QDB: From Quantum Algorithms Towards Correct Quantum Programs

Yipeng Huang, Margaret Martonosi

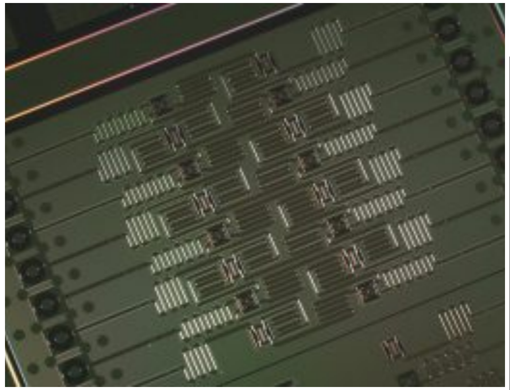
PRINCETON UNIVERSITY

This work is funded in part by EPiQC, an NSF Expedition in Computing, under grant 1730082

Motivation: Race to practical quantum computation

Superconducting qubits

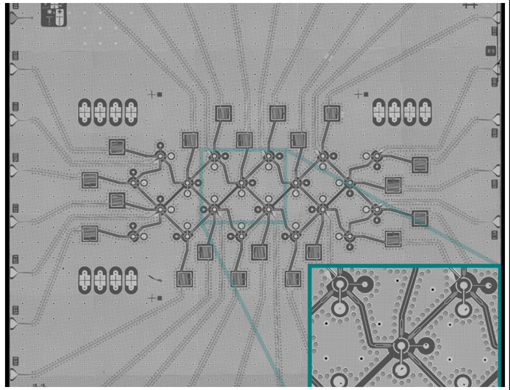
Trapped ion qubits



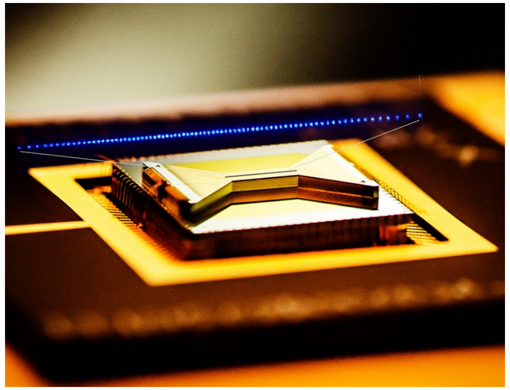
IBM

Google

Intel



Rigetti



University of Maryland / IonQ

Many research teams now competing towards more reliable and more numerous qubits.

Motivation: Race to practical quantum computation

Quantum chemistry algorithms

- Calculating molecule properties from first principles
- Use quantum mechanical system to simulate quantum mechanics!
- Near term: needs few qubits, needs no error correction

Motivation: Race to practical quantum computation

Quantum chemistry algorithms

- Calculating molecule properties from first principles
- Use quantum mechanical system to simulate quantum mechanics!
- Near term: needs few qubits, needs no error correction

Shor's integer factorization quantum algorithm

- Factors large integers in polynomial time!
- (known best classical algorithms take exponential time)
- Distant future: needs many qubits, needs error correction

Motivation: Race to practical quantum computation

Quantum chemistry algorithms

- Calculating molecule properties from first principles
- Use quantum mechanical system to simulate quantum mechanics!
- Near term: needs few qubits, needs no error correction

Shor's integer factorization quantum algorithm

- Factors large integers in polynomial time!
- (known best classical algorithms take exponential time)
- Distant future: needs many qubits, needs error correction

Many more algorithms: https://math.nist.gov/quantum/zoo/

Semantic gap

- Need languages, abstractions…

Tools gap

- Need optimizing compilers, simulators, debuggers…

Infrastructure gap

- Need more abundant, more reliable qubits…

Educational gap

- Need researchers, students…

Quantum algorithms

GAP!

Quantum physical devices

Semantic gap

- Need languages, abstractions…

Tools gap

- Need optimizing compilers, simulators, debuggers….

Infrastructure gap

- Need more abundant, more reliable qubits…

Educational gap

- Need researchers, students…

This paper is about quantum PL support for correctness

Quantum algorithms

Quantum
programming languages

Quantum programming
patterns and antipatterns:
bugs and defenses

Building blocks:
qubits, gates, circuits

Quantum physical devices

This paper is about quantum PL support for correctness

Detailed debugging effort across quantum algorithms

Quantum chemistry, Shor’s factoring, Grover’s search

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices

This paper is about quantum PL support for correctness

Detailed debugging effort across quantum algorithms

Quantum chemistry, Shor’s factoring, Grover’s search

Where possible, validate across quantum languages

Scaffold, ProjectQ, QISKit… compare correctness features

Quantum algorithms

Quantum programming languages

Quantum programming patterns and antipatterns: bugs and defenses

Building blocks: qubits, gates, circuits

Quantum physical devices

This paper is about quantum PL support for correctness

Detailed debugging effort across quantum algorithms

Quantum chemistry, Shor's factoring, Grover's search

Where possible, validate across quantum languages

Scaffold, ProjectQ, QISKit… compare correctness features

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices

This paper is about quantum PL support for correctness

Detailed debugging effort across quantum algorithms

Quantum chemistry, Shor's factoring, Grover's search

Where possible, validate across quantum languages

Scaffold, ProjectQ, QISKit… compare correctness features

Classify quantum bugs in input, operations, and output

Paired with defenses: unit testing, syntax support, assertions

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices

Quantum computing primer to understand debugging

$|0\rangle$

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Quantum computing primer to understand debugging

$$|0\rangle \longrightarrow \boxed{H} \longrightarrow q$$

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Quantum computing primer to understand debugging

$$|0\rangle \longrightarrow \boxed{H} \longrightarrow q$$

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$q = H|0\rangle$$

Quantum computing primer to understand debugging

$$|0\rangle \longrightarrow \boxed{H} \longrightarrow q$$

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$q = H|0\rangle$$

Quantum qubit
Superposition

$$q = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Quantum variables' ability to be in superposition underlies power of quantum computing.

Quantum computing primer to understand debugging

$$|0\rangle \longrightarrow \boxed{H} \longrightarrow q$$

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$$

$$|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$$

$$q = H|0\rangle$$

Quantum qubit
Superposition

$$q = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Quantum variables' ability to be in superposition underlies power of quantum computing.

Quantum computing primer to understand debugging

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$$

$$|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$$

$$q = H|0\rangle$$

Quantum qubit
Superposition

$$q = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Measurement
Collapses state

$$m = \begin{cases}0, P = {}^{1}\!/_{2}\\1, P = {}^{1}\!/_{2}\end{cases}$$

Quantum computing primer to understand debugging

Classical value	**Hadamard gate**	**Quantum qubit**	**Measurement**
Deterministic	A quantum operator	Superposition	Collapses state
$\|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ $\|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$	$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$ $q = H\|0\rangle$	$q = \frac{1}{\sqrt{2}}\|0\rangle + \frac{1}{\sqrt{2}}\|1\rangle$	$m = \begin{cases}0, P = {}^{1}\!/_{2}\\1, P = {}^{1}\!/_{2}\end{cases}$

We cannot pause a quantum computer and "printf debug," because measurement collapses state.

Quantum computing primer to understand debugging

Classical value
Deterministic

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$$

$$|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

Hadamard gate
A quantum operator

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$$

$$q = H|0\rangle$$

Quantum qubit
Superposition

$$q = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Measurement
Collapses state

$$m = \begin{cases}0, P = {}^{1}\!/_{2}\\1, P = {}^{1}\!/_{2}\end{cases}$$

We cannot pause a quantum computer and "printf debug," because measurement collapses state.

Quantum programming is the process of converting quantum circuit diagrams to quantum code.

Quantum computing primer to understand debugging

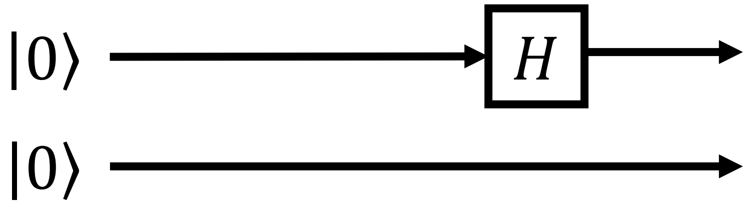
$|0\rangle$

$|0\rangle$

Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

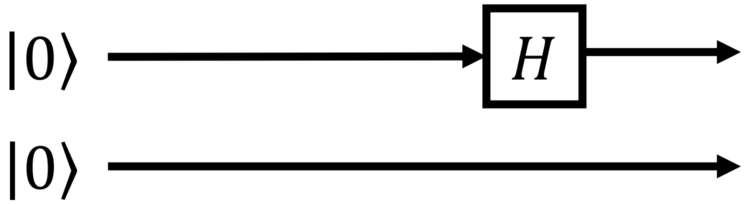
$$|0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$= \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |10\rangle$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

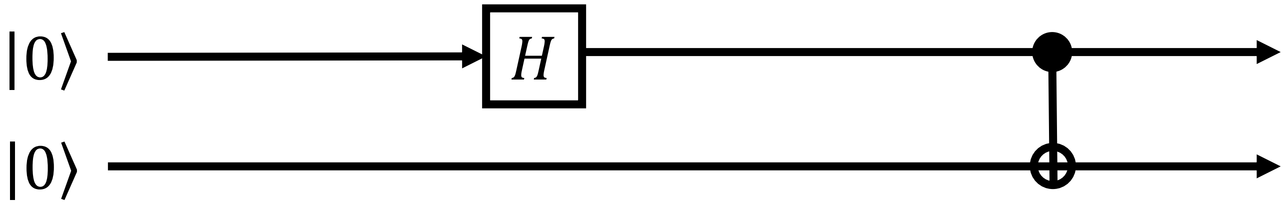
$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

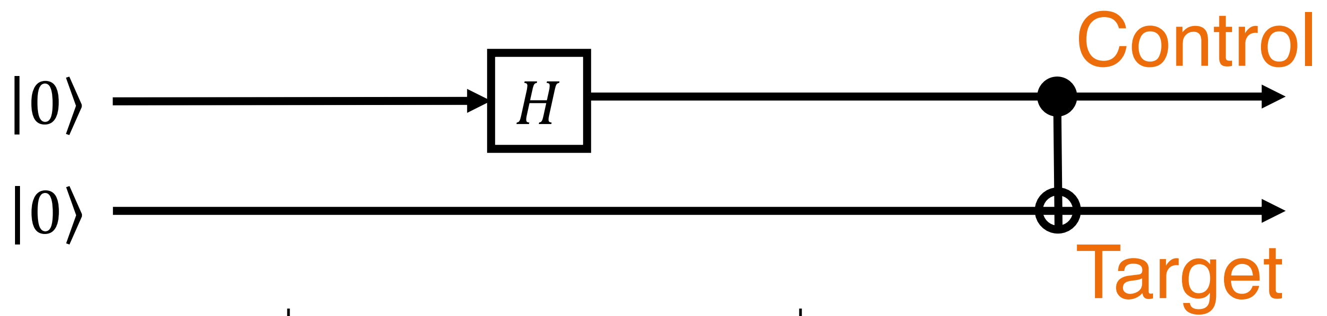
$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Controlled-NOT
Two-qubit operator

$$\mathrm{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

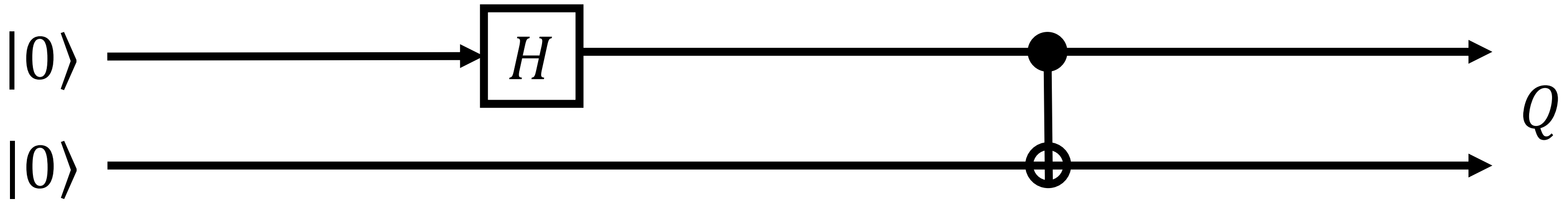
$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Controlled-NOT
Two-qubit operator

$$\text{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

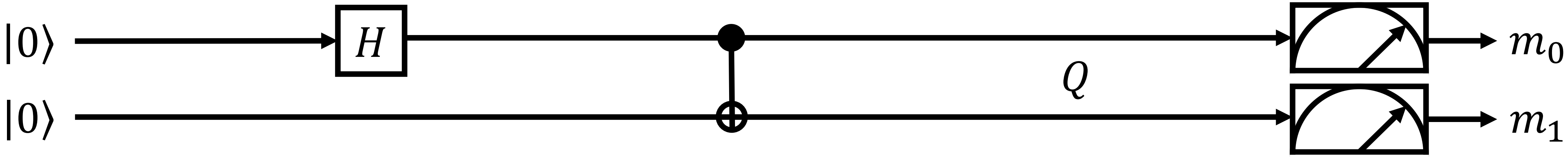
Controlled-NOT
Two-qubit operator

$$\mathrm{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

Entangled state
Cannot be factored

$$Q = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Controlled-NOT
Two-qubit operator

$$\mathrm{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

Entangled state
Cannot be factored

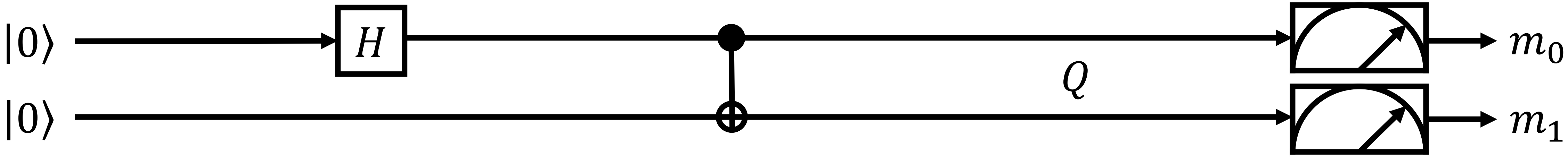
$$Q = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

