# Understanding and Simulating Quantum Circuits

Shane B.
shane@wander.dev

***Abstract*—In this paper we present and clarify the mathematical model behind the tool that we have designed for the purposes of simulating quantum computations using the circuit model, through means of an UI-based user interface running on a general-purpose, classical computer (PC). The aims are (1) to provide a high-level overview of the field; (2) explain the basic theoretical concepts as they are implemented in the current version of our tool at this time; (3) provide several visual demonstrations and explanations of simple quantum algorithms, recreated using our UI tool.**

## I. Distinguishing Characteristics of Quantum Computing

Our physical world can be described and predicted using quantum mechanics. In comparison to classical physics, quantum mechanic posits a *probabilistic model* in which we work with the *probability* of an entity being in a given *state*. In this model, on top of allowing a single concrete state like the binary model (exactly 0 or exactly 1) of classical computing, in certain situations in a certain sense, we can interpret a *physical entity* to be in multiple states "simultaneously". And through the application of so-called *operators* we are able to compute some interesting results directly and instantaneously by leveraging the rules of our reality. Let us introduce the quantum version of *state* through the well-known *Schrödinger's Cat* thought experiment, in which we can consider the cat to have a "state of liveness". We can use its "liveness" to represent our "*qubit*" (for *"qantum bit"*) which we denote $q_{cat}$. Initially, the cat is alive, and hence $q_{cat} = 1$ with a 100% probability ($p_{alive} = 1.0$). Subsequently, the cat is placed in a box with a radioactive source. Then, per *quantum mechanics*, the source may or may not cause the cat to die, both with *equal probability*. In the death outcome, which has a probability $p_{dead} = 0.5$, this would result in a final state $q_{cat} = 0$. The alive outcome has $p_{alive} = 0.5$, and would result in a final state $q_{cat} = 1$. Before we *observe* the cat in the box to determine the actual outcome, the cat's state is in *quantum superposition* meaning it can be interpreted as being "dead" and "alive" simultaneously. Upon *observation* (or *measurement*) of the outcome, we will find one of the two definite results, the probability of both given by the probability distribution $[p_{dead}\ p_{alive}]$. In quantum mechanical terms, it is said that *measurement* results in the *collapse of the wave function*, which produces a *definite state*. In the quantum computing, physicists will opt to use simpler entities (*particles* in particular) to represent *qubits*, such as for example *encoding* our qubit state using the orientation of electrons or photons.

A next important quantum mechanical feature for the purposes of achieving computationally interesting results, is that we can take any $n$ such qubits and "link" any two or more of them together (*entangle* them), such that when the state of one of the qubits is manipulated, the state of the other will *simultaneously* change in a predictable way in turn. Once two qubits are entangled, this simultaneous synchronised effect continues to operate at a potentially large distance — this effect being what Albert Einstein described as "spooky action at a distance" [1]. The *entanglement* is preserved until measurement of any of the involved qubits, upon which a single definite result for all involved qubits is produced and any trace of the entanglement is destroyed. This is notably different from the behaviour of bits in classical computers, where a similar "manipulation" would strictly need to be performed in a "separate step" for each bit, such as in a classical sequential algorithm or a classical parallel electronic circuit needing to manipulate both bits using separate logic gates (Appendix A, Figure 10 & 11).

The final and truly novel property of quantum mechanics is that — on top of directly providing the "randomness" required for *probabilistic computing* — it enables us to directly work with *complex numbers* in computing our state-probabilities. The probabilities of states are encoded in complex numbers (called *amplitudes*) and these *decode* to non-complex *probabilities*. Through creative applications of *operators*, complex numbers allow us to achieve interesting effects in our computations, such as allowing the "cancelling" of the probability of select states being true, which would not have been possible solely using probability theory [2].

The *Quantum* model of computation, then, is built on the idea of directly leveraging these interesting properties of the physical, quantum mechanical world we inhabit exploiting them to the benefit of our physical hardware and algorithms, expanding upon what we can directly compute with classical bits. Within this system, a single *"quantum algorithm"* for solving a specific computational problem encompasses: a particular number $n$ of *qubits*; their initial *states* (or values); and the *operations* performed on the qubits in a well-defined order, which determine the final *probability distribution* for the final states of the qubits, right before *measurement*. A *"quantum computer"* respectively denotes the combination of all such hardware components necessary execute a quantum algorithm. There are various means of representing a quantum algorithm. The most common of such is the *"quantum circuit"* [3]–[5], which is the subject of the tool discussed in this paper.

## II. Anticipated role in Computer Hardware Architecture Model

The ultimate purpose of quantum computing is to leverage the properties of our quantum mechanical world as a "shortcut" to achieve more efficient computations for certain problems. Quantum computers would not necessarily enable us to compute anything "new" — in the sense that any algorithm we may design using quantum computers can also be performed using a classical computer, though perhaps less efficiently. Any quantum computer is Turing-Complete [6], hence is able to execute any classical computer algorithm.

A likely application of quantum computers in our theoretical model of what makes a modern computer would not be to replace classical computers as we currently know them, but as a module within a classical computing system, similar to the *FPU* [7] or *GPU*. GPUs are a relevant comparison, as they are optimized for the specific problem of *parallel computation* and are *programmable*, enabling *parallel computing algorithms* within their own specific paradigm [8], [9].

In the future, provided we can solve the challenging problem of physically engineering a quantum computer that can accurately reflect the theory, as we have achieved with silicon semiconductor chips — which is an active area of research [10] — then we may find ourselves in a world where quantum computers are used as designated and programmable *key components* within various classical computer systems — called a *QPU* (*Quantum Processing Unit*) [11].

## III. Real-World Applications

On a high level, we can distinguish about three broad categories of applications for quantum computing technology. These categories may not be exhaustively representative of all current areas of research.

The first and most obvious use-case for quantum computers is modelling physical and chemical processes directly. Currently, physicists and chemists trying to model large, complex systems within their domains are required to translate said processes in classical computing terms. Often these relations cannot be expressed directly in classical terms, and need to be approximated when converted to classical computing terms. Hence, the quantum mechanical interactions between the objects within some of these processes also have to be computed in classical terms. All these translations form layers of overhead that become incredibly slow at scale, such that in many cases the scientists are required to either use smaller models instead of accurately large ones (which do not produce as accurate results) or depend on expensive supercomputers (which are only helpful up to a point, for very large systems of objects). In such cases, quantum computers would provide a benefit even without needing to consider any complex *quantum algorithms*, as enabling such scientists to model their problem directly *alone* would serve as a large improvement, having a direct impact on commercial areas of research such as material discovery and drug development [12].

Secondly, the next category consists of use-cases leveraging quantum computers in combination with the advanced *quantum algorithms* they would enable. One example of a paradigm-shifting algorithm is *Shor's Algorithm for Prime Factoring* [13]. Currently, a lot of public-key cryptographic systems such as *Finite Field Diffie-Hellman* [14], *RSA* [15] and *ECC*[1] rely on prime factoring as their "mathematically hard problem" of choice[2]. Classically, our best known means of finding the prime factors of an arbitrary integer $n$ — the *General Number Field Sieve* algorithm (*GNFS*) [17] — has an asymptotic time complexity of $O(e^{1.9(\log n)^{1/3}(\log \log n)^{2/3}})$. *Shor's Algorithm* would enable us to compute the prime factors with a *quantum* time complexity[3] of $O((\log n)^2(\log \log n)(\log \log \log n))$ if using *fast multiplication*, or potentially even in $O((\log n)^2(\log \log n))$ if using *Harvey and Van Der Hoven multiplication* [18]. This drastic reduction in time complexity would render these systems far less secure than expected. A visual comparison of these time complexities is given in Figure 12 of Appendix B. Extensive research has been conducted on alternative "hard problems" that are not believed to have any efficient quantum solutions — this is the domain of *Post-Quantum Cryptography* [19].

Within this category we can also consider *exhaustive searching* (or *function inversion* more broadly), which classically has a runtime complexity of $O(n)$. Using the quantum *Grover's algorithm* [20] we can improve this to $O(\sqrt{n})$. Outside of obvious database search applications, this algorithm also provides a general asymptotic improvement for optimisation problems.

Finally, among other solutions directly leveraging quantum mechanical properties, and opposite to Post-Quantum Cryptography, we can find *Quantum Cryptography*. Central to this particular field is *quantum key distribution* (*QKD*) — a protocol that is based on exploiting the fact measuring a quantum superpositon is a destructive operation. Any attempt to learn (and thus *measure*) a non-basis state will result in a change in the state: a definite value being produced according to the probability distribution due to *wave function collapse*, with all information regarding the superposition being "lost" at that instant. This property can be employed to determine if a potential 3rd party attempted to eavesdrop a "quantum communication channel", as the intended recipient of a "quantum message" would then determine that the measurement happened earlier than intended, and take action to work around the problem [2], [21]. Further impactful uses for quantum computing are still an open question, with a lot of investment being made to develop use cases within various domains [22].

[1]ECC in particular relying on a method *close* but more difficult than prime factoring, but which is still reducible to factoring [2], [16]

[2]Cryptographic systems will generally be designed around such *hard problems*: mathematical problems which are easy to solve computationally when sufficient information is known in advance, but hard to "brute-force" without advance knowledge of this key information

[3]A parallel to *quantum time complexity* is the time complexity we compute for *parallel algorithms*, often expressed in terms of the number of processors. *Quantum time complexity* generally doesn't always introduce any new terms, instead simply meaning "if executed on a quantum computer"

## IV. Why Simulate Quantum Circuits?

At present, we are not yet close to having quantum computers that accurately model the theory. The current "era" of quantum computing is generally referred to as the *noisy intermediate-scale quantum* (*NISQ*) era [23], characterised by quantum processors containing up to $n = 1K$ qubits which are neither *fault-tolerant* nor large enough to achieve *quantum advantage* (or *quantum supremacy*): the ability to demonstrate their predicted theoretical time complexity improvement over a classical algorithm for large $n$ in the real world.

Ongoing research into quantum computing currently focuses on two main areas: (1) the physical implementation of the quantum computing systems, particularly sustaining a theory-consistent state of qubits within arbitrary environments (e.g. minimizing the need for extreme temperature-controlled environments) and minimizing "quantum noise" [24] and (2) designing algorithms within the theoretical quantum model that exploit the benefits of the quantum paradigm of computation, which would be theoretically capable of achieving quantum advantage. The software tool that we have designed would aim to assist in the second of above categories, in that it means to enable the higher-level design of quantum algorithms, without needing to understand the underlying technology implementing the quantum computing system on which the algorithm may be executed.

