

AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Computer Science, Electronics and Telecommunications

Department of Computer Science



MASTER OF SCIENCE THESIS

**ASSESSMENT OF IBM-Q QUANTUM
COMPUTER AND ITS SOFTWARE
ENVIRONMENT**

ZUZANNA CHRZĄSTEK

SUPERVISOR:
dr inż. Marian Bubak

CO-SUPERVISOR:
dr inż. Tomasz Stopa

Kraków 2018

Upředzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także upředzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście, samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

.....

SIGNATURE

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr Marian Bubak, for the continuous support of my M.Sc. study, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me a lot during my research and writing of this thesis.

I would also like to thank Dr Tomasz Stopa, for his suggestions and valuable advices, and for provision of the materials used in this study.

I would also thank Dr Katarzyna Rycerz and Dr Piotr Gawron for their support and constructive remarks as well as Dr Janusz Majewski and Prof. Witold Dzwinel for discussions and suggestions during seminars.

Abstract

Nowadays, quantum computing is coming out of research laboratories and starts to become a technology available almost to everyone. Researchers are studying the interesting potential of quantum computers and it seems that practical usage of them is just around the corner. There are many companies that build their own quantum computers, e.g. Google, D-Wave Systems, Rigetti Computing. IBM has contributed significantly to this by opening access to their IBM-Q. Their goal is to build available universal quantum computing systems for business and science.

Quantum computing is a quite new field of study, it is impossible to transfer the style of programming from classical computers to their quantum counterparts. A big challenge is to implement quantum algorithms on a specific quantum computer and fit them to its architecture.

The study begins with short introduction to basics of quantum computing (Chapter 3). Then present physical background of quantum computing (Chapter 4). Next, we put together information about architectures available in the IBM-Q and analyze software environment of the IBM-Q (Chapter 5). After that, we have analyzed capabilities of a tool for creating quantum programs, QISKit. Then, we proposed an extension to software environment of the IBM-Q using some of the features of QuIDE simulator by creating a bridge between those two pieces of software. It was validated with different quantum algorithms already implemented on the IBM-Q (Chapter 6). Finally we have implemented a quantum random walk algorithm to see how accurate is the IBM's computer and we have compared results from real backend with results from the IBM's simulator as well as with from the QuIDE simulator (Chapter 7). On the basis of these study we have assessed the usability of the IBM-Q software environment (Chapter 8).

The thesis showed that there is lack of tools that would help developers to create quantum algorithms. It also showed the importance of quantum simulators, validated the IBM-Q quantum computer and reviewed some available quantum algorithms. It also pointed that decoherence analysis is a big problem and it is the most desirable direction of future work.

In this thesis we applied empirical research strategies and methods: surveys for obtaining background information and for assessment of quantum computers and simulators, case studies for analysis of quantum algorithms execution, computer experiments for validation and evaluation of usability.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem outline	2
1.3	Objectives	2
1.4	Methodology	3
1.5	Content of this work	3
2	Related work	4
3	Basic notions of quantum computing	7
3.1	What is a qubit?	7
3.2	Quantum gates	8
3.3	Measurement	8
3.4	Entanglement	9
3.5	Summary	9
4	How do quantum computers work?	10
4.1	General concept of quantum computers	10
4.1.1	What is quantum computer?	10
4.1.2	Physical phenomena used to build quantum computers	11
4.2	Quantum computer based on superconducting qubits	11
4.2.1	Necessary conditions to build a quantum computer	11
4.2.2	Superconducting qubits	14
4.2.3	How qubits are controlled	15
4.2.4	How measurement is done	15
4.2.5	Fault-tolerant architecture	16
4.3	Quantum volume concept	17
4.4	Overview of IBM quantum computers in the IBM-Q	18
4.4.1	History of the IBM-Q	18
4.4.2	IBM-Q quantum computer's architecture	18
4.4.3	Parameters of the IBM-Q	21

5	Software environment of the IBM-Q	23
5.1	Methods of creating quantum algorithms on the IBM-Q	23
5.1.1	IBM-Q's graphical user interface	24
5.1.2	QISKit - SDK for working with OpenQASM and the IBM Q	26
5.1.3	Assembly language - OpenQASM	28
5.1.4	Representation of quantum operators in QISKit and the IBM-Q Composer	28
5.2	IBM-Q's simulator	30
5.3	Results from the simulator and from the real quantum processor	30
5.4	Matching algorithm with architecture	31
6	Interoperability of the IBM-Q and QuIDE simulators	32
6.1	The need for simulators of quantum computers	32
6.2	Comparison of the IBM-Q and QuIDE software environments	32
6.3	Comparison of the IBM-Q and QuIDE simulators	33
6.4	Converter from QASM to C#	34
6.5	Validation of the converter	36
6.5.1	The Grover algorithm	37
6.5.2	The Deutsch-Jozsa algorithm	40
6.5.3	The Shor algorithm	46
6.6	Summary	49
7	Implementing and running a quantum walk on the IBM-Q	50
7.1	Introduction - classical random walk	50
7.2	Discrete quantum walk	51
7.3	Quantum circuit implementation of quantum walk on a circle	53
7.4	Decomposition of multiple-controlled gates	54
7.5	Implementation of the quantum walk algorithm with a two-qubit walker	57
7.6	Implementation of the quantum walk algorithm with a three-qubit walker	61
7.7	Conclusions	64
8	Assessment of usability of IBM-Q	65
9	Summary, conclusions and future work	66

1. Introduction

1.1. Motivation

Quantum computing is no longer only a theory, nowadays it is becoming a technology available almost to everyone. Researchers and scientists are studying the potential of quantum computers and it seems that practical usage of them is just around the corner. In academia, the field of quantum computation has been growing explosively since its inception in the 1980s and the importance of quantum computers is widely recognized by industry and governments [1]. Developing quantum software is an essential issue as it is crucial in research areas such as:

- *quantum simulation* which includes research on applications of medium-sized quantum computers,
- *algorithms and complexity* which explores capabilities of larger-scale quantum computers (e.g. machine learning, search and optimization problems),
- *cryptography* because larger-scale quantum computers will break our current cryptosystems, but also it is a chance to built new cryptosystems,
- *quantum software framework* which ease the use of quantum computers,
- *quantum information science* which is to provide the mathematical theory behind quantum information processing and it is important for the development of such issues as error correction schemes and a deeper understanding the physical theory behind the hardware.

There are many companies that build their own quantum computers, e.g. Google, D-Wave Systems, Rigetti Computing. IBM also has its contribution to development of the industry. IBM-Q is an initiative taken by IBM and their goal is to build commercially available universal quantum computing systems for business and science. For now they have created the prototype commercial quantum processor that anyone can use. The platform that enables connection to the processor via the IBM Cloud was launched in May 2016 and it is successively developed. At this moment users can choose from three different architectures: two 5-qubit architectures and a 16-qubit architecture.

It is worth to emphasize that it is impossible to transfer the style of programming from classical computers to their quantum counterparts. On classical computers there are debuggers

and other tools that ease program optimization. We can check what exactly is happening at each stage of the execution of a program. Quantum computers cannot give users this possibility as each observation interfere with a state of a quantum system and this is the main reason why the role of quantum computers simulators rises.

When it comes to quantum algorithms, there are many of them created. A collection of about 1000 quantum algorithms is presented in [2]. Now comes a question: how to implement these algorithms on a specific quantum computer and fit them to its architecture?

The questions presented above are the motivation of the investigations presented in this thesis. This work will describe current state of this technology, analyze its current and future applications, as well as results of implementations of selected algorithms on the IBM-Q and propose possible improvements of the accompanying simulators and other software tools.

1.2. Problem outline

The thesis will provide short overview of basic concepts and applications of quantum computing. Next, the state of technology of the IBM quantum computers will be provided as well as the future plans and challenges. The work will include an overview and evaluation of the simulation software and QISKit API capabilities. A set of quantum algorithms of practical importance will be programmed to run on the IBM-Q quantum computer. The work will provide basic knowledge about quantum computing allowing to start deeper scientific work on this subject.

The research question is: what is required to make the existing IBM quantum computer an efficient tool for solving practical problems? As an answer to this question, we pose the following research hypothesis: *quantum computers require appropriate software environments, simulators of quantum computing, programming and compiling tools to be efficiently used for solving practical problems.*

1.3. Objectives

The study will start with short introduction to mathematical basics in quantum computing [3] [4]. Then we will go on to understand physical background of quantum computing, e.g. what type of physical phenomena occurs in different machines [5, 6].

Next, we will gather information about architectures available in the IBM-Q [7] and analyze software environment of the IBM-Q. After that, we will analyze capabilities of a tool for creating quantum programs, QISKIT, a Python API Client to use the IBM-Q [8]. We want to propose

an extension to software environment of the IBM-Q using some of the features of the QuIDE simulator [9] by creating a bridge between those two pieces of software. We will validate our solution with different quantum algorithms already implemented on the IBM-Q.

At the end we plan to implement a quantum random walk algorithm to see how accurate is the IBM's computer [10]. We will compare results from the real backend with results from the IBM's simulator as well as with from the QuIDE simulator.

1.4. Methodology

Empirical research strategies and methods will be applied, namely: surveys for obtaining background information and for assessment of quantum computers and simulators, case studies for analysis of quantum algorithms execution, computer experiments for validation and evaluation of usability.

1.5. Content of this work

In Chapter 2 we present related work to the topic of this thesis, we depict papers that pose a basis to the thesis. In Chapter 3 we introduce the key concepts of quantum information and computation theory. We explain basic notions of quantum computing which the subsequent chapters refer to. Chapter 4 describes a construction of quantum computers. It shows different ways of how to build such a machine as well as a detailed physical description of how the IBM-Q is created. There is also a review of existing the IBM-Q hardware backends. Chapter 5 contains a depiction of software environment created around the IBM-Q. There is a brief description of the graphical interface and frameworks that can be used to create quantum programs. In Chapter 6 we compare the IBM-Q simulator with QuIDE simulator. We also propose extension to the IBM'Q simulator. In Chapter 7 we implement two variants of a quantum walk algorithm and try to execute them on simulators as well as on different backends of the IBM-Q. Chapter 8 contains assessment of usability of the IBM-Q based on experience with implementing and executing quantum algorithms. In Chapter 9 we summarize the thesis, we discuss the goals achieved, present conclusions and propose ideas for potential further work on this topic.

2. Related work

One of the objectives of this thesis is, to present in a concise way, how quantum computers are built nowadays. IBM-Q is based on superconducting qubits. There are papers that deal with this subject. Paper [5] shows properties of superconducting circuits, such as quantized energy levels, superposition of states, and entanglement, which are more commonly associated with atoms and therefore those circuits are ideal as primitive building blocks of quantum computers. Different types of superconducting qubits are described in paper [6]. Researchers from IBM wrote a paper [11] that depicts the important route towards a logical memory with superconducting qubits. They describe the current status of technology with regards to interconnected superconducting-qubit networks and identifies near-term areas of focus to improve devices.

Many companies are constantly trying to create more powerful quantum computer than their competitors. One of them is D-Wave Systems and their D-Wave 2000QTM quantum computer. It uses a process called quantum annealing to search for solutions to a problem. As presented in [12], computation is performed by initializing the quantum processing unit (QPU) into a ground state of a known problem and annealing the system toward the problem to be solved such that it remains in a low energy state throughout the process. At the end of the computation, each qubit ends up as either a 0 or 1. This final state is the optimal or near-optimal solution to the problem to be solved. The creators of this quantum computer says that D-Wave 2000QTM system has "up to 2048 qubits".

Another significant quantum computing company is Rigetti Computing. They created their own 19-qubit quantum system out of superconducting quantum processors and connected their software platform, Forest [13], to it. Rigetti's quantum computer is also available online. People can get access to it and use it on Forest, that enables developers to write and execute quantum software using a quantum-optimized compiler, a software stack for writing hybrid quantum-classical algorithms (PyQuil), and software packages for quantum algorithms (Grove) [14].

The next crucial subject is software for creating quantum programs. User Guide of the IBM-Q [7] presents, next to the basics of quantum computing, how to use the graphical interface of the IBM-Q [15]. There are also many other ways to write an algorithm and then execute it on the IBM-Q. These tools are: The Quantum Composer [15], QASM [16], QISKIT [8]. In [17] the authors present a concept of a software architecture for compiling quantum programs from a high-level language program to hardware-specific instructions. The architecture depicted in

this paper is just a proposal.

In 2017 a group of European scientists created a document called *Quantum Software Manifesto* [18]. The goal of the Manifesto is to show the importance of the quantum software, emphasize the necessity of the quantum software and hardware developers to work together as well as the need of industry involvement in the development of new algorithms. The Manifesto claims that new quantum algorithms are important and also it shows fields where a reliable quantum software is essential. A good-quality software is also needed when it comes to error correction and fault-tolerant computing as well as verification and testing. In the Manifesto, the authors also describe the importance of quantum computer architectures, optimized simulation of quantum systems, quantum machine learning, quantum network software and quantum cryptography.

Later on, we compare two simulators of a quantum computer - the IBM's simulator [7] and the QuIDE created by AGH University of Science and Technology students [9]. There is no work that show differences between those two pieces of software, their advantages and disadvantages and tell when it is best to use each of them. What is more, IBM's simulator does not allow users to check the state of the system during the computation, while the QuIDE has this option. This is a very important feature, because it can be used to test algorithms. Thereupon we would like to create a bridge between the IBM-Q and QuIDE that will convert QASM code into C# code that can be executed on QuIDE.

Quantum computers are designed to outperform standard computers by running quantum algorithms. Areas in which quantum algorithms can be applied include cryptography, search and optimization, simulation of quantum systems and solving large systems of linear equations. In the thesis we also examine some quantum algorithms. There are many of them created already. In [2] one may find a comprehensive catalog of quantum algorithms. They are divided into three sections: Algebraic and Number Theoretic, Oracular, Approximation and Simulation. The author of [19] survey some known quantum algorithms with an emphasis on a broad overview of their applications.

In our analysis we especially put emphasis on quantum walks. A good introduction and overview is in [10] and [20]. The first one focuses mainly on discrete quantum walks and quantum image processing. It begins with a critical and comprehensive assessment of those elements of classical random walks and discrete quantum walks on undirected graphs relevant to algorithm development. The second one provides an introductory survey on quantum random walks.

In [21] the authors discuss an efficient physical realization of topological quantum walks on a one-dimensional finite lattice with periodic boundary conditions. They also describes results of the execution of the quantum algorithm on the IBM-Q five-qubit quantum computer.

Analyzing papers presented above we can notice that there is a need to extend the IBM-Q environment. The results of our research will be used to realize objectives of this thesis. We will describe the IBM-Q, assess the software environment and propose its enhancement.

3. Basic notions of quantum computing

This chapter gives a quick overview of basic notions of quantum computing which are necessary to understand the contents of this thesis.

3.1. What is a qubit?

Classical computers store information on bits, the smallest unit of data, which have 'logical' value, either 0 or 1. Bit is located in a scalar space. Quantum computers store data on bits' counterparts - qubits, units of quantum information. Qubit is a two-state quantum-mechanical system. Quantum mechanics allows the qubit to be in a **superposition** of both states at the same time. It is a property that is fundamental to quantum computing. The state $|\psi\rangle$ of a two-state quantum system can be described by members of two-dimensional complex Hilbert space. This means every state vector is represented by two complex coordinates:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \text{ where } \alpha, \beta \in \mathbb{C}. \quad (1)$$

The coefficients of the linear combinations must follow the rule:

$$|\alpha|^2 + |\beta|^2 = 1. \quad (2)$$

The states $|0\rangle$ and $|1\rangle$ can be treated as orthogonal unit vectors and can be expressed as:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (3)$$

Qubits can be combined in a system of qubits. Suppose we have n qubits and the states of them are $|\phi_1\rangle, |\phi_2\rangle, \dots, |\phi_n\rangle$. The state $|\Phi\rangle$ of the whole system can be expressed as a **tensor product** of the states of the qubits:

$$|\Phi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle \otimes \dots \otimes |\phi_n\rangle. \quad (4)$$

3.2. Quantum gates

State of a qubit can be manipulated by using **quantum gates** which are typically represented as matrices. The result of applying quantum gates can be found by multiplying the matrix representing the gate with the vector representing the quantum state:

$$\text{For } U = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ and } |\phi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad |\phi'\rangle = U|\phi\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a\alpha + b\beta \\ c\alpha + d\beta \end{bmatrix}. \quad (5)$$

As we can see, 2×2 unitary matrix represents a 1-qubit quantum gate. We can act on one qubit with it.

To manipulate an n -qubit system we need an n -qubit quantum gate represented by a unitary matrix $2n \times 2n$. One of the most important multiple qubit gates is a **C-NOT gate** (controlled NOT). Its matrix representation looks as follows:

$$C_{NOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \quad (6)$$

C-NOT manipulates a 2-qubit system. One of the qubits is called **control bit** and when it is in state $|1\rangle$, C-NOT acts as a negation on the second bit, called **target bit**. If control bit is in state $|0\rangle$, the gate has no effect.

An extension of a C-NOT gate is a **Toffoli gate**. It is a 3-qubit gate with two control bits and one target bit. If both control bits are in state $|1\rangle$, the gate acts as a negation on the target bit.

These are only examples from a large set of quantum gates.

3.3. Measurement

To extract information stored in qubits they need to be **measured**. It is an irreversible operation: once qubits are measured we can't restore their original state.

If we would measure a state of a qubit which state is $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we would get either $|0\rangle$ or $|1\rangle$. The result is random and we can only calculate the probabilities. The probability of getting $|0\rangle$ is $|\alpha|^2$ and the probability of getting $|1\rangle$ is $|\beta|^2$ and we see that states $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ are equivalent.

Analogously we may measure the state of n -qubit system $|\Phi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle \otimes \dots \otimes |\phi_n\rangle$. After the measurement the system changes its state to one of the state $|\phi_i\rangle$, where $i \in 1, 2, \dots, n$ with the probability $|\alpha_i|^2$.

To visualize quantum computations it is convenient to use quantum circuits. They clearly show qubits' initialization at the beginning, quantum gates that are used and measurement at the end.

3.4. Entanglement

Entanglement is a phenomenon that occurs when pairs or groups of particles interact in ways such that the quantum state of each particle cannot be described independently of the others. For example, it is possible to prepare two particles in a single quantum state such that when one is observed to be spin-up, the other one will always be observed to be spin-down and vice versa, even though according to quantum mechanics it is impossible to predict which set of measurements will be observed. As a result, measurements performed on one particle seem to be instantaneously influencing other particles entangled with it. Albert Einstein, among others, was studying this phenomena and referred to it as "spooky action at a distance" whereas now, for quantum computing, entanglement is a resource.

