# Outline

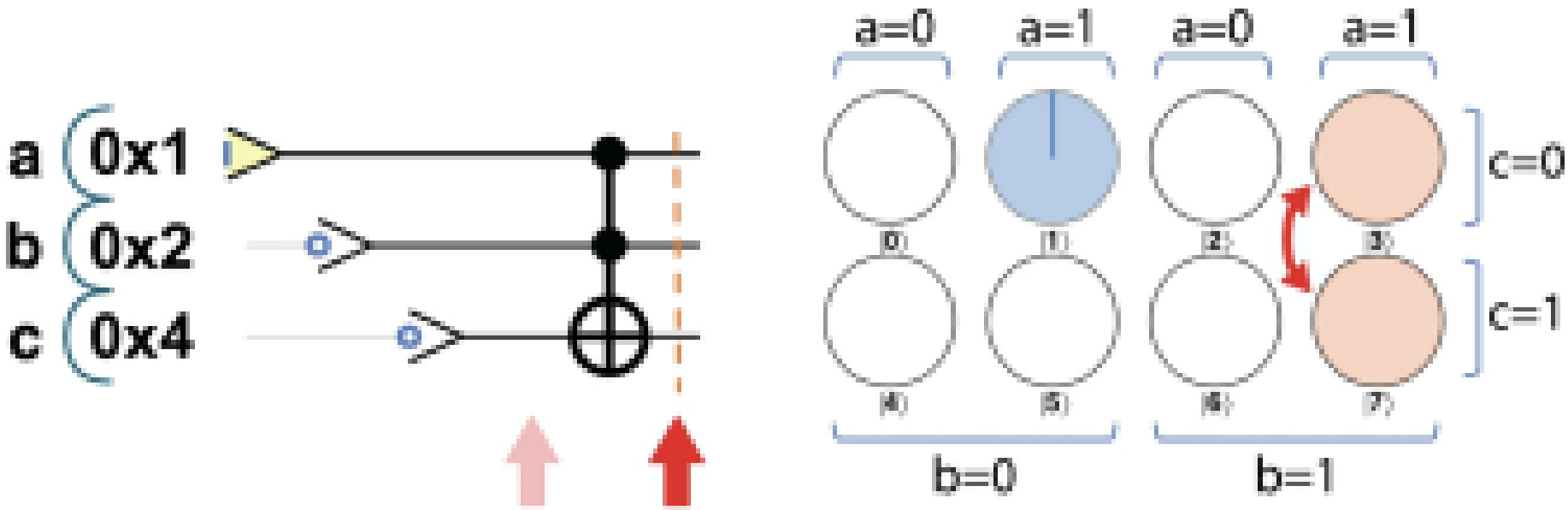



*Figure 5-1. When b=0, this operation has no effect*

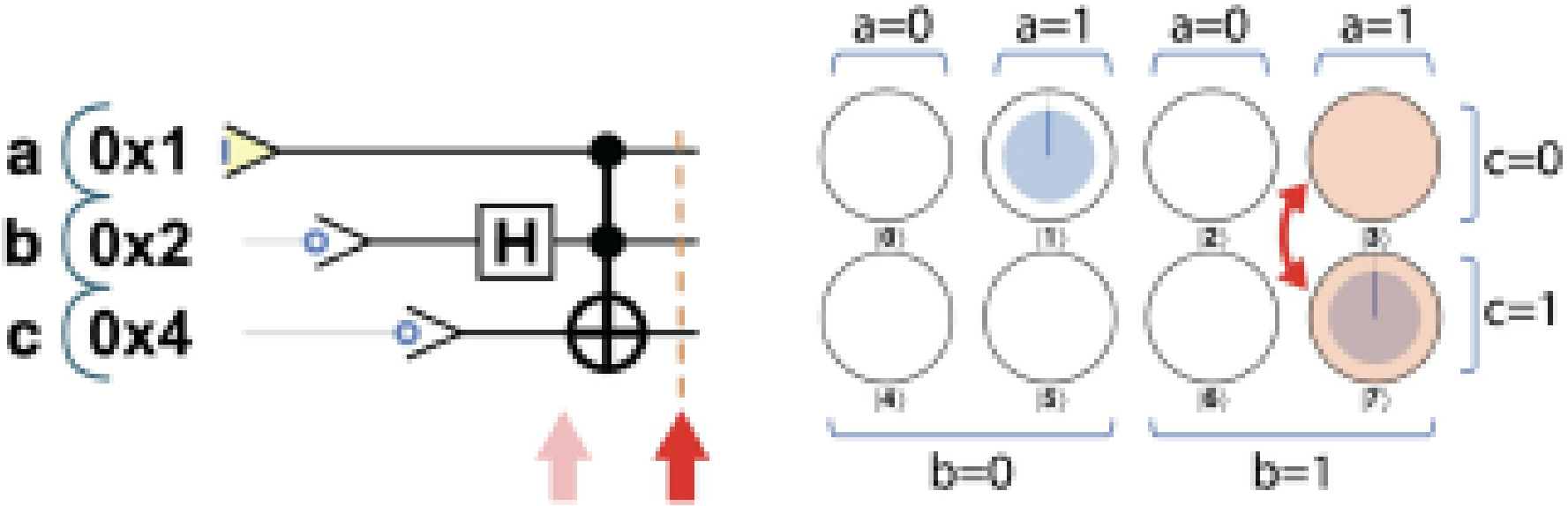



*Figure 5-2. A single gate performs two operations at the same time*

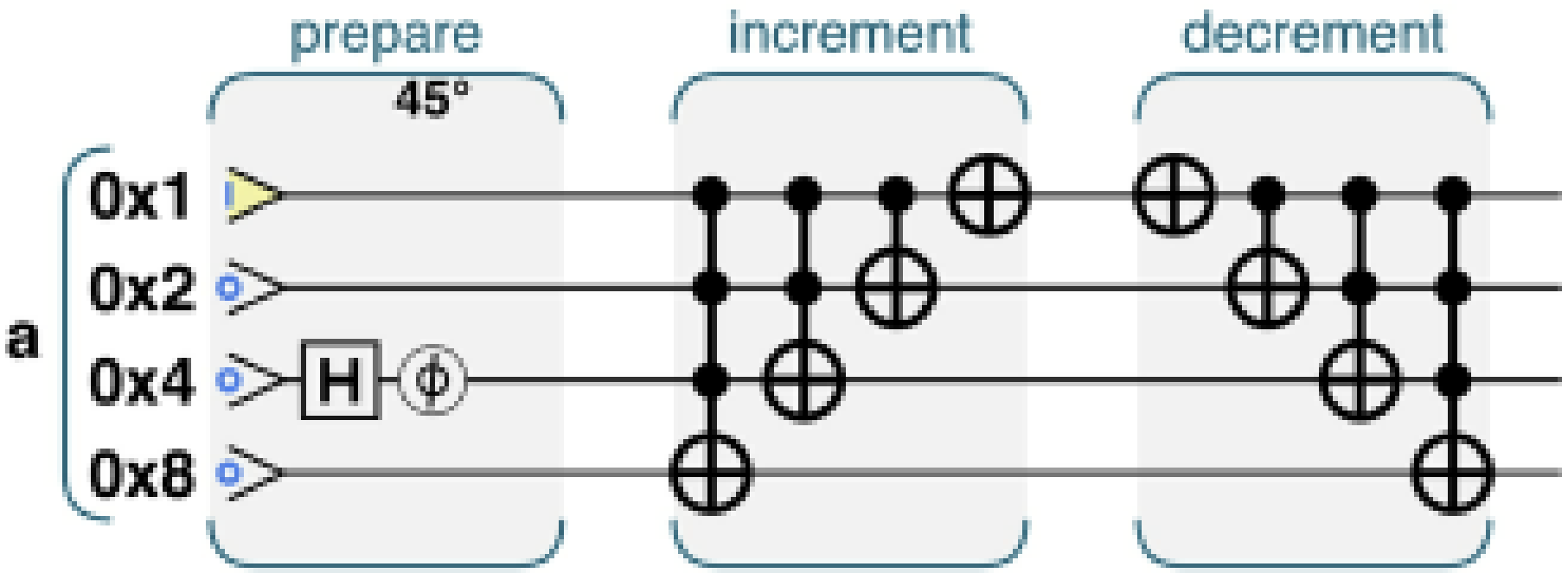



*Figure 5-3. Operations performing increment-by-1 and decrement-by-1*

*Reversibility*

First, and most obviously, the decrement operation is simply the increment with its constituent operations reversed. This makes sense, but may not be obvious if you're used to conventional logic. In conventional logic devices gates tend to have dedicated inputs and outputs, and simply running a device in reverse is likely to damage it, or at least fail to provide a useful result. As we've noted, for quantum operations reversibility is a critical requirement.

