Circuit Implementations

The Mermin–Peres Magic Square Game in Qiskit

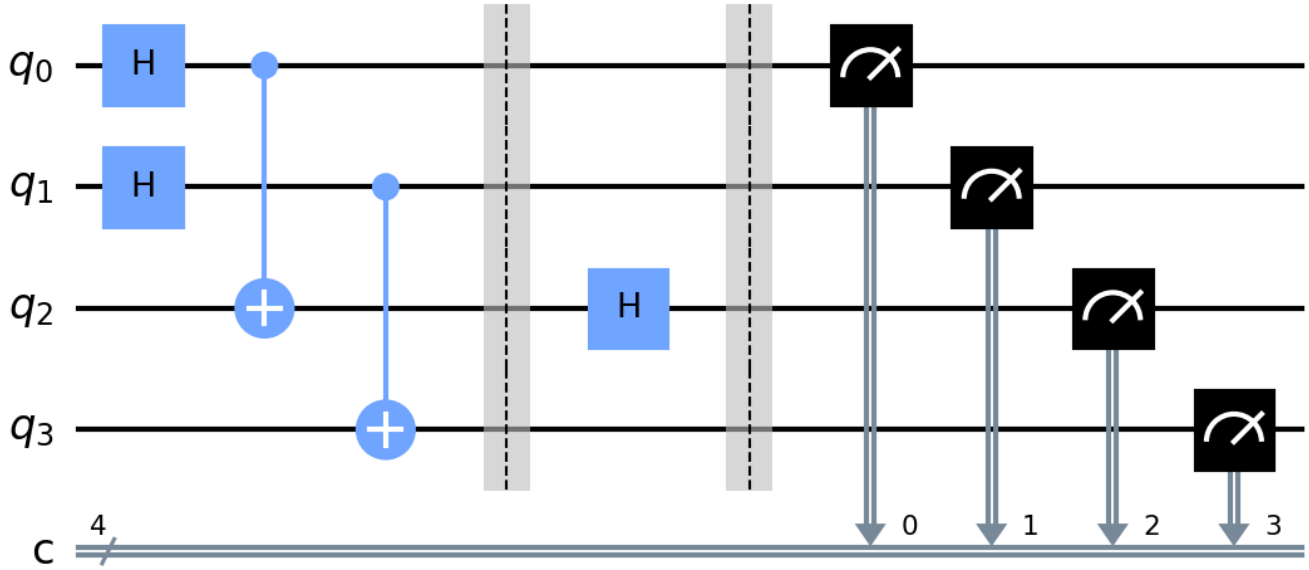
Three worked-out example circuits — from the simplest case to the one requiring joint two-qubit measurement — followed by the complete, tested Python source.

Simulator environment: AerSimulator (backend: aer_simulator), Qiskit Aer v0.17.2, running on Qiskit (Core) v2.5.1 — an ideal, noise-free state-vector simulator; not physical quantum hardware.
Qiskit stands for **Qu**antum **I**nformation **S**cience **Kit** — IBM's open-source software development kit for building, simulating, and running quantum circuits, first released in 2017 (originally spelled "QISKit").

