Rakhim Davletkaliyev

Quantum Computing

for Software Engineers

Quantum Computing for Software Engineers

Rakhim Davletkaliyev

This book is available at
https://leanpub.com/quantum-computing-for-software-engineers

This version was published on 2025-11-17

Contents

Preface

Quantum computing has always seemed a bit like science fiction. On one hand, there are numerous publications by skeptics—from the general public, journalists, and esteemed scientists—explaining in seemingly indisputable detail why quantum computing is at best unachievable in any foreseeable future, or at worst fundamentally impossible. On the other hand, there are dozens of companies of all sizes, from industry behemoths like IBM and Google to smaller startups focusing on one or several niche areas: algorithms, control electronics, calibration, software development frameworks, etc. The industry is experiencing quite a wave of interest, and naturally there is a lot of hype. Some companies make such bold claims that one can understand the skeptics' views, especially when those claims are not demonstrated.

So, who is right? Is quantum computing as unreal as intergalactic space travel, and do we have no chance of seeing it in our lifetimes? Or is it already here, and we just need to invest a bit more time and energy into making it practical?

This may be a bad analogy, but humor me: truly practical quantum computing may be compared to the colonization of the Solar System. In theory, it should be possible. Moreover, we have made a lot of progress in the last century and have solved many extremely hard engineering challenges, allowing us to send probes to distant worlds and even put people on the Moon. General-purpose quantum computing is like having "bases on multiple planets, with a reliable, efficient transport network." I'll be the first to admit how bad the analogy is, especially because quantum computing is simultaneously easier and harder than the colonization of the Solar System. It's easier because, just in terms of resources and the sheer number of engineering problems, it's a much smaller problem than interspace travel. It's much harder because nothing in the observed properties of physics seems to stop us from traveling in space, but the more you try to control the quantum world, the more the universe acts against your pursuit. I can believe that if humanity decides to invest one-tenth of its wealth into space travel, we may have colonies on a few planets and moons in a few decades. I am absolutely not sure the same can be said about quantum computing: investing one-tenth of all wealth may

not give us a machine with a few million stable logical qubits. Space travel needs to solve very hard engineering problems, but they are the same kinds of problems that have always existed: properties of materials, energy efficiency, food production, environment preservation, etc. Quantum computing needs to solve a completely new set of challenges, never seen before.

But this is why it's such a fascinating industry.

Having said all that, I want to stress one more point: quantum computers are already useful. Although no quantum computer has helped solve real-world problems yet (such as large quantum simulations, optimizations, NP-hard problems, cryptography, and other areas promoted by the hype), companies and scientific organizations purchase quantum computers to do research. Today, most QC devices can be considered research tools. There are three main areas of research:

1. Quantum physics
2. Quantum computing
3. Integration

Since quantum computers are essentially analog devices that allow you to control, in a limited fashion, a set of quantum objects, you can do some research in foundational quantum physics. Certain things that require tremendous computational resources on a classical computer may be done easier and faster on a quantum computer. Still, given the current state of the industry, classical computers outperform most quantum systems. But the research applied to smaller QCs can be scaled once the hardware scales.

Of course, the main area is quantum computing itself. From abstract, mathematical notions of algorithms to very low-level questions of calibration, many universities and research organizations are eager to have a quantum computer available to prove their theories and discover new properties. Commercial companies that deal with material science, battery technology, agriculture, and chemistry are buying quantum computers (or at least buying access to one) because they want to be ready if and when truly large-scale QCs become available. It is known that with a robust, large-scale QC one can develop, for example, better chemicals or EV batteries by efficiently simulating complex molecules and their interactions. Although you can't do it today, you can start developing the organizational knowledge and apply it to smaller-scale issues, in the same fashion as many organizations decided to try to "play" with the first

computers back in the day, and as a result came better prepared for the digital age.

And finally, integration research. This is the least known and least discussed topic in the industry but is very important. Its significance is one of the motivations for writing this book. Quantum computers, being research tools, are not normal products. They are driven by software, like anything else, but this software changes rapidly and is rarely written with long-term evolution in mind. If you buy a quantum computer today, chances are your code will not work on any other quantum computer, or even on the next iteration of the same machine. At the same time, researchers often need to work with multiple types of machines simultaneously, and HPC (high-performance computing) centers, i.e. supercomputing data centers, want to integrate quantum computers into their existing infrastructure and provide a "quantum compute" service to their users.

* * *

In the following chapters we will learn the principles behind quantum computers, and some practicalities as well. Before diving in, it's important to keep in mind the following unfortunate facts about the current state of the art.

1. Noise and error rates are overwhelming.
2. The amount of qubits is not a meaningful metric alone.
3. Many current architectures are ultimately not scalable.

First, you'll often hear about "noise" in quantum computing, as well as "error rates" and "error correction" (as well as "error mitigation"). This refers to the fact that quantum systems are inherently difficult to control and keep in a particular state, so whatever information is stored in it, or whatever computation happens, the results are often wrong. But as long as the results are *mostly correct*, there is a way to make it practical.

Second, most quantum hardware companies focus on the number of qubits as a metric of their progress. They themselves of course know better that this metric alone is not very meaningful, but it's understandable that press and media would use it: it's so simple and straight-forward. Just like in the past CPU manufacturers would reveal more and more megahertz and gigahertz, roadmaps and press-releases of IBM, Google, Rigetti and others contain quantum processors with 5, 20, 150, 433, 1121 qubits and beyond. Unlike "hertz" in CPUs, or "bytes" in RAM or disk storage, qubits, at least for now, are not created equal. In superconducting quantum chips (the most popular architecture at the moment) each qubit is uniquely imperfect. The fabrication process is difficult, and often a portion of planned qubits simply don't make it. The lab may try to make a 100 qubit device, but it ends up with e.g. 94 qubits, with 6 permanently "dead". Moreover, some of the remaining alive qubits would be of better quality than others. Some would have longer coherence time, i.e. would hold the state longer. Some would be better suited for certain operations. It may be possible that some computational operations are only available on a subset of qubits. To drive the analogy home, imagine that when you buy a new AMD or Intel CPU, it's clock speed, the number of registers, and the instruction set is unique to this unit. Your friend who bought the same product would get a slightly different "snowflake" chip. And now imagine writing an operating system for this madness.

Third, a lot of current approaches are simply not scalable to the desired levels of practicality. Yes, 1000+ qubits from IBM sounds impressive, but true utility requires orders of magnitude more. Even if the fabrication process improves significantly, and it becomes possible to make 1 million qubit chip, it'll be practically impossible to engineer a cryostat large enough for the millions of cables required to connect the chip to a huge number of control electronic units.

Part 1. Groundwork

In this part of the book we are going to learn the basics of quantum physics and quantum computing. And I mean *basics*, in a popular science way. This is not a replacement to even a first week of introductory university lectures! Luckily, there are enormous number of free resources available online and in libraries. Some recommendations are at the end of the book.

For us it is important to understand the following:

- what qubits are modelled after
- what are quantum gates
- how to implement a qubit with superconductive materials
- what other implementations exist

Chapter 1. Gaming die

If you need to explain regular (classical) computers to a layman, you can come up with pretty good analogies. You can think of a robotic person who, like a computer, performs commands in an exact, precise fashion. Or a calculator that does things you can do in your head or with a pen and paper. It's a lot more difficult to come up with an intuitive analogy for quantum computers because they are so utterly alien to everything we're used to.

But I do have one analogy that served me quite well in the past. It is very remote and very wishy-washy, but I believe it can help build a bit of an intuition.

Imagine you have an unsorted array of numbers. Your goal is to find the index of a particular number. For example, of number `12` in array `[18, 96, 12, 33, 19, 74]`.

Classical computer science says that the best we can do in this case is to just try one by one. Array being unsorted means we can't use any heuristics or optimizations like binary search. If we're really unlucky, the number we're looking for will be the last number we check, so at worst we're gonna perform `N` comparisons (6 in our case).

Now consider a 6-sided gaming die. The one used in simple board games like Monopoly. We throw it and say "whatever it gives, that would be the answer". This is useless. We're just generating random numbers. If the gaming die is fair (balanced), repeating this exercise should generate a roughly equal distribution between number 1,2,3,4,5, and 6.

Figure 1. Fair gaming die produces an equal probability distribution of 6 numbers

This is a “quantum computer” that is not calibrated well, and the program for it is not well written.

Now take a leap of faith with me and consider this magical development where we take this gaming die and modify it in subtle ways. We add a bit of material here and there, cut a tiny piece off of some corners, and make it overall unbalanced, unfair. And we do it in a way that is somehow connected to the shape of the problem - the array of numbers at hand. We aren’t trying to make the die fall on the correct answer directly, because we don’t know the correct answer. But we do know some properties of the array, some deeply mathematical relationships between the numbers, something about the shape of them. And we follow some esoteric mathematical and physical algorithm that tells us how to modify the gaming die accordingly.

Then we throw the die again, many many times. It is still random, but... less random. It gravitates towards one particular number, and it turns out it’s the correct answer!

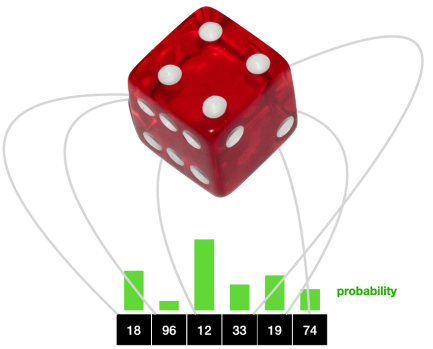


Figure 2. Constructively unfair gaming die produces an unequal probability distribution, gravitating towards the correct answer

This is a “quantum computer” that is well calibrated and well programmed. Quantum computing is basically trying to trick the universe to behave slightly less randomly inside a tiny isolated region of space, for a short period of time. It’s almost like we’re trying to piggyback on the fundamental computation that happens as time goes forward, but skew it towards the problem of ours.

Again, this is very remote and even metaphorical. But the idea of harnessing inherent randomness is important.

(Arguably, this analogy works better for one specific type of quantum

computer – quantum annealing – that isn't actually a universal type unlike superconducting systems. We will briefly discuss the differences in a latter chapter).

Chapter 2. Quantum physics 101

Below is an artist's rendition of an exo-planet with the coolest name WASP-39 b and its star[1].

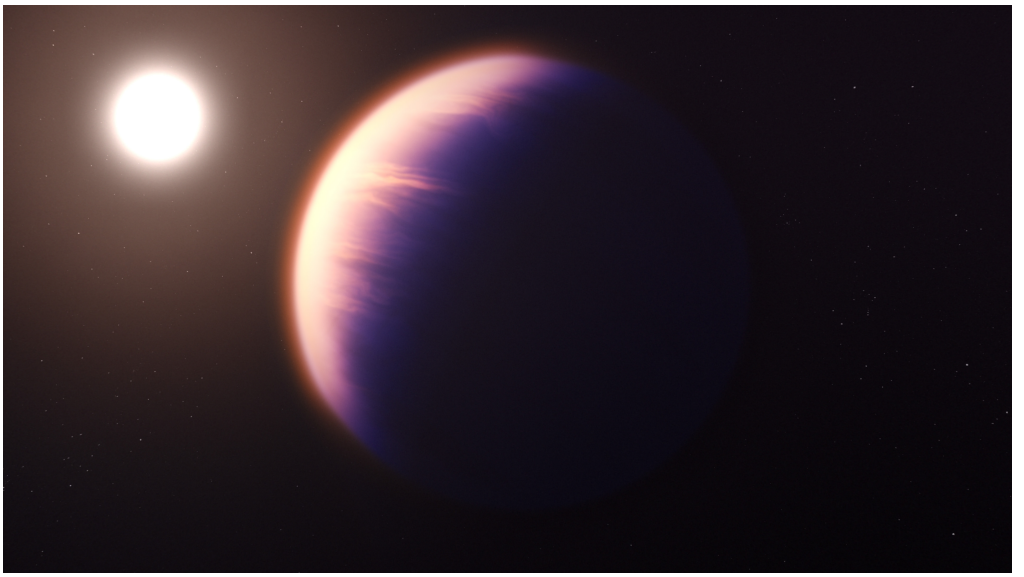


Figure 3. What exoplanet WASP-39 b could look like, based on current understanding. Credits: NASA, ESA, CSA, Joseph Olmsted (STScI)

They are located about 700 light years from Earth. That is 6.62 quadrillion kilometers. For comparison, Jupyter is roughly 40 light minutes away from Earth. So, yeah, WASP-39 b is very far. So far that we can only look at this artistic rendering because no optical telescope can photograph it in any meaningful level of detail. It'd look at best like a blurry bunch of pixels. However, if you search for scientific publications about this planet, or many other distant exo-planets for that matter, you will find some surprising depth of detail about the chemical composition of their atmospheres. On WASP 39 b NASA found sodium, potassium, carbon dioxide, carbon monoxide and other elements.

How?!

You can barely see the planet, it's a blurry set of pixels, yet we know what kinds of molecules can be found floating above it. This seems unreal. The

[1]https://science.nasa.gov/asset/webb/exoplanet-wasp-39-b-and-its-star-illustration/

answer to the "how" question is... spectroscopy[2]. It's a technique that takes advantage of a peculiar property of quantum objects. Scientists had discovered this almost a century ago, and you may have learned it in high school.

Consider an atom of sodium. It has 11 electrons around the nucleus. Now shine a light in a wide spectrum onto it. Wide spectrum means there are photons of different frequencies, or in other words, of different colors. Low frequencies are towards red color, high frequencies are towards violet color. By the way, this is where terms "infrared" and "ultraviolet" come from: infrared is "so red" that it goes below the capabilities of the human eye, and it's very low energy, for example, emitted by warm objects; and ultraviolet is so above the visible light and of such high energy, it is dangerously destructive.

So, you shine these photons of various frequencies through the atom and look at the light on the other side and notice that some parts of the light are gone. Before the atom you had a nice continuos spectrum, and after the atom there are these black gaps of no light. Photons of certain frequencies did not make it through.

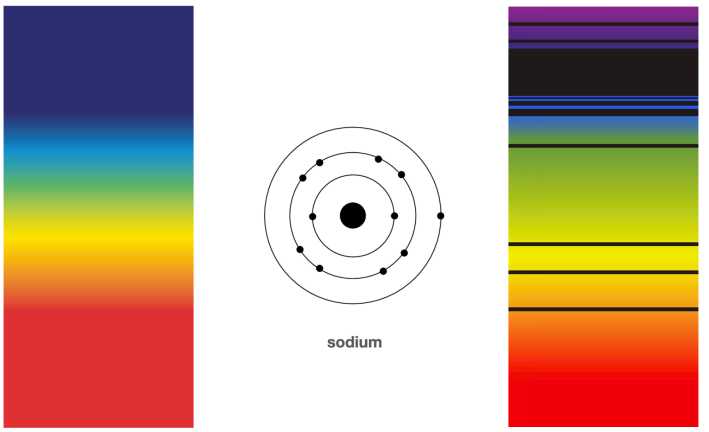


Figure 4. Light of certain frequencies absorbed by the sodium atom

Turns out that photons of certain frequencies got absorbed by certain electrons of that atom. The energy of the photon went into an electron and made it move into a higher energy position. The electron got excited and could not hide it. The resulting pattern of light looks like a barcode and, in essence, it's exactly that. A unique identifier, like a fingerprint, of the sodium atom.

[2]NASA's Webb Detects Carbon Dioxide in Exoplanet Atmosphere, https://science.nasa.gov/missions/webb/nasas-webb-detects-carbon-dioxide-in-exoplanet-atmosphere/

Once the electron goes into a higher energy state, it usually "wants" to go back down, and it does so spontaneously at some point. One of the fundamental laws of the universe is the conservation of energy, so when it falls back to the lower energy state, the same amount of energy that made it excited in the first place is re-emitted in form of a photon indistinguishable from the original one. So later you may track a series of emitted photons that form a broken spectrum that looks like the inverse of the "barcode".

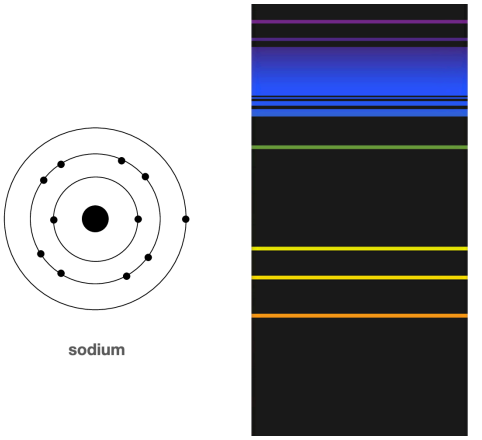


Figure 5. Light of certain frequencies spontaneously emitted by the sodium atom

We have (or can produce) most of the elements here on Earth, so scientists repeated this experiment for all of them and compiled a database of fingerprints for each one.