## V. Software Design & Architecture Summary

The paradigm of implementation of the quantum circuit design tool was a visual, interactive Graphical User Interface (GUI). The main distinguishing feature of our tool, compared to alternatives such as IBM's Qiskit [3] is the ability to design the circuit interactively, through means of cursor-based drag-and-drop actions. Our tool distinguishes itself from Qiskit in particular by avoiding the need to have a working knowledge of Python nor requiring to write any classical computer code.

We have opted for the Python 3 programming language based on the following of its characteristics: (1) it being the standard and widely-known scientific programming language of choice; (2) the OS-independence of the language and its built-in graphical modules; (3) the ability to easily convert datastructures to other widely-used representations such as Qiskit simply by using their existing Python packages; and (4) its support by cloud web service providers [25], allowing reuse of our common modules if deploying on the cloud.

The tool presents the user with a digital "modelling sheet" and the *Basic Quantum Gate Library* ("G-Lib") (Figure 1). The library can be used to recreate the effects of any complex *quantum gate* that is not part of the library, through means of composing (or combining) multiple simple gates, similar to how one might represent some classical logical (binary) gates such as *XOR* using a composition of *AND*, *OR* and *NOT* gates (see Figure 18 of Appendix I).

The code-level implementation of the tool can be described as being composed of two sets of code modules: (1) the "backend" module, providing the data structures to represent a quantum circuit design as well as the logic for interpreting

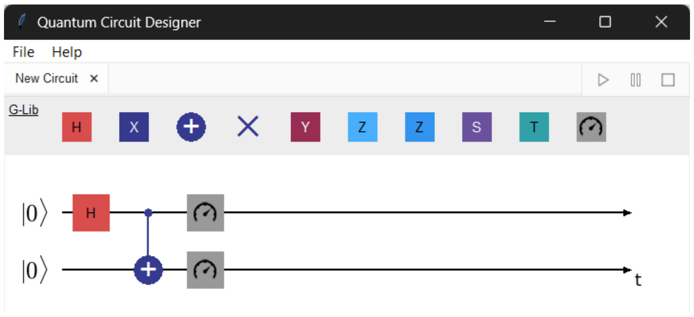


Fig. 1: GUI of the Quantum Circuit Designer, on initial startup

it as the underlying matrix operations it represents, allowing serialisation and deserialization of the data structures, and computing the final probability distribution; and (2) the "front-end" module, supplying the means of rendering the interactive GUI, applying a user's visual manipulations to the backend data structures, and facilitating a clear visualisation of resultant probability distribution.

## VI. Formal Model: Input and Operations

The initial "input state" or *input binary string* to a circuit is typically a *basis state*: a $m = 2^n$ element vector to represent the $n$ (qu)bits of the input string. Inputs are represented using *Bra-ket* notation (also known as *Dirac notation*), which is inherited from the field of physics. In physics, a "Ket" is a $n$-element *complex vertical vector* $\in \mathbb{H}^n$ — that is, it is part of a *complex* Hilbert space rather than just a more "typical" vector $\in \mathbb{R}^n$ — with a notation $|\psi\rangle$. Here, $\psi$ is simply a label, and in our case will typically be a binary string, as in $|01\rangle$. A "Bra" is the *conjugate transpose*[4] of the corresponding Ket, and is thus a *horizontal* vector, written as $\langle\psi|$. Bra-ket notation is useful as for large $n$ it allows us to not need to write out a very large vector containing many elements, particularly exceedingly many 0-elements in $2^n$-sized vectors. One may consider this notation to be equivalent to a high-level programming language function, that produces a vector:

```
enum TYPE { HORIZONTAL, VERTICAL };
function getVectorFor(binNum : string, type : TYPE): vector {
    return // logic to build output vector
}
let ket_01 = getVectorFor("01", TYPE.VERTICAL);
```

For *basis states*, the input binary string is "mapped" to the vector in a straight-forward way. Here we provide an example for the binary string "11", requiring $n = 2$ (qu)bits, and representing the number 3 in binary/base-2. The blue highlight is used for emphasis.

| | |
|---|---|
| Binary string | "11" (2 in *base-2*) |
| Nr of qubits | $n = 2$ (length of the binary string) |
| *Ket* representation | $\lvert 11\rangle$ |
| Nr elements in vector | $m = 2^n = 2^2 = 4$ |

[4]For any matrix, this means we first *transpose* ("mirror" along the diagonal), and then apply *complex conjugation* to every element of the matrix ($a + ib$ becomes $a - ib$)

The corresponding vector is constructed by taking a $m = 2^n$ element *vertical vector*, initialising all elements to zero, except for the element at the (0-indexed) index corresponding to the binary string, counted from top to bottom, which is initialised to 1. Here we do this for index "11" $= 3$ and size $m = 4$:

$$\begin{array}{cc} \text{index} & \text{vector} \\ \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array} & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \equiv |11\rangle \end{array}$$

Bra-ket notation is often combined with another concept, which is also used by the *operators* discussed later and which is often omitted in written Bra-ket formulae: that of the *tensor product* (or *Kronecker product*). A tensor product, typically denoted with the symbol $\otimes$, can be obtained of two matrices of any size (including vectors, which can be interpreted as matrices which have either only 1 row or 1 column) as a generalisation of the following:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1(5) & 1(6) & 2(5) & 2(6) \\ 1(7) & 1(8) & 2(7) & 2(8) \\ 3(5) & 3(6) & 4(5) & 4(6) \\ 3(7) & 3(8) & 4(7) & 4(8) \end{bmatrix}$$

A more elaborate visualisation of a tensor product is provided in Appendix C. One might also write the following, ommitting the $\otimes$ symbol. This also demonstrates a convenient shorthand for building large input strings (here vector indices are added to the left):

$$|1\rangle \otimes |0\rangle \equiv |1\rangle |0\rangle \equiv |1, 0\rangle \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{array}{c} {\scriptstyle 00} \\ {\scriptstyle 01} \\ {\scriptstyle 10} \\ {\scriptstyle 11} \end{array} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \equiv |10\rangle$$

Finally, it is useful to note that all so-called *basis states* are what is defined, in physics, as *pure states*. Considering the case of a *single qubit*, the only (standard) basis states are $|0\rangle$ and $|1\rangle$[5]. For $n = 2$ qubits, the basis states would be a list of *Kets* of all $2^n$ possible binary values $n$ bits can take, namely $|00\rangle, |01\rangle, |10\rangle$ and $|11\rangle$, which are also all pure states.

The opposite to the aforementioned relation does not hold however: not all pure states are basis states. Pure states also encompass *superpositons*: these are non-basis states where, for a *single* qubit, the qubit has a non-zero probability of being in *both* state $|0\rangle$ *and* $|1\rangle$, both probabilities summing to 1. Quantum computing algorithms will exclusively work with *pure states*, typically transforming basis states (classical inputs) into other pure states through the use of operators (usually *quantum logic gates*).

Owing to the fact that the Hilbert space $\mathbb{H}^n$ is complex, all pure states of $n$ qubits can be written as a complex *linear combination* of the basis states. In the case of $n = 1$ we have:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \quad (\alpha, \beta \in \mathbb{C}) \tag{6.1}$$

[5]Alternative "bases" can be considered, such as a basis of $\{|-\rangle, |+\rangle\}$

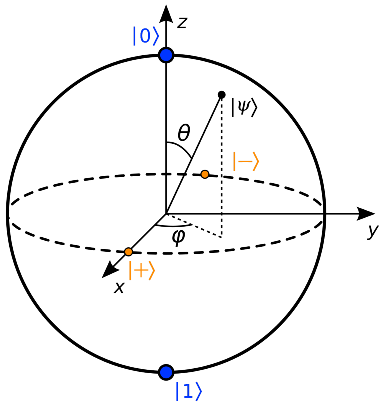


Fig. 2: The Bloch Sphere [26]

Mathematically, using the definition[6] $|x| = abs(x)$, then $\alpha, \beta$ will satisfy $|\alpha|^2 + |\beta|^2 = 1$. Here, $\alpha$ and $\beta$ are called the *amplitudes* of the basis states they are multiplied with. And $|\alpha|^2$ and $|\beta|^2$ are the *probabilities* of their basis states.

A useful concept for visualising a (single) qubit, including the various states it can be in, is the *Bloch Sphere* (Figure 2) — a unit sphere (sphere of $radius = 1$) centred at the origin within a 3D coordinate system. Due to the peculiarities of quantum mechanics, it turns out that any single-qubit state-vector can also be uniquely (re)written as:

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\varphi}\sin(\theta/2)|1\rangle \tag{6.2}$$

With $0 \leq \theta \leq \pi$ and $0 \leq \varphi < 2\pi$ [27]. Noting that now $\alpha$ is always a real number, and $\beta$ a complex number, thus allowing a unique representation for every possible *pure state*. Pure states are all those states (vectors) that point to any point on the *surface* of the sphere, importantly maintaining the property of having $length = 1$, known as the *normalization constraint*. To visualise a state on the Bloch Sphere, given a known pure state $|\psi\rangle$ expressed in terms of (6.1), we solve a system of two linear equations for $\theta$ and $\varphi$ starting from vector equation (6.2). Then, starting from a vector pointing straight up on the Bloch Sphere, we rotate this vector by the obtained angles (in radians): $\theta$ gives the angle between the $zy$ axes, and $\varphi$ gives the angle between the $xy$ axes. An example of such a superpositon is shown visually for one such arbitrary state $|\psi\rangle$, along with for the basis states $|0\rangle$ ($\theta = 0, \varphi = 0$, hence pointing straight up) and $|1\rangle$ ($\theta = \pi, \varphi = 0$, hence pointing straight down) in Figure 2. A solution to the system of linear equations for $|1\rangle$ can be found in Appendix D. ($n > 1$)-qubit systems can be expressed using analogous expressions to (6.1) in terms of their $2^n$ respective basis states, but cannot be as easily visualised geometrically.

Finally and formally, the concrete *probability distribution* of every possible state is given by the *absolute squared values* at all corresponding indices/elements. Formulated another way: if $v$ is a vector and $v[k]$ represents the element of $v$ and index

[6]The "absolute value" is also known as the "modulus". On a complex number $a + ib$, this is $\sqrt{a^2 + b^2}$

$k$ in a 0-indexed indexing scheme — where each element is an *amplitude* that is possibly a complex number — then the probability of *state* $k$ ($k$ taken as a binary string/in base-2) is given by $|v[k]|^2$.