Entangled state is a multi-qubit state that cannot be expressed as the tensor product of 1-qubit states of each its qubits. An example of entangled states are the Bell states:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \quad (7)$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle), \quad (8)$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle), \quad (9)$$

$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle). \quad (10)$$

3.5. Summary

In this chapter we have briefly introduced basic notions of quantum computing. We explained what is a qubit, a quantum gate, measurement and entanglement. This is a basic introduction to issues raised in the next chapters.

4. How do quantum computers work?

Quantum computers are concern of many scientists nowadays. They pose a future of computation, however, there are a number of technical challenges in building them. The most important seems to be achieving a balance between long coherence time, connectivity between qubits and fast control of them.

4.1. General concept of quantum computers

4.1.1. What is quantum computer?

Quantum computers are potentially incredibly powerful machines that take a new approach to processing information. They make use of quantum-mechanical phenomena (e.g. superposition and entanglement) to operate on data.

How did it all begin? In 1981 Richard Feynman, an American theoretical physicist, in his talk at the First Conference on the Physics of Computation, held at MIT, observed that it appeared to be impossible in general to simulate an evolution of a quantum system on a classical computer in an efficient way. He proposed a basic model for a quantum computer that would be capable of such simulations. It was the beginning of quantum computing [22].

Quantum computing uses the possibilities that the laws of quantum mechanics give us to solve computational problems. Classical computers only use a small subset of these possibilities. They compute in the same way that people compute by hand. In this case "efficiently", refers to the idea that the evaluation time doesn't grow too quickly with the size of the input. In quantum computing calculations are performed by unitary transformations on the state of the qubits. Combined with the principle of superposition, this creates possibilities that are not available for classical computing. The quantum algorithms are faster than classical ones thanks to:

- **quantum parallelism:** by using superposition of quantum states, the computer is executing the algorithm on all possible inputs at once;
- **dimension of quantum Hilbert space:** the "size" of the state space for quantum system is exponentially larger than corresponding classical system;

- **entanglement capability:** different qubits in a quantum computer become entangled, exhibiting nonclassical correlations.

Quantum computers differ in physical implementation. That means there are many ways to build physical systems that realize the qubits. In this chapter we will describe what physical laws and phenomena are used and what conditions are required to build a quantum computer.

4.1.2. Physical phenomena used to build quantum computers

There are several approaches proposed when it comes to building a quantum computer. One of them is an **ion trap**. Ions (particles having non-zero electrical charge, such as an atom whose total number of electrons is not equal to its total number of protons) are confined and suspended using electromagnetic fields. Qubits are stored in stable electronic states of each ion, and quantum information can be transferred through the collective quantized motion of the ions in a shared trap [23].

NMR (nuclear magnetic resonance) is another approach to building quantum computers. It uses the spin states of nuclei within molecules as qubits. NMR differs from other implementations of quantum computers in that it uses an ensemble of systems, in this case molecules, rather than a single pure state.

The last approach presented here will be the **superconducting quantum computing**. This is an implementation of a quantum computer in superconductors (materials in which a phenomenon of exactly zero electrical resistance and expulsion of magnetic flux fields occur) [5, 6]. IBM conducts intense research in superconducting quantum computing. Their quantum machine is build using this approach.

4.2. Quantum computer based on superconducting qubits

4.2.1. Necessary conditions to build a quantum computer

Quantum phenomena are hard to observe in everyday life. That's why the physical implementation of qubits and quantum gates is difficult. A number of conditions need to occur to actually implement the qubit.

One of them is extremely low operation temperature. Each material has different temperature in which it becomes a superconductor. For aluminium (IBM's quantum computer is made

of aluminium) it is 1.2 K. Yet, to build a quantum computer, aluminium has to be cooled to even lower temperature, about 15-20 mK.

Electrons inside a quantum system occupy certain allowed orbitals with a specific energy. It means that the energy of an electron is not continuous, but quantized. Energies corresponding to each of the allowed orbitals are called **energy levels**. The energy separation between various levels can be equally split, yet to build a quantum computer the differences between those levels have to be distinct. Let's suppose that three of energy levels are $|0\rangle$, $|1\rangle$ and $|2\rangle$ states. Energy difference between $|0\rangle$ and $|1\rangle$ state is different than between $|1\rangle$ and $|2\rangle$.

All particles pulse. Physicists call this pulsation **thermal fluctuations**. Thermal fluctuations are random deviations of a system from its average state. If the temperature is higher, they are bigger. Even in absolute zero electrons are not still. Their energy is called energy of thermal fluctuations and is of the order of:

$$E_T = k_B T, \quad (11)$$

where k_B is Boltzmann constant and T is temperature.

Let's find out what is the frequency of fluctuations of electrons in the temperature of 15 mK. To know that we need one more formula:

$$E = h\nu, \quad (12)$$

where h is Planck constant and ν is frequency.

$$\nu = \frac{E}{h} = \frac{k_B T}{h}. \quad (13)$$

We calculated that in 15 mK the frequency of fluctuations of electrons is 0.3 GHz. Analogously, for 1 K frequency value is a lot bigger - 20 GHz.

Electrons can "jump" between energy levels if the frequency of thermal fluctuations is bigger than frequency needed to put an electron on a different energy level. This phenomenon can also be controlled by laser with a proper frequency, that can excite an atom e.g. from state $|0\rangle$ to $|1\rangle$. Important information is that thanks to distinct energy separation between levels, the laser with the same frequency can never put the atom in, e.g. state $|2\rangle$.

In IBM's quantum computer frequency of fluctuations amounts to 5 GHz. It means that the

energy separation between energy levels in the IBM-Q must be big enough to make sure that thermal fluctuations would not change the energy level of the electron, or a rather Cooper pair.

Temperature in a quantum computer needs to be small enough, so that energy of thermal fluctuations is much smaller than energy between energy levels of states $|0\rangle$ and $|1\rangle$ not to cause random excitations. If metal is cooled to critical temperature, it goes from a state where it has electrical resistance, to the superconducting state, where there is no resistance to the flow of direct electrical current. What happens there is that the electrons in the metal become bound into so-called Cooper pairs. Above the critical temperature, the net interaction between two electrons is repulsive. Below the critical temperature, though, the overall interaction between two electrons becomes very slightly attractive [5].

Important metrics for the qubits are two time values: T_1 and T_2 . The relaxation time T_1 is the time required for a qubit to relax from the first excited state to the ground state (this process involves energy loss). The dephasing time T_2 is the time over which the phase difference between two eigenstates becomes randomized [5].

4.2.2. Superconducting qubits

Superconducting quantum computing approach assumes using standard circuit elements (such as wires, inductors and capacitors) to create a circuit that has energy levels' structure described above necessary for implementing a qubit [24].

A distinguishing feature of superconducting quantum circuits is the usage of a Josephson junction. Classical circuits (LC) have equally distributed energy levels and because of that they are not good for representing qubits. Josephson junction needs to be introduced to change it: differences between each of the energy levels have to be different.

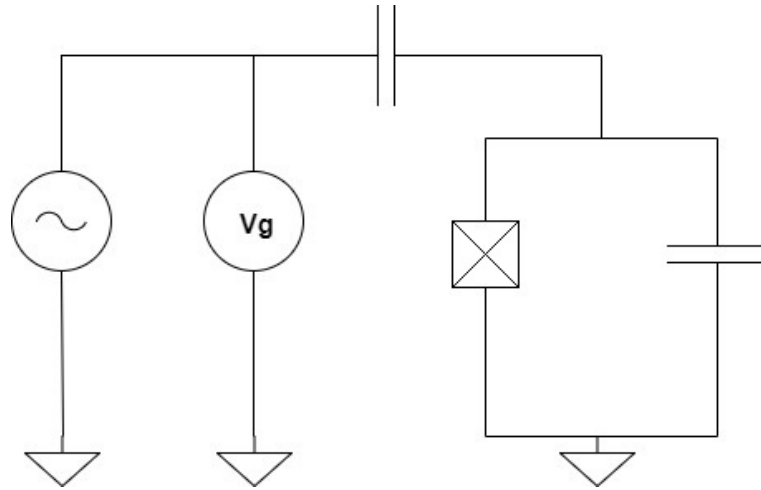


Figure 1: Superconducting quantum computing approach assumes using standard circuit elements (such as wires, inductors and capacitors) to create a circuit that behaves like an atom. A distinguishing feature of superconducting quantum circuits is the usage of a Josephson junction. The figure shows circuit representing a qubit (on the right) with Josephson junction that makes gaps between energy levels different from each other and voltage source to control the energy levels (we can use AC or DC voltage source).

Josephson junction is made by sandwiching a thin layer of a nonsuperconducting material between two layers of superconducting material. Until a critical current is reached, a supercurrent can flow across the barrier (isolator). Electron pairs can tunnel across the barrier without any resistance.

Josephson junction accumulates potential energy E_J (called Josephson energy) when a supercurrent flows through it. It describes the frequency of pairs of electrons' tunneling through Josephson junction. The larger the E_J , the faster the electrons go back and forth through the isolator [24]. E_C is the Coulomb charging energy for the capacitor [6]. In other words: electrostatic

cost (electrostatic energy) that it takes to take a charge from infinity and put it on capacitor.

$$E_C = \frac{e^2}{2C}, \text{ where } C \text{ is the capacity of capacitor.} \quad (14)$$

Important parameter in quantum computing is $\frac{E_J}{E_C}$ ratio. It can be adjusted by engineering the Josephson junction and capacitor in certain way. In various qubit realizations the value of this parameter is different. If $\frac{E_J}{E_C}$ is large (about 50-100) then we call it a transmon qubit and this type of qubit is used in IBM's computer.

4.2.3. How qubits are controlled

Operations on qubits rely on **Rabi oscillations**. Those oscillations appear when qubits are subjected to action of periodic varying electrical or magnetic fields with scrupulously picked influence time. If in time $t = 0$ a qubit is in state $|0\rangle$, the probability $p_{01}(t)$ of transition to state $|1\rangle$ in time t is described by formula:

$$p_{01}(t) = \left(\frac{\omega_1}{\Omega}\right)^2 \sin^2 \frac{\Omega t}{2}, \quad \Omega = \sqrt{(\omega - \omega_0)^2 + \omega_1^2}, \quad (15)$$

the frequency ω_1 is called **Rabi frequency**.

Oscillation amplitude between state $|0\rangle$ and $|1\rangle$ is maximum for $\omega = \omega_0$, that is for *resonance*:

$$p_{01}(t) = \sin^2 \frac{\Omega t}{2}, \quad \omega = \omega_0. \quad (16)$$

To achieve transition from state $|0\rangle$ to state $|1\rangle$ it is sufficient to use rotating field for t equals:

$$\frac{\omega_1 t}{2} = \frac{\pi}{2}, \quad t = \frac{\pi}{\omega_1}. \quad (17)$$

4.2.4. How measurement is done

Measurements of the states of qubits are made with a superconducting quantum interference device (SQUID) [6]. This device, which consists of two Josephson junctions in parallel, is the most sensitive magnetic-flux detector known. For example, one can make a voltmeter that can measure picovolts. That's about 1,000 times more sensitive than other available voltmeters.

4.2.5. Fault-tolerant architecture

Quantum error correction (QEC) helps to perform fault-tolerant quantum computing and it was created by researchers from IBM and was described in [11]. The idea of QEC is to encode information in subsystems of a larger physical space that are immune to noise. A fault-tolerant quantum computing system is shown in figure 2.

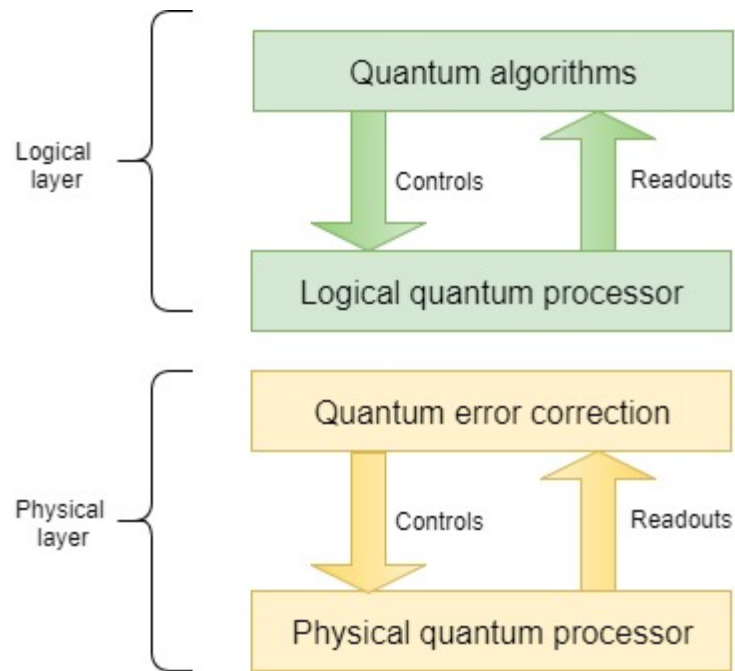


Figure 2: Quantum error correction (QEC) helps to perform fault-tolerant quantum computing. The idea in QEC is to encode information in subsystems of a larger physical space that are immune to noise. In figure there is a full fault-tolerant quantum computing system envisioned within a layered structure. The system consist of two layers: physical qubit layer and logical qubit layer. The lower layer contains physical qubits controlled via a QEC processor. The upper layer functions through control of the physical layer.

The system consist of two layers: the physical qubit layer and the logical qubit layer. The lower layer contains physical qubits controlled via a QEC processor (a classical processor that uses measurement outcomes of the physical qubits to realize a QEC code). The processor keeps track of the physical errors that arise, and implements the appropriate feedback on the controls of the physical qubits. The upper layer functions through control of the physical layer. Here, logical qubits are encoded within the fully error-corrected system of physical qubits, and logical controls and readouts are governed through a processor that determines how to implement difficult quantum algorithms.

4.3. Quantum volume concept

As quantum computers started to become more and more powerful, scientists were thinking about criteria by which their "power" could be measured and compared. Unlike for the classical computing, measuring the speed of computing itself is not enough to define a quantum computer.

IBM researchers created a parameter that describes power of quantum computer called "quantum volume" [25]. The quantum volume measures the useful amount of quantum computing done by a device in space and time. It is not dependent only on number of qubits, but also on error rate. Three factors that influence the error rate are: the accuracy of each operation, number of operation it takes to solve a particular problem and how processor performs these operations. Quantum volume accounts for all of these.

Increasing qubit number does not improve a quantum computer if error rate is high, but improving the error rate will result in a more powerful quantum computer. Quantum volume abstracts the hardware-provided gate set, the qubit connectivity graph, varying fidelities of different operations, possibilities for circuit-rewriting and optimization, available parallelization of operations, etc. It is done by specifying a model algorithm.

How to calculate quantum volume?

First its necessary to know what is the circuit depth. This is a concept created by IBM researchers, that starts with the idea that, because quantum gates can always introduce an error, there is a maximum number of operations that can be performed before it is unreasonable to expect the qubit state to be correct. Circuit depth is that number, multiplied by the number of qubits.

We need to define ϵ_{eff} as the equivalent per-gate error rate that would lead to the same overall error rate, n as number of qubits required to run the algorithm, $d \simeq 1/(n\epsilon_{\text{eff}})$ as achievable circuit depth needed to execute the algorithm with reasonable fidelity to the correct answer and n' (a subset of n) as a number of active qubits, on which to execute the model algorithm. The quantum volume, V_Q , is calculated as:

$$V_Q = \max_{n' \leq n} \min \left[n', \frac{1}{n' \epsilon_{\text{eff}}(n')} \right]^2. \quad (18)$$

This metric quanties the space-time volume occupied by a model circuit with random two- qubit gates that can be reliably executed on a given device.

4.4. Overview of IBM quantum computers in the IBM-Q

4.4.1. History of the IBM-Q

The IBM Quantum Experience (QX) platform was made available in May 2016. It is the first quantum computing platform delivered via the IBM Cloud and accessible by desktop and mobile devices. It enables users to run experiments on the IBM's quantum processor. At the beginning it was only a five qubit quantum processor (ibmqx1 backend) and a simulator. User could only interact with them through a quantum composer. In July the same year IBM launched the community forum, a place where QX users can exchange their knowledge and help each other. On 24th January 2017 IBM launched ibmqx2 backend (5 qubits). Also simulator was expanded to custom topologies up to twenty qubits and quantum language OpenQASM version 2.0 was introduced. User could interact with the processor either through the composer or using OpenQASM. In September 2017 new backends went online: a 5-qubit ibmqx4 and a 16-qubit ibmqx5. In November 2017 IBM Research team announced that they successfully built and tested two more machines [26]:

- one with 20 qubits, which was going to be available to clients of IBM at the end of 2017,
- a prototype with 50 qubits.

Researchers from IBM writes on their blog that the 20-qubit machine has double the coherence time, at an average of 90 μ s, compared to previous generations of quantum processors with an average of 50 μ s. The machine is also designed to scale. The 50-qubit prototype has similar performance.

IBM Quantum Experience platform is constantly developed.

4.4.2. IBM-Q quantum computer's architecture

IBM-Q uses superconducting quantum computing approach. IBM has revolutionized an approach to quantum computers and nowadays everyone can use them. IBM-Q platform contains three types of backend architecture. Two of them are composed of 5 qubits and one is composed of 16 qubits. Descriptions below depict all currently available backends.

The IBMQX2 backend is the first that appeared on the IBM-Q [27]. It went online January 24th 2017. The connectivity map for the CNOTs in this device is:

$$\text{coupling_map} = 0 : [1, 2], 1 : [2], 3 : [2, 4], 4 : [2],$$

where $a : [b]$ means a CNOT with qubit a as control and b as target can be implemented. The connection topology is shown in figure 3.

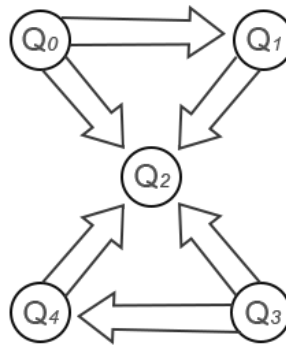


Figure 3: Schema of connection topology of IBMQX2 architecture, one of the IBM-Q's backends. Arrows show which qubit in a pair is a control qubit and which is a target qubit (from CNOT gate).

