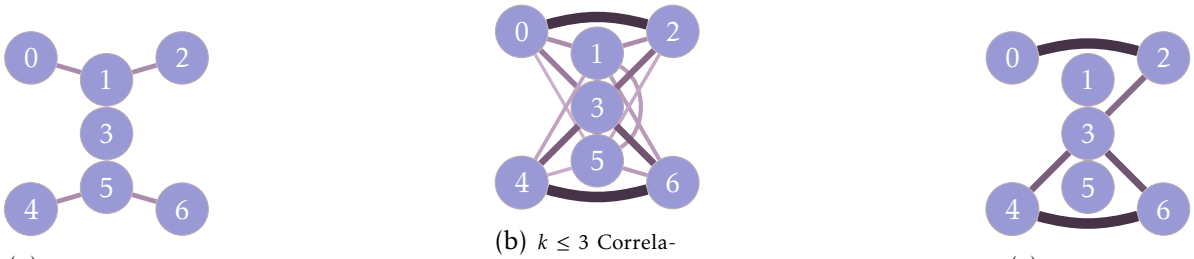
If the number of qubits in a calibration patch is significantly less than the number of qubits in the system, then each individual calibration matrix will be sparse. The maximum number of entries in the measurement vector is bounded by the number of shots performed on the system, and can be periodically culled of very low weight entries. In the regime of a 50+ qubit system, applying a series of sparse matrix-vector products is much more performant than a $2^n \times 2^n$ dense full calibration matrix.

### 6.2.4 ERR: Device Tailored Mitigations

As seen in Fig. 6.1, while highly correlated errors tend to be local, they do not necessarily align to the coupling map. If a device exhibits a systemic correlated error that persists through multiple calibrations, we may modify our coupling map method to instead characterise the noisiest edges. The goal is to construct a coupling map with at most $n$ edges while attempting to maximise the number of highly correlated measurement errors. We present *ERR* an $O(n^2)$ greedy approach to this problem in Algorithm 22 with a graphical example in Fig. 6.8. Here $E$ is the initial coupling map, and $k$ is a locality parameter, such that only two-qubit edges of distance less than $k$ are considered. Note that unlike the computational coupling map there is no requirement that this error coupling map be connected. We can then perform CMC over this error coupling map to construct our sparse measurement calibration matrices.

**Algorithm** 22 Error Coupling Map Construction

error_coupling_map_construct$(E, k)$
Construct all 1 and 2 qubit calibration matrices $C_i$, $C_{i<j}, \forall |i, j| < k \in E$
Calculate all edge weights $w_{i,j} = \|C_i \otimes C_j - C_{ij}\|$
Sort edge weights $W$ in descending order $w_{i,j}$
Initialise an empty graph $E'$
**for** $w_{i,j}$ in $W$ and $|E'| < n$ **do**
  **if** $i$ in $E'$ and $j$ not in $E'$ **then**
    Add $j$ and edge $i, j$ to $E'$
  **if** $j$ in $E'$ and $i$ not in $E'$ **then**
    Add $i$ and edge $i, j$ to $E'$
  **if** $i$ and $j$ not in $E'$ **then**
    Let $w_i$, $w_j$ be the next weighted edges containing $i$ and $j$
    Add $\min_w((i, w_i), (j, w_j))$ and dangling edge $i, j$ to $E'$
return $E'$




[t!] (a) Coupling Map

(b) $k \leq 3$ Correlations

(c) Error Map

Figure 6.8: Construction of a $k = 3$ local error map using ERR.

## 6.3 Methodology

This section will discuss the application and methodology of the designs discussed in Section 6.2 and the construction of the circuits that are executed on the IBMQ devices. These circuits will be used to compare SIM, AIM, Full calibration, Linear calibration, CMC, CMC-ERR, MThree and JIGSAW.

We conducted experiments on the IBMQ, 'Quito' 'Lima', 'Manila', and 'Nairobi' devices. The relevant figures of merit are the overall error rate and the total number of quantum device executions. The goal is to minimise the error rate for a fixed number of measurement shots when comparing circuit selection and averaging techniques (AIM/SIM) against classical post-processing approaches (calibration methods). We define the success probability the one norm distance between a simulated noiseless measurement distribution and a noisy distribution of measurement outcomes.

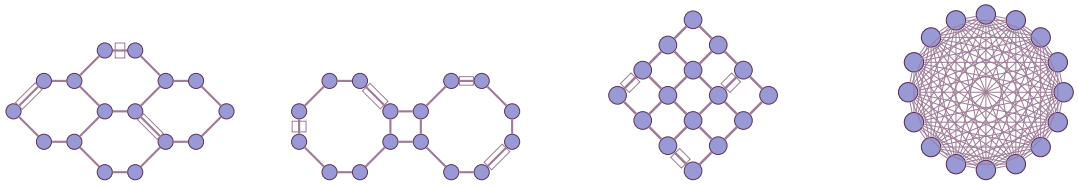
To compare the various measurement error mitigation techniques, we focus on the GHZ benchmark. This benchmark is the smallest circuit that entangles all qubits on a device. We additionally perform QAOA and BV benchmarks on the IBMQ devices.

### 6.3.1 Benchmark Error Simulations

The first benchmarks are a set of simulated measurement errors over a range of states. The simulated error channels exhibit varying amounts of state-dependent, uniform, and correlated errors. These diagrams indicate the probability with which a particular input state is mapped to an output state by the measurement error channel. The size of the squares scales with the probability of that operation. By constructing combinations of these error channels, we compare different mitigation methods against a range of scenarios relating purely to measurement errors. Using these measurement error channels, we can construct errors that probe the responsiveness of different error mitigation techniques to correlated and state-dependent errors. We can also determine the scalability of each of these methods in terms of the total number of shots required to produce a consistent result.

Simulations are then performed by applying the matrices associated with each gate and measurement operator to construct the ideal, noiseless, measurement output vector. We then apply the constructed measurement error channel to this output vector, producing a vector of the probabilities of measuring each state given the measurement error channel. This distribution can then be sampled to produce simulated measurement results.

In addition to these tailored measurement errors, we construct a set of simulated backend devices as seen in Fig. 6.9. The topologies of these simulated devices emulate the coupling map architectures of a range of modern quantum devices. We use these architectures along with the Qiskit statevector simulator [1] to evaluate GHZ circuits for varying device sizes. For these simulations, we set a one qubit gate error rate of 0.1%, a two qubit gate error rate of 1%, and a readout error rate between 2 – 8%, and state-dependent measurement errors between 2 – 8% for each qubit for both $|0\rangle \rightarrow |1\rangle$ and $|0\rangle \rightarrow |1\rangle$. $T_1$ and $T_2$ times are set to infinity. These single and two-qubit error rates are approximately analogous to those exhibited by the current NISQ devices; for example, one calibration of the IBM Quito device reports an average of two-qubit error rate as 0.98%, an average single-qubit $H$ error rate of 0.03% and readout errors between 3% and 7%.



