EE-677 (VLSI-CAD)

Report

Dishant - 22b3984

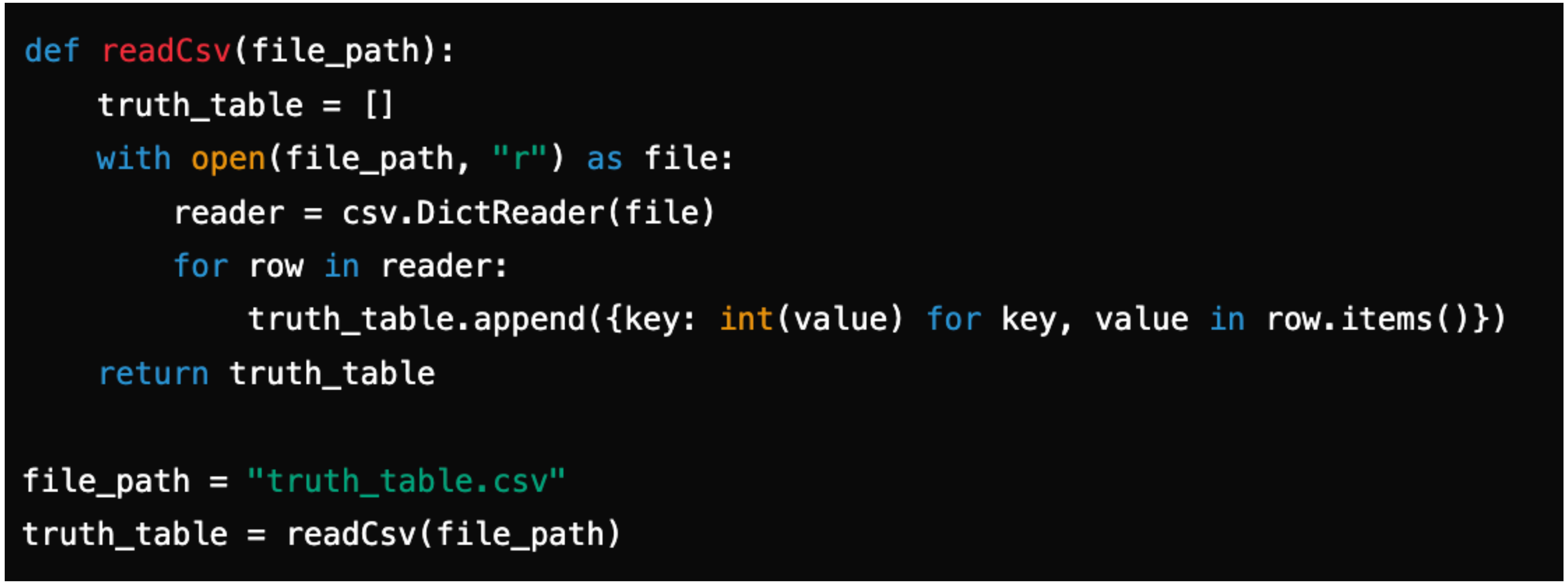
Problem Statement - Design of a quantum circuit from the given Truth table

Introduction

This project focuses on designing a quantum circuit based on a given truth table. By leveraging the principles of Boolean logic and quantum computing, the script automates the process of:

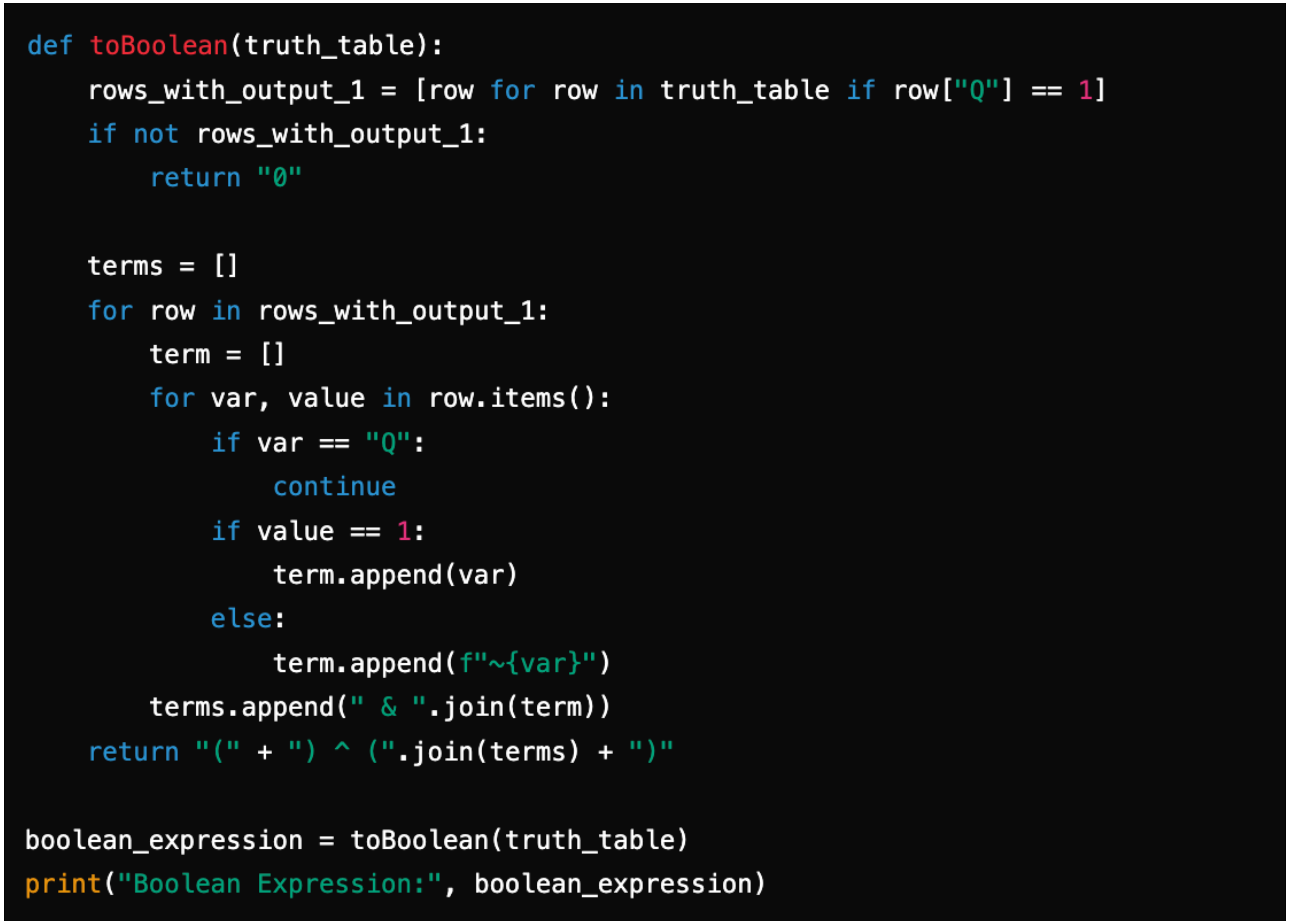
1. Parsing a truth table to derive its corresponding Boolean expression.
2. Translating the Boolean expression into a quantum circuit using Qiskit.
3. Visualizing and outputting the designed circuit.

