T.C.
BAHÇEŞEHİR UNIVERSITY

FACULTY OF ENGINEERING AND NATURAL SCIENCES

CAPSTONE PROPOSAL FINAL REPORT

QUANTUM COMPUTING

Ege Akman – 1729970 (Computer Engineering)

Emir Tolga Demirel – 1730198 (Electrical & Electronics Engineering)

Iren Su Dulger – 1804907 (Electrical & Electronics Engineering)

Iren Su Dulger – 1804907 (Computer Engineering)

Kaan Kocaturk – 1728808 (Computer Engineering)

Esat Canberk Ozcelik – 1805100 (Electrical & Electronics Engineering)

Advisors:

Prof. Seref Kalem (Electrical & Electronics Engineering)

Assist. Prof. Ece Gelal Soyak (Computer Engineering)

ISTANBUL, May 2021

STUDENT DECLARATION

By submitting this report, as partial fulfillment of the requirements of the Capstone course, the students promise on penalty of failure of the course that

- they have given credit to and declared (by citation), any work that is not their own (e.g. parts of the report that is copied/pasted from the Internet, design or construction performed by another person, etc.);
- they have not received unpermitted aid for the project design, construction, report or presentation;
- they have not falsely assigned credit for work to another student in the group, and not take credit for work done by another student in the group.

ABSTRACT

QUANTUM COMPUTING

Ege Akman – 1729970 (Computer Engineering)

Emir Tolga Demirel – 1730198 (Electrical & Electronics Engineering)

Iren Su Dulger – 1804907 (Electrical & Electronics Engineering)

Iren Su Dulger – 1804907 (Computer Engineering)

Kaan Kocaturk – 1728808 (Computer Engineering)

Esat Canberk Ozcelik – 1805100 (Electrical & Electronics Engineering)

Faculty of Engineering and Natural Sciences

Advisor:
Prof. Seref Kalem (Electrical & Electronics Engineering)
Assist. Prof. Ece Gelal Soyak (Computer Engineering)

May 2021

The major point of this project is to get a better understanding of quantum computing, primarily superposition and entanglement. The process will start with connecting Raspberry Pi to IBM's Quantum Computer to simulate the behavior of qubits with Qiskit programming language. We will show the superposition and entanglement principles with Qiskit, and proceed with more complex simulations that also implement quantum computing such as random number generator and qubit reduction with Variational Monte Carlo. The success of our project is determined by the management of IBM connection and the correct implementation of Qubit codes.

Key Words: Quantum, Qubit, Qiskit, Raspberry Pi, IBM, Quantum Computing, Quantum Mechanics, QRasp

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IBM International Business Machines

Qubit Quantum Bit

IEEE The Institute of Electrical and Electronics Engineers

1. OVERVIEW

This project is an implementation of various concepts to understand quantum computing. We will be trying to build three concepts. The first one is creating superposition and entanglement using two qubits. After that, we will measure the results. The expected outcome for this one of states from 00 or 11. When we understand the concept of quantum computing and finished the first part, we will start to build more complex simulation which is quantum random number generation. If we successfully finish the random number generation, we will try to implement Variational Monte Carlo to reduce the number of qubits.

1.1. Identification of the need

In this project, we will try to implement the some algorithms and applications of the quantum computing. To impelement algorithms and applications we need to understand the concept of quantum computing. Quantum computing can be divided into two parts as theoretical and practical implementation. Firstly, we need to understand the theoritical part of the algorithms and applications, this includes complex mathematics and linear algebra. Second part which is the practical part is implementation of the quantum circuits to compare the results with the theoretical part. In the bottom of the circuits, they are unitary matrixes which is used to process on qubits. Due to fact that we can't build our own quantum computer we will use the raspberry pi to simulate. When raspberry pi computational power is not enough, it is possible to use IBM's quantum computer by applying through IBM system.

1.2. Definition of the problem

Monte Carlo simulation gives a random choice for each activity which take place and repeats this process to get a probability distribution. Monte Carlo method is used in optimization, numerical integration in Engineering, Finance and many fields of physics and mathematics. The main constraint of this method is that it requires huge amounts of computational power. Therefore, in classical computing this method is not very efficient to implement. On the other hand, Quantum computers can speed up this process but still there are some measures to be taken into consideration. The number of needed qubits is proportional to the number of random numbers to get the dimension of the integral. High dimensional integrations require high number of random numbers. Quantum computers of today still lack the necessary qubit number. Obligators up to 10^{6} may be needed for computation problems for large portfolios in finance problems as an example whereas largest quantum computer of today has only the qubit number of O(10). Therefore, number of qubits needs to be reduced to be able to perform these processes.

1.2.1. Functional requirements

- Raspberry Pi must be functional. The connection between Raspberry Pi and IBM Quantum Computer must be made.
- Qiskit libraries should be installed. Superposition and entanglement must be shown through Qiskit code.

1.2.2. Performance requirements

The experiments of superposition and entanglement must either be impracticable for standard computers or must execute faster than standard computers.

The codes executed by Qiskit should run in a faster time and must have a higher accuracy rate than that of standard computers if such a comparison can be made.

1.2.3. Constraints

The main constraint is the Raspberry Pi's computational power. Due to lack of Random Access Memory (RAM), number of Qubits that can be used to simulate tasks is limited. Thus, design of the solutions should be carefully designed with inefficacy of RAM in mind.

1.3. Conceptual solutions

We have three conceptual solutions for our topic respectively two qubit superposition and entanglement, random number generation and implementing Variatianol Monte Carlo with random number generation. We will use Raspberry Pi and SenseHAT 8x8 pixel LED to demonstrate our results of the implementations.

First one is creating the superposition of two qubits and entanglement. Expected results is 50% probability of qubits state being 00 or 11. Since qubits are in superposition we cannot know what is the output until their state is measured. We plan to see one of the states on 8x8 pixel LED.

Our second conceptual solution is reducing the number of qubits by building a quantum random number generator. To for implementation on Monte Carlo simulation for the reduction of qubits and thus speeding the process. In order to over come this constraints we will first imply Monte Carlo Simulation techniques and later by implementing pseudo-random number generator (PRNG) estimated average integrand values in order to qunatum circuits for the decrement of the number of qubits required

1.3.1. Literature Review

The smallest unit of data storage in quantum computer is called Qubit as can be seen in Figure 1. Due to qubits are sensitive in nature the information stored in them is unstable [1]. Therefore,

quantum computers such as IBM-Q are operating in very cold temperature and vacuumed place. A qubit can exists in both states simultaneously except logical state 0 or 1. This existance called superposition. During superposition, it is impossible to know qubit's state until it is measured. In other words, a qubit can exits in both states which is called superposition until it is measured. A quntum computer can be considered as a network which combination of a quantum logic gates. Each gate carry out an basic unitary operation on one or more than one qubits [4]. There are lots of gates which be seen in Appendix A.

The simplest quantum logic gate is NOT gate which performs a operation on qubit to change its states. It can be also extended to CNOT and CCNOT. These gates are named respectively Controlled NOT gate and Controlled Controlled NOT gate. They perform the NOT operation on qubit if control qubit is one. There is also important logic gate which is Hadamard gate that creates a superposition. The Hadamard gate will be used in first conceptual solution for our topic [1] [4].

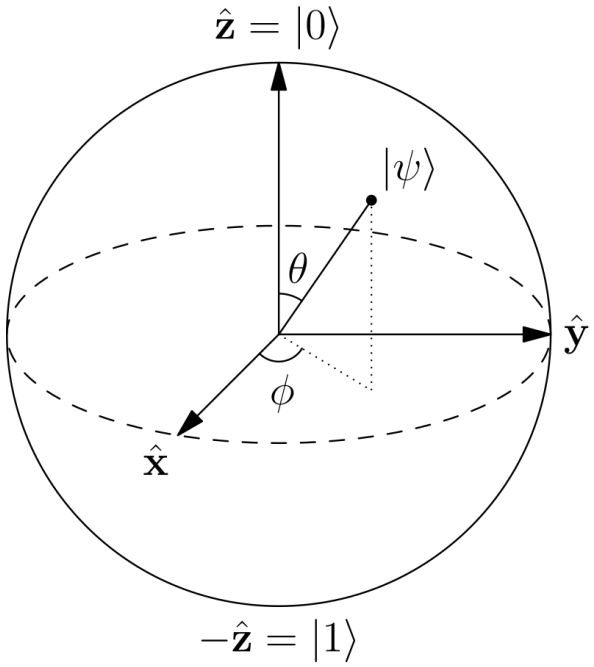


Figure 1. Representation of a Qubit as Bloch Sphere

Monte Carlo Methods

Monte Carlo protocols are a type of computational process that uses repeated random sampling to produce numerical results. The basic idea is to employ randomness to solve problems that are in principle deterministic. They're frequently utilized in physical and mathematical problems, and they're especially valuable when other approaches are difficult or impossible to use. The three main applications of Monte Carlo methods are optimization, numerical integration, and generating

draws from a probability distribution.

Monte Carlo methods are effective for simulating systems with large numbers of associated degrees of freedom, like disorganized elements and fluids, cellular structures and strongly linked solids, in physics problems. Modeling phenomena via inputs with high ambiguity, such as the computation of risk in business and multidimensional definite integrals evaluation with difficult boundary conditions in mathematics, are two further examples. Monte Carlo–based predictions of failure, cost overruns, and schedule overruns are typically better than human intuition or similar "soft" methodologies when applied to systems engineering challenges.

Quantum computers are known to be faster than their conventional counterparts at Monte Carlo simulation. There have already been various proposals for using the quantum algorithm to solve real-world problems, such as quantitative finance. Because many problems in finance where Monte Carlo simulation is applied, problem involves highly high-dimensional integrations, such as risk measurement of credit portfolios, numerous random values are necessary to generate one sample value of the integrand. As a result, in the naive approach, where a quantum register is allocated each random number, the required qubit number is too huge. If a calculation comparable to the classical one, estimating the mean of integrand values sampled by a pseudo-random-number generator (PRNG) executed on a quantum circuit, qubits can be lowered while maintaining quantum speed up. It is possible to offer not only an overview of the concept, but also a practical implementation of PRNG and its application to credit risk measurement. In fact, lowering the number of qubits is a trade-off against increasing circuit depth. As a result, while a complete decrease may be impractical, a trade-off between speed and memory space will be critical in adjusting calculation settings based on machine specs if a quantum computer is utilized to run large-scale Monte Carlo simulations in the upcoming years.

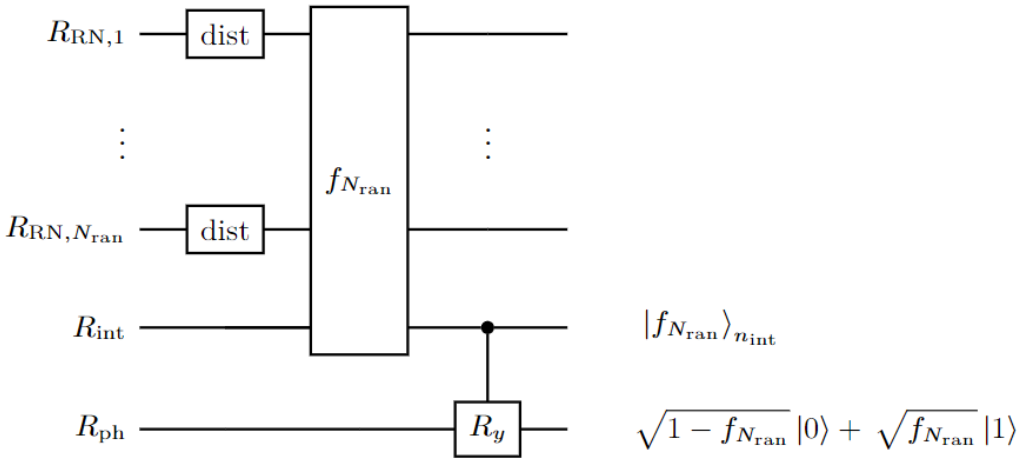


Figure 2. Circuits for the quantum algorithm for Monte Carlo. [5]

1.3.2. Concepts

There are two possible concepts to implement our project. First one is Raspberry Pi and second one is IBM's Quantum Computer.