The second 5-qubit backend, IBMQX4, went online 25th September 2017 [28]. It has a different coupling map than the previous one. Connections between qubits are shown in figure 4.

$$\text{coupling_map} = 1 : [0], 2 : [0, 1], 3 : [2, 4], 4 : [2].$$

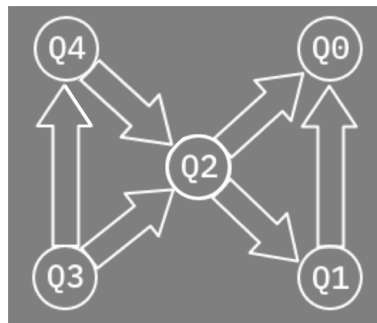


Figure 4: Schema of connection topology of IBMQX4 architecture, one of the IBM-Q's backends. Arrows show which qubit in a pair is a control qubit and which is a target qubit (from CNOT gate).

IBM also created a 16-qubit backend, IBMQX5. It went online 28th September 2017 [29]. Connection topology is shown in figure 5.

$$\text{coupling_map} = \{1 : [2], 2 : [3], 3 : [4, 14], 5 : [4], 6 : [7], 6 : [11], 7 : [10], \\ 8 : [7], 9 : [8, 10], 11 : [10], 12 : [5, 11, 13], 13 : [4, 14], 15 : [0, 2, 14]\}$$

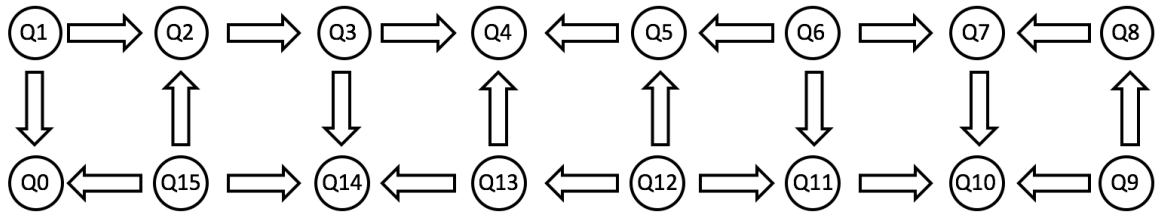


Figure 5: Schema of connection topology of IBMQX5 architecture, one of the IBM-Q's backends. Arrows show which qubit in a pair is a control qubit and which is a target qubit (from CNOT gate).

4.4.3. Parameters of the IBM-Q

Table 1 presents the most important parameters of quantum computers with their general values (most common values for typical quantum computers these days) and values for the IBM-Q.

Parameter	General value	IBM-Q's value
Coherence time T_{coh}	100 μ s	90 μ s
Gate length T_G	10-100 ns	10 ns
Number of quantum operations	$\frac{T_{coh}}{T_G}$	about 10^4
$E_1 - E_0$	4-6 GHz	5 GHz
$E_2 - E_1$	about 5% different than $E_1 - E_0$	

Table 1: Most important parameters of quantum computers. It shows a general value for each parameter and a value for the IBM-Q. The general value is a most common value in quantum computers these days.

Coherence time is a time of survival of a quantum state. In other words, time in which quantum state is unimpaired. *Gate length* is a time in which we can change a qubit state (by performing an operation). The key is to have a quantum state live longer than it takes to change state of a qubit. *Number of quantum operation* is a maximum state changes that can be performed. It is calculated by dividing coherence time by gate length. The last two parameters in the table are *differences in energy levels*. In the table ?? there are parameters of physical realization of the IBM-Q.

Parameter	IBM-Q's value
Temperature of operation T	15-20 mK
Material (superconductor)	aluminium
Critical temperature of aluminium T_C	1,2 K
Energy of Cooper pair for aluminium Δ	5 GHz
Dimensions of the device simulating one qubit	about 500 μm

Table 2: The table shows parameters of physical realization of the IBM-Q - properties of materials used to build the IBM-Q

Temperature of operation is the temperature in which the quantum computer works. It shall be cooled to freely operate between quantum energy levels. The IBM-Q's superconducting circuits are build of aluminium. *Dimensions* depicts the size of the actual device (that simulates one qubit).

5. Software environment of the IBM-Q

IBM has revolutionized an approach to quantum computers by providing by providing an open access to their quantum computer. Nowadays they are no longer machines for scientists from IBM and collaborators, but can be used by everyone. In this chapter we will describe and assess the IBM-Q software environment [15].

5.1. Methods of creating quantum algorithms on the IBM-Q

There are several ways to create a quantum program that can be executed on IBM's quantum computer (see Fig. 6). The easiest way is to use graphical user interface where circuits can be created from gates with drag-and-drop method.

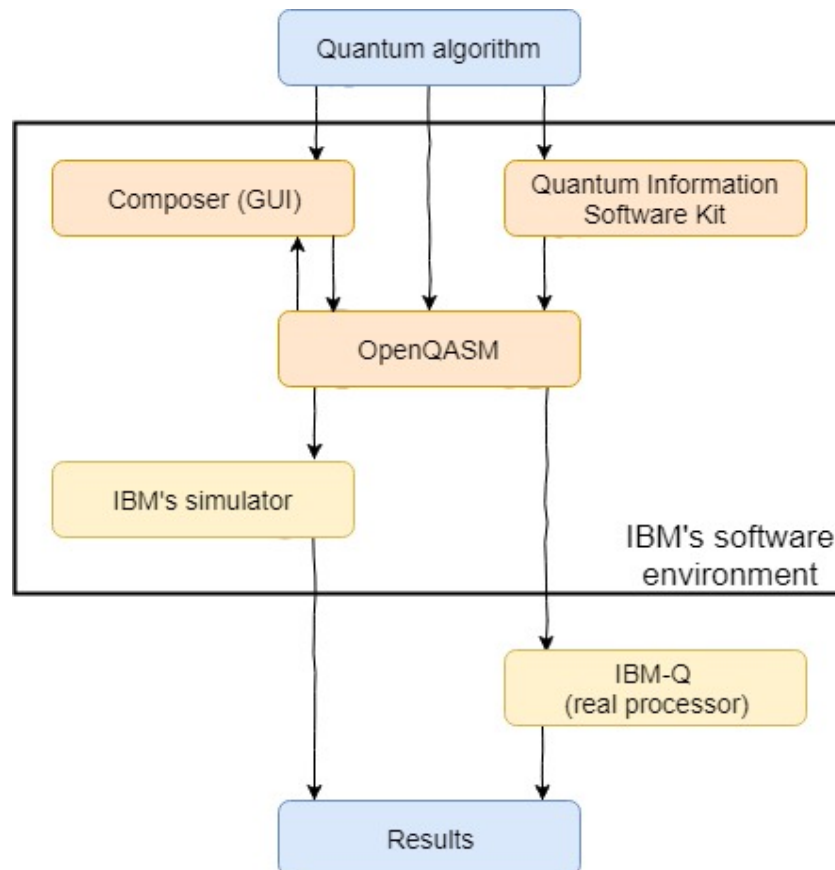


Figure 6: Tools to program the IBM-Q. There are several ways to create a quantum program that can be executed on the IBM's quantum computer. The easiest way is to use graphical user interface. There is also a software development kit called Quantum Information Software Kit (QISKit for short). Both circuit made with GUI and QISKit code can be translated into Qasm code. QASM is a type of assembly language created especially for the IBM-Q. It can be executed on a simulator or a real quantum processor and give some results. Algorithms can also be written directly in QASM.

There is also a software development kit called Quantum Information Software Kit (QISKit for short) [8]. QISKit uses IBM's Python API client to connect to the IBM-Q. Circuits made with GUI and QISKit code can be translated into QASM code. QASM is a type of assembly language created especially for the IBM-Q. It can be executed on a simulator or on a real quantum processor and give some results. Algorithms can also be written directly in QASM.

All of these tools are described below. To present them we will use Grover's algorithm as an example.

5.1.1. IBM-Q's graphical user interface

In Fig. 7 there is a screenshot of the IBM-Q Composer. At the top, there is a list of available backends. It is important to mention, that with GUI users can use only the 5-qubit backends. Inscription "maintenance" indicates that this particular backend is currently being calibrated and is unavailable. Calculations on this architecture will be done when it becomes active. In figure 7 there is an exemplary circuit created from gates listed on the right.

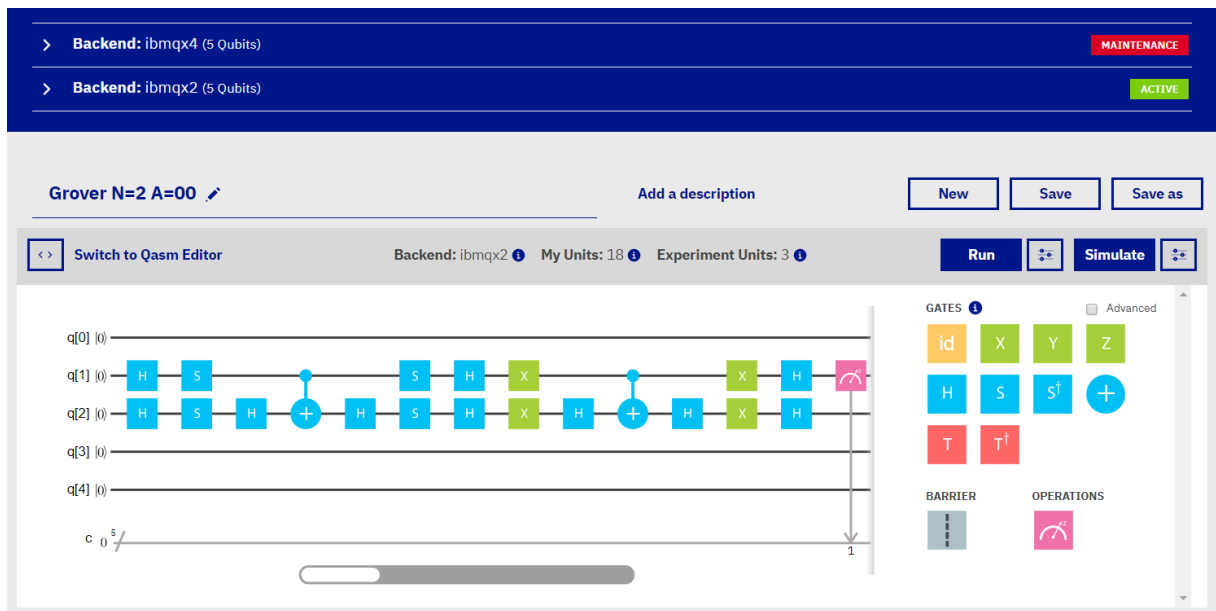


Figure 7: Interface of the IBM-Q Composer [15]. At the top there is a list of available backends. Inscription "maintenance" indicates that this particular backend is currently being calibrated and is unavailable. Calculations on this architecture will be done when it becomes active.

There is a possibility to switch views between composer and QASM editor (shown in Fig. 8). Every instruction in the QASM is the application of a quantum gate. It is possible to begin

building a quantum circuit in the scripting mode. More information about user interface in the IBM-Q can be found in Full User Guide [7] - the documentation of the IBM-Q.

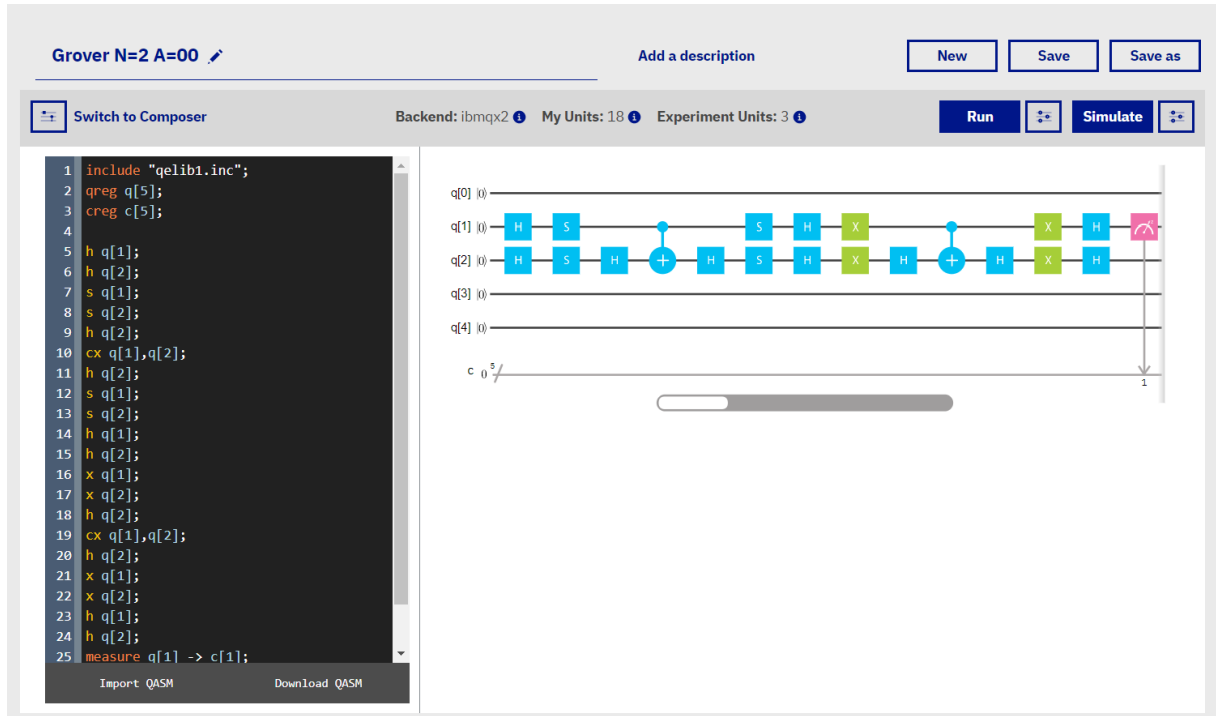


Figure 8: Interface of the IBM-Q QASM editor [16]. Every instruction in the QASM is the application of a quantum gate. It is possible to begin building a quantum circuit from the scripting mode.

After creating a quantum score (circuit), a user can run it on a real quantum processor or simulate it. If we choose the first option, the algorithm is scheduled in a queue and when the execution is finished an email with a proper information is sent to the user. On the other hand, simulation can be done at any time and it permits all-to-all connectivity of qubits.

The system also informs the user if other users have run the same circuit and proposes to present existing results instead of actual run the circuit on quantum processor. That allows users to save Experiment Units. Experiment Units are a kind of credits to execute quantum algorithms on the IBM's processors. Each executions takes some of the Units and they regenerate after 24 hours.

It is also worth to mention that there is a community and a forum where IBM-Q users help each other to solve problems and answer questions they have [30].

5.1.2. QISKit - SDK for working with OpenQASM and the IBM Q

The Quantum Information Software Kit (QISKit) is a software development kit (SDK) for working with OpenQASM and the IBM-Q. To make calculations on online backends, QISKit uses python API client to connect to the IBM-Q [31].

Using the QISKit users can create quantum computing programs, compile them, and execute them on one of several backends (online Real quantum processors, online simulators, and local simulators) [32]. It facilitates a set of functions. All of the functions are described in documentation [33].

As an example of usage of the QISKit, in code listing 1 we present the Grover algorithm implemented in Python with QISKit.

Registers in QISKit are just qubits on which some quantum gates can be executed. Whereas gates are represented by functions (e.g. function `h()` is a representation of Hadamard gate).

Python native instructions and data structures also can help to create a quantum algorithm in QISKit. One of them is `for` loop that can be used to iterate the qubits and perform some operation on each of them. Python's arrays are often used to show connections between qubits. The example also shows a data structure (named `QPS_SPECS`) that provided information about the quantum processor's specifications to the `QuantumProgram()` function.

Code 1: Grover algorithm written in Python with QISKit SDK

```

1 from qiskit import QuantumProgram
2 import Qconfig
3
4 backend = "ibmqx2"
5 coupling_map = {0: [1, 2],
6                 1: [2],
7                 2: [],
8                 3: [2, 4],
9                 4: [2]}
10
11 QPS_SPECS = {
12     "circuits": [{
13         "name": "grover",
14         "quantum_registers": [{
15             "name": "q",
16             "size": 5
17         }],
18         "classical_registers": [
19             {"name": "c",
20              "size": 5}
21         ]
22     }
23 }
24 # set up registers and program
25 qp = QuantumProgram(specs=QPS_SPECS)
26 qc = qp.get_circuit("grover")
27 q = qp.get_quantum_register("q")
28 c = qp.get_classical_register("c")
29
30 # set the APIToken and API url
31 qp.set_api(Qconfig.APIToken, Qconfig.config["url"])
32
33 qc.h(q[1])
34 qc.h(q[2])
35 qc.s(q[1])
36 qc.s(q[2])
37
38 qc.h(q[2])
39 qc.cx(q[1], q[2])
40 qc.h(q[2])
41
42 qc.s(q[1])
43 qc.s(q[2])
44 qc.h(q[1])
45 qc.h(q[2])
46
47 qc.x(q[1])
48 qc.x(q[2])
49
50 qc.h(q[2])
51 qc.cx(q[1], q[2])
52 qc.h(q[2])
53
54 qc.x(q[1])
55 qc.x(q[2])
56
57 qc.h(q[1])
58 qc.h(q[2])
59
60 for i in range(2):
61     qc.measure(q[i], c[i])
62
63 print("map to %s, backend" % backend)
64 result = qp.execute(["grover"], backend=backend,
65                     coupling_map=coupling_map, shots=1024, timeout=120)
66 print(result)
67 print(result.get_counts("grover"))

```

5.1.3. Assembly language - OpenQASM

The same algorithm can be implemented directly in OpenQASM. User can toggle between Composer (GUI) and the QASM editor on the IBM-Q platform.













Grover algorithms's code implemented in Open QASM is shown in code listing 2.

Code 2: OpenQASM

```
1 include "qelib1.inc";
2 qreg q[5];
3 creg c[5];
4
5 h q[1];
6 h q[2];
7 s q[1];
8 s q[2];
9 h q[2];
10 cx q[1],q[2];
11 h q[2];
12 s q[1];
13 s q[2];
14 h q[1];
15 h q[2];
16 x q[1];
17 x q[2];
18 h q[2];
19 cx q[1],q[2];
20 h q[2];
21 x q[1];
22 x q[2];
23 h q[1];
24 h q[2];
25 measure q[1] -> c[1];
26 measure q[2] -> c[2];
```

5.1.4. Representation of quantum operators in QISKit and the IBM-Q Composer

In table 3 we have put together quantum operators, their counterparts in QISKit's set of functions and the IBM-Q Composer's gates representations.