(a) Hexagonal (IBM Washington) [1] (b) Octagonal (Rigetti Aspen) [85] (c) Grid (Google Sycamore) [7] (d) Fully Connected (IonQ Forte) [132]

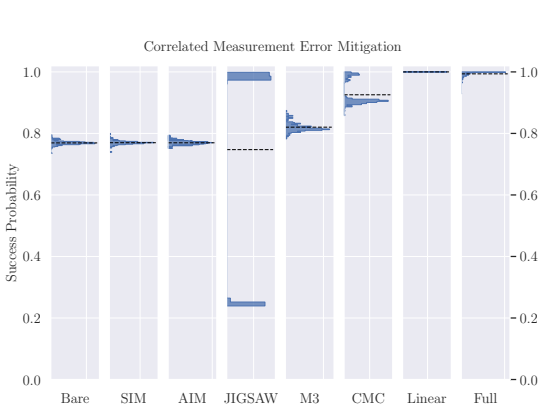
Figure 6.9: Simulated device backends represent the coupling maps from a range of modern NISQ architectures. Devices are constructed where each qubit has a random state-dependent measurement error for both $|0\rangle \rightarrow |1\rangle$ and $|1\rangle \rightarrow |0\rangle$ in the range of 2% to 8%. The boxed edges indicate an example patch on this device.

### 6.3.2 NISQ Device Benchmark Circuits

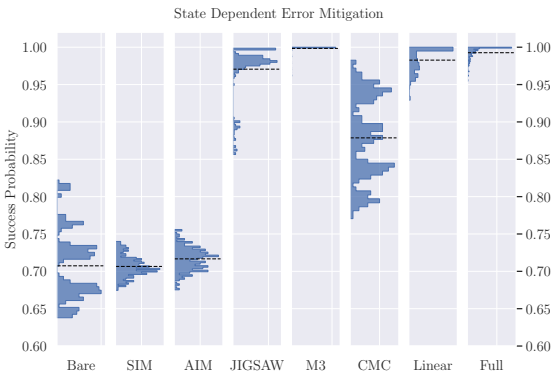
Additionally, we benchmark the performance of these measurement error mitigation methods on IBM quantum devices using GHZ circuits. These represent the smallest full device entangling circuits, minimising the impact of gate errors while producing a non-classical distribution of measurement results. The GHZ circuits are constructed by performing a single-qubit Hadamard gate on qubit 0 of the device, and then performing a breadth first search of two-qubit CNOT operations across the coupling map. This construction ensures that there is no advantage gained by different qubit allocations, routing methods or other compiler optimisations.

## 6.4 Evaluation

### 6.4.1 Simulated Measurement Errors




(a) Correlated Measurement Error Mitigation




(b) State-Dependent Measurement Error Mitigation

Figure 6.10: (a) A sample correlated measurement error mitigation. (b) A sample state-dependent measurement error mitigation. Dashed lines indicate the average of each distribution. The 'Bare' column shows the error distribution without before any mitigations are applied. The bifurcation of JIGSAW occurs where due to the pathological nature of the error model the sub-tables may be missing entries, leading to errors during renormalisation.

We benchmarked a range of measurement error mitigation methods over 136000 trials using a pair of known measurement error operations applied to the full set of $2^n$ computational basis states over four qubits, with results shown in Fig. 6.10. Each mitigation method is afforded an equal number of measurements of the quantum system to perform any calibrations and operations of the circuit. As the classical input state is known, the success probability is the frequency that the classical output state is reported. 'Bare' represents the distribution of errors without any mitigation. The correlated measurement error model applies only two qubit correlated errors. The state-dependent error model applies single qubit state-dependent errors, as a result the $|0\rangle^{\otimes n}$ state experiences no errors at all. AIM and SIM perform their

averaging techniques, which has no overall effect for correlated errors, and narrows the distribution of state-dependent errors. With these focused error models, JIGSAW suffers from an empty sub-table entry with high probability, and should not be considered representative of JIGSAW's performance on more rounded error models. CMC performs well in the presence of both classes of errors, but is outperformed by the Linear and Full methods, which require exponential overheads. As the total number of shots was constrained, the Full model suffers from slight sampling errors leading to a tail in its distribution.

JIGSAW's performance in this benchmark does highlight the problem where renormalisation can promote low probability errors. This can be seen with a circuit that simply prepares and measures the $|0000\rangle$ state and with an error channel composed of weight two correlated errors acting with some probability $\lambda$ between qubits 0, 3 and 1, 2 and JIGSAW patches between qubits 0, 1 and 2, 3. The global PMF in this case would be $\left|0000 : 1-2\lambda \quad 0110 : \lambda \quad 1001 : \lambda\right|$. This problem arises where due to the sparsity of the global PMF as each row contains only a singular value during the coefficient update. These updated coefficients are renormalised, and so the local coefficient table becomes $\left|00|00 : 1 \quad 01|10 : 1 \quad 10|01 : 1\right|$. At this stage all error information from the global PMF has been irrecoverably discarded, increasing the weight of errors independently of $\lambda$.

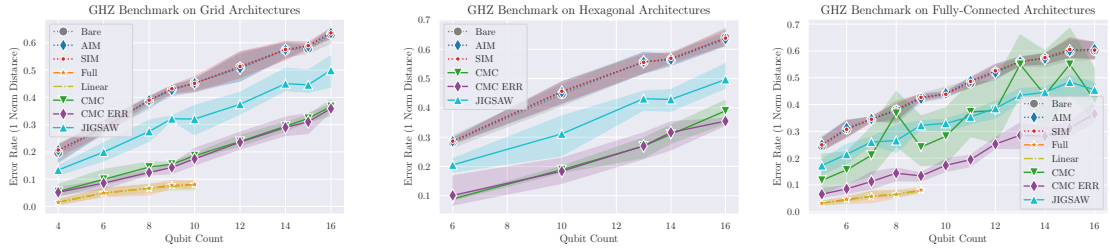
### 6.4.2 Simulated Architectures

Using the architectures described in Fig. 6.9, we consider the performance of our range of error mitigation techniques as a function of the topology and scale of the device. In each trial each method is provided with a fixed 16000 shots to reconstruct a GHZ state from the output of the simulated device. As the Qiskit statevector simulators limit readout errors on a per-qubit basis, the noise in these experiments is biased but not correlated.

In Fig. 6.11a and Fig. 6.11b that CMC and CMC-ERR performs the best of the non-exponential methods over grid and hexagonal architectures. The Full and Linear methods provide the greatest reduction in one-norm distance, but require exponential overheads which limits their application to low qubit counts. AIM and SIM are nearly indistinguishable from the bare error rate. JIGSAW outperforms the averaging methods, but is in turn outperformed by CMC. As the statevector simulator's measurement errors are applied on a per-qubit basis, this guarantees the linearity of the error model. As a result the linear calibration performs as well as the full calibration over these simulations up to the difference in sampling during characterisation.

In the case of the fully connected topology seen in Fig. 6.9d, the number of edges scales quadratically with the number of qubits. As a result the CMC method begins to suffer from a reduced number of shots per coupling map patch, which significantly reduces the accuracy of the method, as can be seen in Fig. 6.11c. For this dense coupling map JIGSAW slightly outperforms CMC, and would be expected to significantly outperform it for larger devices. CMC-ERR outperforms both CMC and JIGSAW, using the constraint of $n$ edges to mitigate the quadratic scaling issues of base CMC.

For completeness, we include the octagonal topology with the same error model as the other families of devices. At 16 qubits, JIGSAW achieves a 23% reduction over the baseline error rate, while *CMC* reduces the error rate by 37%. For the same octagonal device, AIM and SIM are within 1% of the initial error rate.




(a) GHZ error rates for square grid connectivity: Fig. 6.9c.

