DD2367 - Programming Assignment I: Quantum Random Number Generation (QRNG) on IBM Quantum Computers

Authors: Scott McHaffie, Jai Iyer, Venkatesh Elayaraja

Task 0

In [1]:
```
import qiskit, qiskit_ibm_runtime, qiskit_aer
print(qiskit.__version__, qiskit_ibm_runtime.__version__, qiskit_aer.__versi
```

```
2.1.2 0.41.1 0.17.1
```

Task 1

In [2]:
```
from dotenv import load_dotenv
import os

load_dotenv()  # take variables from .env

api_key = os.getenv("API_KEY")
crn = os.getenv("INSTANCE_CRN")
```

In [3]:
```
# imports
from qiskit_ibm_runtime import QiskitRuntimeService
# >>> Edit these two lines:
TOKEN = api_key    # REQUIRED
INSTANCE = crn # OPTIONAL: e.g., "crn:v1:bluemix:public:quantum-computing:us

# Safety check to avoid empty tokens
if not TOKEN or TOKEN.strip() in {"", "<PASTE-YOUR-IBM-QUANTUM-API-KEY-HERE>
    raise ValueError("Please paste your IBM Quantum API key into TOKEN (betw

# Create the service directly (no saved account needed)
service = QiskitRuntimeService(
    channel="ibm_quantum_platform",
    token=TOKEN.strip(),
    instance=(INSTANCE.strip() if isinstance(INSTANCE, str) and INSTANCE.str
)

# Quick sanity check
backends = service.backends(operational=True, simulator=False)
print("OK. Found", len(backends), "real backends. Example:", [b.name for b i
```

```
OK. Found 2 real backends. Example: ['ibm_brisbane', 'ibm_torino']
```

In [4]:
```
# save the CRN and API for future use

QiskitRuntimeService.save_account(
    channel="ibm_quantum_platform",
    token=TOKEN.strip(),
    instance=(INSTANCE.strip() if isinstance(INSTANCE, str) and INSTANCE.str
    set_as_default=True,
    overwrite=True,
)
print("Saved default account for this runtime.")
```

```
Saved default account for this runtime.
```

In [33]:
```
cands = service.backends(simulator=False, operational=True, min_num_qubits=6
# for b in cands: print(b.name, b.num_qubits)

A = service.least_busy(simulator=False, operational=True, min_num_qubits=6)
B = next(b for b in cands if b.name != A.name)

print ("backend A:", A.name, "with", A.num_qubits, "qubits")
print ("backend B:", B.name, "with", B.num_qubits, "qubits")
```