Operator (matrix representation)	QISKit function	IBM-Q gates
$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	iden()	
$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	x()	
$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	y()	
$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	z()	
$H = \begin{bmatrix} 1 & 1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & -1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$	h()	
$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	s()	
$S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$	sdg()	
$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	cx()	
$T = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix}$	t()	
$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1-i}{\sqrt{2}} \end{bmatrix}$	tdg()	
$U_1 = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}$	u1()	
$U_2 = \begin{bmatrix} 1 & e^{i\lambda} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ e^{i\phi} & e^{i\lambda+i\phi} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$	u2()	






Operator (matrix representation)	QISKit function	IBM-Q gates
$U_3 = \begin{bmatrix} \cos \frac{\theta}{2} & -e^{i\lambda} \sin \frac{\theta}{2} \\ e^{i\phi} \sin \frac{\theta}{2} & e^{i\lambda+i\phi} \cos \frac{\theta}{2} \end{bmatrix}$	u3()	
barrier	barrier()	
measurement	measure()	
if		
qubit reset	reset(quantum_reg)	

Table 3: Table shows quantum operators, their counterparts in QISKit's set of functions and the IBM-Q Composer's gates representations.

The barrier operator prevents optimizations that normally occur in software when rearranging gates is possible, e.g., in a circuit with no barriers two Hadamard gates placed one after the other will be combined into an identity and no gate will be applied, whereas, if those two Hadamard gates are separated by a barrier, both of them will be physically implemented.

5.2. IBM-Q's simulator

IBM provides a simulator of quantum computer that can be used after choosing "Custom Topology" option during creating new experiment. User must give number of qubits to be simulated (maximum is 20). Custom processor permits all-to-all connectivity, whereas in real processors connectivity is limited by physical connections. The physical topology and qubit connections were describes in subsection 4.4.2. In simulation the execution of quantum circuits happens immediately, in contrary to running circuits on a real processor which is queued and asynchronous.

5.3. Results from the simulator and from the real quantum processor

After finishing execution of the algorithm we can see the results. It is easy to notice that results from the real processor and from the simulator differ from each other, even though they are both results of the same algorithm.

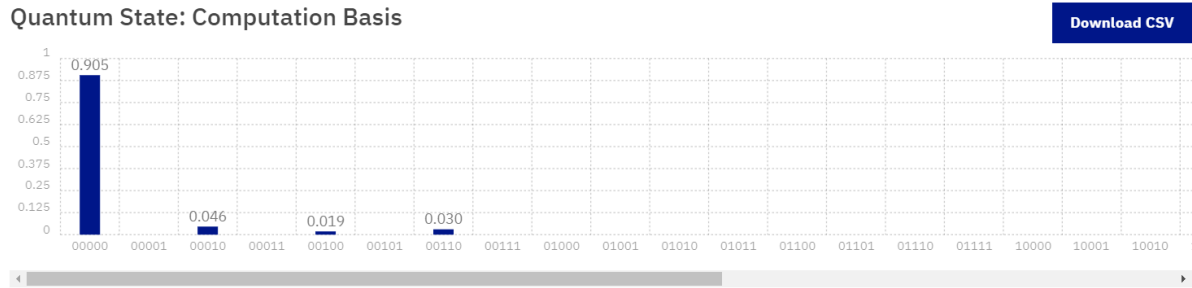


Figure 9: Results of the Grover algorithm execution on the real quantum processor.

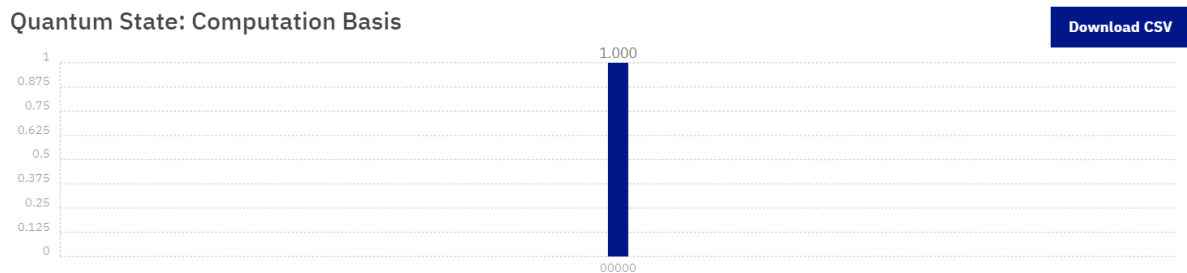


Figure 10: Results of the Grover algorithm execution on the IBM simulator.

Fig. 9 shows output of Grover algorithm executed on the real processor, whereas in the Fig. 10 there are results of the same algorithm run on the simulator. The results from both runs are different. That is because simulator imitates an "ideal" machine and the real processor can be subjected to decoherence.

5.4. Matching algorithm with architecture

A true challenge during creating quantum algorithms is matching the algorithm with particular architecture. It is crucial to remember that each of the backends has different connections between qubits and it is impossible to run the same circuit on all of the backends.

6. Interoperability of the IBM-Q and QuIDE simulators

6.1. The need for simulators of quantum computers

Simulators compute the quantum state we expect a circuit to produce at each stage of processing. They are important when we want to test if the output we got from the real hardware is correct. A simulator can be ideal or realistic. The ideal simulator treats each gate as a unitary matrix and composes them to find the output state. This simulator shows what to expect when all of the operations are perfect. The realistic simulator, on the other hand, numerically solves a system of differential equations, known as a master equation which includes dissipation and phase noise, as well as time-dependent terms for gates and interactions between adjacent qubits. The various interaction strengths are computed from realistic parameters and an effective Hamiltonian. The results are qualitatively similar to what is observed in a typical experiment [34].

As it was explained at the beginning of this thesis, a simulator is even better if it gives a possibility to improve quantum programs and debug them. It can be done by implementing in simulators an option to check inner states of qubits in every step of an algorithm. Unfortunately, the IBM-Q simulators don't have this feature.

6.2. Comparison of the IBM-Q and QuIDE software environments

Before we compare simulators of the IBM-Q and the QuIDE we will review their software environments. In the table 4 there are differences and similarities between environment of the IBM-Q and the QuIDE.

QuIDE	IBM-Q
Non-elementary quantum gates	Only elementary quantum gates (GUI and QASM) or non-elementary quantum gates (QISKit)
C# (.NET Framework)	Graphical interface and QASM or Python (QISKit)
Ability to create quantum programs with code (C# - QuIDE library) as well as with Circuit Designer	Ability to create quantum programs with code (QASM or Python QISKit framework) as well as with Composer
Shows values of registers, probability and amplitude after each step of circuit execution	Shows results after execution of the whole circuit
One library	Multiple frameworks
Transformation from code to circuit and vice versa	
Ability to create reusable subroutines	

Table 4: Differences and similarities between software environment of the IBM-Q and the QuIDE. The set of gates is different in the QuIDE and the IBM-Q. In the QuIDE there is much more gates. Therefore, it would be difficult to map every gate from the QuIDE to a gate from the IBM-Q, however it can be a lot easier in the opposite way.

The set of gates is different in the QuIDE and the IBM-Q. In the QuIDE there is much more gates, therefore, it would be difficult to map every gate from the QuIDE to a gate from the IBM-Q, however, it can be a lot easier in the opposite way.

IBM-Q's GUI and QASM provide ability to create reusable subroutines (nested elementary gates) which technically are multiple-qubit gates. They can be used only with custom topology (not on real devices). The QuIDE also allows to create function that are reusable subroutines.

6.3. Comparison of the IBM-Q and QuIDE simulators

As there are no publications that say how the IBM-Q's simulator is implemented, the only comparison we can make is that comparison of what is visible outside. IBM-Q do not try to provide best in class simulator as it is meant to be only aid to write quantum algorithms and run them on real computers.

QuIDE	IBM-Q
Implemented with hash table technique (dictionary variant)	n.a.
Implemented in C#	n.a.
Optimization: only non-zero amplitudes are kept in memory	n.a.
Standalone application, can run only on Windows (as written in .NET Framework)	Web application, can run on every web browser
Up to 23 in the QuIDE and 26 in QuIDE.dll qubits simulated	Up to 20 qubits simulated

Table 5: Differences between the QuIDE and IBM-Q simulator.

As it was mentioned in subsection 6.1, there are two types of quantum simulators: realistic and ideal. IBM-Q is definitely a realistic simulator, because two executions of the same algorithm can give slightly different results. On the other hand, the QuIDE simulator gives always identical results. Its implementation makes it an ideal simulator that does not take account of decoherence.

Table 5 shows differences between the QuIDE and the IBM-Q simulator. Some information is missing, so we cannot really compare every aspect of both simulators.

6.4. Converter from QASM to C#

Inspired by the fact that IBM's simulator resembles a real hardware and it is impossible to check states of qubits at the individual steps of an algorithm, we have created a converter which converts QASM code (transcript of a quantum circuit) to C# code that can be run on the QuIDE simulator. QuIDE gives a user a possibility to preview the internal state of a quantum system at each stage of the computation. This is impossible on real quantum systems (as well as on IBM's simulator) because each observation interfere with its state.

In the QuIDE users can create quantum circuits with Circuit Designer (similar to the Composer in the IBM-Q) as well as by writing code in C# with the QuIDE libraries. The output c# code that our converter generates can be also transformed into graphical quantum circuit in the QuIDE.

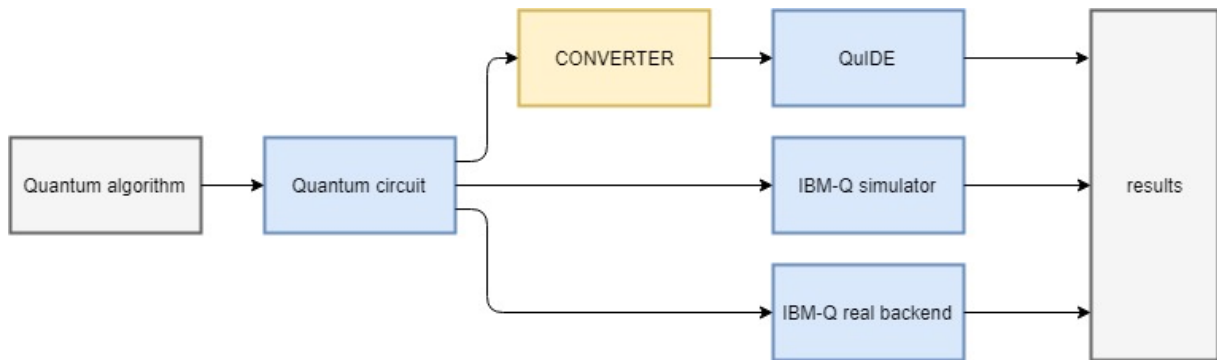


Figure 11: The converter converts QASM code (transcript of a quantum circuit) to C# code that can be run on the QuIDE simulator. QuIDE gives the user a possibility to preview the internal state of a quantum system at each stage of the computation. This is impossible on real quantum systems (as well as on the IBM's simulator). Before execution we have to make sure that we have a proper qasm file with algorithm we want to convert to C#.

Before execution we have to ensure that we have a proper qasm file with algorithm we want to convert to C#. The converter works as showed in algorithm 1.

Algorithm 1 Description of how the converter works

- 1: Program reads a qasm file of which name is given as an input argument at the program execution.
 - 2: Output file (the C# QuIDE file) is being created (it is empty at this stage).
 - 3: Program writes necessary imports of libraries into output file as well as it creates Quantum-Computer instance inside Main() function.
 - 4: The qasm file is parsed line after line.
 - 5: **repeat**
 - 6: Name of the gate is extracted from the line of qasm code.
 - 7: Name of a quantum register is extracted from the line of qasm code. On this register gates will be executed.
 - 8: Parameters of gates are extracted.
 - 9: A corresponding QuIDE function is being found. The function needs to act in the same way as the gate in the parsed line.
 - 10: All extracted arguments are mapped to fit in the QuIDE function.
 - 11: A string with the code of the function with proper parameters is being written into output file.
 - 12: **until** end of file
 - 13: Program closes the output file.
-

The output file can be opened and run in the QuIDE simulator. In step 3 of the Algorithm 1 it was mentioned that necessary imports of libraries are being written into the output file. Those are the libraries that include definitions of functions representing quantum gates as well as the libraries with mathematic functions and matrix representations. In step 8 the parameters that are extracted from the line of QASM code depend on the type of the gate located in that line. In every gate there is a parameter that represents a number of the qubit to perform the execution of the gate on, but other parameters differ depending on gate.

QISKit function	QASM function	QuIDE function
iden(q[i])	id q[i]	-
x(q[i])	x q[i]	q.SigmaX(i)
y(q[i])	y q[i]	q.SigmaY(i)
z(q[i])	z q[i]	q.SigmaZ(i)
h(q[i])	h q[i]	q.Hadamard(i)
s(q[i])	s q[i]	q.PhaseKick(Math.PI/2, i)
sdg(q[i])	sdg q[i]	q.PhaseKick(-Math.PI/2, i)
cx(q[i], q[j])	cx q[i], q[j]	q.CNot(target:j,control:i)
t(q[i])	t q[i]	q.PhaseKick(Math.PI/4, i)
tdg(q[i])	tdg q[i]	q.PhaseKick(-Math.PI/4, i)
u1(lam,q[i])	u1(lam) q[i]	q.PhaseKick(lam, i)
u2(phi,lam,q[i])	u2(phi,lam) q[i]	q.Gate1(U2, i)
u2(theta,phi,lam,q[i])	u3(theta,phi,lam) q[i]	q.Gate1(U2, i)
barrier(*tuples)	barrier q[i], ..., q[j]	-
measure(q[i], c[i])	measure q[i] -> c[i]	q.Measure(i)

Table 6: The table shows quantum gates as QISKit functions, their counterparts in QASM and finally the mapping to functions in the QuIDE.

6.5. Validation of the converter

To validate the converter we took three popular quantum algorithms that are already implemented in the IBM-Q and can be found in the User Guide [7]: the Grover algorithm, the Deutsch-Jozsa algorithm and the Shor algorithm. The existing circuits were transformed in the

IBM-Q into QASM code which was then converted into C# code with the converter. From the C# code a circuit in the QuIDE was generated. We compared the circuit from the IBM-Q and the circuit from the QuIDE. If they were identical, it was a proof that the conversion is correct. The next step was to check internal states in the key stages of the examined algorithm. Finally, we compared results from both simulators as well as from the real processor of the IBM-Q.

6.5.1. The Grover algorithm

To test the converter we started from transforming a quite simple algorithm - the Grover's search algorithm.

Algorithm 2 Description of Grover algorithm [7]

- 1: State $|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$ is being prepared, where N - number of items in given list to search, x - item in the list. At $t = 0$ the initial state $|\psi_0\rangle = |s\rangle$.
 - 2: **repeat**
 - 3: Oracle matrix $U_f |x\rangle = (-1)^{f(x)} |x\rangle$ is defined, where f is an oracle function that returns $f(x) = 0$ for all unmarked items and $f(w) = 1$ for the element w that is being searched.
 - 4: Oracle reflection is applied to the state $U_f |\psi_t\rangle = |\psi_{t'}\rangle$
 - 5: Additional reflection $U_s = 2|s\rangle\langle s| - 1$ about the state $|s\rangle$ is applied. It maps the state $|s\rangle$ to $U_s |s\rangle$ and completes the transformation $|\psi_{t+1}\rangle = U_s U_f |\psi_{t'}\rangle$.
 - 6: **until**
-

The Algorithm 2 describes how the Grover algorithm works and code 2 from subsection 5.1.3 shows an implementation of this algorithm in QASM. It is a very simple case that can be implemented on two qubits, with four possible oracles. Step 1 in the algorithm maps to lines 5, 6 in the code, steps 3 and 4 in the algorithm is realized in lines 7-13 in the code and step 5 is in lines 14-24.

The QASM code of the algorithm presented in code listing 2 is submitted for transformation with the converter. The output C# code is presented in code listing 3.

Code 3: Grover algorithm - code in QuIDE

```

1 using Quantum;
2 using Quantum.Operations;
3 using System;
4 using System.Numerics;
5 using System.Collections.Generic;
6
7 namespace QuantumConsole
8 {
9     public class QuantumTest
10    {
11        public static void Main()
12        {
13            QuantumComputer comp = QuantumComputer.GetInstance();
14            Register q = comp.NewRegister(0,5);
15
16            q.Hadamard(1);
17            q.Hadamard(2);
18            q.PhaseKick(Math.PI/2, 1);
19            q.PhaseKick(Math.PI/2, 2);
20            q.Hadamard(2);
21            q.CNot(target: 2, control: 1);
22            q.Hadamard(2);
23            q.PhaseKick(Math.PI/2, 1);
24            q.PhaseKick(Math.PI/2, 2);
25            q.Hadamard(1);
26            q.Hadamard(2);
27            q.SigmaX(1);
28            q.SigmaX(2);
29            q.Hadamard(2);
30            q.CNot(target: 2, control: 1);
31            q.Hadamard(2);
32            q.SigmaX(1);
33            q.SigmaX(2);
34            q.Hadamard(1);
35            q.Hadamard(2);
36            q.Measure(1);
37            q.Measure(2);
38        }
39    }
40 }
41 }

```

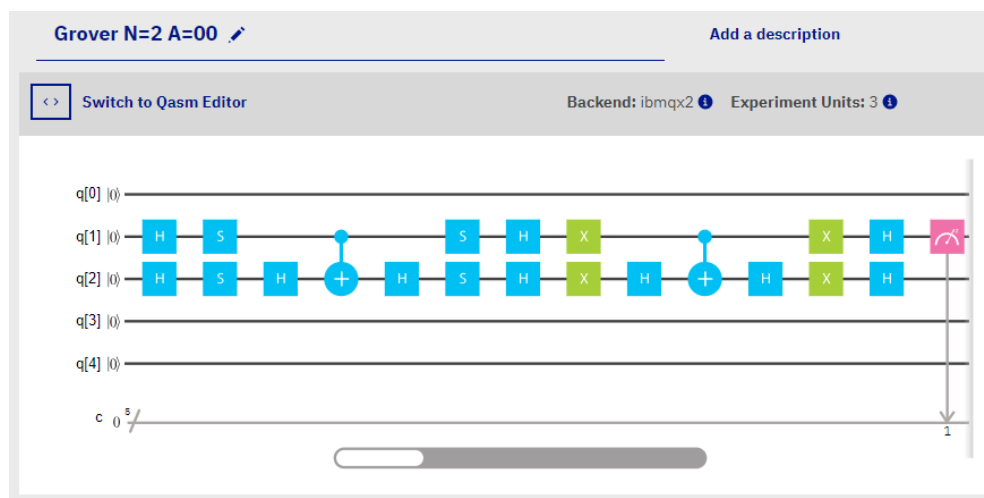


Figure 12: Quantum circuit in the IBM-Q Composer corresponding to the code of Grover algorithm. The algorithm needs only two qubits, so three qubits are not used in this case.

