Quantum Addition using Ripple-Carry Method

Michal Forgó

February 12, 2025

Contents

1 Introduction

Quantum computing introduces novel ways to perform arithmetic operations, including binary addition. This document explains the implementation of quantum addition using the **ripple-carry method**.

2 Ripple-Carry Addition in Quantum Computing

The ripple-carry method works by sequentially computing the sum of two binary numbers and propagating the carry bits. The process involves:

1. Using **CCNOT** gates (CCX) to calculate the carry bit.
2. Using **CNOT** gates (CX) to compute the sum at each position.
3. Reversing unnecessary operations to ensure the reversibility of the quantum computation.

3 Quantum Circuit Representation

Quantum computing enables efficient arithmetic operations using reversible logic. One of the fundamental operations in quantum arithmetic is addition, which can be implemented using quantum gates. The diagram below represents a quantum 4-bit ripple-carry adder, which performs bitwise addition using controlled operations.

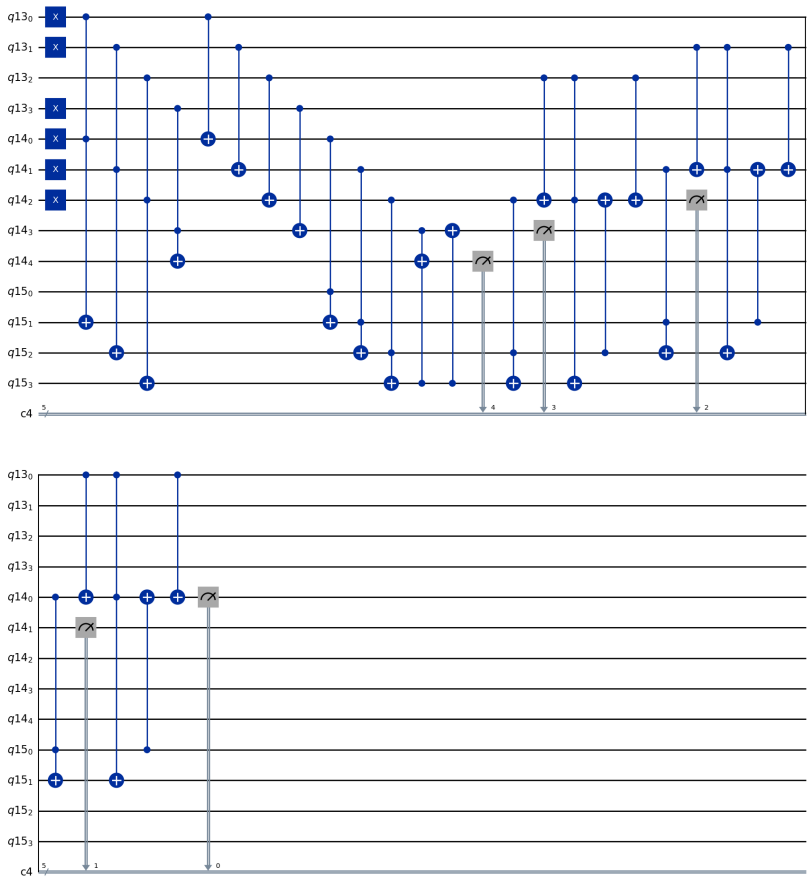

Figure 4: Quantum circuit representation of the ripple-carry addition.

4 Quantum Gates Used

4.1 NOT

- **NOT (X) gate**: Used to set input values in quantum registers. Flips a qubit from $|0\rangle$ to $|1\rangle$ and otherwise.

Figure 1: NOT gate.

4.2 CNOT

- **CNOT (CX) gate**: A controlled NOT gate that flips the target qubit if the control qubit is $|1\rangle$. This is used for the sum computation.

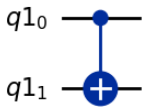

Figure 2: CNOT gate.

4.3 CCNOT

- **CCNOT (CCX) gate**: A controlled-controlled NOT gate, which acts as a carry generator by flipping the target qubit if both control qubits are $|1\rangle$.

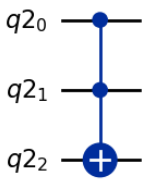

Figure 3: CCNOT gate.

5 Step-by-Step Code Explanation

5.1 Setup

Firstly, we need to import the necessary libraries for the program to work.

```
from qiskit import QuantumCircuit, transpile
from qiskit import QuantumRegister, ClassicalRegister
from qiskit_aer import AerSimulator
```

After that, the program needs an input method. We can hard-code the values like this:

```
firstBinaryNumber = "11001"
secondBinaryNumber = "0101"
```

Or we could let the user choose them like this:

```
while True:
    firstBinaryNumber = input("Enter a binary number: ")
    secondBinaryNumber = input("Enter another binary number: ")

    # Validate input length and that input contains only 1s or 0s
    if len(firstBinaryNumber) > 8 or len(secondBinaryNumber) > 8 or \
    not set(firstBinaryNumber).issubset({'0','1'}) or \
    not set(secondBinaryNumber).issubset({'0','1'}):
        # Error message
        print("Please enter valid inputs. Try again.")
    else:
        break  # Exit loop if input is valid
```

It doesn't matter which method we use; we just need to ensure that the input consists only of ones and zeros and that it is not too large for our registers. We set the maximum allowed bit length to 7.