Dipl. Inf. Ramon Schweiss · QWeb © Research

01

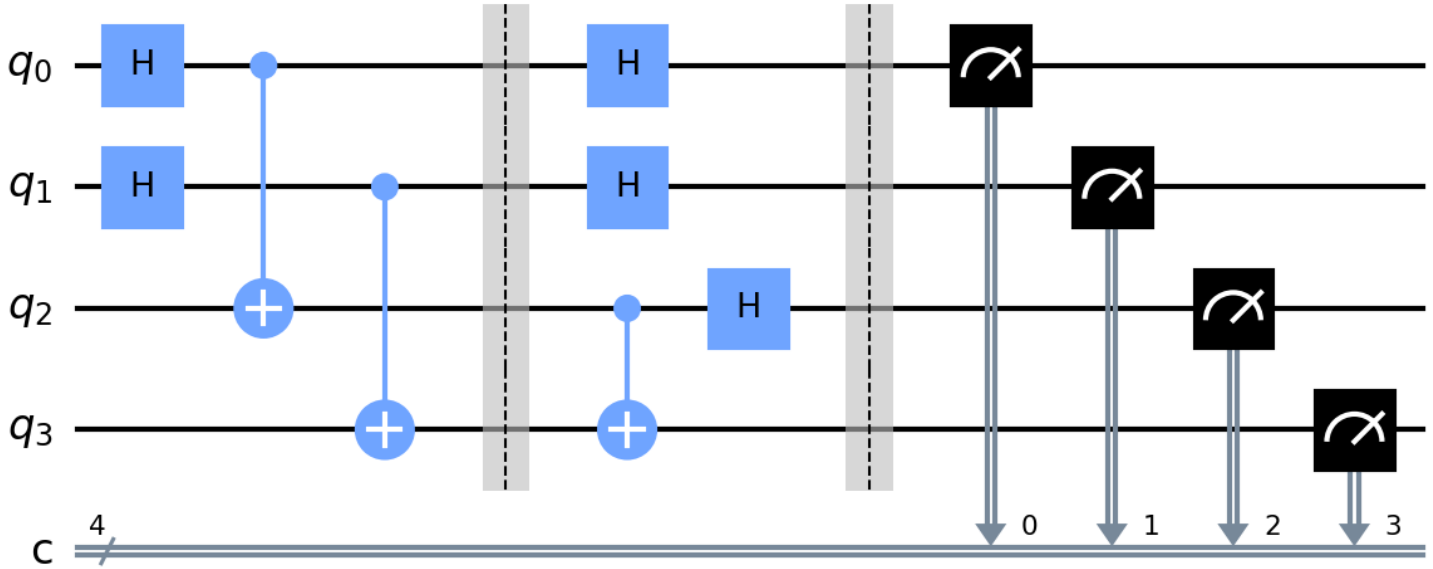
Row 1 / Column 1 — the simplest case



After Bell-pair preparation (H+CNOT on both pairs), Alice needs no further gate at all for row 1 — she measures directly in the Z basis. Bob needs only a single H on B1 for column 1. Not a single entangling gate after preparation.