Steps and Implementation



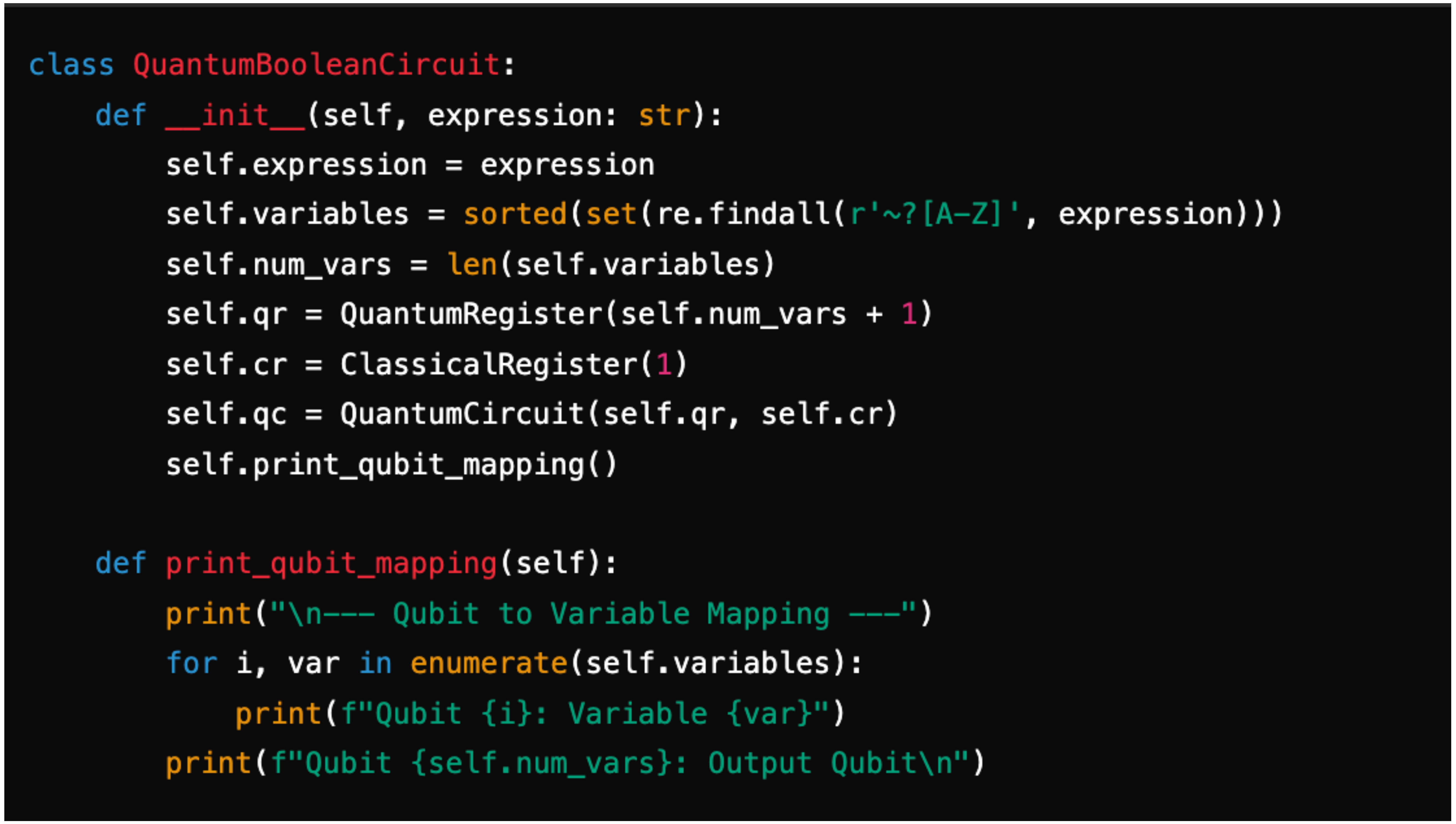
```python
def readCsv(file_path):
    truth_table = []
    with open(file_path, "r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            truth_table.append({key: int(value) for key, value in row.items()})
    return truth_table

file_path = "truth_table.csv"
truth_table = readCsv(file_path)
```

