



# **Compilation of Quantum Programs with Control Flow Primitives in Superposition**

**Master Thesis**

**05.02.25**

**Sascha Thiemann**

**Supervisors: apl. Prof. Dr. Thomas Noll**

**Prof. Dr. rer. nat. Dominique Unruh**

# Outline

---

## Introduction

## Background

- Quantum Control Flow
- Quantum Control Machine

## Language

- Overview
- Syntax
- Translation

## Implementation

- Lexical and Syntactic Analysis
- Semantic Analysis
- Code Generation
- Optimization

## Evaluation

## Conclusion

---

## Introduction

- Idea of Quantum control flow first used by [Altenkirch and Grattage, 2005].
  - Later, it was defined by [Ying et al., 2012].
  - Most languages focus on classical control flow.
  - Only recently, Quantum Control Machine, proposed by [Yuan et al., 2024].
    - An instruction set architecture.
    - Allows for quantum control flow with an assembly-like syntax.
    - Low-level design of the architecture and syntax → complex programs.
- ⇒ Build upon the ideas, define a high-level quantum language with quantum control flow.
- ⇒ Plan to implement a compiler for it.

## Quantum Control Flow

- First used by [Altenkirch and Grattage, 2005] to define functional programming language.
- For example, used to define Hadamard gate as function *had*:

$$\begin{aligned} \textit{had} &: Q \rightarrow Q \\ \textit{had} : x &\mapsto \text{if}^\circ x \\ &\quad \text{then } \{ \textit{false} \mid \neg \textit{true} \} \\ &\quad \text{else } \{ \textit{false} \mid \textit{true} \} \end{aligned}$$

- Later, formally defined by [Ying et al., 2012].
- Allows for the execution of functions based on values in superposition.
- Results in superposition of the results of individual executions.

## Limitations

- Mainly limited by two principles: *reversibility* and *synchronization*.
- Instructions required to be reversible, as they are unitary transformations.
  - ⇒ Control flow, as implemented in classical computers, not possible.
    - Landauer Embedding [Landauer, 1961] seems to offer solution.
    - Can make non-reversible function reversible by outputting input value.
    - However, output includes program history and result depends on the history → become entangled.
    - Leads to disruptive entanglement, invalid results [Yuan et al., 2024].
- Program counter may become entangled with data.
  - ⇒ Disruptive entanglement.
    - The principle of synchronization: *Control flow must become independent from data*.
    - Program may include padding operations to ensure synchronization.
  - ⇒ Loops cannot depend solely on value in superposition.
  - ⇒ Instead, a loop must be bounded by classical value [Yuan et al., 2024].

## Quantum Control Machine

- Quantum Control Machine (QCM), proposed by [Yuan et al., 2024]
- Focused on quantum control flow.
- Syntax and logic similar to classical assembly language
- Employs a branch control register *bcr* to enable reversible jumps.
- Instead of increasing IP by 1 after statement, increased by value of *bcr*.
- The *bcr* can then be reversibly modified.
  - To jump by 5, the *bcr* increased by 5
  - At its destination, decreased by 5 again.

# Background

---

## Instructions

- Here, some instructions of the QCM are listed.
- For every instruction there exists a reversed instruction, e.g., `radd` is the subtraction operation.

| Operation         | Syntax                          | Semantics <sup>1</sup>  |
|-------------------|---------------------------------|---|
| No-op             | <code>nop</code>                | Only increases instruction pointer by the <i>bcr</i> .                    |
| Addition          | <code>add <i>ra rb</i></code>   | Adds register <i>rb</i> to <i>ra</i> .                                    |
| Multiplication    | <code>mul <i>ra rb</i></code>   | Multiplies register <i>ra</i> by <i>rb</i> .                              |
| Jump              | <code>jmp <i>p</i></code>       | Increases <i>bcr</i> by <i>p</i> .  |
| Conditional Jumps | <code>jz <i>p ra</i></code>     | Increases <i>bcr</i> by <i>p</i> if <i>ra</i> is 0.                       |
|                   | <code>jne <i>p ra rb</i></code> | Increases <i>bcr</i> by <i>p</i> if <i>ra</i> is not equal to <i>rb</i> . |

<sup>1</sup> After all operations, the instruction pointer is increased by the value of the *bcr*.