For a sample superpositon of 2 qubits $\frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$:

$$\begin{array}{ccc} \text{index} & \text{state-vector} & \text{probability (of } |index\rangle) \\ \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array} & \begin{bmatrix} 0 \\ 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix} & \begin{array}{c} 0 \\ |\frac{1}{\sqrt{2}}|^2 = 0.5 \\ |\frac{1}{\sqrt{2}}|^2 = 0.5 \\ 0 \end{array} \end{array}$$

The above indicates a probability $p_{01} = 0.5$ of being in state $|01\rangle$ and a probability of $p_{10} = 0.5$ of state $|10\rangle$. Similarly, all *basis states*, having their vector populated with all "0"s with one exception of a single "1"-element, have a 100% probability of being in said initial basis state. The probabilities in any *pure state* will always sum to 1.0, that is, all pure states preserve a *valid probability distribution*.

Additionally, particularly as it concerns physics and the physical implementation of quantum computers, it is important to note that asides from the *pure states*, there are also *mixed states*. In the Bloch Sphere, these mixed states correspond to all such vectors that fall somewhere within the volume of the sphere, being uniquely identified by not only $\theta$ and $\varphi$ but also the radius $r$ that varies in $0 \leq r < 1$, without reaching the surface of the Bloch Sphere as the pure states do. They cannot be represented using formulae 6.1 and 6.2, instead requiring use of the *density operator* [27], which is not covered in this paper.

## VII. Formal Model: Quantum Logic Gates

Of the *operators* previously discussed, within the scope of most quantum circuits, we generally use the so-called *quantum logic gates* (henceforth referred to as *gates*). These gates have interesting properties that ensure we remain in a *pure state* throughout the entirety of our quantum algorithm. The gates provided in our application are part of our so-called "Basic Gate Library", consisting of the most common widely-recognised gates [28] (see Table I of Appendix E), and include all gates needed to create other less-common gates by *composition*.

Gates are applied to a qubit by left-multiplying the *gate matrix* with the current *qubit state vector* (which is initially a basis state). All gates are *unitary matrices* — a definition sourced from linear algebra — which means they are reversible/invertible, complex, square matrices $U$, for which it holds that: $U^*U = UU^* = I$. Here, $U^*$ (written in physics as $U^\dagger$) is the *conjugate transpose* of matrix $U$. This property is of critical importance for the purposes of quantum algorithms, as this is the property that preserves the normalization constraint and thus that at all stages we continue working with a *pure state* and maintain a valid probability distribution. Any such unitary matrix, of the appropriate size, can be considered a valid quantum logic gate.

The gates can be interpreted as "truth tables" or "conditional mapping tables", applied directly to either a *single qubit* or in a "conditional" ("controlled") pattern to multiple qubits. Geometrically, for the case of a *single qubit*, a single-qubit-gate can be seen as rotating the direction the qubit state vector is pointing in along the surface of the Bloch Sphere (Figure 2).

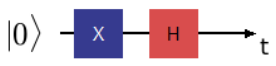


Fig. 3: A single-qubit, two-gate quantum circuit

### A. *Single-Qubit Gate*

The X-gate is the quantum equivalent to the "NOT" logic gate or the "NOT" logical operator ($\neg$) in classical computing. It, along with any single-qubit gate, can be interpreted as mapping (though not necessarily one-to-one) a sole input qubit state to a new state, according to the "NOT" truth table.

Consider the X-gate below annotated with the (0-indexed) row-indices and column-indices in base-2 as seen in the previous section (rows in blue and columns in green) to the left, along with the classical NOT gate truth table to the right:

X (NOT) gate

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} =: X$$

classical NOT

| $A$ | $\neg A$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Given any "input" *pure state*, the a single-qubit gate will "map" to its corresponding output *pure state*. For the single-qubit *basis states*, it maps $|0\rangle \rightarrow |1\rangle$ and $|1\rangle \rightarrow |0\rangle$, as seen in the following example for $|0\rangle$:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

We can consider the initial states, given by the rows, being mapped to the resultant states, given by the corresponding columns: initial state $|0\rangle$ was mapped to resultant state $|1\rangle$ according to the position of the "1" in the "mapping table" (matrix) of the gate, equivalent to the *adjacency matrix* data structure (Appendix F).

In a quantum circuit, we would represent this single-qubit quantum circuit with a single X-gate as in Figure 19 of Appendix J: the initial state of our system of qubits is $|0\rangle$, next comes a "timeline" of the gates that are sequentially applied to the qubit from *left-to-right*. The corresponding matrices are *left-multiplied* to the input vector. An example using two gates (X and then H) is given in (Figure 3), and in the corresponding resultant qubit state vector — a superposition with an equal probability 0.5 for both possible states $|0\rangle$ and $|1\rangle$ — is computed as follows:

$$H(X|0\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \left( \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)$$
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \equiv |-\rangle$$

This result happens to be one of the pair of two common *alternative* basis states, namely the *orthogonal x-basis states*, $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, which point straight along the positive x-axis (for $|+\rangle$) and negative x-axis (for $|-\rangle$) of the Bloch Sphere (Figure 2) respectively.

A notable feature of single-qubit gates demonstrated in Figure 3 is that H (the Hadamard gate) is the only gate capable of producing a superposition[7] out of our entire Basic Gate Library.

Within quantum algorithms, we will generally work with $n$-qubit systems, rather than single-qubit systems. Within such systems, single-qubit gates can be applied to individual qubits to achieve the effect on that (qu)bit within the output binary string. Graphically, this is be represented by a circuit such as in Figure 4. The second line represents the timeline of operations of the second qubit. The input vector in this case is $|0\rangle\,|1\rangle = |01\rangle$, $|0\rangle$ being "qubit 0" and $|1\rangle$ being "qubit 1". Any subsequent qubit added to the circuit below qubit 1 would get numbered $2, 3, \ldots, N-1$ respectively.

In Figure 4, the first the X-gate is applied to qubit 1, toggling it from $|1\rangle$ to $|0\rangle$. Mathematically, we always compute the product starting from any given basis state input vector (in this case the input vector being $|01\rangle$). In order to apply a single-qubit gate to a $n$-gate qubit system, we first create a $2^n \times 2^n$ system-level operation matrix (a *unitary transformation matrix*) that will ensure the gate affects only to the target qubit. This matrix is created by ordering the qubits from qubit 0 (top-most line in the circuit) to qubit $n-1$ (bottom-most line in circuit), knowing the gate was applied to qubit $j$ with $0 \leq j \leq n-1$, we tensor $n-1$ *identity matrices* ($I$) (which do not affect their respective bits) with the corresponding single-qubit-gate at position $j$ from left-to-right, as shown here for gate $X$ (Letting $X_{q_j}$ denote the *unitary transformation matrix* for gate $X$ applied to qubit $j$):

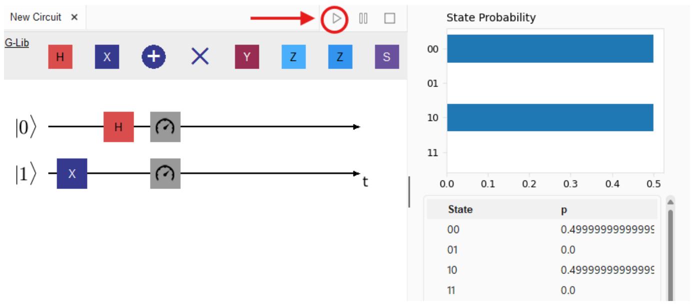
$$X_{q_j} = \underbrace{I \otimes \cdots \otimes I}_{0\ldots(j-1)} \otimes \underbrace{X}_{j} \otimes \underbrace{I \otimes \cdots \otimes I}_{(j+1)\ldots(n-1)} \qquad (6.3)$$

The resultant matrix is left-multiplied to the input-vector:

$$(I \otimes X)\,|01\rangle = \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) |01\rangle$$
$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} |01\rangle = X_1\,|01\rangle = v_1$$

In this case producing intermediate result vector $v_1 = |00\rangle$. Subsequently, the H-gate is expanded to the $4 \times 4$ matrix $H_{q_j = q_0}$ following Equation 6.3, and left-multiplied with $v_1$, to produce the output state vector (annotated with indices to the right):

[7]It is the only gate part of our Basic Gate Library capable of doing this. Other Gates can be considered that have a similar effect, but these other Gates are not strictly necessary for the purposes of quantum computation as their effects can be reproduced using the H-gate



| State | p |
|---|---|
| 00 | 0.499999999999999 |
| 01 | 0.0 |
| 10 | 0.499999999999999 |
| 11 | 0.0 |

Fig. 4: A two-qubit, two single-gate quantum circuit and the result of "running" the circuit

$$(H \otimes I)\,v_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} v_1 = \frac{1}{\sqrt{2}} \begin{array}{c} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array}$$

Analogously, when using the "run" button in our application, the corresponding probability distribution result is produced in a popout right-sidebar as shown in Figure 4.

Another point to consider is the fact that any "empty time slot" on the circuit diagram is equivalent to putting an "identity matrix gate" in that "slot", which would do nothing (result in no change in state for that qubit). Furthermore, the circuits of Figure 4 and Figure 20 in Appendix J are equivalent. This can be seen through the following equations, with the last expression — tensoring the gates to each other directly — being how multiple single-qubit gates placed at the same "time" (same horizontal position aka along the same vertical "column") in the timeline are computed by our application:

$$\left(H_{q_0}\left(X_{q_1}\,|01\rangle\right)\right) = \left(H_{q_0} X_{q_1}\right)|01\rangle = (H \otimes X)\,|01\rangle$$

## B. Multiple-Qubit Gate

Many *multiple-qubit* ($n$-qubit) and *conditional* or *controlled* gates can be distinguished. The idea behind a controlled gate in the case of two-qubit gates is: "If and only if qubit $j$ is in state $|1\rangle$ then apply gate $G_{any}$ to a target qubit $k$". At this time our application only implements two-qubit gates as draggable components, but it should be noted that the behaviours of $(n > 2)$-qubit gates, such as the Toffoli Gate, can be simulated by combining many single-qubit and two-qubit gates in specific way [6]. The subject of this discussion will be the *Controlled NOT (CNOT)* gate (see Table I in Appendix E) which is the "controlled" version of the X-gate. Other "controlled" gates work analogously.

