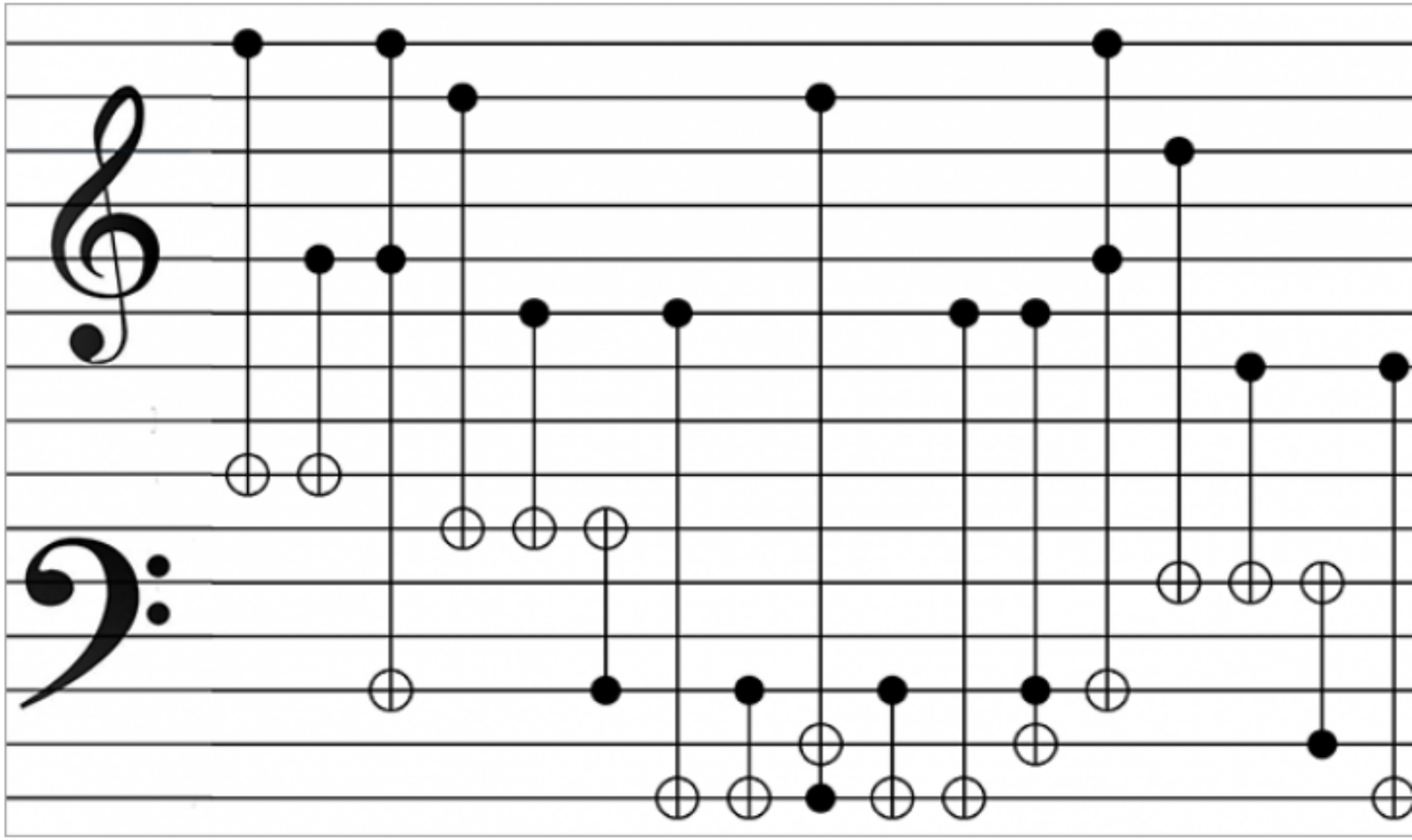
Microsoft

Reversible Computing



Mariia Mykhailova
Principal Software Engineer
Microsoft Quantum Systems

Lecture outline

- Reversible Boolean logic
- Reversible circuit synthesis
- Quantum oracles
- Example: implement oracle for SAT problem

Microsoft

Reversible Boolean logic

Why must quantum gates be reversible?

A reversible gate has a *unique* input state for every output state (i.e., it is a 1:1 mapping of inputs and outputs)

Quantum mechanics dictates that the evolution of a quantum system must be reversible (except measurement)

Quantum computation is a directed evolution of a quantum system

Quantum states are vectors

Quantum gates are unitary matrices:

$$UU^\dagger = U^\dagger U = I$$

†: Adjoint, aka complex conjugate transpose, aka inverse

Is classical Boolean logic reversible?

Are any of these gates reversible?

AND:

x_0	x_1	$x_0 \wedge x_1$
0	0	0
0	1	0
1	0	0
1	1	1

OR:

x_0	x_1	$x_0 \vee x_1$
0	0	0
0	1	1
1	0	1
1	1	1

XOR:

x_0	x_1	$x_0 \oplus x_1$
0	0	0
0	1	1
1	0	1
1	1	0

NOT gate:

x_0	$\neg x_0$
0	1
1	0

NAND gate:

x_0	x_1	$\neg(x_0 \wedge x_1)$
0	0	1
0	1	1
1	0	1
1	1	0

Reversible NOT: Pauli X gate

$$X|\psi\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_0 \end{pmatrix}$$

$$X|\psi\rangle = X(c_0|0\rangle + c_1|1\rangle) = c_0|1\rangle + c_1|0\rangle$$

$$|0\rangle \rightarrow |1\rangle, |1\rangle \rightarrow |0\rangle$$

$$|a\rangle \rightarrow |\neg a\rangle$$

Quantum implementation of the classical NOT gate

Reversible XOR

XOR gate:

x_0	x_1	$x_0 \oplus x_1$
0	0	0
0	1	1
1	0	1
1	1	0

$$00 \rightarrow 00$$
$$01 \rightarrow 01$$
$$10 \rightarrow 11$$
$$11 \rightarrow 10$$

Note that keeping one of the input bits as part of the output yields a unique output.

You could interpret this as flipping (applying X) to the second bit when the first bit is 1.

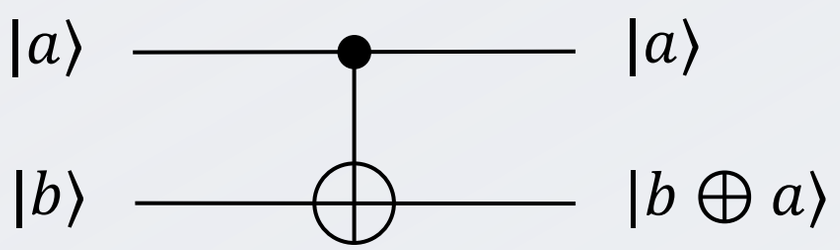
Reversible XOR: controlled-NOT gate

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$|a, b\rangle \rightarrow |a, b \oplus a\rangle$$

If a is 1, then b is flipped
$\Rightarrow$ b undergoes an X gate

$$00 \rightarrow 00$$
$$01 \rightarrow 01$$
$$10 \rightarrow 11$$
$$11 \rightarrow 10$$



