Published: 10 June 2025
Accepted: 06 March 2025
Received: 12 November 2024

RESEARCH-ARTICLE

Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs

ZHICHENG ZHANG, University of Technology Sydney, Sydney, NSW, Australia

MINGSHENG YING, University of Technology Sydney, Sydney, NSW, Australia

Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs

ZHICHENG ZHANG, University of Technology Sydney, Australia
MINGSHENG YING, University of Technology Sydney, Australia

Quantum recursive programming has been recently introduced for describing sophisticated and complicated quantum algorithms in a compact and elegant way. However, implementation of quantum recursion involves intricate interplay between quantum control flow and recursive procedure calls. In this paper, we aim at resolving this fundamental challenge and develop a series of techniques to efficiently implement quantum recursive programs. Our main contributions include:

(1) We propose a notion of *quantum register machine*, the first quantum architecture (including an instruction set) that provides instruction-level support for quantum control flow and recursive procedure calls at the same time.
(2) Based on quantum register machine, we describe the first *comprehensive implementation process* of quantum recursive programs, including the compilation, the partial evaluation of quantum control flow, and the execution on the quantum register machine.
(3) As a bonus, our efficient implementation of quantum recursive programs also offers *automatic parallelisation* of quantum algorithms. For implementing certain quantum algorithmic subroutine, like the widely used quantum multiplexor, we can even obtain exponential parallel speed-up (over the straightforward implementation) from this automatic parallelisation. This demonstrates that quantum recursive programming can be win-win for both modularity of programs and efficiency of their implementation.

CCS Concepts: • **Theory of computation** → **Quantum computation theory**; Abstract machines; • **Software and its engineering** → *Compilers*; • **Computer systems organization** → **Quantum computing**.

Additional Key Words and Phrases: quantum programming languages, recursive definition, quantum architectures, compilation, partial evaluation, automatic parallelisation

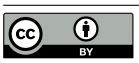
ACM Reference Format:
Zhicheng Zhang and Mingsheng Ying. 2025. Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs. *Proc. ACM Program. Lang.* 9, PLDI, Article 180 (June 2025), 26 pages. https://doi.org/10.1145/3729283

1 Introduction

Recursion in classical programming languages enables programmers to conveniently describe complicated computations as compact programs. By allowing any procedure to call itself, a short static program text can generate (unbounded) long dynamic program execution [34]. Examples of recursion include Hoare's quicksort algorithm [44], various recursive data structures [45], and divide-and-conquer algorithms. The implementation of classical recursion has been well-studied and was an important feature of the celebrated programming language ALGOL 60 [12, 13, 33, 82].

In the context of quantum programming, recursion has been recently studied for similar reasons (e.g., [31, 73, 88, 89, 91]). In particular, a language **RQC**$^{++}$ was introduced in [89, 91] for recursively

Authors' Contact Information: Zhicheng Zhang, University of Technology Sydney, Sydney, Australia, zhicheng.zhang@student.uts.edu.au; Mingsheng Ying, University of Technology Sydney, Sydney, Australia, mingsheng.ying@uts.edu.au.



ACM 2475-1421/2025/6-ART180
https://doi.org/10.1145/3729283

programmed quantum circuits and quantum algorithms. The expressive power of **RQC**$^{++}$ has been demonstrated by various examples.

The aim of this paper is to study **how quantum recursive programs can be efficiently implemented**. We choose to consider quantum recursive programs [1] described by the language **RQC**$^{++}$ [91]. But we expect that the techniques developed in this paper can work for other quantum programming languages that support recursion.

In general, quantum recursion involves the interplay of the following two programming features:

- *Quantum control flow* (in particular, those defined by quantum if-statements [3, 21, 72, 85, 90, 96]) that allow program executions to be in quantum superposition, controlled by some external quantum coin.
- *Recursive procedure calls* that allow a procedure to call itself with different classical parameters.

A good implementation of quantum recursive programs should support the above two features harmoniously. A better implementation should further be efficient.

1.1 Motivating Example: Quantum Multiplexor

> …increase of efficiency always comes down to exploitation of structure …
>
> Edsger W. Dijkstra [35]

To illustrate the basic idea of our implementation, let us start with an algorithmic subroutine called quantum multiplexor [74], and see how quantum recursive programs can benefit its description and implementation. Quantum multiplexor is used in a wide range of quantum algorithms, for example, linear combination of unitaries (LCU) [19, 20, 26, 51], Hamiltonian simulation [9–11, 57], quantum state preparation [6, 56, 98, 99], and solving quantum linear system of equations [25]. Let $N = 2^n$ and $[N] = \{0, 1, \ldots, N-1\}$. A quantum multiplexor can be described by the unitary

$$U = \sum\nolimits_{x\in[N]} |x\rangle\langle x| \otimes U_x. \tag{1}$$

Here, every unitary U_x is described by a quantum circuit, or more generally, a quantum program, say C_x. The quantum multiplexor U applies U_x, conditioned on the state $|x\rangle$ of the first n qubits.

A straightforward implementation of U is by applying a sequential products of N controlled-U_x:

$$\prod\nolimits_{x\in[N]} \Big(|x\rangle\langle x| \otimes U_x + \sum\nolimits_{y\neq x} |y\rangle\langle y| \otimes \mathbb{1} \Big). \tag{2}$$

This implementation has time complexity $O\big(\sum_{x\in[N]} T_x\big)$, where T_x is the time for executing C_x (i.e., implementing U_x). On the other hand, there exists a more efficient parallel implementation [98, 99] of U, with parallel time complexity $O\big(n + \max_{x\in[N]} T_x\big)$ (measured by the quantum circuit depth), using rather involved constructions similar to the bucket-brigade quantum random access memories [38, 39, 41, 42]. The implementation in [98, 99] achieves exponential parallel speed-up (with respect to n) over the straightforward one. The price for obtaining such efficiency is the manual design of rather low-level quantum circuits.

It is natural to ask *if we can design at high-level and still obtain an efficient implementation.* For this example of quantum multiplexor, the intuition is as follows. First, U can be described by a high-level quantum recursive program $\mathcal{P}$, which encapsulates both the control structure in Equation (1) and all programs C_x for describing unitaries U_x. Then, by storing the program $\mathcal{P}$ in a quantum memory, we can design a *quantum register machine* (to be formally defined in this paper) that automatically exploits the structure of $\mathcal{P}$ and executes all C_x's (i.e., implements all U_x's) in quantum superposition, thereby outperforming the straightforward implementation that only sequentially executes C_x's.

[1] As this terminology suggests, the recursion in such programs has a quantum nature.

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow P(n,0)\\
P(k,x) &\Leftarrow \textbf{if } k=0 \textbf{ then } Q[x]\\
&\qquad \textbf{else}\\
&\qquad\quad \textbf{qif}[q[k]]\ |0\rangle \to P(k-1,2x)\\
&\qquad\qquad \square\ |1\rangle \to P(k-1,2x+1)\\
&\qquad \textbf{fiq}\\
&\qquad \textbf{fi}\\
Q[0] &\Leftarrow C_0\\
&\cdots\\
Q[N-1] &\Leftarrow C_{N-1}.
\end{aligned}
$$

Fig. 1. Quantum multiplexor as a quantum recursive program.

Let us make the above intuition more concrete, by describing U in a quantum recursive program (in the language **RQC**$^{++}$ [91]; see Section 2) as in Figure 1. Here, the main procedure $P_{\text{main}}(n)$ describes U, and every $Q[x]$ (or their procedure body C_x) describes U_x. Procedure $P(k,x)$ recursively collects the control information x using the quantum if-statement (**qif** statement) and calls $Q[x]$ when $k=0$. At this point, we only need to note that the program in Figure 1 involves the interplay of the quantum control flow (managed by the **qif** statement) and recursive procedure calls. The **qif** statement in $P(k,x)$ creates two *quantum branches* (in superposition): when $q[k]$ is in state $|0\rangle$, $P(k-1,2x)$ is called; when $q[k]$ is in state $|1\rangle$, $P(k-1,2x+1)$ is called.

If the program in Figure 1 is compiled and stored into a quantum memory, then a quantum register machine that supports quantum control flow and recursive procedure calls can *run through the two quantum branches in superposition*. The cost for executing the **qif** statement only depends on the quantum branch that takes longer running time. This will incur a final time complexity proportional to the maximum $\max_{x\in[N]} T_x$ (compared to the sum $\sum_{x\in[N]} T_x$ in the straightforward implementation) and lead to an exponential parallel speed-up, similar to [98, 99].

1.2 Main Contributions

1.2.1 Architecture: Quantum Register Machine. We propose a notion of quantum register machine, a quantum architecture that provides instruction-level support for quantum control flow and procedure calls at the same time. Its storage components include a constant number of quantum registers (simply called registers in the sequel) and a quantum random access memory (QRAM). The QRAM stores both compiled quantum programs and quantum data. The machine operates on registers like a classical CPU, executing the compiled program by fetching instructions from the QRAM. The machine is also accompanied with a set of low-level instructions, each specifying operations to be carried out by the machine. We briefly explain how the quantum register machine handles the aforementioned two features as follows:

Handle quantum control flow: Inspired by the previous work [96] (which borrows ideas from the classical reversible architectures [8, 36, 81, 84]), we put the program counter into a quantum register, which can be in quantum superposition. However, existing techniques are insufficient to *automatically* handle a challenge introduced by the quantum control flow, known as the *synchronisation problem* [18, 32, 54, 59, 61, 63, 64, 75, 86, 96]. Specifically, previous work [96] circumvents this problem by manually inserting nop (no operation) into the low-level programs. This approach changes the static program text, and is not extendable to handle quantum recursive programs, because the length of dynamic computation generated by quantum recursion cannot be pre-determined from the static program text (see Appendix I.1[2] and Section 8 for further discussion).

In contrast, to automatically handle the synchronisation problem (without changing the static program text), our solution is to use a partial evaluation of quantum control flow (to be explained soon) before execution, and design a few corresponding quantum registers and mechanisms to exploit the partial evaluation result at runtime.

[2]Appendices are available in the full version of this paper [102].

Handle recursive procedure calls: We allocate a call stack in the QRAM. Stack operations are made reversible by borrowing techniques from the classical reversible computing (e.g., [7]). Note that at runtime, all quantum registers and the QRAM (where the dynamic call stack is stored) can be in an entangled quantum state.

It is worth pointing out that the quantum register machine does not aim to model any existing quantum hardware (typically controlled by classical pulses to implement standard quantum circuits). Indeed, quantum register machine should better be thought of as an abstract machine (that does *not* require hardware-level quantum control flow; see also [96]). Its execution is by repeatedly applying some fixed unitary operator per instruction cycle. Such unitary operator will be efficiently implemented by standard quantum circuits composed of one- and two-qubit gates.

1.2.2 Implementation: Compilation, Partial Evaluation and Execution. We propose a comprehensive process of implementing high-level quantum recursive programs (described in the language **RQC**$^{++}$) on the quantum register machine. This includes the following three steps: the first two are purely classical and the last is quantum.

Step 1. Compilation (Section 4): The high-level program in **RQC**$^{++}$ is compiled into a low-level one described by instructions, together with a series of transformations. The low-level instruction set is designed such that the high-level program structure can be exploited for later execution. This step only depends on the static program text and is independent of inputs.

Step 2. Partial evaluation (Section 5): Given the classical inputs (typically specifying the size of quantum inputs), the quantum control flow information of the compiled program is evaluated and stored into a data structure. In later execution, it will be loaded into the QRAM to help address the aforementioned synchronisation problem. This step is independent of quantum inputs.

Step 3. Execution (Section 6): With the compiled program and partial evaluation results loaded into the QRAM, the quantum inputs are finally considered, and the compiled program is executed with the aid of the partial evaluation results. The execution is done by repeatedly applying a fixed unitary (independent of the program) per cycle, which will be eventually implemented by standard quantum circuits with rigorously analysed complexity.

In Section 7, we describe the theoretical complexity of **Step 2** and **3**. More rigorous analysis can be found in Appendices D.4, E.3 and F.1. The final parallel time complexity, measured by the standard asymptotic (classical and quantum) circuit depth, is $O\big(T_{\mathrm{exe}}(\mathcal{P}) \cdot \big(T_{\mathrm{reg}} + T_{\mathrm{QRAM}}\big)\big)$. Intuitively, $T_{\mathrm{exe}}(\mathcal{P})$ is the time for executing the longest quantum branch in program $\mathcal{P}$; and T_{reg} and T_{QRAM} are complexities for elementary operations on registers and the QRAM, independent of the program.

