Final Report

Quantum Computing

Group Quantum Innovators

Name	Roll Number
Jaidev Sanjay Khalane	22110103
John Twipraham Debbarma	22110108
Pranav Joshi	22110197
Vannsh Jani	22110279

▼ Objectives

Initial Aim

"*The aim of this project is to gain an understanding of Quantum Computers as well as explain and contrast their working with respect to classical computers in terms of ISA, Hardware-Software Interface and Microarchitecture. Apart from this, we also aim to develop a simulator application that will enable the user to write a Quantum ISA language code, followed by the simulation of the functioning of the Quantum Computing processes on the classical computer. We intend to develop our own Quantum ISA language for modelling the processes in a QPU of up to 5-8 Qubits. We also aim to explore the current issues that present an obstacle to the efficient usage of quantum computing at a larger scale and come up with ideas to mitigate those issues.*
" as per the Interim Report

Achievements

We have achieved all the initial aims. We have also exceeded our aims by performing more detailed analysis on some more topics such as benchmarking, comparing the basic operations and algorithms on Classical and Quantum Computers, comparing multiple types of ISAs, etc. A brief flow of the work done by us is provided in the next section:

List of Objectives Achieved:

- Introduction: In this section, we start with the description of our understanding of Quantum Computers, Quantum Information, and Quantum Abstract Machines and follow it with a comparison between Bits and Qubits.
- Basic Operations and Gates: Understanding and Comparison: In this section, we will discuss about the basic operations and gates in the Quantum and Classical frameworks and compare them.
- Auxiliary Quantum Circuits to Perform Classical Operations using Quantum Gates: Following the comparison between the Classical and Quantum Gates, we also explored the auxiliary circuits that are required by the quantum computers in order to perform classical operations along with higher-level circuits such as Adder, Subtractor, Multiplexer, etc.
- ISA Comparison: We performed a detailed analysis of some of the Quantum ISAs, namely QSL (Developed by us) and QUASAR ISA for Quantum Computing. We also performed a detailed comparison of these Quantum ISAs wit MIPS ISA in terms of Operation Repertoire, Data Types, Registers, Instruction Format and Addressing Modes.
- Hardware Software Interface of Quantum Computers: We then started the discussion on the Hardware-Software interface of Quantum Computers with the main necessities for Quantum Hardware. We also discussed two views of hardware-software interfaces in Quantum Computers. Apart from this, we also explored the differences between the classical and the quantum Hardware Software interfaces.
- Physical Implementation of Quantum Computers: In this part, we discussed another aspect of Quantum Computers: Their physical implementation. We explored different methods of physically implementing the Qubits along with the ways of implementing single and multi-qubit operations governing their topologies. We also compared this with physical implementation of classical bits and computers.
- Quantum Benchmarking: We also studied various methods of performing benchmarking in quantum computers. Apart from this, we also performed a comparison of these benchmarks with the classical benchmarks as well.

- Comparison of Operations on which Quantum is better, and the Classical is better: We also studied the comparison of the performance of various operations on Quantum Computers and on Classical Computers and discussed which one is better.
- Better Algorithms in Quantum Computers: We also performed a detailed analysis on the quantum algorithms such as Shor's Algorithm, Deustch's Algorithm and Quantum Random Number Generators.
- Current Difficulties in Quantum Computing- Errors and Correction: We also studied some of the prominent issues that are preventing the use of Quantum Computing on a large scale, namely Errors. We also performed a detailed study on some of the methods used in error correction to mitigate these issues.
- Simulation: As mentioned in our initial aims, we developed our own ISA for Quantum Computers (called QSL: Quantum Simulation Language). The detailed documentation of the ISA is also provided at the end of this report. We also developed a Simulator which executes the programs written in this ISA as well as provides a step by step execution for the user to gain an understanding of the evolution of the states of the qubits along with the memory and registers. The details of this are provided in the demonstration video. We also created benchmark programs in our QSL based on OpenQASM benchmarks for the users to execute them, as well as a Benchmark feature in the simulator that would enable the user to benchmark the hardware on which simulator is being run in terms of the CLOPS rating as well as the SPEC type rating that we developed with respect to the IBM Eagle R3 processor (accessed through IBM Kyiv server).
- There are some more topics that we explored (which are beyond the scope of this project) such as Quantum Machine Learning, etc.

Introduction

What are Quantum Computers

A quantum computer is a type of computer that harnesses quantum mechanical phenomena, leveraging the unique behaviors of matter at small scales, where particles exhibit both wave and particle properties. Unlike classical computers, which are rooted in classical physics, quantum computers operate using specialized hardware that exploits these quantum behaviors to perform certain types of calculations exponentially faster. This potential speed advantage has led to great interest in the field, as a scalable quantum computer could theoretically break existing encryption schemes and significantly enhance scientific simulations. However, quantum computing technology is still largely experimental, with many challenges yet to be overcome before it can be widely applied.

The foundational unit of information in quantum computing is the qubit, or "quantum bit," which differs from the classical bit used in traditional computing. While a classical bit exists in a binary state (either 0 or 1), a qubit can exist in a superposition of states, meaning it can represent both 0 and 1 simultaneously. This property allows quantum computers to perform complex calculations with a high degree of parallelism. Despite these theoretical advantages, constructing high-quality qubits has proven difficult, as qubits are highly sensitive to their environment and prone to "decoherence," which introduces noise into calculations.

Quantum Information

In quantum mechanics, every state of a system is characterised by its wave function $\Psi(\vec{x},t)$, or when viewed as an element of a function space, a wave vector $|\Psi\rangle$. This state evolves with time according to the Schrodinger equation $i\hbar\frac{\partial}{\partial t}|\Psi\rangle = \hat{H}|\Psi\rangle$ where $\hat{H}$ is the hamiltonian operator for the system. For a single partcle system, this is given by $\hat{H} = \frac{\hat{p}^2}{2m} + V$ where $\hat{p} = -i\hbar\frac{\partial}{\partial x}$ is the momentum operator and $V(x)$ is the potential. [27]

For any observable O , one can have a hermitian operator $\hat{O}$ representing it. One of the main postulates of Quantum Mechanics state that if $\hat{O}$ has eigenvalues λ_i and (orthonormal) eigenvectors $|i\rangle$, then upon measuring the value of the observable for a system in state $|\Psi\rangle$, one can only get the values λ_i , with probabilities $|\langle i|\Psi\rangle|^2$ where $\langle\cdot|\cdot\rangle$ is the inner dot product defined over the function space. More generally, if $|\Psi\rangle = \sum_i a_i|i\rangle$, then $P_O(\lambda_i) = |a_i|^2$. [3]

Thus, if two observables O_1, O_2 have the same eigenvectors (or an orthogonalised version) for their operators, then each eigenvector $|i\rangle$, also known as a pure state, corresponds to a pair of eigenvalues (α_i, β_i), one for each observable. The probability of obtaining this pair of values as the value of the observables i.e. $P_{(O_1,O_2)}((\alpha_i,\beta_i))$ is given by $|\langle i|\Psi\rangle|^2$. And again, only the eigenvales will be observed. Such a set of observables are called "compatible" . They have the nice property that their commutator $[\hat{O}_1, \hat{O}_2] := \hat{O}_1\hat{O}_2 - \hat{O}_2\hat{O}_1$ is 0 . Similarly, once which don't have a 0 valued commutator are called incompatible. We can never be able to know for sure, both the values for an incompatible pair simultaneously until measurement.

One important set of observables are the x,y,z spin angular momentum of a single electron system, along with the magnitude, i.e S_x, S_y, S_z and $S = \sqrt{S_x^2 + S_y^2 + S_z^2}$. Instead of using S , the observable that we usually use is S^2 . The operators are related by what are known as commutator relations :

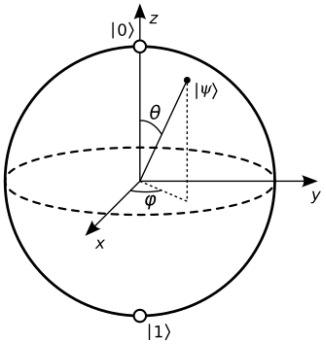
$$[S_x, S_y] = i\hbar S_z$$
$$[S_y, S_z] = i\hbar S_x$$
$$[S_z, S_x] = i\hbar S_y$$
$$[S_x, S] = [S_y, S] = [S_x, S] = 0$$

One can solve this system and get the effect of different operators on the eigenvectors $|0\rangle$, $|1\rangle$ of S_z , with eigenvalues $1, -1$ respectively. Doing so, one gets the well known Pauli matrices for S_x, S_y, S_z, S^2 respectively :

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$
$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$
$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Now, we can represent any state $|\Psi\rangle$, represented as a *spinor* into the eigenbasis of any of these matrices to get the expected value of the corresponding observable. [27]

This mapping from spinnors to a set of expected values is bijective. One can also notice that since $|\Psi\rangle$ gives rise to a probability distribution, thus for any spinor $\begin{bmatrix} a & b \end{bmatrix}^T$ we have $|a|^2 + |b|^2 = 1$, which means that the expected value of S^2 will always be 1 , since it is $|a|^2 \times \lambda_0 + |b^2| \times \lambda_1 = |a|^2 + |b|^2 = 1$. Moreover, from some calculation, one can see that the vector formed by expected values of S_x, S_y, S_z , let's call it $E[\vec{S}]$, has magnitude 1 too, that is, it's on a sphere. This bijective relationship between spinors and points on a sphere gives rise to a concept known as the Bloch sphere, where every state $|\Psi\rangle$ is represented as a point on the Bloch sphere.



Bloch-Sphere

The $|0\rangle$ and $|1\rangle$ states are named so, because these states of a qubit (an electron, or some other particle) are analogous to the 0 (LOW) and 1 (HIGH) states of a classical bit. Unlike classical bits though, there are infinite number of states that a qubit can take, each represented by a spinor. This is what people mean when they say that a qubit can be in both 0 and 1 state at the same time. But this is coming from just one interpretation of quantum mechanics, out of many; the most popular among physicists being "we don't really know, and we don't really care as long as the mathematics works out" , also known as the agnostic position [27].

For multiple qubits, we just take a direct product of the spaces spanned by the spinors to get the space that the full state vector lies in. This is explained in the function of the Quantum Abstract Machine (QAM)

▼ Quantum Abstract Machine

This is an abstract model of a quantum computer, similar to Turing machine model for normal computers.

The QAM is specified by

- N_q qubits, indexed from 0 to $N_q - 1$. The combined state of all qubits $|\Psi\rangle$ is represented as vector in $\Delta(\{0, 1 \dots 2^{N_q} - 1\})$ where $\Delta(S)$ is the simplex of set S .

  The (quantum) state (of the qubits) is initialised to the pure state $|000\dots0\rangle$.

  The k^{th} qubit is refered to as Q_k
- A classical memory C of N_c bits, indexed from 0 to $N_c - 1$, as $C[k]$ with all bits $C[k]$ initialised to 0 for $0 \leq k < N_c$.
- A sequence P of instructions, indexed from 0 to $|P| - 1$.
- A classical integer valued register which is the program counter (PC) . The value is referred to as κ .
  When
  $\kappa = |P|$, the computer has halted.

- A list G of static gates. These are operations that the quantum computer can do on $|\Psi\rangle$, one at a time. Once can represent each static gate as a fixed $2^{N_q} \times 2^{N_q}$ matrix.
- A list G' of parametric gates. These result in again, operations that the quantum computer can do on $|\Psi\rangle$, except that we can also provide different (classical) values of "parameters" from the memory C, or in the program P itself which change the gate's behaviour on the qubits accordingy.

  One can represent parametric gates as functions $M : \mathbb{C}^n \to U(2^{N_q})$ which create matrices $M(\theta) \in \mathbb{C}^{2^{N_q} \times 2^{N_q}}$ which change as the parameters θ change. These matrices $M(\theta)$ act on $|\Psi\rangle$.

Thus, a QAM is given as the 6-tuple $(|\Psi\rangle, C, G, G', P, \kappa)$. Note the similarity with the definition of Turing machines. A QAM, just as a TM specifies a programmable computer programmed for a particular program. [25]

▼ Qubits vs Bits

In classical computing, information is represented by bits, which take on a binary state of either 0 or 1. Each bit encodes a distinct and definite state, enabling calculations to proceed through deterministic sequences of logical operations. This clear, binary system has long been the basis for traditional computational processes, where each bit's value is concrete and can be manipulated in predictable ways through binary logic gates. However, this deterministic approach means classical bits operate on single states at a time, processing one configuration per computational step.

In contrast, quantum computing introduces qubits, which fundamentally differ from classical bits in their ability to exist in a superposition state. Superposition allows a qubit to represent both 0 and 1 simultaneously rather than being confined to one state. This property creates a probabilistic rather than deterministic model, where the outcome of a computation is determined by a probability distribution of the qubit states. Furthermore, qubits can be entangled, linking their states in such a way that the state of one qubit directly influences the state of another, no matter the distance between them. This entanglement, combined with superposition, allows quantum computers to process multiple configurations concurrently, introducing a degree of parallelism unmatched by classical systems. This parallelism enables quantum computers to solve complex, multivariate problems with far fewer steps, presenting transformative potential for fields like cryptography, optimization, and material science.

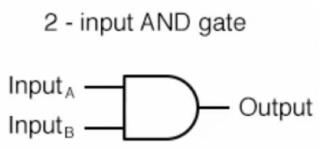
▼ Basic Operations and Gates: Understanding and Comparison

In classical computing, the basic operations are binary logic operations such as AND, OR, NOT, XOR, etc., performed using logic gates on bits. Quantum computing has a set of operations that manipulate qubits based on quantum gates. Quantum gates are reversible and operate differently from classical gates, leveraging properties like superposition and entanglement.

Classical Computing

- **Data Representation and Storage**: Classical computing uses bits as the smallest unit of data, which can be either 0 or 1. Bits store and represent information in a binary format that is immutable without external manipulation. In classical computing, a register with n bits represents exactly 2^n possible states but only one at a time.
- **Classical Logic Gates**: These are the basic operations that are applied on bits such as AND, NOT, OR, XOR, NOR, etc. Classical logic gates are irreversible (except for the NOT gate), meaning the mapping between the inputs and the outputs is not bijective in nature.

  Example of AND Gate:



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Circuit Diagram and Truth Table of AND Gate.

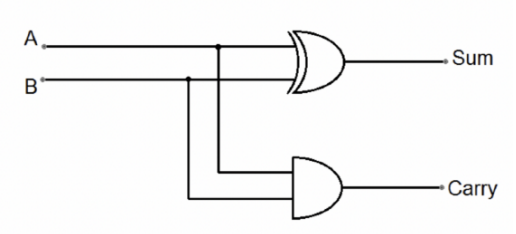
We can clearly see that this mapping is not bijective in nature as if we measure the output bit as 0, we cannot say for certain what the input bits were. Similarly, we can say the same for the other logical operators except the NOT gate, which is reversible in nature.

Here are some of the functions for logical operators:

- **NOT Gate**: Inverts the bit value (0 becomes 1, and 1 becomes 0).
- **AND Gate**: Outputs 1 only if both inputs are 1.
- **OR Gate**: Outputs 1 if at least one input is 1.
- **XOR Gate**: Outputs 1 if the inputs are different.

These gates manipulate individual bits or pairs of bits to execute deterministic logic operations. In classical circuits, combinations of these gates can form more complex logic circuits like adders, multiplexers, and memory units.

- **Arithmetic Operators**: Arithmetic operations are fundamental in any computation, as they allow for processing numerical data.
  - **Addition**:
    - **Half-Adder**: A half-adder is a basic circuit used to add two single bits (A and B). It produces two outputs:
      - Sum: Calculated with an XOR gate ($A \oplus B$), producing 1 if A and B are different.
      - Carry: Calculated with an AND gate ($A \wedge B$), producing 1 only if both inputs are 1.



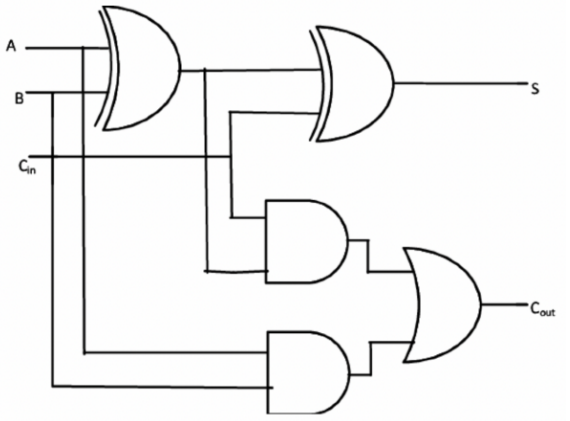
Logic Diagram of Half Adder.

    - **Full-Adder**: A full-adder extends the half-adder by handling a carry input. It uses multiple logic gates (AND, OR, XOR) to manage both the sum and carry from three input bits (two operand bits plus one carry bit). Full-adders are chained together in series to create adders capable of adding multi-bit numbers.$S=(A \oplus B) \oplus Cin Cout=(A \cdot B)+(B \cdot Cin)+(A \cdot Cin)$

      The sum and carry out are given by:

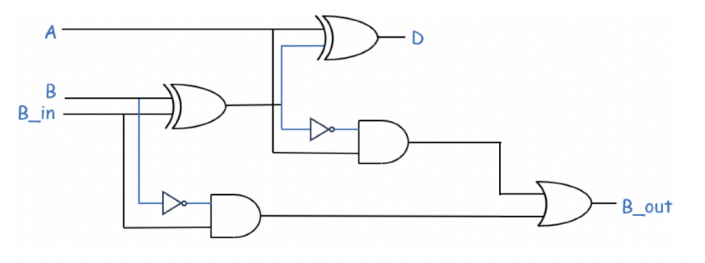
      $S=(A \oplus B) \oplus Cin$

      $Cout=(A \cdot B)+(B \cdot Cin)+(A \cdot Cin)$



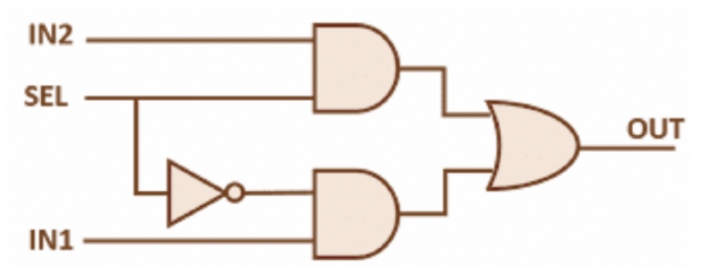
Logic Diagram of Full Adder.