Reversible AND

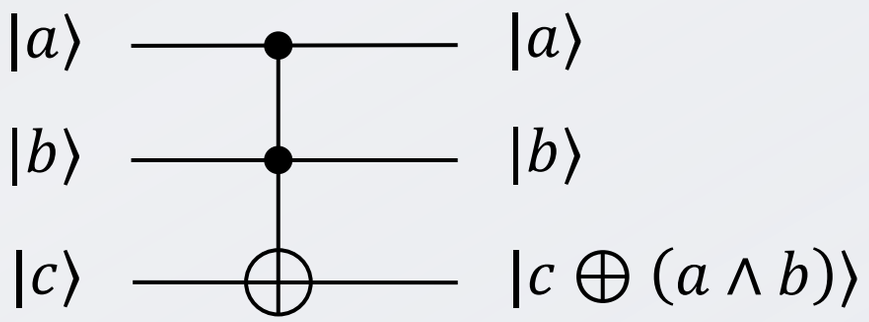
We must keep both inputs and write output to a new bit to make AND reversible

	x_0	x_1	$x_0 \wedge x_1$
AND:	0	0	0
	0	1	0
	1	0	0
	1	1	1

Toffoli (CCNOT) gate:

flips the third (qu)bit if and only if the first two (qu)bits are both $|1\rangle$:

$$|a, b, c\rangle \rightarrow |a, b, c \oplus (a \wedge b)\rangle$$



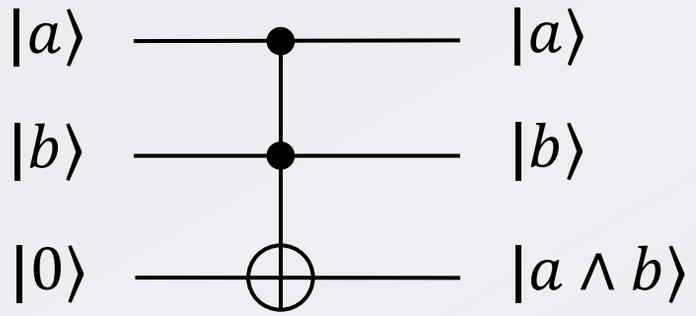
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Universality of Toffoli

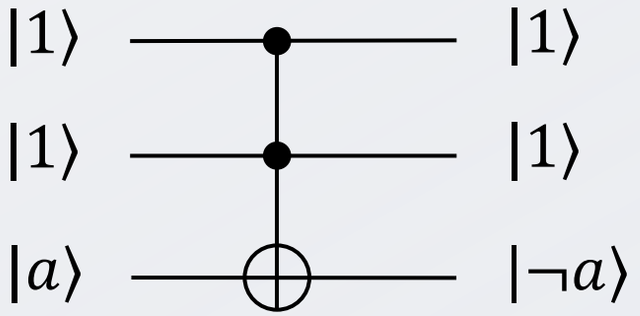
A Toffoli gate can be used to simulate AND, NOT, and FANOUT gates

Assuming we have access to extra qubits in both basis states;
otherwise (if all allocated qubits start in the same state), we need a NOT gate in addition

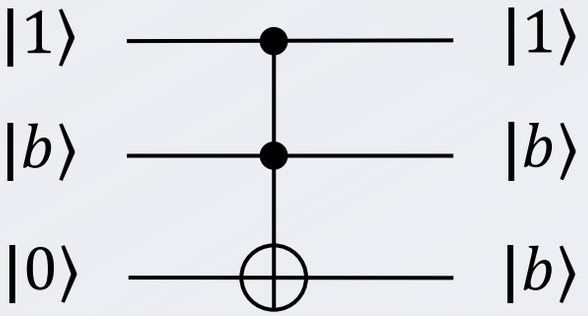
AND



NOT



FANOUT

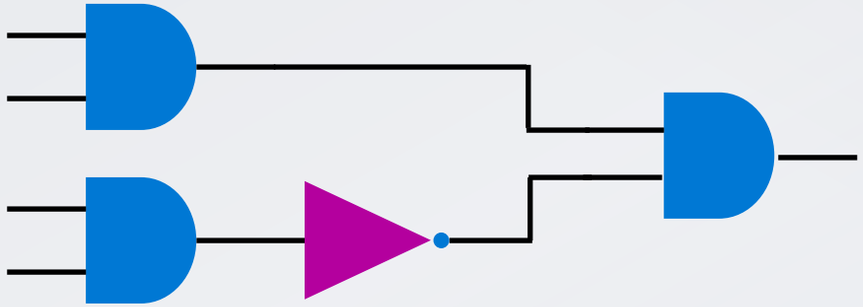


Microsoft

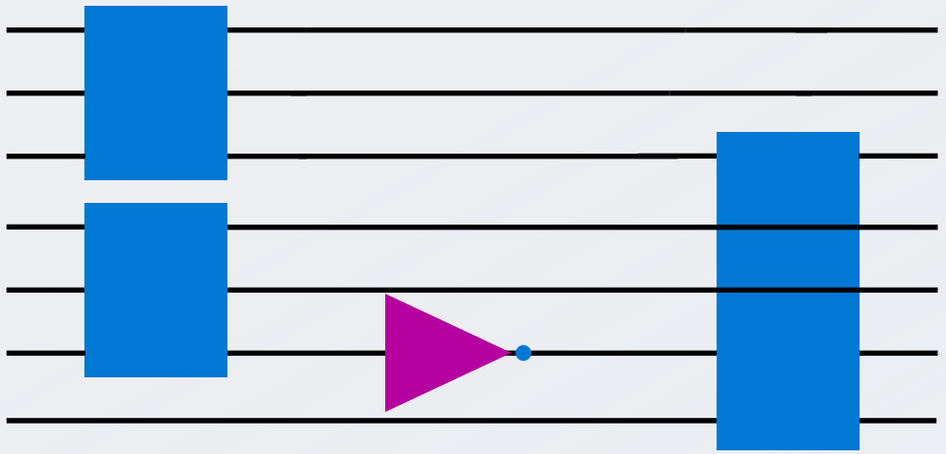
Reversible circuit synthesis

Convert classical functions to quantum circuits

Break down to primitive classical gates:



Replace each classical gate with a reversible equivalent:



Convert a classical circuit to a quantum circuit

Replace each gate with a reversible one

Side-effect:

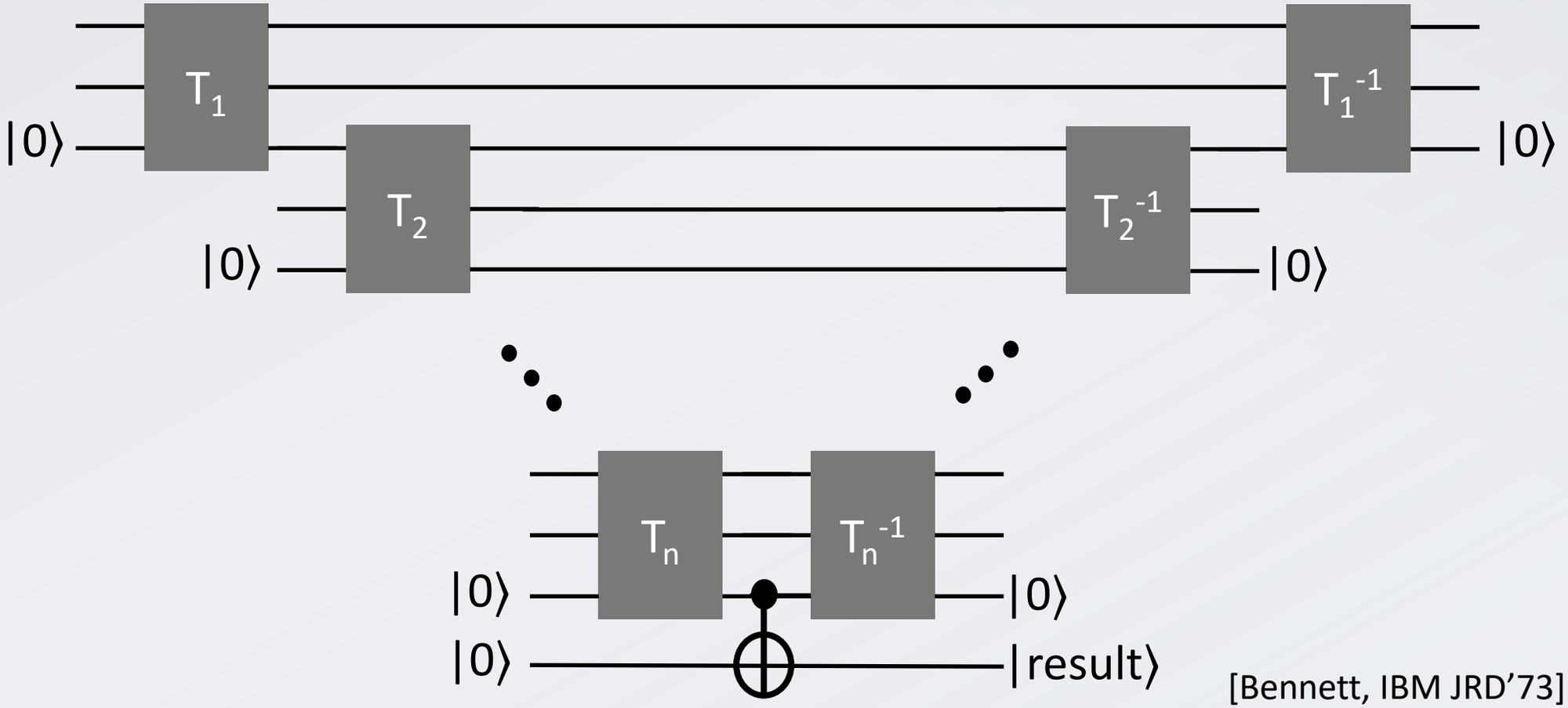
- "Garbage" (scratch or "ancilla") qubits that end up in a non-$|0\rangle$ state (entangled with main qubits)
- Fine for reversible classical computing, **BAD** for quantum computing (prevents interference)

Solution ("Bennett trick"):

Run the computation, copy the result into a new qubit, then "uncompute" the garbage by running the computation backwards

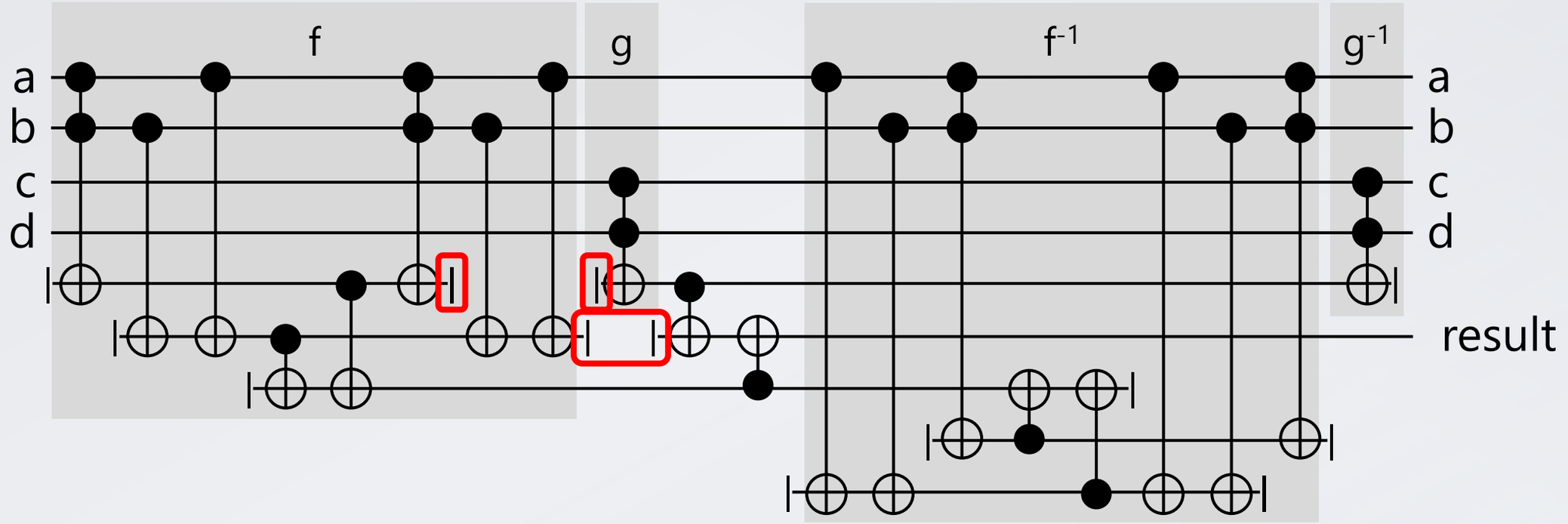
$$
\begin{aligned}
|x\rangle\,|0\rangle\,|0\rangle\,|0\rangle &\longmapsto |x\rangle|f(x)\rangle|garbage(x)\rangle|0\rangle \\
&\longmapsto |x\rangle|f(x)\rangle|garbage(x)\rangle|f(x)\rangle \\
&\longmapsto |x\rangle|0\rangle\,|0\rangle\,|f(x)\rangle
\end{aligned}
$$

Bennett trick



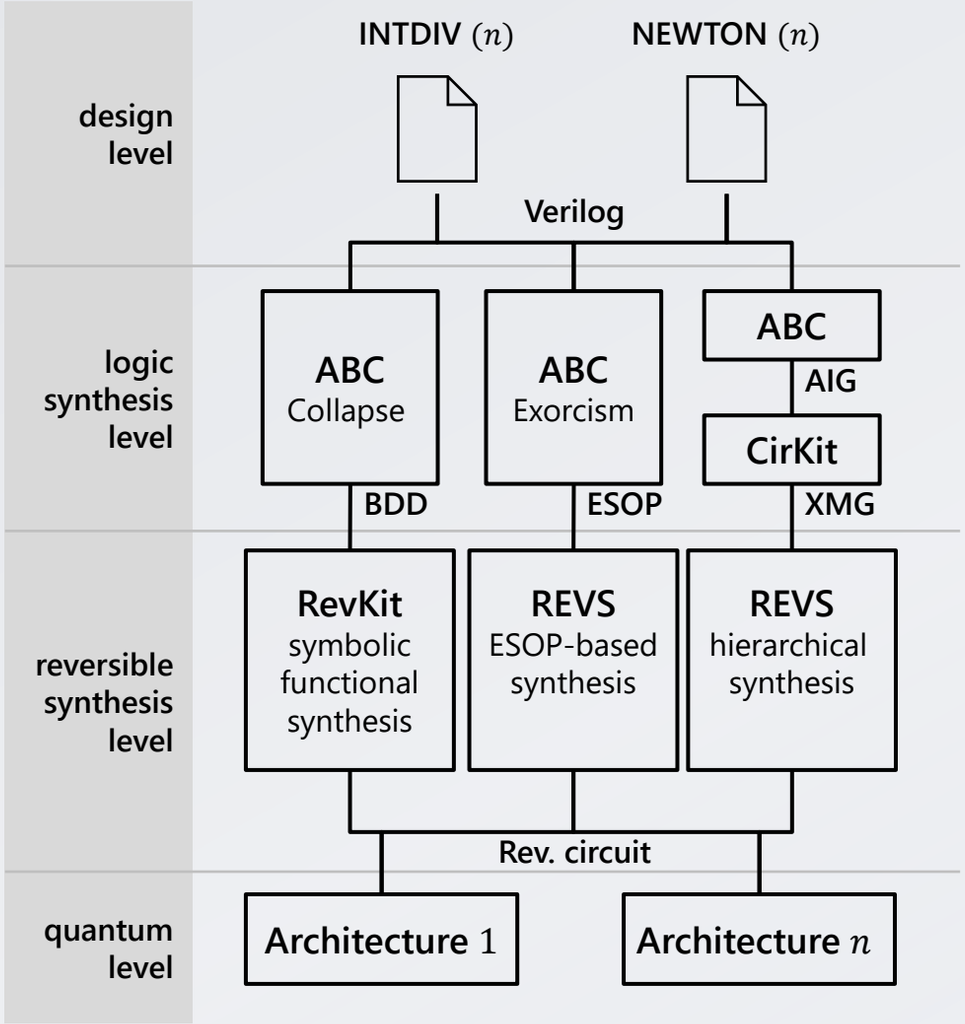
[Bennett, IBM JRD'73]
[Bennett, SIAM J. Comp., 1989]

