# **REFERAT**

# “**Quantum Programming**”

(Book: Thomas G. Wong Introduction to Classical and Quantum Computing, Chapter 5)

# CUPRINS

[INTRODUCERE 3](#_Toc102735812)

[CAPITOLUL 1. IBM QUANTUM EXPERIENCE 5](#_Toc102735813)

[1.1 Services 5](#_Toc102735814)

[1.2 Circuit Composer 8](#_Toc102735815)

[1.3 Quantum Processor 12](#_Toc102735816)

[1.4 Simulator 14](#_Toc102735817)

[CAPITOLUL 2. QUANTUM ASSEMBLY LANGUAGE 17](#_Toc102735818)

[2.1. OpenQASM 17](#_Toc102735819)

[2.2. Quantum Experience Standard Header 18](#_Toc102735820)

[2.3. OpenQASM in IBM Quantum Experience 19](#_Toc102735821)

[2.4. Quantum Adder 20](#_Toc102735822)

[CAPITOLUL 3. QISKIT 23](#_Toc102735823)

[3.1. Circuit Composer 23](#_Toc102735824)

[3.2. Quantum Lab 24](#_Toc102735825)

[CONCLUZII 27](#_Toc102735826)

[BIBLIOGRAFIE 28](#_Toc102735827)

# INTRODUCTION

Computers have come a long way. The first computers were people, not machines, who performed calculations. Indeed, the term “computer” dates as far back as the early 1600’s, centuries before the digital age. Human computers persisted into modern history, with NASA, for example employing people to compute launch trajectories for the space program and other scientific endeavors through the 1960’s. Of course, mechanical and then electronic computers have since taken over, evolving from massive machines that filled entire rooms, to personal computers on our desktops, to smaller and smaller devices. Now, nearly everyone has a electronic computer in their pocket—a smartphone—that is more powerful than the computers that landed people on the moon.

Computers have become so polished that we can use and even program them without understanding how they work at a fundamental level. That’s not a bad thing. It has allowed computers to become tools for more and more people.

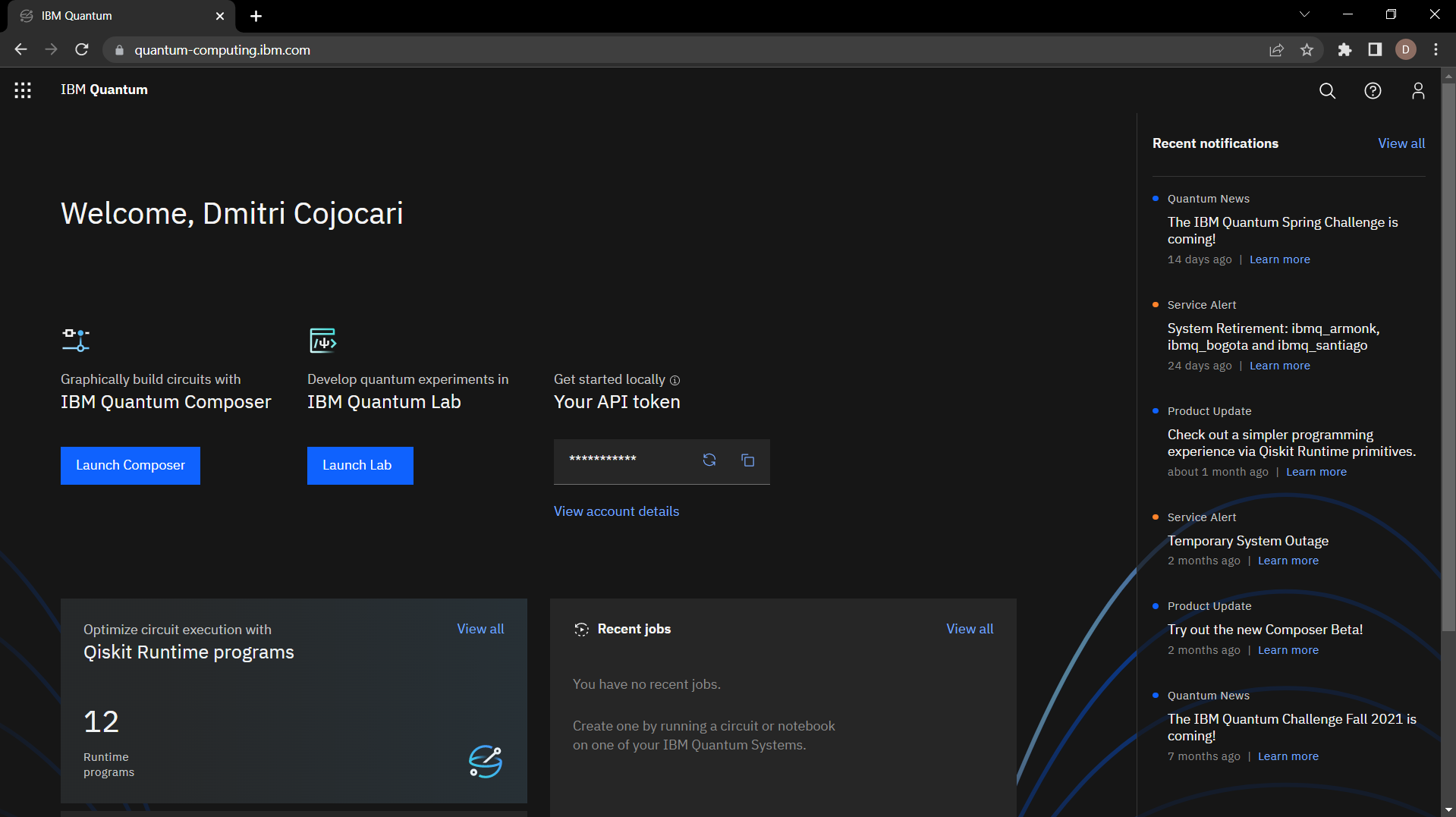
One day, quantum computing will get to this point of accessibility, where we can use and program them without worrying about their details. But we are not quite there, yet. In their development, quantum computers are where classical computers were decades ago. Their inner working still matter, and to understand these inner workings, it is helpful to understand the inner workings of regular, classical computers. So, in this chapter, we will look at the basics of classical computing. If you have studied the fundamentals of classical computing or electrical engineering, this may be review for you. Even so, the topics may be worth seeing again because they have been carefully selected for their quantum analogues in later chapters. Furthermore, quantum computing is not developing in isolation of classical computing. Many of the design decisions for quantum computers stem out of what is done with classical computers. Without knowing classical computing, some aspect of quantum computing may seem arbitrary. A rudimentary understanding classical computing makes it easier to understand quantum computing.

Quantum computing is currently emerging from the research lab onto the marketplace. Many companies are building prototype quantum processors, and although these devices are not yet good enough for fault-tolerant quantum computation, they may still have uses. These rudimentary quantum processors are called *noisy intermediate-scale quantum* (NISQ) devices, where noisy means they suffer from too much decoherence to be fault-tolerant, and intermediate-scale means they have a moderate number of qubits, say roughly fifty to a few hundred. NISQ devices were used to demonstrate quantum computational supremacy, which we briefly discussed in Section 1.8.3. Many companies have made their rudimentary quantum processors available for people to experiment with. In this chapter, we will learn how to program IBM’s quantum computers over the internet. This is not an endorsement of their product or services, and other companies have similar tools for programming their own quantum devices, which you are encouraged to explore on your own. Rather, IBM has made several of their quantum processors freely available to the public, making them a prudent choice for a textbook. Furthermore, after learning one quantum programming toolkit, it will be easier to learn others, as there are many similarities across them.