The truth table is stored in a CSV file, with each row representing an input-output mapping of variables. The script reads this file and converts it into a list of dictionaries for further processing.



```
def toBoolean(truth_table):
    rows_with_output_1 = [row for row in truth_table if row["Q"] == 1]
    if not rows_with_output_1:
        return "0"

    terms = []
    for row in rows_with_output_1:
        term = []
        for var, value in row.items():
            if var == "Q":
                continue
            if value == 1:
                term.append(var)
            else:
                term.append(f"~{var}")
        terms.append(" & ".join(term))
    return "(" + ") ^ (".join(terms) + ")"

boolean_expression = toBoolean(truth_table)
print("Boolean Expression:", boolean_expression)
```

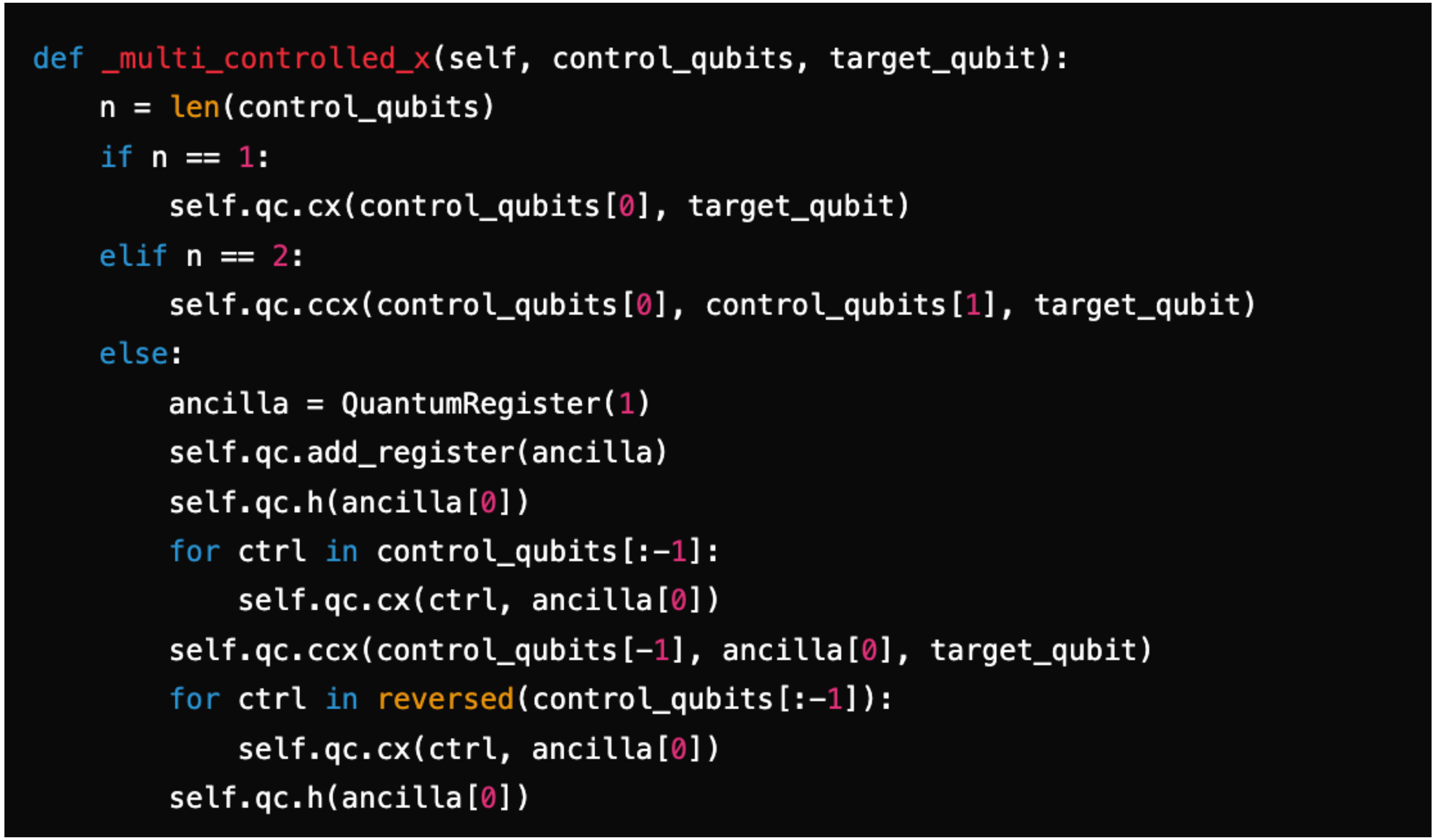
- Each row with `Q=1` is converted into an AND clause of its variables.
- Negations (~) are added for variables with a value of `0`.
- Clauses are combined using XOR (^), resulting in a minimal representation of the Boolean function.



```
class QuantumBooleanCircuit:
    def __init__(self, expression: str):
        self.expression = expression
        self.variables = sorted(set(re.findall(r'~?[A-Z]', expression)))
        self.num_vars = len(self.variables)
        self.qr = QuantumRegister(self.num_vars + 1)
        self.cr = ClassicalRegister(1)
        self.qc = QuantumCircuit(self.qr, self.cr)
        self.print_qubit_mapping()

    def print_qubit_mapping(self):
        print("\n--- Qubit to Variable Mapping ---")
        for i, var in enumerate(self.variables):
            print(f"Qubit {i}: Variable {var}")
        print(f"Qubit {self.num_vars}: Output Qubit\n")
```

