SABRE^{FT+}: Circuit Synthesis for Fault-Tolerant Quantum Computing

Journal:	*ACM Transactions on Quantum Computing*
Manuscript ID	TQC-2023-0005
Manuscript Type:	Research Paper
Date Submitted by the Author:	16-Jan-2023
Complete List of Authors:	Hwang, Yongsoo; Electronics and Telecommunications Research Institute,
Computing Classification Systems:	fault-tolerant quantum computing, quantum circuit
Specialty/Area of Expertise:	fault-tolerant quantum computing

SABRE^{FT+}: Circuit Synthesis for Fault-Tolerant Quantum Computing

Yongsoo Hwang

Electronics and Telecommunications Research Institute, Daejeon, 34129, Republic of Korea

E-mail: yhwang@etri.re.kr

Abstract. To realize universal fault-tolerant quantum computing, there is a need for circuit libraries of universal fault-tolerant quantum protocols that must be executable in terms of locality constraint that a quantum chip has and retain both the fault tolerance and the logic sequence of the protocol at the same time. Unlike topological codes that are fundamentally defined based on local operations, for concatenated codes, we must find such circuits separately. To date, a systematic method for such a task has never been reported, except for manually obtained circuits for small-sized codes. In this work, for the first time, a quantum circuit synthesis algorithm for universal fault-tolerant quantum computing based on concatenated codes has been proposed. We enumerate the requirements and present our approaches and how to implement them with the existing quantum circuit mapping heuristic algorithm. As examples, we show how to synthesize the circuit set of universal fault-tolerant quantum computing based on $[[7, 1, 3]]$ Steane code and the circuit for the syndrome measurement of $[[23, 1, 7]]$ Golay code.

1. Introduction

The application of quantum coding theory and accuracy threshold theorem has enabled reliable preservation and manipulation of quantum states over long periods of time [1, 2, 3, 4, 5, 6, 7, 8]. Thus, the obstacle posed by quantum noise in the implementation of quantum computing has become a nonissue. Nevertheless, realizing error-corrected fault-tolerant quantum computing is still very challenging. It can be achieved through complicated procedures with a large number of high-fidelity quantum and classical resources.

To implement fault-tolerant quantum computing, the most essential ingredient is a set of working quantum circuits of theoretically verified protocols [9]. However, a certain protocol having fault tolerance does not guarantee that it can be run in a fault-tolerant manner in actual practice. This is because it is made without considering the constraints the practical quantum device has. Therefore, we need circuits that work

correctly on practical quantum hardware with geometric locality constraints, retaining the fault tolerance and logic flow of the protocol.

Topological codes represented by surface code [10, 11, 12] and color code [13, 14] are fundamentally defined based on the local quantum operations. Therefore, if the qubit lattice used for the theoretical study of the codes is provided, the protocol itself can then be a working quantum circuit. For example, on Google's quantum processor, Sycamore [15, 16, 17], which has a 2-dimensional square lattice, the surface code can be naturally mapped.

On the contrary, the concatenated codes [7, 18, 19, 20, 21, 22, 23] that are rooted in classical linear codes have been developed without considering such limitation in general. That is, the related protocols are defined based on 2-qubit gates acting on arbitrary qubit pairs, and it is not possible to execute some of them directly on the practical quantum hardware where the spatial locality governs. Therefore, we must separately find the working quantum circuits of the protocols.

Refs. [24, 25, 26] claimed that there exist quantum circuits of fault-tolerant quantum protocols acting in the local setting. However, to the best of the author's knowledge, systematic methods for generating quantum circuits having fault tolerance have never been proposed, except certain results obtained manually for a few well-known small-sized codes that have been reported later [27, 28, 29]. These results may be optimal for the designated setting; however, for other circumstances or quantum codes, in particular, in case of a larger code distance, finding *efficient* fault-tolerant quantum circuits composed of local operations may be non-trivial using their methodology. For this reason, we need to develop a structured method of finding quantum circuits for local fault-tolerant quantum computing.

Quantum circuit synthesis (or *circuit mapping*) is a method that systematically translates a quantum algorithm developed under the assumption of an ideal device into a quantum circuit that works on a real quantum computing hardware with physical and logical constraints. In particular, to run 2-qubit `CNOT` gates for arbitrary qubit pairs under the geometric locality limitation, it introduces a sequence of quantum state moves (via `SWAP` gates). To date, various quantum circuit synthesis algorithms have been proposed to generate compact quantum circuits of ordinary quantum computational algorithms [30, 31, 32]. Most such algorithms create an optimal working quantum circuit that includes the least possible number of `SWAP` gates based on qubit connectivity; however, certain advanced methods [33, 34, 35, 36, 37] also consider the physical performance of a quantum device or the classical control architecture to obtain higher quantum computing performance.

In this work, we use the circuit synthesis methodology to obtain quantum circuits for fault-tolerant quantum computing. We first discuss the case when the conventional quantum circuit synthesis algorithm is applied for the fault-tolerant quantum protocols. Evidently, an optimized quantum circuit in terms of the number of `SWAP` gates is obtained. However, it is likely that some of the `SWAP` gates act on the pair of data qubits. In that case, the fault tolerance of the protocol can not be retained because

the circuit contains the routes for error propagation through data qubits. Besides this, to implement the circuits for fault-tolerant quantum computing correctly, there exist additional considerations not covered in ordinary quantum circuit synthesis. Therefore, existing circuit synthesis algorithms cannot be directly used for the fault-tolerant protocols. In this regard, we need a quantum circuit synthesis algorithm that specializes in fault-tolerant quantum computing.

We thus propose a circuit synthesis algorithm with the following features. First, it finds *fault-tolerant qubit movement paths* that limitedly include `SWAP` gates acting on pairs of data qubits. The circuit then does not allow the error propagation over the data qubits beyond the error-correction capacity. Second, it performs *circuit partitioning* that implements the circuit of a fault-tolerant protocol as a set of several sub-circuits, with each working in a fault-tolerant manner. The classical control processing included in the protocol then can be executed between sub-circuits. Third, it generates a *self-contained* quantum circuit that includes the operations that make all the data qubits move back to their initial positions. The circuit then can be utilized as a library circuit that works correctly in any situation. Last, by applying the above-mentioned functions to the *fault-tolerant quantum protocols categorized hierarchically*, we obtain a full set of quantum circuits for fault-tolerant quantum computing. Readers can access the open-source code of the proposed algorithm online [38].

The remainder of this paper is organized as follows: It first briefly reviews SABRE [32], which lays the groundwork on which the proposed method is based. We then discuss the four requirements of the circuit synthesis for universal fault-tolerant quantum computing, propose our ideas for them, and describe how to implement the ideas with SABRE. As examples, we show the full snapshots of the syndrome measurement circuit of $[[7, 1, 3]]$ Steane code in sequence. We then move into the full set of universal fault-tolerant quantum circuits based on Steane code obtained by applying the proposed algorithm. As the last example, we show how to synthesize the circuit for the syndrome measurement of $[[23, 1, 7]]$ Golay code.

2. Preliminaries

2.1. Fault Tolerance

An $[[n, k, d]]$ quantum error-correcting code encodes a k-qubit logical information into n physical qubits and protects the encoded quantum data from at most $\lfloor (d-1)/2 \rfloor$-qubit quantum errors, where d is the code distance. If the number of quantum errors is no more than $\lfloor (d-1)/2 \rfloor$, it is possible to detect and correct the errors. Theoretically, n *data* qubits and $n-k$ *syndrome* qubits are required to form an error-corrected logical qubit. The number of quantum errors mentioned above as the upper bound of the capacity of error correction refers to errors that occurred on the data qubits.

To conduct fault-tolerant quantum computing based on the encoded logical qubits, we need the error-corrected logical equivalents of the physical quantum gates acting on

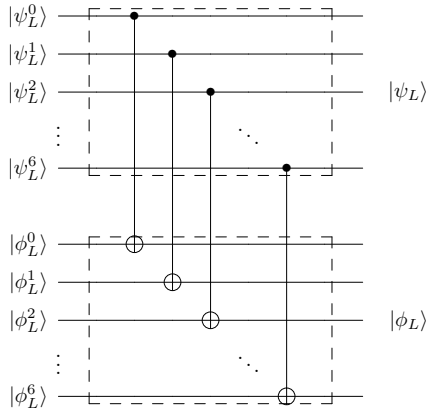


Figure 1. Transversal gate implementation of a logical CNOT gate

physical qubits. A quantum computing protocol acting on an encoded logical qubit is called *fault tolerant* if the probability that a logical qubit experiences a logical error while the protocol is running is lower than the physical error rate that the physical qubits undergo. If the number of quantum errors during the protocol exceeds $\lfloor (d-1)/2 \rfloor$, the encoded logical qubit gets damaged and cannot be recovered via quantum error correction. Therefore, to ensure fault tolerance, there should be no chance for an arbitrary single qubit quantum error that occurs while running the protocol to propagate to more than $\lfloor (d-1)/2 \rfloor$.

A generic fault-tolerant quantum computing protocol is designed to mostly block (noisy) multi-qubit gates between the data qubits to stop the spreading of errors over the data qubits. For example, the *transversal* implementation of a logical quantum gate working in the qubit-wise manner between logical qubits is a preferred method to achieve fault tolerance at a low cost (see Figure 1). Since there are no inter-qubit interactions within a logical qubit code block, it is unlikely that the quantum errors spread over the bound of quantum error correction.

2.2. SABRE

Here, we review the heuristic quantum circuit mapping algorithm SABRE [32]. The algorithm generates an *optimal* quantum circuit by iterating the procedure composed of picking a qubit initial mapping at random and finding the gate sequence based on the mapping as many times as possible. The standard of optimality can be flexibly adopted - the number of quantum gates, the circuit depth, or anything.

The main procedure of SABRE, which decides the gate sequence based on the initial qubit mapping, is composed of iterating graph traversals three times in alternating directions (forward, backward, and forward again). The graph is a directed acyclic graph (DAG) constructed from an input quantum algorithm. A node of DAG includes instruction of a quantum operation and qubits where the gate acts or a certain statement. Figure 2 shows the running flow of SABRE.

Let us denote FL (Front Layer) as the set of root nodes of DAG. The graph traversal

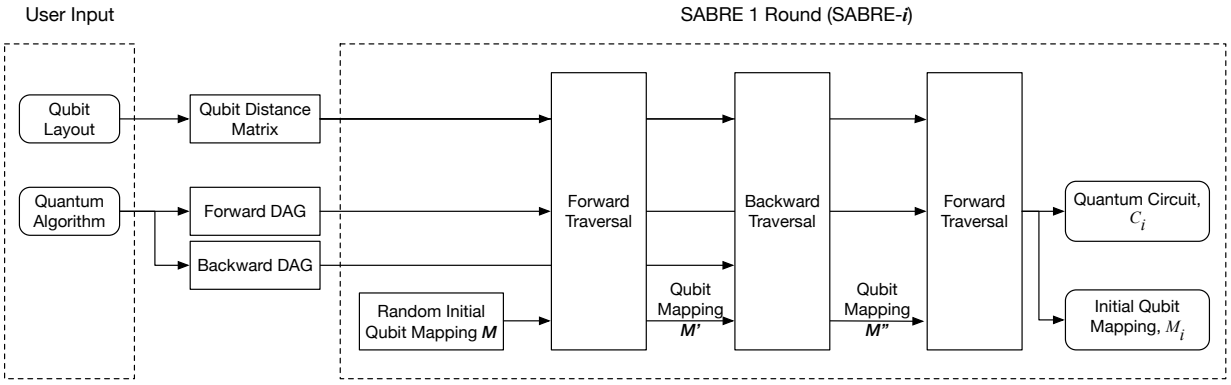


Figure 2. Architecture of SABRE [32]. It is composed of iterating three graph traversals in alternating directions. The user inputs, *Qubit Layout* and *Quantum Algorithm*, are common to all the iterations. Usually, *Qubit Layout* is provided as a physical qubit coupling graph of a quantum chip and Quantum Algorithm is provided in the QASM format [39]. Provided Qubit Layout and Quantum Algorithm, the procedure to make *Qubit Distance Matrix* and *Directed Acyclic Graph* are well described in the literature. The algorithm is composed of multiple iterations of a sequence of graph traversals. The graph traversal is briefly described in Procedure 1. Each iteration begins by picking an initial qubit association at random. After conducting the i-th iteration of SABRE, we will have a quantum circuit $\mathcal{C}_i$ and the associated initial qubit mapping table M_i. The optimal quantum circuit and its associated initial qubit mapping will be obtained by comparing all the results.

is then composed of the following two steps:

(i) Find any executable gates in FL.

(ii) Treat all the executable gates (export them to the circuit and update FL) or find an optimal `SWAP` gate if an executable gate does not exist.

Note that a quantum gate that can run on the given qubit layout under the locality constraint is called *executable*.

Suppose that a node $node_a$ belongs to FL and it is executable. In that case, it is removed from FL and DAG simultaneously and is appended to the quantum circuit that we are generating now. Let us say that $node_b$ is a succeeding node of $node_a$. Conversely, it may then have several preceding nodes including $node_a$ (see Figure 9). Note that the relation between nodes indicates the priority of instructions in the logic sequence of a quantum algorithm.

The node $node_b$ does not belong to FL yet, that is, it is in *non-FL* (the complement set of FL in DAG). If all the preceding nodes of $node_b$ are exported to the circuit and therefore none of them remain in FL, it becomes possible to pull $node_b$ from *non-FL* to FL. Therefore, the algorithm will investigate the instruction of $node_b$ whether it is executable or not, from the following turn. On the contrary, if certain preceding nodes still stay in FL, $node_b$ cannot be moved to FL.

Suppose that none of FL are executable. The algorithm then collects several `SWAP` candidate gates and selects an optimal one by evaluating the costs of all the candidates.

Procedure 1 Graph Traversal of SABRE [32]

Input: Front Layer FL, Qubit Mapping π, Distance Matrix D, DAG, Qubit Coupling Graph G

Output: Gate Scheduling, Final Qubit Mapping π_f

```
while FL ≠ ∅ do
  list_executable_gates = find_executable_gates(FL)
  if list_executable_gates ≠ ∅ then
    Treat list_executable_gates and Update FL
  else
    SWAP_candidate_list = obtain_SWAP(FL, G)
    Evaluate SWAP_candidate_list and Select an optimal one, swap_optimal
    π = π.update(swap_optimal)
  end if
end while
```

By moving qubits via the selected `SWAP` gate, certain gates in FL may become executable, or close to executable, at least. The qubit mapping table should be updated by the `SWAP` gate. In the literature, two kinds of cost functions are defined, but their core is the same as they evaluate the total qubit movement distance required to make all the nodes in FL executable. If FL becomes empty, the graph traversal terminates. Procedure 1 shows the graph traversal briefly.

After each graph traversal, the qubit mapping table that has been updated throughout the graph traversal is provided to the following graph traversal (see Figure 2). Therefore, the initial qubit mapping and the resulting quantum circuit of the last graph traversal become the qubit mapping and the quantum circuit of a current SABRE round. The performance of a quantum circuit obtained in each round is affected by the initial qubit mapping chosen randomly. Therefore, the more the number of iterations, the better is performance of a quantum circuit.

3. Requirements for Circuits of FT Protocols in Fault-Tolerant Quantum Computing

This section enumerates the four characteristics that should be possessed by the quantum circuits of the fault-tolerant quantum protocols utilized in fault-tolerant quantum computing should have. We believe that these to be the minimum, but sufficient, conditions for the circuits. Note that this work assumes that there exist fault-tolerant quantum protocols that are theoretically verified.

(i) ***Fault Tolerance***: The most important characteristic is that the circuit must retain the fault tolerance that the protocol possesses. For the fault tolerance, the number of quantum errors that occurred on the data qubits of a logical qubit should be lower than the upper bound of error correction, $\lfloor (d-1)/2 \rfloor$. For that, the error

propagation through the data qubits should be blocked, or rarely allowed, by not (or hardly) including the multi-qubit operations between the data qubits.

(ii) ***Classical Control Processing***: A generic fault-tolerant quantum protocol contains not only quantum gates but also classical control processing (see Figure 3). To implement the logic of the protocol exactly, the circuit should be logically and physically partitioned based on the classical control operations. In doing so, the classical operation can be carried out after a quantum circuit, and its results can be fed back to the following quantum part.

(iii) ***Library***: The circuit of a fault-tolerant protocol should work as a library. Particularly, we hope that an additional operation depending on the context other than (the circuits of) logical gate must not be necessary. Here, by "extra operation," we mean to refer to, for example, an operation for refreshing a logical qubit depending on the previous logical gate operation. The circuit by itself must include the function instead.

(iv) ***Universal Computing***: For universal fault-tolerant quantum computing, we need a set of quantum circuits of *universal logical gates*, *syndrome measurement*, and *logical ancilla preparations*. All the protocols should act on a logical qubit with an identical qubit arrangement. For that, the circuit synthesis of each protocol must share the positions of data qubits in a logical qubit. Besides, for logical 2-qubit gates including the `T` gate, we must prepare the circuits for all cases of relative arrangements of logical qubits.

4. Circuit Synthesis for Fault-Tolerant Quantum Computing

Here, we describe how to satisfy and implement the requirements of the quantum circuits of fault-tolerant quantum computing presented in the previous section.

4.1. Fault-Tolerant Qubit Move

We earlier mentioned that to retain the fault tolerance the number of the multi-qubit gates between the data qubits should be under control. In this section, we show how to realize that under the locality constraint.

See Figure 3. Suppose that the qubits $|\psi_L^0\rangle$ and $|syndrome^0\rangle$ are located in the nearest neighbor. Then, the `CNOT` gate between both is executable. On the contrary, if the qubits are away, the gate cannot be directly applied to them due to the locality constraint. To resolve the restriction, we should move qubits to place them in the neighborhood first.

To move the qubits, we apply a sequence of (noisy) `SWAP` gates in general. The problem here is that it is likely that such a sequence includes a `SWAP` gate between data qubits. If a quantum error occurs while running a noisy `SWAP` gate acting on the pair of data qubits, both the qubits get corrupted as

$$\tilde{U_{\texttt{SWAP}}}|\psi\rangle|\phi\rangle = (I + \epsilon_2) \cdot U_{\texttt{SWAP}}|\psi\rangle|\phi\rangle$$

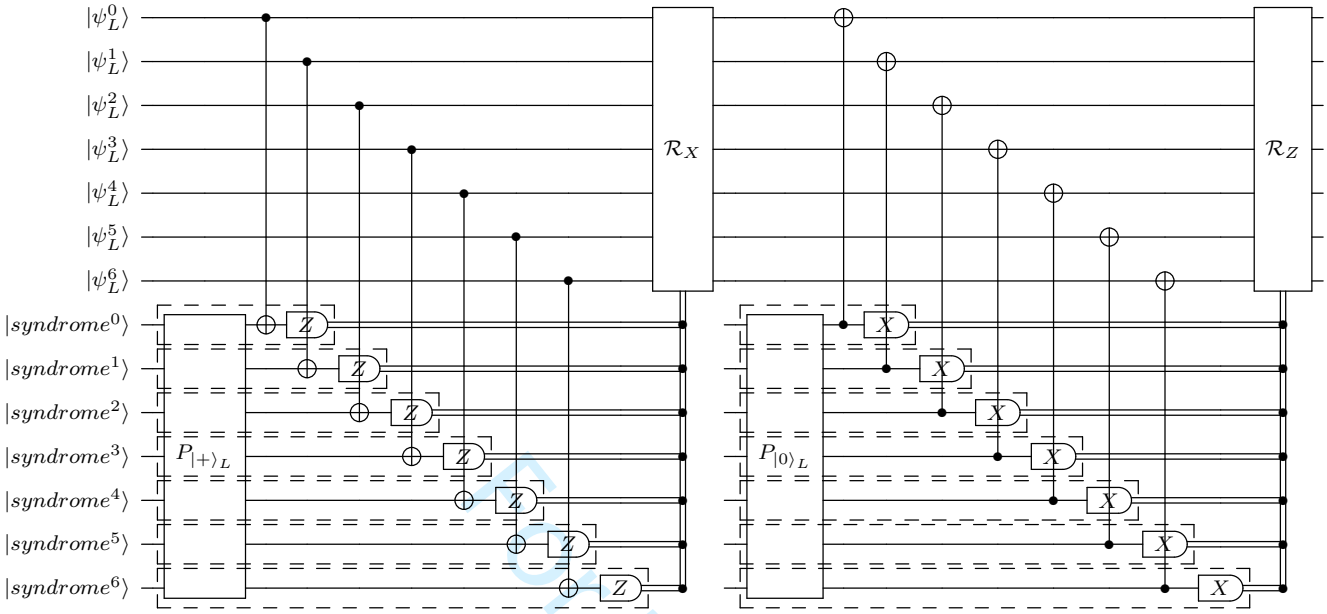


Figure 3. Protocol for the syndrome measurement based on Steane QEC method [40]. The dotted boxes for the ancilla qubit $|anc^i\rangle$ refer to their activation periods. The outsides of the boxes denote the inactivation periods. In the forward-direction graph traversal, the usage status of each qubit is activated by preparation and inactivated by measurement. Alternatively, in the backward direction, the measurement (preparation) operation activates (inactivates) the status of each qubit. In the ideal situation, all the ancilla qubits have the same activation period; however, in practice, due to the locality, certain ancilla qubits have a longer activation period than others. Please note that the data qubits $|\psi_L^i\rangle$ are all activated.

$$= |\tilde{\phi}\rangle|\tilde{\psi}\rangle,$$

where ϵ_2 is an arbitrary 2-qubit error and $U_{\texttt{SWAP}}$ is a noiseless `SWAP` gate. Therefore, in the circuit synthesis for a fault-tolerant quantum protocol, we must avoid or limit the introduction of `SWAP` gates between data qubits.