1.2.3 Bonus: Automatic Parallelisation. We show that quantum recursive programming can be *win-win* for both modularity of programs (demonstrated in [91] via various examples) and efficiency of their implementation (realised in this paper). In particular, as a bonus, the efficient implementation in Section 1.2.2 also offers *automatic parallelisation*. For implementing certain quantum algorithmic subroutine, like the quantum multiplexor in Section 1.1, an exponential speed-up (over the straightforward implementation) can be obtained from this automatic parallelisation, in terms of (classical and quantum) parallel time complexity. Here, the classical parallel time complexity is relevant because the partial evaluation will be performed by a classical parallel algorithm.

For implementing the quantum multiplexor, we obtain the following theorem from the automatic parallelisation, whose proof sketch is to be shown in Section 7.

Theorem 1.1 (Automatic parallelisation of quantum multiplexor). *Via the quantum register machine, the quantum multiplexor in Equation (1) with each U_x consisting of T_x elementary unitary gates can be implemented in (classical and quantum) parallel time complexity (i.e., circuit depth) $\widetilde{O}\big(n \cdot \max_{x\in[N]} T_x + n^2\big)$, where $\widetilde{O}(\cdot)$ hides logarithmic factors.*

Although the complexity in Theorem 1.1 is slightly worse than that in [98, 99] by a factor of $\widetilde{O}(n)$, it is worth stressing that the parallelisation in Theorem 1.1 is obtained automatically. Our framework steps towards a *top-down* design of (parallel) efficient quantum algorithms: the programmer only needs to design the high-level quantum programs (like in Figure 1), and the parallelisation is automatically realised by our implementation based on the quantum register machine. Further comparison of Theorem 1.1 and [98, 99] can be found in Appendix I.2.

1.3 Structure of the Paper

For convenience of the reader, in Section 2 we briefly review the language $\mathbf{RQC}^{++}$ [91] for describing quantum recursive programs. In Section 3, we introduce the notion of quantum register machine. In Section 4, we present the compilation of programs in $\mathbf{RQC}^{++}$ to low-level instructions. Then, in Section 5, we present the partial evaluation of quantum control flow on the compiled program. In Section 6, we present the execution on quantum register machine. Finally, in Section 7, we analyse the efficiency of implementing quantum recursive programs in our framework, and show how it offers automatic parallelisation. In Section 8 we discuss related work, and in Section 9 we conclude and discuss future topics. Further details and examples are presented in the appendices, which are available in the full version of this paper [102].

2 Background on Quantum Recursive Programs

In this section, we briefly introduce the high-level language $\mathbf{RQC}^{++}$ for describing quantum recursive programs, defined in [91]. A more detailed introduction can be found in Appendix A. Two key features of $\mathbf{RQC}^{++}$, compared to other existing quantum programming languages, are quantum control flow and recursive procedure calls, which together support the quantum recursion (different from classical recursion in quantum programs as considered in e.g., [31, 85] and classically bounded recursion in superposition as considered in e.g., [94, 95]). An additional contribution of this paper is providing further insights into $\mathbf{RQC}^{++}$ from an implementation perspective.

2.1 Syntax

The alphabet of $\mathbf{RQC}^{++}$ consists of: (a) Classical variables, often denoted by $x, x_1, x_2, \ldots$; (b) Quantum variables, often denoted by $q, q_1, q_2, \ldots$; (c) Procedure identifiers, often denoted by $P, Q, P_1, P_2, \ldots$; and (d) Elementary unitary gates and elementary classical arithmetic operators. A program in $\mathbf{RQC}^{++}$ describes a *parameterised unitary* without measurements (see Section 8 for discussion about the unitary restriction). Classical variables are solely for specifying the control of programs. For example, in Figure 1, k and x define the formal parameters of $P(k, x)$, and are used in the if-statement and the actual parameters for procedure calls. Classical variables can also store intermediate computation results (see the syntax in Figure 2). We use $\overline{x} = x_1 \ldots x_n$ to denote a list of classical variables. Similar notations apply to quantum variables and procedure identifiers.

Variables can be simple or array variables. The notion of array is standard, e.g., if x is a 1-indexed one-dimensional classical array, then $x[10]$ represents the 10^{th} element in x. Array variables induce subscripted variables: e.g., for a quantum array q, $q[2y + z]$ is an element in q with subscription $2y + z$. For simplicity, in this paper we only consider one-dimensional arrays, and requires that for any classical subscripted variable $x[t]$, the expression t contains no more subscripted variables.

We also consider arrays of procedure and subscripted procedure identifiers, for which notations are similar to that for variables. Moreover, for any procedure identifier P, we associate with it a classical variable $P.ent$, storing the entry address of the declaration of P. The value of $P.ent$ is determined after the program is compiled and loaded into the quantum memory.

$$
\begin{array}{ll}
\mathcal{P} = \{P_1(\overline{u}_1) \Leftarrow C_1, ..., P_n(\overline{u}_n) \Leftarrow C_n\} & \text{Procedure declaration} \\
C ::= \textbf{skip} \mid C_1; C_2 & \text{Sequential composition} \\
\quad \mid \overline{x} := \overline{t} & \text{Classical assignment} \\
\quad \mid U[\overline{q}] & \text{Quantum unitary gate} \\
\quad \mid P(\overline{t}) & \text{Procedure call} \\
\quad \mid \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} & \text{Classical if-statement} \\
\quad \mid \textbf{while } b \textbf{ do } C \textbf{ od} & \text{Classical loop} \\
\quad \mid \textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end} & \text{Local classical variable block} \\
\quad \mid \textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq} & \text{Quantum if-statement}
\end{array}
$$

Fig. 2. The syntax of quantum recursive programming language **RQC**$^{++}$.

$$
\begin{array}{ll}
\text{(SK)} & (\textbf{skip}, \sigma, |\psi\rangle) \to (\downarrow, \sigma, |\psi\rangle) \\
\text{(AS)} & \left(\overline{x} := \overline{t}, \sigma, |\psi\rangle\right) \to \left(\downarrow, \sigma\left[\overline{x} := \sigma(\overline{t})\right], |\psi\rangle\right) \\
\text{(GA)} & \dfrac{\sigma \models Dist(\overline{q})}{(U[\overline{q}], \sigma, |\psi\rangle) \to \left(\downarrow, \sigma, \left(U_{\sigma(\overline{q})} \otimes \mathbb{1}\right)|\psi\rangle\right)} \\
\text{(SC)} & \dfrac{(C_1, \sigma, |\psi\rangle) \to \left(C_1', \sigma', |\psi'\rangle\right)}{(C_1; C_2, \sigma, |\psi\rangle) \to \left(C_1'; C_2, \sigma', |\psi'\rangle\right)} \\
\text{(IF)} & \dfrac{\sigma \models b}{(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, \sigma, |\psi\rangle) \to (C_1, \sigma, |\psi\rangle)}, \quad \dfrac{\sigma \models \neg b}{(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, \sigma, |\psi\rangle) \to (C_2, \sigma, |\psi\rangle)} \\
\text{(LP)} & \dfrac{\sigma \models b}{(\textbf{while } b \textbf{ do } C \textbf{ od}, \sigma, |\psi\rangle) \to (C; \textbf{while } b \textbf{ do } C \textbf{ od}, \sigma, |\psi\rangle)}, \quad \dfrac{\sigma \models \neg b}{(\textbf{while } b \textbf{ do } C \textbf{ od}), \sigma, |\psi\rangle \to (\downarrow, \sigma, |\psi\rangle)} \\
\text{(BS)} & \left(\textbf{begin local } \overline{x} := \overline{t}; C \textbf{ end}, \sigma, |\psi\rangle\right) \to \left(\overline{x} := \overline{t}; C; \overline{x} := \sigma(\overline{x}), \sigma, |\psi\rangle\right) \\
\text{(RC)} & \dfrac{P(\overline{u}) \Leftarrow C \in \mathcal{P}}{\left(P(\overline{t}), \sigma, |\psi\rangle\right) \to \left(\textbf{begin local } \overline{u} := \overline{t}; C \textbf{ end}, \sigma, |\psi\rangle\right)} \\
\text{(QIF)} & \dfrac{|\psi\rangle = \alpha_0 |0\rangle_{\sigma(q)} |\theta_0\rangle + \alpha_1 |1\rangle_{\sigma(q)} |\theta_1\rangle, \qquad (C_i, \sigma, |\theta_i\rangle) \to^* \left(\downarrow, \sigma, |\theta_i'\rangle\right)\ (i = 0, 1)}{(\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq}, \sigma, |\psi\rangle) \to \left(\downarrow, \sigma, \alpha_0 |0\rangle_{\sigma(q)} |\theta_0'\rangle + \alpha_1 |1\rangle_{\sigma(q)} |\theta_1'\rangle\right)}
\end{array}
$$

Fig. 3. Transition rules for defining the operational semantics of **RQC**$^{++}$.

The syntax of **RQC**$^{++}$ is summarised in Figure 2. Here, a program is specified by $\mathcal{P}$, a set of procedure declarations, with a main procedure P_{main}. Each procedure declaration has the form $P(\overline{u}) \Leftarrow C$, where P is the procedure identifier, $\overline{u}$ is a list of formal parameters (which can be empty), and C is the procedure body. The recursion is supported by that C can contains P itself. A statement C is inductively defined, where U represents an elementary unitary gate and b represents a classical binary expression. We further explain as follows.

- The procedure call $P(\overline{t})$ has a list of classical expressions $\overline{t}$ as its actual parameters.
- The block statement **begin local** $\overline{x} := \overline{t}; C$ **end** temporarily sets classical variables $\overline{x}$ to the values of $\overline{t}$ at the beginning of the block, and restores their old values at the end.
- The unitary gate $U[\overline{q}]$ applies the elementary quantum gate U on quantum variables $\overline{q}$.
- The quantum if-statement $\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq}$ executes C_i, conditioned on the qubit variable q (a.k.a., quantum coin): when q is in state $|0\rangle$, C_0 is executed; when q is in state $|1\rangle$, C_1 is executed. Unlike the classical if-statement where the control flow only runs through one of the two branches, the quantum control flow run through both quantum branches created by the **qif** statement, in superposition. Note that the superposition state is held in the composite system including q and the quantum variables in C_0, C_1.

2.2 Semantics

Now we briefly introduce the operational semantics of **RQC**$^{++}$. We use $(C, \sigma, |\psi\rangle)$ to denote a configuration, where C is the remaining statement to be executed or $C = \downarrow$ (standing for termination), σ is the current classical state, and $|\psi\rangle$ is the current quantum state. The operational semantics is defined in terms of transitions between configurations of the form: $(C, \sigma, |\psi\rangle) \to (C', \sigma', |\psi'\rangle)$.

The transition rules for defining the operational semantics of **RQC**$^{++}$ are shown in Figure 3. For simplicity of presentation, we only explain the most non-trivial (QIF) rule. Other rules are rather standard and further explained in Appendix A. In the (QIF) rule, $i = 0, 1$ correspond to the two quantum branches, controlled by the external quantum coin q. Here, $\sigma(q)$ denotes the subsystem specified by q with respect to classical state σ. As usual, $\to^k$ denotes the composition of k copies of $\to$, and $\to^* = \bigcup_{k=0}^{\infty} \to^k$. The semantics of the **qif** statement is exactly a quantum multiplexor [74] with one control qubit q: if each C_i describes a unitary U_i, then **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$**fiq** describes the unitary $U_0 \oplus U_1 = |0\rangle\langle 0|_q \otimes U_0 + |1\rangle\langle 1|_q \otimes U_1$.

Note that in the (QIF) rule, $(C_i, \sigma, |\theta_i\rangle)$ are required to terminate in the same classical state σ for both branches ($i = 0, 1$) to *prevent classical variables from being in superposition*.[3] This requirement seems inevitable to separate classical and quantum variables in the presence of quantum control flow. As a result, only local classical variables can be arbitrarily modified in the **qif** statement. If one wishes to return different data from two quantum branches, then the data becomes intrinsically quantum and should therefore be stored in quantum variables.

2.3 Conditions for Well-Defined Semantics

We present three conditions for a program in **RQC**$^{++}$ to have well-defined semantics, in particular, for the (QIF) rule to be properly and easily applied. The first condition guarantees that in every **qif** statement, q is external to C_0 and C_1. This is introduced for the **qif** statement to be physically meaningful. We use $qv(C, \sigma)$ to denote the quantum variables in statement C with respect to a given classical state σ. Its precise definition is given in Appendix A.3.

Condition 2.1 (External quantum coin). For any procedure declaration $P(\overline{u}) \Leftarrow C \in \mathcal{P}$, and any **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$**fiq** appearing in C, and any classical state σ (of concern), $q \notin qv(C_0, \sigma) \cup qv(C_1, \sigma)$.