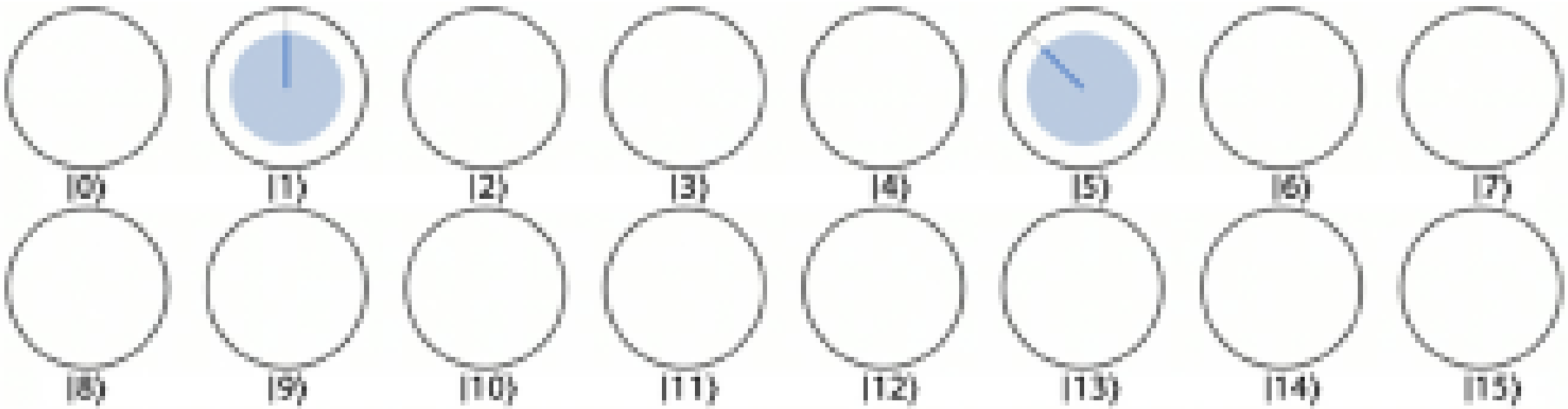



*Figure 5-4. The prepared superposition, before incrementing*

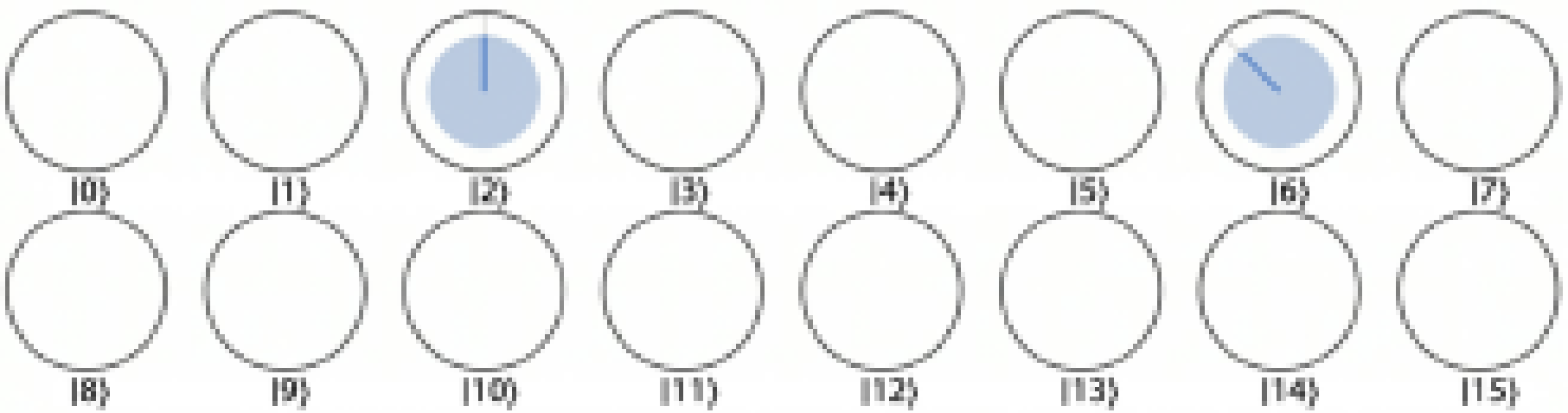



*Figure 5-5. The resulting superposition, after incrementing*

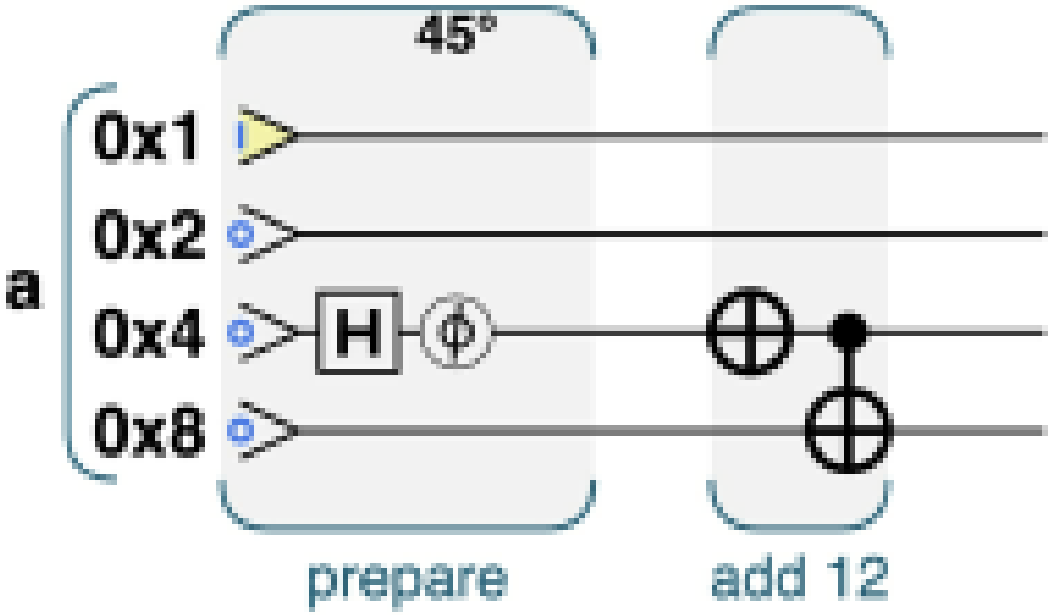
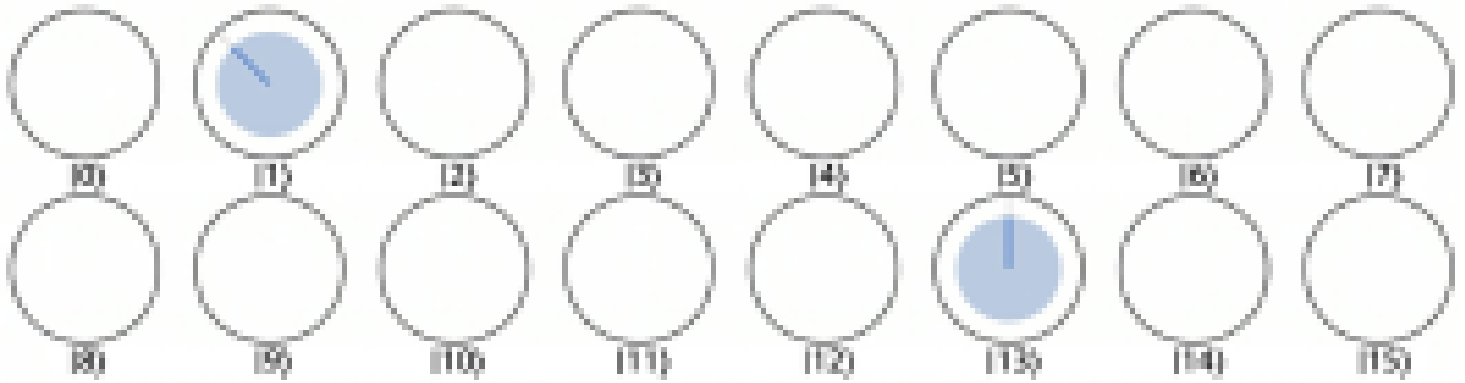



*Figure 5-6. A program to add 12 to a quantum integer*

In this case, Figure 5-7 shows that the input values $|1\rangle$ and $|5\rangle$ will become $|13\rangle$ and $|1\rangle$, since this program will wrap on overflow (just like conventional integer math).