We now discuss the process of choosing `SWAP` gates that do not spread quantum errors over data qubits. Before going further, we first need to see the type of qubits used in a fault-tolerant quantum protocol.

All qubits, regardless of error-corrected logical ones or physical ones constituting a logical qubit, have their lifetime in fault-tolerant quantum computing. A logical qubit has its lifetime determined by a quantum algorithm. However, in the case of physical qubits composing a logical qubit, the lifespan is determined by its role in the fault-tolerant quantum protocol. For example, the data qubits that hold quantum data stay alive for the same duration as a logical qubit. On the contrary, the lifetime of ancilla qubits is bounded by the protocol, in particular the duration of the task where they participate. During the rest process in the protocol, the quantum state that they hold is not critical to quantum computing. Therefore, the qubits, for example, which are originally designed for measuring the error syndrome, can be utilized for other roles such as a communication channel.

To see the lifetime of qubits, we trace their usage status within a fault-tolerant quantum protocol. If a qubit is prepared as a certain quantum state $|0\rangle$ (or $|+\rangle$), its status becomes *activated*. Alternatively, it becomes *inactivated* if the qubit is measured. Therefore, while running a fault-tolerant protocol, the status of a qubit is changed alternatively. In the activated state, a qubit retains a piece of significant information for the protocol; however, when inactivated, the quantum state that the qubit stores is not of concern. For example, please see the dotted boxes in Figure 3.

In light of the usage status, we classify all the component qubits of a logical qubit into two categories: *data-type* and *non-data-type* qubits. A *data-type* (*non-data-type*) qubit is one that is in the activated (inactivated) state. As mentioned above, a data-type qubit carries important quantum information for computing or error correction and therefore we must protect it from noise. Therefore, we must block or limit a `SWAP` gate between the data-type qubits in the circuit synthesis.

So far, we have mentioned that a `SWAP` gate between the data-type qubits should not be allowed or limitedly allowed. This depends on the performance of a quantum error-correcting code. For example, a quantum code of $d = 7$ can correct arbitrary 3 qubit errors, and therefore it is possible to include a one-time `SWAP` gate between data-type qubits. Even though at most a 2-qubit error happens via the interaction, a logical qubit is not corrupted. To conclude, for an $[[n, k, d]]$ quantum code, we allow the introduction of `SWAP` gates between data-type qubits in the resulting circuit upto $\lfloor (d-1)/4 \rfloor$ times.

We now discuss how to implement the above-mentioned method using SABRE. Recall that SABRE is composed of three graph traversals in alternating directions. Also, recall that the usage status of a qubit becomes activated (inactivated) if it is prepared (measured). Such a rule is established based on the normal running flow of a quantum circuit, that is, the forward-direction traversal of *DAG*. Therefore, in the backward traversal, the status change should be caused by the opposite operations instead.

The original SABRE collects all the possible `SWAP` gates that act on the qubits of the nodes in *FL* as `SWAP` candidates and then picks an optimal one. However, the proposed algorithm must selectively collect the `SWAP` candidates to limit the `SWAP` gates on the data-type qubits. For that, when collecting the `SWAP` candidates, we examine the type of qubit pair based on the usage status. There are three cases: 1) both are the non-data-type qubits, 2) one data-type qubit and one non-data-type qubit, and 3) both are the data-type qubits. Evidently, the first and second cases can be collected as the `SWAP` candidates. However, for the third case, we need to do as described in what follows.

Suppose that for certain reasons, which will be discussed later, we must move the quantum state of a data-type qubit to another qubit. Figure 4 shows that $data[0]$ that needs to be moved to a physical qubit denoted as $Target$ is completely surrounded by data-type qubits. In that case, it cannot get to there without a noisy interaction (`SWAP` gate) with one data-type qubit. Figure 5 shows another situation where finding a fault-tolerant movement path may fall into an infinite loop.

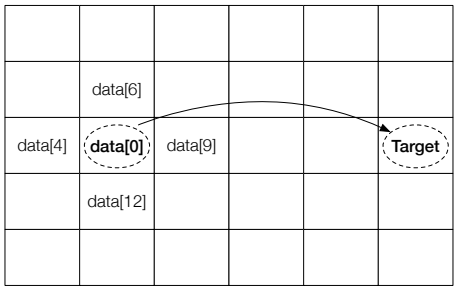


Figure 4. Since the data qubit $data[0]$ is surrounded by data qubits, it cannot be moved to another physical qubit in a fault-tolerant way.

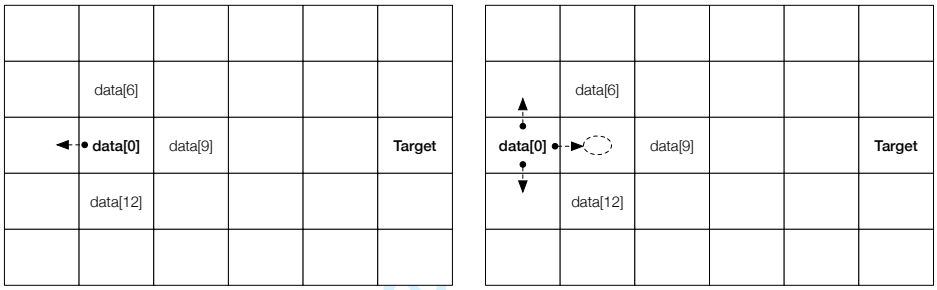


Figure 5. A circuit synthesis falls into an infinite loop when a data qubit surrounded by other data qubits is moved to another place. All the cells indicate physical qubits. (a) $data[0]$ has only one option for the next move, which is the *left* cell. (b) The cost function based on the distance between qubits selects the *right* cell for the next move.

To relieve such congestion, we first must move other data-type qubits around $data[0]$ to somewhere else. A route for the move ($data[0] \rightarrow Target$) can then be made. In this regard, we collect the `SWAP` gates acting on the qubits that are positioned next to the data-type qubit $data[0]$ as `SWAP` candidates according to the type of a qubit pair as well. Figure 6 shows the effect of this rule.

Previously, we mentioned that it is possible to include at most $\lfloor (d-1)/4 \rfloor$ `SWAP` gates between the data-type qubits in the circuit. Therefore, at the beginning of each graph traversal, we initialize a counter that counts the number of `SWAP` gates between the data-type qubits as zero. When collecting the `SWAP` candidate gates, we can add a `SWAP` gate acting on a pair of data-type qubits if the current counter value is less than the bound. Later, after the cost evaluation on the `SWAP` candidates, if a `SWAP` acting on both the data-type qubits is picked, we then increment the counter by 1.

The quantum circuit generated by the rule that we have mentioned so far includes very limited interactions between data-type qubits. Therefore, a quantum error that occurred during running the circuit does not propagate to data qubits going beyond the capacity of the quantum error correction. Then, we can say that the circuit works in a fault-tolerant manner.

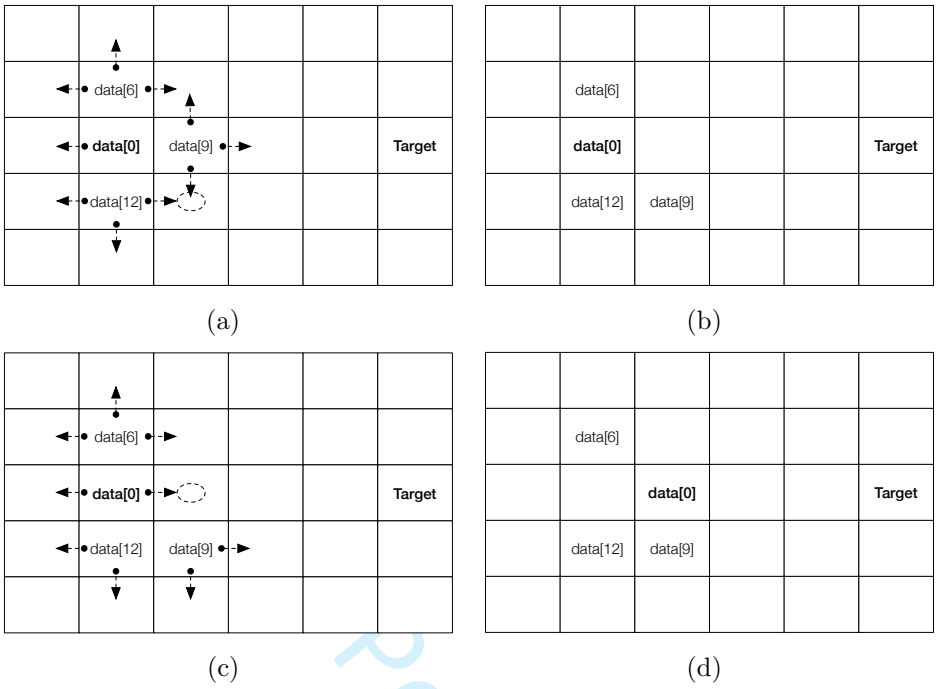


Figure 6. By first moving the data-type qubits located around the qubit that must move to the destination, the congestions shown in Figures 4 and 5 can be relieved and it becomes possible to find a fault-tolerant movement path. (a) Collect `SWAP` gates from both the data-type qubit and its neighbor data-type qubits. (b) Based on the cost evaluation, *data[9]* is moved to its neighbor cell of the dotted circle. (c) Collect `SWAP` gates from both the data-type qubit and its neighbor data-type qubits. (d) Based on the cost evaluation, *data[0]* moves toward the destination *Target* without interaction with any data-type qubits.

4.2. Circuit Partitioning

Owing to the nature of the circuit synthesis, all the quantum gates are placed as forward as possible to minimize the circuit depth. However, to make a working quantum circuit of a fault-tolerant protocol, we must take its logic into account first than just minimizing the circuit depth.

A generic fault-tolerant quantum protocol begins by preparing ancilla qubits as a known state and ends with the measurement of the ancilla qubits and the conditional processing based on the measurement outcomes. The classical control operation may be conducted once at the end or in the middle of the protocol, or several times in case. In the latter case, we must perform the classical operation just after some quantum operations and feed the result back to the following quantum operations. In that case, if the quantum circuit is properly partitioned based on classical control operation, it can be executed easily. Otherwise, it is very tricky to run the circuit correctly and efficiently. In what follows, we discuss how to make the quantum circuit of fault-tolerant quantum protocols partitioned without losing their logic.

We propose two approaches. First, we append a `Barrier` statement [39] to the protocol and treat it in the circuit synthesis. Second, we partition the protocol

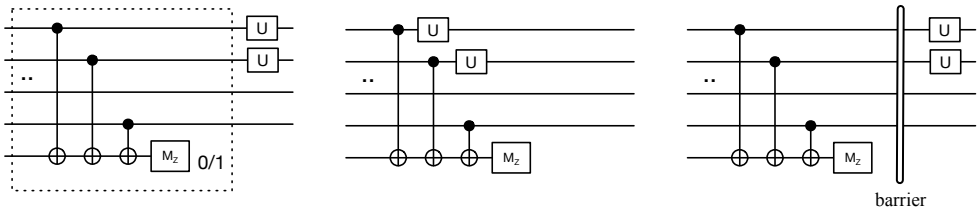


Figure 7. The effect of the circuit partitioning in the circuit synthesis for fault-tolerant quantum protocols. (a) Example algorithm that iterates the dotted box until the measurement comes out 1. (b) The circuit that is synthesized without circuit partitioning. (b) The partitioned circuit by `Barrier` statement.

beforehand, and perform the circuit synthesis on each sub-protocol individually but with sharing of data common to all the tasks. Since the second method is very trivial, we skip its detailed explanation here; however, it will be demonstrated as an example in Section 7.

Suppose that the fault-tolerant protocol described in the quantum assembly code (QASM) includes the `Barrier` statements at proper positions. Here, the proper position for the statement is - upon considering the structure of fault-tolerant quantum protocols - usually when just after all the ancilla qubits that were prepared within the protocol are measured. Please see Figures 3 and 16. Note that for the second method discussed above, we split the entire QASM up based on that position instead of inserting the statement.

We now describe how to deal with the `Barrier` statement in SABRE. Let us imagine the situation that the node with the statement, `Barrier` node, comes in FL after all its preceding nodes are treated and exported to a quantum circuit. When the `Barrier` node stays in FL, we do not append the succeeding nodes of the node that exported to a quantum circuit to FL but keep them in a temporary list (please see Procedure 3). If there is a `Barrier` node only in FL, it can be treated as follows:

(i) The node is removed from FL

(ii) All the nodes stored in the temporary list by the `Barrier` node are moved to FL (see Procedure 4)

By doing so, the circuit can be logically partitioned based on the `Barrier` statement. Please see Figure 7 for the effect of the circuit partitioning.

Before closing this section, let us recall that for finding a fault-tolerant qubit movement path we trace the qubit usage status throughout a protocol. In this regard, the time when the usage status of all the ancilla qubits used for the same task becomes *inactivated* is exactly the above-mentioned proper position for the `Barrier` statement. Therefore, even without introducing the statement, the circuit synthesis algorithm can partition the circuit based on the usage status tracking. However, the present work only considers the circuit partition via an explicit statement.

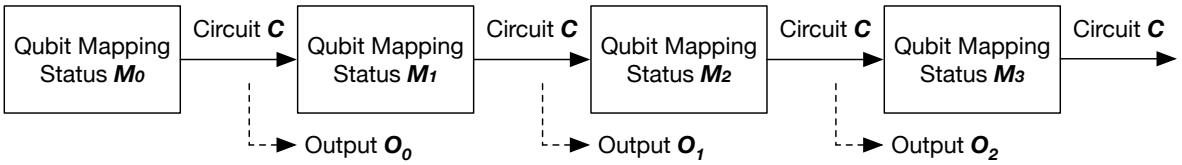


Figure 8. The repetitive execution of a quantum circuit including `SWAP` gates introduced by circuit mapping does not work as expected because of the qubit mapping problem, $(O_0 \neq O_1 \neq O_2 = \cdots)$.

4.3. Self-Contained Quantum Circuit

Suppose that you have a fault-tolerant quantum circuit and the initial qubit mapping for the syndrome measurement protocol. If a logical qubit is configured based on the initial qubit mapping table, it is possible to obtain the error syndrome by running the circuit. Otherwise, if the logical qubit is inconsistent with the mapping table, the circuit does not work as designed. Therefore, the quantum circuit should agree with the initial qubit mapping.

Fault-tolerant quantum computing performs error correction periodically. However, does the above fault-tolerant circuit then always work correctly? Unfortunately, it does not. The circuit will work correctly for the first time only. The reason for this is as follows. If a quantum circuit includes `SWAP` gates introduced by the circuit synthesis, by running the circuit, the qubit mapping will become different from the initial mapping. Therefore, when running the circuit on the qubit mapping again, it will generate outcomes different than expected because the initial qubit mapping for the circuit is inconsistent with the required one. Figure 8 shows such a situation. Please note that `SWAP` gates included in the protocol itself do not pose such a problem.

Therefore, to ensure that the syndrome measurement circuit always works correctly throughout the fault-tolerant quantum computing, we must move the component qubits of the logical qubit back to their initial positions after obtaining the syndrome. Therefore, we include a function in our algorithm that automatically moves the data qubits to the designated positions such as the initial positions or elsewhere, after the main body of a fault-tolerant protocol. Please note that since the lifetimes of the ancilla qubits are bounded by a protocol, their final positions do not affect an incoming logical operation on the logical qubit. In this work, we call the circuit that always works correctly regardless of the situation as a *self-contained* circuit.

To implement the move operation into SABRE, we automatically add the instruction "`Move` *data*[*i*] *destination*[*i*]" for all data qubits before generating *DAG*. The first argument *data*[*i*] is the name of a data qubit used in the protocol, and the second argument *destination*[*i*] is the destination of the move. Note that *destination*[*i*] can be a specific physical qubit index or a symbolic value according to the protocol. Even if it is provided as a symbolic value in the beginning, it is translated to a specific index as the circuit mapping proceeds (see Figure 10).

We now describe how to deal with the `Move` instruction in the proposed algorithm.

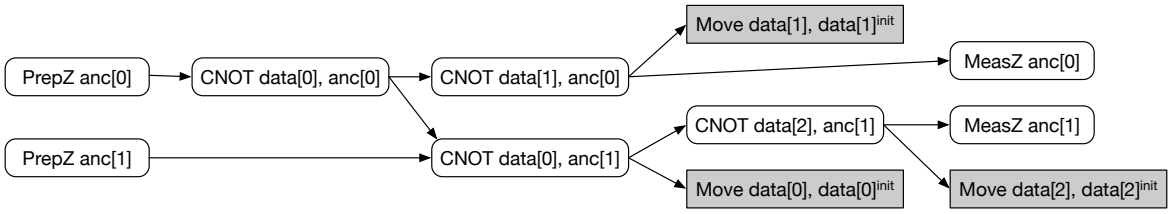


Figure 9. Directed acyclic graph of the 3-qubit code syndrome measurement circuit. The `Move` instructions for all the data qubits are included at the end.

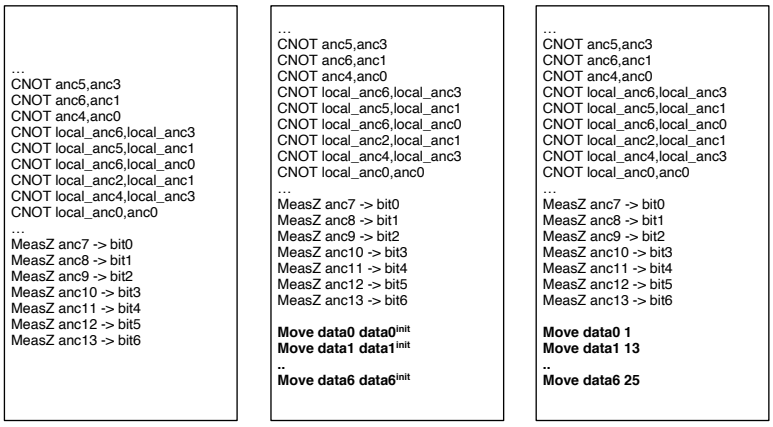
SABRE examines whether a quantum instruction in *FL* is executable or not based on qubit connectivity. In the case of a `CNOT` (or `SWAP`) gate, it determines as executable if a control qubit is positioned next to a target qubit. On the contrary, for the `Move` operation, we check whether the argument qubit is located at the designated position or not (see Procedure 3). If `Move` is decided as being executable, then the algorithm removes it from *FL* and does not anything. This is because for `Move` the most important point is not after the arrival but the middle process, a sequence of `SWAP` gates.

In this work, a bunch of the operations that moves all the data qubits to the designated position is called `MoveBack` because the target position is usually its initial location. The necessity of `MoveBack` depends on the fault-tolerant protocol. Here we describe the details of `MoveBack` for the syndrome measurement protocol, and other protocols will be discussed in the following section.

As mentioned above, only the data qubits should be moved to their initial positions. Please note that, as will be discussed in Section 4.4, we determine the logical qubit configuration from the initial qubit mapping for the synthesized circuit of the syndrome measurement. As when the circuit synthesis is started for the protocol, the specific position of a data qubit $data[i]$ in the qubit layout of a logical qubit is not determined yet, it is provided symbolically as $data[i]^{init}$ (see Figures 9 and 10).

In each SABRE iteration, an initial qubit mapping is picked at random (see Figure 2), and the initial position of $data[i]$ is then fixed at that time. Therefore, with the random initial qubit mapping, the instruction "`Move` $data[i]$ $data[i]^{init}$" is translated to the instruction including a specific qubit index; for example, "`Move` $data[i]$ 3." The circuit mapping algorithm then tries to move $data[i]$ from a current location to the physical qubit with the index 3. Figure 10 shows the conceptual transformation of the protocol description throughout the circuit synthesis.

Sometimes, it may happen that the qubits that already arrived at the designated positions - let us say the initial positions - may be shifted to other places in treating the move operation of other qubits. When such an event occurs, the synthesis algorithm inserts the `Move` about the qubit into *FL* forcibly and treats it again (see Procedure 2). Since the `MoveBack` is conducted at the end after the main body of the protocol, the forcible insertion of the Move statement does not corrupt the logic of the protocol. If all data qubits arrive at the target positions, the `MoveBack` for all the data qubits is completed.



(a) (b) (c)

Figure 10. Conceptual transformation of the QASM by adding the `Move` instruction: (a) → (b) → (c). The directed acyclic graph (DAG) of the protocol is generated from the QASM in (b), and the transformation from (b) to (c) happens in the nodes of DAG. (a) An example of a normal QASM (b) QASM with `Move` instruction with the symbolic representation of the initial position of the data qubits, and (c) QASM with a specific value of the initial position of the data qubits. As mentioned above, at the beginning of each SABRE iteration, the initial qubit mapping table is randomly generated. The specific initial position of the data qubits, e.g. as shown in the figure $\{\cdots, data0^{init}$: 1, $data1^{init}$: 13, .., $data6^{init}$: 25, $\cdots\}$, is determined from the qubit mapping table.