The second condition says that in every **qif** statement, both C_0 and C_1 contain no free changed (classical) variables. A classical variable is *free* if it is not declared as local variable. It is *changed* if it appears on the LHS of an assignment. We use $fcv(C, \sigma)$ to denote the free changed variables in C with respect to σ. See Appendix A.3 for its precise definition. This condition is introduced as the (QIF) rule requires $(C_i, \sigma, |\psi\rangle)$ to terminate in the same classical state σ for both branches $i = 0, 1$.

Condition 2.2 (No free changed variables in **qif** *statements).* For any $P(\overline{u}) \Leftarrow C \in \mathcal{P}$, any **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$**fiq** appearing in C, and any classical state σ (of concern), $fcv(C_0, \sigma) = fcv(C_1, \sigma) = \emptyset$.

The third condition says that every procedure body contains no free changed variables. This condition is introduced to simplify the process of compilation, as it allows the procedure calls to be arbitrarily used together with the **qif** statements without violating Condition 2.2.

Condition 2.3 (No free changed variables in procedure bodies). For any $P(\overline{u}) \Leftarrow C \in \mathcal{P}$ and any classical state σ (of concern), $fcv(C, \sigma) = \emptyset$.

[3] For simplicity of later implementation, this requirement has been made slightly stricter than the original one ("terminating in the same σ'", where σ' may differ from the initial σ) described in [91], but remains easy to meet in practice.

3 Quantum Register Machine

Now we start to consider how to implement quantum recursive programs defined in the previous section. As the basis, let us introduce the notion of quantum register machine, an architecture that provides instruction-level support for quantum control flow and recursive procedure calls at the same time. Unlike most existing quantum architectures that use classical controllers to implement quantum circuits, the quantum register machine stores quantum programs and data in a quantum random access memory (QRAM) and executes on quantum registers. As aforementioned in Section 1, since existing quantum hardware is typically controlled by classical pulses, it would be better to think quantum register machine as an abstract machine (that does not require hardware-level quantum control flow). Like a classical CPU, the machine works by repeatedly applying a fixed unitary U_{cyc} (independent of the program) per instruction cycle, which consists of several stages, including fetching an instruction from the QRAM, decoding it and executing it by performing corresponding operations. To support quantum control flow, additional stages related to the partial evaluation are also needed. The unitary U_{cyc} will be eventually implemented by standard quantum circuits, as described in Section 6 and visualised in Appendix E.1.

In the following, we first explain quantum registers and QRAM, and then describe a low-level instruction set **QINS** (quantum instructions) for the quantum register machine.

3.1 Quantum Registers

The quantum register machine has a constant number of quantum registers (or simply, registers), each storing a quantum word composed of L_{word} (called word length) qubits. Registers are directly accessible. The machine can perform a series of elementary operations on registers, including word-level arithmetic operations (see also Section 3.3), each assumed to take time T_{reg}. The precise definition of elementary operations are presented in Appendix B.1.

Registers are grouped into two types: system and user registers. There are eight system registers. The first five are rather standard and borrowed from the classical reversible architectures [8, 36, 81, 84], as quantum unitaries are intrinsically reversible. We describe their classical effects as follows.

- Program counter *pc* records the address of the current instruction.
- Instruction *ins* records the current instruction.
- Branching offset *br* records the offset of the address of the next instruction to go from *pc*. More specifically, if $br = 0$, then the address of the next instruction will be $pc + 1$. Otherwise, the address of the next instruction will be $pc + br$.
- Return offset *ro* records the offset for *br* in the return of a procedure call.
- Stack pointer *sp* records the current topmost location of the call stack.

In contrast, the last three system registers are novelly introduced to support an efficient implementation of the **qif** statements. They are related to the *qif table*, a data structure generated by the partial evaluation of quantum control flow and used during execution to address the aforementioned synchronisation problem. We briefly describe their classical effects as follows, and will explain further details in Sections 5 and 6.

- Qif table pointer *qifv* records the current node in the qif table.
- Qif wait counter *qifw* records the number of instruction cycles to wait at the current node in the qif table.
- Qif wait flag *wait* records whether the current instruction cycle needs to be skipped.

We also set the initial values of theses registers: *pc*, *sp* and *qifv* are initialised to $|j\rangle$, where j is the starting addresses of the main program, the call stack and the qif table, respectively. Other system and user registers are initialised to $|0\rangle$.

3.2 Quantum Random Access Memory

The quantum register machine has a quantum random access memory (QRAM)[4] composed of N_{QRAM} memory locations, each storing a quantum word. The QRAM is not directly accessible. Like a classical memory, access to QRAM is by providing an address register specifying the address, and a target register to hold the information retrieved from the specified location. Unlike the classical case, the address register can be in quantum superposition, and registers can be entangled with the QRAM. In this paper, we assume the following two types of elementary QRAM accesses.

Definition 3.1 (Elementary QRAM accesses).

- QRAM (swap) load. This access performs the unitary $U_{\text{ld}}(r, a, mem)$ defined by the mapping:
$$|x\rangle_r |i\rangle_a |M\rangle_{mem} \mapsto |M_i\rangle_r |i\rangle_a |M_0, \ldots, M_{i-1}, x, M_{i+1}, \ldots, M_{N_{\text{QRAM}}-1}\rangle_{mem}, \tag{3}$$
for all x, i and $M = (M_0, \ldots, M_{N_{\text{QRAM}}-1})$. Here, r is the target register, a is the address register, and mem is the QRAM.
- QRAM (xor) fetch. This access performs the unitary $U_{\text{fet}}(r, a, mem)$ defined by the mapping:
$$|x\rangle_r |i\rangle_a |M\rangle_{mem} \mapsto |x \oplus M_i\rangle_r |i\rangle_a |M\rangle_{mem}, \tag{4}$$
for all x, i, M.

Moreover, the controlled versions (controlled by a register) of elementary QRAM accesses are also considered elementary, since the number of registers is constant and the control only incurs a constant overhead. Suppose every elementary QRAM access takes time T_{QRAM}.

Some readers might notice that the physical realisation of QRAM is not yet near-term, a challenge shared by most works leveraging QRAM (e.g., [1, 4, 17, 94]). Nevertheless, there are ongoing efforts towards feasible QRAM implementations (e.g., [41, 42, 87]). Importantly, the final complexity of our implementation of quantum recursive programs is measured by the standard circuit depth and unaffected by the near-term feasibility of QRAM.

It is also worth pointing out that managing entanglement between registers and the QRAM is crucial, as improper handling can result in incorrect output states [55]. To this end, the instruction set **QINS** (Section 3.3) and the compilation process (Section 4) are carefully designed to ensure that, after the execution, quantum variables are *disentangled* from other registers and the remaining part of the QRAM. A key of the design is *proper uncomputation* of intermediate results. The idea traces back to [15, 52], and has been applied in [8, 36, 81, 84, 96]. Moreover, in our design, the creation and removal of entanglement during the execution align with the program structure (in **RQC**$^{++}$).

3.2.1 Layout of the QRAM. The QRAM in the quantum register machine stores both programs and data. In particular, it contains the following sections.

(1) Program section stores the compiled program in a low-level language **QINS** (to be defined in Section 3.3).
(2) Symbol table section stores the name of every variable and its corresponding address. Here, unlike in the classical case, the symbol table is used at runtime instead of compile time (see also Appendix C.3), because arrays in **RQC**$^{++}$ are not declared with fixed size.
(3) Variable section stores the classical and quantum variables.
(4) Qif table section stores the qif table (to be defined in Section 5.2).
(5) Stack section stores the call stack to handle the procedure calls. The stack is composed of multiple stack frames, each storing the actual parameters and return offset (from the caller to the callee), and the local data used by the callee, in a procedure call.

[4]In particular, the QRAM considered here is quantum random access quantum memory (QRAQM). Readers are referred to [47] for a review of QRAM.

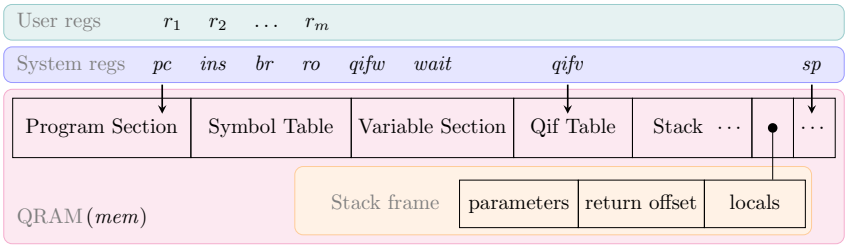


Fig. 4. Storage components of the quantum register machine and the layout of the QRAM. All components can be together in a quantum superposition state.

	Instruction	Classical Effect
Load	`ld(r,i)`	$r \leftrightarrow M_i$
	`ldr(r1,r2)`	$r_1 \leftrightarrow M_{r_2}$
	`fetr(r1,r2)`	$r_1 \leftarrow r_1 \oplus M_{r_2}$
Gate	`uni(G,r1)`	Apply gate G on r_1
	`unib(G,r1,r2)`	Apply gate G on $r_1 r_2$
Arith	`xori(r,i)`	$r \leftarrow r \oplus i$
	`addi(r,i)`	$r \leftarrow r + i$
	`subi(r,i)`	$r \leftarrow r - i$
	`swap(r1,r2)`	$r_1 \leftrightarrow r_2$
	`add(r1,r2)`	$r_1 \leftarrow r_1 + r_2$
	`sub(r2,r2)`	$r_1 \leftarrow r_1 - r_2$
	`neg(r1)`	$r_1 \leftarrow -r_1$
	`ari(op,r1,r2)`	$r_1 \leftarrow r_1 \oplus (op\ r_2)$
	`arib(op,r1,r2,r3)`	$r_1 \leftarrow r_1 \oplus (r_2\ op\ r_3)$
Branch	`bra(i)`	$br \leftarrow br \oplus i$
	`bez(r,i)`	$br \leftarrow br \oplus (i \cdot [r = 0])$
	`bnz(r,i)`	$br \leftarrow br \oplus (i \cdot [r \neq 0])$
	`swbr(r)`	$br \leftrightarrow r$
Qif	`qif(r)`	update *qifv* and *qifw*
	`fiq(r)`	update *qifv*
Special	`start`	none
	`finish`	none

(a) Instructions and corresponding classical effects. Here, $\oplus$ denotes the XOR operator; $[b] = 1$ if b is true and $[b] = 0$ otherwise.

Type	Format (*ins*)				
I	*opcode*	*reg*	*imm*		
	c	*r*	*i*		
R	*opcode*	reg_1	reg_2		
	c	r_1	r_2	0	0
O	*opcode*	*para*	reg_1	reg_2	reg_3
	c	*G*/*op*	r_1	r_2	r_3

$$U_{\text{dec}} = \sum_c |c\rangle\langle c| \otimes \underbrace{\sum_d |d\rangle\langle d| \otimes U_{c,d}}_{U_c}$$

Type	*c*	$U_{c,d}$
I	`ld`	$U_{\text{ld}}(r, imm, mem)$
	`xori`	$U_{\oplus}(r, imm)$
	`bnz`	$\circ(r)$-$U_{\oplus}(br, imm)$
R	`ldr`	$U_{\text{ld}}(r_1, r_2, mem)$
	`swap`	$U_{\text{swap}}(r_1, r_2)$
	`qif`	$U_{\text{qif}}(r_1)$
O	`uni`	$U_G(r_1)$
	`ari`	$U_{op}(r_1, r_2)$

(b) Instruction formats, the decoding unitary U_{dec}, and selected examples of instruction implementations.

Fig. 5. The low-level language QINS and selected examples.

We visualise the quantum register machine and the layout of the QRAM in Figure 4.

3.3 The Low-Level Language QINS

Now we present QINS, an instruction set for describing the compiled programs. Each instruction specifies a series of elementary operations to be carried out by the quantum register machine. There are 22 instructions in QINS, which are listed with their classical effects in Figure 5a. Here, we leave the explanation of instructions `qif` and `fiq` to Section 6. The classical effects of other instructions are lifted to quantum in the standard way when being executed by the quantum register machine.

The design of QINS is inspired by the existing classical reversible instruction sets [8, 36, 81, 84]. Nevertheless, several instructions in QINS are essentially new. The most important are instructions `qif` and `fiq`, which are designed for a *structured management* of quantum control flow (generated by

the **qif** statements in **RQC**$^{++}$), in particular, aiding the partial evaluation and execution. Instructions `uni` and `unib` are designed for quantum unitary gates.

We group the instructions into three types: I (immediate-type), R (register-type) and O (other-type), according to their formats, as shown in Figure 5b. During the execution (to be described in Section 6), we decode the instruction in register *ins* by performing a unitary U_{dec} (see Figure 5b). U_{dec} is a quantum multiplexor, with section *opcode* as its first part of control, and other sections (depending on the type I/R/O) in *ins* as its second part of control. Let c be a computational basis in the first part and d in the second part, then the unitary being controlled is denoted by $U_{c,d}$.