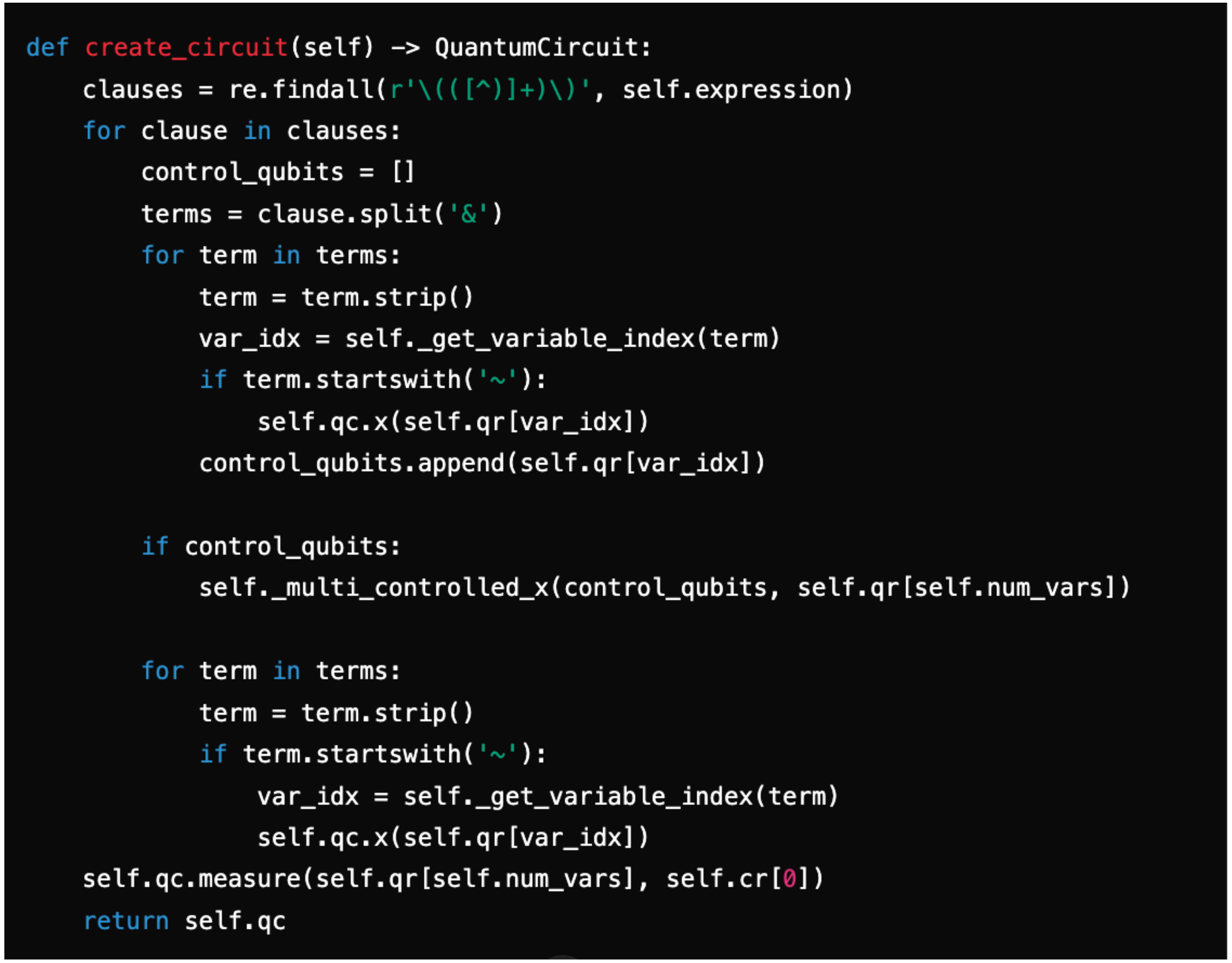
- Initializes quantum and classical registers.
- Maps variables to qubits.
- Translates the Boolean expression into quantum gates.