The CNOT matrix is presented here with its row and column indices annotated in blue and green respectively:

$$\begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 1 & 0 & 0 & 0 \\ 01 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 0 & 1 \\ 11 & 0 & 0 & 1 & 0 \end{array} =: \mathit{CNOT}$$

Interpreted as a "mapping table", attention should be paid in particular to the way these mappings happen. Note the first and second row: indices 00, 01. These row indices "map" the to the same column indices. For row indices 10 and 11, however, we see the "1"s placed *not* along the diagonal, indicating a transformative mapping. Summarised, we have:

$$|00\rangle \xrightarrow{\text{CNOT}} |00\rangle\,, |01\rangle \xrightarrow{\text{CNOT}} |01\rangle$$
$$|10\rangle \xrightarrow{\text{CNOT}} |11\rangle\,, |11\rangle \xrightarrow{\text{CNOT}} |10\rangle$$

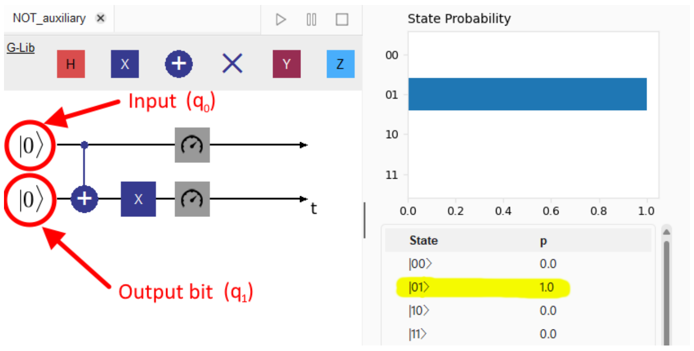
For the first pair of cases, the behaviour of this gate is the same as applying two "identity gates" to both qubits: there is no change in the qubit state vector. For the second pair of cases, when if the left-most qubit is "1", then the value of the second qubit switches from $0 \rightarrow 1$ or $1 \rightarrow 0$, demonstrating the "conditional" or "controlled" behaviour of the Controlled NOT gate. All other "controlled"-series gates work in much the same way.

The other distinct two-qubit gate is the SWAP gate, which unconditionally *swaps* the current values of two distinct qubits. This can likewise be seen from its matrix (Appendix E, Table I).

Figure 21 of Appendix J demonstrates a two-qubit circuit using a H-gate and a CNOT-gate, to create one of the four "Bell States" — the four simplest 2-qubit states giving an example of *quantum entanglement*. This is an interesting example as it shows that through entanglement, we are able to achieve two qubits that are always "in sync", meaning that if we know the value of one of the qubits we would be guaranteed to know the other. This is the idea behind the concept of *quantum teleportation*. If we include the rest of the Bell States (see Appendix H) and now imagine both qubits to become and remain entangled at a (large) distance, combined with the reversibility of the gates, serves the basis of the *quantum communication protocol* known as *superdense coding* [29].

For the purposes of computing the system-level operation matrix (*unitary transformation matrix*) corresponding to a two-qubit gate, similar to the identity matrix tensoring of single-qubit gates discussed in Section VII-A, two strategies can be used to ensure the correct "control-target" relationship of "controlled" gates, particularly in the case of a system of $n > 2$ qubits.

The first of such is by using the SWAP gate/matrix repeatedly, first to place the control and target qubits to the "top" of the circuit (the first two rows at the top), applying the standard "controlled"-series matrix, and moving the "results" on those two qubits back "down" where they were originally situated [6] (Appendix J, Figure 22). The second, which is more friendly to a high-level programming language implementation and which is consequently also the implementation strategy of choice in our application, is by decomposing a "controlled" gate into two parts according to the *algorithmic method* [30]: similarly to Equation 6.3, supposing we have a Controlled NOT/X gate using qubit $j$ for control and targeting qubit $k$, then we build two matrices: (1) the matrix $M_{q_j=|0\rangle}$ that handles the case of "if qubit $j$ is $|0\rangle$, then do nothing to qubit $k$";



| State | p |
|---|---|
| \|00⟩ | 0.0 |
| \|01⟩ | 1.0 |
| \|10⟩ | 0.0 |
| \|11⟩ | 0.0 |

Fig. 5: A circuit outputting the result of a NOT operation into the second qubit

$$M_{q_j=|0\rangle} = I \otimes \cdots \otimes \underbrace{|0\rangle\langle 0|}_{j} \otimes \cdots \otimes \underbrace{I}_{k} \otimes \cdots \otimes I$$

and (2) the matrix $M_{q_j=|1\rangle}$ that handles the case of "if qubit $j$ is $|1\rangle$, then apply the gate to qubit $k$"

$$M_{q_j=|1\rangle} = I \otimes \cdots \otimes \underbrace{|1\rangle\langle 1|}_{j} \otimes \cdots \otimes \underbrace{X}_{k} \otimes \cdots \otimes I$$

The resulting unitary transformation matrix is then given by $CNOT_{j,k} = M_{q_j=|0\rangle} + M_{q_j=|1\rangle}$. This matrix is used to compute the result of the gate at that "time slot". A fully worked out example of the method for a three-qubit system, combined with a single-qubit gate, can be found back in the Appendix (Section G).

## VIII. Classical Circuits using Quantum Circuits

This section gives a brief overview of how we can implement classical logic circuits (Appendix I) using quantum circuits. Here we introduce the concept of quantum circuits often requiring "pre-reserved" *auxiliary/helper*-qubits (used as a "scratch pad") and *output*-qubits to "output results into". A classical (assembly or compiled-to-assembly) computer algorithm, written as a computer program, is represented at a low-level by its corresponding memory layout as shown in Figure 28 of Appendix M. Quantum circuits can be approximately mapped to this model as shown in Figure 29. The following examples do not yet leverage any *superpositions*.

### A. Logical NOT

The X-gate is essentially equivalent to the classical "NOT" gate. Two cases of implementing a "NOT" algorithm can be distinguished: (1) a 1-bit binary input containing the current boolean value, with a single X-gate flipping it (as in Figure 19 of Appendix J); (2) a 2-bit binary input, where the first bit is the "value to be inverted" and the second bit has the result "output" into it (Figure 5).

The second case provides a clear example of explicit *output/auxiliary*-qubits. The input state vector is either $|00\rangle$ or $|10\rangle$. The first qubit, $q_0$, of this string (corresponding to the top line of the circuit) is the actual value to be inverted, and can be either $|1\rangle$ or $|0\rangle$. The second "output"-bit, $q_1$, is always initialised to $|0\rangle$. The "algorithm" consists of first "copying"

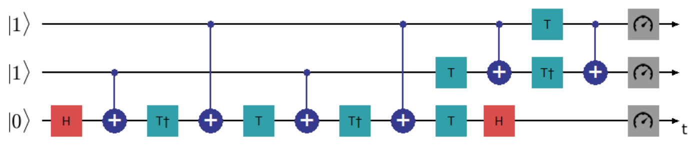


Fig. 6: A quantum circuit constructing the Toffoli gate, outputting the result into the third qubit

the binary value of $q_0$ to $q_1$: if $q_0 = |1\rangle$ then CNOT will (conditonally) flip $q_1 = |0\rangle \rightarrow |1\rangle$. Next, an X-gate is applied to $q_1$, flipping the value of $q_1$ unconditionally. Resulting in an output state of either $|01\rangle$ (with probability 1.0) *iff* the input state was $|00\rangle$, or $|10\rangle$ (with probability 1.0) *iff* the input state was $|10\rangle$. The "probability" result can be interpreted as: "100 out of 100 runs of this algorithm on a quantum computer with a given input *always* produces the corresponding classical result". The "true output" of this simple algorithm is contained in the second qubit/classical output bit, highlighted blue, and needs to be "extracted" from it, along the lines of '*bool result = classicalOutput & 0x01;*' in a classical programming language. Note that in this algorithm the original value of the input-bit is unchanged and could be reused again later. This is similar to what we might do in an assembly language classical algorithm. We might initialise a variable '*bool inverseA = !A;*', which gets put into a register on the CPU, and which gets used later in the program: '*if (inverseA) { /* ... */ }*'. We might choose (either *explicitly* by writing a program on the assembly-level, or *implicitly* by letting the compiler do it for us) to override that register value with a different value later, when we are done using it.

## B. Logical AND

Logical AND is in essence provided by the *Toffoli gate*, which is a two-control-bit CNOT [28]. Since our application does not provide a Toffoli gate as a drag-and-drop component, Figure 6 demonstrates one possible way of constructing a Toffoli gate using a set of "common" one-qubit and two-qubit gates. An alternative and equivalent construction is given in Figure 23 of Appendix J. We will not explain the exact reasoning behind why either two constructions work, as they were generally discovered by brute-force techniques, by matching various configurations against the desired truth table [6] (see Appendix I). Regardless of the implementation, we use qubits $q_0$ and $q_1$ as the control-qubits, and qubit $q_2$ as the target-qubit of the Toffoli gate. The result is output into output-qubit $q_2$. If we wanted to compute $0 \wedge 1$, we would assign $q_0 = |0\rangle$, $q_1 = |1\rangle$. $q_2$ is our reserved output-qubit is always initialised as $|0\rangle$. After applying the Toffoli gate, the result of $0 \wedge 1$ will be "output" into $q_2$, as $q_2$ will only be (conditionally) flipped to $|1\rangle$ if both $q_0$ and $q_1$ are set. The classical result is extracted from the last qubit of the classical output string.

## C. Logical OR

Logical OR can also be easily constructed along the same patterns. Using the fact that $A \vee B = \neg(\neg A \wedge \neg B)$. We use

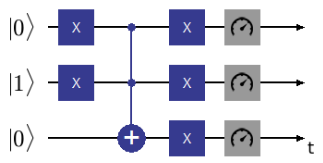


Fig. 7: A quantum circuit implementing OR using X-gates and one Toffoli gate

the more compact graphical representation of the Toffoli gate in the example shown in Figure 7.

The two first qubits, $q_0$ and $q_1$, yet again serve as the "input" to the OR-algorithm. Following the formula above, we first invert their values temporarily using the X-gates (NOT), and then take their AND into $q_2$. We then invert the $q_2$-result by using an X-gate on $q_2$. $q_0$ and $q_1$ are also inverted again in this circuit to reset them to their initial value from before the algorithm, though this step may not strictly be necessary.

## D. Two-Bit Adder

Classically, the implementation of a two-bit adder would be composed of a *half adder* followed by a *full adder* for the carry and the two next bits. The result would be contained in 3 bits [31]. A quantum algorithm achieving the same effect is shown in Figure 30. Here the input is composed of two sets of two bits: input integer A (orange) and input integer B (blue).

Let $(x)_y$ denote explicitly an integer $x$ being represented in its standard symbolic representation in base-$y$. In the example we wish to compute $(2)_{10} + (3)_{10}$, hence $A = (10)_2 = |10\rangle$, and $B = (11)_2 = |11\rangle$. The output is constrained to a maximum of thee bits and is pre-initialised as all-0. The full input state is therefore *encoded* as $|1011000\rangle$.