Quantum circuit in the IBM-Q Composer corresponding to the code of Grover algorithm is presented in figure 12. The algorithm needs only two qubits, so three qubits are not used in this case. The QuIDE circuit is presented in figure 13.

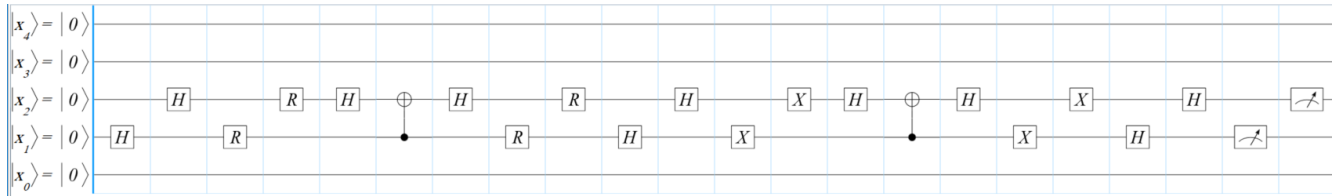


Figure 13: The quantum circuit corresponding to the code of the Grover algorithm in the QuIDE's circuit designer tool generated from the C# code that was converted from QASM code of the Grover algorithm.

We can compare the circuit generated with the QuIDE to the circuit from IBM's Composer. They look almost the same, however there are some differences:

- In the QuIDE some gates have different names (S gate from the IBM-Q is R gate in the QuIDE)
- The QuIDE qubits and the IBM-Q qubits have reversed numbering.

After executing this quantum algorithm on the IBM's simulator, we can see that the probability of getting value $|0\rangle$ is equals to 1. The final results of the execution in the QuIDE look the same as in IBM's simulator, but it also shows the amplitude value $-1.00 - 0.00i$.

As it was mentioned before, the reason why the converter was created is the fact that the QuIDE has a feature that the IBM-Q lacks and this is the possibility to preview of the internal quantum state of the system. In figure 14 there is the same circuit as in figure 13, generated in the QuIDE circuit designer, but "stopped" after step 4 of the algorithm 2.

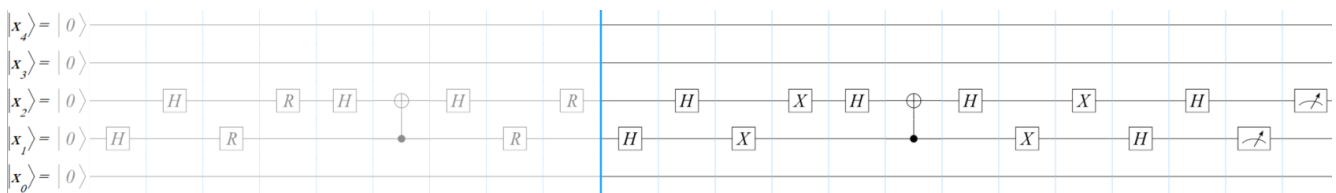


Figure 14: The quantum circuit in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Grover algorithm, the same circuit as in figure 13, "stopped" after step 4 of the algorithm 2.

Figure 15 shows the state of the system exactly in the moment lineated in figure 14. The QuIDE can give users information about value of the registers, states of qubits, probability and

amplitude. It also displays amplitude as a vector in a complex plane. Figure 15 is divided into two parts that present results from one place in algorithm, however the left part shows amplitude on a complex plane of the first state (highlighted in blue) and the right part - of the second state. Directions of arrows are inverted.



Figure 15: The state of the system exactly in the moment lineated in figure 14. QuIDE can give users information about value of the registers, states of qubits, probability and amplitude. It also displays amplitude as a vector in a complex plane. The left part of the figure shows amplitude on a complex plane of the first state (highlighted in blue) and the right part - of the second state. Directions of arrows are inverted.

6.5.2. The Deutsch-Jozsa algorithm

In this subsection we will convert implementation of Deutsch-Jozsa algorithm to code in C# with the converter. It is described in Algorithm 3 and the code listing 4 shows its implementation in QASM.

The Deutsch-Jozsa problem is defined as follows. There is a function $f(x)$ that takes as an input n -bit strings x and returns 0 or 1. The function $f(x)$ is either a constant function that takes the same value $c \in \{0, 1\}$ on all inputs x , or a balanced function that takes each value 0 and 1 on exactly half of the inputs. The goal is to decide whether f is constant or balanced by making as few function evaluations as possible. Classically, it requires $2^{n-1} + 1$ function evaluations in the worst case. Using the Deutsch-Jozsa algorithm, the question can be answered with just one function evaluation. The function f is specified by an oracle circuit U_f , such that $U_f |x\rangle = (-1)^{f(x)} |x\rangle$ (like in the previous section about the Grover algorithm) [7].

Let us suppose we have $n = 3$ and balanced function $f(x) = x_0 \oplus x_1x_2$.

Algorithm 3 Description of Deutsch-Jozsa algorithm [7]

- 1: Initialize n qubits in the all-zeros state $|0, 0, \dots, 0\rangle$. $\triangleright n$ is number of bits in a string.
 - 2: Apply the Hadamard gate H to each qubit.
 - 3: Apply the oracle circuit U_f .
 - 4: Repeat Step 2.
 - 5: Measure each qubit. Let $y = (y_1, \dots, y_n)$ be the list of measurement outcomes.
-

The IBM-Q and the QuIDE initializes their qubits with value 0, so the first point of the algorithm 3 is already fulfilled. Step 2 of the algorithm is realized in lines 7-9 in the code listing 4. Step 3 in algorithm maps to lines 10-13 in the code, step 4 is realized in lines 14-16 in the code and the last step, the measurement is shows in lines 17-19 in the code.

Code 4: Deutsch-Jozsa algorithm - code in QASM

```

1 OPENQASM 2.0;
2 include "qelib1.inc";
3
4 qreg q[5];
5 creg c[5];
6
7 h q[0];
8 h q[1];
9 h q[2];
10 h q[2];
11 z q[0];
12 cx q[1],q[2];
13 h q[2];
14 h q[0];
15 h q[1];
16 h q[2];
17 measure q[0] -> c[0];
18 measure q[1] -> c[1];
19 measure q[2] -> c[2];

```

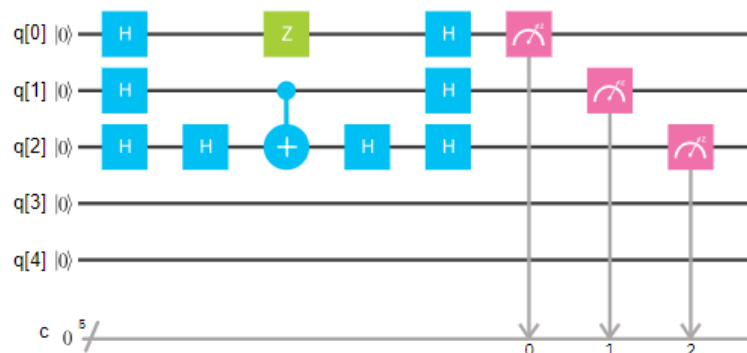


Figure 16: Circuit of the Deutsch-Jozsa algorithm corresponding to the code from listing 4. Two qubits are not used in this case.

The QASM code of the Deutsch-Jozsa algorithm presented in code listing 4 is transformed with the converter into C# code presented in code listing 5.

Code 5: Deutsch-Jozsa algorithm - code in the QuIDE

```

1 using Quantum;
2 using Quantum.Operations;
3 using System;
4 using System.Numerics;
5 using System.Collections.Generic;
6
7 namespace QuantumConsole
8 {
9     public class QuantumTest
10    {
11        public static void Main()
12        {
13            QuantumComputer comp = QuantumComputer.GetInstance();
14            Register q = comp.NewRegister(0,5);
15
16            q.Hadamard(0);
17            q.Hadamard(1);
18            q.Hadamard(2);
19            q.Hadamard(2);
20            q.SigmaZ(0);
21            q.CNot(target: 2, control: 1);
22            q.Hadamard(2);
23            q.Hadamard(0);
24            q.Hadamard(1);
25            q.Hadamard(2);
26            q.Measure(0);
27            q.Measure(1);
28            q.Measure(2);
29        }
30    }
31 }
32

```

Figure 16 shows circuit of the Deutsch-Jozsa algorithm corresponding to the code from listing 4. Again, two qubits are not used in this case. Circuit generated in the QuIDE is shown in figure 17.

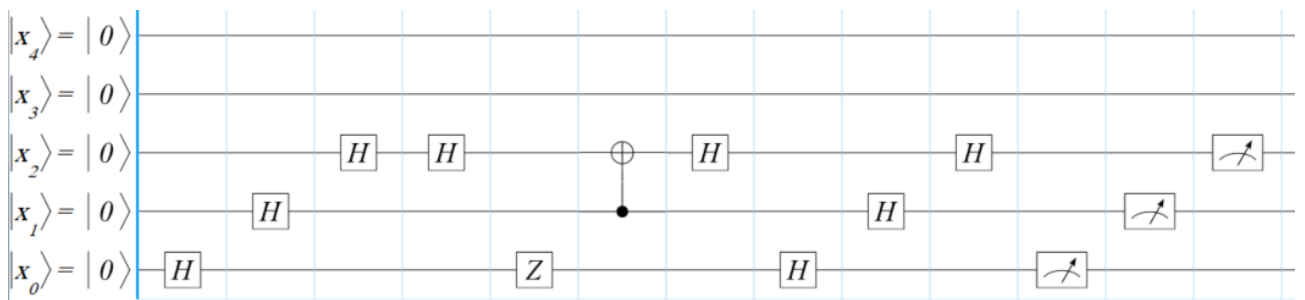


Figure 17: Quantum circuit (corresponding to the code of the Deutsch-Jozsa algorithm) in the QuIDE's circuit designer tool generated from the C# code that was converted from QASM code of the Deutsch-Jozsa algorithm.

DJ N=3 Example

Device: Simulator

Quantum State: Computation Basis

Download CSV

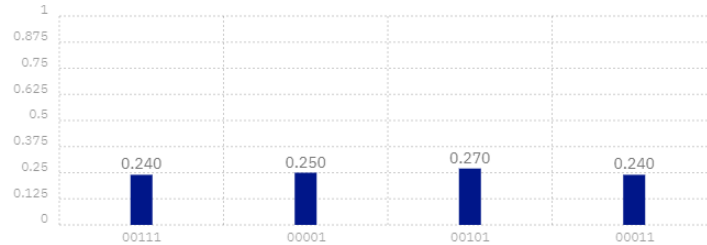


Figure 18: Results of executing the Deutsch-Jozsa algorithm implementation in IBM's simulator.

Value	Qubits	Probability	Amplitude
1>	00001>	P = 0.25	0.50 + 0.00i
3>	00011>	P = 0.25	0.50 + 0.00i
5>	00101>	P = 0.25	0.50 + 0.00i
7>	00111>	P = 0.25	-0.50 + 0.00i

Figure 19: Results of executing the Deutsch-Jozsa algorithm implementation in the QuIDE simulator.

Value	Qubits	Probability	Amplitude
1>	00001>	P = 1	1.00 + 0.00i

Figure 20: Results of executing the Deutsch-Jozsa algorithm implementation in the QuIDE simulator after performing the measurement.

In figure 19 and figure 18 there are results from IBM's simulator and the QuIDE, respectively. The results from both simulators look alike. What is interesting, the outcome from the QuIDE in figure 19 shows the state before the measurement. Results after measurement are depicted in figure 20.

This shows that the simulator of the IBM-Q gives results prior to destroying the quantum state. After performing the measurement we will get one of presented values (in this case 00111, 00001, 00101 or 00011, as shows in figure 18) with given probabilities (0.240, 0.250, 0.270 or 0.240, respectively).

Figure 21 shows the quantum circuit of the Deutsch-Jozsa algorithm generated in the QuIDE circuit designer, "stopped" after step 2 of the algorithm 3.

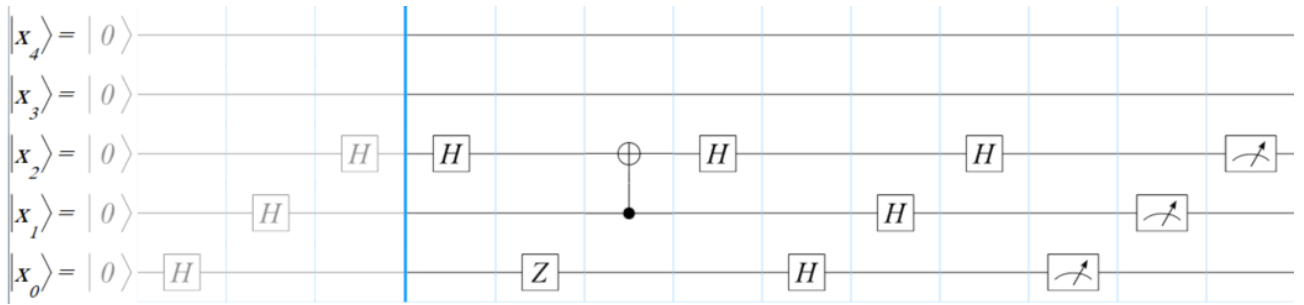


Figure 21: Quantum circuit in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Deutsch-Jozsa algorithm, the same circuit as in figure 17, "stopped" after step 2 of the algorithm 3.

Figure 22 shows the state of the system in the moment marked with a blue line in figure 21. As in case of the Grover algorithm results, we have information about value of the registers, states of qubits, probability, amplitude and its representation as a vector in the complex plane.

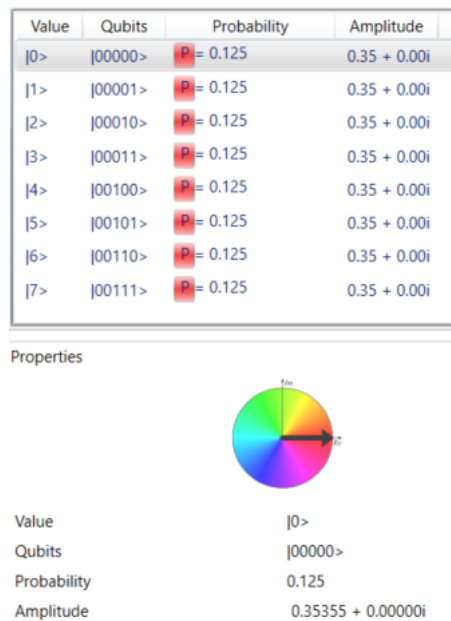


Figure 22: State of the system exactly in the moment lined in figure 21. QuIDE can give users information about value of the registers, states of qubits, probability and amplitude. It also displays amplitude as a vector in the complex plane.

Now we will examine another example. Figure 23 shows the quantum circuit of the Deutsch-Jozsa algorithm generated in the QuIDE circuit designer, "stopped" after step 3 of the algorithm 3.

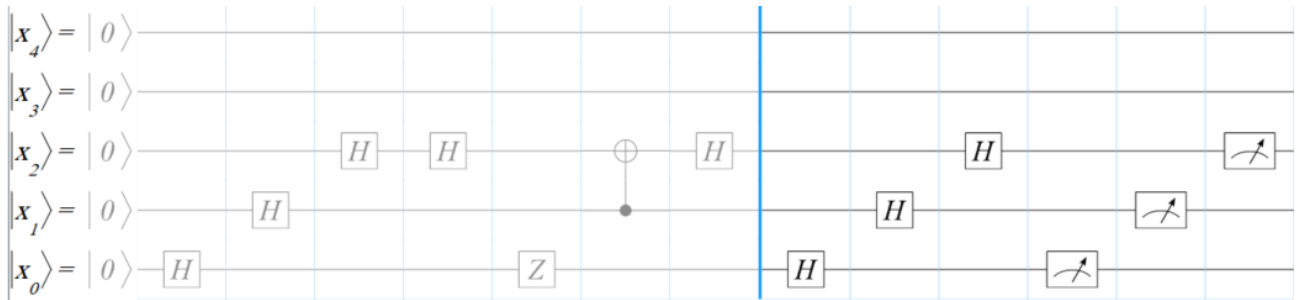


Figure 23: Quantum circuit in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Deutsch-Jozsa algorithm, the same circuit as in figure 17, "stopped" after step 3 of the algorithm 3.

Again, in figure 24 we can see the internal state of the system in the moment marked with a blue line in figure 23. There is information about value of the registers, states of qubits, probability, amplitude and its representation as a vector in a complex plane. Figure 24 shows representation of different amplitudes on a complex plane. The left part shows amplitude of the first state (highlighted in blue) and the right part - of the second state. Directions of arrows are inverted.



Figure 24: State of the system in the moment lineated in figure 23. QuIDE can give users information about value of the registers, states of qubits, probability and amplitude. It also displays amplitude as a vector in a complex plane. The left part of the figure shows amplitude on a complex plane of the first state (highlighted in blue) and the right part - of the second state. Directions of arrows are inverted.

6.5.3. The Shor algorithm

Algorithm 4 Description of Shor algorithm [7][35]

- 1: Choose a random positive integer m . Use the polynomial time Euclidean algorithm to compute the greatest common divisor $gdc(m, N)$. ▷ N is the number we want to factor.

 - 2: **if** $gdc(m, N) \neq 1$ **then**
 - 3: Exit. ▷ We have found a non-trivial factor of N .
 - 4: **end if**
 - 5: Use a QUANTUM COMPUTER to determine the unknown period r of the function

$$\mathbb{N} \xrightarrow{f_N} \mathbb{N}$$

$$a \rightarrow m^a \bmod N$$
 - 6: **if** r is an odd integer **then**
 - 7: **go to** 5
 - 8: **else if** r is an even integer **then**
 - 9: **go to** 11
 - 10: **end if**
 - 11: Since r is even,

$$\left(m^{\frac{r}{2}} - 1\right)\left(m^{\frac{r}{2}} + 1\right) = m^r - 1 = 0 \bmod N.$$
 - 12: **if** $m^{\frac{r}{2}} + 1 = 0 \bmod N$ **then**
 - 13: **go to** 1
 - 14: **else if** $m^{\frac{r}{2}} + 1 \neq 0 \bmod N$ **then**
 - 15: **go to** 17
 - 16: **end if**
 - 17: Use the Euclidean algorithm to compute $d = gdc\left(m^{\frac{r}{2}}, N\right)$. Since $m^{\frac{r}{2}} + 1 \neq 0 \bmod N$, it can be easily shown that d is a non-trivial factor of N .
 - 18: Exit with the answer d .
-

The last algorithm we want to test is the Shor algorithm. It is a quantum algorithm for integer factorization. As written in [7], Shor's algorithm is arguably the most dramatic example of how the paradigm of quantum computing changed our perception of which problems should be considered tractable.