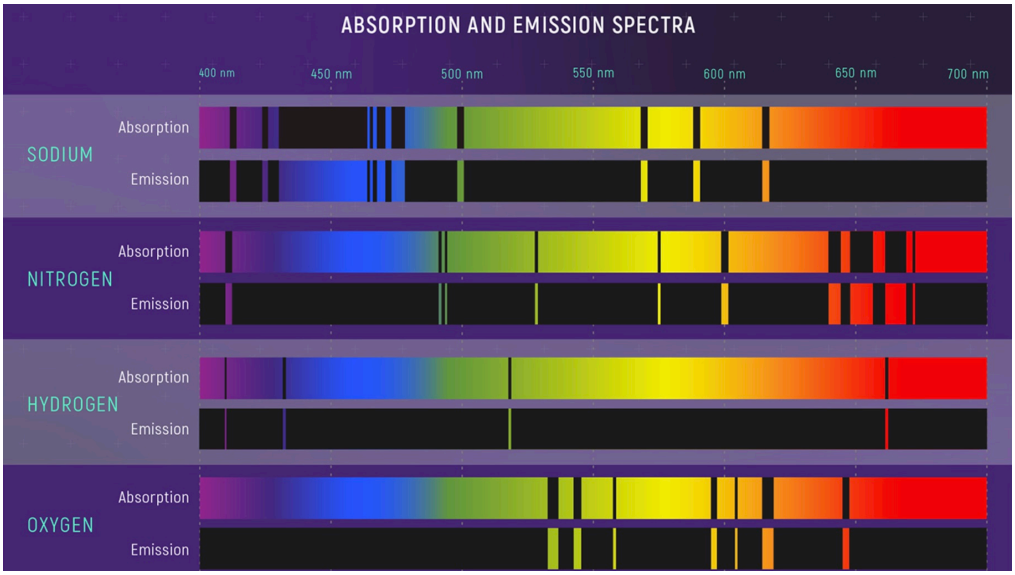


Figure 6. Absorption and emission spectra of sodium, nitrogen, hydrogen, and oxygen. Credits: NASA, ESA, CSA, Leah Hustak (STScI)

Now instead of using an optical telescope, NASA used a set of specialized telescopes that can detect variations in the electromagnetic radiation spectrum, and managed to do so in the exact moment when the WASP-39 b's mother star shines through its atmosphere. The resulting light has those exact bar codes for sodium, potassium, etc.

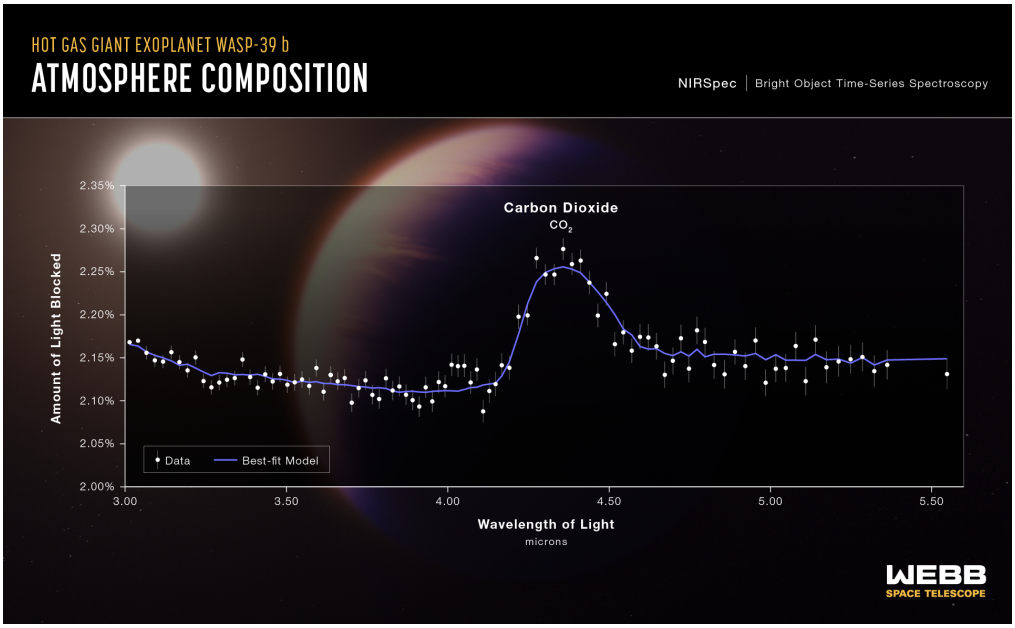


Figure 7. A transmission spectrum of the hot gas giant exoplanet WASP-39 b captured by Webb's Near-Infrared Spectrograph (NIRSpec). Credits: NASA, ESA, CSA, Leah Hustak (STScI), Joseph Olmsted (STScI)

This is truly amazing! And if we could simulate these interactions between light and electrons, or more generally, simulate quantum interactions, on a regular computer, we could develop better materials, chemicals, pharmaceuticals, design more efficient car batteries and discover better processes for growing food. Simulating would be so much simpler, faster, and cheaper than actually experimenting and producing those new materials and chemicals in real life. Imagine designing new drugs by precisely modeling molecular interactions, or discovering novel materials with extraordinary properties before ever synthesizing them in a lab.

The description of the process seemed pretty straight-forward, right? Feels like we could easily simulate this with an if-else. There is some state, and known energy levels, and if they match, then the state is changed.

When scientists started to think about this problem – and they started a long time ago, when computers were barely a thing – they realized that even if computers become billions of times faster, it still won't be enough. The problem is that the state of a quantum object is not discrete like we have observed, it's not just 0 or 1, ground or excited. And I don't mean that there are more states: yes, there could be more discrete states (think 0, 1, 2, etc.); those two or more discrete states can be observed, but in order to simulate a quantum system,

you have to express and store the state before the observation. And according to quantum physics, the state is expressed with a formula that happens to be the same as for describing waves, like that of water on the surface of a lake.

For a system that can be observed in one of two states, like that electron, to fully express the state you need to store two complex numbers called amplitudes. The formula below contains a so-called bra–ket notation, also called Dirac notation.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

To simplify, we can think of them as probabilities (in reality they are numbers from which you can calculate probabilities, but this distinction is not very important right now). So, the true state is not zero or one, but instead a probability distribution that tells you how often would you observe zero and how often would you observe one if you had an opportunity to measure the state multiple times.

A complex number is like a 2-dimensional number. It's an extension of real numbers (those that may have digits after the dot, like `14.665` or `3.14`). A complex number is a combination of some real number and an imaginary number; every complex number can be expressed in the form `a+bi`, where `a` and `b` are real numbers, and `i` is such a number that multiplying it by itself results in `-1`.

"There's Plenty of Room at the Bottom: An Invitation to Enter a New Field of Physics"[3] was a lecture given by physicist Richard Feynman at the annual American Physical Society meeting at Caltech on December 29, 1959. He imagined nano-scale machines capable of direct manipulation of individual atoms and producing materials of any kind on demand; and tiny surgical robots that would be ingested by a patient and perform operations on the smallest scale inside the body.

Two decades later, as computers became more viable and scientists and engineers started to see the limitations on the horizon, Feynman presented "Simulating Physics with Computers"[4]. He explained that classical computers can efficiently simulate classical physics; that is, physics models prior to

[3] https://en.wikipedia.org/wiki/There%27s_Plenty_of_Room_at_the_Bottom

[4] Simulating Physics with Computers, Richard P. Feynman, Department of Physics, California Institute of Technology, Pasadena, California, May 7, 1981 https://s2.smu.edu/~mitch/class/5395/papers/feynman-quantum-1981.pdf

quantum, such as Newton's laws of motion. After the discovery of quantum mechanics, it was clear that classical physics is merely an approximation, albeit a very good one. Newton's laws of motion can be used to successfully travel to the moon and back, because the approximation is good enough at that scale. But if we want to simulate complex quantum systems, or the long-term evolution of the universe, these approximations fall apart. Feynman then proceeds to argue that classical computers can still be used to simulate true quantum systems, because in the end the laws of quantum physics are just mathematical equations parametrized by time. I.e. you can calculate true quantum states by hand with a pen and paper. Computers are pretty good at math, and are certainly faster than humans, but here comes the critical part of Feynman's 1981 paper: computers are still too slow and too small.

This isn't a 1981 problem. Sure, computers were orders of magnitude slower back then compared to today, and had much smaller memory to work with. Feynman and his colleagues could project the growth of computer architecture and still see that exponential growth in computational power and memory is required. A system of N quantum objects requires describing 2^N amplitudes.

You can consider the size of the universe and roughly calculate how a computer can be built in principle. A classical computer of the size of the universe would have huge, but still limited powers and memory. You can then consider a relatively modest quantum system that needs to be simulated, and realize that even the computer of the size of the whole universe is not enough.

Yet this "computation" happens! The universe itself can be viewed as a computer that calculates the evolution of everything every "tick" and at every point in space.

Thus the only way forward is quantum computers: machines that themselves operate on quantum object, using them to store information and perform computation. So, instead of using many bits to describe the state of a single quantum object, a quantum computer would use a single qubit - a quantum bit. The challenge is, of course, how to get and control those qubits. Perhaps, just get those atoms with their electrons?

This paper can be considered the founding document of the field of quantum computation, a new kind of computation designed not just to crunch numbers, but to emulate the very fabric of the quantum world.

Chapter 3. Qubits and quantum gates

Before we go into practical matters of building and operating a real quantum computer, let's simply assume one exists. How would it look like from the point of view of a programmer? What computing primitives would be offered?

In classical computing there is a clear and simple model of logical gates: `AND`, `OR`, `XOR`, etc. All programming languages support such operations, but the important part is that processors themselves support those operations. One can build those gates physically with electric circuits or even large-scale objects like dominoes[5].

A very limited number of gates (called "basis gates") can be used to express all other gates, and the rest of computing. The entirety of math necessary to build any software from "Hello, World" to whole operating systems can be decomposed to a small number of logic gates. For example, an adder of two binary numbers can be constructed with two `XOR` gates, two `AND` gates and an `OR` gate.

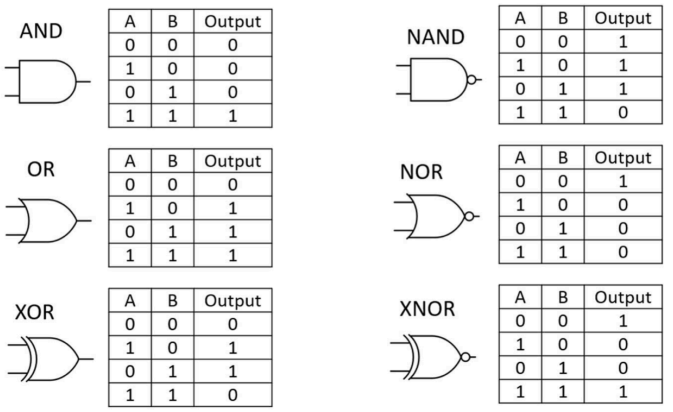


Figure 8. Basic logic gates. Credits: gsnetwork.com

In a similar fashion, there is a notion of quantum gates. Just like logic gates, quantum gates come in different forms: some operate on a single qubit, some on pairs, some on three or more qubits. You can think of a gate as applying an operation to qubits, so the state of the qubit before and after the gate may

[5] https://en.wikipedia.org/wiki/Domino_computer

differ. The most important part to understand here is that gates do not operate on observed bits, but on unobserved quantum states.

A classical logic gate can either change the bit or leave it unchanged. The result if always either `0` or `1`. But the result of applying a quantum gate is just a different state among virtually infinite number of possibilities. Say, the state of a qubit was $\alpha|0\rangle + \beta|1\rangle$ before the gate, and became $\alpha\prime|0\rangle + \beta\prime|1\rangle$ after the gate: slightly different complex numbers. This can already tell you how much more information and computation is packed into a quantum computer.

Unlike many classical logic gates, all quantum logic gates are reversible. It means that no information is ever lost in the process of computation until the measurement (observation) is performed. Compare this to e.g. the classical `AND` gate: its output is a single bit from which there's no way to reconstruct the inputs. (However, it is still possible to perform classical computing by picking only reversible gates.)

This reversibility requirement has practical consequences. If information somehow manages to escape the system during computation it means the state had been observed (e.g. by the environment into which the information had escaped), and thus the fragile quantum system had experienced decoherence and the complex amplitudes are gone; no quantum computation can be done at this point, only classical bit manipulation. Thus, the quantum computer must be completely isolated from the rest of the universe. In practice, depending on the architecture, it may require physical isolation from electromagnetic radiation, from any particles (so, a vacuum is required) and from any energy (so, a near absolute zero temperature is required).

A qubit can be modeled mathematically in different ways. One way is a Bloch sphere, named after the Swiss-American theoretical Felix Bloch.

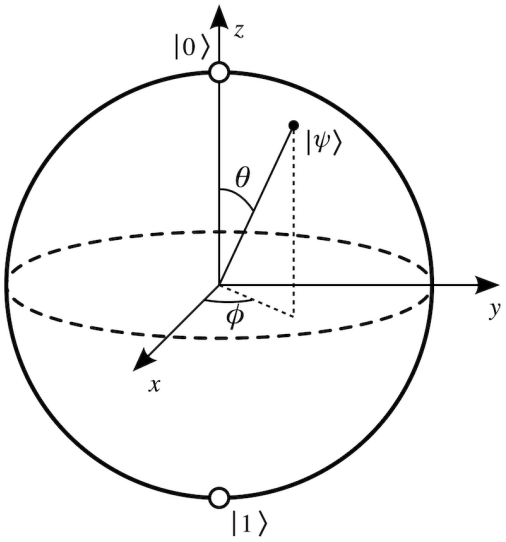


Figure 9. Bloch sphere. Credits: Wikipedia

It's a sphere, like a planet, and there's a vector pointing from its center to the surface. It can point to any position on the surface, and each such possibility represents a distinct state. When it points straight to the north pole it represents the state `0`, and when it points straight to the south pole it represents `1`. If the vector points to anywhere on the equator, exactly in between the `0` and `1`, then there an equal 50% probability of observing the state in either `0` or `1`. When observation is made, it is always either `0` or `1`, and you never see the vector pointing anywhere else. But before the observation, the vector can point anywhere, and applying quantum gates can change where the vector points to.

Do not let the visual nature of Bloch spheres mislead you. A planet-like sphere does not mean the qubit is a spherical, or has any sort of meaningful orientation in space, or a direction. The Bloch sphere is as abstract as it gets, and is just a way to visualize numbers.

A nice thing about vectors is that they can be described as matrices (or rather, columns of a matrix). Quantum gates can be described as simple or complex matrices, and this gives us a nice model of applying gates: multiply the state vector by the gate matrix, and the result is the new state. This is how you can do quantum computing on paper: it's just linear algebra!

Similar to classical logic gates, we can take a limited amount of quantum gates — a universal gate set — and use them as a foundation to describe all

other gates, and as a consequence, the entirety of quantum computing.

A quantum program looks a bit like musical notation. Horizontal lines are qubits, and elements on them are gates. Time goes from left to right. This representation is called a quantum circuit[6]. Generally, it does not have a notion of timing, only relative timing. It means that the order here matters, but the exact number of seconds (or rather nanoseconds) between the operations is not part of the circuit.

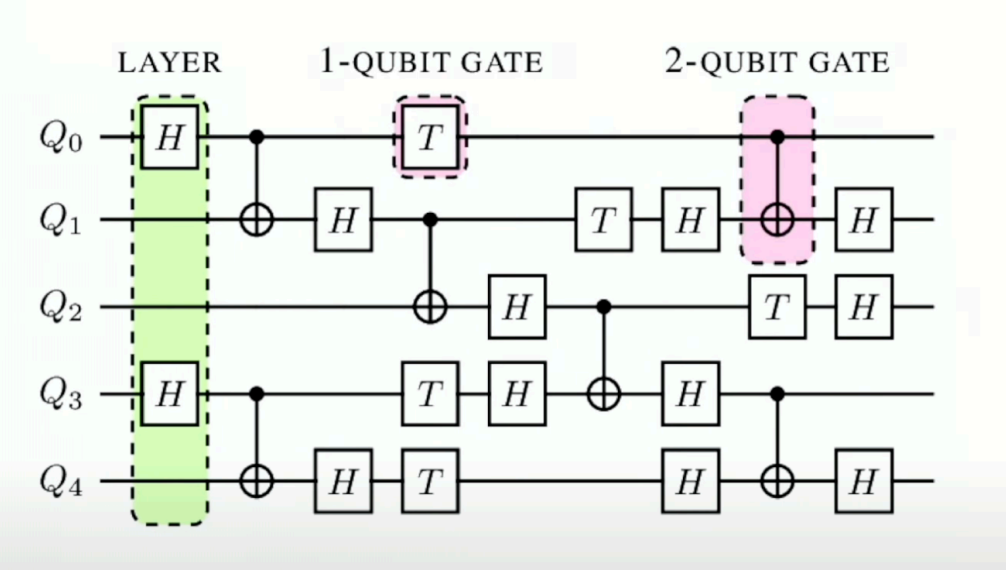


Figure 10. Example of a 5-qubit quantum circuit, with each horizontal line representing the time-evolution of the state of a single logical qubit. Credits: Ferrari, Davide & Cacciapuoti, Angela Sara & Amoretti, Michele & Caleffi, Marcello. (2021). Compiler Design for Distributed Quantum Computing.