The first three gates handle the job of a *half adder*: If both the least-significant-bits of A and B ($q_{A1}$ and $q_{B1}$) are set, the one-to-last-significant-bit of the output ($q_{r1}$ or "carry") is set. If only the LSB of A ($q_{A1}$) or of B ($q_{B1}$) is set, then the LSB of the output ($q_{r2}$) gets set. The next set of gates are a *full adder*, and handle all possible carry-producing cases, flipping the "used-up" bits to 0. Three such carry-producing cases are possible: $q_{A0}$+$q_{r1}$, $q_{B0}$+$q_{r1}$, $q_{A0}$+$q_{B0}$. At the end, the case of a non-carried bit is handled. The full adder is "destructive" to the input bits due to the "use-up"-step. Many full adders could be chained/repeated to enable larger ($> 2$ bit large) inputs.

At the end the output result is produced into the last three bits. And at every step, the entire algorithm has a 1.0 probability of the correct result. The result of $|0001101\rangle$ indicates that with a probability of 100%, if this algorithm were executed on a quantum computer, in "100 out of 100 cases", given a (classical) input of "1011000", the output of the algorithm will be the binary string "0001101". The intended output — the "101"-part — needs to be extracted from the full string.

## IX. Deutsch's Algorithm: Taking a Problem from Exponential to Polynomial

*Deutsch's algorithm* (alternatively known as a $n = 1$ case of the *Deutsch-Jozsa algorithm*) is a simple algorithm which demonstrates a case where a quantum algorithm can solve a problem in exponentially less time than any possible deterministic classical algorithm, and somewhat less time than a randomized classical algorithm. It was the first quantum algorithms to demonstrate such a result while retaining absolute certainty of the correctness of its answer, unlike quantum algorithms that were published prior to it [32], [33]. (Though, the algorithm itself has little practical use).

*Deutsch's algorithm* tackles the following problem: given a function $f : \{0,1\} \to \{0,1\}$ we would like to quickly determine if $f(x)$ is *constant* — that is, always produces 0 or always produces 1 for both possible inputs — or *balanced* — produces 0 for exactly half the possible inputs, and 1 for the other half. $f$ is guaranteed to be one of both. Note that the definition of "balance" is most relevant for the *Deutsch-Jozsa* extension of the problem, which deals with any function $f : \{0,1\}^n \to \{0,1\}$.

We can already consider how we would solve this problem classically for $n = 1$: we would have to write an algorithm that checks for both possible inputs if the output is constant or not:

```
function solveDeutschClassic(f: Function) {
    return f(0) == f(1) ? "constant" : "balanced";
}
```

Thus, for the classical case, *the function needs to be evaluated twice* to come to an answer.

We briefly consider what a *randomized* (*stochastic*) classical algorithm could look like in the case of $n > 1$. Given $2^n$ possible unique binary inputs, one may evaluate $m < 2^n/2$ possible unique inputs picked at random from the total $2^n$ possible inputs. While evaluating said outputs, keep track of whether we ever encounter a differing output of $f(x)$. If it is encountered, we know for certain that $f(x)$ is not constant. On average, about $m = 3$ evaluations of $f(x)$ would be sufficient to see if $f$ is *not* constant. If we do not encounter an output that immediately does not match to the previous outputs, then we can only say with some probability $p$ that $f(x)$ *may actually be constant*, unless we actually evaluate $f$ for exactly $2^n/2+1$ inputs (i.e. more than half the possible unique inputs). This latter option corresponds to the deterministic classical algorithm. The randomized classical algorithm and the deterministic classical algorithm have similar worst-case time complexities due to requiring $2^n/2+1$ invocations, and thus have an *exponential* time complexity in the worst case.

We will now return to $n = 1$ and consider a gate $U_f$, named the "*oracle* for $f$", which applies to an input-qubit (and in the extended algorithm to $n$ input-qubits) $|x\rangle$ and one output-qubit $|y\rangle$. This matrix implements the logic of $f$, by ensuring an input state $|x\rangle |y\rangle$ gets mapped to the output state $|x\rangle |y \oplus f(x)\rangle$:

$$|x\rangle |y\rangle \xrightarrow{U_f} |x\rangle |y \oplus f(x)\rangle \tag{8.1}$$

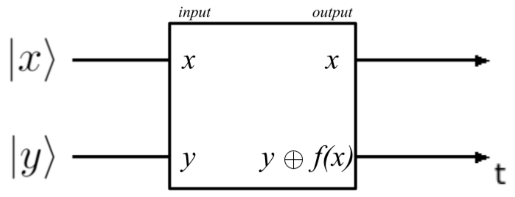


Fig. 8: The gate-representation of the *oracle for* $f$, $U_f$

The $\oplus$ (XOR or *addition mod* 2) against $y$ is necessary in this definition to preserve the property of $U_f$ being a unitary matrix (i.e. being reversible). $U_f$ otherwise acts as a "black box" to us, and we do not know and cannot discover by which means $f(x)$ is evaluated/implemented. Figure 8 demonstrates the representation of such an *oracle* gate $U_f$ on a two-qubit circuit. As seen prior, we know that $U_f$ will be a $2^2 \times 2^2 = 4 \times 4$ matrix, given that it operates on 2 qubits.

For the $n = 1$ case, we can exhaustively determine all the possible *unitary transformation matrices* that we can use as $U_f$ by using Equation 8.1 directly, of which all possibilities are listed in Appendix K. For the algorithm, we will initialise $|x\rangle$ as $|0\rangle$ and $|y\rangle$ as $|1\rangle$. Next, we first consider the two *constant* cases: $f(x) = 0$ and $f(x) = 1$. As shown in Appendix K, their corresponding oracle $U_f$ is the identity matrix $I_4$ (or $I \otimes I$) for $f(x) = 0$, and $U_f = I \otimes X$ for $f(x) = 1$. This corresponds to the circuits in Figure 24 and 25 of Appendix J, which show Deutsch's algorithm for these two oracles. Note that a see-through "box" is used to show the "internal implementation" of the circuit, which would be hidden from us in the problem.

It is easy to see that the top line and the bottom line are independent in both constant cases. The first two H-gates put both qubits in superposition, meaning that all $2^2 = 4$ possible states have probability 0.25 associated with them, or more specifically, the linear combination representing the state becomes:

$$\frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \tag{8.2}$$

A detailed work-out of this is found in Appendix K, Equation A.1. Subsequently, we either don't use any gate at all for $f(x) = 0$ or use the X-gate on the second qubit for $f(x) = 1$. This does not affect the first qubit, as only gates graphically involving the first qubit line would affect it (unless it was already *entangled*). Subsequently, the first qubit has another H-gate applied to it. Due to the fact that all gates are *unitary matrices*, H is *reversible*, meaning that $HH^* = HH = I$ (as $H^* = H$ for the Hadamard gate), and hence the superposition on the first line is reverted to being always $|0\rangle$ again. The final state-vector result for $f(x) = 0$ is a linear combination $\frac{1}{\sqrt{2}}(|00\rangle - |01\rangle)$. For $f(x) = 1$, the final state-vector result is $\frac{1}{\sqrt{2}}(-|00\rangle + |01\rangle)$, since the X-gate produces a *phase factor* of $-1$ The final *probability* results (absolute squared value) for both are the same: probability 0.5 for $|00\rangle$ and probability 0.5 for $|01\rangle$. This means that if this algorithm were executed on a real quantum computer, over many runs, we would get

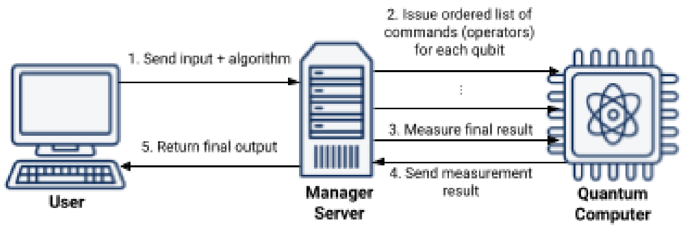

2. Issue ordered list of commands (operators) for each qubit
1. Send input + algorithm
3. Measure final result
5. Return final output
4. Send measurement result
User
Manager Server
Quantum Computer

Fig. 9: High level overview of a remote user's interaction with a quantum computer

back a classical binary string result of "00" half the time, and a result of "01" the other half of the time.

Note the "Measure" gate in both circuits is applied to the first qubit ($q_0$) only, similar to the final example of Section VIII. This indicates that we are only interested in extracting out the value of $q_0$. The value of the second qubit ($q_1$) is not of interest.

We next consider the *balanced* cases of $f(x) = \neg x$ and $f(x) = x$. As per the exhaustive list of Appendix K, we use the associated Gates $U_f$ to both functions and implement these as in Figure 26 and Figure 27 of Appendix J. We can observe immediately that the probability distribution is inverted. With probability $p = 0.5$, both cirquits using a *balanced* $f$ in $U_f$ will produce either $|10\rangle$ or $|11\rangle$. We can also see that both circuits use a CNOT gate, thereby *entangling* $q_0$ and $q_1$ (that is, by using a two-qubit cirquit on two qubits that are in *superposition*). The algebraic proof for this graphical result can be found back in Appendix L. As shown in the UI, for both *balanced* cases, the first qubit $q_0$ will always be 1. The second qubit $q_1$ will yet again vary, resulting in output states $|10\rangle$ and $|11\rangle$ being *measured* with equal probability. As before, $q_1$ is ignored when "extracting" the result from this algorithm.

In conclusion, we see that for the problem that for $n = 1$ would have classically required two function evaluations of function $f$, Deutsch's algorithm has enabled us to solve the problem with only a single 'oracle'-evaluation of $U_f$! It further turns out that the concept can be extended to $n > 1$ to achieve an improvement from a classically *exponential* to a quantum *polynomial* asymptotic runtime complexity [32]!

## X. Mathematics vs Reality

The linear algebra discussed in Section VI defines the theory behind quantum circuits, which allows us to predict the behaviour of qubits, and use it to design quantum algorithms, similar to how we can use equations/formulae to compute expected outcomes in classical physics (e.g. "how far will the car move"). Our application does not yet allow executing the algorithms on a real-world quantum computer, only to compute the predicted outcomes. In the real world a quantum circuit designed by a user can be translated into *a set of ordered operations* that a physical quantum computer needs to perform on its qubits to achieve the desired state transformations. This is shown in Figure 9. The computer will first need to set the input state, then apply the ordered list of operations to all involved qubits, and finally *measure* (read out) the result on the qubits. In doing so, it would trigger the collapse of the wave function, resulting in a read-out of a state according to the probability distribution of the state vector. The output would need to be extracted and reassembled for the user and returned to her.