For illustration, selected instructions and corresponding $U_{c,d}$ are presented in Figure 5b. Here, U_{ld} is the QRAM access in Definition 3.1. U_{qif} (and similarly U_{fiq} for `fiq`) will be defined in Section 6. Other unitaries are elementary operations on registers: (a) $U_{\oplus}$ performs the mapping $|x\rangle\,|y\rangle \mapsto |x \oplus y\rangle\,|y\rangle$; (b) $\circ(r)$-U stands for the controlled version $|0\rangle\langle 0|_r \otimes U + \sum_{x\neq 0} |x\rangle\langle x|_r \otimes \mathbb{1}$ of unitary U; (c) U_{swap} performs the mapping $|x\rangle\,|y\rangle \mapsto |y\rangle\,|x\rangle$; (d) U_G applies the elementary gate G (chosen from a fixed set $\mathcal{G}$ of size $O(1)$); (e) U_{op} performs the mapping $|x\rangle\,|y\rangle \mapsto |x \oplus (op\ y)\rangle\,|y\rangle$ for unary operator *op* (chosen from a fixed set $\mathcal{OP}$ of size $O(1)$).

Further details of **QINS** are provided in Appendix B.2.

4 Compilation

As usual, the first step in the implementation of quantum recursive programs is their compilation. The compilation of a program $\mathcal{P}$ in **RQC**$^{++}$ consists of the following passes.

(1) First, a series of high-level transformations are performed on the original $\mathcal{P}$ to obtain $\mathcal{P}_h$, which simplify the program structure and make it easier to be further compiled.
(2) Then, the transformed program $\mathcal{P}_h$ is translated into an intermediate program $\mathcal{P}_m$ in the mid-level language composed of instructions similar to those in **QINS** but more flexible.
(3) Finally, the mid-level $\mathcal{P}_m$ is translated into a program $\mathcal{P}_l$ in the low-level language **QINS**.

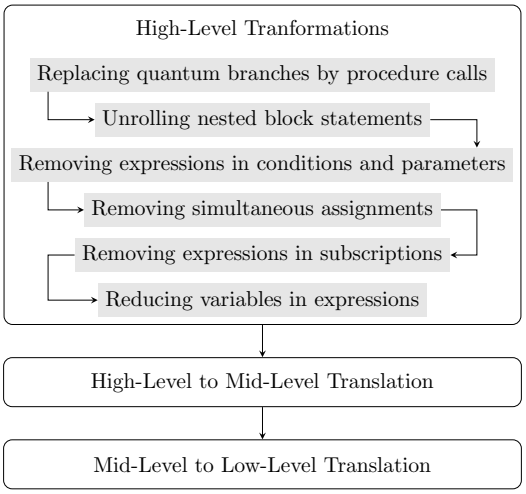


Fig. 6. The compilation process.

The compilation process is visualised in Figure 6. The remainder of this section is devoted to describe these passes carefully. In the sequel, we always assume the source program $\mathcal{P}$ to be compiled satisfies Conditions 2.1 to 2.3 in Section 2.3 and *has well-defined semantics*. We will not bother checking the syntax and semantics of $\mathcal{P}$.

4.1 High-Level Transformations

In this section, we describe the first pass of high-level transformation from $\mathcal{P}$ to $\mathcal{P}_h$. The major target of this pass is to simplify the automatic uncomputation of classical variables in later passes.

A program $\mathcal{P}$ in **RQC**$^{++}$ may contain irreversible classical statements, e.g., assignment $x := 1$. Reversibly implementing these statements introduces garbage data, e.g., through the standard Landauer [52] and Bennett [15] methods. For the overall correctness of the quantum computation,

these garbage data should be properly uncomputed. Moreover, the block statement in **RQC**$^{++}$ explicitly requires uncomputation of local variables at the end of the block.

In the execution of a program, when should we perform uncomputation? First, we realise a difficulty from the uncomputation of local variables in nested block statements. Consider the example in Figure 7. The inner block modifies x, which is used by the outer block. If one tries to uncompute the local variable y at the end of the inner block, the change on x (by the inner block) is also uncomputed, which is an undesirable side effect.

begin local $x := 1$;
 begin local $y := 2x$;
  $x := y + 1$;
  $y := 2x$
 end;
 $U[q[x+2]]$
end

Fig. 7. An example of nested block statement.

To overcome this difficulty, we will perform a series of transformations on the source program $\mathcal{P}$, such that the transformed $\mathcal{P}_h$ no longer contains nested block statements. Along the way, we also simplify the structure of the program. Consequently, for $\mathcal{P}_h$, we only need to perform uncomputation at the end of every procedure body (of procedure declarations), which will be automatically done in the high-level to mid-level translation (in Section 4.2).

An overview of high-level transformations is already shown in Figure 6. In the following we only select the first two steps for explanation, while other steps are rather standard (see e.g., the textbook [2]) and presented in Appendix C.1.

4.1.1 Replacing Quantum Branches by Procedure Calls. In this step, we replace the program in every quantum branch of every **qif** statement by a procedure call. More specifically, for every **qif** statement, if C_0, C_1 are not procedure identifiers or **skip** statements, then we introduce fresh procedure identifiers P_0, P_1, perform the replacement:

$$\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq} \qquad \Rightarrow \qquad \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq},$$

and add new procedure declarations $P_i \Leftarrow C_i$ (for $i \in \{0,1\}$) to $\mathcal{P}$. If only one of C_0, C_1 is procedure identifier or **skip**, then the replacement is performed only for the other branch. It is easy to see the above transformation does not violate Conditions 2.1 to 2.3.

4.1.2 Unrolling Nested Block Statements. In this step, we unroll all nested block statements. The program after this step is promised to no more contain block statements, but uncomputation of classical variables needs to be done at the end of every procedure body when the program is implemented, for it to preserve its original semantics. To do this, for any $P(\overline{u}) \Leftarrow C' \in \mathcal{P}$ and every block statement appearing in C', we perform the replacement:

$$\textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end} \qquad \Rightarrow \qquad \overline{x'} := \overline{t}; C[x'/x],$$

where $\overline{x'}$ is a list of fresh variables, and $[x'/x]$ stands for replacing variable x by x'. Also, we append $\overline{x'} := \overline{0}$ at the beginning of C'. The above transformation keeps Conditions 2.1 to 2.3 too.

Note that after this step, the program is technically in some new language with the same syntax as **RQC**$^{++}$, but whose semantics requires the uncomputation of classical variables at the end of every procedure body.

4.1.3 After the High-Level Transformations. We observe that the program $\mathcal{P}_h = \{P(\overline{u}) \Leftarrow C\}_P$ after the high-level transformations in Figure 6 has the following simplified syntax:

$$\begin{aligned} C ::= {} & \textbf{skip} \mid x := t \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{x}) \mid \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } x \textbf{ do } C \textbf{ od} \\ & \mid \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq}, \end{aligned}$$

where every subscripted variable and procedure identifier has a basic classical variable as its subscription (e.g., $x[y]$), and every expression has the form $t \equiv op\, x$ or $t \equiv x\ op\ y$. As aforementioned, the semantics is slightly changed: uncomputation of classical variables are needed at the end of

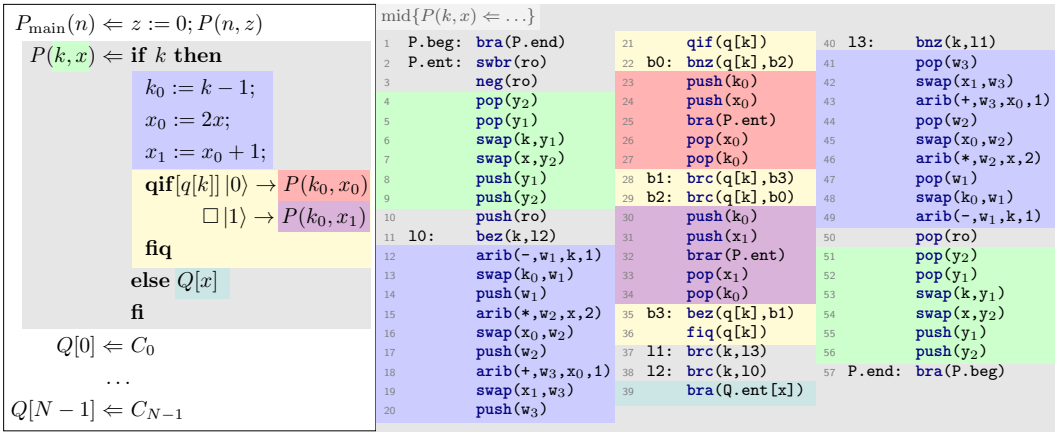


Fig. 8. Example of high-level transformations and high-to-mid-level translation. The original program is the quantum multiplexor program in Figure 1. Here, on the LHS is the program after the high-level transformations. On the RHS is the high-to-mid-level translation of the procedure $P(k, x)$. Their connections are highlighted in colors. Note that new variables x_0, x_1, k_0 are introduced by the rather standard third step of high-level transformations (see also Appendix C.1), and some transformations have no effect on this example.

every procedure body in the implementation, (which will be automatically done in the high-level to mid-level translation in Section 4.2).

For illustration, on the LHS of Figure 8, we show an after-the-high-level-transformations version of the quantum multiplexor program in Figure 1.

4.2 High-Level to Mid-Level Translation

Now we translate the transformed high-level program $\mathcal{P}_h$ obtained in the previous subsection into $\mathcal{P}_m$ in a mid-level language, which is different from the low-level language **QINS** (defined in Section 3.3) in the following aspects:

- We do not consider the memory allocation. Thus, instructions `ld`, `ldr` and `fetr` are not needed at this stage.
- Beyond registers and numbers, instructions can also take variables and labels as input. Here, like in the classical assembly language, a label is an identifier for the address of an instruction. (When the program is further translated into **QINS**, in the next section, every label l will be replaced by the offset of the address of where l is defined from the address of where l is used.)
- We have additional instructions `push` and `pop` for stack operations. Also, an additional branching instruction `brc` will be used in pair with `bez` (or `bnz`). In particular, `brc(x,1)`, compared to `bra(1)`, has the additional information of some variable x.

The high-to-mid-level translation also automatically handles the initialisation of formal parameters and the uncomputation of classical variables at the end of procedure bodies (see Section 4.1).

Let us use mid$\{D\}$ to denote the high-to-mid-level translation of a statement (or declaration) D in **RQC**$^{++}$. In Figure 9, we present selected examples of the high-to-mid-level translation, and more details are shown in Appendix C.2. Here, init$\{\cdot\}$ and uncp$\{\cdot\}$ denote the initialisation of formal parameters and uncomputation of classical variables, respectively. We further explain them as follows.

mid{$U[q]$}
```
uni(U,q)
```
mid{**while**}
```
10:  bnz(y,12)
11:  bez(x,13)
     add(y,1)
     mid{C}
12:  brc(y,10)
13:  brc(x,11)
```
mid{**qif** ... **fiq**}
```
     qif(q)
10:  bnz(q,12)
     mid{C0}
11:  brc(q,13)
12:  brc(q,10)
     mid{C1}
13:  bez(q,11)
     fiq(q)
```
mid{$P(\overline{x})$}
```
push(x1)
...
push(xn)
bra(P.ent)
pop(xn)
...
pop(x1)
```
mid{$P(\overline{u}) \Leftarrow C$}
```
P.beg:  bra(P.end)
P.ent:  swbr(ro)
        neg(ro)
        init{u}
        push(ro)
        mid{C}
        uncp{C}
        pop(ro)
        init{u}
P.end:  bra(P.beg)
```
init{$\overline{u}$}
```
pop(yn)
...
pop(y1)
swap(u1,y1)
...
swap(un,yn)
push(y1)
...
push(yn)
```
uncp{$P(\overline{x})$}
uncp{**qif** ... **fiq**}
$\emptyset$
uncp{**while**}
```
10:  bnz(x,12)
11:  bez(y,13)
     uncp{C}
     sub(y,1)
12:  brc(x,10)
13:  brc(y,11)
```

Fig. 9. Selected examples of the high-to-mid-level translation. Here, init{·} and uncp{·} stand for the initialisation of formal parameters and uncomputation of classical variables, respectively. $y, y_1, \ldots, y_n$ are all fresh variables. Also, mid{**while**} and uncp{**while**} use the same fresh variable y.

