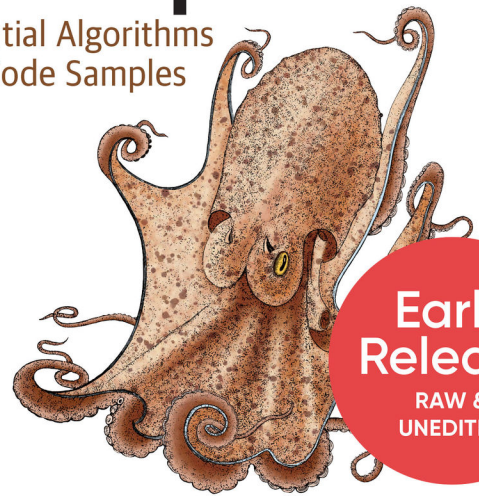
O'REILLY®

Programming Quantum Computers

Essential Algorithms and Code Samples



Early Release

RAW & UNEDITED

Eric R. Johnston, Nic Harrigan
& Mercedes Gimeno-Segovia

Programming Quantum Computers

Essential Algorithms and Code Samples

Eric R. Johnston, Nic Harrigan,
and Mercedes Gimeno-Segovia

Practical Programming on Quantum Computers

by Mercedes Gimeno-Segovia , Nic Harrigan , and Eric R. Johnston

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Editor: Mike Loukides
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- July 2019: First Edition

Revision History for the Early Release

- 2019-01-24: First release
- 2019-03-21: Second release

See http://oreilly.com/catalog/errata.csp?isbn=9781492039686 for release details.

978-1-492-03961-7

Chapter 1. Introduction

Quantum computers are no longer theoretical devices.

The authors of this book believe that the best uses for a new technology are not necessarily discovered by its inventors, but by domain experts experimenting with it as a new tool for their work. Discovering new applications for quantum computing means making tools accessible to these experts. With that in mind, this book is a hands-on programmer's guide to using quantum computing technology.

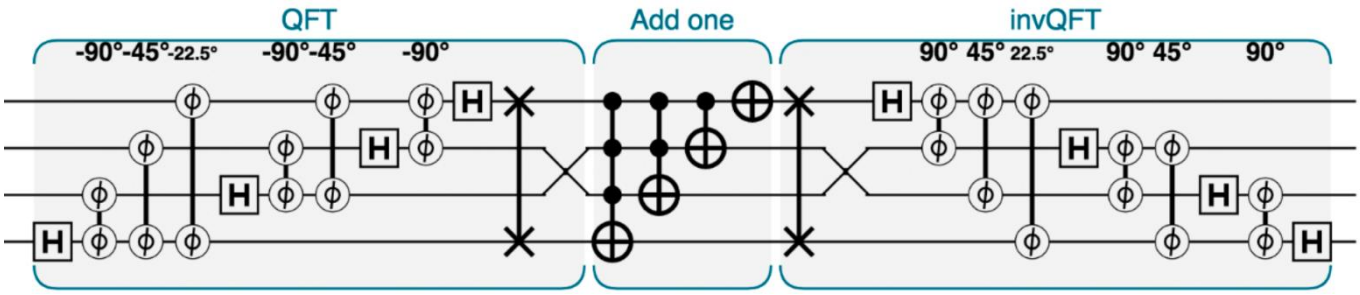


Figure 1-1. Quantum logic can look a bit like sheet music

Whether you're an expert in software engineering, computer graphics, data science, or just a curious computerphile, this book is designed to let you learn how the power of quantum computing might be relevant to you, by actually starting to use it. In the chapters ahead, you will become familiar with symbols such as those in Figure 1-1, and the code used in applying them to problems you care about.

To facilitate this, the following chapters do *not* contain thorough explanations of quantum physics (the laws underlying quantum computing) or even quantum information theory (how those laws talk about processing information). Instead, they provide working examples demonstrating capabilities of this exciting new technology. Most importantly, the code we present for these examples can be tweaked and adapted. This allows you to learn from them in the most effective way possible - by being hands-on and trying things out. Along the way, core concepts are explained as they are used, and only in so far as they build an intuition for writing quantum programs.

While quantum computing technologies span a very wide range of applications such as quantum chemistry and physical simulation, this book will focus on computational primitives.

Our humble hope is that interested readers might be able to wield these insights to apply and augment applications of quantum computing in fields that physicists may not even have heard of. Admittedly, hoping to help spark a quantum revolution isn't *that* humble, but it's definitely exciting to be a pioneer.

Required background

The physics underlying the qubits of quantum computing is full of dense mathematics. But then so is the physics underlying the transistor, and yet learning C++ needn't involve a single physics equation. In this book we take a similarly *programmer-centric* approach, circumventing any significant mathematical background. That said, here is a short list of background knowledge that may be helpful in digesting the new concepts that we introduce:

1. Familiarity with programming control structures (`if`, `while`, etc.). JavaScript is used in this book to provide lightweight access to samples which can be run online. If you're new to JavaScript but have some prior programming experience then the level of background you need could likely be picked up in less than an hour. For a more thorough introduction to JavaScript see Learning Javascript.
2. Occasionally we will employ some relevant programmer-level mathematics, such as:
    - An understanding of using mathematical functions.
    - Familiarity with trigonometric functions.
    - Comfort manipulating binary numbers and converting between binary and decimal representations.
3. A very elementary understanding of how to assess the computational complexity of an algorithm (i.e. *big-o* notation).

One part of the book that reaches beyond these requirements is [Link to Come] - where we survey a number of applications of quantum computing to machine learning. Due to space constraints our survey gives only very cursory introductions to each machine learning application before showing how a quantum computer can provide an advantage. Although we intend the content to be understandable to a general reader, those wishing to really experiment with these applications will require a bit more of a machine learning background (for any reader interested in developing this, we provide some introductory references in [Link to Come]).

This book is about programming (not building, nor researching) quantum computers, which is why we can do without advanced mathematics and quantum theory. However, for those interested in exploring the more academic literature on quantum computing, [Link to Come] provides some good initial references and shows how you can begin to link the concepts we introduce to the mathematical notations commonly used by the quantum computing research community.

What is a QPU?

Despite its ubiquity, the term "Quantum Computer" can be a bit misleading. It conjures images of an entirely new kind of machine, probably having some kind of glowing liquid keyboard and trans-dimensional display - one that supplants all existing computing software with a futuristic alternative.

At the time of writing this is a common, albeit huge, misconception. The promise of quantum computers stems not from them being a *conventional computer killer*, but rather from their ability to dramatically *extend* the kind of problems that are tractable within computing. There are *some* specific, yet important, computational problems that are easily calculable on a quantum computer, but that would quite literally be impossible on any conceivable standard computing device that we could ever hope to build[1].

But crucially, these kind of speedups have only been seen for certain problems (many of which we later elucidate on), and although it is anticipated that more will be discovered, it's highly unlikely that it would ever make sense to run *all* computations on a quantum computer. For most of the tasks taking up your laptop's clock-cycles a quantum computer would perform no better.

In other words - from the programmer's point of view - a quantum computer is really a *co*-processor. In the past, computers have used a wide variety of co-processors, each suited to their own specialties, such as floating-point arithmetic, signal processing, and real-time graphics. With this in mind, we will use the term **QPU** (Quantum Processing Unit) to refer to the device on which our code samples run. We feel this reinforces the important context within which quantum computing should be placed.

As with other co-processors such as the GPU (Graphics Processing Unit), programming for a QPU involves the programmer writing code which will primarily run on the CPU of a normal computer. The CPU issues the QPU co-processor commands only to initiate tasks suited to its capabilities.

A Hands-on Approach

The hands-on samples we present form the backbone of this book. But at the time of writing a full-blown general purpose QPU does not exist - so how can you hope to ever run the code that we present? Fortunately (and excitingly), even in 2019 a few prototype QPUs *are* currently available from companies such as IBM, and can be accessed on the cloud. Furthermore, for smaller problems it's possible to *simulate* the behavior of a QPU on conventional computing hardware. Although simulating larger QPU programs becomes impossible, for smaller code snippets it's a much more convenient way to learn how we might control an actual QPU. The code samples in this book are compatible with both of these scenarios, and will remain both usable and pedagogical even as more sophisticated QPUs become available.

To ensure that our code samples are widely employable we provide them, whenever possible, in each of the following languages:

- **QASM:** QASM is a quantum assembly language, which can run directly on hardware and simulators available from IBM, and is provided online for free public use.
- **Python:** Qiskit is a free and open source Python library for quantum computation. Programs can be run in simulation as well as on IBM's more advanced hardware.
- **JavaScript:** QCEngine is a free online quantum computation simulator, allowing users to run samples in a browser, with no software installation at all. This simulator was developed by the authors, initially for their own use and now as a companion for this book. QCEngine is especially useful for us, both because it can be run without the need to download any software, and also because it incorporates the *circle notation* that we use as a visualization tool throughout the book.

There are many other QC simulators, libraries, and systems available. A list of links to several well-supported systems is available at *http://oreilly-qc.github.io*. The three primary platforms chosen for the book represent a balance between ease-of-reading and ability to run on actual hardware available at the time of publication.

To prevent code samples from overwhelming the text, we provide samples only in JavaScript for QCEngine. This simulation engine has the advantages of being easily available online, very fast, and tailored to our approach. Full implementations of the books code samples in the other languages can be found online at *http://oreilly-qc.github.io*.

It's worth noting that the quantum languages we include are by no means an exhaustive list of all those available at the time of writing. While the syntax may differ, most quantum languages share the common structure and operations of the QPU. The intuitions and core concepts built by this book can readily be applied to other existing (and future) quantum programming languages.

A QCEngine primer

Since we'll use QCEngine throughout the book, it's worth spending a little time to see how to navigate the simulator - which may be found at *http://oreilly-qc.github.io*, along with more detail its use.

RUNNING CODE

The QCEngine web interface shown in Figure 1-2 allows you to easily produce the various visualizations we rely on throughout the book to get to grips with the unintuitive nature of qubits. With zero install and minimal friction, you can create these visualizations simply by entering code into the QCEngine code editor:

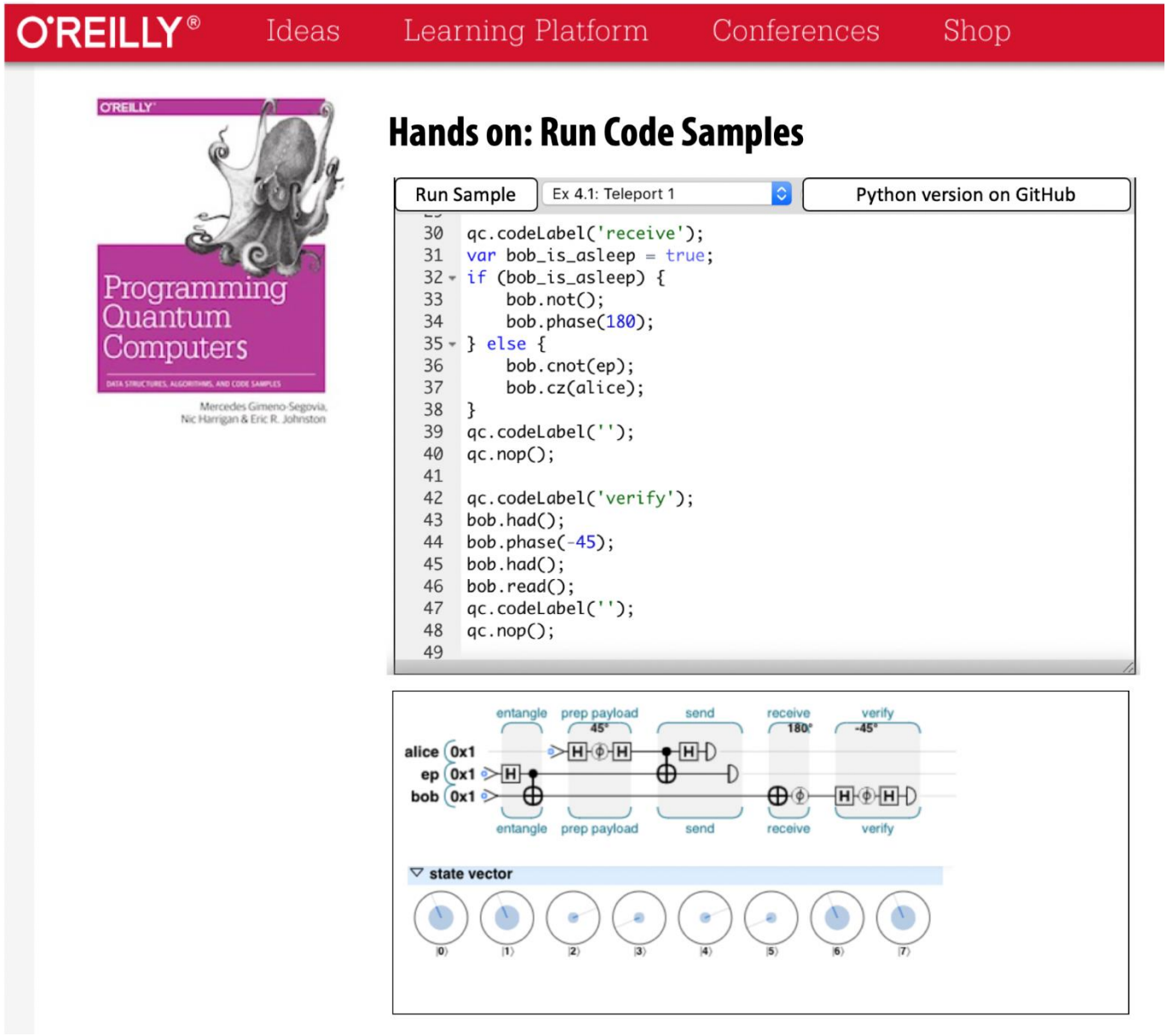


Figure 1-2. The QCEngine UI

You can also find the code samples from the book already online. To run one as-is, select it from the list at the top of the editor and click `Run Sample` in order to execute the code using the QCEngine. The quantum simulation will then run locally on your machine.

After clicking the `Run Sample` button, some new interactive UI elements will appear for visualizing the results of running your code:

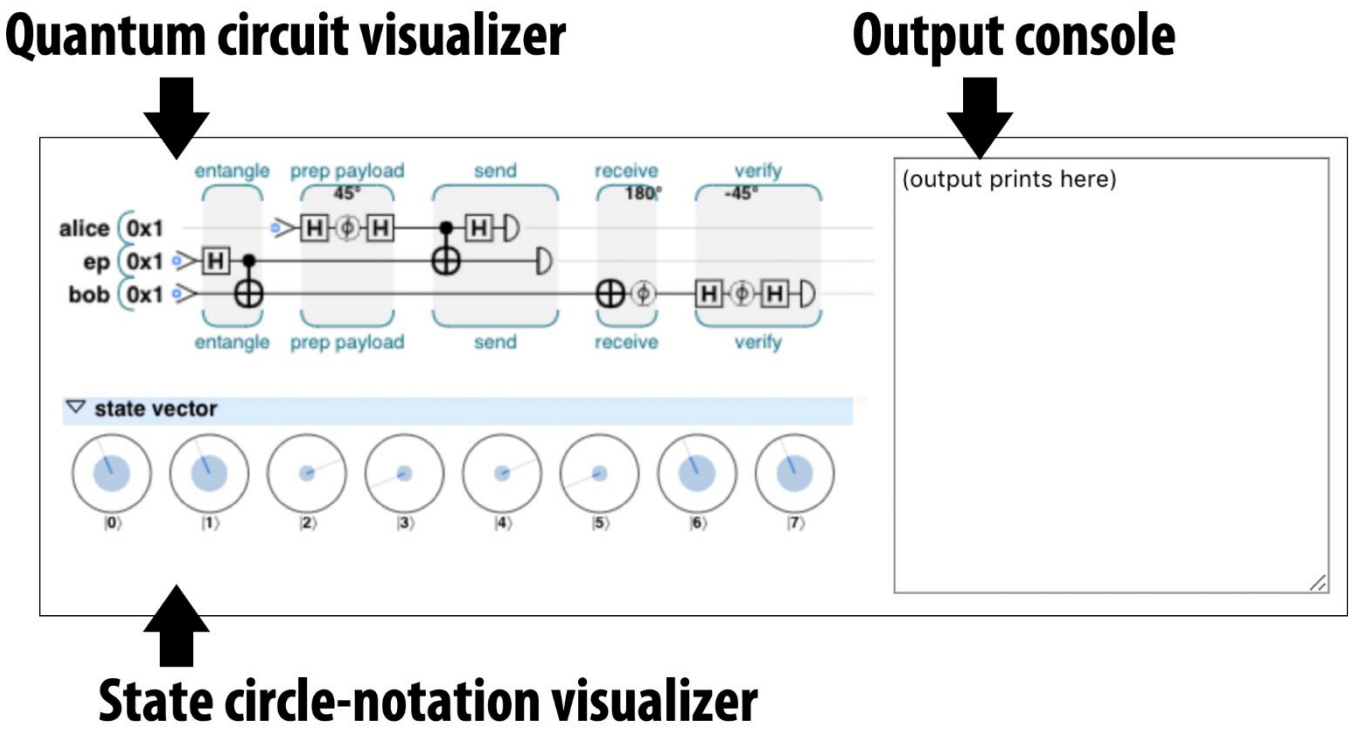


Figure 1-3. QCEngine UI elements for visualizing QPU results

Circuit visualizer: This element presents a visual representation of the circuit that your code represents. We'll introduce the symbols used in these circuits in Chapter 2 and Chapter 3. As we describe below, this view can also be used to interactively step through the program. We will use this visualizer to represent both classical and quantum information. whenever a qubit is carrying quantum information, the line will be black, and whenever there is only transmission of classical information, the line will be gray. We already show you an example of this in Figure 1-3, soon you will be able to understand all the elements of the diagram.

State circle-notation visualizer: This displays the so-called *circle-notation* visualization of qubits in the QPU (or simulator) that's running your code. We explain how to read and use this notation in Chapter 2.

QCEngine output console: This is where any text appears that may have been printed from within your code (i.e. for debugging) using the `qc.print()` command. Anything printed with the standard JavaScript `console.log()` function will still go to your web browser's JavaScript console. As an example, if you add `qc.print("No cat puns allowed");` to the end of the default QCEngine code snippet and click `Run Sample` again, then you should see this text in the output console.

DEBUGGING CODE

Debugging QPU programs can be tough. As each line of code executes, the configuration of qubits inside the QPU changes, and quite often the easiest way to understand what a program is doing is to slowly step through it, inspecting a qubit visualization at each step. Hovering your mouse over the **circuit visualizer**, you should see a vertical orange line appear at a fixed position and a gray vertical line wherever in the circuit your cursor happens to be. The orange line shows which position in the circuit (and therefore the code) the circle notation visualizer currently represents. By default this is the end of the program, but by clicking on other parts of the circuit, you can have

the circle-notation visualizer show the configuration of qubits at that point in the program. For example, here's how the circle-notation visualizer changes as we switch between two different steps in the default QCEngine program:

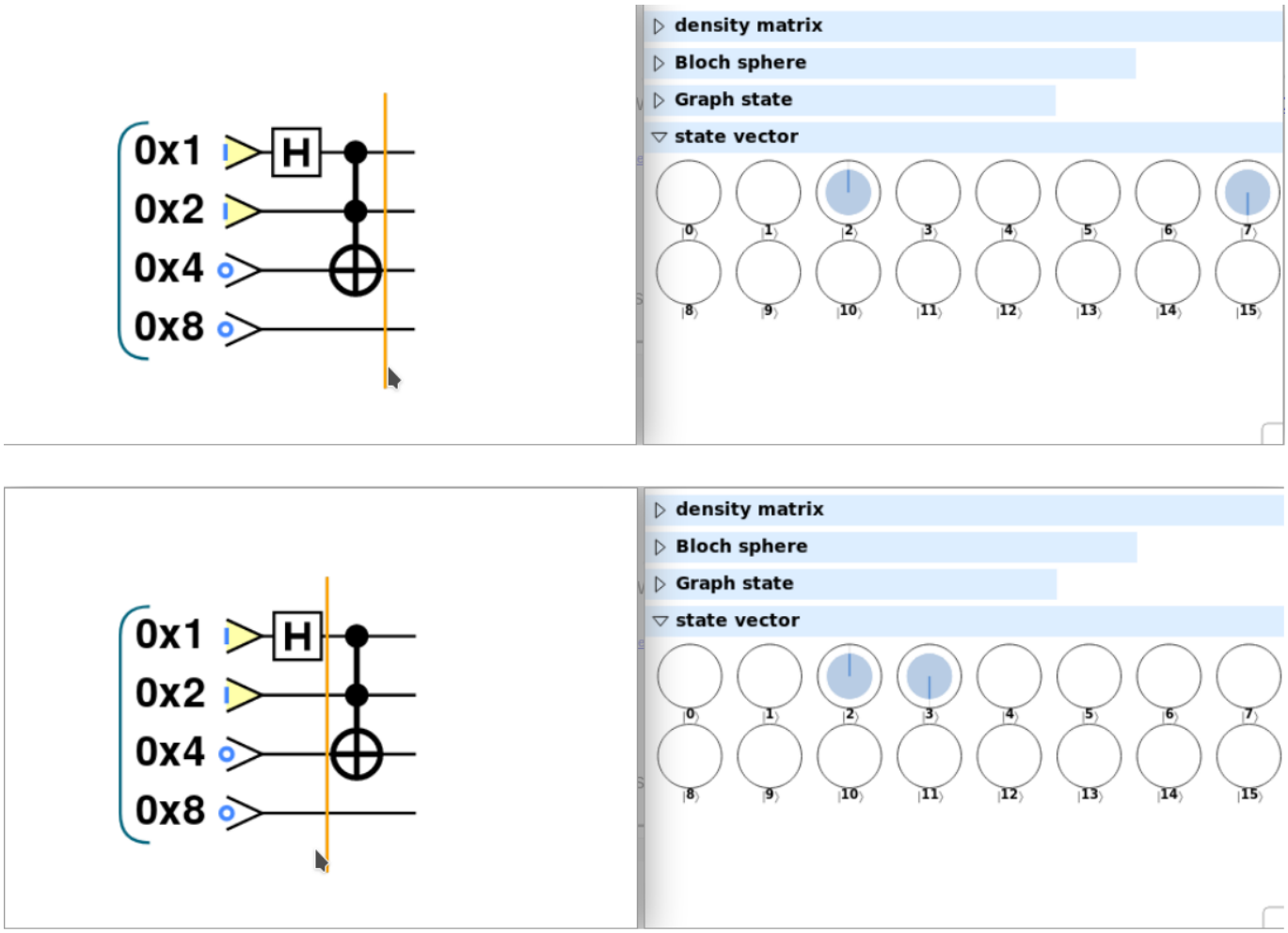


Figure 1-4. Stepping through a QCEngine progam using the circuit and circle-notation visualizers

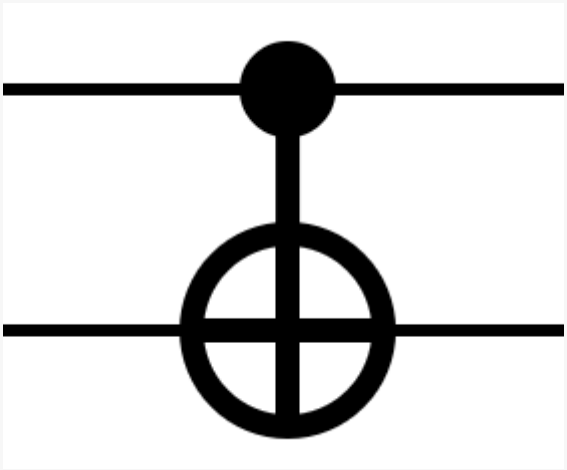
Now that you have access to a QPU simulator, you're probably keen to start tinkering. Don't let us stop you! In Chapter 2 we'll start to walk through the code for increasingly complex QPU programs.

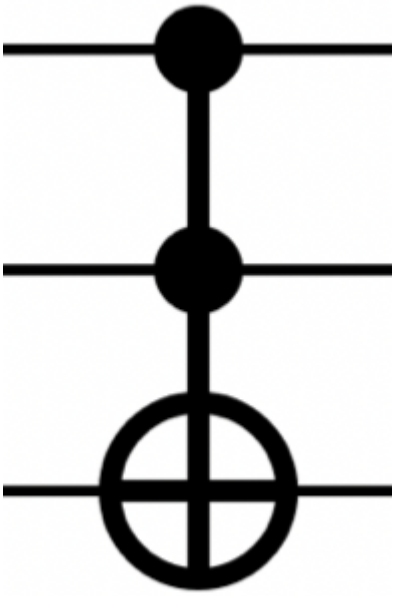
Native QPU Instructions

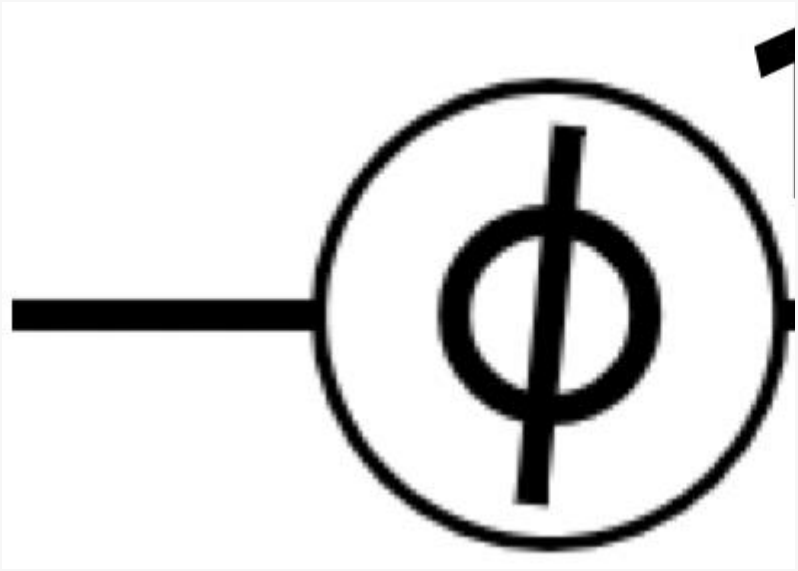
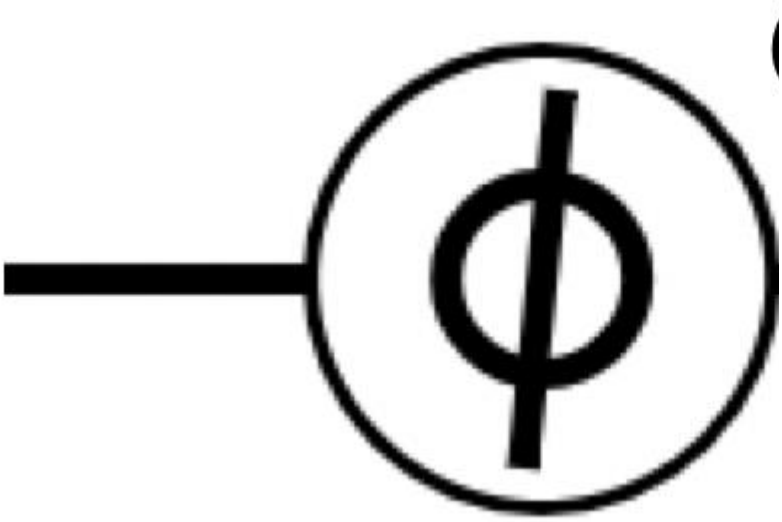
QCEngine is one of several tools that allow us to run and inspect QPU code, but what will QPU code actually look like? In conventional CPU programming, software stacks connect many layers of abstractions and tools, from raw hardware controls, to assembly-language instructions, right through to applications written in high-level languages. A similar stack exist for programming a QPU, one that incorporates both quantum and classical elements. Conventional high-level languages are commonly used to control lower level QPU instructions (as we've already seen with JavaScript-based QCEngine). In this book we'll regularly cross between these layers. Describing the programming of a QPU with machine-level operations helps us get to grips with fundamental novel logic of a QPU, whilst seeing how we to manipulate these operations from higher level conventional

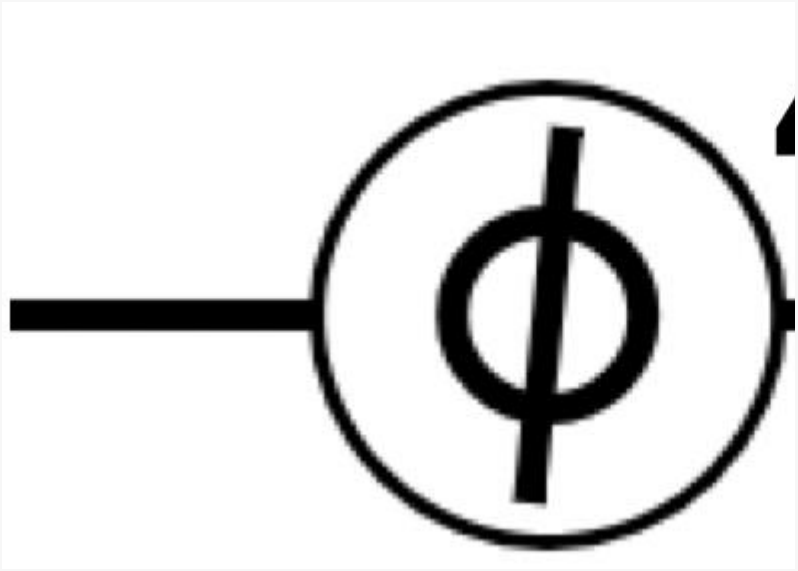
languages like JavaScript, Python or {cpp} gives us a more pragmatic paradigm for actually writing code.

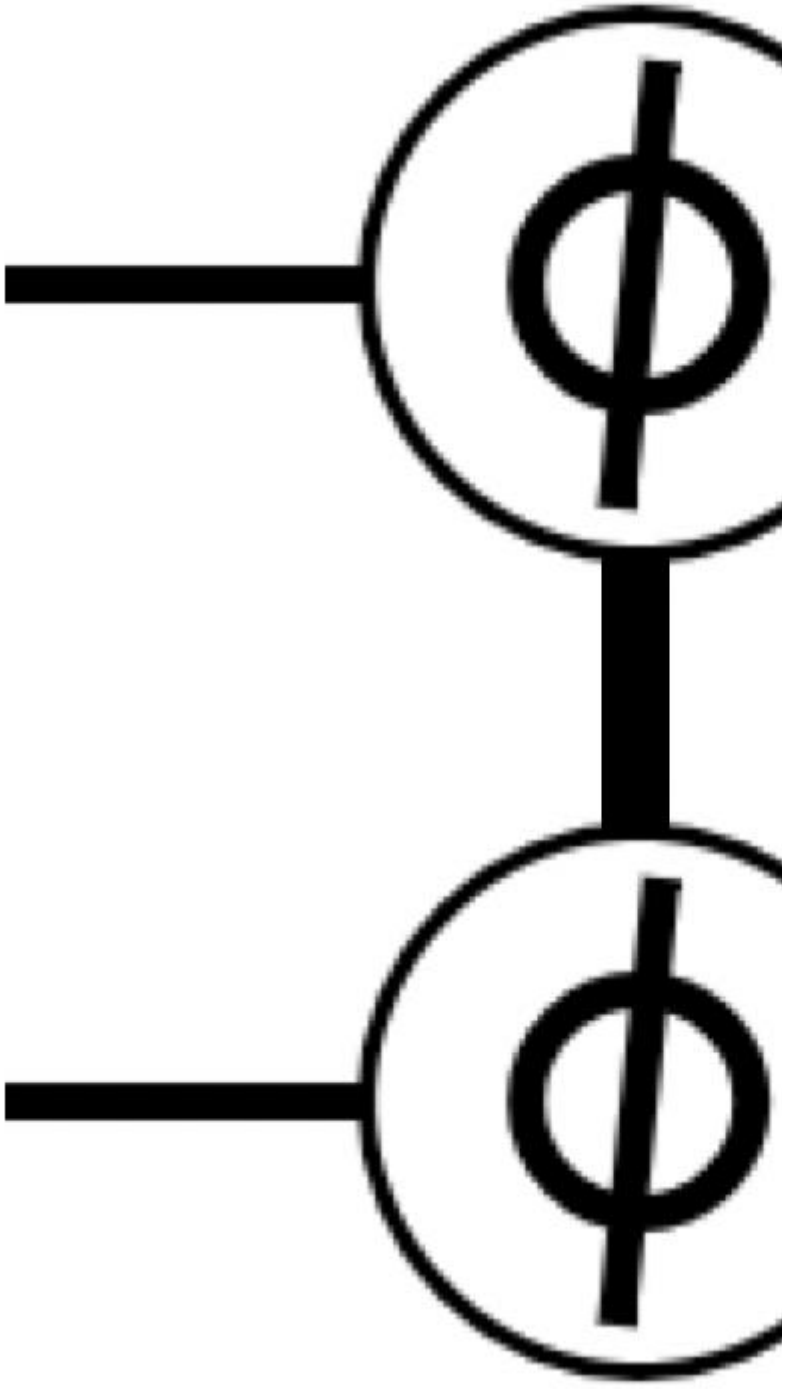
To whet your appetite, we list some of the fundamental QPU instructions in Table 1-1 below - each of which will be explained in more detail within the chapters ahead. It is worth noting that it is not necessary for a QPU to natively feature all of these instructions; in fact all quantum algorithms can be implemented with a very small set of native gates, in the same way as any classical computation can be made exclusively from NAND gates. However, most QPUs will have an extended set of gate available for use, and this list contains the most common ones.