  - **Full Subtractor**: A full subtractor calculates the subtraction of two bits, taking into account a "borrow" from a previous, less significant bit. It has three inputs: the two bits to be subtracted (let's call them `A` and `B`) and a borrow-in (`Bin`). The circuit produces two outputs: the difference and the borrow-out. The difference output shows the result of `A - B - Bin`, while the borrow-out indicates if a "borrow" is needed for the next stage.



Logic circuit of a full subtractor.

- **Control Operations**:
  - **2:1 Multiplexer**: A 2:1 multiplexer (MUX) acts like a decision-maker in a circuit. It has two inputs In1 and In2 and a "sel" switch. Depending on the position of this switch, the MUX chooses which input to pass through to the output.



Logic circuit of 2:1 Multiplexer

Quantum Computing

- **Data Representation and Storage**: Quantum computing uses qubits, which can be in a state of $|0\rangle$, $|1\rangle$, or any superposition of these states, represented as $\alpha|0\rangle+\beta|1\rangle$, where α and β are complex probability amplitudes. Qubits can represent a combination of both states simultaneously, allowing for more data density and computational flexibility. In contrast to classical computing, n qubits can represent 2^n states simultaneously through superposition, vastly increasing the information processing capability.
- **Quantum Logic Gates**: Quantum gates operate on qubits and use transformations that maintain the qubits' quantum properties, like superposition and entanglement.
  - **Single Qubit Gates**: These are the quantum gates that operate on one qubit. Some common single qubit gates are given below.
    - **I (Identity Gate)**: The Identity gate does not affect the state of any qubit. Hence, we can say that $I|0\rangle=|0\rangle$ and $I|1\rangle=|1\rangle$.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

    - **Pauli-X Gate**: The Pauli-X gate flips the state of a qubit, meaning it maps $|0\rangle$ to $|1\rangle$ and vice-versa. Hence, this gate is analogous to the NOT gate in classical computing. Hence, we can say that $X|0\rangle=|1\rangle$ and $X|1\rangle=|0\rangle$.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

    - **Pauli-Y Gate**: The Pauli-Y gate introduces a bit-flip and phase-flip operation to a qubit. Hence, we can say that $Y|0\rangle=i|1\rangle$ and $Y|1\rangle=-i|0\rangle$. Along with the bit-flip, the phase of the state also flips.

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

- **Pauli-Z Gate**: The Pauli-Z gate, also known as the phase-flip gate, affects only the phase of the qubit's state. The Pauli-Z gate does not change the $|0\rangle$ state but flips the sign of the $|1\rangle$ state, effectively introducing a phase shift. Hence, we can say that $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$.

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- **Hadamard Gate (H)**: The Hadamard gate transforms the basis states $|0\rangle$ and $|1\rangle$ into equal superpositions (equal probability of being in $|0\rangle$ and $|1\rangle$) of each other. Specifically:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

For a general qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the Hadamard gate produces:

$$H|\psi\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} \alpha + \beta \\ \alpha - \beta \end{pmatrix}$$

- **S Gate**: The **S gate** (also called the π/2 phase gate) introduces a relative (local) phase shift of π/2 (or 90 degrees) to the qubit. It is sometimes considered a "square root" of the Z gate because applying it twice results in a Z gate operation. Hence, we can say that $S|0\rangle = |0\rangle$ and $S|1\rangle = i|1\rangle$. For a general qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the S gate transforms it as follows:

$$S|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ i\beta \end{pmatrix}$$

- **T Gate**: The **T gate** (also called the π/4 phase gate) introduces a relative (local) phase shift of π/4 (or 45 degrees) to the qubit. It is often used in quantum circuits as a "fourth root" of the Z gate, meaning that applying it four times results in an S gate and eight times results in the identity operation. Hence, we can say that $T|0\rangle = |0\rangle$ and $T|1\rangle = e^{i\pi/4}|1\rangle$. For a general qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the T gate transforms it by applying a phase shift to the $|1\rangle$ component.

$$T|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ e^{i\pi/4}\beta \end{pmatrix}$$

- **Rz Gate (Rotation around Z-axis)**: The Rz gate applies a rotation around the z-axis by an angle θ to a single qubit. It is a phase shift gate. When this gate is applied to a general qubit state $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$, the Rz gate introduces no phase shift for the $|0\rangle$ component but introduces a phase shift of e^iθ for the $|1\rangle$ component. This allows for precise control of the qubit's phase.

$$R_z(\theta)\,|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ e^{i\theta}\beta \end{pmatrix}$$

- **Reset Gate**: The reset gate is a single-qubit gate that resets the state of a qubit to the basis state $|0\rangle$, regardless of the qubit's prior state. It is a non-unitary operation and is often used to clear qubits for reuse in a quantum circuit. After applying the reset gate to a qubit state, the qubit will always be set to $|0\rangle$, independent of its initial state.
- **Measure Operation**: The measure operation is a single-qubit operation that reads the state of a qubit, causing it to collapse into one of the classical basis states, $|0\rangle$ or $|1\rangle$. This measurement is a probabilistic and irreversible process that converts quantum information into a classical value. When the measurement is performed in the standard computational basis, the probabilities of the qubit collapsing to each basis state are determined by the magnitudes of the complex amplitudes of the qubit's state. After measurement:
  - If the outcome is $|0\rangle$, the qubit's state becomes $|0\rangle$.
  - If the outcome is $|1\rangle$, the qubit's state becomes $|1\rangle$.

  Thus, the measure operation **collapses** the superposition into a definite state based on the probability distribution of the amplitudes.

$$P(|0\rangle) = |\alpha|^2 \quad \text{and} \quad P(|1\rangle) = |\beta|^2$$

The Identity gate and the Pauli X, Y, and Z gates, as well as the Hadamard gate mentioned above, are both **Unitary** and **Hermitian** in nature (the matrix is equal to its conjugate transpose), making these gates **Involutory**. This means that the square of the matrices is the identity matrix.

- **Double Qubit Gates**: These are quantum gates that operate on two qubits. Some common double qubit gates are as follows:
  - **CNOT (Controlled-NOT) Gate**: The CNOT gate flips the second (target) qubit if the first (control) qubit is $|1\rangle$. This gate is fundamental for creating entanglement in quantum circuits. The CNOT gate has the following behavior:
    - CNOT $|00\rangle = |00\rangle$
    - CNOT $|01\rangle = |01\rangle$
    - CNOT $|10\rangle = |11\rangle$
    - CNOT $|11\rangle = |10\rangle$

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

    This means that the second qubit flips when the first qubit is in state $|1\rangle$. This gate is also known as the CX Gate.
  - **SWAP Gate**: The SWAP gate exchanges the states of two qubits. It can be implemented using three CNOT gates in the following sequence:
    1. Apply a CNOT gate with qubit A as the control and qubit B as the target.

2. Apply a CNOT gate with qubit B as the control and qubit A as the target.
3. Apply a CNOT gate with qubit A as the control and qubit B as the target.

This sequence swaps the states of the qubits. The SWAP gate operates on the basis states as follows:

- SWAP $|00\rangle = |00\rangle$
- SWAP $|01\rangle = |10\rangle$
- SWAP $|10\rangle = |01\rangle$
- SWAP $|11\rangle = |11\rangle$

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

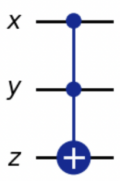
- **CZ (Controlled-Z) Gate**: The CZ gate is a controlled phase-flip gate. It flips the phase of the second (target) qubit if the first (control) qubit is in state $|1\rangle$. The action of the CZ gate on basis states is:
  - CZ $|00\rangle = |00\rangle$
  - CZ $|01\rangle = |01\rangle$
  - CZ $|10\rangle = |10\rangle$
  - CZ $|11\rangle = -|11\rangle$

$$\text{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

- **CP (Controlled-Phase) Gate**: The CP gate applies a phase shift of θ to the target qubit if the control qubit is $|1\rangle$. For $\theta = \pi$, the CP gate becomes the CZ gate.

$$\text{CP}(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}$$

- **Some Other Quantum Gates**:
  - **Toffoli Gate (CCNOT Gate)**: The Toffoli gate, or CCNOT gate, is a three-qubit gate that flips the target qubit $|z\rangle$ if both control qubits $|x\rangle$ and $|y\rangle$ are in state $|1\rangle$. This gate is commonly used in quantum circuits to implement classical AND operations.



▼ Auxiliary Quantum Circuits to perform Classical Operations using Quantum Gates

In quantum computing, auxiliary circuits are supplemental or supporting circuits that help carry out particular tasks required to run or manage a quantum system. By bridging the gap between classical and quantum functions, these circuits aid in activities including directing operations, managing qubits, stabilizing quantum states, and facilitating measurements.

Quantum System: A computer system that uses the concepts of quantum mechanics, mainly qubits (quantum bits), which are capable of existing in several states at once (superposition) and having entangled states. A pure Quantum System is hard to implement, so we will try to relate and compare with the classical systems to gain an understanding of the core concepts and try to simulate the operations.

Classical System: A traditional computing system that uses binary bits (0 and 1) to perform computations. We are already learning it from our course contents, so I have referred whatever taught in the class and used the concepts below to make the required actions (like making circuits, comparing with quantum and implementing classical operations using quantum gates).

Hybrid Quantum-Classical Systems: Systems that integrate both quantum and classical components, where quantum circuits perform computations and classical components handle auxiliary functions or manage quantum operations. Shor's algorithm which I have discussed in the subsequent headings, uses a hybrid quantum-classical system for it to implement.

Purpose of Auxiliary Circuits in Quantum-Classical Systems

The purpose of auxiliary circuits is to **facilitate communication, control, and synchronization** between classical and quantum systems. Since classical and quantum systems process data differently, auxiliary circuits ensure they can operate cohesively. Here are some purposes:

- **Data Conversion**: Translating quantum states into classical data (measurement).
- **Control Signals**: Sending pulses and timing signals to quantum gates and qubits.
- **Error Correction**: Detecting and correcting errors in quantum computations.

Example: A scenario where a classical controller is sending timing signals to a quantum processor to execute a specific operation at a precise moment. Auxiliary circuits act as an intermediary, ensuring that the controller's instructions translate correctly into quantum operations. [15][16]

Breaking Down the Role of Auxiliary Circuits

Their main roles and functions:

- **Initialization**: Auxiliary circuits help initialize qubits into the desired quantum state.
- **Measurement**: They convert quantum data back into classical information, as quantum results must be observed in classical form.
- **Stabilization and Error Correction**: They assist in stabilizing qubit states, managing decoherence, and applying error correction codes.
- **Gate Operations**: Auxiliary circuits aid in sending signals to quantum gates, ensuring qubits undergo correct transformations.

Creating auxiliary circuits for quantum computing involves designing and implementing both the **hardware components** and the **control software** that enables communication and management between quantum and classical systems.

1. **Understanding Auxiliary Circuit Design: Hardware Perspective**

- In actual hardware, building auxiliary circuits is complex and generally requires knowledge of **quantum hardware** (such as superconducting qubits, ion traps) and **classical electronics** (timing controls, signal processors).
- There are various types of quantum circuits, few of which are **pulse control circuits** (to send signals to qubits), **timing circuits** (for synchronization), and **measurement circuits** (to convert quantum states into classical results).

Note: Since physical implementation is advanced and requires specialized equipment, we'll primarily focus, in this project, on understanding the functions of these components rather than physically building them.

2. **Simulating Auxiliary Circuit Functions: Coding and Software Perspective**

For our project, simulating the behavior of auxiliary circuits using **quantum programming frameworks** will be more practical than building actual physical circuits. Quantum programming libraries, like **Qiskit** (IBM's library for quantum computing in Python), allow us to simulate auxiliary circuits and understand their interaction with qubits.

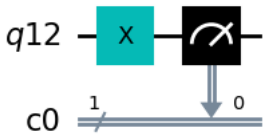
The following libraries and dependencies are used: [18]

```
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import IBMQ, Aer, execute
from qiskit.visualization import plot_bloch_multivector
```

NOT Gate

NOT Gate can be implemented using an X quantum gate. [17]

The circuit diagram for the algorithm may be shown as follows:



The truth table for NOT Gate is given below:

Input	Output
A	Y
0	1
1	0

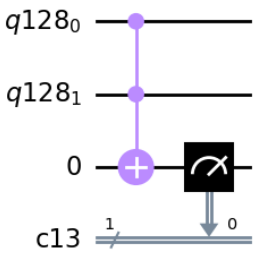
The code below demonstrates the same:

```
# Creating a Quantum Circuit with 1 quantum register and 1 classical register:
q = QuantumRegister(1)  # Quantum Register
c = ClassicalRegister(1)    # Classical Register
qc = QuantumCircuit(q,c)    # Quantum Circuit with quantum and classical registers
qc.x(q[0])  # Applying X-gate
qc.measure(q[0], c[0])    # Mapping the quantum measurement to the classical bits
qc.draw(output='mpl')
```

AND Gate

To implement an AND gate using quantum gates, we can use a Toffoli gate (CCX gate), which is a controlled-controlled-X gate. This gate flips the target qubit if both control qubits are in the state $|1\rangle$. In this configuration, if both q0=1 and q1=1, the target qubit becomes $|1\rangle$; otherwise, it remains $|0\rangle$. (Note that here, the number 128 is just the count of iterations the code is run.) The final output is stored in c[0].

The circuit diagram for the algorithm may be shown as follows:



The truth table for AND Gate is given below:

A	B	Output
0	0	0
1	0	0
0	1	0
1	1	1

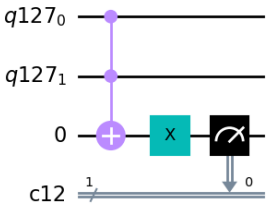
The code below demonstrates the same:

```
q = QuantumRegister(2)
zero = QuantumRegister(1, "0")
c = ClassicalRegister(1)
qc = QuantumCircuit(q,zero,c)
qc.ccx(q[0], q[1], zero)
qc.measure(zero, c[0])
qc.draw(output='mpl')
```

NAND Gate

To implement a NAND gate using quantum gates, we can use a Toffoli (CCX) gate followed by an X gate. The Toffoli gate computes the AND operation on two control qubits and a target qubit, and the X gate inverts the result to achieve the NAND operation.

The circuit for the algorithm may be shown as follows:



The truth table for the NAND Gate is given below:

A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

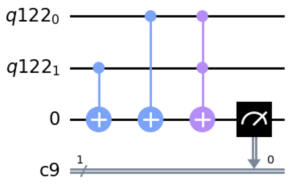
The code below demonstrates the same:

```
q = QuantumRegister(2)
zero = QuantumRegister(1, "0")
c = ClassicalRegister(1)
qc = QuantumCircuit(q,zero,c)
qc.ccx(q[0], q[1], zero)
qc.x(zero)
qc.measure(zero, c[0])
qc.draw(output='mpl')
```

OR Gate

Similarly, the OR Gate can be implemented as below:

The circuit for the algorithm may be shown as below:



The truth table for the OR Gate is:

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	1

The code below demonstrates the same:

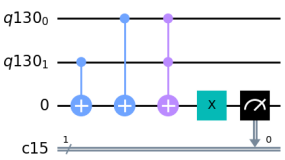
```
q = QuantumRegister(2)
zero = QuantumRegister(1,"0")
c = ClassicalRegister(1)
qc = QuantumCircuit(q,zero,c)

qc.cx(q[1], zero)
qc.cx(q[0], zero)
qc.ccx(q[0], q[1], zero)
qc.measure(zero, c[0])
qc.draw(output='mpl')
```

NOR Gate

The NOR gate outputs `1` only when both inputs are `0`. This can be represented using NOT and OR logic. The final output is stored in c[0].

The circuit for the algorithm may be shown as below:



The truth table for NOT Gate is:

A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0

The code is provided below:

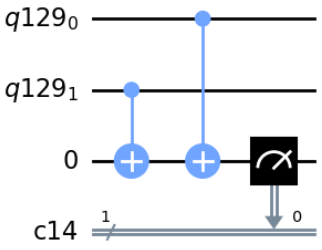
```
q = QuantumRegister(2)
zero = QuantumRegister(1, "0")
c = ClassicalRegister(1)
qc = QuantumCircuit(q,zero,c)

qc.cx(q[1], zero)
qc.cx(q[0], zero)
qc.ccx(q[0], q[1], zero)
qc.x(zero)
qc.measure(zero, c[0])
qc.draw(output='mpl')
```

XOR Gate

Similarly, the XOR Gate can be implemented as below. The final output A⊕B is stored in D[0].