- To reversibly implement **while** x **do** C **od**, a fresh variable y is introduced to count the number of loops. Similar to the classical reversible architectures [8, 36, 81, 84], we use a pair of branching instructions (e.g., `bnz` and `brc`) to realise reversible (conditional) branching.
- In the translation of **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$**fiq**, we have a pair of instructions `qif(q)` and `fiq(q)`, which indicate the creation and join of quantum branching, respectively. They will be used in the partial evaluation of quantum control flow and the final execution.
- The translations of procedure call $P(\overline{x})$ and declaration $P(\overline{u}) \Leftarrow C$ are inspired by their counterparts in classical reversible computing [7]. Here, the biggest difference is our design of the *automatic uncomputation* of classical variables, performed by the program uncp{C} at the end of procedure body C (see also Section 4.1.2), which reverses the changes on classical variables in C. The uncomputation uncp{·} is also recursively defined, where uncp{$P(\overline{x})$} and uncp{**qif** ... **fiq**} are set to empty, due to Conditions 2.2 and 2.3.

For illustration, on the RHS of Figure 8, we present an example of the high-to-mid-level translation of the quantum multiplexor program. For simplicity, we only show the translation of the recursive procedure $P(k, x)$. The full translation can be found in Appendix C.2.

4.3 Mid-Level to Low-Level Translation

Now we are ready to describe the last pass in which the mid-level program $\mathcal{P}_m$ obtained in the previous sections is translated into a program $\mathcal{P}_l$ in the low-level language **QINS** and thus executable on the quantum register machine. In this pass, instructions that take variables and labels as inputs will be translated to instructions that only take registers and immediate numbers as inputs. The additional instructions `push`, `pop` and `brc` also need to be translated. To do this, we need the load instructions `ld`, `ldr` and `fetr`. Let us use low{i} to denote the mid-to-low-level translation of an instruction i. In Figure 10, we present selected examples of the mid-to-low-level translation, and more details are shown in Appendix C.4. We further explain them as follows.

- The translation of `uni(U,q[x])` shows how to handle inputs containing subscripted quantum variables. We use $@x$ to denote the address of the name x (in the symbol table section of the QRAM; see Section 3.2.1). The word at $@x$ stores the address $\&x$ of the variable x (in the variable section). Lines 1–2 load the value of x into free register r_2. To obtain the address of $q[x]$, we add the address $\&q = \&(q[0])$ and the value of x, in Lines 3–4. Line 5 loads the value of $q[x]$ into free register r_4, on which the instruction `uni(U,`r_4`)` is executed. Lines 7–11 reverse the effects of Lines 1-5. Further details of the symbol table and memory allocation of variables can be found in Appendix C.3.

low{uni(U,q[x])}

```
1  ld(r1,@x)
2  ldr(r2,r1)
3  ld(r3,@q)
4  add(r3,r2)
5  ldr(r4,r3)
6  uni(U,r4)
7  ldr(r4,r3)
8  sub(r3,r2)
9  ld(r3,@q)
10 ldr(r2,r1)
11 ld(r1,@x)
```

low{bra(P.ent)}

```
1  ld(r1,@P.ent)
2  fetr(r2,r1)
3  ld(r1,@P.ent)
4  sub(r2,pc)
5  subi(r2,2)
6  swbr(r2)
7  addi(r2,2)
8  sub(r2,pc)
9  neg(r2)
10 ld(r1,@P.ent)
11 fetr(r2,r1)
12 ld(r1,@P.ent)
```

low{push(r)}

```
1 addi(sp,1)
2 ldr(r,sp)
```

low{pop(r)}

```
1 ldr(r,sp)
2 subi(sp,1)
```

low{add(x,1)}

```
1 ld(r1,@x)
2 ldr(r2,r1)
3 addi(r2,1)
4 ldr(r2,r1)
5 ld(r1,@x)
```

low{l0: bez(x,l1)}

```
1       ld(r1,@x)
2       ldr(r2,r1)
3  l0:  bez(r2,l1)
4       ldr(r2,r1)
5       ld(r1,@x)
```

low{l1: brc(x,l0)}

```
1       ld(r1,@x)
2       ldr(r2,r1)
3  l1:  bra(l0)
4       ldr(r2,r1)
5       ld(r1,@x)
```

Fig. 10. Selected examples of the mid-to-low-level translation. Here, all registers r_i are free registers.

- In the translation of `bra(P.ent)`, recall that the classical variable $P.ent$ corresponds to some procedure identifier P. Here, Lines 1–3 loads the value of $P.ent$ into free register r_2. Note that Line 2 uses `fetr` instead of `ldr` to preserve the copy of $P.ent$ in the QRAM for recursive procedure calls. Lines 4–5 calculate in r_2 the offset of $P.ent$ from the address of Line 6. When the branching occurs after Line 6, note that registers r_1 and r_2 are cleared. Lines 7–12 are similar.
- The translations of `push` and `pop` are rather simple. Note that they are reversible, e.g., if an element is pushed into the stack, the original register r will be cleared.
- The translations of `bez` and `brc` are related when they are used in pairs: they use the same free registers r_1 and r_2.

To end the compilation, we need to replace every label l in the compiled program by the offset of the address of where l is defined from where l is used. The compiled program is not yet loaded into the QRAM, but stored classically for later partial evaluation in Section 5. We present the example of mid-to-low-level translation of the quantum multiplexor program in Appendix C.4.

5 Partial Evaluation of Quantum Control Flow

At the end of the last section, a compiled program in the low-level language **QINS** is obtained. For its execution on the quantum register machine, we need to first perform a partial evaluation of quantum control flow to generate a data structure called qif table, to be loaded into the QRAM. In this section, we carefully describe this partial evaluation.

5.1 The Synchronisation Problem

Programs with only classical control flow can be straightforwardly executed without partial evaluation. However, for programs with quantum control flow, there is an obstruction known as the synchronisation problem [18, 54, 59, 61, 63, 64, 75, 86, 96] (see further discussion in Appendix I.1). In our case, it means in executing the statement $\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}$, C_0 and C_1 can take different numbers of instruction cycles to terminate. Consequently, the arrival times of two control flows (corresponding to two branches, in superposition) at the **fiq** are asynchronous, and hence they cannot be correctly merged into one control flow, in the same cycle. Another way to view the synchronisation problem is from the (QIF) rule in Figure 3. The problem occurs when $(C_0, \sigma, |\theta_0\rangle) \to^{k_0} (\downarrow, \sigma', |\theta_0'\rangle)$ and $(C_1, \sigma, |\theta_1\rangle) \to^{k_1} (\downarrow, \sigma', |\theta_1'\rangle)$ for some $k_0 \neq k_1$.

The synchronisation problem becomes more complicated for general quantum recursive programs. Note that C_0 and C_1 can further contain quantum recursion, and the number of nested

procedure calls involved cannot be determined before hand. The program might not even terminate. How to deal with the probably unbounded quantum recursion?

Our solution is by partial evaluation of the quantum control flow. When the classical inputs are given (while the quantum inputs remained unknown), we can check whether the compiled program $\mathcal{P}$ terminates in some practical (manually set) running time $T_{\text{prac}}(\mathcal{P})$. If the program terminates in $T_{\text{prac}}(\mathcal{P})$ cycles, for every **qif** statement, we can count the number of cycles for executing C_0 and C_1, as well as determine the structure of nested quantum branching induced by nested procedure calls. These can be gathered into a classical data structure called *qif table*, which will be used later in quantum superposition at runtime to synchronise two quantum branches in every **qif** statement. Note that this process is only dependent on the classical inputs but *independent* of the quantum inputs, and it does not change the static program text (compared to [96]). Also, our partial evaluation is different from those (e.g., [49, 53]) that aim at optimising the programs.

Along with generating the qif table, given the classical inputs, we can also determine the sizes of all arrays and allocate the addresses for variables (including determining the symbol table). This task is simple and we will not describe its details.

5.2 Qif Table

Now let us introduce the notion of qif table, storing the history information of quantum branching for an execution of the compiled program $\mathcal{P}$, within a given practical running time $T_{\text{prac}}(\mathcal{P})$. The qif table is a classical data structure that will be used *in quantum superposition* at runtime.

5.2.1 Nodes and Links in Qif Table.

Definition 5.1 (Qif table). A qif table is composed of linked nodes. There are two types of nodes in the qif table. Each node of type • represents an instantiation of **qif** ... **fiq**; i.e., an execution running through the `qif` to the corresponding `fiq` once. Nodes of type ∘ are ancilla nodes for the qif table to be *reversibly used*. Each node v of type • records the following information:

(1) (Next link $v.nx$): If v has a *continuing* non-nested instantiation v' of **qif** ... **fiq**, then $v.nx = v'$. Otherwise, we set $v.nx = v''$ for some node v'' of type ∘, enabling the qif table to be reversibly used (no matter whether v has a continuing instantiation of **qif** ... **fiq**) in later execution.

(2) (First children links $v.fc_i$) and (Last children links $v.lc_i$) for $i \in \{0, 1\}$: If v has *enclosed* nested instantiations of **qif** ... **fiq**, then $v.fc_0$ and $v.fc_1$ links to the first two children nodes, representing the first two enclosed instantiations of **qif** ... **fiq** (corresponding to branches $|0\rangle$ and $|1\rangle$ from v, respectively). Moreover, $v.lc_0$ and $v.lc_1$ links to the last two children nodes, which are the two next nodes (specified by the next link nx and of type ∘) of the last two enclosed instantiations of **qif** ... **fiq** (corresponding to branches $|0\rangle$ and $|1\rangle$ from v, respectively).
Otherwise, $v.fc_0 = v.lc_0 = v'$ and $v.fc_1 = v.lc_1 = v''$ for some nodes v', v'' of type ∘.

Further, each node v of either type • or ∘ records the following information:

(1) (Wait counter $v.w$): It stores the number of cycles to wait at node v.
(2) ($v.pr$): pr is the inverse link of nx. If $v.nx = v'$, then $v'.pr = v$.
(3) ($v.cf$): cf is the inverse link of fc_0 and fc_1. If $v.fc_0 = v_0$ and $v.fc_1 = v_1$, then $v_0.cf = v_1.cf = v$.
(4) ($v.cl$): cl is the inverse link of lc_0 and lc_1. If $v.lc_0 = v_0$ and $v.lc_1 = v_1$, then $v_0.cl = v_1.cl = v$.

In Figure 11, we give an example of a program and its corresponding qif table. We only show the links fc_i, lc_i and nx, and omit w, cf, cl and pr for simplicity of presentation. The partial evaluation should be done on the compiled program, but for clarity we present the original program written in **RQC**$^{++}$. We also show the correspondence between nodes in the qif table and the instantiations of **qif** statements in the program. It is easy to verify that the links are consistent with Definition 5.1.

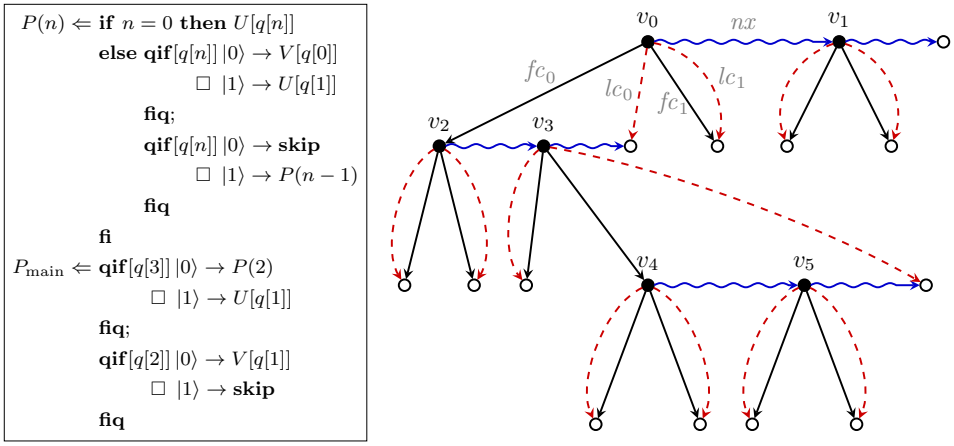


Fig. 11. Example of a program in $\mathbf{RQC}^{++}$ and its corresponding qif table. In the qif table, we only show the links fc_i (colored in black), lc_i (colored in red and dashed) and nx (colored in blue and squiggled), while w, cf, cl and pr are omitted for simplicity. The correspondence between the nodes on the RHS and the instantiations of **qif** statements on the LHS is as follows: (1) v_0: instantiation of the first **qif** ... **fiq** in P_{main}. (2) v_1: instantiation of the second **qif** ... **fiq** in P_{main}. (3) v_2 and v_4: the first and second instantiations of the first **qif** ... **fiq** in $P(n)$. (4) v_3 and v_5: the first and second instantiations of the second **qif** ... **fiq** in $P(n)$.

Additionally, we remark that to store the qif table in the QRAM, we need to encode all links and counter recorded at a node. The simplest way is to store them into a tuple, where links like $v.nx$ records the base address of the tuple of the corresponding node. Further discussion can be found in Appendices D.1 and D.2.