## Adding Two Quantum Integers

Suppose we have two QPU registers a and b (bearing in mind that each of these could potentially store a superposition of integer values), and we ask for a simple + operation that would store the result of their addition in a new register c. This is analogous to how a CPU performs addition with conventional digital registers. However, there's a problem — this approach violates *both* the reversibility and the no-copying restrictions on QPU logic:

- Reversibility is violated by $c = a + b$ because the prior contents of c are lost.

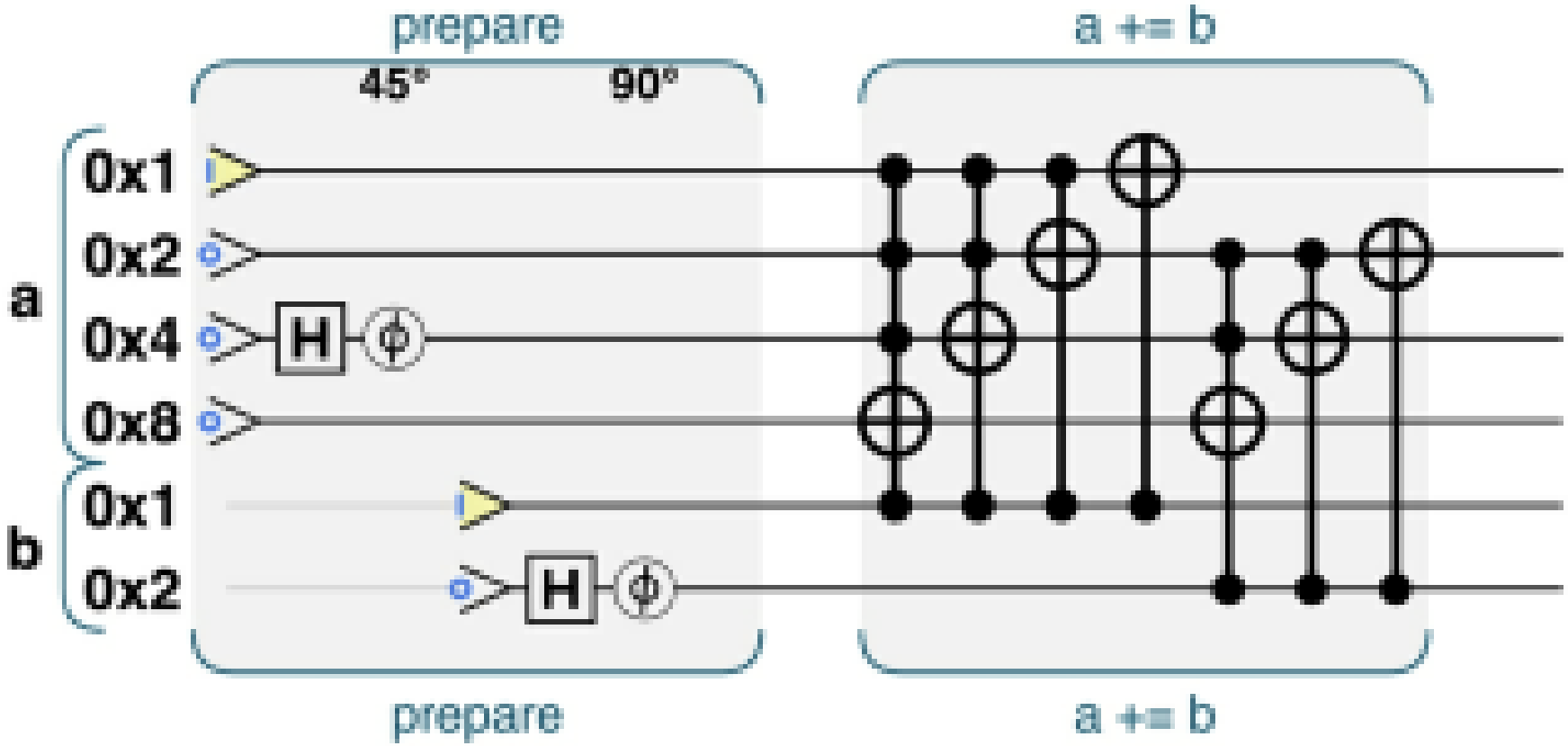
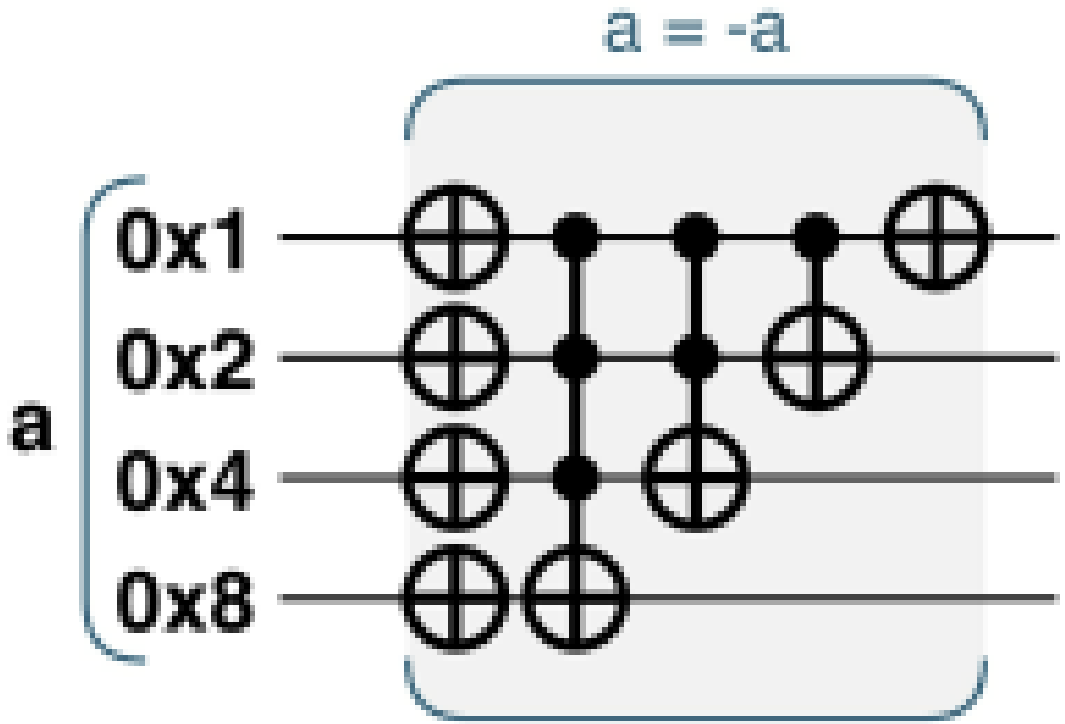



*Figure 5-8. Operations assembled to perform a += operation*

*Table 5-1. Two's complement for binary representation of negative integers*

| 0 | 1 | 2 | 3 | −4 | −3 | −2 | −1 |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |




Changement de signe : **Apply NOT on all bits and add** 1. $\rightsquigarrow$ vérifier sur la slide d'avant que ça marche.

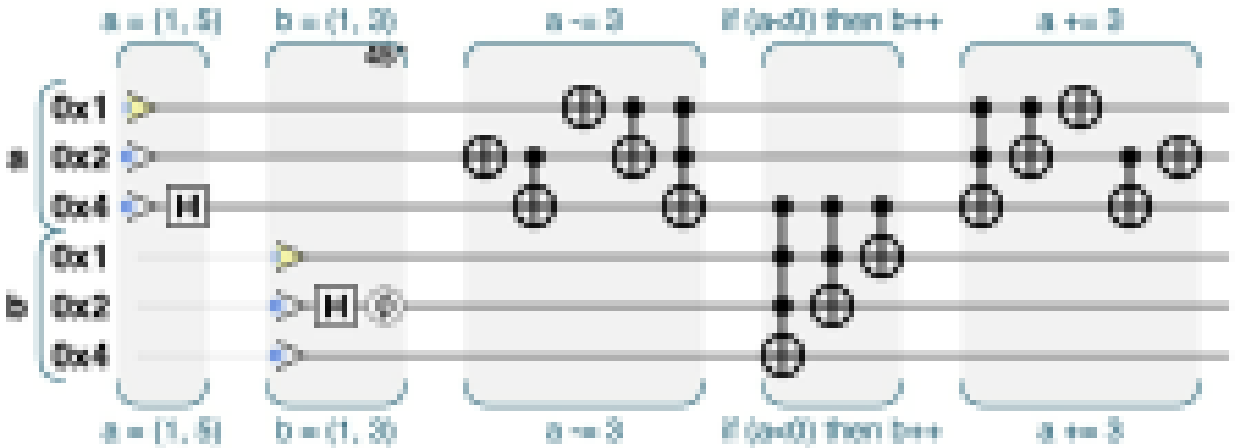



*Figure 5-11. Conditional execution*

**SAMPLE CODE**

Run this sample online at http://oreilly-qc.github.io?p=5-4

*Example 5-4. Conditional execution*

```
a.subtract(3);
// If high bit of a is set then b += 1
b.add(1, a.bits[0x4]);
a.add(3);
```

Running Example 5-4, circle notation reveals that only parts of the quantum state where a is less than 3 or greater than 6 are incremented.