02

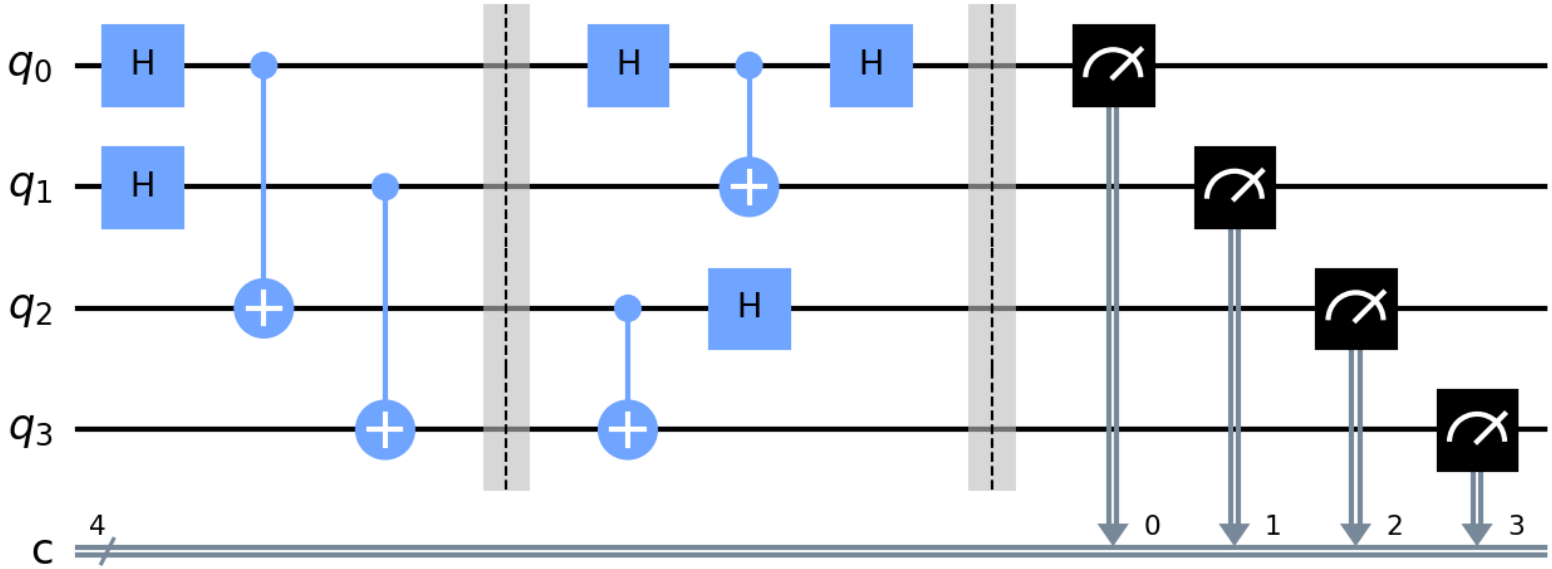
Row 2 / Column 3 — the mixed case



Alice rotates both qubits into the X basis via H — simple, local, no entanglement needed. Bob, however, needs his CNOT+H for column 3: the Bell-state measurement circuit, because his line's operators include Y⊗Y.

03

Row 3 / Column 3 — the most demanding case



Here both sides need their own entangling gate: Alice her H–CNOT–H for row 3, Bob his CNOT–H for column 3. Four gate layers in total after Bell-pair preparation, cleanly organized by the barriers (dashed lines) into preparation → Alice's measurement setup → Bob's measurement setup → measurement.

What stays the same across all three circuits

The first two gate blocks — H on q0, CNOT q0→q2, H on q1, CNOT q1→q3 — always produce the same two Bell pairs. That is the shared resource, identical in every one of the nine (row, column) circuits.

Only the gates that follow differ, depending on which row or column the referee is asking about.

Why row / column 3 are "more expensive"