Additionally, we must distinguish some quantum algorithms that are probabilistic in nature. This type of algorithm would have a non-100%-probability guarantee of producing the correct result. *Shor's algorithm* is an example of one such algorithm. Here, the probability of a correct result in one run might only be $2/3$, and to reach a desired low probability of error we would rerun the algorithm repeatedly, outputting the *majority result*, thereby *boosting* the precision of the final result (as per the *Chernoff bound*) for a time trade-off.

Regardless, in the physical world, the operators/gates (commands) might take the shape of specific laser pulses being applied to a type of particle, transforming its state in accordance to the theory of the operator. In Figure 9, a "Manager Server" sits between a remote user and the quantum computer and is responsible for performing the task of translating the user's quantum circuit design into the list of corresponding commands, and issuing those commands (e.g. through laser pulses) to the physical quantum computer setup. Here, it is of extreme importance that the quantum computer is in fact able to satisfy the normalization constraint at all times, and does not degrade from a pure state into a mixed state if undesired. Since in large quantum computing systems this may be especially difficult to maintain, large quantum computers (1K+ qubits) will generally have to employ error correction protocols at the level of algorithm or the "Manager Server".

## Conclusions

We hope to have sufficiently explained theory behind our tool with this paper. The tool we have built is intended to be used within a research setting after some necessarily feature-additions. Long-term prospects for the tool include enabling linking the tool to a physical quantum computer setup and, through a (cloud-hosted) operation-queueing-and-translation layer, enabling users to send valid quantum algorithms as "commands" to any external quantum computer, without needing to know the details of the physical setup outside of the number of qubits available at their disposal[8]. This is similar to what is provided as a paid service by companies such as IBM [3], Amazon [36] and Google [5]. We will also work on introducing a novel "pulse protocol" to the tool, to enable expert users to step away from the unitary model of quantum logic gates and pure states, and instead also allow work on the level of mixed states, integrating them with the existing quantum circuit model and drag-and-drop mechanism. While mixed states are not of interest for developing quantum algorithms, being undesireable degradations of the system, they are of particular interest in physics research.

[8]Compare to something like *MapReduce* [34] or *HPC* [35] scheduling

## Acknowledgments

I would like to thank Prof. Van Den Bussche, Prof. Nesladek and Dr. Shukla for their assistance, feedback and suggestions over the course of me working on this project.

## References

[1] The Royal Swedish Academy of Sciences. (2022) The Nobel Prize in Physics 2022. https://www.nobelprize.org/prizes/physics/2022/summary/.

[2] S. Aaronson, *Quantum Computing since Democritus*. Cambridge University Press, 2013.

[3] International Business Machines Corporation, "IBM Qiskit SDK: Quantum circuit model," https://docs.quantum.ibm.com/api/qiskit/circuit, accessed: 2024-05-24.

[4] Microsoft Corporation, "Explore quantum: Quantum circuits," https://quantum.microsoft.com/en-us/explore/concepts/quantum-circuits, accessed: 2024-05-24.

[5] Google LLC, "Google Cirq: Circuits," https://quantumai.google/cirq/build/circuits, accessed: 2024-05-24.

[6] R. J. Lipton and K. W. Regan, *Introduction to Quantum Algorithms via Linear Algebra, Second Edition*. Cambridge, Massachusetts: The MIT Press, 2021.

[7] Wikipedia contributors, "Floating-point unit — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Floating-point_unit&oldid=1233408872, 2024, [Online; accessed 11-August-2024].

[8] NVIDIA Corporation, *CUDA C++ Programming Guide*, 12th ed., https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, May 2024.

[9] Advanced Micro Devices, Inc, "AMD ROCm™ documentation," https://rocm.docs.amd.com/en/latest/what-is-rocm.html, accessed: 2024-05-24.

[10] C. Ferrie, "Quantum computers in 2023: how they work, what they do, and where they're heading," https://theconversation.com/quantum-computers-in-2023-how-they-work-what-they-do-and-where-theyre-heading-215804, accessed: 2024-05-24.

[11] Wikipedia contributors, "List of quantum processors — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=List_of_quantum_processors&oldid=1237714483, 2024, [Online; accessed 19-August-2024].

[12] F. Bova, A. Goldfarb, and R. G. Melko, "Commercial applications of quantum computing," *EPJ quantum technology*, vol. 8, no. 1, p. 2, 2021.

[13] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[14] D. K. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," RFC 7919, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7919

[15] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, feb 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[16] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85 Proceedings*, H. C. Williams, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426.

[17] Wikipedia contributors, "General number field sieve — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=General_number_field_sieve&oldid=1207450577, 2024, [Online; accessed 24-May-2024].

[18] D. Harvey and J. Van Der Hoeven, "Integer multiplication in time o(nlog n)," *Annals of Mathematics*, vol. 193, no. 2, pp. 563–617, 2021.

[19] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[20] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[21] S. Pirandola, U. L. Andersen, L. Banchi, M. Berta, D. Bunandar, R. Colbeck, D. Englund, T. Gehring, C. Lupo, C. Ottaviani, J. L. Pereira, M. Razavi, J. S. Shaari, M. Tomamichel, V. C. Usenko, G. Vallone, P. Villoresi, and P. Wallden, "Advances in quantum cryptography," *Adv. Opt. Photon.*, vol. 12, no. 4, pp. 1012–1236, Dec 2020. [Online]. Available: https://opg.optica.org/aop/abstract.cfm?URI=aop-12-4-1012

[22] X. Foundation, "Quantum for real-world impact," https://www.xprize.org/prizes/qc-apps, accessed: 2024-04.

[23] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[24] S. Resch and U. R. Karpuzcu, "Benchmarking quantum computers and the impact of quantum noise," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–35, 2021.

[25] Amazon.com, Inc, "Amazon serverless compute," https://aws.amazon.com/serverless/, accessed: 2024-04.

[26] Wikipedia, Smite-Meister, CC BY-SA 3.0, via Wikimedia Commons, "The bloch sphere, a geometric representation of a two-level quantum system," https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg, 2009.

[27] I. Oliveira, R. Sarthour, T. Bonagamba, E. Azevedo, and J. Freitas, *NMR Quantum Information Processing*. Elsevier Science, 2011.

[28] Wikipedia contributors, "Quantum logic gate — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Quantum_logic_gate&oldid=1232422776, 2024, [Online; accessed 3-August-2024].

[29] D. Meyer, "A Few Notes on Bell States, Superdense Coding, and QuantumTeleportation," https://davidmeyer.github.io/qc/bell.pdf, 2023.

[30] P. S. (https://quantumcomputing.stackexchange.com/users/4359/perry sakkaris), "How to construct matrix of regular and "flipped" 2-qubit CNOT?" Quantum Computing Stack Exchange, (version: 2020-02-22). [Online]. Available: https://quantumcomputing.stackexchange.com/a/5192

[31] Wikipedia contributors, "Adder (electronics) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Adder_(electronics)&oldid=1238197425, 2024, [Online; accessed 9-August-2024].

[32] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proceedings: Mathematical and Physical Sciences*, vol. 439, no. 1907, pp. 553–558, 1992. [Online]. Available: http://www.jstor.org/stable/52182

[33] Wikipedia contributors, "Deutsch–jozsa algorithm — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Deutsch%E2%80%93Jozsa_algorithm&oldid=1236296482, 2024, [Online; accessed 10-August-2024].

[34] Apache Software Foundation, "Mapreduce tutorial," https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2023, [Online; accessed 18-August-2024].

[35] HPC Wiki contributors, "Scheduling basics," https://hpc-wiki.info/hpc/Scheduling_Basics, 2019, [Online; accessed 18-August-2024].

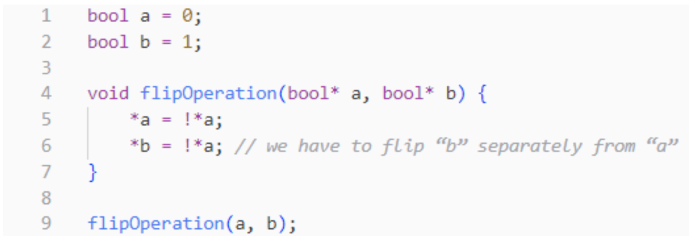
[36] Amazon.com, Inc, "Amazon braket documentation," https://docs.aws.amazon.com/braket/, accessed: 2024-04.

[37] Wikipedia, Trex4321, CC0, via Wikimedia Commons, "An xor gate using a 2-2 ano-or-invert gate and negated inputs," https://commons.wikimedia.org/wiki/File:XOR_gate_based_on_2-2_AOI_gate.svg, 2024.

[38] S. Gharibian, "Lecture 6: Deutsch's algorithm," https://people.vcu.edu/~sgharibian/courses/CMSC491/notes/Lecture%206%20-%20Deutsch's%20algorithm.pdf, 2015.

## Appendix

### A. An Overview of Classical Bit-Level State Management



```c
bool a = 0;
bool b = 1;

void flipOperation(bool* a, bool* b) {
    *a = !*a;
    *b = !*a; // we have to flip “b” separately from “a”
}

flipOperation(a, b);
```

Fig. 10: a simple C program that flips two bits, ensuring one remains the opposite of the other.

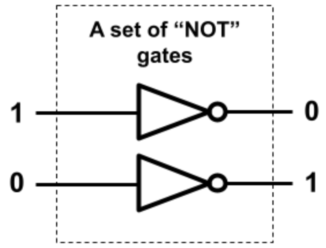


Fig. 11: two bits being inverted using a set of 2 logical NOT gates in a classical computer circuit diagram.