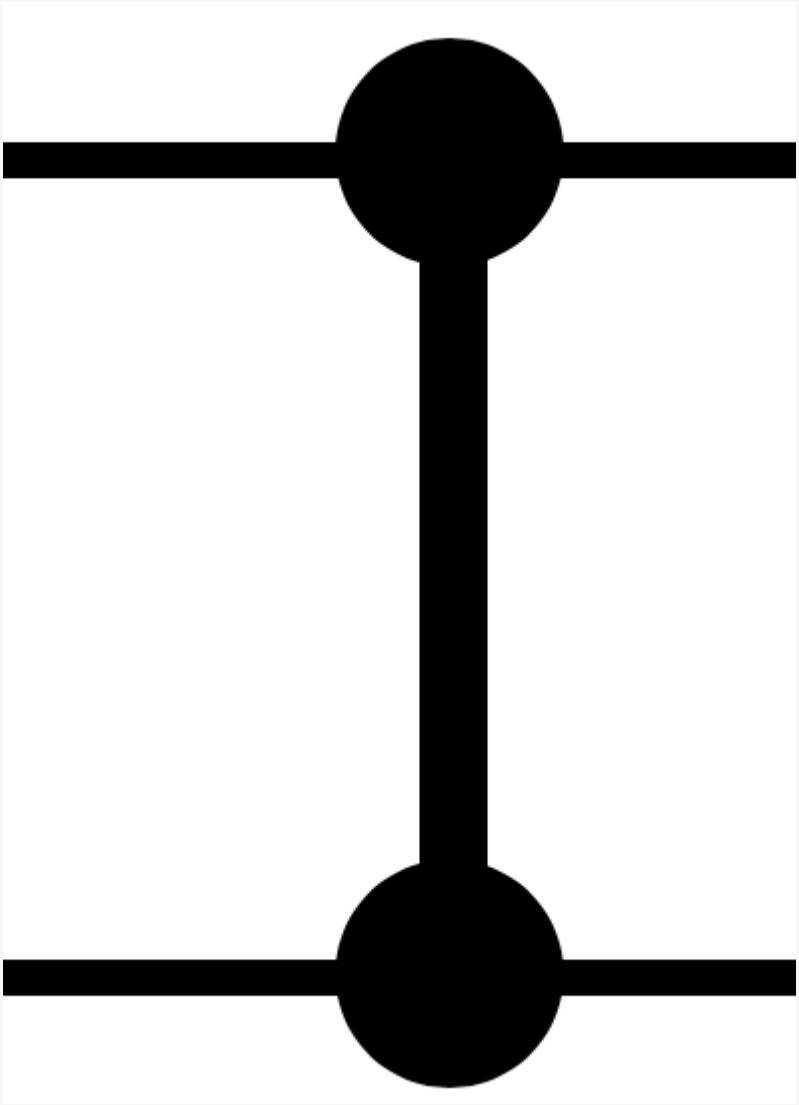
Symbol	Name	Usage	Description
	NOT (also X)	`qc.not(t)`	Logical bitwise NOT
	CNOT	`qc.cnot(t,c)`	Controlled NOT: If (c) then NOT(t)

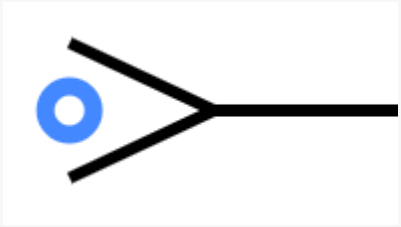
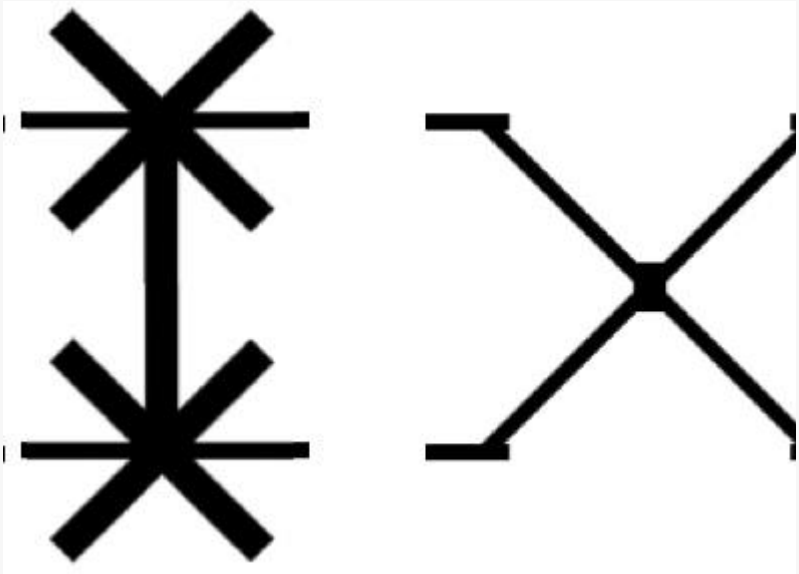
Symbol	Name	Usage	Description
	CCNOT (Toffoli)	`qc.cnot(t,c1|c2)`	If (c1 and c2) then NOT(t)
	HAD (Hadamard)	`qc.had(t)`	Hadamard gate (normally used to create quantum superposition)
	PHASE	`qc.phase(angle,c)`	Relative phase rotation

Symbol	Name	Usage	Description
	Z	`qc.phase(180,c)`	Relative phase rotation by 180 degrees
	S	`qc.phase(90,c)`	Relative phase rotation by 90 degrees

Symbol	Name	Usage	Description
	T	`qc.phase(45, c)`	Relative phase rotation by 45 degrees

Symbol	Name	Usage	Description
	CPHASE	`qc.phase(angle,c1|c2)`	Conditional phase rotation

Symbol	Name	Usage	Description
	CZ	`qc.phase(180,c1|c2)`	Conditional phase rotation by 180 degrees
	READ	`val = qc.read(t)`	Read and collapse qubits, returning digital data

Symbol	Name	Usage	Description
	WRITE	`qc.write(t,val)`	Write digital data to qubits
	ROOTNOT	`qc.rootnot(t)`	Root-of-NOT operation.
	SWAP (EXCHANGE)	`qc.exchange(t1|t2)`	Exchange two qubits

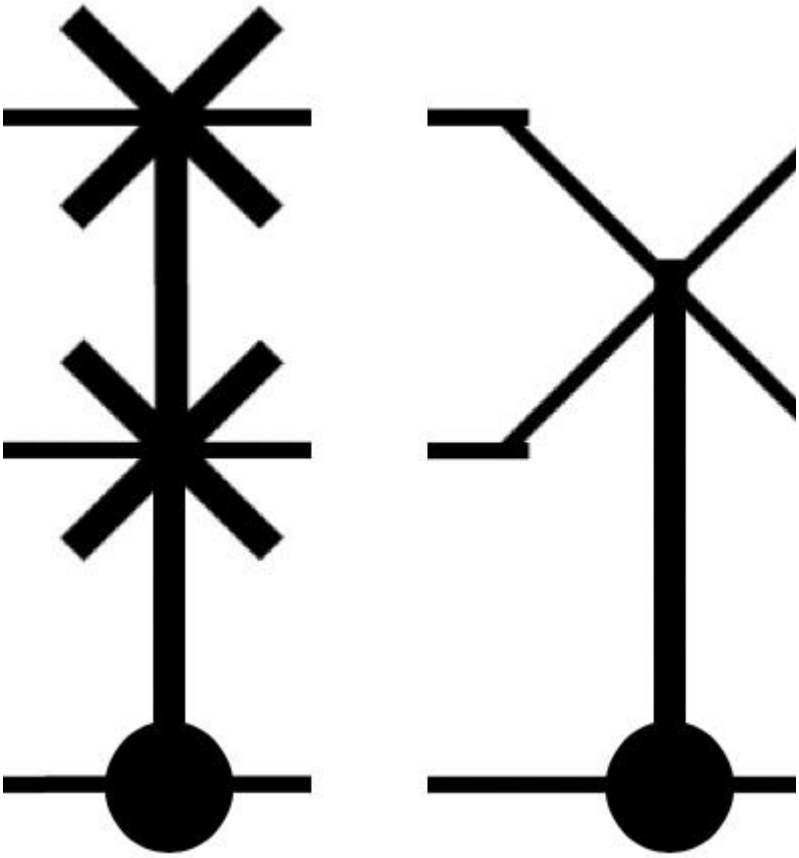
Symbol	Name	Usage	Description
	CSWAP	`qc.exchange( t1|t2, c)`	Conditional exchange: if (c) then SWAP(t1

Table 1-1. Essential QPU Instruction Set

With each of these operations, the specific instructions and timing will depend on the processor brand and architecture. However, this is an essential set of basic operations expected to be available on all machines, and these operations form the basis of our QPU programming, just as instructions like `MOV`and `ADD` do for CPU programmers.

Simulator Limitations

Although simulators offer us a fantastic opportunity to prototype small QPU programs built from the kind of operations described above, when compared to real QPUs they are hopelessly under-powered. One measure of the power of a QPU is the number of *qubits* it has available to operate on[2] (the quantum equivalent of bits, on which we'll have much more to say shortly).

At the time of this book's publication, the world record for the largest QPU that can be simulated with a conventional computer stands at 51 qubits. That's sort of like simulating a digital computer with 51 total bits of RAM (just over 6 bytes). In practice, the simulations and resources available to most readers of this book will typically be able to handle 26 or so qubits before grinding to a halt.

The examples in this book have been written with these limitations in mind. They make a great starting point, but each qubit added to them will double the memory required to run the simulation, cutting its speed in half.

Hardware Limitations

Conversely, at the time this book is being written, the largest actual QPU hardware available has around 70 *physical* qubits, whilst the largest QPU available to the public (through the QISKit project) is 16 physical qubits[3]. By *physical*, as opposed to *logical*, we mean that these 70 qubits have no error correction - making them noisy and unstable. Qubits are much more fragile than their conventional counterparts - the slightest interaction with the environment (even something as unassuming as the absorption of a single photon or a collision with a single atom) can derail the computation. What seems an insurmountable hurdle can be overcome by one of the most remarkable results in the field of quantum computing: *the threshold theorem*. This theorem proves that there exist quantum error correction protocols that allow us to build clean logical qubits from noisy imperfect ones, as long as the imperfections in our hardware are below a certain threshold value.

Dealing with logical qubits allows a programmer can be agnostic about the QPU hardware, and implement any textbook algorithm without having to worry about specific hardware constraints[4]. In this book, we focus solely on programming with logical qubits, and while the examples we provide are small enough to be run on smaller QPUs (such as the ones available at the time of publication), abstracting away physical hardware details means that the skills and intuitions you develop will remain invaluable as future hardware develops[5].

QPU vs. GPU: Some Common Characteristics

The idea of programming an entirely new kind of processor can be intimidating (even if it does already have its own StackExchange community. Here's a list of pertinent facts about what it's like to program a QPU:

- It is very rare that a program will run *entirely* on a QPU. Usually, a program running on the CPU will issue QPU instructions, and later retrieve the results.
- Some tasks are very well suited to the QPU, and others are not.
- The QPU runs on a separate clock from the CPU, and usually has its own dedicated hardware interfaces to external devices (optical outputs and such).
- A typical QPU has its own special RAM which the CPU cannot efficiently access.
- A simple QPU will be one chip accessed by a laptop, or even perhaps eventually an area within another chip. A more advanced QPU is a large and expensive add-on, and always requires special cooling.
- Early QPUs, even simple ones, are the size of refrigerators and require special high-amperage power outlets.
- When a computation is done, a projection of the result is returned to the CPU, and most of the QPU's internal working data is discarded.

- QPU debugging can be very tricky, requiring special tools and techniques. Single-stepping a program is sometimes not possible, and often the best approach is to make changes to the program and observe their effect on the output.
- Optimizations which speed up one QPU may slow down another.

Sounds pretty challenging. But here's the thing, you can replace *QPU* with *GPU* in each and every one of those statements and they're still entirely valid.

Although QPUs are an almost alien technology of incredible power, when it comes to the problems we might face in learning to program them, a generation of software engineers have seen it all before. It's true of course that there are some nuances to QPU programming that are genuinely novel (this book wouldn't be necessary otherwise!), but the uncanny number of similarities should be reassuring. We can do this!

How this book is structured

A tried-and-tested approach for getting hands-on with new programming paradigms is to learn a set of conceptual primitives. For example, anyone learning GPU programming should focus on mastering parallelism and pixel shader concepts, rather than on syntax or hardware specifics.

The heart of our book focuses on building an intuition for a set of quantum primitives – ideas forming a toolbox of building-blocks for problem solving with a QPU. To prepare us for these primitives we first introduce the basic concepts of qubits (the *rules of the game* if you like). Following outlining the primitives we show how they can be used as building blocks within useful QPU applications

As a consequence, this book is divided into three parts. The reader is encouraged to become familiar with Part I and gain some hands-on experience before proceeding to more advanced parts.

- **Part I: Programming for a QPU** - Here we introduce the core concepts required to program a QPU, such as qubit essentials, instructions, superposition logic and even quantum teleportation. Examples are provided, which can be easily run using simulators or a physical QPU.
- **Part II: QPU Primitives** - The next part provides detail on some essential algorithms and techniques at a higher level. These include *amplitude amplification*, the *Quantum Fourier Transform*, and *phase estimation*. These can be considered "library functions" that programmers will call to build applications. Understanding how they work is essential to becoming a skilled QPU programmer. There is an active community of researchers developing new QPU Primitives, therefore we expect this library to grow in the future. An appendix with resources from the research literature is provided for the interested reader who wants to keep up with the state-of-the-art.
- **Part III: QPU Applications** - The world of QPU applications is evolving as rapidly as the machines themselves. Here we introduce a few examples of existing applications, as well as some fun speculation on others.
- **Part IV: Into the Future** - From machine learning to astronomy, the final section is an exploration of where quantum computation is headed.

By the end of these three parts of the book we hope to provide the reader with an understanding of what quantum applications can do, what makes them powerful, and how to identify the kinds of problems that they can solve.

Enough introduction. Let's begin!

1 One of our favorite ways to drive this point home is the following result of a back-of-the-envelope calculation: Suppose conventional transistors could be made atom-sized, and we aimed to build a warehouse-sized conventional computer able to match even a modest quantum computers performance at factoring prime integers. We would need to pack transistors so densely that we would create a gravitational singularity. Gravitational singularities make computing (and existing) very hard.

2 Despite its popularity in the media as a benchmark for quantum computing, counting the number of qubits that a piece of hardware can handle is really an oversimplification, and much subtler considerations are necessary to assess a QPU's true power.

3 Although it's possible this figure may even become out of date whilst this book is in press!

4 For example, as well as performing no error correction, current 70 qubit QPUs have restrictions on which qubits can interact with each other - requiring special care if one is working with *that*particular piece of hardware.

5 Circa 2019, QPUs with logical (error-corrected) qubits do not yet exist. As a consequence discussions of quantum computing sometimes center on applications specifically contrived for less powerful (noisier) QPUs. These are commonly referred to as *Noisy Intermediate-Scale Quantum*, or NISQ, applications. We will not cover these applications, but rather focus on programming a fully functioning QPU.

Chapter 2. One Qubit

What exactly is a qubit? How can we visualize one? How is it useful? These are short questions with complicated answers. In this chapter we will answer them in a practical way, by describing a single qubit so that we can immediately being making use of it. The additional complexity of *multi*-qubit systems will be covered in the next chapter. The code samples that we include may be run on the QCEngine simulator (introduced in Chapter 1), using the provided links.

Qubits vs. Bits

Conventional bits have one binary parameter that we can play with - we can initialize a bit in either state 0 or state 1. This makes the mathematics of binary logic simple enough to use as-is, but we *could*visually represent the possible 0/1 values of a bit, using two separate filled/empty circles.

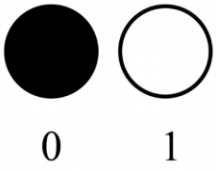
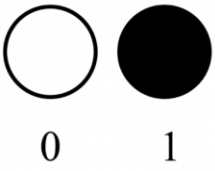
Possible values of a bit	Graphical representation
0	
1	

Figure 2-1. Possible values of a conventional bit - a graphical representation

Now onto qubits. In some sense, qubits are very similar to bits – whenever you read the value on a qubit, you'll always find a value of either 0 or 1. So *after the readout* of a qubit, we can always describe it as shown in Figure 2-1.

But characterizing qubits *before* readout isn't so black and white, and requires a more sophisticated description. Before readout, qubits can exist in a *superposition* of states, which is a state very different from a probabilistic *mixture* states, that have a non-zero probability of being in one state or another[1]. In fact, the number of possible superpositions that a qubit can exist in before being read forms an infinite continuum. Figure 2-2 lists just some of the variations we could dial up with particular choices of the magnitudes and relative phases of a superposition. Although we'll always end up reading out 0 or 1 at the end, if we're clever then it's the availability of these extra states that will allow us to perform some very powerful computing.

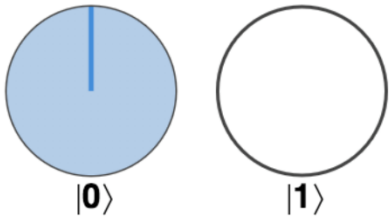
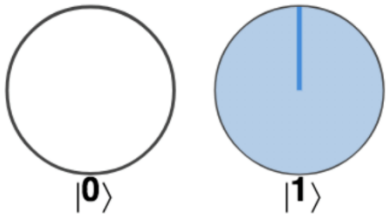
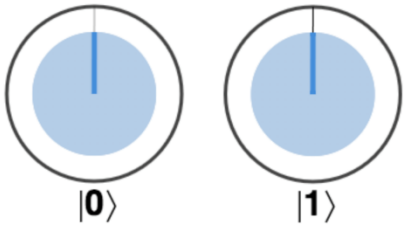
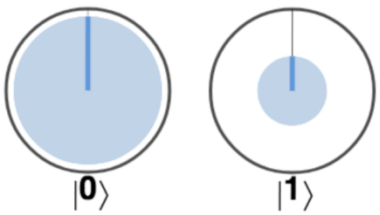
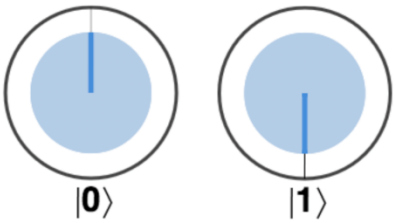
Possible values of a qubit	Graphical representation
$\|0\rangle$	
$\|1\rangle$	
$0.707\ \|0\rangle + 0.707\ \|1\rangle$	
$0.95\ \|0\rangle + 0.35\ \|1\rangle$	
$0.707\ \|0\rangle - 0.707\ \|1\rangle$	

Figure 2-2. Some possible values of a qubit

The first two rows in Figure 2-2 show the quantum equivalents of the states of a conventional bit - with no superposition at all. A qubit prepared in state |0⟩ is equivalent to a conventional bit being 0 - it will always give a value 0 on read-out - and similarly for |1⟩. If our qubits were only ever in states |0⟩ or |1⟩ (and none of the more exotic combinations illustrated in the other rows) then we'd just have a very expensive conventional bit.

But as soon as we start considering superpositions, the mathematical representations of our qubit shown in the left-hand column of Figure 2-2 begin getting much more involved than the simple binary logic of conventional bits. Fortunately, the right-hand column also presents an equivalent pictorial *circle notation*. Since our goal is building a fluent and pragmatic intuition for what goes on inside a QPU without needing to entrench ourselves in opaque mathematics, from now on we'll think of qubits entirely in terms of this circle notation. Let's see how this notation represents the amplitudes of a qubit in superposition[2].

BRA-KET NOTATION

In the mathematics within Figure 2-2 we've changed our labels from 0 and 1 to |0⟩ and |1⟩. This is called *bra-ket notation*, and is commonly used in quantum computing. As a casual rule of thumb, any possible value the qubits might (or will) have upon readout is represented using the bra-ket notation. When it *has been* read out, we just use the number to represent the resulting digital value.

Introducing Circle Notation

From experimenting with photons we've seen that there are two aspects of a qubits general state that we need to keep track of in a QPU - the **magnitude** of its superposition amplitudes and the **relative phase** between them. Circle notation displays these parameters as follows:

- The **magnitude** of the amplitude associated with each value a qubit can assume (|0⟩ or |1⟩) is related to the *fraction* of filled-in area shown for each of the |0⟩ or |1⟩ circles - or more colloquially, the *size* of each circle.
- The **relative phase** between these values amplitudes is indicated by the *rotation* of the |1⟩ circle relative to the |0⟩ circle (a darker line is drawn in the circles to make this rotation apparent).

We'll be relying on circle notation throughout the book, so it's worth taking a little more care to see precisely how the sizes and rotations of these circles capture these concepts.

Size of the circles:

The area shaded in each of the circles is directly proportional to the probability of obtaining that circle's value (0 or 1) if we read out the qubit. The examples in Figure 2-3 show the circle notation for different qubit states and the chance of reading out a 1 in each case.

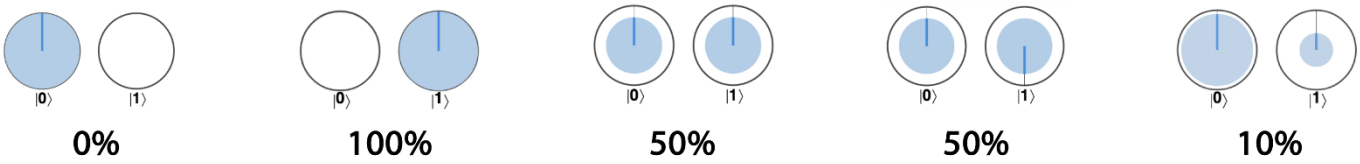


Figure 2-3. Probability of reading the value 1 for different superpositions represented in circle notation

DESTRUCTIVE OBSERVATION

Reading a qubit destroys information. In all of the above cases, reading the qubit will produce either a 0 or a 1, and when that happens, the qubit will change its state to match the observed value. So even if a qubit was initially in a more sophisticated state, once you readout 1 you'll always get 1 if you immediately continue to try and read it again.

Notice that as the area shaded in the |0⟩ circle gets larger, there's more chance you'll read out a 0, and of course that means that the chance of getting a 1 outcome decreases (being whatever is left over). In the last example in Figure 2-3, there is a 90% of reading out the qubit as 0, and therefore a corresponding 10% chance of reading out a 1.

It's easy to forget, although important to remember that in circle notation the size of a circle associated with a given outcome does *not* represent the full superposition *amplitude*. The important additional information that we're missing is the relative phase of our superposition.

Relative rotation of the circles:

Some QPU instructions will also allow us to alter the relative rotations of a qubits |0⟩ and |1⟩ circles. This represents the *relative phase* of the qubit. The relative phase between a qubits state can take any value from 0° to 360°, and a few examples are shown below.

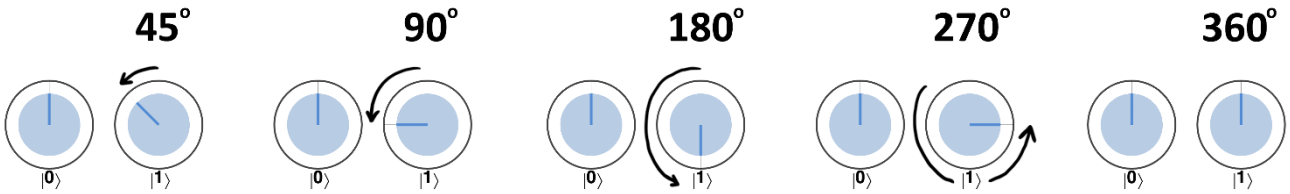


Figure 2-4. Example relative phases in a single qubit

WARNING

Our convention for rotating the circles in circle notation within this book is that a positive angle rotates the relevant circle *counter-clockwise*, as illustrated above.

In all the above example we have only rotated the |1⟩ circle. Why not the |0⟩ circle as well? As the name suggests, it's only the *relative* phase in a qubit's superposition that ever makes any difference. And consequently only the *relative* rotation between our circles[3] is of interest[4]. If a QPU operation were to apply a rotation to both circles then we can always consider them both to be rotated such that only the |1⟩ circle is rotated - making the *relative* rotation more readily apparent:

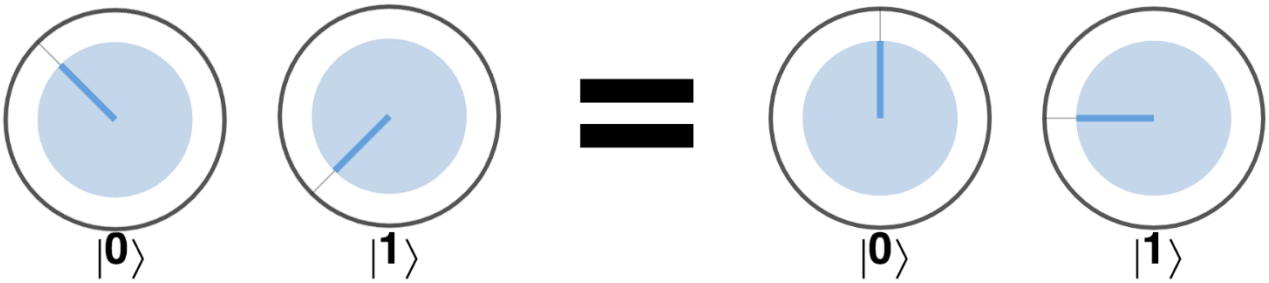


Figure 2-5. Only relative rotations matter in circle notation.

So although sometimes a simulation might show both circles rotated, it's worth bearing in mind that we can imagine rotating all the circles if it helps us make sense of things.

Note that the relative phase can be varied independently of the magnitude of a superposition (the one exception being that it doesn't make sense to talk about relative phase if a qubit state is entirely either |0⟩ or |1⟩). This independence also works the other way, comparing the third and fourth examples in Figure 2-3 we can see that the relative phase between outcomes for a single qubit has no direct effect on the chances of us reading them out.

The fact that the relative phase of a single qubit has no effect on the magnitudes in a superposition means that it has no influence on observable readout results. Although this may make the relative phase property seem inconsequential, this could not be further from the truth. In quantum computations involving *multiple* qubits, we can crucially take advantage of this rotation to cleverly and indirectly affect the chances we will eventually read out different values. In fact, by manipulating qubits with carefully chosen operations, well engineered relative phases can provide an astonishing computational advantage. We'll now introduce some of these operations - in particular those that act only on a single qubit - and we'll visualize their effects using circle notation.

A Quick Look at a Physical Qubit

Conventional programming guides almost never mention the physical nature of bits and bytes, despite the fascinating truth behind modern silicon fabrication. In fact, the ability to abstract away the physical nature of information is what makes writing complex programs of any kind possible. Similarly, this book will focus on what qubits *do* rather than what they physically *are*.

That said, we'd be remiss if we didn't give some idea of what a qubit *actually looks like*. Although the transistor has won its way to modern-day ubiquity, a visit to any computer history museum will testify that there are many ways to build digital bits. Similarly, there are many ways to physically create qubits (although at the time of writing we don't yet know which will become ubiquitous). Any physical object that can be isolated enough from the rest of the universe to display some of the more delicate phenomena of quantum physics can, in principle, be used as a qubit[5].

One object that readily demonstrates the quantum effects required of a qubit is a single photon. To illustrate this, let's take a step back and suppose we tried to use the *location* of a photon to represent a conventional digital *bit*. In the device shown in Figure 2-6, a switchable mirror (that can be set as either reflective or transparent) allows us to control whether a photon ends in up in one of two paths - corresponding to an encoding of either 0 or 1.

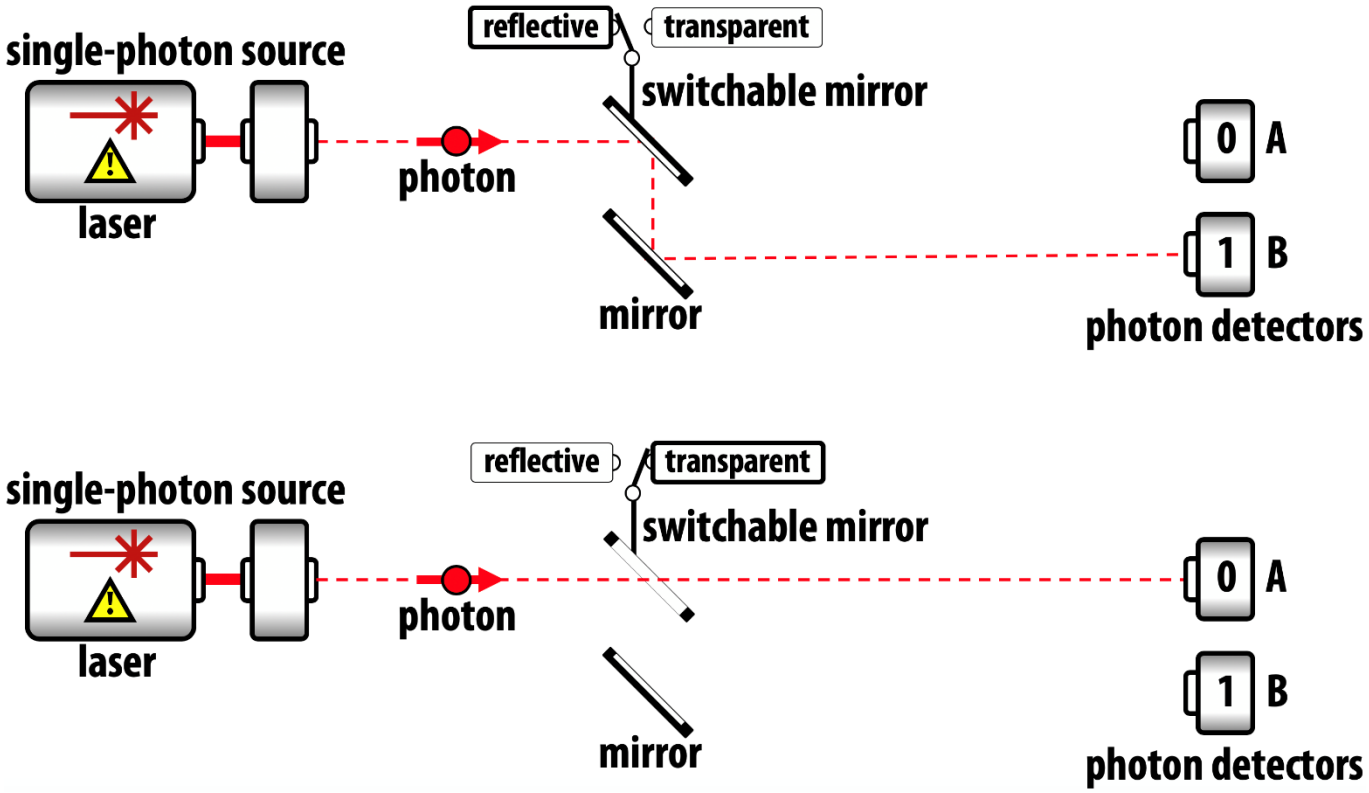


Figure 2-6. Using a photon as a conventional bit

Devices like this actually exist in digital communication technology, but nevertheless a single photon clearly makes a very fiddly bit (for starters, it won't stay in any one place for very long). To use this setup to demonstrate some qubit properties, suppose we replace the switch we use to *set* the photon as 0 or 1 with a *half*-silvered mirror.

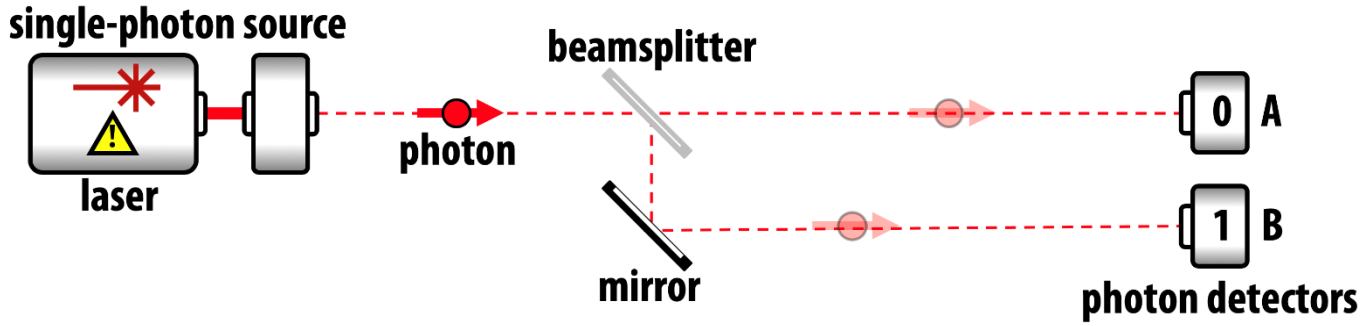


Figure 2-7. A simple implementation of one photonic qubit

A half silvered mirror, as shown in Figure 2-7 (also known as a *beamsplitter*), is a semi-reflective surface that would, with a 50% chance, either deflect light into the path we associate with 1, or allow it to pass straight through to the path we associate with 0. There are no other options.

When a single, indivisible, photon hits this surface it suffers a sort of identity crisis. In a sense that has no digital equivalent, it ends up existing in a state where it can be influenced by effects in both the 0 path and the 1 path. We say that the photon is in a *superposition* of traveling in each possible path. In other words, we no longer have a conventional bit, but a *qubit* that can be in a superposition of values 0 and 1.

It's very easy to misunderstand the nature of superposition (as many popular quantum computing articles do). It's *not* correct to say that the photon is in both the 0 *and* 1 paths at the same time. There is only one photon, so if we put detectors in each path, as shown in Figure 2-7, only one will go off. When this happens, it will reduce the quantum state of the photon into a digital bit and give a definitive 0 *or* 1 result. Yet, as we'll explore shortly, there are computationally useful ways a QPU can interact with a qubit in superposition before we need to read it out through such a detection.

The kind of quantum superposition shown in Figure 2-7 will be central to leveraging the quantum power of a QPU. As such, we'll need to describe and control quantum superpositions a little more quantitatively. When our photon is in a superposition of paths, we say it has an *amplitude* associated with each path. There are two important aspects to these amplitudes - two *knobs* we can twiddle to alter the particular configuration of a qubit's superposition:

- The **magnitude** associated with each path of the photon's superposition is an analog value determining the probability that the photon will be detected in that path. This magnitude is a measure of how much the photon has *spread* into each path. In Figure 2-7 we could twiddle the magnitudes of the amplitudes associated with each path by altering the reflectivity of the beamsplitter.
- The **relative phase** between the different paths in the photon's superposition captures the amount by which the photon is *delayed* on one path relative to the other. This is also an analog value which can be controlled by the path length difference. Note that we could change the relative phase without affecting the change of the photon being detected in each path.

WARNING

For the mathematically inclined, the amplitudes associated with different paths in a superposition are generally *complex numbers*. The *magnitude* associated with an amplitude is its absolute square, whilst its *relative phase* is the angle if the complex number is expressed in polar form. For the mathematically uninclined, we will shortly introduce a visual notation so that you need not worry about such complex issues (pun intended).

The magnitude and relative phase are values available for us to exploit when computing, and we can think of them as being *encoded* in our qubit. But if we're ever to readout any information from it, the photon must eventually strike a detector. At this point both these analog values vanish - the quantumness of the qubit is gone. Herein lies the crux of quantum computing - finding a way to exploit these ethereal quantities such that some useful remnant remains after the destructive act of readout.

HANDS-ON

This setup in Figure 2-7 is equivalent to the code sample we will shortly introduce in Example 2-1 in the case where photons are used as qubits.

Everything we've discussed so far has been specifically related to photons, and we're in danger of getting a little too attached to the idea of them forming our qubits. There are many physical ways to implement qubits, so we'd like to have a functional model of superposition, magnitude, and relative phase which is not tied to any particular implementation.

This is a programmer's guide, not a physics textbook, let's abstract away the physics and see how we can describe and visualize qubits in a manner as detached from photons and quantum physics as binary logic is from semiconductor physics.

The First Few QPU Operations

Like their CPU counterparts, single-qubit QPU instructions take an input and transform it into an output. Only now of course our inputs and outputs are qubits rather than bits. Many QPU instructions have an associated inverse, which can be useful to know about. In this case a QPU operation is said to be *reversible*, which ultimately means that no information is lost or discarded when it is applied. Some QPU operations however are *irreversible* and have no inverse (somehow they result in the loss of information). We'll eventually come to see that whether or not an operation is reversible can have important ramifications for how we make use of it.

Some of these QPU instructions may seem strange and of questionable use - but after only introducing a handful of them we'll quickly begin putting them to use.

QPU Instruction: NOT

The NOT is the quantum equivalent of a digital NOT operation on a bit. Zero becomes one, and vice versa. However, unlike its conventional cousin the QPU NOT operation also works on a qubit in superposition.

In circle notation this results, very simply, in the swapping of the |0⟩ and |1⟩ circles, as in Figure 2-8.

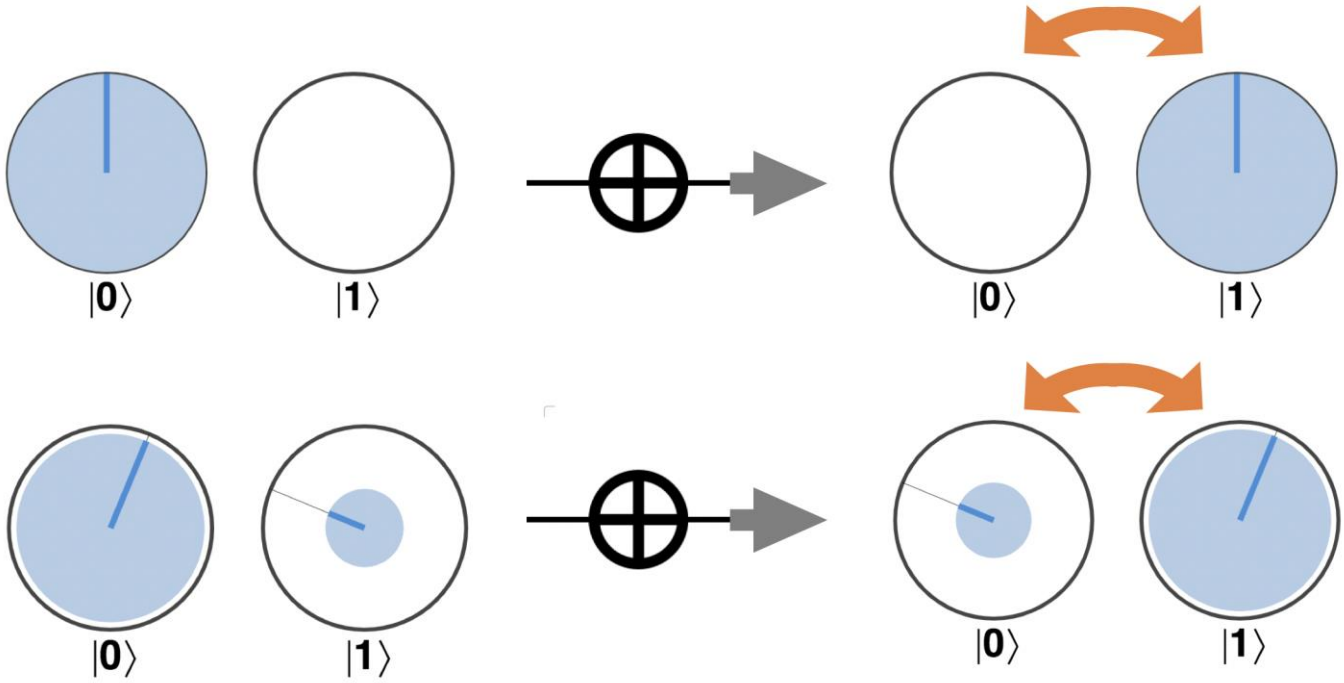


Figure 2-8. The NOT operator in circle notation

Reversibility: Just as in digital logic, the NOT operation is its own inverse; applying it twice returns the qubit to its original value.