Only circles in columns 0, 1, 2, and 7 in Figure 5-12 are affected by the incrementing logic.

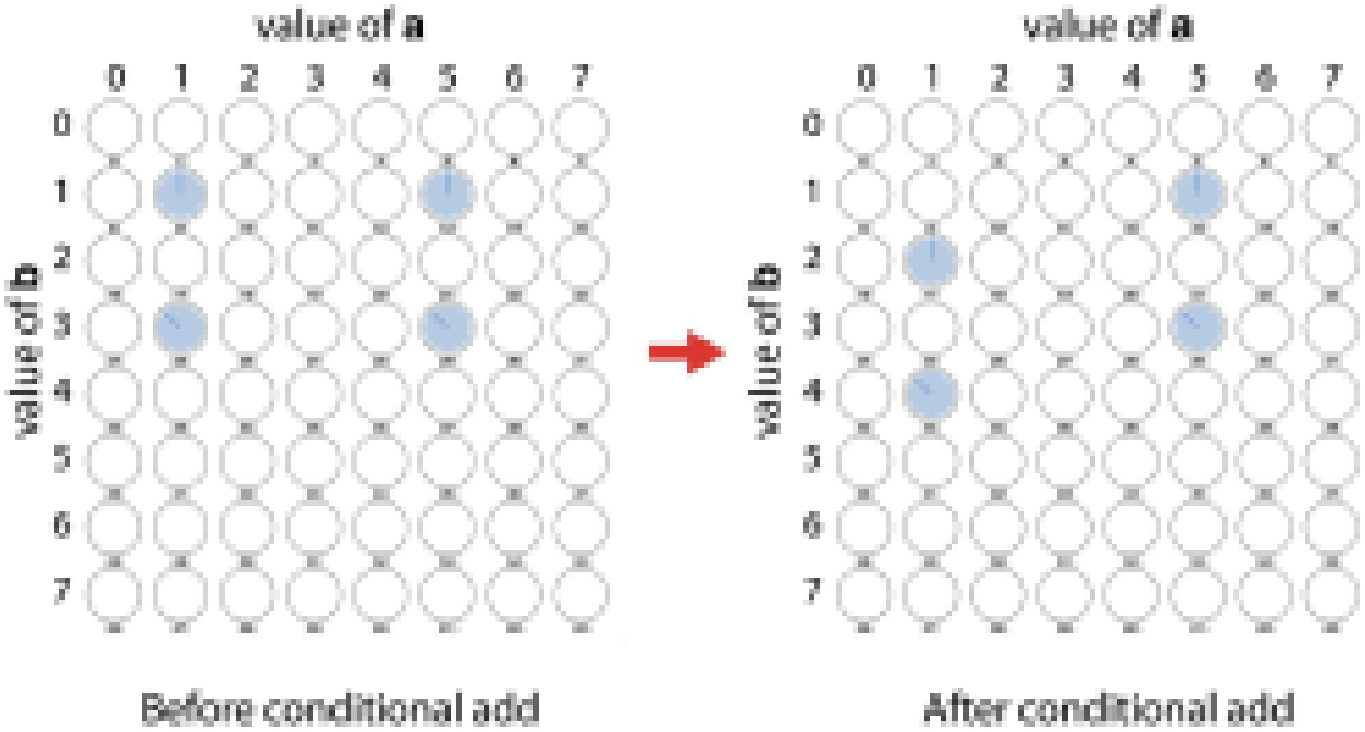



Figure 5-12. Conditional addition

On exécute : si $a < 3$, $b++$.

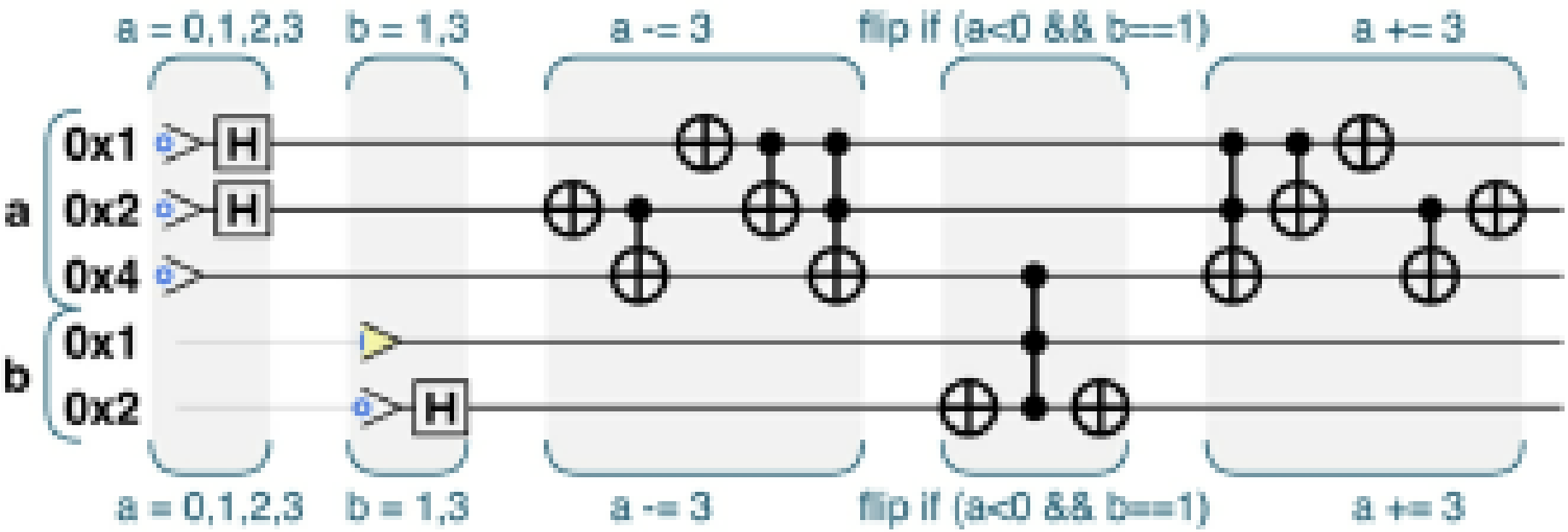



*Figure 5-13. Phase-encoding the result*

**SAMPLE CODE**

Run this sample online at http://oreilly-qc.github.io?p=5-5.

*Example 5-5. Phase-encoding the result*

```
// Flip the phase if a is less than 3 and b is equal to 1
a.subtract(3);
b.not(~1);
qc.phase(180, a.bits(0x4), b.bits());
b.not(~1);
a.add(3);
```