Measurement
Results correlated

$$(m_0, m_1)$$

$$= \begin{cases}(0,0), P = {}^{1}\!/_{2}\\(1,1), P = {}^{1}\!/_{2}\end{cases}$$

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Controlled-NOT
Two-qubit operator

$$\text{CNOT} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$$

Entangled state
Cannot be factored

$$Q = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

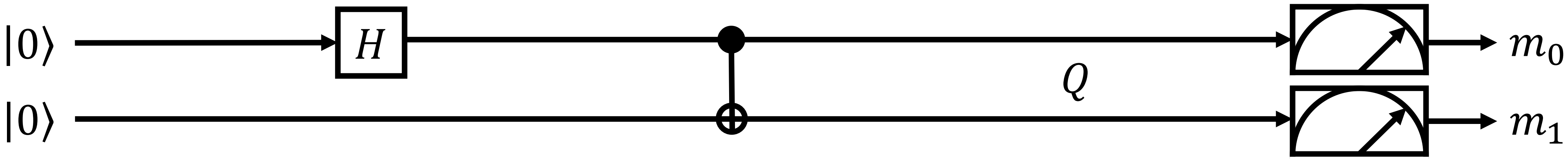
Measurement
Results correlated

$$(m_0, m_1)$$

$$= \begin{cases}(0,0), P = {}^1\!/_2\\(1,1), P = {}^1\!/_2\end{cases}$$

Possibility of entanglement leads to huge state space underlying power of quantum computing.

Quantum computing primer to understand debugging



Two qubits
Tensor product

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |00\rangle$$

Product state
Can be factored

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix}$$

$$= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Controlled-NOT
Two-qubit operator

$$\mathrm{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

Entangled state
Cannot be factored

$$Q = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

Measurement
Results correlated

$$(m_0, m_1)$$

$$= \begin{cases}(0,0), P = {}^{1}/_{2}\\(1,1), P = {}^{1}/_{2}\end{cases}$$

Possibility of entanglement leads to huge state space underlying power of quantum computing.

Huge state space limits us to simulating only toy-sized quantum programs.

Now that we understand what quantum programming means, what is the prior work on debugging?

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the entanglement of qubits and the inability to copy.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “simulate all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUi|>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable” [Paykin+, “QWIRE,” POPL 2017]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the **entanglement** of qubits and the **inability to copy**.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “simulate all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUi|>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable” [Paykin+, “QWIRE,” POPL 2017]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the entanglement of qubits and the inability to copy.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “**simulate** all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUil>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable” [Paykin+, “QWIRE,” POPL 2017]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the entanglement of qubits and the inability to copy.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “simulate all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUil>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and **testing using simulations is not scalable**” [Paykin+, “QWIRE,” POPL 2017]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the entanglement of qubits and the inability to copy.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “simulate all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUil>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable” [Paykin+, “QWIRE,” POPL 2017]

…and will need novel ideas.

- “full CS ecosystem needed… languages, simulators, **debuggers**” [Martonosi, keynote, PPoPP / HPCA 2018]
- “Can we **check useful properties** in polynomial time for programs with quantum supremacy?” [Chong, keynote, ASPLOS 2018]

QC researchers anticipate debugging will be important…

- “methods for debugging quantum programs in situ...will soon be important” [Green+, “Quipper,” PLDI 2013]

…but it will be difficult…

- “Debugging tools will require innovative approaches due to the entanglement of qubits and the inability to copy.” [Svore+, “The Quantum Future of Computation,” Computer 2016]
- “simulate all operations... to enable algorithm development… and verification of correctness.” [Wecker+, ”LIQUil>,” PPoPP 2015]
- “debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable” [Paykin+, “QWIRE,” POPL 2017]

…and will need novel ideas.

- “full CS ecosystem needed… languages, simulators, debuggers” [Martonosi, keynote, PPoPP / HPCA 2018]
- “Can we check useful properties in polynomial time for programs with quantum supremacy?” [Chong, keynote, ASPLOS 2018]

Despite all the interest in debugging, little concrete has been written. Define bugs? Defenses?

Quantum algorithms

Quantum programming languages

Quantum programming patterns and antipatterns: bugs and defenses

Building blocks: qubits, gates, circuits

Quantum physical devices

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation
4. Algorithm progress assertions
5. Postconditions

A first taxonomy of quantum program bugs and defenses.

Quantum program bug types

Defenses, debugging, and assertions

1. **Quantum initial values**
2. **Basic operations**
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Detailed debugging of Shor’s factorization algorithm

Detailed debugging of Shor's factorization algorithm

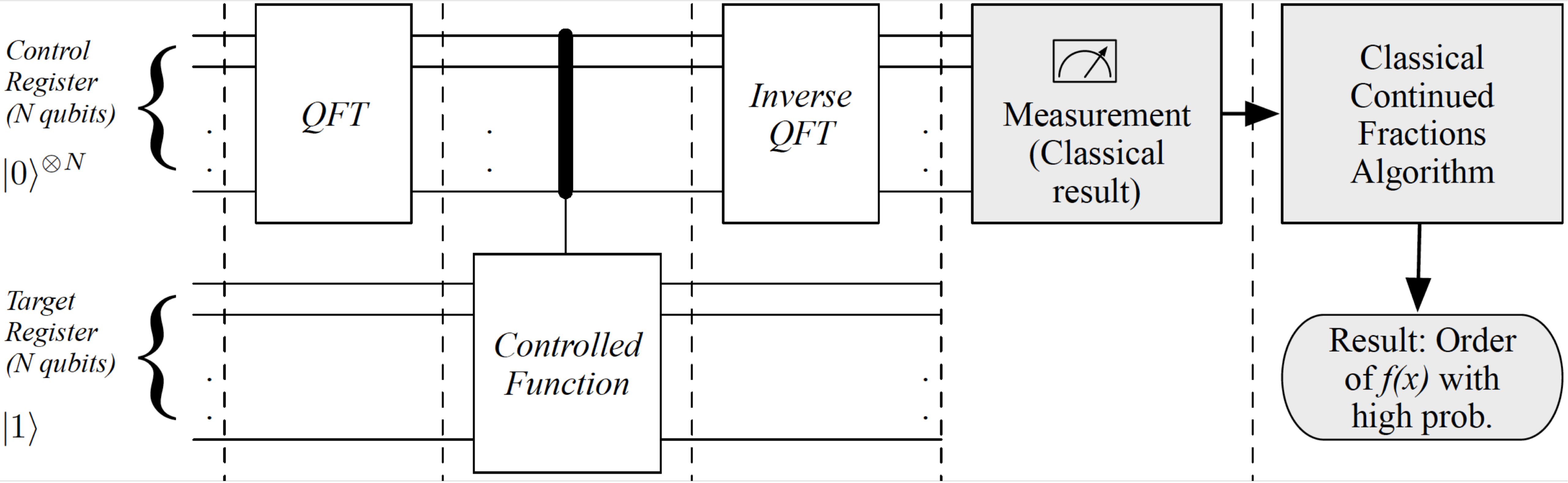


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Detailed debugging of Shor's factorization algorithm

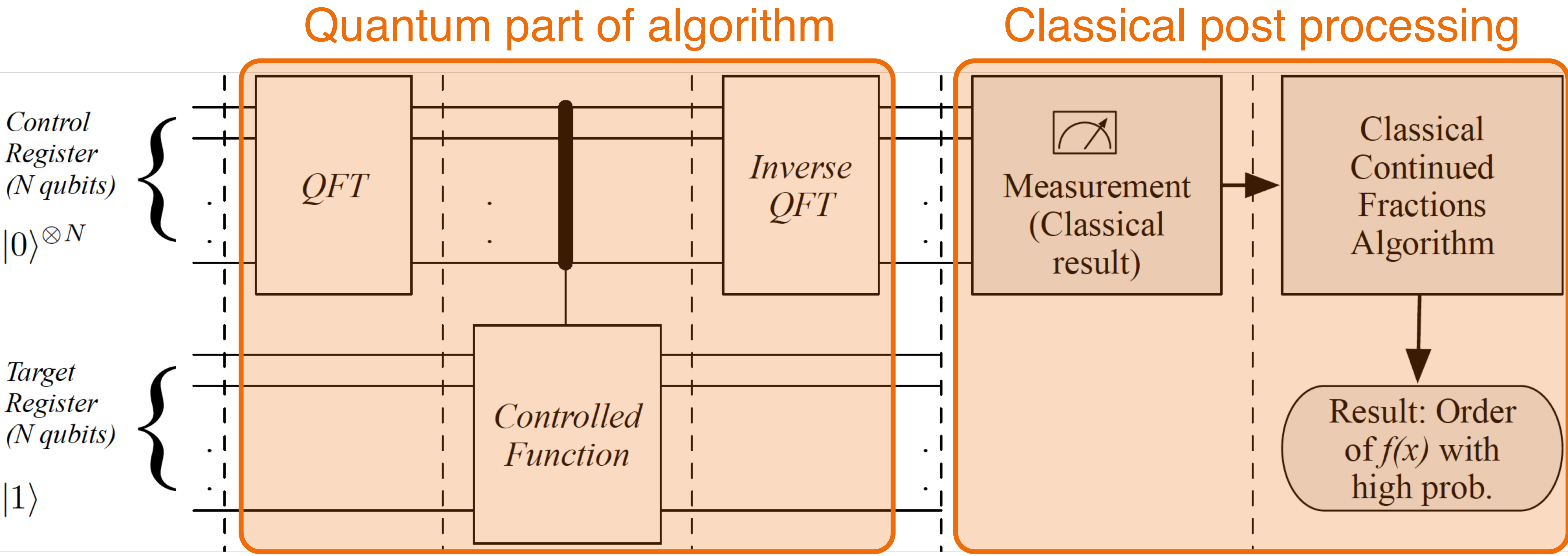
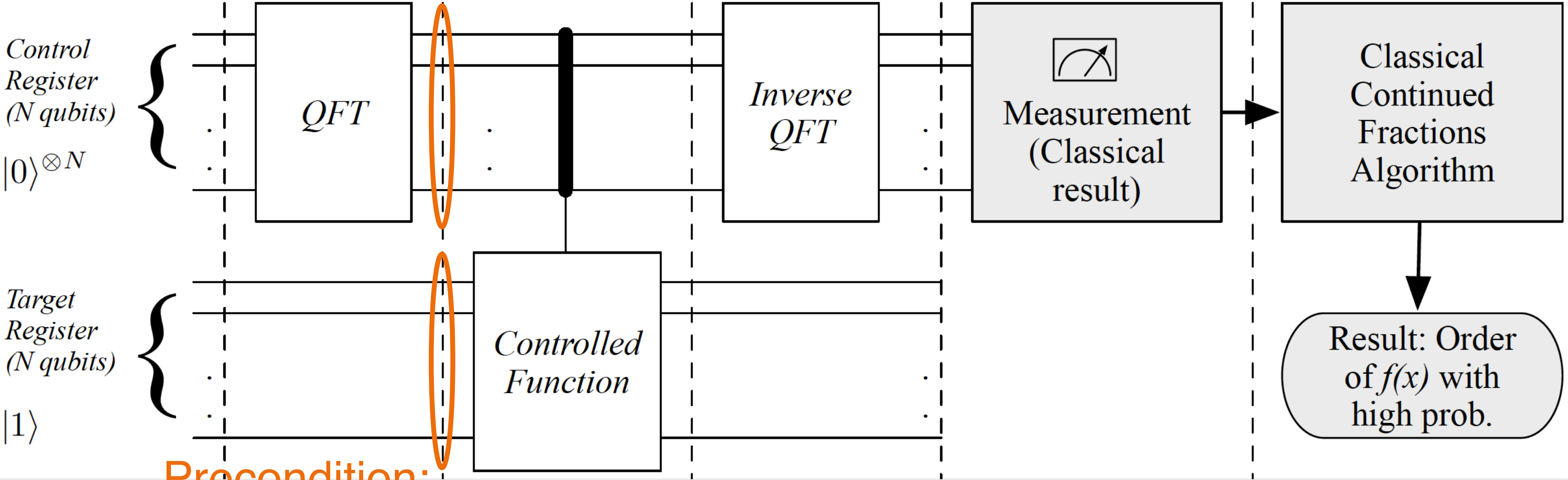