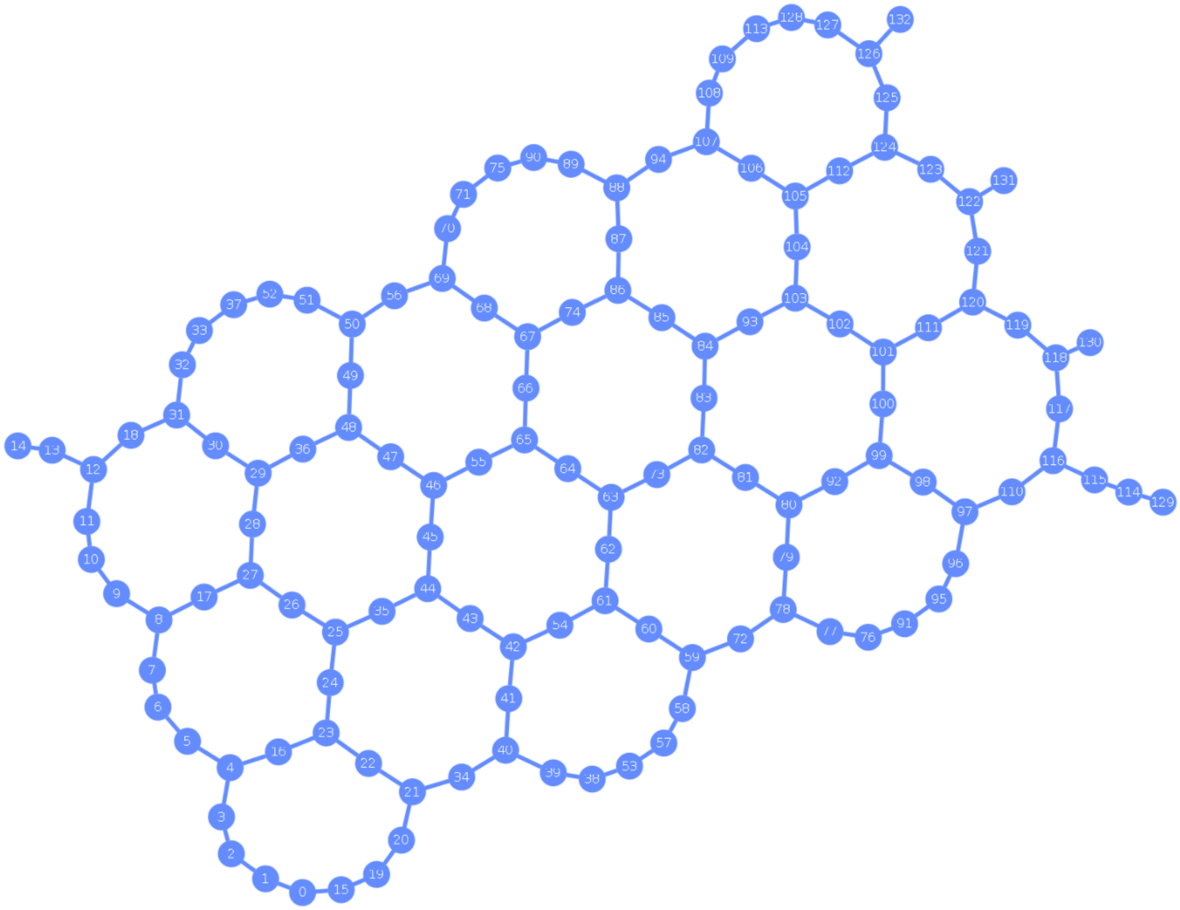
```
backend A: ibm_torino with 133 qubits
backend B: ibm_brisbane with 127 qubits
```

In [6]:
```
cfgA = A.configuration(); cfgB = B.configuration()
print("A basis_gates:", cfgA.basis_gates)
print("B basis_gates:", cfgB.basis_gates)
cmapA = A.coupling_map; cmapB = B.coupling_map
```

```
A basis_gates: ['cz', 'id', 'rz', 'sx', 'x']
B basis_gates: ['ecr', 'id', 'rz', 'sx', 'x']
```

In [7]:
```
from qiskit.visualization import plot_coupling_map
plot_coupling_map(A.num_qubits, None, cmapA.get_edges())
```

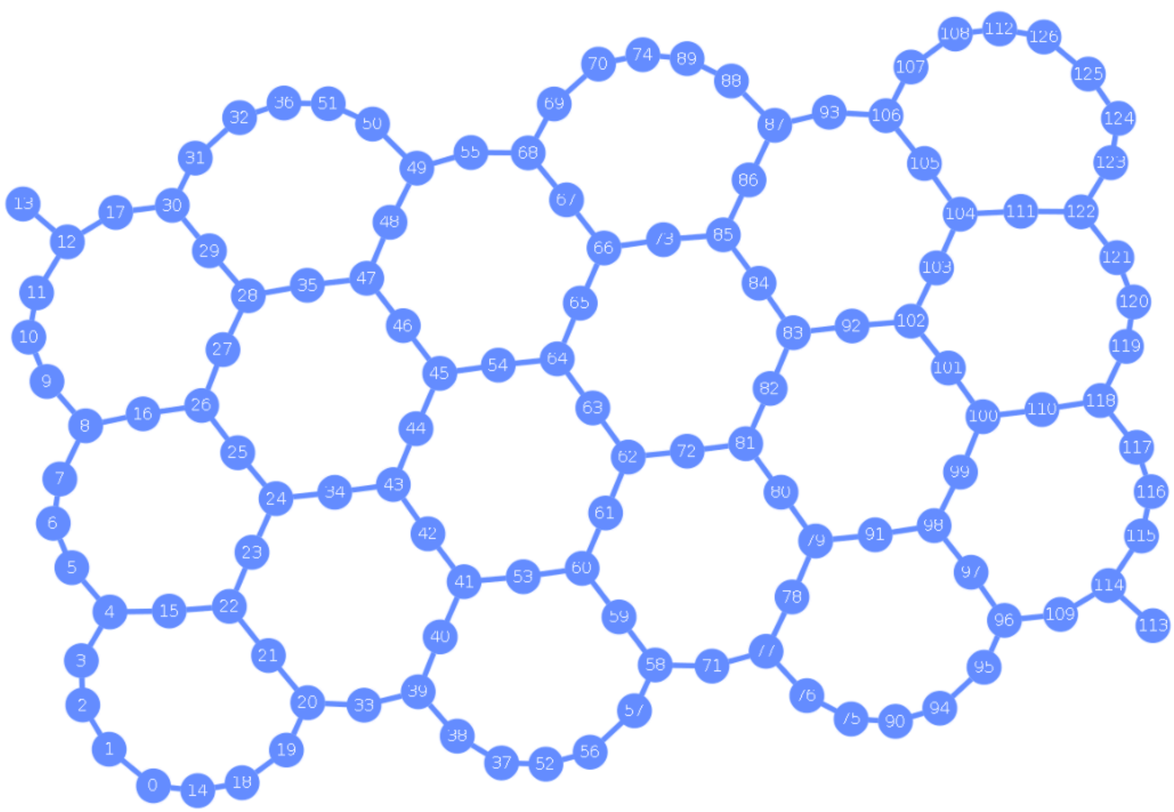
Out[7]:



In [8]:

```
plot_coupling_map(B.num_qubits, None, cmapB.get_edges())
```

Out[8]:



Task 2

In [9]:
```python
from qiskit import QuantumCircuit

# One-bit QRNG
# qc1 = QuantumCircuit(1)
# qc1.h(0)          # coin-flip on qubit 0
# qc1.measure_all() # record the outcome as a classical bit
# qc1.draw()

# k-bit QRNG
def qrng(k: int):
    qc = QuantumCircuit(k)
    for q in range(k):
        qc.h(q)          # one coin-flip per qubit
    qc.measure_all()
    return qc

k = 3
qc = qrng(k)
# print (qc)
qc.draw()
```

Out[9]:
```
        ┌───┐ ░ ┌─┐      
   q_0: ┤ H ├─░─┤M├──────
        ├───┤ ░ └╥┘┌─┐   
   q_1: ┤ H ├─░──╫─┤M├───
        ├───┤ ░  ║ └╥┘┌─┐
   q_2: ┤ H ├─░──╫──╫─┤M├
        └───┘ ░  ║  ║ └╥┘
meas: 3/═════════╩══╩══╩═
                 0  1  2 
```

Review. We designed a 3-Qubit Random Number Generator circuit using only **single-qubit Hadamard gates**, and single qubit measurements in the computational basis $\{|0\rangle, |1\rangle\}$. This circuit is comprised of 3 stages:

1. **Qubit State Preparation**: Each of the three qubits (labelled q_0, q_1, and q_2) are initialised to the basis state $|0\rangle$.
2. **Uniform Randomization**: We individually transform each qubit (q_i) to the equal superposition state $|q_i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ for $i = \{0, 1, 2\}$. This primes the measurement outcome for each qubit state $\{|0\rangle, |1\rangle\}$ to have equal probablility (50%).
3. **Measurement**: Each qubit is measured in the computational basis individually, and we report a 3-bit string of $\{0, 1\}$ corresponding to each measurment outcome of $\{|0\rangle, |1\rangle\}$ respectively.