4.4. Circuit Synthesis of Hierarchically Categorized FT Protocols

To implement universal fault-tolerant quantum computing, we need a set of fault-tolerant quantum protocols (and therefore circuits): logical `H`, `T`, and `CNOT` gates with the fault-tolerant preparation `PrepZ` and measurement `MeasZ` of the logical state in the Z basis. All the protocols (circuits) should act on the logical qubit having the identical qubit arrangement. For that, the circuit synthesis of each protocol must share the positions of the data qubits of a logical qubit. Otherwise, if we perform the circuit synthesis of each protocol independently, we will obtain the most optimized circuit of the protocol, but it will work on a different qubit arrangement than others. In that case, as mentioned before, we need an extra operation that tunes the position of the data qubits.

To this end, we divide all the protocols into two categories, *pivot* and *non-pivot* protocols, perform the circuit mapping on the pivot protocols first, and then use their result as a reference for the task regarding the non-pivot protocols. In the circuit mapping on the non-pivot protocols, we anchor the positions of the data qubits from the results of the circuit mapping on pivot protocols when choosing a random initial mapping.

Then, with which criterion can the protocols be categorized? The circuit of a pivot protocol is generated from scratch without any limitation on qubit allocation and therefore it can be highly optimized by the circuit mapping method itself. However, the circuit of a non-pivot protocol is limited by the result of the pivot protocol. In

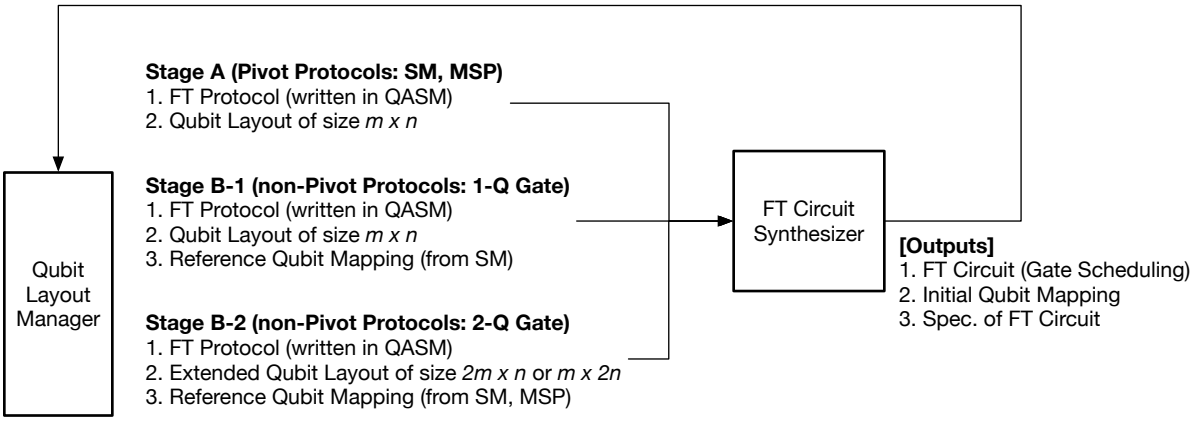


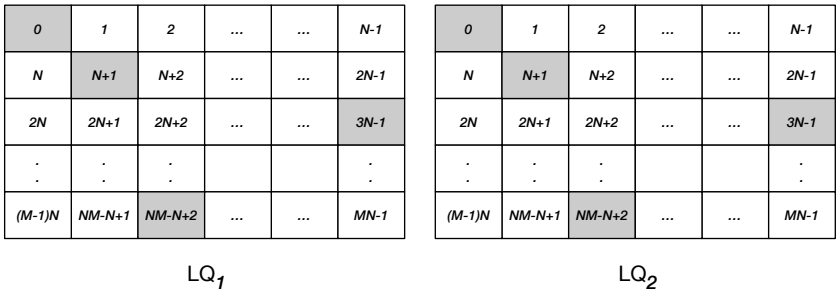
Figure 11. Hierarchical fault-tolerant circuit synthesis for universal fault-tolerant quantum computing. *FT Circuit Synthesizer* is implemented by applying the proposed ideas enumerated in Sections 4.1-4.3. *Qubit Layout Manager* plays a role in making an extended qubit layout according to the relative position between the logical qubits (or magic state). Note that SM and MSP respectively indicate syndrome measurement and magic state preparation. The numbered items in each stage indicate the inputs for the circuit synthesis.

this circumstance, for better computing performance, it is reasonable to choose the most frequently executed protocols as the pivot, and such is, evidently, the syndrome measurement in fault-tolerant quantum computing. We, therefore, configure the logical qubit, the arrangement of physical qubits, from the initial qubit mapping of the circuit mapping on the syndrome measurement. Please see Figure 11 for the entire circuit synthesis procedure for universal FT quantum computing.

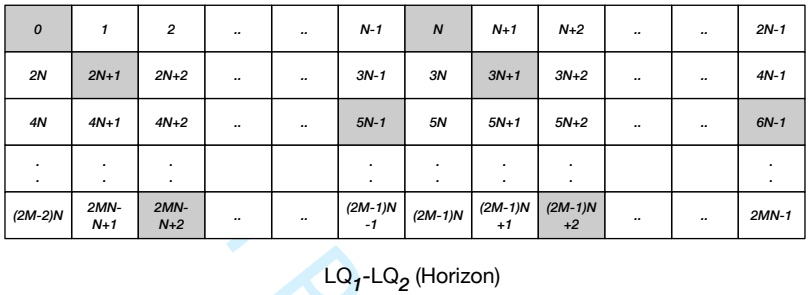
We first focus on the 1-qubit logical operations: `PrepZ`, `MeasZ`, and `H` gate. For the operations that can be implemented as transversal, the circuit mapping is very trivial. Therefore, here we discuss how to implement the fault-tolerant circuit for `PrepZ` only.

`PrepZ` makes the quantum state of a logical qubit (or a code block) to a logical zero state $|0\rangle_L$. Note that a logical plus state $|+\rangle_L$, $(|0\rangle_L + |1\rangle_L)/\sqrt{2}$, is implemented by additionally applying a logical `H` gate to $|0\rangle_L$. Before `PrepZ`, all the component qubits of a logical qubit have garbage state and therefore need not be logically distinguished. However, by running the circuit of `PrepZ`, the data qubits of a logical qubit should be placed at the designated locations that are determined in the circuit synthesis of the syndrome measurement protocol. Therefore, the `MoveBack` operation should be performed after the main body of `PrepZ`; however, the destination of each data qubit is specified from the logical qubit.

For 2-qubit logical gates, such as the `CNOT` gate, the proposed method treats the protocol as one that acts on a single extended qubit layout. Suppose that a logical qubit is defined on the 2-dimensional rectangular qubit layout of size $m \times n$. We then extend the layout as $2m \times n$ vertically or $m \times 2n$ horizontally, and allocate the component qubits of two logical qubits on it properly (see Figure 12). In doing so, all the physical operations for the 2-qubit logical gate act within two logical qubits. For reference,



(a)



(b)

Figure 12. Example of the qubit layout extension. (a) Two copies of a single logical data block of the size $m \times n$ and (b) The horizontally extended qubit layout, of the size $m \times 2n$. The qubit layout can be extended in the vertical direction as well. Gray cells indicate the data qubits in the logical qubit block. The physical index of each data qubit should be relabeled in the extended qubit layout.

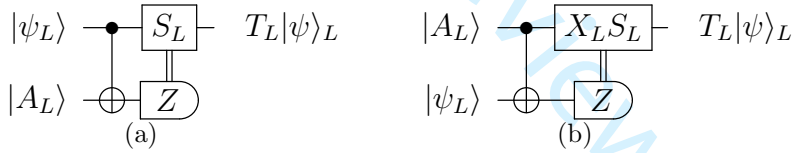


Figure 13. Logical `T` gate protocol that measures the magic state $|A\rangle_L$. `S` gate correction is required conditioned on the measurement outcome. (a) Protocol that measures the magic state and (b) Protocol that measures the logical data qubit [41].

Ref. [27] uses the qubits outside the two logical qubits concerned as a communication channel.

A certain error-correcting code implements a logical `T` (and `S`) gate as a 2-qubit gate protocol acting on a logical data qubit and a magic state (see Figure 13). In that case, the magic state preparation protocol (see Figure 13 of Ref. [21]) should be synthesized first and the circuit of the `T` gate then is synthesized on the extended qubit layout. The layout extension is based on the qubit mappings of both the logical qubit and the magic state.

The circuit of a logical `T` gate can be synthesized using two approaches: a combination of basic quantum gates and a single complex gate. In the first, as shown in Figure 15 (a), we assemble the circuits of the logical `CNOT`, `MeasZ`, and `S` gates. For this,

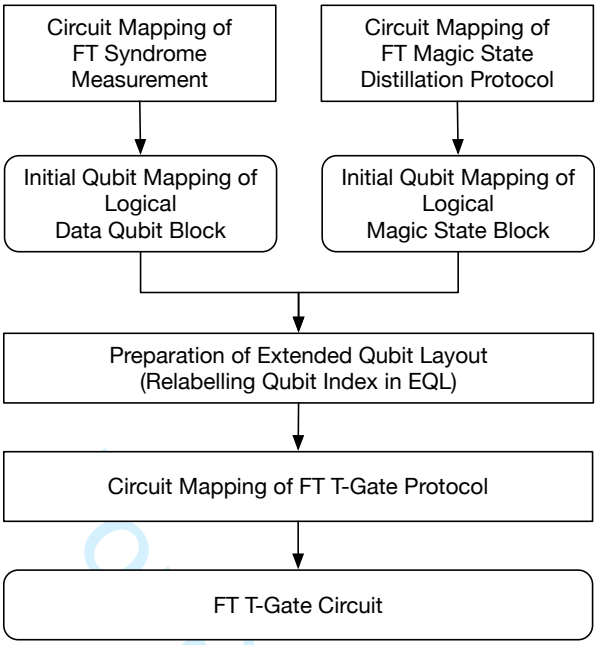


Figure 14. Flow of the circuit synthesis for the logical `T` gate protocol. After mapping the pivot protocols (syndrome measurement and magic state preparation) respectively, it is possible to determine the extended qubit layout for the logical `T` gate protocol.

the *magic qubits* of a magic state should be placed at the corresponding positions of the data qubits of a logical qubit to perform a logical `CNOT` gate. Please note that in this paper, we call the data qubits of a magic state the magic qubits. That is, the circuit of a magic state preparation should include the `MoveBack` operation for the magic qubits with respect to the logical qubit configuration. For that, the following instruction is required: "`Move` $magic[i]$ $destination[i]$" where $destination[i]$ corresponds to the index of the $data[i]$ of the logical qubit in the extended layout to perform the logical `CNOT` gate circuit that is already synthesized.

On the contrary, the circuit can be synthesized as a single complex gate as well. In this case, unlike the first case, we do not need to consider the `MoveBack` in the magic state preparation because it is possible to generate a circuit based on the current positions of magic qubits $magic[i]$ immediately after the main body of the preparation protocol. Besides, after the `CNOT` gate between logical data qubit and magic state, the magic qubits do not need to be back at their initial position. Note that the logical `CNOT` gate circuit includes `MoveBack` of both logical qubits. Therefore, this synthesis requires fewer qubit moves than the former approach. To make a compact circuit, it is better to adopt the second approach. Figure 15 shows a comparison between the two approaches. Note that we applied this approach to the circuit synthesis for the magic state preparation that includes syndrome measurement as a sub-process.

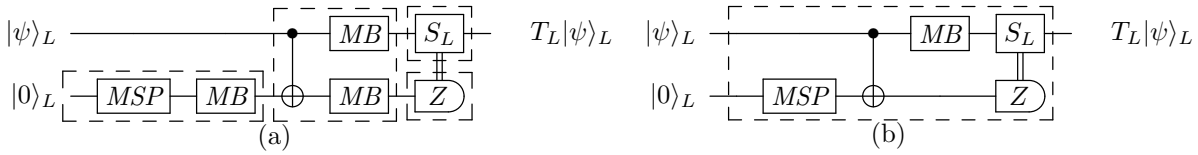


Figure 15. Comparison of the `T` circuits according to the synthesis approach. (a) The combination of basic gates. (b) Single complex gate. Note that *MSP* and *MB* refer to the protocol of the magic state preparation and the `MoveBack` operation, respectively. In (b), *MB* can be applied after S_L-correction as well.

Table 1. Conditions for the operations introduced in this work to be executable

operation	condition
`Move`	a qubit arrives at the destination
`Barrier`	`Barrier` node only remains in *FL*

4.5. Summary of Proposed Circuit Synthesis

We summarize the proposed circuit synthesis by representing the pseudocodes. Overall, the entire structure is similar to that of SABRE, shown in Figure 2, except that before building *DAG* we insert the `MoveBack` operation for the data qubits. The introduction of `MoveBack` can be done via two options: automatic insertion in the synthesizer and manual insertion by user.

The graph traversal and its sub-procedures have been customized for our purpose as follows. First, the graph traversal is outlined in Procedure 2. The lists shown in lines 1 and 2 are used to process the `Barrier` statement for the circuit partitioning and `MoveBack` for the self-contained circuit. The list *list_qubits_moved_back* is filled with the qubits that already arrived at their initial locations when finding executable gates (shown in line 4). Later, the algorithm examines whether a qubit that stays at its home is shifted to another place later (see lines 11-15). If the qubit is moved out, it is appended to the list *list_qubits_moved_back* again.

The key differences between the proposed method and the original SABRE are as follows. First, ***find_executable_gates_FT()*** shown in line 4, besides the conventional quantum gates, examines whether the additional instructions, `Move` and `Barrier` statement, are executable. Table 1 lists the executable conditions for those instructions again. The lists *list_executable_gates* and *list_qubits_moved_back* respectively holds the executable gates and the data qubits that arrived at their initial positions by `MoveBack`. A detailed description is included in Procedure 3. As shown in line 6, the executable gates need to be exported to a circuit being generated; then, update *FL* by checking the logical dependency. If the executable gate corresponds to `PrepZ` or `MeasZ` then the qubit's usage status table also should be changed (see Procedure 4). Writing an executable gate on the circuit is carried out in the final forward graph traversal only, not in the first and second traversals. The statements in lines 4-15 assign the usage status of a qubit according to a quantum gate and those

Procedure 2 Outline of The Graph Traversal for FT Circuit

Input: Front Layer FL, Qubit Mapping π, Distance Matrix D, DAG, Qubit Coupling Graph G

Output: Gate Scheduling, Final Qubit Mapping π_f

1: List for `Barrier`, $list\_for\_barrier = []$
2: List for `MoveBack`, $list\_for\_moveback = []$
3: **while** $FL \neq \emptyset$ **do**
4: $list\_executable\_gates$, $list\_qubits\_moved\_back$ = ***find_executable_gates_FT(FL)***
5: **if** $list\_executable\_gates \neq \emptyset$ **then**
6: **Treat** $list\_executable\_gates$ **for FT Circuit and Update** FL
7: **else**
8: $SWAP\_candidate\_list$ = ***obtain_SWAP_FT(FL, G)***
9: Evaluate $SWAP\_candidate\_list$ and Select an optimal one, $swap_{optimal}$
10: $\pi = \pi.update(swap_{optimal})$
11: **for** $qubit$ in [$swap_{optimal}.qubit1$, $swap_{optimal}.qubit2$] **do**
12: **if** $qubit \in list\_qubits\_moved\_back$ **then**
13: $list\_for\_moveback.append(qubit)$
14: **end if**
15: **end for**
16: **end if**
17: **if** FL is $\emptyset$ **then**
18: **while** $list\_for\_barrier \neq \emptyset$ **do**
19: $gate = list\_for\_barrier.pop()$
20: $FL.append(gate)$
21: **end while**
22: **while** $list\_for\_moveback \neq \emptyset$ **do**
23: $gate = list\_for\_moveback.pop()$
24: $FL.append(gate)$
25: **end while**
26: **end if**
27: **end while**

in lines 23-37 update FL after checking the logical dependency of the executable gate within DAG.

Second, ***obtain_SWAP_FT()*** in line 8 collects the `SWAP` candidates by limitedly taking a `SWAP` gate according to the type of qubits (see Procedure 5). It is the most critical function in retaining fault tolerance. For the `Barrier` statement and the `MoveBack` operation in the backward traversal, we do not need to perform `SWAP` gates (lines 3-5) because the purpose of the backward graph traversal is to find a proper initial mapping for the last forward traversal [32]. As mentioned in Section 4.1, when a data qubit is surrounded by all the data qubits, we need to move its neighbor qubits to somewhere else first (lines 20-35). However, for the `Move` operation, collecting `SWAP` candidates acting on the target qubit depends on the type of quantum information that stays there (lines 14-18).

The procedure to use the proposed method to find a set of fault-tolerant quantum

Procedure 3 *find_executable_gates_FT()*

Input: Front Layer *FL*, Qubit Coupling Graph *G*
Output: List of executable gates *list_executable_gates*, List of qubits moved back to initial position *list_qubits_moved_back*

```
list_executable_gates = []
list_qubits_moved_back = []
for gate in FL do
    if gate ∈ list of 1-qubit gates then
        list_executable_gates.append(gate)
    else if gate ∈ list of 2-qubit gates then
        if gate.ctrl ∈ G[gate.trgt] then
            list_executable_gates.append(gate)
        end if
    else if gate == Move then
        if gate.qubit == gate.destination then
            list_executable_gates.append(gate)
            list_qubits_moved_back.append(gate.qubit)
        end if
    else if gate == Barrier then
        if there does not exist any non-trivial gates except for the Barrier in FL then
            list_executable_gates.append(gate)
        end if
    end if
end for
```

circuits is shown in Figure 11. Therefore, we have not described the details for generating the circuit libraries.

5. Example : Syndrome Measurement of $[[7, 1, 3]]$ Steane Code

We present the fault-tolerant quantum circuit for the stabilizer measurement of $[[7, 1, 3]]$ Steane code. We apply the Steane-EC method, which utilizes logical ancilla states ($|0\rangle_L$, $|+\rangle_L$) of the code [21, 40, 42] to extract syndrome (see Figure 3). The protocol is composed of the Z-stabilizer measure and the X-stabilizer measure in serial, and the Z (X)-stabilizer measure is composed of the fault-tolerant preparation of $|0\rangle_L$ ($|+\rangle_L$) and a sequence of `CNOT` gates between data qubits and the syndrome qubits.

For the fault-tolerant preparation of $|0\rangle_L$ ($|+\rangle_L$), we apply Goto's single qubit verification method [43] rather than the method comparing two copies of logical states [21]. In Goto's method, a non-FT logical ancilla state is examined using a single checkup qubit. If the verification succeeds, the main body of the syndrome measurement begins. Otherwise, the logical ancilla state must be prepared again. However, we do not take the repetition caused by the non-deterministic feature into account throughout the present work as its purpose is to generate a fault-tolerant circuit.

Figure 16 shows the QASM of the protocol that includes the `Barrier` statements.