Raspberry Pi is a small single-board computer which is used for general purposes. We will use it in our project to simulate some quantum computing concepts. However, due to fact that it is still classical computer it has less computational power due to RAM (Random Access Memory) size. It limits the number of qubit that can be used in quantum computing.

On the other hand, IBM Quantum Computer is much more powerfull due to fact that it is a real quantum computer. It can compute more than 100 qubits with the latest developments. Implementing complex quantum computing concepts will require to use IBM's Quantum Computer.

Table 1. Comparison of the two conceptual solutions.

	Raspberry Pi	IBM Quantum Computer
Cost	high	low
Complexity	medium	low
Performance	medium	high
Features	high	high

1.4. Physical architecture

Quantum von Neumann architecture:

A quantum bus system allows quantum information to be moved between the various components of a quantum computer. The quantum arithmetic logic unit (QALU), which is the most

hardware demanding portion of the quantum computer because quantum gate operations are performed here, is where quantum information is manipulated. Quantum data is stored in a quantum memory, which need to depend on multiplexing technology to provide huge storage capacity. In addition, an input and output region serves as an interface to the classical world, where the qubit state can be detected or started.

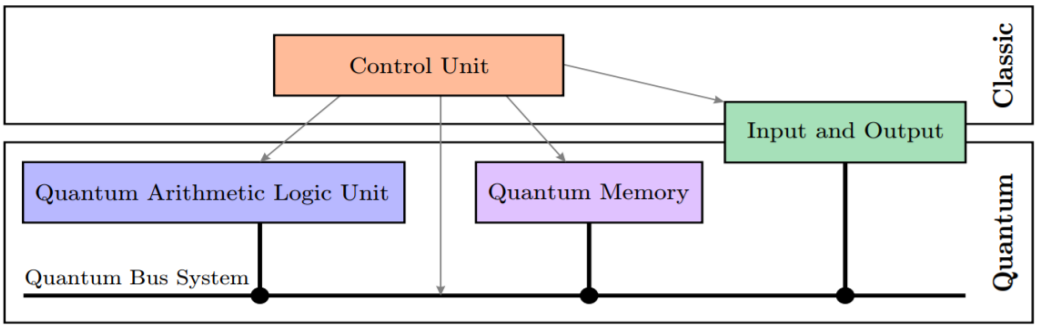


Figure 3. The Quantum von Neumann architecture. [1]

The quantum von Neumann architecture is based on the same concepts as the classical version. A quantum von Neumann machine conducts a sequence of quantum gate operations by loading the qubits to be manipulated into the QALU's quantum registers. The quantum register length can be as long as you want it to be. To entangle the two qubits from arbitrary positions in the memory, they are loaded from the quantum memory into the QALU, where gate operations are performed. After the quantum gate operations, the qubits can either stay in the QALU for extra processing or be placed back into the quantum memory. To detect quantum states, the appropriate quantum information can be transmitted to an output, or detecting, region. It is possible to execute QIP in the QALU and detection in the output region at the same time since this detection zone is independent from the QALU.

After detecting the quantum state, the qubits can be relocated to an input region to be initialized into a single state before being returned to the quantum memory (now initialized). If a more complicated starting state is required, such as an entangled state, the initialized qubits can be placed in the QALU, which performs quantum gate operations to generate the desired first/inital quantum state.

Quantum Memory Region:

Field effect transistor (FET) and a capacitor, as shown in figure a, are the components of a dynamic RAM (DRAM) needed to store one bit of information in a traditional computer. The FET used to access the DRAM cell is controlled by digital multiplexing logic. Because of the minimal hardware demand per bit, DRAM chips can store a lot of data.

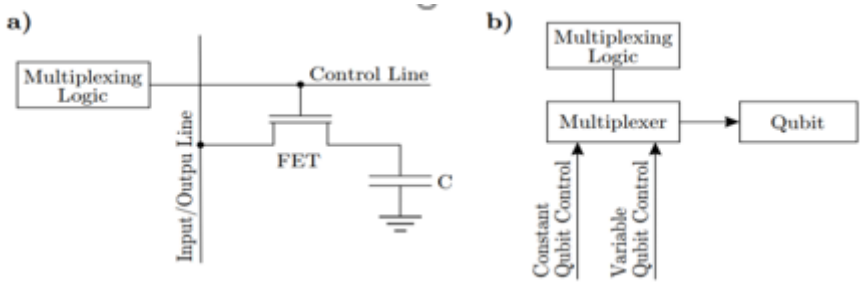


Figure 4. A memory cell of a DRAM(a) and multiplexing quantum memory(b). [1]

Quantum data storage capabilities of significant size must be realized in large-scale quantum computers while the quantum memory hardware requirement is modest (classical). Rent's rule would be violated if hardware demand expanded in lockstep with the number of qubits stored, and control hardware would become too sophisticated and costly for large-scale quantum computers with thousands or millions of qubits.

Multiplexing circuits, as shown in the figure b upside, can be used to reduce hardware consumption. As a result, the capacity to store quantum data/information with a group of parameters that remain constant is required.

To store an ion chain in a segmented Paul trap, for example, all that is required is a negative DC voltage at the ion string's position and positive DC voltages surrounding it, forming an axial confinement for the ion string. In theory, these few voltages may be utilized to store an infinite number of ion strings. This collection of parameters (for trapped ions, a set of DC voltages) is applied to all qubits in the quantum memory during storage. Multiplexing technology allows a shift of this set of parameters to another set that can be adjusted independently in order to access a specific memory cell. This set of parameters allows the quantum information of any memory cell to be moved out of the memory cell out of the quantum memory.

The quantum bus system for quantum information transit, which must be conducted with high fidelity to allow fault-tolerant operation, is one of the most important components of this quantum von Neumann architecture.

Quantum information transport:

Because quantum information cannot be copied, it can only be conveyed via quantum information. Moving qubits physically, quantum teleportation, and photon coupling are all possible. In atomic or molecular qubit systems, quantum information can be transferred by physically moving the qubit from one spatial point to another. For example, in segmented trapped ion systems, adjust the confining axial DC potential to shift Paul traps, ions, or ion strings. In general, solid-state systems are incapable of such mobility. Solid state technologies, such as spins in silicon, do, nevertheless, allow for the movement of qubits.

An entangled qubit pair, each of which is at the destination, is required for quantum

teleportation. The qubit at the destination, from which quantum information is extracted, is the first, while the qubit at the source is the second. When a conditional quantum state is generated, it is also necessary to perform a gate, which needs a qubit measurement across a classical channel to the destination. Because quantum data must be stored and accessed in the quantum memory, each site must have read-out and quantum-gate capabilities. This is in direct contradiction to DiVincenzo's requirement for particular hardware for each criterion, which necessitates more hardware than physical movement. However, in a number of solid-state systems, it could be an effective method.

Qubits to photons mapping has been demonstrated in atomic or molecule qubit systems, as well as solid-state systems. As with quantum teleportation, this approach demands quantum logic at each memory location and is hence hardware heavy. The capacity to transition from one qubit system to another is one of the benefits of quantum information transit via quantum teleportation or photon mapping over qubit mobility. QIP might be performed, for example, using superconducting circuit QED systems, and nitrogen vacancy centers in diamond could be exploited for long-term quantum memory storage.

Because quantum gate operations and quantum state readout are needed at each point in the quantum memory, these technologies have a higher memory hardware requirement than systems that enable qubit mobility. Quantum teleportation and photon mapping will be restricted to small and medium-scale devices if this challenge cannot be overcome. Large-scale systems with low hardware needs per stored qubit may be required to transfer the qubits in the quantum computer.

The SenseHAT 8x8 pixel LED which is drived by Raspberry Pi will be used to display results of the computation. We will observe the output of the quantum circuit which will be used for random number generation. The interface diagram can be seen in Figure 2.

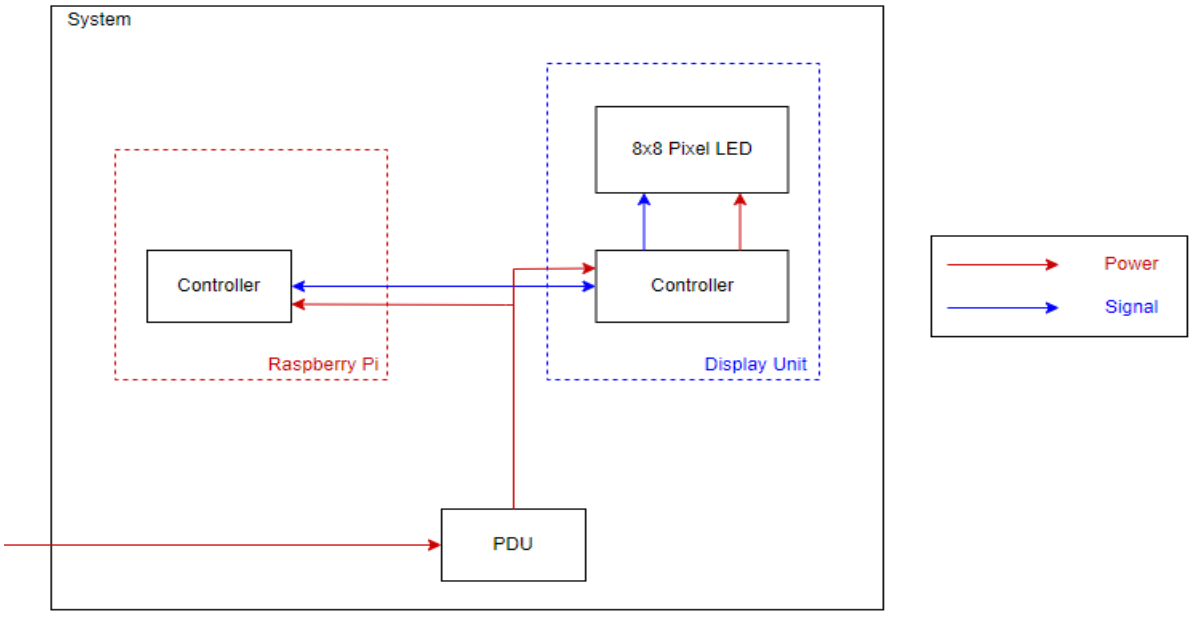


Figure 5. Interface diagram for the system.

When the device turned on it will check firstly is device successfully booted. If not booted, it we will reboot the system until device is ready. When device ready to use, qiskit program that we codded will be run. If program successfully run, device will send the results to 8x8 pixel LED to display. If everything successfully done, program can be re-run or device can be turned off. Our system will work as it has been shown in Figure 3.