An excerpt of the QCM instruction set with instructions used in later examples.

# Background

---

## (Non-) Reversible Example

- Example of a classical program and the reversible equivalent.
- Both programs calculate  $x^y$ .
- Neither are synchronized because loop is not bound by classical value.

```
1      add    res $1
2      add    r1  y
3  l1:  jz     l2  r1
4      mul    res x
5      radd   r1  $1
6      jmp    l1
7  l2:  nop
```

A non-reversible exponentiation algorithm.

```
1      add    res $1
2      add    r1  y
3  l1:  rjne   r11 r1  y
4  r12: jz     l2  r1
5      mul    res x
6      radd   r1  $1
7  r11: jmp    l1
8  l2:  rjmp   l2
```

A reversible exponentiation algorithm.



# Background

---

## Reversible Synchronized Example

- Synchronized implementation, calculating  $x^{\min y, \max}$ .
- *max* is a classical bound for loop.
- Line 9 includes padding instruction.

```
1      add    res    $1
2      add    r1     max
3  l1:    rjne   r11   r1    max
4  r12:    jz     l2     r1
5  r13:    jg     l3     r1    y
6      mul    res    x
7  r14:    jmp    l4
8  l3:    rjmp   r13
9      nop                    ; padding
10 l4:    rjle   r14    r1    y
11      radd   r1     $1
12 r11:    jmp    l1
13 l2:    rjmp   r12
```

A synchronized, reversible exponentiation algorithm.

## Language Overview

- Idea for language: Provide high-level language with the capabilities of the QCM.
- Want to remove low-level concepts, add high-level ones.
- Jump instructions in superposition are removed → need to add other control flow primitives.
- Introduce multiple high-level concepts and control flow statements:
  - blocks and scopes,
  - different data types,
  - composite gates,
  - loop statements, unrolled at compile time, and
  - quantum if- and else-statements.

# Language

## Syntax

- Define  $CFG_{Luie}$  for our language.
- Start symbol is the program, consisting of gate declarations and a block.
- A block is a list of translatables, either statements or declarations.

$$\begin{aligned} CFG_{Luie} &= (V_{Luie}, \Sigma_{Luie}, R_{Luie}, prg_{Luie}) \\ V_{Luie} &= \{exp, rExp, gate, prg_{Luie}, \dots\} \\ \Sigma_{Luie} &= \{.., range, (, ), \dots\} \end{aligned}$$

$$\begin{aligned} Program : prg_{Luie} & ::= gDcl_1 \dots gDcl_n blk \mid \\ & blk \\ Block : blk & ::= t_1 \dots t_n \mid \\ & \epsilon \end{aligned}$$

```
1  /* Gate Declaration */
2  gate c_h_reg(control, reg) do
3      /* Block Start */
4
5      qif control do
6          for i in range(sizeof(reg)) do
7              h reg[i];
8          end
9      end
10
11     /* Block End */
12 end
13
14 /* Block Start */
15 const regSize : int = 3; // Decl.
16 qubit c;                // Decl.
17 qubit[regSize] a;        // Decl.
18 c_h_reg c, a;
19 /* Block End */
```

Luie program with structure highlighted.

## Syntax

- Three different statements: quantum if-statement, a loop statement, gate application.
- To combine qubit or register access: qubit argument.
- Expressions for numeric values or ranges.
- Fix set of defined gates  $\rightarrow$  differentiate translations.