[6]Do not confuse quantum circuits with electrical circuits which are used to build physical qubits. A quantum circuit is a mathematical concept, and a clearer name for it would be "quantum program".

Chapter 4. Crafting a qubit with superconductivity

Quantum computing is often imagined as something delicate and ethereal: electrons hovering in superposition, photons zipping through fiber-optic mazes. The sodium atom we've considered earlier, together with its electrons, is just another example of a "natural" qubit.

And while those images reflect real research directions, one of the most powerful approaches to building quantum computers is surprisingly tangible: circuits made of wire and metal, etched onto chips. In this chapter, we explore how quantum information or the elusive qubit can live not just in particles, but in entire electrical circuits, made practical by the strange and remarkable world of superconductivity.

For me this fact was probably the most mind-blowing. I knew about quantum computers in general before starting to work in the industry, but I always thought a quantum computer is built with stereotypical quantum things. The majority of popular science books and even university textbooks always describe quantum physics via properties of photons and electrons, so naturally when it comes to utilizing the quantum effects for computation, the same objects are re-used. Apart from engineering concerns, it doesn't really matter which one to use, just like with classical computers we could built processors out of vacuum tubes or transistors or dominoes. The theoretical computation is equivalent, but the engineering tradeoffs are huge.

At its core, a qubit is just a two-level quantum system. Any physical object that is capable of existing in two distinct quantum states—and, crucially, in superpositions of those states—can serve as a qubit. Electrons, with their spin states, or photons, with their polarization, are natural candidates. But quantum mechanics doesn't limit us to the microscopic. With careful engineering, even something as macroscopic as a loop of wire can be coaxed into behaving quantum mechanically.

Indeed, we can create a qubit out of a simple electrical circuit, such as a loop containing a capacitor (usually denoted `C`) and an inductor (usually denoted `L`). The capacitor stores energy in an electric field, the inductor in a magnetic field. Together, they form an LC oscillator—essentially a quantum harmonic oscillator when cooled and isolated enough. The energy in this circuit oscillates back and forth between the electric and magnetic fields, just like a mass on a spring.

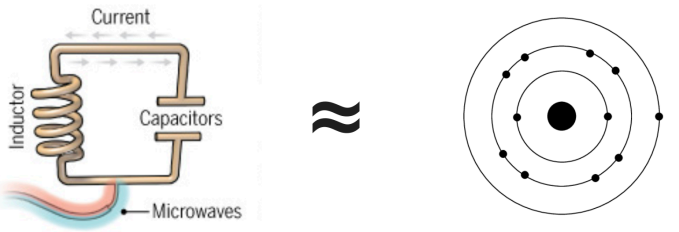


Figure 11. Specially constructed electrical circuit can behave like an atom for certain scenarios and under certain conditions

But real wires aren't perfect. In ordinary circuits, resistance drains energy over time, converting it into heat. This dissipation erases the quantum information stored in the circuit's oscillations. A qubit that leaks energy is like a memory cell that forgets its value—it's useless for computation.

To maintain coherence or the ability to hold quantum information we need to eliminate resistance. Luckily, the rules of the universe happen to have a property that can help us: superconductivity.

Superconductors are materials that, when cooled below a certain temperature, exhibit zero electrical resistance. Current can flow essentially forever in a superconducting loop without any loss. When we build our LC circuit from superconducting materials, we get a high-quality quantum oscillator that can retain energy—and quantum information—for much longer.

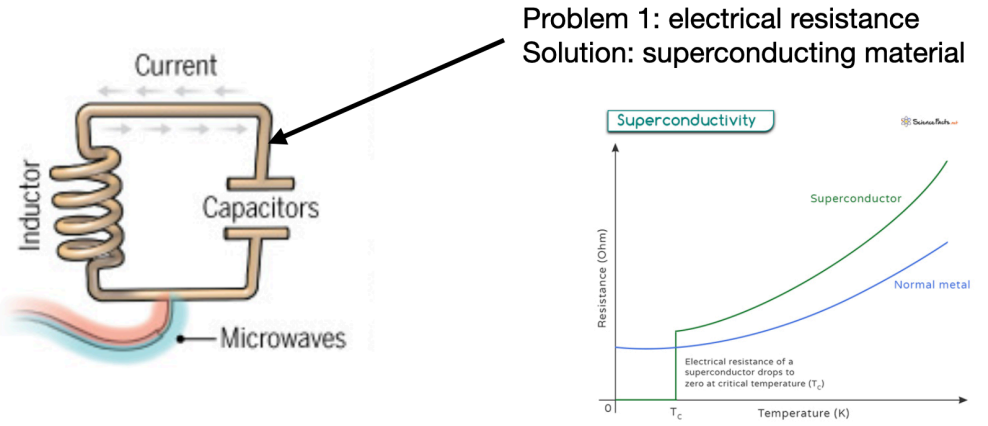


Figure 12. Superconductivity can solve the problem of unwanted electrical resistance

With superconductivity, our circuit is no longer just an analog of quantum mechanics—it becomes a bona fide quantum system. It exhibits quantized

energy levels, and under the right conditions, can even show superpositions and entanglement. But there's still a problem: such a circuit has evenly spaced energy levels. It's like a ladder with rungs at perfectly regular intervals. That's fine for physics experiments, but not great for quantum computing.

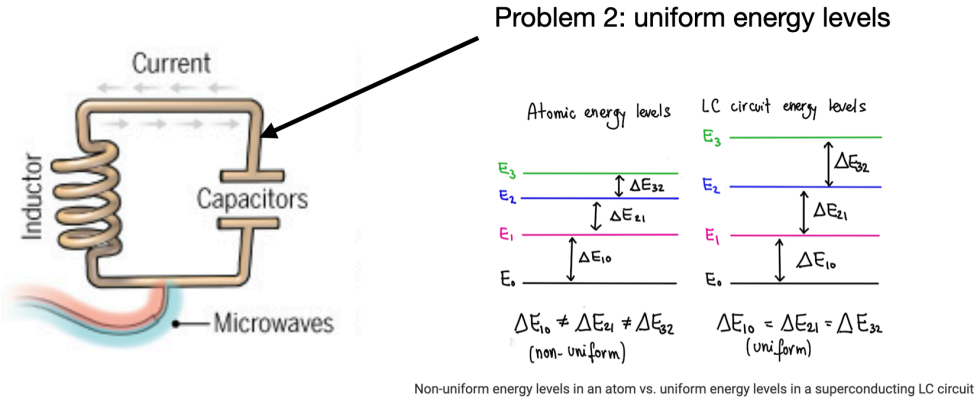


Figure 13. Observed energy levels are uniform

To perform quantum gates, we need to isolate just two energy levels—say, the ground state and the first excited state—and control transitions between them. But in a harmonic oscillator, applying energy that flips the qubit from $|0\rangle$ to $|1\rangle$ can just as easily excite it from $|1\rangle$ to $|2\rangle$, or beyond. That means our circuit isn't just a qubit—it's a "qutrit," or worse. It's hard to address just two levels in a harmonic system. To solve this, we need to make the energy levels uneven—*anharmonic*.

The breakthrough came with the Josephson junction: a thin insulating barrier between two superconductors. It behaves in a non-linear way, introducing exactly the anharmonicity we need. By adding a Josephson junction to the circuit we create a non-linear oscillator whose energy levels are no longer equally spaced.

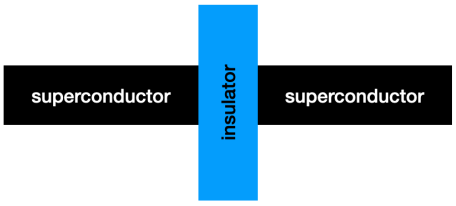


Figure 14. Josephson junction: a thin insulating barrier between two superconductors

Now, the transition from $|0\rangle$ to $|1\rangle$ requires a different energy than the transition from $|1\rangle$ to $|2\rangle$. This spacing allows us to selectively excite and manipulate just the lowest two levels, effectively creating a true qubit. These are called *transmon qubits*, one of the most widely used types in today's superconducting quantum computers.

Note that accessing higher state like $|2\rangle$ can still be useful for certain areas of research and even for certain computational tasks. Some scientists working on simulating chemical reactions would like to be able to access higher states inside of a quantum processor because those states might have better correlation with the underlying models they're trying to simulate. But for general purpose quantum computing just two states $|0\rangle$ and $|1\rangle$ are enough, and unlocking higher states does not expand the domain of problems that can be solved.

There are different ways to implement superconducting qubits, depending on which variable—charge or flux—you use to store and manipulate quantum information.

- **Charge qubits** rely on the number of Cooper pairs (paired electrons) on a small superconducting island. They're sensitive to charge fluctuations. While they can be manipulated quickly, they're also vulnerable to noise.
- **Flux qubits**, on the other hand, encode information in the direction of current flowing around a superconducting loop. This current generates a magnetic flux, hence the name. Flux qubits are typically more robust against charge noise, but can be more complex to control and fabricate.

There's a third kind, called the *phase qubit*, which uses the phase difference across a Josephson junction as its state variable. And the *transmon* qubit—a sort of refined charge qubit with reduced sensitivity to noise—has become the dominant platform in many quantum computing systems today.

Don't worry, we don't have to dive much deeper than this. I mean, you totally can if this sounds interesting, but we are going to move up the ladder of abstraction now and start treating qubits as generic objects with certain limited amount of properties. However, you will soon start noticing that in modern quantum computing most abstractions leak, both upwards and downwards.

Chapter 5. Other modalities

Superconducting quantum computers are the most popular approach nowadays in terms of commercial interest and recent pace of progress. Companies developing hardware in this area are IBM, Google, IQM, SpinQ.

Below is a short overview of other modalities of quantum computing.

Trapped Ions

Trapped-ion quantum computers use individual charged atoms (ions) as their physical qubits. Certain properties of the atoms are used to represent the state, and they happen to be stable and well-isolated from environmental noise (e.g., magnetic field fluctuations). This results in the longest coherence times (T_2) of any leading qubit modality, often measured in seconds or even minutes. Compared to superconducting quantum computers, trapped ions enjoy orders of magnitude longer coherence. However, the time it takes to perform any operation on qubits is longer as well. Operations are executed by directing precisely tuned lasers at individual ions.

In press-releases or magazine articles about quantum computers people often make comparisons across different modalities without context. Companies try to push the strong aspects of their chosen technology while downplaying the weak aspects. So, you can sometimes see how trapped ions are described as being orders of magnitude more stable than superconducting quantum computers because of aforementioned coherence time. While technically true, it's not a good metric, because the slower operations compensate for the increased coherence. It's like saying a new CPU can perform 10x more operations, but each operation is 10x slower.

Companies building trapped ion quantum computers: IonQ, Honeywell Quantum Solutions, Alpine Quantum Technologies.

Photonic

Remember how I said that in the past I naively thought that quantum computers must be made of photons? Well, this one is. Photonic quantum computers use particles of light as qubits. Photons can carry quantum information, and are generated, manipulated, and measured using optical components like

beamsplitters, mirrors, and waveguides. Quantum operations are performed using properties of photons such as polarization or phase. Photonic qubits are difficult to keep stable, and to scale to large numbers.

Unlike several other modalities, photonic QCs don't require extremely low temperatures to operate. They also open up the possibility of using long-established technologies like fiber optic cables to transfer quantum state across relatively large distances at the speed of light, and in a "native" format of the computer. Compare this to a superconducting quantum computer, where there isn't a natural, easy way to e.g. connect two systems and transfer quantum data; but IBM have been hinting at some sort of modular design in their roadmaps and presentations.

Companies building photonic quantum computers: PsiQuantum, Xanadu Quantum Technologies, Quantum Circuits.

Quantum Dots

A quantum dot is a nanoscale semiconductor structure that exhibits optical properties. These structures are typically a few nanometers in size and can confine electrons in three dimensions, leading to discrete energy levels similar to those of atoms. With careful manipulation, they can be used as qubits.

Quantum dots can be manufactured using standard semiconductor fabrication processes, which potentially makes this technology very scalable compared to others. The electronics necessary for manipulating them, however, is still very complex and not easily miniaturized. Just like superconducting QCs, quantum dots are highly sensitive to external noise.

Companies developing quantum dots quantum computers: Intel, Microsoft.

Topological qubits

This is the most exotic approach today. You're now familiar with how fragile a superconducting qubit's state is; a stray microwave photon or bit of magnetic noise can cause it to decohere and lose its quantum information. Topological quantum computing aims to build a qubit that is inherently protected from this kind of local noise. The idea is to encode the qubit's state non-locally, "smearing" it across exotic particles (like Majorana zero modes) in a special topological material.

For a software engineer, this concept is mind-bending. Okay, that's not fair: this approach is mind-bending for anyone! The information (the qubit's state) isn't stored in one place, but in the relationship or topology of a larger structure. Computation isn't done with pulses, but by "braiding" these particles around each other in spacetime. The promise is a near-total elimination of the massive error-correction overhead that plagues other modalities.

Sounds too good to be true? Well, yes. It's the least realistic modality as of today. The only major company that is actively pursuing this technology is Microsoft. In 2025 they have unveiled the Majorana 1[7] chip, the world's first quantum processor powered by topological qubits.

Annealing

The modalities we've discussed so far are all aimed at building a universal gate-based quantum computer, or a machine that can run any quantum algorithm. This means that in theory you can write a quantum program, cross-compile it for different architectures, and expect similar results from each (albeit at different speeds). In this sense, those types of QCs are like different CPU architectures (x86, ARM, RISC, etc.)

(To be fair, comparing different architectures of quantum computers with different CPU families is a stretch. CPU families differ in the way components are chosen and laid out, the instruction sets, voltages, etc. Ultimately, all modern CPUs are of the same physical architecture: silicon-based miniaturized transistors. The types of QCs we've covered so far are completely different to each other on a physical architecture level. If we had CPUs made of water pipes and CPUs made of vibrating membranes in addition to silicon, then the comparison would've been fair.)

Quantum Annealing is fundamentally different. It is not a universal computer; it is a specialized, analog optimizer. Instead of applying a precise sequence of logical gates, an annealer works by finding the lowest energy state (the "ground state") of a complex, programmable system of qubits.

As a developer, you don't program an annealer with gates. Instead, you map your optimization problem (e.g., "what's the most efficient route for my delivery fleet?") onto the hardware's specific format. While its universality is

[7] Microsoft's Majorana 1 chip carves new path for quantum computing, https://news.microsoft.com/source/features/innovation/microsofts-majorana-1-chip-carves-new-path-for-quantum-computing/

limited, quantum annealing is commercially available today from companies like D-Wave.

It is unfortunate that in the eyes of media and pop science, quantum annealing computers are put in the same category as other quantum computers. This is inherently wrong, like comparing analog rulers with digital calculators.

Part II. Levels of Abstraction of a Superconducting Quantum Computer

This is the most important part of the book. We are going to traverse the full path from an abstract quantum algorithm, to code, then go through multiple transformations and compilation steps, all the way to "bare metal" of the control instruments and the quantum chip, then raise back up into tangible data. At each step, we will zoom in and explore little details and caveats.

This part is vaguely based on an illustration I had made for IQM Quantum Computers back in 2022 titled "The Journey of a Quantum Algorithm". Although it is somewhat tied to the particular architecture and implementation of IQM's machines as of 2022, the overall structure is fundamental to all superconducting quantum computers, and many parts apply even to other types of QCs.