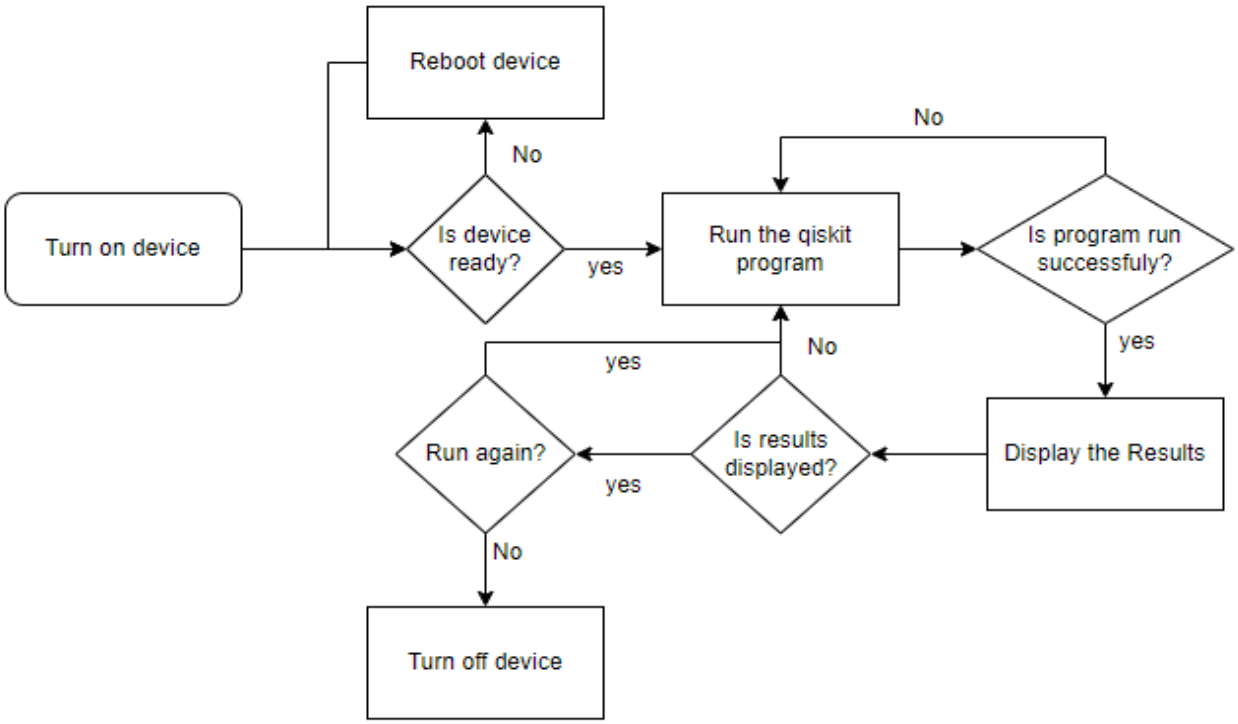


Figure 6. Process chart for the system.

2. WORK PLAN

2.1. Work Breakdown Structure (WBS)

The necessary works break down into two sub systems as hardware and software as in Figure 4. The main challenging part is the software development where we will be focusing on implementing the study of random number generation in quantum computer.

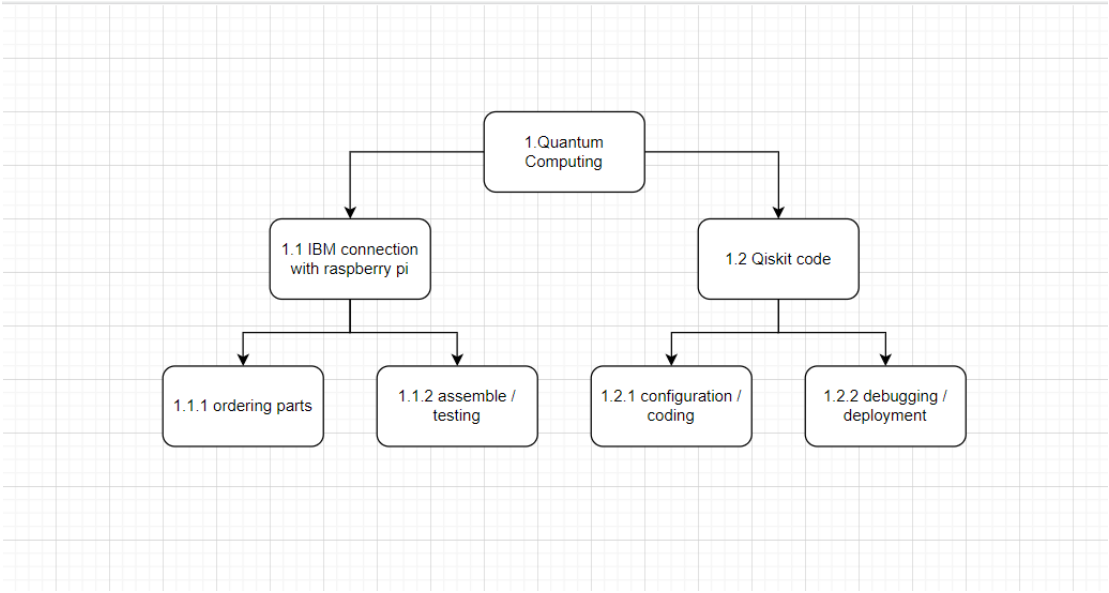


Figure 7. Work breakdown structure for the project.

2.2. Responsibility Matrix (RM)

In this project, we have divided into two subsystems as Computer Engineering and Electrical and Electronics Engineering. Both departments have people will be responsible from writing report and management. The responsibility matrix can be investigated in Table 2.

Electrical and Electronics Engineers will be responsible from the studying Quantum Information Processing for designing quantum circuit.

Computer Engineers will be responsible from configuration of Raspberry Pi for quantum simulation and implementing the code for the quantum circuit. Furthermore, computer engineers will be implementing Quantum Information Processing study which is done by Electrical and Electronics Engineers.

Table 2. Responsibility Matrix for the team

	Electrical & Electronics Engineering			Computer Engineering		
Task	Iren	Canberk	Tolga	Ege	Iren	Kaan
Management	R		S		R	S
Reporting	S	S	R	R	S	S
Configuration of Rapsberry Pi				R	S	S
Quantum Information Processing	R	R	S			
Quantum circuit design	S	R	R			
Impelementing the Qiskit Code				S	R	R
Implementing Random Number Generation				S	S	R

2.3. Project Network (PN)R

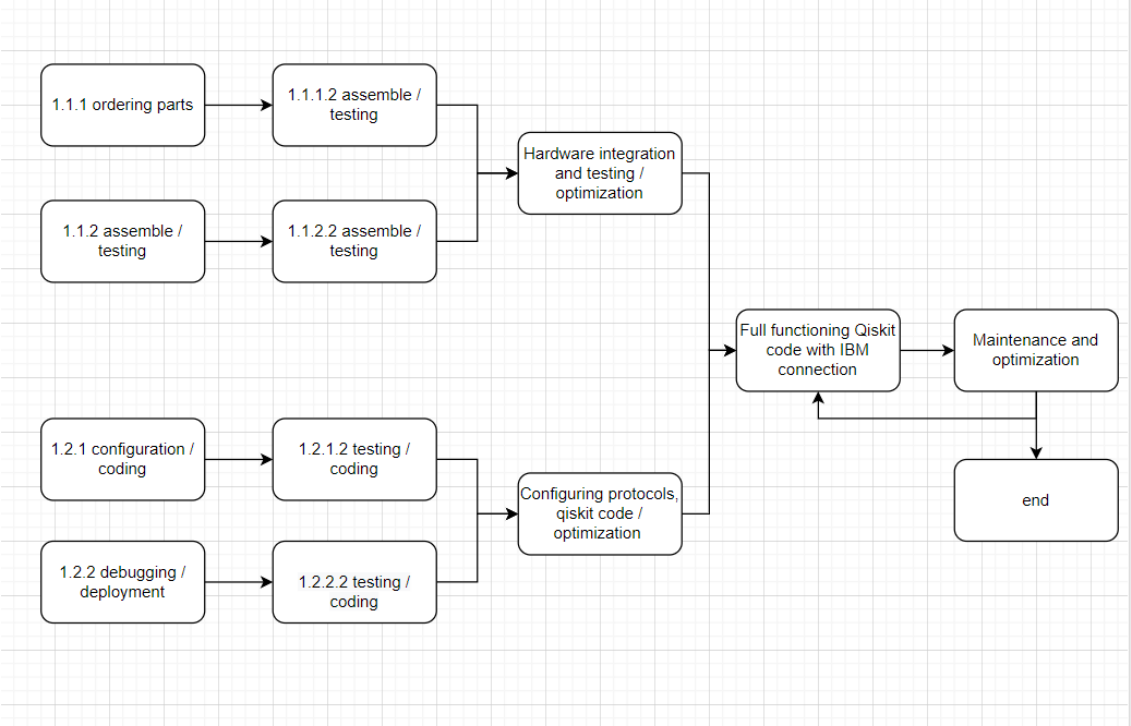


Figure 8. The project network.

2.4. Gantt char

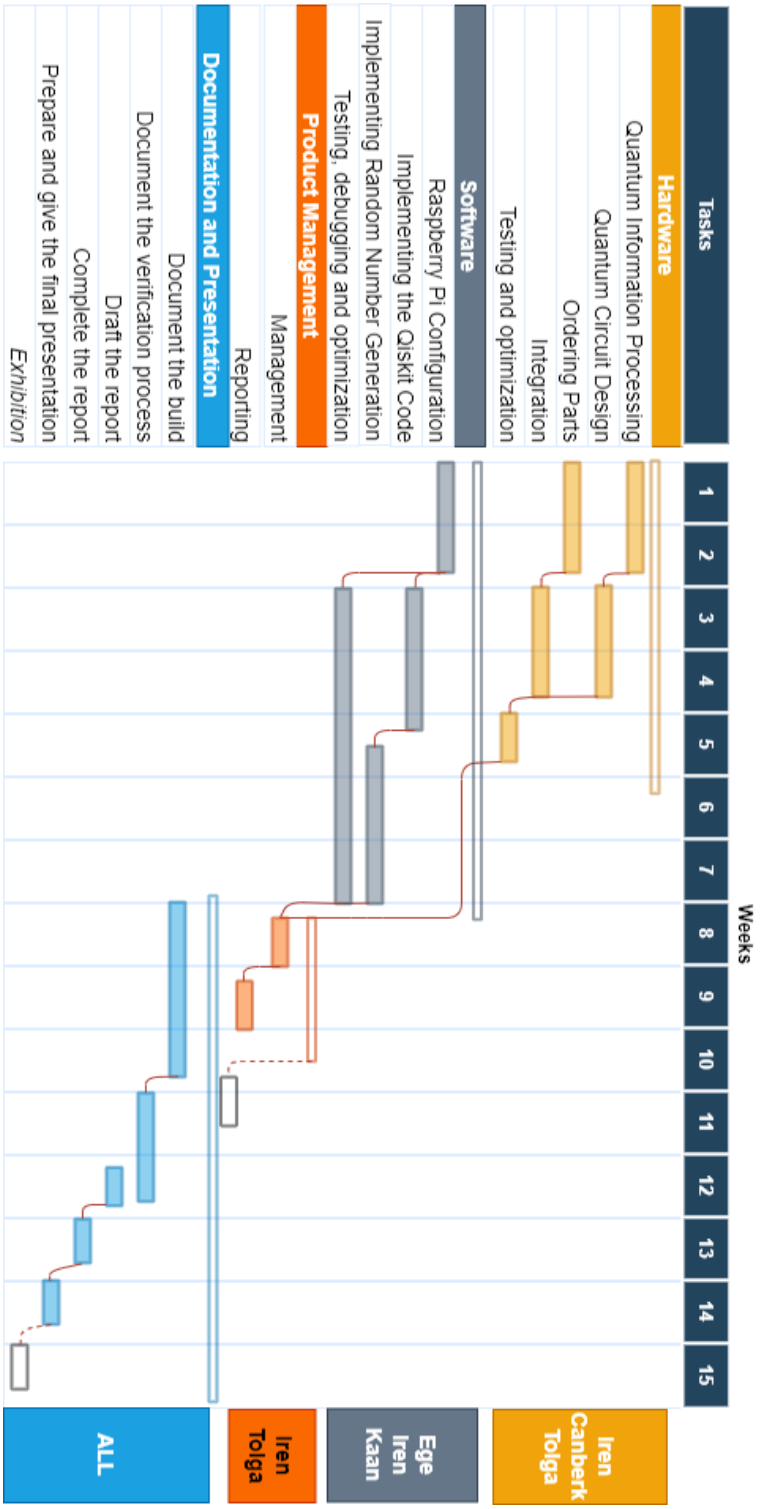


Table 3. Gantt chart for the materialisation phase of the project.