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Bug type 1: mistake in quantum initial values

Precondition:
uniform distribution



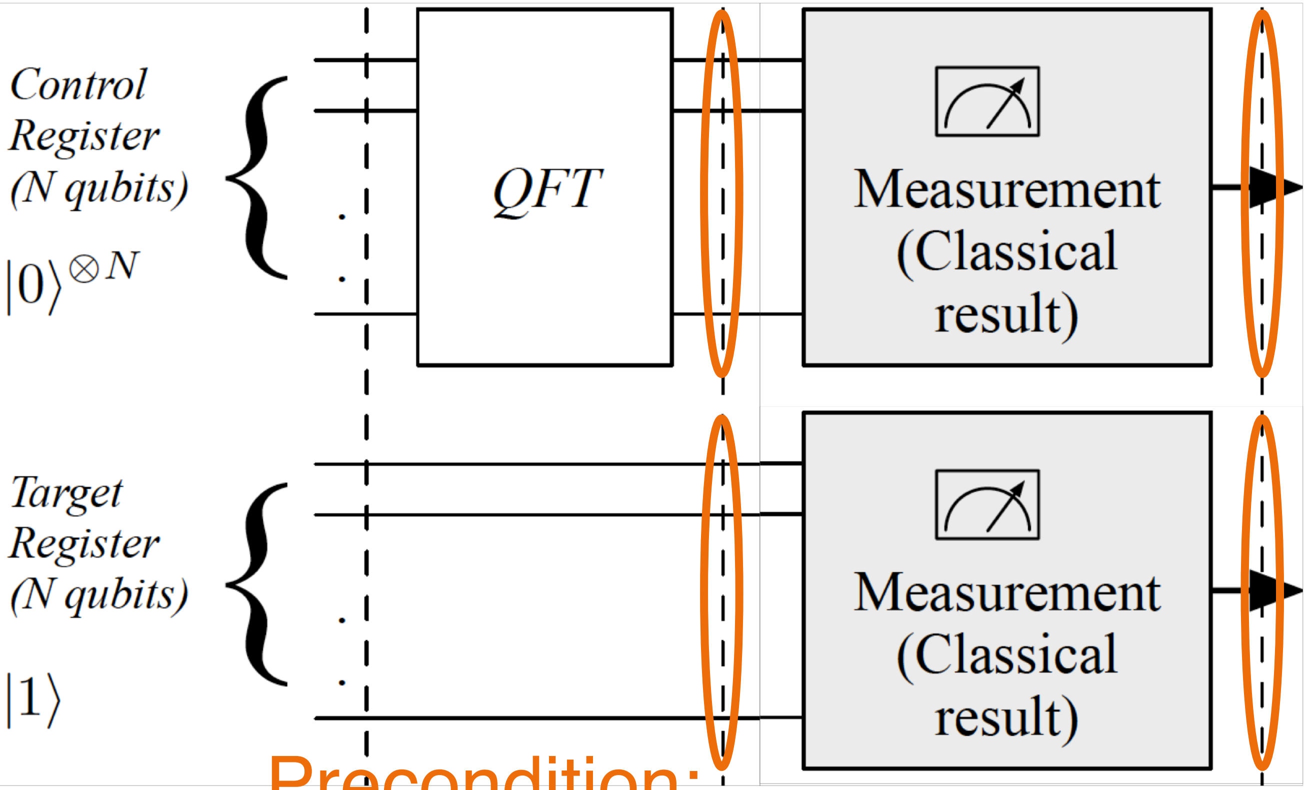
Precondition:
any integer

Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Defense type 1: check for precondition assertions

Precondition:
uniform distribution

Validate
precondition



Measure prematurely, check preconditions met, then restart execution.

Precondition:
any integer

Validate
precondition

Bug type 2: mistake in coding basic operations

Consists of
controlled rotate-Z

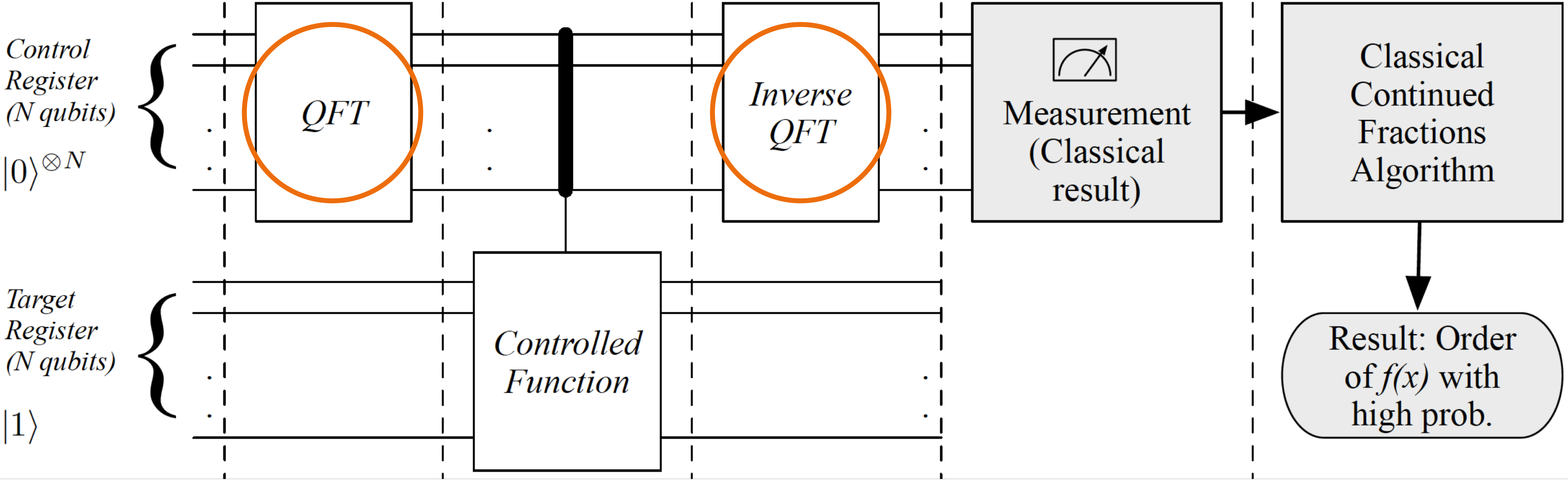
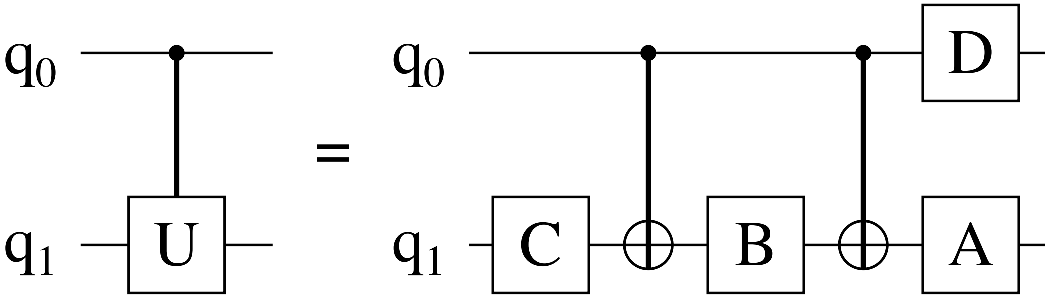


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

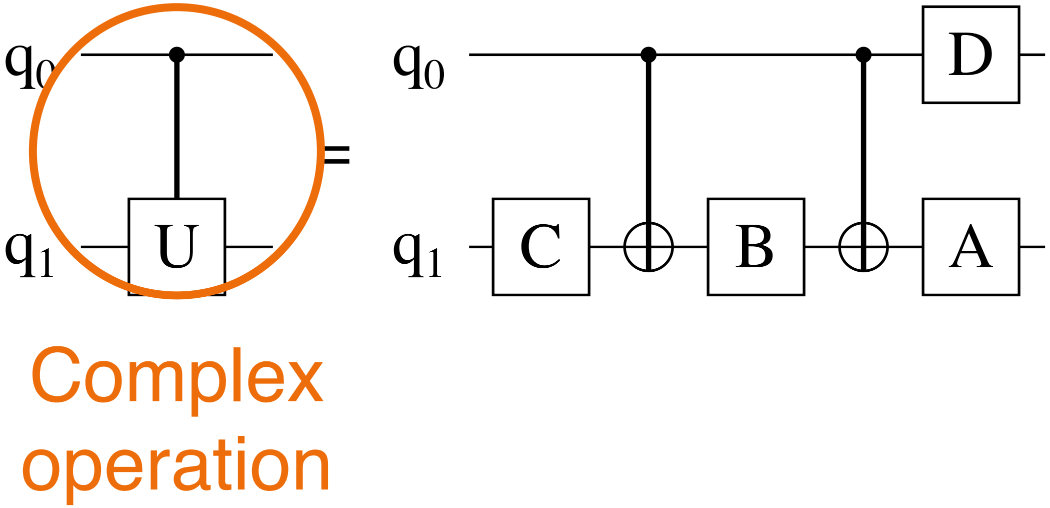
Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:



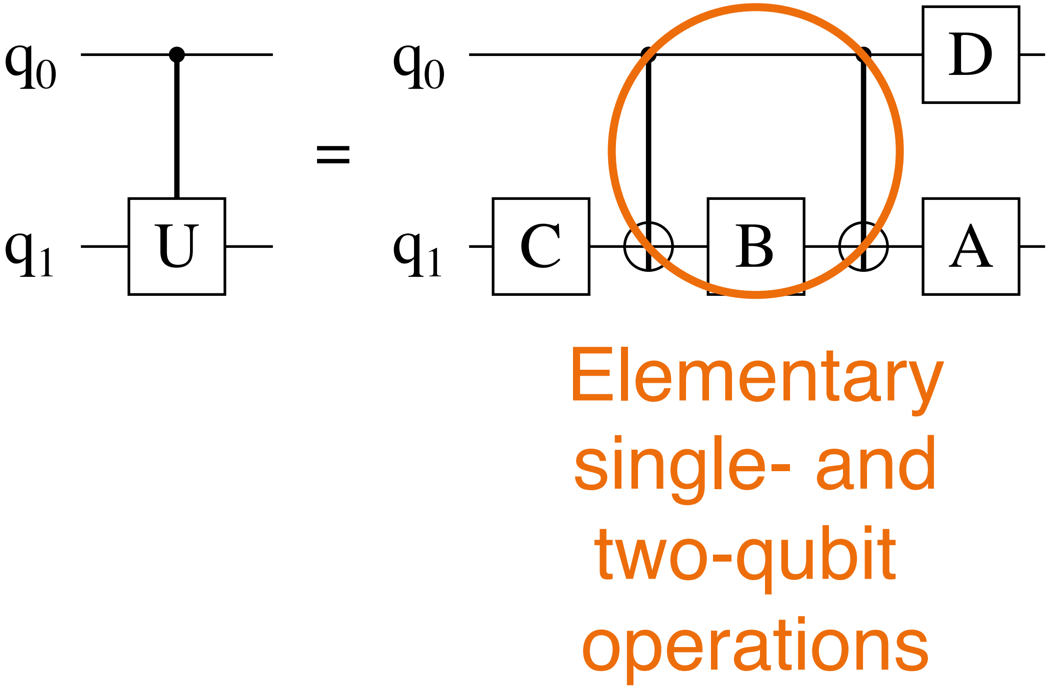
Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:



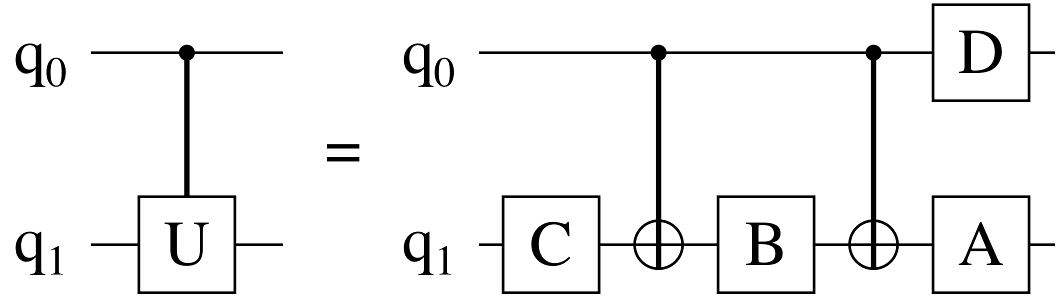
Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:



Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:



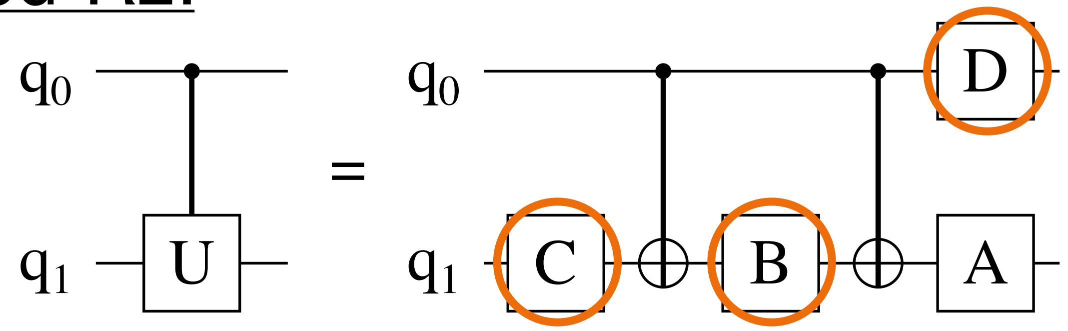

```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

Correct,
operation A unneeded

Scaffold language

Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:

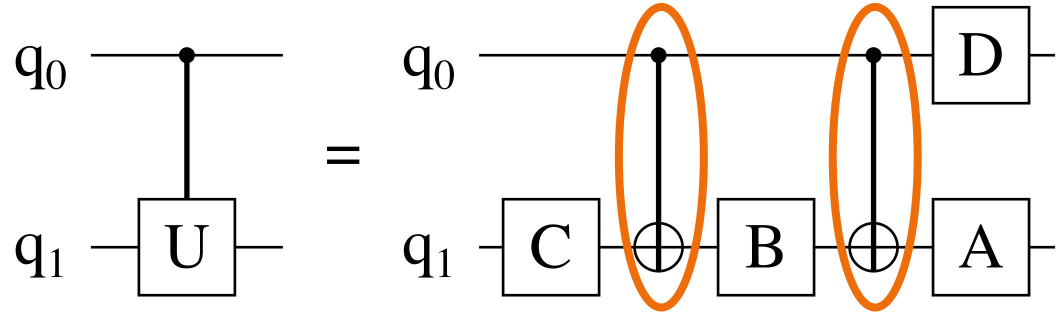


Elementary single-qubit operations

`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
Correct, operation A unneeded	

Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:



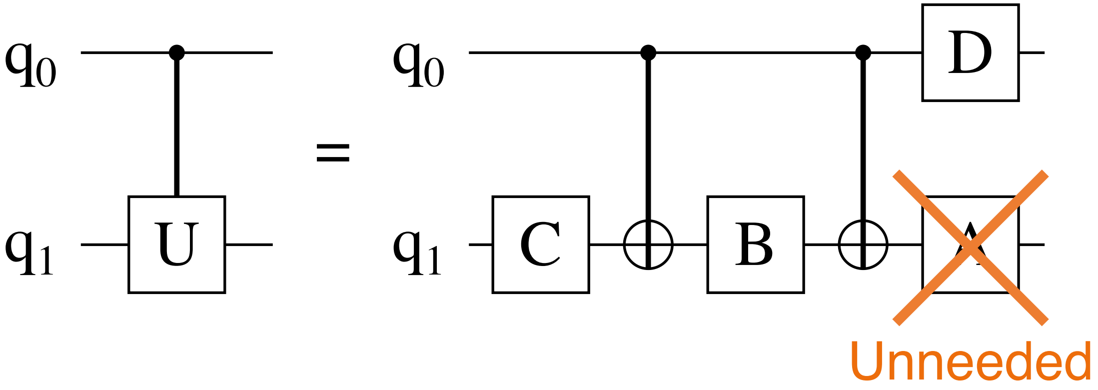
Elementary two-qubit operations