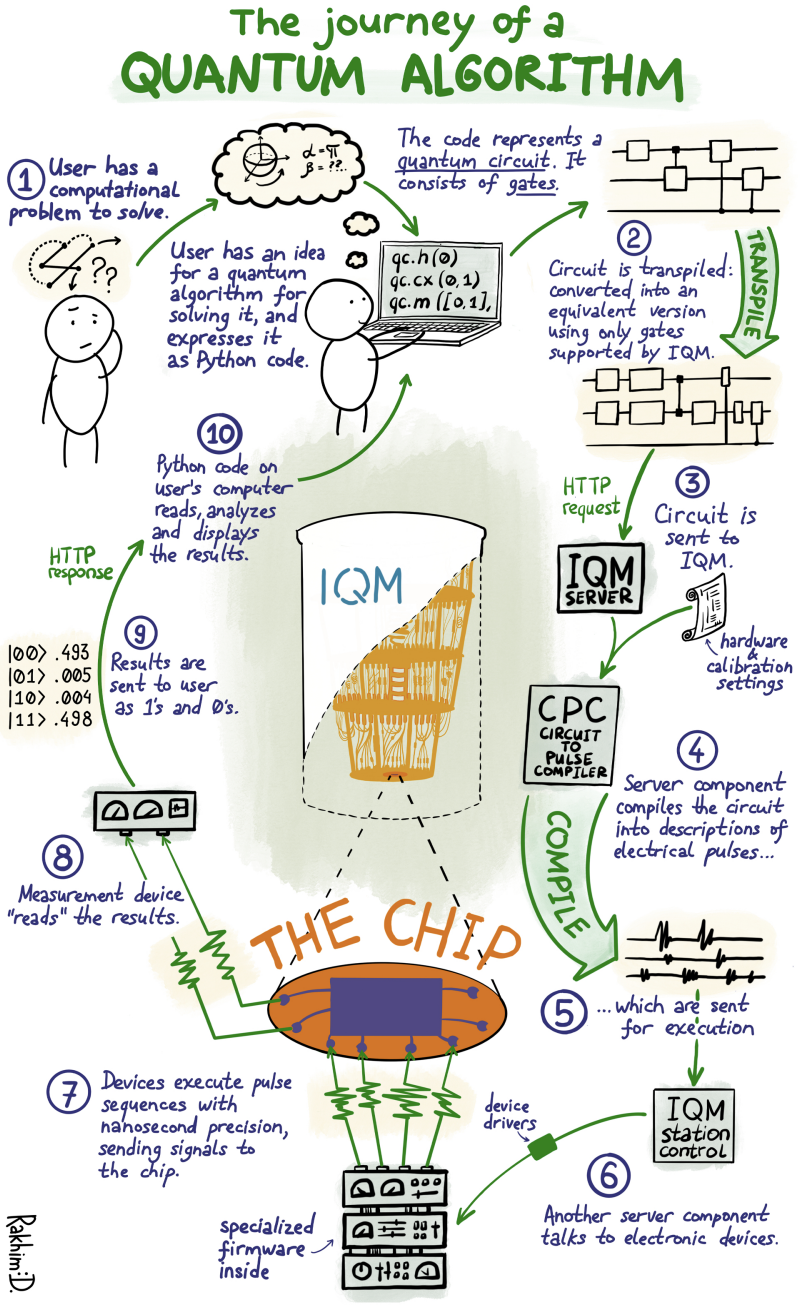


Figure 15. The Journey of a Quantum Algorithm. Credits: Rakhim Davletkaliyev, Olli Ahonen, IQM Quantum Computers

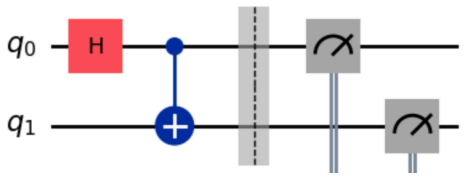
Chapter 6. Quantum Circuits

It all starts with the user having an idea. Just like with regular computers, the task often comes down to converting a mathematical description of an algorithm into actual code. Today most likely this means writing code in Python and using some popular quantum computing SDK (software development kit) like Qiskit (see Chapter 3 for more details on various SDKs and formats).

There are different ways to express a quantum program, but as of today the user is most likely to define a quantum circuit. The definition looks slightly different depending on the platform and language, but follows the same basic structure: there are qubits and gates. An aptly named single-qubit gate is applied to a single qubit only, two-qubit gates to two, and so on. Consider this abstract code that vaguely reminds of Qiskit:

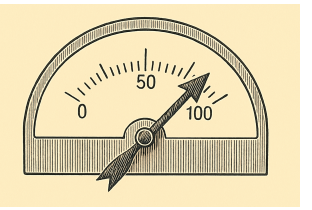
```
circuit.h(0)
circuit.cx(0,1)
circuit.m([0,1])
```

The first line adds a Hadamard gate to qubit `0`. The second line adds a two-qubit CX gate to qubits `0` and `1`. The last line adds a measurement gate to qubits `0` and `1`. Tools like Qiskit can generate visualizations of circuits, and our example would look like so:



The red square represents the Hadamard gate, and the blue structure connecting the two qubits represents the `cx` gate; in this notation, the smaller blue dot denotes the source qubit, and the bigger circle with a plus sign denotes the target qubit.

The two grey boxes on the right are the measurements; the icons inside the boxes look like dial gauges.

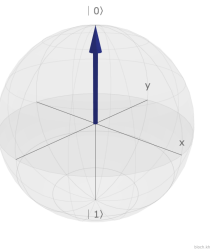


The vertical line separating the gates is called a barrier. We'll discuss it in more detail later while talking about optimization. Note that it was added automatically by Qiskit, and this is a pretty common behavior.

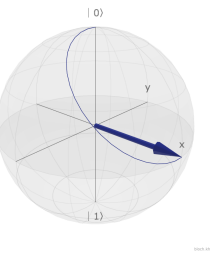
Hadamard gate

Usually, the assumption is that all qubits start in state `0`. The Hadamard gate transforms such state into a superposition of `0` and `1`. This means that if you measure it you will get `0` with 50% probability or `1` with an equal 50% probability. Just repeating this process over and over is akin to flipping a coin.

We can visualize the qubit's initial state of `0` (ground state) as a Bloch sphere with the vector pointing to the North pole:



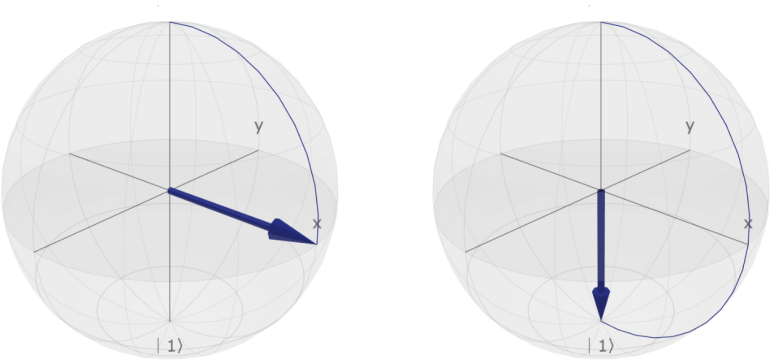
The Hadamard gate rotates the qubit so that it points onto the equator:



Remember, when the qubit is measured, its state can only be one of the two: either `0` or `1`, i.e. either north pole or south pole. The state after applying the Hadamard gate is not observable, it only exists as long as the qubit stays isolated from the universe inside a functioning quantum computer. Intuitively, having the arrow point to the equator means that upon observation the arrow has 50% chance of collapsing into `0` and 50% chance of collapsing into `1`. It is now perfectly between the two states, or in other words it's in a superposition. (note that 50%/50% split is not necessary to call this a superposition; any other probability split is also considered a superposition)

The Hadamard gate can be expressed as a 90° rotation around the Y-axis, followed by a 180° rotation around the X-axis. You might wonder why do a 180° rotation around the X-axis: this doesn't move the arrow as it is pointing exactly in line with the X-axis.

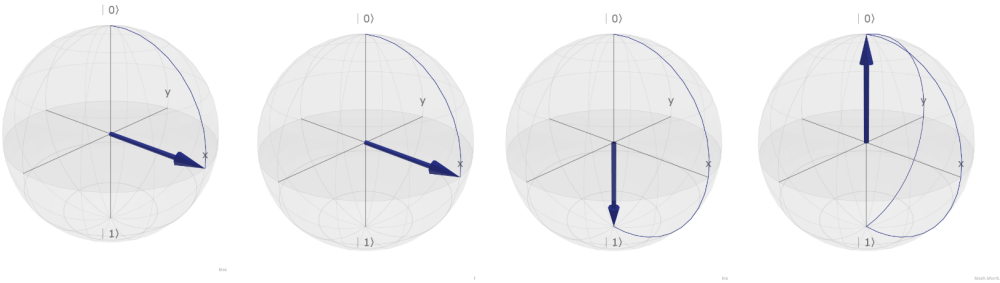
The Hadamard gate applied twice should return the qubit into its initial state. This means that the state is not really lost after putting the qubit into a superposition, and the operation is truly reversible. But if the operation was only a 90° rotation around the Y-axis, then applying it again would put the state into the opposite value. The image below illustrates this.



The Hadamard gate on the other hand always restores the initial state when applied twice, regardless of the initial position. Below is an illustration depicting the 4 steps:

1. Rotate 90° around the Y-axis.
2. Rotate 180° around the X-axis (no change). At this point Hadamard is applied once.

3. Rotate 90° around the Y-axis.
4. Rotate 180° around the X-axis (return to the original state). At this point Hadamard is applied twice.



CX gate

The `cx` gate is a conditional flip. A naive analogy is this:

```
if (qubit_0 == 1):
    flip(qubit_1)
```

But this analogy is *very* bad. The `cx` gate is **not** "measure qubit 1 and if it is 1 then flip qubit 2". There is no measurement involved here! After `cx` gate is applied to the pair of qubits, the state of qubit 1 is tied to the state of qubit 0, but the state of qubit 0 is still undetermined. Now, according to the theoretical mathematical models, measuring those two qubits should always yield the same pairs: either both `0` or both `1`.

So, instead of `if-then`, think of `cx` as:

```
entangle_inverse(qubit_0, qubit_1)
```

Where `entangle_inverse` is a unique function that makes the state of qubit 1 inversely dependent on the state of qubit 0.

Shots

Running a circuit just once rarely makes sense. We want to see a good statistical proof, so we should run the circuits hundreds or thousands of times in a row, collect all measurement results and observe the stochastic patterns. The number of executions is usually called "shots", and quantum APIs offer an argument with that or similar name. In abstract, sending a circuit for execution may look like this:

```
results = qc.execute(shots=10000)
```

The results can be:

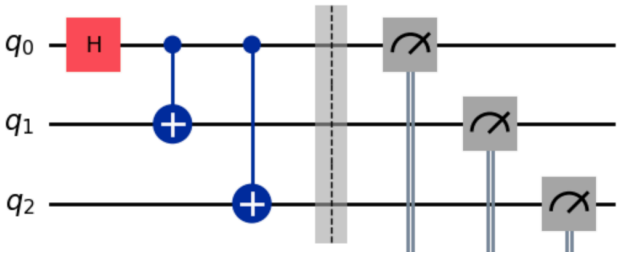
- a complete raw blob of measurements from all shots, i.e. data like `[0,0], [0,0], [1,1], [1,0],...`
- a processed histogram in a compact form stating how many results of each kind were obtained, e.g. `{[0,0]: 4883, [1,1]: 4912, [1,0]: 99, [0,1]: 106}`
- normalized histogram showing observed probabilities, e.g. `{[0,0]: 0.4883, [1,1]: 0.4912, [1,0]: 0.0099, [0,1]: 0.0106}`
- some other application- or algorithm-specific format. This may include "rawer" data read from the qubits without post-processing, but we're not gonna discuss those now.

GHZ (Greenberger–Horne–Zeilinger) state

A GHZ (Greenberger-Horne-Zeilinger) state circuit is a quantum circuit used to entangle multiple qubits, typically consisting of a Hadamard gate on the first qubit followed by a series of CNOT gates. GHZ is a common benchmark for quantum computers. The more qubits you can demonstrate to entangle this way, the better your QPU is. This kind of circuit is also an important part of some quantum algorithms.

Let's construct a simple circuit of that kind and use it as an example in our journey down the ladder of abstraction.

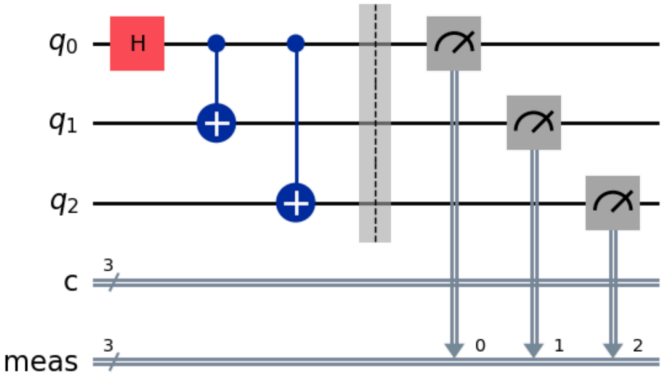
```
circuit.h(0)
circuit.cx(0, 1)
circuit.cx(0, 2)
circuit.m([0,1,2])
```



We put qubit `0` into a superposition by applying the Hadamard gate. Then apply the `cx` gate to pairs `0-1` and `0-2`. Essentially, we have entangled the qubits, and we expect the resulting measurements to always yield either `000` or `111`.

The measurement gate is a command for the instruments to read the state of given qubits and (normally) send the data downstream for post-processing. This is a destructive operation, because the complex quantum state is destroyed upon observation. So, our qubit 0 being in a superposition thanks to Hadamard gate will collapse to one of the two possible states and no longer be in a superposition.

Note the measurement gates in the visualization above. They have vertical lines leading down somewhere. The image is actually cropped; the complete version looks like so:



Recall that these visualizations are done by Qiskit, a popular quantum SDK. Qiskit's interface is built around the idea of quantum and classical registers, which vaguely remind of CPU registers, as well as measurement keys to identify the qubits from which a particular measurement outcome was gathered. The full image shows that qubits `0`, `1` and `2` were measured into keys `meas_0`, `meas_1` and `meas_2` of a special classical register.

You can also measure multiple qubits into the same classical register, and read an integer value out of the register. For example, if each of the three qubits have value `1`, and they are measured into the same classical register, then the stored value will be `111` which can be read as integer `3`.

Since in this book we try to focus, as much as possible, on the universal concepts rather than particular designs or implementations, we are going to mostly ignore those parts and focus on qubits and gates only. The Qiskit visualizer is handy, though, so we'll continue cropping the images when necessary.

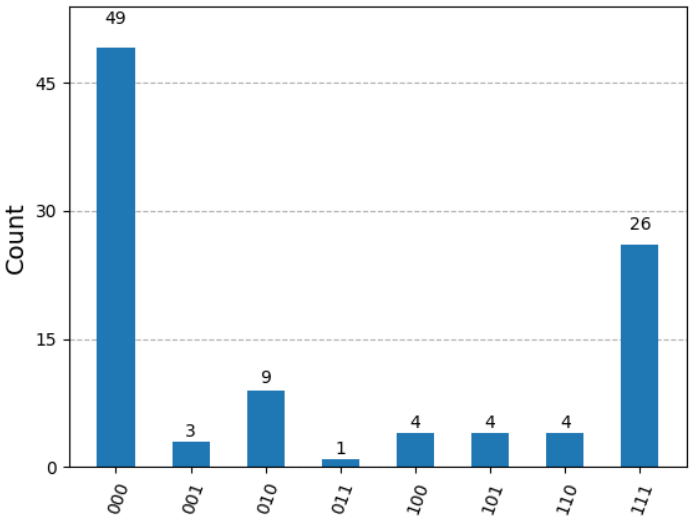
After running 100 shots (which is a very small number) on a quantum computer that is not very well calibrated, we get a data structure from the server:

```
{'meas_3_1_2': [[1.0], [0.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
↪  [1.0], [0.0], [0.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0],
↪  [0.0], [1.0], [0.0], [1.0], [0.0], [0.0], [1.0], [0.0], [1.0], [0.0],
↪  [0.0], [1.0], [0.0], [1.0], [0.0], [0.0], [0.0], [1.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
↪  [0.0], [0.0], [1.0], [1.0], [1.0], [1.0], [0.0], [0.0], [0.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
↪  [1.0], [1.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0],
↪  [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0], [1.0],
↪  [1.0]],

'meas_3_1_1': [[1.0], [0.0], [1.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
↪  [0.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0],
↪  [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [0.0], [0.0], [0.0],
↪  [0.0], [1.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0],
↪  [0.0], [0.0], [1.0], [1.0], [1.0], [1.0], [0.0], [0.0], [1.0], [0.0],
↪  [0.0], [1.0], [1.0], [1.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0],
↪  [1.0], [1.0], [0.0], [0.0], [1.0], [1.0], [0.0], [1.0], [1.0], [0.0],
↪  [1.0], [1.0], [0.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0], [0.0],
↪  [0.0]],

'meas_3_1_0': [[1.0], [0.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
↪  [1.0], [0.0], [0.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0],
↪  [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [1.0], [1.0], [0.0],
↪  [0.0], [1.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [1.0],
↪  [0.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0], [1.0], [0.0], [0.0],
↪  [1.0], [0.0], [1.0], [1.0], [1.0], [1.0], [0.0], [0.0], [0.0], [0.0],
↪  [0.0], [1.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [1.0], [0.0],
↪  [0.0], [1.0], [0.0], [0.0], [0.0], [1.0], [0.0], [0.0], [0.0], [0.0],
↪  [0.0], [0.0], [0.0], [0.0], [0.0], [1.0], [1.0], [0.0], [0.0], [1.0],
↪  [0.0]]}
```

Qiskit and other SDKs usually convert this data into a neat histogram-like structure:

```
{'111': 26, '000': 49, '010': 9, '101': 4, '100': 4, '001': 3, '011': 1, '110':
↪  4}
```

And we can visualize it like so:



You can see that generally there is a clear-ish distribution that matches the theory. Most results are `000` or `111`. However, `000` is very close to the expected: 49 out of 100, or 49%, while `111` is only 26%. Somehow, the entanglement between qubits was better in those cases that led to the first qubit measuring as `0`.

