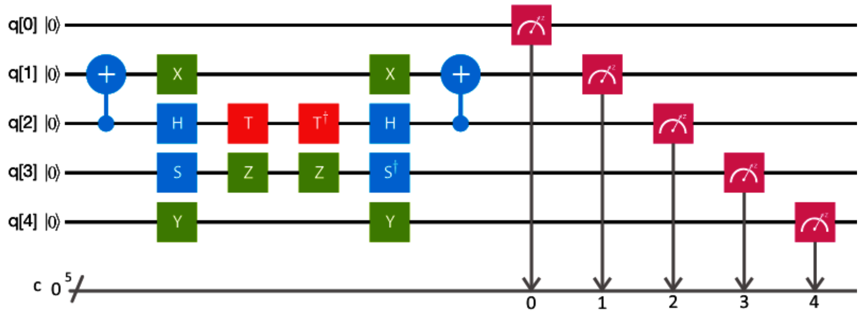
Challenges



1. Implement a function `create_circuit_matrix(qiskit_representation)` that constructs the overall matrix for the circuit shown in the image. The function should ignore measurement operations. If the parameter `qiskit_representation` is `True`, return the matrix in Qiskit's qubit ordering format. If `False`, return the matrix in the standard qubit ordering.

2. Implement a function `create_EPRPair(input_state, shots, seed)` that initializes a single-qubit quantum state from `input_state` and prepares a Bell state using a quantum circuit. The function should simulate the circuit with the specified `seed` and return measurement results (counts).

3. Implement a function `create_W_state(shots, seed)` that generates a W state starting from the initial state $|000\rangle$. The function should use the provided `shots` to specify the number of measurements and the `seed` for the simulator's random number generator. The circuit should prepare the W state and return the measurement results (counts).

4. Write a function `prepare_ghz_state(num_qubits, shots, seed)` that generates a generalized $|GHZ\rangle$ state for a given number of qubits (`num_qubits`). The function should take the number of measurements (`shots`) and a seed (`seed`) for reproducibility. After preparing the GHZ state, simulate the circuit, perform the specified number of measurements, and return the measurement results (counts).

5. Write a function `normalize_state(alpha, beta)` that accepts two complex numbers, `alpha` and `beta`, representing the coefficients of an unnormalized quantum state $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$. The function should return the normalized state $|\psi'\rangle=\alpha'|0\rangle+\beta'|1\rangle$ (ensuring that $|\alpha'|2+|\beta'|2=1$) as an array.

6. Write a function `inner_product(state1, state2)` that calculates the inner product ⟨state1|state2⟩ between two quantum states, represented as arrays of complex coefficients. The function should:

- Verify that both states are normalized; raise an error with the message `'state is not normalized'` if either state is not normalized.
- Ensure that the states have the same dimension; raise an error with the message `'the vectors do not have the same dimension'` if they differ.
- Compute and return the inner product.
- The input states are provided as row vectors.

7. Write a function `measure_state(state_vector, shots, seed)` that simulates the measurement of a quantum state. The input `state_vector` is a list or array of complex coefficients representing the state, `shots` is the number of measurements to perform, and `seed` ensures reproducibility. The function should simulate the specified number of measurements and return a list of individual measurement outcomes (memory) from the simulation.

8. Write a function `is_quantum_gate(matrix)` that determines whether a given matrix represents a valid quantum gate. The function should return `True` if the matrix satisfies the necessary conditions for a quantum gate and `False` otherwise.

9. Write a function `kronecker_product(state1, state2)` that computes the Kronecker product of two quantum states, each represented as a list or array of complex coefficients. The function should return the resulting combined quantum state as an array of complex coefficients.

10. Write a function `quantum_fourier_transform(num_qubits, shots, seed, input_state)` that creates and simulates a Quantum Fourier Transform (QFT) circuit. The function should:

- Initialize the qubits with the given `input_state`.
- Apply the QFT to the qubits.
- Perform the specified number of `shots` measurements.
- Use the provided `seed` for reproducibility.
- Include the final swap of the algorithm

11. Write a function `inverse_quantum_fourier_transform(num_qubits, shots, seed, input_state)` that creates and simulates an inverse Quantum Fourier Transform (QFT) circuit. The function should:

- Initialize the qubits with the provided `input_state`.
- Apply the inverse QFT to the qubits.
- Include the final swap to the algorithm
- Perform the specified number of `shots` measurements.
- Use the given `seed` for reproducibility.
- The function should return a list of measurement outcomes from the simulation (counts).