*Statement* : *stm* ::= *qif* *qArg* *do blk end* |  
                  *for id in rExp do blk end* |  
                  *id qArg*<sub>1</sub>, ..., *qArg*<sub>n</sub>;  
*QubitArg* : *qArg* ::= *id* | *id[exp]*

```
1 gate c_h_reg(control, reg) do
2   /* If-Statement */
3   qif control do
4     /* Loop Stat. |  $\nabla$  Expression */
5     for i in range(sizeof(reg)) do
6       /*  $\uparrow$  Range Exp. */
7       h reg[i];
8       /*  $\uparrow$  Qubit Argument */
9     end
10  end
11 end
12
13 const regSize : int = 3;
14 qubit c;
15 qubit[regSize] a;
16 /* Composite Gate Application */
17 c_h_reg c, a;
18   /*  $\uparrow$   $\uparrow$  Qubit Argument */
```

Luie program with statements and arguments highlighted.

## Symbol Table

- Saves symbol information relevant for translation.
- Contains four types of symbols:
  1. named constants,
  2. quantum registers and qubits,
  3. qubit arguments, and
  4. composite gates.

$$\begin{aligned} \textit{SymbolTable} := \{st \mid st : \textit{Identifier} \dashrightarrow & (\{\textit{const}\} \times \mathbb{Q}) \\ & \cup (\{\textit{qubit}\} \times \mathbb{N} \times \textit{Identifier}) \\ & \cup (\{\textit{arg}\} \times \textit{QubitArgument}) \\ & \cup (\{\textit{gate}\} \times \textit{Block} \times \textit{Identifier}^+)\} \end{aligned}$$

## Translation Function and Block Translation

- *trans* function translates program.
- Symbol table updated with gate declaration by *up* function.
- Block translation *bt* translates all translatable.
- Updates symbol table used in next translation.

```
trans(gate c_h_reg...) = OPENQASM 3.0;  
                        include "stdgates.inc";  
                        bt(const regSize = 3; ..., up(stε, gate c_h_reg...))  
st1 = up(stε, gate c_h_reg...) = [c_h_reg ↦ (gate, qif control do ... end,  
                                         control, reg)]
```

```
bt(const regSize = 3; ..., st1) = tr1    where (tr1, st2) = tt(const regSize = 3; , st1)  
                                tr2    where (tr2, st3) = tt(qubit c; , st2)  
                                tr3    where (tr3, st4) = tt(qubit[regSize] a; , st3)  
                                tr4    where (tr4, -) = tt(c_h_reg c, a; , st4)
```

# Implementation

---

## Overview

- Four compilation stages:
  1. the lexical and syntactic analysis,
  2. semantic analysis,
  3. code generation, and
  4. optimizations
- Process managed by a static compiler class.
  - Parses command line parameters.
  - Handles input and output of files.
  - Calls the different stages.
  - Handles logging and error messages.

```
1 ./LUIECompiler --input "./program.luie" \
2                  --output "./build/program.qasm" \
3                  --optimization nullgate+peepingcontrol
```

A command line interface example.

## Lexical and Syntactic Analysis

- First compilation stage is lexical and syntactic analysis.
- Both lexer and parser created with the ANTLR4 tool.
- Generates the source code based on a given grammar.
- Implementation of the grammar is more elaborate version of previously disused one.

```
1 parse      : mainblock EOF;  
2  
3 mainblock  : gateDeclaration* (declaration | statement)*;  
4  
5 block      : (declaration | statement)*;
```

The basic structure of parsing rules for Luie.



## Semantic Analysis

- Analyses non-syntactic constraints of program, mainly:
  1. declaration analysis and
  2. type checking.
- Declaration analysis ensures all identifiers used were previously declared and all identifiers used in declarations are not already declared.
- Type checking ensures that symbols are used in the correct context.

```
1 qubit c;           /* ↙ Type error (cannot use qubits in arith. expression) */
2 const regSize : int = c + 2;
3                   /* ↙ Already declared */
4 qubit[regSize] c;
5                   /* ↙ Type error (expects a register or qubit) */
6 c_h_reg c, a, regSize;
7                   /* ↗ Undeclared identifier */
```

Luise program with semantic errors highlighted.