```
Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
```

Correct,
operation A unneeded

Bug type 2: mistake in coding basic operations

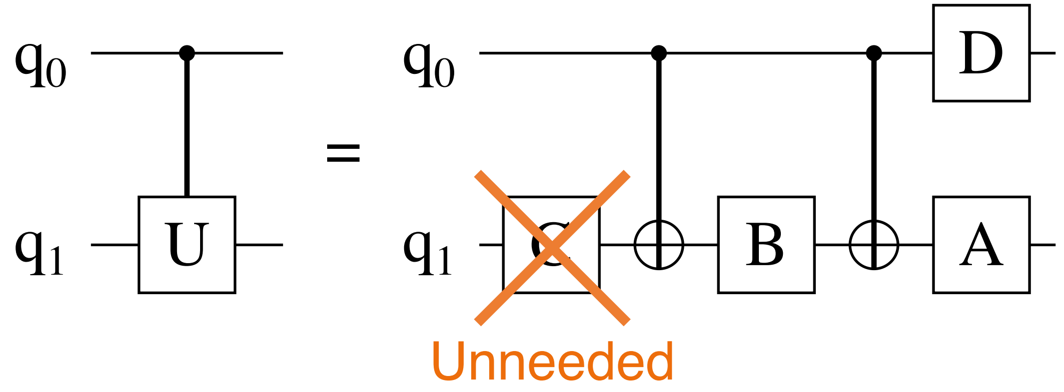
E.g., controlled-Rz:



`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
Correct, operation A unneeded	

Bug type 2: mistake in coding basic operations

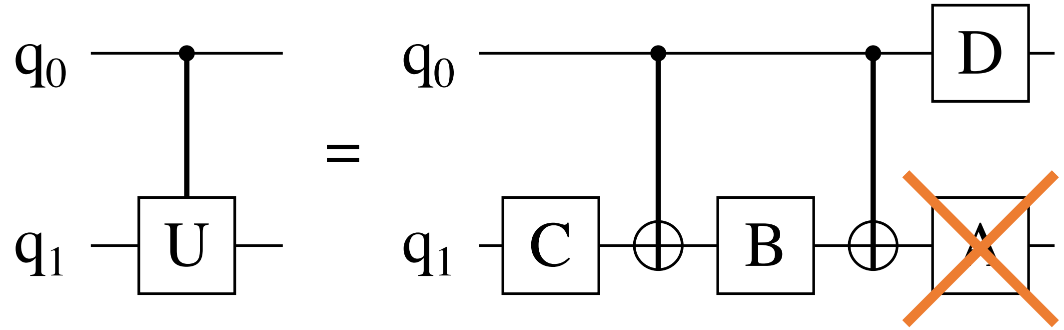
E.g., controlled-Rz:



`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
Correct, operation A unneeded
	`CNOT(q0, q1);`
	`Rz(q1, -angle/2); // B`
	`CNOT(q0, q1);`
	`Rz(q1, +angle/2); // A`
	`Rz(q0, +angle/2); // D`
	
	Correct, operation C unneeded	

Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:

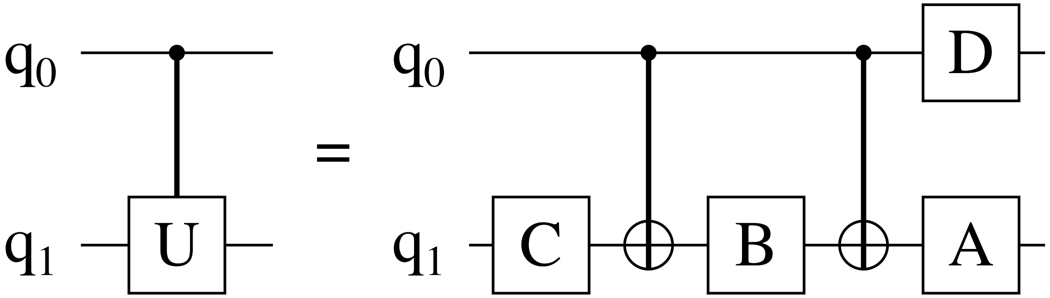


Unneeded?
But signs on angles wrong!

`Rz(q1, +angle/2); // C`	`CNOT(q0, q1);`	`Rz(q1, -angle/2);`
`CNOT(q0, q1);`	`Rz(q1, -angle/2); // B`	`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`	`CNOT(q0, q1);`	`Rz(q1, +angle/2);`
`CNOT(q0, q1);`	`Rz(q1, +angle/2); // A`	`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`	`Rz(q0, +angle/2); // D`	`Rz(q0, +angle/2); // D`
Correct, operation A unneeded	**Correct, operation C unneeded**	**Incorrect, angles flipped**

Bug type 2: mistake in coding basic operations

E.g., controlled-Rz:

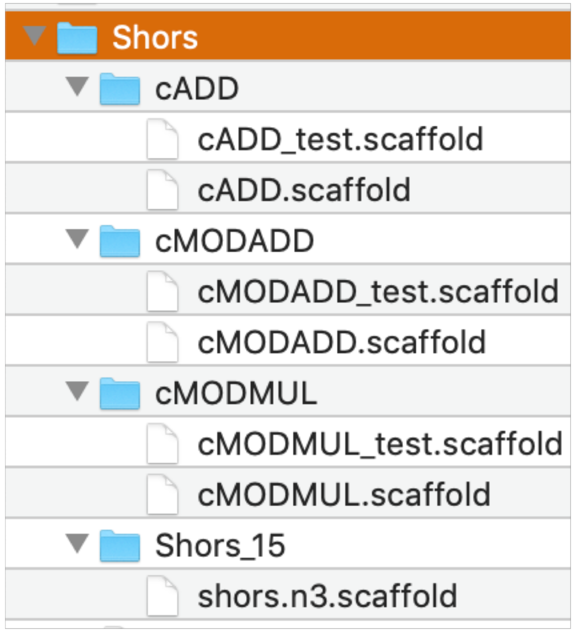


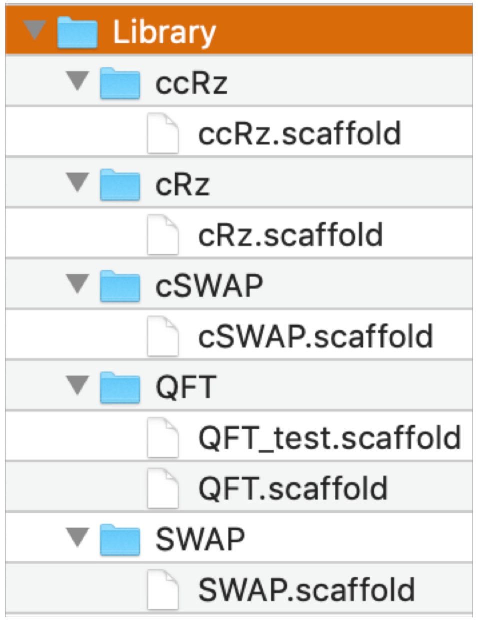
`Rz(q1, +angle/2); // C`
`CNOT(q0, q1);`
`Rz(q1, -angle/2); // B`
`CNOT(q0, q1);`
`Rz(q0, +angle/2); // D`
 	
**Correct,
operation A unneeded**
	`CNOT(q0, q1);`
	`Rz(q1, -angle/2); // B`
	`CNOT(q0, q1);`
	`Rz(q1, +angle/2); // A`
	`Rz(q0, +angle/2); // D`
	
	**Correct,
	operation C unneeded**
		`Rz(q1, -angle/2);`
		`CNOT(q0, q1);`
		`Rz(q1, +angle/2);`
		`CNOT(q0, q1);`
		`Rz(q0, +angle/2); // D`
		
		**Incorrect,
		angles flipped**

Many ways to translate basic quantum operations to program code—many details to get right!

Defense type 2: support for subroutines / unit tests

E.g., Shor's subroutines:

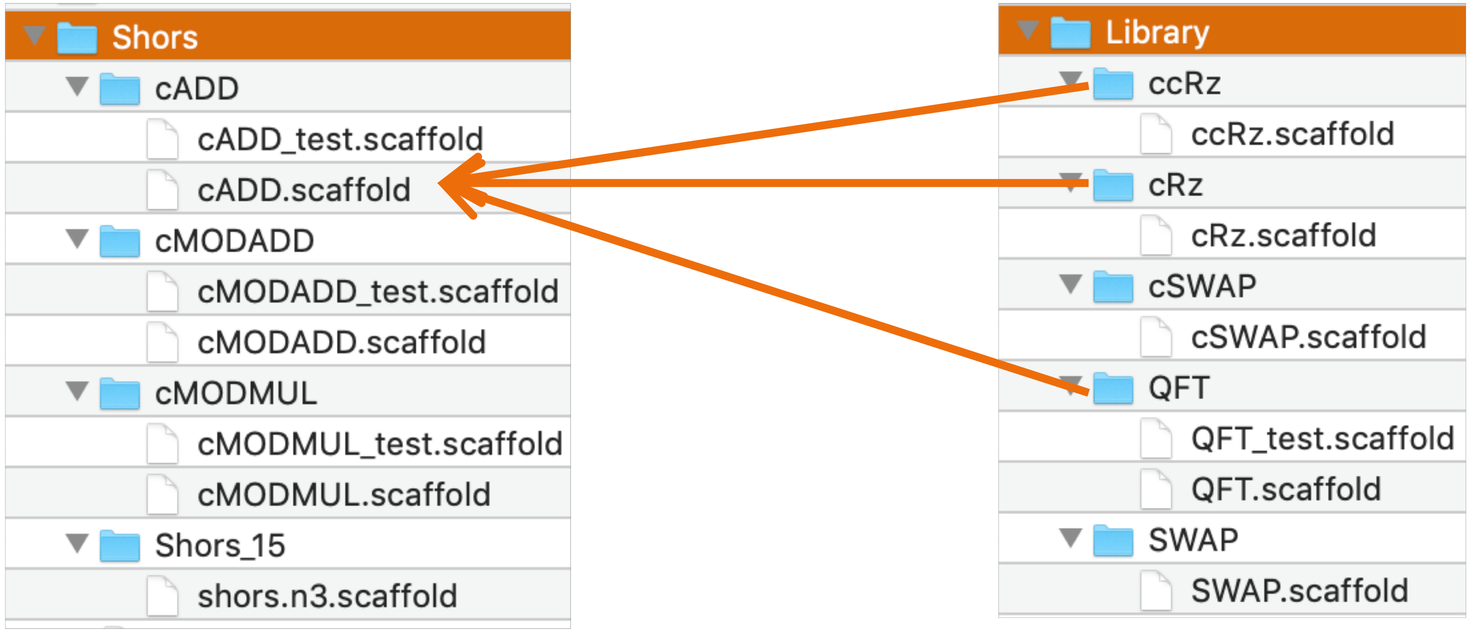




- Unit (stress) testing
- Code reuse

Defense type 2: support for subroutines / unit tests

E.g., Shor's subroutines:



- Unit (stress) testing
- Code reuse

Quantum program bug types

1. **Quantum initial values**
2. **Basic operations**
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. **Preconditions**
2. **Subroutines / unit tests**

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. **Composing operations**
   A. **Iteration**
   B. **Mirroring**
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests

Bug type 3-A: mistake in composing gates using iterations

Compose basic gates
through iteration

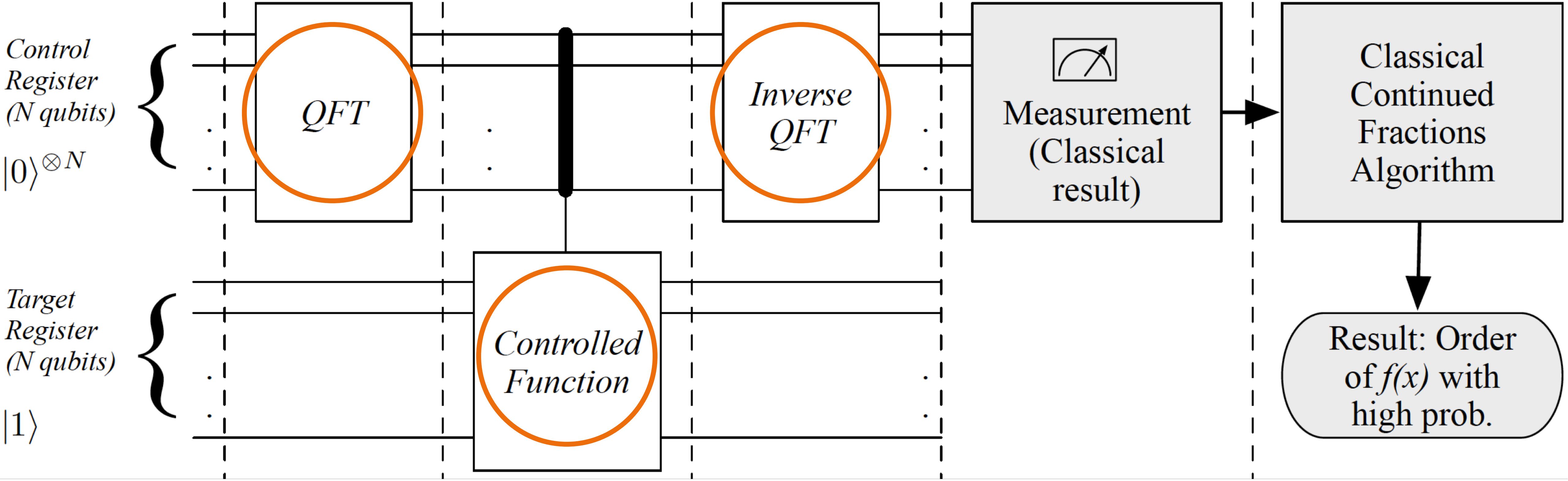


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Bug type 3-A: mistake in composing gates using iterations

E.g., quantum Fourier transform:

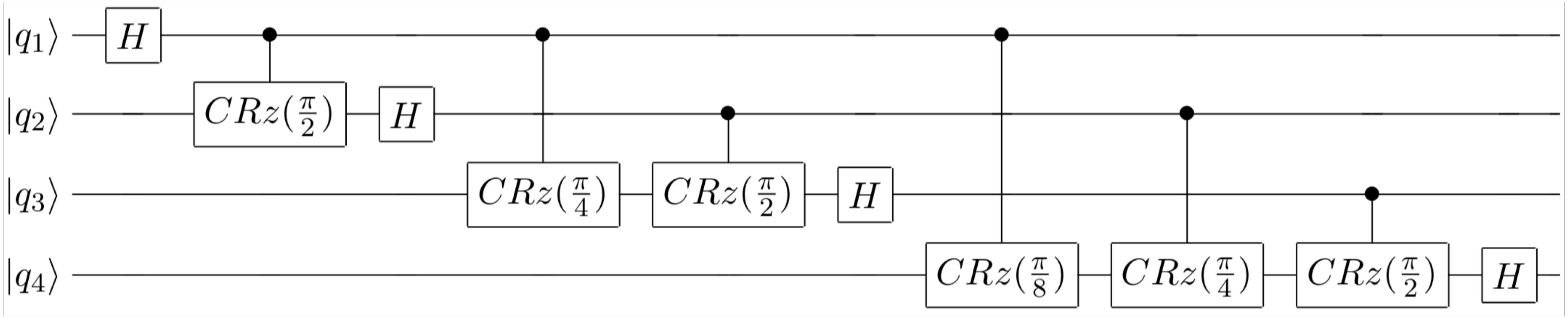


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Tricky iterations—

Bug type 3-A: mistake in composing gates using iterations

E.g., quantum Fourier transform:

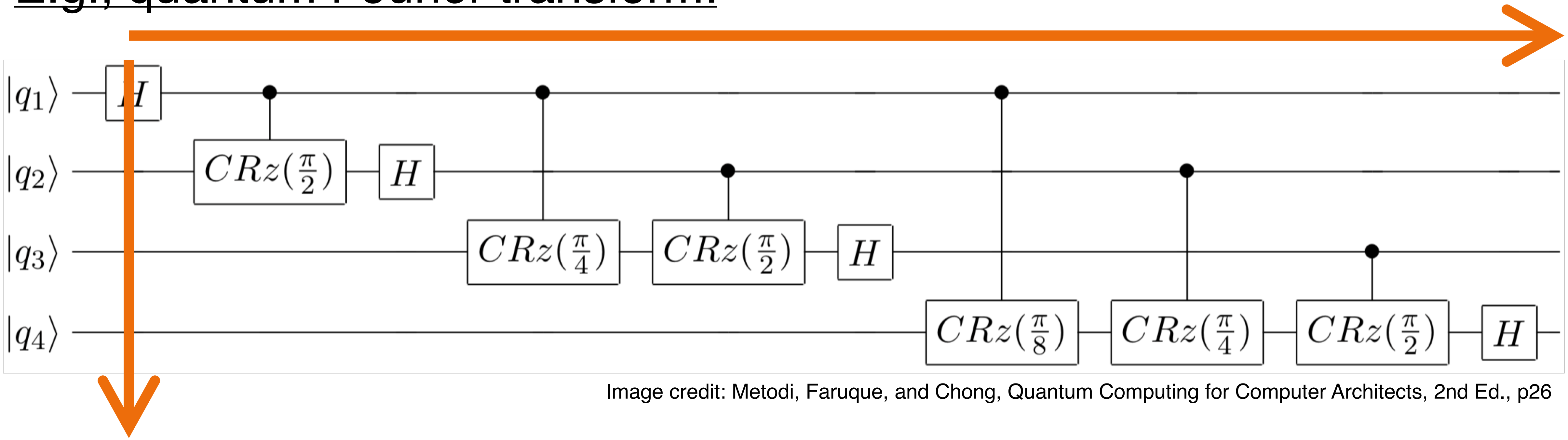


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Tricky iterations—two dimensional loop, indexing

Bug type 3-A: mistake in composing gates using iterations

E.g., quantum Fourier transform:

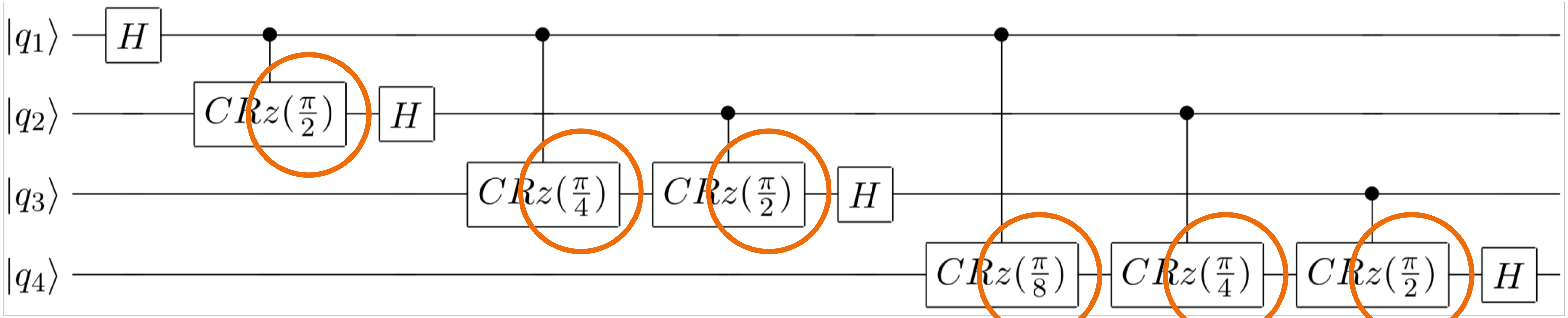


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Tricky iterations—two dimensional loop, indexing, bit shifting, endianness

Bug type 3-A: mistake in composing gates using iterations

E.g., quantum Fourier transform:

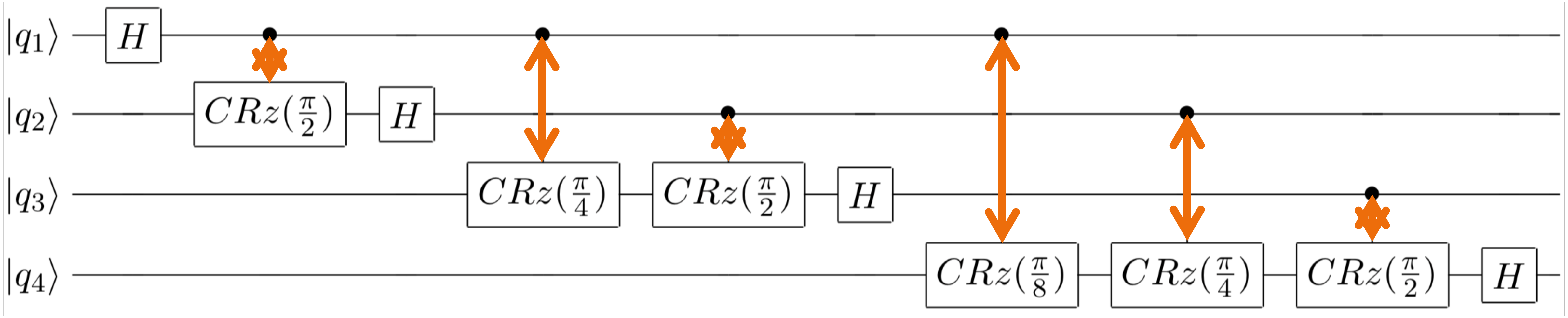


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Tricky iterations—two dimensional loop, indexing, bit shifting, endianness, control-target order…

Bug type 3-A: mistake in composing gates using iterations

E.g., Scaffold controlled adder:

```
module cADD (
    const unsigned int c_width, // number of control qubits
    qbit ctrl0, qbit ctrl1, // control qubits
    const unsigned int width, const unsigned int a, qbit b[]
) {
    for (int b_indx=width-1; b_indx>=0; b_indx--) {
        for (int a_indx=b_indx; a_indx>=0; a_indx--) {
            if ((a >> a_indx) & 1) { // shift out bits in constant a
                double angle = M_PI/pow(2,b_indx-a_indx); // rotation angle
                    switch (c_width) {
                        case 0: Rz ( b[b_indx], angle ); break;
                        case 1: cRz ( ctrl0, b[b_indx], angle ); break;
                        case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
}}}}}
```

Tricky iterations—two dimensional loop, indexing, bit shifting, endianness, control-target order…

Defense type 3-A: support for numeric data types

E.g., ProjectQ controlled adder:

```
def add_constant(eng, c, quint):

    with Compute(eng):
        QFT | quint

    for i in range(len(quint)):
        for j in range(i, -1, -1):
            if ((c >> j) & 1):
                R(math.pi / (1 << (i - j))) | quint[i]