Procedure 4 How to treat the executable gates and update *FL*

Input: *list_executable_gates*, Front Layer *FL*, Qubit Coupling Graph *G*, Qubit Usage Status *qubit_status*, Graph Traversal Direction, *direction*, Temp List for `Barrier` *list_for_barrier* Temp List for Move-Back *list_for_moveback*

Output: *SWAP_candidate_list*, *qubit_status*

```
List of executed gates, list_executed_gates = []
for gate in list_executable_gates do
    Export gate to the circuit (only for the last forward graph traversal)
    if gate == PrepZ then
        if direction == forward then
            qubit_status[gate.qubit] = activated
        else
            qubit_status[gate.qubit] = inactivated
        end if
    else if gate == MeasZ then
        if direction == forward then
            qubit_status[gate.qubit] = inactivated
        else
            qubit_status[gate.qubit] = activated
        end if
    else if gate == Barrier then
        if list_for_barrier ≠ ∅ then
            FL.extend(list_for_barrier)
            Initialize list_for_barrier
        end if
    end if
    FL.remove(gate)
    list_executed_gates.append(gate)
    for successor in gates.successor_gates do
        ancestors = successor.predecessor_gates
        if ancestors ⊂ list_executed_gates then
            if gate == MOVE then
                list_for_moveback.append(successor)
            else
                if Barrier ∈ FL then
                    list_for_barrier.append(successor)
                else
                    FL.append(successor)
                end if
            end if
        end if
    end for
end for
```

After the translation of the QASM to a DAG, the circuit synthesis begins. For the 2-dimensional qubit layout of size 5×7, we have obtained a fault-tolerant quantum

Procedure 5 *obtain_SWAP_FT()*

Input: Front Layer FL, Qubit Coupling Graph G, Graph Traversal Direction *direction*, Qubit Mapping π, Qubit Usage Status *qubit_status*

Output: *SWAP_candidate_list*

SWAP_candidate_list = ∅
for *gate* in *FL* **do**
  if (*gate* == `Barrier`) or (*direction* == backward and *gate* == `Move`) **then**
    continue
  end if
  if *gate* ∈ [`CNOT`, `SWAP`, `CZ`] **then**
    physical[*gate.ctrl*] = π(*gate.ctrl*)
    physical[*gate.trgt*] = π(*gate.trgt*)
  else if *gate* == MOVE **then**
    physical[*gate.ctrl*] = π(*gate.ctrl*)
    physical[*gate.trgt*] = *gate.trgt*
  end if
  for *qubit* in [*gate.ctrl*, *gate.trgt*] **do**
    if *qubit* == *gate.trgt* and *gate* == `Move` **then**
      if *qubit* ∉ [*data*, *magic*] **then**
        temp_swap = [(π^{-1}(*qubit*), π^{-1}(*neighbor*)) for *neighbor* in *G*[*physical*[*qubit*]] if *qubit_status*[π^{-1}(*neighbor*)] == *inactivated*]
        SWAP_candidate_list.extend(*temp_swap*)
      end if
    else
      if *qubit_status*[*qubit*] == *inactivated* **then**
        temp_swap = [(*qubit*, π^{-1}(*neighbor*)) for *neighbor* in *G*[*physical*[*qubit*]]]
        SWAP_candidate_list.extend(*temp_swap*)
      else
        for *neighbor* in *G*[*physical*[*gate.ctrl*]] **do**
          if *qubit_status*[*neighbor*] == *inactivated* **then**
            SWAP_candidate_list.append((*qubit*, π^{-1}(*neighbor*)))
          else
            if *counter* < *max_allowable_interaction* **then**
              SWAP_candidate_list.append((*qubit*, π^{-1}(*neighbor*)))
            end if
            temp_swap = [(π^{-1}(*neighbor*), π^{-1}(*j*)) for *j* in *G*[*neighbor*] if *qubit_status*[π^{-1}(*j*)] == *inactivated*]
            SWAP_candidate_list.extend(*temp_swap*)
          end if
        end for
      end if
    end if
  end for
end for

circuit of depth 35. Please see Figures A1-A7; each figure shows each part of the circuit: FT preparation of $|+\rangle_L$, `CNOT` between *data*[i] and *syndrome*[i] over $i = 0 \cdots 6$,

```
PrepZ syndrome[0]              PrepZ syndrome[0]
PrepZ syndrome[1]              PrepZ syndrome[1]
PrepZ syndrome[2]              PrepZ syndrome[2]
PrepZ syndrome[3]              PrepZ syndrome[3]
PrepZ syndrome[4]              PrepZ syndrome[4]
PrepZ syndrome[5]              PrepZ syndrome[5]
PrepZ syndrome[6]              PrepZ syndrome[6]
PrepZ checkup[0]               PrepZ checkup[0]
H syndrome[0]                  H syndrome[1]
H syndrome[4]                  H syndrome[2]
H syndrome[5]                  H syndrome[3]
H syndrome[6]                  CNOT syndrome[1], syndrome[0]
CNOT syndrome[0], syndrome[1]  CNOT syndrome[3], syndrome[5]
CNOT syndrome[5], syndrome[3]  CNOT syndrome[2], syndrome[6]
CNOT syndrome[6], syndrome[2]  CNOT syndrome[1], syndrome[4]
CNOT syndrome[4], syndrome[1]  CNOT syndrome[2], syndrome[0]
CNOT syndrome[0], syndrome[2]  CNOT syndrome[3], syndrome[6]
CNOT syndrome[6], syndrome[3]  CNOT syndrome[1], syndrome[5]
CNOT syndrome[5], syndrome[1]  CNOT syndrome[6], syndrome[4]
CNOT syndrome[4], syndrome[6]  CNOT syndrome[0], checkup[0]
CNOT checkup[0], syndrome[0]   CNOT syndrome[5], checkup[0]
CNOT checkup[0], syndrome[5]   CNOT syndrome[6], checkup[0]
CNOT checkup[0], syndrome[6]   MeasZ checkup[0]
MeasZ checkup[0]               Barrier
Barrier                        CNOT syndrome[0], data[0]
CNOT data[0], syndrome[0]      CNOT syndrome[1], data[1]
CNOT data[1], syndrome[1]      CNOT syndrome[2], data[2]
CNOT data[2], syndrome[2]      CNOT syndrome[3], data[3]
CNOT data[3], syndrome[3]      CNOT syndrome[4], data[4]
CNOT data[4], syndrome[4]      CNOT syndrome[5], data[5]
CNOT data[5], syndrome[5]      CNOT syndrome[6], data[6]
CNOT data[6], syndrome[6]      H syndrome[0]
MeasZ syndrome[0]              H syndrome[1]
MeasZ syndrome[1]              H syndrome[2]
MeasZ syndrome[2]              H syndrome[3]
MeasZ syndrome[3]              H syndrome[4]
MeasZ syndrome[4]              H syndrome[5]
MeasZ syndrome[5]              H syndrome[6]
MeasZ syndrome[6]              MeasZ syndrome[0]
Barrier                        MeasZ syndrome[1]
                               MeasZ syndrome[2]
                               MeasZ syndrome[3]
                               MeasZ syndrome[4]
                               MeasZ syndrome[5]
                               MeasZ syndrome[6]
```

Figure 16. QASM of the stabilizer measurement of $[[7, 1, 3]]$ Steane code [40, 21, 42]. Note that the declaration of qubits and classical bits has been omitted. The Z (X)-type stabilizer measure is shown in the left (right) column. From the first line to the line just before the `Barrier` statement the procedure of the logical ancilla state preparation (of Ref. [43]) is described, and the main procedure of the syndrome measure is shown thereafter.

FT preparation of $|0\rangle_L$ and `CNOT` between *syndrome*[i] and *data*[i] over $i = 0 \cdots 6$. The initial qubit mapping for the circuit is shown in Figure 17. Figure A8 shows the text representation of the circuit in JSON format, which is generated from our implementation of the circuit synthesis algorithm.

By processing the `Barrier` statements, the entire circuit is separated into 4 sub-circuits (please see Figures A1-A7). Besides, using the `MoveBack` operation, all data qubits will be placed at their initial positions after all the quantum operations. Please

data[5]	**data[3]**	syndrome[5]	dummy[5]	**data[1]**	**data[0]**	dummy[18]
dummy[13]	dummy[10]	syndrome[3]	dummy[6]	syndrome[1]	dummy[14]	**data[2]**
dummy[3]	dummy[2]	syndrome[6]	checkup[0]	syndrome[0]	dummy[17]	dummy[15]
data[6]	dummy[0]	syndrome[2]	dummy[7]	dummy[1]	syndrome[4]	dummy[19]
dummy[11]	dummy[16]	dummy[4]	dummy[8]	dummy[9]	dummy[12]	**data[4]**

Figure 17. The initial qubit mapping for the fault-tolerant circuit of the syndrome measurement that is shown continuously over Figures A1-A7.

Table 2. Protocols for Steane code based universal FTQC

Type	Protocols
Pivot	Syndrome Measurement, Magic State Preparation
Non-Pivot	`PrepZ`, `MeasZ`, `H`, `T`, `S`, `CNOT`

compare Figure 17 with Figure A7 (f).

6. Fault-Tolerant Circuits of $[[7,1,3]]$ Steane code FTQC

This section shows how to prepare a full set of fault-tolerant quantum circuits for $[[7,1,3]]$ Steane code. For this, we first need to determine the protocols required and then classify them into two categories: *pivot* and *non-pivot*. Table 2 shows the protocols required for universal fault-tolerant quantum computing divided into two categories.

Before proceeding to the circuit synthesis for the pivot protocols, let us discuss the qubit layout first. The qubit layout affects the cost and the performance of logical qubits, logical gates, and therefore that of the entire quantum computing. Thus, there is a need to pay close attention to it. We believe that there exists an optimal qubit layout in terms of the circuit size KQ defined as $\#qubit \times circuit\_depth$ [44]. As the size of the layout grows, the number of qubits increases; however, the circuit depth decreases until the lower bound. In this regard, we compare the circuit sizes over various qubit layouts and select an optimal one.

6.1. Optimal Qubit Layout

Over 2-dimensional qubit layouts of size 5×6 - 7×8, we quantify the size of the quantum circuit for the syndrome measurement based on two error correction methods, Shor-EC and Steane-EC. For Steane-EC [21, 40, 42], we exploit the fault-tolerant preparation of the logical ancilla states $|0\rangle_L$ (and similarly $|+\rangle_L$) of Ref. [43] (see Figure 1 therein). For Shor-EC [5, 21, 42], we apply the fault-tolerant preparation of the length-4 cat state described in Ref. [21] (see Figure 6 therein). To select the best qubit layout, we evaluate the circuits in terms of their size. In this work, we focus on the case that prepares ancilla states in serial, and not in parallel.

Table 3. Size of the fault-tolerant circuits about the syndrome measurement over various 2-dimensional qubit layouts. In the table, *ideal* indicates the static analysis of the protocol itself. The number in parentheses indicates the number of qubits of the corresponding qubit layout. Note that the protocols based on Steane-EC and Shor-EC are composed of 15 and 12 qubits.

Method	item	*ideal*	5×6 (30)	5×7 (35)	6×6 (36)	5×8 (40)	6×7 (42)	6×8 (48)	7×7 (49)	7×8 (56)
Steane-EC	*depth*	18	43	35	41	40	41	41	40	38
	#gates	83	156	168	162	158	164	165	175	155
	KQ	270	1,290	1,225	1,476	1,600	1,722	1,968	1,960	2,128
Shor-EC	*depth*	44	55	53	54	55	53	55	52	54
	#gates	144	206	204	199	224	212	212	224	216
	KQ	528	1,650	1,855	1,944	2,200	2,226	2,640	2,548	3,025

Table 3 shows the circuit sizes of fault-tolerant syndrome measurements over the qubit layouts by adopting *#gates* and *circuit_depth* as the performance evaluation criteria. It is well known that, theoretically, Shor-EC based protocol requires fewer qubits but more quantum gates than Steane-EC based one [21, 27]. Therefore, for fault tolerance, the latter is preferred. As presented in the table, our results show the same tendency. Based on the table, in the following section, we develop the circuits of fault-tolerant quantum protocols with a qubit layout of size 5×7. For reference, in Ref. [27] the qubit layout of size 6×8 is applied to optimize the threshold than to optimize the space-time resource in the local setting.

Before going further, we emphasize again that since our proposed algorithm is heuristic and non-deterministic, the results the table shows are not fixed for the layout. They are just the most compact ones we have obtained so far, but there may exist more optimized circuits.

6.2. Full Set of Fault-Tolerant Quantum Circuits

We synthesize the protocols listed in Table 2 in the following order. The first is the syndrome measurement and the magic state preparation. Since the fault-tolerant quantum circuit and the associated qubit mapping of the syndrome measurement are already described in Section 5, we discuss the magic state preparation only.

We take the protocol described in Ref. [21] (see Figure 13 therein). It is composed of the repetition of the preparations of ancilla states (logical zero state and 7-qubit cat state), $\bar{\mathtt{T}}\bar{\mathtt{X}}\bar{\mathtt{T}}^{\dagger}$ measurement, and error detection. Since the ancilla preparations and error detection include selection of the following operation conditioned on the measurement outcomes, we apply the circuit partitioning we discussed earlier. For that, we insert the `Barrier` statements in the protocol - a total of 10 times. The fault-tolerant preparations of the logical ancilla and the cat state respectively require a single checkup qubit. Therefore, by reusing qubits, the protocol is composed of at least 15 qubits: 7 for data, 7 for ancilla, and 1 for verification. Finally, we emphasize that in the circuit

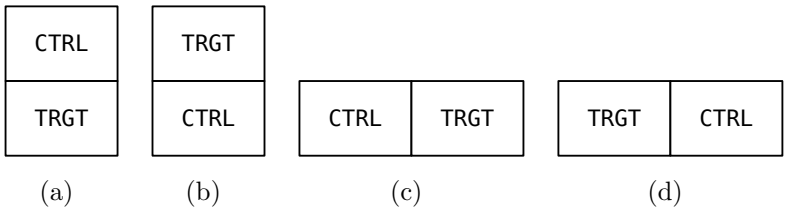


Figure 18. The relative arrangement of two logical qubits.

Table 4. Static analysis of the fault-tolerant quantum circuits for universal fault-tolerant quantum computing. The numbers of `SWAP` gates and `Barrier` indicate the quantities of those introduced in the circuit synthesis. Note that the quantities of the other gates in the circuit are precisely the same as those included in the protocol. Note that MB indicates the `MoveBack` operation.

Operation	Protocol				Circuit			
	Reference	#Qubits	Depth	Gates	Depth	#SWAP	MB	#Barrier
Syndrome Measurement	[40, 21]	15	20	CNOT: 36, H: 15, PrepZ: 16, MeasZ: 16	35	80	O	3
Encoder	[43]	8	8	CNOT: 11, H: 3, PrepZ: 8, MeasZ: 1	18	31	O	-
Magic State Preparation	[21]	15	75	CNOT: 113, H: 49, PrepZ: 49, MeasZ: 49, T: 14, $T^\dagger$: 14	162	323	X	10
h-CNOT	Transversal	14	1	CNOT: 7	21	136	O	-
v-CNOT	Transversal	14	1	CNOT: 7	13	80	O	-
h-T (d_e, m_w)	Figure 15 (b)	14	3	CNOT: 7, MeasZ: 7, S: 7	22	96	O	1
h-T (d_w, m_e)	Figure 15 (b)	14	3	CNOT: 7, MeasZ: 7, S: 7	20	104	O	1
v-T (d_n, m_s)	Figure 15 (b)	14	3	CNOT: 7, MeasZ: 7, S: 7	17	71	O	1
v-T (d_s, m_n)	Figure 15 (b)	14	3	CNOT: 7, MeasZ: 7, S: 7	24	81	O	1

synthesis of the magic state preparation (or distillation for other code), the final qubit mapping after running the circuit is critical to the `T` gate circuit.

For the 1-qubit gate, the circuit synthesis of `PrepZ` is sufficient because the other gates can be implemented as the transversal gates. As mentioned several times, we take the protocol that consumes a single checkup qubit protocol [43]. In the circuit of `PrepZ`, with respect to the position of the data qubits, the final qubit mapping should be consistent with that in the logical qubit.

For the 2-qubit gates such as `CNOT` and `T`, we must generate the circuits based on all the cases of how two qubits are relatively arranged on the logical qubit layout (see Figure 18). However, in the case of `CNOT`, all the data qubits in both logical qubits must move back to their initial positions. Therefore, the circuits for layouts (a) and (c) can directly be used for layouts (b) and (d) by changing the `CNOT` directions. On the contrary, for the `T` gate, as mentioned before, the `MoveBack` for the magic state is not required, and therefore the circuit for layout (a) cannot be applied for case (b). Similarly for layouts (c) and (d). Therefore, we conduct the circuit synthesis of the `T` gate over all four cases, while the same is performed for only two cases for the `CNOT` gate. Figures A9-A13 show all the snapshots of the `CNOT` gate for the qubits arranged vertically, ($ctrl_n$, $trgt_s$), while Figures A14-A20 show all the snapshots of the `T` gate for the qubits arranged vertically, ($data_n$, $magic_s$).

To conclude this section, we list the setting and the result of the circuit synthesis

over all the protocols shown in Table 2 (see Table 4). Note that for `S` gate correction in the `T` gate, we apply the transversal gate rather than the state injection method like the `T` gate protocol. For fault tolerance, the state injection based method is preferred [27]; however, for the purpose of this work, the transversal gate protocol is sufficient.

7. [[23, 1, 7]] Golay Code

[[23, 1, 7]] Golay code is well known as a high error-correction threshold [21, 22]; however, the realization of universal fault-tolerant quantum computing with that code has rarely been studied. Due to its large code block (logical qubit), it is challenging to find fault-tolerant quantum circuits [23, 45].

Further, it is also tricky and time-consuming to find the circuits with the proposed method. In this work, we apply the *divide-and-conquer* approach to find the fault-tolerant quantum circuit of the syndrome measurement efficiently. That is, we divide the entire protocol into several sub-protocols, obtain the fault-tolerant circuit for each sub-protocol by applying the proposed algorithm, and finally combine all the sub-circuits.

We hire the preparation of a logical zero state in Ref. [23], and divide it into the non-FT preparation stage of $|0\rangle_L$ and the verification stage (see Figures 3 and 4 therein). We then conduct the circuit synthesis of the former and apply the result to the synthesis of the latter. Please note that the fact that the former acts in a non-fault-tolerant manner does not imply that its non-FT circuit is enough for that. By using the divide-and-conquer method, the entire circuit can be naturally partitioned, and classical control operations can be conducted between the circuits.

Let us assume that the size of the qubit layout for the above non-FT preparation circuit is $m \times n$. We then need an extended qubit layout of size $2m \times 2n$ of the fault-tolerant circuit for the verification part (see Figure 20). The initial qubit mapping is determined by four copies of the final mapping of the non-FT preparation circuit (see Figure 19). The method of extending the qubit layout has already been discussed in Section 4.4.

We now apply the circuit synthesis algorithm to find the fault-tolerant circuit functioning for the verification and then combine the resulting circuit with four copies of the non-FT circuit for preparing $|0\rangle_L$. We now have a fault-tolerant circuit to prepare a logical zero state. Since the code is self-dual, the fault-tolerant circuit for the preparation of $|+\rangle_L$ is almost the same as the FT circuit for preparing $|0\rangle_L$ in terms of the circuit synthesis [23].

As mentioned earlier, to define an optimal configuration of a logical qubit, we focus on the syndrome measurement protocol. In Golay code, the non-FT preparation of logical states $|0\rangle_L$ and $|+\rangle_L$ is the dominant part of the syndrome measurement protocol. Therefore, in this paper, we determine the configuration of a logical qubit from the circuit synthesis of the non-FT preparation protocol. Besides, since the fault-tolerant preparation of the logical states takes considerable time and space (qubits), we assume that the logical ancilla states are supplied from a factory, and not generated

data[3]				data[12]	data[8]	data[16]
	data[1]		data[18]			
data[15]	data[20]	data[22]	data[11]	data[5]	data[21]	
data[13]	data[6]	data[4]			data[9]	
			data[7]	data[17]		data[0]
			data[14]		data[19]	data[2]
	data[10]					

Figure 19. A qubit layout of size 7×7 for a logical zero state $|0\rangle_L$ in $[[23, 1, 7]]$ Golay code of Ref. [23]. Based on this layout of 49 qubits, we obtain a fault-tolerant circuit of the preparation protocol (Figure 4 of Ref. [23]). The circuit includes 269 `SWAP` gates to make 57 `CNOT` gates executable on the qubit layout without noisy `SWAP` gates between data qubits. Empty cells indicate dummy qubits working as communication channels.

$\|\bar{0}\rangle_1$	$\|\bar{0}\rangle_2$
$\|\bar{0}\rangle_3$	$\|\bar{0}\rangle_4$

(a)

$\|\overline{data}\rangle$
$\|\overline{anc}\rangle$

(b)

Figure 20. Qubit layouts for $[[23, 1, 7]]$ Golay code. (a) Extended qubit layout for encoding $|0\rangle_L$ in a fault-tolerant manner [23] (see Figure 3 therein). Each cell, shown in Figure 19, indicates the code block for the corresponding logical zero state $|\bar{0}\rangle_i$ prepared in a non-fault-tolerant manner. (b) Extended qubit layout for a logical qubit composed of the code blocks of data qubits $|\overline{data}\rangle$ and logical ancilla $|\overline{anc}\rangle$. Each cell is arranged as the code block shown in Figure 19, but the roles of the qubits in the block differ.

at the site. Therefore, with the 2-dimensional qubit layout of size 14 x 7, we define a logical qubit made of the 23 data qubits, the 23 ancilla qubits for a logical ancilla state, and some dummy qubits for the communication channel. A detailed description of the method to supply the logical ancilla states practically has been left as a future work.

Since the code distance of the Golay code is 7, it is possible to correct arbitrary 3-qubit errors. Therefore, as mentioned above, it is possible to allow a one-time noisy `SWAP` gate between data-type qubits during the syndrome measurement, $\lfloor (7-1)/4 \rfloor = 1$. Note that in the fault-tolerant preparation of logical ancilla states, we do not permit any noisy `SWAP` gate between the data-type qubits. Table 5 describes the procedure and the static data of the circuit synthesis of the fault-tolerant preparation of the logical zero states and the syndrome measurement. Owing to a lack of space, we did not present the full circuit of the syndrome measurement here.

Table 5. Process and static analysis for the circuit synthesis of $[[23, 1, 7]]$ Golay code. The entire circuit synthesis is performed in sequence: 1) Non-FT preparation of $|0\rangle_L$ ($|+\rangle_L$), 2) verification, and 3) syndrome measurement. Note that the value in parentheses in Circuit Depth indicates the circuit depth at the protocol level without the locality effect of the qubit layout. The circuit depth includes the physical preparation of $|0\rangle$.