The circuit for the algorithm may be shown as below:



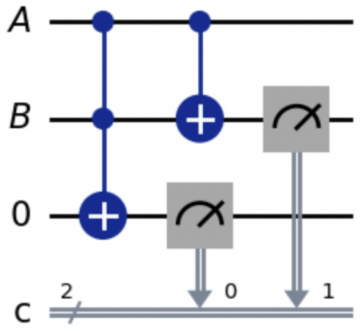
The truth table for XOR Gate is:

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

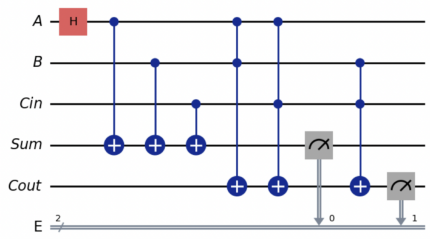
The code is provided below:

```
q = QuantumRegister(2)
zero = QuantumRegister(1, "0")
c = ClassicalRegister(1)
qc = QuantumCircuit(q,zero,c)
qc.cx(q[1], zero)
qc.cx(q[0], zero)
qc.measure(zero, c[0])
qc.draw(output='mpl')
```

- **Arithmetic Operations**: Arithmetic operations can be implemented using quantum gates.
  - **Half Adder**: The Half Adder takes two input bits, AA and BB, and produces two outputs:
    1. **Sum (S)**: The XOR of A and B.
    2. **Carry (C)**: The AND of A and B.The quantum circuit for the Half Adder is constructed as follows: A CNOT gate computes the sum $S=A \oplus B$, and a Toffoli gate computes the carry $C=A \wedge B$. The carry output is represented by c[0] and the sum is represented by c[1].



  - **Full Adder**: The Full Adder takes three input bits, A, B, and a Carry-in (Cin), and produces two outputs:
    1. **Sum (S)**: The XOR of A, B, and Cin.
    2. **Carry-out** (Cout): This is true if at least two of the inputs are 1.The quantum circuit for the Full Adder uses two CNOT gates and one Toffoli gate to compute the sum $S=A \oplus B \oplus Cin$. Additional Toffoli gates are used to compute the Carry-out (Cout), which represents the carry resulting from adding the three bits. The sum is represented by E[0] and the carry-out by E[1].



- **Full Subtractor**: To calculate the difference, a series of CNOT gates perform XOR operations between the qubits representing AA (q0), BB (q1), and the borrow-in BinBin (q2). The difference is calculated as A⊕B⊕Bin.

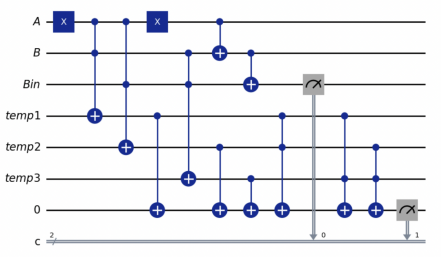
  The process is as follows:

  1. **CNOT(A, B)**: Updates B to A⊕B.
  2. **CNOT(B, Bin)**: Updates Bin to A⊕B⊕Bin. (B already has the updated value A⊕B).

  To calculate **Borrow-out**, Toffoli (CCX) gates and CNOT gates are used to implement the AND and OR operations:

  1. **NOT A AND B**: Apply an X gate to invert A, then use a Toffoli gate with NOT A,B, and a temporary qubit to set the borrow-out if A=0 and B=1.
  2. **NOT A AND Bin**: Use a Toffoli gate with NOT A,Bin to set the borrow-out if A=0 and Bin=1.
  3. **B AND Bin**: Use a Toffoli gate with B,Bin to set the borrow-out if both B=1 and Bin=1.
  4. Finally, reset A using a Pauli-X gate.

  The difference is represented by c[0] and the borrow-out by c[1].

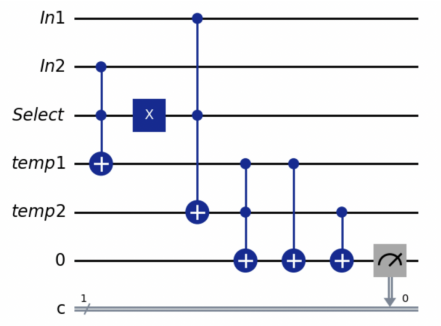


- **Control Operations**
  - **2:1 Multiplexer**: A 2:1 multiplexer (MUX) is a decision-making device in a circuit. It has two inputs (In1 and In2) and one "select" switch. Based on the position of the switch, the MUX decides which input to send to the output.

    The quantum circuit for the 2:1 multiplexer is implemented using CCNOT gates, Pauli-X gates, and CNOT gates. The process is as follows:

    1. **temp1**: This represents temp1=In2∧select.
    2. **temp2**: This represents temp2=In1∧(~select), where the negation is achieved using the Pauli-X gate.
    3. Finally, the output is computed as OUT=temp1∨temp2.

    The final output is represented by c[0].



▼ ISA Comparison

An Instruction Set Architecture (ISA) serves as an abstract model of a computer, providing a programmer's perspective on its operation and capabilities. It acts as a "contract" or a clearly defined interface between hardware and software, specifying the fundamental capabilities that the hardware must provide to ensure that all software operations can be expressed and executed reliably within this framework.

Different ISAs can drastically influence the design, performance, and efficiency of the hardware, especially when comparing classical architectures like MIPS to more novel models such as quantum ISAs.

In comparing ISAs, we focus on three primary aspects:

1. **Operation Repertoire**: Examines the types and complexity of operations available.
2. **Data Types and Registers**: Identifies the various data formats supported and how they can be manipulated. Discusses the number and function of accessible registers.
3. **Instruction Format and Addressing Modes**: Analyzes how instructions are structured and encoded. Evaluates the methods by which data is accessed and manipulated.

Operation Repertoire

The operation repertoire specifies the variety and complexity of operations that the ISA can execute, such as arithmetic, logical, and control instructions. It also indicates the level of abstraction the architecture supports: complex ISAs (CISCs) may support more intricate operations directly in hardware, while simpler ISAs (RISCs) often rely on fewer, more streamlined operations that can be executed quickly and efficiently.

MIPS ISA

MIPS is a RISC (Reduced Instruction Set Computing) architecture, meaning it emphasizes a simplified set of operations that can execute quickly and efficiently. MIPS operations can be categorized into five categories:

- **Data Manipulations**: These instructions contain arithmetic and logical instructions such as `add`, `addi`, `sub`, `AND`, `OR`, `NOT`, etc., which are applied on data.
- **Data Access/Transfer Operations**: These operations typically include load, store, and move instructions, which allow the processor to fetch data from memory, save data to memory, or transfer data between registers.
  - **Load Instructions**: Load instructions in MIPS are used to transfer data from memory into a register. Some common examples are:
    - `Load Word (lw)`: Loads a 32-bit word from memory into a register.
    - `Load Half-Word (lh)`: Loads a 16-bit half-word from memory into a register, sign-extending it to 32 bits.
    - `Load Half-Word Unsigned (lhu)`: Loads a 16-bit half-word, zero-extending it to 32 bits.
    - `Load Byte (lb)`: Loads an 8-bit byte, sign-extending it to 32 bits.
    - `Load Byte Unsigned (lbu)`: Loads an 8-bit byte, zero-extending it to 32 bits.
  - **Store Instructions**: Store instructions move data from a register to memory, storing values of different data sizes:
    - `Store Word (sw)`: Stores the contents of a 32-bit register into a 32-bit word in memory.
    - `Store Half-Word (sh)`: Stores the lower 16 bits of a 32-bit register into a 16-bit half-word in memory.
    - `Store Byte (sb)`: Stores the lower 8 bits of a 32-bit register into an 8-bit byte in memory.
  - **Move Instructions**: Move instructions transfer data directly between registers or to specific locations:
    - `move` is a pseudo-instruction in MIPS, meaning it does not have its own opcode. Instead, it is translated by the assembler into an equivalent instruction, typically `add $d, $s, $0`, where the source register `$s` is added to `$0`, effectively copying the value to the destination register `$d`.
    - `Move from HI (mfhi)`: Retrieves the contents of the special-purpose HI register and copies it into a general-purpose register.
    - `Move from LO (mflo)`: Retrieves the contents of the special-purpose LO register and copies it into a general-purpose register.
- **Control Transfer Operations**: Control transfer operations alter the normal sequential flow of execution in a program. These instructions are essential for implementing functions, loops, and conditional execution. They fall into two categories: unconditional and conditional.

    - **Unconditional**:
        - Unconditional control transfers always jump to a specified address.
        - A common example is the procedure call, where the program jumps to a new address to execute a function or subroutine. In MIPS, the `Jal (Jump and Link)` instruction is used for procedure calls, which saves the return address in the `$ra` register.
        - The `Jump` and `Jr (Jump Register)` instructions also perform unconditional control transfers.
    - **Conditional**:
        - Conditional control transfers change program flow based on specific conditions, allowing the program to make decisions and handle branches. In MIPS, common conditional control instructions include `beq (branch if equal)` and `bne (branch if not equal)`.
- **Special Instructions**: These instructions are designed for system-level tasks, debugging, handling exceptional conditions, and controlling the processor state.
    - `NOP (No Operation)`: Does nothing and passes through the pipeline without affecting registers or memory. Often used to resolve hazards or timing issues.
    - `syscall`: Triggers a system call, switching control to a predefined location in the operating system for executing operating system-level functions like I/O operations.

Quantum ISA (QSL)

We have developed a custom Quantum ISA inspired by the MIPS architecture, organizing our instructions into four distinct categories. We're implementing a RISC architecture similar to MIPS, but with a streamlined subset of instructions to enhance simplicity.

- **Quantum Logical Gates**: Includes a variety of single and multi-qubit operations. These instructions are classified as Q-Type instructions.
    - **Single-Qubit Gates**:
        - `H (Hadamard)`: Creates superposition.
        - `T` and `S`: Phase shift gates.
        - `X`, `Y`, `Z`: Pauli gates.
        - `RESET`: Resets the qubit to a standard state.
        - `MEASURE`: Measures the qubit and outputs the result to a dedicated "measure" bit.
    - **Two-Qubit Gates**:
        - `CNOT (Controlled-NOT)`.
        - `CZ (Controlled-Z)`.
        - `SWAP`: Exchanges the states of two qubits.
- **Control Instructions**:
    - `JMP (Jump)`: An unconditional jump instruction, similar to the `J` instruction in MIPS.
    - `JPP (Jump Positive)`: A conditional jump based on the value of a specific bit in a memory location, similar to `beq` in MIPS.
    - `JPN (Jump Negative)`: A conditional jump based on the value of a specific bit in a memory location, similar to `bne` in MIPS.
- **Classical Instructions**: These are R-Type instructions:
    - `AND(reg1, bit1, reg2, bit2)`: Performs a bitwise AND between specific bits in two registers.
    - `IOR(reg1, bit1, reg2, bit2)`: Performs a bitwise OR between specific bits in two registers.
    - `XOR(reg1, bit1, reg2, bit2)`: Performs a bitwise XOR between specific bits in two registers.
    - `NOT(reg1, bit1)`: Inverts the bit at a specified position in a register.
    - `NOR(reg1, bit1, reg2, bit2)`: Performs a bitwise NOR between specific bits in two registers.
    - `MOV(reg1, bit1, reg2, bit2)`: Copies a bit from one register to another.
    - `SET(reg1, bit1)`: Sets a specific bit in a register to 1.
    - `CLR(reg1, bit1)`: Clears a specific bit in a register to 0.

- **Memory Access Instructions**: These instructions allow reading from and writing to memory, similar to MIPS:
  - `LDB(reg1, bit1, reg2, offset)`: Loads a byte from memory into a specific bit in a register.
  - `STB(reg1, bit1, reg2, offset)`: Stores a byte from a specific bit in a register to memory.
- **No Operations (NOP) Instruction**: A "do nothing" instruction that allows the processor to continue to the next instruction.
- **Halt (HLT) Instruction**: Stops the program immediately.

Data Types and Registers

MIPS ISA

- **Data Types**:
  - **Integers**: MIPS32 handles 32-bit signed integers, but smaller (8-bit, 16-bit) or larger (64-bit) integers can be represented using multiple 32-bit registers.
  - **Floating Point**: Supports single-precision (32-bit) and double-precision (64-bit) floating-point numbers. These are handled by a separate floating-point unit (FPU) and stored in registers $f0 to $f31.
  - **Memory Addresses**: 32-bit addresses, allowing up to 4GB of memory. Instructions like `lw`, `sw`, `lb`, and `sb` use these addresses.
  - **Byte and Halfword (8-bit and 16-bit)**: MIPS allows operations on smaller data types using instructions such as `lb`, `lh`, `sb`, and `sh`.
- **Registers**:
  - **General-Purpose Registers**: 32 registers, each 32 bits wide, for data and intermediate results:
    - $0–$31 with specific roles (e.g., $0 is always 0, $29 is the stack pointer, $31 is the return address register).
  - **Special-Purpose Registers**:
    - **Program Counter (PC)**: Tracks the current instruction address.
    - **HI/LO Registers**: Store the results of multiplication and division, with HI storing the upper 32 bits and LO storing the lower 32 bits.

Quantum ISA (QSL)

- **Data Types**:
  - **Qubits**: The Quantum ISA uses 5 qubits, allowing 32 quantum states. These qubits are interconnected for entanglement and direct interactions.
  - **Classical Registers**: 7 classical registers (32 bits each) store results from quantum measurements or support data for classical operations.
  - **Buffer Register**: A dedicated 32-bit buffer register stores quantum measurement results, typically in the 0th bit.
  - **Primary Memory**: 96 bytes of memory (24 words, 32 bits each) used for storing classical data. Memory access is via pseudo-registers that reference memory blocks.
  - **Instruction Memory**: Up to 1024 instructions, each 18 bits, with separate instruction and data memories (Harvard architecture).
- **Registers**:
  - **Classical Registers**: 8 registers ($0 to $7) for storing classical data, with $0 reserved for measurement results.
  - **Primary Memory (Pseudo Registers)**: 24 pseudo-registers indexed $8–$31, representing memory locations in the primary memory. These are bit-addressable and used for memory operations like `LDB` and `STB`.
  - **Instruction Memory**: Separated from data memory, containing up to 1024 instructions.

Instruction Format

The MIPS architecture, developed by John Hennessy and his colleagues at Stanford in the 1980s, has been widely used in various commercial systems like Silicon Graphics, Nintendo, and Cisco. It is also known as a Load/Store Architecture. The design principles of MIPS, as articulated by Hennessy and Patterson, emphasize key concepts for efficiency and simplicity:

1. **Simplicity favors regularity.**
2. **Make the common case fast.**
3. **Smaller is faster.**
4. **Good design demands good compromises.**

MIPS ISA

MIPS32 follows a fixed instruction format with a 32-bit instruction length, making it simpler for both hardware implementation and performance optimization. Below are the key aspects of the MIPS32 instruction format:

- **Instruction Length**:

  MIPS32 uses a fixed instruction length of 32 bits for all instructions. Fixed-length instructions simplify instruction decoding and allow for more predictable execution in pipelines.

- **Address Space**:

  In MIPS, the address space is 4 GB, which equals 232232 bytes. Since each word is 32 bits (4 bytes), the number of addressable locations is calculated by dividing the total memory size by the size of each word. This results in 2324=2304232=230, meaning MIPS can address 1,073,741,824 (approximately 1 billion) word locations within its 4 GB address space.

- **Instruction Types**:

  MIPS32 instructions are divided into three formats:

  1. R-format (Register format)
  2. I-format (Immediate format)
  3. J-format (Jump format)

  These 3 formats are consistent with each other, favoring simplicity.

- **R-format (Register format)**:

  The R-format is used for arithmetic, logical, and shift operations involving only registers. It has the following fields:

  - Opcode (6 bits): Specifies the operation.
  - rs (5 bits): First source register.
  - rt (5 bits): Second source register.
  - rd (5 bits): Destination register.
  - shamt (5 bits): Shift amount (only for shift operations).
  - funct (6 bits): Function code that specifies the exact operation (e.g., add, subtract).

  Example: `add $t1, $t2, $t3`

- **I-format (Immediate format)**:

  The I-format is used for operations that involve immediate values, such as loading data from memory or performing arithmetic with constants or branch type instructions. It has the following fields:

  - Opcode (6 bits): Specifies the operation.
  - rs (5 bits): The source register.
  - rt (5 bits): The destination register or the register where the result will be stored.
  - Immediate (16 bits): A constant value (or offset) that is used in the operation.

  Example: `addi $t1, $t2, 10` (adds the constant 10 to the value in `$t2` and stores the result in `$t1`).

  The 16-bit immediate field in I-type instructions can be used for various purposes, such as loading a constant into a register or for branching operations.

  In branch instructions, the 16-bit immediate represents a signed offset (positive or negative) that is added to the address of the instruction immediately after the branch (the next instruction). This offset allows the processor to calculate the target address to jump to when the branch condition is met. The 16-bit immediate is sign-extended to 32 bits before it is used. This allows the offset to represent both positive and negative values, giving the ability to branch forward or backward in the program.

The effective target address for the branch is calculated by adding the sign-extended immediate value (multiplied by 4, because MIPS instructions are word-aligned) to the current PC value. This is done after the next instruction's address (PC + 4), since the 16-bit immediate is always specified relative to the instruction that follows.

- **J-format (Jump format)**:

  The J-format is used for jump operations, which alter the flow of control in a program. It has the following fields:

  - Opcode (6 bits): Specifies the jump operation.
  - Address (26 bits): The address of the target instruction (used with the current PC value to compute the full jump address).