Task 3

In [21]:

```
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manag

pmA = generate_preset_pass_manager(optimization_level=3, backend=A)
isaA = pmA.run(qc)

pmB = generate_preset_pass_manager(optimization_level=3, backend=B)
isaB = pmB.run(qc)

print("A ops:", isaA.count_ops(), "depth:", isaA.depth())
print("B ops:", isaB.count_ops(), "depth:", isaB.depth(), "\n")

# (Optional) See which physical qubits were chosen for your logical qubits 0
print("ISA A initial_index_layout:", isaA.layout.initial_index_layout())
print("ISA A routing_permutation: ", isaA.layout.routing_permutation())
print("ISA A final_index_layout:  ", isaA.layout.final_index_layout(), "\n")

print("ISA B initial_index_layout", isaB.layout.initial_index_layout())
print("ISA B routing_permutation", isaB.layout.routing_permutation())
print("ISA B final_index_layout:  ", isaB.layout.final_index_layout(), "\n")

# (Optional) Peek at the device’s native gate names (you don't need to know
print("A basis gates:", A.configuration().basis_gates)
print("B basis gates:", B.configuration().basis_gates, "\n")
# Draw the transpiled circuit
isaA.draw()
```

```
A ops: OrderedDict([('rz', 6), ('sx', 3), ('measure', 3), ('barrier', 1)]) d
epth: 4
B ops: OrderedDict([('rz', 6), ('sx', 3), ('measure', 3), ('barrier', 1)]) d
epth: 4

ISA A initial_index_layout: [37, 87, 117, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 10
6, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 118, 119, 120, 121, 12
2, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132]
ISA A routing_permutation:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1
4, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
3, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 5
2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 7
1, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 9
0, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 10
7, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 12
2, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132]
ISA A final_index_layout:   [37, 87, 117]

ISA B initial_index_layout [118, 103, 90, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 104, 105, 10
6, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 119, 120, 121, 12
2, 123, 124, 125, 126]
ISA B routing_permutation [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107,
108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 1
23, 124, 125, 126]
ISA B final_index_layout:   [118, 103, 90]

A basis gates: ['cz', 'id', 'rz', 'sx', 'x']
B basis gates: ['ecr', 'id', 'rz', 'sx', 'x']
```

Out[21]:

```
global phase: 3π/4
               ┌─────────┐┌────┐┌─────────┐ ░ ┌─┐
  q_0 -> 37 ─┤ Rz(π/2) ├┤ √X ├┤ Rz(π/2) ├─░─┤M├──────
               ├─────────┤├────┤├─────────┤ ░ └╥┘┌─┐
  q_1 -> 87 ─┤ Rz(π/2) ├┤ √X ├┤ Rz(π/2) ├─░──╫─┤M├───
               ├─────────┤├────┤├─────────┤ ░  ║ └╥┘┌─┐
 q_2 -> 117 ─┤ Rz(π/2) ├┤ √X ├┤ Rz(π/2) ├─░──╫──╫─┤M├
               └─────────┘└────┘└─────────┘ ░  ║  ║ └╥┘
    meas: 3/═══════════════════════════════════╩══╩══╩═
                                               0  1  2
```

Review. We address how the transpiler assigns qubits for a quantum circuit, for two different quantum computers.

1. The instruction counts and gates used for both quantum computers are identical, since the implementation of this circuit requires only **rz** and **sx** gates which both quantum computers use.

2. There are no 2-qubit operations in the implementation of this circuit in either quantum computer. This concurs with the theoretical circuit design.

3. The transpiler assigned our logical qubits to physical qubits as $q_0 \to 37, q_1 \to 87, q_2 \to 117$. The physical qubits are distant from one another (≥ 10 qubits in between each pair).

   3.1 `isaA.layout.initial_index_layout()` consists of all the qubits assigned for implementation of the circuit.

   3.2 `isaA.layout.routing_permutation()` is a mapping which indicates the order in which qubit operations are conducted, including **SWAP** operations. This happens when the circuit contains multi-qubit operations, and the assigned qubits are distant from one another. In our case, the array returned is the trivial version (`[0,1,2,...]`) since our circuit has no multi-qubit gates.

   3.3 `isaA.layout.final_index_layout()` consists of the final positions of the qubits in the order of operations (including **SWAP** operations) assigned for the implementation of the circuit. Additionally, this does not include unassigned qubits, as in the previous two arrays.