(b) GHZ error rates for hexagonal connectivity: Fig. 6.9a.

(c) Error rates for full connectivity: Fig. 6.9d.

Figure 6.11: Error rates for GHZ state preparation for families of simulated devices with either square-grid connectivity (Fig. 6.11a), hexagonal connectivity (Fig. 6.11b), or full connectivity(Fig. 6.11c). Each method is permitted 16000 shots with which to reconstruct a $\mathrm{GHZ}_n$ state over $n$ qubits. This poses some problems for the CMC method as the number of edges for the complete coupling map scales quadratically with the number of qubits.

| **Method** | GHZ Circuit Benchmarks | | | |
|---|---|---|---|---|
| | Manila - 5 | Lima - 5 | Quito - 5 | Nairobi - 7 |
| Bare | $0.20^{+0.10}_{-0.04}$ | $0.23^{+0.02}_{-0.01}$ | $0.26^{+0.01}_{-0.01}$ | $0.56^{+0.02}_{-0.01}$ |
| Full | $0.10^{+0.11}_{-0.04}$ | $0.07^{+0.02}_{-0.01}$ | $0.04^{+0.02}_{-0.01}$ | N/A |
| Linear | $0.11^{+0.02}_{-0.02}$ | $0.06^{+0.02}_{-0.01}$ | $0.08^{+0.02}_{-0.02}$ | N/A |
| AIM | $0.18^{+0.03}_{-0.02}$ | $0.23^{+0.01}_{-0.01}$ | $0.26^{+0.01}_{-0.01}$ | $0.57^{+0.01}_{-0.01}$ |
| SIM | $0.19^{+0.03}_{-0.02}$ | $0.23^{+0.01}_{-0.01}$ | $0.27^{+0.01}_{-0.01}$ | $0.62^{+0.01}_{-0.02}$ |
| JIGSAW | $0.18^{+0.06}_{-0.06}$ | $\mathbf{0.17^{+0.06}_{-0.05}}$ | $0.21^{+0.04}_{-0.04}$ | $0.52^{+0.19}_{-0.23}$ |
| *CMC* | $0.14^{+0.09}_{-0.05}$ | $\mathbf{0.17^{+0.01}_{-0.04}}$ | $\mathbf{0.16^{+0.01}_{-0.01}}$ | $0.64^{+0.02}_{-0.06}$ |
| $CMC_{ERR}$ | $\mathbf{0.13^{+0.08}_{-0.03}}$ | $0.25^{+0.03}_{-0.03}$ | $0.17^{+0.10}_{-0.03}$ | $\mathbf{0.33^{+0.41}_{-0.12}}$ |

Table 6.2: GHZ benchmarks of the 1-norm distance between output distribution and ideal $|+\rangle^{\otimes n}$ state. The relative performance of CMC and $\mathrm{CMC_{ERR}}$ depends on the 'uniformity' of the error distributions for that device as seen in Fig. 6.1 The best non-exponential method is bolded in each column.

### 6.4.3 NISQ Device Benchmarks

Lastly we compare the performance of these methods on physical quantum devices. IBM Manila and Nairobi have local but non-coupling map aligned correlated errors, while Lima and Quito have locally uniform error profiles, as seen in Fig. 6.1. From this we would expect a relative advantage for CMC-ERR on Manila and Nairobi, and for CMC on Lima and Quito. This is supported by the results in Table 6.2. Each method is allocated 32000 shots to perform both calibration and any required circuit executions. For all five qubit devices exponential methods achieve the best performance. However, at the seven qubit mark these methods begin to encounter scaling issues, with the Full calibration approach exceeding 100 calibration circuits. Of the non-exponential methods CMC and CMC-ERR beat or match JIGSAW on all devices.

| Method | QAOA Circuit Benchmarks | | |
|---|---|---|---|
| | Manila - 5 | Lima - 5 | Nairobi - 7 |
| Bare | $0.29^{+0.04}_{-0.03}$ | $0.26^{+0.03}_{-0.02}$ | $0.77^{+0.13}_{-0.10}$ |
| Full | $0.23^{+0.03}_{-0.03}$ | $0.24^{+0.09}_{-0.04}$ | N/A |
| Linear | $0.22^{+0.01}_{-0.01}$ | $0.23^{+0.05}_{-0.04}$ | N/A |
| AIM | $0.37^{+0.01}_{-0.01}$ | $0.44^{+0.03}_{-0.03}$ | $0.85^{+0.07}_{-0.04}$ |
| SIM | $0.38^{+0.02}_{-0.01}$ | $0.42^{+0.04}_{-0.02}$ | $0.86^{+0.08}_{-0.07}$ |
| JIGSAW | $0.28^{+0.05}_{-0.06}$ | $0.26^{+0.17}_{-0.13}$ | $0.79^{+0.19}_{-0.16}$ |
| MTHREE | $0.23^{+0.01}_{-0.01}$ | $0.23^{+0.04}_{-0.03}$ | $\mathbf{0.70^{+0.13}_{-0.11}}$ |
| *CMC* | $0.24^{+0.02}_{-0.04}$ | $0.19^{+0.05}_{-0.04}$ | $0.77^{+0.13}_{-0.16}$ |
| $CMC_{ERR}$ | $\mathbf{0.23^{+0.01}_{-0.02}}$ | $\mathbf{0.13^{+0.04}_{-0.04}}$ | $0.77^{+0.14}_{-0.16}$ |

Table 6.3: Max-cut QAOA benchmarks of the 1-norm distance between output distribution and ideal state.

| Method | BV Circuit Benchmarks | | |
|---|---|---|---|
| | Manila - 5 | Lima - 5 | Nairobi - 7 |
| Bare | $0.34^{+0.14}_{-0.26}$ | $0.33^{+0.13}_{-0.19}$ | $0.67^{+0.19}_{-0.23}$ |
| Full | $0.18^{+0.21}_{-0.18}$ | $0.16^{+0.13}_{-0.16}$ | N/A |
| Linear | $0.20^{+0.19}_{-0.20}$ | $0.14^{+0.14}_{-0.14}$ | N/A |
| AIM | $0.29^{+0.09}_{-0.16}$ | $0.25^{+0.06}_{-0.12}$ | $0.59^{+0.15}_{-0.28}$ |
| SIM | $0.31^{+0.12}_{-0.15}$ | $0.29^{+0.03}_{-0.11}$ | $0.61^{+0.12}_{-0.16}$ |
| JIGSAW | $\mathbf{0.16^{+0.11}_{-0.13}}$ | $\mathbf{0.13^{+0.07}_{-0.10}}$ | $\mathbf{0.45^{+0.33}_{-0.34}}$ |
| MTHREE | $0.20^{+0.19}_{-\mathbf{0.20}}$ | $0.16^{+0.14}_{-\mathbf{0.16}}$ | $0.50^{+0.28}_{-0.29}$ |
| *CMC* | $0.26^{+0.13}_{-0.22}$ | $0.26^{+0.14}_{-0.17}$ | $0.54^{+0.24}_{-0.26}$ |
| $CMC_{ERR}$ | $0.23^{+0.16}_{-0.22}$ | $0.24^{+0.13}_{-0.16}$ | $0.58^{+0.21}_{-0.26}$ |

Table 6.4: BV benchmarks of the 1-norm distance between output distribution and a set of randomly sampled integer states.

