APPENDIX A REORDERING ALGORITHM

The reordering algorithm allows the user to compare an ordered set of qubits to the output of the quantum computation. Such an algorithm is necessary as while internal to the quantum computer abstraction qubits can be in arbitrary order grouped in arbitrary quantum registers, the user desires output in a specified order. This section presents the details of this algorithm.

There are some requested configurations which would be impossible to provide without merging quantum registers, and the first step of the reordering algorithm computes and merges quantum registers as needed so that it may be possible to sort and return the desired configuration. For example, if the user requests the order “q0”,“q1”,“q3”,“q4” and internally “q0” and “q4” are members of a single quantum register, and “q1” and “q3” are members of another, these two quantum registers will have to be merged before sorting.

After the first step, we know we have a set of quantum registers that it is possible to reorder into the requested order. However it could still be the case that we cannot return exactly the requested order. For example if “q0”,“q1”,“q3”,“q4” were again requested but in this case internally all 5-qubits reside in the same quantum register, it will be in general not possible to separate out “q2”. This is checked for and the algorithm throws an exception if sorting is not possible at this stage.

Since we are only dealing with a small number of qubits (5) it is possible to use a simplistic sorting algorithm for clarity; in this case bubble sort is chosen. With each step in the sorting algorithm, we must also rearrange the state of the quantum register involved to correspond to the new order. Using a sorting algorithm with simple well defined operations, bubble sort with its in place swaps, makes it easy to apply the necessary matrix operations to the quantum register.

The bubble sort algorithm is simple to describe: it steps through a list comparing adjacent items and swaps them as necessary, and repeats this stepping through until the list is sorted. It has a worst case performance of $\mathcal{O}(n^2)$. Since $n = 5$ in our case this is not a big penalty to pay for simplicity, and the nature of quantum computation makes this the least of our worries were **Quintuple** attempted to be extended to large n. The bubble sort algorithm is explicitly coded so that as we swap the qubits to match the desired order, we also rearrange the state of the quantum register involved to correspond with the new order. This is done by computing the permutation matrix corresponding to the rearranging prescribed by bubble sort, and applying this permutation matrix to the state. This is done with every swap that bubble sort prescribes of the qubit list, meaning that the state is in the corresponding order when the qubit list is sorted.

The final step of the reordering algorithm is to just return a single state representing the qubits of interest; this is possible as was ensured in the previous step. For example if “q0”,“q1”,“q3”,“q4” are requested, and “q2” resides in a separate quantum register than any of these qubits, then “q2” is ignored. The result is then easily computed as the ordered tensor product the quantum registers solely containing ordered qubits of interest. This result can be compared to the expected state supplied by the user.

The pseudo-code for the algorithm is included below:

Algorithm 1 Reordering

1: **function** REORDER(O: requested order)

Phase 1 - Merge quantum registers

Require: O is in increasing order
2: **for** $q \in O$ **do**
3: **for** $r \in R \leftarrow$ quantum registers **do**
4: $rmin \leftarrow$ smallest qubit in r
5: $rmax \leftarrow$ largest qubit in r
6: $S \leftarrow$ all qubits between (inclusive) $rmin$ and $rmax$
7: **if** $q \notin r \ \ \& \ \ q \in S$ **then**
8: $r_q \leftarrow$ the register q belongs to
9: MERGE(r_q, r)

Phase 2 - Sort quantum registers

Ensure: Every quantum register has qubits that are either all in O or none are in O
10: **for** $r \in R \leftarrow$ quantum registers **do**
11: $Q \leftarrow$ qubits in r
12: **if** $Q \cap O \notin \{\varnothing, Q\}$ **then**
13: **return** $failure$
14: **if** Q not ordered **then**
15: $n \leftarrow length(Q)$
16: $swapped \leftarrow true$
17: **while** $swapped \neq false$ **do**
18: $swapped \leftarrow false$
19: **for** $i = 0$ to $n - 1$ **do**
20: **if** $Q[i] > Q[i+1]$ **then**
21: SWAP$(r, i, i+1)$
22: $swapped \leftarrow true$

Phase 3 - Create combined answer state

23: $answer \leftarrow nil$
24: **for** $r \in R \leftarrow$ quantum registers **do**
25: $Q \leftarrow$ qubits in r
26: **for** $q \in Q$ **do**
27: **if** $Q \in O$ **then**
28: **if** $answer = nil$ **then**
29: $answer \leftarrow q$
30: **else**
31: $answer \leftarrow answer \otimes q$
32: **return** $answer$

Algorithm 2 Swap

procedure SWAP(r, i, j)
    $Q \leftarrow$ qubits in r
    $state \leftarrow$ state of r

Phase 1 - Permute the state

    $n \leftarrow length(Q)$
    $L \leftarrow$ all possible states of n qubits in canonical ordering
    $permute \leftarrow Id_{n \times n}$ ▷ $n \times n$ identity matrix
    $swapped \leftarrow \varnothing$
    for $c \in L$ **do**
        $newc \leftarrow c$
        SWAPHELPER($newc, i, j$)
        if $newc \neq c$ **then**
            $i_{\text{per}} \leftarrow$ index of c in L
            $j_{\text{per}} \leftarrow$ index of $newc$ in L
            $swap \leftarrow \{i_{\text{per}}, j_{\text{per}}\}$
            if $swap \notin swapped$ **then**
                $swapped \leftarrow swapped \cup swap$
                SWAPHELPER($permute.rows, i_{\text{per}}, j_{\text{per}}$)
    $state \leftarrow permute \cdot state$

Phase 2 - Swap the qubits in the register

    SWAPHELPER(Q, i, j)

Algorithm 3 Swap Helper

procedure SWAPHELPER(l, i, j)
    $tmp \leftarrow l[i]$
    $l[i] \leftarrow l[j]$
    $l[j] \leftarrow tmp$