Code 6: Shor algorithm - code in QASM

```

1 OPENQASM 2.0;
2 include "qelib1.inc";
3
4 qreg q[5];
5 creg c[5];
6
7 x q[4];
8 x q[1];
9 x q[2];
10 x q[3];
11 x q[4];
12 cx q[3],q[2];
13 cx q[2],q[3];
14 cx q[3],q[2];
15 cx q[2],q[1];
16 cx q[1],q[2];
17 cx q[2],q[1];
18 cx q[4],q[1];
19 cx q[1],q[4];
20 cx q[4],q[1];
21 measure q[1] -> c[1];
22 measure q[2] -> c[2];
23 measure q[3] -> c[3];
24 measure q[4] -> c[4];

```

Code 7: Shor algorithm: looking for a period r of the function - code in QASM.

```

1 using Quantum;
2 using Quantum.Operations;
3 using System;
4 using System.Numerics;
5 using System.Collections.Generic;
6
7 namespace QuantumConsole
8 {
9     public class QuantumTest
10    {
11        public static void Main()
12        {
13            QuantumComputer comp = QuantumComputer.GetInstance();
14            Register q = comp.NewRegister(0,5);
15
16            q.SigmaX(4);
17            q.SigmaX(1);
18            q.SigmaX(2);
19            q.SigmaX(3);
20            q.SigmaX(4);
21            q.CNot(target: 2, control: 3);
22            q.CNot(target: 3, control: 2);
23            q.CNot(target: 2, control: 3);
24            q.CNot(target: 1, control: 2);
25            q.CNot(target: 2, control: 1);
26            q.CNot(target: 1, control: 2);
27            q.CNot(target: 1, control: 4);
28            q.CNot(target: 4, control: 1);
29            q.CNot(target: 1, control: 4);
30            q.Measure(1);
31            q.Measure(2);
32            q.Measure(3);
33            q.Measure(4);
34        }
35    }
36 }

```

The steps of the factorization are shown in algorithm 4, but only step 5 requires usage of quantum computer. All other steps can be executed on a classical computer.

The QASM code of the Shor algorithm presented in code listing 6 is transformed with the converter into C# code presented in code listing 7.

Figure 25 shows circuit of the Shor algorithm corresponding to the code from listing 6. In this case one qubit is not used. Circuit generated in the QuIDE is shown in figure 26.

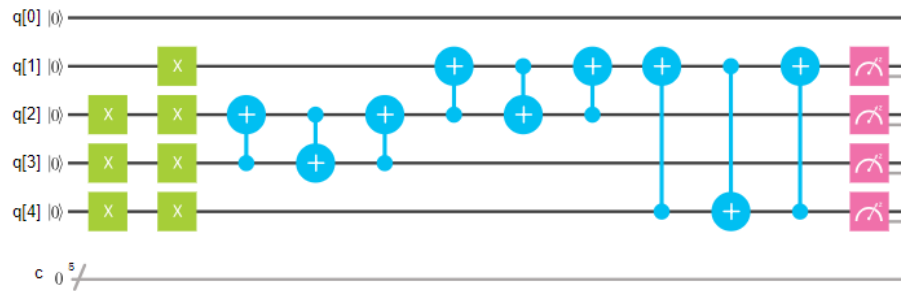


Figure 25: Circuit of the Shor algorithm corresponding to the code from listing 6. One qubit is not used.

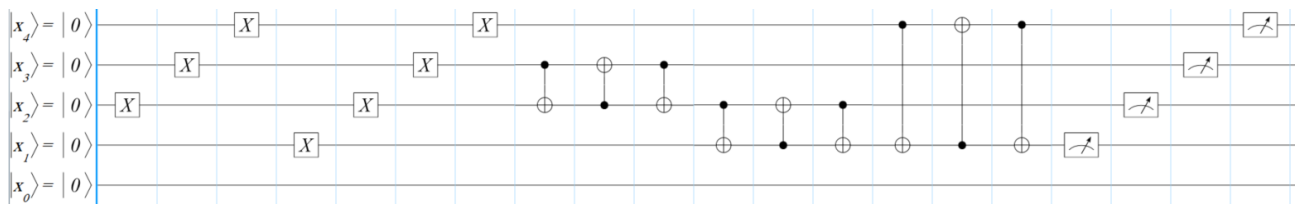


Figure 26: Quantum circuit (corresponding to the code of the Shor algorithm) in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Shor algorithm.

After executing this quantum algorithm on the IBM's simulator we can see that the probability of getting value $|4\rangle$ is equals to 1. The final results of the execution on the QuIDE look the same as in IBM's simulator, but it also shows the amplitude value $-1.00 + 0.00i$.

We will take a closer look on the internal state of the quantum system. There are two places that we chose in the quantum circuit from figure 26. They are shown in figures 27 and 28.

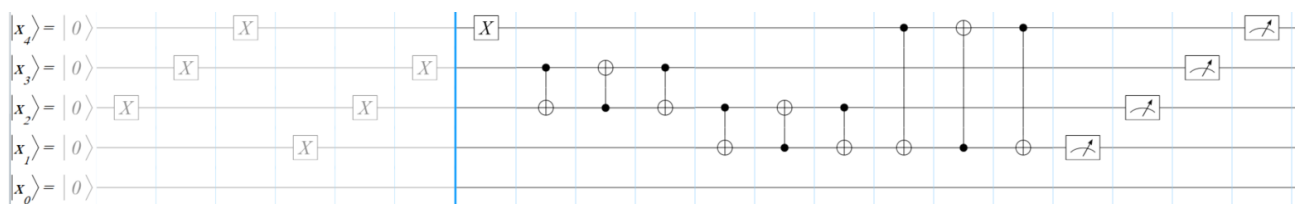


Figure 27: Quantum circuit in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Shor algorithm, the same circuit as in figure 26, "stopped" before performing one of NOT gates.

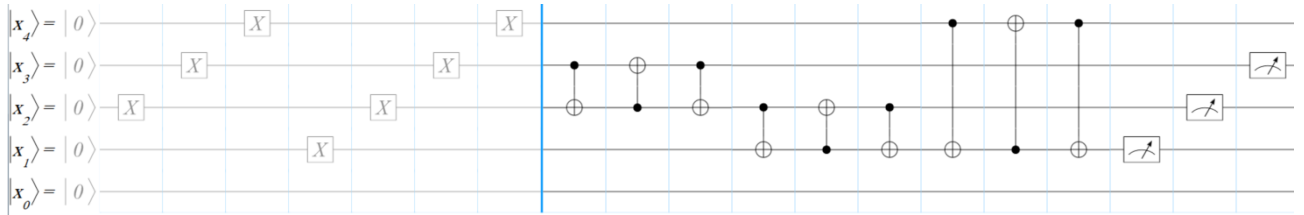


Figure 28: Quantum circuit in the QuIDE's circuit designer generated from the C# code that was converted from QASM code of the Shor algorithm, the same circuit as in figure 26, "stopped" after performing one of NOT gates.

Those two places are quantum states before and after performing NOT gate on the fourth qubit. In figures 29 and 30 there is information about those states, respectively. In the 'Qubits' column we can see that the value of the first qubit (which is actually the last, fourth qubit) changes.

Value	Qubits	Probability	Amplitude
18>	10010>	P = 1	1.00 + 0.00i

Figure 29: State of the system in the moment marked in figure 27. QuIDE can give users information about value of the registers, states of qubits, probability and amplitude.

Value	Qubits	Probability	Amplitude
2>	00010>	P = 1	1.00 + 0.00i

Figure 30: State of the system in the moment marked in figure 28. QuIDE can give users information about value of the registers, states of qubits, probability and amplitude.

6.6. Summary

The validation showed that the circuits from the IBM-Q and after the conversion, from the QuIDE are identical which means that the converter works correctly. Results from both simulators and the real processor coincide with each other and are consistent with algorithms description.

7. Implementing and running a quantum walk on the IBM-Q

7.1. Introduction - classical random walk

A classical random walk is a stochastic process that describes a path derived from a series of random steps on a mathematical space. In this section, we will depict a random walk on a space of integers (discrete random walk).

The most basic type of a random walk is a random walk in one dimension - random walk on a line of integers. We start at 0 and move one step to the left or to the right at a time, according to some probability distribution. This process can be illustrated as follows. A walker is placed at 0 on a line of integers and a coin is flipped. Depending on which side it lands, the walker moves one unit to the right or one unit to the left. After five flips of the coin, the walker could be on 1, -1, 3, -3, 5 or -5. We can interpret one-dimensional random walk also as a Markov chain with state space given by integers $s_i = 0, \pm 1, \pm 2, \dots$, transition matrix P if each time the walker is in state s_i there is some fixed probability p_{ij} that it will be in state s_j and p_{ij} does not depend upon which states the chain was in before the current state [10].

Transition matrix P for $i = 6$ looks as follows:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (19)$$

There are also different types of random walks, e.g. a random walk on a graph. In this case the walker now has more move possibilities than left and right. The random walk on a graph is presented in Algorithm 5.

Algorithm 5 Random walk on a graph

- 1: $s = u$ ▷ starting point
 - 2: **repeat**
 - 3: choose a neighbor v of u according to a certain probability distribution P
 - 4: $u = v$
 - 5: **until** stop condition
-

Classical random walks can be used to create randomized algorithm to solve e.g. 2-satisfiability (2-SAT) in polynomial or a very efficient algorithm (though still exponential) to solve 3-SAT [10]. Classical random walks are also used in different field of studies, e.g. in physics as simplified models of physical Brownian motion or as method of Fermi estimation.

7.2. Discrete quantum walk

Quantum walks are quantum counterparts of classical random walks. There are two models of them: discrete quantum walks and continuous quantum walks. In the first model an evolution operator is applied to a walker and a coin only in discrete time steps. In the second model it can be applied at any time. We will concentrate only on the discrete quantum walks.

Analogously to the classical random walk, where the walker's current state is described by a probability distribution over positions, the walker in a quantum walk is in a superposition of positions. Evolution operator is applied to the initial quantum state several times without performing intermediate measures. Quantum walks can be performed on graphs. In this subsection we will discuss a special type of them - quantum walk on an unrestricted line, which is performed on a graph $G(V, E)$ of degree $|V| = 2$. They can be used to test to which extent given quantum computer is a real quantum computer.

Components of the quantum walk are:

- a walker - quantum system living in a Hilbert space of infinite but countable dimension, its initial position is usually at 'origin' of the line:

$$|position\rangle_{initial} = |0\rangle_p, \quad (20)$$

- a coin - quantum system living in a 2-dimensional Hilbert space, it can take the canonical basis states $|0\rangle$ or $|1\rangle$ as well as any superposition of these states. Hadamard operator (shown in eq. 21) is usually used as a coin operator:

$$\hat{H} = \frac{1}{\sqrt{2}}(|0\rangle_{cc} \langle 0| + |0\rangle_{cc} \langle 1| + |1\rangle_{cc} \langle 0| - |1\rangle_{cc} \langle 1|), \quad (21)$$

- a conditional shift operator - applied to the whole quantum system, allows walker to go one step forward if the accompanying coin state is one of the two basis states (e.g. $|0\rangle$),

or one step backwards if the accompanying coin state is the other basis state (e.r. $|1\rangle$):

$$\hat{S} = |0\rangle_{cc} \langle 0| \otimes \sum_i |i+1\rangle_{pp} \langle i| + |1\rangle_{cc} \langle 1| \otimes \sum_i |i-1\rangle_{pp} \langle i|, \quad (22)$$

- a set of observables - defined according to the basis states that have been used to define coin and walker, they allow to extract information from the quantum system. Observables can be used to perform a measurement, e.g. we may first perform a measurement on the coin using observable:

$$\hat{M}_c = \alpha_0 |0\rangle_{cc} \langle 0| + \alpha_1 |1\rangle_{cc} \langle 1| \quad (23)$$

and then perform a measurement on the position states of the walker by using the operator:

$$\hat{M}_p = \sum_i a_i |i\rangle_{pp} \langle i|. \quad (24)$$

As an illustrative example we shortly present quantum walk on an infinite line. Following [10] quantum walk on an infinite line can be written as:

$$|\psi\rangle = \sum_k [a_k |0\rangle_c + b_k |1\rangle_c] |k\rangle_p, \quad (25)$$

where $|0\rangle_c, |1\rangle_c$ are the coin state components and $|k\rangle_p$ are the walker state components.

We will now introduce an example analysis of first three steps of a quantum walk for an initial state $|\psi\rangle_0 = |0\rangle_c \otimes |1\rangle_p$.

First step:

$$|\psi\rangle_1 = \frac{1}{\sqrt{2}} |0\rangle_c |1\rangle_p + \frac{1}{\sqrt{2}} |1\rangle_c |-1\rangle_p. \quad (26)$$

Second step:

$$|\psi\rangle_2 = \left(\frac{1}{2} |0\rangle_c + 0 |1\rangle_c \right) |2\rangle_p + \left(\frac{1}{2} |0\rangle_c + \frac{1}{2} |1\rangle_c \right) |0\rangle_p + \left(0 |0\rangle_c - \frac{1}{2} |1\rangle_c \right) |-2\rangle_p. \quad (27)$$

Third step:

$$\begin{aligned}
 |\psi\rangle_3 = & \left(\frac{1}{2\sqrt{2}} |0\rangle_c + 0 |1\rangle_c \right) |3\rangle_p + \left(\frac{1}{\sqrt{2}} |0\rangle_c + \frac{1}{2\sqrt{2}} |1\rangle_c \right) |1\rangle_p + \\
 & \left(\frac{-1}{2\sqrt{2}} |0\rangle_c + 0 |1\rangle_c \right) |-1\rangle_p + \left(0 |0\rangle_c + \frac{1}{2\sqrt{2}} |1\rangle_c \right) |-3\rangle_p.
 \end{aligned} \tag{28}$$

These were the first three steps. Algorithm 6 shows quantum walk on an infinite line with infinite possible steps.

Algorithm 6 Quantum walk

- 1: **repeat**
 - 2: choose a starting point, e.g. $|0\rangle_p$
 - 3: **repeat**
 - 4: toss a quantum coin

$$H |0\rangle_c = \frac{1}{\sqrt{2}} |0\rangle_c + \frac{1}{\sqrt{2}} |1\rangle_c$$
 or
$$H |1\rangle_c = \frac{1}{\sqrt{2}} |0\rangle_c - \frac{1}{\sqrt{2}} |1\rangle_c$$
 - 5: move one qubit left and right according to qubit state
 - 6: **until** stop condition
 - 7: **until** stop condition
-

This algorithm can also use to implement a quantum walk on a circle. The implementation of a walker will be different in that case. We will present the implementation this algorithm in the next subsection.

7.3. Quantum circuit implementation of quantum walk on a circle

As a part of assessment of the IBM-Q software environment we will implement the quantum walk algorithm in both the IBM-Q and the QuIDE. After that, we will compare results from the IBM-Q real processor, the IBM-Q simulator and the QuIDE simulator.

To implement a quantum walk we followed the article [36]. Authors of this article proposed implementations of a quantum walk on various topologies, including a circle. Figures 31 and 32 show realizations of step forward and backward, respectively, on a cyclic permutations of the node states.

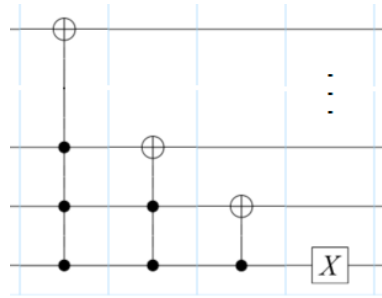


Figure 31: Increment gate on n qubits, producing cyclic permutations in the 2^n bit-string states.

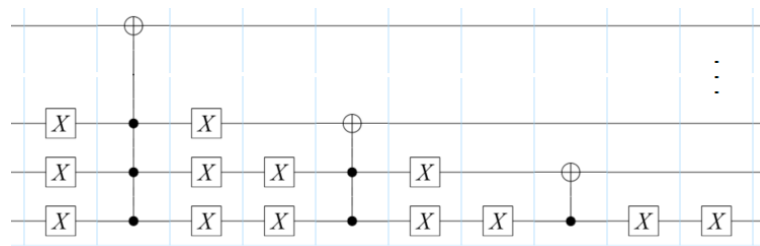


Figure 32: Decrement gate on n qubits, producing cyclic permutations in the 2^n bit-string states.

To implement a walk along a cycle of size 2^n we require $n + 1$ qubits: n qubits to encode the nodes and an additional qubit for the coin. The coin operator can be implemented by a single Hadamard gate. Figure 33 presents an example of the circuit for a cycle of size 16.

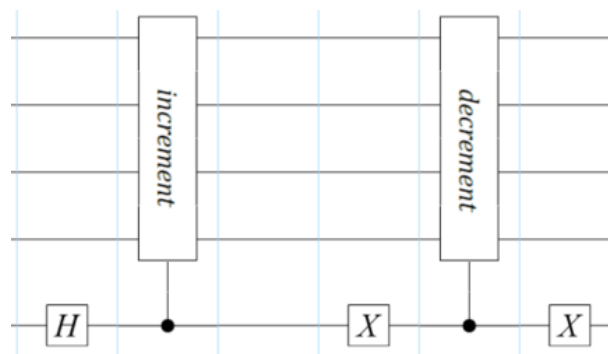


Figure 33: Quantum circuit implementing one step of a quantum walk along a 16-length cycle. The walker is implemented on four qubits, so number of possible states of the walker is equals to $2^4 = 16$.

7.4. Decomposition of multiple-controlled gates

As it was shown in the previous subsection, the particular implementation of the quantum walk we want to realize requires multiple-controlled quantum gates. Multiple-controlled NOT gates can be easily implemented in the QuIDE, but QISKit allows to create only the Toffoli gate. However, to implement the quantum walk on three qubits we need triple-controlled NOT gate.

Paper [37] shows how to decompose n -bit quantum gates to elementary gates. Figures 34 and 35 presents simulations of any double- and triple-controlled unitary gate U with single-controlled gates. In the first example $V^2 = U$, whereas in the second one $V^4 = U$.