Y⊗Y forces genuine two-qubit entanglement in the measurement setup, while the other two rows and columns get by with pure single-qubit rotations — the circuit diagrams make that cost visible at a glance.

```
The Mermin-Peres Magic Square Game — Qiskit implementation
============================================================

Two Bell pairs are shared in advance:
    Pair 1: A1 (qubit 0) - B1 (qubit 2)
    Pair 2: A2 (qubit 1) - B2 (qubit 3)
Alice holds A1, A2. Bob holds B1, B2. No communication after this point.

The referee picks a row r in {1,2,3} for Alice and a column c in {1,2,3}
for Bob, independently and uniformly at random. Alice must report three
values (one per column of her row) whose product is +1. Bob must report
three values (one per row of his column) whose product is -1. They win
the round if, in addition, their reported values at the shared cell
(row r, column c) agree.

The operator square (corrected Mermin-Peres construction, with the two
sign-flipped cells in row 3):

               Column 1     Column 2     Column 3
    Row 1        I⊗Z          Z⊗I          Z⊗Z
    Row 2        X⊗I          I⊗X          X⊗X
    Row 3       -X⊗Z         -Z⊗X          Y⊗Y

Row products are all +1, column products are all -1 -- a pattern proven
impossible for any classical (non-communicating) pre-agreed table, which
can win at most 8 of the 9 possible (row, column) combinations. With two
shared Bell pairs, Alice and Bob win every round, with certainty.

Measurement strategy
--------------------
Rows 1, 2 and columns 1, 2 only involve *local* single-qubit Pauli
operators multiplied together, so a single fixed basis choice per qubit
(no entangling gate) reveals all three eigenvalues of the line at once:

  Row 1 / Col 1,2: Z basis (no gate) or a single H on one qubit.
  Row 2:           X basis on both qubits (H, H).

Row 3 and column 3 contain Y⊗Y, which is NOT expressible as independent
single-qubit bases shared across all three operators of the line. Their
three operators only commute as a *group* (like a two-qubit stabilizer
group), so reading out all three eigenvalues at once requires a genuine
two-qubit diagonalizing circuit -- the same idea used in a Bell-state
measurement (as in entanglement swapping / quantum teleportation).

Run this file directly to see one worked-out round and then a long run
of randomly chosen rounds, confirming a 100% win rate.
```

```python
import random
from qiskit import QuantumCircuit , transpile
from qiskit_aer import AerSimulator


# ---------------------------------------------------------------------
# 1. The shared resource: two Bell pairs, Phi+ = (|00> + |11>) / sqrt(2)
#    Qubit layout: 0 = A1, 1 = A2 (Alice's qubits)
#                  2 = B1, 3 = B2 (Bob's qubits)
# ---------------------------------------------------------------------
def build_game_circuit (row: int, col: int) -> QuantumCircuit :
    qc = QuantumCircuit (4, 4)

    # --- prepare the two entangled Bell pairs ---
    qc.h(0)
    qc.cx(0, 2)   # Pair 1: A1-B1
    qc.h(1)
    qc.cx(1, 3)   # Pair 2: A2-B2
    qc.barrier()

    # --- Alice's basis-change circuit, chosen by her row ---
    if row == 1:
        pass                        # I⊗Z, Z⊗I, Z⊗Z -- already diagonal
    elif row == 2:
        qc.h(0)
        qc.h(1)                     # X⊗I, I⊗X, X⊗X -- rotate to X basis
    elif row == 3:
        qc.h(0)
        qc.cx(0, 1)
        qc.h(0)                     # -X⊗Z, -Z⊗X, Y⊗Y -- joint circuit
    else:
        raise ValueError ("row must be 1, 2 or 3 ")

    # --- Bob's basis-change circuit, chosen by his column ---
    if col == 1:
        qc.h(2)                     # I⊗Z, X⊗I, -X⊗Z
    elif col == 2:
        qc.h(3)                     # Z⊗I, I⊗X, -Z⊗X
    elif col == 3:
        qc.cx(2, 3)
        qc.h(2)                     # Z⊗Z, X⊗X, Y⊗Y -- Bell-basis measurement
    else:
        raise ValueError ("col must be 1, 2 or 3 ")

    qc.barrier()
    qc.measure(range(4), range(4))
    return qc
```

```python
# ---------------------------------------------------------------------
# 2. Classical decoding: turn the two measured bits from each side into
#    the three +/-1 eigenvalues of that row / column.
# ---------------------------------------------------------------------
def decode_alice(row: int, b0: int, b1: int) -> dict:
    """b0 = measured bit of A1, b1 = measured bit of A2 (post basis change).
    Returns {column_number: eigenvalue} for Alice's three reported values."""
    if row == 1:
        return {1: (-1) ** b1, 2: (-1) ** b0, 3: (-1) ** (b0 ^ b1)}
    if row == 2:
        return {1: (-1) ** b0, 2: (-1) ** b1, 3: (-1) ** (b0 ^ b1)}
    if row == 3:
        return {1: (-1) ** (b1 ^ 1), 2: (-1) ** (b0 ^ 1), 3: (-1) ** (b0 ^ b1)}
    raise ValueError("row must be 1, 2 or 3 ")


def decode_bob(col: int, d0: int, d1: int) -> dict:
    """d0 = measured bit of B1, d1 = measured bit of B2 (post basis change).
    Returns {row_number: eigenvalue} for Bob's three reported values."""
    if col == 1:
        return {1: (-1) ** d1, 2: (-1) ** d0, 3: (-1) ** (d0 ^ d1 ^ 1)}
    if col == 2:
        return {1: (-1) ** d0, 2: (-1) ** d1, 3: (-1) ** (d0 ^ d1 ^ 1)}
    if col == 3:
        return {1: (-1) ** d1, 2: (-1) ** d0, 3: (-1) ** (d0 ^ d1 ^ 1)}
    raise ValueError("col must be 1, 2 or 3 ")
```