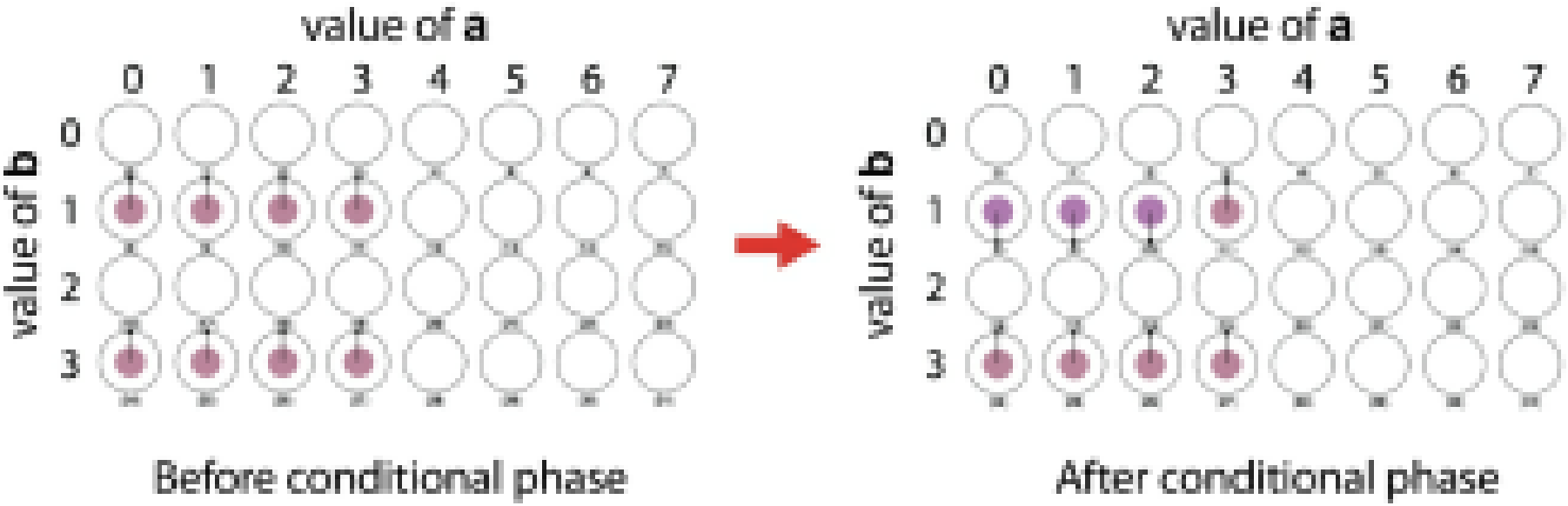



*Figure 5-14. The effect of phase encoding*

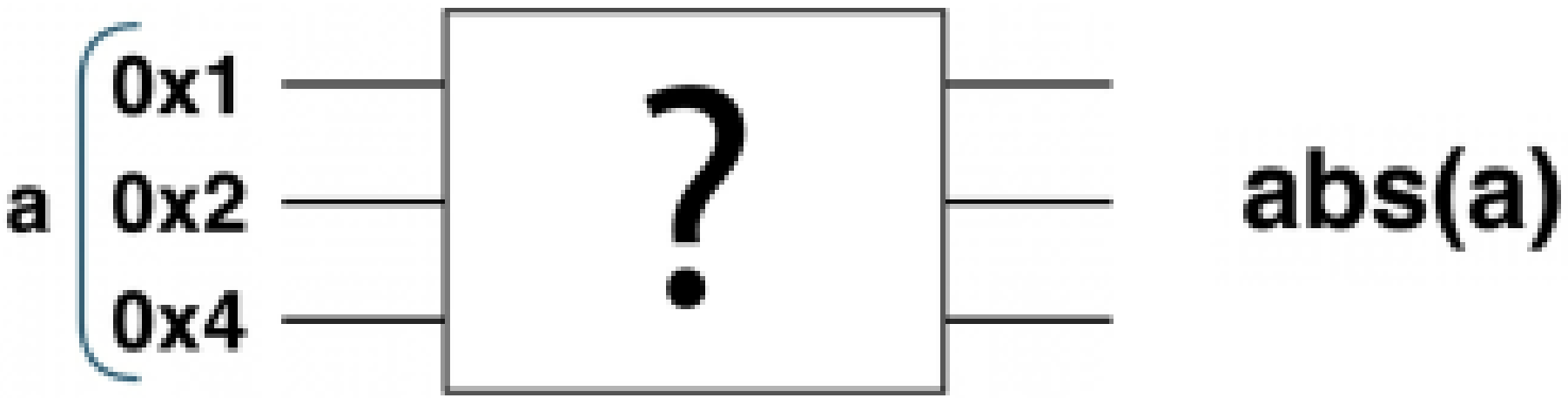



*Figure 5-15. What QPU operations can compute the absolute value?*

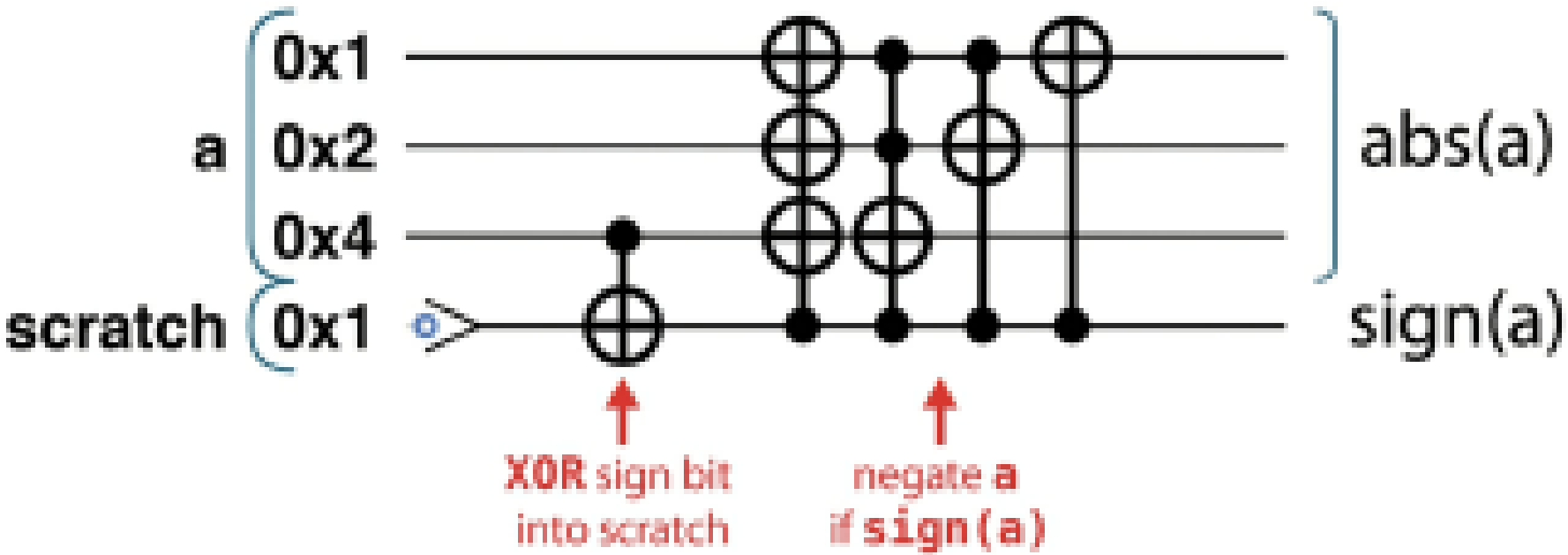



*Figure 5-16. Scratch qubits can make an irreversible operation reversible*

With the scratch qubit involved, we now *can* find a set of gates that will ultimately implement `abs` on our `a` register. But before we verify in detail that the circuit in Figure 5-16 achieves this, note that the `CNOT` operation we use between our scratch qubit and the `0x4` qubit of `a` (which tells us the sign of its integer value in two's complement) does not *copy* the sign qubit exactly. Instead, the `CNOT` *entangles* the scratch qubit with the sign qubit. This is an important distinction to note, since we also know that QPU operations cannot copy qubits.