# Implementation

---

## Code Generation

- Parse tree is traversed and source code is translated to in-memory representation.
- Source code representation (SCR) is translated to target code representation (TCR).
- TCR can be translated directly to the textual OpenQASM code.
- Example code generation with program:

```
1 gate c_h_reg(control, reg) do
2     qif control do
3         for i in range(sizeof(reg)) do
4             h reg[i];
5         end
6     end
7 end
8
9 qubit c;
10 qubit[3] a;
11 c_h_reg c, a;
```

An example Luie program to show the code generation process.

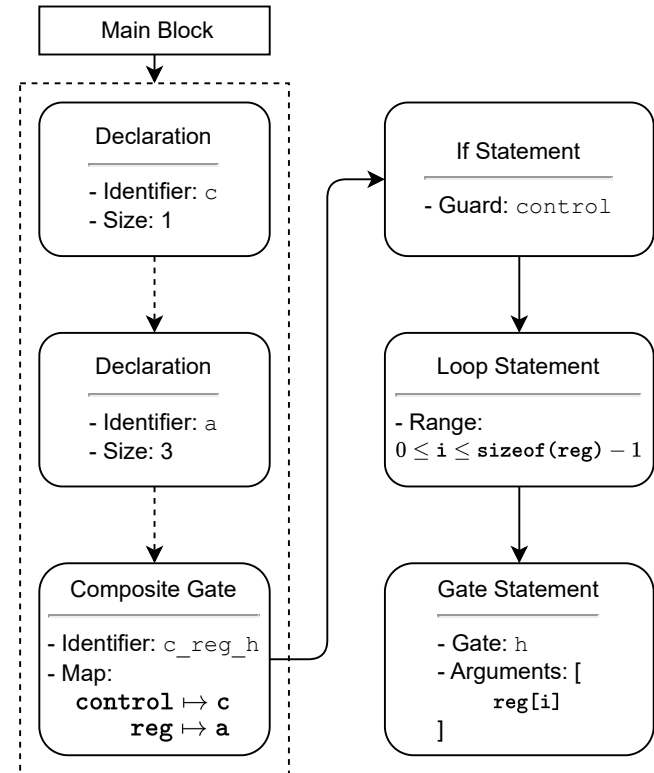
# Implementation

## Source Code Representation

- Three main classes:
  1. Code block: contains list of translatables.
  2. Declaration: only register declaration, constants compile-time only.
  3. Statements: variety of different classes.
- Example contains three translatables.
  - First two declarations, last is gate statement.
  - Gate's body  $\supset$  if-statement  $\supset$  loop statement  $\supset$  gate application.

```
1 gate c_h_reg(control, reg) do
2   qif control do
3     for i in range(sizeof(reg)) do
4       h reg[i];
5   end end end
6
7 qubit c; qubit[3] a;
8 c_h_reg c, a;
```