12. Write a function `qft_then_inverse_qft(num_qubits, input_state)` that constructs a quantum circuit for a given number of qubits (`num_qubits`) and an `input_state` represented as a list or array of complex coefficients. The circuit should apply the Quantum Fourier Transform (QFT) followed by its inverse to the `input_state`. The function should return the final state vector after these operations.

13. Write a function `prepare_binary_state(a)` that takes an integer `a` as input and creates a quantum circuit to prepare a quantum state representing the binary encoding of `a`. The binary representation should map `qubit_0` to the most significant bit (MSB) and `qubit_n` to the least significant bit (LSB), with `qubit_0` being the first qubit in the circuit as per standard notation. Combine all operations into a single gate and return the resulting gate as a quantum circuit.

14. Write a function `quantum_addition(a, b)` that performs quantum addition of two integers, `a` and `b`. The function should:

- Prepare the input quantum state based on the binary representation of `a`, with `qubit_0` as the least significant bit (LSB) and `qubit_n` as the most significant bit (MSB), where `qubit_0` is the first qubit in the circuit, as conventionally represented.
- Construct a quantum circuit that adds `b` to `a`, ensuring the resulting quantum state encodes the binary representation of a+b
- Combine all operations into a single gate and return the resulting gate as a quantum circuit.

15. Write a function `quantum_modular_addition(a, b, N)` that performs modular addition of two integers a, b mod N. The function should:

- Prepare the input quantum state based on the binary representation of aaa, with `qubit_0` as the least significant bit (LSB) and `qubit_n` as the most significant bit (MSB), where `qubit_0` is the first qubit in the circuit, as conventionally represented.
- Construct a quantum circuit that computes (a+b)mod N, ensuring the resulting quantum state encodes the binary representation of the modular sum.
- Combine all operations into a single gate and return the resulting gate as a quantum circuit.
    - **Note**: Ensure that a and b are less than N.

16. Write a function `bit_flip_code(input_state, error)` that implements a repetition code for detecting bit-flip errors. The function should:

- Encode a single-qubit quantum state (`input_state`) into a three-qubit (repetition code) state using a stabilizer code.
- Apply bit-flip errors (X gates) based on the qubit indices provided in the `error` list.
- Decode the three-qubit state and return the error syndrome after the decoding process.
- The function should use stabilizer code principles to detect bit-flip errors.

17. Write a function `phase_flip_code(input_state, error)` that implements a repetition code for detecting phase-flip errors. The function should:

- Encode a single-qubit quantum state (`input_state`) into a three-qubit (repetition code) state using a stabilizer code.
- Apply phase-flip errors (Z gates) based on the qubit indices provided in the `error` list.
- Decode the three-qubit state and return the error syndrome after the decoding process.
- The function should use stabilizer code principles to detect phase-flip errors. HINT. The circuit can be constructed starting from the previous one (problem 16) and adapt the way the Z gate can be simplified.

18 Create a function `normalize_state(state_vector)` that accepts an array of complex numbers representing the coefficients of an unnormalized quantum state for multiple qubits. The function should normalize the state vector such that the sum of the squared magnitudes of all coefficients equals 1, and return the normalized state vector.

19. Write a function `quantum_or_gate(input1, input2)` that simulates a classical OR gate using a quantum circuit. The circuit should take the input state $|\text{input1},\text{input2},0\rangle$ and produce the output state $|\text{input1},\text{input2},(\text{input1 OR input2})\rangle$. The function should:

- Initialize the qubits based on the binary values of `input1` and `input2`.
- Apply the appropriate quantum gate/s to compute the OR operation.
- Combine all operations into a single gate and return the resulting quantum circuit.

  Note: Do not include measurement operations.