Reusing qubits



- all scratch qubits are returned to the $|0\rangle$ state (indicated by "|")
- some scratch qubits are reused in the circuit (red frames above)
- space savings comparted to alternative methods (e.g., original Bennett trick)

Circuit synthesis for classical subroutines



Example: integer division $x \mapsto 2^n/x$,

x is an n-bit (unsigned) integer; result is rounded to the closest integer.

Design:

High-level implementation of division in Verilog:

- Integer long division (divide $2^n = qx + r$)
- Newton-Raphson

Logic synthesis:

- Convert Verilog to logical netlist in AIG format (And-Inverter Graphs) using tool ABC
- Convert AIG to ESOP format (Exclusive Sums of Products) using tool XOR-cism
- Convert ESOP to Toffoli networks using different tools (REVS, RevKit)

Several passes for various parameter settings allow T-count/space/ compile time tradeoffs.

[Soeken et al., arXiv:1612.00631]

Q# tools: Toffoli simulator

https://docs.microsoft.com/quantum/machines/toffoli-simulator

Supports only X, CNOT and multi-controlled X gates

Represents state as an array of bits

No superposition or entanglement! Acts only on basis states

Useful for verifying reversible circuits

If the results on the basis states are correct and only these quantum gates were used (no measurements and no relative phases), the result on any superposition states will also be correct

Microsoft

Exercises

Exercises

1. Implement the following function using CCNOT gates:

$$c_1, c_2, \ldots, c_n, 0_1, 0_2, \ldots, 0_{n-1} \rightarrow$$
$$c_1, c_2, \ldots, c_n, c_1 \wedge c_2, c_1 \wedge c_2 \wedge c_3, \ldots, c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

2. Implement the following function using CCNOT gates:

$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

3. You're given implementations of functions $f_1(c)$, $f_2(c)$:

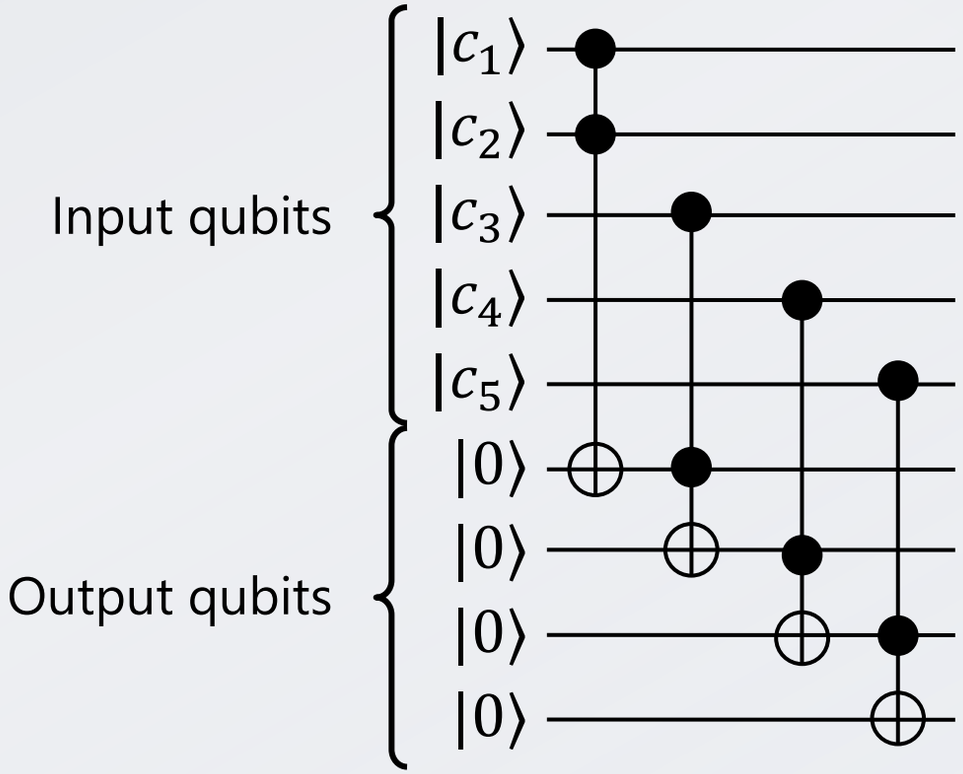
$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus f_i(c)$$

Implement the function

$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus (f_1(c) \wedge f_2(c))$$