QPU Instruction: HAD

The HAD operation (short for Hadamard) is the first QPU instruction we will encounter that cannot be performed on a digital bit.

HAD essentially creates an equal superposition when presented with either a |0⟩ or |1⟩ state. This operation is our gateway drug into using the bizarre and delicate parallelism of quantum superposition!

In circle notation, this results in the output qubit having the same amount of area filled-in for both |0⟩ and |1⟩, as in Figure 2-9.

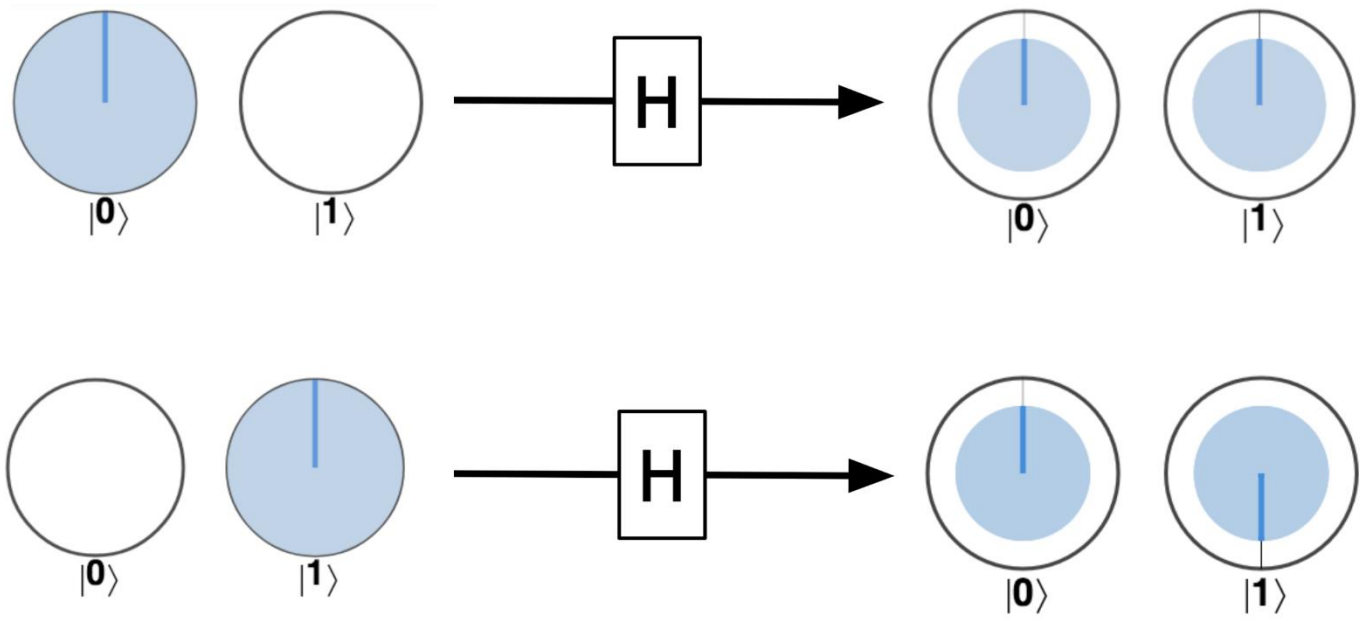


Figure 2-9. Hadamard applied to some basic states

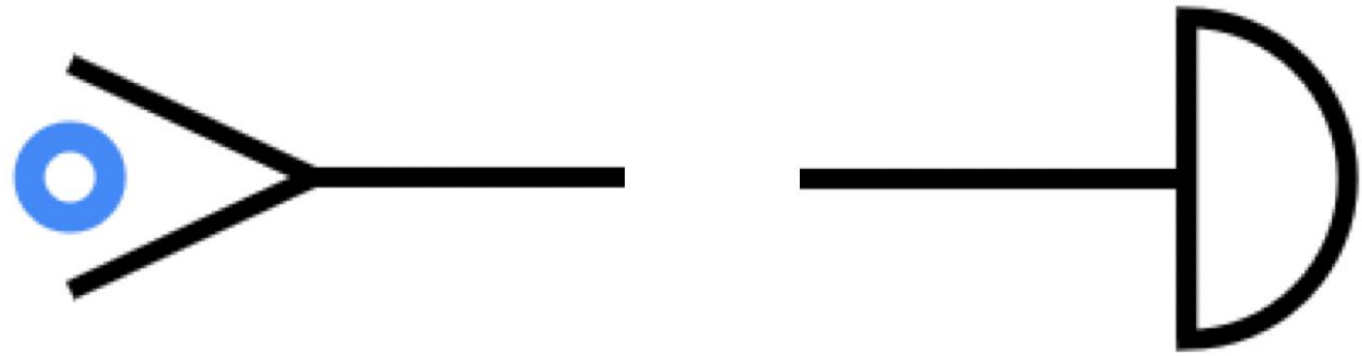
This allows HAD to often be used to produce *uniform superpositions* of outcomes in a qubit - i.e. a superposition where each outcome is equally likely. Notice also that the action on qubits initially in the states |0⟩ and |1⟩ is slightly different - the output of acting HAD on |1⟩ yields a non-zero rotation (relative phase) of one of the circles, whereas the output from acting it on |0⟩ doesn't.

We can also apply HAD to qubits which are *already in a superposition*. In this case the |0⟩ value of the output is the *sum* of the amplitudes from the two input states, and the |1⟩ value of the output is the *difference* between these amplitudes - each part being weighted by the original superpositions amplitudes (note that amplitudes are not the same as magnitudes).

We won't worry about being able to reproduce this example with the (complex) mathematics of superposition amplitudes. In practice we'll rely on a simulator such as QCEngine to determine the actions of HAD for us in such situations.

Reversibility: Similar to NOT, the HAD operation is its own inverse; applying it twice returns the qubit to its original value.

QPU Instruction: READ and WRITE



The READ operation is the formal expression of the *readout* process on a qubit that we've talked about previously. READ is unique in being the only part of a QPU's instruction set that potentially returns a *random* result.

READ will return a value of either 0 or 1 with a probability determined by the magnitude associated with each outcome in a qubits state (ignoring the relative phase). Following a READ operation a qubit is left in a state |0⟩ if the 0 outcome is obtained and state |1⟩ if the 1 outcome is obtained. In other words, any superposition is irreversibly destroyed.

In circle notation an outcome occurs with a probability determined by the filled area in each associated circle. We then shift the filled in area between circles to reflect this result - the circle associated with the occurring outcome becomes entirely filled-in, whilst the remaining circle becomes empty. This is illustrated in Figure 2-10 for READ being performed on two different example superpositions.

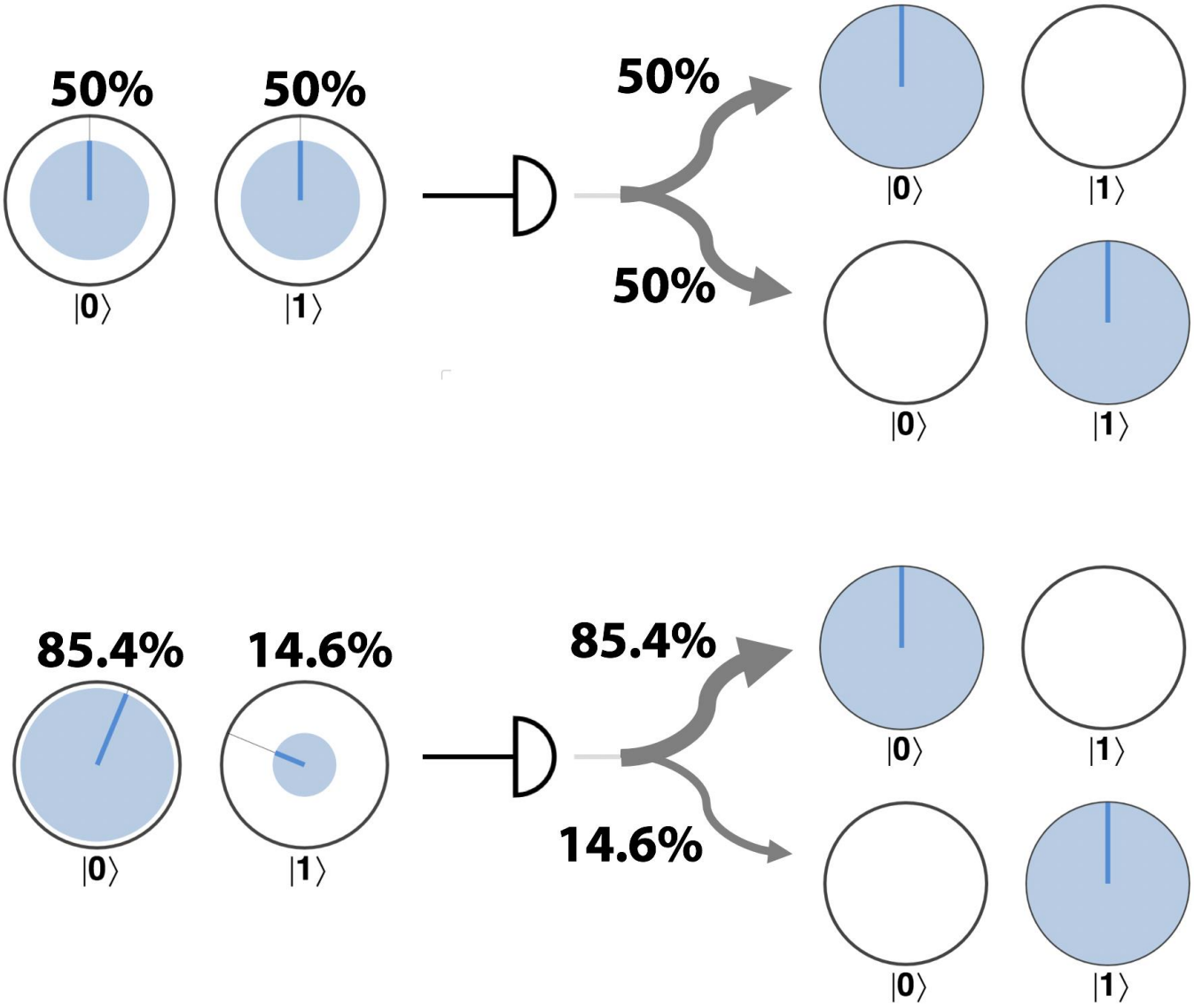


Figure 2-10. The READ operator produces random results

WHAT HAPPENED TO THE PHASE?

In the second example of Figure 2-10, the READ operation removes all meaningful amplitude and phase information. As a result, we are free to draw the remaining circle at any orientation.

Using READ, we can also construct a simple WRITE operation that allows us to prepare a qubit in a desired state of either |0⟩ or |1⟩. This may be performed by combining READ and NOT. First we READ the qubit, and then if the value does not match the value we plan to WRITE, we perform a NOT operation. On some QPU hardware, qubits are initialized to |0⟩, so simply applying a NOT will set the value to |1⟩. Note that this WRITE operation does *not* allow us to prepare a qubit in an arbitrary superposition (with arbitrary magnitude and relative phase), but only in either state |0⟩ or state |1⟩[6]

Reversibility: The READ and WRITE ops are ***not reversible***. They cause the quantum state to collapse, and lose information. Once that is done, the analog values of the qubit (both amplitude and phase) are gone forever.

Hands-on: A perfectly random bit

Before moving on to introduce a few more single qubit operations, let's pause to see how - armed with the HAD, READ and WRITE operations - we can already perform a task that is impossible on any conventional computer. We will generate a truly random bit.

Throughout the history of computation, a vast amount of time and effort has gone into developing *PRNG* (Pseudo-Random Number Generator) systems, which find usage in applications ranging from cryptography to weather forecasting. PRNGs are *pseudo* in the sense that if you know the contents of the computer's memory and the PRNG algorithm, you can predict the next number in the sequence.

According to the known laws of physics the readout behavior of a qubit in superposition is fundamentally and perfectly unpredictable. This allows a QPU to create the worlds greatest random number generator by simply preparing a qubit in state |0⟩, applying the HAD instruction, and then reading out the qubit. We can illustrate this combination of QPU operations using a *quantum circuit*diagram, where a line moving left to right illustrates the sequence of different operations that are performed on our (single) qubit.

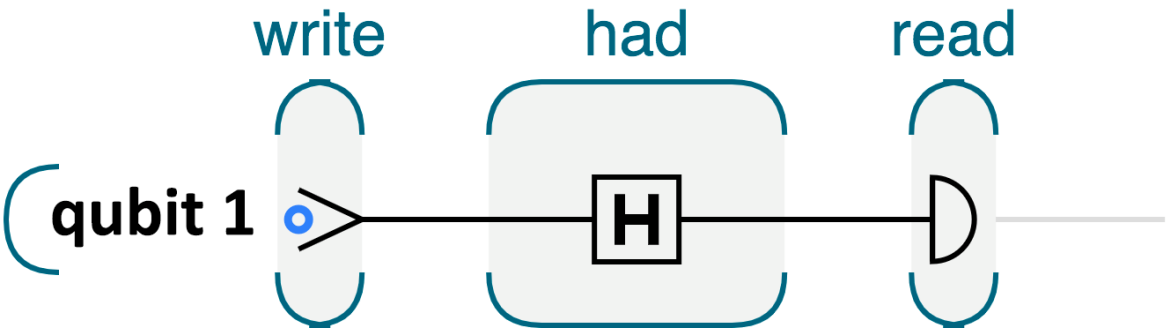


Figure 2-11. Generating a perfectly random bit with a QPU

It might not look like much, but there we have it, our first quantum application - a Quantum Random Number Generator (QRNG)! We can simulate this using the code snippet in Example 2-1. If you repeatedly run these four lines of code on the QCEngine simulator, you'll receive a binary

random string. Of course CPU-powered simulators like QCEngine are approximating our QRNG with a PRNG, but running the equivalent code on a real QPU will produce a perfectly random binary string.

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=2-1*

***Example 2-1.** One random bit*

```
qc.reset(1);          // allocate one qubit
qc.write(0);          // write the value zero
qc.had();             // place it into superposition of 0 and 1
var result = qc.read();  // read the result as a digital bit
```

ABOUT THESE CODE SAMPLES

All of the code samples in this book can be found online at *http://oreilly-qc.github.io*, and may be run either on QPU simulators, or else on actual QPU hardware. Running these samples is an essential part of learning to program a QPU! For more information on running them, see the notes online, or Chapter 1.

Since it might be your first quantum program (congratulations!), let's break it down just to be sure each step makes sense:

- `qc.reset(1)` sets up our simulation of the QPU, requesting one qubit. All the programs we write for QCEngine will initialize a set of qubits with a line like this.
- `qc.write(0)` simply initializes our single qubit in the |0⟩ state – the equivalent of a conventional bit being set to the value 0.
- `qc.had()` applies HAD to our qubit, placing it into a superposition of |0⟩ and |1⟩, just as in Figure 2-9.
- `var result = qc.read()` reads out the value of our qubit at the end of the computation, as a random digital bit - assigning the value to the `result` variable.

Since HAD leaves our qubit with its weighting equally spread over |0⟩ and |1⟩, then – pragmatically - after applying HAD to a qubit and reading it out we receive either a 0 or 1 with precisely 50% probability – i.e. we randomly read out a 0 or 1.

It might look like all we've really done here is find a very expensive way of flipping a coin, but this underestimates the power of HAD. If you could somehow *look inside* HAD you would find neither a pseudo nor hardware random generator. Unlike these, HAD is guaranteed unpredictable by the laws of quantum physics. Nobody in the known universe can do any better than a hopeless random guess as to whether a qubit's value following a HAD will be read to be 0 or 1 – even if they know exactly the instructions we are using to generate our random numbers.

So, a QPU's instruction set contains a fundamental single instruction that can produce truly random bits for us - straight out of the box!

In fact, although we'll properly introduce dealing with multiple qubits in the next chapter, we can easily see, how by running our program for a single random qubit in parallel eight times, we can produce a random *qubyte*. Example 2-2 shows what this looks like.

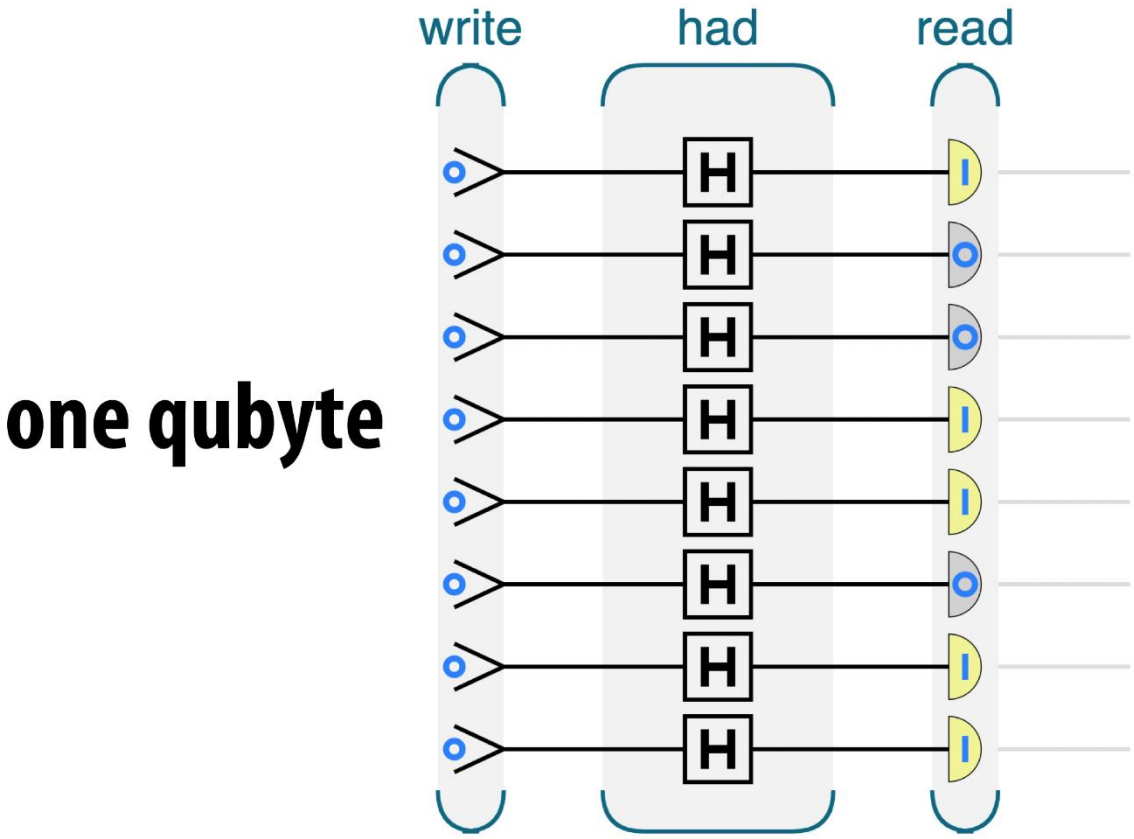


Figure 2-12. One random byte

This code in Example 2-2 for creating a random qubyte is of course almost identical to Example 2-1, just expanded to eight qubits:

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=2-2*

***Example 2-2.** One random byte*

```
qc.reset(8);
qc.write(0);
qc.had();
result = qc.read();
qc.print(result);
```

Note that above we make use of the fact that in QCEngine operations like WRITE and HAD default to being applied to all qubits that we have initialized, unless we explicitly pass specific qubits for them to act on.

EIGHT SEPARATE QUBITS

Although Example 2-2 uses multiple qubits, there are no actual multi-qubit operations that take more than on of the qubits as input. The same program could be run on a simple 1-qubit processor, by generating random bits one at a time.

QPU Instruction: PHASE(θ)

The PHASE(θ) operator, like HAD, has no classical-bit equivalent. This instruction allows us to directly manipulate the *relative phase* of a qubit - changing it by some specified angle. Consequently, as well as a qubit to operate on, the PHASE(θ) operation also takes an additional (conventional) parameter as an input - the angle to rotate by. For example, PHASE(45), denotes a PHASE operation that performs a 45 degree rotation.

In circle notation the effect of PHASE(θ) is to simply rotate the circle associated with |1⟩ by the angle we specify. This is shown in Figure 2-13 for the case of PHASE(45).

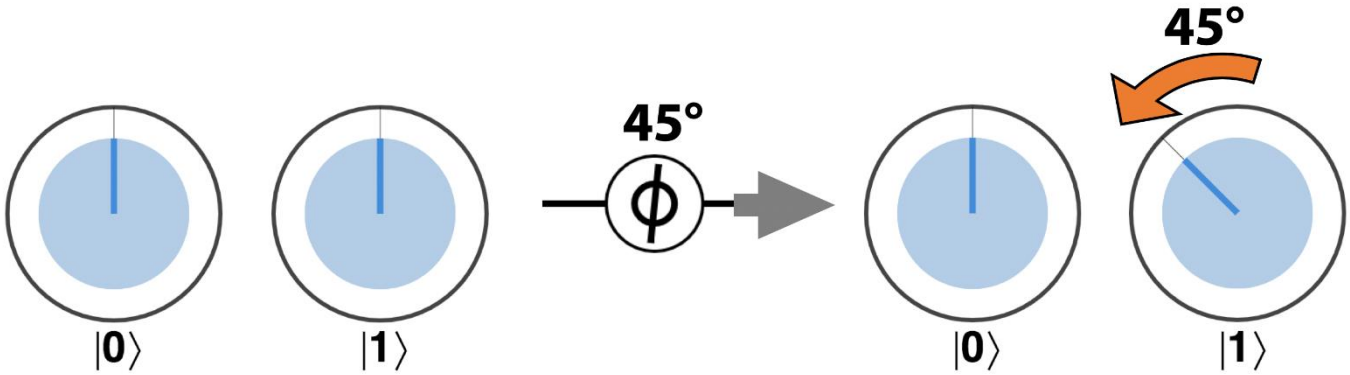


Figure 2-13. Operation of a PHASE gate

Note that the PHASE operation only rotates the circle associated with the |1⟩ state, and has no effect on a qubit in the |0⟩ state.

Reversibility: PHASE operations are reversible, although they are not generally their own inverse. The PHASE operation may be reversed by applying a PHASE with the *negative* of the original angle. In circle notation, this corresponds to *undoing* the rotation, by rotating in the opposite direction.

Using HAD and PHASE, we can produce some single-qubit quantum states which are so commonly used that they've been named: |+⟩, |-⟩, |+Y⟩, and |-Y⟩, as seen in Figure 2-14. If you're feeling like flexing your QPU muscles, see whether you can determine how to produce these states using HAD and PHASE operations (each superposition shown has an equal magnitude in each of the |0⟩ and |1⟩ states).

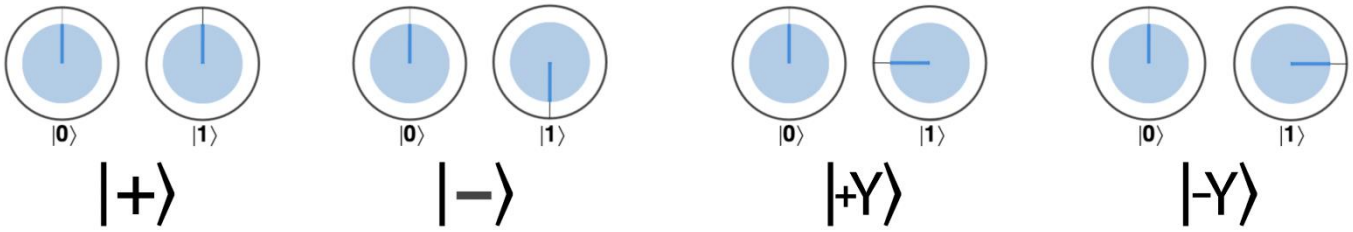


Figure 2-14. Four very commonly used single-qubit states

These four states will be used in Example 2-4, and in fact although they can be produced using HAD and PHASE, we can also understand them as being the result of so-called single-qubit *rotation*operations. These rotations do not correspond to rotations in the circle notation, but to rotations in a different visualization of qubits that is used in the academic literature (if you want to learn about it, you can check chapter [Link to Come]).

One final note on PHASE(θ) - At a glance, the ability of PHASE to rotate a term may seem less powerful than HAD's ability to create superposition and allow a sort of *parallel* computation. This is far from the truth, as a computation that only involves HAD gates and not PHASE can be efficiently simulated on a classical computer. It is only when those gates are combined (together with the two qubit gates that we will see in chapter Chapter 3) that the full computational power of the quantum computer can be attained.

QPU Instructions: ROTX(θ) and ROTY(θ) rotation operations

We've seen that PHASE rotates the relative phase of a qubit - and that in circle notation this corresponds to rotating the circle associated with the |1⟩ value. There are two other common operations related to PHASE called ROTX(θ) and ROTY(θ), which also perform kinds of rotations on our qubit.

Here's what the application of ROTX(45) and ROTY(45) looks like on the |0⟩ and |1⟩ states in our circle notation:

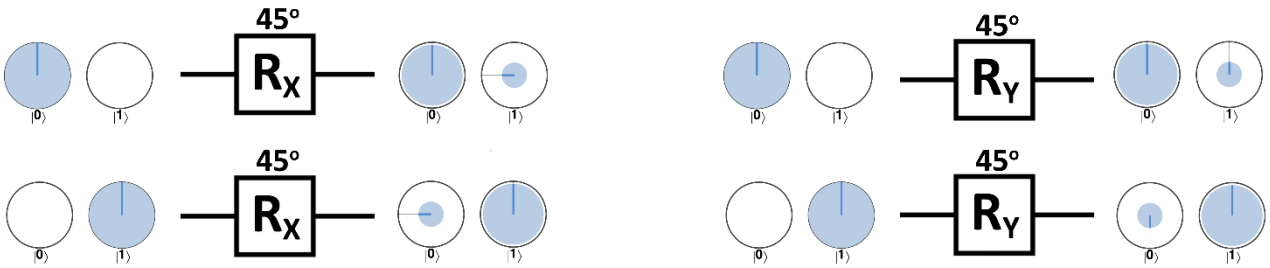


Figure 2-15. ROTX and ROTY actions on 0 and 1 input states.

In our circle notation these operations don't look like very intuitive *rotations*, at least not as obviously as PHASE did. However, their rotation names stem from their action in another common visual representation of a single qubits state, known as the *Bloch sphere*. In the Bloch sphere representation, a qubit is visualized by a point somewhere on the surface of three-dimensional

sphere. We use circle notation instead of the Bloch sphere visualization in this book as the Bloch sphere doesn't scale up well when dealing with multiple qubits. But to satisfy any etymological curiosity, if we were to represent a qubit on the Bloch sphere then ROTY and ROTX operations would correspond to rotating the point representing the qubit about the sphere's Y and X axes respectively[7] - although this meaning is lost in our circle notation, since we use two 2D circles rather than a three dimensional sphere. In fact, the PHASE operation actually corresponds to a rotation about the Z axis when visualizing qubits in the Bloch sphere, and so you may also here it referred to as ROTZ.

2D ROTATIONS ON A 4D OBJECT

If you *do* want to try visualizing ROTX and ROTY as *rotations* similar to PHASE (also called ROTZ), consider this: In circle notation, each of our 2D circles for |0⟩ and |1⟩ has two axes. PHASE performs a 2D rotation using the two axes in the |1⟩ circle, while ROTX and ROTY do the same thing, but using one axis from each circle.

You may have noticed that the states in Figure 2-15 are precisely those we highlighted in Figure 2-14. In fact the states |+⟩, |-⟩, |+Y⟩, and |-Y⟩ are precisely what we obtain by *rotating* our the states |0⟩ and |1⟩ with ROTX and ROTY through 90 degrees.

The missing operation

Looking at all the single qubit operations we have presented so far, it might look like the gateset for quantum computation is simply an advanced package of the gateset available to conventional computer. This is not true, as there is one operation that not only quantum computers cannot do better, they cannot do at all: COPY. A strange consequence from the nature of the physical laws that govern quantum systems, the *no-cloning theorem* states that it is impossible to replicate an *unknown* quantum state. Of course through repeated preparation, we can make many copies of a *known* state, but there is no way of copying a state without first determining what that state is.

This is a bit of an inconvenience at first, but as we will learn in the following chapters, the possibilities available with the new gates make up for loosing COPY.

Combining QPU operations

We now have NOT, HAD, PHASE, READ and WRITE at our disposal. It's worth mentioning that, just like digital logic, combining these operations can allow us to find alternative ways to realize operations, and even allow us to create entirely new ones. For example, suppose your QPU has the HAD and PHASE instructions, but the NOT is missing. A PHASE(180) gate can be combined with two HADs to produce the exact equivalent of a NOT gate, as shown in Figure 2-16

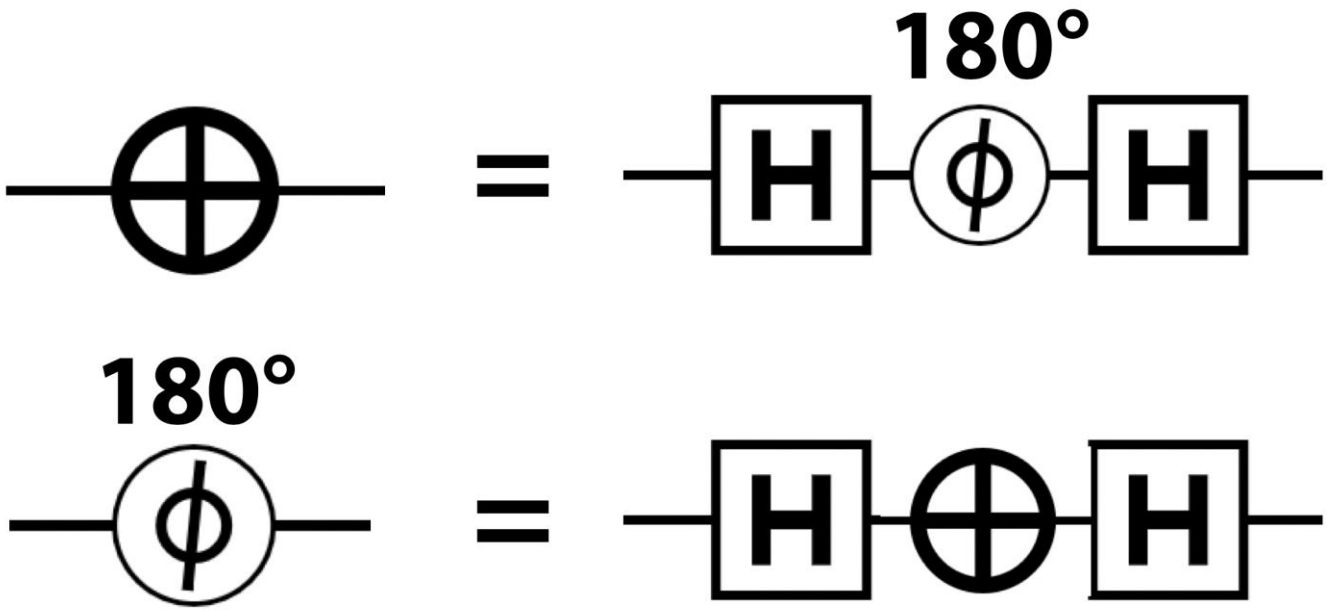


Figure 2-16. Building equivalent gates

See whether you can program the above combination of operations in QCEngine and setup through the resulting circle notation to see how HAD, followed by a PHASE(180) and another HAD results in the same action as a simple NOT.

WARNING

We use degrees to represent angles throughout this book, but if you ever deal with the mathematics of quantum computing you'll more often see angles represented in radians, which divide the interval between 0 and 2π to cover 360°. It's easy enough to convert between the two, and anywhere you see a value in degrees you can replace it with the value in radians, by multiplying by π/180.

QPU Combo Instruction: ROOT-of-NOT

Combining gates also lets us produce interesting new gates which do not exist at all in the world of digital logic. The ROOT-of-NOT gate (RNOT) is a great example. It's quite literally the square-root of the NOT gate, in the sense that, when applied twice, it performs a single NOT operation:

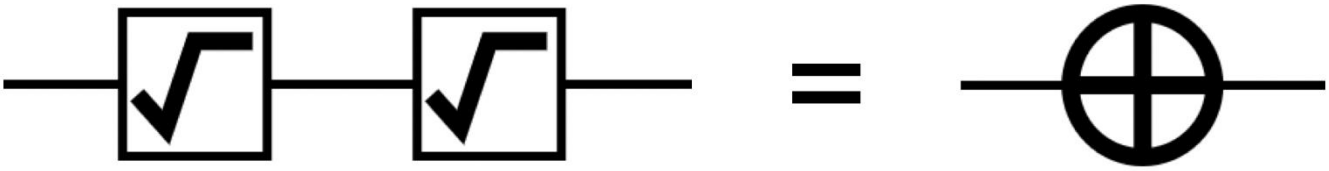


Figure 2-17. An impossible operation for classical bits

There's more than one way to construct this gate, but here's one simple implementation:

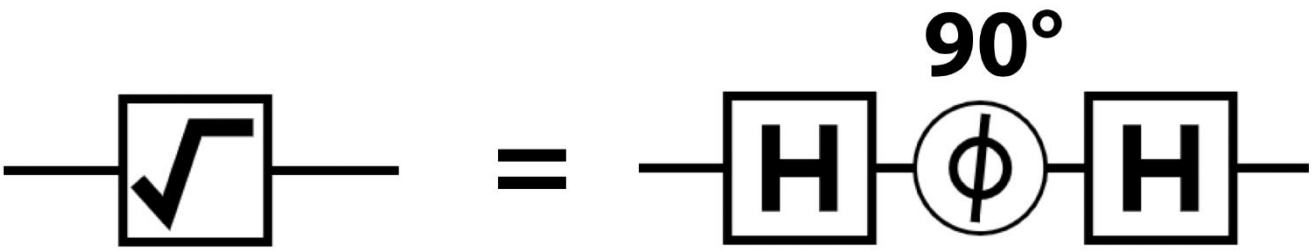


Figure 2-18. Recipe for root-of-not

We can check that applying this set of gates twice does indeed yield the same result as a NOT by running it in our simulator:

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=2-3*

***Example 2-3.** Showing the action of RNOT*

```
qc.reset(1);
qc.write(0);
qc.codeLabel("RNOT")
qc.had();
qc.phase(90);
qc.had();
qc.codeLabel();
qc.nop()
qc.codeLabel("RNOT")
qc.had();
qc.phase(90);
qc.had();
```

NOTE

`qc.codelabel("op1")` is a QCEngine function that allowing us to tag blocks of code with the label specified inside (`op1` in this case). The labels show up in QCEngines quantum circuit visualizer and can be useful for making sense of large quantum circuits. Any code following a `qc.codelabel("op1")` call will be assigned the label we pass to the function. To end the block of code that is tagged, sinply use `qc.codelabel()`.

In circle notation we can visualize each step involved in implementing an RNOT operation (a PHASE(90) between two HADs):

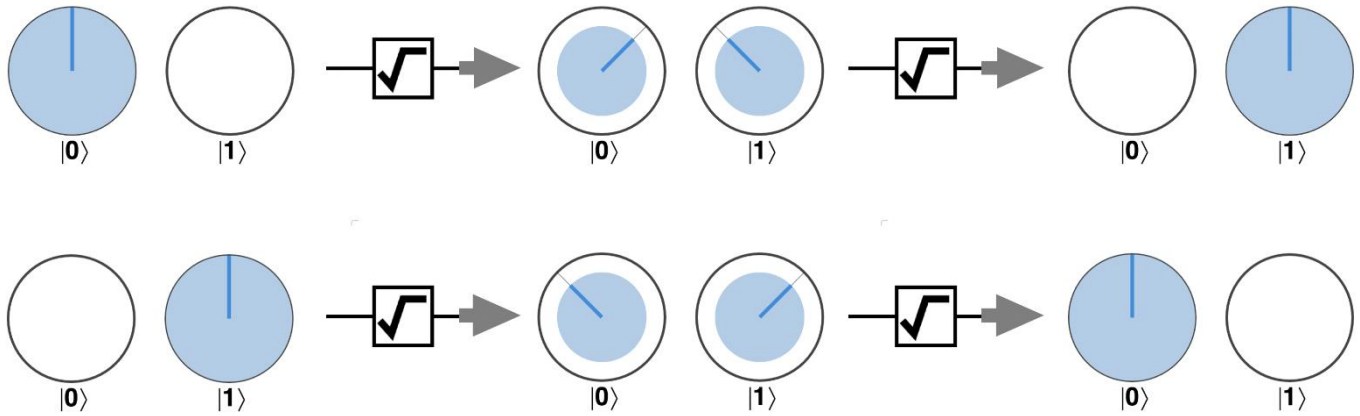


Figure 2-19. Operation of the ROOT-of-NOT gate

Following the evolution of our qubit in circle notation helps us see how RNOT is able to get us halfway to a NOT operation. Recall from Figure 2-16 that if we HAD a qubit, then rotate its relative phase by *180* degrees, another HAD would result in a NOT operation. RNOT performs half of this rotation (a PHASE(90)), so that two applications will result in the HAD-PHASE(180)-HAD sequence that is equivalent to a NOT. It might be a bit mind-bending at first, but see if you can piece together how the RNOT gate cleverly performs this feat when applied twice (it might help to remember that HAD is it's own inverse, so a sequence of two HADs does nothing at all).