	Protocol		
	Non-FT preparation of $\|0\rangle_L$ ($\|+\rangle_L$)	Verification	Syndrome Measurement
Qubit Mapping	$M_{init} \to \cdots \to M_{tempLQ}$	$M_{tempLQ} \to \cdots \to M_{LQ}$	$M_{LQ} \to \cdots \to M_{LQ}$ (`MoveBack`)
Size of Qubit Layout	$m \times n$	$2m \times 2n$	$2m \times n$ (or $m \times 2n$)
Allowable Noisy `SWAP` between Data Qubits	0	0	1
Circuit Depth (Ideal Depth)	51 (9)	21 (4)	11 (5)
# `SWAP`	269	296	50

8. Discussion

The circuit synthesis for an ordinary quantum algorithm always succeeds in finding a quantum circuit unless the size of a quantum chip (qubit layout) is less than the number of qubits in the quantum algorithm. According to a target case (quantum algorithm, qubit connectivity, and an initial qubit mapping), the quality of the resulting circuit or the efficiency of the synthesis procedure may be very poor; however, nothing makes the circuit mapping impossible fundamentally.

On the contrary, the situation differs for a fault-tolerant quantum protocol because the qubits must move in a fault-tolerant manner. If the size of a target quantum chip is equal to or is modestly bigger than the number of qubits in the fault-tolerant protocol, the circuit synthesis may fail to find a fault-tolerant circuit. It sometimes falls into an infinite loop. Particularly, the proposed algorithm tries to find a fault-tolerant quantum circuit based on a randomly picked initial qubit mapping - such a phenomenon that a circuit algorithm cannot generate a fault-tolerant circuit happens more frequently when the size of the quantum chip is not remarkably greater. In our implementation, we apply the time limit for the graph traversal. If the graph traversal is not completed within the time limit, we forcibly turn off the current SABRE iteration and begin the next iteration. The amount of time limit is proportional to the size of the protocol; in particular, the number of `CNOT` gates.

In the present work, we have not derived a lower bound on the size of a qubit layout for an n-qubit fault-tolerant quantum protocol. Finding such a bound is combinatorially challenging. It depends on the initial qubit mapping, the property of a quantum protocol (highly parallel or serial), the shape of the qubit layout, and so on. In this paper, we applied a *trial-and-error* approach to see a loose bound. We observed that for the Steane-EC based syndrome measurement protocol, the proposed algorithm succeeds in

finding a fault-tolerant quantum circuit acting on the qubit layout of 4×5, but of longer depth 72. Since the protocol itself is designed as using 15 qubits, the minimum requirement for the fault-tolerant communication may not be so large but at the cost of the circuit depth.

The proposed algorithm continuously selects a locally optimal `SWAP` gate. In that case, even though a selected `SWAP` gate was evaluated as optimal at a cost evaluation at the time, it may not play a significant role in the resulting circuit. Occasionally, for example, consecutively `SWAP` gates for a pair of qubits may be involved in the circuit. Note that this does not happen for the synthesis of ordinary computational algorithms.

In this work, even though we have not represented it explicitly, we deal with the problem in two stages. First, during the circuit mapping, if a selected `SWAP` gate at the current cost evaluation is the same as the previous one, then we reject the selection and take a second optimal one. Second, after each SABRE iteration, as post-processing for the resulting circuit, we cancel out the quantum instructions repeated consecutively on the same qubits. It may happen that a `SWAP` gate is applied on a pair of two qubits repeatedly at a distance of time.

The error-correction strength of the concatenated code is enhanced by recursively encoding local qubits with the same or another quantum code [20]. Throughout the paper, we have shown that the fault-tolerant quantum circuit in the concatenation level 1. The circuits for concatenation level l where $l > 1$ can be obtained by encoding the circuit in level 1 with the circuits in level $l - 1$ by following the principle of the code concatenation. In case a different quantum code is hired for a higher level, we first need to synthesize a set of the circuits for that code and then encode the bottom-level circuits with them.

This work defines the qubit configuration of a logical qubit from the circuit synthesis of the syndrome measurement, and then obtains compact fault-tolerant quantum circuits based on that configuration. For the 2-qubit gates, we have considered all the possible relative arrangements of logical qubits. As shown in Table 4, the sizes of the circuits are different according to the arrangement. This is because both the layout for a logical qubit and the arrangement of physical qubits in the logical qubit is not symmetric. Even though an individual quantum circuit is optimized in terms of the circuit depth, operating universal fault-tolerant quantum computing may be tricky because according to the qubit arrangement, the `CNOT` (`T`) gate works differently. In this regard, we need to consider a method to operate universal fault-tolerant quantum computing efficiently with the quantum circuits of non-uniform performance.

9. Conclusion

To date, various concatenated quantum codes and their fault-tolerant protocols have been studied theoretically; however, their realization in the local setting has been rarely discussed. For a few well-known small-sized codes, manually obtained fault-tolerant circuits have been reported. Even though the results may be optimal for the case, those

results or methodologies cannot be directly applied to different qubit layouts or quantum codes, in particular, one of larger code distance. Therefore, we strongly believe that an algorithmic method to synthesize fault-tolerant quantum circuits is required.

In the present work, as a systematic approach to the problem, we raised four requirements, presented our approaches for them, and described how to implement them with the existing heuristic quantum circuit mapping algorithm. As a result, it is now possible to obtain a set of fault-tolerant quantum circuits for an arbitrary concatenated quantum code by running the algorithm. However, since the proposed algorithm does not work deterministically, the presented static analysis data about the circuits are not completely fixed for the protocol and the qubit layout. To get the most optimized circuits, we need to iterate the synthesis rounds as much as possible.

Acknowledgments

YH is grateful to Jeongho Bang for the helpful comments. This work was partly supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00003, "Research and Development of Core Technologies for Programming, Running, Implementing and Validating of Fault-Tolerant Quantum Computing System" & No.2022-0-00463, "Development of a Quantum Repeater in Optical Fiber Networks for Quantum Internet") and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2019M3E4A1080146).

Reference

[1] Gottesman, D. *Stabilizer Codes and Quantum Error Correction.* Ph.D. thesis (1997).

[2] Gottesman, D. Theory of fault-tolerant quantum computation. *Physical Review A* **57**, 127–137 (1998). quant-ph/9702029.

[3] Knill, E., Laflamme, R. & Zurek, W. H. Resilient Quantum Computation. *Science* **279**, 342–345 (1998).

[4] Preskill, J. Reliable quantum computers. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **454**, 385–410 (1998).

[5] Shor, P. Fault-tolerant quantum computation. *Proceedings of 37th Conference on Foundations of Computer Science* 56–65 (1996).

[6] Knill, E. Quantum computing with realistically noisy devices. *Nature Publishing Group* **434**, 39 – 44 (2005). URL https://www.nature.com/articles/nature03350.

[7] Steane, A. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **452**, 2551–2577 (1996). URL http://rspa.royalsocietypublishing.org/content/452/1954/2551.

[8] Steane. Error Correcting Codes in Quantum Theory. *Physical review letters* **77**, 793–797 (1996).

[9] Preskill, J. The Physics of Quantum Information. *arXiv* (2022). 2208.08064.

[10] Kitaev, A. Y. Fault-tolerant quantum computation by anyons. *Annals of Physics* **303**, 2 – 30 (2003-01).

[11] Raussendorf, R. & Harrington, J. Fault-Tolerant Quantum Computation with High Threshold in Two Dimensions. *Physical Review Letters* **98**, 97 – 4 (2007).

[12] Fowler, A. G., Mariantoni, M., Martinis, J. M. & Cleland, A. N. Surface codes: Towards practical large-scale quantum computation. *Physical Review A* **86**, 8 – 48 (2012).

[13] Bombín, H. & Martin-Delgado, M. A. Topological Quantum Distillation. *Physical Review Letters* **97**, 180501 – 4 (2006).

[14] Kubica, A. & Beverland, M. E. Universal transversal gates with color codes: A simplified approach. *Physical Review A* **91**, 032330 – 12 (2015).

[15] Arute, F. *et al.* Quantum supremacy using a programmable superconducting processor. *Nature Publishing Group* 1 – 7 (2019).

[16] Chen, Z. *et al.* Exponential suppression of bit or phase errors with cyclic error correction. *Nature* **595**, 383–387 (2021).

[17] Acharya, R. *et al.* Suppressing quantum errors by scaling a surface code logical qubit. *arXiv* (2022). `2207.06431`.

[18] Shor, P. W. Scheme for reducing decoherence in quantum computer memory. *Physical Review A* **52**, R2493–R2496 (1995).

[19] Laflamme, R., Miquel, C., Paz, J. P. & Zurek, W. H. Perfect Quantum Error Correcting Code. *Physical Review Letter* **77**, 198—201 (1996).

[20] Knill, E. & Laflamme, R. Concatenated Quantum Codes (1996). URL `https://arxiv.org/abs/quant-ph/9608012`.

[21] Aliferis, P., Gottesman, D. & Preskill, J. Quantum accuracy threshold for concatenated distance-3 codes. *Quantum Information & Computation* **6**, 97–165 (2018).

[22] Cross, A. W., Divincenzo, D. P. & Terhal, B. M. A comparative code study for quantum fault tolerance. *Quantum Information and Computation* **9**, 541–572 (2009).

[23] Paetzick, A. & Reichardt, B. W. Fault-Tolerant Ancilla Preparation and Noise Threshold Lower Bounds for the 23-Qubit Golay Code. *Quantum Information and Computation* **12**, 1034–1080 (2012).

[24] Aharonov, D. & Ben-Or, M. Fault-Tolerant Quantum Computation with Constant Error Rate. *SIAM Journal on Computing* **38**, 1 – 241 (2008).

[25] Gottesman, D. Fault-tolerant quantum computation with local gates. *Journal of Modern Optics* **47**, 333–345 (2000).

[26] Svore, K. M., Terhal, B. M. & DiVincenzo, D. P. Local fault-tolerant quantum computation. *Physical Review A* **72**, 022317 (2005).

[27] Svore, K. M., DiVincenzo, D. P. & Terhal, B. M. Noise Threshold for a Fault-Tolerant Two-Dimensional Lattice Architecture. *Quantum Information and Computation* **7**, 297–318 (2007).

[28] Lai, C.-Y., Paz, G., Suchara, M. & Brun, T. A. Performance and Error Analysis of Knill's Postselection Scheme in a Two-Dimensional Architecture. *Quantum Information & Computation* **14**, 807–822 (2014). `1305.5657`.

[29] Spedalieri, F. M. & Roychowdhury, V. P. Latency in local, two-dimensional, fault-tolerant quantum computing. *Quantum Information & Computation* **9**, 666–682 (2009). `0805.4213`.

[30] Pedram, M. & Shafaei, A. Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures. *IEEE Circuits and Systems Magazine* **16**, 62 – 74 (2016).

[31] Zulehner, A., Paler, A. & Wille, R. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **38**, 1226–1236 (2019).

[32] Li, G., Ding, Y. & Xie, Y. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1001–1014 (2019).

[33] Murali, P., Baker, J. M., Javadi-Abhari, A., Chong, F. T. & Martonosi, M. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1015–1029 (2019).

[34] Tannu, S. S. & Qureshi, M. K. Not All Qubits Are Created Equal: A Case for Variability-Aware

Policies for NISQ-Era Quantum Computers. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 987–999 (2019).

[35] Niu, S., Suau, A., Staffelbach, G. & Todri-Sanial, A. A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era. *IEEE Transactions on Quantum Engineering* **1**, 1–14 (2020). 2010.03397.

[36] Nishio, S., Pan, Y., Satoh, T., Amano, H. & Van Meter, R. Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation. *ACM Journal on Emerging Technologies in Computing Systems* **16**, 1–25 (2020).

[37] Lao, L., Someren, H. v., Ashraf, I. & Almudever, C. G. Timing and Resource-Aware Mapping of Quantum Circuits to Superconducting Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **41**, 359–371 (2022). 1908.04226.

[38] Hwang, Y. Fault-Tolerant Quantum Circuit Synthesis. URL https://github.com/ETRIQuantum/FTCircuitSynthesis.

[39] Cross, A. W., Bishop, L. S., Smolin, J. A. & Gambetta, J. M. Open Quantum Assembly Language. *https://arxiv.org/abs/1707.03429* 1 – 24 (2017). URL https://arxiv.org/abs/1707.03429.

[40] Steane, A. M. Active Stabilization, Quantum Computation, and Quantum State Synthesis. *Physical Review Letters* **78**, 2252–2255 (1997). quant-ph/9611027.

[41] Zhou, X., Leung, D. W. & Chuang, I. L. Methodology for quantum logic gate construction. *Physical Review A* **62**, 385 (2000).

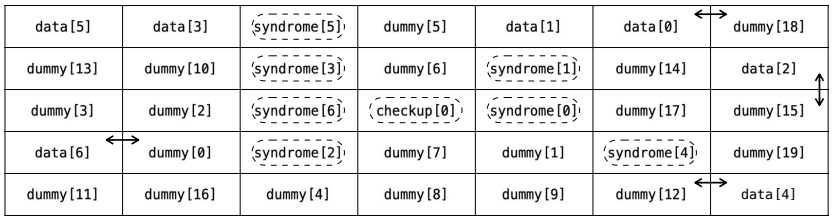
[42] Weinstein, Y. S. Syndrome measurement order for the [[7,1,3]] quantum error correction code. *Quantum Information Processing* **15**, 1263 – 1271 (2015).

[43] Goto, H. Minimizing resource overheads for fault-tolerant preparation of encoded states of the Steane code. *Scientific Reports* **5**, 1 – 7 (2016).

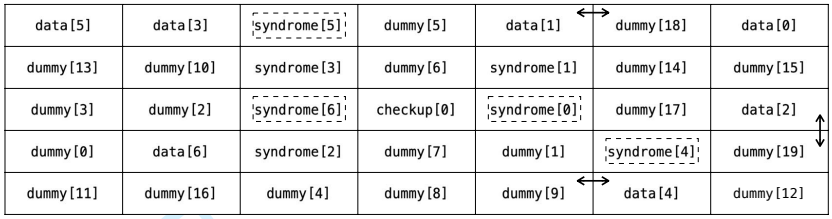
[44] Steane, A. M. Overhead and noise threshold of fault-tolerant quantum error correction. *Physical Review A* **68**, R2493 – 19 (2003).

[45] Zheng, Y.-C., Lai, C.-Y. & Brun, T. A. Efficient preparation of large-block-code ancilla states for fault-tolerant quantum computation. *Physical Review A* **97**, 032331 (2018).

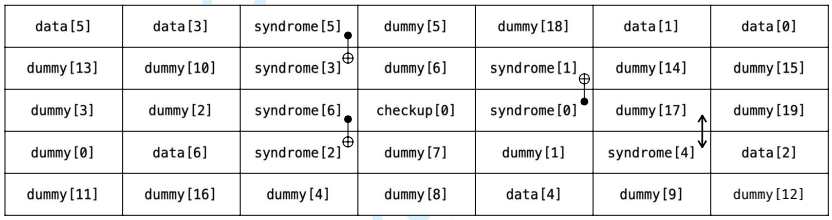
Appendix



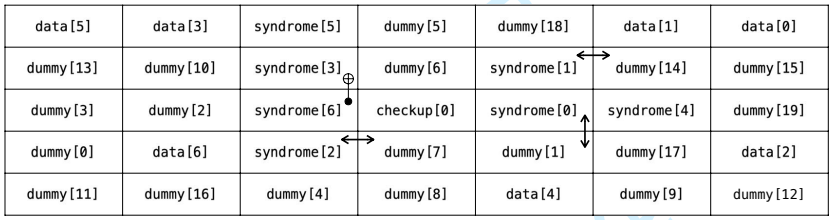
(a) Step 1



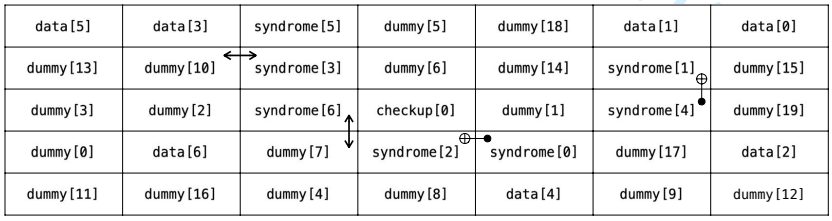
(b) Step 2



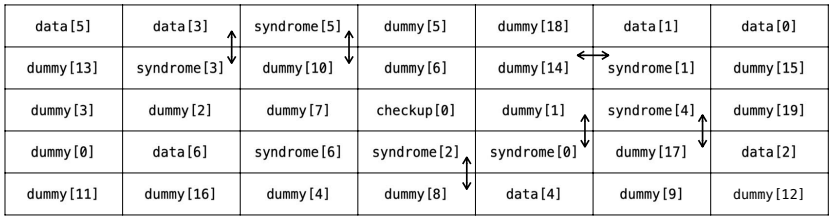
(c) Step 3



(d) Step 4



(e) Step 5



(f) Step 6

Figure A1. The first part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Fault-tolerant preparation of the logical state $|+\rangle_L$. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.

data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	dummy[6]	syndrome[1]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	dummy[7]	checkup[0]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	dummy[8]	dummy[1]	syndrome[4]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(a) Step 7

data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	syndrome[1]	dummy[6]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	dummy[7]	checkup[0]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	dummy[8]	syndrome[4]	dummy[1]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(b) Step 8

data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	syndrome[1]	dummy[6]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	checkup[0]	dummy[7]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	syndrome[4]	dummy[8]	dummy[1]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(c) Step 9

data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	syndrome[1]	dummy[6]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	checkup[0]	dummy[7]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	syndrome[4]	dummy[8]	dummy[1]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(d) Step 10

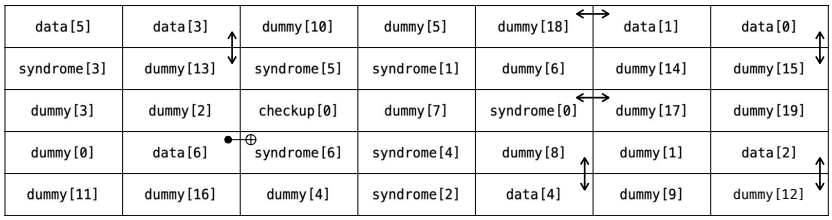
data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	syndrome[1]	dummy[6]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	checkup[0]	dummy[7]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	syndrome[4]	dummy[8]	dummy[1]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(e) Step 11

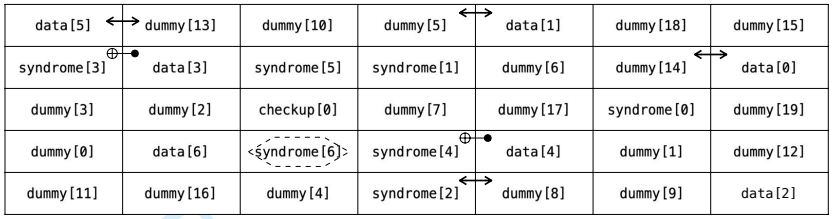
data[5]	data[3]	dummy[10]	dummy[5]	dummy[18]	data[1]	data[0]
syndrome[3]	dummy[13]	syndrome[5]	syndrome[1]	dummy[6]	dummy[14]	dummy[15]
dummy[3]	dummy[2]	checkup[0]	dummy[7]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	data[6]	syndrome[6]	syndrome[4]	dummy[8]	dummy[1]	data[2]
dummy[11]	dummy[16]	dummy[4]	syndrome[2]	data[4]	dummy[9]	dummy[12]

(f) Step 12 (`Barrier`)

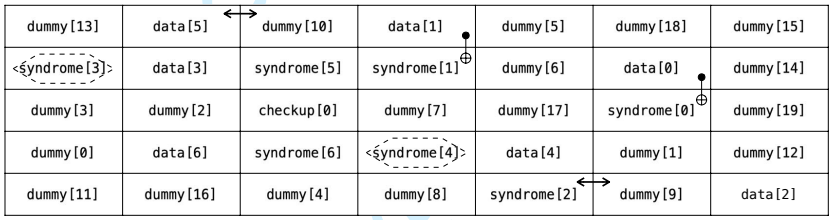
Figure A2. (Continued from Figure A1) The first part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Fault-tolerant preparation of the logical state $|+\rangle_L$. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.