5.2.2 Generation of Qif Table. For a compiled program $\mathcal{P}$, we fix a practical running time $T_{\text{prac}}(\mathcal{P})$. The partial evaluation is performed by multiple parallel processes. We classically emulate the execution of the compiled program, neglecting all quantum inputs and unitary gates. Whenever a `qif` is met, the current process forks into two sub-processes, each continuing the evaluation of the corresponding quantum branch. Whenever a `fiq` is met, the current process waits for its pairing sub-process, and collects information from both sub-processes to merge into one process. Every process only goes into a single quantum branch and therefore contains no quantum superposition.

For each process, we maintain the following classical information. We have 6 system registers pc, ins, br, sp, v, t and a constant number of user registers. Here, v points to the current node in the qif table, and t is a counter that records the number of instructions already executed. We also have a classical memory M storing classical variables and the stack. Let M_i be the value stored at the memory location i.

The algorithm for partial evaluation of quantum control flow and generation of the qif table is presented as Algorithm 1. The major part of the function QEva is the loop between Lines 5–25, which consists of three stages that also appear in the execution in Section 6. The first and the last stages are similar to their classical counterparts. The handling of instructions `qif` or `fiq` in the stage **(Decode & Execute)** is highlighted. Algorithm 1 returns a timeout error if t exceeds the practical running time $T_{\text{prac}}(\mathcal{P})$. Otherwise, we obtain the actual running time $t = T_{\text{exe}}(\mathcal{P})$ for later use in Section 6. More detailed explanation of Algorithm 1 is provided in Appendix D.3.

Algorithm 1 Partial evaluation of quantum control flow and generation of qif table.

function QEvA
  Initialise $pc \leftarrow$ starting address of the compiled main program and $t \leftarrow 0$
  Create an initial node v
  while $t \leq T_{\text{prac}}(\mathcal{P})$ **do**
    (Fetch): Let $ins \leftarrow M_{pc}$ and $t \leftarrow t + 1$
    (Decode & Execute):
    if $ins =$ `qif(q)` **then** ▷ *creation of quantum branching*
      Create nodes v_0 and v_1. Set $v.fc_i, v.lc_i \leftarrow v_i$ and $v_i.cf, v_i.cl \leftarrow v$ ▷ *for the enclosed branches*
      Fork into two sub-processes QEvA$_0$ and QEvA$_1$. For QEvA$_i$, set $v \leftarrow v_i$
    else if $ins =$ `fiq(q)` **then** ▷ *join of quantum branching*
      Wait for the pairing sub-process QEvA$'$ with $v'.cl = v.cl = \hat{v}$ for some parent node $\hat{v}$
      $\hat{t} \leftarrow \max\{t, t'\}$, $v.w \leftarrow \hat{t} - t$ and $v'.w \leftarrow \hat{t} - t'$ ▷ *v', t' are corresponding v, t in QEvA$'$.*
      Merge with the pairing sub-process QEvA$'$ by letting $t \leftarrow \hat{t}$ and $v \leftarrow \hat{v}$
      Create node u. Set $v.nx \leftarrow u$ and $u.pr \leftarrow v$. ▷ *for the continuing branch*
      Suppose $v.cl = \hat{u}$ and $\hat{u}.lc_x = v$ for some $\hat{u}$ and x. Let $u.cl \leftarrow \hat{u}$, $\hat{u}.lc_x \leftarrow u$ and $v.cl \leftarrow 0$
      Update $v \leftarrow u$
    else if $ins =$ `finish` **then** ▷ *termination*
      return t
    else if $ins \notin \{$`uni(G,r)`, `unib(G,r1,r2)`$\}$ **then** ▷ *neglect quantum gates*
      Update registers and M according to Figure 5a
    (Branch):
    if $br \neq 0$ **then**
      Let $pc \leftarrow pc + br$
    else
      Let $pc \leftarrow pc + 1$
  return Timeout error

6 Execution on Quantum Register Machine

Now we are ready to describe how the compiled program $\mathcal{P}$ is executed, with the aid of partial evaluation results (including symbol table and qif table), on the quantum register machine. Let us load all these instructions and data into the QRAM, according to the layout described in Section 3.2.1.

6.1 Unitary U_{cyc} and Unitary U_{exe}

Algorithm 2 presents the execution on quantum register machine, which consists of repeated cycles, each performing the unitary U_{cyc}. We fix the number of repetitions to be $T_{\text{exe}} = T_{\text{exe}}(\mathcal{P})$, obtained from Algorithm 1.

In U_{cyc}, we need to decide whether to wait (i.e., skip the current cycle) or execute, according to the wait counter information stored in the current node of the qif table. To reversibly implement this procedure, U_{cyc} consists of three stages and exploits the registers *qifw* and *wait*. As a subroutine of U_{cyc}, the unitary U_{exe} consists of four stages, inspired by the design of classical reversible processor (e.g., [81]). In the **(Decode & Execute)** stage, the unitary U_{dec} is defined in Figure 5b. We provide more detailed discussion on Algorithm 2 and its visualisation as quantum circuits in Appendix E.1.

6.2 Unitaries for Executing Qif Instructions

It remains to define the unitaries U_{qif} and U_{fiq} that are unspecified in Figure 5a. We present their constructions in Algorithm 3. Additional remarks are as follows.

Algorithm 2 Execution on quantum register machine.

Unitary U_{main}
  Initialise registers according to Section 3.1
  for $t = 1, \ldots, T_{\text{exe}}$ **do**
    Apply the unitary U_{cyc} (defined below)

Unitary U_{cyc}
  (Set wait flag): Conditioned on *qifw*, set the wait flag in *wait*:
    Perform $\sum_{w,z} |w\rangle\langle w|_{qifw} \otimes |z \oplus [w > 0]\rangle\langle z|_{wait}$
  (Execute or wait): Conditioned on *wait*, apply the unitary U_{exe} (defined below), or wait and decrement the value in *qifw*:
    Perform $|0\rangle\langle 0|_{wait} \otimes U_{\text{exe}} + \sum_{z\neq 0,w} |z\rangle\langle z|_{wait} \otimes |w-1\rangle\langle w|_{qifw} \otimes \mathbb{1}$
  (Clear wait flag): Conditioned *qifw* and *qifv*, uncompute the wait flag in *wait*:
    Perform $\sum_{w,v,z} |w\rangle\langle w|_{qifw} \otimes |v\rangle\langle v|_{qifv} |z \oplus [w < v.w]\rangle\langle z|_{wait}$

Unitary U_{exe}
  (Fetch): Apply the unitary $U_{\text{fet}}(ins, pc, mem)$ ▷ U_{fet} *is defined in Definition 3.1.*
  (Decode & Execute): Apply the unitary U_{dec} ▷ U_{dec} *is defined in Figure 5b.*
  (Unfetch): Apply the unitary $U_{\text{fet}}(ins, pc, mem)$ again
  (Branch): Update *pc*, conditioned on *br*:
    Apply $U_+(pc, br)$ ▷ U_+ *performs the mapping* $|x\rangle\,|y\rangle \mapsto |x+y\rangle\,|y\rangle$.
    Apply $\circ(br)$-$\sum_x |x+1\rangle\langle x|_{pc}$ ▷ $\circ(\cdot)$-U *is defined in Section 3.3.*

Algorithm 3 The unitaries U_{qif} and U_{fiq} in Figure 5b.

Unitary $U_{\text{qif}}(q)$
  Conditioned on q, move *qifv* to its first children node in the qif table via the links fc_0 and fc_1; i.e., perform the following series of unitaries:
    $V_{fc} = \sum_{v,x,u} |v\rangle\langle v|_{qifv} \otimes |x\rangle\langle x|_q \otimes |u \oplus v.fc_x\rangle\langle u|_r$, where r is a free register
    $V_{cf} = \sum_{v,u} |v \oplus u.cf\rangle\langle v|_{qifv} \otimes |u\rangle\langle u|_r$
    $U_{\text{swap}}(r, qifv)$, which also clears register r ▷ U_{swap} *is defined in Section 3.3.*
  Update *qifw* with the wait counter information corresponding to *qifv*; i.e., perform:
    $\sum_{w,v} |v\rangle\langle v|_{qifv} \otimes |w \oplus v.w\rangle\langle w|_{qifw}$
Unitary $U_{\text{fiq}}(q)$
  Conditioned on q, move *qifv* to its parent node in the qif table via the inverse link *cl*; i.e., perform the following series of unitaries:
    $V_{cl} = \sum_{v,u} |v\rangle\langle v|_{qifv} \otimes |u \oplus v.cl\rangle\langle u|_r$, where r is a free register
    $V_{lc} = \sum_{v,x,u} |v \oplus u.lc_q\rangle\langle v|_{qifv} \otimes |x\rangle\langle x|_q \otimes |u\rangle\langle u|_r$
    $U_{\text{swap}}(r, qifv)$, which also clears register r
  Move *qifv* to the next node in the qif table via the link *nx*; i.e., perform the following series of unitaries:
    $V_{nx} = \sum_{v,u} |v\rangle\langle v|_{qifv} \otimes |u \oplus v.nx\rangle\langle u|_r$, where r is a free register
    $V_{pr} = \sum_{v,u} |v \oplus u.pr\rangle\langle v|_{qifv} \otimes |u\rangle\langle u|_r$
    $U_{\text{swap}}(r, qifv)$, which also clears register r

- For U_{qif}, note that we are promised that *qifw* is initially in state $|0\rangle$, because U_{qif} is used as a subroutine in U_{exe}, which will only be called by U_{cyc} when *qifw* is in state $|0\rangle$.
- The information fc_x, *cf*, lc_x, *cl*, *nx*, *pr* and *w* are stored in the qif node, and need to be fetched using U_{fet} into free registers before being used, of which details are omitted for simplicity.

Further explanation of Algorithm 3 is provided in Appendix E.2.

Now we remark on how the qif table as a classical data structure is used *in quantum superposition* during the execution on quantum register machine. Recall that at runtime, the value in register *qifv* indicates the current node in the qif table. Register *qifv* can be in a quantum superposition state, in particular, entangled with the quantum coin q (as well as other register and the QRAM) when instruction `qif(q)` is executed (see Algorithm 3). For example, after the unitary $U_{\text{qif}}(q)$ is performed, the state of the quantum register machine can be $\frac{1}{\sqrt{2}}|0\rangle_q|v_1\rangle_{qifv}|\psi_0\rangle + \frac{1}{\sqrt{2}}|1\rangle_q|v_2\rangle_{qifv}|\psi_1\rangle$, where $|\psi_0\rangle$ and $|\psi_1\rangle$ are states of the remaining quantum registers and the QRAM. In this way, the information in the qif table is used in quantum superposition.

7 Efficiency and Automatic Parallelisation

An implementation of quantum recursive programs has been presented in the previous sections. In this section, we analyse its efficiency, and further show that as a bonus, such implementation also offers *automatic parallelisation*. For implementing certain algorithmic subroutine, like the quantum multiplexor introduced in Section 1.1, we can even obtain exponential parallel speed-up (over the straightforward implementation) from this automatic parallelisation. This steps towards a *top-down* design of efficient quantum algorithms: we only need to design high-level quantum recursive programs, and let the machine automatically realise the parallelisation (whose quality, of course, still depends on the program structure). The intuition for the automatic parallelisation was already pointed out in Section 1.1: (1) with quantum control flow, the quantum register machine can go through quantum branches in superposition; and (2) with recursive procedure calls, the program can generate exponentially many quantum branches (as each instantiation of the **qif** statement creates two quantum branches).

In the following, we briefly describe the complexity of implementing quantum recursive programs; in particular, for partial evaluation and execution. We first describe the complexity in terms of elementary operations on registers and the QRAM, and then refine it into parallel time complexity measured by the standard (classical and quantum) circuit depth. The full analysis can be found in Appendices D.4, E.3 and F.1. Recall that, intuitively, $T_{\text{exe}}(\mathcal{P})$ correspond to the time for executing the longest quantum branch in program $\mathcal{P}$.

(1) Algorithm 1 takes $O(T_{\text{exe}}(\mathcal{P}))$ classical parallel elementary operations. Here, "elementary" means the operation only involves a constant number of memory locations in the classical RAM (as the partial evaluation is performed classically). "Parallel" means multiple elementary operations performed simultaneously are counted as one parallel elementary operation, like in the standard parallel computing. The intuition for this complexity is that in the partial evaluation, each of the classical parallel processes only evaluate one quantum branch.
(2) Algorithm 2 takes $O(T_{\text{exe}}(\mathcal{P}))$ quantum elementary operations, including on registers and QRAM accesses (see Definition 3.1). The intuition was already presented in Section 1.1.