Reversibility: While RNOT operations are never their own inverse, the inverse of the operation in Figure 2-18 may be constructed by using the negative phase value, as shown in Figure 2-20.

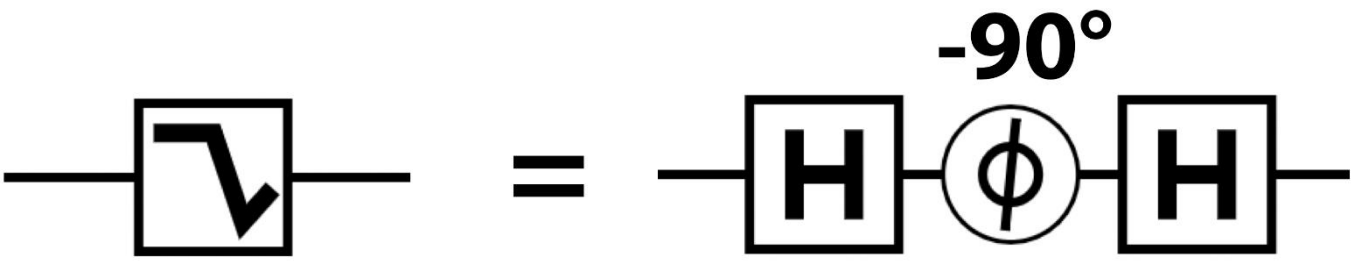


Figure 2-20. Inverse of root-of-not

Although it might seem like an esoteric curiosity, the RNOT gate teaches us the important lesson that by carefully placing information in the relative phase of a qubit we can perform entirely new kinds of computation.

Hands-on: Quantum Spy Hunter

To give a more practical demonstration of the power in manipulating the relative phases of qubits, let's finish this chapter with a more complex program. One that uses some of the simple single-qubit QPU operations that we've learned so far to detect a spy. This technique is a simplified version of *QKD*(*quantum key distribution*) which is the core of so-called quantum encryption. The

main feature of quantum (as opposed to conventional) key distribution, is that we have the ability to unambiguously detect when a key has been compromised.

We suppose that two QPU programmers, Alice and Bob, are sending data to each other via a communication channel capable of transmitting qubits (turns out that a fiber-optic link will do the job). Once in a while, they send the specially constructed "spy hunter" qubit shown in this example, which they use to test whether their communication channel has been compromised.

Any spy who tries to read one of these qubits has a 25% chance of getting caught. So even if Alice and Bob only use 50 of them in the whole transfer, the spy's chances of getting away are far less than one in a million.

Alice and Bob can then detect whether their key has been compromised by exchanging some digital (non-quantum) information, which does not need to be private or encrypted. After exchanging their message they test a few of their qubits by reading them out and checking that they agree with the digital in a certain expected way. If any disagree, then they know someone was listening in.

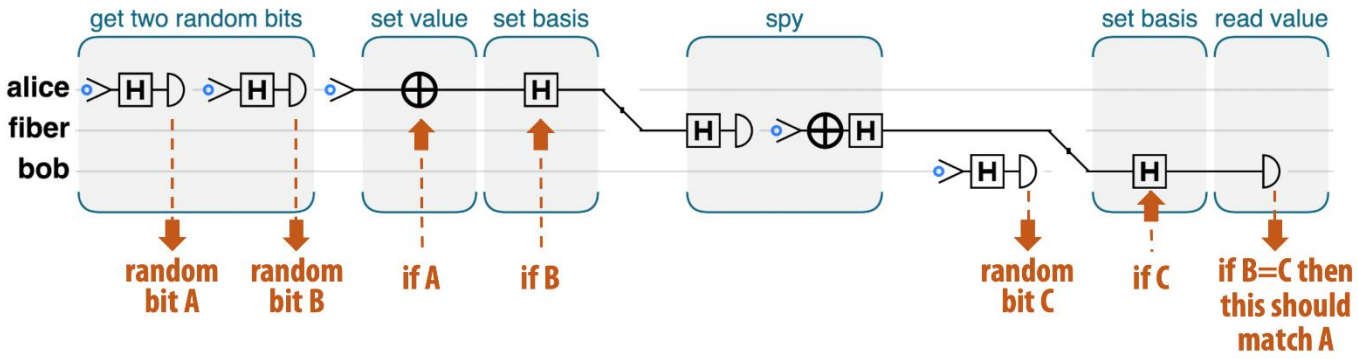


Figure 2-21. Quantum spy hunter

Here's the code. We strongly recommend trying out Example 2-4 on your own, by pasting the code into the QCEngine workbench link, and tweaking and testing like you would with any other code snippet.

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=2-4*

Example 2-4. *Quantum Random Spy Hunter*

```
qc.reset(3);
qc.discard();
a = qint.new(1, 'alice');
fiber = qint.new(1, 'fiber');
b = qint.new(1, 'bob');

function random_bit(q) {
    q.write(0);
    q.had();
    return q.read();
}
```

```
// Generate two random bits
send_basis = random_bit(a);
send_val = random_bit(a);

// Prepare Alice's qubit
a.write(send_val);
if (send_basis)
    a.had();

// Send the qubit!
fiber.exchange(a);

// Activate the spy
var spy_is_present = false;
if (spy_is_present) {
    var spy_basis = 0;
    if (spy_basis)
        fiber.had();
    stolen_data = fiber.read();
    fiber.write(stolen_data);
    if (spy_basis)
        fiber.had();
}

// Receive the qubit!
recv_basis = random_bit(b);
fiber.exchange(b);
if (recv_basis)
    b.had();
recv_val = b.read();

// Now Alice emails Bob to tell
// him her basis and value.
// If the basis matches and the
// value does not, there's a spy!
if (send_basis == recv_basis)
    if (send_val != recv_val)
        qc.print('Caught a spy!\n');
```

In Example 2-4, Alice and Bob each have access to a simple QPU containing a single qubit, and can send their qubits along a fiber-optic link. There might be a spy listening to that link; in the sample code you can control whether or not a spy is present by toggling the `spy_is_present` variable.

The fact that quantum cryptography can be performed with such relatively small QPU's is one of the reasons why it has begun to see commercial application far before more powerful general purpose QPU's are available.

Let's walk through the code one step at a time to see how Alice and Bob's simple QPU's allow them to perform this feat. We'll use the comments from the code-snippet as markers:

`// Generate two random bits`

> Alice uses her one-qubit QPU as a simple QRNG, exactly as we did in Example 2-2, to generate two secret random bits known only to her, which we label as `basis` and `value`.

`// Prepare Alice's qubit`

Using her two random bits, Alice prepares the "spy hunter" qubit. She sets it to `value`, and then uses `basis` to decide whether to apply a HAD. In effect, she is preparing her qubit randomly into one of the states |0⟩, |1⟩, |+⟩, |-⟩, and not (yet) telling anyone *which* of the states it is. If she does decide to apply a HAD then if Bob wants to extract whether she intended a 0 or 1, he will have to apply the inverse of HAD (another HAD) before performing a READ.

```
// Send the qubit!
```

Alice sends her qubit into the fiber connection, on its way to Bob. For clarity in this example, we are using another qubit to represent the fiber.

```
// Activate the spy
```

If this were **non-quantum** digital data, the spy would simply make a copy of the bit. Mission accomplished! With qubits, that's not possible. Recall that there is no copy operation, so the only thing the spy can do is READ the qubit Alice sent, and then carefully send one just like it to Bob to avoid detection. Remember that reading a qubit destroys information, we are only left with the classical bit of the readout. The spy doesn't know whether or not Alice performed a HAD. As a result he won't know whether to apply a second (*inverting*) HAD before performing his READ. If he simply performs a READ he won't know whether he's receiving a random value from a qubit in superposition or a value actually encoded by Alice. Not only does this mean he won't be able to reliably extract Alice's bit, but he also won't know what the right state is to send on to Bob to avoid detection.

```
// Receive the qubit
```

Like Alice, Bob randomly generates a basis bit, and uses that to decide whether to apply a HAD before applying a READ to Alice's qubit, resulting in his value bit. This means that sometimes Bob will (by chance) correctly decode a binary value from Alice and other times he won't.

```
// If the basis matches and the value does not, there's a spy!
```

Now that the qubit has been received, Alice and Bob can openly compare in which cases their choices of applying HAD's or not correctly matched up. If they randomly happened to both apply (or not apply) a HAD (this will be about half the time) then their value bits should match - i.e. Bob correctly decoded Alice's message. If, in these *correctly decoded messages* their values *don't*agree, then they can conclude that the spy must have READ their message and sent on an *incorrect*replacement qubit to Bob - messing up his decoding.

Conclusion

In this chapter we have introduced a logical model for a single qubit, as well as a variety of QPU instructions used to manipulate it. The quantum random property of the READ operation was used to construct a random number generator, and control over the phase of a qubit was used to detect a spy.

The ability to combine single-qubit operations into new operations is important to understand, and can be critical when using a QPU program on hardware or simulators with differing instruction sets. Similar construction methods will be used to build new operations in the chapters ahead.

The circle notation used to visualize the state of a qubit is also used extensively in the chapters ahead, where it is expanded to display the state of multi-qubit systems. In the next chapter, we will

explore the logical behavior of these multi-qubit systems, and also introduce new QPU operations used to work with them.

1 The differences between superpositions and mixtures of quantum states requires some subtlety, if you are interested in understanding this distinction more deeply, we address it in the [Link to Come].

2 In Appendix ?? we explain in more detail how this pictorial representation maps to the mathematics of quantum computation, allowing the interested reader to port understanding they acquire from circle notation to mathematics used in other literature.

3 A single qubits is represented by two circles, therefore is the relative rotation between these that matter. If we were dealing with a state of 3 qubits, which is represented by 8 circles, it is the relative phase between all eight circles that matters.:

4 That only *relative* phases are of importance stems from the underlying quantum mechanical laws governing qubits

5 Note that the ability of an object to exhibit quantum effects is not necessarily dependent on its size. Generally speaking it's easier to observe quantum phenomena in atomic-scale objects, but single photons and superconducting qubits are two examples that buck this trend.

6 We will see that being able to prepare an arbitrary superposition is useful, especially in quantum machine learning applications, and we'll introduce an approach for this in chapter xx.

7 Some examples of the these rotations in the Bloch sphere are shown in chapter [Link to Come]:

Chapter 3. Multiple Qubits

As useful as qubits are individually, they are much more useful (and more intriguing) in groups. We've already seen in Chapter 2 how the distinctly quantum phenomenon of superposition introduces new parameters that we can use for computation (magnitude and phase). But when our QPU has access to more than one qubit, we can also start to make use of a second powerful quantum phenomenon known as *quantum entanglement*. Quantum entanglement is a very particular kind of powerful interaction between qubits. In this chapter, we'll see entanglement in action, and develop complex and sophisticated ways to utilize it.

But before we can explore some of the abilities groups of multiple qubits have to offer, we'll need ways to visualize them.

Circle notation for multi-qubit registers

How can we extend our circle notation to multiple qubits? If we were only ever interested in qubits that *didn't* interact with one another, then we could simply employ the representation we used for a single qubits, but multiple times - using a pair of circles for the |0⟩ and |1⟩ states of each qubit. Although this naive representation allows us to describe a superposition of any one *individual* qubit, there's are ways we might superpose groups of qubits that it cannot represent (as we'll see below).

So how else could we represent the state of a *register* of multiple qubits? Just as is the case with conventional bits, a register of N qubits can be used to represent one of 2^N different values. For example, a register of three qubits, having qubits in the states |0⟩|1⟩|1⟩ can represent a decimal value of *3*. When talking about multi-qubit registers, we'll often describe the decimal value that the register represents in the same quantum notation that we used for a single qubit. So whereas a single qubit can be in the states |0⟩ and |1⟩, a two-qubit register has possible states |0⟩, |1⟩, |2⟩ and |3⟩. Making use of the quantum nature of our qubits we can also create superpositions of these different values. This being the case, to represent N qubits, let's use a separate circle for each of the 2^N different values that an N-bit number can assume.

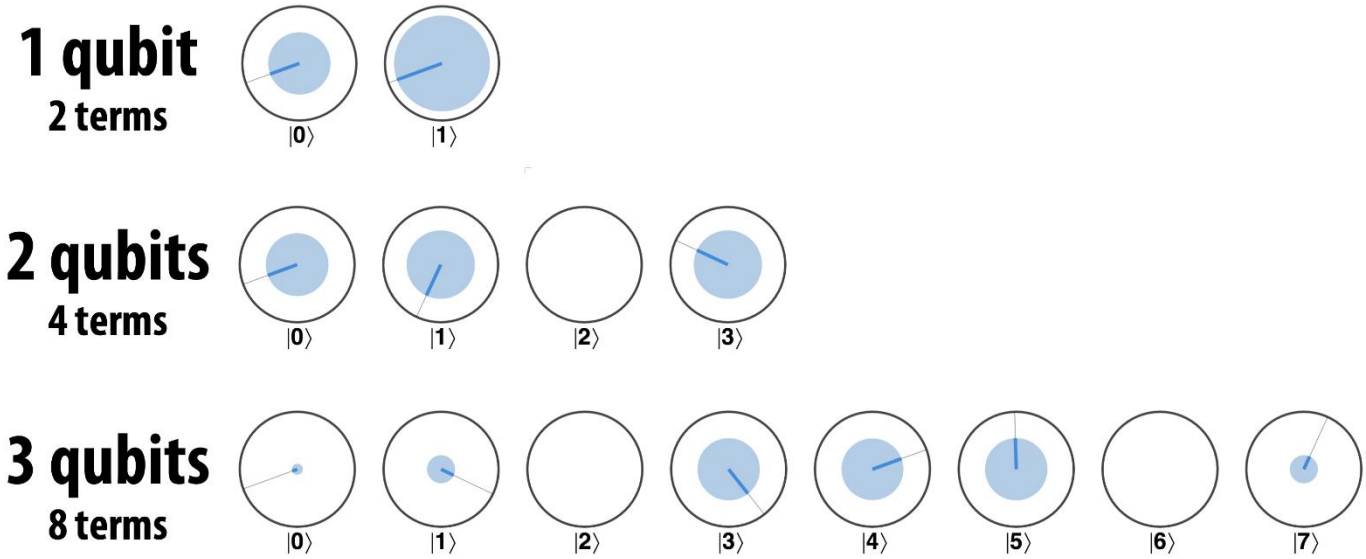


Figure 3-1. Circle notation for various numbers of multiple qubits

In Figure 3-1, we see the familiar 2-circle |0⟩, |1⟩ representation for a single qubit. For two qubits we have circles for |0⟩, |1⟩, |2⟩, |3⟩. This is not "one pair of terms per qubit"; instead, it is one term for each possible two-bit number you may get by reading these qubits. For four qubits, the terms are |0⟩ … |7⟩. As shown above, this means that we can now associate a magnitude and relative phase with each of these 2^N values. In the case of the 3-qubit example, the magnitude of each of the circles represents the probability that a specific 3-bit value (0-7) will be observed when *all three* qubits are read.

You may be wondering what such a superposition of a registers value looks like in terms of the states of the individual qubits making up the register. In some cases we can easily see what the individual qubit states must be. For example, in Figure 3-2 the three-qubit register superposition of the states |0⟩, |2⟩, |4⟩ and |6⟩ can easily be expressed in terms of each individual qubit's state.

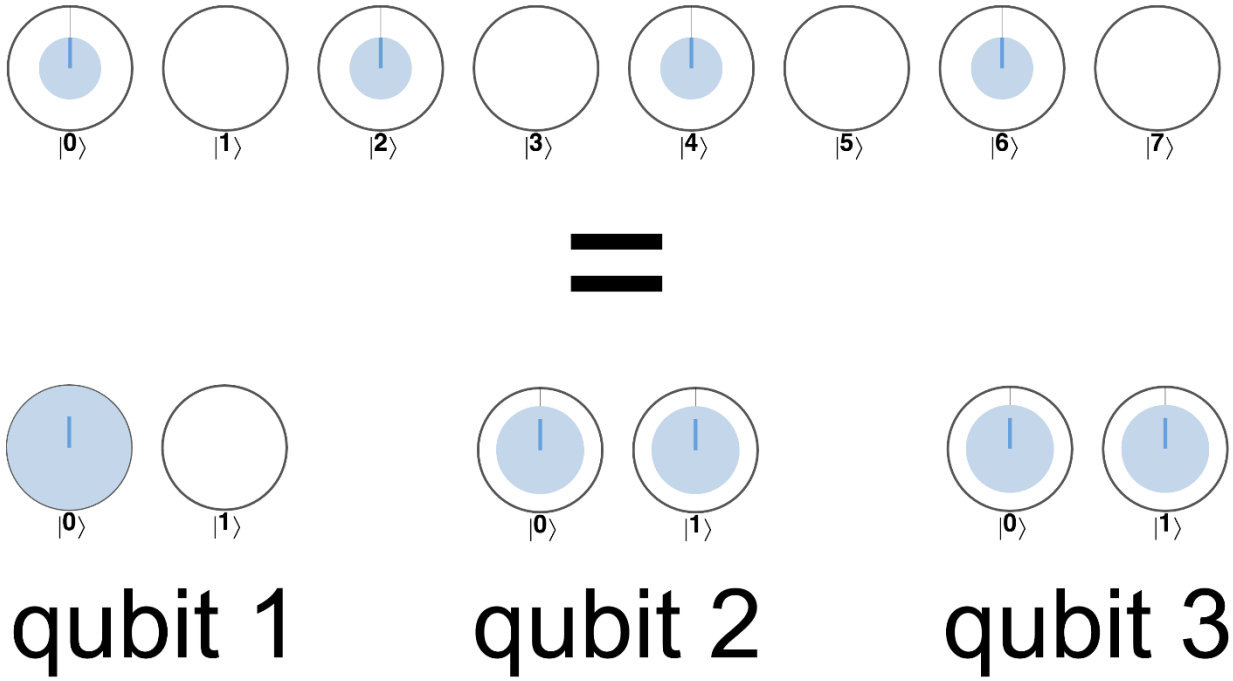


Figure 3-2. Some multi-qubit quantum states can be understood in terms of single qubit states

In fact, this multiqubit state can be generated simply using two single-qubit HAD operations, as shown by Example 3-1.

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=3-1*

***Example 3-1.** Creating a multi-qubit state that can be expressed in terms of its qubits*

```
qc.reset(3);
qc.write(0);
var qubit1 = qint.new(1, 'qubit 1');
var qubit2 = qint.new(1, 'qubit 2');
var qubit3 = qint.new(1, 'qubit 3');
qubit2.had();
qubit3.had();
```

TIP

Example 3-1 introduces some new QCEngine notation. When dealing with multi-qubit registers we'll have a lot of qubits to keep track of. The `qint` object allows us to label our qubits and treat them more like a standard programming variable. Once we've setup our register with some qubits (using `qc.reset`), `qint.new()` allows us to assign them to `qint`s. The first argument to `qint.new()` specifies how many qubits to assign to this `qint` from the stack created by `qc.reset()` (we'll later find it useful to sometimes group sub-registers of qubits). The second argument takes a label that is used in the circuit visualizer. `qint` objects have many methods allowing us to apply QPU operations directly to a given qubit. For example, above we use `qint.had()`.

So we can understand the multi-qubit state in Figure 3-2 in terms of its constituent qubits. But take a look at the state of a three-qubit register shown in Figure 3-3.

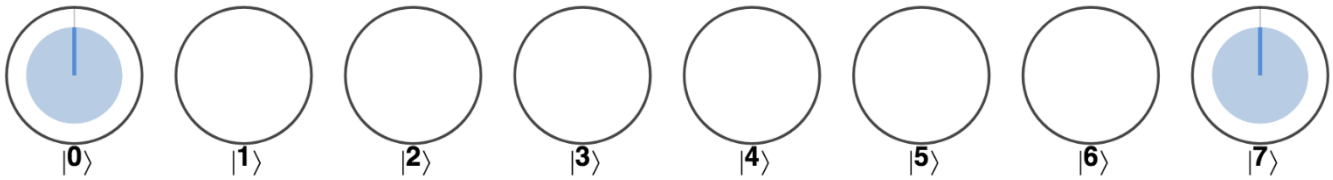


Figure 3-3. Quantum relationships between multiple qubits.

This represents the state of a three qubits in equal superposition of |0⟩ and |7>. Can we visualize this state in terms of what each individual qubit is doing like we did in Figure 3-2? Since 0 and 7 are 000 and 111 in binary, we have a superposition of all three qubits being in the state |0⟩ (|0⟩|0⟩|0⟩) and all three qubits being in state |1⟩ (|1⟩|1⟩|1⟩)). Surprisingly, in this case, there is no way to write down circle representations for the individual qubits! Notice that for this three qubit state a readout of the qubits always results in us finding them to have the *same* values (with 50% probability that value will be 0, and with 50% probability it will be 1). So clearly the three qubits must, in some sense, be linked - to ensure that their outcomes are the same.

This link is the new and powerful *entanglement* phenomenon that we previously mentioned. Entangled states can't be described only in terms of what the individual qubits are doing - although you're welcome to try! The entanglement link is only describable in the configuration of the whole multi-qubit register. It also turns out to be impossible to *produce* entangled states from only *single-qubit operations*. So to explore entanglement in more detail, we'll need to introduce multi-qubit operations, and for that we'll need to be able to deftly apply everything we've learned so far to circuits containing*registers* of many qubits.

HOW MANY BITS PER QUBIT?

In articles about quantum computing, be wary when authors make assertions such as "…12 qubits, the equivalent of 4096 bits…". As you have hopefully discovered already, bits and qubits are different units, and there is no straightforward conversion.

Drawing a Multi-Qubit Register

We now know how to describe the configuration of *N* qubits in circle notation using 2^N circles, but what about drawing multi-qubit quantum circuits? Since our multi-qubit circle notation considers each qubit to take a position in a length *N* bit-string, it will be convenient to label each qubit according to it's binary value.

As an example, let's take another look at the random qubyte circuit we introduced in Chapter 2. There we simply labelled the eight qubits as `qubit 1`, `qubit 2`, etc… But here's how that circuit looks if we properly label each qubit with the binary value it represents. Note that we follow the standard memory addressing practise of using hexadecimal:

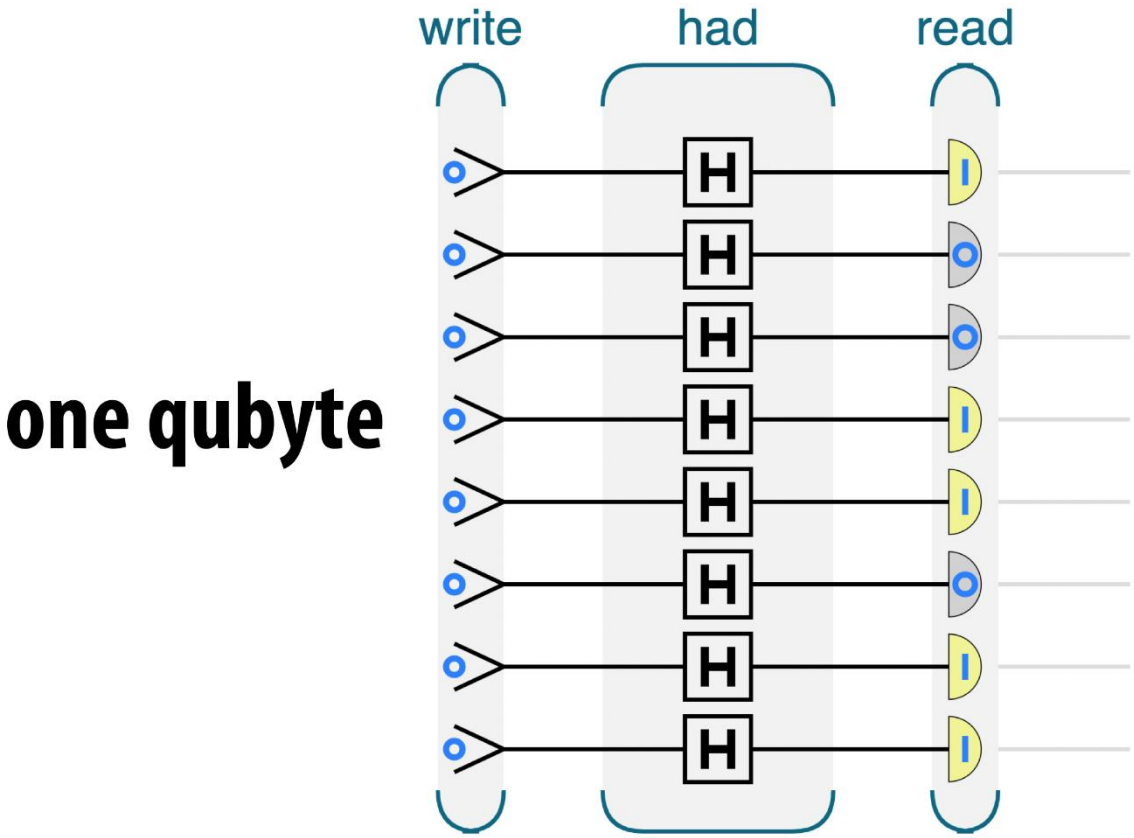


Figure 3-4. One random byte

NAMING QUBITS

Note that in Figure 3-5 we write our qubit values in hexadecimal, using notation such as **0x1** and **0x2**. This is standard programmer notation for hexadecimal values, and we'll use this throughout the book as a very convenient notation to clarify when we're talking about a specific qubit - even in cases when we have large numbers of them.

Single-qubit Operations in Multi-Qubit Registers

Now we know how to draw circuits of multiple qubits, and how to represent them in circle notation, let's start doing something with them. Starting with the operations we've already covered, what happens (in circle notation) when we apply single-qubit operations such as NOT, HAD and PHASE to a multi-qubit register? The operation is almost identical to the single-qubit case. The only difference is that the circles are operated on in certain *operator pairs* specific to the qubit that the operation acts on.

To identify a qubit's operator pairs, match each circle with the one whose value differs by the qubit's bit-value, as shown in Figure 3-5.

Once these operator pairs have been identified, the operation is performed on *each pair*, just as if the members of a pair were the |0⟩ and |1⟩ values of a single-qubit register. For a NOT operation, the circles in each pair are simply swapped as in Figure 3-5.

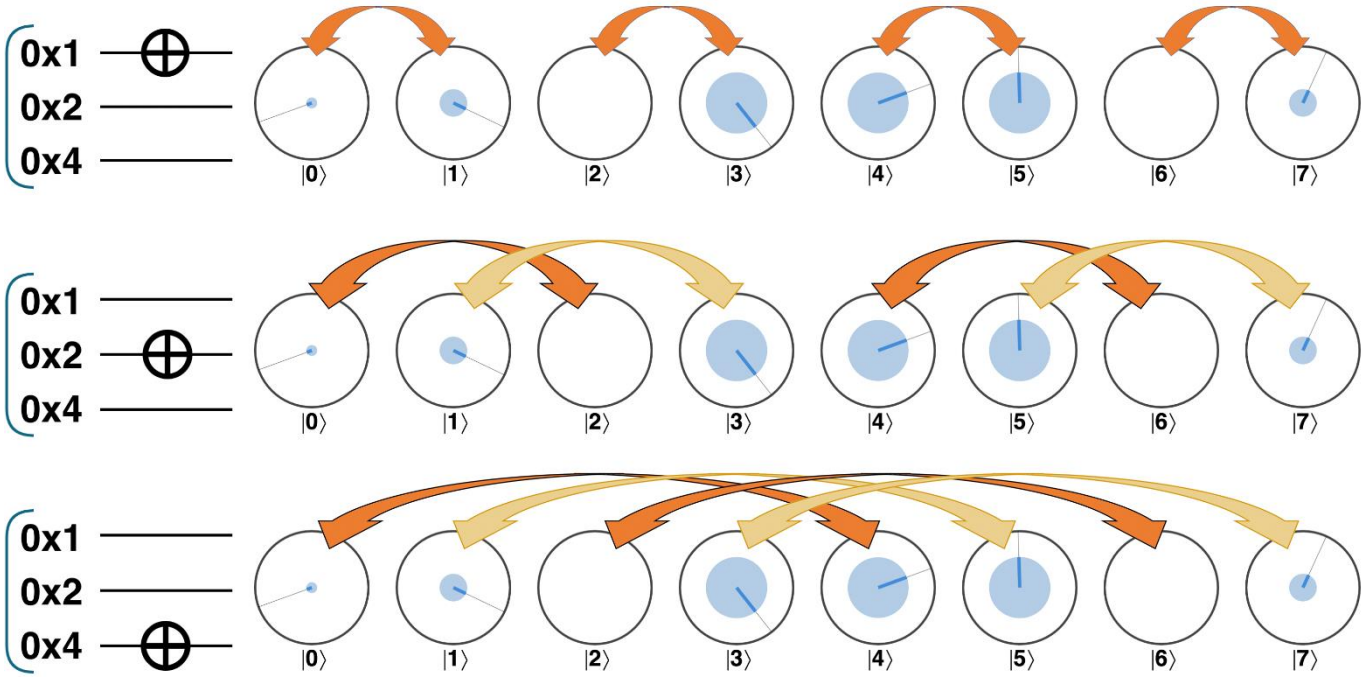


Figure 3-5. The NOT operation swaps terms in each of the qubit's operator pairs.

For a single-qubit PHASE operation, the right-hand circle of each pair is rotated by the phase angle, as in Figure 3-6

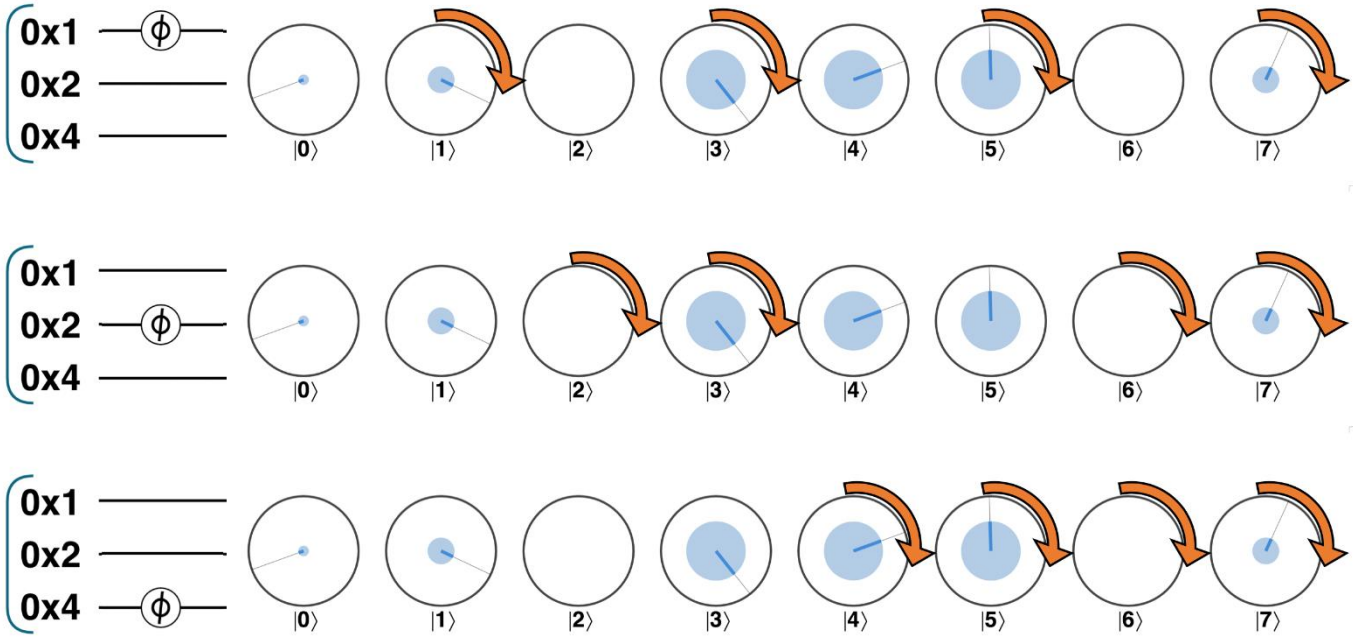


Figure 3-6. Single-qubit phase in a multi-qubit register.

Thinking in terms of operator pairs is a good way to quickly visualize the action of single-qubit operations on a register. For a deeper understanding of why this works we need to think about the effect that an operation on a given qubit has on the binary representation of the whole register. For example, the circle-swapping action of a NOT on the second qubit in Figure 3-5 corresponds to simply flipping the second bit in each term's binary representation. Similarly, a single-qubit PHASE operation on (for example) the third qubit rotates each circle for which the third bit is 1. A single-

qubit PHASE will always cause exactly half of the terms to be rotated, and *which* half just depends on which qubit is the target of the operation.

The same kind of reasoning helps us think about the action of any other single qubit operation on qubits from larger registers.

Reading a qubit in a multi-qubit register

What should we expect when we perform a READ operation on a single qubit from a multi-qubit register? A READ operation also functions using operator pairs. If we have a multi-qubit circle representation then we can determine the probability of obtaining a "0" outcome for one single qubit by adding the magnitudes of all of the circles on the |0⟩ (left-hand) side of that qubits operator pairs. Similarly we can determine the probability of "1" by adding the magnitudes of all of the circles on the |1⟩ (right-hand) side of the qubits operator pairs.

Following a READ, the state of our multi-qubit register will change to reflect which outcome occurred. All circles which do not agree with the result, will be eliminated as shown in Figure 3-7(following this elimination, the state will also have the remaining terms *'renormalized'* so that their magnitudes add up to 100%). Figure 3-7 below illustrates how the READ operation affects a multi-qubit register in circle notation:

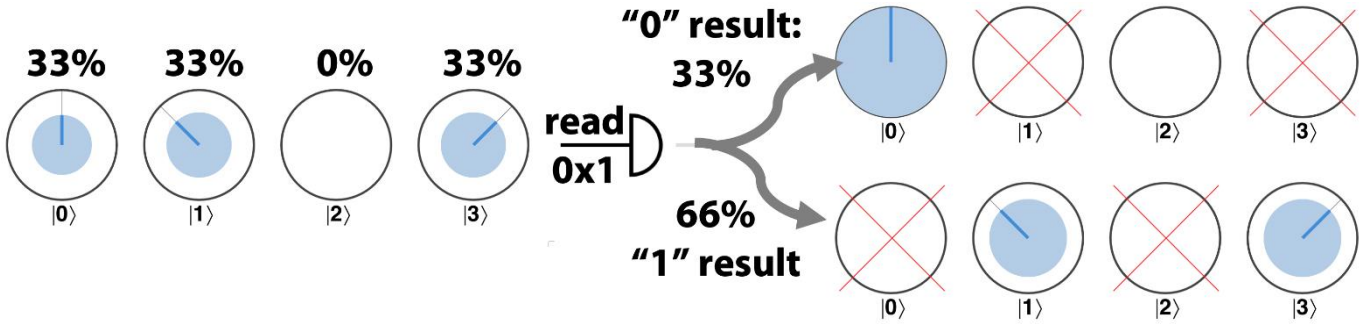


Figure 3-7. Reading one qubit in a multi-qubit register

To read more than one qubit, each single-qubit read operation can be performed individually according to this operator pair prescription.

Visualizing Larger Numbers of Qubits

Since *N* qubits require 2^N terms (circles in circle notation), each additional qubit we might add to our QPU doubles the number of terms (or circles) we need to keep track of. As Figure 3-8 shows, the number of terms quite quickly increases to the point where our circle notation circles become vanishingly small.