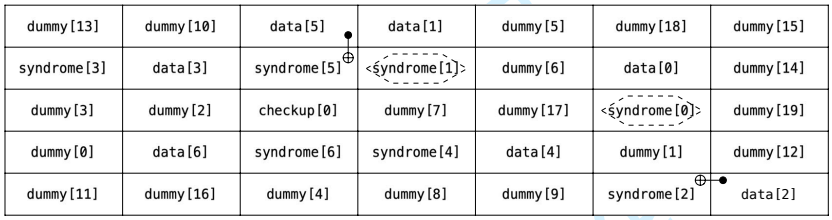
(a) Step 13



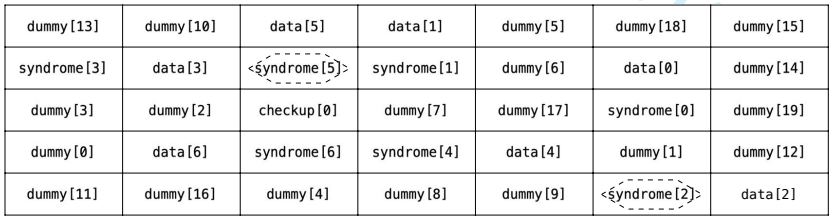
(b) Step 14



(c) Step 15

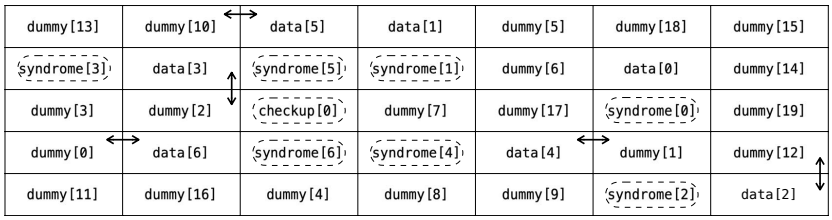


(d) Step 16

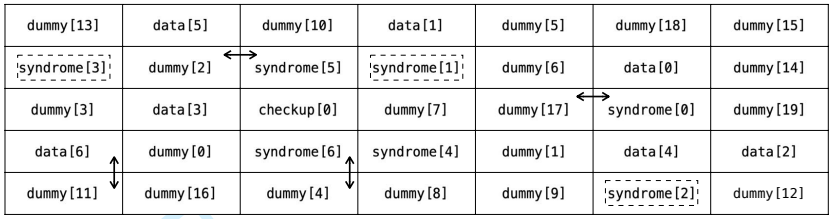


(e) Step 17 (`Barrier`)

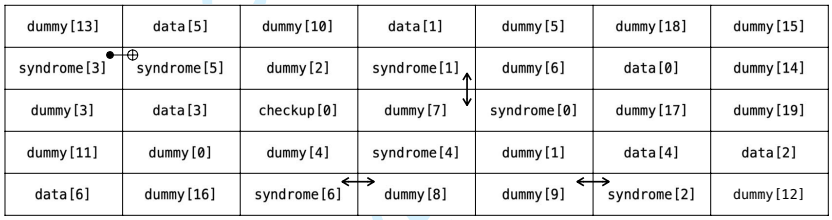
Figure A3. The second part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Transversal `CNOT` between data qubits and syndrome qubits. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.



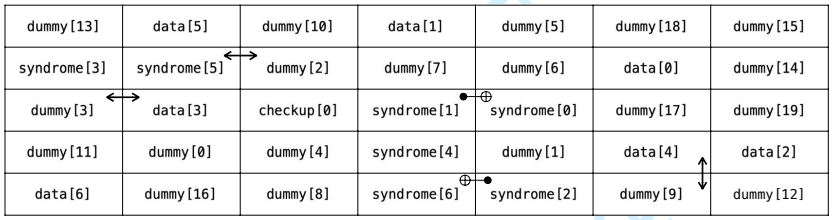
(a) Step 18



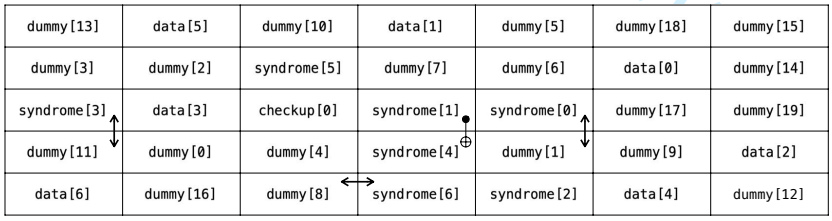
(b) Step 19



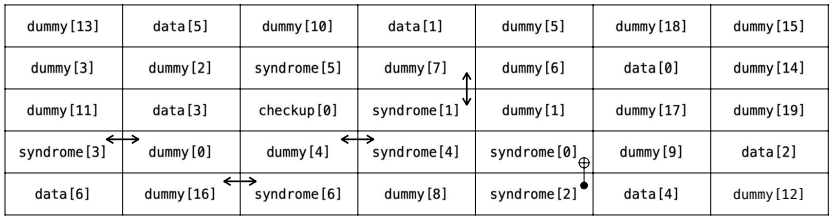
(c) Step 20



(d) Step 21

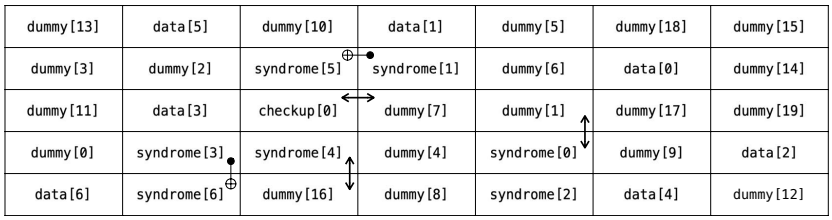


(e) Step 22

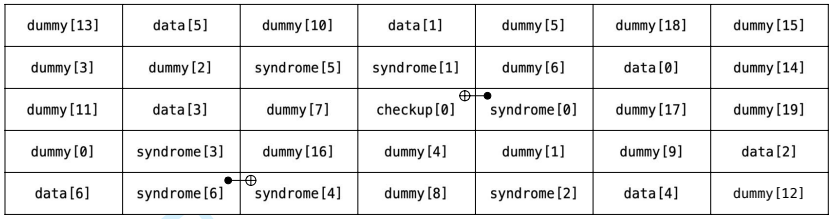


(f) Step 23

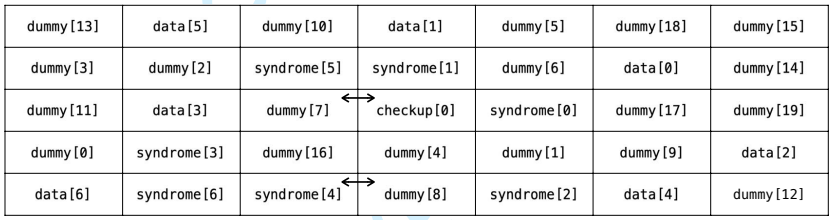
Figure A4. The third part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7,1,3]]$ Steane code: Fault-tolerant preparation of the logical state $|0\rangle_L$. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.



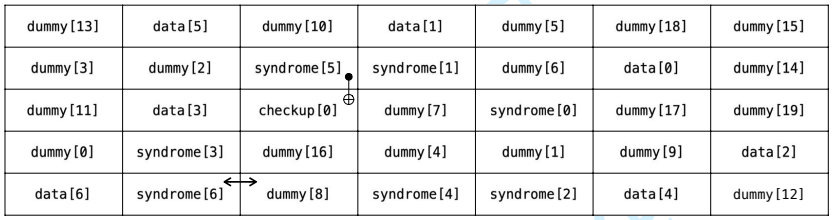
(a) Step 24



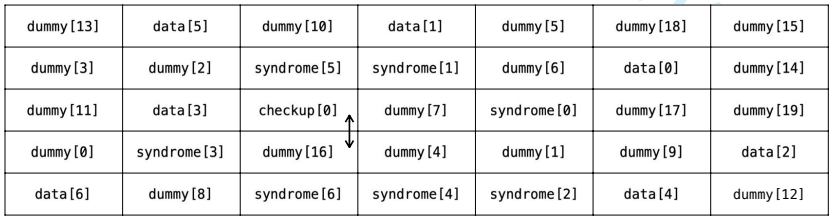
(b) Step 25



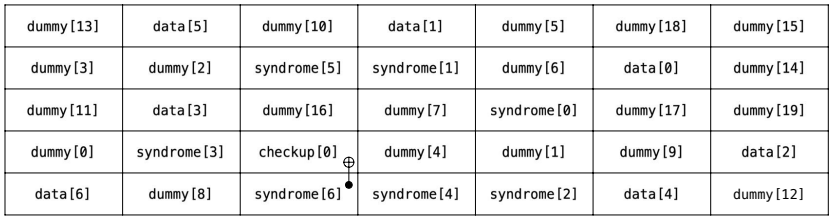
(c) Step 26



(d) Step 27



(e) Step 28



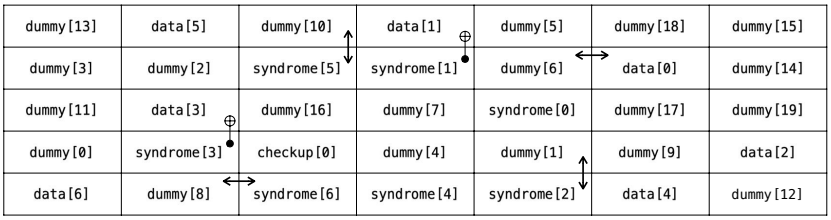
(f) Step 29

Figure A5. (Continued from Figure A4) The third part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Fault-tolerant preparation of the logical state $|0\rangle_L$. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.

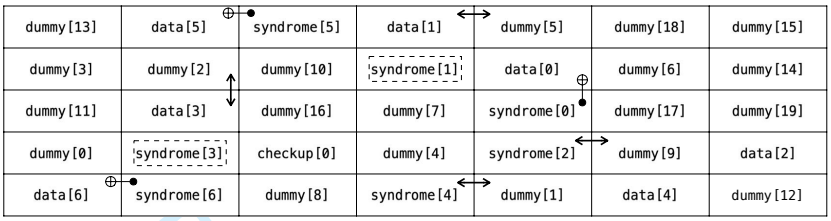
dummy[13]	data[5]	dummy[10]	data[1]	dummy[5]	dummy[18]	dummy[15]
dummy[3]	dummy[2]	syndrome[5]	syndrome[1]	dummy[6]	data[0]	dummy[14]
dummy[11]	data[3]	dummy[16]	dummy[7]	syndrome[0]	dummy[17]	dummy[19]
dummy[0]	syndrome[3]	checkup[0]	dummy[4]	dummy[1]	dummy[9]	data[2]
data[6]	dummy[8]	syndrome[6]	syndrome[4]	syndrome[2]	data[4]	dummy[12]

(a) Step 30 (`Barrier`)

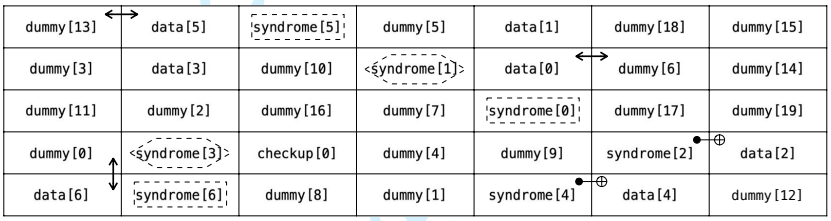
Figure A6. (Continued from Figure A5) The third part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Fault-tolerant preparation of the logical state $|0\rangle_L$. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow respectively indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.



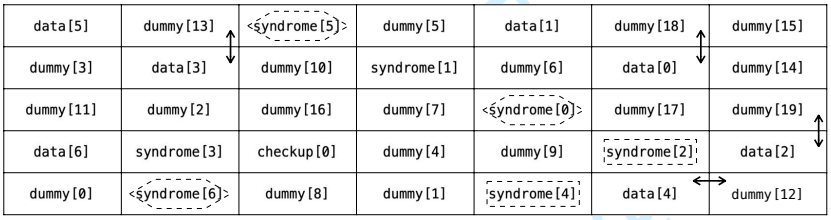
(a) Step 31



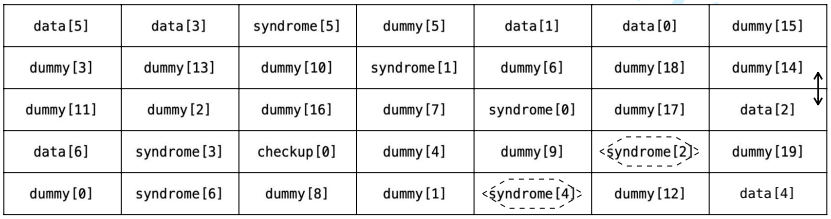
(b) Step 32



(c) Step 33



(d) Step 34



(e) Step 35

data[5]	**data[3]**	syndrome[5]	dummy[5]	**data[1]**	**data[0]**	dummy[15]
dummy[3]	dummy[13]	dummy[10]	syndrome[1]	dummy[6]	dummy[18]	**data[2]**
dummy[11]	dummy[2]	dummy[16]	dummy[7]	syndrome[0]	dummy[17]	dummy[14]
data[6]	syndrome[3]	checkup[0]	dummy[4]	dummy[9]	syndrome[2]	dummy[19]
dummy[0]	syndrome[6]	dummy[8]	dummy[1]	syndrome[4]	dummy[12]	**data[4]**

(f) Final Qubit Mapping

Figure A7. The fourth part of the fault-tolerant quantum circuit of stabilizer measurement of $[[7, 1, 3]]$ Steane code: Transversal `CNOT` between the data qubits and the syndrome qubits. By the `MoveBack` operation, all the data qubits arrives at the initial positions. Compare (f) with Figure 17. Note that the rectangles, rounded rectangles, hexagons and bi-directed arrow indicate `H`, `PrepZ`, `MeasZ` and `SWAP` gates.

```
{
    "circuit" : {
         "0" : ["PrepZ 18", "PrepZ 11", "PrepZ 23", "PrepZ 9", "PrepZ 26", "PrepZ 2",
                "PrepZ 16", "PrepZ 17", "SWAP 20,13", "SWAP 6,5", "SWAP 22,21","SWAP 33,34"],
         "1" : ["H 18", "H 26", "H 2", "H 16", "SWAP 27,20", "SWAP 5,4", "SWAP 32,33"],
         "2" : ["CNOT 18,11", "CNOT 2,9", "CNOT 16,23", "SWAP 19,26"],
         "3" : ["CNOT 16,9", "SWAP 24,23", "SWAP 12,11", "SWAP 25,18"],
         "4" : ["CNOT 19,12", "CNOT 25,24", "SWAP 8,9", "SWAP 23,16"],
         "5" : ["SWAP 18,25", "SWAP 11,12", "SWAP 26,19", "SWAP 31,24", "SWAP 9,2", "SWAP 7,8"],
         "6" : ["CNOT 17,18", "SWAP 25,26", "SWAP 10,11"],
         "7" : ["CNOT 9,10", "SWAP 16,17", "SWAP 24,25"],
         "8" : ["CNOT 16,9", "CNOT 24,23"],
         "9" : ["CNOT 16,23"],
        "10" : ["H 16"],
        "11" : ["MeasZ 16", "Barrier"],
        "12" : ["CNOT 22,23", "SWAP 13,6", "SWAP 8,1", "SWAP 4,5", "SWAP 25,32", "SWAP 34,27", "SWAP 19,18"],
        "13" : ["MeasZ 23", "CNOT 8,7", "SWAP 12,13", "SWAP 1,0", "CNOT 25,24", "SWAP 3,4", "SWAP 32,31"],
        "14" : ["MeasZ 7", "SWAP 2,1", "MeasZ 24", "CNOT 3,10", "CNOT 12,19", "SWAP 33,32"],
        "15" : ["CNOT 2,9", "MeasZ 10", "MeasZ 19", "CNOT 34,33"],
        "16" : ["MeasZ 9", "MeasZ 33", "Barrier"],
        "17" : ["PrepZ 19", "PrepZ 10", "PrepZ 33", "PrepZ 7", "PrepZ 24", "PrepZ 9", "PrepZ 23", "PrepZ 16",
                "SWAP 15,8", "SWAP 27,34", "SWAP 1,2", "SWAP 26,25", "SWAP 21,22"],
        "18" : ["H 10", "H 33", "H 7", "SWAP 18,19", "SWAP 8,9", "SWAP 30,23", "SWAP 28,21"],
        "19" : ["SWAP 17,10", "CNOT 7,8", "SWAP 32,33", "SWAP 31,30"],
        "20" : ["CNOT 17,18", "SWAP 9,8", "CNOT 32,31", "SWAP 14,7", "SWAP 33,26"],
        "21" : ["CNOT 17,24", "SWAP 25,18", "SWAP 30,31", "SWAP 21,14"],
        "22" : ["SWAP 10,17", "CNOT 32,25", "SWAP 23,24", "SWAP 29,30", "SWAP 22,21"],
        "23" : ["CNOT 10,9", "SWAP 18,25", "SWAP 17,16", "CNOT 22,29", "SWAP 30,23"],
        "24" : ["CNOT 18,17", "CNOT 29,30"],
        "25" : ["SWAP 16,17", "SWAP 31,30"],
        "26" : ["CNOT 9,16", "SWAP 30,29"],
        "27" : ["SWAP 23,16"],
        "28" : ["CNOT 30,23"],
        "29" : ["MeasZ 23", "Barrier"],
        "30" : ["CNOT 10,3", "CNOT 22,15", "SWAP 11,12", "SWAP 2,9", "SWAP 25,32", "SWAP 29,30",
                "SWAP 20,13"],
        "31" : ["H 10", "H 22", "CNOT 18,11", "CNOT 2,1", "CNOT 29,28", "SWAP 26,25",
                "SWAP 32,31", "SWAP 4,3", "SWAP 6,13", "SWAP 8,15"],
        "32" : ["MeasZ 10", "MeasZ 22", "H 18", "H 2", "H 29", "CNOT 26,27", "CNOT 32,33",
                "SWAP 12,11", "SWAP 0,1", "SWAP 21,28"],
        "33" : ["MeasZ 18", "MeasZ 2", "H 26", "H 32", "MeasZ 29",
                "SWAP 5,12", "SWAP 34,33", "SWAP 20,27", "SWAP 1,8"],
        "34" : ["MeasZ 26", "MeasZ 32", "SWAP 13,20"],
    },
    "final_mapping" : {"data0" : 5, "data1" : 4, "data2" : 13, "data3" : 1, "data4" : 34,
                       "data5" : 0, "data6" : 21},
    "initial_mapping" : {"data0" : 5, "data1" : 4, "data2" : 13, "data3" : 1, "data4" : 34,
                         "data5" : 0, "data6" : 21,
                         "syndrome0" : 18, "syndrome1" : 11, "syndrome2" : 23, "syndrome3" : 9,
                         "syndrome4" : 26, "syndrome5" : 2, "syndrome6" : 16}
}
```


Figure A8. The JSON format representation of the fault-tolerant quantum circuit shown in Figures A1 $\sim$ A7.

LQ1-data[5]	**LQ1-data[3]**	LQ1-syndrome[5]	LQ1-dummy[5]	**LQ1-data[1]**	**LQ1-data[0]**	LQ1-dummy[18]
LQ1-dummy[13]	LQ1-dummy[10]	LQ1-syndrome[3]	LQ1-dummy[6]	LQ1-syndrome[1]	LQ1-dummy[14]	**LQ1-data[2]**
LQ1-dummy[3]	LQ1-dummy[2]	LQ1-syndrome[6]	LQ1-checkup[0]	LQ1-syndrome[0]	LQ1-dummy[17]	LQ1-dummy[15]
LQ1-data[6]	LQ1-dummy[0]	LQ1-syndrome[2]	LQ1-dummy[7]	LQ1-dummy[1]	LQ1-syndrome[4]	LQ1-dummy[19]
LQ1-dummy[11]	LQ1-dummy[16]	LQ1-dummy[4]	LQ1-dummy[8]	LQ1-dummy[9]	LQ1-dummy[12]	**LQ1-data[4]**
LQ2-data[5]	**LQ2-data[3]**	LQ2-syndrome[5]	LQ2-dummy[5]	**LQ2-data[1]**	**LQ2-data[0]**	LQ2-dummy[18]
LQ2-dummy[13]	LQ2-dummy[10]	LQ2-syndrome[3]	LQ2-dummy[6]	LQ2-syndrome[1]	LQ2-dummy[14]	**LQ2-data[2]**
LQ2-dummy[3]	LQ2-dummy[2]	LQ2-syndrome[6]	LQ2-checkup[0]	LQ2-syndrome[0]	LQ2-dummy[17]	LQ2-dummy[15]
LQ2-data[6]	LQ2-dummy[0]	LQ2-syndrome[2]	LQ2-dummy[7]	LQ2-dummy[1]	LQ2-syndrome[4]	LQ2-dummy[19]
LQ2-dummy[11]	LQ2-dummy[16]	LQ2-dummy[4]	LQ2-dummy[8]	LQ2-dummy[9]	LQ2-dummy[12]	**LQ2-data[4]**