2.5. Costs

Cost of the Raspberry Pi and pixel LED is affected with exchange rates. So, prices could be different when placing order.

Table 4. Costs

Cost	
SenseHAT 8x8 Pixel Led	450,00
Total	450,00

2.6. Risk assessment

There are several risks which cannot be neglected. It is difficult to measure how much time we need to complete our quantum circuit construction and implementing it in qiskit, since the python code heavily rely on constructed circuit. Furthermore, this, also, leads to failure of integration of subsystem which is again important to test project.

Since this project will be implemented on Raspberry Pi, it is difficult the RAM will be enough for computational power. If it is not enough, we will use IBM-Q which is a real quantum computer built by IBM. If we cannot connect to the IBM-Q only choice is the use our personal computer which RAM can be increase if needed.

Table 5. Risk assessment

Failure Event	Probability	Severity	Risk Level	Plan of Action
Covid-19 Lockdown	Possible	Minor	Low	Online systems will be used
Failure of Integration of Subsystems	Possible	Major	High	Consultancy will be received from professionals in the field
Raspberry Pi and 8x8 Led malfunction	Possible	Minor	Low	Connecting to IBM-Q remotely
Bugs	Likely	Moderate	High	Solving bugs
Unable to connect IBM-Q remotely	Possible	Minor	Low	Using personal computers for simulation
Failure of construction of quantum circuit	Possible	Major	High	Consultancy will be received from professionals in the field

3. SUB-SYSTEMS

3.1. Electrical & Electronics Engineering

3.1.1. Requirements

Since Electrical & Electronics Engineering will be responsible from the Raspberry Pi configuration and will be developing quantum computing solutions with Computer Engineering sub-system. Necessary information related to Raspberry Pi and Qiskit should be know by sub-system. To build solutions using Qiskit every sub-system member should be aware of the algorithms of quantum computing and the Qiskit quantum gates.

3.1.2. Technologies and methods

Raspberry Pi is a name for series of single-board computers developed by Raspberry Pi Foundation. Raspberry Pi is a very cheap computer that runs Linux operating system while providing a set of GPIO (General Purpose Input/Output) pins. It has many use of areas from basic learn purpose to implementing complex IoT solutions.

The quantum computer is composition of quantum logic gates. There are many gates which performing an elementary unitary operation on qubits. The qubit is the basis unit of a information like bit in the classical computer. This qubit's behaviour differs from classical bits while representing a 1 bit information. Unlikely to classical bits qubit also able to exists in superposition. The superposition is a state that qubit can be exists as a 0 and 1 at the same time [3] [7].

In quantum computing states are represented with vector called statevectors. The simple quantum logic gates' statevectors shown in Appendix A. The NOT gate is the simplest gate which changes the state of the qubit. If state of the qubit is 0, it is changed to 1 and vice versa. There are two more NOT gates,which they have control qubit(s), such as CNOT and CCNOT. The operation of changing state of the qubit is controlled by another qubit or qubits. If control qubit's state is 1 the target qubit's state is changed, if control qubit is not in state 0 target qubit's state stays still. One of the important quantum logic gates is Hadamard gate which is used to create superposition [1] [4].

3.1.3. Conceptualization

Raspberry Pi is a small computer that runs on Linux and is used in areas such as robotics. IBM Quantum Composer and Lab is a hardware, made by quantum circuits and runs on multiple machines, that can be remotely accessed via IBM Cloud, or via state-of-the-art software. Connection to IBM Quantum computer with Raspberry is generally named like RasQberry or Qrasp.

For connection, a raspberry and a Sense HAT LED display is used. The process starts with setting up Raspberry Pi, Python environment, instillation of necessary dependencies, Qiskit and so

on. After the setup of Jupyter Notebook for Qiskit on Raspberry Pi, commands for IBM Quantum providers and backends are executed. After this, Qiskit files are available for execution. With installation and enabling of Sense HAT the set up and configurations are done. Most of this process is open-source and if any problems arise, they will be solved depending on the error such as running different commands on shell, upgrading of Raspberry Pi system, using Sense HAT emulator.

3.1.4. Physical architecture

We will be using Raspberry Pi 4 and 8x8 pixel LED to obsver outcomes of the implementation of conceptual solutions. Corresponding diagram for the system can be seen in Fingure 7.

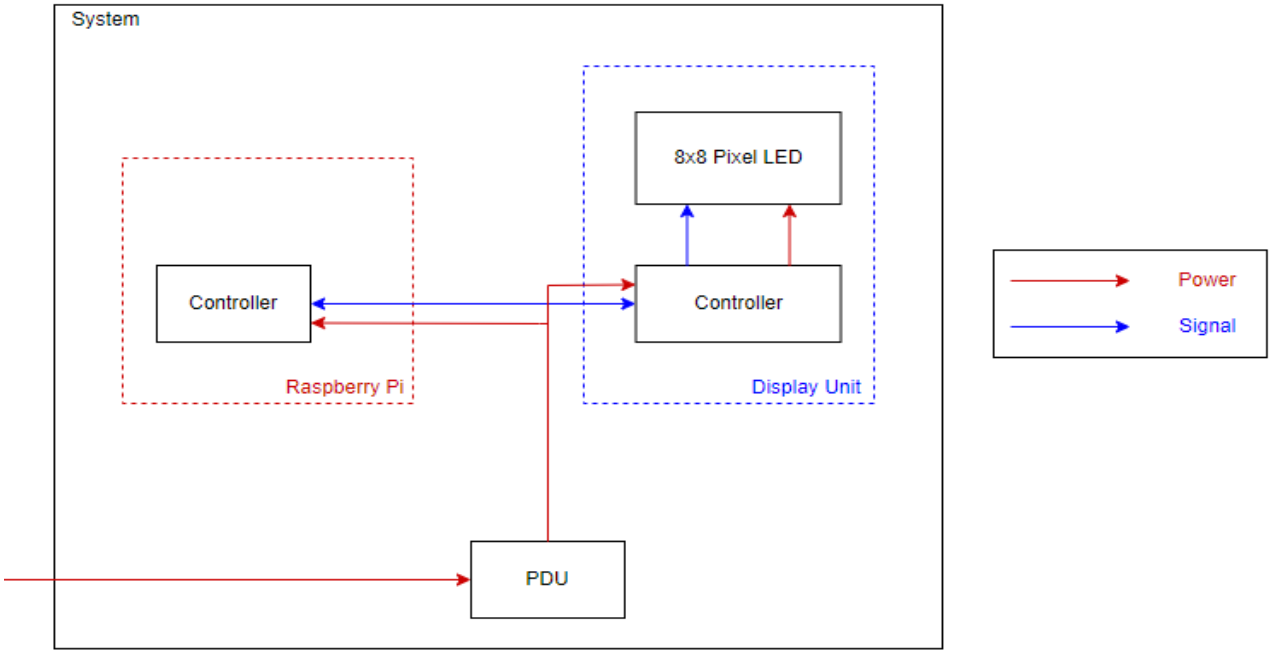


Figure 9. Interface diagram for the system

When Figure 8 is investigated, it is seen that when device turned on system will be checked the device ready to operate if not it will try to reboot device. When device is ready, Qiskit program corresponding to our project will be run and it will be check whether the program successfully run or not. If it is run successfully, we will display the result on 8x8 LED, if it is not displayed program will run again. When all program successfully worked, it is up to user to run program again to see different outputs or turn off the device.

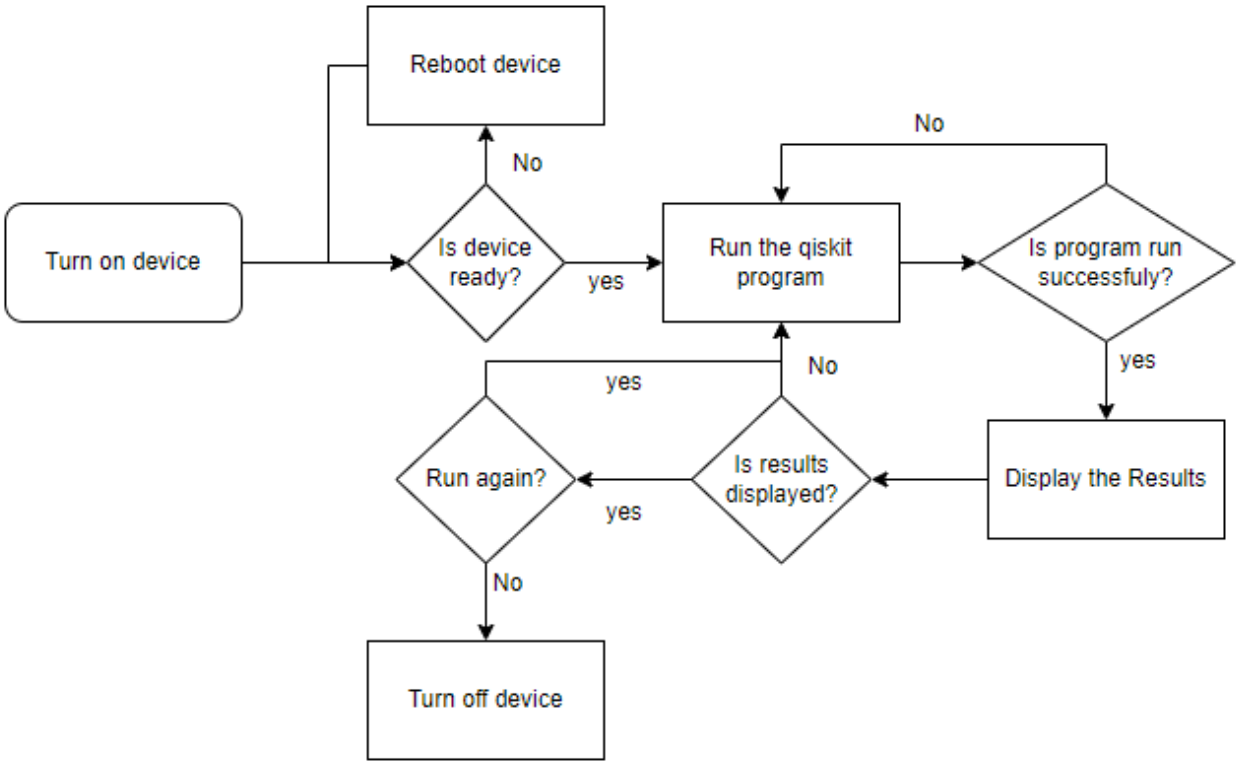


Figure 10. Process chart for the system

3.1.5. Materialization

Since our project rely on construction and implementing quantum circuit and software. We will only use Raspberry Pi 4 and 8x8 pixel LED as materials. The 8x8 pixel LED will be connected to Raspberry Pi 4 and will be drived by Raspberry Pi. As Electrical and Electronics Engineers, we firstly set up the raspberry pi later we combined the SenseHat with Raspberry Pi. Since our part finished, we started to help Computer Engineers while they are programming. In order to achieve our goals we proceed step by step from fundamentals to complex implementation.

Superposition is where the quantum states of a physical system is added together, giving an unknown system of probability until it is observed. It is demonstrated by Schrödinger's Wave Equations and explained by the renowned Schrödinger's Cat.

$$i\hbar \frac{\partial \psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m} \frac{\partial^2 \psi(x,t)}{\partial x^2} + V(x,t)\,\psi(x,t)$$

In a quantumly entangled group of particles measurements of momentum, spin and polarization are correlated. No matter the distance, they can show the perfectly correlated values of these measurements, if this is the case information shall travel instantaneously but according to relativistic theory this is a violation of the physical laws of the universe, therefore Albert Einstein famously called this phenomenon "Spooky action at a distance" and ridiculed the theory saying that the formulations must have been incomplete. Nevertheless, this phenomenon today constitutes one

of the main pillars of quantum computing and its implementation.