### B. Runtime Complexity Comparison of Factoring Algorithms

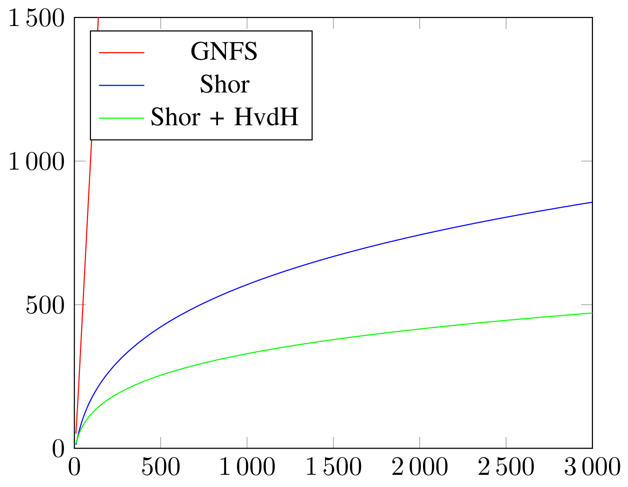


Fig. 12: A comparison of integer factoring algorithms

### C. Alternative Visualisation for Tensor Product

A tensor product of two matrices can (informally) also be visualised as the following, where the “nested” matrices are expanded at the end (the brackets are removed) into one big resultant matrix:

$$\begin{bmatrix}1&2\\3&4\end{bmatrix}\otimes\begin{bmatrix}5&6\\7&8\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}1\begin{bmatrix}5&6\\7&8\end{bmatrix}\\3\begin{bmatrix}5&6\\7&8\end{bmatrix}\end{bmatrix}&\begin{bmatrix}2\begin{bmatrix}5&6\\7&8\end{bmatrix}\\4\begin{bmatrix}5&6\\7&8\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

$$=\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}1(5)&1(6)\\1(7)&1(8)\end{bmatrix}\\\begin{bmatrix}3(5)&3(6)\\3(7)&3(8)\end{bmatrix}\end{bmatrix}&\begin{bmatrix}\begin{bmatrix}2(5)&2(6)\\2(7)&2(8)\end{bmatrix}\\\begin{bmatrix}4(5)&4(6)\\4(7)&4(8)\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

$$=\begin{bmatrix}1(5)&1(6)&2(5)&2(6)\\1(7)&1(8)&2(7)&2(8)\\3(5)&3(6)&4(5)&4(6)\\3(7)&3(8)&4(7)&4(8)\end{bmatrix}$$

This is not a formal definition but merely a visualisation strategy, as nested matrices can have a different and concrete meaning in mathematics.

Certainly thinking about this example from a high-level-programming-language point of view, there would be a difference between a 2-level nested list/array such as this in python:

```python
list1 = [
    [1*5, 1*6, 2*6, 2*6],
    [1*7, 1*8, 2*7, 2*8],
    [3*5, 3*6, 4*5, 4*6],
    [3*7, 3*8, 4*7, 4*8]
]
```

And a more convoluted 4-level nested list:

```python
list2 = [
    [
        [[1*5, 1*6], [1*7, 1*8]],
        [[2*5, 2*6], [2*7, 2*8]]
    ],
    [
        [[3*5, 3*6], [3*7, 3*8]],
        [[4*5, 4*6], [4*7, 4*8]]
    ],
]
```

Even if physically, especially in a lower-level language such as C, such datastructures would end up being stored in memory as a single continuous “array” (chunk) of 32 or 64-bit integers.

### D. Bloch Sphere Parameters Solution for the Basis State $|1\rangle$

This section provides a written out solution to computing the Bloch Sphere parameters $0 \leq \theta \leq \pi$ and $0 \leq \varphi < 2\pi$ (both radians), for visualusing the one-qubit basis state $|1\rangle$. $|0\rangle$ does not need to be computed as it has $\theta = 0, \varphi = 0$. Other pure states (superpositons) are computed analogously to the following.

We start by writing down the state vector in terms of (Equation 6.1) and equating it to (Equation 6.2). For the pure states this

is obviously quite simple, $|1\rangle$ having $\alpha = 0, \beta = 1$ (and $|0\rangle$ having $\alpha = 1, \beta = 0$).
Solving this for $|1\rangle$:

$$\begin{aligned}|1\rangle &= \cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle \\ \Leftrightarrow \begin{bmatrix}0\\1\end{bmatrix} &= \begin{bmatrix}\cos(\theta/2)\\ e^{i\varphi}\sin(\theta/2)\end{bmatrix} \Leftrightarrow \begin{cases}\cos(\theta/2) = 0\\ e^{i\varphi}\sin(\theta/2) = 1\end{cases}\end{aligned}$$

We solve for $\theta$:

$$\begin{aligned} &\cos^{-1}(0) &&= \theta/2 \\ \Leftrightarrow\ &2\cos^{-1}(0) &&= \theta \\ \Leftrightarrow\ &2(\tfrac{\pi}{2}) &&= \theta \qquad [\text{since } 0 \le \theta \le \pi] \\ \Leftrightarrow\ &\pi &&= \theta \end{aligned}$$

We next solve for $\varphi$ using $\theta$:

$$\begin{aligned} &e^{i\varphi}\sin(\pi/2) &&= 1 \quad [\text{fill in } \theta = \pi] \\ \Leftrightarrow\ &e^{i\varphi}(1) &&= 1 \\ \Leftrightarrow\ &\ln(e^{i\varphi}) &&= \ln(1) \\ \Leftrightarrow\ &i\varphi &&= 0 \\ \Leftrightarrow\ &\varphi &&= 0/i \\ \Leftrightarrow\ &\varphi &&= 0 \end{aligned}$$

This solution, $\theta = \pi, \varphi = 0$, in turn gives a vector pointing straight down on the Bloch Sphere (Figure 2) to represent $|1\rangle$.

### *E. Basic Gate Library*

The “Basic Gate Library” of all quantum logic gates included in our application is given in Table I.

### *F. Graph Data Structure Example Table*

The two main methods for storing the widely-known ”graph” data structure (directed or undirected) in memory are: (1) the adjacency matrix; and (2) adjacency list. This section provides an example of an adjacency matrix representation of a graph. Of interest is the *concept* of “mapping” items in a “from-to” relationship using a matrix.
Given the following directed graph:

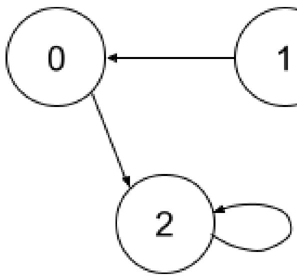
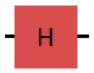
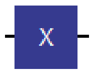
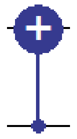
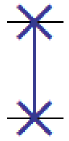
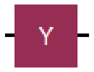
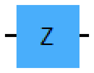
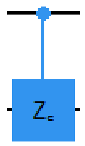
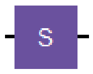
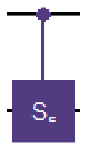
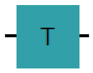
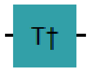
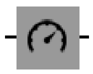


Fig. 13: A basic directed graph consisting of 3 nodes: 0, 1, and 2

This graph can be represented by the following adjacency matrix:

| | **0** | **1** | **2** |
|---|---|---|---|
| **0** | 0 | 0 | 1 |
| **1** | 1 | 0 | 0 |
| **2** | 0 | 0 | 1 |

TABLE I: Basic Gate Library

| gate | name (alternative names) | matrix |
|---|---|---|
|  | H (Hadamard) | $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$ |
|  | X (Pauli-X, NOT) | $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$ |
|  | CNOT (Controlled NOT) | $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$ |
|  | SWAP | $\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}$ |
|  | Y (Pauli-Y) | $\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$ |
|  | Z (Pauli-Z) | $\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$ |
|  | CZ (Controlled Z) | $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{bmatrix}$ |
|  | S (Phase, P) | $\begin{bmatrix}1 & 0\\0 & i\end{bmatrix}$ |
|  | Controlled S (Phase, P) | $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&i\end{bmatrix}$ |
|  | T ($\pi/8$) | $\begin{bmatrix}1 & 0\\0 & e^{i\pi/4}\end{bmatrix}$ |
|  | T$^{\dagger}$ (T$^{*}$) | $\begin{bmatrix}1 & 0\\0 & e^{-i\pi/4}\end{bmatrix}$ |
|  | Measure | - |

As is surely widely understood, cells with “0” represent the absence of a direct edge/link between the node in the row/column, and a “1” represents the presence of such a link. In this example it was chosen to have the “rows” represent the “from” and the “columns” represent the “to” of an edge/link.

### *G. Algorithmic Method for Controlled Gate Construction: Example*

This section provides an example of how one can construct a Controlled NOT matrix for a system of three qubits, as shown in (Figure 14), using qubit 1 ($q_1$) as the *control-qubit* (middle line), and qubit 2 ($q_2$) (bottom line) as the *target-qubit*. Qubit

0 ($q_0$) (top line) will remain unaffected by this matrix.

**"If qubit $j = 1$ is $|0\rangle$, then do nothing to qubit $k = 2$":**

$$M_{q_j=|0\rangle} = I \otimes \underbrace{|0\rangle\langle 0|}_{j=1} \otimes \underbrace{I}_{k=2} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\0 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

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

**"If qubit $j = 1$ is $|1\rangle$, then apply gate X to qubit $k = 2$":**

$$M_{q_j=|1\rangle} = I \otimes \underbrace{|1\rangle\langle 1|}_{j=1} \otimes \underbrace{X}_{k=2} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \otimes \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} \otimes \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$$

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

**Resulting unitary transformation matrix ("system-level operation matrix"):**

$$CNOT_{j,k} = M_{q_j=|0\rangle} + M_{q_j=|1\rangle} =$$

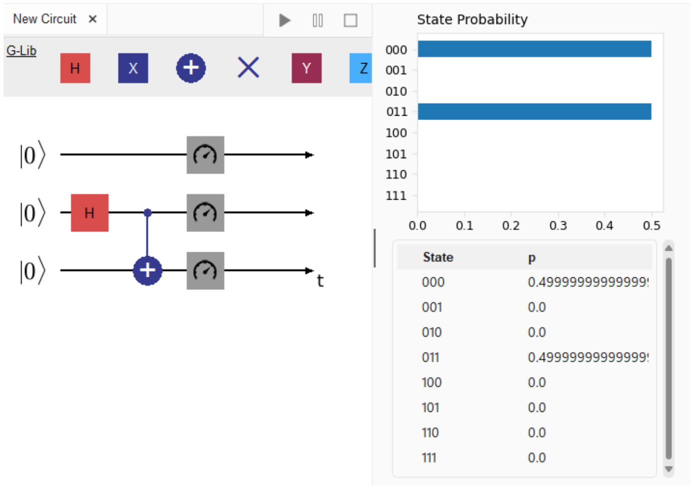
$$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix} + \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$$

$$= \begin{array}{c|cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111\\ \hline 000 & 1&0&0&0&0&0&0&0\\ 001 & 0&1&0&0&0&0&0&0\\ 010 & 0&0&0&1&0&0&0&0\\ 011 & 0&0&1&0&0&0&0&0\\ 100 & 0&0&0&0&1&0&0&0\\ 101 & 0&0&0&0&0&1&0&0\\ 110 & 0&0&0&0&0&0&0&1\\ 111 & 0&0&0&0&0&0&1&0 \end{array}$$

All the $2^3 = 8$ mappings are given below, with the non-identity mappings highlighted (control bit blue, target bit red):

$$\textbf{input} \xrightarrow{\text{CNOT}} \textbf{output}$$

$$|000\rangle \longrightarrow |000\rangle$$
$$|001\rangle \longrightarrow |001\rangle$$
$$|010\rangle \longrightarrow |011\rangle$$
$$|011\rangle \longrightarrow |010\rangle$$
$$|100\rangle \longrightarrow |100\rangle$$
$$|101\rangle \longrightarrow |101\rangle$$
$$|110\rangle \longrightarrow |111\rangle$$
$$|111\rangle \longrightarrow |110\rangle$$



| State | p |
|---|---|
| 000 | 0.49999999999999! |
| 001 | 0.0 |
| 010 | 0.0 |
| 011 | 0.49999999999999! |
| 100 | 0.0 |
| 101 | 0.0 |
| 110 | 0.0 |
| 111 | 0.0 |

Fig. 14: Three qubit circuit with H targetting qubit 1 followed by Controlled NOT targetting qubit 2, using qubit 1 as control

To fully explain (Figure 14), we can observe that the initial state of the qubit system is $|000\rangle$. Next, a H (Hadamard) gate is applied to the middle qubit — qubit 1, highlighted blue — making a superposition with two possible states with equal probability: one where qubit 1 is $|0\rangle$ and one where qubit 1 is $|1\rangle$. Qubit 0 (top most in the circuit, also left-most in the input state vector and in the binary string) is always $|0\rangle$.