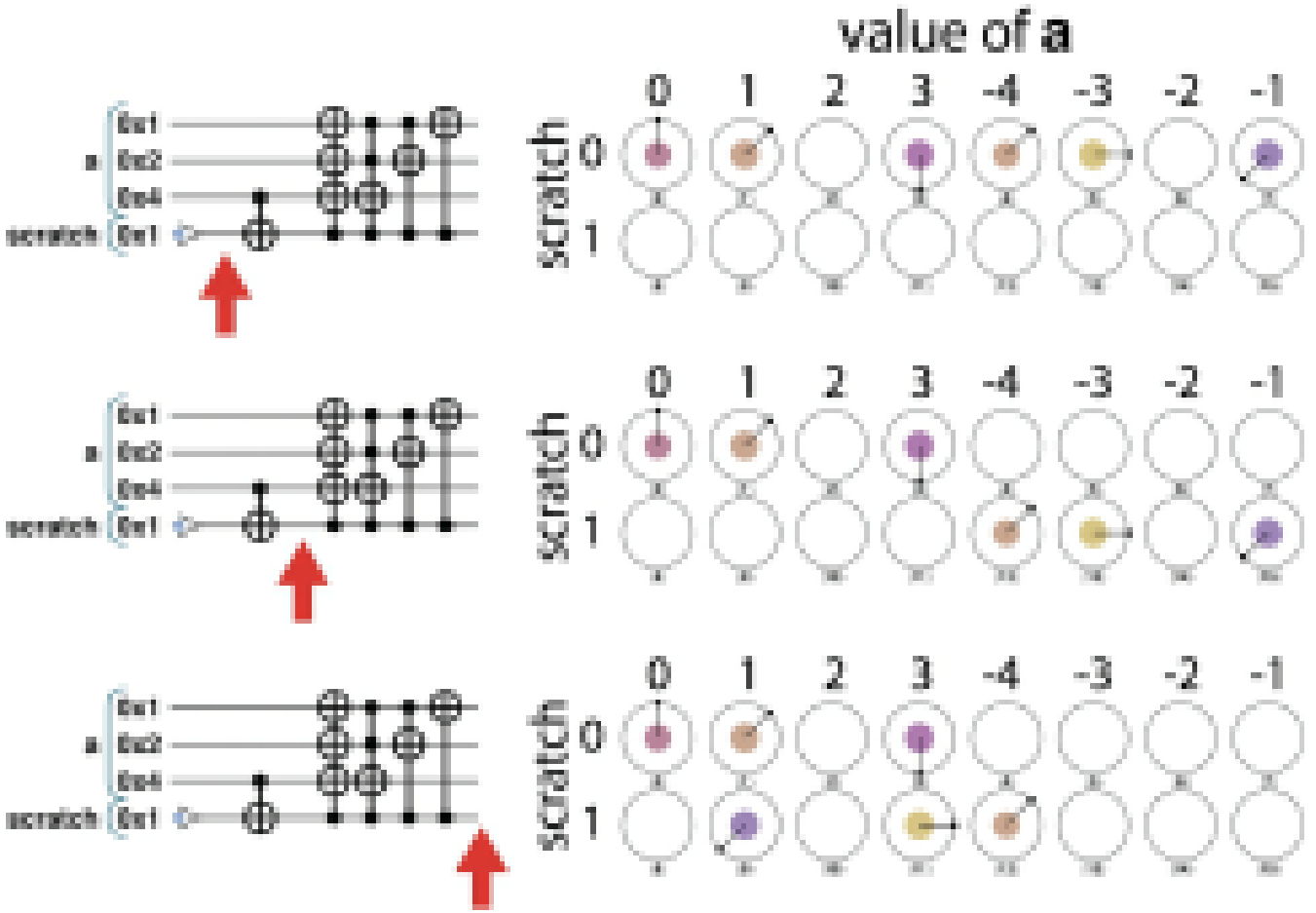



*Figure 5-17. Circle notation steps for the absolute value*

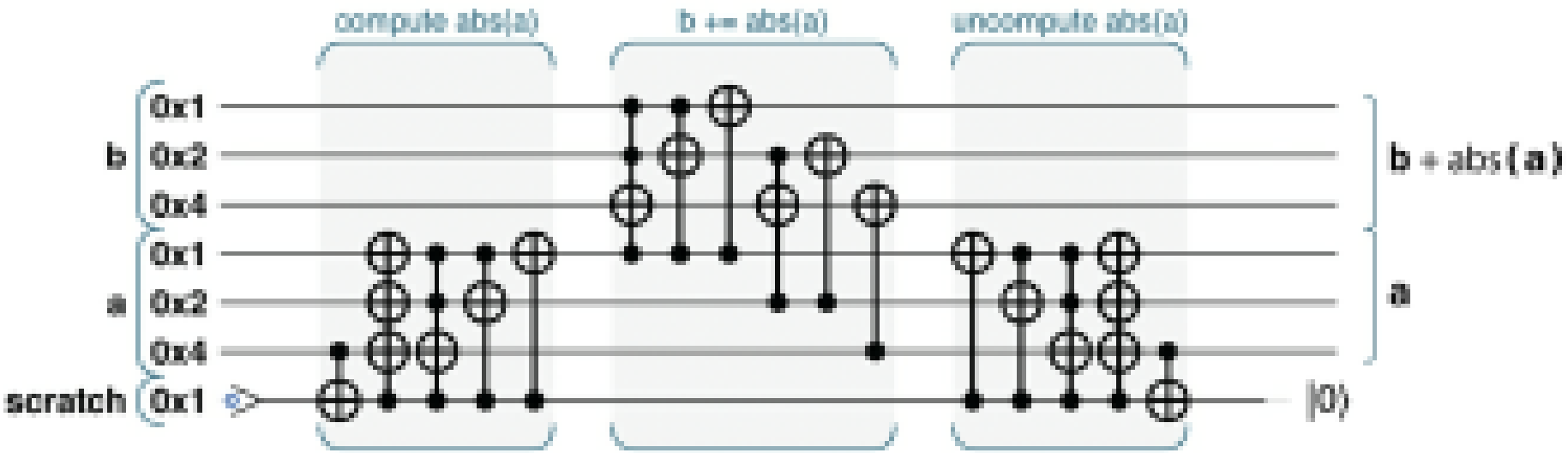



*Figure 5-18. Using uncomputation to perform b += abs(a)*

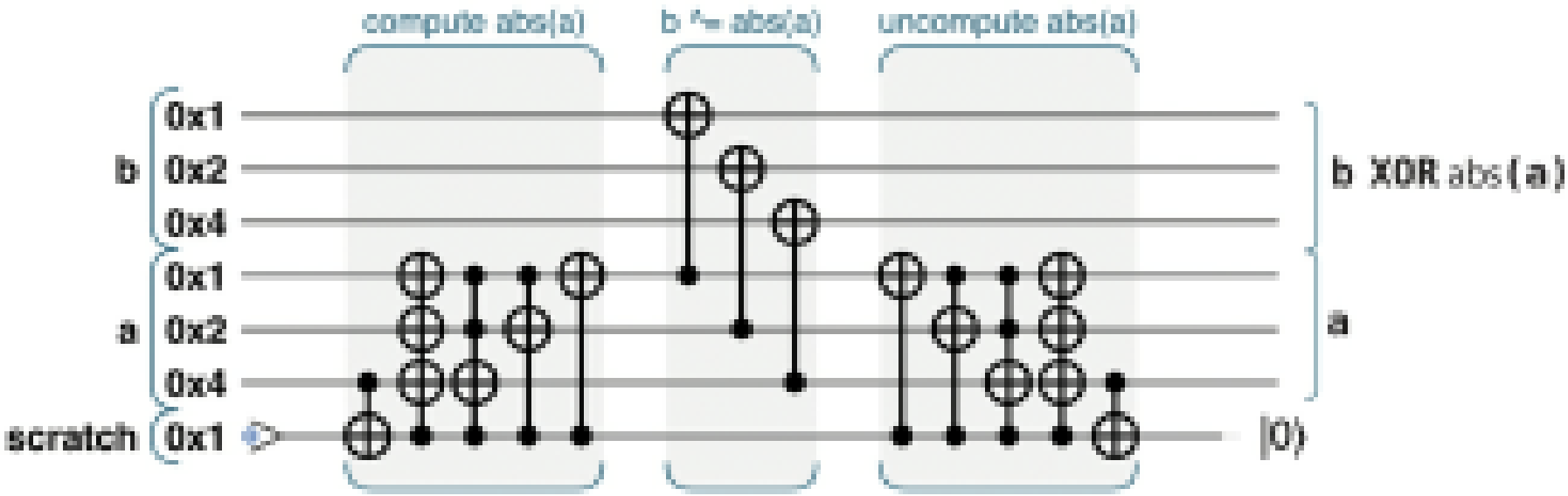



*Figure 5-19. Using uncomputation to produce b xor abs(a)*

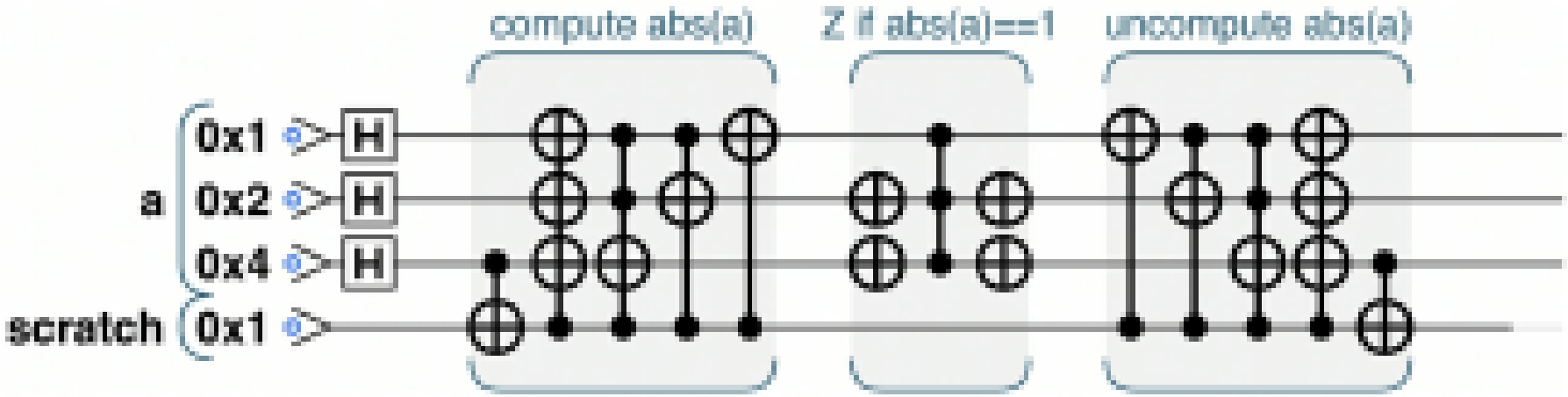



*Figure 5-20. Using uncomputation to perform phase(180) if abs(a)==1*