  Example: `j 1000`

  In MIPS J-type instructions, the 26-bit immediate is extended to a 32-bit address in a specific way. First, two zeros are appended to the right of the 26-bit immediate, which is equivalent to shifting the value left by 2 bits (or multiplying by 4). This results in a 28-bit number, as all addresses in MIPS are word-aligned and therefore multiples of 4. Next, the top 4 bits of the current **PC + 4** (the address of the next instruction) are concatenated to the left side of the 28-bit number, forming the final 32-bit target address. This method allows the 26-bit immediate to efficiently encode the target address for jumps while maintaining word alignment.

- **Addressing Modes**:

  MIPS32 uses a register + offset addressing mode, where the address of an operand in memory is computed by adding a register value to an offset.

Quantum ISA (QSL)

Our Quantum ISA follows these same core design principles. Like MIPS, we focus on simplicity and regularity to ensure efficient operations, prioritize the most common operations for speed, and aim for an architecture that is both streamlined and effective.

- **Instruction Length**:

  Our Quantum ISA uses a fixed 18-bit instruction length, compared to the 32-bit instruction length in MIPS32 ISA, which streamlines instruction decoding.

- **Address Space**:

  In our Quantum ISA, the address space consists of 1024 addressable locations, with each location holding 8 bits (1 byte). The total memory size is therefore 1024 × 8 bits, or 1 KB.

- **Instruction Types**:

  Our Quantum ISA instructions are divided into four main categories:

  1. **Q-Type instructions** – Quantum operations on one or two qubits, including `H`, `T`, `X`, `Y`, `Z`, `S`, `RESET`, `MEASURE`, `CNOT`, `CZ`, and `SWAP`.
  2. **Control Instructions** – Control flow operations, including `JMP`, `JPP`, and `JPN`.
  3. **Classical Instructions (R-Type)** – Classical bitwise operations, such as `AND`, `IOR`, `XOR`, `NOT`, `NOR`, `MOV`, `SET`, and `CLR`.
  4. **Memory Access Instructions** – Operations to load and store data, including `LDB` and `STB`.
  5. **Special Instructions** – `NOP` (No operation) instruction.

Q-Type (Qubit Operations)

- **Opcode** (first 3 bits): The opcode `000` signifies a Q-type instruction for qubit operations.
- **qs** (second 3 bits): Specifies the first source quantum register for the operation.
- **Reserved** (third 3 bits): This field is reserved (set to `000`) and is not used in Q-type instructions.
- **qt** (fourth 3 bits): Specifies the target or second source quantum register for two-qubit operations.
- **func** (fifth 3 bits): Indicates the specific operation or gate to apply (e.g., `H`, `T`, `X`).
- **func** (sixth 3 bits): Additional bits for specifying operations, if required.

R-Type (Classical Operations)

- **Opcode** (first 3 bits): The opcode `001` signifies an R-type instruction for classical operations.
- **rs** (second 3 bits): Specifies the source register and the destination register (the final value is stored in this register).
- **sb** (third 3 bits): Specifies the bit position within the source register.
- **rt** (fourth 3 bits): Specifies the target / second source register.

- **tb** (fifth 3 bits): Specifies the bit position within the target register.
- **func** (sixth 3 bits): Defines the specific classical operation (e.g., `AND`, `OR`).

JPN (Jump if Negative)

- **Opcode** (first 3 bits): The opcode `010` denotes a JPN instruction.
- **rs** (second 3 bits): Specifies the register to check.
- **sb** (third 3 bits): Specifies the bit within the register for condition checking.
- **imm** (fourth 3 bits): Part of the 9-bit immediate value for jump address calculation.
- **imm** (fifth 3 bits): Continuation of the immediate field.
- **imm** (sixth 3 bits): Final part of the immediate value.

Here, `imm` provides the target address for the jump.

JPP (Jump if Positive)

- **Opcode** (first 3 bits): The opcode `011` denotes a JPP instruction.
- **rs** (second 3 bits): Specifies the register to check.
- **sb** (third 3 bits): Specifies the bit within the register for condition checking.
- **imm** (fourth 3 bits): Part of the 9-bit immediate value for jump address calculation.
- **imm** (fifth 3 bits): Continuation of the immediate field.
- **imm** (sixth 3 bits): Final part of the immediate value.

Here, `imm` provides the target address for the jump.

JMP (Unconditional Jump)

- **Opcode** (first 3 bits): The opcode `100` signifies an unconditional jump.
- **imm** (second 3 bits): Part of the 15-bit immediate value, specifying the target jump address.
- **imm** (third 3 bits): Continuation of the immediate field.
- **imm** (fourth 3 bits): Continuation of the immediate field.
- **imm** (fifth 3 bits): Continuation of the immediate field.
- **imm** (sixth 3 bits): Final part of the immediate value.

Here, `imm` provides the target address for the jump.

Load (Memory Access)

- **Opcode** (first 3 bits): The opcode `101` signifies a load instruction.
- **rs** (second 3 bits): Specifies the register where the data loaded is stored.
- **sb** (third 3 bits): Specifies the bit position within the `rs` register.
- **rt** (fourth 3 bits): Specifies the base register for the memory address.
- **imm** (fifth 3 bits): Part of the 6-bit immediate value, used as an offset for the memory address.
- **imm** (sixth 3 bits): Final part of the immediate offset.

Here, `imm` is used to determine an offset that is adjusted based on the content of a register and represents a location within `dataMem`.

Store (Memory Access)

- **Opcode** (first 3 bits): The opcode `110` signifies a store instruction.
- **rs** (second 3 bits): Specifies the source register for data to be stored.
- **sb** (third 3 bits): Specifies the bit position within the source register.
- **rt** (fourth 3 bits): Specifies the base register for the memory address.
- **imm** (fifth 3 bits): Part of the 6-bit immediate value, used as an offset for the memory address.
- **imm** (sixth 3 bits): Final part of the immediate offset.

Here, `imm` is used to determine an offset that is adjusted based on the content of a register and represents a location within `dataMem`.

NOP (No Operation)

- **Opcode** (first 3 bits): The opcode `111` identifies this instruction as NOP.
- **Remaining Fields** (second to sixth 3 bits): All bits are set to `000`, indicating no operation or effect.

The special instruction `halt` is represented by setting all 18 bits of the instruction to 1, signaling the processor to terminate execution.

Addressing Mode:

Our Quantum ISA utilizes a direct addressing mode for memory access, meaning that to access data at a specific memory location, the exact memory address must be provided. This approach simplifies memory access by removing the need for an offset calculation, directly specifying the target address in memory access instructions:

- `LDB` and `STB` directly use the specified memory address to load or store data without any offset.

This direct addressing mode streamlines data transfer between memory and registers by eliminating the need for base register calculations.

Brief Comparison of MIPS ISA and QUASAR ISA

Here, we compare the classical MIPS ISA with the quantum-oriented QUASAR ISA, specifically focusing on differences in operational capabilities, data handling, and instruction encoding methods critical for quantum control.

Comparative Analysis

1. **Operation Repertoire**
   - **MIPS ISA**:

     MIPS is a Reduced Instruction Set Computer (RISC) architecture that emphasizes simplicity and efficiency. The MIPS operation set is primarily classical, including:

     - **Arithmetic and Logical Operations**: Basic add, subtract, and multiply operations.
     - **Data Transfer Operations**: Load and store operations for moving data between registers and memory.
     - **Control Operations**: Branching and jump operations for program flow control.
   - **QUASAR ISA**:

     Designed as a RISC-V quantum extension, QUASAR introduces specialized quantum operations to manage qubit states, gate sequences, and the timing crucial for quantum circuit control. QUASAR's operation repertoire includes:

     - **Quantum Gate Operations**: QUASAR supports up to 15 unique quantum gates, including single-qubit gates (e.g., Pauli rotations) and two-qubit operations (e.g., Controlled-NOT).
     - **Measurement Operations**: Allows for qubit measurement and the retrieval of classical data from quantum states.
     - **Timing Control Operations**: Specialized timing instructions ensure that quantum gates are applied within precise time constraints necessary for quantum coherence.
     - **Masking and Parallelism**: QUASAR's mask-based operations enable simultaneous application of single-gate operations across multiple qubits, optimizing for Single Instruction, Multiple Data (SIMD) patterns essential for quantum circuit execution.
2. **Data Types and Registers**
   - **MIPS ISA**:

     MIPS relies on fixed-size data types and general-purpose registers (32 registers, each 32-bits), primarily supporting integers and floating-point data for classical computation.
   - **QUASAR ISA**:

     The QUASAR ISA is tailored for quantum control, addressing data requirements specific to qubit operations and quantum states:

     - **Qubit Registers and Addressing**: QUASAR supports up to 512 qubits with various addressing modes, allowing flexible targeting of qubits for operations.
     - **Quantum-Specific Data Types**: The ISA accommodates complex quantum data, including superposition and entangled states, through register configurations that can support quantum state vectors and measurement results.

- **Register-Based Addressing for Quantum Masks**: QUASAR uses a 32-bit mask register format, with a sliding mask approach allowing multiple qubits to be targeted by a single instruction, supporting efficient batch operations on quantum data.
- **Immediate and Direct Addressing**: For immediate addressing, qubit IDs are encoded in a 9-bit field, while larger groups of qubits are specified using the mask format.

3. **Instruction Format and Addressing Modes**

- **MIPS ISA**:

  MIPS uses a straightforward 32-bit instruction format with immediate, register, and base displacement addressing modes. These allow for straightforward encoding and decoding, but are limited in supporting complex quantum operations or parallelism.

- **QUASAR ISA**:

  QUASAR modifies the RISC-V format with a fixed 32-bit instruction length but includes specialized fields for quantum control:

  - **Dual Addressing Modes**: QUASAR includes both direct (immediate) and mask addressing modes. In immediate mode, a 9-bit immediate field is used for single-qubit identification. The mask mode allows for operations on multiple qubits simultaneously, utilizing a 4-bit immediate value to indicate mask offset, supporting up to 32 qubits per mask.
  - **Quantum Gate and Qubit Encoding**: The instruction format includes fields for specifying gate types (up to 15 unique gates) and qubit targets, as well as timing synchronization. This allows QUASAR to handle complex quantum gate sequences efficiently.
  - **Pipeline Optimization and Timing Constraints**: QUASAR's encoding approach is optimized for pipeline stages to meet the precise timing demands of quantum gates. Instructions are structured to minimize pipeline delay, accommodating the real-time control required by quantum circuits where operations must align with qubit coherence times.
  - **Instruction Compactness and Scalability**: The encoding efficiency of QUASAR allows for a dense representation of quantum operations, reducing overall instruction count and memory footprint. This compactness is critical for handling larger quantum circuits and for enabling the ISA to scale effectively with future quantum chips of greater qubit count.

▼ Hardware Software Interface of Quantum Computers

Necessities for Quantum Computing (Hardware Aspect)

Based on the paper [1] by David DiVincenzo in the year 2000, the following five "necessities" were given for performing Quantum Computation. The first of them is measurability. In simple words, every single qubit, even in a multi-qubit system, should be individually measurable (to a high degree of accuracy). The second necessity was of Universality. According to this, all the quantum operations that we would perform in a quantum computer should be decomposable to a set of some "universal gates". This sounds analogous to the ability to decompose all the operations in a conventional ALU to the basic operations such as NOT, AND, OR, and XOR (all of which can be modelled by just using NAND or NOR, which are said to be universal). The third condition is scalability. By this, we should be able to increase the number of Qubits and the logical operations so that the cost does not get more than the benefits. The next component of this is the initializability. In this, we should be able to initialize a qubit with |0> state, which is necessary for the predictability of the computation and the creation of consistent results. The fifth and last property to have is coherence. This is a major problem in Quantum computers that tend to have the issue of losing coherence before the quantum gate execution due to the interference of Noise, Heat, Electromagnetic fluctuations, Vibrations or Particle spins in the environment.

The Hardware Software Interface of a Practical Quantum Computer

As we can understand from [2], to realize quantum computation, we need to look at the structure of quantum computers. Similar to the usual computers, the highest layer is still the application layer. It essentially provides the environment for the programmer, OS and UI. This application level has to be independent of the underlying hardware, the way Microsoft Teams is independent of whether it runs on an i5 or an AMD processor. The application layer is then connected on a lower level to the classical processing layer. This layer has three main tasks. The first task is to perform the compilation, the way a compiler decomposes a high-level instruction to micro instructions that are essentially "understandable" by the device in the form of an ISA. Apart from this, since the result of the quantum program is obtained by measuring the qubits at the final stages, this also processes the measurements performed by lower levels. It also takes care of the calibration and tuning required for the layers below it. Apart from this, since the quantum algorithms may also have a component of classical computation along with them, this layer also performs the classical computations required. The microinstructions from this layer then travel down to the digital processing layer, which translates the microinstructions that we made in the previous layer to some pulses in digital pulses, which further go ahead to instruct how the analog signals to the QPU should be. In the other direction, it also serves as a layer that feeds the measurement from the lower layers to the higher layers, ensuring that the measure values are compatible

with the classical hardware part. Based on the inputs from digital layer, the analog processing layer creates various signals that will be sent to the underlying quantum hardware to perform qubit operations. These are mainly in the form of phase and amplitude-modulated voltage steps and bursts of microwave pulses. These signal then are used by the Quantum Processing layer, which then performs the required operations and returns the output (after measurement) to the higher layers, which are transferred from analog formats to digital formats as we move above. The figure given below describes this visually.

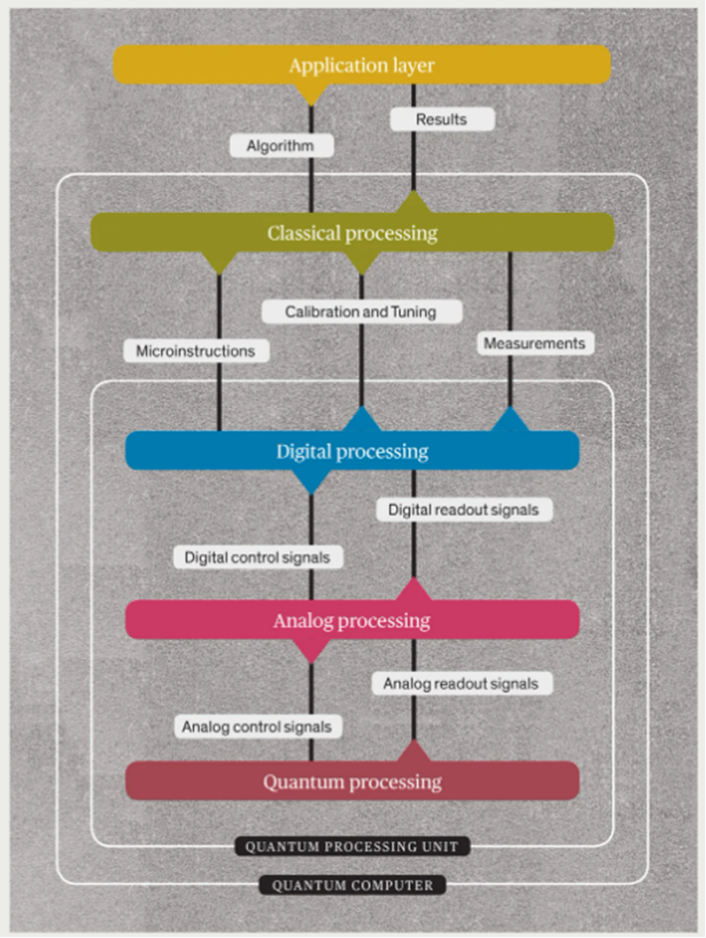


Figure: Layer Cake: The components of a practical quantum computer can be divided into five sections, each carrying out different kinds of processing. Source [2]

There is another way of visualising this as proposed by the National Academic Press, Washington DC in the book on Quantum Computing: Progress and Prospects [3]. In chapter 5, they describe the essential hardware of a Quantum Computer, where they divide the Quantum Computing hardware of the quantum computer into various levels such as the “quantum data plane”, “control and measurement plane”, the “control processor plane”, and the “host processor plane”. Starting with the Quantum data plane, we can observe the presence of physical qubits and the structures to hold them in their place, along with additional circuitry to perform the gate-level implementations as well as measurements. Thus is the Quantum processing layer as described in the previous model. The control and measurement plane converts the control signals from digital signals to analog signals for performing the necessary computation. It also converts the analog data from the measurement operation to the classical bits for further computation. This is nothing but a combination of the Digital Processing and Analog processing layers in the previous paragraph. The control processor plane processes the input instructions from the host processor and creates a sufficient sequence of quantum gate operations and the measurements that will be sent to the lower levels for further processing. Interestingly, it also has the task to perform the error correction algorithms if the machine has error corrections in it. This is analogous to the classical layer in the previous model. The next plane corresponds to the host processor, which is basically a conventional computer that runs software such as Operating Systems and Compilers (which are based on the ISAs). This is analogous to the application layer in the previous models.