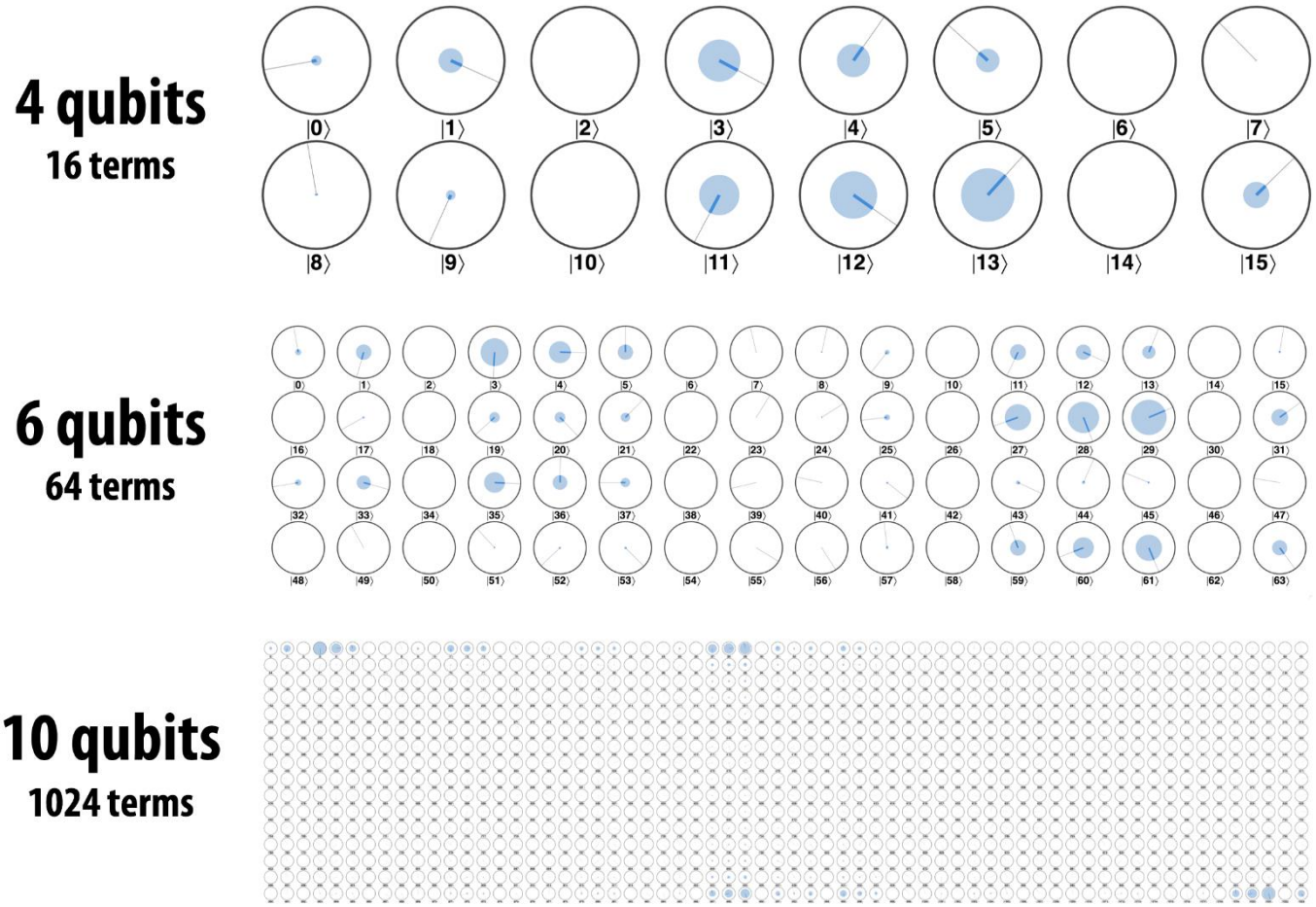


Figure 3-8. Circle notation for larger qubit counts

With so many tiny circles, the circle-notation visualization becomes useful for seeing *patterns* instead of individual values, and we can zoom in to any areas we want a more quantitative view of. Nevertheless, we can improve clarity in these situations by exaggerating the amplitudes, making the phase-needles bold, and also using differences in color or shading to emphasize differences in phase as shown in Figure 3-9.

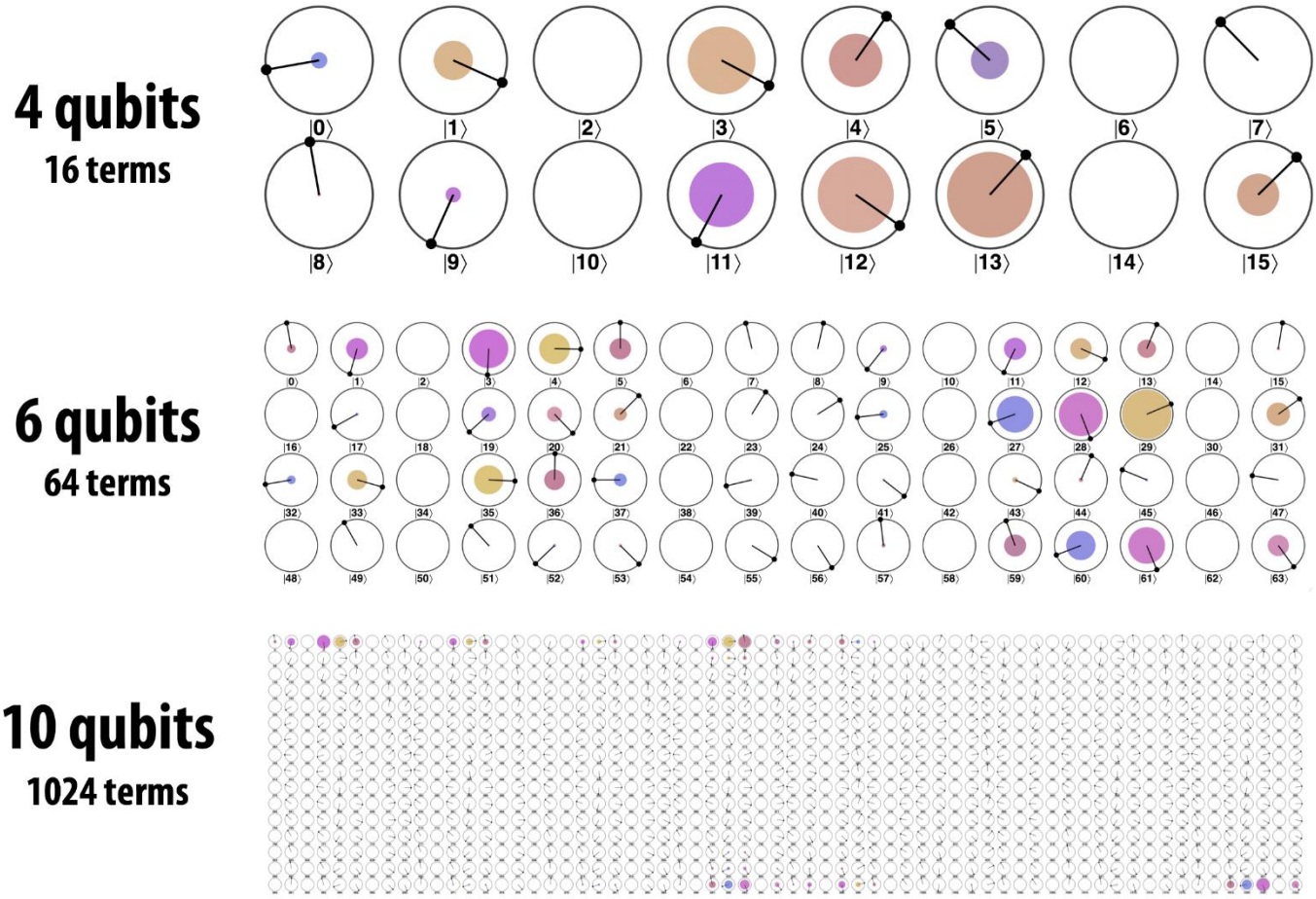


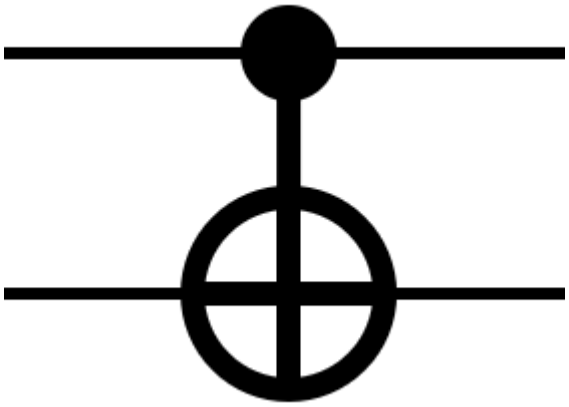
Figure 3-9. Sometimes exaggeration is warranted

In the chapters ahead, we will sometimes make use of these techniques to illustrate aspects of quantum states. But even these eyeball-hacks are only useful to a point; for a 32-qubit system there are 4,294,967,296 circles. Displaying them in a 65,536x65,536 grid of circles is too much information for most displays and eyes alike.

TIP

In your own QCEngine programs you can add the line `qc_options.color_by_phase = true;` to the very beginning to enable the *phase coloring* shown in Figure 3-9. The *bold needles* can also be toggled by including the line `qc_options.book_render = true;`.

QPU Instruction: CNOT



Now we've flexed our multi-qubit muscles implementing our familiar single qubit operations on larger QPU registers, it's time to introduce some definitively multi-qubit QPU operations. By *multi-qubit*operations we mean ones that have mult-qubit inputs, *requiring* more than one qubit to operate. The first of these we'll consider is the powerful CNOT operation. CNOT works on *two* qubits and can be thought of as an "if" programming construct with the following condition: "Apply the NOT operation normally to a target qubit, but *only* if a condition qubit has the value 1." The circuit symbol used for CNOT shows this logic by connecting two qubits with a line. A filled dot represents the control qubit whilst a NOT symbol show the target qubit to be conditionally operated on.

The idea of using *condition* qubits to selectively apply actions is used in many other QPU operations, but CNOT is perhaps the most basic example. Figure 3-10 illustrates the difference between applying a NOT operation to a qubit within a register versus applying CNOT (conditioned on some other *control*qubit).

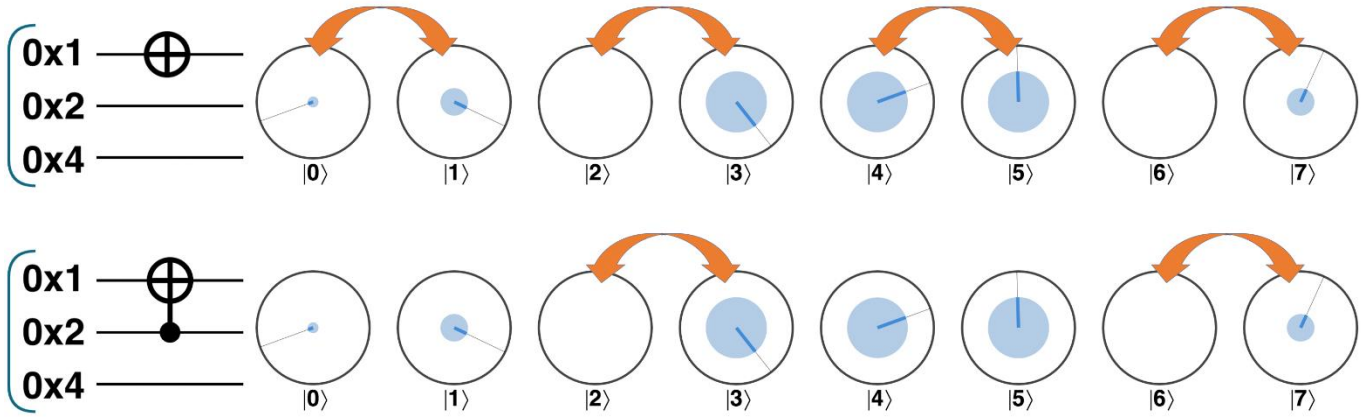


Figure 3-10. CNOT in operation

The orange arrows show which operator pairs have their circles swapped in circle notation. We can see that the essential operation of CNOT is the same as that of NOT, only more selective - applying the NOT operation only to terms whose binary representations (in this example) have a 1 in the second bit (eg: 2=010, 3=011, 6=110 and 7=111).

Reversibility: Like the NOT operation, CNOT is its own inverse; applying the CNOT operation twice will return the multi-qubit register state that we began with.

On its own there's nothing particularly quantum about CNOT (conditional logic is of course a fundamental feature of conventional CPU's). But armed with the conditional logic of the CNOT we can now ask an interesting and distinctly quantum question. What would happen if the control qubit of a CNOT operation is in a superposition? We can simulate this situation for a 2 qubit register using the circuit in Figure 3-11.

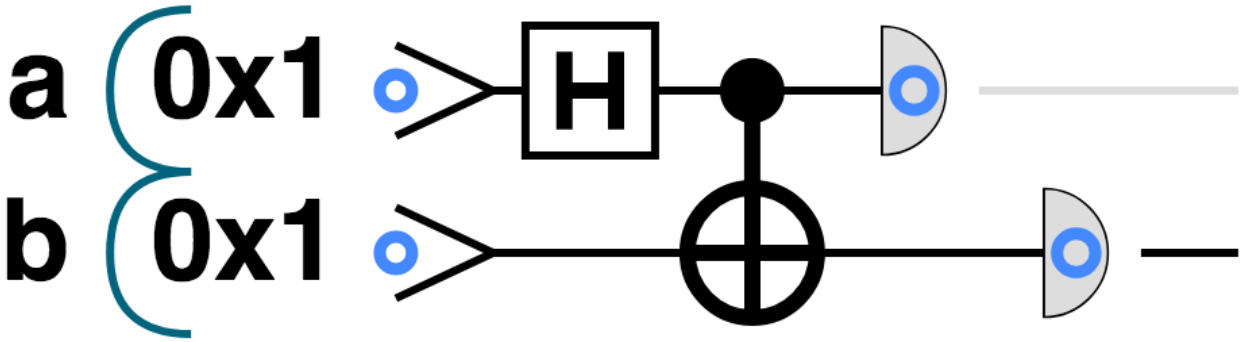


Figure 3-11. CNOT with a target qubit in superposition

Note that, for ease of reference, we've temporarily labeled our two qubits as `a` and `b` (rather than using hexadecimal). Starting with our register in the |0⟩ state, let's walk through the circuit and see what happens.

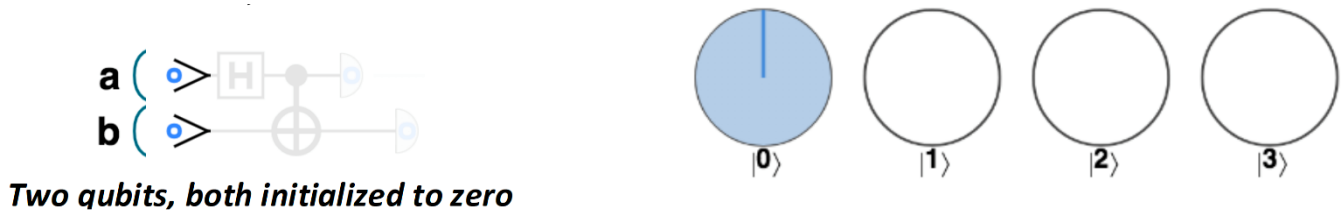


Figure 3-12. Bell Pair step 1

First we apply HAD to qubit `a`. Since `a` is the lowest weight qubit in the register, this creates a superposition of the terms |0⟩ and |1⟩, visualized below in circle notation:

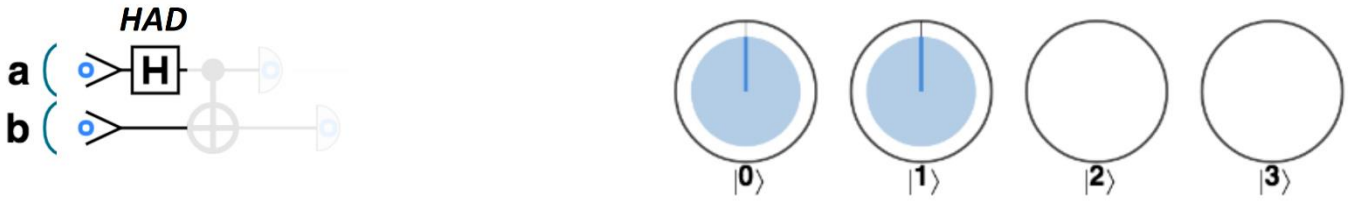


Figure 3-13. Bell Pair step 2

Next we apply the CNOT operation such that qubit `b` is *conditionally* flipped, dependent on the state of qubit `a`.

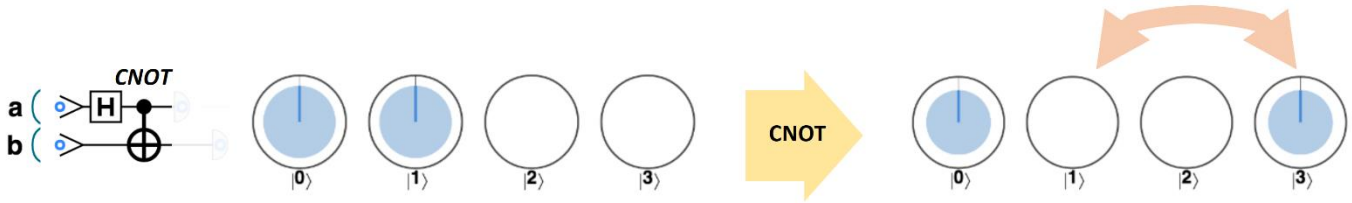


Figure 3-14. Bell Pair step 3

The result is a superposition of |0⟩ and |3⟩. This makes sense, since *if* qubit `a` had taken value |0⟩ then no action would occur on `b`, and it would remain in the state |0⟩ - leaving the register in a total state of |0⟩|0⟩. However, *if* `a` had been in state |1⟩ then a NOT would be applied to `b` and the register would have a value of |1⟩|1⟩=|3⟩. Another, way to understand the CNOT's operation in Figure 3-14 is that it simply follows the rule of changing the value of qubit `b` - which implies swapping the states |1⟩ and |3⟩ (as is shown by the orange arrow in Figure 3-14). In this case, one of these circles just so happens to be in superposition.

What we obtain in Figure 3-14 turns out to be a very powerful resource. In fact, it's the two-qubit equivalent of the *entangled state* we first saw in Figure 3-3. We've already noted that these entangled states demonstrate a kind of interdependence between qubits - if we read out the two qubits shown in Figure 3-14, then although the outcomes will be random they will always agree (i.e. be either `00` or `11`, but with 50% chance for each). And we've also mentioned that these states can't be described only in terms of the individual qubits making up the register.

The one exception we made so far to our mantra of *"avoid physics at all costs"*, was to give a slightly deeper insight into superposition. The phenomenon of entanglement is equally important within a QPU, and so - for one final time - we'll indulge ourselves in a few sentences of physics to give you a better feel for *why* entanglement is so powerful. Again, if you prefer code samples over physics insight, then you can happily skip the next couple of paragraphs without side effect.

TIP

If you want to steer clear of physics altogether then you can skip the next couple of paragraphs explaining entanglement without any ill effect.

From what we've seen so far, entanglement might not necessarily strike you as strange. Finding agreement between the values of bits certainly isn't cause for concern - not even if they randomly assume correlated values. In conventional computing if two otherwise random bits are always found to agree on readout then there are two entirely rational possible explanations.

1. Some mechanism in the past has coerced their values to be equal (which we simply discover on reading them out) - what we might call a *common cause*. Their randomness is therefore illusory.
2. The two bits truly do randomly assume particular values at the very moment of readout, but by some method they are able to communicate with each other, to ensure they correlate.

In fact, with a bit of thought is possible to see that - for conventional bits - these are the *only* two ways we could explain how two bits can randomly agree.

Yet through a clever experiment, initially proposed by the great Irish physicist John Bell, it's possible to conclusively demonstrate that entanglement allows such agreement, but without either of these two reasonable explanations being responsible! This is the sense in which you may hear it said that entanglement is a kind of distinctly quantum link between qubits that is *stronger than could ever be conventionally possible*. As we start programming more complex QPU applications, entanglement will begin popping up everywhere. You won't need to explicitly think about it too hard, but it's helpful to have a litle insight into what's going on under the hood.

Hands-on: Using Bell Pairs for shared randomness

The entangled state we created in Figure 3-14 is commonly referred to as a Bell pair1. Let's see a quick and easy way to put the powerful link within this state to use.

We saw in the previous chapter how simply measuring a qubit in superposition provides us with a Quantum Random Number Generator. Similarly, the reading out of a Bell pair acts like a QRNG, only now we will obtain agreeing random values on two qubits.

A surprising fact about entanglement is that the qubits involved remain entangled no matter how far apart we may move them. Thus we can easily use Bell pairs to generate *correlated random bits* at different locations. Such bits can be the basis for establishing secure shared randomness - something that critically underlies the modern internet.

The code snippet in Example 3-2 shows how we can implement this prescription to generate shared randomness, by creating a Bell pair and then reading out a value from each qubit.

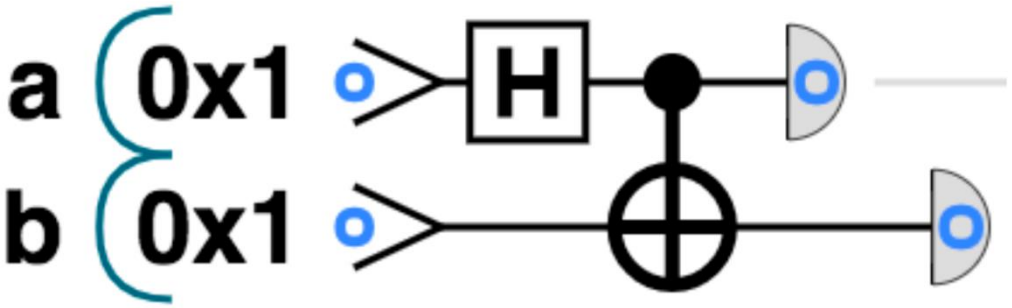


Figure 3-15. Bell Pair circuit

SAMPLE CODE

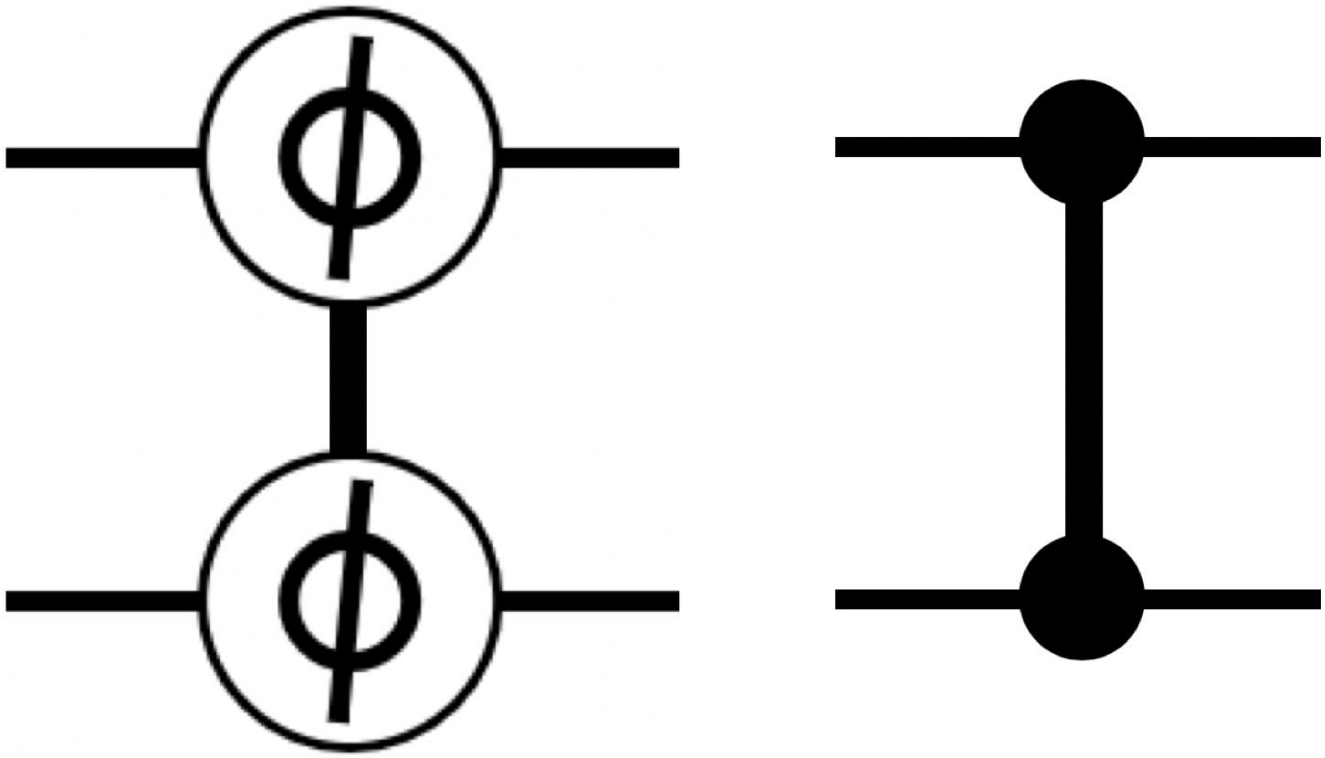
Run this sample online at *http://oreilly-qc.github.io?p=3-2*

***Example 3-2.** Make a Bell Pair*

```
qc.reset(2);
a = qint.new(1, 'a');
b = qint.new(1, 'b');
qc.write(0);
a.had();            // Place into superposition
```

```
b.cnot(a);          // Entangle
a_result = a.read();
b_result = b.read();
qc.print(a_result);
qc.print(b_result);
```

QPU Instruction: CPHASE and CZ



Another very common two-qubit operation is CPHASE(θ). Like the CNOT operation, CPHASE employs a kind of conditional logic that can generate entanglement between qubits. Recall from Figure 3-6 that the single-qubit PHASE(θ) operation acts on a register to rotate (by angle θ) the |1⟩ values in that qubits operator pairs. As CNOT did for NOT, CPHASE restricts this action on some target qubit to only occur when another control qubit assumes a value |1⟩. Note that CPHASE only acts when its control qubit is |1⟩, and when acting only affects target qubit states having value |1⟩. This means that a CPHASE(θ) applied to, say, qubits `0x1` and `0x4` results in the rotation (by θ) of all circles for which *both* these two qubits have a value of 1. Because of this particular propert of the action of PHASE, CPHASE has a symmetry between its inputs not shared by CNOT. Unlike most other controlled operations, it's irrelevant which qubit we consider to be the target and which we consider to be the control for CPHASE.

Below we compare the operation of a CPHASE between the `0x1` and `0x4` qubits with individual PHASE operations on these qubits:

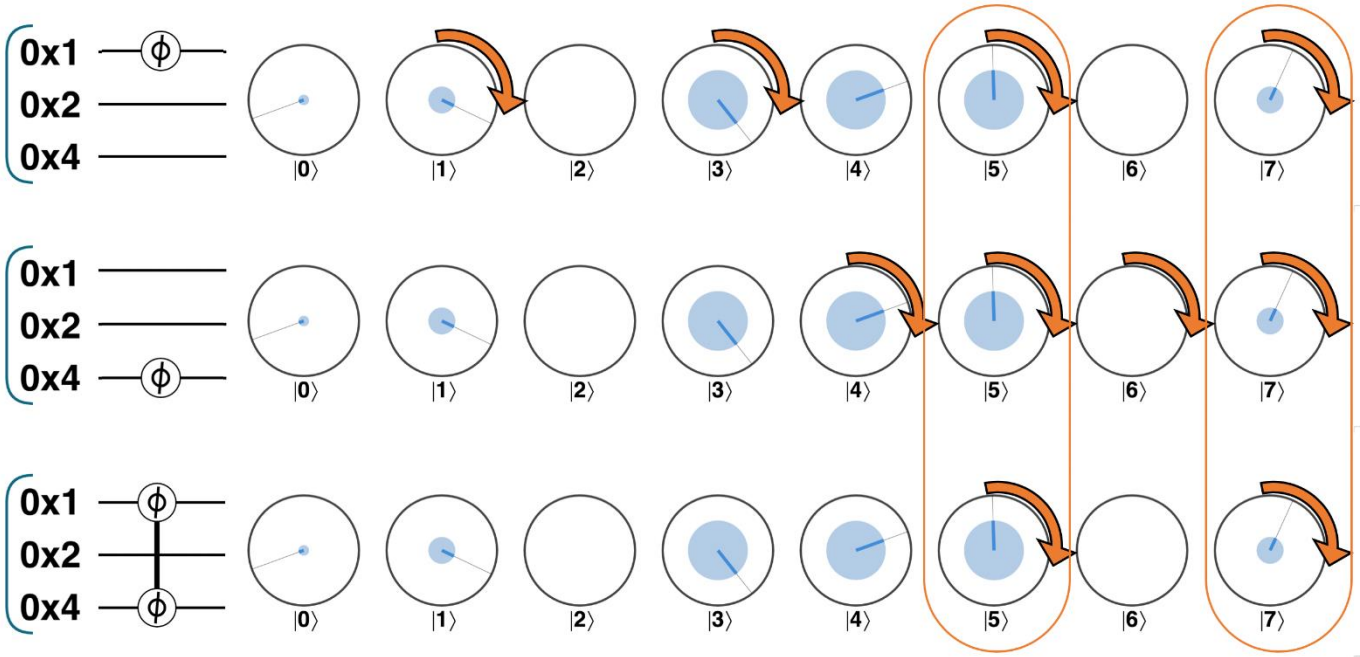


Figure 3-16. Applying CPHASE in circle notation

On their own, single-qubit PHASE operations will rotate the relative phase of *half* of the circles associated with a QPU register. Adding a condition cuts the number of rotated circles in half. We can also add further conditional qubits to our CPHASE, each time cutting the number of terms we act on in half again. In general, the more that we *condition* QPU operations, the more selective we can be with which terms of a QPU we manipulate.

QPU programs very frequently employ the CPHASE(θ) operation with a phase of $\theta = 180^\circ$, and consequently this particular implementation of CPHASE is given its own name: *CZ*, along with its own simplified symbol shown in Figure 3-17. Interestingly, CZ can be constructed from HAD and CNOT very easily. Recall from Figure 2-16 that the Z operation can be made from two HAD operations and a NOT. Similarly, CZ can be made from two HAD operations and a CNOT, as shown in Figure 3-17

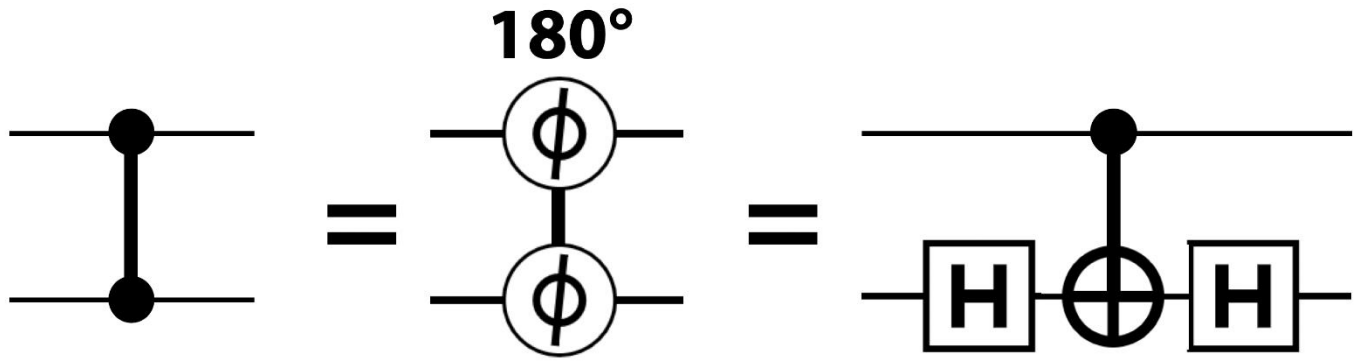


Figure 3-17. CZ, CPHASE and CNOT

Phase kickback

Once we start thinking about altering the phase of one QPU register *conditioned* on the values of qubits in some other register we can produce a surprising and useful effect known as *phase kickback*. Take a look at the below circuit:

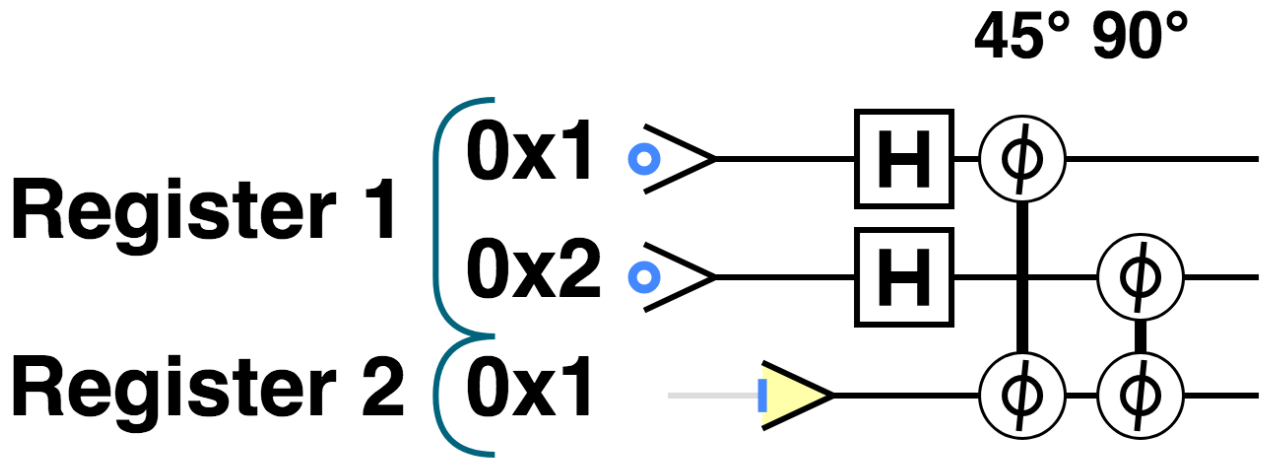


Figure 3-18. Circuit for demonstrating phase kickback trick

One way to think of this circuit is that, after placing `Register 1` in a superposition of all of its $2^2 = 4$ possible values, we've rotated the phase of `Register 2` conditional on the values taken by the qubits of `Register 1`. However, looking at the resulting individual states of *both* registers in circle notation we see that something interesting has also happened to the state of `Register 1`:

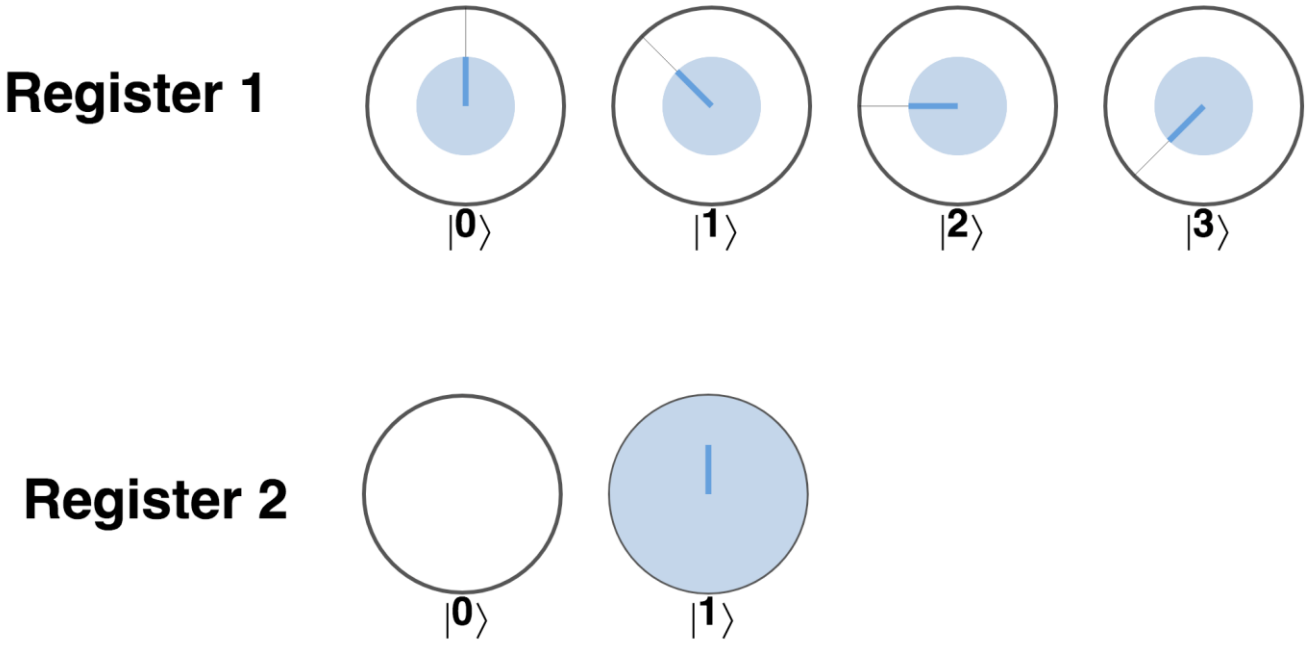


Figure 3-19. States of both registers involved in phase kickback

The phase rotations we tried to apply to the whole of the second register (conditioned on qubits from the first) have also affected different terms from the first register! More specifically, here's what we're seeing happen in the above circle notation representations:

- The 45° rotation we performed on `Register 2` conditioned on the lowest weight qubit from `Register 1` has *also* rotated every term in `Register 1` for which this lowest weight qubit is activated (the |1⟩ and |3⟩ terms).
- The 90° phase rotation conditioned on `Register 1`s highest weight qubit has also been 'kicked-back' onto all terms in `Register 1` having this qubit activated (|2⟩ and |3⟩).