In [11]:
```
from qiskit_ibm_runtime import SamplerV2 as Sampler

# Hardware: target a specific backend (backend B)
sampler = Sampler(mode=B) # or mode=B
resultB = sampler.run([isaB], shots=4000).result()
countsB = resultB[0].data.meas.get_counts() # {'010011': n, ...}
total = sum(countsB.values())
probs = {bitstr: count / total for bitstr, count in countsB.items()}
print (probs)
```

```
{'100': 0.12625, '001': 0.126, '110': 0.123, '111': 0.12975, '011': 0.125,
'101': 0.13125, '010': 0.12125, '000': 0.1175}
```

In [36]:
```
# Simulator with the same result schema (backend B), for use in Task
from qiskit.primitives import BackendSamplerV2
from qiskit_aer import AerSimulator

sim_resultB = BackendSamplerV2(backend=AerSimulator()).run([isaB], shots=400
sim_countsB = sim_resultB[0].data.meas.get_counts()
sim_total = sum(sim_countsB.values())
```

```
sim_probs = {bitstr: count / sim_total for bitstr, count in sim_countsB.item
print (sim_probs)
```

```
{'000': 0.12225, '001': 0.116, '110': 0.1355, '011': 0.123, '010': 0.12425,
'100': 0.11825, '111': 0.12675, '101': 0.134}
```

Task 4

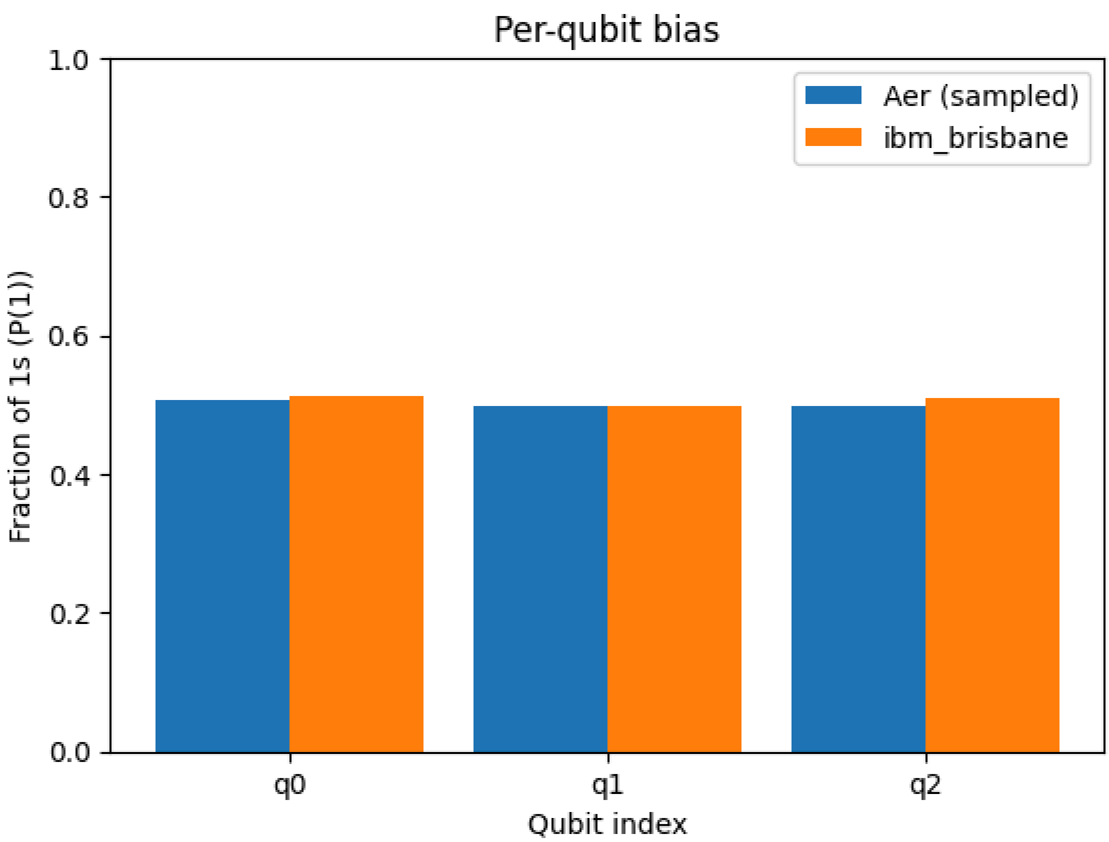
In [13]:
```
# Sampled counts from simulator A. We skip this cell due to extremely long r
# sim_resultA = BackendSamplerV2(backend=AerSimulator()).run([isaA], shots=4
# sim_countsB = sim_resultB[0].data.meas.get_counts()
# sim_countsA = sim_resultA[0].data.meas.get_counts()
```