There was one case of `011` and one case of `100`, meaning the first qubit was measured as `0` or `1`, but the two other qubits got the opposite values. These are arguably the worst cases, but fortunately it's only 5% in total.

The cases where one of the two target qubits got entangled correctly are `001`, `010`, `110` and `101`. We can read this as semi-successful entanglement.

GHZ as a benchmark

Consider a larger circuit with the same structure:

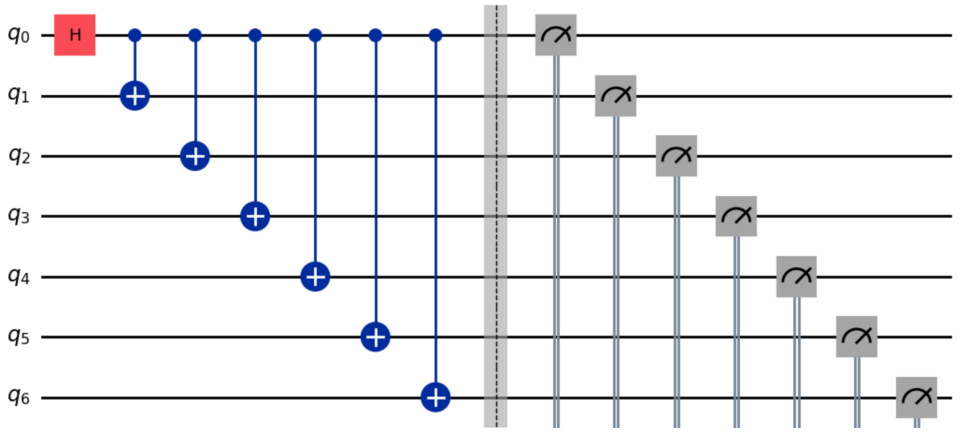


Figure 16. Circuit with CNOT depth of 6

This circuit has a `CNOT` depth of 6. Each `CNOT` (i.e. `cx` gate) operation takes time, and we normally want to minimize the time, as decoherence of the quantum state can introduce errors, and a longer time spent applying successive `CNOT`s can lead to more decoherence. The circuit can be improved so that `CNOT`s are applied in parallel when possible:

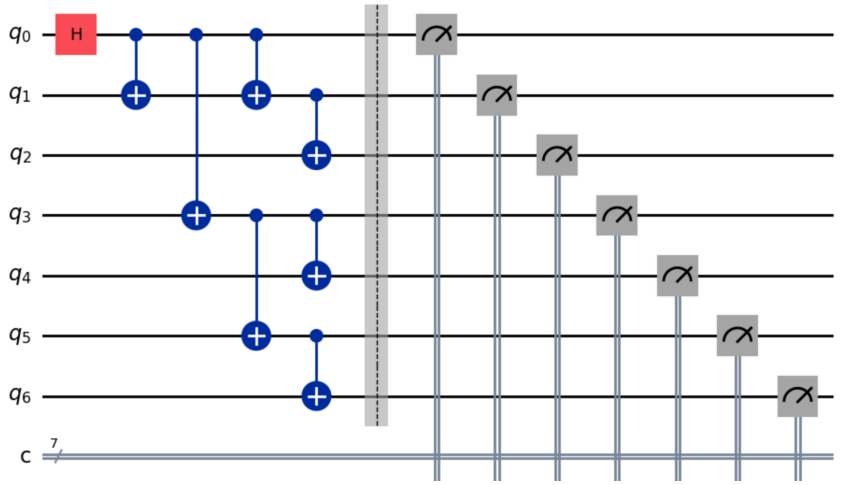


Figure 17. Circuit with improved CNOT depth

As mentioned earlier, GHZ is often used as a benchmark. For example, IBM had repeatedly published results of generating whole-processor multi-qubit entanglement. In 2021, their result featured generating whole-processor entanglement on 27- and 65-qubit quantum systems[1]. In 2025, IBM reported

[1]https://www.ibm.com/quantum/blog/whole-device-entanglement

a 120-qubit GHZ[^ibm_ghz_2]. In both cases, a lot of optimization heuristics were used, including a smarter placement of `CNOT` gates like described above.

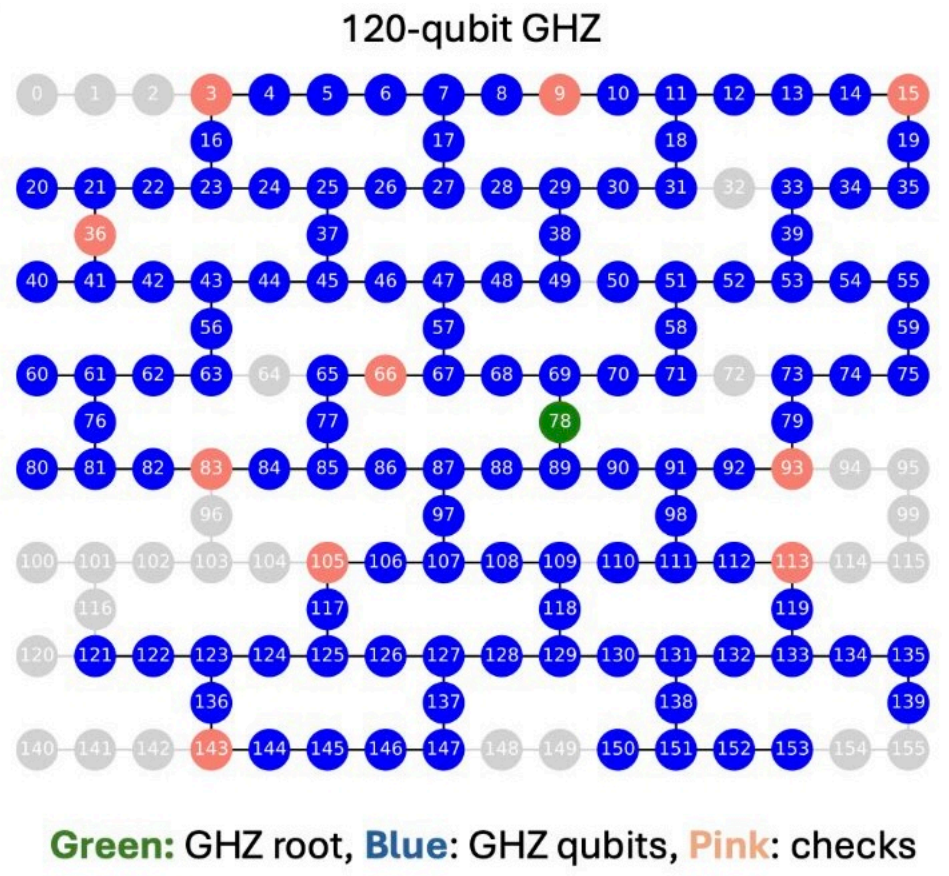


Figure 18. IBM whole device entanglement. Credits: Jay GambettaJ, Director of IBM Research, LinkedIn.

Chapter 7. Transpilation, routing, and optimization

Native gates

Now that we have a quantum circuit, it may seem like it should be straightforward to "apply" it onto a physical quantum chip. But it's not; or at least, not usually.

When hardware vendors design and build quantum chips and control electronics, they usually have a small set of operations in mind. Let's call them "native operations" or "native gates". This set needs to be universal, in other words, it should be possible to represent any operation from the theory of quantum computing as a combination of native gates.

Like described earlier, it's very similar to logic gates of classical computers. When building a CPU, you can choose to support `AND`, `OR` and `NOT` gates only. All other gates (e.g. `XOR`, `NAND`, etc.) can be expressed as combinations of those universal gates.

Note that in most cases, the choice of native quantum gates is not set in stone. It's not a physical property of the chip, or the manufacturing process. It's implied by those physical properties, but is ultimately driven by calibration.

The quantum computer at this moment in time may report that it supports only `PRX` and `CZ` gates. So, how do we run our Hadamard- (`H`) and CX-based circuit there? We first must convert those gates into equivalent native gates. This process is usually called transpilation, although some vendors use different terms such as synthesis.

Transpilation in computing usually refers to a process of converting source code from one language to another language on the same level of abstraction. For example, transpiling TypeScript code into JavaScript. As opposed to compilation, which usually refers to a process of converting source code from one level of abstraction onto a lower level. For example, compiling Java code into bytecode.

Transpilation can be done by hand, but quantum SDKs like Qiskit and Cirq have a transpiler module built in. It needs to know what is the native gateset of the target machine, and if there are any specific rules that define transpilation. Most vendors ship special adapters for Qiskit and other popular frameworks that allow to easily transpile circuits into their target architecture.

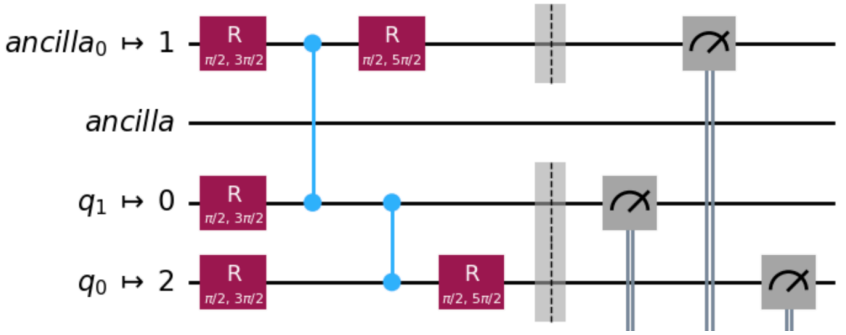


Figure 19. 3-qubit GHZ circuit transpiled to one of IQM's chips

A few other notes on this topic:

- Superconducting QPUs may support different gates at different locations. This, again, is dependent of calibration. It may be that, e.g. qubits 1, 2, 3, 5 support gates `A` and `B`, while qubit 4 supports gates `B` and `C`.
- Two-qubit gates normally operate not directly on the two qubits, but on a component that connects the qubits. In practice this information is reported as e.g. "gate `D` is supported between qubits 1 and 3", while in reality that gate is supported on the component (e.g. a so-called tunable coupler) that physically connects qubits 1 and 3.
- Even the native gates are still an abstraction. In superconducting QCs a gate application consists of a series of pulses; there may be more than one way to put the qubit(s) into the desired state, i.e. more than one way to define pulse sequences. Different ways are often referred to as "implementations", and, like always, this depends on calibration. For example, the `prx` gate in IQM's systems may be offered in multiple implementations: `drag_gaussian`, `drag_crf`, `drag_gaussian_sx`.

Routing

This transition from "abstract qubits in vacuum" to real qubits on a real chip involves another step: routing. When defining our circuit, we haven't thought about connectivity. Or rather, we assumed that all qubits are inter-connected, and we are allowed to apply e.g. two-qubit gates on any two pairs of qubits. Our toy example circuit utilized only two qubits, but even there we made this assumption. In reality, the connectivity of superconducting chips is heavily limited. Here is a realistic topology of a 5-qubit chip:

```
      QB0
       |
QB1 — QB2 — QB3
       |
      QB4
```

There is no direct connection between qubits 0 and 1, so an operation `cx(0,1)` is not physically possible. Assuming the user does not really care which physical qubits are used for the computation, one obvious way to map the circuit onto this topology is to choose a pair that is physically connected, for example `QB2` and `QB3` and assign "logical" qubits to physical qubits like so:

- logical `qubit 0` to physical `QB2`
- logical `qubit 1` to physical `QB3`

(Note that we use the term "logical qubit" to denote a single qubit in our primitive circuit; there is another, more common meaning for this term that is related to quantum error correction, in which a single logical qubit can be mapped to multiple physical qubits in order to achieve redundancy and better fidelity. This approach is similar to classical computing where in order to minimize noise and random errors, a single bit of information is stored in multiple bits of physical memory, and extra bits are used to correct potential spontaneous errors or bit flips. Since this book is mostly focused on the current state of the quantum computing industry where true quantum error correction, or fault-tolerant quantum computing, is not yet a complete reality, we are going to keep using the term "logical qubit" like we did.)

Our case was very simple, but now consider this circuit:

```
circuit.cx(0,1)
circuit.cx(1,2)
circuit.cx(0,2)
```

We want to apply two-qubit gates on pairs `0,1`, `1,2` and `0,2`. It is not possible to map this to the given topology because it requires this kind of loop:

```
QB0 ——— QB1
 |       |
  —— QB2 ——
```

Thankfully, there is a way around it: swap the state between qubits whenever needed. In other words, use one of the unused physical qubits to store a state. The following sequence of diagrams shows the steps involved. Physical qubits are denoted with `QB`, the corresponding mapped logical qubits are in brackets `()`, and * denotes the location of the currently discussed operation.

Step 1: pick a connected pair to perform `cx(0,1)`:

```
          QB0(0)*
           |
QB1( ) – QB2(1)* – QB3( )
           |
          QB4( )
```

Step 2: pick a qubit `QB4` connected to `QB2` to perform `cx(1,2)`:

```
          QB0(0)
           |
QB1( ) – QB2(1)* – QB3( )
           |
          QB4(2)*
```

Step 3: Now we need to perform `cx(0,2)`, and those states currently reside in `QB0` and `QB4` which aren't connected. We keep `QB0` as is, but move the state out of `QB2` to `QB3`:

```
          QB0(0)
           |
QB1( ) – QB2( ) – QB3(1)*
           |
          QB4(2)
```

Step 4: then move the state of `QB4` into `QB2`; now we can perform `cx(0,2)`:

```
          QB0(0)*
           |
QB1( ) — QB2(2)* — QB3(1)
           |
          QB4( )
```

In the end the mapping is as follows:

- logical `qubit 0` to physical `QB0` (started there and never moved)
- logical `qubit 1` to physical `QB3` (started at `QB2` and moved to `QB3`)
- logical `qubit 2` to physical `QB2` (started at `QB4` and moved to `QB2`)

These temporary swaps are expressed in form of `SWAP` gates. The router algorithm has to modify the circuit and insert those swap gates. It would look like this:

```
circuit.cx(QB0,QB2)
circuit.cx(QB2,QB4)
circuit.swap(QB2,QB3) # inserted swap
circuit.swap(QB4,QB2) # inserted swap
circuit.cx(QB0,QB2)
```

SWAPs are not free. They take some time, which is very limited. We only have a few hundred microseconds of coherence time at our disposal (with superconducting QCs), so every non-essential operation is basically wasted time. SWAPs are also not always perfect, so in general we want to minimize them.

Optimization

A quantum circuit is not just a list of operations; it's a program that a transpiler will often try to optimize, by default. The transpiler's job is to rewrite your circuit to run as efficiently as possible on the target quantum hardware, but most SDKs allow you to control this behavior. For example, in Qiskit you can select one of multiple optimization levels, `0` being no optimizations at all.

A common optimization is gate cancellation. If you apply a Hadamard (H) gate twice in a row, you've done essentially nothing (recall the rotations involved in the Hadamard implementation, applying them twice returns the vector of the Bloch sphere to its original position). The transpiler is smart

enough to see this and will just delete both gates. This is a somewhat silly example: why would you put two Hadamard gates in a sequence? One reason is that we just want to provide a very simple but illustrative example.

Another reason is that this kind of double operation can be part of a legitimate benchmarking program. A well-calibration quantum computer would execute two Hadamard gates and return the qubit to its initial state, and by doing this in a row multiple times and repeatedly measuring the final state one can make conclusions about the quality of calibration. Other trivial examples of optimization are combining multiple rotations into one operation, cancelling out rotations, etc.

Sometimes a transpiler may even replace some already native gates with other gate(s) if this is deemed more optimal, for example, because of a reduction in length (in time). In some cases, you'd want to keep optimizations enabled, but control a specific cancellation at a specific location. A tool that can help here is a so-called Barrier gate.

Barrier

Most quantum circuit SDKs and interfaces include a "barrier" gate. Unlike other gates, barrier does not represent a mathematical operation or any direct manipulation of the qubit. Barrier is not truly a gate in this sense, but rather an instruction for the scheduler that allows the user to separate operations explicitly.

Imagine you have two threads in a classical (not quantum) program. You need thread A to finish writing to a variable before thread B reads it. If the compiler reorders your instructions for "efficiency," you could get a race condition. To prevent this, you use tools like mutexes or semaphores. These tools essentially tell the compiler: "Do not reorder operations across this point. All operations before this fence must complete before any operations after it begin."

As discussed earlier, one of the trivial transpiler optimizations was removing two consecutive Hadamard gates because they cancel out. If your goal is to actually execute multiple Hadamard gates just like you wrote them in the original circuit, then you have to disable the optimizations. But if you want to keep the optimizations enabled for other parts of the circuit, you can use the Barrier gate to instruct the transpiler not to perform any operations across a certain line.