The net result that we see on `Register 1` is the combination of these phase rotations that have been *kicked back* onto it from our intended target of `Register 2`.

Phase kickback is a very useful idea, as we can use it to apply phase rotations to specific terms in a register (`Register 1` in the above example). We can do this by performing a phase rotation on some *other* register conditioned on qubits from the register we really care about that chosen to specifically pick out the terms we want to rotate.

NOTE

For phase kickback to work, we always need to initialize the second register in the |1⟩ value. Notice that although we are applying two-qubit operations between the two registers, be are not creating any entanglement, hence we can fully represent the state as separate registers. Two qubit gates not always generate entanglement between registers, we will see why in [Link to Come].

Don't feel bad if this phase-kickback trick initially strikes you as a little mind-bending, it can take a while to get used to. The easiest way to get a feel for it is to play around with some examples to build intuition for how entanglement is linking the two registers phases. To get you started, Example 3-3contains QCEngine code for reproducing the two register example we described above.

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=3-3*

***Example 3-3.** Phase Kickback*

```
qc.reset(3);
// Create two registers
reg1 = qint.new(2, 'Register 1');
reg2 = qint.new(1, 'Register 2');
reg1.write(0);
reg2.write(1);
// Place the first register in superposition
reg1.had();
// Perform phase rotations on second register,
//conditioned on qubits from the first
```

```
qc.phase(45, 0x1|0x4);
qc.phase(90, 0x2|0x4);
```

We'll make use of phase-kickback in [Link to Come] to understand the inner workings of a QPU primitive, and again in <<>> to explain how a QPU circuit can help us solve systems of linear equations. The wide utility of phase kickback stems from the fact that it doesn't only work for CPHASE operations, but any conditional operation that generates a change in phase on a register (even if the phase is a global, unREADable one). This is as good a reason as any to understand how we might construct more general *conditional operations*.

Constructing Any Conditional Operation

We've introduced CNOT and CPHASE operations, but is there such thing as a CHAD (conditional HAD), or a CRNOT (conditional RNOT)? There are, and even if a conditional version of some single qubit operation is missing from the instruction set of a particular QPU, there's a process by which we can "convert" single qubit operations into a multi-qubit conditional ones.

The general prescription for *conditioning* a single qubit operation involves a little more mathematics than we want to cover here, but seeing a couple of examples will help us feel more comfortable with the idea (and also be useful to us later on). The key idea is breaking our single qubit operation into smaller steps. It turns out that it's always possible to break a single qubit operation into a set of steps such that we can use CNOTs to conditionally *undo* our operation.

This is much easier to see with a simple example: suppose we are writing software for a QPU whose instruction set includes CNOT, CZ and PHASE, but no instruction to perform CPHASE. This is actually a very common case with modern QPUs, but fortunately we can easily add a CPHASE at the compiler stage.

Recall that the desired effect for a 2-qubit CPHASE is to rotate the phase for any term for which *both*qubits are take value |1⟩ (in this example we consider CPHASE(90)):

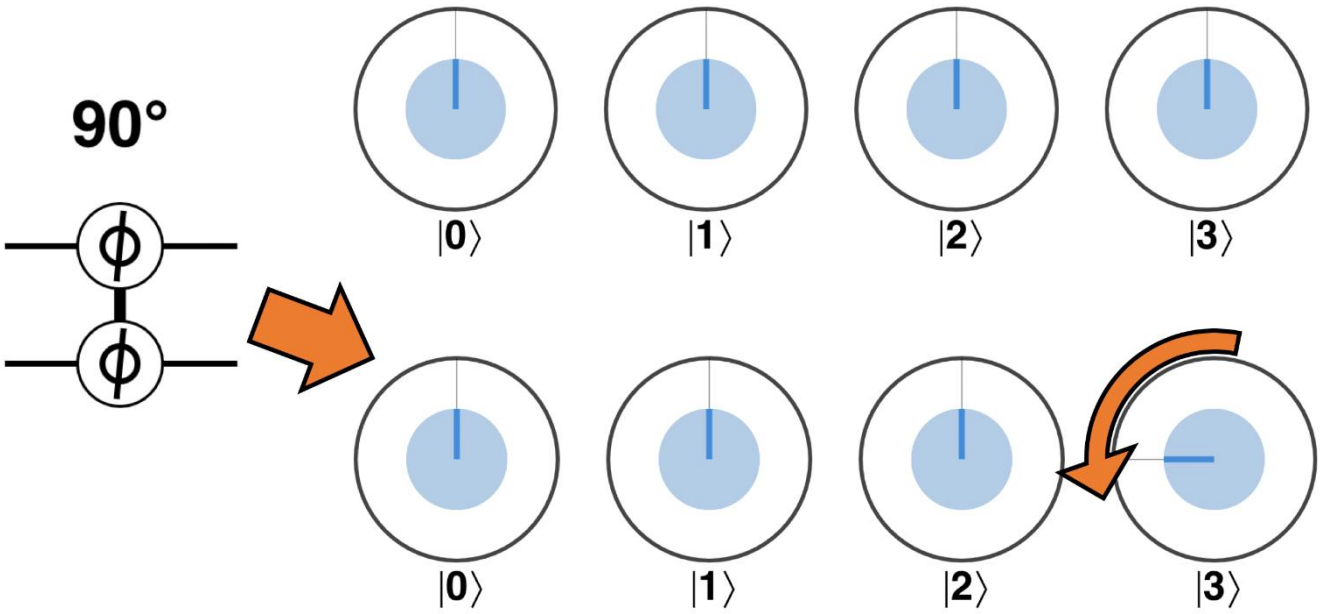


Figure 3-20. Desired operation of a CPHASE(90) operation

We can easily break down a PHASE(90) operation into smaller pieces by rotating through smaller angles. For example, PHASE(90)=PHASE(45)PHASE(45). We can also *undo* a rotation by rotating in the opposite direction, for example the operation PHASE(45)PHASE(-45) is the same as doing nothing to our qubit. With these facts in mind, we can construct the CPHASE(90) operation described in Figure 3-20 using the gates in Figure 3-21, as imlemented in Example 3-4.

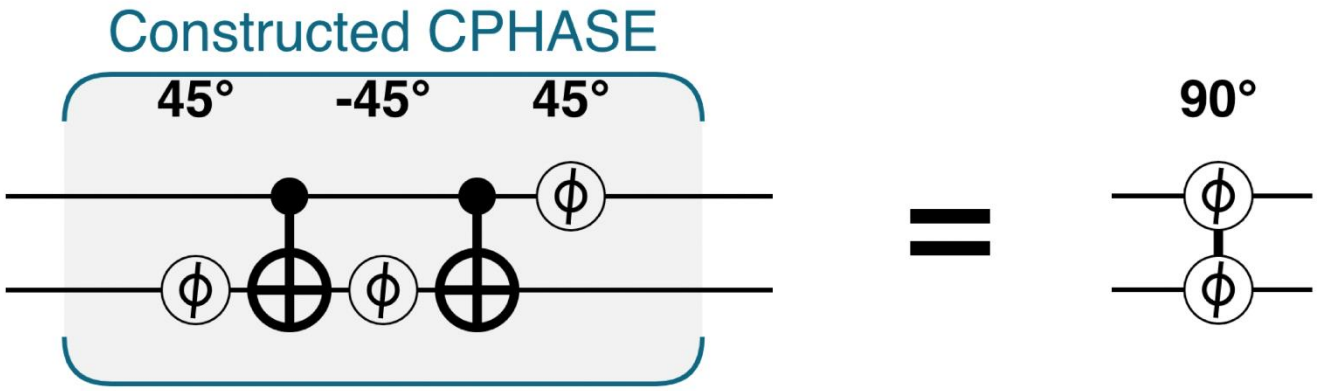


Figure 3-21. Constructing a CPHASE operation

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=3-4*

***Example 3-4.** Custom conditional-phase*

```
theta = 90;

// Using two CNOTs and three PHASEs...
qc.phase( theta / 2, 0x2);
qc.cnot(0x2, 0x1);
qc.phase(-theta / 2, 0x2);
qc.cnot(0x2, 0x1);
qc.phase( theta / 2, 0x1);

// Builds the same operation as a 2-qubit CPHASE
qc.phase(theta, 0x1 | 0x2);
```

Following this circuit's operation on different possible inputs we can see how it works to only apply PHASE(90) when both qubits are 1 (an input of 11). To see this it helps to recall that PHASE has no effect on a qubit in the state $|0\rangle$. Alternatively, we can follow the action of Figure 3-22 using circle notation.

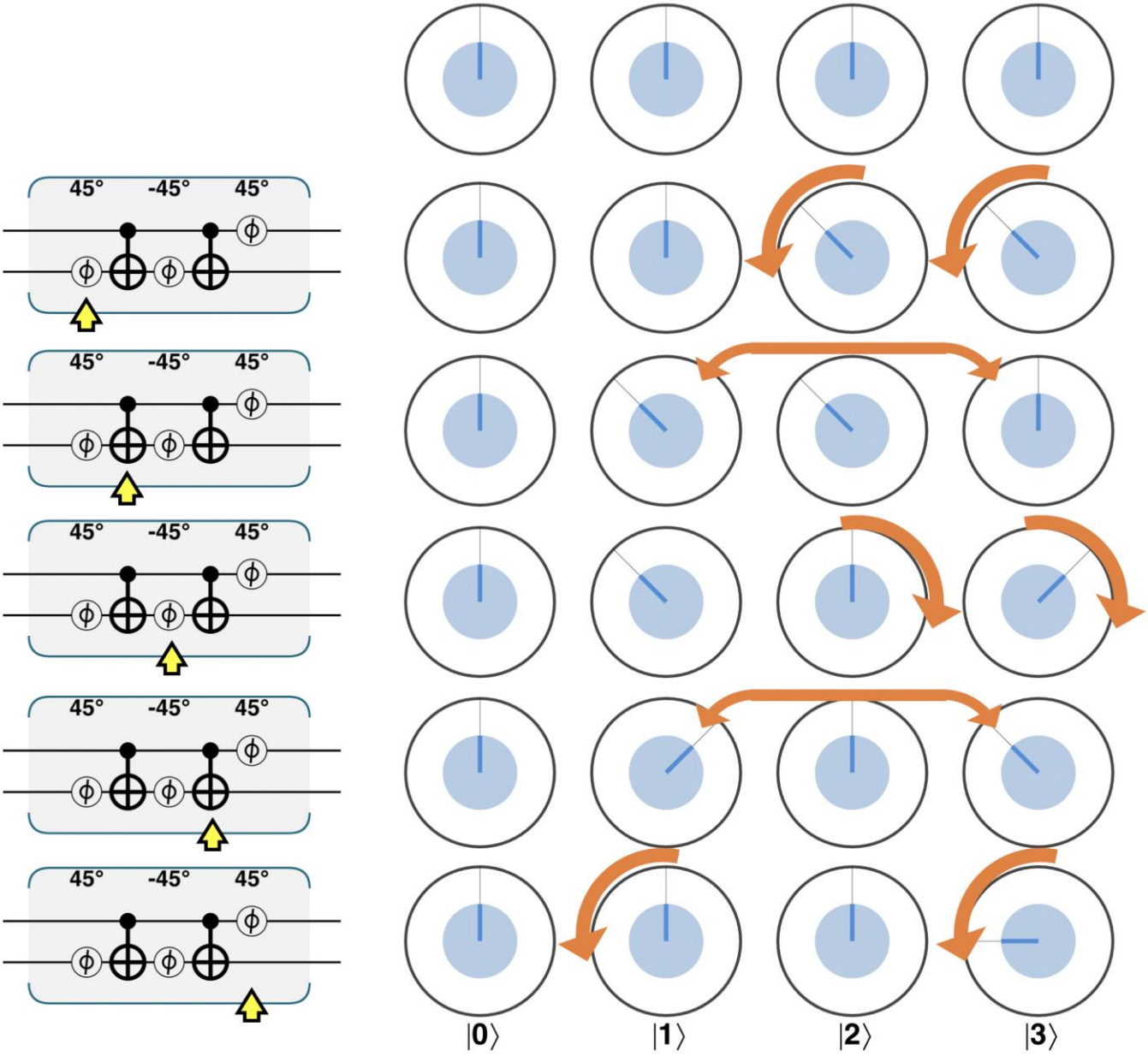
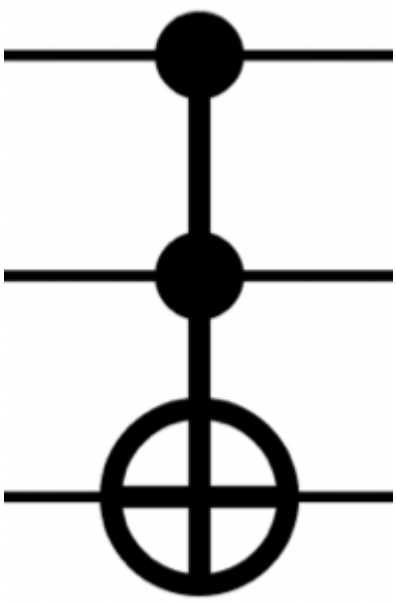


Figure 3-22. Step-by-step through the constructed CPHASE operation

Another example of conditioning a single qubit operation that will be useful to us in later chapters is ROTY…

[[Example of conditioning ROTY to be added]]

QPU Instruction: CCNOT (Toffoli)



We've previously noted that multi-qubit conditional operations can be made more selective by performing operations conditioned on more than one qubit. Let's see this in action and generalize the CNOT operation by adding multiple conditions. A CNOT with two condition qubits is commonly referred to as a CCNOT operation. The CCNOT is also sometimes called a Toffoli gate, after the eponymously titled equivalent gate from conventional computing

With each condition added, the NOT operation stays the same, but the number of operator pairs affected in the registers circle notation is reduced by half. We show this below by comparing a NOT operation on the first qubit in a three-qubit register alongside the associated CNOT and CCNOT operations.

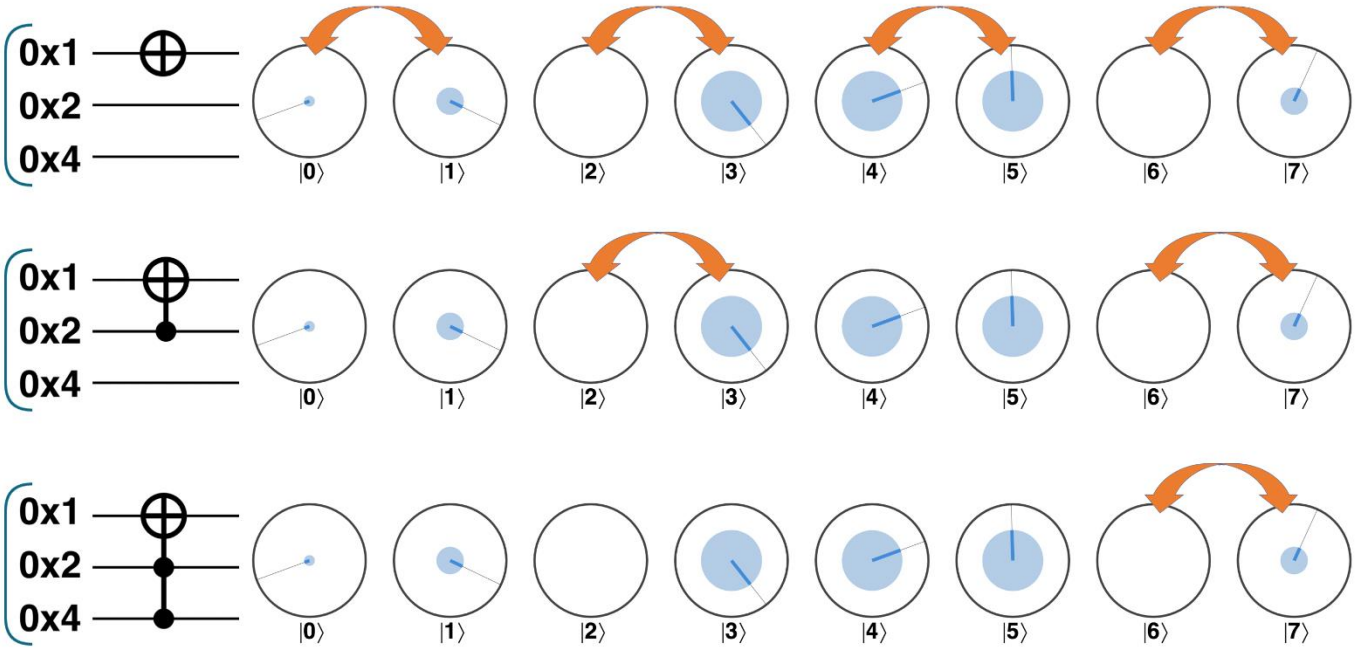


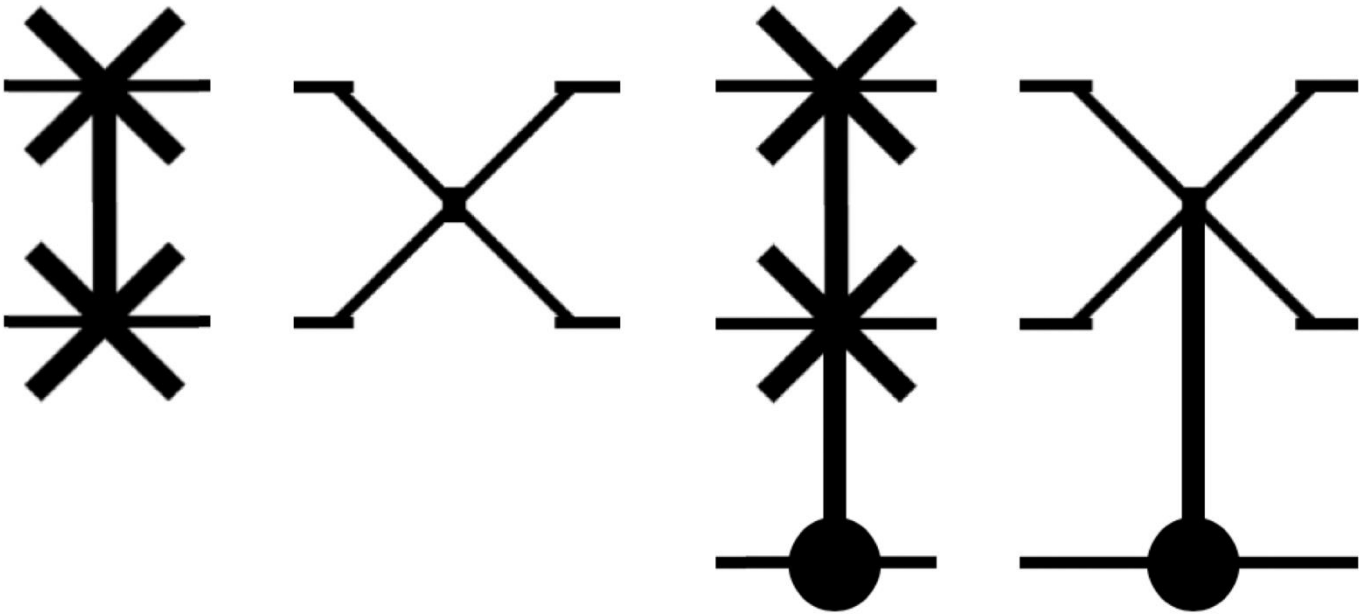
Figure 3-23. Adding conditions makes the NOT operation more selective

In a sense, a Toffoli can be interpreted as an operation implementing "if A AND B then flip C". For performing basic logic, the Toffoli gate is arguably the single most useful QPU operation. Multiple Toffoli gates can be combined and cascaded to produce a wide variety of logic functions, as we will explore in [Link to Come].

FAN IN AND FAN OUT

The number of *condition* qubits which may be applied to a single CNOT-style operation on a QPU is sometimes referred to as the *fan in* of the QPU. The number of *target* qubits is referred to as the *fan out*.

QPU Instruction: SWAP and CSWAP



Another very common operation in quantum computation is SWAP (also called *exchange*), which simply exchanges two qubits. If the architecture of a QPU allows it then SWAP may be a truly fundamental operation in which the physical objects representing qubits are actually moved to swap their positions. Alternatively, a SWAP can be performed by exchanging the *information* contained in two qubits (rather than the qubits themselves) using three CNOT operations, as shown in Figure 3-24.

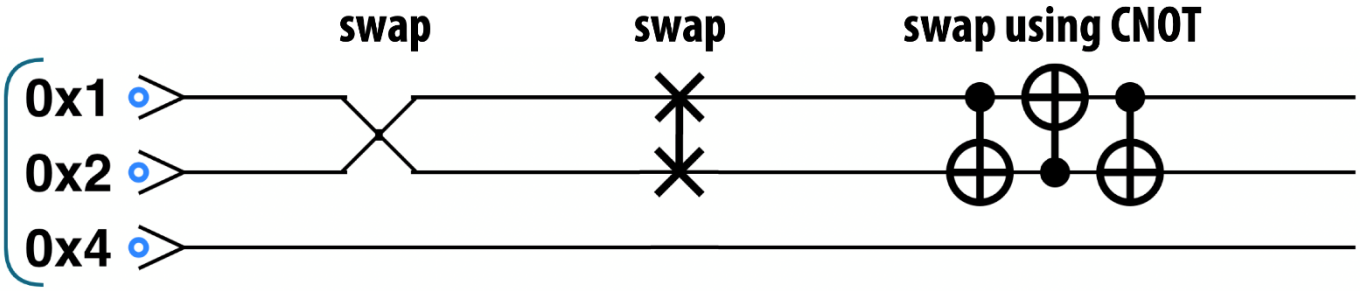


Figure 3-24. Swap can be made from CNOT operations

A DIGITAL PARALLEL

In this book we have two ways to indicate swap operations. In the general case we use a pair of connected X's. When the swapped qubits are adjacent to one another, it is often simpler and more intuitive to cross the qubit lines. The operation is identical in both cases.

You may be wondering why SWAP is a useful operation. Why not simply re-name our qubits instead? On a QPU, SWAP comes into its own when we consider generalizing it to a conditional operation called CSWAP, or *conditional exchange*. CSWAP can be implemented using three Toffoli gates, as shown in Figure 3-25.

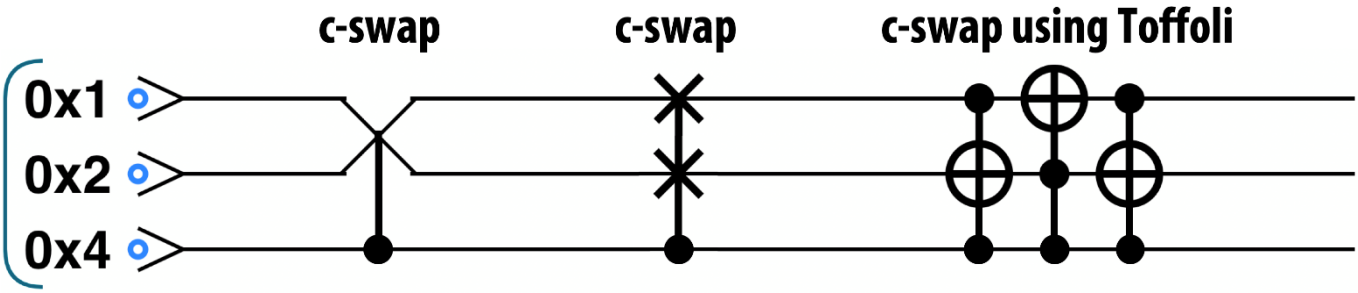


Figure 3-25. CSWAP constructed from Toffoli gates

If the condition qubit for a CSWAP operation is in superposition then we end up with a superposition of our two qubits being exchanged, and also being not exchanged. In [Link to Come] and [Link to Come], we'll see how this feature of CSWAP allows us to perform multiplication-by-2 in quantum superposition.

The swap test

SWAP operations allow us to build a very useful circuit known as a *swap test*. A swap test circuit solves the following problem: If you're given two qubit registers, how do you tell if they are in the *same* state? By now we know only too well that in general we can't use the destructive READ operation to completely learn the state of each register in order to make the comparison. The SWAP operation does something a little more sneaky. Without telling us what either state is, it simply lets us determine whether or not they're equal.

The in a world where we can't necessarily learn precisely what's in an output register, the abilities of the swap test can be an invaluable tool. Figure 3-26 shows a circuit for implementing a swap test.

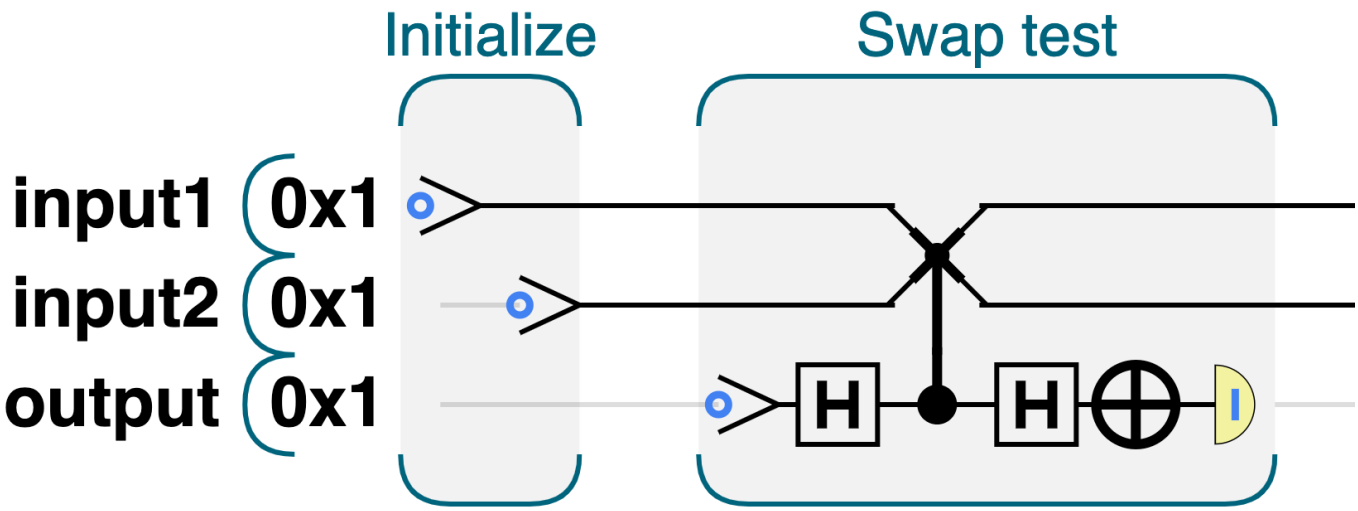


Figure 3-26. Using the swap test to determine whether two states match

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=3-5*

Example 3-5. The swap test

```
// In this example the swap test should reveal
// the equality of the two input states
qc.reset(3);
var input1 = qint.new(1, 'input1');
var input2 = qint.new(1, 'input2');
var output = qint.new(1, 'output');

// Initialize to any states we want to test
input1.write(0);
input2.write(0);

// The swap test itself
output.write(0);
output.had();
input1.exchange(input2, 0x1, output.bits());
output.had();
output.not();
result = output.read();
// result is 1 if inputs are equal
```

In Example 3-5 we use the swap test to compare the states of two single-qubit registers, but the same circuit can easily be extended to compare multi-qubit registers. The result of the swap test is found when we READ the single qubit `output` register (no matter how large are input registers, the output remains a single qubit). By changing the lines `input1.write(0)` and `input2.write(0)` you can experiment with what `output` the swap test produces for inputs that differ to varying extentsfootnote:[]. You should find that if the two input states are equal then the `output` register results in a state $|1\rangle$, and so we definitely obtain a 1 outcome when applying a READ to this register. However, as the two inputs become increasingly more different, the probability of obtaining a 1 outcome upon READing the `output` register decreases, eventually becoming 50% in the case where we have `input1` being state $|0\rangle$ and `input2` being state $|1\rangle$. The below plot shows precisely how the

probability of getting outcome 1 in the `output` register changes as the difference between the two input registers is increased.

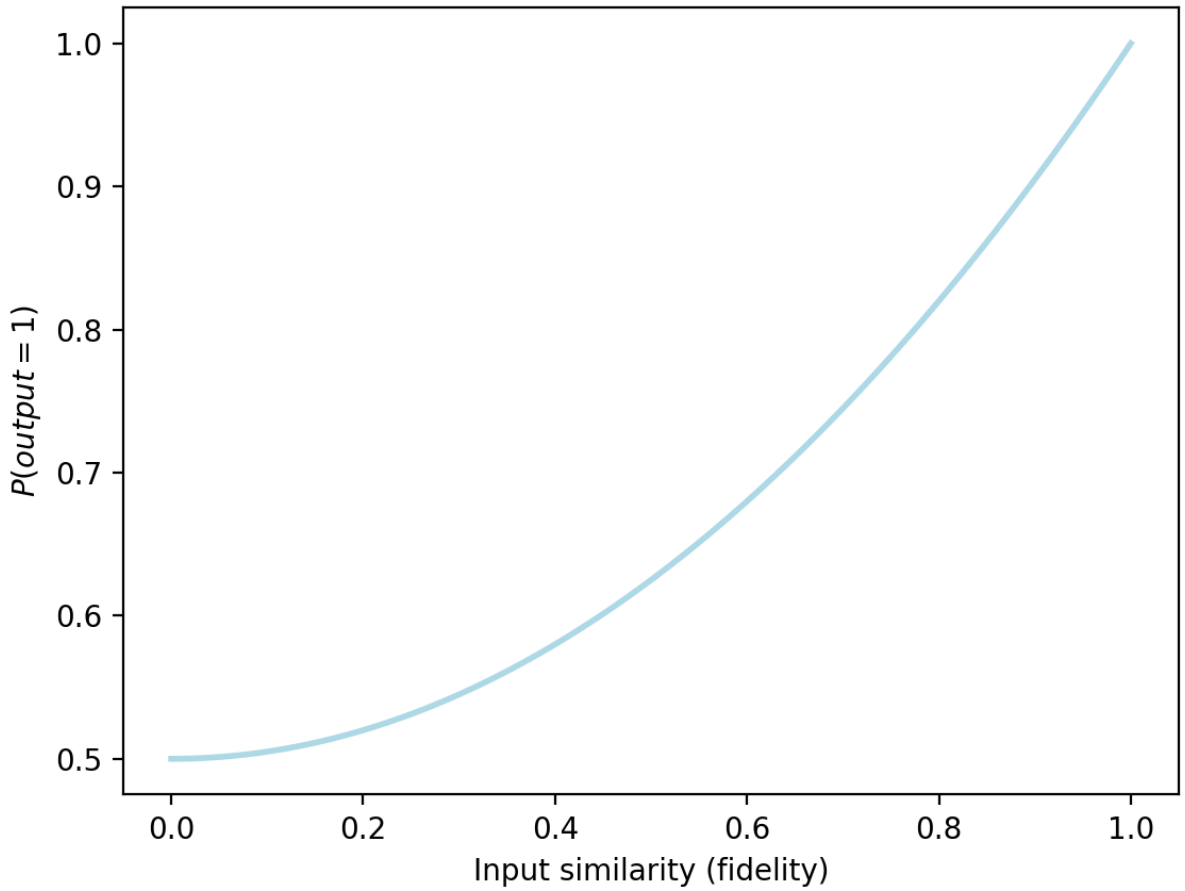


Figure 3-27. How output of swap test varies as input states are made increasingly different

The x-axis on this plot uses a numerical measure of the distance between two states known as the *fidelity*. We won't go into the mathematical detail of how the fidelity between the states of two QPU registers is calculated, but it mathematically encapsulates a way of comparing two superpositions. The idea is that a single qubit register having a state in superposition of almost entirely |0⟩, with a small contribution of |1⟩, is very close to a register in the state |0⟩. As we increase the proportion of |1⟩ in the superposition this registers state gradually becomes increasingly *different* from a register in state |0⟩. The fidelity is actually the *overlap* (inner product) between the complex vectors representing the two input states (in their full-blown quantum mechanical mathematical description).

By re-running the swap test circuit multiple times we can keep track of the outcomes we obtain. The more runs for which we observe a 1 outcome the more convinced we can be that the two input states were identical. Precisely how many times we would need to repeat the swap test depends on how confident we want to be that the two inputs are identical, and how close we would allow them to be in order to be called identical. The below plot shows how many swap tests we would need to run and obtain 1 outcomes for (y-axis) in order to be 99% confident that our inputs are identical.

The x-axis shows how this number changes as we relax the requirement for inputs being identical (measuring similarity with the previously mentioned notion of fidelity). Note that if at any time we obtain a 0 outcome then we know that the two input states are not identical[2].

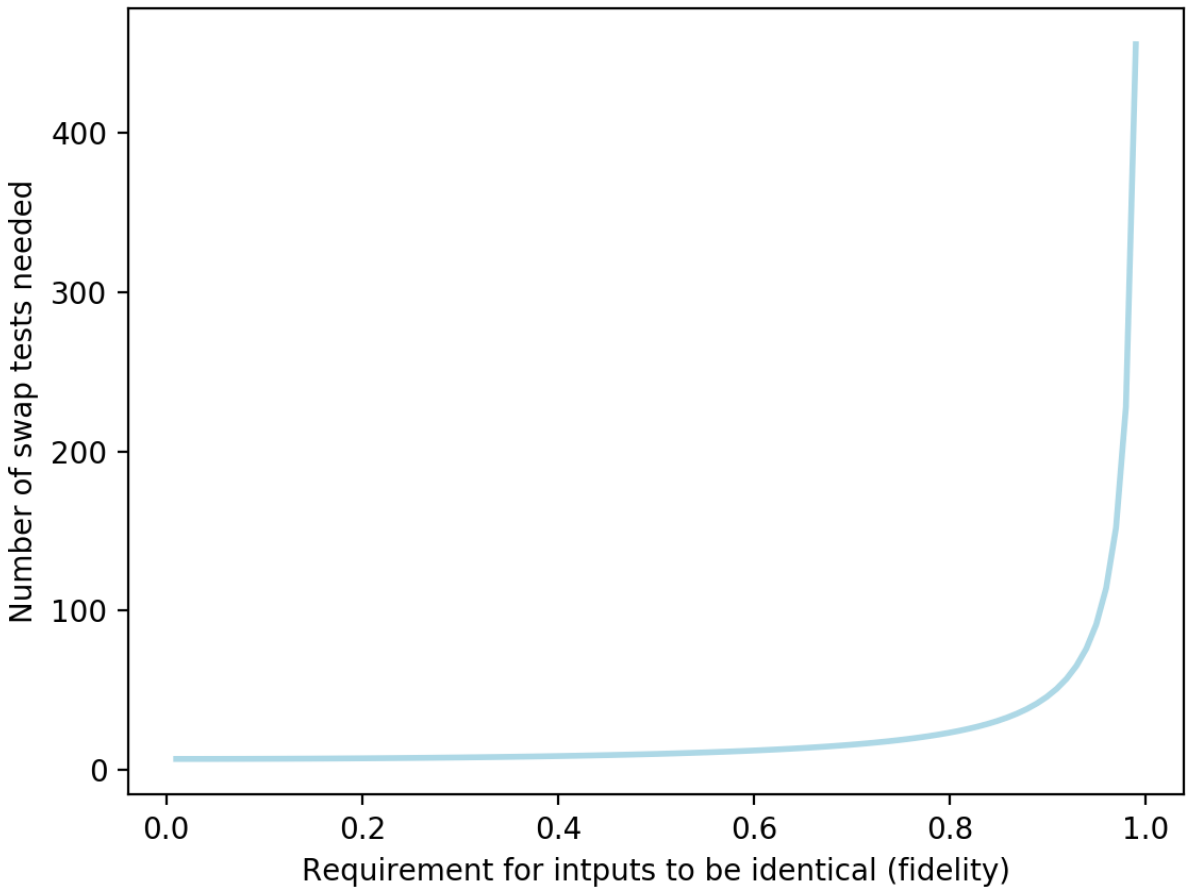


Figure 3-28. Number of swap tests that would need to return an outcome of 1 for us to be 99% confident inputs are identical

Rather than looking at the swap test as a yes/no way of ascertaining whether two states are equal, another useful interpretation is to note that Figure 3-27 and Figure 3-28 shows us that the *probability* that we get a 1 in `outcome` is a measure of just *how* identical the two inputs are (their fidelity). If we repeat the swap test enough times then we can estimate this probability and therefore their fidelity. In this way we can estimate the overlap between two quantum states - something we'll find useful when summarizing the operation of quantum machine learning applications later on.