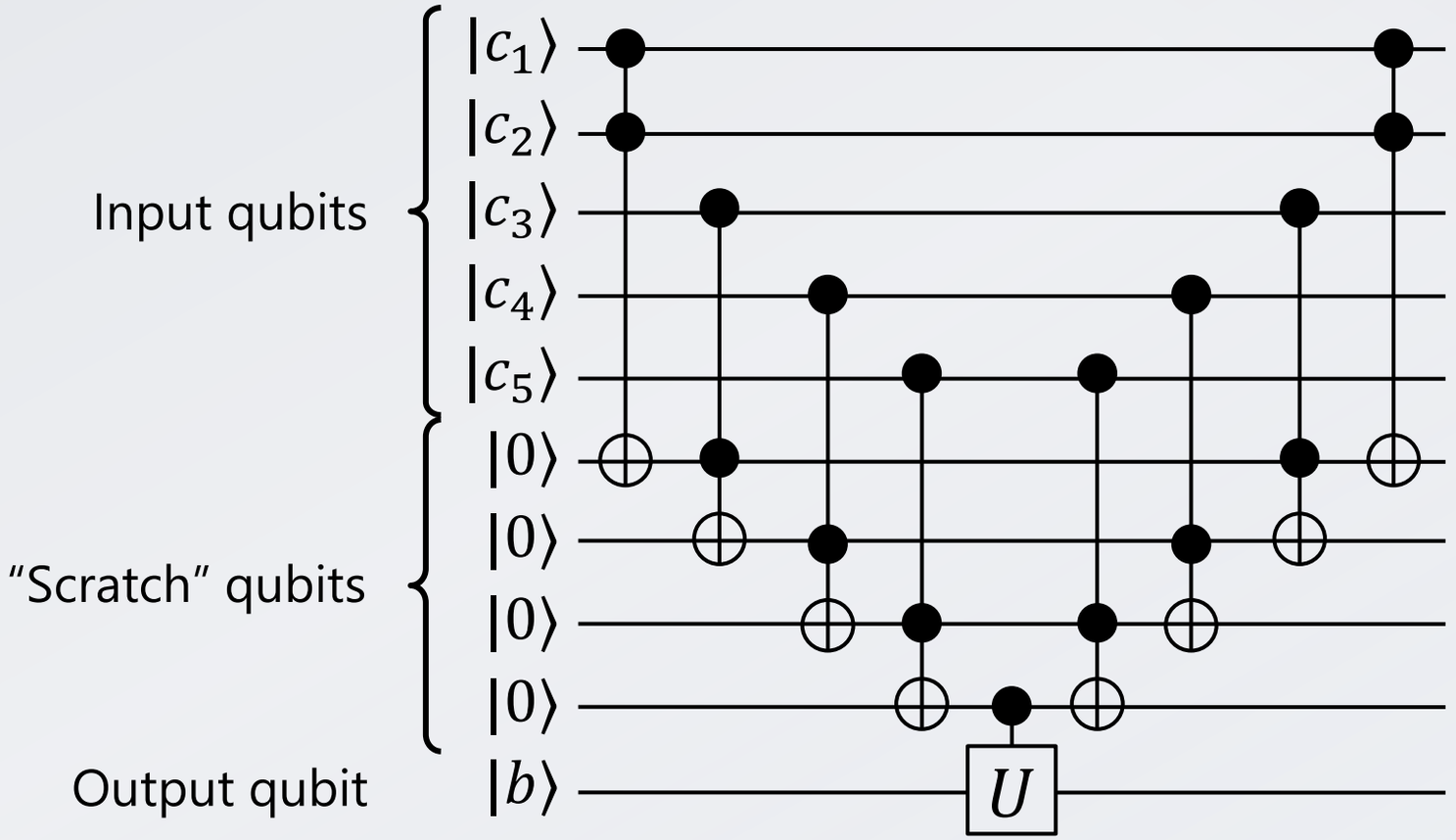
Exercise 1

$$c_1, c_2, \ldots, c_n, 0_1, 0_2, \ldots, 0_{n-1} \rightarrow$$
$$c_1, c_2, \ldots, c_n, c_1 \wedge c_2, c_1 \wedge c_2 \wedge c_3, \ldots, c_1 \wedge c_2 \wedge \cdots \wedge c_n$$



Exercise 2: Multi-controlled NOT (or another gate)

$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus c_1 \wedge c_2 \wedge \cdots \wedge c_n$$



Network implementing the Cn(U) operation
[Nielsen/Chuang, p. 184]

Exercise 3: Combining multiple functions

You're given implementations of functions $f_1(c)$, $f_2(c)$:

$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus f_i(c)$$

Implement the function

$$c_1, c_2, \ldots, c_n, b \rightarrow c_1, c_2, \ldots, c_n, b \oplus (f_1(c) \wedge f_2(c))$$

1. Allocate 2 extra qubits a_1 and a_2
2. Compute $f_i(c)$ into a_i
3. CCNOT with a_1 and a_2 as controls and b as target
4. Uncompute $f_i(c)$ and release a_i

Microsoft

Quantum oracles

Classical (non-quantum) oracles

A function f is given to us as a "black box" or oracle:

Goal:

We want to determine information about f by using the black box (or "querying it") as few times as possible.

Think of f as an actual function written in some language.

Python

```
def f(x):
    return abs(x-2)+1
```

C#

```
int f(int x)
{return Math.Abs(x-2)+1;}
```

White box vs. black box

Example:

Given $f: \{1, \ldots, N\} \to \{1, \ldots, N\}$. Find $\min_x f(x)$.

Best black box algorithm:

Query $f(1), f(2), \ldots, f(N)$. Output the smallest number returned by the oracle.

We can do better if we know f's source code! (white box)

Python

```
def f(x):
    return abs(x-2)+1
```

C#

```
int f(int x)
{return Math.Abs(x-2)+1;}
```

Minimum value of this function is $f(2) = 1$.

Why treat f as a black box?

- Maybe it really is a black box to us? f is computed by someone else (e.g., on the cloud, a remote database, etc.)
- We have the source code of f, but it is obfuscated
- We have the source code of f, but we don't know how to use it to make our goal easier
- We want to design an algorithm so general that it does not depend on f's source code at all (i.e., works for all f)!

Quantum oracles: general definition

Quantum oracle **is a black box** ***unitary*** **operation U_f that implements some classical function $f(x)$**

$$f : \{0,1\}^N \to \{0,1\}$$ (N-bit input, 1-bit output)

U_f implements $f(x)$ in a ***reversible*** **way**

The oracle must work properly on all superposition states

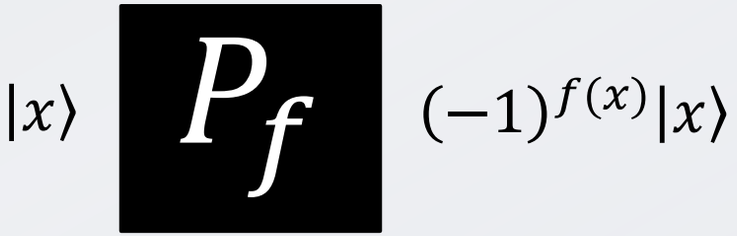
Review: Phase oracles

Phase oracles encode $f(x)$ into the phase of the state

$$U_f\,|x\rangle = (-1)^{f(x)}|x\rangle$$

If $f(x) = 0$, the phase doesn't change

If $f(x) = 1$, the phase is multiplied by -1

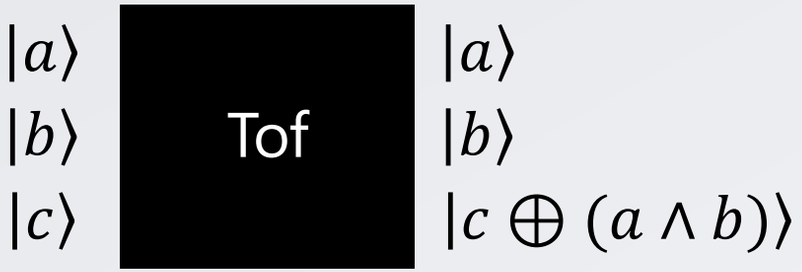


Behavior on superposition states follows from linearity of the oracle:

$$U_f \sum c_i|x_i\rangle = \sum c_i U_f|x_i\rangle = \sum (-1)^{f(x_i)} c_i|x_i\rangle$$

Building quantum oracles

Given a classical circuit that calculates a function, we can convert it to quantum using reversible computing tricks

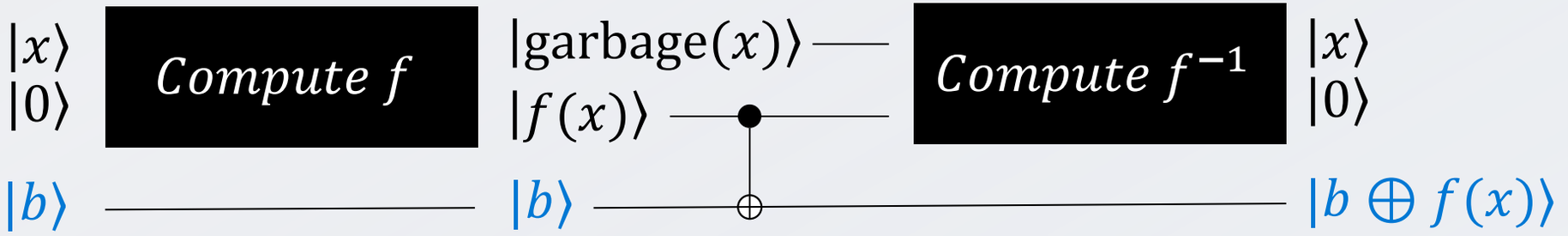


Quantum Toffoli gate

Classical AND gate

Classical NOT gate

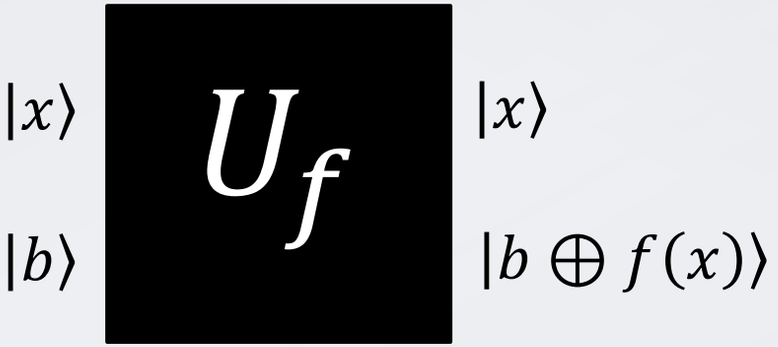
Results in a quantum circuit like this for $f: \{0,1\}^n \to \{0,1\}$:



Marking oracles

This construction yields the following quantum oracle:

also known as the "bit flip oracle" or "marking oracle"



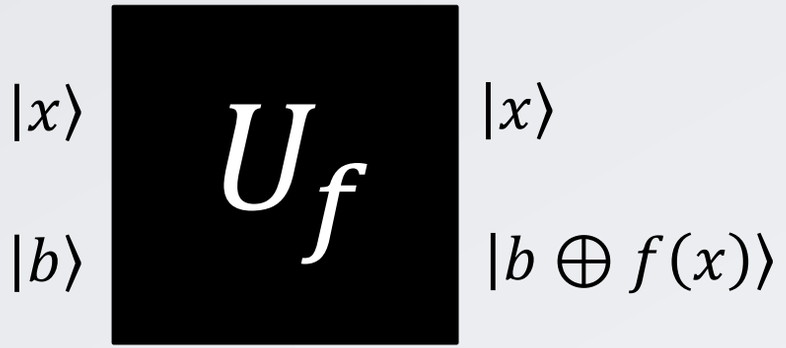
Can we define a quantum oracle that maps $|x\rangle \rightarrow |f(x)\rangle$?

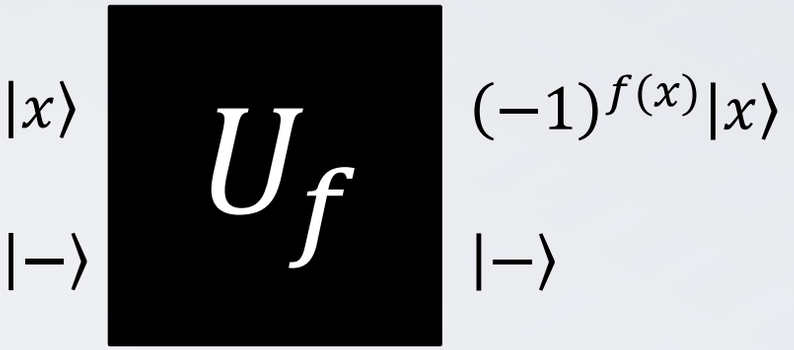
Not necessarily unitary and not necessarily reversible

If $f(x) = x$, such oracle will be unitary (implemented by identity gate)

If $f(x) = 0$ (or, more generally, has collisions), such oracle won't be unitary

Convert marking oracle to phase oracle



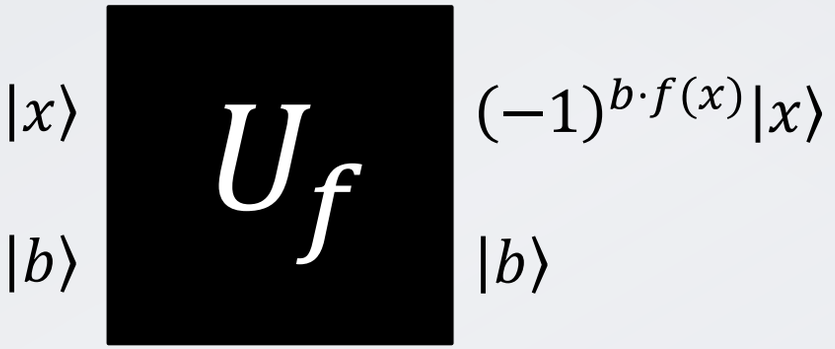


Phase kickback trick:

$$U_f|x\rangle|-\rangle = |x\rangle\frac{1}{\sqrt{2}}(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) =$$

$$= \begin{cases} |x\rangle|-\rangle, \text{if } f(x) = 0 \\ -|x\rangle|-\rangle, \text{if } f(x) = 1 \end{cases} = (-1)^{f(x)}|x\rangle|-\rangle$$

Another type of phase oracle



Allows to preserve more information about $f(x)$

(traditional phase oracle cannot distinguish $f(x)$ from $1 - f(x)$, since they only differ by a global phase)

Microsoft

Example: Implement an oracle for SAT problem

SAT problem: definition

$$f(x) = \bigwedge_i \bigvee_k y_{ik}, \text{where } y_{ik} = \text{either } x_j \text{ or } \neg x_j$$

Example:

$$f(x_1, x_2) = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

Is there an easier way to write this function?

$$f(x_1, x_2) = x_1 \oplus x_2$$

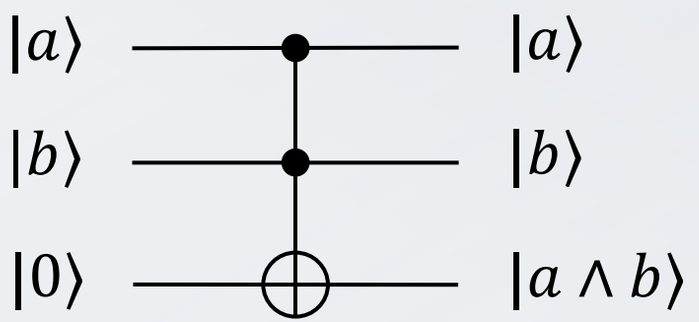
But let's stick to the SAT representation for now

Building blocks: NOT and AND

NOT


```
X(q);
```

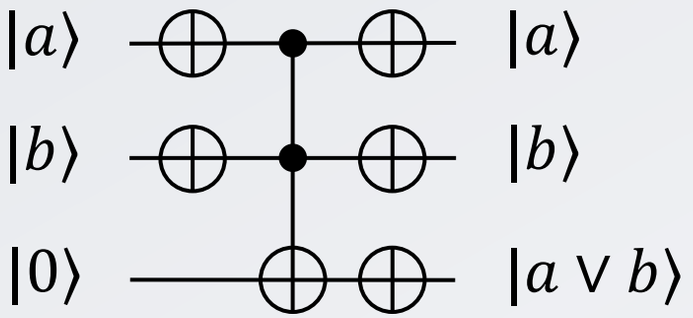
AND

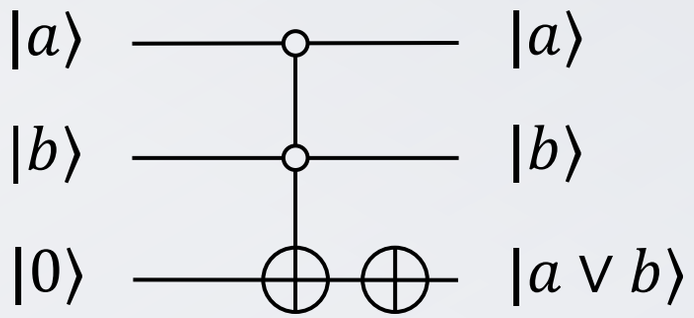



```
CCNOT(a, b, y);
Controlled X([a, b], y);
```