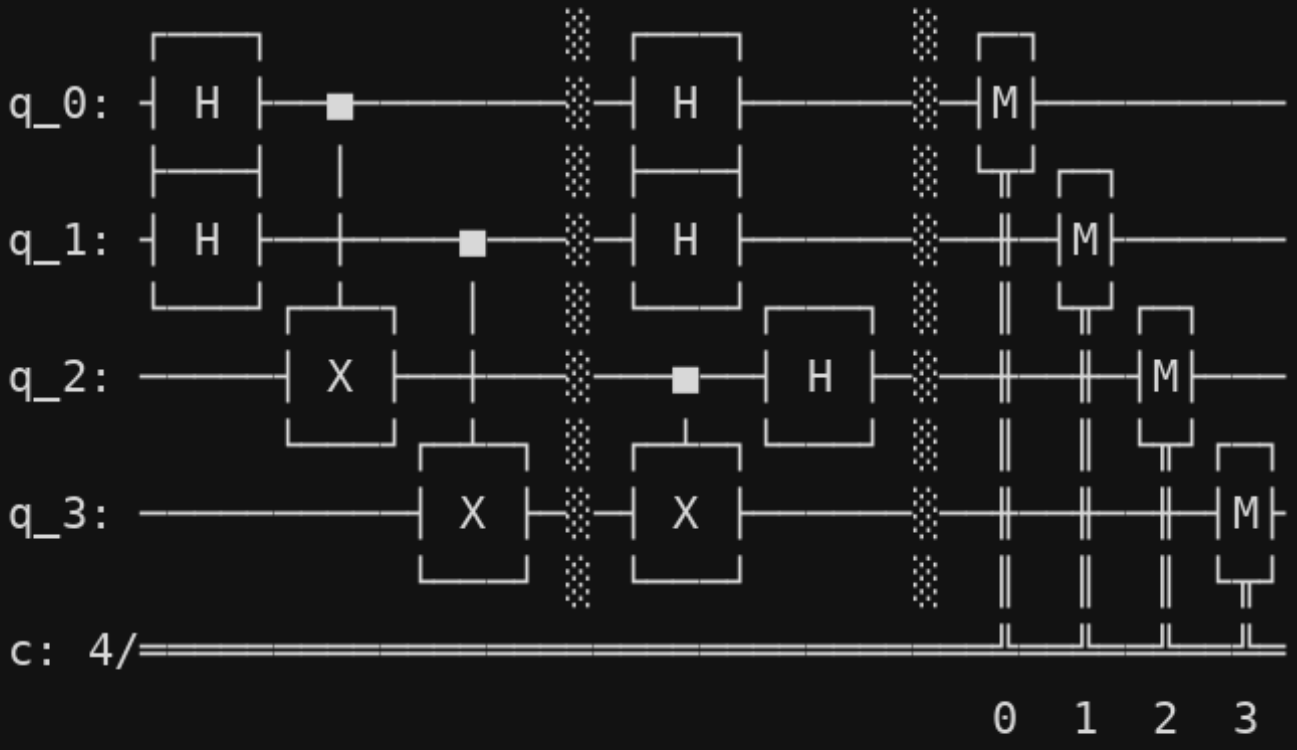
mermin_peres_magic_square.py — part 3/4: Batched shot evaluation against the win condition

```python
# ---------------------------------------------------------------------
# 3. Evaluate many shots of a single (row, col) circuit at once -- far
#    faster than calling simulator.run() once per round.
# ---------------------------------------------------------------------
def evaluate_shots(simulator: AerSimulator, qc_t, row: int, col: int, shots: int):
    result = simulator.run(qc_t, shots=shots, memory=True).result()
    memory = result.get_memory(qc_t)

    wins = 0
    for bitstring in memory:
        bits = bitstring[::-1]   # qubit 0 is the rightmost character
        b0, b1, d0, d1 = (int(bits[i]) for i in range(4))

        alice = decode_alice(row, b0, b1)
        bob = decode_bob(col, d0, d1)

        alice_ok = alice[1] * alice[2] * alice[3] == 1
        bob_ok = bob[1] * bob[2] * bob[3] == -1
        shared_ok = alice[col] == bob[row]

        wins += int(alice_ok and bob_ok and shared_ok)
    return wins
```

```python
if __name__ == "__main__":
    simulator = AerSimulator()

    # Pre-build and transpile all nine (row, col) circuits once.
    circuits = {
        (r, c): transpile(build_game_circuit(r, c), simulator)
        for r in (1, 2, 3) for c in (1, 2, 3)
    }

    # --- One fully worked-out round, shown in detail ---
    row, col = 2, 3
    result = simulator.run(circuits[(row, col)], shots=1, memory=True).result()
    bitstring = result.get_memory(circuits[(row, col)])[0]
    bits = bitstring[::-1]
    b0, b1, d0, d1 = (int(bits[i]) for i in range(4))
    alice = decode_alice(row, b0, b1)
    bob = decode_bob(col, d0, d1)

    print(f"Referee asks: row = {row}, column = {col}\n")
    print(f"Alice's answer (columns 1,2,3):  "
          f"{alice[1]:+d}, {alice[2]:+d}, {alice[3]:+d}   "
          f"(product = {alice[1]*alice[2]*alice[3]:+d}, needs +1)")
    print(f"Bob's answer   (rows 1,2,3):     "
          f"{bob[1]:+d}, {bob[2]:+d}, {bob[3]:+d}   "