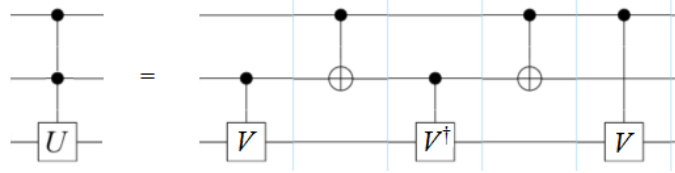


Figure 34: Simulation of any double-controlled unitary gate U using CNOT gates, V gates and a V^\dagger gate, where $V^2 = U$.

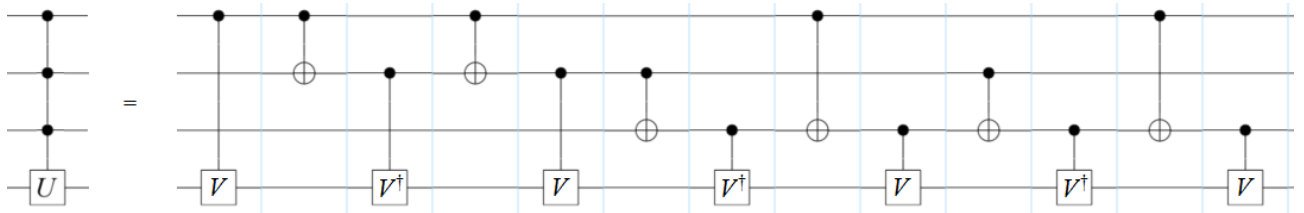


Figure 35: Simulation of any triple-controlled unitary gate U using CNOT gates, V gates and a V^\dagger gate, where $V^4 = U$.

As both the QISKit and the QuIDE have functions realizing the Toffoli gate with two control qubits, we needed to decompose cccNOT gate for the QISKit implementation of the quantum walk. Simulation of this gate is shown in figure 36.

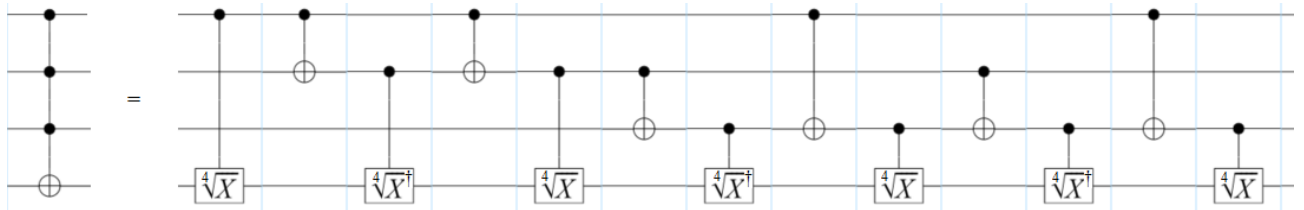


Figure 36: Simulation of triple-controlled NOT gate using CNOT gates, $\sqrt[4]{X}$ gates and $\sqrt[4]{X}^\dagger$ gates.

The $\sqrt[4]{X}$ gate, as well as the $\sqrt[4]{X}^\dagger$ gate are not implemented in the IBM-Q. We had to realize them with general unitary gates U . In the QISKit tutorial [38] there is a matrix that represents the U gate:

$$U = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i\lambda+i\phi} \cos(\theta/2) \end{bmatrix}. \quad (29)$$

This is the most general form of a single qubit unitary gate. QISKit provides the U gate through the $u3$ gate:

$$u3(\theta, \phi, \lambda) = U(\theta, \phi, \lambda). \quad (30)$$

The $u2(\phi, \lambda) = u3(\pi/2, \phi, \lambda)$ is also useful gate as it allows us to create superpositions:

$$u2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{bmatrix}. \quad (31)$$

The $u1(\lambda) = u3(0, 0, \lambda)$ is a useful as it allows us to apply a quantum phase:

$$u1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}. \quad (32)$$

Matrix of the gate realizing rotation around X in the Bloch sphere axis has the form:

$$\sqrt[4]{X} = \begin{bmatrix} \cos(\theta/2) & -i \sin(\theta/2) \\ -i \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}. \quad (33)$$

The $\sqrt[4]{X}$ gate rotates qubit by $\pi/4$ around the X axis, so its matrix has the form:

$$\sqrt[4]{X} = \begin{bmatrix} \cos(\pi/8) & -i \sin(\pi/8) \\ -i \sin(\pi/8) & \cos(\pi/8) \end{bmatrix}. \quad (34)$$

This explanation was needed to understand how we translated $\sqrt[4]{X}$ and $\sqrt[4]{X}^\dagger$ gates into $u3$ gates. After creating and solving a simple system of equations we found values of the required parameters. In $\sqrt[4]{X}$ they are as follows: $\theta = \pi/4, \phi = -\pi/2, \lambda = \pi/2$. Whereas in $\sqrt[4]{X}^\dagger$: $\theta = -\pi/4, \phi = -\pi/2, \lambda = \pi/2$, so

$$\sqrt[4]{X} \rightarrow u3(\pi/4, -\pi/2, \pi/2), \quad (35)$$

$$\sqrt[4]{X}^\dagger \rightarrow u3(-\pi/4, -\pi/2, \pi/2). \quad (36)$$

7.5. Implementation of the quantum walk algorithm with a two-qubit walker

At first, we implemented a very simple quantum walk with two-qubit walker. The walker can move forward and backward on four nodes (number of nodes equals 2^n , where $n = 2$) located on a circle.

We implemented the algorithm on both the IBM-Q simulator and the QuIDE simulator as well as on the IBM-Q real backend. We used simulators to check how ideal results should look like and then compare them with results from the IBM-Q processor. That way we were able to assess usability of the IBM-Q. In this subsection we will present results from both simulators and from the real processor.

In figure 37 there is a quantum circuit created in the QuIDE representing the step gate from figure 38. The lowest qubit is the coin qubit and the two upper qubits are the walker qubit.

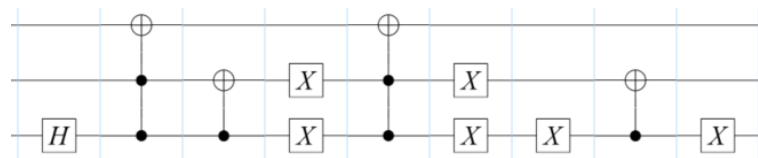


Figure 37: Quantum circuit representing the step gate from figure 38. The lowest qubit is the coin qubit and the two upper qubits are the walker qubit.

Figure 38 presents quantum circuit of the algorithm implemented in the QuIDE. QuIDE allows users to create their own gates (a C# function in code is interpreted as a gate). In our implementation one step of the walker is a quantum gate.

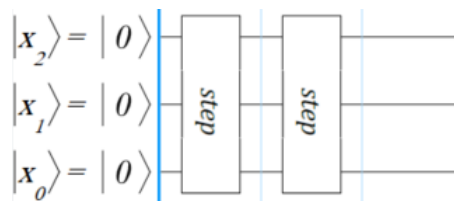


Figure 38: Quantum circuit of the quantum walk algorithm implemented in the QuIDE. QuIDE allows users to create their own gates. In our implementation one step of the walker is a quantum gate.

Results from execution of this circuit are shown in figure 39. It is important to notice that the QuIDE counts the coin qubit as a part of the quantum state of the system, however, we only need values of the walker qubits. Therefore, the last qubit value in the 'Qubits' column is unnecessary in this instance and we have two possible states of walker: $|00\rangle$ and $|10\rangle$.

Value	Qubits	Probability	Amplitude
$ 0\rangle$	$ 000\rangle$	$P = 0.25$	$0.50 + 0.00i$
$ 1\rangle$	$ 001\rangle$	$P = 0.25$	$-0.50 + 0.00i$
$ 4\rangle$	$ 100\rangle$	$P = 0.25$	$0.50 + 0.00i$
$ 5\rangle$	$ 101\rangle$	$P = 0.25$	$0.50 + 0.00i$

Figure 39: Results from execution of the circuit from figure 38. QuIDE counts the coin qubit as a part of the quantum state of the system, however, we only need values of the walker qubits. Therefore, we have two possible states of walker: $|00\rangle$ and $|10\rangle$.

The next step was to implement the same quantum walk with two-qubit walker on the IBM-Q and execute it on both the simulator and the real processor. However, this time it was essential to remember about mapping the algorithm to the topology of the architecture of the ibmqx4 processor (we chose this particular processor to execute the quantum walk on). The topology was presented in figure 4.

This algorithm is quite simple and it was easy to do the mapping just by analyzing the topology and the circuit generated in the QuIDE. In the QISKit implementation of the quantum walk qubit q_3 is now the coin qubit and qubits q_2 and q_4 are the walker qubits. The circuit is presented in figure 40.

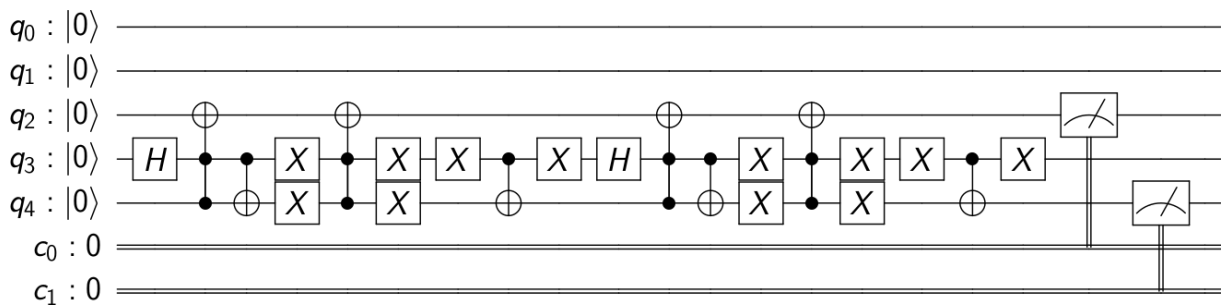


Figure 40: The circuit of the quantum walk mapped to the ibmqx4 architecture and implemented in QISKit. The two measured qubits are the walker qubits.

Results from the IBM-Q simulator after executing only the first step of the walker are shown in figure 41. The histogram depicts that there is about 50% probability that the walker from the initial position $|00\rangle$ moved either to $|10\rangle$ or $|11\rangle$ state.

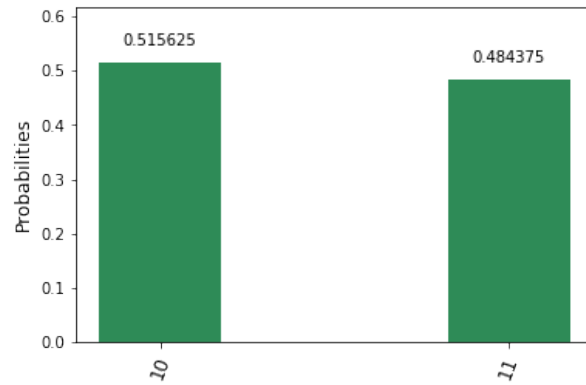


Figure 41: Results from execution of the first step of the quantum walk algorithm with 2-qubit walker on the IBM-Q simulator.

We also executed the first step of the quantum walk on ibmqx4 architecture. Results from this execution are in figure 42. It is clearly visible that the results differ significantly. The reason for that is decoherence, which is definitely a big weakness of quantum computers nowadays.

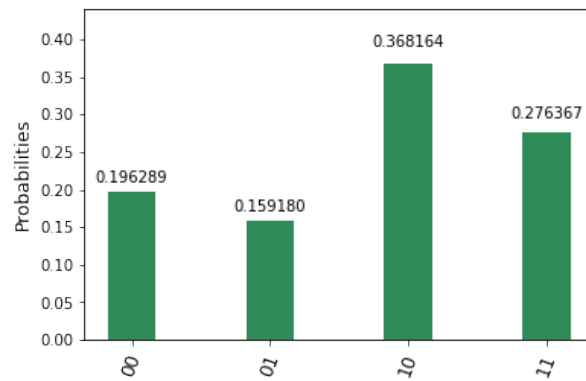


Figure 42: Results from execution of the first step of the quantum walk algorithm with 2-qubit walker on ibmqx4 backend.

Figure 43 shows results from execution of all two steps of the algorithm on the IBM-Q simulator. It is plain to see that they are very similar to the results from the QuIDE.

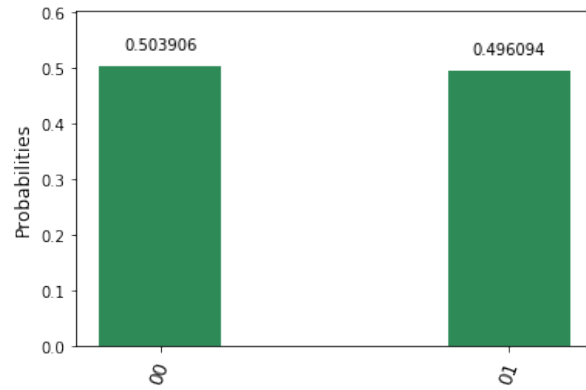


Figure 43: Results from execution of both two steps of the quantum walk algorithm with 2-qubit walker on the IBM-Q simulator.

In figure 44 there are results from execution of the two steps of the algorithm on the ibmqx4 processor. Comparing them with those from figure 43 one can see decoherence is significant.

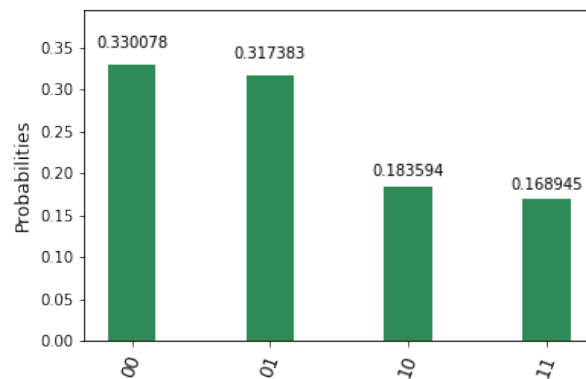


Figure 44: Results from execution of both two steps of the quantum walk algorithm with 2-qubit walker on the ibmqx4 backend.

To make sure that those differences in results from the simulator and the real processor are due to decoherence indeed, we decided to test it. The IBM-Q User Guide [7] provides quantum programs that check T_1 and T_2 times. We described those parameters in subsection 4.2.1.

The test functions as follows: we apply X gate on one qubit and after that we put many identity gates separated by barriers. The number of identity gates is roughly the same as number of elementary gates in the quantum walk algorithm, namely around 25 for one step. We should expect the state of the qubit to be $|1\rangle$, because identity gates should do nothing. However, the results are quite surprising. They are presented in figure 45.

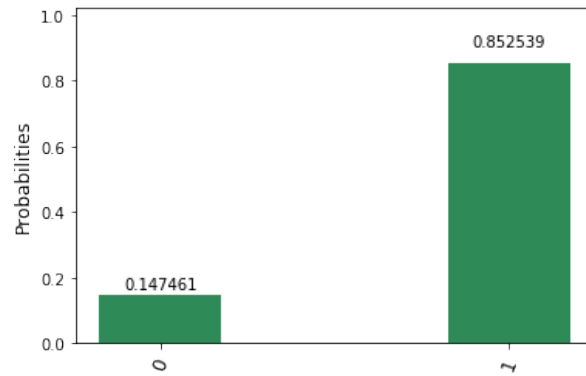


Figure 45: Results from decoherence test performed on ibmqx4 architecture.

Decoherence is explicitly visible here. The qubit has no possibility to be in $|0\rangle$ state, nonetheless the histogram shows 0.147461 probability of that situation. That means that results of the quantum walk could be distorted.

Authors of the paper [21] also had problems with decoherence in their quantum walk. They wrote that performing more than a single step requires time which is longer than coherence time, and the algorithm fails on the IBM-Q after the first step.

7.6. Implementation of the quantum walk algorithm with a three-qubit walker

In this subsection we will discuss implementation and results of the quantum walk algorithm with three-qubit walker.

Like in the previous subsection, we started from implementing the algorithm in the QuIDE. Implementation of the step gate is shown in figure 46. The coin qubit is the one in the bottom. The walker is realized on the three upper qubits.

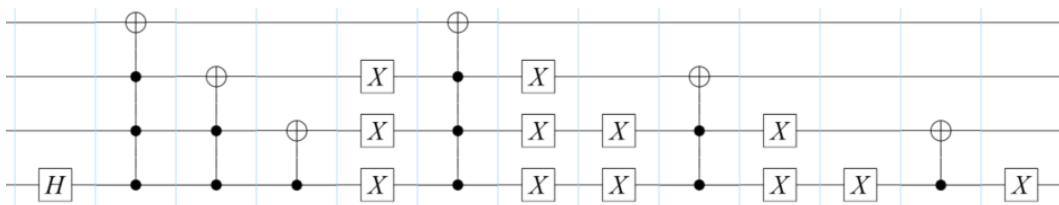


Figure 46: Quantum circuit representing the step gate from figure 47. The lowest qubit is the coin qubit and the two upper qubits are the walker qubit.

Figure 47 shows the quantum circuit with four step gates.

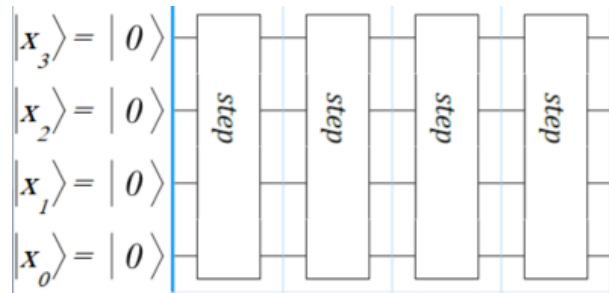


Figure 47: Quantum circuit of the quantum walk implementation in the QuIDE with four step gates. QuIDE allows users to create their own gates. In our implementation one step of the walker is a quantum gate.

Results from the algorithm execution in the QuIDE are presented in figure 48. As in the previous two-qubit variant of the quantum walk, this time it is also crucial to remember that the last qubit value in the 'Qubits' column is value of the coin. This, the possible states of the walker are: $|000\rangle$, $|010\rangle$, $|100\rangle$, $|110\rangle$.