On the Bernstein-Vazirani (BV) benchmarks in table 6.4 we see that the gate noise is sufficiently dominant that Jigsaw performs the best. Lastly, on the QAOA benchmarks in table 6.3 CMC performs best on the five qubit benchmarks while gate noise is sufficiently high at the seven qubit mark that few significant improvements in the error rate are achieved by any method.

These results reveal a strong association on the performance of each method with the underlying error model of the device. Correlated errors on IBMQ-Nairobi are almost anti-aligned with the device's coupling map, as seen in Fig. 6.8. This explains the poor performance of base CMC on this device which failed to characterise most of the correlated errors. Similarly JIGSAW, which relies on calibrating against random pairs of qubits, performs best on relatively even error distributions. In several instances JIGSAW's best case matched CMC or CMC-ERR's best case, but had a worse average performance due to its reliance on the randomised calibration pairs. CMC-ERR exhibited a 41% reduction in the average error rate on IBMQ Nairobi, demonstrating the efficacy of tailoring correlated error mitigation methods to device noise profiles.

Overall we observe that M3 provides the best performance and scalability for non-correlated measurement errors, JIGSAW's inclusion of gate noise provides advantages in some situations and CMC best handles correlated errors.

## 6.5 Discussion

### 6.5.1 Scalability

The IBMQ implementations of the Full and Linear calibration matrices quickly encounter scaling issues. These methods require the construction of a $2^n \times 2^n$ matrix. The Full method performs this using $2^n$ calibration circuits. For $n > 10$ it becomes unfeasible to queue and execute all the required calibration circuits. Alongside the constraints of constructing this matrix we must also consider the classical computational overheads. For example at $n = 14$ the dense calibration matrix (using 32 bit floating point representation) occupies 32GB of RAM. Calculating the inverse of this matrix and applying it to the sparse output vector of the measurement results is computationally unfeasible. By comparison using a naïve COO sparse matrix representation with CMC, 32GB affords us 32 qubits. More memory efficient sparse matrix constructions may

be made given the regular structure of the CMC matrices.

For calibration matrix methods (Full, Linear and CMC), as long as the error profile of the device does not drift significantly, it is possible to apply the same calibration matrix to multiple circuits executed on the same device. Circuit dependent methods (AIM, SIM, and JIGSAW) must be re-run from scratch for every new circuit that is executed. We find that ERR characterisations are stable for a given device on the order of weeks between significant recalibrations.

### 6.5.2 Performance and Stability

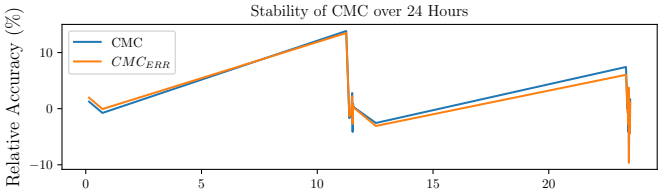



Figure 6.12: The performance of CMC on IBMQ Quito over the space of 24 hours since it was constructed benchmarked using a GHZ circuit. The reported accuracy is the relative difference in accuracy, representing an absolute error rate of between 15% and 19%.

A pure python implementation of CMC characterises 20 qubits in 60 seconds on a quad core i7-6700 CPU. Further performance improvements may be obtained as each sparse matrix may be permuted to $C_{ij} \otimes I^{\otimes n-1}$, it is possible to construct sparse block diagonal forms of each matrix with only 16 unique coefficients.

Fig. 6.12 demonstrates that CMC is stable for at least a day after it is first constructed. This limits the number of repeated calibrations when compared to circuit specific methods.

### 6.5.3 Linearity of Edge Counts on NISQ Devices

The main concern for the scalability of CMC is the number of edges in the coupling map. A non-sparse coupling map increases the number of two qubit calibrations that are performed and decreases the number of calibration patches that may be performed simultaneously. Table 6.5 shows the number of edges as a function of the number of qubits for a range of modern architectures.

IonQ's fully-connected devices [132](Fig. 6.9d) are the only family of architectures with a greater than linear growth in the number of edges, hence it is not scalable to perform bare CMC over this fully connected graph. The construction of CMC-ERR avoids this scaling problem bounds that the total number of calibration patches by the number of qubits.

In this chapter we have demonstrated CMC and CMC-ERR, which are novel methods for efficiently constructing and joining sparse and scalable measurement calibration matrices. These methods do not increase the number of measurement shots on the device. Results on IBMQ devices have achieved up to a 41% reduction in the error rate of the circuit, averaging at 35% over the experiments performed. Additionally our simulations demonstrate that CMC outperforms all other non-exponential methods on correlated error simulations. Our results demonstrate a strong association between the performance of measurement error mitigation methods with

| Architecture | Example | Edge Count($n$) |
|---|---|---|
| Linear | Honeywell H1 [113] | $n-1$ |
| Grid(Fig. 6.9c) | Google Sycamore [7] | $2n+c+r$ |
| Local Grid(Fig. 6.4) | IBM Tokyo [1] | $4n+c+r$ |
| Hexagonal(Fig. 6.9a) | Rigetti ACORN[122] | $(n-1)+cr$ |
| Heavy Hex(Fig. 6.9a) | IBM Washington[1] | $(n-1)+cr$ |
| Octagonal(Fig. 6.9b) | Rigetti ASPEN[85] | $\frac{3n}{2}-2r-2c$ |
| Fully Connected(Fig. 6.9d) | IonQ Forte[132] | $\frac{1}{2}(n^2-n)$ |

Table 6.5: Edge count as a function of the number of qubits $n$, rows $r$ and columns $c$ for a range of device architectures. For all architectures $n \geq rc$. All architectures excepting IonQ's grow the number of edges linearly with the number of qubits.

the underlying error model of the device. Future work may involve merging the sparse linear error mitigation techniques of M3 with the improved correlated error mitigations of CMC, and enhancing CMC's matrix inversions with better stochastic matrix inverse methods. All code required to replicate these results can be found in the corresponding git repository.

# Bibliography


[1] Gadi Aleksandrowicz et al. *Qiskit: An Open-source Framework for Quantum Computing*. Version 0.7.2. Jan. 2019. DOI: 10.5281/zenodo.2562111. URL: https://doi.org/10.5281/zenodo.2562111.

[2] Yuri Alexeev et al. "Quantum Computer Systems for Scientific Discovery". In: *PRX Quantum* 2 (1 2021), p. 017001. DOI: 10.1103/PRXQuantum.2.017001. URL: https://link.aps.org/doi/10.1103/PRXQuantum.2.017001.

[3] Panos Aliferis, Daniel Gottesman, and John Preskill. *Quantum accuracy threshold for concatenated distance-3 codes*. 2005. arXiv: quant-ph/0504218 [quant-ph].

[4] J. Altepeter, D. James, and Paul Kwiat. "Quantum State Tomography". In: 2017.

[5] Dong An and Lin Lin. *Quantum linear system solver based on time-optimal adiabatic quantum computing and quantum approximate optimization algorithm*. 2019. arXiv: 1909.05500 [quant-ph]. URL: https://arxiv.org/abs/1909.05500.

[6] Srinivasan Arunachalam et al. "On the robustness of bucket brigade quantum RAM". In: *New Journal of Physics* 17.12 (2015), p. 123010. DOI: 10.1088/1367-2630/17/12/123010.