20. Write a function `measure_in_basis(state_vector, basis_matrix)` that measures a qubit initially in the computational basis in an arbitrary basis defined by `basis_matrix`. The matrix represents the basis of transformation. The function should:

- Verify that `basis_matrix` is unitary; if not, raise an error with the message `'matrix is not unitary'`.
- Compute and return the probabilities of measurement outcomes in the specified basis as an array.

21. Write a function `transform_to_basis(input_state, new_basis)` that transforms a quantum state vector from the computational basis to a specified new basis, which is given by a list of its eigenvectors The function should:

- Verify that the vectors in the list `new_basis` are orthonormal; if not, raise an error with the message `'new basis is not orthonormal'`.
- Transform the `input_state` into the `new_basis` using the provided eigenvectors.
- Return the transformed state vector in the new basis as an array
    - **Note**: Be aware that the eigenvectors are obtained starting from some observables, using the specific function in numpy. Thus, the order of the eigenvectors could be reversed than the normal way of writing transformation matrices (we will use common ones)

22. Write a function `create_u3_gate(theta, phi, lambda)` that generates the matrix representation of a custom U3 gate based on the input rotation angles θ, ϕ, and λ. The function should compute and return the U3 gate matrix. Use the normal way of representing qubits (not the one Qiskit does).

23. Write a function `create_cu3_gate(theta, phi, lambda, num_qubits, control_qubit, target_qubit)` that generates the matrix representation of a custom controlled-U3 (CU3) gate. Return the matrix of the gate. Use the normal way of representing qubits (not the one Qiskit does).

24. Write a Python function `multi_controlled_not(num_qubits, control_qubits, target_qubit)` that constructs and returns the matrix representation of a multi-controlled NOT gate. The function should:

- Accept `num_qubits` as the total number of qubits in the system.
- Take `control_qubits` as a list of indices representing the control qubits.
- Specify `target_qubit` as the index of the target qubit.
- Return the resulting matrix in the standard format of quantum computing (not Qiskit's representation).

25. Write a function `custom_measurement_circuit` that takes the following arguments: `input_state` (a state in the computational basis), `observable` (a Hermitian operator to measure), `seed` (to initialize a quantum simulator), and `shots` (the number of measurement runs). The function should construct a quantum circuit that measures the `input_state` in the eigenbasis of the `observable`. It should then return the expected value of the measurement based on the simulator's initialization and the specified number of shots.

26. Create a function `parity_check_circuit` that constructs a quantum circuit to calculate the parity of an integer `a`, using quantum gates. The function should measure the output to determine whether the number of 1s in the binary representation of `a` is even or odd. It should return a boolean value (`True` for even parity, `False` for odd parity) along with the Qiskit matrix representation of the circuit (excluding measurement gates).

28. Create a Python function `xor_transform_gate(x, k, num_bits)` that constructs a quantum circuit and returns a custom gate encapsulating all operations. The function should:

- Prepare the quantum state ∣xk⟩|xk\rangle∣xk⟩ by encoding the binary representations of `x` and `k` (padded with leading zeros to match `num_bits`).
- Apply a quantum operation to transform the state ∣xk⟩⟩ into ∣x(xi⊕ki)⟩⟩, where xi⊕kix_i \oplus xi⊕ki represents the XOR operation applied bitwise.
- Combine all gates into a single Qiskit matrix representation that implements and returns the transformation.

29. Create a function `verify_orthonormality(basis_vectors)` that accepts a list of basis vectors and determines whether they form an orthonormal set. The function should return `True` if the vectors are orthonormal; otherwise, return `False`.

30. Create a function `bloch_sphere_angles(state_vector)` that computes the angles required for representing a single-qubit quantum state on the Bloch sphere. The function should:

- Validate that the input state is normalized, and raise an error with the message `'state is not normalized'` if it is not.
- Return the polar angle (θ), the azimuthal angle (ϕ), and the global phase (ω) as an array.

31. Create a function `toffoli_with_basic_gates(num_qubits, control1, control2, target)` that constructs a Toffoli (CCX) gate using only fundamental quantum gates such as CNOT and single-qubit gates. The function should represent the construction in the standard, paper-based notation of quantum gates (not specific to Qiskit) and encapsulate the resulting circuit into a single gate. Return the final constructed gate.