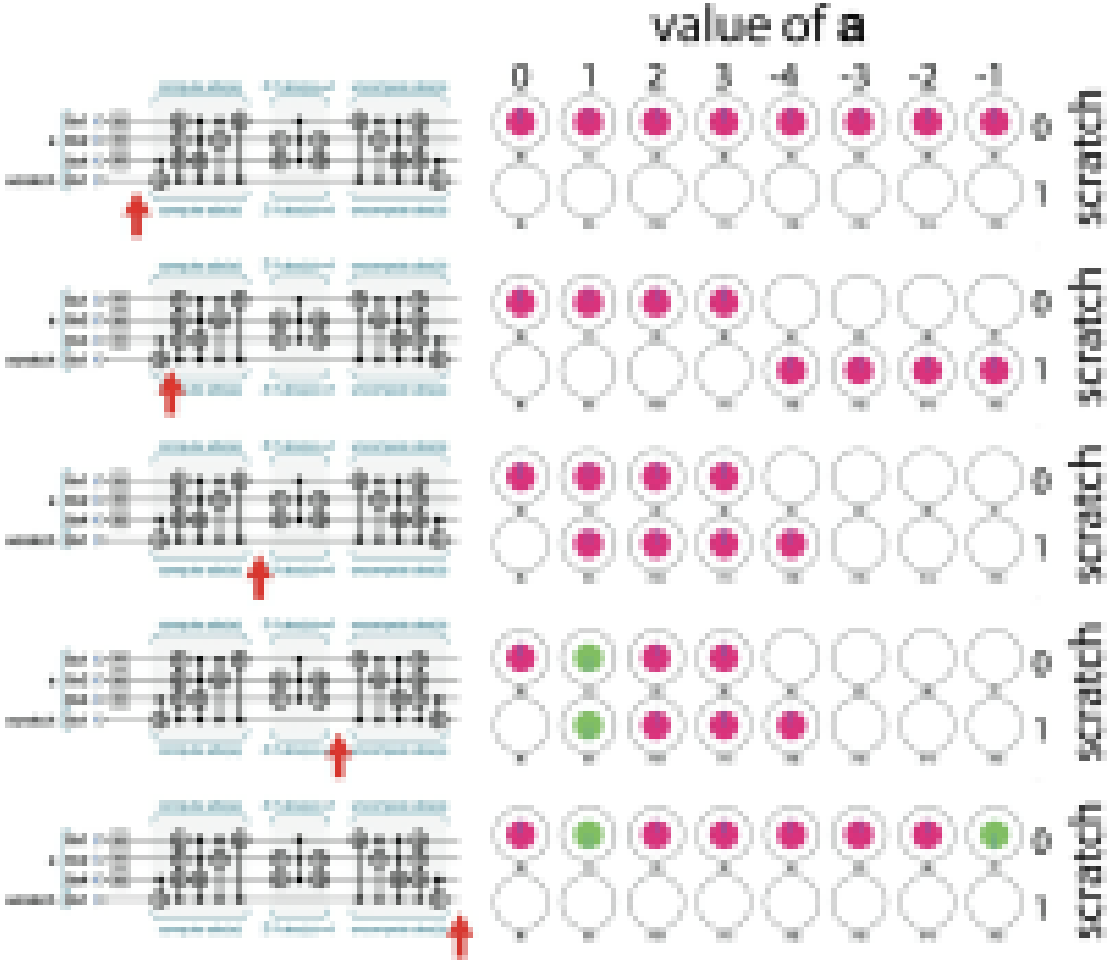



*Figure 5-31. Step-by-step walkthrough of using uncompute for conditional phase inversion*

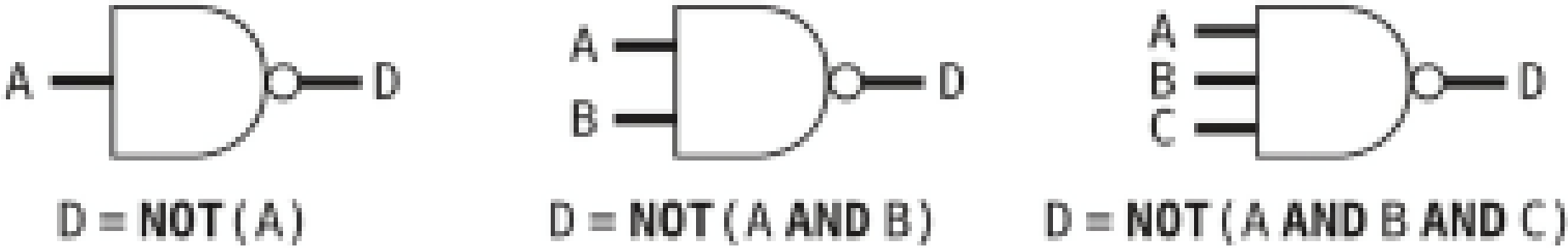



*Figure 5-22. Digital NAND gates in different sizes*

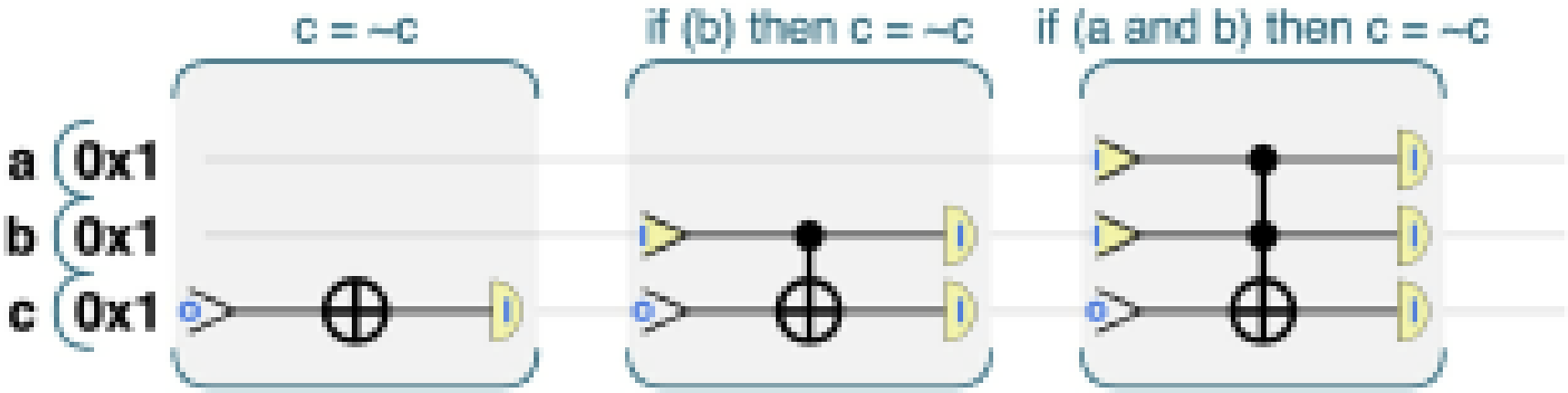



*Figure 5-23. Quantum CNOT gates in different sizes*

**SAMPLE CODE**

Run this sample online at http://oreilly-qc.github.io?p=5-6.