- **Qubit Mapping**: Each variable in the Boolean expression is assigned a qubit. An additional qubit is reserved for the output.



```
def _multi_controlled_x(self, control_qubits, target_qubit):
    n = len(control_qubits)
    if n == 1:
        self.qc.cx(control_qubits[0], target_qubit)
    elif n == 2:
        self.qc.ccx(control_qubits[0], control_qubits[1], target_qubit)
    else:
        ancilla = QuantumRegister(1)
        self.qc.add_register(ancilla)
        self.qc.h(ancilla[0])
        for ctrl in control_qubits[:-1]:
            self.qc.cx(ctrl, ancilla[0])
        self.qc.ccx(control_qubits[-1], ancilla[0], target_qubit)
        for ctrl in reversed(control_qubits[:-1]):
            self.qc.cx(ctrl, ancilla[0])
        self.qc.h(ancilla[0])
```

Multi-controlled `X` gates are implemented recursively, using ancilla qubits for cases with more than two control qubits.



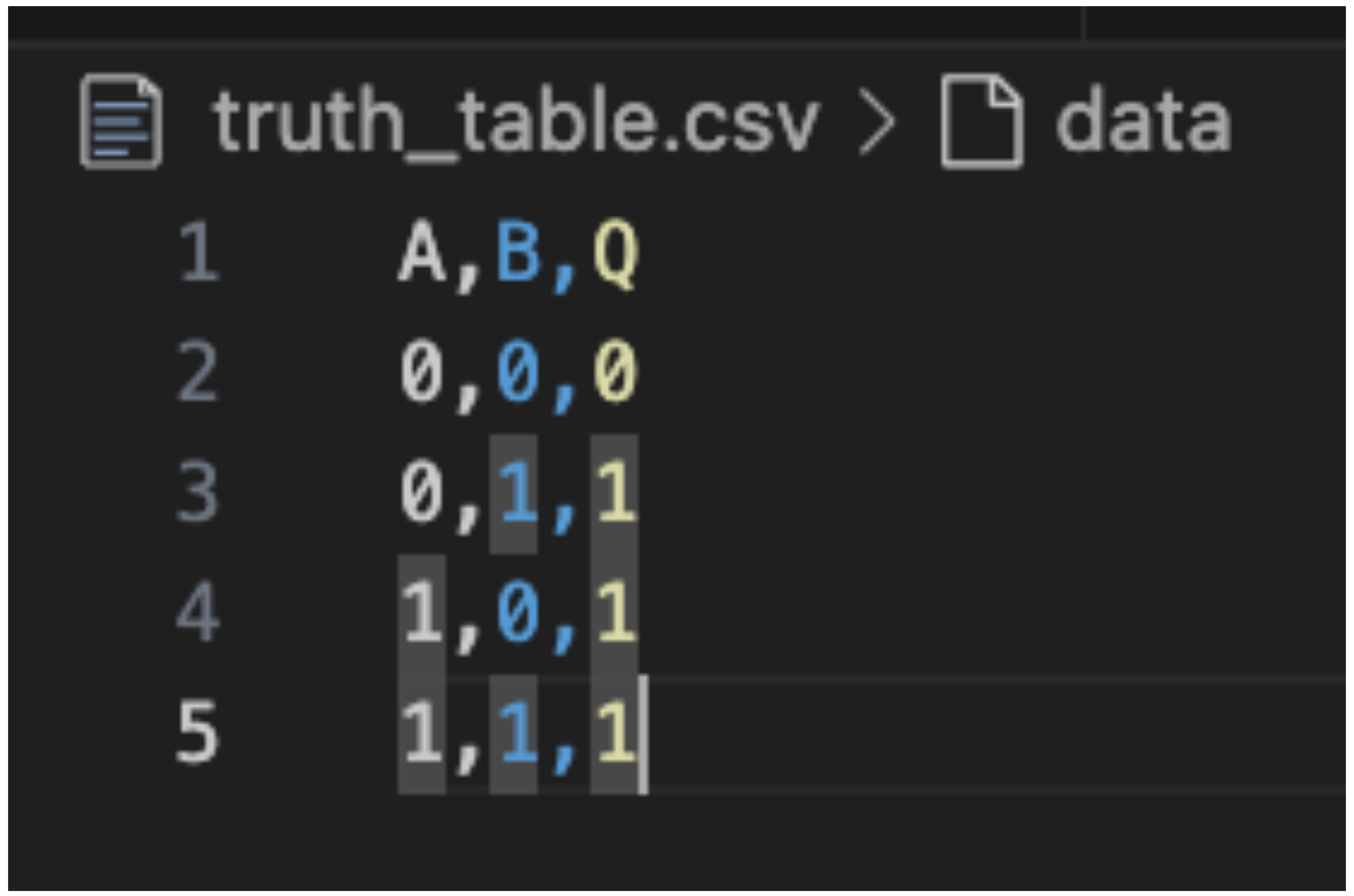
```python
def create_circuit(self) -> QuantumCircuit:
    clauses = re.findall(r'\(([^)]+)\)', self.expression)
    for clause in clauses:
        control_qubits = []
        terms = clause.split('&')
        for term in terms:
            term = term.strip()
            var_idx = self._get_variable_index(term)
            if term.startswith('~'):
                self.qc.x(self.qr[var_idx])
            control_qubits.append(self.qr[var_idx])

        if control_qubits:
            self._multi_controlled_x(control_qubits, self.qr[self.num_vars])

        for term in terms:
            term = term.strip()
            if term.startswith('~'):
                var_idx = self._get_variable_index(term)
                self.qc.x(self.qr[var_idx])
    self.qc.measure(self.qr[self.num_vars], self.cr[0])
    return self.qc
```

The Boolean expression is parsed into clauses, which are then translated into gates:

The output qubit is measured into a classical bit for result verification.