          f"(product = {bob[1]*bob[2]*bob[3]:+d}, needs -1)")
    print(f"Shared cell (row {row}, col {col}): "
          f"Alice reports {alice[col]:+d}, Bob reports {bob[row]:+d}  "
          f"-> {'MATCH' if alice[col]==bob[row] else 'MISMATCH'}")
    print()
    print(build_game_circuit(row, col).draw())

    # --- Confirm the win rate for every one of the 9 (row, col)
    #     combinations directly, with many shots each ---
    shots_per_combo = 2000
    print(f"\nWin rate for each (row, column) combination,   "
          f"{shots_per_combo} shots each:")
    total_wins, total_shots = 0, 0
    for row in (1, 2, 3):
        for col in (1, 2, 3):
            wins = evaluate_shots(simulator, circuits[(row, col)], row, col,
                                  shots_per_combo)
            total_wins += wins
            total_shots += shots_per_combo
            print(f"  row={row}, col={col}: {wins}/{shots_per_combo} "
                  f"= {wins/shots_per_combo:.2%}")

    print(f"\nOverall win rate over {total_shots} shots:  "
          f"{total_wins}/{total_shots} = {total_wins/total_shots:.4%}")
```

$ python3 mermin_peres_magic_square.py — output 1/2: one worked-out round in full detail

```
Referee asks: row = 2, column = 3

Alice's answer (columns 1,2,3): -1, +1, -1   (product = +1, needs +1)
Bob's answer   (rows 1,2,3):    +1, -1, +1   (product = -1, needs -1)
Shared cell (row 2, col 3): Alice reports -1, Bob reports -1  -> MATCH
```



$ python3 mermin_peres_magic_square.py — output 2/2: win rate across all nine (row, column) combinations

```
Win rate for each (row, column) combination, 2000 shots each:
  row=1, col=1: 2000/2000 = 100.00%
  row=1, col=2: 2000/2000 = 100.00%
  row=1, col=3: 2000/2000 = 100.00%
  row=2, col=1: 2000/2000 = 100.00%
  row=2, col=2: 2000/2000 = 100.00%
  row=2, col=3: 2000/2000 = 100.00%
  row=3, col=1: 2000/2000 = 100.00%
  row=3, col=2: 2000/2000 = 100.00%
  row=3, col=3: 2000/2000 = 100.00%

Overall win rate over 18000 shots: 18000/18000 = 100.0000%
```