[7] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5. URL: https://doi.org/10.1038%2Fs41586-019-1666-5.

[8] Alán Aspuru-Guzik et al. "Simulated Quantum Computation of Molecular Energies". In: *Science* 309.5741 (2005), pp. 1704–1707. ISSN: 0036-8075. DOI: 10.1126/science.1113479. eprint: https://science.sciencemag.org/content/309/5741/1704.full.pdf. URL: https://science.sciencemag.org/content/309/5741/1704.

[9] Ryan Babbush et al. "Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity". In: *Phys. Rev. X* 8 (4 2018), p. 041015. DOI: 10.1103/PhysRevX.8.041015.

[10] Niel de Beaudrap and Dominic Horsman. "The ZX calculus is a language for surface code lattice surgery". In: *Quantum* 4 (Jan. 2020), p. 218. ISSN: 2521-327X. DOI: 10.22331/q-2020-01-09-218. URL: http://dx.doi.org/10.22331/q-2020-01-09-218.

[11] David Beckman et al. "Efficient Networks for Quantum Factoring". In: *Physical Review A* 54.2 (Aug. 1996), pp. 1034–1063. DOI: 10.1103/PhysRevA.54.1034. (Visited on 10/12/2016).

[12] Dominic W. Berry et al. "Simulating Hamiltonian Dynamics with a Truncated Taylor Series". In: *Phys. Rev. Lett.* 114 (9 2015), p. 090502. DOI: 10.1103/PhysRevLett.114.090502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.114.090502.

[13] D.W. Berry et al. “How to perform the most accurate possible phase measurements”. In: *Phys. Rev. A* 80 (5 2009), p. 052114. DOI: 10.1103/PhysRevA.80.052114. URL: https://link.aps.org/doi/10.1103/PhysRevA.80.052114.

[14] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. “Surface Code Compilation via Edge-Disjoint Paths”. In: *PRX Quantum* 3.2 (May 2022). ISSN: 2691-3399. DOI: 10.1103/prxquantum.3.020342. URL: http://dx.doi.org/10.1103/PRXQuantum.3.020342.

[15] Michael E. Beverland et al. *Assessing requirements to scale to practical quantum advantage*. 2022. arXiv: 2211.07629 [quant-ph].

[16] Robin Blume-Kohout et al. *Wildcard error: Quantifying unmodeled errors in quantum processors*. 2020. DOI: 10.48550/ARXIV.2012.12231. URL: https://arxiv.org/abs/2012.12231.

[17] Alex Bocharov, Martin Roetteler, and Krysta M. Svore. “Efficient Synthesis of Universal Repeat-Until-Success Quantum Circuits”. In: *Phys. Rev. Lett.* 114 (8 2015), p. 080502. DOI: 10.1103/PhysRevLett.114.080502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.114.080502.

[18] Matthias F. Brandl. “A Quantum von Neumann Architecture for Large-Scale Quantum Computing”. In: (2017). eprint: arXiv:1702.02583. URL: https://arxiv.org/abs/1702.02583.

[19] Sergey Bravyi and Jeongwan Haah. “Magic-state distillation with low overhead”. In: *Physical Review A* 86.5 (Nov. 2012). ISSN: 1094-1622. DOI: 10.1103/physreva.86.052329. URL: http://dx.doi.org/10.1103/PhysRevA.86.052329.

[20] Sergey Bravyi and Alexei Kitaev. “Universal quantum computation with ideal Clifford gates and noisy ancillas”. In: *Physical Review A* 71.2 (Feb. 2005). ISSN: 1094-1622. DOI: 10.1103/physreva.71.022316. URL: http://dx.doi.org/10.1103/PhysRevA.71.022316.

[21] Sergey Bravyi et al. “Mitigating measurement errors in multi-qubit experiments”. In: *arXiv* (2020).

[22] Andrew C. Bray, Adrian C. Dickents, and Mark A. Holmes. *The Advanced User Guide for the BBC Microcomputer*. The Cambridge Microcomputer Centre, 1983.

[23] Benjamin J. Brown et al. “Poking Holes and Cutting Corners to Achieve Clifford Gates with the Surface Code”. In: *Physical Review X* 7.2 (May 2017). ISSN: 2160-3308. DOI: 10.1103/physrevx.7.021029. URL: http://dx.doi.org/10.1103/PhysRevX.7.021029.

[24] Vera von Burg et al. *Quantum computing enhanced computational catalysis*. 2020. arXiv: 2007.14460 [quant-ph]. URL: https://arxiv.org/abs/2007.14460.

[25] Colin Campbell et al. *Superstaq: Deep Optimization of Quantum Programs*. 2023. arXiv: 2309.05157 [quant-ph].

[26] Earl T. Campbell and Dan E. Browne. *On the Structure of Protocols for Magic State Distillation*. 2009. arXiv: 0908.0838 [quant-ph].

[27] Earl T. Campbell, Barbara M. Terhal, and Christophe Vuillot. “Roads towards fault-tolerant universal quantum computation”. In: *Nature* 549.7671 (Sept. 2017), 172–179. ISSN: 1476-4687. DOI: 10.1038/nature23460. URL: http://dx.doi.org/10.1038/nature23460.

[28] Daan Camps et al. *Explicit Quantum Circuits for Block Encodings of Certain Sparse Matrices*. 2023. arXiv: 2203.10236 [quant-ph].

[29] Alba Cervera-Lierta. "Exact Ising model simulation on a quantum computer". In: *Quantum* 2 (2018), p. 114. DOI: 10.22331/q-2018-12-21-114. URL: https://doi.org/10.22331%2Fq-2018-12-21-114.

[30] Andrew M. Childs, Robin Kothari, and Rolando D. Somma. "Quantum Algorithm for Systems of Linear Equations with Exponentially Improved Dependence on Precision". In: *SIAM Journal on Computing* 46.6 (Jan. 2017), pp. 1920–1950. DOI: 10.1137/16m1087072. URL: https://doi.org/10.1137/16m1087072.

[31] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. "Circuit Transformations for Quantum Architectures". en. In: Schloss Dagstuhl - Leibniz-Zentrum fur Informatik, 2019. DOI: 10.4230/LIPICS.TQC.2019.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TQC.2019.3.

[32] Andrew M. Childs and Nathan Wiebe. "Hamiltonian simulation using linear combinations of unitary operations". In: *Quantum Information and Computation* 12.11&12 (2012), pp. 901–924. DOI: 10.26421/qic12.11-12-1.

[33] Control Data Corporation. *160-A Computer Programming Manual*. 1963.

[34] Alexander Cowtan et al. "On the Qubit Routing Problem". In: *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*. Ed. by Wim van Dam and Laura Mancinska. Vol. 135. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 5:1–5:32. ISBN: 978-3-95977-112-2. DOI: 10.4230/LIPIcs.TQC.2019.5. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10397.

[35] Steven A. Cuccaro et al. *A new quantum ripple-carry addition circuit*. 2004. arXiv: quant-ph/0410184 [quant-ph].

[36] Andrew S. Darmawan et al. "Practical Quantum Error Correction with the XZZX Code and Kerr-Cat Qubits". In: *PRX Quantum* 2.3 (Sept. 2021). ISSN: 2691-3399. DOI: 10.1103/prxquantum.2.030345. URL: http://dx.doi.org/10.1103/PRXQuantum.2.030345.