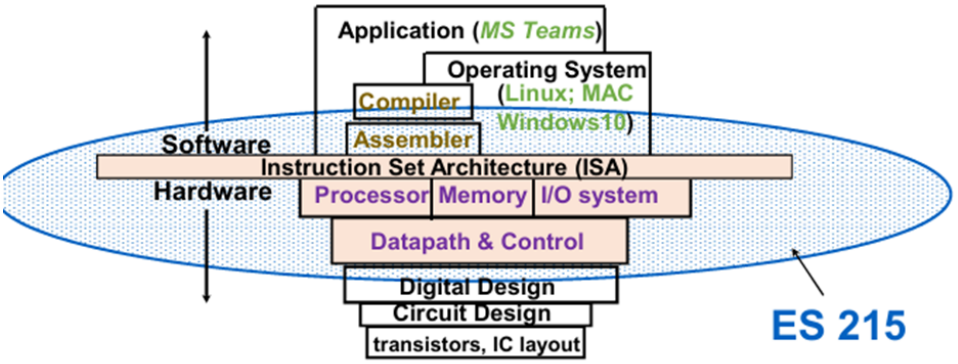


Figure: Classical Computer Architecture. Source: Lecture Slides, ES215 (2024), IIT Gandhinagar

On the whole, we have reviewed two models of quantum computer architectures. We can see that both the models, though described in different ways, convey the same information. However, the architecture mentioned in the former is more detailed. Similar to the classical computers, we can see that there is an application layer, which deals with providing an interface for the user and, to some extent, provides a stage for the software, like the operating systems and compilers, to function. Similar to the classical systems, their OS, compilers and other system-level software are also based on their ISAs. The areas where we can find the most differences are in the lower levels. While conventional computers have a setup of processors along with Memory and IO systems with control and the lower levels performed in the digital system, the digital operations themselves are performed using transistor-based circuits, the quantum computer, on the other hand, has all this in the classical processing layer and to some extent in the digital layer, the core of the things start happening from the analog layer and the quantum processing layer as we discussed previously. In simple words, while the conventional processor works directly with digital signals, the quantum processor requires analog signals, as we discussed previously. Another striking difference can be seen in still lower levels. While the present-day transistor implementations for the conventional computers are based on semiconductors, namely silicon, our discussion of the quantum level/plane is quite abstract, and this leads us to more in-depth discussion on the physical implementation of this quantum layer/level.

▼ Physical Implementation of Quantum Computers

While present-day classical computers are developed on semiconductor (principally silicon) based transistors, the Quantum level/plane in the quantum computers are experimentally built on many different bases as discussed in the source [4], [5] and [6].

Though there exist some models of quantum computing, such as quantum annealers and quantum gate-based models, they formally lack universality and as a result, there are many methods that are used to implement the latter (gate-based models). The Qubits in this can be made of multiple types of implementations such as photons, ions, superconductors, etc., contrary to the semiconductor transistor-based implementation of the current conventional processors.

The implementation of these qubits, as discussed above, can be done in multiple ways. One of the most common ones is to use Superconductor-based qubits. They are made using the artificial atoms from Josephson junction-based nonlinear superconducting circuits. In a Josephson junction, we have a flow of a supercurrent even in the absence of a voltage across the Josephson junction. This acts like a quantum oscillator with two-state systems. In this, every single qubit is coupled with a microwave resonator, which performs single-qubit operations, while the multi-qubit gates are implemented using microwave resonators that are connected to multiple qubits. They have become some of the most widely used methods of implementing quantum computers because of their rapid growth of coherence time. A schematic diagram corresponding to the superconductor-based qubit implementation is given below.

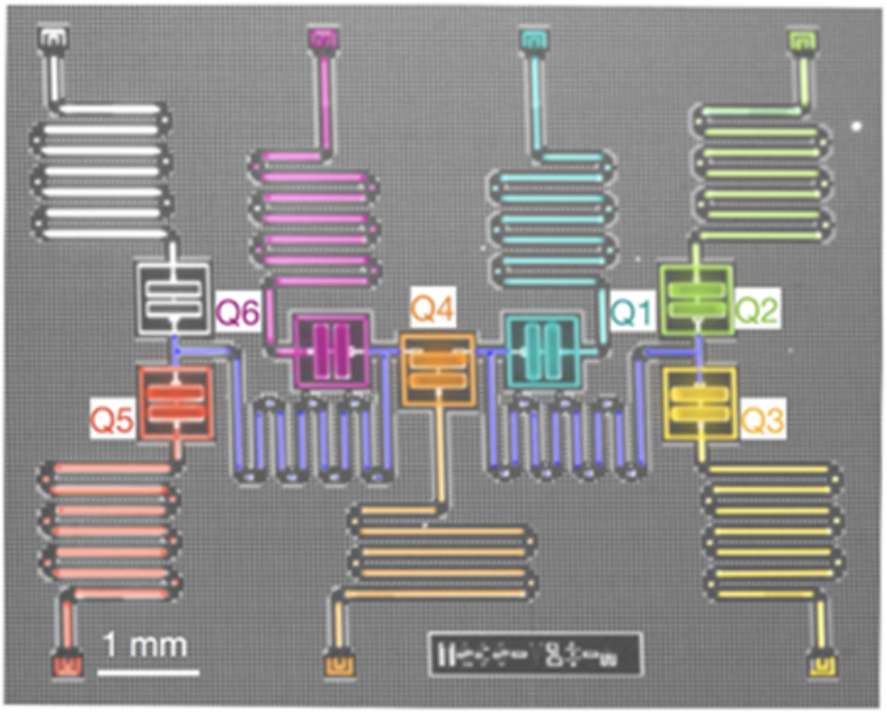


Figure: Superconductor based qubit implementation. Source [5]

Another currently used method is based on NMR (Nuclear Magnetic Resonance). It is done upon the immersion of many atoms into a magnetic field of 0 frequency and then subsequently undergo fields exhibiting low and high frequencies. We can generate qubits upon the interaction of the spin inherent to the electrons with respect to the outer nuclear spin. The alignment of that can either be in the parallel or antiparallel styles. Upon the usage of radio-frequency magnetic fields, we can get the single qubit gates, while for multiple qubit gates, we use the interaction of spins within one molecule. However, it faces the scalability issue which we had discussed previously. Because of this, as the number of qubits grows, the decoherence time significantly reduces. The attached figure describes the hardware for NMR-based implementation.

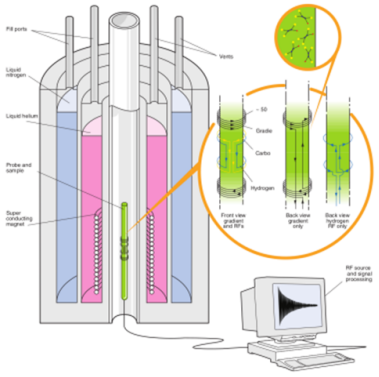


Figure: NMR-based implementation. Source [5]

There also exist implementations based on semiconductors. These are electrons that spin in the 2-dimensional electron gases inside the structure of a semiconductor. This is also called quantum dots which correspond to the electrons in the lattice of the semiconductor. The qubit states are corresponding to the spins of the electrons. The single-qubit gates are implemented using time-dependent magnetic fields to manipulate the single electron spins, while the multi-qubit gates are implemented using the exchange interactions between the spins of the neighbouring dots. The visual representation of this methodology is given in the figure below.

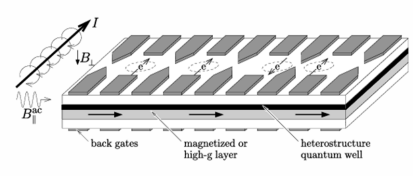


Figure: Semiconductor-based quantum dot implementation. Source [5]

We also have implementations based on topological qubits and trapped ion qubits. In topological architecture, they are made using exotic quasiparticles, which we call as anyons. They are used to generalize the bosons and fermions, and due to this, they exhibit some non-trivial quantum properties described by their topologies. In the trapped ion-based architecture, we have atomic ions that are cooled by lasers in a high vacuum and trapped using high-power electric fields. This implementation is also quite popular due to the longer coherence times. To implement single qubit gates in the electromagnetic traps, we use laser beams. We utilize the ion-ion interactions as well for multi qubit gate implementation. The visual representation of this implementation is provided in the figure given below.

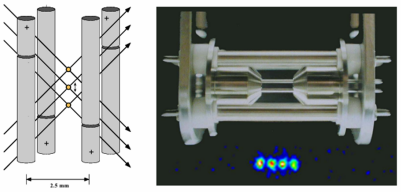


Figure: Implementation of Ions in Electromagnetic traps based-qubit implementation. Source [5]

As we can compare the various methods of implementing the quantum gate-based computers as given above, we can compare them with the classical computers right from the implementation. While the classical computers have a fixed bit value (either 0 or 1) and are currently implemented using flip flops (generally), which are, in turn, transistors based on semiconductors, we have the qubits based on many different implementations as described above. Moreover, they do not have a deterministic state at a particular instance of time and it is a superposition of the pure states, owing to the quantum nature of the qubits. On the whole, these differences, along with the ones stated above, contribute significantly to the differences between conventional and quantum computers.

▼ Quantum Benchmarking

Classical Benchmarking Methods

As discussed in the class as well as from the paper [7], we will first start with the understanding of the meaning of what benchmark means. In simple words, we need some standardised tools for evaluating and comparing the processors in terms of their performance, dependability and security. There are two principal types of standardised benchmarks, namely SPEC and TPC. The SPEC and TPC benchmarks usually align themselves to the comparison based on various specialised operations in their entire test bench, which also includes a comparison of speed, throughput, power consumption, etc., for various types of operations, such as integer-based operations and floating-point operations. We also have certain properties that need to be satisfied for a particular set of tools to be used as benchmarks. These include relevance, which indicates how closely the benchmark program behaves to the real-world application of the same type. We also should consider the reproducibility of the results, which is the ability to consistently produce the same results at each test stage. There should also be fairness, which is the absence of any artificial constraints in smoothly executing the benchmarks. The results that are generated should also be verifiable. This would ensure that the benchmark results are accurate and can be verified conveniently. The final requirement is of usability. The benchmark should not be extremely complex or should not have inaccessible libraries used, which would create difficulties in using the benchmark programs.

Quantum Benchmarking Methods

For the process of benchmarking quantum processors, we usually have to look for three main factors, namely the Hardware factors, the Algorithmic (concerning QPU Evaluation on the whole) factors and the Application factors [8]. The Hardware factors matter because, as we discussed in the physical implementation section, all the hardware has certain advantages and disadvantages. Similarly, there may be differences between the applications for which the particular processors were built. There may also be the differences between the algorithms utilized, which will directly have implications over the depth of the circuits and finally over other important parameters like the error tolerance. This leads us to the division of the quantum benchmarks into three main categories, namely physical benchmarks, aggregated benchmarks and application-level benchmarks. The division of the benchmarks based on categories is as given in the Figure below.

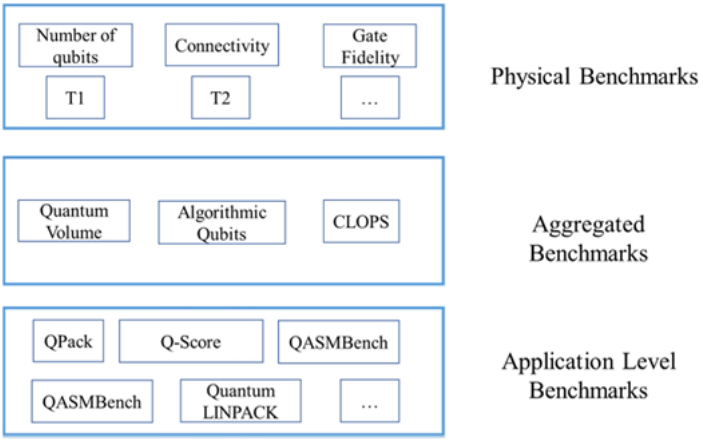


Figure: Overview of Quantum Benchmarks. Source [8]

Physical Benchmarks

There are many ways in which quantum computers are actually implemented. We had actually performed a detailed analysis on this in the Physical Implementations section. Due to this variation in the physical implementation, there are also certain differences in the performances of the devices. The major physical parameters which are considered in the benchmarking for quantum computers include:

- Number of Qubits: This indirectly describes the capacity of the quantum computer in terms of the number of qubits it can compute (or operate on).
- Qubit Connectivity: One major aspect that we saw in the Physical Implementations section was that, in order to implement the physical qubits with multiple qubit gates, we should have the qubits interacting (connected) to each other. It is, therefore, not possible that all the bits are connected to all the other qubits. The connectivity between the qubits is, therefore, represented as a graph and is called the qubit topology. An interesting thing to understand here is that the performance will have an improvement upon an improvement in the qubit connectivity because we will be able to operate on “farther” bits more conveniently.
- T1 and T2: The T1 value corresponds to the energy relaxation time, which is the time required for the quantum state to decompose from a high-energy state to a low-energy state, while the T2 time is the inverted and weighted sums of the pure dephasing time and T1. These times give an idea about the stability of the qubit and the influence that noise may have over the qubit values.
- Fidelity: Due to the noise at every level of quantum gate operations, we may have the possibility of errors. With more and more errors at every level of operations, we may tend to move away from the ideally expected results. This divergence from the ideal values is measured by the fidelity values. Basically, fidelity is also a measure of the impact of the errors at all these stages, which is the value of the divergence of the ideal outputs that we actually expect from the actual values that we get.

Aggregated Benchmarks

The metrics that we discussed above are based on the physical implementations of the quantum computers. However, they do not involve the performance of the quantum processor as a whole, which consists of different qubits, which are, in turn, arranged in multiple qubit topologies. This is evaluated by the aggregated benchmarks. Some of them are:

- QV: QV (Quantum Volume) was a popular aggregated benchmark released by IBM which tries to remove the susceptibility of factors such as number of qubits, topology, error rate, etc. when we perform the operations. QV is taken as 2^k where it runs k-layer gate operations (of Haar-random SU(4) unitaries) on k qubits and considers both the number of qubits and the quality of gate operations and measurements, after which it finds the qubit operations that are above a particular level of thresholds of correctness. It basically rewards for better fidelity.
- Algorithmic Qubits: The AQ metric overcomes the “very large QV metric” problem (because of the exponential nature of QV by introducing the idea of measuring how large Quantum Circuits a quantum computer can handle (execute). It also considers other parameters, such as the error corrections of the qubits and the number of qubits in the processor, for a proper determination of the AQ value. It was given by IonQ.
- CLOPS: There are also certain properties that we need to consider for evaluating the performance of a quantum computer, namely the quality, speed and scale. The quality can be measured by metrics such as QV and fidelity. The scale can be measured to a good extent by the AQ and the number of qubits. However, there was no such metric for speed until CLOPS was proposed. CLOPS stands for the Circuit Level Operations Per Second. This is the metric that considers the number of QV layers that are executed per second and takes into account all the latencies, data transfer times, gate times, etc. and finally returns the CLOPS rating.

Application-Based benchmarks

Different applications require different types of metrics. Some applications that focus on a particular application may require a particular type of processor, while the other applications may not favour that. The type of processor is also dominated by the application and the types of gates that it will run. A processor running a particular type of operation much better than the other processor will be preferable for applications that heavily use that particular operation. The following are some application-based quantum benchmarks:

- Qpack: It mainly deals with problems such as Max-Cut, Dominating Set and the Travelling Salesman Problem (TSP) and considers the runtime, best approximation error, success probability and performance scaling.
- Q-Score: It mainly has applications based on TSP and Max-Cut and evaluates the QPU hardware along with the software stack by assigning a Q-Score.
- F-VQE: This is a benchmark for the performance of random weighted Max-Cut problem efficiently.
- Quantum Chemistry: This is a quantum chemistry benchmark that tends to specialize in the benchmarking of the series of electronic structure calculations instances. This is open source.
- OpenQASM: This is a low-level benchmark based on assembly language OpenQASM. It deals with various domains, which include Chemistry, Simulations, ML, Linear Algebra, cryptography, etc. and evaluates the depth and width of the circuit, gate density, retention lifespan, etc.

There are some more types of application-level benchmarks, including DDQCL (Data-Driven Quantum Circuit Learning Algorithm), Quantum LINPACK, etc. In contrast to the conventional benchmarks, however, some striking differences we can observe are in terms of the metrics, which also include the circuit depth, the width, the error rate (fidelity) and the coherence times. These are especially necessary because quantum circuits are much more error-susceptible in their operations than conventional computers. This also has an implication over the number of qubits and directly, the scale of operations that they can work on.

▼ Comparison of Operations on which Quantum is better and the Classical is better

The sources for the following study were [15], [16].

1. Data Representation

- **Quantum Computers are better**: For problems where data can benefit from superposition, such as complex probabilistic or combinatorial problems, quantum data representation (qubits) allows multiple states to be represented at once, potentially speeding up certain computations. This concept is also discussed above in detail, so I am not going into the details here.
- **Classical Computers are better**: For deterministic applications or general-purpose computing tasks, classical data representation (bits) is more stable, precise, and reliable, as classical bits do not face issues with noise and decoherence. From the course on Computer Organisation and Architecture, I learned that though quantum computers may be very efficient in some cases but it is still in development so when it comes to stability of operations, then classical computers do take the lead advantage because of the thorough and rigorous research in this field since long.

2. Logical Operations

- **Better on Quantum**: Quantum gates are advantageous in problems that require parallel state manipulation, like factorization (using Shor’s algorithm) or search algorithms (using Grover’s algorithm). Quantum gates can perform complex state transformations in fewer steps than classical logic for these tasks. As I learned from the books [15] and [16], the superposition and entanglement of the states are the most advantageous elements in contrast to the classical computers.
- **Better on Classical**: Classical logic gates are ideal for most day-to-day tasks and applications where deterministic, sequential processing is essential. For tasks involving basic computations, classical logic is efficient, reliable, and widely optimized in current technology.

3. Memory Architecture and Data Access

- **Quantum takes the lead on**: Quantum memory architecture is suited for specialized computations where data in quantum registers can be processed in superposition, offering advantages in quantum simulations and certain scientific computations.
- **Classical takes the lead on**: Classical memory (RAM, cache) provides fast, direct access to data, making it vastly superior for applications requiring quick, reliable storage and retrieval. Random access and direct addressing in classical computers support a wider range of applications and general-purpose computing.