Firstly, 2 qubit entanglement and superposition circuits are implemented on jupyter notebook and observed the results programmatically. The first implementation of the project was 2 qubit entanglement. To achieve the entanglement, we applied respectively Hadamard Gate to first qubit and CNOT Gate to second qubit. After that we measured the qubits. The circuit drawing by of the entanglement in qiskit can be seen in figure 11.

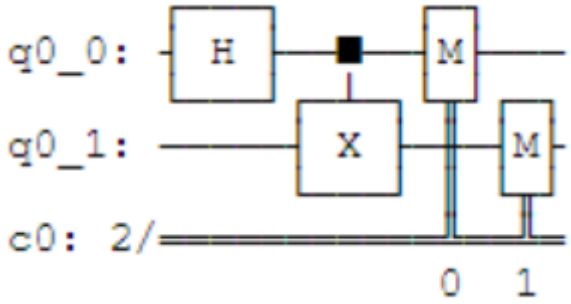


Figure 11. 2 Qubit entanglement circuit.

It is observed that the results were close to %50 probability of occurance of one state. The results can be seen in figure 12. These results are subject to change when running the simulation.

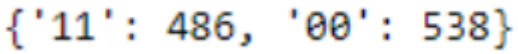


Figure 12. 2 Qubit entanglement results.

Later, we constructed the superposition circuit by using quantum gates on jupyter notebook as shown in figure 13. In order to construct the circuit, we applied Hadamard Gate to both qubits.

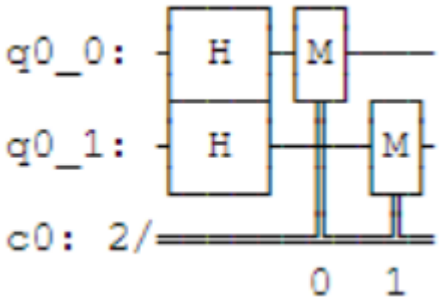


Figure 13. 2 Qubit superposition circuit.

Compared to entanglement, the results show that each states' probability close to %25 as can be seen in figure 14.

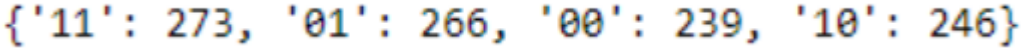


Figure 14. 2 Qubit superposition results.

Furthermore, when we finished the 2 qubit superposition and entanglement we further

constructed the 3 qubit versions of both of them. Since increase in qubit number doesn't affect the probability of entanglement, main difference was in superposition. Each states' probability changes to %12.5 from %25 since the state number doubled compared to 2 qubit.

In order to construct the circuit for 3 qubit entanglement one more CNOT Gate added. Which is affecting the third qubit's state when second qubit's state changed. This circuit let us observe the probability of occurrence of 000 or 111 states. Below, the circuit for 3 qubit entanglement is shown in figure 15.

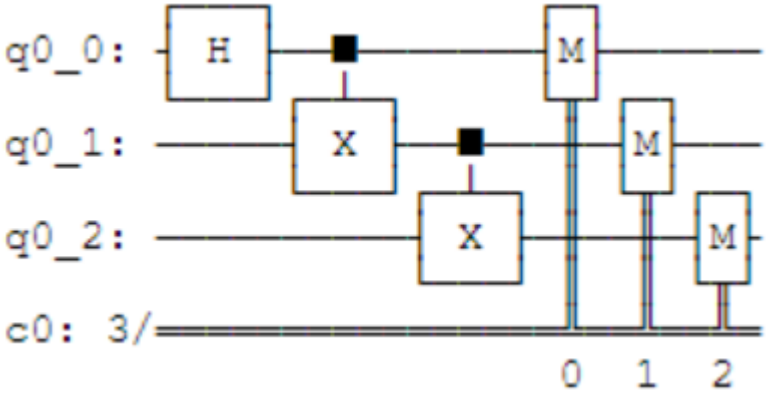


Figure 15. 3 Qubit entanglement circuit.

Since the probability of occurrence is %50 theoretically any of the states from 000 or 111. Results are close to the each other. Later we constructed the circuit for 3 qubit superposition which is let the probability of occurrence change from %25 to %12.5 due to state number is doubled. In order to construct circuit we applied Hadamard Gate to the third qubit. The circuit can be observed in figure 16.

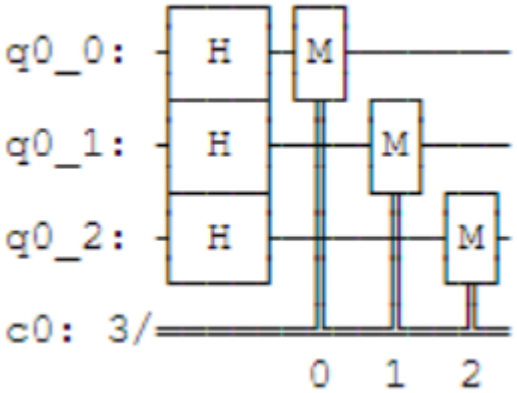


Figure 16. 3 Qubit superposition circuit.

Finally, Computer Engineer programmed algorithm to show the results on SenseHAT. (Algoritma anlatılacak ama tam değil genel hatlarıyla bilgi verilmesi gerek)

3.1.6. Evaluation

At the first step of test, we will run example codes to see if our integration of system working

properly. When we see that system works under preferred conditions, we will start to implement our project step by step. Each step will be tested when the step is done. Expected outcomes of the project for each conceptual solution is differs. For two qubit superposition and entanglement simulations expected outcome is 00 or 11 with probability of 50%. If the result is one of the states that can be 00 or 11, program is working as intended.

3.2. Computer Engineering

3.2.1. Requirements

Computer Engineering will handle the programming and software development relying on quantum mechanics and states with correlation in Electrics & Electronical Engineering department. This phases will be demonstrated on Raspberry Pi boards and these boards will be programmed using QISKIT software development kit and Python programming language.

3.2.2. Technologies and methods

QISKIT is a free, open-source software development kit for dealing with quantum computers at the circuit, pulse, and algorithm level. It includes tools for writing and altering quantum programs, as well as running them on IBM Quantum Experience prototype quantum devices or local computer simulators.

Python is a dynamically semantic, interpreted, object-oriented high-level programming language. Its high-level built-in data structures, together with dynamic typing and dynamic binding, making it ideal for Rapid Application Development and as a scripting or glue language for connecting existing components.

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy. Machine learning starts with data The data is gathered and prepared to be used as training data, or the information the machine learning model will be trained on. The more data, the better the program.

Quantum Machine Learning is constructed on two concepts : hybrid quantum-classical models and quantum data. Quantum data is any data source that occurs in a natural or artificial quantum system. This can be data generated by a quantum computer.

Algoritmhs

A. Deutsch-Jozsa Algorithm

Deutsch-Jozsa Problem

A concealed Boolean function f is supplied to us, which takes a sequence as input and returns either 0 or 1, as follows:

$$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1 \text{ , where } x_n \text{ is 0 or 1}$$

The given Boolean function has the feature of being guaranteed to be either balanced or constant. For any input, a constant method returns all 0s or all 1, whereas a balanced program runs 0s for half of all entries and 1s with the other half.

The Classical Solution

In the best case scenario, two queries to the oracle can identify whether the concealed Boolean equation, f(x), is balanced: for example, if there are both f(0,0,0,...)→0 and f(1,0,0,...)→1, it is known that the function is balanced because there are two different outputs.

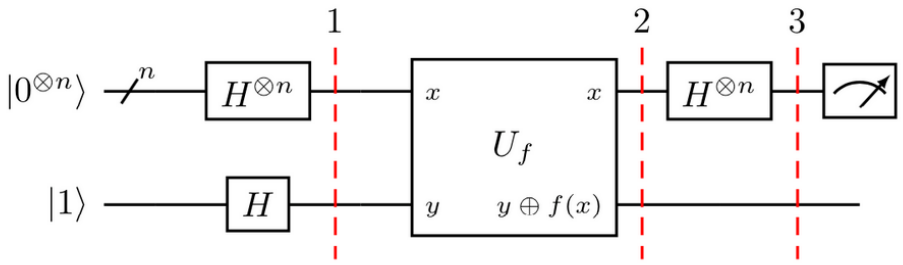
In the worst-case scenario, if the same result for each input is gotten, exactly half of all potential inputs plus one have to be tested to be sure that f(x) is constant. 2(n1)+1 trial inputs are required to be positive that f(x) is constant in the worst scenario, because the set of all possible inputs is 2n. If 8 of the 16 possible options are checked for a 4-bit string and obtain all 0s, it is still feasible that the 9th input yields a 1 and f(x) is balanced. This is a very extremely rare event in terms of probability.

$$P_{\text{constant}}(k) = 1 - \frac{1}{2^{k-1}} \qquad \text{for } 1 < k \leq 2^{n-1}$$

In reality, if we were above x percent certain, we could truncate our classical algorithm early. However, if we want to be absolutely certain, we'll need to verify 2(n1)+1 inputs.

The Quantum Solution

This problem can be solved with 100 percent certainty using a quantum computer after only one call to the function f(x), as long as the function f is implemented as a quantum oracle that transfers the state $|x\rangle|y\rangle$ to $|x\rangle|y \oplus f(x)\rangle$, where is addition modulo 2. The Deutsch-Jozsa algorithm's generic circuit is shown below.



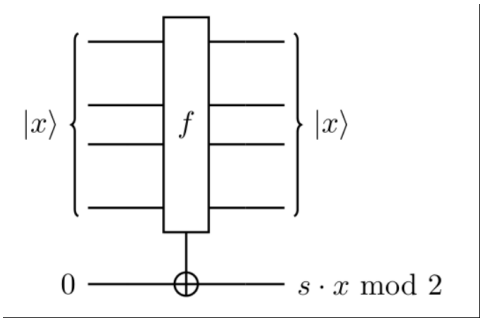
B. The Bernstein-Vazirani Algorithm

The Bernstein-Vazirani Problem

It is given another black-box function, f, which accepts a string of bits (x) as input and returns either 0 or 1, as follows:

$$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1 \text{ where } x_n \text{ is } 0 \text{ or } 1$$

Rather than being balanced or constant, as in the Deutsch-Jozsa issue, the function must now return the bitwise product of the input with some string, s. To put it another way, given an input x, f(x)=sx (mod 2). It is expected that s will be discovered. The Bernstein-Vazirani oracle looks like this as a standard reversible circuit:



The Classical Solution

Classically, the oracle returns:

$$f_s(x) = s \cdot x \mod 2$$

Given an x as an input. By querying the oracle with the following sequence of inputs, the concealed bit string s can be revealed:

Input(x)
100...0
010...0
001...0
000...1

Where each query yields a distinct piece of data (the bit si). For example, x = 1000...0 can be used to determine the least relevant bit of s, and x = 0100...0 can be used to get the next least significant bit, and so on. This implies the fs(x) method would have to be called n times.

The Quantum Solution

This problem can be solved with 100 percent confidence using a quantum computer after only one call to the function f. (x). To identify the hidden bit string, the quantum Bernstein-Vazirani algorithm is quite simple:

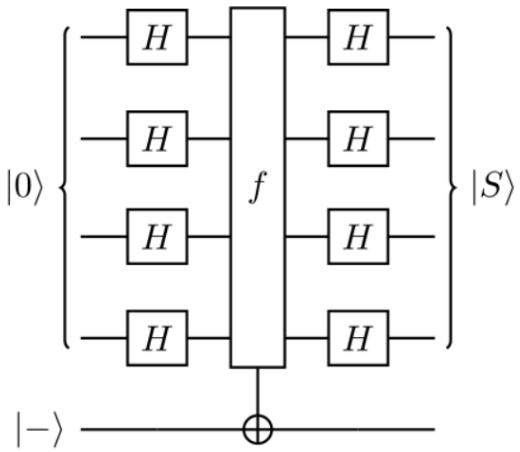
1- Set the input qubits to the |0n state and the output qubit to the |0n state.