*Example 5-6. Logic using CNOT gates*

```
// a = ¬a
qc.write(0);
qc.not();
qc.read();

// if (b) then a = ¬a
qc.write(2, 2|4);
qc.cnot(b);
qc.read(2|4);

// if (a and b) then a = ¬a
qc.write(1|2);
qc.ccnot(4, 1|2);
qc.read(1|2|4);
```

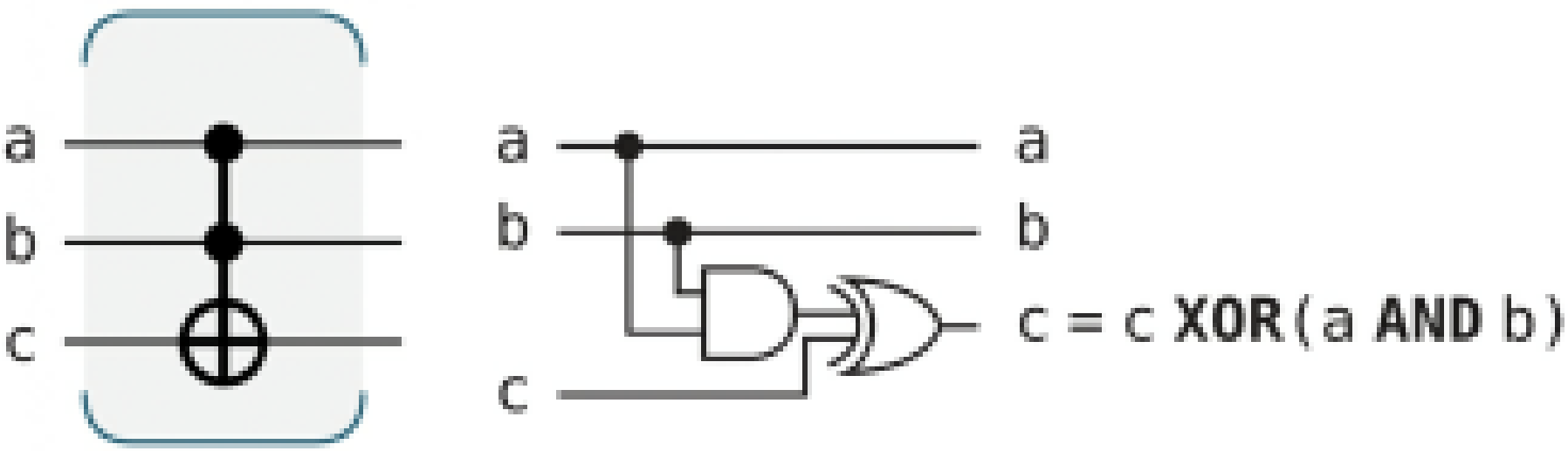
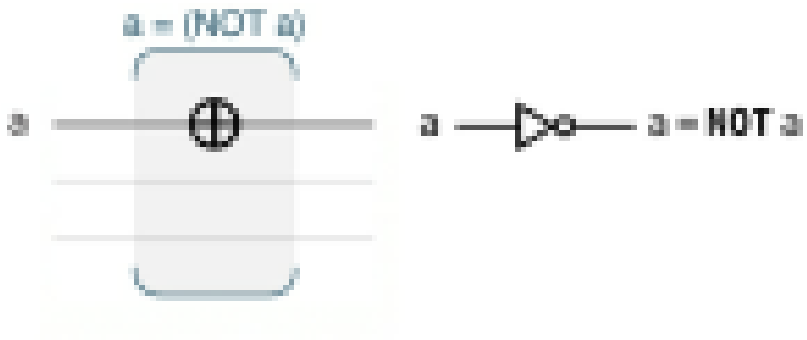
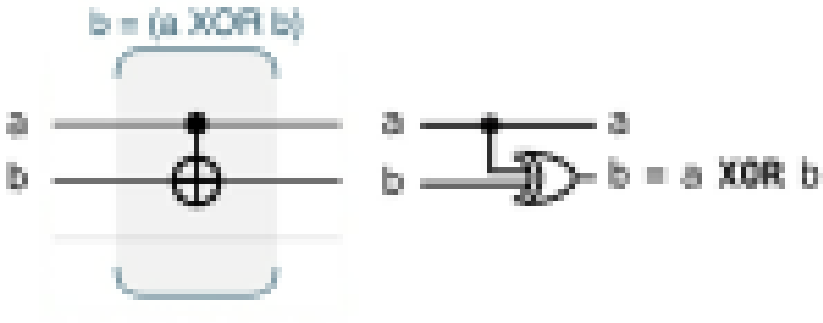
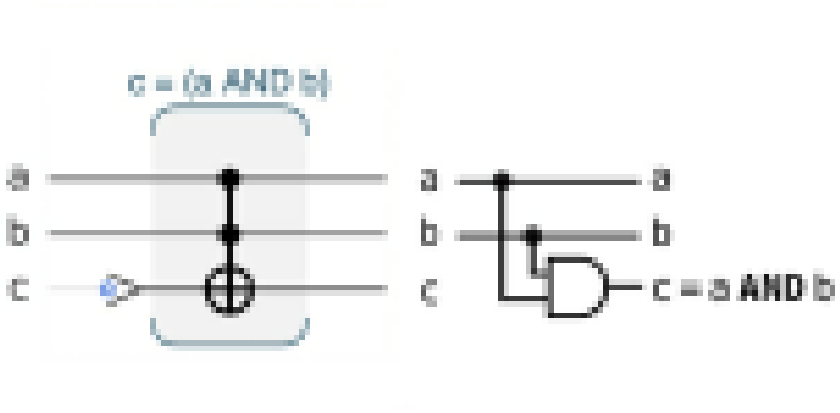
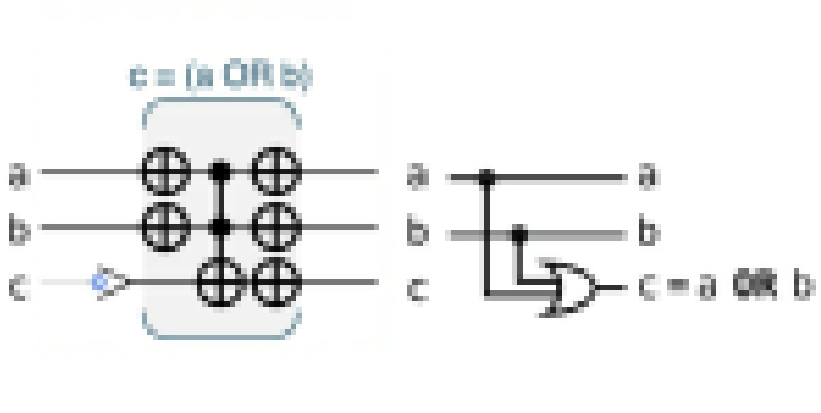



*Figure 5-24. The exact digital logic equivalent of our multicondition CNOT gate*

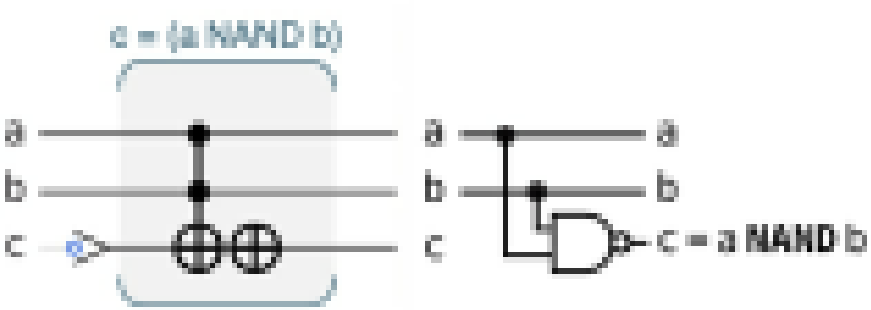

a = (NOT a)
a
a
a = NOT a

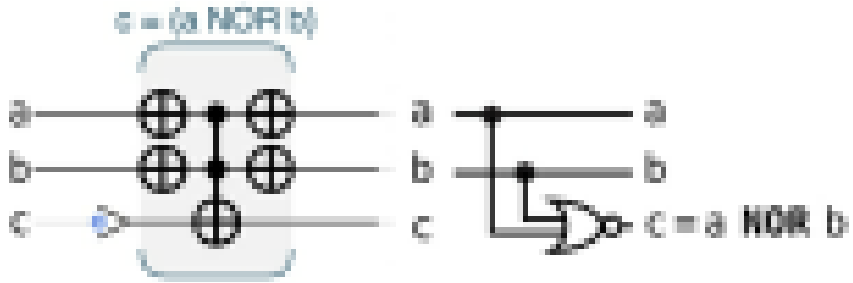

b = (a XOR b)
a
b
a
b
a
b = a XOR b




Remarquer l'utilisation astucieuse du NOT sur le AND pour transformer en OR.


c = (a NAND b)
a
b
c
a
b
c
c = a NAND b


c = (a NOR b)
a
b
c
a
b
c
c = a NOR b

→ voir exercice addition de qubits.