The above complexities are in terms of elementary operations. As mentioned in Section 1, the implementation will be eventually quantum circuits, so we need to translate elementary operations into quantum circuits. The overall (classical and quantum) parallel time complexity of Algorithms 1 and 2 will be

$$O\big(T_{\text{exe}}(\mathcal{P}) \cdot \big(T_{\text{reg}} + T_{\text{QRAM}}\big)\big),$$

where T_{reg} and T_{QRAM} are parallel time complexities for elementary operations on registers and QRAM accesses, as aforementioned. Here, we assume that classical elementary operations are cheaper than their quantum counterparts.

For concreteness, let us return to the example of quantum multiplexor program $\mathcal{P}$ in Figure 1. Recall that in Figure 8 the programs after high-level transformations and high-to-mid-level translation are already presented. Now we provide a proof sketch of Theorem 1.1, whose full proof can be found in Appendix F.2.

Proof Sketch of Theorem 1.1. Since each C_x only consists of T_x quantum unitary gates, the number of instructions in the compiled program of $P[x] \Leftarrow C_x$ will be $O(T_x)$. As a result, the whole compiled quantum multiplexor program (presented in Appendix C.4) contains $\Theta\big(\sum_{x\in[N]} T_x\big)$ instructions. It is easy to verify that $T_{\text{exe}}(\mathcal{P}) = O\big(\max_{x\in[N]} T_x + n\big)$.

Let us determine T_{reg} and T_{QRAM} by implementing the quantum register machine in the more common quantum circuit model. We can calculate the size N_{QRAM} of the QRAM and the word length L_{word} for implementing $\mathcal{P}$. In particular, taking $N_{\text{QRAM}} = \Theta(\sum_x T_x)$ is sufficient. To see this, we can calculate that the sizes of the program, symbol table and variable sections are upper bounded by $\Theta(\sum_x T_x)$. The size of the qif table is $\Theta(2^n)$. The size of the stack is upper bounded by $\Theta(\sum_x T_x) + \Theta(n)$. To store an address in such QRAM, taking $L_{\text{word}} = \Theta\big(\log N_{\text{QRAM}}\big)$ is sufficient.

By lifting results from classical parallel circuits for elementary arithmetic [14, 62, 70], we have $T_{\text{reg}} = O\big(\log^2 L_{\text{word}}\big)$. By extending existing circuit QRAM constructions (e.g., [39, 41]), we have $T_{\text{QRAM}} = O\big(\log N_{\text{QRAM}} + \log L_{\text{word}}\big)$. The above calculations are carried out in terms of parallel time complexity, i.e., quantum circuit depth. Combining the above arguments leads to Theorem 1.1. □

8 Related Work

Low-level quantum instructions. Several quantum instruction set architectures have been proposed in the literature, e.g., OpenQASM [28], Quil [76], eQASM [37]. Only the architecture introduced in [96], called quantum control machine, supports program counter in superposition (and hence quantum control flow), and the others do not support quantum control flow at the instruction level. Quantum control machine also supports conditional jumps, but different from quantum register machine defined in this paper, it does not support arbitrary procedure calls.

Automatic parallelisation. Numerous efforts have been devoted to parallelisation of quantum circuits of specific patterns, e.g., [16, 27, 40, 46, 48, 60, 71, 77, 78, 80, 97, 98, 100, 101]. Other than the quantum circuit model, the measurement-based quantum computing [69] is also shown to provide certain benefits for parallelisation [22, 23, 29, 30, 50, 67]. These techniques of parallelisation are at the low level. In comparison, the automatic parallelisation from our implementation is at the high level: the quantum register machine automatically exploits parallelisation opportunities in the structures of the high-level quantum recursive programs.

Automatic uncomputation. Silq [21] is the first quantum programming language that supports automatic uncomputation, which was further investigated in [65, 66, 83, 85, 94, 95]. Silq's uncomputation is for quantum programs lifted from classical ones, or in their terminology, lifted functions (whose semantics can be described classically and preserves the input). Later works like [83, 85] also considered uncomputation of quantum programs but they do not support quantum recursion. In comparison, **RQC^{++}** supports quantum recursion, where classical variables are used solely for specifying the control (not data). The automatic uncomputation in our implementation is of these classical variables in quantum recursive programs.

Classical reversible languages. There are extensive works in classical reversible programming languages, including the high-level language Janus [58, 92, 93], low-level instruction set architectures PISA [8, 36, 84] and BobISA [81]. Some of these reversible languages support local variables, specified by a pair of local-delocal statements, which have explicitly reversible semantics. In **RQC^{++}**,

irreversible classical computation can be done on local variables, but their translations into low-level instructions become reversible.

Quantum control flow and data structures. Many works [3, 21, 24, 72, 85, 90, 95, 96] on quantum programming languages include quantum control flow as a feature. Some [24, 96] discuss the limitation of quantum control flow. For example, it is shown in [24] that the semantics of quantum recursion cannot be defined using Tarski's fixpoint theorem, when quantum measurements are involved. However, **RQC**$^{++}$ considered in this paper only describes unitary operators and therefore circumvents this issue. A similar unitary restriction is used in [96] to support instruction-level quantum control flow.

Another related topic is data structures in superposition [94, 95]. The language Tower in [94] can describe recursive programs in superposition, which allows a *single* layer of interleaving between quantum control flow and recursion. However, their syntax does not contain unitary gates and quantum if-statement, which cannot express the most general form of quantum recursion. In contrast, **RQC**$^{++}$ allows *arbitrary* interleaving between quantum if-statements and recursive procedure calls. Such expressive power of **RQC**$^{++}$ also makes our implementation non-trivial.

9 Conclusion

We propose the notion of quantum register machine, an architecture that provides instruction-level support for quantum control flow and recursive procedure calls at the same time. We design a comprehensive process of implementing quantum recursive programs on the quantum register machine, including compilation, partial evaluation of quantum control flow and execution. As a bonus, our implementation offers automatic parallelisation, from which we can even obtain exponential parallel speed-up (over the straightforward implementation) for implementing some important quantum algorithmic subroutines like the quantum multiplexor.

To conclude this paper, let us list several topics for future research. Firstly, an immediate next step is to develop a software that realises our implementation of quantum recursive programs for actual execution on future quantum hardware. Moreover, one can consider certifying such software implementation, like in recent verified quantum compilers, e.g., [5, 43, 53, 68, 79]. Secondly, our implementation is designed to be simple for clarity. It is worth extending the features of the quantum register machine and further optimise the steps in the compilation, partial evaluation and execution. Thirdly, it is interesting to see what other quantum algorithms (except those considered in [91] and this paper) can be written in quantum recursive programs and benefit (with possible speed-up) from the efficient implementation of the quantum register machine.

Acknowledgments

Zhicheng Zhang thanks Qisheng Wang for helpful discussions about the halting schemes of quantum Turing machine related to the synchronisation problem in Section 5.1 (see also Appendix I.1). We thank the anonymous reviewers for their valuable comments. This work was partly supported by the Australian Research Council (Grant Number: DP250102952). Zhicheng Zhang was supported by the Sydney Quantum Academy, NSW, Australia.

References

[1] Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. 2020. On the quantum complexity of closest pair and related problems. In *35th Computational Complexity Conference (CCC 2020)*, Vol. 169. 16:1–16:43. doi:10.4230/LIPIcs.CCC.2020.16

[2] V. Aho Alfred, S. Lam Monica, Ravi Sethi, and D. Ullman Jeffrey. 2007. *Compilers: principles, techniques & tools.* Pearson Education.

[3] Thorsten Altenkirch and Jonathan Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. 249–258. doi:10.1109/LICS.2005.1

[4] Andris Ambainis. 2007. Quantum walk algorithm for element distinctness. *SIAM J. Comput.* 37, 1 (2007), 210–239. doi:10.1137/S0097539705447311

[5] Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. *Verified compilation of space-efficient reversible circuits*. 3–21. doi:10.1007/978-3-319-63390-9_1

[6] Israel F. Araujo, Daniel K. Park, Francesco Petruccione, and Adenilton J. da Silva. 2021. A divide-and-conquer algorithm for quantum state preparation. *Scientific reports* 11, 1 (2021), 6329. doi:10.1038/s41598-021-85474-1

[7] Holger Bock Axelsen. 2011. Clean translation of an imperative reversible programming language. In *International Conference on Compiler Construction*. 144–163. doi:10.1007/978-3-642-19861-8_9

[8] Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. 2007. Reversible machine code and its abstract processor architecture. In *Computer Science–Theory and Applications: Second International Symposium on Computer Science in Russia (CSR 2007)*. 56–69. doi:10.1007/978-3-540-74510-5_9

[9] Ryan Babbush, Dominic W. Berry, Ian D. Kivlichan, Annie Y. Wei, Peter J. Love, and Alán Aspuru-Guzik. 2016. Exponentially more precise quantum simulation of fermions in second quantization. *New Journal of Physics* 18 (2016), 033032. doi:10.1088/1367-2630/18/3/033032

[10] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. 2018. Encoding electronic spectra in quantum circuits with linear T complexity. *Physical Review X* 8, 4 (2018), 041015. doi:10.1103/PhysRevX.8.041015

[11] Ryan Babbush, Nathan Wiebe, Jarrod McClean, James McClain, Hartmut Neven, and Garnet Kin-Lic Chan. 2018. Low-depth quantum simulation of materials. *Physical Review X* 8 (2018), 011044. Issue 1. doi:10.1103/PhysRevX.8.011044

[12] John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM* 3, 5 (1960), 299–311. doi:10.1145/367236.367262

[13] John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. 1963. Revised report on the algorithmic language ALGOL 60. *Commun. ACM* 6, 1 (1963), 1–17. doi:10.1145/366193.366201

[14] Paul W. Beame, Stephen A. Cook, and H. James Hoover. 1986. Log depth circuits for division and related problems. *SIAM J. Comput.* 15, 4 (1986), 994–1003. doi:10.1109/SFCS.1984.715894

[15] Charles H. Bennett. 1973. Logical reversibility of computation. *IBM Journal of Research and Development* 17, 6 (1973), 525–532. doi:10.1147/rd.176.0525

[16] Debajyoti Bera, Frederic Green, and Steven Homer. 2007. Small depth quantum circuits. *ACM SIGACT News* 38, 2 (2007), 35–50. doi:10.1145/1272729.1272739

[17] Daniel J. Bernstein, Stacey Jeffery, Tanja Lange, and Alexander Meurer. 2013. Quantum algorithms for the subset-sum problem. In *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013*. 16–33. doi:10.1007/978-3-642-38616-9_2

[18] Ethan Bernstein and Umesh Vazirani. 1993. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 11–20. doi:10.1145/167088.167097

[19] Dominic W. Berry, Andrew M. Childs, Richard Cleve, Robin Kothari, and Rolando D. Somma. 2015. Simulating Hamiltonian dynamics with a truncated Taylor series. *Physical Review Letters* 114 (2015), 090502. Issue 9. doi:10.1103/PhysRevLett.114.090502

[20] Dominic W. Berry, Andrew M. Childs, and Robin Kothari. 2015. Hamiltonian simulation with nearly optimal dependence on all parameters. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS '15)*. 792–809. doi:10.1109/FOCS.2015.54

[21] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–300. doi:10.1145/3385412.3386007

[22] Anne Broadbent and Elham Kashefi. 2009. Parallelizing quantum circuits. *Theoretical computer science* 410, 26 (2009), 2489–2510. doi:10.1016/j.tcs.2008.12.046

[23] Dan Browne, Elham Kashefi, and Simon Perdrix. 2011. Computational depth complexity of measurement-based quantum computation. In *Theory of Quantum Computation, Communication, and Cryptography: 5th Conference, TQC 2010*. 35–46. doi:10.1007/978-3-642-18073-6_4

[24] Costin Bădescu and Prakash Panangaden. 2015. Quantum alternation: prospects and problems. In *Proceedings of the 12th International Workshop on Quantum Physics and Logic (EPTCS, Vol. 195)*. 33–42. doi:10.4204/EPTCS.195.3

[25] Andrew M. Childs, Robin Kothari, and Rolando D. Somma. 2017. Quantum algorithm for systems of linear equations with exponentially improved dependence on precision. *SIAM J. Comput.* 46, 6 (2017), 1920–1950. doi:10.1137/16M1087072

[26] Andrew M. Childs and Nathan Wiebe. 2012. Hamiltonian simulation using linear combinations of unitary operations. *Quantum Information & Computation* 12, 11–12 (2012), 901–924. doi:10.26421/QIC12.11-12-1

[27] Richard Cleve and John Watrous. 2000. Fast parallel circuits for the quantum Fourier transform. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS '00)*. 526–536. doi:10.1109/SFCS.2000.892140

[28] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. arXiv:1707.03429 [quant-ph]

[29] Raphael Dias da Silva, Einar Pius, and Elham Kashefi. 2013. Global quantum circuit optimization. arXiv:1301.0351 [quant-ph]

[30] Vincent Danos, Elham Kashefi, and Prakash Panangaden. 2007. The measurement calculus. *Journal of the ACM (JACM)* 54, 2 (2007), 8–es. doi:10.1145/1219092.1219096

[31] Haowei Deng, Runzhou Tao, Yuxiang Peng, and Xiaodi Wu. 2024. A case for synthesis of recursive quantum unitary programs. *Proceedings of the ACM on Programming Languages* 8 (2024), 1759–1788. doi:10.1145/3632901

[32] David Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400, 1818 (1985), 97–117. doi:10.1098/rspa.1985.0070

[33] Edsger W. Dijkstra. 1960. Recursive programming. *Numer. Math.* 2, 1 (1960), 312–318. doi:10.1007/bf01386232

[34] Edsger W. Dijkstra. 1970. Notes on structured programming. (1970). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF circulated privately.