2- To the input register, apply Hadamard gates.

3-Query the oracle

4- To the input register, apply Hadamard gates.

5-Measure



Let's take a closer look at what happens when an H-gate is applied to each qubit to better understand the process. If there is an n-qubit state, |a⟩, and apply the H-gates, the transformation can be saw :

$$|a\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle.$$

When a quantum register |00...0> is chosen and n Hadamard gates are applied to it, the familiar quantum superposition is obtained:

$$|00\ldots0\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle$$

Because a=0 in this situation, the phase term (-1)(ax) vanishes, and (-1)(ax)=1.

The traditional oracle fs returns 1 if the input x is such that s·x mod2=1, and 0 otherwise. The following transformation is obtained if the same phase kickback trick from the Deutsch-Jozsa algorithm is applied to a qubit in the state |−⟩:

$$|x\rangle \xrightarrow{f_s} (-1)^{s\cdot x}|x$$

By querying the quantum oracle fs with the quantum superposition acquired from the Hadamard transformation of |00...0⟩, the procedure to expose the concealed bit string follows naturally. Namely,

$$|00\ldots0\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \xrightarrow{f_a} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle$$

Since the inverse of the n Hadamard gates is also the n Hadamard gates itself, it can be obtain "a" by

$$\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle \xrightarrow{H^{\otimes n}} |a\rangle$$

C. Shor’s algorithm

Shor's algorithm is well-known for its ability to factor integers in polynomial time. The widely used cryptosystem, RSA, relies on factoring being difficult for large enough integers since the best-known classical algorithm requires superpolynomial time to factor the product of two primes.

For Shor's algorithm implementation, you'll need qiskit and a few Python modules:

```
import matplotlib.pyplot as plt
import numpy as np
from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_histogram
from math import gcd
from numpy.random import randint
import pandas as pd
from fractions import Fraction
print("Imports Successful")
```

Period Finding

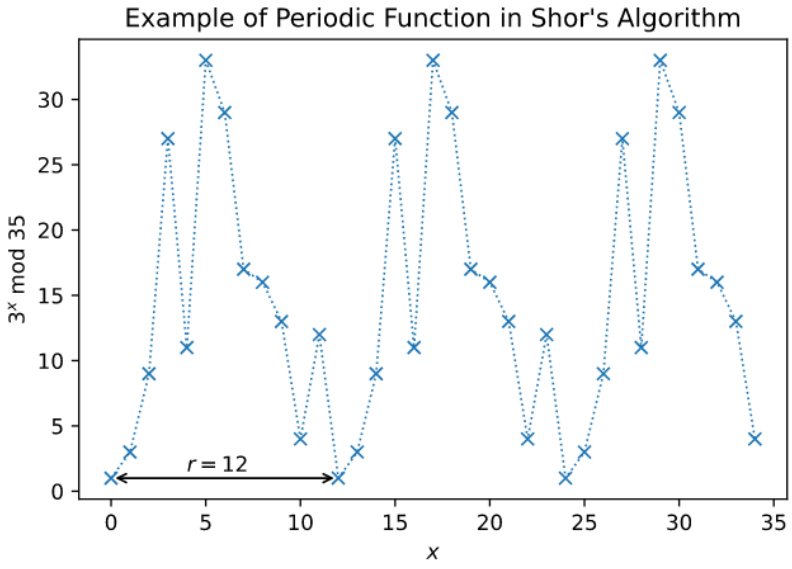
Periodic Function:

$$f(x) = a^x \bmod N$$

where "a" and "N" are both positive numbers, an is smaller than N, and they have no factors in common. The period (r) is the smallest non-zero integer that:

$$a^r \bmod N = 1$$

The graph below shows an example of this function:



Solution

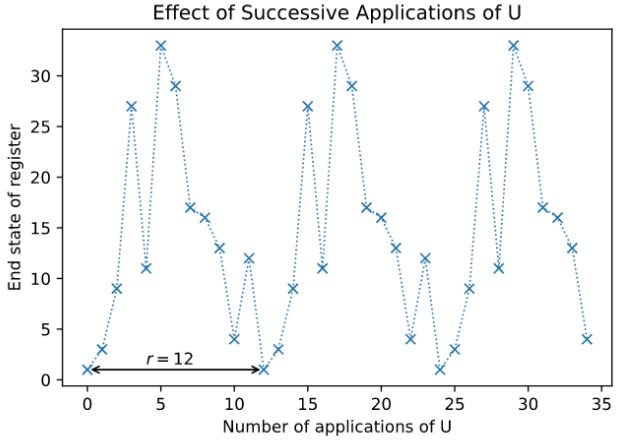
Shor devised a method for estimating quantum phase on the unitary operator:

$$U|y\rangle \equiv |ay \bmod N\rangle$$

If the state $|1\rangle$ is given, it can be observed that each subsequent application of U multiplies

the state of our register by a(modN), and after r applications, the state of our register is returned to the state |1⟩. With a=3 and N=35, for example:

$$
\begin{aligned}
U|1\rangle &= |3\rangle \\
U^2|1\rangle &= |9\rangle \\
U^3|1\rangle &= |27\rangle \\
&\vdots \\
U^{(r-1)}|1\rangle &= |12\rangle \\
U^r|1\rangle &= |1\rangle
\end{aligned}
$$



As a result, an eigenstate of U would be a superposition of the states in this cycle (|u0):

$$|u_0\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |a^k \bmod N\rangle$$

The eigenvalue of this eigenstate is 1. Consider, for example, the case whenever the phase of the kth state is proportional to k:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i k}{r}} |a^k \bmod N\rangle$$

$$U|u_1\rangle = e^{\frac{2\pi i}{r}} |u_1\rangle$$

Because it contains r, this eigenvalue is especially noteworthy. In fact, the phase differences between the r computational basis states must be equal, therefore r must be included. This isn't the only eigenstate with this behavior; to broaden the scope, an integer, s, can be multiply to this phase difference, which will show up in our eigenvalue:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i s k}{r}} |a^k \bmod N\rangle$$

$$U|u_s\rangle = e^{\frac{2\pi i s}{r}} |u_s\rangle$$

For each integer value of s, there is a distinct eigenstate.

$$0 \leq s \leq r-1.$$

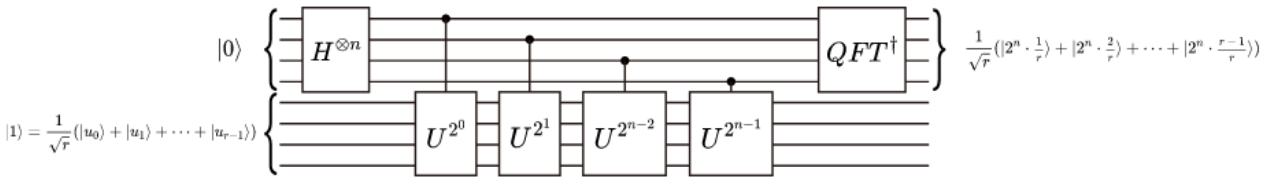
When all of these eigenstates are added together, the varied phases cancel out all of the computational basis states save |1⟩:

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle$$

Because the computational basis state |1⟩ is a superposition of these eigenstates, a phase is observed when QPE is performed on U using the state |1⟩:

$$\phi = \frac{s}{r}$$

Where s is a random integer ranging from 0 to r1. To find r, the continuing fractions algorithm on ϕ is utilized. This is the circuit schematic (notice that it follows Qiskit's qubit ordering convention):



D. Grovers Algoritm

The problem's complexity In the classic situation, the answer is O(n). In the quantum scenario, however, it will be the square root of n. As a result, desired elements in lengthy lists can be found using quadratic speed simultaneous computation.

```
In [3]:
my_list=[1,3,5,2,4,9,5,8,0,7,6]

In [4]:
def the_oracle(my_input):
    winner=7
    if my_input is winner:
        response = True
    else:
        response = False
    return response

In [5]:
for index, trial_number in enumerate(my_list):
    if the_oracle(trial_number) is True:
        print('Winner found at index %i'%index)
        print('%i calls to the Oracle used'%(index+1))
        break

Winner found at index 9
10 calls to the Oracle used
```

By adding a negative phase to the solution states, grover's algorithm solves oracles.

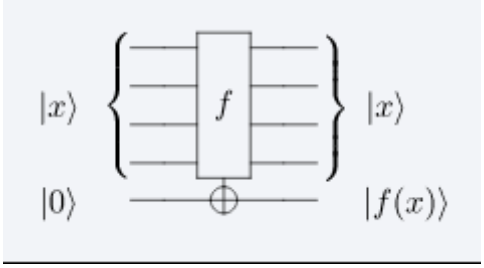
$$U_\omega|x\rangle = \begin{cases} |x\rangle & \text{if } x \neq \omega \\ -|x\rangle & \text{if } x = \omega \end{cases}$$

The element in the diagonal matrix that corresponds to the marked item will have a negative phase.

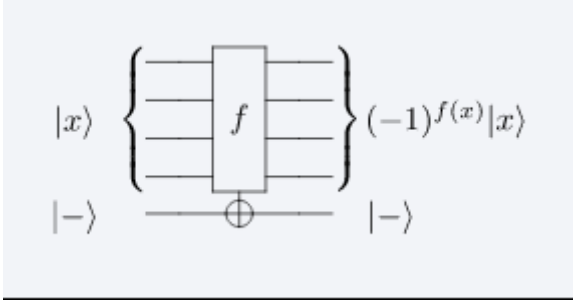
$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \leftarrow \omega = 101$$

Circuit structure of a Grover Oracle

A traditional function can be transformed into a reversible circuit of the following type:



The phase kickback effect transforms the 'output' qubit into a Grover oracle (identical to the Deutsch-Jozsa oracle) if it is initialized in the state $|-\rangle$:



The auxiliary ($|-\rangle$) qubit is then ignored.

Application of Quantum Circuits with Qiskit

We programmed a Raspberry-PI to act like a quantum computer and used a sense-hat module to create quantum circuits showing superposition and quantum entanglement (GHZ and Bell States) on real-life.

Raspberry PI runs Python 3.5, and Qiskit is a Python-based software so we did not face any struggles on hardware. Also we wanted our program to work on IBM Quantum machines for further access.

We imported and used sense-hat, qiskit and numpy libraries for our cause.

A. Two Qubit Entanglement

For our Algorithm at first we valued the number of Qubits we are going to use which is 2. Then from Qiskit we imported Quantum register, Classical Register, Quantum Circuit; we defined them on our code and outlined our Quantum Circuit. On our circuit we applied Hadamard gate to first qubit and CNOT gate to second qubit for achieving the entanglement.

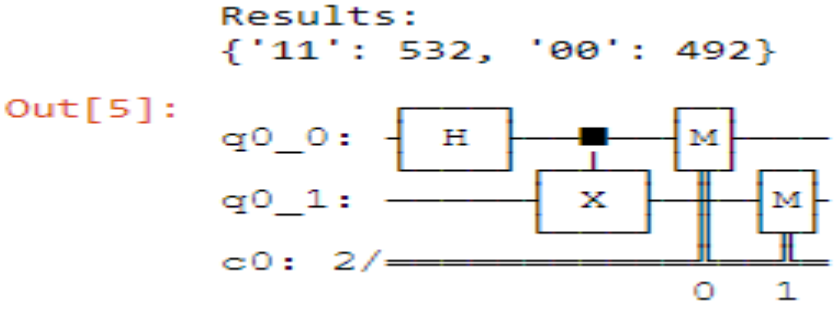
Final stage was creating qDict dictionaries to both get the results of probability of quantum states from our algorithm, drawing the circuit, and display the quantum dictionary as a bar graph on the SenseHat 8x8 pixel display.