    Uncompute(eng)
```

Defense type 3-A: support for numeric data types

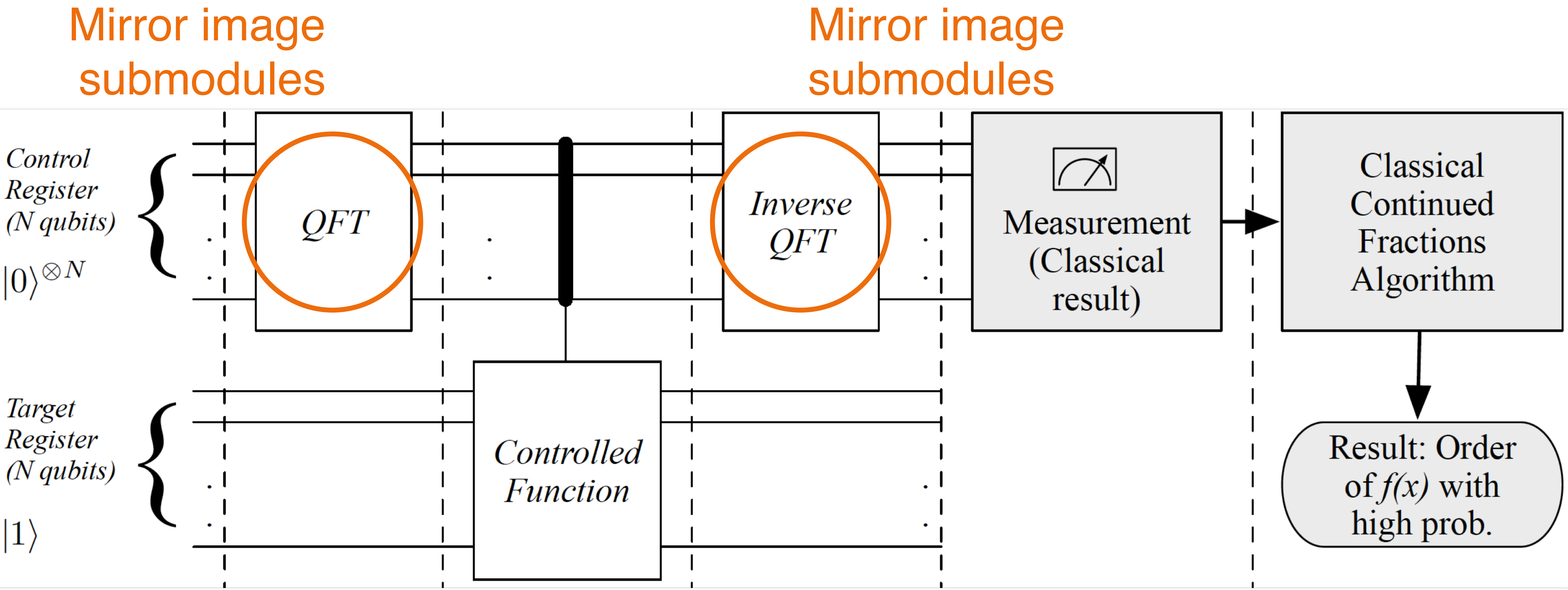
E.g., ProjectQ controlled adder:

```
def add_constant(eng, c, quint):

    with Compute(eng):
        QFT | quint

    for i in range(len(quint)):
        for j in range(i, -1, -1):
            if ((c >> j) & 1):
                R(math.pi / (1 << (i - j))) | quint[i]

    Uncompute(eng)
```

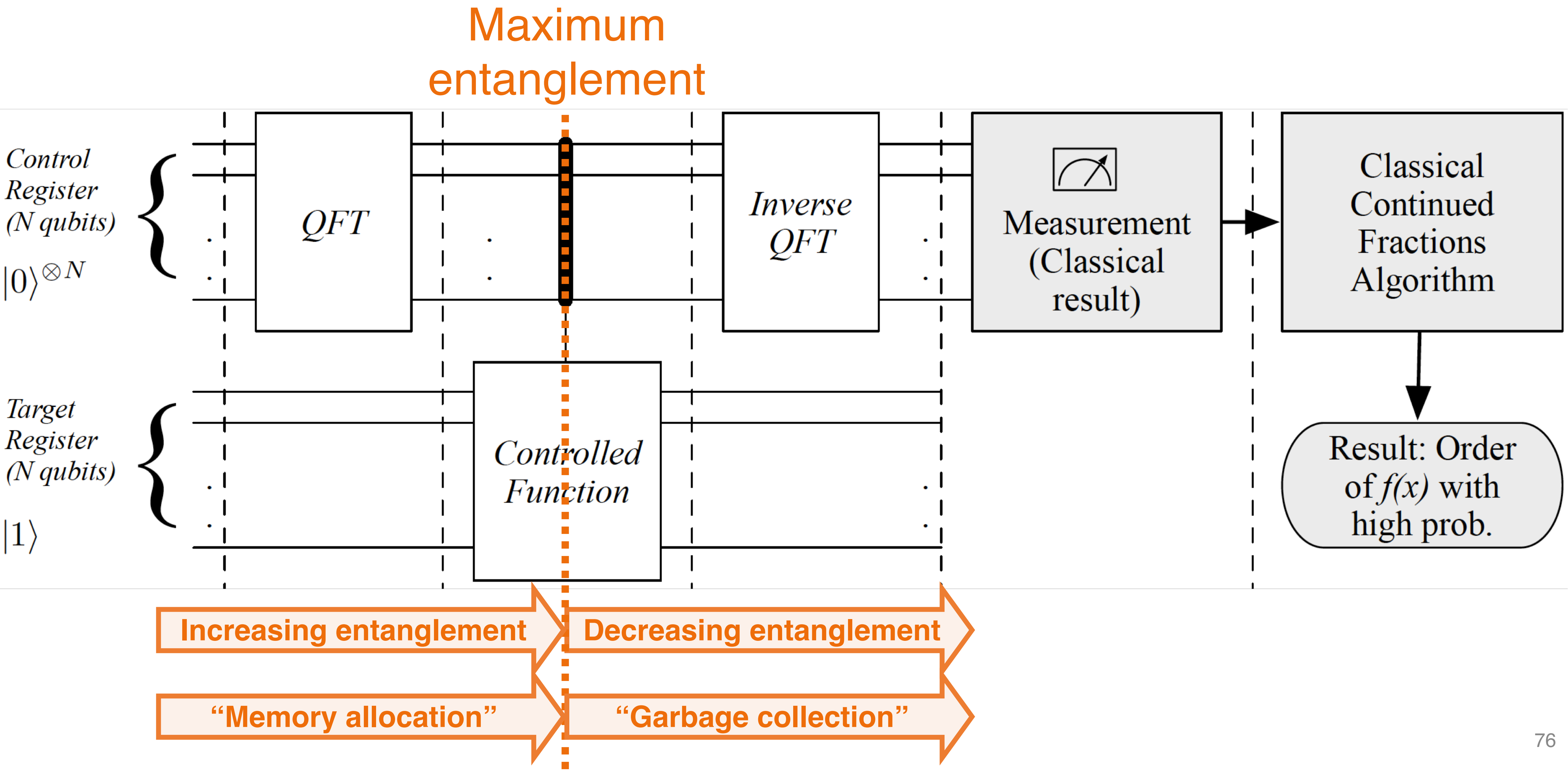
Greater abstraction than raw qubits

Language support for numerical data types reduces confusion

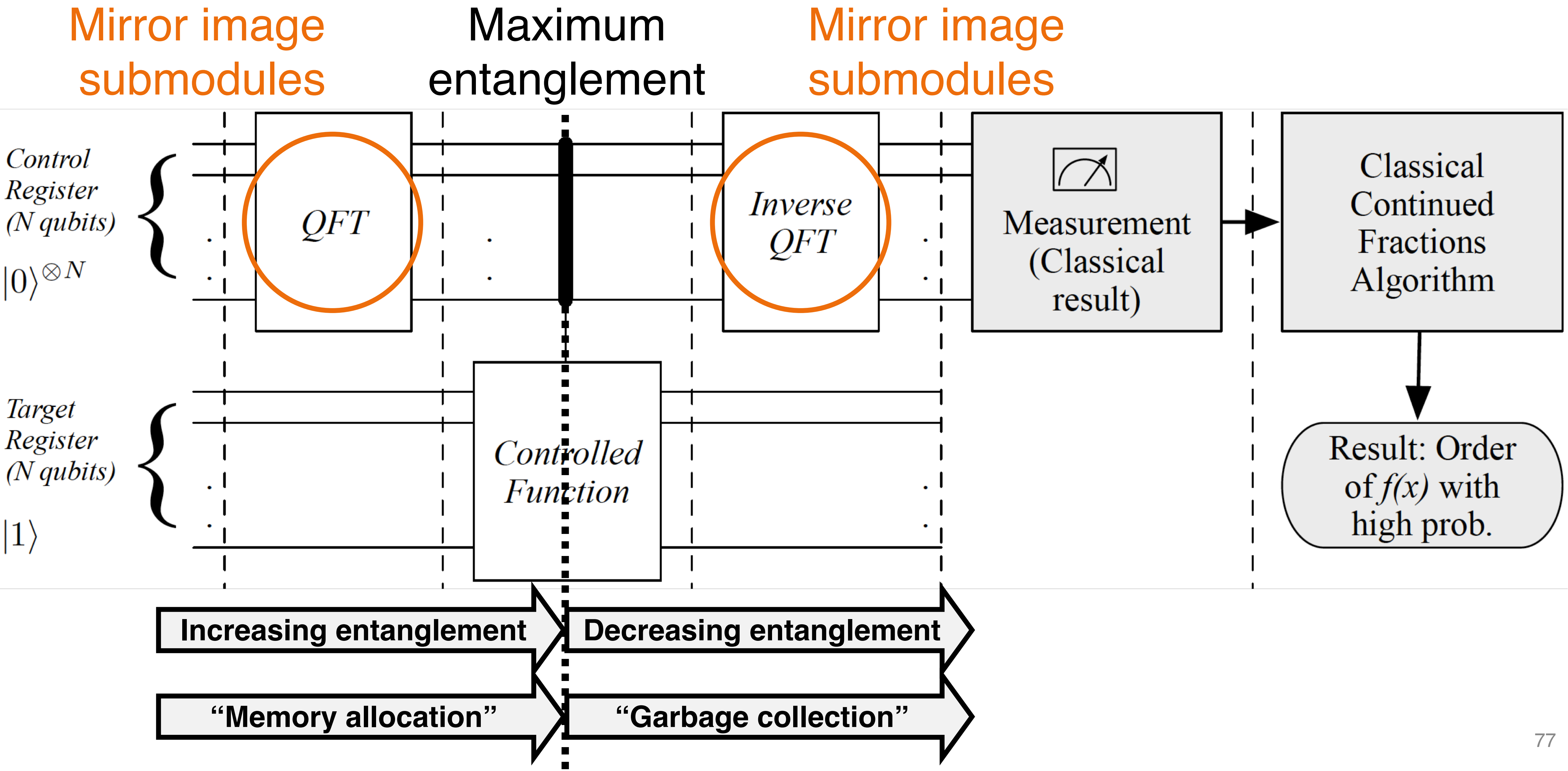
Bug type 3-B: mistake in composing gates using mirroring



Bug type 3-B: mistake in composing gates using mirroring



Bug type 3-B: mistake in composing gates using mirroring



Bug type 3-B: mistake in composing gates using mirroring

E.g., Scaffold controlled adder:

```
module cADD (
    const unsigned int c_width, // number of control qubits
    qbit ctrl0, qbit ctrl1, // control qubits
    const unsigned int width, const unsigned int a, qbit b[]
) {
    for (int b_indx=width-1; b_indx>=0; b_indx--) {
        for (int a_indx=b_indx; a_indx>=0; a_indx--) {
            if ((a >> a_indx) & 1) { // shift out bits in constant a
                double angle = M_PI/pow(2,b_indx-a_indx); // rotation angle
                switch (c_width) {
                    case 0: Rz ( b[b_indx], angle ); break;
                    case 1: cRz ( ctrl0, b[b_indx], angle ); break;
                    case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
}}}}}
```

Mirror image subroutines need careful reversal of each operation and each iteration.

Defense type 3-B: support for reversible computation

E.g., ProjectQ controlled adder:

```
def add_constant(eng, c, quint):

    with Compute(eng):
        QFT | quint

    for i in range(len(quint)):
        for j in range(i, -1, -1):
            if ((c >> j) & 1):
                R(math.pi / (1 << (i - j))) | quint[i]