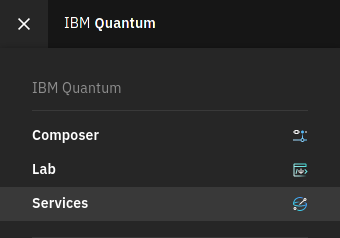
# 1. IBM QUANTUM EXPERIENCE

## 1.1 IBM Quantum Services

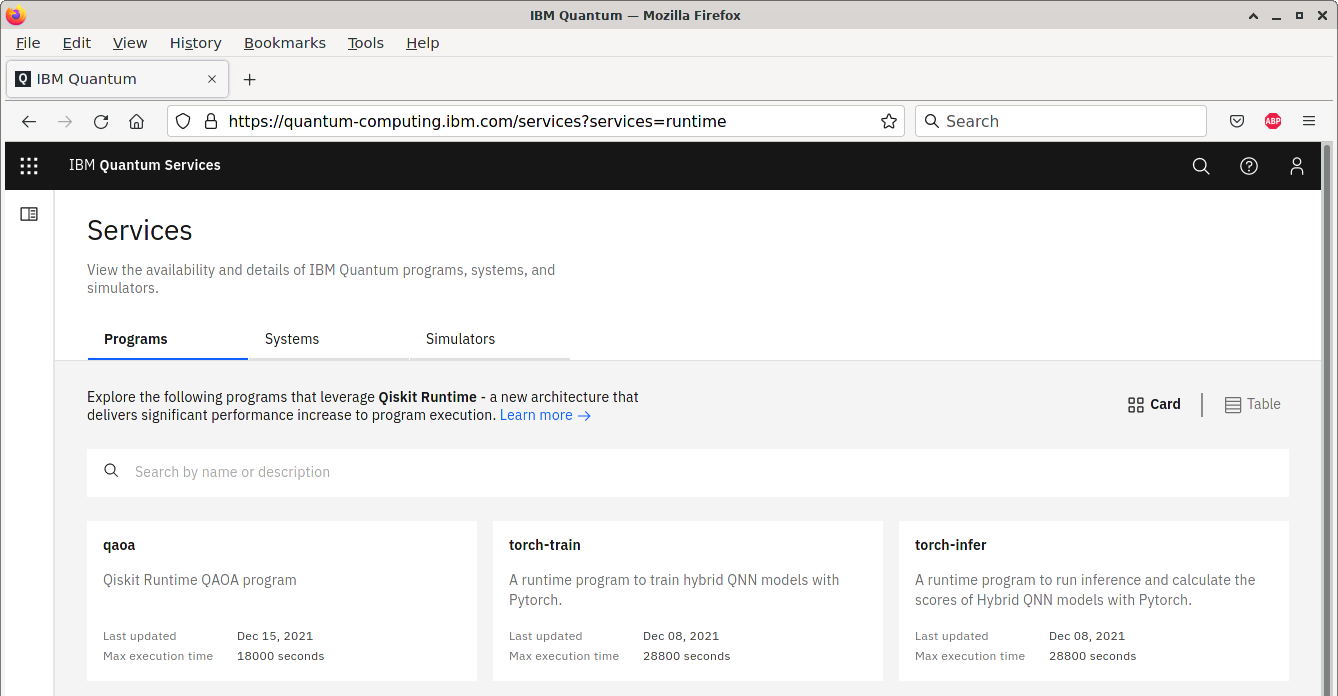
IBM was the first to make their quantum processors available over the internet (over the “cloud”), and their online platform is called IBM Quantum Experience. It can be accessed at https://quantum-computing.ibm.com. Their smaller quantum processors are available to the public, and access to their larger, newer processors is available commercially. When we log in, we first see the Dashboard:



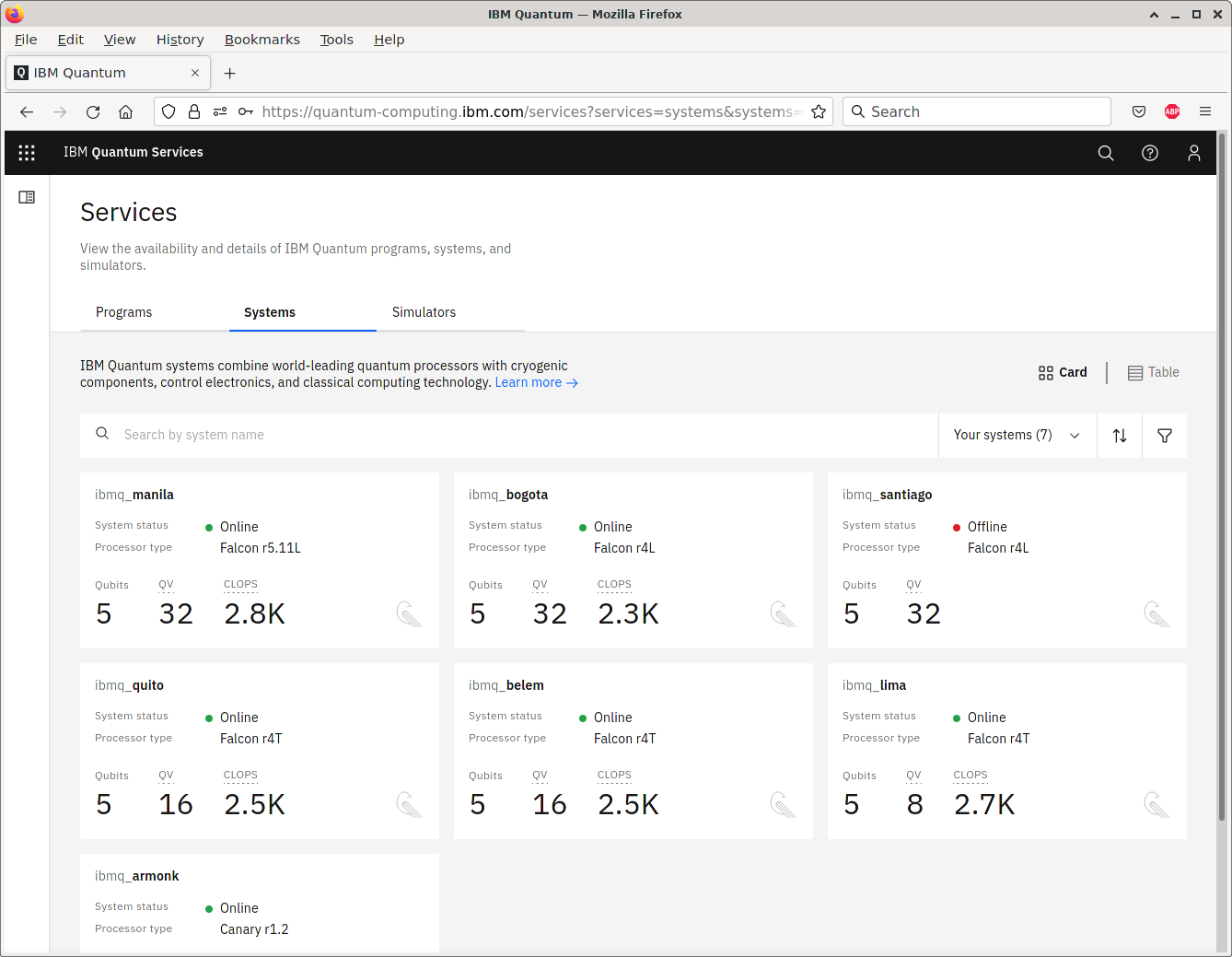
We can go to the Services page to view a list of quantum processors available to us. To get to the Services page, we can click on the menu icon in the top-left corner of the Dashboard, then click “Services:”