32. Write a function `implement_mod2_oracle(num_qubits)` that constructs a quantum oracle for the function f(x)=xmod 2. The function should take the number of input qubits (`num_qubits`) as a parameter and create a quantum circuit where the input state $|x0\rangle$ is transformed into $|x\,(x \bmod 2)\rangle$, with the result stored in an ancilla qubit. The function should return the quantum circuit implementing this oracle. Construct just the circuit that would implement this functionality.

33. **Grover's Algorithm**

- **Challenge:** Implement a function **grovers_algorithm(target_state, seed)** that constructs a quantum circuit to perform Grover's search algorithm and returns the result.
- **Details:**
- **Initialize the Circuit:** Create a quantum circuit with enough qubits to represent the search space and an additional qubit for the ancilla.
- **Apply Hadamard Gates:** Place all qubits into a superposition state by applying Hadamard gates.
- **Oracle Implementation:** Create an oracle that flips the amplitude of the target state. This can be done using a multi-controlled Toffoli gate.
- **Diffusion Operator:** Apply the diffusion operator to amplify the probability of the target state.
- **Iteration:** Repeat the application of the oracle and diffusion operator $\sqrt{N}$ times (where N is the number of states).
- **Measurement:** Measure the qubits to find the target state.

34. **Quantum Teleportation**

- **Challenge:** Implement a function **quantum_teleportation(input_state, seed)** that constructs a quantum circuit to perform quantum teleportation and returns the result.
- **Details:**
- **Initialize the Circuit:** Create a quantum circuit with three qubits.
- **Prepare Bell State:** Apply Hadamard and CNOT gates to the first two qubits to create an entangled Bell state.
- **Entangle and Measure:** Entangle the input state with the first qubit and measure the first two qubits.
- **Apply Corrections:** Apply appropriate X and Z gates to the third qubit based on the measurement results to complete the teleportation.

35. **Quantum Machine Learning Challenge**

- **Challenge**: Implement a function **`qnn(dataset, seed)`** that constructs a quantum circuit to train a simple quantum neural network (QNN) on the provided training data and returns the result and accuracy of the trained model.
- **Details:**
- **Initialize the Quantum Circuit**: Create a quantum circuit with enough qubits to encode the features of the data points.
- **Define Ansatz**: Choose a simple parameterized quantum circuit as the ansatz for the QNN.
- **Encode Training Data**: Use parameterized gates to encode the training data into the quantum circuit.
- **Train QNN**: Use a classical optimizer to train the QNN by minimizing the loss function.
- **Evaluate Model**: Test the trained QNN on a validation dataset and return the result and accuracy.

36. **Circuit Width Optimization with Classiq**
   - **Challenge:** Implement a function **optimize_circuit(`quantum_model`)** that synthesizes two quantum circuits based on the **`quantum_model`**, one without constraints and the other one optimized such that it has minimum width. You must use the Classiq SDK and the function should return the width ratio between the two circuits: width_optimized_circuit/ width_simple_circuit.

37. **Circuit Depth Optimization with Classiq**
   - **Challenge:** Implement a function **optimize_circuit(`quantum_model`)** that synthesizes two quantum circuits based on the **`quantum_model`**, one without constraints and the other one optimized such that it has minimum depth. You must use the Classiq SDK and the function should return the depth ratio between the two circuits: depth_optimized_circuit/ depth_simple_circuit.

38. **Implementing a polynomial function with Classiq**
   - **Challenge:** Implement a function **polynomial(num_qbits_x, a)** that creates a circuit to solve the equation x^2 + a. The input state **x** will be an equal superposition of **num_qbits_x** qubits and the function should return the results of the execution job.

39. **Quantum conditional statements with Classiq**
   - **Challenge:** Implement a function **q_if(initial_state, test_value)** that creates a circuit to check if the initial state is equal to the **test_value** without collapsing the state. The function should return the results of the measurement which will highlight with a target qubit the value 1 if the initial state matches the test_value, or 0 otherwise.