    Uncompute(eng)
```

Language support for automatically generating reversed computation cuts mistakes, lines of code

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. **Composing operations**
   A. **Iteration**
   B. **Mirroring**
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. **Quantum specific language support**
   A. **Numeric data types**
   B. **Reversible computation**

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. **Classical input parameters**
5. **Garbage collection of qubits**

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation

Classical input parameters for Shor's factoring algorithm

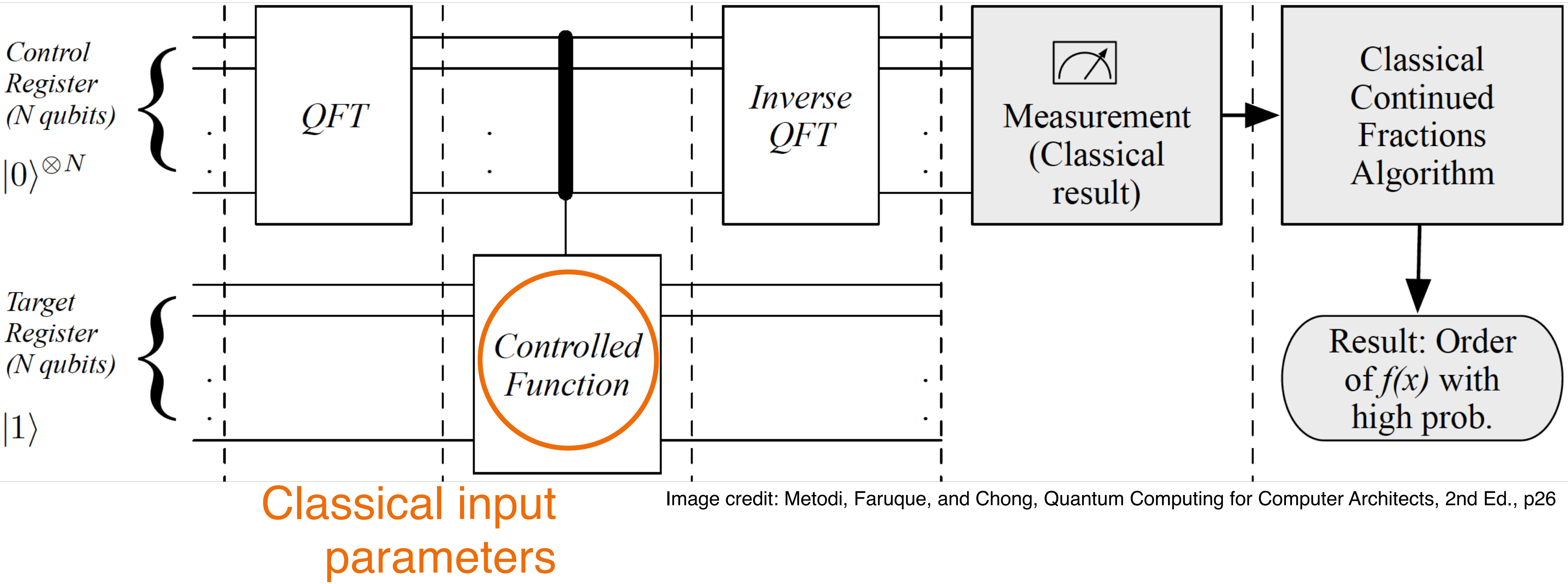
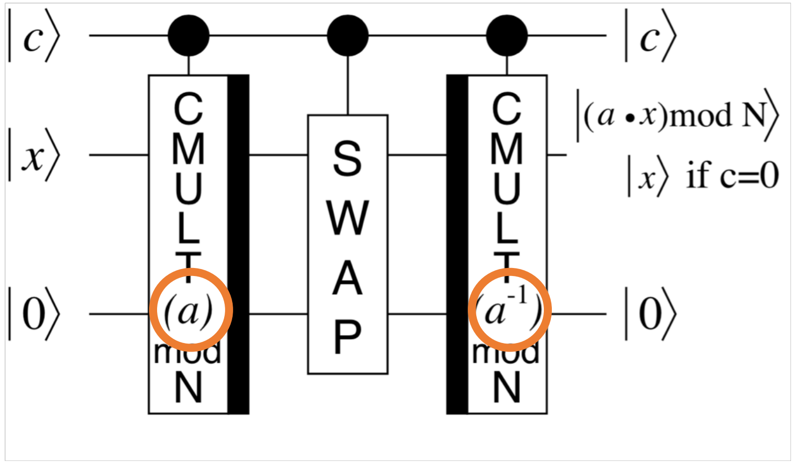


Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Classical input parameters for Shor's factoring algorithm

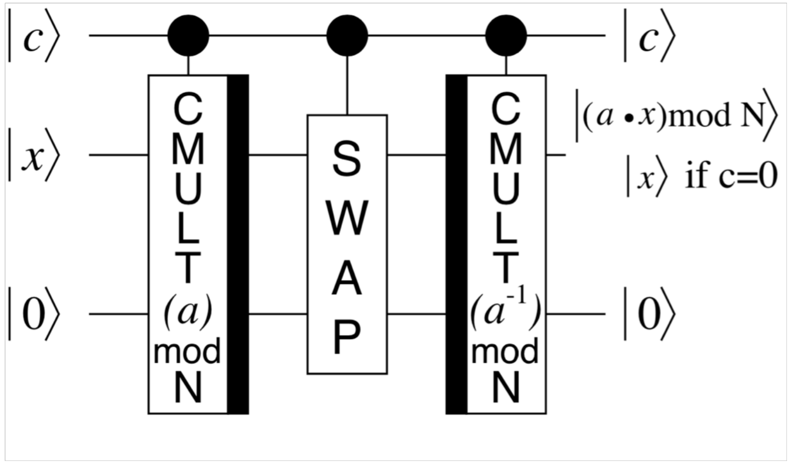


Classical input parameters

Image credit: Beauregard, 2003

k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	a^{-1}, $a \times a^{-1} \equiv 1 \bmod 15$

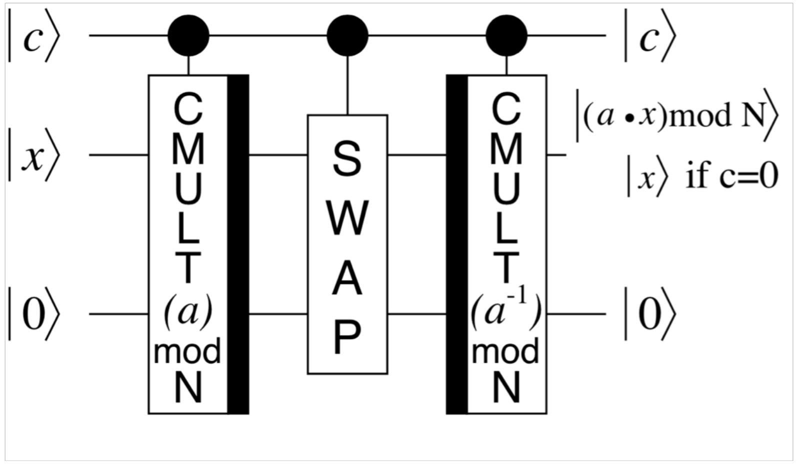
Classical input parameters for Shor's factoring algorithm



A guess number: 7 Number to factor: 15

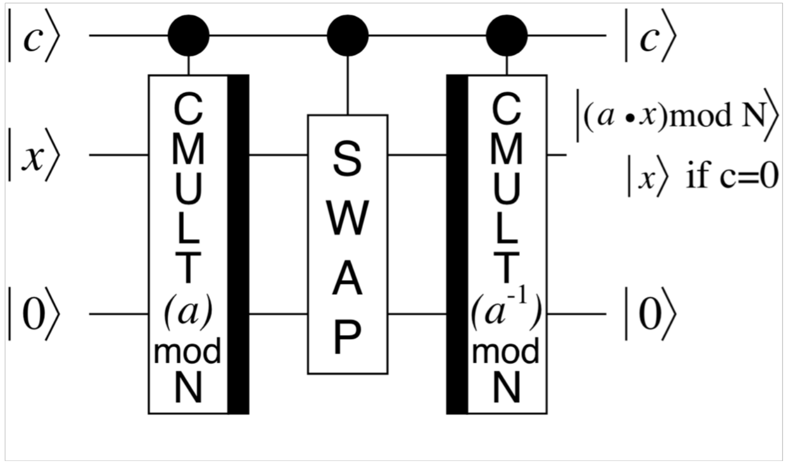
k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$

Classical input parameters for Shor's factoring algorithm



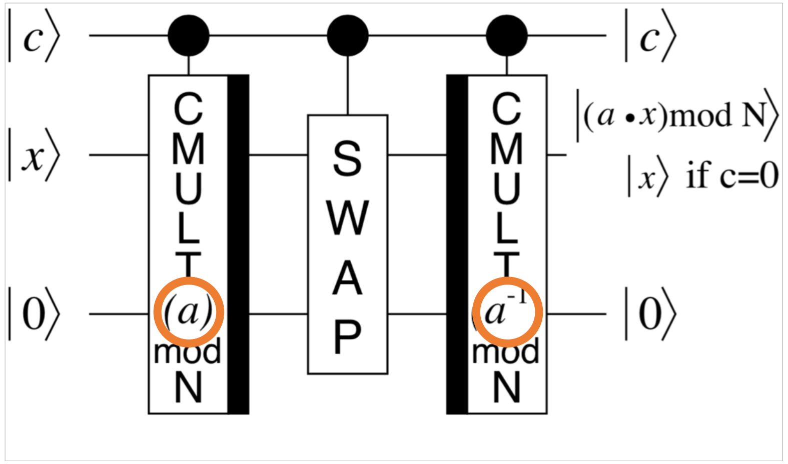
k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$
0	7	13

Classical input parameters for Shor's factoring algorithm



k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$
0	7	13
1	4	4

Classical input parameters for Shor's factoring algorithm



Classical input parameters

k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$
0	7	13
1	4	4
2	1	1
3	1	1
…	…	…

Output measurement for Shor's factoring algorithm

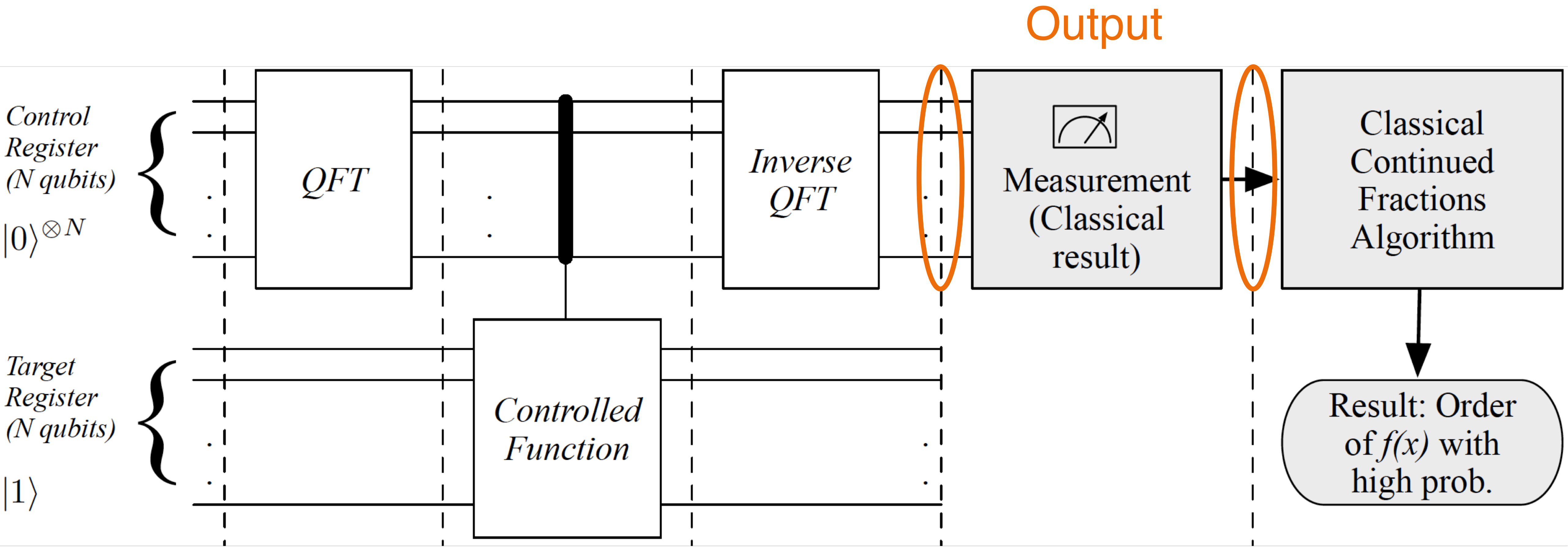


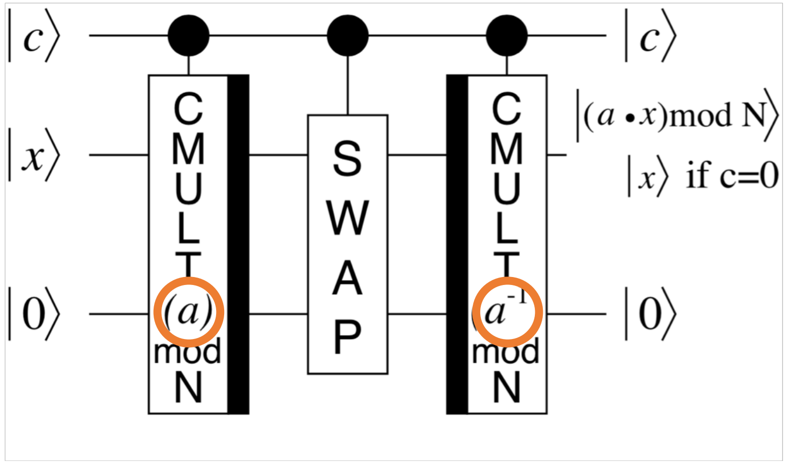
Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

Output measurement for Shor's factoring algorithm

	output							
	0	1	2	3	4	5	6	7
probability	1/4	0	1/4	0	1/4	0	1/4	0

Shor's factoring ancilla and output with good inputs

Bug type 4: incorrect classical input parameters



Suppose incorrect input

k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$
0	7	~~13~~ **12**
1	4	4
2	1	1
3	1	1
…	…	…

Defense type 4: algorithm progress checks

	output							
	0	1	2	3	4	5	6	7
probability	3/16	1/16	3/16	1/16	3/16	1/16	3/16	1/16

Shor's factoring ancilla and output with bad inputs

Algorithm progress checks (integration testing) detect errors in classical input parameters.

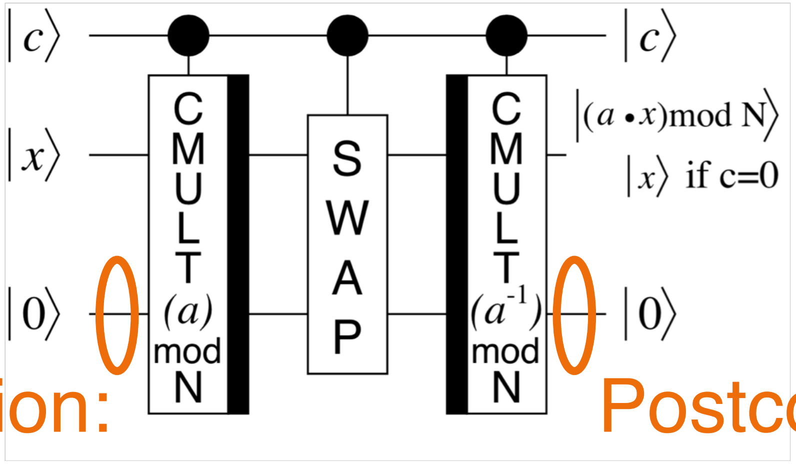
Defense type 4: algorithm progress checks

	output							
	0	1	2	3	4	5	6	7