Building blocks: OR

De Morgan's laws: $a \vee b = \neg(\neg a \wedge \neg b)$



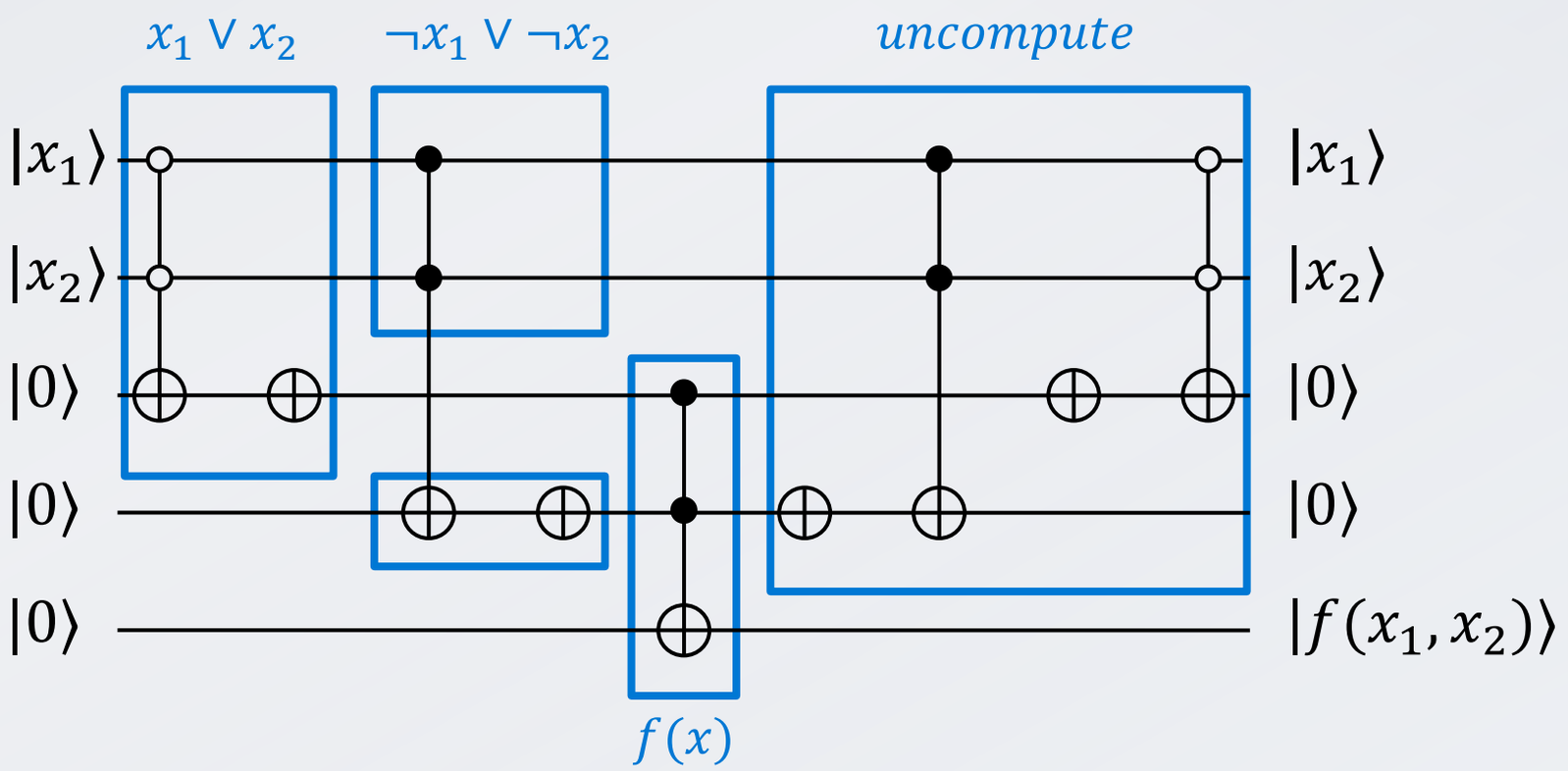



```
within {
    X(a); X(b);
} apply {
    CCNOT(a, b, y);
}
X(y);
```

```
ControlledOnInt(0, X)([a, b], y);
X(y);
```

Putting the blocks together: the circuit

$$f(x_1, x_2) = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$



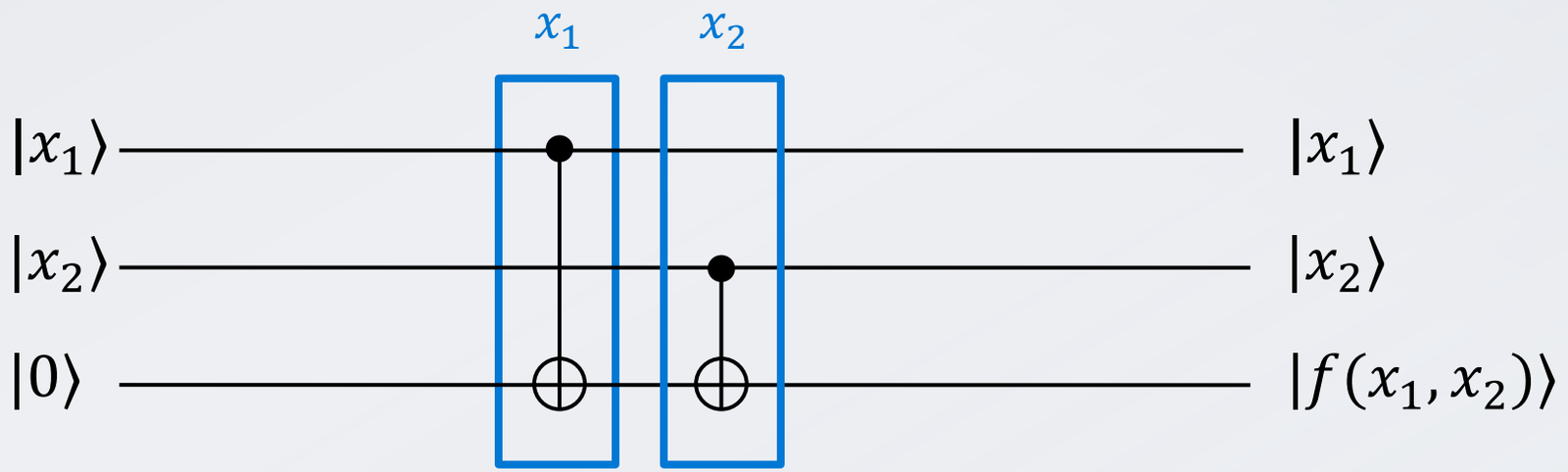
Putting the blocks together: Q#

$$f(x_1, x_2) = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

```
// Allocate auxiliary qubits to store results of clauses evaluation
use aux = Qubit[2];
within {
    // The first clause is OR of two inputs
    ControlledOnInt(0, X)(x, aux[0]);
    X(aux[0]);
    // The second clause is OR of two negated inputs
    Controlled X(x, aux[1]);
    X(aux[1]);
} apply {
    // The result is AND of two clauses
    Controlled X(aux, target);
}
```

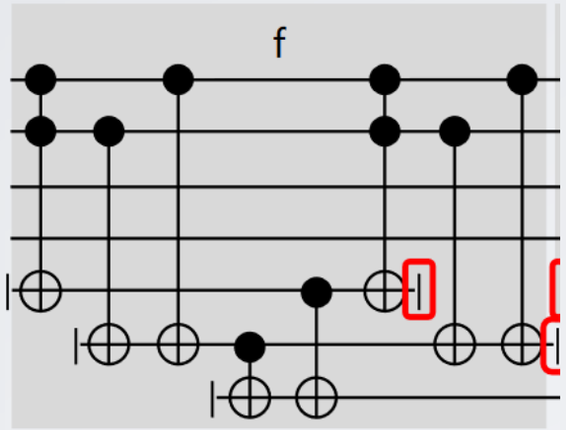
The same circuit based on the simpler representation