So, to preserve the two H gates in this circuit:

```
circuit.h(0)
circuit.h(0)
```

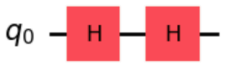


Figure 20. Two Hadamard gates placed in sequence on the same qubit

We would need to put a barrier gate between them like so:

```
circuit.barrier([0, 1])
```

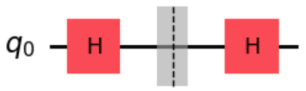


Figure 21. A barrier placed between two Hadamard gates

Calibration-aware transpilation and routing

We've discussed transpilation with the assumption that all qubits and their connections are equivalent. When choosing which physical qubits to map onto, and which SWAPs to introduce, if needed, we haven't considered the fact that not all physical qubits are created equal. Due to fabrication imperfections, the quality of qubits vary. This results in different coherence times, and different error rates for specific operations. In addition, the connections between qubits may differ from one another, which results in different 2-qubit gate fidelities.

Moreover, these differences are not static. Yes, many of them originate from the fabrication process, but the actual values depend on the calibration. We'll discuss calibration more in Chapter 11, but for now it's important to understand that the overall condition of the QPU is dynamic, changing from day to day. Any combination of the following parameters can change pretty much at any moment:

- the amount of qubits (e.g. a qubit may practically "die" or its quality may degrade)
- the connections between qubits (e.g. a component that facilitates the connection may "die" or its quality may degrade)
- coherence time of any qubit (i.e. how long the qubit can preserve quantum state)

The program that performs the computation must also remember this mapping in order to report the measurement results correctly, so that the values can be mapped back to the original logical qubits. But I'm afraid we're not ready to go into processing the results, because at this point all we have is an abstract circuit represented as static data. Sure, it is now transpiled to the native gate set and routed to the correct topology, but what next? How do these text symbols translate into actual quantum hardware? Let's cover one more aspect of gates before diving into pulse representation.

Chapter 8. Mid-circuit measurement

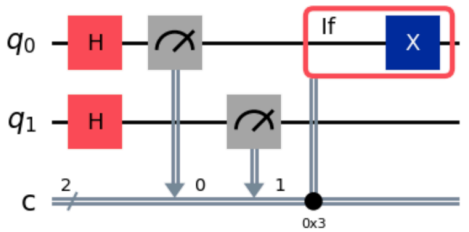
Clasically-controled gates

Recall the `cx` – a "quantum CNOT" gate. It looked like an `if` statement, but wasn't really that. Instead, it was a quantum entanglement between states of two qubits.

In contrast to quantum `cx`, there is an actual `if`-like operation in quantum circuits, but not all hardware currently supports it fully. The idea is simple: measure the state of one or more qubits, and if the result is `1` then apply some gate on one or more other qubits, otherwise do nothing. This requires an aptly called "mid-circuit measurement": measuring the state not just at the end, but in the middle of the circuit, and continuing with other operations after the measurement.

In code it may look like this:

```
qr = QuantumRegister(2, "q")
cr = ClassicalRegister(2, "c")
qc = QuantumCircuit(qr, cr)
qc.h(0)
qc.h(1)
qc.measure(qr, cr)

# Apply an X to qubit 0 if both measurements were 1.
with qc.if_test((cr, 3)):
    qc.x(0)
```



There is one aspect that becomes more important for mid-circuit measurements: Quantum non-demolition (QND) measurement. If you read some introductory literature about quantum computing and the properties of collapsing

wave functions, you'll see an example repeated all the time: after measuring a qubit, all subsequent measurements yield the same result, because the wave function had collapsed, and there is no uncertainty anymore. In theory this is correct, but in practice the process of measuring may be destructive. Yes, it would read and report the correctly observed value, but it does not necessarily guarantee that the qubit stays in that state afterwards. In my experience, as of 2024-2025, not all commercially available quantum computers guarantee QND by default. This often comes down to a specific implementation and calibration of a measurement operation.

The measurement is usually not some hardcoded action of the instruments, but instead it can be considered another operation for which calibration is required. A simple measurement operation that does not guarantee quantum non-demolition (or quantum non-destructiveness) is easier to calibrate for.

QND is important for mid-circuit measurements because it may be needed to apply the `if` operation multiple times from the same source qubit.

Fast feedback

Fast Feedback (or real-time feedback) is the ability to perform a mid-circuit measurement, communicate the classical result to a classical processing unit, have that unit perform a classical computation, and use the result of that computation to conditionally trigger a subsequent quantum gate, all within a timeframe significantly shorter than the qubit's coherence time.

Any real-time feedback loop must complete its entire cycle well within this coherence window. Let's analyze the steps in this "latency budget":

1. Measurement (Readout): a microwave pulse is sent to the readout resonator coupled to the qubit. The reflected pulse is captured. This whole operation is not instantaneous.
2. Signal Propagation: The faint analog signal travels from the chip up a chain of amplifiers and coaxial cables to the room-temperature control electronics. This can take 50-100 ns.
3. Classical processing out: the analog signal is digitized. A classical processor must then execute the conditional logic. If this is a simple if (bit == 1) check like with `if_test` example from the previous section, then it's fast. But for more complex operations (especially ones relevant for correcting errors) the process takes longer. This logic must be deterministic and execute in nanoseconds.

4. Signal propagation in: the command for the next gate travels from the room-temperature electronics back down the control lines, through attenuators, to the target qubit. This again takes 50-100 ns.
5. Gate Application: the microwave pulse for the quantum gate itself must be applied, taking another 10-50 ns.

A typical "fast feedback" round-trip time for current superconducting systems might be on the order of hundreds of nanoseconds to microseconds. Best superconducting quantum computers can achieve hundreds of microseconds, and in some isolated (not yet practical) cases certain physical qubits have been demonstrated to stay "alive" for almost a millisecond[2].

A general-purpose CPU running a high-level operating system (like Windows, macOS, or Linux) is non-deterministic. It contends with thread schedulers, interrupts, and I/O, making it impossible to guarantee a 50 ns response time. For fast feedback, more precise control and predictability are required. In some commercially available electronic devices regular CPUs (especially highly efficient ARM chips) are used. Exact nanosecond-precision guarantees are still not available, but within a budget of few CPU ticks people have managed to use the for fast feedback.

It is the opinion of many specialists in the industry that the correct technology for fast feedback is the Field-Programmable Gate Array (FPGA). An FPGA is a chip containing a set of reconfigurable logic blocks and memory. As a software engineer, you can think of it as a piece of hardware that you can program to become a specific digital circuit. You don't "run" code on it in the traditional sense; you synthesize your logic (written in a hardware description language like VHDL or Verilog) into a physical configuration of gates. The fast feedback loop, implemented in hardware, looks like this:

1. Measurement.
2. The analog signal travels to the control rack.
3. The signal is digitized by an ADC, then the digital stream is fed directly into an FPGA. The FPGA performs demodulation and thresholding to get a classical bit. This bit is routed immediately (within the same clock cycle) into a pre-loaded logic block on the same FPGA. This logic block executes the operation. The output code is sent to an the waveform generator

[2]New quantum record: Transmon qubit coherence reaches millisecond threshold. https://www.aalto.fi/en/news/new-quantum-record-transmon-qubit-coherence-reaches-millisecond-threshold

or another device that's responsible for sending signals back to the chip. Ideally, this device is also controlled by the FPGA.
4. The signal travels back into the chip.
5. Gate is applied.

Another simple example of fast feedback is active reset: changing the state of the qubit to `0` by first measuring it, and applying the necessary pulse if the state is not `0` already.

Chapter 9. Compilation to pulse representation

At this stage we have a circuit that is perfectly suited for the hardware at hand. All non-native gates are decomposed into equivalent native ones, logical qubits are mapped to physical qubits correctly, and `SWAP`s are introduced where needed. We are now ready to finally convert these mathematical operations into analog signals.

Each gate will end up being one or more microwave pulses of precise shape(s). The calibration data dictates those shapes (more on calibration in chapter 11). We are not going to provide precise mathematical mappings from high-level concepts like `Hadamard` or `CNOT` into rotations and corresponding pulses; this alone is worthy of a book of its own. But by now you should have a conceptual idea along these lines:

1. A qubit's state is not a physical arrow and a sphere, but its mathematical view can be represented with a Bloch sphere.
2. A gate is an operation that changes the state, i.e. rotates the vector (arrow) in a precise manner; a gate may be a single rotation, or a series of rotations.
3. A physical qubit in a superconducting QPU reacts to microwaves.
4. We need a way to find out which kinds of microwaves to generate in order to change the state in line with the math.

An example of a single pulse that may correspond to a realistic gate (e.g. Hadamard) would look like this in data form:

```
{
 wave_i: {
  n_samples: 80,
  full_width: 1.5,
  center_offset: 0
 },
 wave_q: {
  n_samples: 80,
  full_width: 1.5,
  center_offset: 0
 },
 scale_i: 0.088,
 scale_q: -0.005,
 phase: -1.571,
```

```
 modulation_frequency: 0,
 phase_increment: 0,
 duration: 40
}
```

Note the duration: this number is typically in nanoseconds. A single gate can be applied within a few nanoseconds!

Visualized, it may look like this:

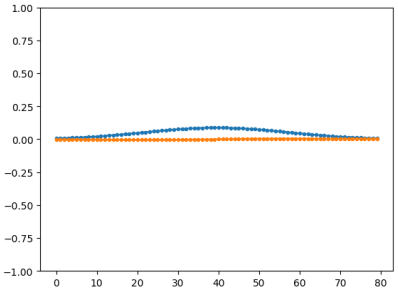


Figure 22. IQ pulse

A two-qubit gate would inevitably involve sending a pulse to a coupler: the component connecting the two qubits. Here is an example of the pulse that corresponds to a `CNOT` gate:

```
{
 wave: {
  n_samples: 96,
  full_width: 0.843,
  rise_time: 0.104,
  center_offset: 0
 },
 scale: 0.09,
 duration: 48
}
```

This pulse has a different shape, and is ultimately a constant application for a "long" period of time, with a sharp rise and fall. It's quite fascinating that sending this simple microwave pulse for 48 nanoseconds (in our example) somehow entangles the quantum states of two qubits it connects.

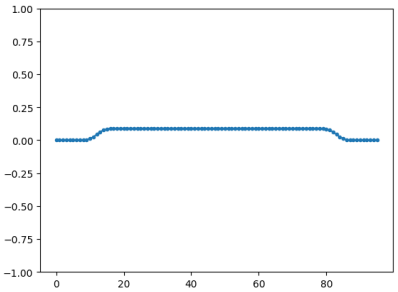


Figure 23. Real pulse

Our example circuit involved 3 qubits, which means there are at least 5 components on the physical chip that require manipulation: the qubits themselves and the connectors (couplers) between them. Qubits are manipulated via so-called drive channels; essentially, wires going from the instruments to particular qubits. Couples have their corresponding channels. We also need to measure the state of all three qubits in the end, and this may require sending a complex signal on one or more readout channels.

The entire set of pulses grouped by those channels can be visualized like so:

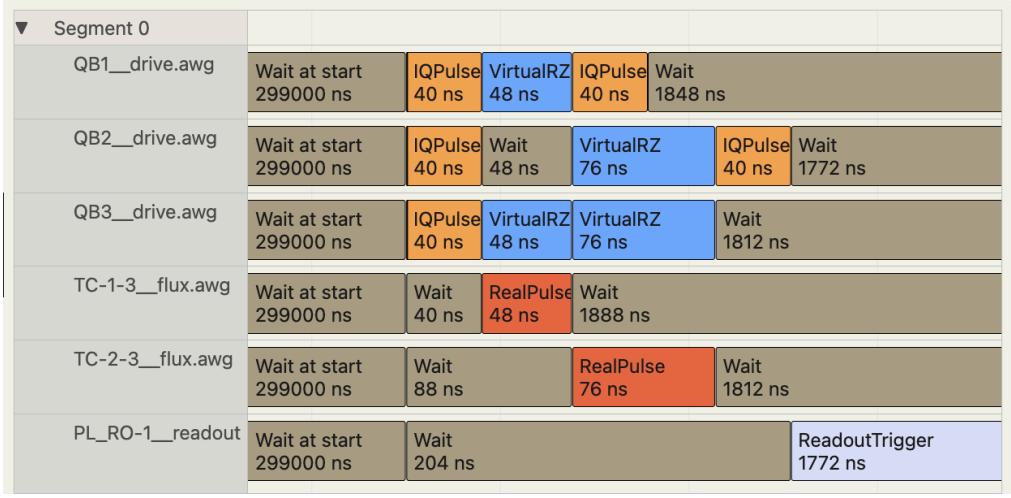


Figure 24. IQM's pulse playlist visualizer

Compilation to instrument instructions

This is the final stage of the "lowering" from abstract circuits into concrete actions on the physical device. The pulse representation that was generated

in the previous section is still only data. It's very similar to an audio file. It describes the shapes and timing of sound waves, but does not represent the actual instructions for the speaker membrane on how to vibrate in order to generate such audio waves.

In regular computers and digital audio systems there must be a device called DAC: digital-to-analog converter; it converts digital audio data into an analog audio signal that speakers or headphones can play. We need something equivalent in order to convert the pulse data into specific instructions for the control instruments.

There are companies that specialize in producing electronics that is suitable for operating a quantum computer. Some QC vendors also produce their own instruments. A typical commercial quantum computer you can find deployed in universities or research centers often contains a combination of various instruments, sometimes from different manufacturers. The instruments involved are AWGs (Arbitrary Waveform Generators), signal generators, amplifiers, and almost always include some sort of central control unit.

Because of extreme timing precision requirements, it's very difficult to synchronize and orchestrate the exact series of pulses in time when there are multiple instruments involved. A central control unit usually takes care of uploading data into corresponding devices and then acting as the dispatcher, triggering the start of the "firing" of pulses. In order to make signals synchronized on a nanosecond level, the central unit has to be aware of the length of each cable connecting it to other instruments, because in one nanosecond, light travels approximately 30 centimeters, which is slightly less than one foot. This is often simplified to "a foot per nanosecond" in discussions about computing.

The way to convert pulse data into instrument instructions is dictated by the instruments, or rather the software that comes with the instruments. Often, they are given in form of some high-level programming library acting like a driver. It provides an interface and as output generates the payload to be uploaded into the instruments. Some instruments also come with a web-based service software that allows to upload, control, and visualize payloads and current status of the instruments.

This is another huge area of development. Control instruments have to evolve significantly in order to support future needs of larger quantum computers. At the very least, they need more memory and very high speed

buses between memory and actual signal triggers.

Chapter 10. Pulse-level control

In the previous chapters, we've treated quantum gates as the fundamental building blocks of our programs. We've seen how a high-level quantum circuit is transpiled into an "ideal" circuit for a specific hardware topology, and then compiled into a time-ordered sequence of analog pulses. For most application developers, this level of abstraction is perfect. You write qc.cx(0, 1), and the compiler and control electronics conspire to execute the best-calibrated CNOT gate the machine has.

But what if that compiled CNOT isn't good enough? What if you want to invent a new gate? Or what if your goal isn't to run an algorithm, but to measure the precise lifetime of your qubit?

For this we need to access a lower level, a bit like a "systems programming" layer of the quantum stack. This is referred to as pulse-level access or control: the features allowing to bypass the high-level gate abstraction and directly command the hardware to produce specific microwave pulses at specific times. This is the layer where quantum physics and electrical engineering meet software, and it's where the most advanced research and performance-tuning happens.

Between 2023 and 2025 I have been involved in the development and maintenance of IQM Pulla (**pul**se **l**eve**l** **a**ccess), so I'm going to use it as an example. It's open source and has very simple and clean model of various abstraction levels, and I'm most familiar with this product.

Note that many quantum hardware vendors do not offer pulse-level access altogether, and some, like IBM, are starting to wind down the support for such feature set.

* * *

For a software engineer, working at the gate level is like writing in Python or Java. You have a powerful, expressive, high-level language. Working at the pulse level is like dropping down to assembly or even writing GPU shader code. It's more complex and hardware-specific, but it gives you ultimate control and performance.

This control is essential for several key tasks that are critical to advancing the field. The list below is loosely ordered by complexity, starting from conceptually and technically simple tasks and going all the way to "hardcore" mode of experimental access.

- Peeking into the compiled form.
- Tweaking the calibration.
- Defining custom implementations for native gates.
- Defining custom new gates.
- Hardware Characterization and Benchmarking: How do you even know what a gate is? How do you find a qubit's exact frequency? You can't do this with a CNOT. You do it by performing a spectroscopy experiment: sweeping a low-power pulse across a range of frequencies and seeing where the qubit "wakes up." This is a fundamental pulse-level task. Similarly, benchmarking gate fidelity (e.g., with Randomized Benchmarking) often requires fine-grained control over the exact pulses being sent.
- Gate Calibration: The "default" X gate on a quantum computer is just a pulse that was calibrated to work well. This calibration drifts over time as the hardware's environment changes. Researchers constantly run calibration routines—like the Rabi experiment we'll see later—to find the exact pulse amplitude and duration needed to perform a perfect π-pulse (an X gate).

What is a pulse?