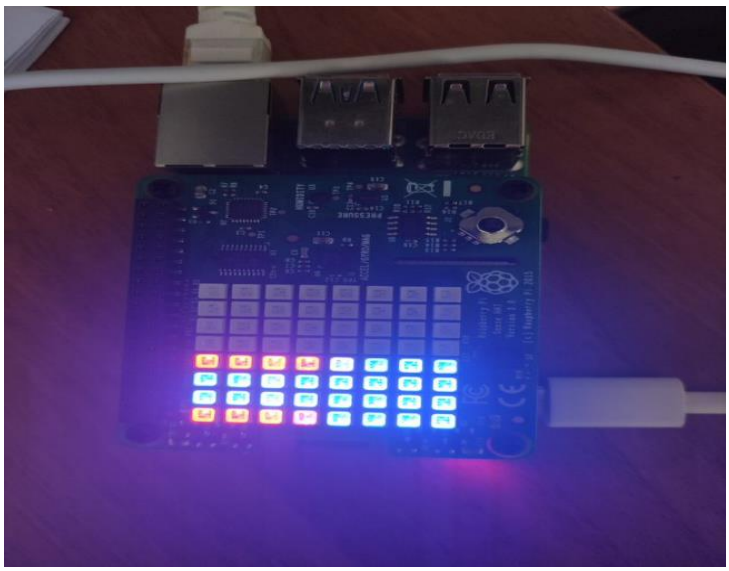
We achieved:

```
{'11': 486, '00': 538}
{'00': 538, '01': 0, '10': 0, '11': 486}
```

the results show that each states' probability of occurence is close to %50.

Our circuit:





B. Two Qubit Superposition

Altering our Two Qubit Entanglement algorithm we could achieve Superposition. Mainly at the skeleton of our Quantum circuit implementation. Instead of putting Hadamard gate on the first qubit and applying CNOT gate to the second qubit to achieve Entanglement we applied Hadamard gates to both qubits to achieve Superposition. The functions we had used and libraries were basically the same except the deletion of CNOT gate and addition of the second Hadamard gate. But result of probability of states drastically changed

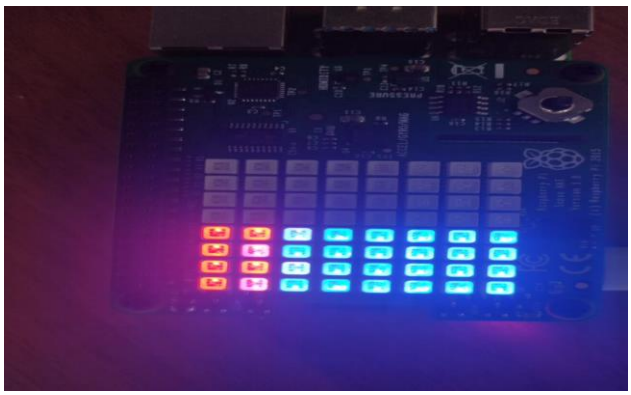
We achieved:

```
{'11': 273, '01': 266, '00': 239, '10': 246}
{'00': 239, '01': 266, '10': 246, '11': 273}
```

the results show that each states' probability of occurrence is close to %25.

Our circuit:

```
Results:
{'11': 273, '01': 266, '00': 239, '10': 246}
Out[5]:
q0_0: ─H─M─
q0_1: ─H───M─
c0: 2/═══╩═╩═
         0 1
```



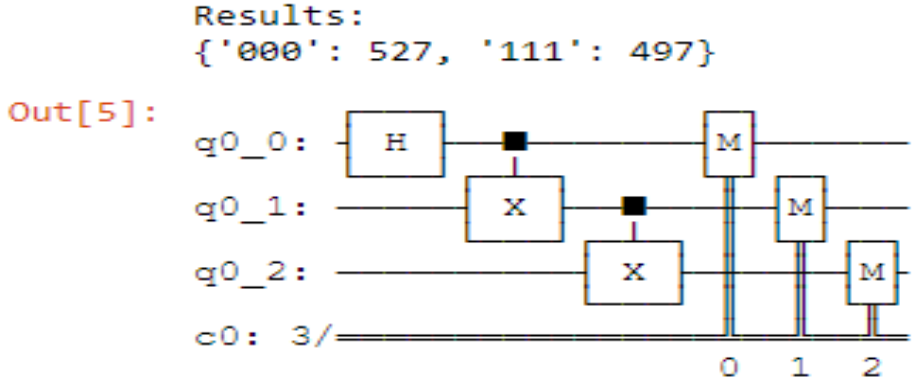
C. Three Qubit Entanglement.

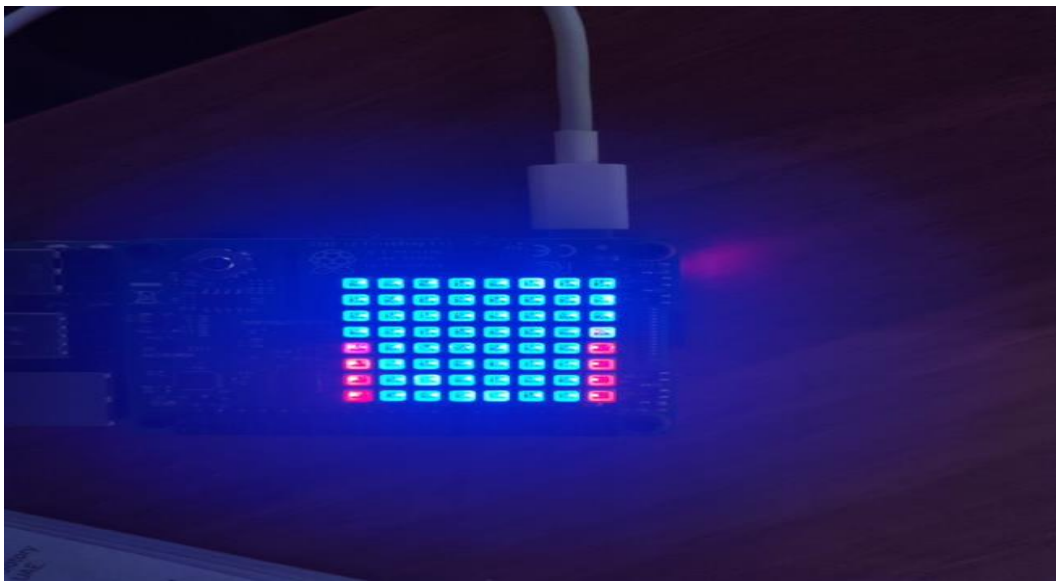
Our Code once changed again to increase the number of qubits and prior to gates we used to achieve Two Qubit Entanglement we added an extra gate of CNOT. We put the Hadamard gate to first qubit then to second and third we used CNOT gates to achieve entanglement on our quantum circuit.

We achieved:

```
{'000': 527, '111': 497}
{'000': 527, '001': 0, '010': 0, '011': 0, '100': 0, '101': 0, '110': 0, '111': 497}
```

Increase in the number of qubits did not affect the states' probability of occurrence to much, and is also close to %50.





D. Three Qubit Superposition

The difference of this stage from the Two Qubit Superposition one is that we have a 3 qubit quantum circuit. Still having Hadamard gates on all of the Qubits to achieve superposition.

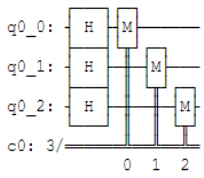
We achieve:

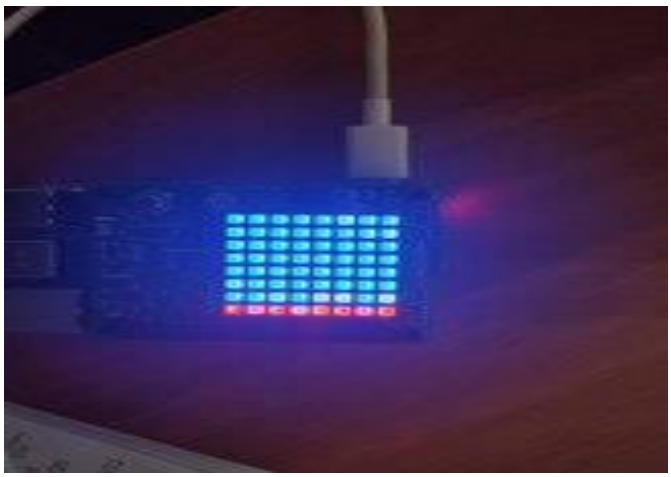
```
{'110': 120, '101': 132, '011': 144, '111': 132, '010': 134, '000': 141, '100': 109, '001': 112}
{'000': 141, '001': 112, '010': 134, '011': 144, '100': 109, '101': 132, '110': 120, '111': 132}
```

Probability of Occurrence decreased to approximately 12,5.

```
Results:
{'110': 120, '101': 132, '011': 144, '111': 132, '010': 134, '000': 141, '100': 109, '001': 112}
```

Out[5]:





E. Random Number Generation:

There are two types of RNGs: Pseudo-RNGs (PRNGs) and True RNGs (TRNGs). Even if pseudo-RNG output is 'random enough' for many uses, it is not genuinely and statistically random.

A True RNG, on the other hand, requires an actual piece of hardware to measure some random process in the real world, as no computer program could ever be truly random. These gadgets range from air noise measurement equipment to radioactive material connected through USB.

Therefore, the reason we used qiskit to generate random numbers is that all randomness in the cosmos is the outcome of quantum systems collapsing when they are measured. In a sense, this is pure randomness, and it is the underlying source of unpredictability in any TRNG.

qRNG is a python-based open-source quantum random number generator. It accomplishes this by communicating with any of IBM's publicly accessible quantum computers using the Qiskit API.

```
5   import qrng
```

qRNG connects to IBMQ and generate some numbers:

```
7   qrng.set_provider_as_IBMQ('')
8   qrng.set_backend()
9
10  random_number = qrng.get_random_int(0,255)
```

Then, using Sense Hat library, random generated number is shown in the rasqberry :

```
12  from sense_hat import SenseHat
13
14  sense = SenseHat()
15  sense.rotation = 180
16  sense.low_light = True
17  sense.show_message("1 Byte Random")
18  sense.show_message(str(random_number))
19
```

3.2.3. Conceptualization

Algorithms that are made to prove that quantum computing has advantages over classical computing are examined. Deutsch-Jozsa, Bernstein-Vazirani, Shor's and Grover's algorithms are chosen as they are algorithms that are solved differently for classical and quantum computing.

Each of these algorithms are executed with Qiskit and are analysed for the difference in runtimes and outcomes between classical and quantum computing.

3.2.4. Physical architecture

Our project has two physical attributes, Raspberry boards to be programmed, computers to get through software development phase.

3.2.5. Materialization

We will out-buy the raspberry pi board so we can program it to be a small-scale quantum computer with QISKIT software development kit afterwards we can run through our algorithms and solve our problems.

3.2.6. Evaluation

After the configuration of Raspberry algorithms, quantum properties such as superposition and entanglement will be shown. In addition, the chosen algorithms created for quantum computing will be executed by Qiskit software, to show the structure of Qiskit programming and the relative outcomes. Through these algorithms quantum properties are shown and proved.

These algorithms are chosen to implement the quantum properties and quantum computing to see the behaviour of quantum system and analyse the outcome with comparison to classical computing.

4. INTEGRATION AND EVALUATION

Begin the first paragraph here.

Begin the second paragraph here.

4.1. Integration

4.2. Evaluation

Begin the first paragraph here.

Begin the second paragraph here.

5. SUMMARY AND CONCLUSION

In summary, ...Begin the first summary paragraph here.

Begin the second paragraph here.