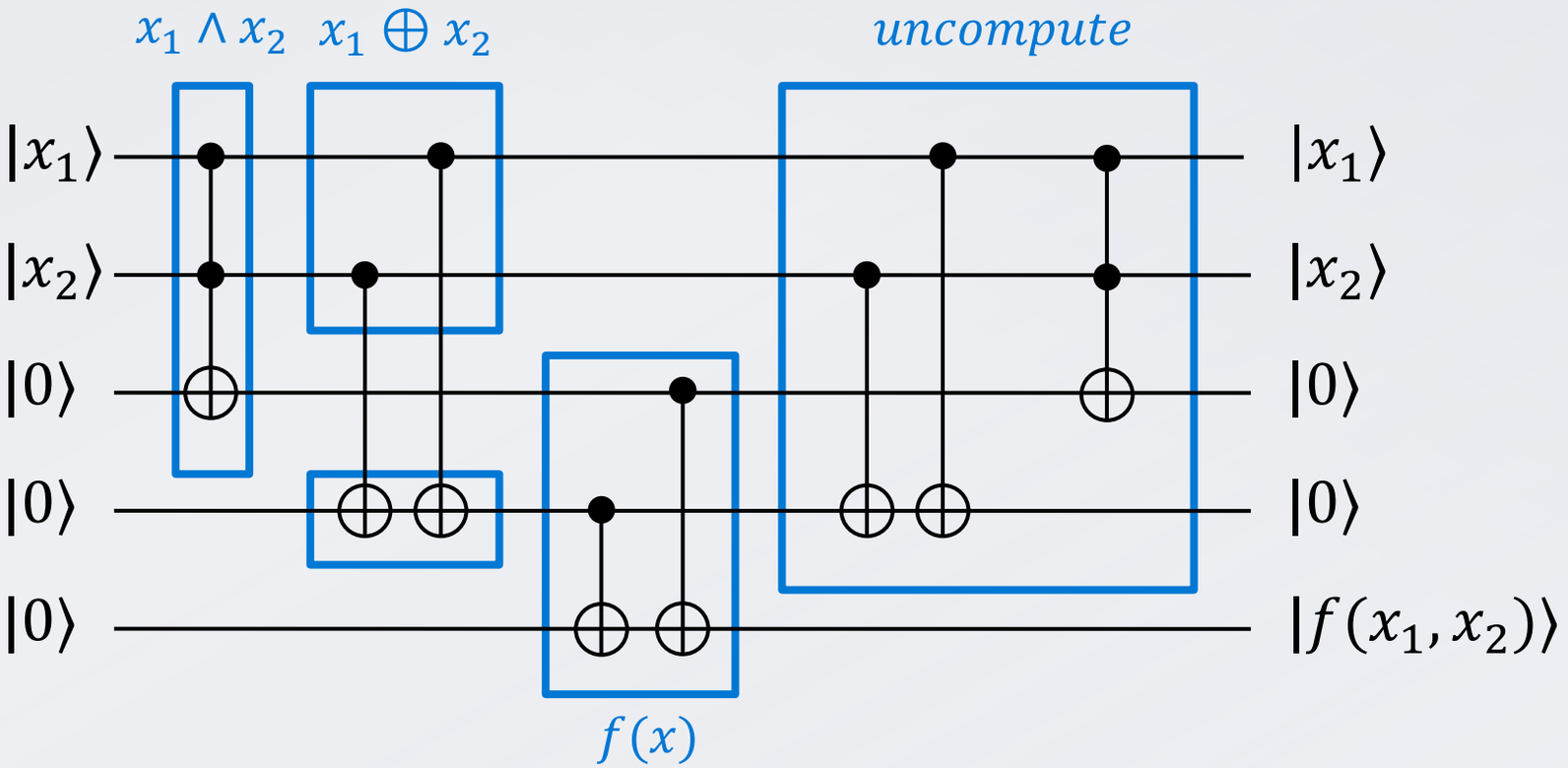
$$f(x_1, x_2) = x_1 \oplus x_2$$



Using a better representation of the computation done by the circuit can be more efficient than direct circuit implementation and optimization

Interpreting and optimizing a circuit, part 1: interpret





$$f(x_1, x_2) = x_1 \oplus x_2 \oplus x_1 \wedge x_2$$

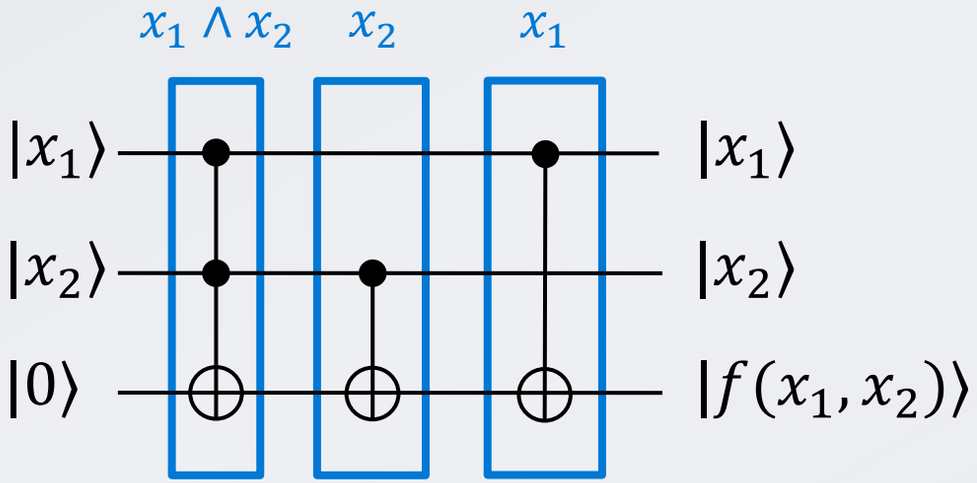
Interpreting and optimizing a circuit, part 2: optimize

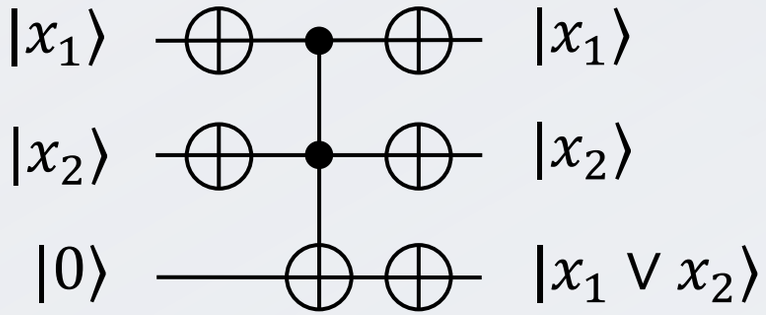
$$f(x_1, x_2) = x_1 \oplus x_2 \oplus x_1 \wedge x_2$$

x_0	x_1	$f(x)$
0	0	0
0	1	1
1	0	1
1	1	1

$$f(x_1, x_2) = x_1 \vee x_2$$

Compute XORs in-place
(no need for auxiliary qubits)





Circuit optimization tips and tricks

Circuits have two main parameters: number of qubits (“width”) and time it takes to execute the circuit (“depth”)

For reversible computations, depth depends mostly on the number of Toffoli (CCNOT) gates

Multi-controlled X gates are decomposed into CCNOTs automatically using extra qubits

How to optimize circuits?

- Swap gates with different targets, unless targets of one gate are controls for another
- Cancel pairs of adjacent gates of form “U” + “adjoint U”
- Try to reduce the number of helper qubits allocated (for example, by computing XORs in-place)
- Try to reduce the number of controls in controlled gates
- “Controlled U” followed by “ControlledOnZero U” can be replaced with “U”