So, what is a pulse in this context? As discussed earlier, we control a superconducting qubit by sending microwave signals to it. An arbitrary pulse is defined by a few key parameters:

1. Duration: How long the pulse lasts, typically in nanoseconds (e.g., 20 ns).
2. Carrier Frequency: The "base" frequency of the microwave. This is set to be resonant with the qubit's transition frequency (e.g., 5.0 GHz).
3. Amplitude: The "strength" or "height" of the pulse. This (along with duration) controls the angle of rotation on the Bloch sphere.
4. Phase: The relative phase of the carrier wave. This controls the axis of rotation (e.g., a 0° phase might give an X-gate, while a 90° phase gives a Y-gate).

5. Envelope (Shape): This is the most critical part for advanced control. Instead of just switching the microwave on and off (a "square pulse"), we smoothly "ramp" the amplitude up and down. This shape is called the envelope. A common envelope is a simple Gaussian shape, but complex, optimized shapes (like DRAG) are also used.

Digital control electronics use a standard radio-frequency technique called IQ modulation. A "baseband" signal is generated, called I (In-phase) and Q (Quadrature). In a pulse-level programming environment, a "pulse" is ultimately defined as two arrays of numbers: the I samples and the Q samples.

When you drop to the pulse level, you are no longer building a QuantumCircuit. Instead, you are building a sequence of pulses. In IQM's lingo it's called a `Schedule`. Think of a Schedule as a timeline or a music sequencer. You don't just say "do this, then that." You say "at time `t=0ns`, do pulse_A on channel_1" and "at time `t=30ns`, do pulse_B on channel_2."

In other words, you get partial or complete control over the data on the pulse level – the data we have briefly discussed and visualized in the previous chapter. One suitable analogy is this: compiling C code into assembly, and then inspecting and manually editing the assembly code.

Chapter 11. Calibration

As a software engineers, we are accustomed to deterministic, digital systems. When you execute a function `add(a, b)`, you expect it to return the same, correct result every time for the given values of `a`, `b`. The underlying hardware (transistors, logic gates, etc.) is abstracted away so completely that it is, for all practical purposes, a perfect, logical machine.

However, said underlying hardware is not actually perfect. Chips and memory aren't 100% deterministic, bits flip randomly sometimes, the communication between components is ever so slightly imperfect. Data in RAM may be corrupted at any moment. But these events are rare and limited enough so that certain error-correcting techniques make it feel like the hardware is perfect. This error correction was much more visible in the early days of computing.

Quantum computers, particularly those built from superconducting circuits, are all still in those early days, but the story is much, much worse. A QPU is a complex, analog physics experiment. We've seen how quantum circuits, which assume the idealized notion of a quantum computing device, are converted into analog signals that attempt to manipulate the physical qubits in precise manner. The exact way the final signals are defined depends on the current conditions across the combination of all components on a quantum chip. These conditions aren't stable, they change over time and also due to miniscule changes in temperature and other physical properties.

This is why a quantum computer must be calibrated. Not just once, but continuously. The information obtained by the calibration process (often simply called "calibration data") is stored, usually in a database or object storage of some sort, and then utilized during the compilation.

Simply put, if a gate `A` calls for a rotation of `N` degrees around some axis, and we generally know that this requires sending a known pulse to the qubit, then the precise time and amplitude of the pulse depends on the calibration data. The better the calibration data, the more precise pulse shape will be computed, which will result in more accurate rotation (or any other manipulation).

In practice, there are two distinct types of calibration in superconducting QCs:

1. Initial calibration.
2. Recalibration.

Initial calibration

In an ideal world, we would build a QPU and characterize it once. Characterization is essentially looking at the chip with precise scientific devices and determining its precise properties. If we could rely on the initial characterization and simply hard-code the observed control parameters, then we wouldn't need calibration at all.

There are three big problems that make it impossible to avoid calibration today:

1. **Manufacturing inhomogeneity**. Even though those properties were known in the design long before the fabrication, due to the imperfection of the fabrication process those values are just targets. Each modern QPU is kind of a snowflake: close to the target design, but not exactly matching. Manufacturing classical CPUs and GPUs is also a lottery, and the quality of silicon varies even across a single wafer; but with quantum it's a lot worse. Each qubit has a unique resonant frequency, energy-level anharmonicity (the difference between ground state and first excited state), and coupling strength to its neighbors and readout resonator. Therefore, the pulse required to perform e.g. a `NOT` gate on `Q1` is different from the pulse for `Q1`.
2. **Environmental sensitivity and temporal drift**. Superconducting qubits are exquisitely sensitive to their environment, which is what makes them good sensors, but difficult to control. Their properties drift over time due to: temperature fluctuations (millikelvin-level changes in the dilution refrigerator), stray electromagnetic fields (noise from the control electronics or even external sources), random state changes or flips due to inevitable defects in the substrate materials.
3. **Cross-talk**. Qubits are not isolated. Applying a pulse to `Q1` can electromagnetically "leak" to `Q2`, causing a small, unwanted operation. This is known as cross-talk.

So, once the chip is fabricated and characterized, we need to determine exact properties that will allow to generate correct pulses for different operations on different locations on the QPU. The steps performed are often referred to as "experiments". Calibration is not a single experiment but a "stack" of sequential procedures, where each step relies on the parameters found in the previous one. This process typically runs automatically, managed by a classical control

system, and can take hours to complete for a large QPU. Some basic types of experiments relevant during initial calibration:

Finding the qubits and resonators (Spectroscopy)

The goal is to locate the resonant frequencies of the readout resonator and the qubit. For a simple analogy, think of a wine glass. If you make a noise in a specific frequency, the glass will start vibrating. This means you have found the resonant frequency of the glass and can affect its state by making that sound. We want to find such frequencies for each qubit and each resonator.

The corresponding experiment is usually called spectroscopy. The idea is to generate a microwave signal sweeping across a range of frequencies until the qubit or the resonator "responds" strongest. The number found this way is stored as part of the calibration data. Superconducting qubits generally have frequencies within the range of 4 to 8 GHz[3].

Single-qubit gate parameters (Rabi & Ramsey)

The goal is to determine the pulse parameters to perform `NOT` gate (at least). The experiment is called Rabi Oscillation. The qubit first has to be put into the ground state `0`, therefore we must have completed the spectroscopy part first. From here, a drive pulse is applied with varying duration or amplitude. The probability of measuring $|1\rangle$ is plotted against the varied parameter. The resulting plot shows a sinusoidal oscillation (a "Rabi oscillation"). Certain pairs of amplitude and duration are derived from the results and stored as part of the calibration data.

Single-qubit pulses, two-qubit tuning, and more

Next we build up on the initial results and find next pieces of the puzzle, such as:

- Single-qubit errors, i.e. properties of a special operation that compensates for occasional leakage from state `1` into higher states.
- Two-qubit tuning for gates such as `CNOT` and `CZ`. This is often the most complex and time-consuming part, because the process involves multiple parameters.

[3]Why 4-8 GHz? The rationale behind common qubit frequencies. https://quantumobserver.substack.com/p/why-4-8-ghz

- Measurement. There may be multiple ways to measure the state of a qubit, and each one is considered an operation with its own set of calibrated values.
- Active reset. One easy way to put a qubit into the ground state is to wait long enough (not long until the state decoheres completely, but long enough within the miniscule timeframe of the coherence time; recall the pulse visualization from chapter 9 – it included a 299000 nanosecond wait time at start of each qubit and coupler). An active reset is a special operation that "actively" changes the state to `0` on demand by first measuring the state and then applying a pulse that compensates and achieves the ground state; this is important because it's generally much faster than waiting, so we can save a lot of time at the beginning of a job and use that saved up time to perform more operations in the circuit. Understandably, active reset requires at least efficient measurement to be available.

Normally, the process would also include measuring some values along the way, such as T1 (simplifying: coherence or "staying alive" time of a component). Once initial calibration is complete, we end up with a data structure like this:

```
{
  "q0": {
    "f_01_GHz": 5.12345,
    "f_readout_GHz": 6.8761,
    "T1_us": 120.5,
    "T2_ramsey_us": 90.2,
    "pi_pulse_amplitude_V": 0.456,
    "pi_pulse_duration_ns": 20,
    "pi_half_pulse_amplitude_V": 0.228,
    "drag_alpha": 0.881
  },
  "q1": {
    "f_01_GHz": 5.09876,
    "f_readout_GHz": 6.9012,
    "T1_us": 105.1,
    /* ... etc. ... */
  },
  "q0_q1_coupler": {
    "cz_pulse_shape_I": [0.01, 0.03, ..., 0.02],
    "cz_pulse_shape_Q": [0.00, 0.01, ..., 0.00],
    "cz_pulse_duration_ns": 40
  }
}
```

This kind of data is the source of truth for what kinds of gates are supported at which qubits or pairs. In other words, if you don't calibrate e.g. `CNOT` between qubits 4 and 5, then the transpiler will disallow such operation and will either reject your circuit, or re-route the circuit accordingly (if possible), or introduce additional `SWAP`s.

Initial calibration on a modern superconducting QPU with dozens or few hundred qubits may take many hours.

* * *

(I had noticed a bit of inconsistency in the lingo between different people and companies when it comes to calibration. Some people would say "I need access to calibration data", but they mean fidelity or quality metric data, i.e. they want those metrics that describe how good the calibration data is. Other people actually want calibration data: those numbers that define the pulse shapes, amplitudes, etc. And some people assume that a single "calibration data" structure would include both.)

Recalibration

The initial calibration set is good, but the properties of the chip keep changing over time and due to changing environment. Occasional recalibration is needed. It's much faster than initial calibration. Think of it as fine-tuning the mostly well tuned system.

In theory, recalibration can take part on a subset of the QPU while another subset is busy executing some user's circuit. This possibility depends on the hardware and software of the machine.

Evaluating calibration quality: fidelity

A calibration is only as good as the gate quality it produces. The ultimate metric for quality is fidelity: What is the probability that the physical operation performed by the hardware is identical to the ideal, logical operation defined by theory?

This is often measured using Randomized Benchmarking (RB). The procedure is roughly as follows:

1. A random sequence of logical gates is generated. Gates that are easy to simulate classically are preferred.
2. The classical computer calculates what the net effect of this sequence is. It then appends one final "inverse" gate that should return the qubit to its initial state if all gates were perfect.
3. The full sequence (random sequence + inverse gate) is compiled using the current calibration set and run on the actual QPU. The qubit is measured.
4. The process is repeated hundreds of times with different random sequences of the same length. This gives a "survival probability" or the probability of successfully returning to $|0\rangle$.
5. The entire process is repeated for longer sequences (e.g., 10, 20, 100, 200, etc.).

As the sequence length increases, the probability of an error accumulating increases. A diagram is plotted that usually shows exponential decay. The rate of decay of this curve is directly related to the average fidelity of the gates. The result is a single number: the average gate fidelity (e.g., 99.92%). This is the "pass/fail" grade for the calibration. Interleaved RB is a variation used to find the fidelity of one specific gate (like the `CNOT`).

As you can imagine, this takes a long time. It's one of the heaviest things you can do, basically stress-testing everything including the transpiler (as it handles longer and longer circuits), compiler, pulse scheduler, client-server interaction, data transfer, job scheduler, control electronics, and, ultimately, the QPU.

Part III. Industry landscape

Chapter 12. Software ecosystems

This part is an overview of existing, popular programming frameworks and libraries that allow us to define and manipulate quantum circuits, send them for execution, and retrieve and process the measurement results. The artificial syntax we've used before (e.g. code like `circuit.h(0)`) is a common structure among popular Python-based frameworks, as you are about to see.

SDKs and formats (Qiskit, Cirq, CUDA, QASM, QIR, etc.)

Just as you wouldn't program a classical computer by manually flipping transistors, you don't program a quantum computer by directly manipulating microwave pulses. The raw complexity of the underlying physics and hardware needs to be abstracted away. This is the role of quantum software development kits (SDKs) and standardized data formats. They provide a bridge between the theoretical world of quantum algorithms and the practical, noisy reality of quantum hardware. The primary goal of a quantum framework is to let researchers and developers focus on the logic of their quantum algorithms, not the peculiarities of a specific quantum chip. In practice, many of the popular packages allow you to define quantum circuits, manipulate them, and ultimately execute them on either a simulator or a real quantum processing unit (QPU).

You might notice that the vast majority of these frameworks are based on Python. This isn't a coincidence. Quantum computation, especially in the current era, is an experimental science. The classical code that builds, submits, and analyzes the results of quantum circuits does not need to be exceptionally performant. Python, with its simple syntax and vast ecosystem of scientific libraries like NumPy and Matplotlib, is the perfect language for this kind of orchestration. It allows for rapid prototyping and easy integration with the powerful classical pre- and post-processing that most hybrid quantum algorithms require.

However, this is just the surface. Many popular scientific Python packages use low-level, more performant languages under the hood to do the actual heavy lifting. For example, roughly third of Numpy (a very popular scientific package) is written in C, with a little sparkle of C++ on top. Some parts of Qiskit – a flagship quantum computing framework developed and maintained by IBM – are being written in Rust. This doesn't mean that users (scientists and researchers) have to suddenly learn C or Rust, but rather that you as an aspiring quantum software engineer will likely have to, at some point, deal with the complexities of inter-operability between Python and lower-level languages. As you might have guessed, the reason behind C- or Rust-based parts is performance. Multiplying huge matrices is just not fast enough in Python. Another reason is better compatibility with GPUs (graphics processing unit), which often requires C-compatible bindings.

A mature quantum framework typically offers at least these features:

- Circuit Definition: An intuitive interface to create quantum circuits, add qubits and classical bits, and apply quantum gates from a standard library.
- Circuit Manipulation: Tools to combine smaller circuits into larger ones, inspect their properties (like depth and gate count), and modify them programmatically.
- Transpilation and Optimization: The process of converting an abstract quantum circuit into a new one that can actually run on specific hardware. This involves replacing the gates with the hardware's native gate set and optimizing the circuit to reduce its depth and gate count, minimizing errors.
- Routing: A crucial step that maps the abstract "logical" qubits from the circuit onto the physical qubits of a QPU, taking into account the chip's limited connectivity.
- Execution: A unified interface to send the prepared circuit to different backends, whether they are local simulators, powerful cloud-based simulators, or real quantum hardware from various vendors.
- Result Handling: Tools to retrieve and interpret the measurement outcomes from an experiment, typically providing counts, probabilities, and visualization tools.

We will explore transpilation and routing in detail in the next section, as they are complex and critical steps in making a quantum program executable.

A more comprehensive framework might also offer things like:

- Pulse-level access: A way to break through an abstraction barrier of quantum circuits and at least see, but ideally also modify the pulse representation that is generated from the original quantum circuits.
- Control calibration: Access calibration data obtained from the vendor (usually this means a remote, cloud-based quantum computer) to at least see, but ideally also modify the parameters that define the implementations of native gates for a given QPU.
- Define custom gates. This may be a simple case of defining composite gates from multiple basic gates; or a much more complex task of defining completely new gates from scratch, which usually requires pulse-level access and access to calibration.

Qiskit from IBM

When it comes to popularity and breadth, it's hard to overstate the influence of IBM's Qiskit. It is a comprehensive open-source framework that provides tools for almost every level of the quantum computing stack. Its rich set of features, extensive documentation, and active community have made it the de facto standard for many newcomers and researchers. Qiskit is structured into modular components, including Terra (the core for circuit creation and transpilation), Aer (for high-performance simulations), and modules for studying applications in areas like chemistry and optimization.

Qiskit also used to include pulse-level access features, but in the newer versions IBM had started to wind down that part of the stack. One way to explain this move is to look at IBM's overall strategy and business model when it comes to quantum. Unlike many smaller companies in the industry, IBM does not sell their quantum computer hardware. You cannot simply purchase a machine and install it in your data center. So, ultimately, the scope of what you can do on their hardware is controlled by IBM, and it may not be in their best interest to allow everyone to have a very low-level access to this technology. But perhaps a more fair explanation is that IBM's ambitious roadmap targets high-level goals of functional error correction and fault-tolerant quantum computers. The company is already "in the future", where the notion of controlling individual pulses is as niche and unnecessary for 99.99% of cases as it is in classical computing of 21st century. Just like nobody expects to program voltages on the motherboard to do useful programming today, IBM and other major players don't expect users of their software libraries to deal with pulse level control and calibration.

This makes sense, but the reality on the ground today is that the vast majority of algorithm researchers, scientists, software integrators and admins of quantum computing service providers demand low-level access of some sort. Sometimes they require simple read-only data access to the current state of the calibration and some quality metrics for particular gate implementations (which can be viewed as pulse-level access because those metrics are obtained directly from the concrete numerical parameters of gates, like waveforms, durations of pulses, or even voltages); in other cases, researchers need to define custom gate implementations or brand new gates on the fly, dynamically, and control the way the execution is done. A relatively thin slice of science can be done in this area while being restricted to pure quantum circuits only.