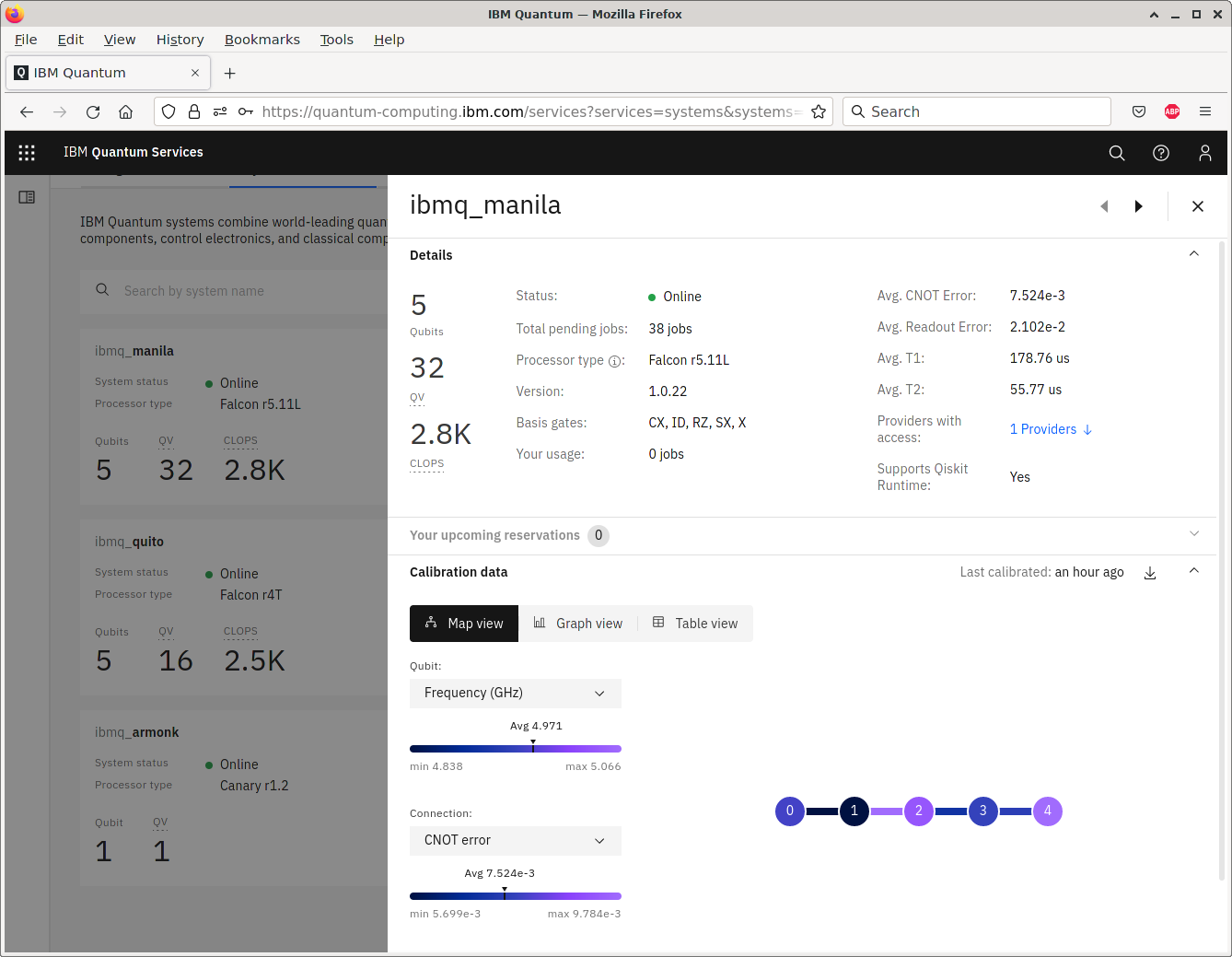
This brings us to the Services page:



On the Services page, we can click the “Systems” tab and then filter by “Your systems:”



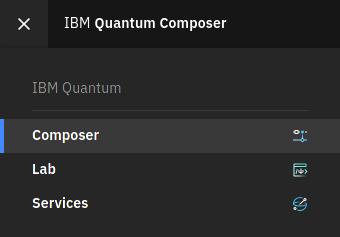
This is the list of quantum processors that are available to us. If we click on a processor, such as ibmq manila, we can see more information about it:

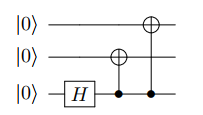


We see that this quantum processor has five qubits arranged in a line. This arrangement, or *topology*, can affect which quantum gates can be naturally applied. For example, we can naturally apply CNOT between qubits 0 and 1. If we want to apply CNOT between qubits 0 and 2, however, we would need to, for example, SWAP qubits 2 and 1, apply CNOT between 0 and 1, then SWAP 1 back with 2.

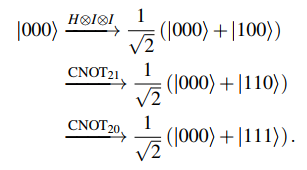
## 1.2 Circuit Composer

The Circuit Composer provides a drag-and-drop interface for programming quantum circuits. To get to the Circuit Composer, we can click on the menu icon in the top-left corner and then click “Composer:”

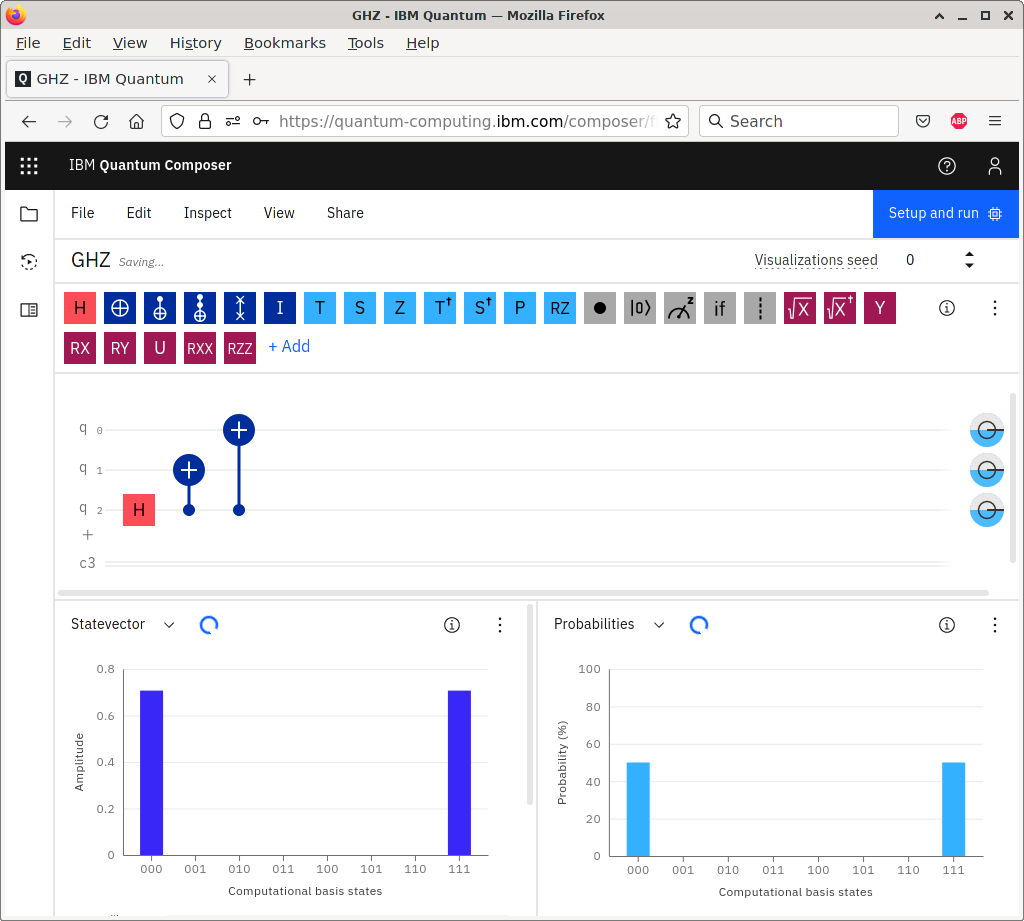
 For example, let us program the following quantum circuit:



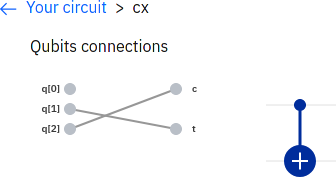
This circuit produces the following state:



This state is known as the Greenberger–Horne–Zeilinger state (GHZ state). It is an entangled state, and we will revisit it in the next chapter. If we measure it, we find that all the qubits are 0 with probability 1*/*2 or all 1 with probability 1*/*2. Using the Circuit Composer, we can create this circuit by dragging a Hadamard gate and two CNOT gates onto the circuit:

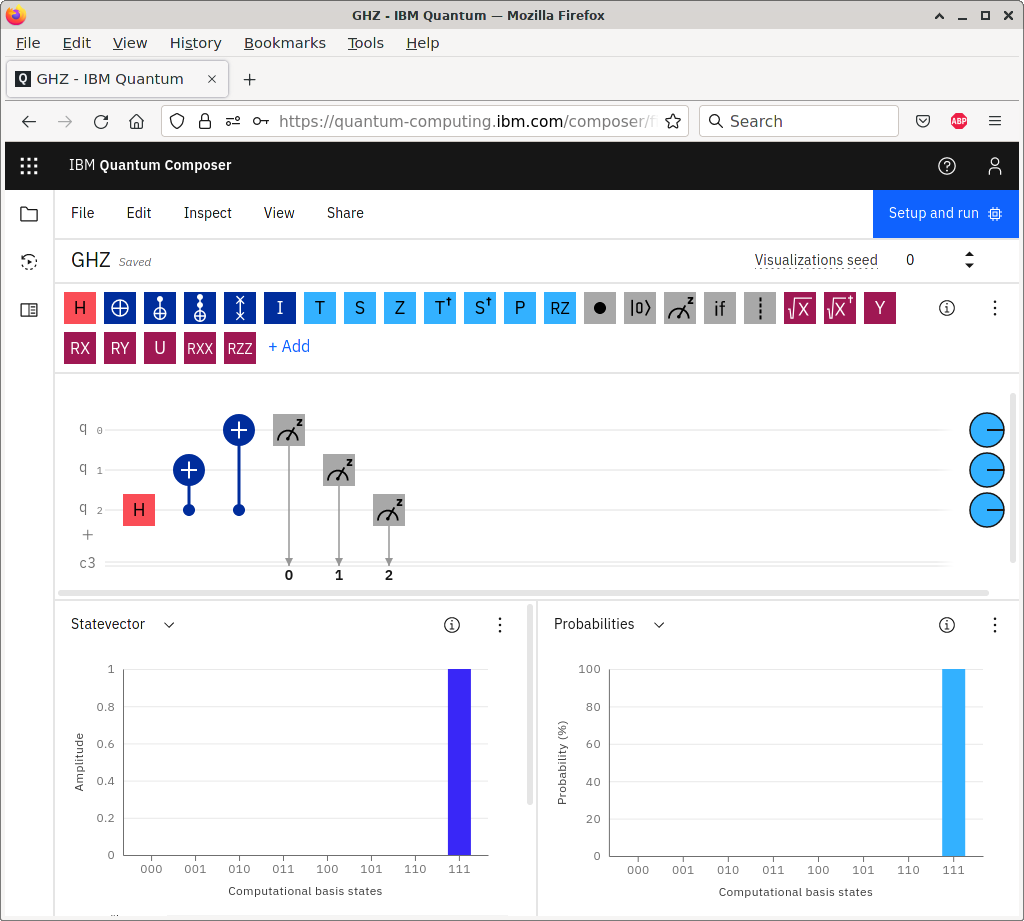


To change the control and target of CNOTs, we double-clicked on them and modified which qubit was the control and which was the target. For example, for CNOT21, the control and target were set as shown below:



In the Circuit Composer, we also deleted some qubits so that there are only three. At the bottom of the webpage, the Circuit Composer automatically simulated the circuit, showing histograms indicating that the circuit yields |000⟩ with probability 50% or *|*111*⟩* with probability 50%, as expected.

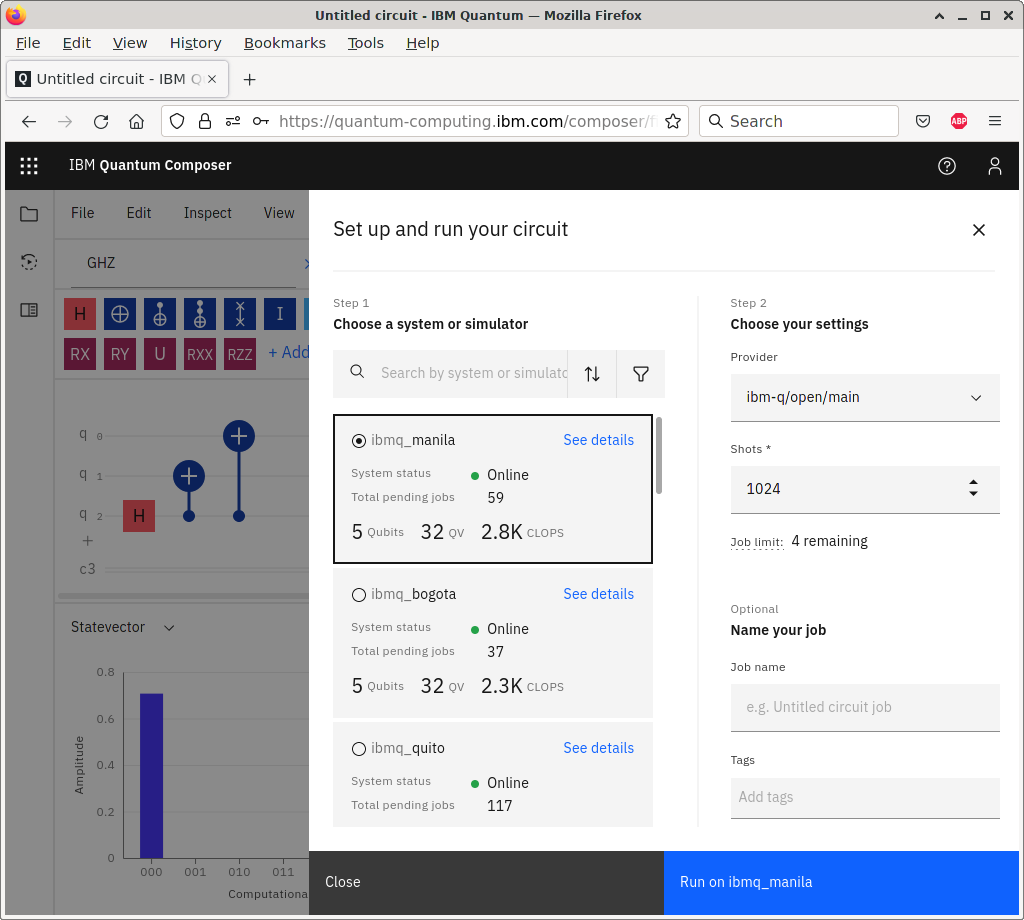
To run this on an actual quantum processor, we need to add at least one measurement. Let us measure all three qubits by adding measurement gates to the Circuit Composer:



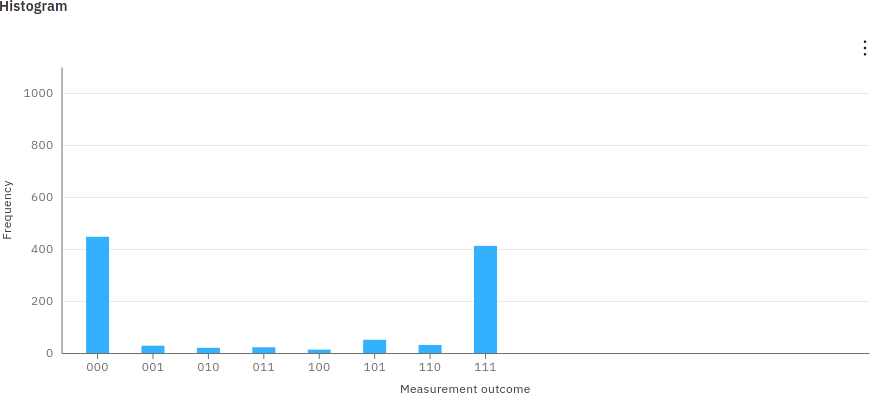
Now, the histograms at the bottom of the screen have changed. Instead of giving both *|*000*⟩* and *|*111*⟩*, we only get one of them. This is because when measuring the qubits, we only get *|*000*⟩* or *|*111*⟩*, not both. The Circuit Composer is choosing one of them using a pseudo-random number generator. At the top-right corner of the screen, we can change the “Visualization seed,” which is a number that the pseudorandom number generator starts with to generate pseudo-random numbers.