4. Processing and Parallelism

- **Quantum is advantageous because**: Quantum processors excel in tasks that can leverage **quantum parallelism**. Problems like optimization, complex simulations, and cryptography benefit from this type of parallelism, as quantum systems can explore multiple states or solutions simultaneously.
- **Classical is advantageous because**: Classical processors, especially with multi-core architectures and GPUs, are effective for a broad range of parallel processing tasks and can handle most current-day applications efficiently. Classical parallelism, based on deterministic cores and pipelines, is superior for routine computational tasks and workloads involving massive data handling.

5. Error Handling and Correction

- **Better on Classical**: Error correction in classical computers is straightforward and highly reliable due to low error rates in classical circuits. Classical computers are thus better suited for tasks requiring strict accuracy and reliability, such as financial computations or medical applications.
- **Better on Quantum**: Quantum error correction schemes are specialized but critical in quantum computing. While challenging to implement, they are essential for long-duration quantum computations. As quantum error correction evolves, it may enable quantum systems to tackle more complex problems with higher reliability, although classical error handling remains more efficient and practical for most uses.

6. Readout and Output

- **Classical takes the lead for the fact that**: Classical computers offer immediate, precise, and repeatable outputs. For applications where deterministic results are necessary, such as real-time processing, classical readout is superior.
- **Better on Quantum**: Quantum readout allows for probabilistic outputs useful in scenarios where multiple possible states need to be averaged or explored, such as in probabilistic models, simulations, and machine learning applications. However, it requires multiple measurements, making it less practical for everyday deterministic tasks.

7. Energy Efficiency and Cooling

- **Better on Classical**: Classical systems are optimized for energy efficiency with standard cooling methods, making them practical for general use, including mobile devices, servers, and supercomputers.
- **Better on Quantum**: Quantum computers are not yet optimized for energy efficiency due to extreme cooling requirements. While theoretically they could perform specific tasks more efficiently, current quantum systems require significant infrastructure, making them less practical in terms of energy use for general applications.

To summarise:

Category	Better on Quantum	Better on Classical
Data Representation	Quantum data representation (qubits) allows multiple states to be represented at once, useful for **complex probabilistic or combinatorial problems**.	Classical data representation (bits) is stable, precise, and reliable, ideal for **deterministic tasks without issues from noise and decoherence**.
Logical Operations	Quantum gates are effective for **parallel state manipulation, like in Shor's algorithm (factorization)** and Grover's algorithm (search).	Classical logic gates are efficient and reliable for most day-to-day tasks requiring **deterministic, sequential processing**.
Memory Architecture and Data Access	Quantum memory is advantageous in specialized **computations using superposition, benefiting quantum simulations and scientific computations**.	Classical memory (RAM, cache) **supports fast, reliable storage and retrieval with direct addressing**, suitable for a wide range of applications and general computing.
Processing and Parallelism	Quantum processors **excel in tasks that benefit from quantum parallelism, such as optimization, complex simulations, and cryptography**.	**Classical multi-core processors and GPUs are efficient in a wide range of parallel processing tasks**, handling data-intensive and routine computational workloads.
Error Handling and Correction	Quantum error correction schemes **are evolving to support long-duration quantum computations**, though still challenging to implement.	Classical error correction is **straightforward and reliable due to low error rates, suited for tasks needing strict accuracy like financial and medical computations**.
Readout and Output	**Quantum readout allows probabilistic outputs, helpful for exploring multiple states, in probabilistic models, simulations, and machine learning**.	Classical readout provides **precise, repeatable, and immediate results, essential for real-time processing and deterministic applications**.
Energy Efficiency and Cooling	Theoretically, quantum computing **may be more energy-efficient for specific tasks, but current systems require extreme cooling, limiting practicality for general applications**.	Classical systems are **optimized for energy efficiency and cooling, practical for mobile devices, servers, and supercomputers** in day-to-day applications.

▼ Better Algorithms in Quantum Computers than Classical Computers

Shor's Algorithm [19][20][21]

Integer factorization is the process of determining a given number's prime factors. Though classical computers are very efficient in finding the product of numbers but they struggle a lot when it comes to the Interger factorization. Shor's method is a quantum algorithm designed to address this issue. This approach has significant implications for cryptography due to its strong factoring capabilities, which can render some encryption methods (such as RSA) vulnerable.

Shor's algorithm utilizes Quantum Fourier Transform (QFT) and Quantum Phase Estimation (QPE) to efficiently find the period of a function, which is then used to factor N . Below are the main steps involved in Shor's algorithm.

Steps in Shor's Algorithm

1. **Classical Preprocessing:**
   (a) Choose a random integer a such that $1 < a < N$.
   (b) Compute $\gcd(a, N)$. If $\gcd(a, N) \neq 1$, then a is a non-trivial factor of N, and the algorithm terminates with a as a factor.
   (c) If $\gcd(a, N) = 1$, proceed with the quantum part to find the period r of the function $f(x) = a^x \mod N$.
2. **Quantum Phase Estimation (QPE) to Find the Period (r):**
   (a) Prepare two quantum registers:
      - The first register, an n-qubit register, initialized to the state $|0\rangle^{\otimes n}$.
      - The second register, a single-qubit register, initialized to $|1\rangle$.
   (b) Apply a Hadamard transform to each qubit in the first register:
   (c) Implement the unitary operation $U_{a,N}$ defined by $U_{a,N}|y\rangle = |a^y \mod N\rangle$ on the second register, conditioned on the first register. This entangles the two registers.
3. **Apply Quantum Fourier Transform (QFT) to the First Register:**
   (a) Perform the Quantum Fourier Transform (QFT) on the first register to extract the periodicity information.
   (b) Let us measure register 1. The measurement gives us $k \cdot 2^n/r$, where k is a random variable uniformly distributed from 0 to $r - 1$. It is easy to see that with high probability $\gcd(k, 2^n/r) = 1$. If so, then by computing $\gcd(k \cdot 2^n/r, 2^n)$, we obtain 2^n. Since we know 2^n, from $2^n/r$ it is straightforward to compute r as an approximation.
4. **Classical Post-Processing:**
   (a) If r is odd or $a^{r/2} \equiv -1 \mod N$, then repeat the algorithm with a different random a.
   (b) Otherwise, r is the period of $f(x)$. Use r to find factors of N by computing $\gcd(a^{r/2} \pm 1, N)$.

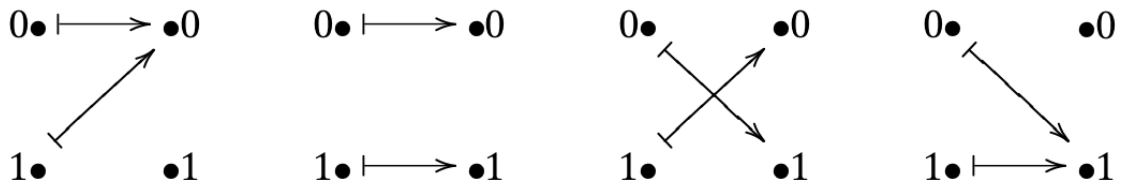
So, Shor's algorithm gives quantum computers a significant edge since it can factor large numbers exponentially quicker than traditional computers.

While classical algorithms have sub-exponential time complexity, Shor's algorithm operates in polynomial time, specifically O((log N)^3), where N is the number to be factored.

Therefore, Shor's method demonstrates one of the most compelling advantages of quantum computing: it can be utilized to tackle problems that conventional computers cannot handle.

Deutsch's Algorithm [15][16]

Deutsch's Algorithm is one of the simplest quantum algorithms yet a nice and very useful algorithm that is concerned with the functions from the set {0, 1} to the set {0, 1}. There are four such combinations/functions which can be visualised as:



A function f: {0,1} → {0,1} is balanced if f (0) != f (1), i.e., it is one-to-one. In contrast, a function f: {0,1} → {0,1} is called constant if f(0)= f(1). Of the four functions immediately above, two are balanced, and two are constant. [16]

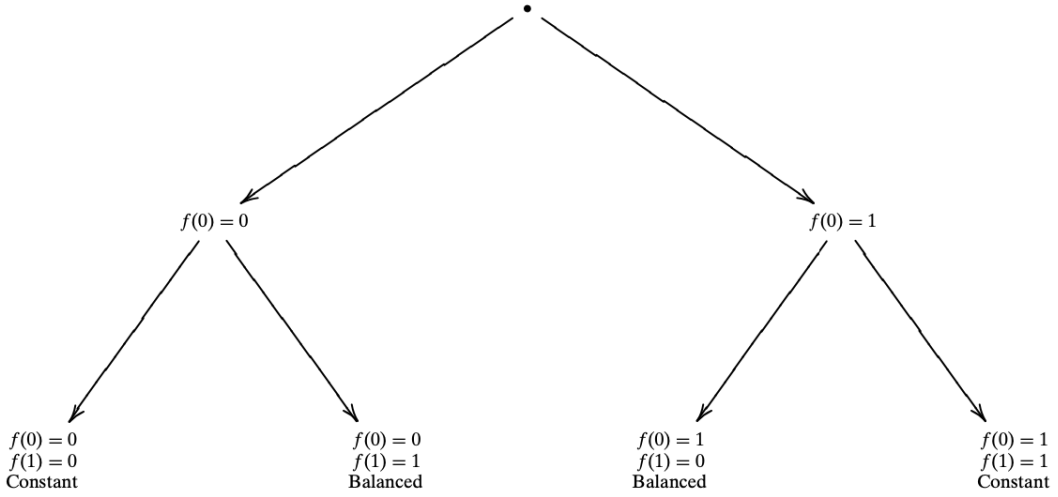
The problem that the algorithm solves is:

Imagine a black-box function (also called an "oracle") that takes a bit (0 or 1) as input and gives an output. The function can either be **constant** or **balanced**:

- **Constant**: The function always gives the same output, whether the input is 0 or 1.
- **Balanced**: The function gives different outputs for 0 and 1 (one of the outputs will be 0, and the other will be 1).

Our goal is to figure out whether the function is constant or balanced. With a classical computer, we'd need to evaluate the function at least twice (once for 0 and once for 1) to be sure. But with Deutsch's quantum algorithm, we can solve the problem with **just one evaluation** of the function! [16]

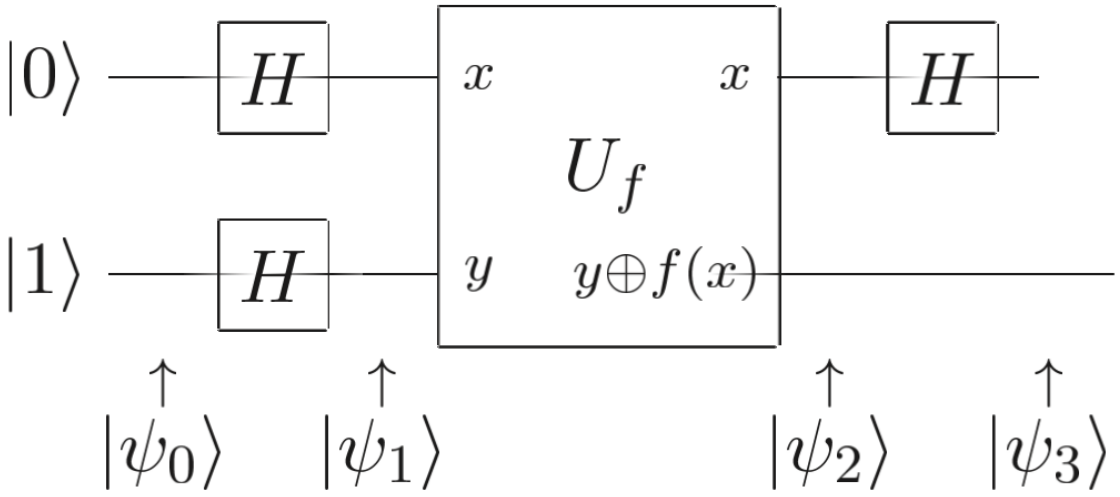
The following decision tree shows what a classical computer must do:



Deutsch’s algorithm uses a property of quantum mechanics known as interference in combination with quantum parallelism.

Note: “Quantum interference is a phenomenon in quantum mechanics that arises from the wave-like nature of quantum particles such as electrons or photons. When a particle is in a superposition of multiple states, these states can interfere with each other leading to constructive or destructive interference. (As a reminder, superposition describes the condition in which a quantum system can exist in multiple states or configurations simultaneously.) This interference can result in specific patterns of outcomes when the particle is measured. In quantum computing, interference is used in various ways to manipulate and control quantum states and to perform computational tasks. By applying quantum gates that create superpositions of qubits, and by controlling the relative phases of the states, interference can be used to amplify certain outcomes and suppress others.” [17]

Below is the diagram for the algorithm which uses quantum gates, where ψ represents each stage of the algorithm. [16]



The input state

$$|\psi_0\rangle = |01\rangle$$

is passed through the two Hadamard gates to yield

$$|\psi_1\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right]$$

If we apply Uf to the state $|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$ then we obtain the state $(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$.

Therefore, applying U_f to $|\psi_1\rangle$, we get,

$$|\psi_2\rangle = \begin{cases} \pm\left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) = f(1) \\ \pm\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) \neq f(1). \end{cases}$$

Finally, the Hadamard gate on the first qubit yields

$$|\psi_3\rangle = \begin{cases} \pm|0\rangle \left[\dfrac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) = f(1) \\ \pm|1\rangle \left[\dfrac{|0\rangle - |1\rangle}{\sqrt{2}}\right] & \text{if } f(0) \neq f(1). \end{cases}$$

Realizing that f(0)⊕f(1) is 0 if f(0)=f(1) and 1 otherwise, we can rewrite this result concisely as

$$|\psi_3\rangle = \pm|f(0) \oplus f(1)\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right]$$

So, by measuring the first qubit we may determine f(0)⊕f(1).

With just one evaluation of f(x), we can now establish a global attribute of f(x), namely f(0)⊕f(1), thanks to the quantum circuit! Compared to a classical apparatus, which would need at least two evaluations, this is quicker.

Quantum Random Number Generators (QRNGs) [22]

Using quantum mechanics, a Quantum Random Number Generator (QRNG) produces actual random numbers. Unlike classical random number generators that rely on algorithms or physical processes that may eventually become predictable, QRNGs are really unpredictable and suitable for high-security applications by using the inherent unpredictability of quantum occurrences.

Steps to build the QRNG circuit:

1. **Initialize the Quantum Circuit:** Define a quantum circuit with 4 qubits using `num_qubits = 4` and create the circuit as follows:

   `qc = QuantumCircuit(num_qubits)`

2. **Apply Hadamard Gates:** To create superposition, apply a Hadamard gate H to each qubit, making each qubit equally likely to be $|0\rangle$ or $|1\rangle$:

   $$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

   This is done with the code:

   `for qubit in range(num_qubits): qc.h(qubit)`

3. **Measure the Qubits:** Measure each qubit to collapse it to either 0 or 1 based on its superposition state:

   `qc.measure_all()`

4. **Run the Circuit on a Simulator:** Execute the circuit on the Qiskit `qasm_simulator` with a single shot to get one random 4-bit outcome:

   `job = execute(qc, simulator, shots=1)`

5. **Output the Random Number:** Convert the binary result to decimal format:

   `random_number_int = int(random_number, 2)`

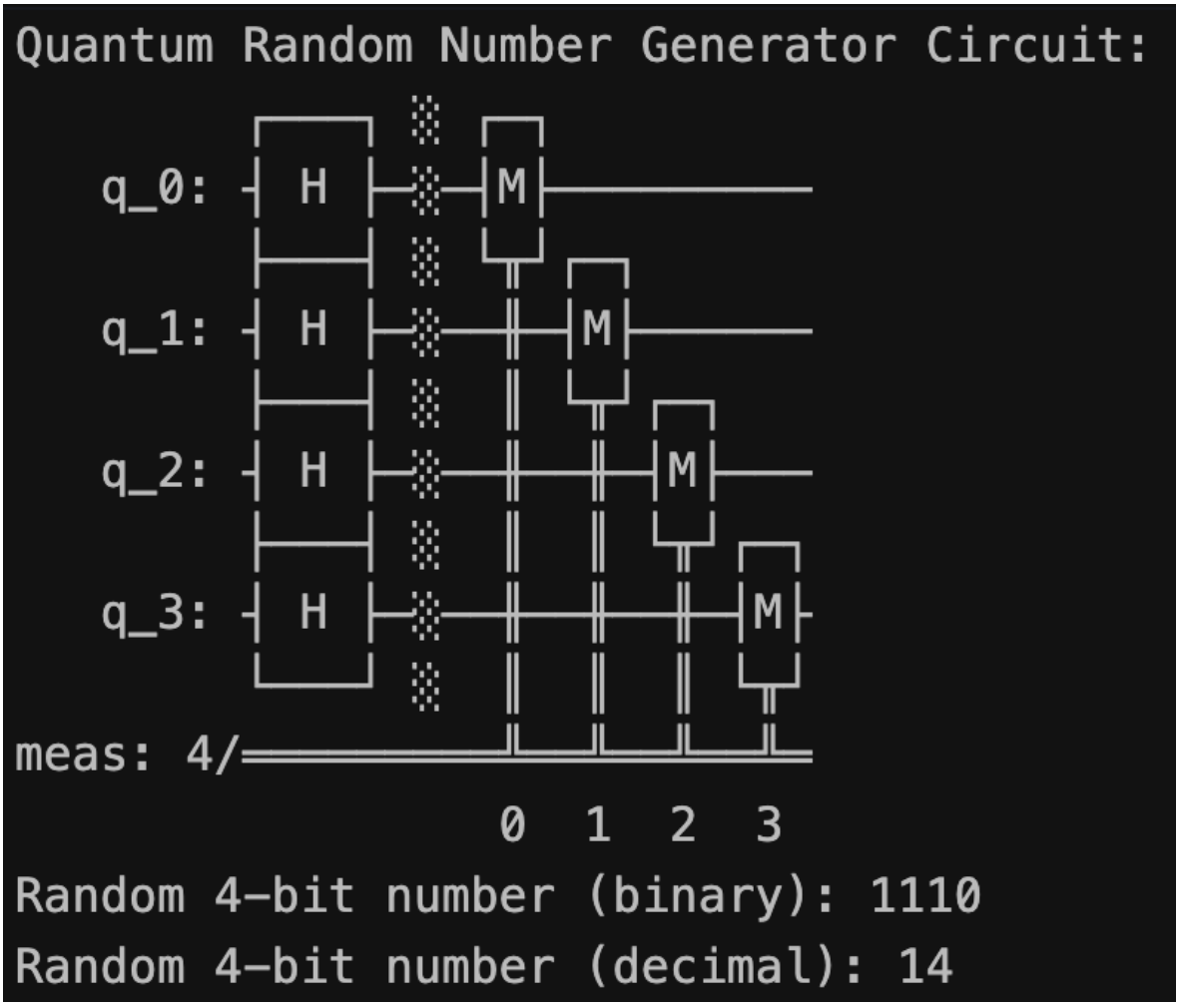
```
from qiskit import QuantumCircuit, Aer, execute
import numpy as np

num_qubits = 4
qc = QuantumCircuit(num_qubits)
# Apply a Hadamard gate to each qubit to create superposition
for qubit in range(num_qubits):
    qc.h(qubit)
qc.measure_all()
print("Quantum Random Number Generator Circuit:")
print(qc.draw())
# Simulate the circuit
simulator = Aer.get_backend('qasm_simulator')
job = execute(qc, simulator, shots=1)
result = job.result()
counts = result.get_counts(qc)
# Binary format
random_number = list(counts.keys())[0]
print("Random 4-bit number (binary):", random_number)
# Binary to integer
random_number_int = int(random_number, 2)
print("Random 4-bit number (decimal):", random_number_int)
```

The above steps outline a simple Quantum Random Number Generator using 4 qubits in superposition. The randomness arises from the quantum properties of superposition and measurement, ensuring unpredictable outcomes suitable for applications like cryptography.