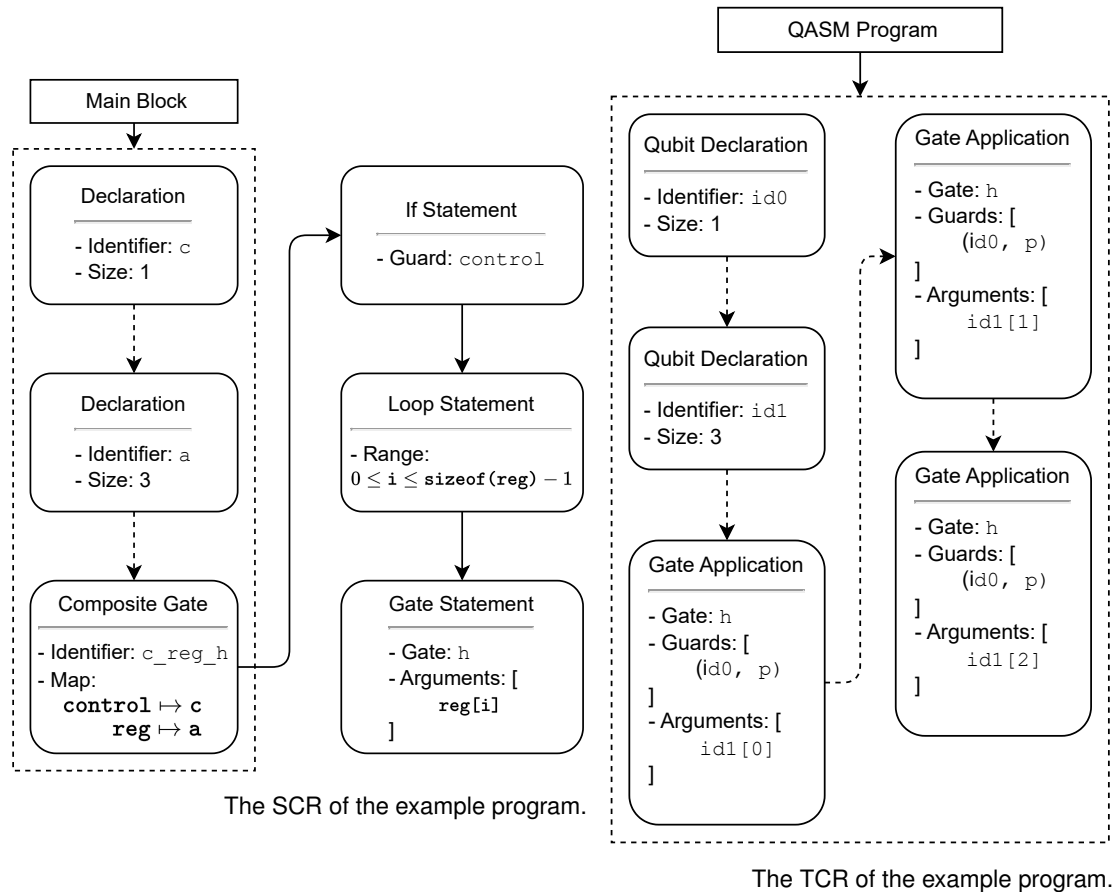
An example Luie program to show the code generation process.



The SCR of the example program.

## Target Code Representation

- TCR based on `QASMP` program object.
- Contains list of `Code` objects, either gate or declaration.
- All SCR objects translated to list of code objects and appended to program object.
- Translate program object: code iterated, converted to text, written file.



## Translated Example Program

- TCR converted to OpenQASM program.
- Version string and include header prepended to the code.
- For each quantum register, classical one is declared, the registers are measured and saved to registers.
- Additions are performed right before result is written to output and after optimization.

```
1 OPENQASM 3.0;  
2 include "stdgates.inc";  
3 qubit id0;  
4 qubit[3] id1;  
5 ctrl(1) @ h id0, id1[0];  
6 ctrl(1) @ h id0, id1[1];  
7 ctrl(1) @ h id0, id1[2];  
8 bit id0_measurement;  
9 measure id0 -> id0_measurement;  
10 bit[3] id1_measurement;  
11 measure id1 -> id1_measurement;
```

The OpenQASM translation of the example Luie program.

## Optimization

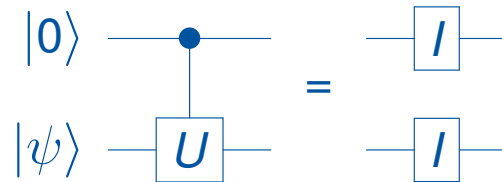
- Compiler can perform optimizations to reduce number of gates and qubits.
- Optimizations are performed to TCR → allow for more optimizations at cost of performance.
- Compiler performs general peephole optimizations based on rules presented by [Garcia-Escartin and Chamorro-Posada, 2011].
- Can be divided into four rules:
  1. Null gate rule.
  2. Peeping control rule.
  3. Hadamard reduction rule (omitted).
  4. Control reversal (omitted).