[37] Poulami Das, Swamit Tannu, and Moinuddin Qureshi. "JigSaw: Boosting Fidelity of NISQ Programs via Measurement Subsetting". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, 937–949. ISBN: 9781450385572. DOI: 10.1145/3466752.3480044. URL: https://doi.org/10.1145/3466752.3480044.

[38] Christopher M. Dawson and Michael A. Nielsen. *The Solovay-Kitaev algorithm*. 2005. eprint: arXiv:quant-ph/0505030. URL: https://www.arxiv.org/abs/quant-ph/0505030.

[39] Eric Dennis et al. "Topological quantum memory". In: *Journal of Mathematical Physics* 43.9 (Sept. 2002), 4452–4505. ISSN: 1089-7658. DOI: 10.1063/1.1499754. URL: http://dx.doi.org/10.1063/1.1499754.

[40] David Deutsch and Richard Jozsa. "Rapid Solution of Problems by Quantum Computation". en. In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 439.1907 (Dec. 1992), pp. 553–558. ISSN: 1364-5021, 1471-2946. DOI: 10.1098/rspa.1992.0167. (Visited on 10/12/2016).

[41] Dhruv Devulapalli et al. *Quantum Routing with Teleportation*. 2022. arXiv: 2204.04185 [quant-ph].

[42] Y. Ding et al. "SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 570–583. DOI: 10.1109/ISCA45697.2020.00054.

[43] Yongshan Ding et al. "Magic-State Functional Units: Mapping and Scheduling Multi-Level Distillation Circuits for Fault-Tolerant Quantum Architectures". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2018. DOI: 10.1109/micro.2018.00072. URL: http://dx.doi.org/10.1109/MICRO.2018.00072.

[44] David P DiVincenzo. "Two-bit gates are universal for quantum computation". In: *Physical Review A* 51.2 (1995), p. 1015.

[45] Steven T. Flammia and Yi-Kai Liu. "Direct Fidelity Estimation from Few Pauli Measurements". In: *Physical Review Letters* 106.230501 (2011).

[46] Simon Forest et al. "Exact synthesis of single-qubit unitaries over Clifford-cyclotomic gate sets". In: *Journal of Mathematical Physics* 56.8 (2015), p. 082201. DOI: 10.1063/1.4927100. URL: https://doi.org/10.1063/1.4927100.

[47] Austin G. Fowler and Simon J. Devitt. *A bridge to lower overhead quantum computation*. 2013. arXiv: 1209.0510 [quant-ph].

[48] Austin G. Fowler and Craig Gidney. *Low overhead quantum computation using lattice surgery*. 2019. arXiv: 1808.06709 [quant-ph].

[49] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Phys. Rev. A* 86 (3 2012), p. 032324. DOI: 10.1103/PhysRevA.86.032324.

[50] Craig Gidney. "Halving the cost of quantum addition". In: *Quantum* 2 (June 2018), p. 74. ISSN: 2521-327X. DOI: 10.22331/q-2018-06-18-74. URL: https://doi.org/10.22331/q-2018-06-18-74.

[51] Craig Gidney. *Inplace Access to the Surface Code Y Basis*. 2023. arXiv: 2302.07395 [quant-ph].

[52] Craig Gidney and Martin Ekerå. "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits". In: *Quantum* 5 (Apr. 2021), p. 433. ISSN: 2521-327X. DOI: 10.22331/q-2021-04-15-433. URL: https://doi.org/10.22331/q-2021-04-15-433.

[53] Craig Gidney and Martin Ekerå. *How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits*. 2019. arXiv: 1905.09749 [quant-ph]. URL: https://www.arxiv.org/abs/1905.09749.

[54] Craig Gidney and Austin Fowler. *A slightly smaller surface code S gate*. 2017. arXiv: 1708.00054 [quant-ph].

[55] Craig Gidney and Austin G. Fowler. "Efficient magic state factories with a catalyzed CCZ to 2T Transformation". In: *Quantum* 3 (Apr. 2019), p. 135. ISSN: 2521-327X. DOI: 10.22331/q-2019-04-30-135. URL: http://dx.doi.org/10.22331/q-2019-04-30-135.

[56] Craig Gidney and Austin G. Fowler. *Flexible layout of surface code computations using AutoCCZ states*. 2019. arXiv: 1905.08916 [quant-ph].

[57] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum Random Access Memory". In: *Phys. Rev. Lett.* 100 (16 2008), p. 160501. DOI: 10.1103/PhysRevLett.100.160501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.100.160501.

[58] Daniel Gottesman. "An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation". In: *quant-ph* (). URL: arXiv:0904.2557v1.

[59] Daniel Gottesman. "Stabilizer Codes and Quantum Error Correction". PhD thesis. Pasadena, California: California Institute of Technology, May 1997. URL: https://thesis.library.caltech.edu/2900/2/THESIS.pdf.

[60] Christopher Granade, Joshua Combes, and D G Cory. "Practical Bayesian tomography". In: *New Journal of Physics* 18.3 (2016), p. 033024. DOI: 10.1088/1367-2630/18/3/033024. URL: https://doi.org/10.1088/1367-2630/18/3/033024.

[61] Christopher Granade, Christopher Ferrie, and Steven T Flammia. "Practical adaptive quantum tomography". In: *New Journal of Physics* 19.11 (2017), p. 113017. DOI: 10.1088/1367-2630/aa8fe6. URL: https://doi.org/10.1088%2F1367-2630%2Faa8fe6.

[62] Christopher Granade et al. *QInfer: Library for Statistical Inference in Quantum Information*. Sept. 2016. DOI: 10.5281/zenodo.157007. URL: http://dx.doi.org/10.5281/zenodo.157007.

[63] Christopher E Granade et al. "Robust online Hamiltonian learning". In: *New Journal of Physics* 14.10 (2012), p. 103013. DOI: 10.1088/1367-2630/14/10/103013. URL: https://doi.org/10.1088%2F1367-2630%2F14%2F10%2F103013.

[64] Daniel Greenbaum. *Introduction to Quantum Gate Set Tomography*. 2015. DOI: 10.48550/ARXIV.1509.02921. URL: https://arxiv.org/abs/1509.02921.

[65] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *arXiv:quant-ph/9605043* (May 1996). arXiv: quant-ph/9605043. (Visited on 10/12/2016).

[66] Riddhi Swaroop Gupta et al. "Autonomous adaptive noise characterization in quantum computers". In: *npj Quantum Inf* 6 (2020). DOI: 10.1038/s41534-020-0286-0.

[67] R. Harper, Steven T Flammia, and J. J. Wallman. "Efficient Learning of Quantum Noise". In: *Nat. Phys* (2020).

[68] Robin Harper and Steven T. Flammia. "Fault-Tolerant Logical Gates in the IBM Quantum Experience". In: *Physical Review Letters* 122.8 (Feb. 2019), p. 080504. DOI: 10.1103/PhysRevLett.122.080504. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.080504.

[69] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum Algorithm for Linear Systems of Equations". In: *Phys. Rev. Lett.* 103 (15 2009), p. 150502. DOI: 10.1103/PhysRevLett.103.150502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502.

[70] Daniel Herr, Franco Nori, and Simon J Devitt. "Lattice surgery translation for quantum computation". In: *New Journal of Physics* 19.1 (Jan. 2017), p. 013034. ISSN: 1367-2630. DOI: 10.1088/1367-2630/aa5709. URL: http://dx.doi.org/10.1088/1367-2630/aa5709.