To perform quantum addition, we must allocate space to store values. First, we determine the largest input and take its size. Based on that, we define the necessary quantum and classical registers:

```
maxInputLength = max(len(firstBinaryNumber), len(secondBinaryNumber))

regA = QuantumRegister(maxInputLength)
regB = QuantumRegister(maxInputLength + 1)
regC = QuantumRegister(maxInputLength)
regD = ClassicalRegister(maxInputLength + 1)
```

- *regA* (Quantum Register, *maxInputLength* qubits)
  - This register stores the first binary number.
  - Each qubit represents a bit of the input, initialized in the $|0\rangle$ or $|1\rangle$ state based on the binary number.
- *regB* (Quantum Register, $maxInputLength + 1$ qubits)
  - This register stores the second binary number and also serves as the output register.
  - The extra qubit (+1) is necessary to accommodate possible carry-over in the final sum.
- *regC* (Quantum Register, *maxInputLength* qubits)
  - This register handles carry bits during the addition process.
  - It temporarily holds information that ensures correct addition by simulating classical carry propagation.
- *regD* (Classical Register, $maxInputLength + 1$ bits)
  - This is a classical register used to store the final measured result.
  - After running the quantum computation, the output is measured and saved into *regD*.

Now we can initializes quantum circuit using the defined registers.

```
qc = QuantumCircuit(regA, regB, regC, regD)
```

Right now, the registers are empty. So we need to put our data in them. Because all of the individual qbits are set to $|0\rangle$ as default we just need to use the NOT gate on the qbits that we want to flip into 1s.

```
for idx, val in enumerate(firstBinaryNumber):
    if val == "1":
        qc.x(regA[len(firstBinaryNumber) - (idx+1)])

for idx, val in enumerate(secondBinaryNumber):
    if val == "1":
        qc.x(regB[len(secondBinaryNumber) - (idx+1)])
```

5.2 Carry Gate Logic for Quantum Addition

In this section, we implement the carry gate logic required for performing binary addition in a quantum circuit. The process consists of computing carry values, updating them, and ensuring proper reset to maintain correct operation.

5.2.1 Carry Computation and Propagation

The carry bits are calculated and propagated through the circuit using CCNOT (CCX) and CNOT (CX) gates. The process ensures that the addition follows classical ripple-carry logic while leveraging quantum entanglement.

```
# Implementing carry gate logic for addition
for i in range(maxInputLength - 1):
    qc.ccx(regA[i], regB[i], regC[i + 1])  # Compute carry
    qc.cx(regA[i], regB[i])  # Partial sum
    qc.ccx(regC[i], regB[i], regC[i + 1])  # Update carry
```

Each iteration of the loop processes a bit pair from `regA` and `regB`, computing the carry using a CCNOT gate and updating the partial sum using a CNOT gate.

5.2.2 Final Carry Computation

To handle the most significant bit (MSB), a final carry computation is performed, ensuring that the carry propagates correctly.

```
qc.ccx(regA[maxInputLength - 1], regB[maxInputLength - 1],
        regB[maxInputLength])
qc.cx(regA[maxInputLength - 1], regB[maxInputLength - 1])
qc.ccx(regC[maxInputLength - 1], regB[maxInputLength - 1],
        regB[maxInputLength])
```

5.3 Resetting Carry Bits

To ensure correct operation in subsequent calculations, carry operations are reversed, resetting all carry bits to the $|0\rangle$ state.

```
# Undo last carry operation to reset the state
qc.cx(regC[maxInputLength - 1], regB[maxInputLength - 1])

# Reverse the carry operations to reset all carry bits to |0>
for i in range(maxInputLength - 1):
    qc.ccx(regC[(maxInputLength - 2) - i],
        regB[(maxInputLength - 2) - i],
        regC[(maxInputLength - 1) - i])
    qc.cx(regA[(maxInputLength - 2) - i],
        regB[(maxInputLength - 2) - i])
    qc.ccx(regA[(maxInputLength - 2) - i],
        regB[(maxInputLength - 2) - i],
        regC[(maxInputLength - 1) - i])

    # These operations flips regB if control is |1>
    qc.cx(regC[(maxInputLength - 2) - i],
    regB[(maxInputLength - 2) - i])
    qc.cx(regA[(maxInputLength - 2) - i],
    regB[(maxInputLength - 2) - i])
```

By applying the carry reversal operations in reverse order, the circuit ensures that all intermediate carry values return to the initial state.

5.4 Measuring the Final Result

After the computation, the final sum is extracted by measuring the qubits in `regB` and storing the classical result in `regD`.

```
# Measure qubits and store results in the classical register
for i in range(maxInputLength + 1):
    qc.measure(regB[i], regD[i])
```

This step collapses the quantum state into a classical binary output that represents the sum of the two input numbers.

5.5 Executing and Visualizing the Results

Now that the circuit is fully constructed, we need to simulate it to obtain the results. We initialize a quantum simulator, compile our circuit for execution, and let it compute the result.

```
simulator = Aer.get_backend('aer_simulator')
circ = transpile(qc, simulator)
result = simulator.run(circ).result()
counts = result.get_counts(circ)
print(*counts)
```

6 Conclusion

Quantum addition using the ripple-carry method is an essential step toward more complex arithmetic operations in quantum computing. The use of CCNOT and CNOT gates ensures logical correctness while maintaining quantum reversibility.