## Null Gate and Peeping Control Rules

- Null gates are combinations of gates under specific conditions equivalent to  $I$ .
  - Simplest null gate version is twofold application of self-inverse gate.
  - Can be removed entirely from circuit.
- Our peeping control rules are special case of null gates.
  - Control is  $|1\rangle \rightarrow$  remove control.
  - Control is  $|0\rangle \rightarrow$  remove gate.



Null gates of self-inverse gates.



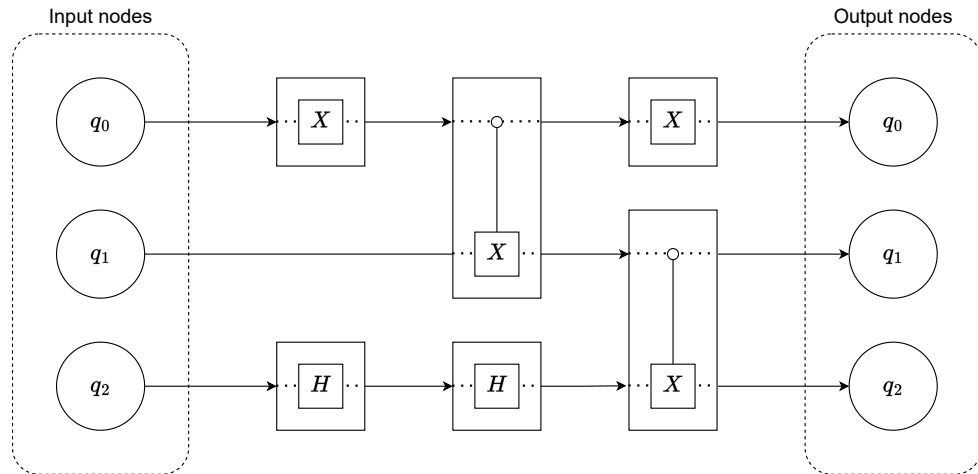
Null gates for gates in specific conditions.

## Circuit Graph

- Rules not directly applied to program, but to circuit graph.
- Program  $\xrightarrow{\text{Translation}}$  Graph  $\xrightarrow{\text{Optimization}}$  Graph  $\xrightarrow{\text{Translation}}$  Program
- Circuit graph is acyclic and directed.
  - Nodes split into input, output, gate nodes.
  - Input, output nodes have only one outgoing, incoming edge.
  - Each input, output, edge corresponds belongs to qubit in program.
  - Qubit wire represented by path from input to output node over corresponding edges.

```
1 qubit[3] q;  
2 x q[0];  
3 cx q[0], q[1];  
4 x q[0];  
5 h q[2];  
6 h q[2];  
7 cx q[1], q[2];
```

A simple, unoptimized program.



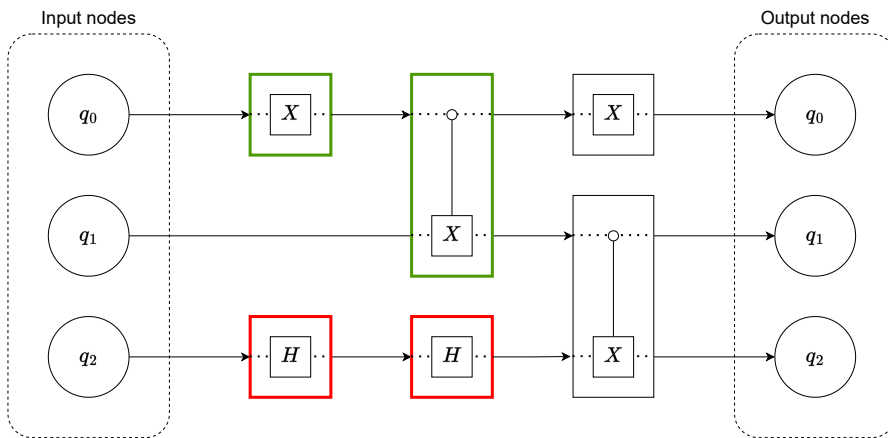
An example of a simple, unoptimized circuit graph.