## 1.3 Quantum Processor

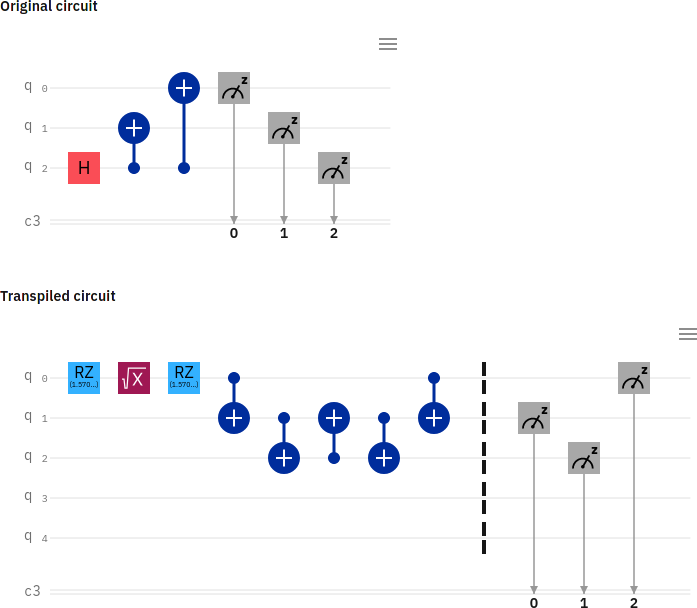
We can run the quantum circuit on one of IBM’s actual quantum processors. At the top of the Circuit Composer, there is a button that says “Setup and run.” Clicking it shows the following menu:



From this menu, we can select the quantum system on which to run the circuit. The number of shots defaults to 1024, meaning it will run our circuit 1024 times and return a histogram of the measurement outcomes. Ideally, we expect to get *|*000*⟩* 512 times and *|*111*⟩* 512 times. We also see our job limit. Each user is limited to having five jobs in the queue at a time. Clicking “Run on ibmq xxx” adds our job to the queue. When it is done, we can click it to see the results. Here is what we got for the histogram:



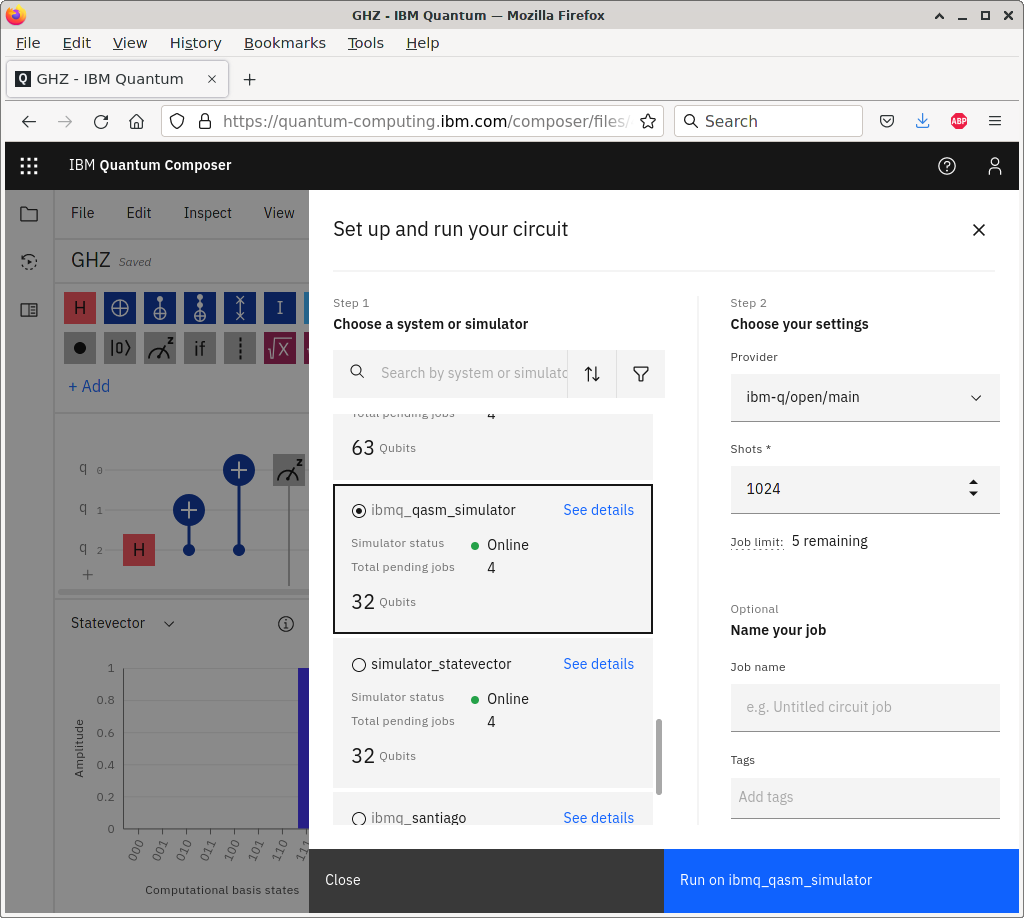
Theoretically, we expect 000 or 111, each half the time. Due to a limited number of shots and decoherence in the quantum processor, however, the results deviated from our expectations. The results page also shows the actual quantum circuit that was run, which is called the *transpiled* circuit:



That is, it might not be possible to run our original quantum circuit on the device due to the topology or gate set available to the processor, so the software will transpile or convert our circuit to an equivalent one that can be physically run.

## 1.4. Circuit Simulator

Sometimes, it can take a long time for a job to make it through the queue for an actual quantum processor. Or, the available quantum processors have too few qubits. In these cases, using a simulator rather than an actual quantum processor may be favorable. Let us try this for the previous circuit that creates the GHZ state. Clicking “Setup and run,” let us run the circuit on ibm qasm simulator:



This results in the following histogram:

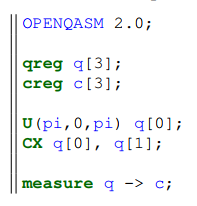


Out of the 1024 shots, the simulator measured *|*000*⟩* 493 times and *|*111*⟩* 531 times. Due to the limited number of shots and the use of a pseudo-random number generator to simulate the results, the results were not perfectly 50% each, but they are pretty close.

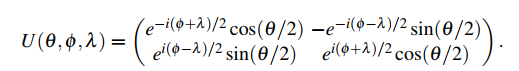
# 2. QUANTUM ASSEMBLY LANGUAGE

## 2.1. OpenQASM

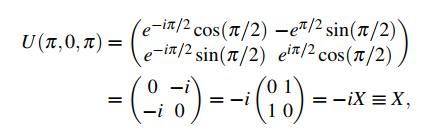
Rather than dragging and dropping quantum gates to create a circuit, they can also be written using programming languages. We can describe quantum circuits using *OpenQASM*, where Open refers to the specification being open or freely available, and QASM (pronounced kazm) stands for quantum assembly language. Despite the name “assembly language,” it is really more of a hardware description language like Verilog (see Section 1.3), where we defined registers and wires, listed logic gates with their inputs and outputs, and defined modules/functions. A document describing OpenQASM is available at https://arxiv.org/abs/1707.03429. Here is an example of a simple OpenQASM program:



The first line specifies that it is an OpenQASM program, version 2.0. Then we define a quantum register or array named q consisting of 3 qubits, *|q*2*⟩|q*1*⟩|q*0*⟩*. All of these qubits are initially *|*0*⟩*, so q is initially *|*000*⟩*. This is followed by a classical register named c consisting of 3 bits, also indexed *c*2*c*1*c*0, and all these bits are initially 0. Next, we apply a one-qubit quantum gate U(pi,0,pi) to qubit q[0], where the one-qubit gate is parameterized as