[71] Oscar Higgott and Craig Gidney. *Sparse Blossom: correcting a million errors per core second with minimum-weight matching*. 2023. arXiv: 2303.15933 [quant-ph].

[72] H.M.Wiseman et al. "Adaptive Measurements in the Optical Quantum Information Laboratory". In: *quant-ph* ().

[73] Adam Holmes et al. "Resource optimized quantum architectures for surface code implementations of magic-state distillation". In: *Microprocessors and Microsystems* 67 (June 2019), 56–70. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2019.02.007. URL: http://dx.doi.org/10.1016/j.micpro.2019.02.007.

[74] Dominic Horsman et al. "Surface code quantum computing by lattice surgery". In: *New Journal of Physics* 14.12 (2012), p. 123011. DOI: 10.1088/1367-2630/14/12/123011. URL: https://dx.doi.org/10.1088/1367-2630/14/12/123011.

[75] M Howard et al. "Quantum process tomography and Linblad estimation of a solid-state qubit". In: *New Journal of Physics* 8.3 (2006), pp. 33–33. DOI: 10.1088/1367-2630/8/3/033. URL: https://doi.org/10.1088%2F1367-2630%2F8%2F3%2F033.

[76] Mike West Jane Liu. "Combined Parameter and State Estimation in Simulation-Based Filtering". In: *Springer-Verlag, New York* (2001).

[77] Samuel Jaques and Arthur G. Rattew. "QRAM: A Survey and Critique". In: (2023). arXiv: 2305.10310 [quant-ph].

[78] N. Cody Jones et al. "Layered Architecture for Quantum Computing". In: *Phys. Rev. X* 2 (3 2012), p. 031007. DOI: 10.1103/PhysRevX.2.031007. URL: https://link.aps.org/doi/10.1103/PhysRevX.2.031007.

[79] Ivan Kassal et al. "Polynomial-Time Quantum Algorithm for the Simulation of Chemical Dynamics". en. In: *Proceedings of the National Academy of Sciences* 105.48 (Feb. 2008), pp. 18681–18686. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.0808245105. (Visited on 10/12/2016).

[80] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. "Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates". In: (2012). eprint: arXiv:1206.5236. URL: https://www.arxiv.org/abs/1206.5236.

[81] Sebastian Krinner et al. "Realizing repeated quantum error correction in a distance-three surface code". In: *Nature* 605.7911 (May 2022), 669–674. ISSN: 1476-4687. DOI: 10.1038/s41586-022-04566-8. URL: http://dx.doi.org/10.1038/s41586-022-04566-8.

[82] Antonio A. Lagana, M. A. Lohe, and Lorenz von Smekal. "Construction of a universal quantum computer". In: *Phys. Rev. A* 79 (5 2009), p. 052322. DOI: 10.1103/PhysRevA.79.052322. URL: https://link.aps.org/doi/10.1103/PhysRevA.79.052322.

[83] L Lao et al. "Mapping of lattice surgery-based quantum circuits on surface code architectures". In: *Quantum Science and Technology* 4.1 (2018), p. 015005. DOI: 10.1088/2058-9565/aadd1a. URL: https://dx.doi.org/10.1088/2058-9565/aadd1a.

[84] Jessica Lemieux et al. "Efficient Quantum Walk Circuits for Metropolis-Hastings Algorithm". In: *Quantum* 4 (June 2020), p. 287. ISSN: 2521-327X. DOI: 10.22331/q-2020-06-29-287. URL: https://doi.org/10.22331/q-2020-06-29-287.

[85] Andy C. Y. Li et al. *Benchmarking variational quantum eigensolvers for the square-octagon-lattice Kitaev model*. 2021. DOI: 10.48550/ARXIV.2108.13375. URL: https://arxiv.org/abs/2108.13375.

[86] Gushu Li, Yufei Ding, and Yuan Xie. "Tackling the qubit mapping problem for NISQ-era quantum devices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 1001–1014.

[87] Daniel Litinski. "A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery". In: *Quantum* 3 (Mar. 2019), p. 128. ISSN: 2521-327X. DOI: 10.22331/q-2019-03-05-128.

[88] Daniel Litinski. "Magic State Distillation: Not as Costly as You Think". In: *Quantum* 3 (Dec. 2019), p. 205. ISSN: 2521-327X. DOI: 10.22331/q-2019-12-02-205. URL: http://dx.doi.org/10.22331/q-2019-12-02-205.

[89] Daniel Litinski and Felix von Oppen. "Lattice Surgery with a Twist: Simplifying Clifford Gates of Surface Codes". In: *Quantum* 2 (May 2018), p. 62. ISSN: 2521-327X. DOI: 10.22331/q-2018-05-04-62. URL: https://doi.org/10.22331/q-2018-05-04-62.

[90] Seth Lloyd. "Universal Quantum Simulators". en. In: *Science* 273.5278 (Aug. 1996), pp. 1073–1078. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.273.5278.1073. (Visited on 10/12/2016).

[91] Easwar Magesan, J. M. Gambetta, and Joseph Emerson. "Scalable and Robust Randomized Benchmarking of Quantum Processes". In: *Phys. Rev. Lett.* 106 (18 2011), p. 180504. DOI: 10.1103/PhysRevLett.106.180504. URL: https://link.aps.org/doi/10.1103/PhysRevLett.106.180504.

[92] Easwar Magesan, Jay M. Gambetta, and Joseph Emerson. "Characterizing Quantum Gates via Randomized Benchmarking". In: *Physical Review A* 85.4 (Apr. 2012). ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.85.042311. arXiv: 1109.6887. (Visited on 10/28/2016).

[93] Mariantoni. "Implementing the Quantum von Neumann Architecture with Superconducting Circuits". In: *Science* 334.6052 (2011), pp. 61–65. ISSN: 0036-8075. DOI: 10.1126/science.1208517. eprint: https://science.sciencemag.org/content/334/6052/61.full.pdf. URL: https://science.sciencemag.org/content/334/6052/61.

[94] Olivia Di Matteo, Vlad Gheorghiu, and Michele Mosca. "Fault-Tolerant Resource Estimation of Quantum Random-Access Memories". In: *IEEE Transactions on Quantum Engineering* 1 (2020), 1–13. ISSN: 2689-1808. DOI: 10.1109/tqe.2020.2965803. URL: http://dx.doi.org/10.1109/TQE.2020.2965803.

[95] Seth T. Merkel et al. "Self-consistent quantum process tomography". In: *Phys. Rev. A* 87 (6 2013), p. 062119. DOI: 10.1103/PhysRevA.87.062119. URL: https://link.aps.org/doi/10.1103/PhysRevA.87.062119.

[96] Pierre Del Moral, Aranaud Doucet, and Ajay Jasra. *Sequential Monte Carlo for Bayesian Computation*. Oxford University Press, 2006.

[97] Jonathan E. Moussa. "Transversal Clifford gates on folded surface codes". In: *Phys. Rev. A* 94 (4 2016), p. 042316. DOI: 10.1103/PhysRevA.94.042316. URL: https://link.aps.org/doi/10.1103/PhysRevA.94.042316.

[98] Prakash Murali et al. "Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, 1001–1016. ISBN: 9781450371025. DOI: 10.1145/3373376.3378477. URL: https://doi.org/10.1145/3373376.3378477.