This being said, we will still use Qiskit throughout this book as a language for defining circuits and running quantum jobs. Here is an example of a Qiskit program:

```
# Define a circuit for 5 qubits
qc = QuantumCircuit(5)

qc.h(0) # Hadamard gate on qubit 0

for qb in range(1, 5):
    qc.cx(0, qb)  # cx gates between qubit 0 and each other gate

qc.barrier()      # barrier gate on all qubits
qc.measure_all()  # measurement operation on all qubits

# Transpile the circuit
qc_transpiled = transpile(qc, backend)
```

Cirq, Pytket, and others

While Qiskit is dominant, the ecosystem is vibrant with powerful alternatives. Some notable tools include:

Cirq: Developed by Google, Cirq is an open-source framework designed with the Noisy Intermediate-Scale Quantum (NISQ) era in mind. It places a strong emphasis on giving users fine-grained control over circuit construction and optimization to extract the maximum performance from today's noisy hardware. Some people find Cirq's structure and abstractions to be cleaner and simpler to understand compared to Qiskit.

```
q0, q1 = cirq.LineQubit.range(2)
qc = cirq.Circuit(cirq.H(q0), cirq.CX(q0, q1), cirq.measure(q0, q1, key='b'))
```

Pytket is a hardware-agnostic quantum SDK developed by Quantinuum that aims to be a powerful compiler. The focus is on circuit optimization and retargeting capabilities, allowing users to write a circuit once and then efficiently compile it to run on a wide variety of different quantum backends, from IBM to Google to Quantinuum's own trapped-ion machines. The syntax looks very similar to Qiskit:

```
qc = Circuit(2, 2)
qc.H(0)
qc.CX(0, 1)
qc.measure_all()
```

Microsoft Q# (pronounced "q-sharp"), a high-level, open-source programming language developed by Microsoft for writing quantum programs. Q# is included in the Quantum Development Kit (QDK), which also contains simulators and debugging tools.Example of code in Q#:

```
operation BellPair(qb1 : Qubit, qb2 : Qubit) : Unit
{
    H(qb1);
    CNOT(qb1, qb2);
}
```

PennyLane is designed with a big of a higher level of abstraction in mind and is geared towards applications in machine learning. But one can still write direct gates using a familiar syntax:

```
qml.Hadamard(wires=0)
qml.CNOT(wires=[0, 1])
return qml.probs(wires=[0, 1])
```

AWS Braket is Amazon's quantum computing cloud offering, which includes a python-based SDK for creating and submitting quantum programs. Braket acts like a proxy towards multiple vendors, so it's an abstraction that allows you to run the same algorithm on multiple different quantum computers without rewriting the code. Its Python implementation features nice chaining that is familiar to some software engineers:

```
qc = Circuit().h(0).cnot(control=0, target=1)
```

QASM

I don't think there was ever a successful "common data format" in the history of computing. There are always competing formats and vendors pursuing their own goals. But there are always attempts to define a common format

or a common language, in any area. In the quantum computing this attempt is QASM (Quantum Assembly Language). It is a simple, human-readable text format that describes a sequence of quantum operations. It has become an acceptable standard for representing circuits, allowing for interoperability between different software tools and hardware platforms. Think of it as the assembly language for quantum computers.

However, despite its widespread adoption, OpenQASM 2.0 (the most common version) has significant shortcomings that have prevented it from becoming a true lingua franca. Its most critical limitation is the lack of support for complex classical control flow. Modern hybrid algorithms often require real-time classical processing based on measurement results to decide the next steps in the computation (a feature sometimes called "dynamic circuits"). QASM 2.0 was not designed for this. To work around these limitations, different frameworks often resorted to proprietary extensions, leading to a fragmented ecosystem where a QASM file from one tool might not be fully compatible with another. This defeats the purpose of a universal standard and highlights the need for a more powerful and expressive representation.

Here is an example of OpenQASM code:

```
OPENQASM 3;
include "stdgates.inc";

const n = 3; // number of qubits
qubit[n] q;  // a register 'q' of n qubits
bit[n] c;    // a register 'c' of n classical bits

h q[0];      // Hadamard gate
for k in [0:n-1] {
  cnot q[k], q[k+1]; // Controlled-NOT from control qubit q[k] to target qubit
  ↪  q[k+1]
}

c = measure q; // measure quantum register
```

CUDA-Q

Not all quantum programming is done in Python. As the field moves towards integrating quantum processors into high-performance computing (HPC) centers, the need for more performant classical languages becomes apparent. CUDA-Q is NVIDIA's answer to this challenge. It is a platform designed for

building and running hybrid quantum-classical applications, allowing developers to integrate quantum kernels written in C++ or Fortran with large-scale classical computations running on GPUs. This approach is essential for applications where the classical processing part is a significant bottleneck.

QIR

While Qiskit, Cirq, and other SDK-based circuits (in the form of Python objects) or QASM are good at representing circuits, they are not great at integrations and inter-operability. Each quantum hardware vendor has to implement adapters for each format. This situation is very similar to various computer architectures (e.g. x86, ARM, RISC, etc.) and/or operating systems (Windows, macOS, Linux, etc.) having to deal with programs in different languages.

In classical compilers, it's common to translate source code into an intermediate representation (IR) before turning it into machine code. Think of it as a middle step: the front end of the compiler takes care of translating from the programming language into IR, and then the back end translates that IR into machine instructions. The nice thing about this setup is that it decouples programming languages from hardware. You can add support for a new language by just writing a new front end, and you can target new hardware by just writing a new back end. The IR sits in the middle and makes everything more modular and reusable.

IRs are usually flexible enough to represent many different languages, and at this stage you can also apply optimizations or reorganize code to make it run more efficiently. Once you know the target hardware, the IR is compiled into actual executable code. This approach means that:

- Lots of different languages can share the same optimization and code generation tools
- A single language can be compiled to many different hardware platforms
- Compiler development becomes more efficient since a lot of the heavy lifting is shared

One notable toolkit in this area is LLVM, a target-independent optimizer and code generator, and a collection of modular and reusable compiler and toolchain technologies. It can be used to develop a frontend for any programming language and a backend for any instruction set architecture.

Quantum Intermediate Representation (QIR) is just this idea applied to quantum computing. It's a common middle layer that sits between quantum programming languages/frameworks and the quantum hardware they run on, similar to LLVM's IR. Instead of every language having to talk directly to every type of hardware, QIR provides a shared way to describe quantum programs in a format that's independent of both the programming language and the hardware. It's built on top of LLVM IR, a widely used compiler framework, and it's being developed by the QIR Alliance (Microsoft is one of the members).

QIR is built on LLVM, which is already used by tons of classical programming languages. Instead of reinventing the wheel, QIR just defines some rules for how quantum constructs should be represented in LLVM IR. The cool part is that this means QIR can naturally handle both classical and quantum logic—super important for hybrid quantum–classical algorithms. It also lets us reuse existing compiler tools and optimizations from the classical world, which saves time and effort. QIR is already being picked up by major players in the quantum space. NVIDIA, Oak Ridge National Lab, IQM, Quantinuum, and Rigetti are all building tools around it.

QIR is not meant to be easily read by humans, but it is still somewhat human readable. Here is an example of a portion of QIR code:

```
define void @BellPair__body(%Qubit* %qb1, %Qubit* %qb2) {
entry:
  call void @__quantum__qis__h(%Qubit* %qb1)
  call void @__quantum__qis__cnot(%Qubit* %qb1, %Qubit* %qb2)
  ret void
}
```

Chapter 13. Hybrid computation

In any discussion about practicalities of running quantum algorithms, there is always an elephant in the room: hybrid computation. For a huge portion of useful algorithms, and even for quantum error correction, a combination of quantum and non-quantum (classical) computation is required.

What is quantum-classical hybrid computation

A "hybrid algorithm" is a very wide term. It can be as simple as classical control where a measurement result from one qubit controls the flow of execution akin to a simple if/then statement. Or it be a complex machine learning algorithm that needs to run as close to the quantum chip as possible to process some measurement results and affect the parameters of the subsequent gates. Overall, this is still an unsolved problem.

There are 3 distinct types of hybrid computing. Note that this categorization is not an industry standard, and different people may mean different things when they say "hybrid". Heck, even words like "calibration" and "circuit" are ambiguous sometimes. This just shows, again, how young the industry is.

The three types we can observe are

1. Hybrid remote.
2. Hybrid adjacent.
3. Hybrid tight.

Hybrid remote

Hybrid remote is when the quantum part of the job is sent to the remote hardware backend via the network (e.g. the internet), and the classical part is performed on the client side. There is no true integration of any sort.

This is the most common method of accessing quantum computers nowadays. For example, you can sign up for some cloud access on IBM, IQM or Amazon, and you'll receive details about the remote URL and credentials. You then spin up a local program, like a Python process or a jupyter notebook, install the necessary libraries, prepare your circuits and submit them by defining the "backend" using the provided URL and credentials. Every time you submit a

circuit (or, more likely, a batch of circuits), the data is sent to the server and placed in a queue among jobs from other users. A single quantum computer can perform a single thing at a time, so some sort of queueing is necessary. The backend may be implemented with a naive first-in-first-out (FIFO) queue, or a more complicated system with priorities and optimizations. Ironically, the problem of optimal scheduling is very difficult, and quantum computers can in theory be applied to find better solutions.

Once your job completed, you receive the results into the same running Python process or a notebook. It's usually post-processed in some way and converted into the object representation of the SDK you use (be it Qiskit, Cirq or something else). Your task may require using the measurement results to construct the next batch of circuits, run them, and repeat the process. In a way, this is the most inefficient way to do hybrid computation. Certain algorithms require a rapid succession of circuit executions, and the classical part in between is often very simple and quick. The majority of time is therefore spent on network communication, serialization-deserialization of data, and waiting in the queue.

Naturally, there is no way to continue from some quantum state after a given job completes. Each new job runs in a complete blank slate. If the algorithm requires the quantum state to be preserved while the classical part is being computed, this approach will not work.

Apart from that aspect, some cloud providers provide ways to partially mitigate the slowness. For example, repeated job submissions from the same user may be prioritized for a short period of time after the first job. Or, more commonly, users simply book dedicated time on the machine, and therefore their job always run first as the queue is not accessible to other users during the booking period. This is very bad for QPU utilization and leads to the QPU being unused most of the time, but you can probably already think of ways of combining booking and sharing. This remind a lot of time-sharing systems of early classical computers.

Often, people in the industry do not see this model as hybrid at all.

Hybrid adjacent

Hybrid adjacent is similar to hybrid remote, except the classical process runs on the server side, ideally as close to the quantum hardware as possible. For example, some cloud providers allow you to run a session in the browser and,

use jupyter notebooks and other software remotely. This shortens the link between the classical process and the quantum process, but does not change things fundamentally. All drawbacks of the previous approach are still present.

Hybrid tight

Tight integration is the ultimate goal. We want to be able to run classical code (on a CPU or a GPU or an FPGA) within the coherence time, i.e. in the middle of the circuit, or at least very close to that timing. In a way, mid-circuit measurement and classical control are the simplest one-bit-one-branch versions of it.

This goal is especially important in the context of quantum error correction. Maintaining the state of a logical qubit which is implemented with a dynamic set of physical qubits requires constant background classical computation.

Currently, Nvidia is actively cooperating with several quantum hardware startups in an attempt to build an open system architecture for tightly coupling the GPU computing structures with quantum processors to ultimately build accelerated quantum supercomputers. Recently in 2025 Quantum Machines, one the manufacturers of control electronics and cryogenic electronics, announced its integration with NVIDIA's NVQLink platform, focusing on real-time orchestration between quantum and classical computing systems.

This is an extremely important area. Arguably, the most important one outside of the pure quantum physics/fabrication/algorithms.

HPC integration

HPC integration is a constantly hot topic in the industry. Aforementioned NVIDIA's NVQLink platform is probably the biggest, and most known project in this area.

HPC stands for high-performance computing. Also known as simple supercomputers. An HPC cluster contains multiple high-speed computer servers networked with a centralized scheduler that orchestrates the massively parallel computing workload. The computers, called nodes, use either high-performance multi-core CPUs or—more likely today—GPUs, which are well suited for rigorous mathematical calculations, machine learning (ML) models and graphics-intensive tasks. A single HPC cluster can include 100,000 or more nodes.

For good old HPC centers, GPUs are commonly viewed as just specific computing nodes. They can be connected to existing systems, clusterized, controlled efficiently and precisely. GPUs are also very fast, both in their computing power and in communication. Ideally, HPC centers would like to connect quantum computers in a similar fashion. They want to have a very clear and strict interface towards QCs, allowing for deep integration with the existing scheduling and balancing software.

Several HPC centers around the world already host quantum computers on their premises and allow their users to run quantum jobs. For example, scientists working in chemistry or material science often use GPUs to simulate quantum systems, but today can also offload a subset of their workload onto a real quantum computer. This is often required to validate their research results.

MLIR

In a previous chapter we've seen multiple DSLs (domain-specific languages) or formats to define quantum circuits. One of them was different to everything else: QIR. It was developed because standard LLVM IR is not inherently equipped to represent the unique semantics of quantum operations, such as quantum gates, measurements, or qubit management. QIR currently is limited to gates. But what if we need to combine a rich multi-layered cake of hybrid and pulse and whatever else may be invented in the quantum domain?

To tackle this, some folks are trying to adopt the Multi-Level IR (MLIR) framework, which originates as a sub-project of LLVM. MLIR is a more general and extensible compiler infrastructure designed to address the representation of diverse and domain-specific abstractions. It can be thought of as a "meta-IR," a framework for building other IRs. Instead of a single, fixed set of instructions like LLVM IR, MLIR provides a system for defining "dialects." Each dialect is a collection of custom operations and types tailored to a specific domain, such as quantum computation, high-performance computing, or machine learning accelerators.

A single program representation in MLIR can contain operations from multiple dialects simultaneously. This is exceptionally well-suited for hybrid quantum-classical computing. One can represent the classical control flow (e.g., loops, conditionals) using a standard dialect, while representing the quantum circuit operations (e.g., Hadamard gates, CNOTs) using a quantum-specific

dialect, all within a unified representation. One can define a dialect for pulse-level access and then combine it with other existing dialects, achieving a super-hybrid program format of sorts. MLIR's infrastructure allows for progressive lowering, where high-level, abstract operations are gradually transformed into lower-level, more hardware-specific representations through a series of dialect-aware passes. This structured approach allows for optimizations at multiple levels of abstraction, from high-level algorithmic rewrites down to hardware-specific gate decompositions.

The output of all of this is a assembly-like source code, and the big question is how to run it. Many vendors today simply cannot accept QIR or MLIR. They expect only simple circuits in some format equivalent to Qiskit or Cirq, often transmitted to the server in a simple JSON form or some other encoding. These circuits cannot contain arbitrary classical code.

Chapter 14. What's next

We have only scratched the surface of the enormously large and complex topic of quantum computation. Things to explore on your own include:

- Real-time or near real-time control. Continuing on the topic of tight hybrid integration, various methods, and the debate between Quantum-GPU or Quantum-CPU or Quantum-FPGA integration for quantum error correction.
- Simulation vs. real hardware. Circuit-level and pulse-level simulation.
- Quantum communication. How to transfer quantum state between distinct machines.
- Biggest challenges in scaling, especially scaling superconducting quantum computers. Do look into what it takes to build a million (physical) qubits system with today's approach; for fun, try to calculate the total length of cables necessary to wire up the 1M qubit QPU.

* * *

I strongly believe that the quantum computing industry needs more professional software engineers to join the ranks. Given the academic and institutional roots, a lot of the problems and design requirements mirror classical systems, with an inevitable quantum twist. A lot of existing, established solutions can be applied to quantum software systems along at least the following aspects:

- task scheduling and optimization
- modular code architecture
- memory management
- efficient data representation, (de)serialization, encoding and decoding
- sustainable software development in a rapidly changing industry
- DX (developer experience), a highly underrated topic in all of software, but especially so in quantum software

I honestly struggle to convey the scale of things that need to be designed, implemented, and maintained. I haven't seen such an open field of exciting foundational problems in any other area. Modern software engineers, myself included, often lament about the "good old days", and imagine how exciting it must've been to be there at the beginning, designing first programming languages, first compilers, operating systems, network protocols, etc. Quantum computing is your chance to be there! With an additional benefit of having all the knowledge of the classical computing history and the impressive modern toolset of languages and resources.

I hope this short book has been interesting and useful to you. My primary goal was to give a wide enough overview of the land. When quantum companies hire software engineers, they rarely expect them to know much about quantum, understandably. By reading this book and, hopefully, reading some other resources online, you will have a big advantage.

So, would you like to join the exciting wild west world of a new computing paradigm?

Recommended educational resources and books about quantum computing and quantum physics:

- Quantum Country. A free introduction to quantum computing and quantum mechanics (https://quantum.country/)
- IBM courses and tutorials (https://quantum.cloud.ibm.com/learning/en)
- Google quantum AI resources (https://quantumai.google/resources)
- IQM Academy (https://www.iqmacademy.com/)
- Basics of quantum programming with Pennylane (https://pennylane.ai/qml)

* * *

This book was made possible thanks to a grant by Unitary Foundation (https://unitary.foundation/), a non-profit helping create a quantum technology ecosystem that benefits the most people. The cover image was created by Natalia Bass (https://tashabass.com/).

Espoo, Finland, 2025.