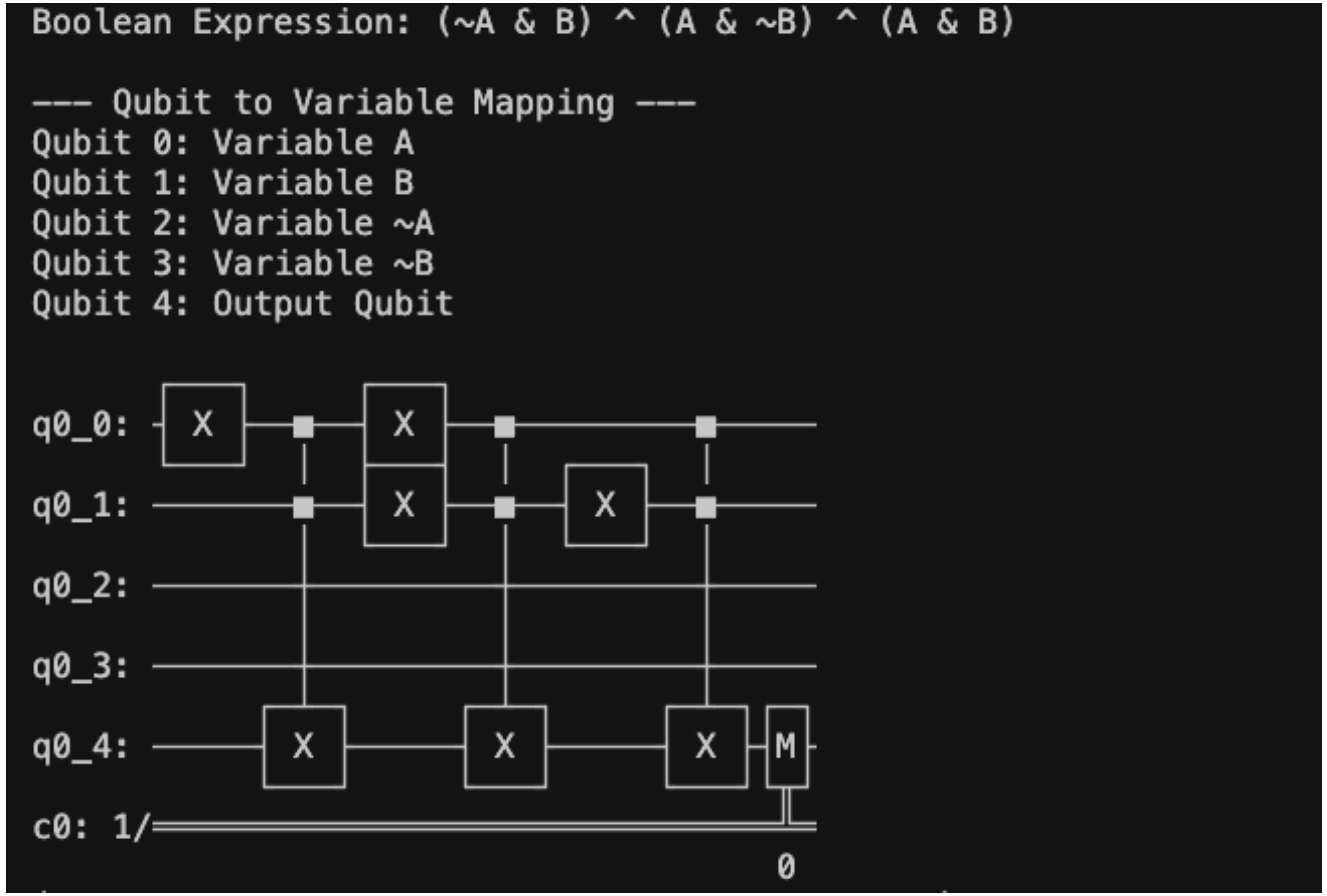
Final Output after Running the code

Example truth table



truth_table.csv > data

```
A,B,Q
0,0,0
0,1,1
1,0,1
1,1,1
```



```
Boolean Expression: (~A & B) ^ (A & ~B) ^ (A & B)

--- Qubit to Variable Mapping ---
Qubit 0: Variable A
Qubit 1: Variable B
Qubit 2: Variable ~A
Qubit 3: Variable ~B
Qubit 4: Output Qubit
```