(a) Initial Mapping based on Vertically Extended Layout

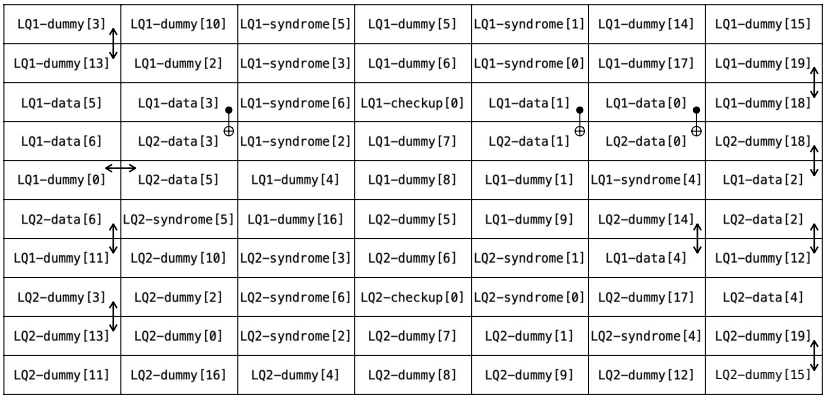
LQ1-dummy[13] ↕	LQ1-dummy[10] ↕	LQ1-syndrome[5]	LQ1-dummy[5]	LQ1-syndrome[1] ↕	LQ1-dummy[14] ↕	LQ1-dummy[18]
LQ1-data[5]	LQ1-data[3]	LQ1-syndrome[3]	LQ1-dummy[6]	LQ1-data[1]	LQ1-data[0]	LQ1-dummy[15] ↕
LQ1-dummy[3]	LQ1-dummy[2]	LQ1-syndrome[6]	LQ1-checkup[0]	LQ1-syndrome[0]	LQ1-dummy[17]	LQ1-data[2]
LQ1-data[6]	LQ1-dummy[0]	LQ1-syndrome[2]	LQ1-dummy[7]	LQ1-dummy[1]	LQ1-syndrome[4]	LQ1-dummy[19]
LQ2-data[5] ↕	LQ2-data[3] ↕	LQ1-dummy[4]	LQ1-dummy[8]	LQ2-data[1] ↕	LQ2-data[0] ↕	LQ2-dummy[18] ↕
LQ1-dummy[11]	LQ1-dummy[16]	LQ2-syndrome[5]	LQ2-dummy[5]	LQ1-dummy[9]	LQ1-dummy[12]	LQ1-data[4]
LQ2-dummy[13]	LQ2-dummy[10]	LQ2-syndrome[3]	LQ2-dummy[6]	LQ2-syndrome[1]	LQ2-dummy[14]	LQ2-data[2]
LQ2-data[6] ↕	LQ2-dummy[2]	LQ2-syndrome[6]	LQ2-checkup[0]	LQ2-syndrome[0]	LQ2-dummy[17]	LQ2-dummy[15]
LQ2-dummy[3]	LQ2-dummy[0]	LQ2-syndrome[2]	LQ2-dummy[7]	LQ2-dummy[1]	LQ2-syndrome[4]	LQ2-data[4] ↕
LQ2-dummy[11]	LQ2-dummy[16]	LQ2-dummy[4]	LQ2-dummy[8]	LQ2-dummy[9]	LQ2-dummy[12]	LQ2-dummy[19]

(b) Step 1

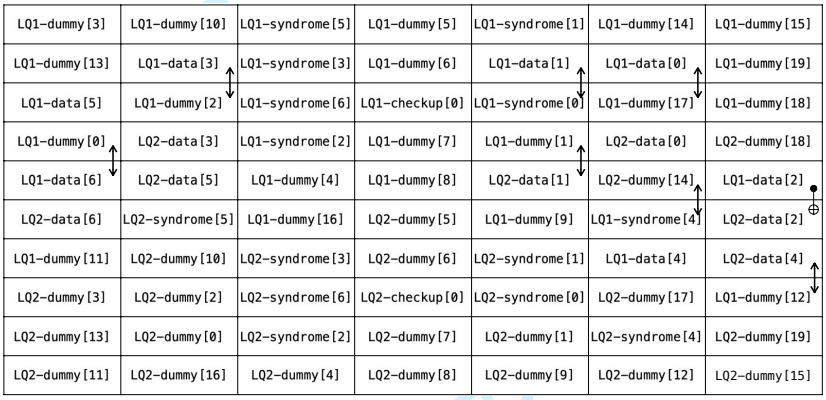
LQ1-dummy[13]	LQ1-dummy[10]	LQ1-syndrome[5]	LQ1-dummy[5]	LQ1-syndrome[1]	LQ1-dummy[14]	LQ1-dummy[15] ↕
LQ1-dummy[3] ↕	LQ1-dummy[2] ↕	LQ1-syndrome[3]	LQ1-dummy[6]	LQ1-syndrome[0] ↕	LQ1-dummy[17] ↕	LQ1-dummy[18]
LQ1-data[5]	LQ1-data[3]	LQ1-syndrome[6]	LQ1-checkup[0]	LQ1-data[1]	LQ1-data[0]	LQ1-dummy[19] ↕
LQ1-data[6]	LQ2-data[3] ↕	LQ1-syndrome[2]	LQ1-dummy[7]	LQ2-data[1] ↕	LQ2-data[0] ↕	LQ1-data[2]
LQ2-data[5]	LQ1-dummy[0]	LQ1-dummy[4]	LQ1-dummy[8]	LQ1-dummy[1]	LQ1-syndrome[4]	LQ2-dummy[18]
LQ1-dummy[11] ↔	LQ2-syndrome[5] ↔	LQ1-dummy[16]	LQ2-dummy[5]	LQ1-dummy[9]	LQ1-data[4] ↔	LQ1-dummy[12]
LQ2-data[6] ↕	LQ2-dummy[10]	LQ2-syndrome[3]	LQ2-dummy[6]	LQ2-syndrome[1]	LQ2-dummy[14]	LQ2-data[2]
LQ2-dummy[13]	LQ2-dummy[2]	LQ2-syndrome[6]	LQ2-checkup[0]	LQ2-syndrome[0]	LQ2-dummy[17]	LQ2-data[4] ↕
LQ2-dummy[3]	LQ2-dummy[0]	LQ2-syndrome[2]	LQ2-dummy[7]	LQ2-dummy[1]	LQ2-syndrome[4]	LQ2-dummy[15]
LQ2-dummy[11]	LQ2-dummy[16]	LQ2-dummy[4]	LQ2-dummy[8]	LQ2-dummy[9]	LQ2-dummy[12]	LQ2-dummy[19]

(c) Step 2

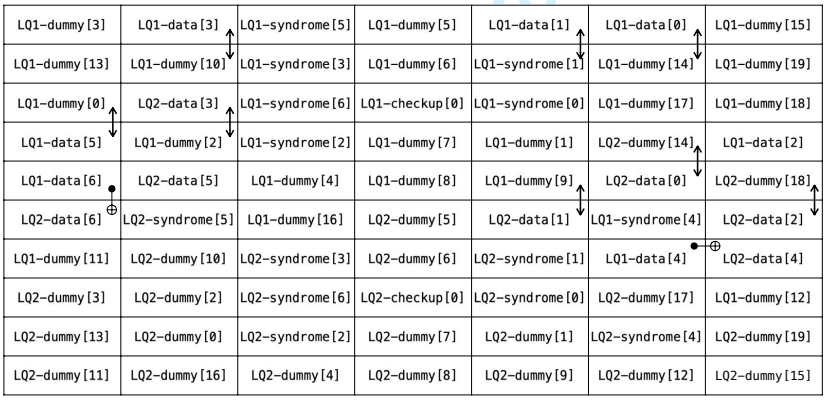
Figure A9. Logical `CNOT` gate operations for qubits arranged vertically. (a) The vertically extended qubit layout of logical qubits, LQ1 and LQ2. Note that the bi-directed arrow indicates `SWAP` gate.



(a) Step 3



(b) Step 4



(c) Step 5

Figure A10. (Continued from Figure A9) Logical `CNOT` gate operations for qubits arranged vertically. Note that the bi-directed arrow indicates `SWAP` gate.

LQ1–dummy[3]	LQ1–data[3]	LQ1–syndrome[5]	LQ1–dummy[5]	LQ1–data[1]	LQ1–data[0]	LQ1–dummy[15]
LQ1–dummy[13]	LQ1–dummy[10]	LQ1–syndrome[3]	LQ1–dummy[6]	LQ1–syndrome[1]	LQ1–dummy[14]	LQ1–dummy[19]
LQ1–dummy[0]	LQ2–data[3]	LQ1–syndrome[6]	LQ1–checkup[0]	LQ1–syndrome[0]	LQ1–dummy[17]	LQ1–data[2]
LQ1–data[5]	LQ2–data[5]	LQ1–syndrome[2]	LQ1–dummy[7]	LQ1–dummy[1]	LQ2–dummy[14]	LQ1–dummy[18]
LQ1–data[6]	LQ1–dummy[2]	LQ1–dummy[4]	LQ1–dummy[8]	LQ1–dummy[9]	LQ1–syndrome[4]	LQ2–dummy[18]
LQ1–dummy[11]	LQ2–syndrome[5]	LQ1–dummy[16]	LQ2–dummy[5]	LQ2–data[1]	LQ2–data[0]	LQ2–data[2]
LQ2–data[6]	LQ2–dummy[10]	LQ2–syndrome[3]	LQ2–dummy[6]	LQ2–syndrome[1]	LQ1–data[4]	LQ1–dummy[12]
LQ2–dummy[3]	LQ2–dummy[2]	LQ2–syndrome[6]	LQ2–checkup[0]	LQ2–syndrome[0]	LQ2–dummy[17]	LQ2–data[4]
LQ2–dummy[13]	LQ2–dummy[0]	LQ2–syndrome[2]	LQ2–dummy[7]	LQ2–dummy[1]	LQ2–syndrome[4]	LQ2–dummy[19]
LQ2–dummy[11]	LQ2–dummy[16]	LQ2–dummy[4]	LQ2–dummy[8]	LQ2–dummy[9]	LQ2–dummy[12]	LQ2–dummy[15]

(a) Step 6

LQ1–dummy[3]	LQ1–data[3]	LQ1–syndrome[5]	LQ1–dummy[5]	LQ1–data[1]	LQ1–data[0]	LQ1–dummy[15]
LQ1–dummy[13]	LQ1–dummy[10]	LQ1–syndrome[3]	LQ1–dummy[6]	LQ1–syndrome[1]	LQ1–dummy[14]	LQ1–data[2]
LQ1–dummy[0]	LQ2–data[3]	LQ1–syndrome[6]	LQ1–checkup[0]	LQ1–syndrome[0]	LQ1–dummy[17]	LQ1–dummy[19]
LQ1–data[5]	LQ2–data[5]	LQ1–syndrome[2]	LQ1–dummy[7]	LQ1–dummy[1]	LQ2–dummy[14]	LQ1–dummy[18]
LQ1–data[6]	LQ2–syndrome[5]	LQ1–dummy[4]	LQ1–dummy[8]	LQ1–dummy[9]	LQ1–syndrome[4]	LQ2–dummy[18]
LQ1–dummy[11]	LQ1–dummy[2]	LQ1–dummy[16]	LQ2–dummy[5]	LQ2–data[1]	LQ2–data[0]	LQ1–dummy[12]
LQ2–dummy[3]	LQ2–dummy[10]	LQ2–syndrome[3]	LQ2–dummy[6]	LQ2–syndrome[1]	LQ1–data[4]	LQ2–data[2]
LQ2–data[6]	LQ2–dummy[2]	LQ2–syndrome[6]	LQ2–checkup[0]	LQ2–syndrome[0]	LQ2–dummy[17]	LQ2–dummy[19]
LQ2–dummy[13]	LQ2–dummy[0]	LQ2–syndrome[2]	LQ2–dummy[7]	LQ2–dummy[1]	LQ2–syndrome[4]	LQ2–data[4]
LQ2–dummy[11]	LQ2–dummy[16]	LQ2–dummy[4]	LQ2–dummy[8]	LQ2–dummy[9]	LQ2–dummy[12]	LQ2–dummy[15]

(b) Step 7

LQ1–dummy[3]	LQ1–data[3]	LQ1–syndrome[5]	LQ1–dummy[5]	LQ1–data[1]	LQ1–data[0]	LQ1–dummy[15]
LQ1–dummy[13]	LQ1–dummy[10]	LQ1–syndrome[3]	LQ1–dummy[6]	LQ1–syndrome[1]	LQ1–dummy[14]	LQ1–data[2]
LQ1–data[5]	LQ2–data[3]	LQ1–syndrome[6]	LQ1–checkup[0]	LQ1–syndrome[0]	LQ1–dummy[17]	LQ1–dummy[19]
LQ1–dummy[0]	LQ2–syndrome[5]	LQ1–syndrome[2]	LQ1–dummy[7]	LQ1–dummy[1]	LQ2–dummy[14]	LQ1–dummy[18]
LQ1–data[6]	LQ2–data[5]	LQ1–dummy[4]	LQ1–dummy[8]	LQ1–dummy[9]	LQ1–syndrome[4]	LQ1–dummy[12]
LQ1–dummy[11]	LQ2–dummy[10]	LQ1–dummy[16]	LQ2–dummy[5]	LQ2–data[1]	LQ2–data[0]	LQ2–dummy[18]
LQ2–dummy[3]	LQ1–dummy[2]	LQ2–syndrome[3]	LQ2–dummy[6]	LQ2–syndrome[1]	LQ1–data[4]	LQ2–data[2]
LQ2–dummy[13]	LQ2–dummy[2]	LQ2–syndrome[6]	LQ2–checkup[0]	LQ2–syndrome[0]	LQ2–dummy[17]	LQ2–dummy[19]
LQ2–data[6]	LQ2–dummy[0]	LQ2–syndrome[2]	LQ2–dummy[7]	LQ2–dummy[1]	LQ2–syndrome[4]	LQ2–dummy[15]
LQ2–dummy[11]	LQ2–dummy[16]	LQ2–dummy[4]	LQ2–dummy[8]	LQ2–dummy[9]	LQ2–dummy[12]	LQ2–data[4]

(c) Step 8

Figure A11. (Continued from Figure A10) Logical `CNOT` gate operations for qubits arranged vertically. Note that the bi-directed arrow indicates `SWAP` gate.

LQ1−dummy[3]	LQ1−data[3]	LQ1−syndrome[5]	LQ1−dummy[5]	LQ1−data[1]	LQ1−data[0]	LQ1−dummy[15]
LQ1−data[5]	LQ1−dummy[10]	LQ1−syndrome[3]	LQ1−dummy[6]	LQ1−syndrome[1]	LQ1−dummy[14]	LQ1−data[2]
LQ1−dummy[13]	LQ2−syndrome[5]	LQ1−syndrome[6]	LQ1−checkup[0]	LQ1−syndrome[0]	LQ1−dummy[17]	LQ1−dummy[19]
LQ1−data[6]	LQ2−data[3]	LQ1−syndrome[2]	LQ1−dummy[7]	LQ1−dummy[1]	LQ2−dummy[14]	LQ1−dummy[18]
LQ1−dummy[0]	LQ2−data[5]	LQ1−dummy[4]	LQ1−dummy[8]	LQ1−dummy[9]	LQ1−dummy[12]	LQ1−syndrome[4]
LQ2−dummy[10]	LQ1−dummy[11]	LQ1−dummy[16]	LQ2−dummy[5]	LQ2−data[1]	LQ2−data[0]	LQ2−dummy[18]
LQ2−dummy[3]	LQ1−dummy[2]	LQ2−syndrome[3]	LQ2−dummy[6]	LQ2−syndrome[1]	LQ1−data[4]	LQ2−data[2]
LQ2−dummy[13]	LQ2−dummy[2]	LQ2−syndrome[6]	LQ2−checkup[0]	LQ2−syndrome[0]	LQ2−dummy[17]	LQ2−dummy[19]
LQ2−data[6]	LQ2−dummy[0]	LQ2−syndrome[2]	LQ2−dummy[7]	LQ2−dummy[1]	LQ2−syndrome[4]	LQ2−dummy[15]
LQ2−dummy[11]	LQ2−dummy[16]	LQ2−dummy[4]	LQ2−dummy[8]	LQ2−dummy[9]	LQ2−dummy[12]	LQ2−data[4]

(a) Step 9

LQ1−data[5]	LQ1−data[3]	LQ1−syndrome[5]	LQ1−dummy[5]	LQ1−data[1]	LQ1−data[0]	LQ1−dummy[15]
LQ1−dummy[3]	LQ1−dummy[10]	LQ1−syndrome[3]	LQ1−dummy[6]	LQ1−syndrome[1]	LQ1−dummy[14]	LQ1−data[2]
LQ1−dummy[13]	LQ2−syndrome[5]	LQ1−syndrome[6]	LQ1−checkup[0]	LQ1−syndrome[0]	LQ1−dummy[17]	LQ1−dummy[19]
LQ1−data[6]	LQ2−data[3]	LQ1−syndrome[2]	LQ1−dummy[7]	LQ1−dummy[1]	LQ2−dummy[14]	LQ1−dummy[18]
LQ1−dummy[0]	LQ1−dummy[11]	LQ1−dummy[4]	LQ1−dummy[8]	LQ1−dummy[9]	LQ2−data[0]	LQ1−syndrome[4]
LQ2−dummy[10]	LQ2−data[5]	LQ1−dummy[16]	LQ2−dummy[5]	LQ2−data[1]	LQ1−dummy[12]	LQ2−dummy[18]
LQ2−dummy[3]	LQ1−dummy[2]	LQ2−syndrome[3]	LQ2−dummy[6]	LQ2−syndrome[1]	LQ1−data[4]	LQ2−data[2]
LQ2−dummy[13]	LQ2−dummy[2]	LQ2−syndrome[6]	LQ2−checkup[0]	LQ2−syndrome[0]	LQ2−dummy[17]	LQ2−dummy[19]
LQ2−data[6]	LQ2−dummy[0]	LQ2−syndrome[2]	LQ2−dummy[7]	LQ2−dummy[1]	LQ2−syndrome[4]	LQ2−dummy[15]
LQ2−dummy[11]	LQ2−dummy[16]	LQ2−dummy[4]	LQ2−dummy[8]	LQ2−dummy[9]	LQ2−dummy[12]	LQ2−data[4]

(b) Step 10

LQ1−data[5]	LQ1−data[3]	LQ1−syndrome[5]	LQ1−dummy[5]	LQ1−data[1]	LQ1−data[0]	LQ1−dummy[15]
LQ1−dummy[3]	LQ1−dummy[10]	LQ1−syndrome[3]	LQ1−dummy[6]	LQ1−syndrome[1]	LQ1−dummy[14]	LQ1−data[2]
LQ1−dummy[13]	LQ2−syndrome[5]	LQ1−syndrome[6]	LQ1−checkup[0]	LQ1−syndrome[0]	LQ1−dummy[17]	LQ1−dummy[19]
LQ1−data[6]	LQ1−dummy[11]	LQ1−syndrome[2]	LQ1−dummy[7]	LQ1−dummy[1]	LQ2−dummy[14]	LQ1−dummy[18]
LQ1−dummy[0]	LQ2−data[3]	LQ1−dummy[4]	LQ1−dummy[8]	LQ1−dummy[9]	LQ2−data[0]	LQ1−syndrome[4]
LQ2−data[5]	LQ2−dummy[10]	LQ1−dummy[16]	LQ2−dummy[5]	LQ2−data[1]	LQ1−data[4]	LQ2−dummy[18]
LQ2−dummy[3]	LQ1−dummy[2]	LQ2−syndrome[3]	LQ2−dummy[6]	LQ2−syndrome[1]	LQ1−dummy[12]	LQ2−data[2]
LQ2−dummy[13]	LQ2−dummy[2]	LQ2−syndrome[6]	LQ2−checkup[0]	LQ2−syndrome[0]	LQ2−dummy[17]	LQ2−dummy[19]
LQ2−data[6]	LQ2−dummy[0]	LQ2−syndrome[2]	LQ2−dummy[7]	LQ2−dummy[1]	LQ2−syndrome[4]	LQ2−dummy[15]
LQ2−dummy[11]	LQ2−dummy[16]	LQ2−dummy[4]	LQ2−dummy[8]	LQ2−dummy[9]	LQ2−dummy[12]	LQ2−data[4]

(c) Step 11

Figure A12. (Continued from Figure A11) Logical `CNOT` gate operations for qubits arranged vertically. By the `MoveBack` operations, the data qubits after all the quantum operations are placed in their initial positions (compare (b) and Figure A9 (a)). Note that the bi-directed arrow indicates `SWAP` gate.