In [14]:
```
# Sampled counts from backend A, we skip this cell due to extremely long rur
# resultA = Sampler(mode=A).run([isaA], shots=4000).result()
# countsA = resultA[0].data.meas.get_counts()
```

In [15]:
```
# Sampled counts from backend B and simulations based on backend B are alrea
```

In [16]:
```
import numpy as np, matplotlib.pyplot as plt

def per_qubit_p1(counts, n):
    shots = sum(counts.values())
    p = np.zeros(n, dtype=float)
    for s, c in counts.items():                       # s like '0101' (qubit 0 is
        for j, ch in enumerate(reversed(s)):          # map column 0 -> qubit 0
            if ch == '1':
                p[j] += c
    return p / max(shots, 1)

k = qc.num_qubits  # or isaA.num_qubits
p_sim = per_qubit_p1(sim_countsB, k)
p_B   = per_qubit_p1(countsB,     k)

x = np.arange(k); w = 0.42
fig, ax = plt.subplots()
# Plot the bars
ax.bar(x - w/2, p_sim, width=w, label="Aer (sampled)")
ax.bar(x + w/2, p_B,   width=w, label=B.name)

ax.set_xlabel("Qubit index")
ax.set_ylabel("Fraction of 1s (P(1))")
ax.set_title("Per-qubit bias")
ax.set_xticks(x, [f"q{j}" for j in range(k)])
ax.set_ylim(0, 1)
ax.legend()
plt.tight_layout()

# Table data
columns = ["Qubit", "Aer (sampled)", B.name]
cell_text = [[f"q{j}", f"{p_sim[j]:.3f}", f"{p_B[j]:.3f}"] for j in range(k)

# Add table below the plot
```

```
table = plt.table(
    cellText=cell_text,
    colLabels=columns,
    loc="bottom",
    cellLoc="center",
    bbox=[0, -0.5, 1, 0.3]
)

# Adjust layout so table fits under plot
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.show()
```



Qubit	Aer (sampled)	ibm_brisbane
q0	0.506	0.512
q1	0.499	0.499
q2	0.498	0.510

Bonus Tasks

B.1 Monobit bias (per qubit and overall)

In [22]:
```
import numpy as np
```

```python
# defining an array to make printing easier in this Bonus section
name_list = np.array([countsB, sim_countsB])
result_list = np.array([resultB, sim_resultB])
print_list = np.array(["Backend B", "Simulation of Backend B"])

def monobit_summary(counts, res, k):
    shots = sum(counts.values())
    bitstrings = res[0].data.meas.get_bitstrings()
    M = np.array([[int(b) for b in s[::-1]] for s in bitstrings], dtype=int)
    p = M.mean(axis=0)                       # per-qubit fraction of 1s
    overall = float(p.mean())
    se = np.sqrt(0.25/shots)                 # rough expected fluctuation for a fa
    suspect = np.abs(p - 0.5) > 3*se         # rule-of-thumb: outside ±3·SE
    return p, overall, se, suspect

for name, res_name, print_name in zip (name_list, result_list, print_list):
    p, overall, se, flag = monobit_summary(name, res_name, k)
    print (print_name, "per-qubit P(1):", np.round(p, 3), "overall:", round(
    print (print_name, "suspect qubits:", np.where(flag)[0].tolist(), "\n")
```

```
Backend B per-qubit P(1): [0.512 0.499 0.51 ] overall: 0.507 SED~ 0.0079
Backend B suspect qubits: []

Simulation of Backend B per-qubit P(1): [0.506 0.5   0.498] overall: 0.501 S
ED~ 0.0079
Simulation of Backend B suspect qubits: []
```

B.2 Runs test (temporal alternation)

In [23]:

```python
def runs_fraction_per_qubit(counts, res, k):
    bitstrings = res[0].data.meas.get_bitstrings()
    M = np.array([[int(b) for b in s[::-1]] for s in bitstrings], dtype=int)
    flips = (M[1:] != M[:-1]).mean(axis=0)    # fraction of shot-to-shot flip
    return flips

for name, res_name, print_name in zip (name_list, result_list, print_list):
    flips = runs_fraction_per_qubit(name, res_name, k)
    print (print_name, "runs (flip fraction) per qubit:", np.round(flips, 3)
```

```
Backend B runs (flip fraction) per qubit: [0.491 0.499 0.477]

Simulation of Backend B runs (flip fraction) per qubit: [0.502 0.506 0.502]
```