Output:

After running the code, we get our random number, as below:



▼ Current Difficulties: Quantum Errors and Correction

▼ Importance

Although quantum computing has fascinating capacity to speed up algorithms, its use remains limited because of the inaccuracy of the gate operations and measurement, as well as the tendency of the system to go to the thermal state. Due to these phenomenon, the final state of qubits after some operation or even after some idle time, may be different than what is required for the algorithm being run. For many quantum algorithms, this can cause failure instantly.

There are 3 main ways of mitigating this issue

1. Better implementation of qubits. This is a physics problem largely.
2. Reducing the number of operations performed. This can be done by
    a. Better Quantum Algorithms
    b. Quantum Circuit Optimisation.
3. Quantum Error correction.

Although there has been progress in Quantum Error Correction, the current number of qubits on quantum computers are not sufficient to be able to build a “logical qubit” with low enough error rate to run most Quantum Algorithms reliably. [33]

Thus, searching for better methods of quantum error correction has become important.

▼ General Scheme

Most quantum algorithms are designed considering the gate operations to be perfect, and that the qubit does not deviate from its state over time when kept idle. Such a qubit is a mathematical construct, and is called a “logical qubit”. Practically, this term is used for a group of n qubits which together represent the state of only k of those qubits with some redundancy, where k is considerable smaller than n . For example, for the popular 3 qubit bit flip code, we have $n = 3$ and $k = 1$.

Most of the times, the errors that a n qubit state accumulates can be represented completely using the Pauli operators X and Z , which correspond to flipping the qubit state and a sign change respectively. [32,33]

▼ Popular Methods

1. Bit Flip Code
   This is analogous to repetition code in classical computing, except that due to the no-cloning theorem, we cannot simply copy the qubits states, and instead rely of entanglement.
2. Sign Flip Code

   Using Hadamard gates, we can go back and forth between the basis $\{|0\rangle, |1\rangle\}$ and $\{|+\rangle, |-\rangle\}$, and thus, represent sign filps as bit flips. This allows us to use the Bit Flip Code here as well.
3. Linear Codes and Syndrome Extraction.

   In classical computing, linear codes, such as Hamming codes for example project a k bit string x into a subspace C of ${\mathbb{Z}_2}^n$ using a matrix $G \in {\mathbb{Z}_2}^{n \times k}$ called the generator matrix as $x' = G^T x$. This longer bit string is now operated on, which may result in error, causing the final value to be x'' , which is $wt(e)$ bit flips away from the correct value x' , where $e = x'' - x'$. Then, we use a matrix $H \in {\mathbb{Z}_2}^{d \times n}$ to calculate the “syndrome” for x'' as $s = Hx'' = He$. The syndrome uniquely determines the error e with $wt(e) \leq t$ where t is the threshold for error correction.

   Similar to that, Quantum Error Correction codes, such as Shor code use syndrome analysis to perform error correction for both Bit Flip and Sign Flip errors simultaneously. [32]

▼ Example : Bit Flip Code

The simplest Quantum Error Correction Code is the 3 qubit bit flip code.

For a given state $a|0\rangle + b|1\rangle$ of a qubit q_0, we use the CNOT gates on two other qubits q_1, q_2 in states $|0\rangle$ each, keeping q_0 as the control qubit, to give us the final state as

$$|\psi\rangle = a|000\rangle + b|111\rangle$$

Now, due to some errors, there might be a flip ,i.e. an X operation on any of the qubits with probability p (for a particular qubit). This means that the new state $|\psi'\rangle$ could be any one of these

$$\begin{gathered} a|000\rangle + b|111\rangle \\ a|100\rangle + b|011\rangle \\ a|010\rangle + b|101\rangle \\ a|001\rangle + b|110\rangle \end{gathered}$$

with probabilities $(1-p)^3, p(1-p)^2, p(1-p)^2, p(1-p)^2$.

We don’t consider more than one bit flip since that would be too improbable.

Thus, the probability of error is around $3p(1-p)^2$.

Now, for finding whether or not there was an error in $|\psi'\rangle$ and if there was, then what qubit was flipped, we use 2 spare qubits q_3, q_4 , called the ancilla qubits. This is not part of the code, but rather used for decoding the encoded information.

We perform CNOT on q_3 two times, once with q_0 as the control qubit, and then, with q_1 as the control qubit.

Similarly, we perform CNOT on q_4 using q_1, q_2 as control qubits.

Then, the state of the full system (including q_3, q_4) is one of these values

$$\begin{gathered} a|000\ 00\rangle + b|111\ 00\rangle \\ a|100\ 10\rangle + b|011\ 10\rangle \\ a|010\ 11\rangle + b|101\ 11\rangle \\ a|001\ 01\rangle + b|110\ 01\rangle \end{gathered}$$

with probabilities $(1-p)^3, p(1-p)^2, p(1-p)^2, p(1-p)^2$ respectively.

Next, we can simply measure the ancilla qubits. This results doesn't change the quantum state in any of the cases, because both the pure states have the same values for the ancilla qubits, thus revealing no information about other qubits due to the measurement.

This gives us the measured values 00 , 10, 11, and 01 for the different cases, in order.

The value 00 corresponds to no error , the values 10,11 and 01 tell us that q_0, q_1, q_2 have been flipped , respectively.

Then, we can correct the error by applying an X gate where the flip occurred.

The only scenario where this scheme fails is when there are more than 1 flip. This happens with a probability of around $3p^2(1-p)$, which is the probability of 2 flips, and is the significant portion for very small p (on the order of 10^{-2})
This is almost
$1/p$ (on the order of 10^2) times lower than the probability that we say for errors, when there was no correction mechanism in place. [33]

▼ Our own Quantum Assembly Language and Simulator

We have also developed our own Quantum Assembly Language and Simulator.

The documentation can be accessed from the page: https://jaidevsk.github.io/QSL.html

OR the link:

QSL

QSL is a low level assembly language for quantum computers. It is designed to be a simple language that can be used to write quantum circuits and classical logic. The language is designed to be simple and easy to understand, and is meant to be used as a stepping stone to more complex quantum programming languages.

https://jaidevsk.github.io/QSL.html

The documentation pdf is also attached to the end of this report.

▼ QSL Simulator Demonstration

The link to the demonstration video is: Demonstration.mp4

OR

https://iitgnacin-my.sharepoint.com/:v:/g/personal/22110103_iitgn_ac_in/EUYWUNsE3EpNlVzOTN7YDyUBZO7pDT5S4aw8uuqPQcoKWg?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpY

▼ Key Results

We first started with the understanding on Quantum Computers, Quantum Information, Quantum Abstract Machines and followed it with a comparison between Bits and Qubits. After this, we explored the functioning of basic operations and gates in the Quantum and Classical frameworks and compared them. Along with this, we also explored the auxiliary circuits that are required by the quantum computers in order to perform classical operations along with higher-level circuits such as Adder, Subtractor, Multiplexer, etc. Importantly, we also performed a detailed analysis of some of the Quantum ISAs, namely QSL (Developed by us) and QUASAR ISA for Quantum Computing. We also performed a detailed comparison of these Quantum ISAs wit MIPS ISA in terms of Operation Repertoire, Data Types, Registers, Instruction Format and Addressing Modes. We then explored the Hardware-Software interface of Quantum Computers along with the main necessities for Quantum Hardware. In total, we explored two views of hardware-software interfaces in Quantum Computers. Apart from this, we also explored the differences between the classical and the quantum Hardware Software interfaces. We then understood another aspect of Quantum Computers: Their physical implementation. We explored different methods of physically implementing the Qubits along with the ways of implementing single and multi-qubit operations governing their topologies. We also compared this with physical implementation of classical bits and computers. After this, we studied various methods of performing benchmarking in quantum computers. Apart from this, we also performed a comparison of these benchmarks with the classical benchmarks as well. We also studied the comparison of the performance of various operations on Quantum Computers and on Classical Computers and discussed which one is better. We also performed a detailed analysis on the quantum algorithms such as Shor's Algorithm, Deustch's Algorithm and Quantum Random Number Generators. Then we understood the current difficulties in Quantum Computing- Errors and Correction: We also studied some of the prominent issues that are preventing the use of Quantum Computing on a large scale, namely Errors. We also performed a detailed study on some of the methods used in error correction to mitigate these issues. Finally coming to the simulation section, as mentioned in our initial aims, we developed our own ISA for Quantum Computers (called QSL: Quantum Simulation Language). The detailed documentation of the ISA is also provided at the end of this report. We also developed a Simulator which executes the programs written in this ISA as well as provides a step by step execution for the user to gain an understanding of the evolution of the states of the qubits along with the memory and registers. The details of this are provided in the demonstration video. We also created benchmark programs in our QSL based on OpenQASM benchmarks for the users to execute them

as well as a Benchmark feature in the simulator that would enable the user to benchmark the hardware on which simulator is being run in terms of the CLOPS rating as well as the SPEC type rating that we developed with respect to the IBM Eagle R3 processor (accessed through IBM Kyiv server).

There are some more topics that we explored (which are beyond the scope of this project) such as Quantum Machine Learning, etc.

▼ Challenges and Open Issues

One of the major challenges faced by us was due the unavailability of the detailed information on the architecture and organisation of Quantum Computers. This may be because of the fact that there is a very tough competition between various companies in this sector. That is why this information is not available freely. We also faced some difficulties in executing the “actual” quantum benchmark programs on our machines since the large number of states are beyond the computational limits of our classical devices. Therefore, certain smaller example programs were used for benchmarking (in the simulations section).

▼ Member Contributions

Jaidev

- Hardware Software Interface
- Physical Implementation
- Quantum Benchmarking
- Development of the Quantum Assembly Language (QSL) and the Simulator

Pranav

- Introduction
- Current Difficulties: Quantum Errors and Correction
- Development of the Quantum Assembly Language (QSL) and the Simulator
- Extras: Quantum Machine Learning.

Vannsh

- Introduction
- ISA Comparison
- Basic Operations and Gates

John

- Auxiliary Quantum Circuits to perform Classical Operations using Quantum Gates
- Comparison of Operations on which Quantum is better, and the Classical is better
- Better Algorithms in Quantum Computers than Classical Computers

▼ Summary and Takeaways

On the whole, this project was a great source for us to delve into the fast emerging field of Quantum Computing. Through this project, we got the opportunity to explore various concepts of Computer organisation and architecture in Quantum Computing domain as well as observe the variation as we move from the Classical framework to Quantum computing. We could also get a good experience in developing ISA and Simulator in this course, that too for Quantum Computers.

▼ Future Prospects

Since we have already explored the organisation as well as the architecture of Quantum computers along with the development of a Quantum Assembly language, we aim to take this assembly language forward along with its simulator to the Compilers course. In the future, we aim to increase the scale of the quantum as well as classical operations (with quantum gates) as well as design a compiler that would compile the code from High Level languages to our own Quantum Assembly Language (QSL), which will be executed on our simulator.

▼ References

[1] DiVincenzo, D.P. (2000), The Physical Implementation of Quantum Computation. Fortschr. Phys., 48: 771-783. https://doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E

[2] https://spectrum.ieee.org/heres-a-blueprint-for-a-practical-quantum-computer

[3] https://nap.nationalacademies.org/read/25196/chapter/7#114

[4] Ugwuishiwu, Chikodili & Ayegbusi, Oluwole. (2021). Towards physical implementation of Quantum Computation. Advances in Computer Science and Engineering. 10.30534/ijatcse/2020/147952020

[5] https://quantumtheory-bruder.physik.unibas.ch/fileadmin/user_upload/quantumtheory_bruder_physik/People/Martin_Koppenhoefer/Quantum_Computing_and_

[6] Dejpasand, M.T.; Sasani Ghamsari, M. Research Trends in Quantum Computers by Focusing on Qubits as Their Building Blocks. Quantum Rep. 2023, 5, 597–608. https://doi.org/10.3390/quantum5030039

[7] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15). Association for Computing Machinery, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[8] Wang, J.; Guo, G.; Shan, Z. SoK: Benchmarking the Performance of a Quantum Computer. Entropy 2022, 24, 1467. https://doi.org/10.3390/e24101467

[9] https://docs.quantum.ibm.com/api/qiskit/circuit_library

[10] https://ieeexplore.ieee.org/abstract/document/9325421

[11] https://learning.quantum.ibm.com

[12] https://www.youtube.com/watch?v=42OiBzfdE2o&list=PLOFEBzvs-VvqKKMXX4vbi4EB1uaErFMSO&index=1

[13] https://medium.com/analytics-vidhya/quantum-gates-7fe83817b684

[14] https://www.youtube.com/watch?v=8J-H7gbDCos&list=PL4wzlfHhrqQzJfrxDv2nYmLwvDBZmPb9-

[15] Quantum Computing for Computer Scientists by Noson S. Yanofsky, Mirco A. Mannucci

[16] Quantum Computation And Quantum Information Nielsen Chuang

[17] https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://iontrap.umd.edu/wp-content/uploads/2016/01/Quantum-Gates-c2.pdf&ved=2ahUKEwi1jsnt4NOJAxU7S2cHHfG1M9YQFnoECDEQAQ&usg=AOvVaw1-UQDCs6YFpM0vBCa_NSR0

[18] https://docs.quantum.ibm.com/guides

[19] https://people.eecs.berkeley.edu/~vazirani/f04quantum/notes/lec9.pdf

[20] https://www.cs.cmu.edu/~odonnell/quantum15/lecture09.pdf

[21] https://www.qi.damtp.cam.ac.uk/files/QIC-12.pdf

[22] https://quantumcomputinguk.org/tutorials/16-qubit-random-number-generator

[23] Python 3 https://www.python.org/

[24] TKinter https://docs.python.org/3/library/tkinter.html

[25] A Practical Quantum Instruction Set Architecture

[26] Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights

[27] Introduction to quantum mechanics / David J. Griffiths (link)

[28] Rigetti, The Lifecycle of a Program (link)

[29] Lecture Notes, ES215, Computer Organisation and Architecture, IIT Gandhinagar

[30] Wack, Andrew & Paik, Hanhee & Javadi-Abhari, Ali & Jurcevic, Petar & Faro, Ismael & Gambetta, Jay & Johnson, Blake. (2021). Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers. 10.48550/arXiv.2110.14108.

[31] S. Stein, N. Wiebe, J. Ang and A. Li, "Benchmarking Quantum Processor Performance through Quantum Distance Metrics Over An Algorithm Suite," 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lyon, France, 2022, pp. 618-624, doi: 10.1109/IPDPSW55747.2022.00106.

[32] Quantum error correction, *Wikipedia*

[33] A Tutorial on Quantum Error Correction, *Andrew M. Steane*

Quantum Simulator Language

Contents

Description

QSL is a low level assembly language for quantum computers. It is designed to be a simple language that can be used to write quantum circuits and classical logic. The language is designed to be simple and easy to understand, and is meant to be used as a stepping stone to more complex quantum programming languages.

This language was first developed by the Quantum Innovators group as a course project in the Computer Organisation and Architecture Course at IIT Gandhinagar, India.

This language is also accompanied by a simulator, whose detailes will be discussed in the later sections.