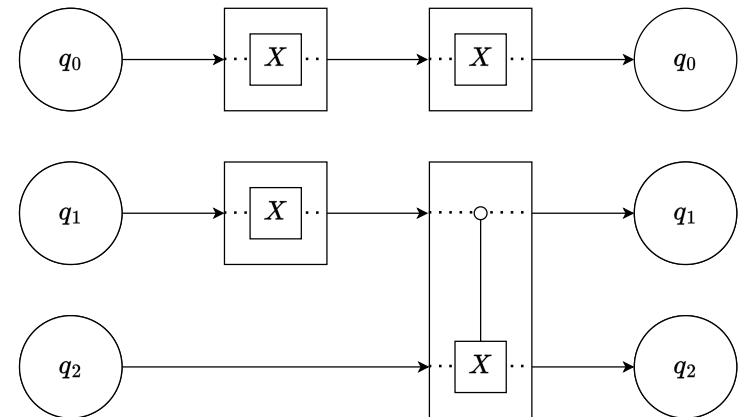
# Implementation

## Example Optimization Process I

- To optimize the graph, each qubit wire is iterated.
- All subpaths (up to max. length) checked for alternatives.
- Example:
  - **First wire:** Peeping control rule can be applied to first **CX** gate.
  - **Third wire:** **HH** null gate can be removed.



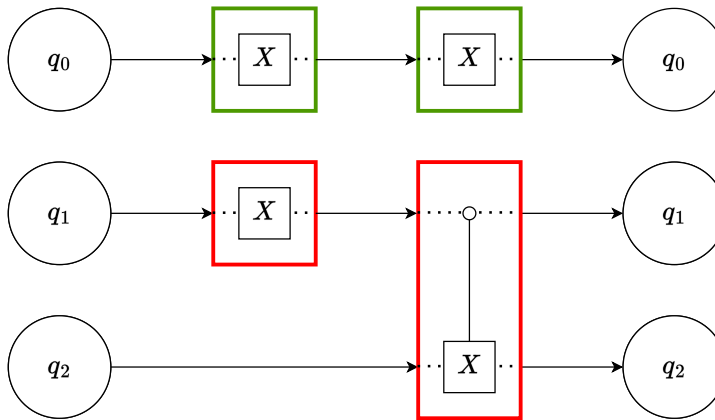
An example of a simple, unoptimized circuit graph.



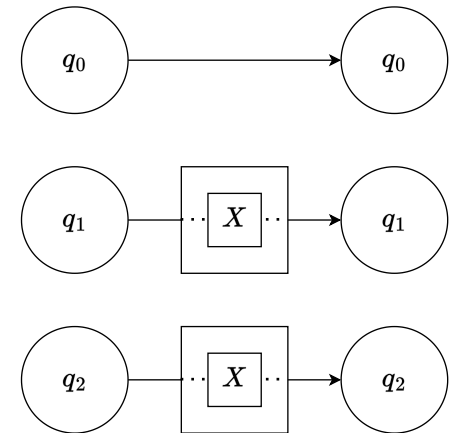
The circuit graph after the first optimization step.

## Example Optimization Process II

- While all qubit wires iterated, still possible optimizations.
  - Applying optimizations may enable others!
- ⇒ Optimization repeated as long as previous iteration applied optimizations.
- Example:
    - **First wire:**  $XX$  null gate can be removed.
    - **Second wire:** Peeping control rule can be applied to  $CX$  gate.



The circuit graph after the first optimization step.



The completely optimized graph.