LQ1-data[5]	LQ1-data[3]	LQ1-syndrome[5]	LQ1-dummy[5]	LQ1-data[1]	LQ1-data[0]	LQ1-dummy[15]
LQ1-dummy[3]	LQ1-dummy[10]	LQ1-syndrome[3]	LQ1-dummy[6]	LQ1-syndrome[1]	LQ1-dummy[14]	LQ1-data[2]
LQ1-dummy[13]	LQ2-syndrome[5]	LQ1-syndrome[6]	LQ1-checkup[0]	LQ1-syndrome[0]	LQ1-dummy[17]	LQ1-dummy[19]
LQ1-data[6]	LQ1-dummy[11]	LQ1-syndrome[2]	LQ1-dummy[7]	LQ1-dummy[1]	LQ2-dummy[14]	LQ1-dummy[18]
LQ1-dummy[0]	LQ2-dummy[10]	LQ1-dummy[4]	LQ1-dummy[8]	LQ1-dummy[9]	LQ2-data[0]	LQ1-syndrome[4]
LQ2-data[5]	LQ2-data[3]	LQ1-dummy[16]	LQ2-dummy[5]	LQ2-data[1]	LQ2-dummy[18]	LQ1-data[4]
LQ2-dummy[3]	LQ1-dummy[2]	LQ2-syndrome[3]	LQ2-dummy[6]	LQ2-syndrome[1]	LQ1-dummy[12]	LQ2-data[2]
LQ2-dummy[13]	LQ2-dummy[2]	LQ2-syndrome[6]	LQ2-checkup[0]	LQ2-syndrome[0]	LQ2-dummy[17]	LQ2-dummy[19]
LQ2-data[6]	LQ2-dummy[0]	LQ2-syndrome[2]	LQ2-dummy[7]	LQ2-dummy[1]	LQ2-syndrome[4]	LQ2-dummy[15]
LQ2-dummy[11]	LQ2-dummy[16]	LQ2-dummy[4]	LQ2-dummy[8]	LQ2-dummy[9]	LQ2-dummy[12]	LQ2-data[4]

(a) Step 12

LQ1-data[5]	LQ1-data[3]	LQ1-syndrome[5]	LQ1-dummy[5]	LQ1-data[1]	LQ1-data[0]	LQ1-dummy[15]
LQ1-dummy[3]	LQ1-dummy[10]	LQ1-syndrome[3]	LQ1-dummy[6]	LQ1-syndrome[1]	LQ1-dummy[14]	LQ1-data[2]
LQ1-dummy[13]	LQ2-syndrome[5]	LQ1-syndrome[6]	LQ1-checkup[0]	LQ1-syndrome[0]	LQ1-dummy[17]	LQ1-dummy[19]
LQ1-data[6]	LQ1-dummy[11]	LQ1-syndrome[2]	LQ1-dummy[7]	LQ1-dummy[1]	LQ2-dummy[14]	LQ1-dummy[18]
LQ1-dummy[0]	LQ2-dummy[10]	LQ1-dummy[4]	LQ1-dummy[8]	LQ1-dummy[9]	LQ2-dummy[18]	LQ1-data[4]
LQ2-data[5]	LQ2-data[3]	LQ1-dummy[16]	LQ2-dummy[5]	LQ2-data[1]	LQ2-data[0]	LQ1-syndrome[4]
LQ2-dummy[3]	LQ1-dummy[2]	LQ2-syndrome[3]	LQ2-dummy[6]	LQ2-syndrome[1]	LQ1-dummy[12]	LQ2-data[2]
LQ2-dummy[13]	LQ2-dummy[2]	LQ2-syndrome[6]	LQ2-checkup[0]	LQ2-syndrome[0]	LQ2-dummy[17]	LQ2-dummy[19]
LQ2-data[6]	LQ2-dummy[0]	LQ2-syndrome[2]	LQ2-dummy[7]	LQ2-dummy[1]	LQ2-syndrome[4]	LQ2-dummy[15]
LQ2-dummy[11]	LQ2-dummy[16]	LQ2-dummy[4]	LQ2-dummy[8]	LQ2-dummy[9]	LQ2-dummy[12]	LQ2-data[4]

(b) Step 13 (Final Qubit Mapping)

Figure A13. (Continued from Figure A11) Logical `CNOT` gate operations for qubits arranged vertically. By the `MoveBack` operations, the data qubits after all the quantum operations are placed in their initial positions (compare (b) and Figure A9 (a)). Note that the bi-directed arrow indicates `SWAP` gate.

LQ1–data[5]	**LQ1–data[3]**	dummy[16]	dummy[50]	**LQ1–data[1]**	**LQ1–data[0]**	dummy[39]
dummy[22]	dummy[37]	dummy[21]	dummy[45]	dummy[11]	dummy[35]	**LQ1–data[2]**
dummy[10]	dummy[25]	dummy[43]	dummy[49]	dummy[19]	dummy[28]	dummy[32]
LQ1–data[6]	dummy[51]	dummy[33]	dummy[18]	dummy[41]	dummy[53]	dummy[26]
dummy[0]	dummy[42]	dummy[29]	dummy[15]	dummy[38]	dummy[34]	**LQ1–data[4]**
dummy[31]	**LQ2–magic[4]**	dummy[9]	dummy[14]	dummy[5]	dummy[47]	dummy[13]
LQ2–magic[3]	dummy[20]	dummy[24]	dummy[55]	**LQ2–magic[6]**	dummy[46]	**LQ2–magic[2]**
dummy[30]	**LQ2–magic[1]**	dummy[40]	dummy[17]	dummy[52]	dummy[1]	dummy[48]
dummy[3]	dummy[36]	dummy[2]	dummy[6]	dummy[8]	dummy[27]	**LQ2–magic[0]**
dummy[4]	dummy[23]	**LQ2–magic[5]**	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(a) Initial Mapping based on Vertically Extended Layout, $(data_n, magic_s)$

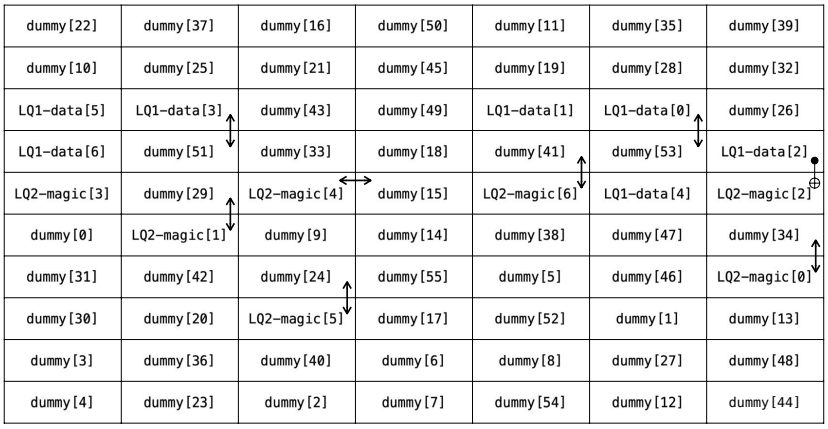
LQ1–data[5]	LQ1–data[3]	dummy[16]	dummy[50]	LQ1–data[1]	LQ1–data[0]	dummy[39]
dummy[22]	dummy[37]	dummy[21]	dummy[45]	dummy[11]	dummy[35]	LQ1–data[2]
dummy[10]	dummy[25]	dummy[43]	dummy[49]	dummy[19]	dummy[28]	dummy[32]
LQ1–data[6]	dummy[51]	dummy[33]	dummy[18]	dummy[41]	dummy[53]	dummy[26]
dummy[0]	dummy[42]	dummy[29]	dummy[15]	dummy[38]	dummy[34]	LQ1–data[4]
dummy[31]	LQ2–magic[4]	dummy[9]	dummy[14]	dummy[5]	dummy[47]	dummy[13]
LQ2–magic[3]	dummy[20]	dummy[24]	dummy[55]	LQ2–magic[6]	dummy[46]	LQ2–magic[2]
dummy[30]	LQ2–magic[1]	dummy[40]	dummy[17]	dummy[52]	dummy[1]	dummy[48]
dummy[3]	dummy[36]	dummy[2]	dummy[6]	dummy[8]	dummy[27]	LQ2–magic[0]
dummy[4]	dummy[23]	LQ2–magic[5]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(b) Step 1

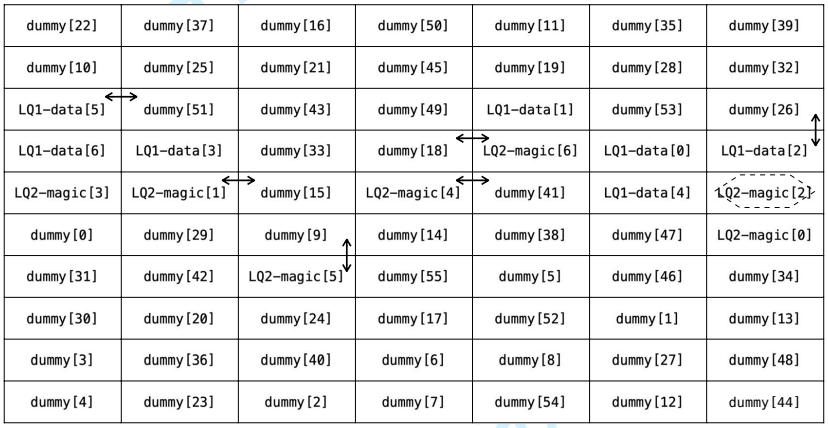
dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
LQ1–data[5]	LQ1–data[3]	dummy[21]	dummy[45]	LQ1–data[1]	LQ1–data[0]	dummy[32]
dummy[10]	dummy[25]	dummy[43]	dummy[49]	dummy[19]	dummy[28]	LQ1–data[2]
LQ1–data[6]	dummy[51]	dummy[33]	dummy[18]	dummy[41]	dummy[53]	dummy[26]
dummy[0]	LQ2–magic[4]	dummy[29]	dummy[15]	dummy[38]	LQ1–data[4]	dummy[34]
LQ2–magic[3]	dummy[42]	dummy[9]	dummy[14]	LQ2–magic[6]	dummy[47]	LQ2–magic[2]
dummy[31]	LQ2–magic[1]	dummy[24]	dummy[55]	dummy[5]	dummy[46]	dummy[13]
dummy[30]	dummy[20]	dummy[40]	dummy[17]	dummy[52]	dummy[1]	LQ2–magic[0]
dummy[3]	dummy[36]	LQ2–magic[5]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(c) Step 2

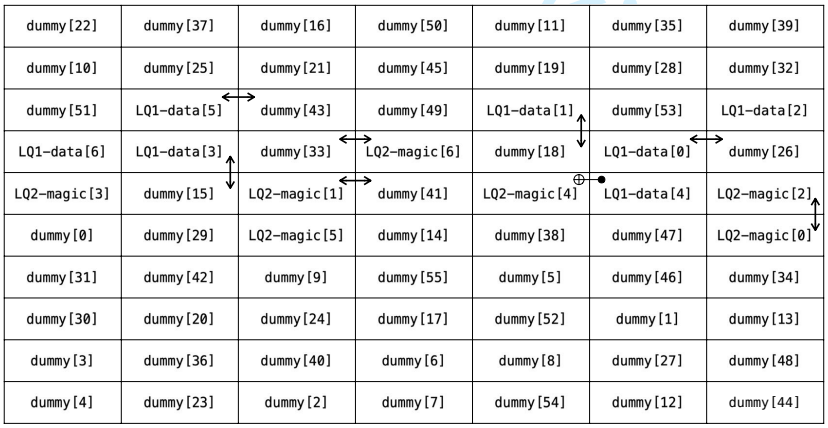
Figure A14. The first part of logical `T` gate operations for qubits arranged vertically: Transversal `CNOT` between data qubits $data[i]$ and magic qubits $magic[i]$ and `MeasZ` on magic qubits. (a) The vertically extended qubit layout of logical qubits: data qubits in the north direction and magic qubits in the south direction. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.



(a) Step 3

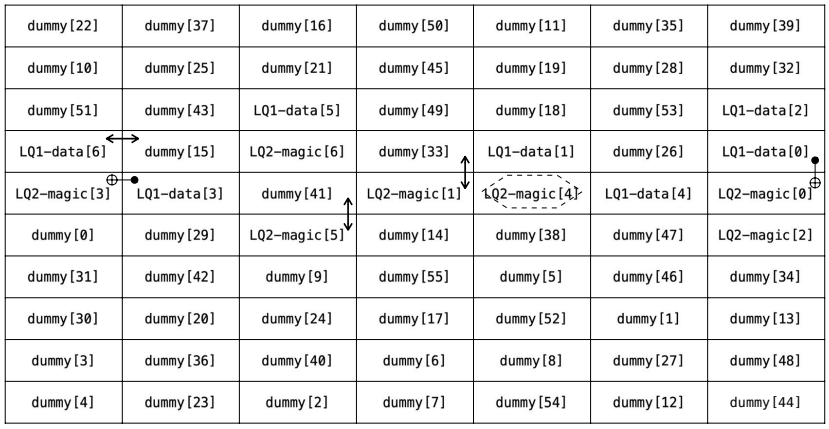


(b) Step 4

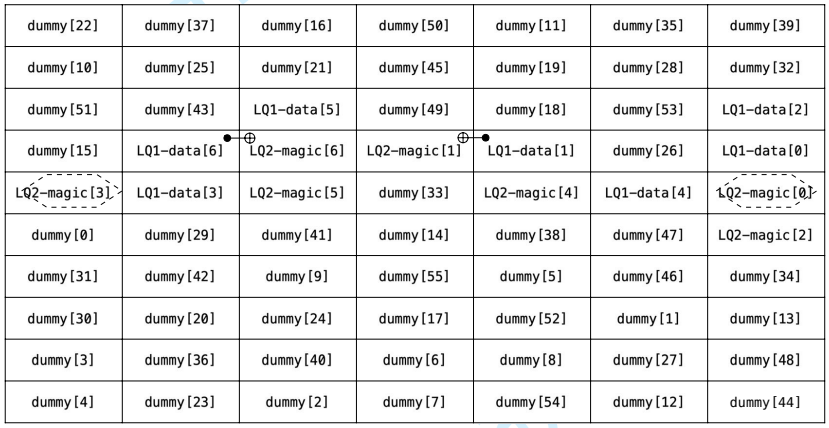


(c) Step 5

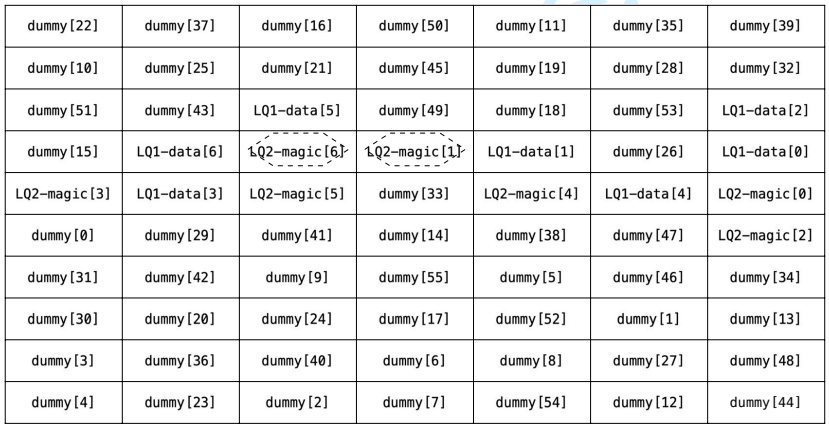
Figure A15. (Continued from Figure A14) The first part of logical `T` gate operations for qubits arranged vertically: Transversal `CNOT` between data qubits $data[i]$ and magic qubits $magic[i]$ and `MeasZ` on magic qubits. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.



(a) Step 6



(b) Step 7



(c) Step 8

Figure A16. (Continued from Figure A15) The first part of logical `T` gate operations for qubits arranged vertically: Transversal `CNOT` between data qubits $data[i]$ and magic qubits $magic[i]$ and `MeasZ` on magic qubits. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	dummy[21]	dummy[45]	dummy[19]	dummy[28]	dummy[32]
dummy[51]	dummy[43]	LQ1-data[5]	dummy[49]	dummy[18]	dummy[53]	LQ1-data[2]
dummy[15]	LQ1-data[6]	LQ2-magic[6]	LQ2-magic[1]	LQ1-data[1]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[5]	dummy[33]	LQ2-magic[4]	LQ1-data[4]	LQ2-magic[0]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(a) Step 9

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	dummy[21]	dummy[45]	dummy[19]	dummy[28]	dummy[32]
dummy[51]	dummy[43]	LQ1-data[5]	dummy[49]	dummy[18]	dummy[53]	LQ1-data[2]
dummy[15]	LQ1-data[6]	LQ2-magic[5]	LQ2-magic[1]	LQ1-data[1]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ1-data[4]	LQ2-magic[0]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(b) Step 10

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	dummy[21]	dummy[45]	dummy[19]	dummy[28]	dummy[32]
dummy[51]	dummy[43]	LQ1-data[5]	dummy[49]	dummy[18]	dummy[53]	LQ1-data[2]
dummy[15]	LQ1-data[6]	LQ2-magic[5]	LQ2-magic[1]	LQ1-data[1]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ1-data[4]	LQ2-magic[0]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(c) Step 11 (`Barrier`)

Figure A17. (Continued from Figure A16) The first part of logical `T` gate operations for qubits arranged vertically: Transversal `CNOT` between data qubits $data[i]$ and magic qubits $magic[i]$ and `MeasZ` on magic qubits. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	dummy[21]	dummy[45]	dummy[19]	dummy[28]	dummy[32]
dummy[51]	dummy[43]	LQ1-data[5]	dummy[49]	dummy[18]	dummy[53]	LQ1-data[2]
dummy[15]	LQ1-data[6]	LQ2-magic[5]	LQ2-magic[1]	LQ1-data[1]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ1-data[4]	LQ2-magic[0]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(a) Step 12 (S-correction)

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	dummy[21]	dummy[45]	dummy[19]	dummy[28]	dummy[32]
dummy[51]	dummy[43]	LQ1-data[5]	dummy[49]	dummy[18]	dummy[53]	LQ1-data[2]
dummy[15]	LQ1-data[6]	LQ2-magic[5]	LQ2-magic[1]	LQ1-data[1]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ1-data[4]	LQ2-magic[0]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(b) Step 13

dummy[22]	dummy[37]	dummy[16]	dummy[50]	dummy[11]	dummy[35]	dummy[39]
dummy[10]	dummy[25]	LQ1-data[5]	dummy[45]	dummy[19]	dummy[28]	LQ1-data[2]
dummy[51]	dummy[43]	dummy[21]	dummy[49]	LQ1-data[1]	dummy[53]	dummy[32]
LQ1-data[6]	dummy[15]	LQ2-magic[5]	LQ2-magic[1]	dummy[18]	dummy[26]	LQ1-data[0]
LQ2-magic[3]	LQ1-data[3]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ2-magic[0]	LQ1-data[4]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(c) Step 14

Figure A18. The second part logical `T` gate operations for qubits arranged vertically: Logical `S` gate correction and `MoveBack`. Please compare the final mapping with the initial mapping (Figure A14 (a)). The positions of the data qubits are the same in both mappings, but the magic qubits are not. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.

dummy[22]	dummy[37]	LQ1–data[5]	dummy[50]	dummy[11]	dummy[35]	LQ1–data[2]
dummy[10]	dummy[25]	dummy[16]	dummy[45]	LQ1–data[1]	dummy[28]	dummy[39]
dummy[51]	dummy[43]	dummy[21]	dummy[49]	dummy[19]	dummy[53]	LQ1–data[0]
LQ1–data[6]	LQ1–data[3]	LQ2–magic[5]	LQ2–magic[1]	dummy[18]	dummy[26]	dummy[32]
LQ2–magic[3]	dummy[15]	LQ2–magic[6]	dummy[33]	LQ2–magic[4]	LQ2–magic[0]	LQ1–data[4]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2–magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(a) Step 15

dummy[22]	LQ1–data[5]	dummy[37]	dummy[50]	LQ1–data[1]	LQ1–data[2]	dummy[35]
dummy[10]	dummy[25]	dummy[16]	dummy[45]	dummy[11]	dummy[28]	LQ1–data[0]
dummy[51]	LQ1–data[3]	dummy[21]	dummy[49]	dummy[19]	dummy[53]	dummy[39]
LQ1–data[6]	dummy[43]	LQ2–magic[5]	LQ2–magic[1]	dummy[18]	dummy[26]	dummy[32]
LQ2–magic[3]	dummy[15]	LQ2–magic[6]	dummy[33]	LQ2–magic[4]	LQ2–magic[0]	LQ1–data[4]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2–magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(b) Step 16

LQ1–data[5]	dummy[22]	dummy[37]	dummy[50]	LQ1–data[1]	dummy[28]	LQ1–data[0]
dummy[10]	LQ1–data[3]	dummy[16]	dummy[45]	dummy[11]	LQ1–data[2]	dummy[35]
dummy[51]	dummy[25]	dummy[21]	dummy[49]	dummy[19]	dummy[53]	dummy[39]
LQ1–data[6]	dummy[43]	LQ2–magic[5]	LQ2–magic[1]	dummy[18]	dummy[26]	dummy[32]
LQ2–magic[3]	dummy[15]	LQ2–magic[6]	dummy[33]	LQ2–magic[4]	LQ2–magic[0]	LQ1–data[4]
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2–magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(c) Step 17

Figure A19. (Continued from Figure A18) The second part logical `T` gate operations for qubits arranged vertically: Logical `S` gate correction and `MoveBack`. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.

LQ1-data[5]	**LQ1-data[3]**	dummy[37]	dummy[50]	**LQ1-data[1]**	**LQ1-data[0]**	dummy[28]
dummy[10]	dummy[22]	dummy[16]	dummy[45]	dummy[11]	dummy[35]	**LQ1-data[2]**
dummy[51]	dummy[25]	dummy[21]	dummy[49]	dummy[19]	dummy[53]	dummy[39]
LQ1-data[6]	dummy[43]	LQ2-magic[5]	LQ2-magic[1]	dummy[18]	dummy[26]	dummy[32]
LQ2-magic[3]	dummy[15]	LQ2-magic[6]	dummy[33]	LQ2-magic[4]	LQ2-magic[0]	**LQ1-data[4]**
dummy[0]	dummy[29]	dummy[41]	dummy[14]	dummy[38]	dummy[47]	LQ2-magic[2]
dummy[31]	dummy[42]	dummy[9]	dummy[55]	dummy[5]	dummy[46]	dummy[34]
dummy[30]	dummy[20]	dummy[24]	dummy[17]	dummy[52]	dummy[1]	dummy[13]
dummy[3]	dummy[36]	dummy[40]	dummy[6]	dummy[8]	dummy[27]	dummy[48]
dummy[4]	dummy[23]	dummy[2]	dummy[7]	dummy[54]	dummy[12]	dummy[44]

(a) Final Mapping

Figure A20. (Continued from Figure A19) The second part of logical `T` gate operations for qubits arranged vertically: Logical `S` gate correction and `MoveBack`. Please compare the final mapping with the initial mapping (Figure A14 (a)). The positions of the data qubits are the same in both mappings, but the magic qubits are not. Note that the rectangles, hexagons and bi-directed arrow respectively indicate `S`, `PrepZ`, `MeasZ` and `SWAP` gates.