By running the swap test circuit an adequate number of times we can, with some confidence, determine whether a state in a quantum register is equal to some other reference state. This can be an invaluable tool for utilizing outputs from quantum algorithms that may otherwise seem *trapped* in quantum superposition. We'll make use of this tool in later chapters.

Hands-on: Remote-controlled randomness

Armed with multi-qubit operations we're now in a position to start making use of the power of quantum entanglement. We can explore some interesting and non-obvious properties of entanglement using a small QPU program for *remote controlled* random number generation. This program will generate two qubits, such that reading out one instantly affects the probabilities for obtaining a random bit read out from the other. Moreover, this effect occurs over any distance of space or time. This seemingly impossible task, enabled by a QPU, is surprisingly simple to implement.

Here's how the remote control will work: We will manipulate a pair of qubits such that reading one qubit (either one) returns a 50/50 random bit which tells you the "modified" probability of the other. If the result is 0, then the other qubit will have 15% probability of being 1. Otherwise, if the qubit you read returns 1, then the other one will have 85% probability of being 1.

The sample code in Example 3-6 shows how to implement this remote controlled random number generator. As in Example 3-1 we make use of the QCEngine `qint` object to be able to properly address different qubits directly as objects:

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=3-6*

Example 3-6. *Remote-controlled Randomness*

```
qc.reset(2);
a = qint.new(1, 'a');
b = qint.new(1, 'b');
qc.write(0);
a.had();
// now prob of a is 50%
b.had();
b.phase(45);
b.had();
// now prob of b is 15%
b.cnot(a);
// Now, you can read *either*
// qubit and get 50% prob.
// If the result is 0, then
// the prob of the *remaining*
// qubit is 15%, else it's 85%
a_result = a.read();
b_result = b.read();
qc.print(a_result + ' ');
qc.print(b_result + '\n');
```

We can follow the effect of each operation within this circuit in circle notation:

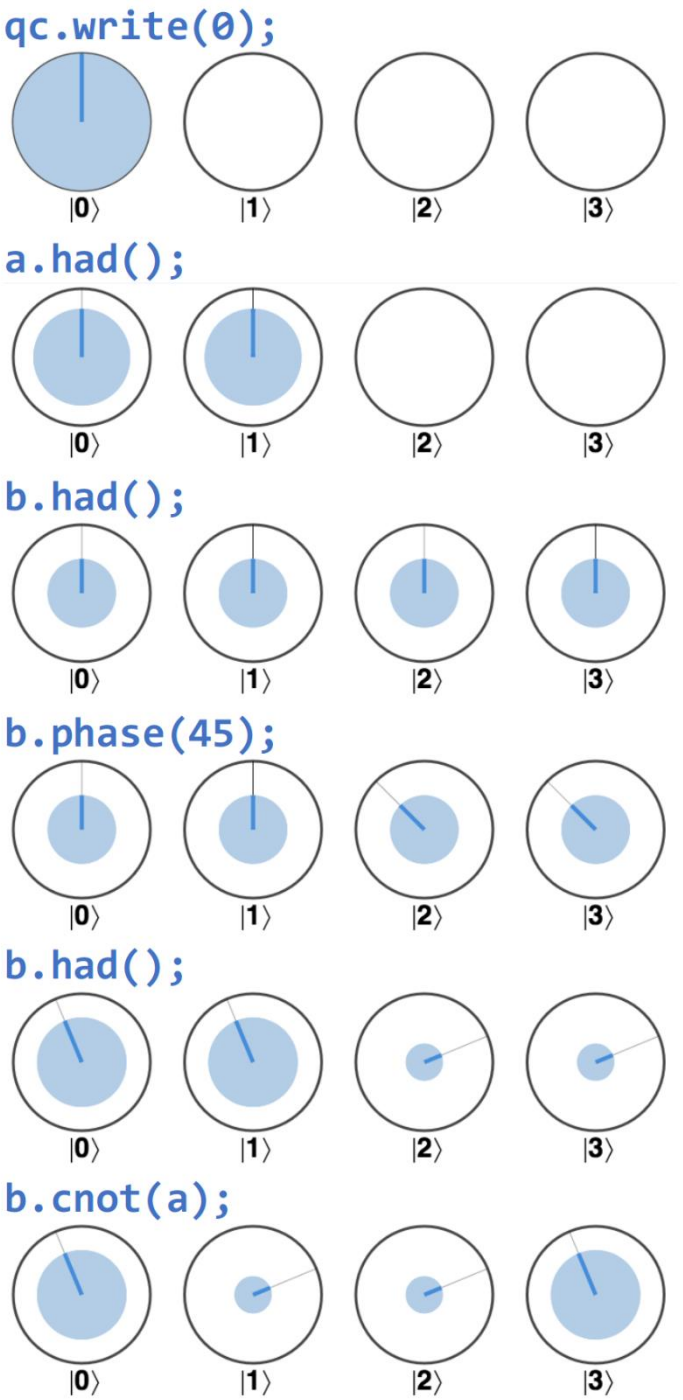


Figure 3-29. The remote-random sample, step by step

After these steps are completed we're left with an entangled state of the two qubits shown at the end of Figure 3-29 - for which we have amplitudes in each possible state |0⟩, |1⟩, |2⟩ and |3⟩.

Let's consider what would happen if we prepare this state and then read out qubit `a`. If qubit `a` were read out to have value 0 (as it will with 50% probability), then only the circles compatible with this state of affairs would remain. These are the terms for |0⟩|0⟩ = |0⟩ and |1⟩|0⟩ = |2⟩, and so the state of the two qubits becomes the state shown in Figure 3-30.

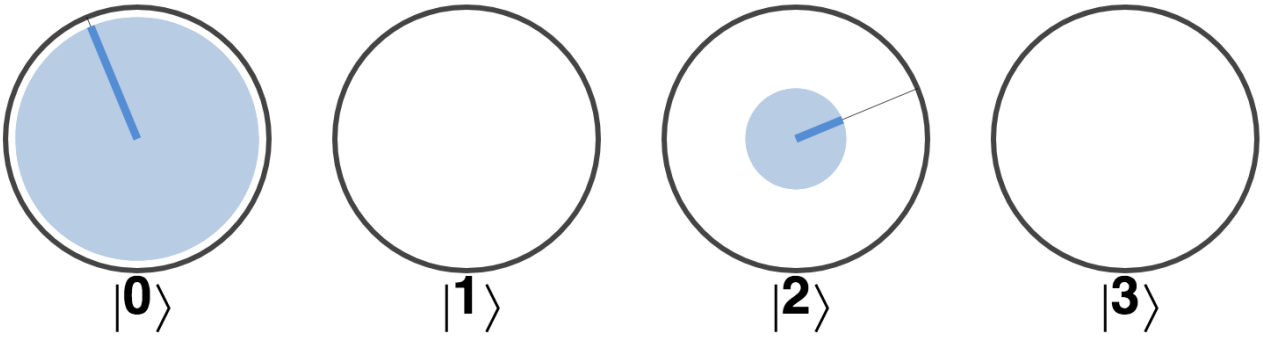


Figure 3-30. The state of one qubit in the remote control after the other is READ to be 0.

The two terms in this state have non-zero probabilities, both for obtaining 0 and for obtaining 1 when reading out qubit *+b+*. In particular, qubit a has 70%/30% probabilities of reading out 0/1:

However, suppose that our initial measurement on qubit a had yielded 1 (which also occurs with 50% probability). Then only the |0⟩|1⟩=|1⟩ and |1⟩|1⟩=|3⟩ terms will remain after the readout:

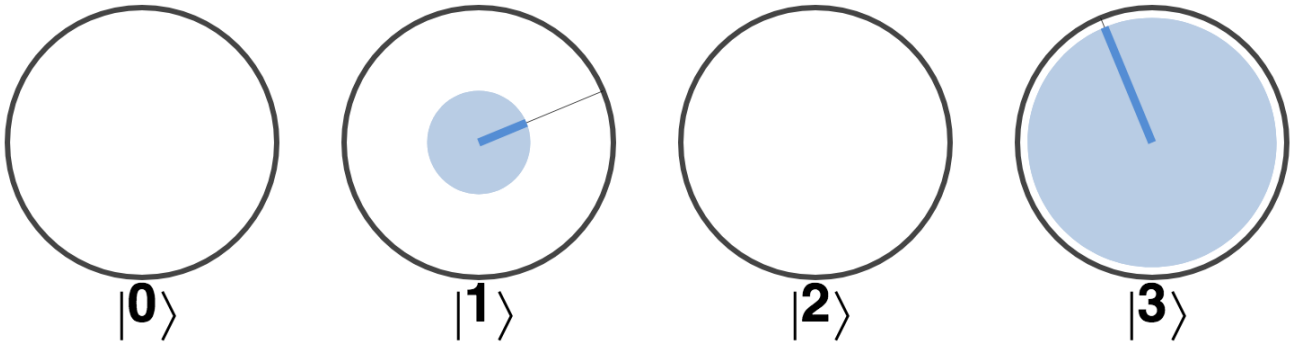


Figure 3-31. The state of one qubit in the remote control after the other is READ to be 1.

And the probabilities for reading out the 0/1 outcomes on qubit b are now 30%/70%.

Note that although we were able to change the probability distribution of readout values for qubit b instantaneously, we're unable to do so with any intent - we can't choose whether to cause the 70%/30% or the 30%/70% distribution - since the outcome we get when measuring qubit a is obtained randomly. It's a good job too, since if we could make this change deterministically then we could send *information* instantaneously using entanglement - i.e. faster than light. Although sending signals faster than light sounds like fun, were this possible, bad things would happen[3]. In fact one of the strangest things about entanglement is that although it allows us to change the states of qubits instantaneously across arbitrary distances, it always does so in such way that we can never use the effect to send intelligible, pre-determined information. It would seem that the universe isn't a big fan of sci-fi.

Conclusions

We've seen how single and multi-qubit operations can allow us to manipulate the properties of superposition and entanglement in the qubits of a QPU. With these operations at our disposal we're ready to see how they can really allow us *compute* with a QPU in new and powerful ways. In [Link to Come] we'll see how fundamental digital logic can be re-imagined within a QPU, but first in the next chapter we take a hands-on exploration of quantum teleportation. Not only is quantum teleportation a fundamental component of many quantum applications, but exploring it will also put to the test everything we've learned so far about describing and manipulating registers of qubits.

1 This, along with several other states carry that name because they were in fact the states that John Bell used in his proposed experiment for demonstrating the inexplicable nature of the correlations in entangled states

2 In reality we would want to allow for *some* zero occurences if we were genuinely content with close but not identical states - and also perhaps to allow for the possibility of errors in the computation. For this simple analysis we have ignored these factors.

3 Sending-information-back-in-time-causality-violating kinds of bad things. The venerated work of Dr Emmett Brown attests to the dangers of such hijinks.

Chapter 4. Quantum Teleportation

In this chapter we'll really start getting our feet wet using QPU operations, through some code allowing us to immediately and physically teleport an object 3.1 millimeters! (The same code will work over interstellar distances, given the right equipment).

Although teleportation might conjure up images of magician's parlor tricks we'll see that the kind of *quantum* teleportation we can perform with a QPU is equally impressive, yet far more practical - and is in fact an essential conceptual component of programming a QPU.

Hands on: Let's teleport something

The best way to learn about teleportation is to try and do it. Keep in mind that throughout all of human history, only a few thousand people have actually performed physical teleportation of any kind, so just running the following code makes you a pioneer.

For this example, rather than a simulator, we will use IBM's 5-qubit actual QPU. You'll be able to paste the below sample code into the IBM simulators website, click a button, and confirm that your teleportation was successful.

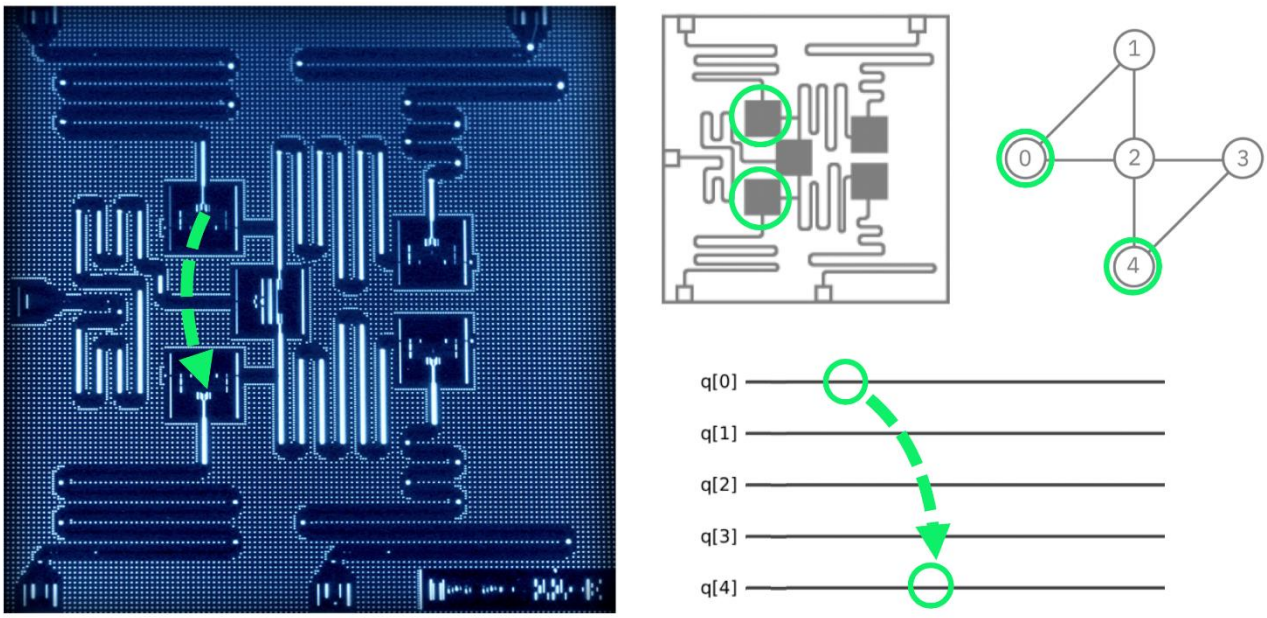


Figure 4-1. This chip is very small, so the qubit does not have far to go.[1]

To try this sample code, see the instructions and links online at *http://oreilly-qc.github.io?p=4-1*

Note that this code sample is *not* JavaScript to be run on QCEngine, but instead can be run online through IBM's cloud interface. Doing so allows you to not just simulate, but actually *perform* the teleportation of a qubit at IBM's research center in Yorktown Heights, New York (at least at the time

of writing). Below we'll walk you through how to do this, by following through the code in detail we'll be able to understand precisely how quantum teleportation works.

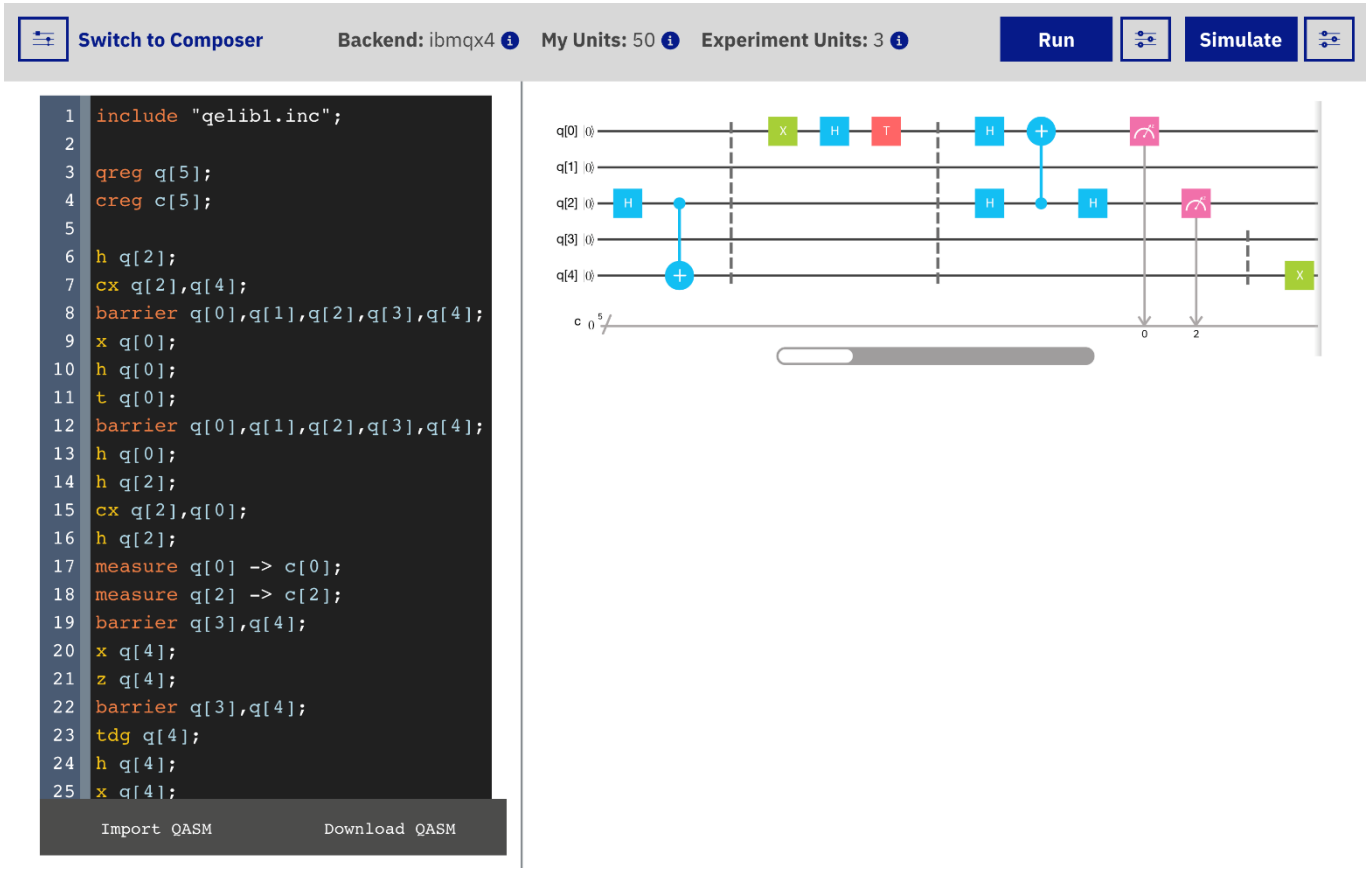


Figure 4-2. The IBM Quantum Experience QASM online editor[2]

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=4-1*

Example 4-1. Teleport and verify

```
include "qelib1.inc";
qreg q[5];
creg c[5];

// Step 1: Create an entangled pair
h q[2];
cx q[2],q[4];
barrier q[0],q[1],q[2],q[3],q[4];

// Step 2: Prepare a payload
x q[0];
h q[0];
t q[0];
barrier q[0],q[1],q[2],q[3],q[4];

// Step 3: Send
h q[0];
h q[2];
cx q[2],q[0];
h q[2];
measure q[0] -> c[0];
```

```
measure q[2] -> c[2];
barrier q[3],q[4];

// Step 4: Receive
x q[4];
z q[4];
barrier q[3],q[4];

// Step 5: Verify
tdg q[4];
h q[4];
x q[4];
measure q[4] -> c[4];
```

Before we get into details it's worth making some clarifying points. Firstly, by *quantum teleportation*we mean the ability to transport the precise state (i.e. magnitudes and relative phase) of one qubit to another. We are making a perfect copy of all the information contained in the first qubit and putting it in the second qubit. If you recall, quantum information cannot be replicated, hence the information on the first qubit is destroyed when we teleport onto the second one. Since a quantum description is the most complete description of a physical object, this is actually precisely what you might colloquially think of as teleportation - only at the quantum level.[3]

With that out the way, let's teleport! The textbook introduction to quantum teleportation begins with a story that goes like this: A pair of qubits in an *entangled* state are shared between two parties, Alice and Bob (physicists take delight in anthropomorphizing the alphabet). These entangled qubits are a resource that Alice will use to teleport the state of some other qubit to Bob. So teleportation involves three qubits: the *payload* qubit that Alice wants to teleport, and an entangled pair of qubits that she shares with Bob, having operational access to one each (and that acts a bit like a quantum ethernet cable). Alice prepares her payload and then, using HAD and CNOT operations, she entangles this payload qubit with the other qubit she holds (which is *already* entangled with Bob's qubit). She then destroys both her payload and entangled qubit using READ operations. The results from these READs yield two conventional bits of information that she sends to Bob. Since these are *bits*, rather than qubits, she can use a *conventional* ethernet cable for this part. Using those two bits, Bob performs some single-qubit operations on his half of the entangled pair of qubits that he shared with Alice and, lo and behold, it *becomes* the payload qubit that Alice intended to send (and destroyed in the process).

Before we see and run the detailed protocol of quantum operations allowing Alice and Bob to achieve this, you may have a concern in need of addressing. "Hang on…" (we imagine you saying), "…if Alice is sending Bob conventional information through an ethernet cable…" (you continue), "…then surely this isn't that impressive at all". Excellent observation! It is certainly true that quantum teleportation crucially relies on the transmission of conventional bits for its success, it is this very fact that can assure you that we are not violating Einstein's theory of relativity by signalling faster than the speed of light (in case you were worrying about that too!). We've already seen that the magnitudes and relative phases needed to fully describe a general qubit can take on a continuum of values. Crucially, this protocol works even in the case when Alice does not know the value of her qubit. This is particularly important since it is impossible to determine the magnitude and relative phase of a qubit in an unknown state[4]. And yet - with the help of an entangled pair of qubits - only two conventional bits were needed to effectively transmit the precise configuration of Alice's qubit, whatever its amplitudes were - to potentially an infinite number of bits of precision!

So how do we engineer this magic? Here's the full protocol. The below circuit shows the operations we need to employ on three involved qubits. All the operations they need have been introduced in Chapter 2 and Chapter 3.

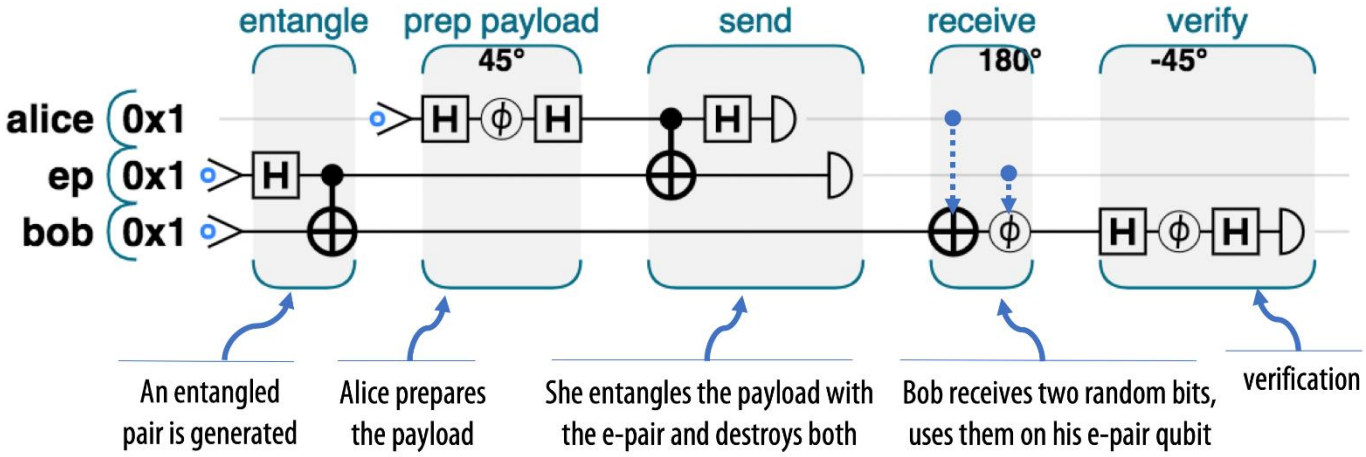


Figure 4-3. Complete teleportation circut

If you paste the code for this circuit from Example 4-1 into the IBM QX system, the user interface will display the circuit as shown for QCEngine in Figure 4-4. Note that this is exactly the same program as we've shown in Figure 4-3, just displayed slightly differently. The quantum gate notation we've been using is standardized, and so you can expect to find it used outside this book.[5]

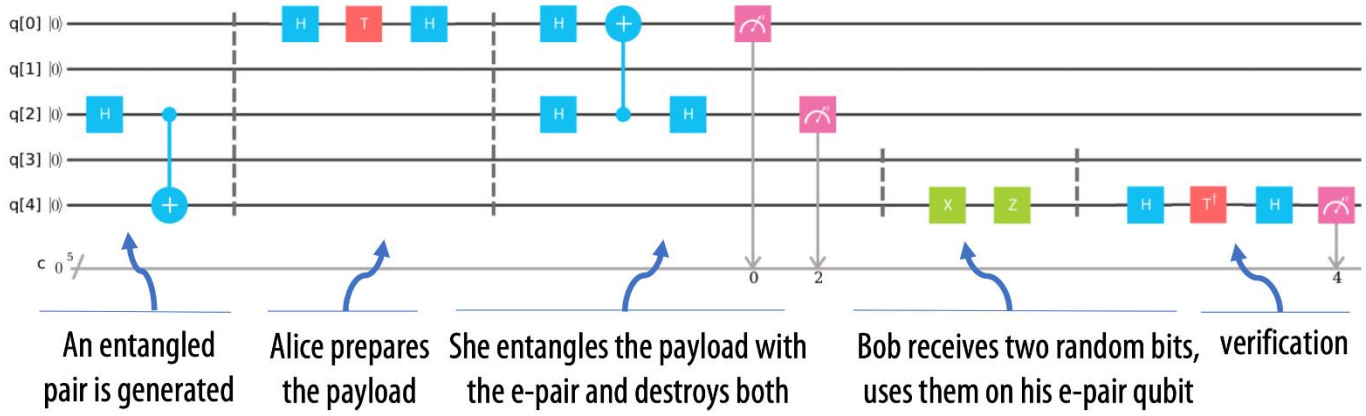


Figure 4-4. Teleportation circut in IBM QX[6]

When you click **Run**, IBM will run your program 1024 times (this is adjustable) and when it's finished, display some statistics describing all of these runs. Once you've run the program you can expect to find something similar (although not identical) to the bar-chart shown below:

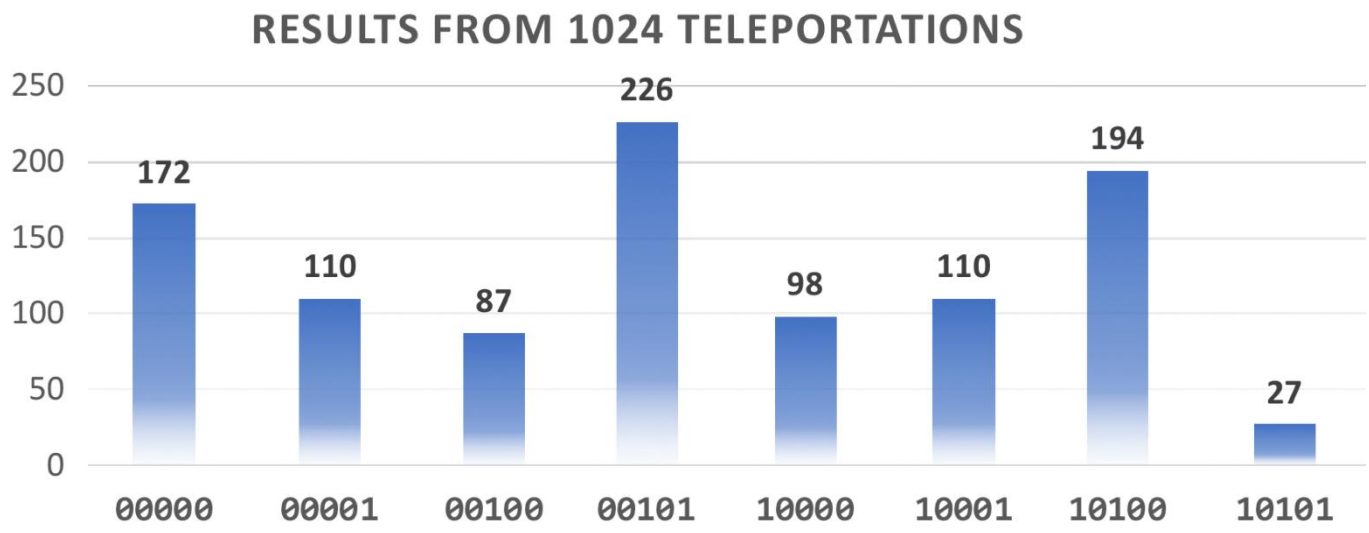


Figure 4-5. Teleportation success? How can we tell?

Success? Maybe! In order to know how to read and interpret these results, let's walk through each step in the QPU program with a little more detail, using circle-notation to visualize what's happening to our qubits7(*http://oreilly-qc.github.io?p=4-1*).].

WARNING

At the time of writing the circuits and results displayed by IBM QX show what's happening with all *five* qubits from the QPU, even if we're not using them all. This is why there are two empty *qubit lines* in the circuit shown in Figure 4-4, and why the bars showing results for each output in Figure 4-5 are labelled with a 5 bit (rather than 3 bit) binary number - even though only the $2^3 = 8$ bars corresponding to the three qubits we used are actually shown.

Program Step-by-step Walkthrough

Since we use three qubits in our teleportation example, a full description of all the qubits involved will need $2^3 = 8$ circles (one for each possible combination of three bits). We'll also arrange these eight circles in two rows, which helps us to visualize the effects that operations have on the three different qubits involved. In the circle notation in Figure 4-6 we've labelled each row or column of circles that correspond to a given qubit having a particular value - you can check that these labels are correct by considering the binary value of the register that each circle corresponds to.

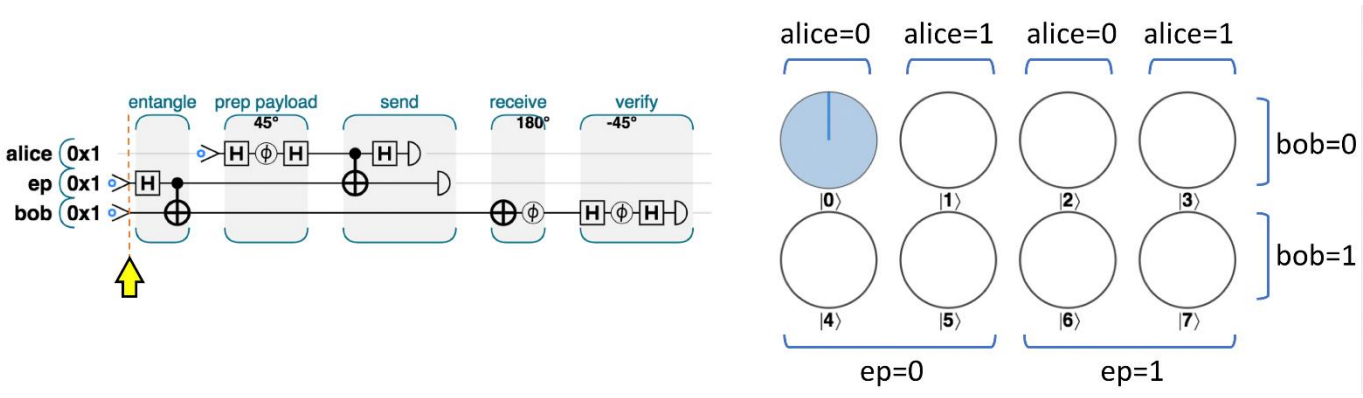


Figure 4-6. The complete teleport-and-verify program

NOTE

When dealing with multi-qubit registers we'll often arrange our circle notation in rows and columns as we do in Figure 4-6. This is always a useful way of more quickly spotting the behavior of individual qubits by picking out the relevant rows or columns.

At the beginning of the program, all three qubits are initialized to state |0⟩, as indicated in Figure 4-6 - the only possible value is the one where `alice=0` and `ep=0` and `bob=0`.

STEP 1: CREATE AN ENTANGLED PAIR

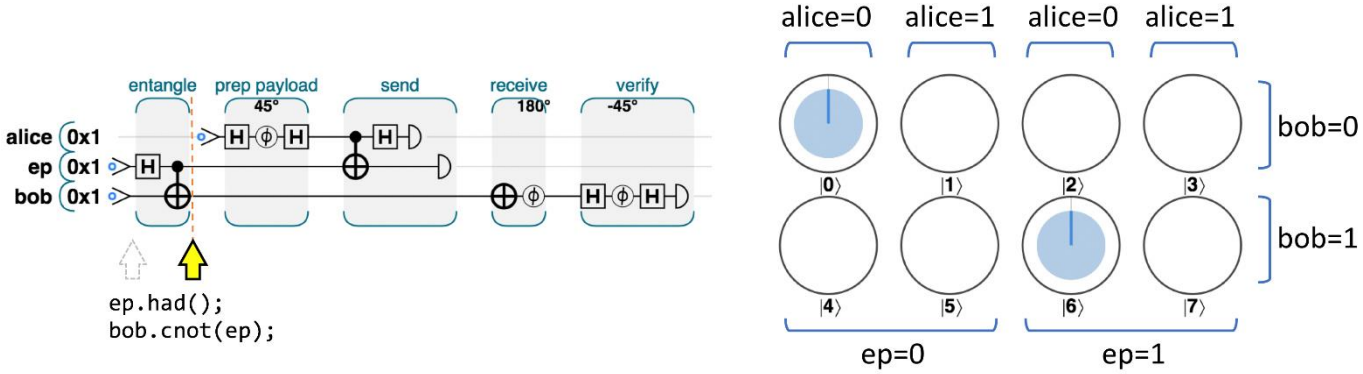


Figure 4-7. Step 1: Create an entangled pair

The first task we need to undertake to be able to teleport is the establishing of an entangled link. The HAD and CNOT combination that does this for us is the same process we used in Chapter 3 to create the specially named *Bell pair* entangled state of two qubits. We can readily see from the circle notation in Figure 4-7 that if we read `bob` and `ep`, the values are 50/50 random, but guaranteed to match each other, à la entanglement.

STEP 2: PREPARE THE PAYLOAD

Having established an entanglement link, Alice can prepare her payload to be sent. How she prepares it depends of course on the nature of the (quantum) information that she wants to send to Bob. She might write a value to transform it, entangle it with some other QPU data, or even receive her payload qubit from a previous computation in some entirely separate part of her QPU.

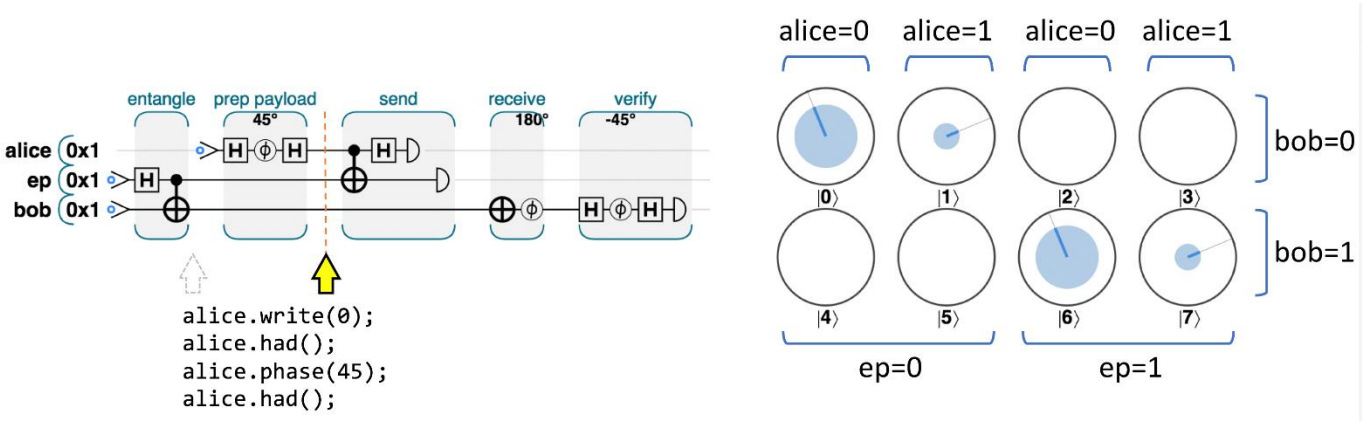


Figure 4-8. Step 2: Prepare the payload

For our example here we'll disappoint Alice by asking her to prepare a particularly simple payload qubit - using only HAD and PHASE operations. This has the benefit of producing a payload with a readily decipherable circle-notation pattern - as shown in Figure 4-8. We can see that the `bob` and `ep`qubits are still dependent on one another (the circles only have magnitudes for values corresponding to them having equal values). We can also see that the value of `alice` is not dependent on either of the other two qubits, and furthermore that her payload preparation produced a qubit which is 85.4% zero and 14.6% one, with a relative phase of -90 degrees (the circles corresponding to `alice`=1 are at 90 degrees to the **right** of the `alice`=0 circles).