[35] Edsger W. Dijkstra. 1979. Programming considered as a human activity. In *Classics in software engineering*. 1–9. https://www.cs.utexas.edu/~EWD/ewd01xx/EWD117.PDF

[36] Michael Patrick Frank. 1999. *Reversibility for efficient computing*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[37] X. Fu, L. Riesebos, M. A. Rol, Jeroen van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2019. eQASM: An executable quantum instruction set architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 224–237. doi:10.1109/HPCA.2019.00040

[38] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Architectures for a quantum random access memory. *Physical Review A* 78, 5 (2008), 052310. doi:10.1103/PhysRevA.78.052310

[39] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Quantum random access memory. *Physical Review Letters* 100, 16 (2008), 160501. doi:10.1103/PhysRevLett.100.160501

[40] Frederic Green, Steven Homer, Cristopher Moore, and Christopher Pollett. 2002. Counting, fanout and the complexity of quantum ACC. *Quantum Information & Computation* 2, 1 (2002), 35–65. doi:10.26421/QIC2.1-3

[41] Connor T. Hann, Gideon Lee, S. M. Girvin, and Liang Jiang. 2021. Resilience of quantum random access memory to generic noise. *PRX Quantum* 2, 2 (2021), 020311. doi:10.1103/PRXQuantum.2.020311

[42] Connor T. Hann, Chang-Ling Zou, Yaxing Zhang, Yiwen Chu, Robert J. Schoelkopf, Steven M. Girvin, and Liang Jiang. 2019. Hardware-efficient quantum random access memory with hybrid quantum acoustic systems. *Physical Review Letters* 123, 25 (2019), 250501. doi:10.1103/PhysRevLett.123.250501

[43] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434318

[44] Charles Antony Richard Hoare. 1961. Algorithm 64: quicksort. *Commun. ACM* 4, 7 (1961), 321. doi:10.1145/366622.366644

[45] Charles Antony Richard Hoare. 1975. Recursive data structures. *International Journal of Computer & Information Sciences* 4, 2 (1975), 105–132. doi:10.1007/BF00976239

[46] Peter Høyer and Robert Špalek. 2005. Quantum fan-out is powerful. *Theory of Computing* 1, 5 (2005), 81–103. doi:10.4086/toc.2005.v001a005

[47] Samuel Jaques and Arthur G. Rattew. 2023. QRAM: A survey and critique. arXiv:2305.10310 [quant-ph]

[48] Jiaqing Jiang, Xiaoming Sun, Shang-Hua Teng, Bujiao Wu, Kewen Wu, and Jialin Zhang. 2020. Optimal space-depth trade-off of CNOT circuits in quantum logic synthesis. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '20)*. 213–229. doi:10.1137/1.9781611975994.13

[49] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

[50] Richard Jozsa. 2005. An introduction to measurement based quantum computation. arXiv:quant-ph/0508124 [quant-ph]

[51] Robin Kothari. 2014. *Efficient algorithms in quantum query complexity*. Ph. D. Dissertation. University of Waterloo.

[52] Rolf Landauer. 1961. Irreversibility and heat generation in the computing process. *IBM journal of research and development* 5, 3 (1961), 183–191. doi:10.1147/rd.53.0183

[53] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified compilation of quantum oracles. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 589–615. doi:10.1145/3563309
[54] Noah Linden and Sandu Popescu. 1998. The halting problem for quantum computers. arXiv:quant-ph/9806054 [quant-ph]
[55] Chenxu Liu, Meng Wang, Samuel A. Stein, Yufei Ding, and Ang Li. 2023. Quantum memory: a missing piece in quantum computing units. arXiv:2309.14432 [quant-ph]
[56] Guang Hao Low, Vadym Kliuchnikov, and Luke Schaeffer. 2024. Trading T gates for dirty qubits in state preparation and unitary synthesis. *Quantum* 8 (2024), 1375. doi:10.22331/q-2024-06-17-1375
[57] Guang Hao Low and Nathan Wiebe. 2019. Hamiltonian simulation in the interaction picture. arXiv:1805.00675 [quant-ph]
[58] Christopher Lutz and Howard Derby. 1986. Janus: a time-reversible language. *Letter to Rolf Landauer* 2 (1986).
[59] Takayuki Miyadera and Masanori Ohya. 2005. On halting process of quantum turing machine. *Open Systems & Information Dynamics* 12, 3 (2005), 261–264. doi:10.1007/s11080-005-0923-2
[60] Cristopher Moore and Martin Nilsson. 2002. Parallel quantum computation and quantum codes. *SIAM J. Comput.* 31, 2 (2002), 799–815. doi:10.1137/S0097539799355053
[61] John M. Myers. 1997. Can a universal quantum computer be fully quantum? *Physical Review Letters* 78, 9 (1997), 1823. doi:10.1103/PhysRevLett.78.1823
[62] Yu Ofman. 1963. On the algorithmic complexity of discrete functions. In *Sov. Math. Dokl.*, Vol. 7. 589.
[63] Masanao Ozawa. 1998. Quantum nondemolition monitoring of universal quantum computers. *Physical Review Letters* 80, 3 (1998), 631. doi:10.1103/PhysRevLett.80.631
[64] Masanao Ozawa. 1998. Quantum Turing machines: local transition, preparation, measurement, and halting. In *Quantum Communication, Computing, and Measurement 2*. 241–248. doi:10.1007/0-306-47097-7_32
[65] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: synthesizing uncomputation in quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 222–236. doi:10.1145/3453483.3454040
[66] Anouk Paradis, Benjamin Bichsel, and Martin Vechev. 2024. Reqomp: space-constrained uncomputation for quantum circuits. *Quantum* 8 (2024), 1258. doi:10.22331/q-2024-02-19-1258
[67] Einar Pius. 2010. *Automatic parallelisation of quantum circuits using the measurement based quantum computing model*. Master's thesis. University of Edinburgh.
[68] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: reasoning about reversible quantum circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (2019), 299–312. doi:10.4204/eptcs.287.17
[69] Robert Raussendorf and Hans J. Briegel. 2001. A one-way quantum computer. *Physical Review Letters* 86, 22 (2001), 5188. doi:10.1103/PhysRevLett.86.5188
[70] John H. Reif. 1986. Logarithmic depth circuits for algebraic functions. *SIAM J. Comput.* 15, 1 (1986), 231–242. doi:10.1137/0215017
[71] Gregory Rosenthal. 2023. Query and depth upper bounds for quantum unitaries via Grover search. arXiv:2111.07992 [quant-ph]
[72] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From symmetric pattern-matching to quantum control. In *Foundations of Software Science and Computation Structures: 21st International Conference, FOSSACS 2018*. 348–364. doi:10.1007/978-3-319-89366-2_19
[73] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586. doi:10.1017/S0960129504004256
[74] Vivek V. Shende, Stephen S. Bullock, and Igor L. Markov. 2005. Synthesis of quantum logic circuits. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. 272–275. doi:10.1109/TCAD.2005.855930
[75] Yu Shi. 2002. Remarks on universal quantum computer. *Physics Letters A* 293, 5-6 (2002), 277–282. doi:10.1016/S0375-9601(02)00015-4
[76] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2017. A practical quantum instruction set architecture. arXiv:1608.03355 [quant-ph]
[77] Xiaoming Sun, Guojing Tian, Shuai Yang, Pei Yuan, and Shengyu Zhang. 2023. Asymptotically optimal circuit depth for quantum state preparation and general unitary synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 10 (2023), 3301–3314. doi:10.1109/TCAD.2023.3244885
[78] Yasuhiro Takahashi and Seiichiro Tani. 2013. Collapse of the hierarchy of constant-depth exact quantum circuits. In *Proceedings of the 28th IEEE Conference on Computational Complexity*. 168–178. doi:10.1109/CCC.2013.25
[79] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: Push-button verification for the Qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 641–656. doi:10.1145/3519939.3523431

[80] Barbara M. Terhal and David P. DiVincenzo. 2004. Adptive quantum computation, constant depth quantum circuits and arthur-merlin games. *Quantum Information & Computation* 4, 2 (2004), 134–145. doi:10.26421/QIC4.2-5
[81] Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück. 2012. A reversible processor architecture and its reversible logic design. In *Reversible Computation: Third International Workshop, RC 2011*. 30–42. doi:10.1007/978-3-642-29517-1_3
[82] Gauthier van den Hove. 2015. On the origin of recursive procedures. *Comput. J.* 58, 11 (2015), 2892–2899. doi:10.1093/comjnl/bxu145
[83] Hristo Venev, Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2024. Modular synthesis of efficient quantum uncomputation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2097–2124. doi:10.1145/3689785
[84] Carlin James Vieri. 1999. *Reversible computer engineering and architecture.* Ph. D. Dissertation. Massachusetts Institute of Technology.
[85] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A unified language for quantum and classical computing. *Proceedings of the ACM on Programming Languages* 7 (2023), 921–951. doi:10.1145/3571225
[86] Qisheng Wang and Mingsheng Ying. 2023. Quantum random access stored-program machines. *J. Comput. System Sci.* 131 (2023), 13–63. doi:10.1016/j.jcss.2022.08.002
[87] Shifan Xu, Alvin Lu, and Yongshan Ding. 2025. Fat-tree QRAM: A high-bandwidth shared quantum random access memory for parallel queries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25, Vol. 2)*. 390–406. doi:10.1145/3676641.3716256
[88] Zhaowei Xu, Mingsheng Ying, and Benoît Valiron. 2021. Reasoning about recursive quantum programs. arXiv:2107.11679 [cs.LO]
[89] Mingsheng Ying. 2016. *Foundations of quantum programming.* Morgan Kaufmann. doi:10.1016/C2014-0-02660-3
[90] Mingsheng Ying, Nengkun Yu, and Yuan Feng. 2012. Defining quantum control flow. arXiv:1209.4379 [quant-ph]
[91] Mingsheng Ying and Zhicheng Zhang. 2024. Verification of recursively defined quantum circuits. arXiv:2404.05934 [quant-ph]
[92] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers*. 43–54. doi:10.1145/1366230.1366239
[93] Tetsuo Yokoyama and Robert Glück. 2007. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. 144–153. doi:10.1145/1244381.1244404
[94] Charles Yuan and Michael Carbin. 2022. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 259–288. doi:10.1145/3563297
[95] Charles Yuan and Michael Carbin. 2024. The T-complexity costs of error correction for control flow in quantum computation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 492–517. doi:10.1145/3656397
[96] Charles Yuan, Agnes Villanyi, and Michael Carbin. 2024. Quantum control machine: The limits of control flow in quantum programming. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1–28. doi:10.1145/3649811
[97] Pei Yuan and Shengyu Zhang. 2023. Optimal (controlled) quantum state preparation and improved unitary synthesis by quantum circuits with any number of ancillary qubits. *Quantum* 7 (2023), 956. doi:10.22331/q-2023-03-20-956
[98] Xiao-Ming Zhang, Tongyang Li, and Xiao Yuan. 2022. Quantum state preparation with optimal circuit depth: Implementations and applications. *Physical Review Letters* 129, 23 (2022), 230504. doi:10.1103/PhysRevLett.129.230504
[99] Xiao-Ming Zhang and Xiao Yuan. 2024. Circuit complexity of quantum access models for encoding classical data. *npj Quantum Information* 10, 1 (2024), 42. doi:10.1038/s41534-024-00835-8
[100] Xiao-Ming Zhang, Man-Hong Yung, and Xiao Yuan. 2021. Low-depth quantum state preparation. *Physical Review Research* 3, 4 (2021), 043200. doi:10.1103/PhysRevResearch.3.043200
[101] Zhicheng Zhang, Qisheng Wang, and Mingsheng Ying. 2024. Parallel quantum algorithm for hamiltonian simulation. *Quantum* 8 (2024), 1228. doi:10.22331/q-2024-01-15-1228
[102] Zhicheng Zhang and Mingsheng Ying. 2024. Quantum register machine: efficient implementation of quantum recursive programs. arXiv:2408.10054 [quant-ph]

Received 2024-11-12; accepted 2025-03-06