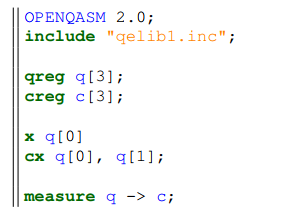
With appropriate choices for the angles, any one-qubit gate can be written this way, up to a global phase. Technically, this is a rotation about the *z*-axis of the Bloch sphere by *λ*, followed by a rotation about the *y*-axis by *θ*, followed by another rotation about the *z*-axis, but by *φ*. In this example, when (*θ,φ,λ*) = (*π,*0*,π*), we get



where in the last step, *≡* means “equivalent to” because the global phase of *-i* can be dropped. So, this gate transforms q[0] from *|*000*⟩* to *-i|*001*⟩*, but the global phase can be ignored, so it is just *|*001*⟩*. Next, CNOT (CX) is applied with q[0] as the control and q[1] as the target, transforming the state from *|*001*⟩* to *|*011*⟩*. Finally, q is measured, and the resulting bits are placed in the classical register c. So, c[2] = 0, c[1] = 1, and c[0] = 1. *U*(*θ,φ,λ*) and CX are the only two gates that OpenQASM has built-in because they form a universal gate set. That is, recall from Section 4.6 that the set *{*CNOT, all single-qubit gates*}* is universal for quantum computing.

## 2.2. Quantum Experience Standard Header

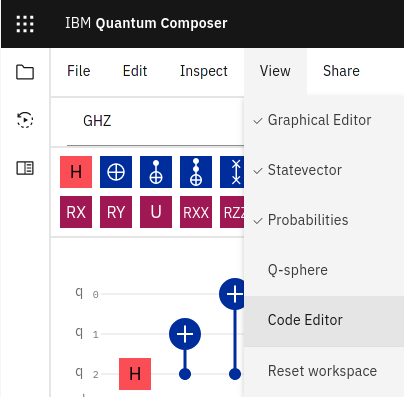
Rather than writing all one-qubit gates in the form *U*(*θ,φ,λ*) or defining them ourselves, it would be convenient if commonly used quantum gates like *X*, *Y* , *Z*, *H*, and others were predefined. Thankfully, these and many of the gates used by IBM Quantum Experience are defined in the library qelib1.inc, called the *IBM Quantum Experience standard header*, which we can include in OpenQASM. So our previous code can be written as:



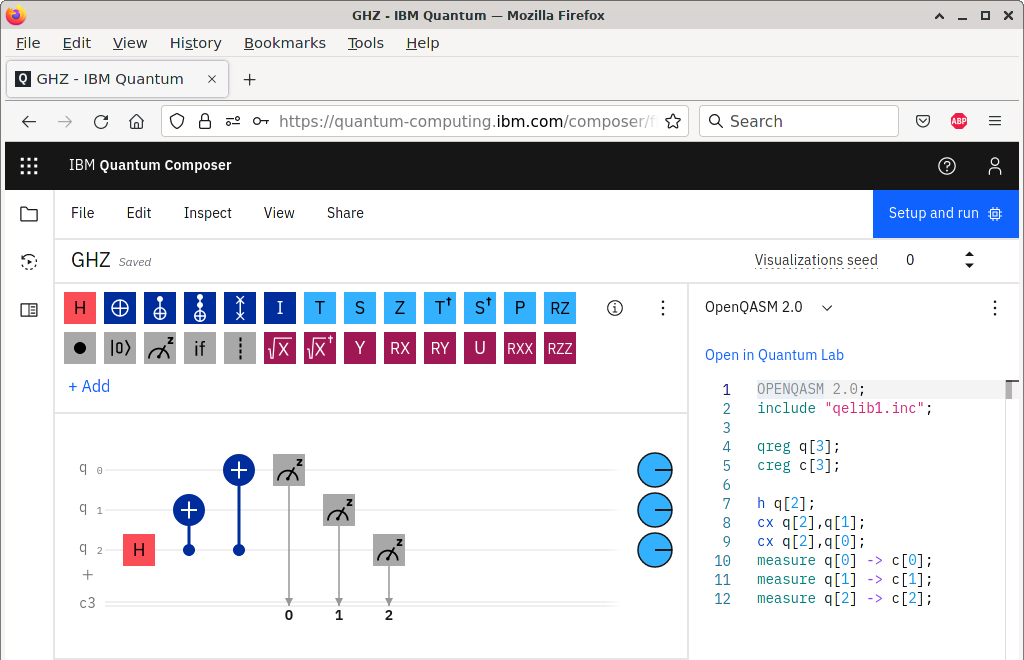
Note CX has been replaced by cx, since we are now using CNOT defined in the Quantum Experience standard header instead of the CNOT that is native to OpenQASM

## 2.3. OpenQASM in IBM Quantum Experience

Besides dragging and dropping quantum gates, IBM Quantum Experience also supports programming using OpenQASM. From the previous circuit for the GHZ state, we can go to the menu and select “View” and then “Code Editor:”

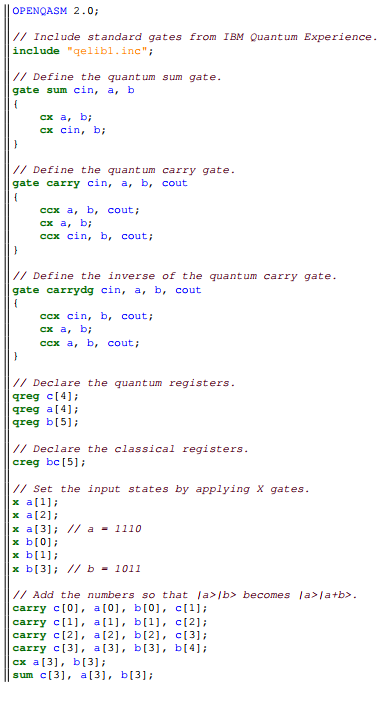


Then, the Code Editor will appear on the right side of the screen:

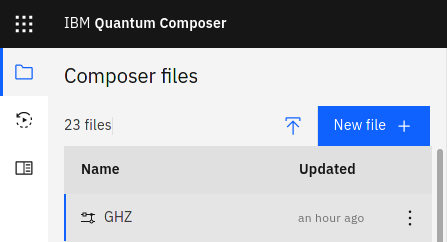


## 2.4. Quantum Adder

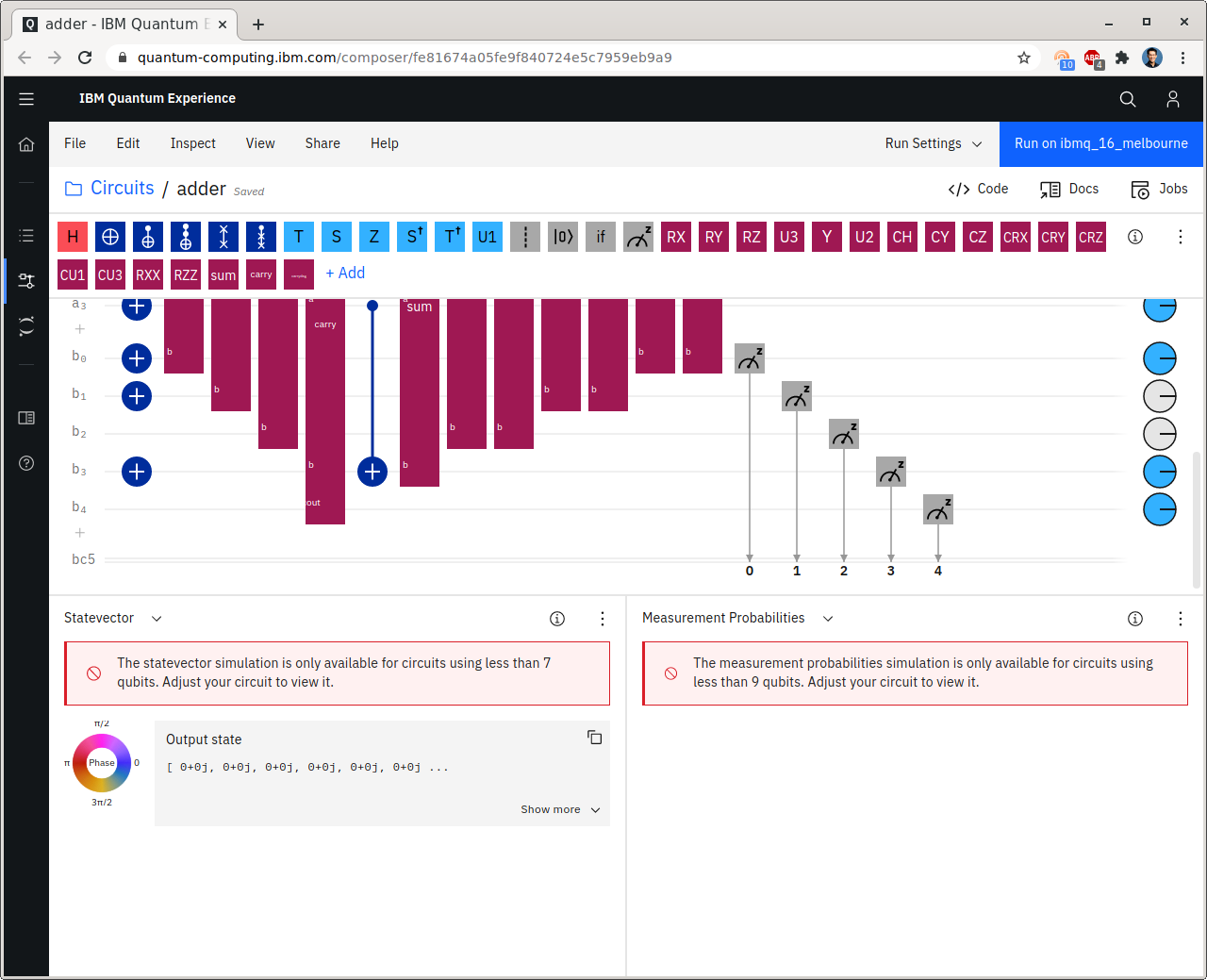
Now, let us write some OpenQASM code to add 1110 + 1011 = 11001 using the quantum ripple-carry adder in Section 4.5.6 and simulate it in IBM Quantum Experience. We can define our own quantum gates to implement the sum *S*, carry *C*, and inverse carry *C*†. Note s is the *S* gate, so we cannot use it as an identifier/name.



In the Circuit Composer, if we click the button on the left to view the “Composer files,” there is a button to upload an OpenQASM circuit:



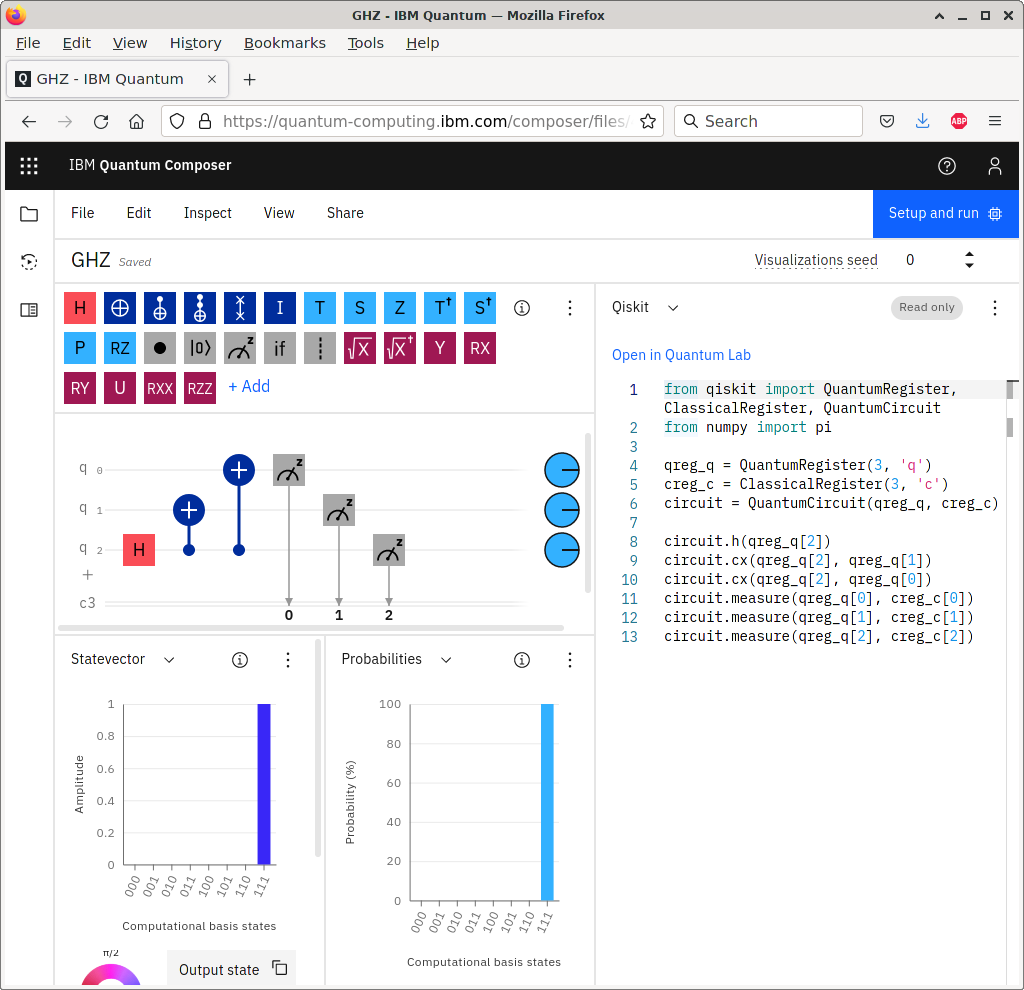
Uploading it, we get a new circuit:



# 3. QISKIT SDK

## 3.1. Circuit Composer

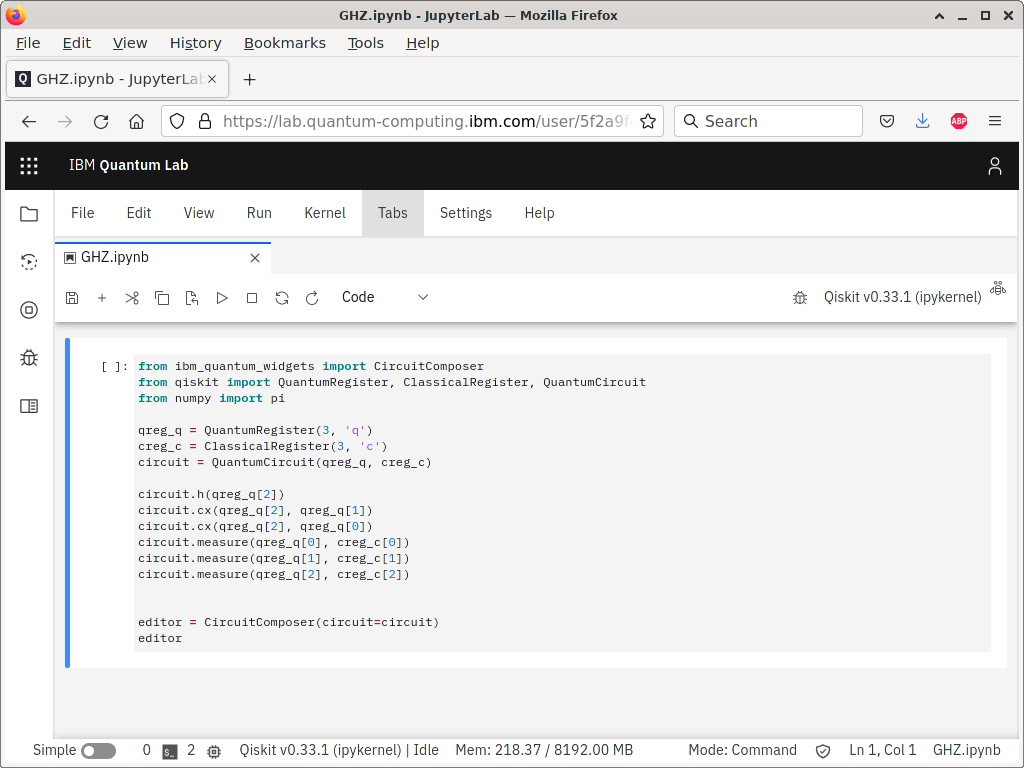
Besides the Circuit Composer and OpenQASM editor, IBM has provided another way to program their quantum processors. It is called *Qiskit*, where QIS stands for quantum information science, and kit refers to a software development kit (SDK). Qiskit is pronounced “kiz kit,” although some variants exist, like, “quiz kit” and “kiss kit.” Qiskit is not a programming language, but is rather a toolkit or package for the Python programming language. Qiskit is the most powerful way to program IBM’s quantum computers because it provides more functionality than the other approaches, and it also allows users to use Python’s vast network of packages and libraries. More information about Qiskit is available at qiskit.org. You can use Qiskit inside IBM Quantum Experience. To view a circuit as Qiskit code, in the Circuit Composer, just select “Qiskit” in the Code Editor. For the GHZ state, we get the following:



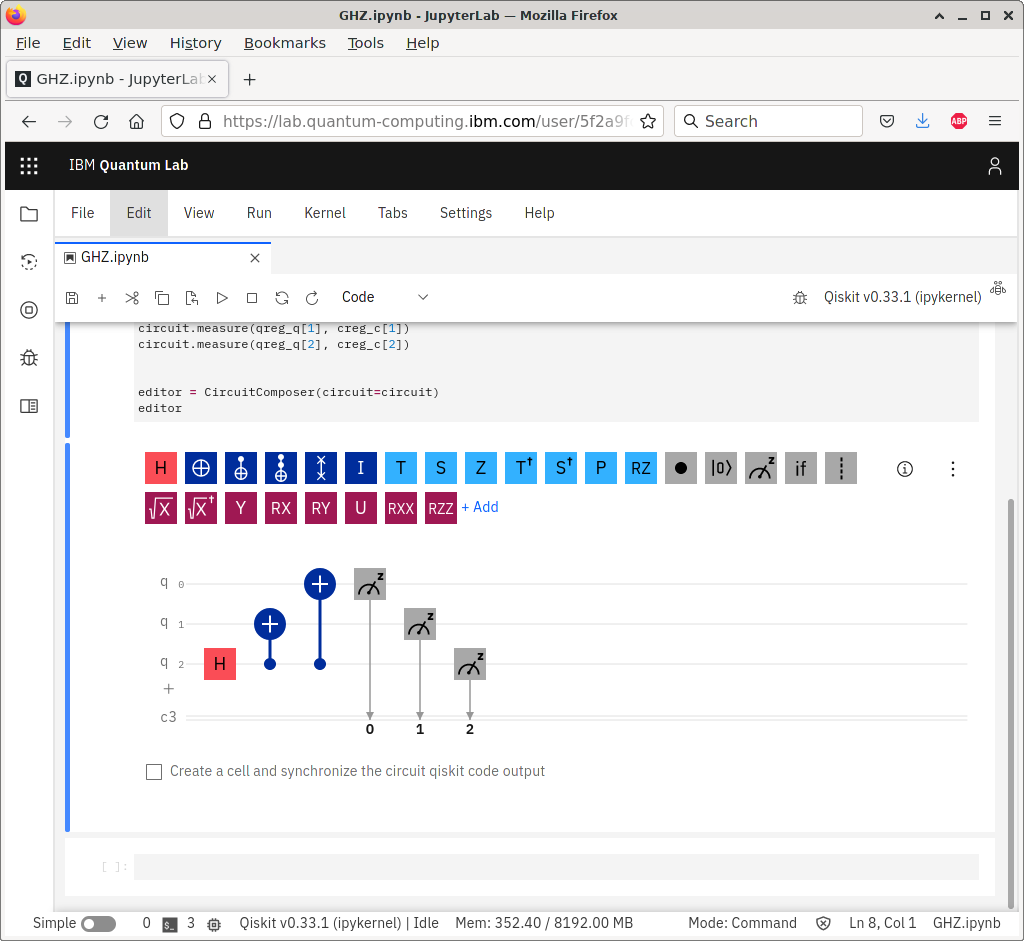
The first line imports from the Qiskit package functions to define quantum registers, classical registers, and quantum circuits. In the second line, we import from the numpy package the number pi. Although it is not used in this circuit, it is used in many circuits, so it is included here for convenience. In the next block of lines, the code defines a quantum register of length 3, labeled q, with the variable name qreg q. Then, the three qubits would be qreg q[0], qreg q[1], and qreg q[2]. Similarly, the next line defines a classical register of length 3, labeled c, with the variable name creg c, so the bits are creg c[0], creg c[1], and creg c[2]. After that, a quantum circuit is created containing the quantum and classical registers, and we name it circuit. Finally, in the last block of 6 lines, we add a Hadamard gate to our quantum circuit, and it is applied to qubit qreg q[2]. Then, we add a CNOT (C*X*) gate, with qreg q[2] as the control and qreg q[1] as the target. Then, we add another CNOT gate, again with qreg q[2] as the control, but now with qreg q[0] as the target. In the final three lines, we add measurements to the circuit, and the result of measuring qubit qreg q[0] is placed in the classical bit creg c[0], and so forth.

## 3.2. Quantum Lab

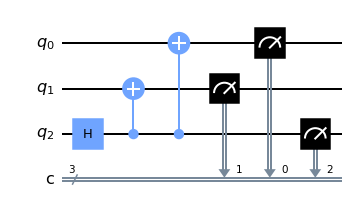
In the Circuit Composer, the Qiskit code is “read only,” so it cannot be modified. To modify it, we click “Open in Quantum Lab.” This opens a Jupyter notebook, where Python code can be executed and the results displayed in an interactive manner:



The first line loads a package that will allow us to view the Circuit Composer from within the Quantum Lab. The second-to-last line creates the Circuit Composer as an object named editor, and the last line displays the editor. We can run this cell by selecting the cell and clicking the *▷* Run button, or by pressing Shift+Enter on your keyboard. When we do, we get the following:



After running the cell, a second, empty cell appears below. We can put any Python code we would like. For example, we can draw the circuit without using the entire Circuit Composer using the draw function within QuantumCircuit to draw a picture of our quantum circuit: QuantumCircuit.draw(circuit) The output of this is



# CONCLUSIONS

Quantum computing is progressing from an academic research interest to a nascent industry, and the existence of this chapter on quantum programming is evidence of this. Actual quantum devices are being developed, and the tools described in this referat provide an introduction for how to use them.

Quantum computing, long confined to the halls of academic research, is now an emerging industry. As such, there are many companies involved in quantum computing, and they could be grouped into various types:

* *Traditional technology companies.* Many well-established computer companies have noted that quantum computing may be the future of computer technology, and they want to be leaders in the field. As such, they are investing heavily in building quantum hardware and/or developing quantum software expertise.
* *Technology startup companies.* Since quantum computing is a relatively new technological field, there is still plenty of room for new companies with new ideas. As a result, startups have also entered the nascent quantum computing industry. Some specialize in hardware, others specialize in software, while others are attempting both.
* *Companies that use computing technology.* Banks, car companies, airplane manufacturers, and accounting firms are all examples of companies that have been hiring experts in quantum computing. They are not interested in building quantum computers themselves, but they want to know how future quantum computers can be used for each of their businesses. If they wait for fault-tolerant quantum computers to be built before investigating their uses, they will be left behind by competitors.

These companies are desperately trying to hire qualified individuals. Some of the jobs are quantumly technical, such as building quantum computers and developing quantum algorithms. Other jobs are classically technical. For example, web programmers and software engineers were needed to create the IBM Quantum Experience website, and these jobs require little or no prior experience with quantum computing. As another example, electrical engineers with experience with radiofrequency devices can easily pivot to helping to build superconducting qubits, where radio frequency interactions are very important.

# BIBLIOGRAPHY