STEP 3.1: LINK THE PAYLOAD TO THE ENTANGLED PAIR

In Chapter 2 we saw that the conditional nature of the CNOT operation can entangle the states of two qubits. Alice now uses this fact to entangle her payload qubit with her half of the entangled pair that she shares with Bob. She does this simply by performing a CNOT between the payload and her part of the entangled pair. In terms of circle notation this action swaps circles around as shown in Figure 4-9.

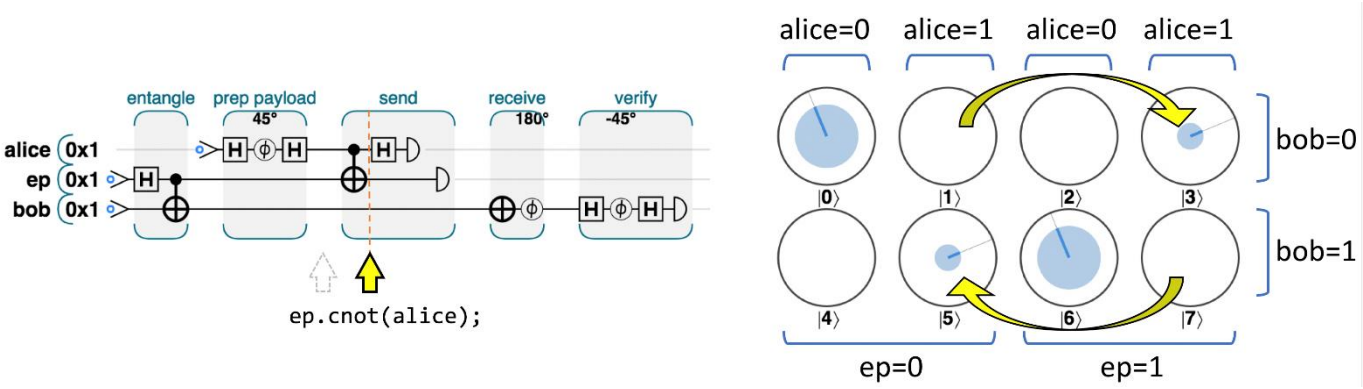


Figure 4-9. Step 3.1: Link the payload to the network

Now that we have *multiple* entangled states, there's the potential for a little confusion - so let's be clear. Alice and Bob *already* each held one of two entangled qubits (produced in Step 1). Now Alice has entangled *another* (payload) qubit onto her half of this (already entangled) pair. Intuitively we can see that in some sense Alice has, by proxy, now linked her payload to Bob's half of the entangled pair - although her payload qubit is still unchanged, its READ results will be logically linked with

those of the other two qubits. We can see this link in circle notation since the state vector only contains entries where the XOR of all three qubits is zero. Formerly this was true of `ep` and `bob`, but now it is true for all three as a *three qubit* entangled group.

STEP 3.2: PUT THE PAYLOAD INTO A SUPERPOSITION

To make this link that she's created for her payload actually useful, Alice needs to finish by performing a HAD operation on her payload, shown in Figure 4-10.

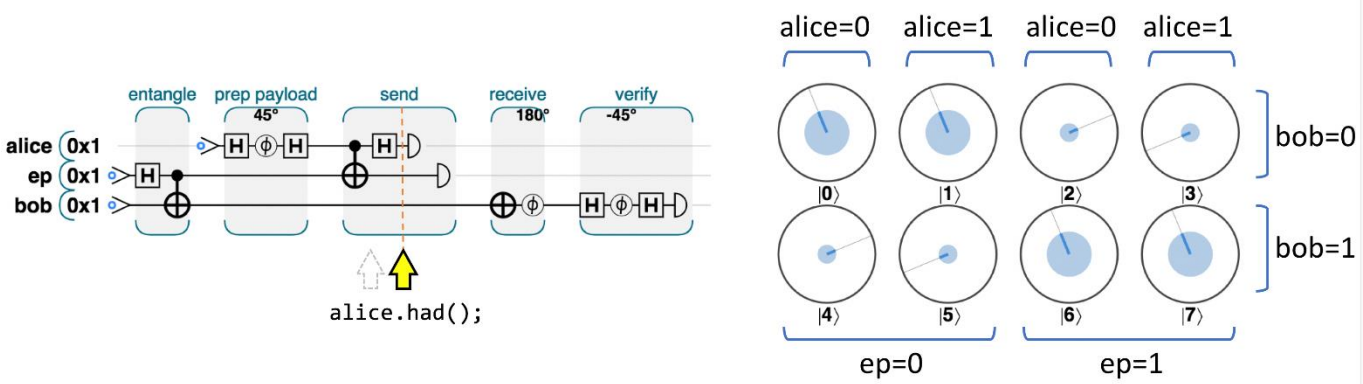


Figure 4-10. Step 3.2: Put the payload into a superposition

To see why Alice needed this HAD, take a look at the circle notation of the three qubits states shown above. In each **column** is a pair of circles, showing a qubit that Bob might receive (precisely which one he *would* receive depends on the results of the READs that Alice performs - as we'll see shortly). Interestingly we can see that the four potential states Bob could receive are all different variations on Alice's original payload. In the first column (where `alice=0` and `ep=0`), we have Alice's payload, exactly as she prepared it. In the second column, we have the same thing, except with a PHASE(180) applied. In the third column, we see the correct payload, but with a NOT having been applied to it (|0⟩ and |1⟩ are flipped). Finally, the last column is both flipped and phase shifted (i.e. a PHASE(180) followed by a NOT). Applying this HAD has allowed Alice to maneuver the state of Bob's qubit closer to that of her payload.

Looking at the magnitudes in Figure 4-10 reveals that the result of a READ on any or all of the three qubits is 50/50 random, and the outcome of each is ***completely independent*** of the others. So although there is clearly information contained in the entanglement between these qubits, reading each qubit individually would result in random values[8].

STEP 3.3: READ BOTH SEND QUBITS

Next Alice performs a READ operation on her two qubits (the payload and her half of the entangled pair she shares with Bob). This READ irrevocably destroys both these qubits. You may wonder why Alice bothers to do this. Why not just leave her payload as is, and teleport a *copy* to Bob? As we'll see, it turns out that the results of this unavoidably destructive READ operation are crucial for the teleportation protocol to work. Rather than an inconvenience, this turns out to be a fundamental fact about the states of qubits - you can teleport, but never copy them. When teleporting, we **must** destroy the original.

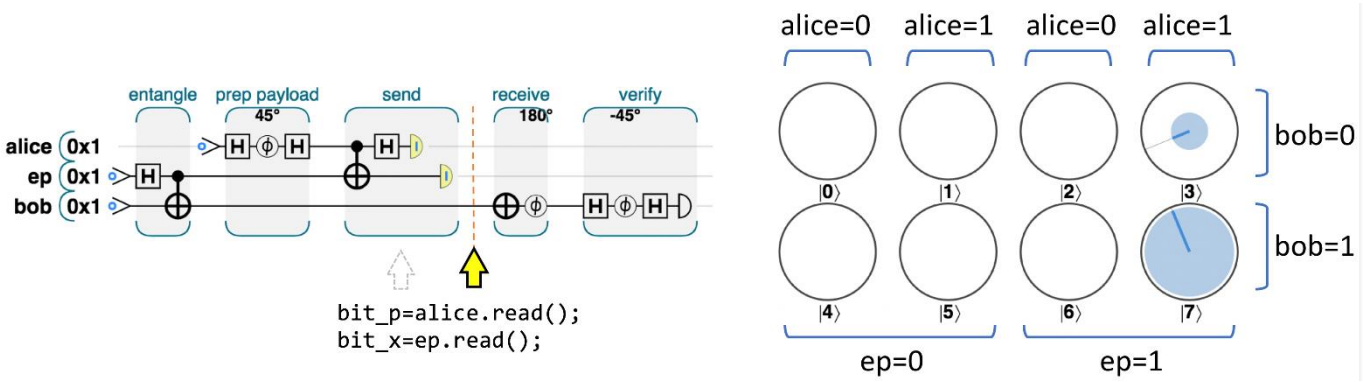


Figure 4-11. Step 3.3: READ both send qubits

Above, Alice performs these READ operations on her payload and her half of the entangled pair. This operation returns two bits, which will help Bob to reconstruct Alice's payload on his qubit.

In terms of circle notation, Figure 4-11 shows that by reading the values Alice has really selected one column (which one she gets being dependent on the random READ results), resulting in the circles outside this column being set to zero.

STEP 4: RECEIVE AND TRANSFORM

We've already seen when discussing Step 3.2 that Bob's qubit could end up in one of four states - each of which is simply related to Alice's payload by HAD and/or PHASE(180) operations. If Bob could learn which of these four states he possessed, he could apply the necessary operations to convert it back to Alice's original payload. And the two bits Alice has from her READ operations are precisely this information that Bob needs! So at this stage, *Alice picks up the phone and transmits two bits of conventional information to Bob*.

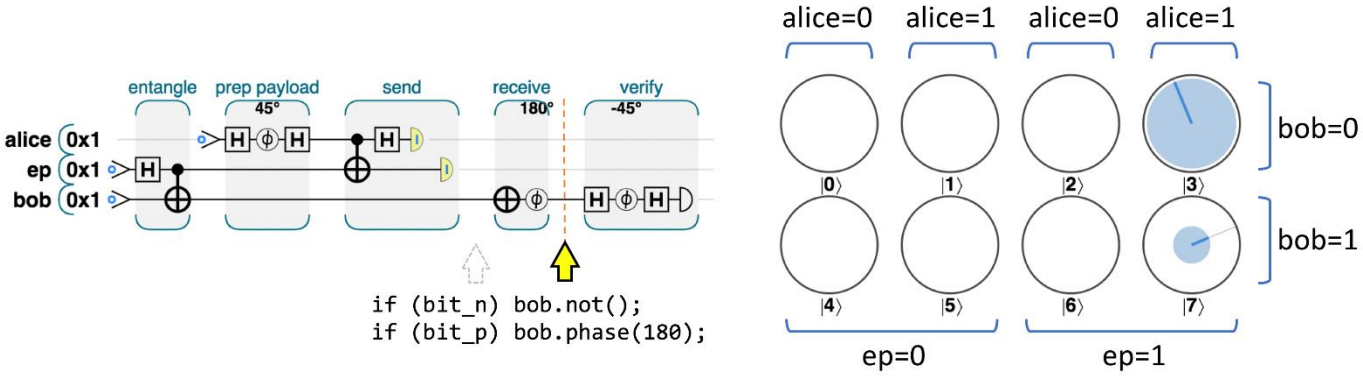


Figure 4-12. Step 4: Receive and transform

Based on which two bits he receives, Bob knows which column from our circle notation view represents his qubit. If the first bit he receives from Alice is 1, he performs a NOT operation on the qubit. Then, if the second bit is 1 he also performs a PHASE(180) - as illustrated above.

This completes the teleportation protocol - Bob now holds a qubit indistinguishable from Alice's initial payload qubit.

NOTE

Like many early QPUs, the current IBM QX hardware does not support the kind of *feed-forward* operation, we need to allow the (completely random) bits from Alice's `READ` to control Bob's actions. This shortcoming can be circumvented by using ***postselection***. In this case, we just assume Bob is asleep at the switch, so he performs the same operation no matter what Alice sends. Then we look at all of the output, and discard any results where Bob did the wrong thing (we ***post-select*** only on the cases where Bob happened to do what he would have with the benefit of information from Alice).

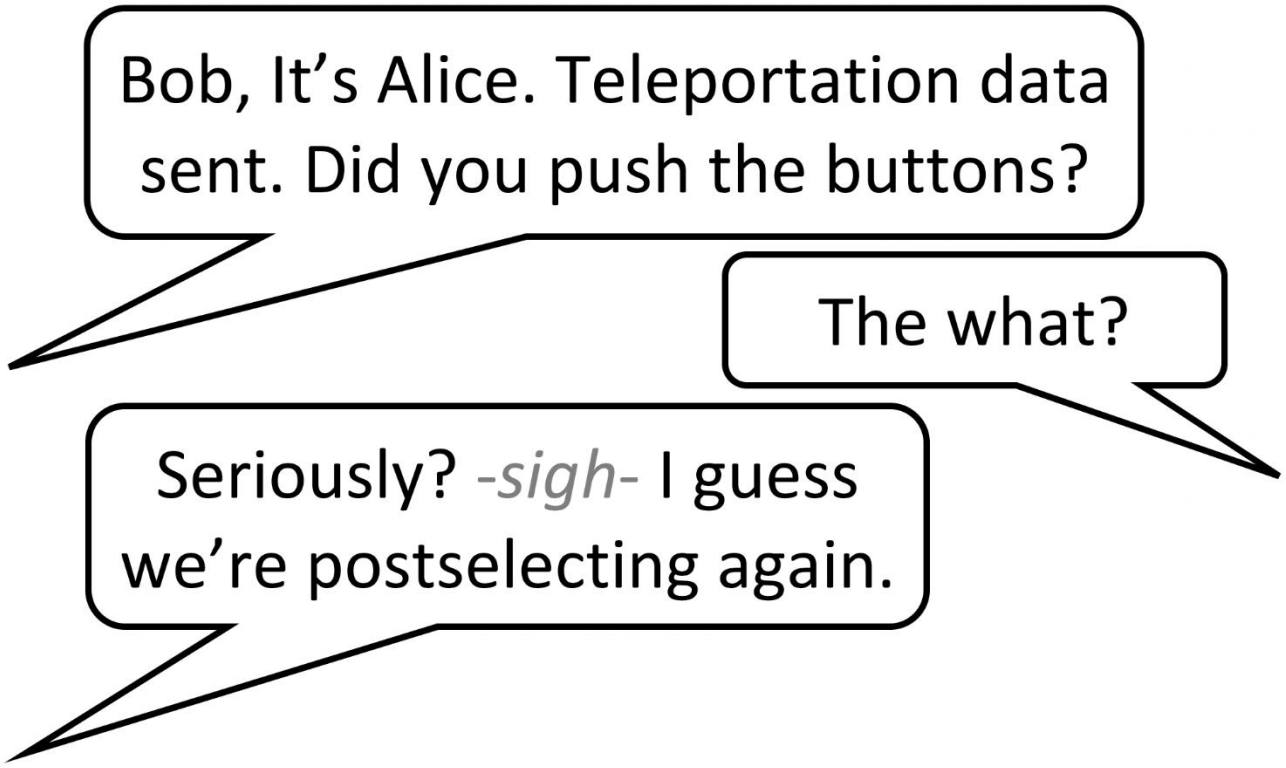


FIGURE 4-13. SOMETIMES, BOB CAN BE THE WEAK LINK.

STEP 5: VERIFY THE RESULT

If Alice and Bob were using this teleportation in serious work, they'd be finished. Bob would take the teleported qubit from Alice and continue to use it in whatever larger quantum application they were working on. So long as they trust their QPU hardware they can rest assured that Bob has the qubit Alice intended.

But, what about cases where we'd like to *verify* that the hardware has teleported a qubit correctly (even if we don't mind destroying the teleported qubit in the process)? Our only option is to READ Bob's final qubit. Of course we can never expect to learn (and therefore verify) the state of his qubit from a single READ, but by repeating the whole teleportation process and making multiple READ's we can start to build up a picture of Bob's state.

In fact the easiest way for us to verify the teleport protocols success on a physical device would be for Bob to run the "prep the payload" steps that Alice performs on a $|0\rangle$ state to create her payload, on his final qubit, only in reverse. If the qubit Bob has truly matches the one Alice sent, this should

leave Bob with a |0⟩ state, and Bob's final verification READ should only ever return a 0. If Bob ever reads this test qubit as nonzero, the teleportation failed. This additional step for verification is shown below:

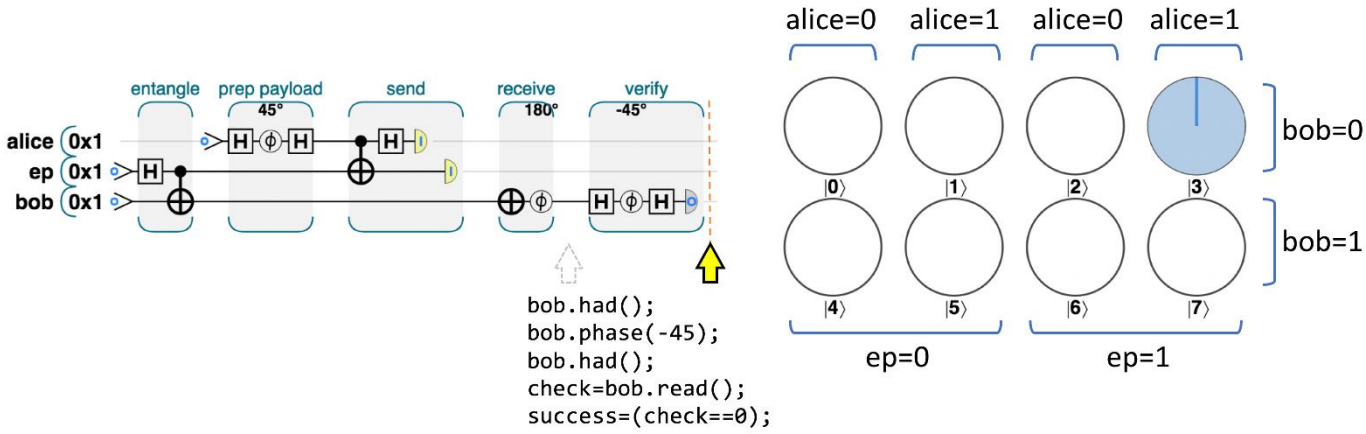


Figure 4-14. Step 5: Verify the result

Even if Alice and Bob are doing serious teleporter work, they will probably want to intersperse their actual teleportations with many verification tests such as these, just to be sure that their QPU is working correctly.

Interpreting the results

Armed with this fuller understanding of the teleportation protocol and its subtleties, let's return to the results obtained from our physical teleportation experiment on the IBM QX. We now have the necessary knowledge now to decode them.

There are three READ operations performed in the entire protocol: two by Alice as part of the teleportation, and one by Bob for verification purposes. The bar chart in Figure 4-5 enumerates the number of times each of the eight possible combinations of these outcomes occurred during 1024 teleportation attempts.

As we noted above, we'll be *postselecting* on the IBM QX for instances where Bob happens to perform the right operations (as if he had acted on Alice's READ results). In the example circuit we gave the IBM QX in Figure 4-4, Bob always performs both a HAD and a PHASE(180) on his qubit - so we need to post-select on cases where Alice's two READ operations gave `11`. This leaves two sets of actually useful results where Bob wasn't *asleep at the wheel*, and his actions were correctly matched up with Alice's READ results:

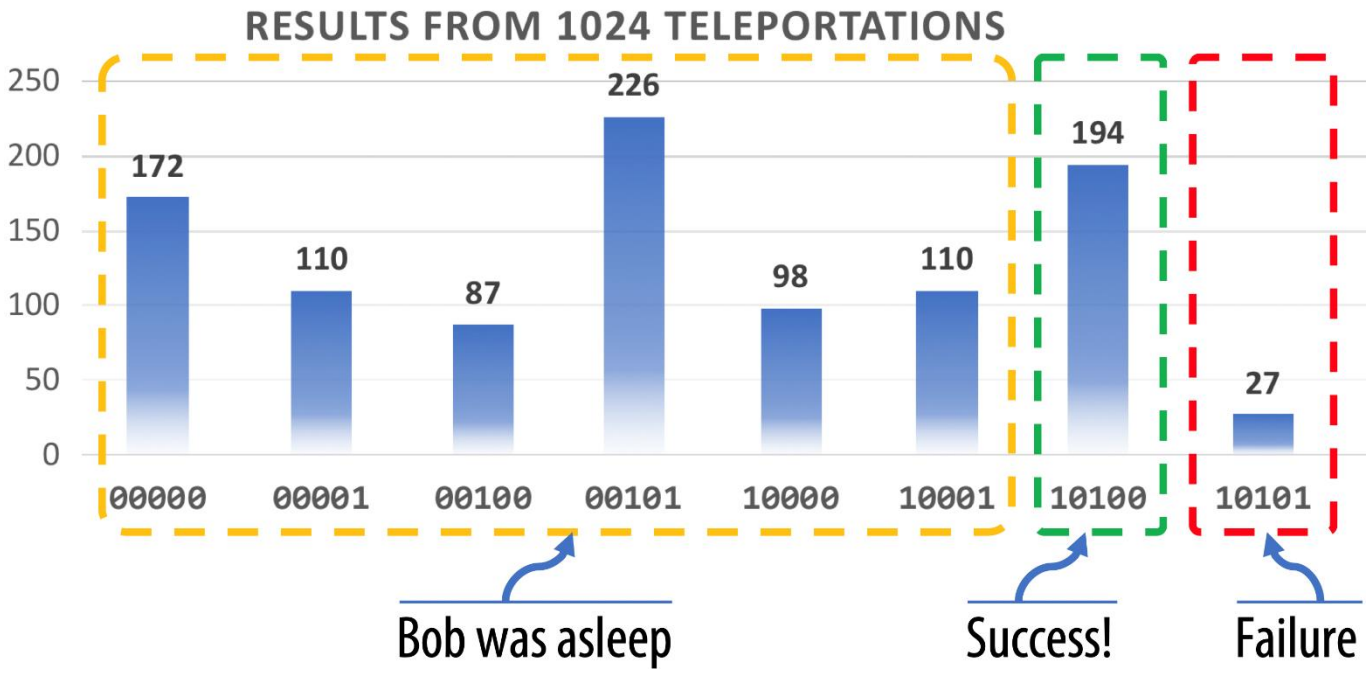


Figure 4-15. Interpreted teleportation results

Of the 221 times where Bob did the correct thing, the teleportation succeeded when Bob's verification READ gave a value of 1 (since he uses the verification step we discussed previously). This means the teleportation succeeded 194 times, and failed 27 times. A success rate of 87.8% is not bad, considering here Alice and Bob are using the best available equipment in 2019. Still, they may think twice before sending anything important.

WARNING

If Bob receives an erroneously teleported qubit which is *almost* like the one Alice sent, this test is likely to report success. Only by running this test many times can we gain confidence that the device is working well.

Fun with famous teleporter accidents

As science fiction nerds, our personal favorite use of teleportation is the classic film *The Fly*. Both the original from 1958 and the modern 1986 Jeff Goldblum version feature an error-prone teleportation experiment. After the protagonist's cat fails to teleport correctly, he decides that the next logical step is to try it himself, but without his knowledge a fly has gotten into the chamber with him.

Even exploring the *possibility* of teleporting actual insects is way *way* outside the scope of this book. To try and make it up to you, we've put together some fun sample code to teleport a quantum *image* of a fly, with a small amount of error. The code sample can be run online at *http://oreilly-qc.github.io?p=4-2*, and the horrifying result it produces is shown in Figure 4-16.

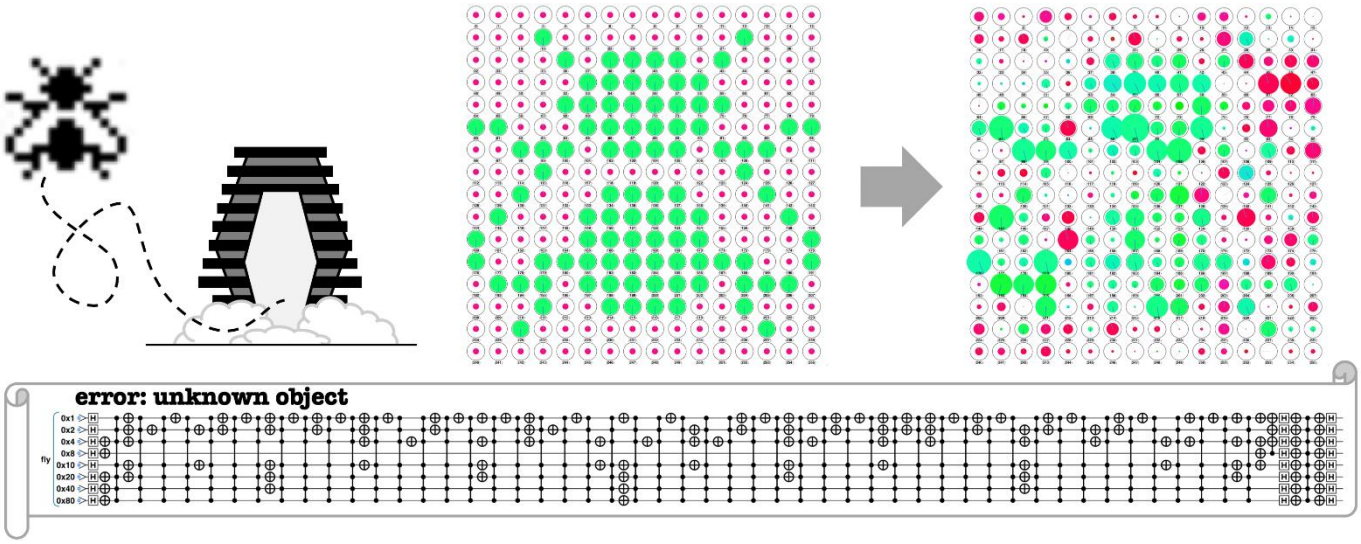


Figure 4-16. Be afraid. Be very afraid.

Gate teleportation

A really interesting feature of quantum teleportation is that we can not only teleport *information*, but we can also teleport *gates*. This means that during the course of a computation, different agents can have access to the qubits and the quantum gates and the computation can still be performed. This might be sounding a little bit esoteric, so let's look at a couple of simple examples to see how it works. You will be surprised to see it's only a small addition to the material we have covered so far!

Quantum Computing in the cloud

Imagine the following scenario: Alice wants to perform a gate on her qubit, however she only has access to a restricted number of operations (she is missing `PHASE`) and cannot do the gate she wants[9]. Luckily for her, she shares an entangled pair of qubits with Bob, and he is willing to do the computation for her. But first he needs the qubit that Alice has.

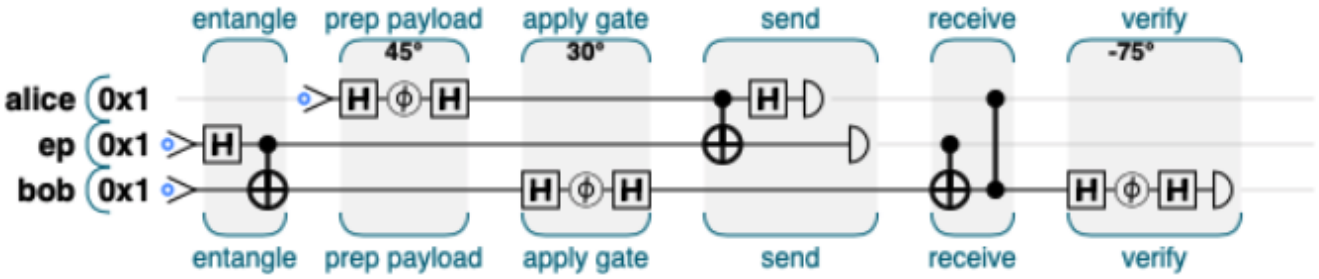


Figure 4-17. Quantum Computing in the cloud

It turns out that they can use the simple qubit teleportation protocol from Figure 4-3 with one small modification: Bob needs to apply the gate that Alice wants to his half of the entangled pair, and then Alice simply teleports her qubit to Bob through their entanglement channel. Bob still has to apply the corrections necessary for qubit teleportation, but at the end of the procedure, Bob will be left holding Alice's qubit with the gate applied to it, exactly as she wanted! Now Bob can teleport the

qubit back to Alice (through another entangled pair) or he can apply more gates to it following Alice's instructions.

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=4-3*

Example 4-2. *Quantum Computing in the cloud*

```
qc.reset(3);
var alice = qint.new(1, 'alice');
var ep    = qint.new(1, 'ep');
var bob   = qint.new(1, 'bob');

// Initialize and entangle
ep.write(0);
bob.write(0);
ep.had();
bob.cnot(ep);

// Alice prep payload
alice.write(0);
alice.had();
alice.phase(45);
alice.had();

// Send
ep.cnot(alice);
alice.had();
alice.read();
ep.read();

// Apply gate [ej TODO: Check code order issue]
bob.had();
bob.phase(30);
bob.had();

// Receive
bob.cnot(ep);
bob.cz(alice);

// Verify
bob.had();
bob.phase(-45-30);
bob.had();
bob.read();
```

It is easy to see the potential of this procedure to do quantum computing in the cloud. At no point in this procedure does Bob need to know what the information stored in Alice's qubit is to perform the computation, and at the same time, Alice can have access to all possible computational procedures on her quantum information while only being able to do very simple operations herself! Of course, for a full quantum computing in the cloud there needs to be considerations of security, as well as verifications that the sever has performed the operation correctly. Those schemes are outside the scope of the book, but in [Link to Come] we mention some references to find more about them.

Teleporting entanglement

So far we have used the teleportation protocol to communicate information, as well as to use remote computing capabilities. There's one last trick we can do with teleportation that can become quite handy when doing complex quantum computation: we can use teleportation to create entanglement between qubits[10]. This can be applied for the case where Alice and Bob want to do a joint computation on some of their qubits by applying an entangling gate, but they are far away, so they can only apply gates locally to qubits they have in their machines. What we will see in Figure 4-18 is that they can use an entangled pair of qubits to create entanglement between qubits in the distant labs by only doing local operations!

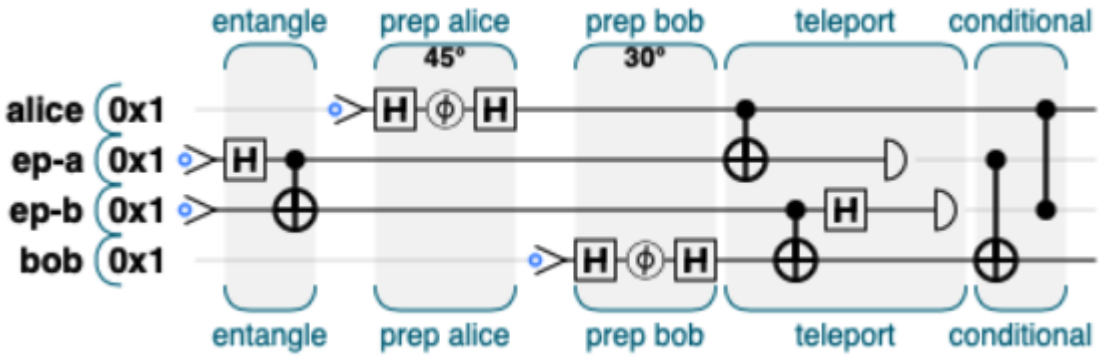


Figure 4-18. Teleporting entanglement

SAMPLE CODE

Run this sample online at *http://oreilly-qc.github.io?p=4-4*

***Example 4-3.** Teleporting entanglement*

```
qc.reset(4);
var alice  = qint.new(1, 'alice');
var ep1    = qint.new(1, 'ep-a');
var ep2    = qint.new(1, 'ep-b');
var bob    = qint.new(1, 'bob');

// Initialize and entangle
ep1.write(0);
ep2.write(0);
ep1.had();
ep2.cnot(ep1);

// Prep Alice
alice.write(0);
alice.had();
alice.phase(45);
alice.had();

// Prep Bob
bob.write(0);
bob.had();
bob.phase(30);
bob.had();

// Teleport
```

```
ep1.cnot(alice);
bob.cnot(ep2);
ep2.had();
r1 = ep1.read();
r2 = ep2.read();

// Conditional
bob.cnot(ep1);
alice.cz(ep2);
```

Alice and Bob share one entangled pair of qubits (**ep-a** and **ep-b**), and each of them has an additional computational qubit. Both of them prepare their computational qubits in some state they want (which can be the result of a previous computation) and then perform a variation of the teleportation protocol we've seen earlier in the chapter. Both Alice and Bob perform a `CNOT` between the qubits they each have, Alice uses her computational qubit as control while Bob uses his computational qubit as target. Bob then applies a `HAD` gate on **ep-b** and they then both measure the qubits from the entangled pair. Notice that, before the `READ` operation, they have created an entangled state of four qubits, by choosing the right place to apply `HAD` before measurement, they are able to retain the entanglement between their computational qubits. As a fun exercise, try removing the `HAD` operation before reading **ep-b** and see whether you can tell what's happening!

How is teleportation actually used?

Working through the teleportation protocol has been a good way for us to get some practical experience using QPU operations and circle notation, but it has also taught us an important lesson about entanglement that's worth emphasizing. Operations such as CNOT and CZ between qubits allow us to form an *entanglement network*, along which (quantum) information and operations can travel. Movement along this network can take the information through any direction or distance in time and space. The important caveat, however, is that each time a link is crossed, the entanglement is spent, the original qubit destroyed, and the bridge has been burned. As such, entangled pairs of qubits, sometimes referred to as *ebits* are a consumable resource.

In this sense teleportation is a surprisingly fundamental part of the operation of a QPU - even in straight computational applications that have no obvious *communication* aspect at all. It allows us to shuttle and input information between quantum systems while working around the *no-cloning theorem*constraint. In fact, most of the actual practical use of teleportation is over very short distances within a QPU as an integral part of quantum algorithms[11]. In the examples in upcoming chapters, most quantum operations for two or more qubits function by forming various types of entanglement. The use of these quantum links to perform computation can usually be seen as an application of the general concept of teleportation. Even a single CNOT gate, the most basic two-bit logic function in all of quantum computation, is applied by the hundreds or thousands in any interesting program. Though we may not explicitly acknowledge teleportations role in the algorithms and applications we cover, its plays a fundamental role.

[1] Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

2 Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

3 The caveat of course is that Jeff Goldblum is made up of many, many quantum states, and so teleporting the states of qubits is far removed from any ideas of teleportation as a mode of transportation. In other words, although teleportation is an accuracte description, Lt. Reginald Barclay doesn't have anything to worry about just yet…

4 There are protocols that allow us to do this, but only in the case where we have many identical copies to work with.

5 Gates such as CNOTs and HADs can be combined in different ways to produce the same result. Do you spot the operation that has different decomposition in IBM QX and QCEngine?

6 Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

7 The complete source code for our below examples can be found at [*http://oreilly-qc.github.io?p=4-1*

8 **Quantum Cloaking Device:** Also at this point, there is something *really* interesting about these three qubits. Remember that on a physical quantum computer it is impossible to view the state vector. This means that if right at this moment spies stormed in on Alice and Bob and confiscated all of the qubits and inspected everything, they would simply get three random values, with no correlation at all. In a sense, Alice has hidden her qubit in Hilbert space, where it is safely locked away.

9 This scenario assumes that Alice already has her qubit ready (and does not have to prepare the payload) as well as the entangled pair with Bob, so the example assumes the scenario starts after the preparation of both the entangled pair and the payload. In the code sample, we have to prepare both, but not in the real scenario.

10 The protocol described in this section is also called *entanglement swapping*.

11 Taking the computational capabilities of teleportation to its limits, there's an entire quantum computational architecture based on the premise that we can teleport information and gates between quantum systems. In [Link to Come] we give some useful references to find out more information about this.

About the Authors

Eric Johnston (“EJ”) is the creator of the QCEngine simulator, and also an acrobat and competitive gymnast. He studied Electrical Engineering and Computer Science at U. C. Berkeley, and worked as a researcher in Quantum Engineering at the University of Bristol Centre for Quantum Photonics. EJ worked at Lucasfilm for 24 years as a software engineer for video games and movie effects, along with motion-capture stunt performance.

Nicholas Harrigan Received a PhD for his research into quantum computing and the foundations of quantum mechanics. He’s since been a a data-scientist and a passionately prolific science communicator. Throughout he’s been fascinated by the relationship between physics and computing, and finds inspiration and motivation in the Unix command `yes`

Mercedes Gimeno-Segovia Develops architectures for quantum computing, during her PhD she designed the first photonic quantum architecture compatible with the silicon industry. She is interested in full-stack architectures, from the control interface with hardware devices to high level programming languages. She also plays the violin and is an avid reader and runner.