B.3 Lag-1 autocorrelation (temporal dependence)

In [24]:

```python
def autocorr_lag1(counts, res, k):
    bitstrings = res[0].data.meas.get_bitstrings()
    M = np.array([[int(b) for b in s[::-1]] for s in bitstrings], dtype=int)
    X = M - M.mean(axis=0, keepdims=True)
    num = (X[1:]*X[:-1]).sum(axis=0)
    den = (X[:-1]**2).sum(axis=0)
    ac1 = np.divide(num, den, out=np.zeros_like(num, dtype=float), where=der
```

```
    return ac1

for name, res_name, print_name in zip (name_list, result_list, print_list):
    ac1 = autocorr_lag1(name, res_name, k)
    print (print_name, "lag-1 autocorr per qubit:", np.round(ac1, 3), "\n")
```

```
Backend B lag-1 autocorr per qubit: [0.018 0.002 0.045]

Simulation of Backend B lag-1 autocorr per qubit: [-0.003 -0.011 -0.004]
```

B.4 Inter-qubit correlation (spatial dependence)

In [25]:
```
import itertools

def interqubit_corr(counts, res, k):
    bitstrings = res[0].data.meas.get_bitstrings()
    M = np.array([[int(b) for b in s[::-1]] for s in bitstrings], dtype=int)
    X = M - M.mean(axis=0, keepdims=True)
    cov = (X.T @ X) / (len(M)-1)
    std = X.std(axis=0, ddof=1)
    R = cov / (std[:,None]*std[None,:])
    np.fill_diagonal(R, 1.0)
    return R

for name, res_name, print_name in zip (name_list, result_list, print_list):
    R = interqubit_corr(name, res_name, k)
    flags = [(i,j,float(R[i,j])) for i,j in itertools.combinations(range(k),
    print (print_name, "suspicious pairs:", flags[:10])
```

```
Backend B suspicious pairs: []
Simulation of Backend B suspicious pairs: []
```

B.5 Interpretation

Monobit bias: the computer had an overall p-value of 5.07, which is a deviation of only 0.07 from the ideal p-value of 0.5. The p-value of the computer varied only slightly from the p-value of the simulator, which was 0.502.

Temporal alternation: qubits 0 and 1 of the computer performed well, achieving shot-to-shot flip fractions of 0.491 and 0.499, respectively. Qubit 2 however achieved a shot-to-shot flip fraction of only 0.477, indicating its tendency to "stick" towards 0.

Autocorrelation: The quantum computer showed higher autocorrelations for qubits q_0 (0.018) and q_2 (0.045) compared to the simulator, yet these values are still very low, demonstrating that the overall stickiness of the qubits are fairly low.

Inter-qubit correlation: The selected qubit set showed no suspicious pairs for the $R \geq 0.1$ threshold. This concurs with the simulator result, demonstrating that the qubits in the quantum computer are evolving as single qubit systems.

Computer-to-computer comparisons: we were not able to compare the Torino and Brisbane backends due to runtime issues with IBM Torino.

Generative AI Disclosure

We used ChatGPT-5 and ChatGPT-4 for AI assistance during this assignment. A breakdown of the usage per task is shown below:

0. Setup: Initially we used ChatGPT to help us with the keywords to initialize and clone a public GitHub repository for version control and collaboration. The conversation can be found in full here. We validated this by pushing changes and seeing that the repository was initialized and behaving as expected.
1. Task 1: We used ChatGPT to create a hidden .env folder where we could store our API key locally and read it into a given variable without leaking it on GitHub. The conversation is found here. We validated this by consulting package documentation.
2. Task 2: We did not use any AI tools.
3. Task 3: We did not use any AI tools.
4. Task 4: We used ChatGPT to help us with syntax for creating a table with the empirical fraction of 1s per qubit. The conversation is found here. We validated the code by comparing the values in the table to the values in each array and improved the aesthetics of the plot by manually adjusting the location of the table.
5. Bonus: We used ChatGPT to understand the syntax of the bitstring slicing. Specifically, to understand that the two semicolons were reversing each bit in the bitstring. The conversation is found here.