probability	3/16	~~1/16~~	3/16	~~1/16~~	3/16	~~1/16~~	3/16	~~1/16~~

Shor's factoring ancilla and output with bad inputs

Are there other symptoms we can observe??

Bug type 5: incorrect garbage collection of qubits

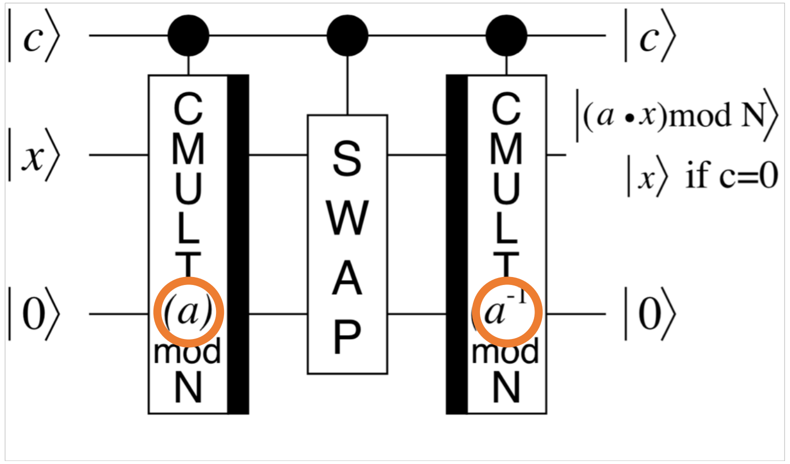


Precondition:
ancilla = zero

Postcondition:
ancilla = zero

Reversed computation needed to properly disentangle (garbage collect) temporary qubits.

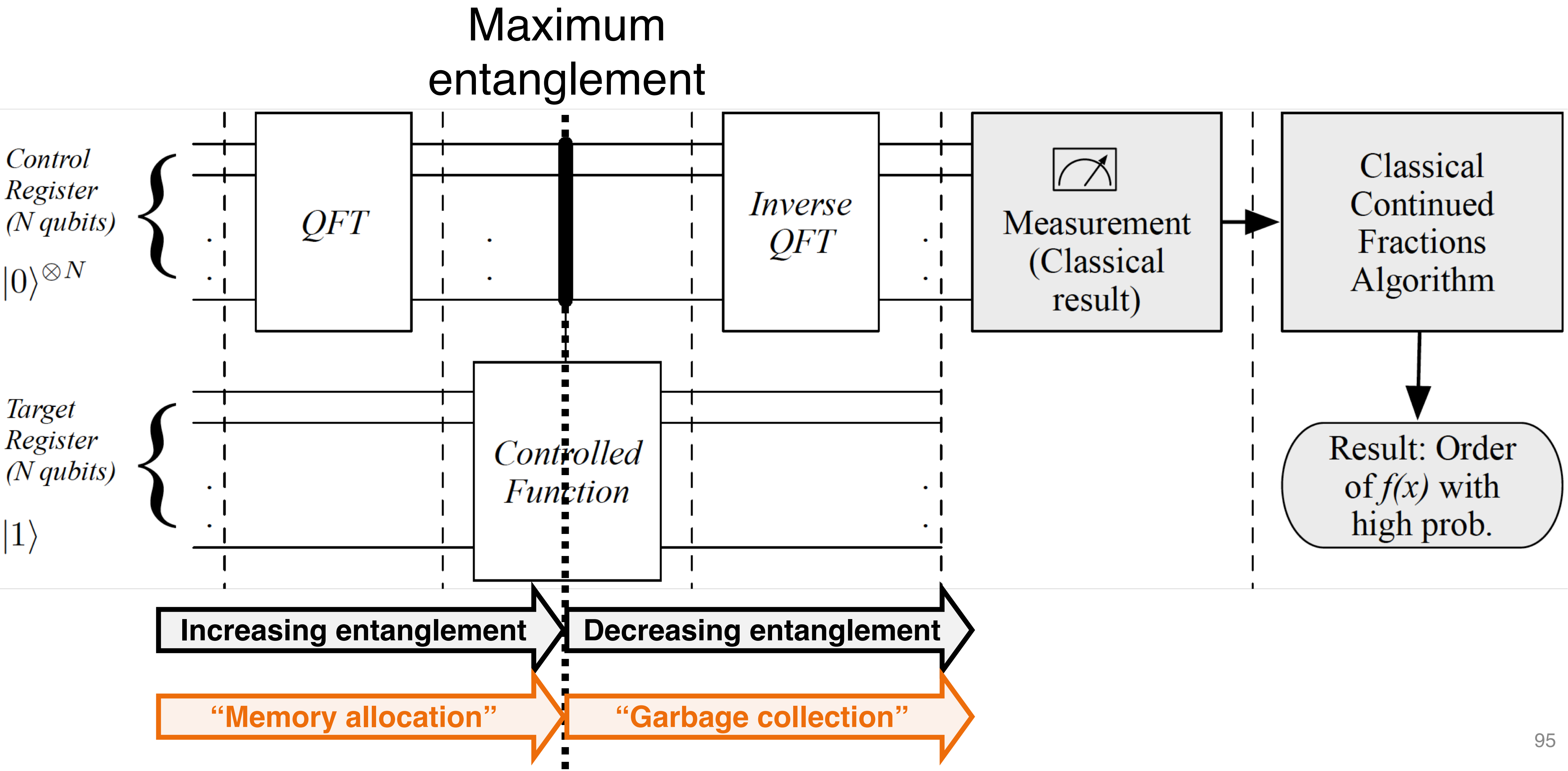
Bug type 5: incorrect garbage collection of qubits



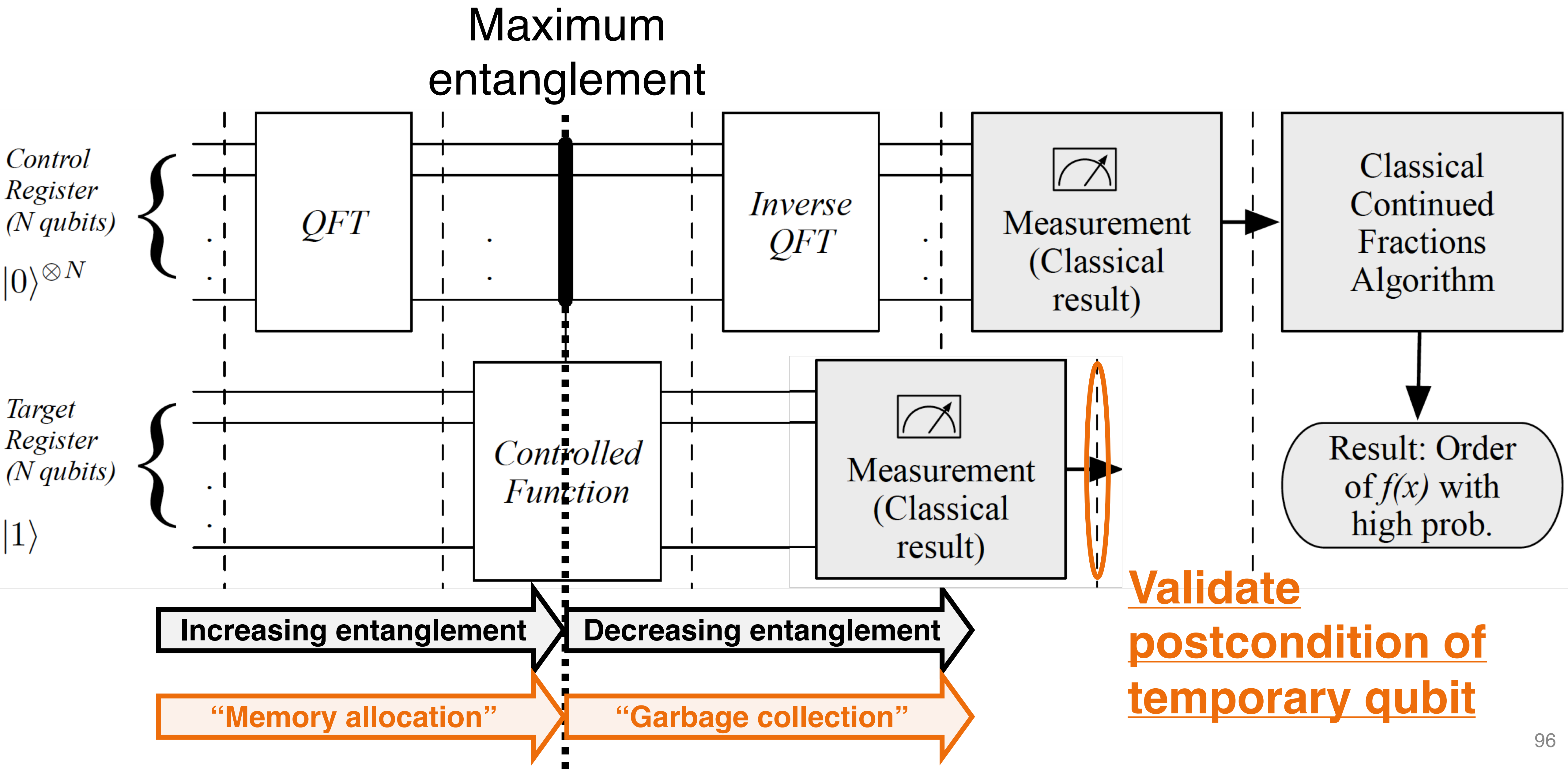
Incorrect reversed computation, incorrect garbage collection

k, the algorithm iteration	$a = 7^{2^k} \bmod 15$	$a^{-1}; a \times a^{-1} \equiv 1 \bmod 15$
0	7	~~13~~ 12
1	4	4
2	1	1
3	1	1
…	…	…

Defense type 5: check for postcondition assertions



Defense type 5: check for postcondition assertions



Validate postcondition of temporary qubit

Defense type 5: check for postcondition assertions

probability		output 0	1	2	3	4	5	6	7
temporary variable	0	1/8	0	1/8	0	1/8	0	1/8	0
	4	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

Defense type 5: check for postcondition assertions

probability		output							
		0	1	2	3	4	5	6	7
temporary variable	0	1/8	0	1/8	0	1/8	0	1/8	0
	4	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

P(temporary variable=0) = 0.5

Indicates algorithm failed

Defense type 5: check for postcondition assertions

probability		output							
		0	1	2	3	4	5	6	7
temporary variable	0	1/8	0	1/8	0	1/8	0	1/8	0
	4	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

P(temporary variable=0) = 0.5

Indicates algorithm failed

Postcondition check on temporary qubits detects errors in garbage collection.

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. <u>**Classical input parameters**</u>
5. <u>**Garbage collection of qubits**</u>

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation
4. <u>**Algorithm progress assertions**</u>
5. <u>**Postconditions**</u>

Quantum program bug types

1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation
4. Algorithm progress assertions
5. Postconditions

A first taxonomy of quantum program bugs and defenses.

This paper is about quantum PL support for correctness

Detailed debugging effort across quantum algorithms

Quantum chemistry, Shor's factoring, Grover's search

Where possible, validate across quantum languages

Scaffold, ProjectQ, QISKit… compare correctness features

Classify quantum bugs in input, operations, and output

Paired with defenses: unit testing, syntax support, assertions

Quantum algorithms
Quantum programming languages
Quantum programming patterns and antipatterns: bugs and defenses
Building blocks: qubits, gates, circuits
Quantum physical devices