## Evaluation

- Evaluation consists of two aspects:
  1. Optimizations performed.
  2. Execution time stages.
- Example program: quantum ripple-carry adder proposed by [Cuccaro et al., 2004].
  - Takes two registers *a* and *b*, two qubits *cin* and *cout*.
  - Adder adds *a* register to *b* register.
  - *cin* and *cout* qubits are used as input, output carry bits.
- Our implementation consists only of *CX* and *CCX*.

## Optimization Evaluation

- For optimization evaluation, classical inputs and in superposition.
- First inputs:  $a = |1\rangle$  and  $b = |15\rangle$ .
  - Classical inputs  $\rightarrow$  peeping control rules, null gate rules can be applied.
  - Optimized circuit: only gates that initialize the result.
  - Only two  $X$  gates remain.
  - First  $X$  flips  $a[0]$  qubit  $\rightarrow a = |1\rangle$ , second flips  $cout$ , indicating result of  $|16\rangle$ .
- Second inputs:  $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$  and  $b = |4\rangle$ .
  - Inputs in superposition  $\rightarrow$  peeping control rules, null gate rules only applied *partially*.
  - Only  $\frac{12}{25}$  gates can be optimized.
  - Other inputs in superposition: even fewer gates optimized.

# Evaluation

---

## Performance Evaluation

- Compiled adder with input of  $a = \frac{1}{\sqrt{2}}(|0\rangle + |3\rangle)$  and  $b = |15\rangle$  for different register sizes  $n$ .
- Program size does not change  $\rightarrow$  execution times of the semantic analysis remain constant.
- Code generation stage: linear increase.
  - Unrolled loop  $\rightarrow$  compiled program increases linearly.
- Optimization worst performance, approximate quadratic increase.

| Register Size $n$ | Execution Time of Stages in ms |                 |              |
|-------------------|--------------------------------|-----------------|--------------|
|                   | Semantic Analysis              | Code Generation | Optimization |
| 64                | 27.3                           | 47.8            | 711.6        |
| 128               | 26.3                           | 50.4            | 2292.4       |
| 256               | 26.2                           | 59.7            | 10755.7      |
| 512               | 25.8                           | 74.9            | 60204.7      |
| 1024              | 26.1                           | 109.1           | 405376.6     |






The execution times compiling a quantum ripple-carry adder with different register sizes.

## Conclusion

- Multiple aspects that can be improved.
- Optimizations:
  - Focused on high-level optimizations → work best in tandem with other tools.
  - Addition of hardware-focused features and transpilation to concrete devices.
  - Improve performance of optimization stage by parallelization of wire traversals.
  - Increases complexity of program, e.g., race conditions need to be considered.
- General improvements:
  - Addition of type casting for named constants.
  - Expansion of predefined function; addition of constants, e.g.,  $\pi$  or  $e$ .
  - Addition of explicit measurement.


## References

---

-  [Altenkirch, T. and Grattage, J. \(2005\).](#)  
A functional quantum programming language.  
[In 20th Annual IEEE Symposium on Logic in Computer Science \(LICS' 05\), pages 249–258. IEEE.](#)
-  [Cuccaro, S. A., Draper, T. G., Kutin, S. A., and Moulton, D. P. \(2004\).](#)  
A new quantum ripple-carry addition circuit.
-  [Garcia-Escartin, J. C. and Chamorro-Posada, P. \(2011\).](#)  
Equivalent quantum circuits.
-  [Landauer, R. \(1961\).](#)  
Irreversibility and heat generation in the computing process.  
[IBM Journal of Research and Development, 5\(3\):183–191.](#)
-  [Ying, M., Yu, N., and Feng, Y. \(2012\).](#)  
Defining quantum control flow.

## References

---

-  Yuan, C., Villanyi, A., and Carbin, M. (2024).  
Quantum control machine: The limits of control flow in quantum programming.  
[Proceedings of the ACM on Programming Languages](#), 8(OOPSLA1):1–28.