Value	Qubits	Probability	Amplitude
$ 0\rangle$	$ 0000\rangle$	P = 0.0625	$-0.25 + 0.00i$
$ 1\rangle$	$ 0001\rangle$	P = 0.0625	$0.25 + 0.00i$
$ 4\rangle$	$ 0100\rangle$	P = 0.0625	$0.25 + 0.00i$
$ 5\rangle$	$ 0101\rangle$	P = 0.0625	$-0.25 + 0.00i$
$ 8\rangle$	$ 1000\rangle$	P = 0.0625	$0.25 + 0.00i$
$ 9\rangle$	$ 1001\rangle$	P = 0.0625	$-0.25 + 0.00i$
$ 12\rangle$	$ 1100\rangle$	P = 0.5625	$0.75 + 0.00i$
$ 13\rangle$	$ 1101\rangle$	P = 0.0625	$0.25 + 0.00i$

Figure 48: Results from the algorithm execution of the circuit from figure 47 in the QuIDE. QuIDE counts the coin qubit as a part of the quantum state of the system, however, we only need values of the walker qubits. Therefore, we have four possible states of walker: $|000\rangle$, $|010\rangle$, $|100\rangle$ and $|110\rangle$.

Implementing this algorithm in QISKit was not trivial, because the cccNOT gate had to be decomposed to elementary gates that QISKit would be able to read. Operations described in subsection 7.4 had been done and circuit shown in figure 49 was created.

The values of the parameters in $U3$ gates are the parameters depicted in subsection 7.4 converted to double. The QISKit does this conversion by default.

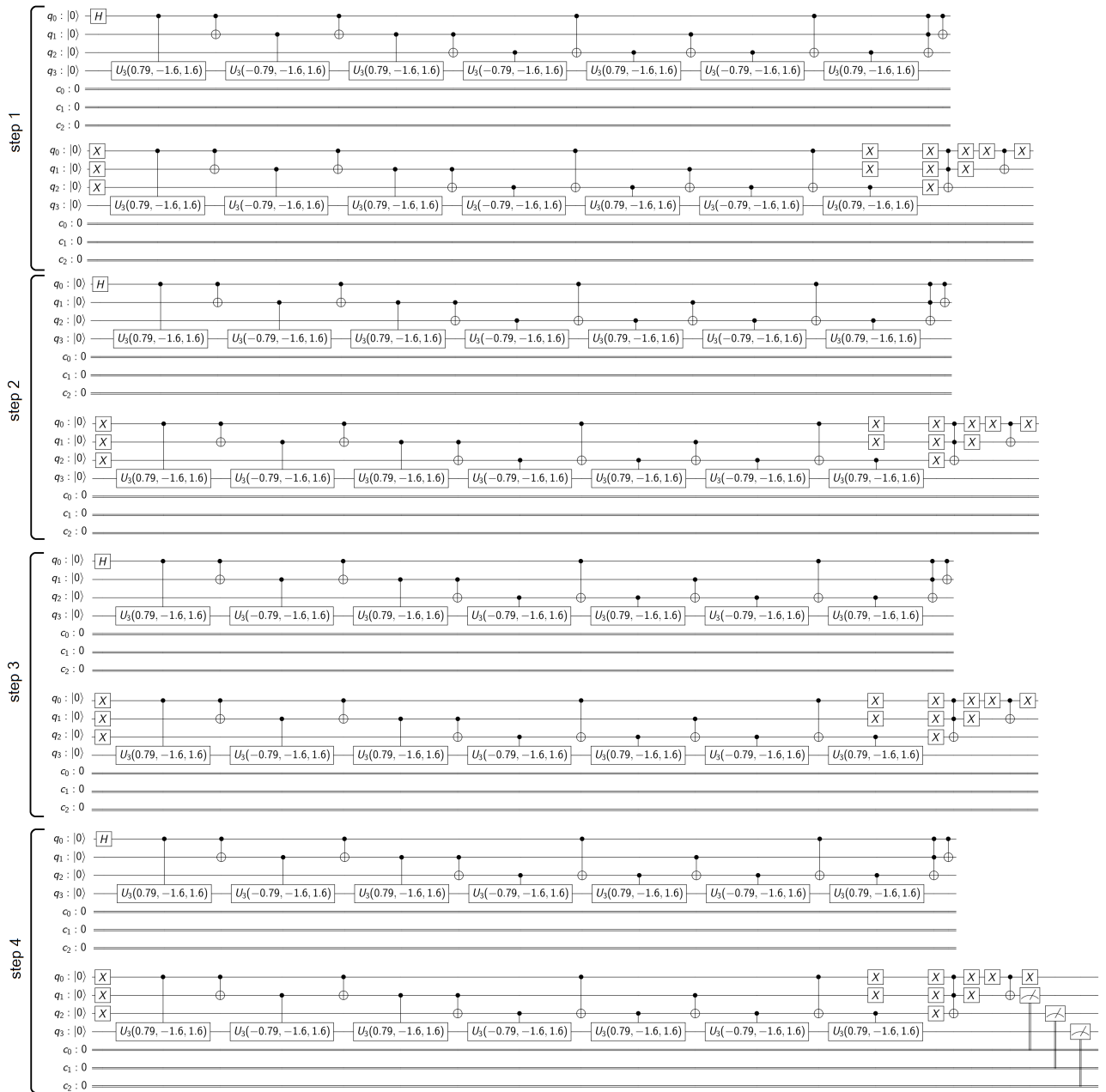


Figure 49: The quantum circuit of the quantum walk implementation in QISKit. In the figure the particular steps are labeled. The values of the parameters in U_3 gates are the parameters depicted in subsection 7.4 converted to double. QISKit does this conversion by default.

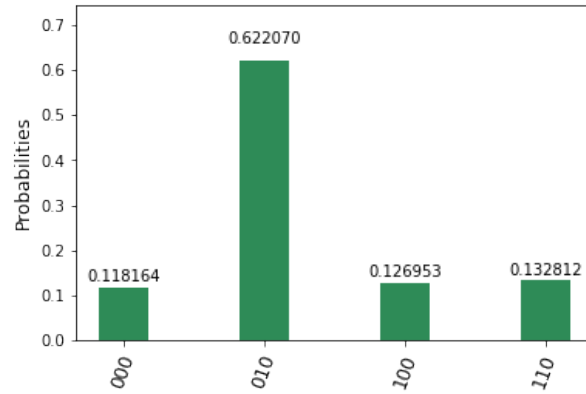


Figure 50: Results from execution of the circuit from figure 49 with 3-qubit walker on the IBM-Q simulator.

Results from execution on the simulator of the circuit from figure 49 are shown in figure 50. These are the idealistic results. We wanted to weigh them against results from the real 16-qubit processor just as in the previous subsection. There occurred the architecture-matching problem. After analysis of both 5-qubit architectures we verified that the algorithm cannot be mapped to neither of them in an easy way.

There is a tool described in article [39] that maps an algorithm written in QASM to the 16-qubit architecture of the IBM-Q. Unfortunately, this tool could not identify the U gates, which are essential in our quantum walk algorithm. We could not use the mapper, so we do not have results from a real processor for this variant of the algorithm.

7.7. Conclusions

We developed quantum circuits that realize the quantum walk algorithm on a circle with four nodes and with eight nodes. Results from simulators, both the QuIDE and IBM-Q, show that our implementations are correct. Two biggest problems that we encountered were that architecture mapping issue and decoherence.

When it comes to the first on those, with easy algorithms the mapping can be done without any tool. However, with more complicated algorithms it would be very difficult and could generate a lot of bugs. Unfortunately, there is lack of mapping tools.

Decoherence is noticeable in the IBM-Q backends. When algorithm consists of several dozen of gates it is sometimes difficult to distinguish correct results from those which decohered. It handicaps reading results from longer and more complicated circuits.

8. Assessment of usability of IBM-Q

IBM-Q is definitely a revolutionary tool for creating and running quantum algorithms. It allows students, researchers, scientists and literally everyone to write and execute their own quantum program. The only requirement is the Internet access.

The graphical interface is quite limited. It provides access to only two out of three available architectures (the 16-qubit one is not available through the graphical interface). It also has narrower set of gates than, e.g. QISKit. Arguably, the clarity issue was critical in this case. The graphical interface can be used at the learning stage as well as to implement less complicated quantum algorithms. It is very easy to use.

Algorithms like quantum walk that we implemented could not be created in the graphical interface. The Toffoli gate is not available there and after decomposing it to elementary gates the quantum circuit was too long to display it in the Composer. The Composer managed to create only a few first gates and the latter ones were removed by the software.

QISKit provides more gates [33], nevertheless for our quantum walk with three-qubit walker we needed cccNOT gate, which is not available in QISKit. However, it was possible to implement this gate after decomposition to elementary gates. In QISKit we can also use standard Python instructions such as loops, which significantly speeds up the process of implementation.

For educational and debugging purposes, it is useful to visualize the quantum state. QISKIT provides tools for creating histograms. The measurement results from many executions of the quantum circuit can be represented as a probability distribution over the possible outcomes. For a quantum circuit which has previously run on a backend, a histogram visualizing the probability distribution can be obtained. The height of each bar represents the fraction of instances the corresponding outcome is obtained within the total number of shots on the backend. QISKit also allows to draw a circuit, providing it has a code implementation.

A serious disadvantage is the fact that during implementation of any quantum algorithm programmers need to remember about topologies of particular architectures of the IBM-Q backends. It can be very difficult to match the algorithm to the specific architecture and there are few tools that can help with this problem.

Another obstacle is the decoherence issue. After a number of gates decoherence significantly interrupts executing of algorithms. Quantum computers are still imperfect and they cannot keep the quantum state very long.

9. Summary, conclusions and future work

Quantum computing is no longer only a theory, nowadays it is becoming a technology available almost to everyone. Researchers and scientists are studying the potential of quantum computers and it seems that practical usage of them is just around the corner.

The thesis began with a brief introduction to quantum computing. We described basic notions, such as a qubit, a quantum gate and an entanglement. Then we presented physical phenomena used to build quantum computers, realization of qubits in superconducting quantum computing as well as how to measure the power of quantum computers. Next architectures of the IBM-Q and its parameters were delineated. After that we described software environment of the IBM-Q, its features and their applications. There is also a chapter about interoperability of the IBM-Q and the QuIDE simulators. We presented our idea for an extension of the IBM's simulator so that programmers would have a possibility to preview the internal state of a quantum system at each stage of the computation. We created a converter that takes QASM code and outputs C# code that can be executed in the QuIDE simulator. QuIDE has the mentioned feature. It can act as a debugger in classical computing. Afterwards we took a closer look at quantum walk algorithms. We also implemented and executed one on both the IBM-Q and the QuIDE simulators as well as on the real quantum processor of IBM. The histograms were compared and verified a big impact of decoherence on the results from the real backend, which is definitely a disadvantage. However, decoherence in the IBM-Q requires further research.

The thesis showed the importance of quantum simulators, validated the IBM-Q quantum computer and reviewed some available quantum algorithms.

The whole software environment of the IBM-Q is very easy to use, nevertheless it needs some improvement. Flexibility of architectures' topology could be bigger. Decoherence looms large in the process of computation and it needs to be decreased. This issue has to be exhaustively analyzed.

Decoherence analysis is the most desirable direction of future work. Especially its impact on the state of the system during usage of multiqubit gates. We only tested one-qubit identity gates. Another interesting idea for further research is the issue of matching algorithms to a particular architecture of quantum computing. We implemented two variants of quantum walk: with 2-qubits walker and with 3-qubit walker. While the first one was quite simple and the process of mapping to the ibmqx4 architecture could be one after analyzing the topology of the backend,

References

- [1] Jop Briët and Simon Perdrix. “Quantum Computation and Information”. *ERCIM News* 112 (Jan. 2018), pp. 8, 9.
- [2] Stephen Jordan. “Quantum Zoo”. <https://math.nist.gov/quantum/zoo/>.
- [3] Leonard Susskind Art Friedman. *Mechanika kwantowa. Teoretyczne minimum*. Prószyński i S-ka, 2016. http://www.proszynski.pl/Mechanika_kwantowa__Teoretyczne_minimum-p-33197-.html.
- [4] Sławomir Brzezowski. *Wstęp do mechaniki kwantowej*. Instytut Fizyki Uniwersytetu Jagiellońskiego, 2006. http://www.if.uj.edu.pl/documents/3830070/31778969/Slawomir_Brzezowski_Wstep_do_Mechaniki.pdf.
- [5] John Clarke and Frank K. Wilhelm. “Superconducting quantum bits”. *Nature* 453 (2008), pp. 1031–1042. DOI: 10.1038/nature07128. <http://dx.doi.org/10.1038/nature07128>.
- [6] Hans Mooij. “Superconducting quantum bits”. *Physics World* 17.12 (2004), p. 29. <http://stacks.iop.org/2058-7058/17/i=12/a=30>.
- [7] “IBM Q Experience Library”. <http://research.ibm.com/ibm-q/qx/>.
- [8] “Quantum Information Software Kit”. <https://www.qiskit.org/>.
- [9] Joanna Patrzyk, Bartłomiej Patrzyk, Katarzyna Rycerz, and Marian Bubak. “Towards A Novel Environment for Simulation of Quantum Computing”. *Computer Science* 16.1 (2015), p. 103. ISSN: 2300-7036. <https://journals.agh.edu.pl/csci/article/view/1268>.
- [10] Salvador Venegas-Andraca. “Discrete Quantum Walks and Quantum Image Processing”. *University of Oxford* (2005).
- [11] Jay M. Gambetta, Jerry M. Chow, and Matthias Steffen. “Building logical qubits in a superconducting quantum computing system”. *npj Quantum Information* 3 (2017). ISSN: 2056-6387. DOI: 10.1038/s41534-016-0004-0. <https://doi.org/10.1038/s41534-016-0004-0>.

-
- [12] “D-Wave 2000QTM Quantum Computer Technology Overview”. https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_0117F2.pdf.
- [13] “Forest - An API for quantum computing in the cloud”. <https://www.rigetti.com/forest>.
- [14] Chad Rigetti. “Introducing the Rigetti Tech Blog”. *Rigetti Tech Blog*. <https://rigetticomputing.github.io/blog/2018/01/31/Introducing-the-Rigetti-Tech-Blog.html>.
- [15] “IBM-Q experience - Composer”. <https://quantumexperience.ng.bluemix.net/qx/editor>.
- [16] “IBM-Q experience - OpenQASM editor”. <https://quantumexperience.ng.bluemix.net/qx/qasm>.
- [17] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. “A software methodology for compiling quantum programs”. *Quantum Science and Technology* 3.2 (2018), p. 020501. <http://stacks.iop.org/2058-9565/3/i=2/a=020501>.
- [18] Andris Ambainis, Harry Buhrman, Elham Kashefi, Adrian Kent, Iordanis Kerenidis, Frederik Kerling, Noah Linden, Ashley Montanaro, Floor van de Pavert, and Thomas Strohm. “Quantum Software Manifesto” (2017). <http://www.qusoft.org/quantum-software-manifesto/>.
- [19] Ashley Montanaro. “Quantum algorithms: an overview”. *Npj Quantum Information* 2 (Jan. 2016). <http://dx.doi.org/10.1038/npjqi.2015.23>.
- [20] J Kempe. “Quantum random walks: An introductory overview”. *Contemporary Physics* 44.4 (2003), pp. 307–327. DOI: 10.1080/00107151031000110776. <https://doi.org/10.1080/00107151031000110776>.
- [21] Radhakrishnan Balu, Daniel Castillo, and George Siopsis. “Physical realization of topological quantum walks on IBM-Q and beyond”. *Quantum Science and Technology* 3.3 (2018), p. 035001. <http://stacks.iop.org/2058-9565/3/i=3/a=035001>.
- [22] R. P. Feynman. “Simulating Physics with Computers”. *International Journal of Theoretical Physics* 21.6 (1982), pp. 467–488.

-
- [23] Philipp Schindler, Daniel Nigg, Thomas Monz, Julio T Barreiro, Esteban Martinez, Shannon X Wang, Stephan Quint, Matthias F Brandl, Volckmar Nebendahl, Christian F Roos, Michael Chwalla, Markus Hennrich, and Rainer Blatt. “A quantum information processor with trapped ions”. *New Journal of Physics* 15.12 (2013), p. 123012. <http://stacks.iop.org/1367-2630/15/i=12/a=123012>.
- [24] Alexandre Blais. *Quantum Computing with Superconducting Qubits*. 2012. https://www.youtube.com/watch?v=t5nxusm_Umk.
- [25] Lev S. Bishop, Sergey Bravyi, Andrew Cross, Jay M. Gambetta, and John Smolin. “Quantum Volume” (2017). https://dal.objectstorage.open.softlayer.com/v1/AUTH_039c3bf6e6e54d76b8e66152e2f87877/community-documents/quatnum-volumehp08co1vbo0cc8fr.pdf.
- [26] Dario Gil. “The future is quantum”. <https://www.ibm.com/blogs/research/2017/11/the-future-is-quantum/>.
- [27] “IBMQX 2 - backend information”. <https://github.com/QISKit/ibmqx-backend-information/blob/master/backends/ibmqx2/README.md>.
- [28] “IBMQX 4 - backend information”. <https://github.com/QISKit/ibmqx-backend-information/blob/master/backends/ibmqx4/README.md>.
- [29] “IBMQX 5 - backend information”. <https://github.com/QISKit/ibmqx-backend-information/blob/master/backends/ibmqx5/README.md>.
- [30] “IBM-Q experience Community”. <https://quantumexperience.ng.bluemix.net/qx/community>.
- [31] “Python API client IBM Quantum Experience github page”. <https://github.com/QISKit/qiskit-api-py>.
- [32] “QISKit github page”. <https://github.com/QISKit/qiskit-sdk-py>.
- [33] “QISKit Documentation”. <https://www.qiskit.org/documentation/index.html>.
- [34] “QISKit: Frequently Asked Questions”. https://github.com/QISKit/ibmqx-user-guides/blob/master/rst/full-user-guide/000-FAQ/000-Frequently_Asked_Questions.rst.
- [35] Jr Samuel J. Lomonaco. “Shor’s Quantum Factoring Algorithm”. *AMS Proceedings of Symposium in Applied Mathematics* 58 (Nov. 2000).

-
- [36] B.L. Douglas and J.B. Wang. “Efficient quantum circuit implementation of quantum walks” (2009). <https://arxiv.org/abs/0706.0304v3>.
- [37] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. “Elementary gates for quantum computation”. *Phys. Rev. A* 52 (5 Nov. 1995), pp. 3457–3467. DOI: 10.1103/PhysRevA.52.3457. <https://link.aps.org/doi/10.1103/PhysRevA.52.3457>.
- [38] “QISKit - Basic Quantum Operations”. https://nbviewer.jupyter.org/github/QISKit/qiskit-tutorial/blob/master/reference/tools/quantum_gates_and_linear_algebra.ipynb.
- [39] A. Zulehner, A. Paler, and R. Wille. “Efficient mapping of quantum circuits to the IBM QX architectures”. *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 1135–1138.
- [40] “QISKit ACQUA Algorithms and Circuits for QUantum Applications”. <https://qiskit.org/acqua>.