[99] Paul D. Nation et al. "Scalable Mitigation of Measurement Errors on Quantum Computers". In: *PRX Quantum* 2 (4 2021), p. 040326. DOI: 10.1103/PRXQuantum.2.040326. URL: https://link.aps.org/doi/10.1103/PRXQuantum.2.040326.

[100] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. DOI: 10.1017/CBO9780511976667.

[101] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2009. DOI: 10.1017/cbo9780511976667.

[102] Jeremy L O'Brien et al. "Quantum process tomography of a controlled-NOT gate". In: *Physical review letters* 93.8 (2004), p. 080502.

[103] Dahl O.J, Dijkstra E.W., and Hoare C.A.R. *Structured Programming*. Academic Press, 1972.

[104] Joe O'Gorman and Earl T. Campbell. "Quantum computation with realistic magic-state factories". In: *Physical Review A* 95.3 (Mar. 2017). ISSN: 2469-9934. DOI: 10.1103/physreva.95.032338. URL: http://dx.doi.org/10.1103/PhysRevA.95.032338.

[105] Gurleen Padda et al. *Improving Qubit Routing by Using Entanglement Mediated Remote Gates*. 2023. arXiv: 2309.13141 [quant-ph].

[106] Adam Paetznick and Krysta M. Svore. "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries". In: (2013). eprint: arXiv:1311.1074. URL: https://www.arxiv.org/abs/1311.1074.

[107] Alexandru Paler and Robert Basmadjian. *Clifford Gate Optimisation and T Gate Scheduling: Using Queueing Models for Topological Assemblies*. June 2019. arXiv: 1906.06400 [quant-ph].

[108] Alexandru Paler and Robert Basmadjian. *Clifford Gate Optimisation and T Gate Scheduling: Using Queueing Models for Topological Assemblies*. 2019. arXiv: 1906.06400 [quant-ph].

[109] Alexandru Paler and Austin G. Fowler. *OpenSurgery for Topological Assemblies*. 2020. arXiv: 1906.07994 [quant-ph].

[110] Alexandru Paler, Oumarou Oumarou, and Robert Basmadjian. "Parallelizing the queries in a bucket-brigade quantum random access memory". In: *Physical Review A* 102.3 (Sept. 2020). ISSN: 2469-9934. DOI: 10.1103/physreva.102.032608. URL: http://dx.doi.org/10.1103/PhysRevA.102.032608.

[111] T. Patel and D. Tiwari. "VERITAS: Accurately Estimating the Correct Output on Noisy Intermediate-Scale Quantum Computers". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–16. DOI: 10.1109/SC41405.2020.00019.

[112] Fernando Pérez and Brian E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. (Visited on 10/28/2016).

[113] J. M. Pino et al. "Demonstration of the trapped-ion quantum CCD computer architecture". In: *Nature* 592.7853 (2021), pp. 209–213. DOI: 10.1038/s41586-021-03318-4. URL: https://doi.org/10.1038%2Fs41586-021-03318-4.

[114] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331/q-2018-08-06-79.

[115] Timothy Proctor et al. "Measuring the capabilities of quantum computers". In: *Nature Physics* 18.1 (2021), pp. 75–79. DOI: 10.1038/s41567-021-01409-7. URL: https://doi.org/10.1038%2Fs41567-021-01409-7.

[116] Neil J. Ross and Peter Selinger. *Optimal ancilla-free Clifford+T approximation of z-rotations*. 2016. arXiv: 1403.2975 [quant-ph].

[117] Yuval R. Sanders et al. "Black-Box Quantum State Preparation without Arithmetic". In: *Phys. Rev. Lett.* 122 (2 2019), p. 020502. DOI: 10.1103/PhysRevLett.122.020502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.020502.

[118] Yuval R. Sanders et al. "Compilation of Fault-Tolerant Quantum Heuristics for Combinatorial Optimization". In: *PRX Quantum* 1 (2 2020), p. 020312. DOI: 10.1103/PRXQuantum.1.020312. URL: https://link.aps.org/doi/10.1103/PRXQuantum.1.020312.

[119] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *arXiv:quant-ph/9508027* (Aug. 1995). arXiv: quant-ph/9508027. (Visited on 10/12/2016).

[120] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Review* 41.2 (Jan. 1999), pp. 303–332. DOI: 10.1137/s0036144598347011. URL: https://doi.org/10.1137/s0036144598347011.

[121] P.W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. SFCS-94. IEEE Comput. Soc. Press. DOI: 10.1109/sfcs.1994.365700.

[122] Robert S. Smith, Michael J. Curtis, and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. 2016. DOI: 10.48550/ARXIV.1608.03355. URL: https://arxiv.org/abs/1608.03355.

[123] Swamit Tannu and Moinuddin Qureshi. "Mitigating Measurement Errors in Quantum Computers by Exploiting State-Dependent Bias". In: *MICRO* (2019).

[124] Himanshu Thapliyal et al. *Quantum Circuit Designs of Integer Division Optimizing T-count and T-depth*. 2018. arXiv: 1809.09732 [quant-ph].

[125] W.G. Unruh. "Maintaining Coherence in Quantum Computers". In: 51 (1995).

[126] Joshua Viszlai et al. *An Architecture for Improved Surface Code Connectivity in Neutral Atoms*. 2023. arXiv: 2309.13507 [quant-ph].

[127] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37. (Visited on 10/28/2016).

[128] George Watkins et al. *A High Performance Compiler for Very Large Scale Surface Code Computations*. 2023. arXiv: 2302.02459 [quant-ph].

[129] Christopher J. Wood, Jacob D. Biamonte, and David G. Cory. "Tensor networks and graphical calculus for open quantum systems". In: (2011). DOI: 10.48550/ARXIV.1111.6950. URL: https://arxiv.org/abs/1111.6950.

[130] W.K. Wooters and W.H. Zurek. "A single quantum cannot be cloned". In: *Nature* 299 (1982), pp. 802–803.

[131] W. K. Wootters and W. H. Zurek. "A single quantum cannot be cloned". In: *Nature* 299.5886 (1982), pp. 802–803. DOI: 10.1038/299802a0.

[132] K. Wright et al. “Benchmarking an 11-qubit quantum computer”. In: *Nature Communications* 10.1 (2019). DOI: 10.1038/s41467-019-13534-2. URL: https://doi.org/10.1038%2Fs41467-019-13534-2.

[133] Yanqing Wu et al. “Quantum Behavior of Graphene Transistors near the Scaling Limit”. In: *Nano Letters* 12.3 (Mar. 2012), pp. 1417–1423. ISSN: 1530-6984. DOI: 10.1021/nl204088b. (Visited on 10/12/2016).

[134] Charles Yuan and Michael Carbin. “Tower: data structures in Quantum superposition”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (Oct. 2022), 259–288. ISSN: 2475-1421. DOI: 10.1145/3563297. URL: http://dx.doi.org/10.1145/3563297.

[135] Charles Yuan, Agnes Villanyi, and Michael Carbin. *Quantum Control Machine: The Limits of Control Flow in Quantum Programming*. 2023. arXiv: 2304.15000 [cs.PL].

[136] Pei Yuan and Shengyu Zhang. *Optimal (controlled) quantum state preparation and improved unitary synthesis by quantum circuits with any number of ancillary qubits*. 2023. arXiv: 2202.11302 [quant-ph].