In conclusion, ...Begin the first conclusion paragraph here.

Begin the second paragraph here.

ACKNOWLEDGEMENTS

We wish to thank our advisers Prof. Seref KALEM and Assist. Prof Ece Gelal SOYAK for giving all their supports and sharing their experiences with us to develop our project during researching.

This work was partly funded by Bahçeşehir University.

REFERENCES

[1] M. Kiili, «Quantum Computing,» HAAGA-HELIA University of Applied Sciences, 2014.

[2] M. F. Brandl, «A Quantum von Neumann Architecture for Large-Scale Quantum Computing,» Cornell University, 2017.

[3] J. West, «The Quantum Computer,» *XOOTIC,* cilt 10, no. 3, 2003.

[4] R. Garipelly, P. M. Kiran ve A. S. Kumar, «A Review on Reversible Logic Gates and their Implementation,» *International Journal of Emerging Technology and Advanced Engineering,* cilt 3, no. 3, 2013.

[5] K. Miyamoto ve K. Shiohara, «Reduction of Qubits in Quantum Algorithm for Mone Carlo Simulation by Pseuda-random Number Generator,» *APS Physics,* no. 102, 2020.

[6] X. Ma, X. Yuan, Z. Cao, B. Qİ ve Z. Zhang, «Quantum Random Number Generation,» *Nature Partner Journals,* cilt 2, pp. 1-9, 2016.

[7] T. Hey, «Quantum Computing: An Introduction,» *Computing & Control Engineering Journal,* cilt 10, no. 3, pp. 105-112, 1999.

[8] A. Montanaro, «Quantum Speedup of Monte Carlo Methods,» *Royal Society,* cilt 471, no. 2181, 2015.

[9] J. Preskill, «Quantum Computing: Pro and Con,» *Royal Society,* cilt 454, no. 1969, 1998.

[10] K. Tamura ve Y. Shikano, «Quantum Random Number Generation with the Superconducting Quantum Computer IBM 20Q Tokyo,» TUCS Lecture Notes, 2020.

APPENDIX A

Quantum Logic Gates [1].

Not Gate = Pauili X = $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

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

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

Hadamard = $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

CNOT = $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

TOFFOLI = CCNOT = $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

SWAP = $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

FREDKIN = CSWAP = $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

APPENDIX B

2 Qubit Entanglement Python Code

```
#!/usr/bin/env python

from sense_hat import SenseHat
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import Aer
from time import sleep
import numpy as np

numberOfQubits = 2
shots = 1024

qr = QuantumRegister(numberOfQubits)
cr = ClassicalRegister(numberOfQubits)
circuit = QuantumCircuit(qr, cr)

circuit.h(qr[0])
circuit.cx(qr[0], qr[1])

circuit.measure(qr[0], cr[0])
circuit.measure(qr[1], cr[1])

backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=shots)
result = job.result()
print ("Results:")
Qdictres = result.get_counts(circuit)
print(Qdictres)
circuit.draw()

sense = SenseHat()
sense.rotation = 180
```

```
sense.low_light = True
sense.show_message("2Q Entanglement")

R = [255,   0,   0] # Red
B = [0,     0, 255] # Blue
W = [255, 255, 255] # White

# Create a default Qdict dictionary with all values 0
global lst
lst = [bin(x)[2:].rjust(numberOfQubits, '0') for x in range(2**numberOfQubits)]
values = [0]*pow(2,numberOfQubits)
Qdict = dict(zip(lst,values))

# Update the dictionary with the actual dictionary values sent to the function.
Qdict.update(Qdictres)

# Scale by dividing by 1024 (shots) - For now assuming 1024, which is set by the sh parameter.
# Qdict.update({m: (1/sh) * Qdict[m] for m in Qdict.keys()})

print(Qdictres)
print(Qdict)

# Defining the display colors.
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

sense.clear()
sense.rotation = 90
# Writing to the SenseHat display pixels.
for key in Qdict:
    y=7-int(key,2) # Cycle through the states
    for x in range (0,8): # Cycle through the pixels
        val = ((x+1)*128)-Qdict[key] # The difference between the state result and the pixel x
position
```

```
        if val<0:
            #If the state result is greater than the pixel, set pixel color red.
            color=red
        else:
          if val>0 and val<128:
            #If the state result is within the pixel, set pixel color gradient.
            fade = (255-(2*val),0,(2*val))
            color=fade
          else:
            #If the state result is less than the pixel, set pixel color blue.
            color=blue
        #Set pixel color.
        sense.set_pixel(x, y, color)

sleep(5)
sense.clear()
```

APPENDIX C

2 Qubit Superposition Python Code

```
#!/usr/bin/env python

from sense_hat import SenseHat
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import Aer
from time import sleep
import numpy as np

numberOfQubits = 2
shots = 1024

qr = QuantumRegister(numberOfQubits)
cr = ClassicalRegister(numberOfQubits)
circuit = QuantumCircuit(qr, cr)

circuit.h(qr[0])
circuit.h(qr[1])
circuit.measure(qr[0], cr[0])
circuit.measure(qr[1], cr[1])

backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=shots)
result = job.result()
print ("Results:")
Qdictres = result.get_counts(circuit)
print(Qdictres)
circuit.draw()

sense = SenseHat()
sense.rotation = 180
sense.low_light = True
```

```
sense.show_message("2Q Superposition")

R = [255,   0,   0] # Red
B = [0,     0, 255] # Blue
W = [255, 255, 255] # White

# Create a default Qdict dictionary with all values 0
global lst
lst = [bin(x)[2:].rjust(numberOfQubits, '0') for x in range(2**numberOfQubits)]
values = [0]*pow(2,numberOfQubits)
Qdict = dict(zip(lst,values))

# Update the dictionary with the actual dictionary values sent to the function.
Qdict.update(Qdictres)

# Scale by dividing by 1024 (shots) - For now assuming 1024, which is set by the sh parameter.
# Qdict.update({m: (1/sh) * Qdict[m] for m in Qdict.keys()})

print(Qdictres)
print(Qdict)

# Defining the display colors.
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

sense.clear()
sense.rotation = 90
# Writing to the SenseHat display pixels.
for key in Qdict:
    y=7-int(key,2) # Cycle through the states
    for x in range (0,8): # Cycle through the pixels
        val = ((x+1)*128)-Qdict[key] # The difference between the state result and the pixel x position
        if val<0:
```

```
                #If the state result is greater than the pixel, set pixel color red.
                color=red
        else:
            if val>0 and val<128:
                #If the state result is within the pixel, set pixel color gradient.
                fade = (255-(2*val),0,(2*val))
                color=fade
            else:
                #If the state result is less than the pixel, set pixel color blue.
                color=blue
        #Set pixel color.
        sense.set_pixel(x, y, color)

sleep(5)
sense.clear()
```

APPENDIX C

3 Qubit Entanglement Python Code

```
#!/usr/bin/env python

from sense_hat import SenseHat
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import Aer
from time import sleep
import numpy as np

numberOfQubits = 3
shots = 1024

qr = QuantumRegister(numberOfQubits)
cr = ClassicalRegister(numberOfQubits)
circuit = QuantumCircuit(qr, cr)

circuit.h(qr[0])
circuit.cx(qr[0], qr[1])
circuit.cx(qr[1], qr[2])

circuit.measure(qr[0], cr[0])
circuit.measure(qr[1], cr[1])
circuit.measure(qr[2], cr[2])

backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=shots)
result = job.result()
print ("Results:")
Qdictres = result.get_counts(circuit)
print(Qdictres)
circuit.draw()
```

```
sense = SenseHat()
sense.rotation = 180
sense.low_light = True
sense.show_message("3Q Entanglement")

R = [255,   0,   0] # Red
B = [0,     0, 255] # Blue
W = [255, 255, 255] # White

# Create a default Qdict dictionary with all values 0
global lst
lst = [bin(x)[2:].rjust(numberOfQubits, '0') for x in range(2**numberOfQubits)]
values = [0]*pow(2,numberOfQubits)
Qdict = dict(zip(lst,values))

# Update the dictionary with the actual dictionary values sent to the function.
Qdict.update(Qdictres)

# Scale by dividing by 1024 (shots) - For now assuming 1024, which is set by the sh parameter.
# Qdict.update({m: (1/sh) * Qdict[m] for m in Qdict.keys()})

print(Qdictres)
print(Qdict)

# Defining the display colors.
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

sense.clear()
sense.rotation = 90
# Writing to the SenseHat display pixels.
for key in Qdict:
    y=7-int(key,2) # Cycle through the states
    for x in range (0,8): # Cycle through the pixels
```

```
        val = ((x+1)*128)-Qdict[key] # The difference between the state result and the pixel x position
        if val<0:
            #If the state result is greater than the pixel, set pixel color red.
            color=red
        else:
           if val>0 and val<128:
            #If the state result is within the pixel, set pixel color gradient.
            fade = (255-(2*val),0,(2*val))
            color=fade
           else:
            #If the state result is less than the pixel, set pixel color blue.
            color=blue
        #Set pixel color.
        sense.set_pixel(x, y, color)

sleep(5)
sense.clear()
```

APPENDIX D

3 Qubit Superposition Python Code

```
#!/usr/bin/env python

from sense_hat import SenseHat
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import Aer
from time import sleep
import numpy as np

numberOfQubits = 3
shots = 1024

qr = QuantumRegister(numberOfQubits)
cr = ClassicalRegister(numberOfQubits)
circuit = QuantumCircuit(qr, cr)

circuit.h(qr[0])
circuit.h(qr[1])
circuit.h(qr[2])
circuit.measure(qr[0], cr[0])
circuit.measure(qr[1], cr[1])
circuit.measure(qr[2], cr[2])

backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=shots)
result = job.result()
print ("Results:")
Qdictres = result.get_counts(circuit)
print(Qdictres)
circuit.draw()

sense = SenseHat()
```

```
sense.rotation = 180
sense.low_light = True
sense.show_message("3Q Superposition")

R = [255,   0,   0] # Red
B = [0,     0, 255] # Blue
W = [255, 255, 255] # White

# Create a default Qdict dictionary with all values 0
global lst
lst = [bin(x)[2:].rjust(numberOfQubits, '0') for x in range(2**numberOfQubits)]
values = [0]*pow(2,numberOfQubits)
Qdict = dict(zip(lst,values))

# Update the dictionary with the actual dictionary values sent to the function.
Qdict.update(Qdictres)

# Scale by dividing by 1024 (shots) - For now assuming 1024, which is set by the sh parameter.
# Qdict.update({m: (1/sh) * Qdict[m] for m in Qdict.keys()})

print(Qdictres)
print(Qdict)

# Defining the display colors.
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

sense.clear()
sense.rotation = 90
# Writing to the SenseHat display pixels.
for key in Qdict:
    y=7-int(key,2) # Cycle through the states
    for x in range (0,8): # Cycle through the pixels
        val = ((x+1)*128)-Qdict[key] # The difference between the state result and the pixel x
```

```
position
        if val<0:
            #If the state result is greater than the pixel, set pixel color red.
            color=red
        else:
           if val>0 and val<128:
            #If the state result is within the pixel, set pixel color gradient.
            fade = (255-(2*val),0,(2*val))
            color=fade
           else:
            #If the state result is less than the pixel, set pixel color blue.
            color=blue
        #Set pixel color.
        sense.set_pixel(x, y, color)

sleep(5)
sense.clear()
```

APPENDIX E

1 Byte Random Number Generator Python Code

```
# Open Source qRNG link: https://github.com/ozaner/qRNG
import qrng

qrng.set_provider_as_IBMQ('')
qrng.set_backend()

random_number = qrng.get_random_int(0,255)

from sense_hat import SenseHat

sense = SenseHat()
sense.rotation = 180
sense.low_light = True
sense.show_message("1 Byte Random")
sense.show_message(str(random_number))
```