Instructions

QSL has a set of instructions that can be used to write quantum programs and classical programs. The basic instructions are divided into several categories:

- Quantum Operations (Q type)
- Classical Logical Operations (R type)
- Control Flow instructions
- Data transfer instructions
- Special Instructions
- Comments
- Labels

Quantum Operations (Q type)

Quantum operations are used to manipulate qubits in a quantum circuit. The following quantum operations are supported:

- RESET(q) : Collapses the wave-function so that qubit number q goes to the pure state |0>
- H(q) : Hadamard gate on qubit number q
- T(q) : T gate applied on qubit number q
- X(q) : Pauli-X gate on qubit number q
- Y(q) : Pauli-Y gate on qubit number q
- Z(q) : Pauli-Z gate on qubit number q
- S(q) : S gate on qubit number q
- MEASURE(q) : Collapses wavefunction so that qubit number q enters a pure state, and stores the result (0 or 1) in the 0 index bit of register 0 (or s0)
- CNOT(qs,qt) : Applies CNOT gate on qubits qs and qt where qt is the control qubit, and qs is the controlled qubit (one that changes its state).
- CZ(qs,qt) : Controlled Z gate on qs , with qt as control qubit.

- SWAP(qs,qt) : Swaps the states of qubits qs and qt .

Classical Logical Operations (R type)

Classical logical operations are used to manipulate classical bits in a classical circuit. The following classical logical operations are supported:

- AND(rs,sb,rt,tb) : Stores the result of rs[sb] AND rt[tb] in bit rs[sb] .
- IOR(rs,sb,rt,tb) : Stores the result of rs[sb] OR rt[tb] in bit rs[sb] .
- XOR(rs,sb,rt,tb) : Stores the result of rs[sb] XOR rt[tb] in bit rs[sb] .
- NOT(rs,sb) : Stores the result of NOT rs[sb] in bit rs[sb] .
- NOR(rs,sb,rt,tb) : Stores the result of NOR rs[sb] OR rt[tb] in bit rs[sb] .
- MOV(rs,sb,rt,tb) : Stores the value of rs[sb] in bit rt[st] .
- SET(rs,sb) : Sets the bit rs[sb] to 1
- CLR(rs,sb) : Sets the bit rs[sb] to 0

Control Flow instructions

Control flow instructions are used to control the flow of the program. The following control flow instructions are supported:

- JPN(rs,sb,label) : Jumps to instruction labelled label if rs[sb] has value 0
- JPP(rs,sb,label) : Jumps to instruction labelled label if rs[sb] has value 1
- JMP(label) : Jumps to instruction labelled label

Data transfer instructions

Data transfer instructions are used to transfer data between registers and memory. The following data transfer instructions are supported:

- LDB(rs,sb,rt,imm) : Let x be the unsigned value of register rt added to the value of the 6 bit field imm, interpreted as a signed number . Then this instruction sets rs[sb] to the value of bit x of data memory .
- STB(rs,sb,rt,imm) : Similar to LDB, except bit x of data memory is set to the value of rs[sb] .

Special Instructions

Special instructions are used for special purposes. The following special instructions are supported:

- NOP() : Does nothing
- HLT() : Stops (Halts) the program. This is automatically added at the end.

Comments

On every line, everything after the first '#' character, including the '#' character itself, is disregarded during compilation.

Example:

```
MOV(0,0,1,1) # This text is going to be disregarded
```

Labels

Everything before the first ':' character , after discarding the comment is the first label. Discard that (including ':') and repeat the process to get the second label, and so on.

Example:

```
label1:NOP()
label2:label3:NOP()
```

Processor Design for Simulating

The processor that we have designed this ISA for is a quantum processor with the following specifications:

- Number of Qubits: 5
- Possible Number of Quantum States: 32
- Qubit Topology: Fully Connected
- Number of Classical Registers: 7
- Number of Buffer Registers: 1
- Number of Addressable Primary Memory Locations: 24
- Size of Classical Registers: 32 Bits
- Size of Buffer Register: 32 Bits
- Size of Memory Word: 32 Bits
- Maximum Number of Instruction: 1024 (in 2024 version)
- Size of Instructions: 18 Bits
- Primary Data Memory: 24*32 Bits = 96 Bytes

Other Points

- The Buffer Register is the place the Measurement value will be stored. It is indexed as $0, the 0th register.
- The Measurement by default will be stored at its 0th bit.
- This follows the Harvard Type Architecture, that is, the Data Memory is separate from the instruction Memory.
- Here, the register file also acts as a bit addressable data memory (as of 2024 Version 1).
- There are 8 actual registers ($0,$1…$7) which are addressable using fields rs and rt.
- The primary memory, which is reprsented as “pseudo” registers ($8 to $31) which are accessible only using LDB and STB instructions. We are calling them “registers” just for our ease, but each of them is just a block of 32 bits in the data memory and nothing more.

Machine Code

The Machine Code is of 18 bits in total. Since the instructions are divided based on their types, the machine codes and their formats are also based on that. The opcode, which corresponds to the first three bits of tthe instruction machine code decides whether the instruction is Q-type, R-type, etc. The next 15 bits depend on the type of instruction for their format. We use Big Endian Notation, that is, the MSB is on left. The table for the types of instructions with their respective opcodes is as given below:

Instruction Type	Opcode - first 3 bits
Q	000
R (classical)	001
JPN	010
JPP	011

JMP	100
Load	101
Store	110
NOP/HLT	111

Q Type Instructions

For Q Type Instructions, the instruction is of 18 bits. The first 3 bits are the opcode, the next 3 bits are the qubit number, the next 3 bits are 000, the next 3 bits are the target qubit number, and the last 6 bits are the function code. The table for the function codes is as given below:

Part of Machine Code	**Opcode**	**qs**	**000**	**qt**	**Function Code**
Number of Bits Taken	3 Bits	3 Bits	3 Bits	3 Bits	6 Bits

The meanings of the individual parts is as follows:

- Opcode: The first 3 bits of the instruction, which is 000 for Q type instructions.
- qs: The first qubit number on which the operation is to be performed.
- 000: A 3 bit field which is always 000.
- qt: The second qubit number for the operation. It is 000 for single qubit operations.
- Function Code: The 6 bit function code for the operation.

The table for the function codes is as given below:

Instruction	**6 Bit Function Code**
RESET	000000
H	000001
T	000010
X	000011
Y	000100
Z	000101
S	000110
MEASURE	000111
CNOT	001000

CZ	001001
SWAP	001010

R Type Instructions

For R Type Instructions, the instruction is of 18 bits. The first 3 bits are the opcode, the next 3 bits are the register number, the next 3 bits are the bit number, the next 3 bits are the target register number, the next 3 bits are the target bit number, and the last 3 bits are the function code. The table for the function codes is as given below:

Part of Machine Code	Opcode	rs	sb	rt	tb	Function Code
Number of Bits Taken	3 Bits	3 Bits	3 Bits	3 Bits	3 Bits	3 Bits

The meanings of the individual parts is as follows:

- Opcode: The first 3 bits of the instruction, which is 001 for R type instructions.
- rs: The first register number on which the operation is to be performed.
- sb: The bit number in the first register on which the operation is to be performed.
- rt: The second register number for the operation.
- tb: The bit number in the second register on which the operation is to be performed.
- Function Code: The 3 bit function code for the operation.

The table for the function codes is as given below:

Instruction	3 Bit Function Code
AND	000
IOR	001
XOR	010
NOT	011
NOR	100
MOV	101
SET	110
CLR	111

Control Flow Instructions

For Control Flow Instructions, the instruction is of 18 bits. The first 3 bits are the opcode, the next 3 bits are the register number, the next 3 bits are the bit number, and the last 9 bits are the label. The table for

the function codes is as given below:

Part of Machine Code	Opcode	rs	sb	imm
Number of Bits Taken	3 Bits	3 Bits	3 Bits	9 Bits

The meanings of the individual parts is as follows:

- Opcode: The first 3 bits of the instruction, which is 010 for JPN type instructions, 011 for JPP type instructions, and 100 for JMP type instructions.
- rs: The register number on which the operation is to be performed.
- sb: The bit number in the register on which the operation is to be performed.
- imm: The label (immediate) to which the control flow is to be transferred.

Data Transfer Instructions

For Data Transfer Instructions, the instruction is of 18 bits. The first 3 bits are the opcode, the next 3 bits are the register number, the next 3 bits are the bit number, the next 3 bits are the target register number, the next 3 bits are the target bit number, and the last 3 bits are the immediate value. The table for the function codes is as given below:

Part of Machine Code	Opcode	rs	sb	rt	tb	imm
Number of Bits Taken	3 Bits	3 Bits	3 Bits	3 Bits	3 Bits	3 Bits

The meanings of the individual parts is as follows:

- Opcode: The first 3 bits of the instruction, which is 101 for Load type instructions and 110 for Store type instructions.
- rs: The register number from which the data is to be loaded or stored.
- sb: The bit number in the register from which the data is to be loaded or stored.
- rt: The register number to which the data is to be loaded or stored.
- tb: The bit number in the register to which the data is to be loaded or stored.
- imm: The immediate value to be loaded or stored.

Special Instructions

For Special Instructions, the instruction is of 18 bits. The first 3 bits are the opcode, and the next 15 bits are Function Code. The table for the function codes is as given below:

Part of Machine Code	Opcode	Function Code
Number of Bits Taken	3 Bits	15 Bits

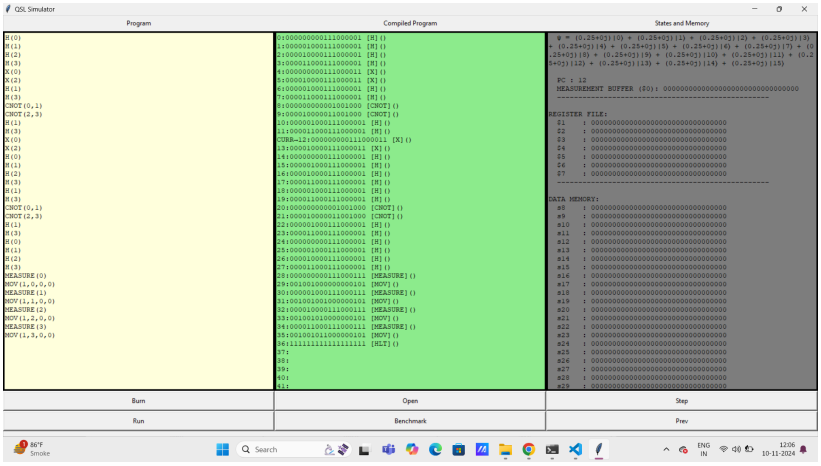
The meanings of the individual parts is as follows:

- Opcode: The first 3 bits of the instruction, which is 111 for NOP and HLT type instructions.
- Function Code: The 15 bit function code for the operation.

Instruction	Function Code
NOP	000000000000000
HLT	111111111111111

Simulator

We have also developed a simulator for the QSL Language that allows the user to write programs in QSL assembly language and visualise the execution with respect to the variation in the quantum states, memory and the sequence of instructions executed.

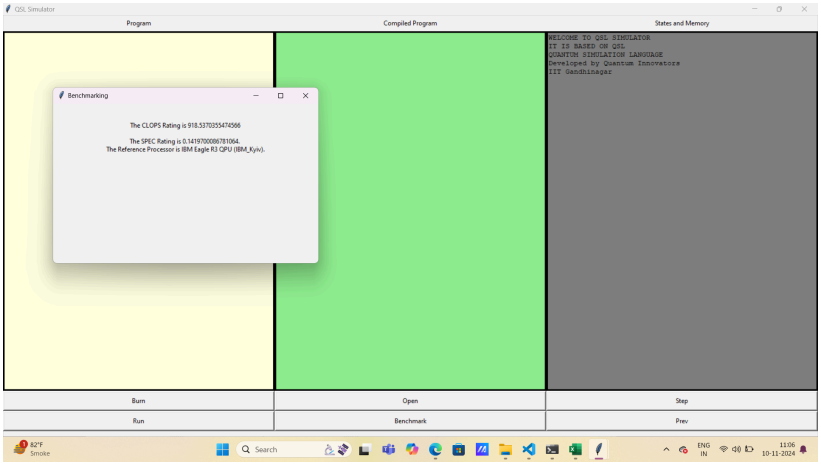


The Simulator is a desktop based application made using Python (Tkinter). The application can be downloaded by following the instructions provided in the Downloads section. Some of the main features of the simulator application are:

- PROGRAM Window: It is the coding window, present on the leftmost side of the simulator. It allows the user to write the assembly program in the QSL language. Apart from the main code, it also accepts comments and labels, which simplify the execution.
- COMPILED PROGRAM Window: After the user executes the program using the Burn button, the simulator takes the assembly code and converts it into the machine code (in binary format). This binary code is displayed in the compiled program window. It also consists of a pointer "CURR->", which displays the current instruction that is being executed. This is an important feature as it enables the user to keep a track of the instruction that is currently under execution.
- STATES & MEMORY Window: This window displays the state vector and the memory of the quantum processor. The state vector (32 states) represents the quantum states of the qubits. The memory consists of a 32-bit vectors that represent the classical memory of the processor. The simulator allows the user to view the changes in the state vector and memory after each instruction execution. It consists of the following information:
    - State Vector ψ: It is the vector that represents the state of the qubits. Since we have a 5-qubit system, this vector is of size 32.
    - PC: This is the Program Counter, it displays the index of the instruction that is currently under execution.
    - IR: Since we display the "CURR->" pointer in the COMPILED PROGRAM Window, we need not show the intruction register here again.
    - MEASUREMENT BUFFER: It displays the state of the Measurement Buffer, where the measurement value will be stored.
    - REGISTER FILE

    - DATA MEMORY
    - INSTRUCTION MEMORY
- BURN Button: This button is necessary to execute the program and generate a step by step trace of the program execution.
- STEP Button: This button steps to the next instruction. It is useful for the user to visualise a step by step execution. In simple words, STEP is the Next Instruction. It is used after BURN button.
- PREV Button: This button moves to the previous instruction. By pressing this, the user will be able to move the data memory, register files and the quantum states also to the previous states. It is also used after BURN button.
- RUN Button: This button is used to directly run the program to the end, and it displays the final output. The execution trace of the program is not stored in this. It is useful for directly moving to the final state in long programs.
- OPEN Button: This button is meant for opening the Assembly Code files. The Assembly Code files for QSL have to be stored and opened in .qsl format.
- BENCHMARK Button: This button is used to generate the benchmark ratings of the simulator on the user's hardware in some standardised metrics. The exact details are given in the Benchmarking section.

Benchmarks

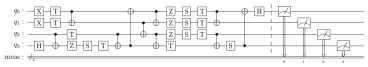


In order to provide the user an idea of the performance of the simulator on their hardware with respect to the actual quantum hardware, we have provided a benchmarking feature in the simulator. Since we are executing this simulator on a classical machine, the "typical" quantum metrics of evaluation such as Fidelity, Error Rate, etc. are not applicable. Hence, we have devised some execution time centric metrics that are more relevant to the classical machine on which the simulator is running. The metrics are as follows:

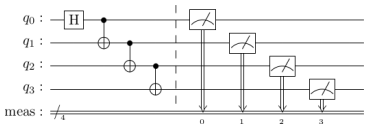
- CLOPS Rating: This provides the user the CLOPS rating of the simulator running on the classical hardware. More information on CLOPS Rating can be found from its original paper "Scale, Quality, and Speed: three key attributes to measure the performance of near-term quantum computers". The CLOPS rating is calculated as the number of CLOPS (Classical Logical Operations per Second) that the simulator can perform on the user's hardware. The CLOPS rating is calculated as the number of CLOPS that the simulator can perform in a second. The higher the CLOPS rating, the better the performance of the simulator on the user's hardware. We have used the parameters as Depth (Number of QV layers) D=100, Number of Qubits N=5, Number of Shots S=10, Number of templates M=10, and Number of parameters updated K=1. The CLOPS rating is calculated as: CLOPS = D * S * M * K / time_taken.
- SPEC- Rating: This rating is developed by us based on some smaller versions of Benchmark programs available for OpenQASM in our own QSL Assembly Language. These programs were executed on IBM Eagle R3 machine (Quantum Processor) and then, based on the execution time, a SPEC style rating was calculated (which is the Geomatric Mean of all the speedup). The codes for the benchmark are also provided in the project repository in .qsl format.

Some of the benchmark programs currently included along with their circuit diagrams are as presented below, The diagrams are made using IBM Qiskit software:

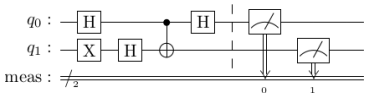
- ADDER.qsl:



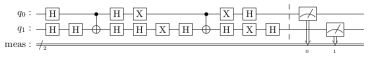
- CAT.qsl:



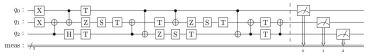
- DEUSTCH.qsl:



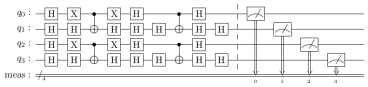
- GROVER.qsl:



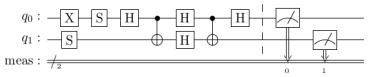
- FREDKIN.qsl



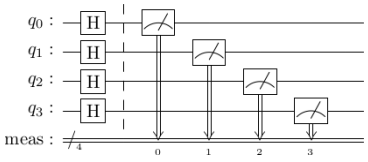
- HS4.qsl:



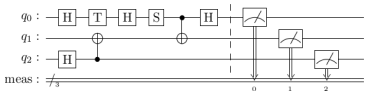
- ISWAP.qsl:



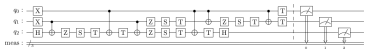
- QRNG.qsl:



- TELEPORTATION.qsl:



- TOFFOLI.qsl:



Downloads

To Download this code for the Simulator, run the following command in your terminal:

```
git clone https://github.com/JaidevSK/QSL.git
```

To install the pip based dependencies in requirements.txt file, run the command:

```
pip install -r requirements.txt
```

After cloning the repository, you can run the simulator by running the following command:

```
python3 SIMULATOR.py
```

For Downloading this user manual in PDF format, click the following button:

Download PDF