If qubit 1 is $|1\rangle$, we invert qubit 2 to be $|1\rangle$, which is $|0\rangle$ initially (highlighted red in the input vector). If qubit 1 is $|0\rangle$, we don't do anything with qubit 2 and it stays $|0\rangle$.

The only possible output states in this case are:
(1) $|\mathbf{000}\rangle$ ($q_1$ not flipped, hence CNOT no effect on $q_2$); and
(2) $|\mathbf{011}\rangle$ ($q_1$ was flipped, hence CNOT also flipped $q_2$ conditionally).
With equal probability $p = 0.5$ each.

### *H. Bell States*

| Bell State | Formula |
|---|---|
| $\lvert\phi^+\rangle$ | $\frac{\lvert 00\rangle + \lvert 11\rangle}{\sqrt{2}}$ |
| $\lvert\phi^-\rangle$ | $\frac{\lvert 00\rangle - \lvert 11\rangle}{\sqrt{2}}$ |
| $\lvert\psi^+\rangle$ | $\frac{\lvert 01\rangle + \lvert 10\rangle}{\sqrt{2}}$ |
| $\lvert\psi^-\rangle$ | $\frac{\lvert 01\rangle - \lvert 10\rangle}{\sqrt{2}}$ |

## *I. Classical Logic Circuits Reference*

This section provides an overview of the standard classical logic gates and their truth tables. This section may be of interest when comparing to the corresponding quantum implementations of these basic operations.

**Classical NOT**

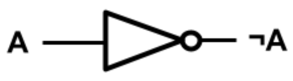


Fig. 15:  A classical logic circuit NOT gate

| $A$ | $\neg A$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Classical AND**

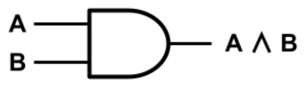


Fig. 16:  A classical logic circuit AND gate

| $A$ | $B$ | $A \wedge B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Classical OR**

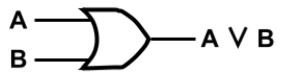


Fig. 17:  A classical logic circuit OR gate

| $A$ | $B$ | $A \vee B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**XOR implemented by composing basic classical logic gates**

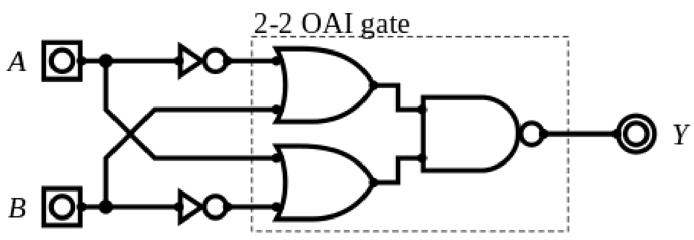


Fig. 18:  'XOR' implemented using a combination of classical logic gates [37]

## *J. Quantum Circuit Examples*

This section provides more quantum circuit examples that were left out of the main text for the sake of brevity.

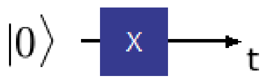


Fig. 19:  A single-qubit, single-X-gate quantum circuit

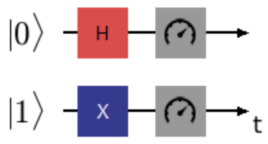
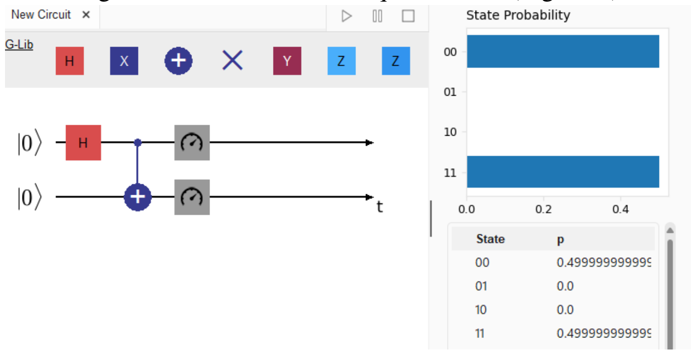
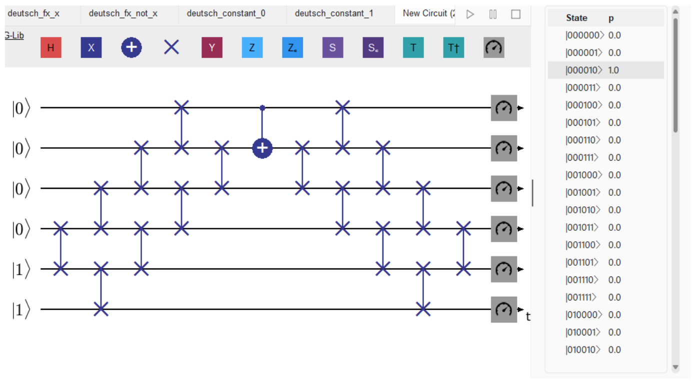


Fig. 20:  A circuit that is equivalent to (Figure 4)



| State | p |
|---|---|
| 00 | 0.4999999999999 |
| 01 | 0.0 |
| 10 | 0.0 |
| 11 | 0.4999999999999 |

Fig. 21:  A circuit composed of a Hadamard and Controlled NOT gate, producing output state $|\phi^+\rangle$ — one of the *Bell States*



| State | p |
|---|---|
| \|000000⟩ | 0.0 |
| \|000001⟩ | 0.0 |
| \|000010⟩ | 1.0 |
| \|000011⟩ | 0.0 |
| \|000100⟩ | 0.0 |
| \|000101⟩ | 0.0 |
| \|000110⟩ | 0.0 |
| \|000111⟩ | 0.0 |
| \|001000⟩ | 0.0 |
| \|001001⟩ | 0.0 |
| \|001010⟩ | 0.0 |
| \|001011⟩ | 0.0 |
| \|001100⟩ | 0.0 |
| \|001101⟩ | 0.0 |
| \|001110⟩ | 0.0 |
| \|001111⟩ | 0.0 |
| \|010000⟩ | 0.0 |
| \|010001⟩ | 0.0 |
| \|010010⟩ | 0.0 |

Fig. 22:  One approach to interpret applying an operation to two qubits that are not situated on the first two lines

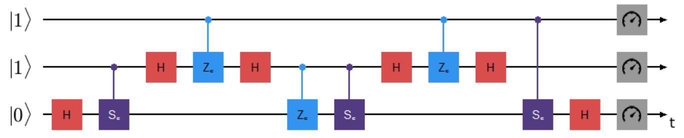
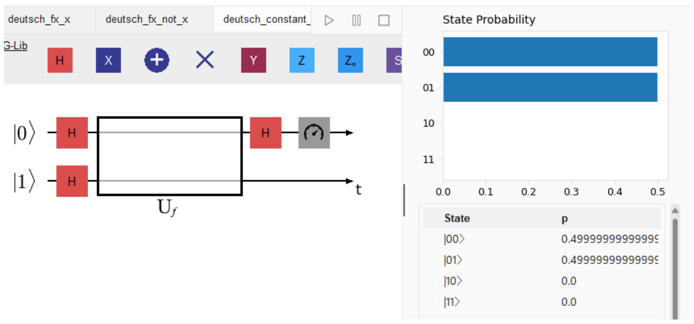
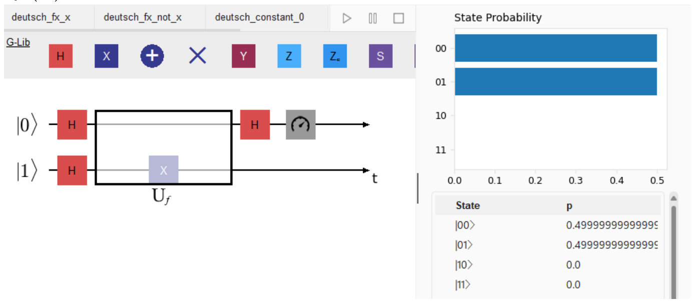


Fig. 23:  An alternative way to implement a Toffoli gate, equivalent to (Figure 6) [6]



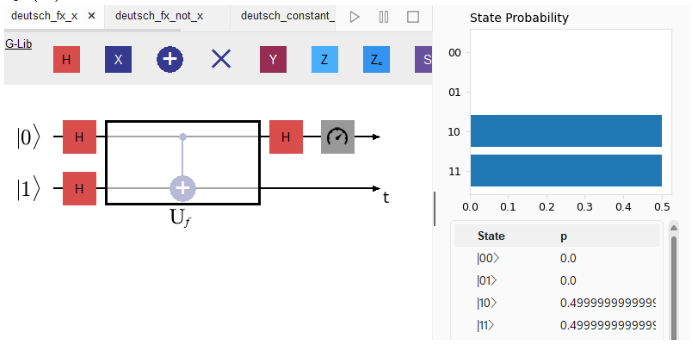
| State | p |
|---|---|
| \|00〉 | 0.49999999999999 |
| \|01〉 | 0.49999999999999 |
| \|10〉 | 0.0 |
| \|11〉 | 0.0 |

Fig. 24: The circuit for the Deutsch algorithm with $U_f$ using $f(x) = 0$



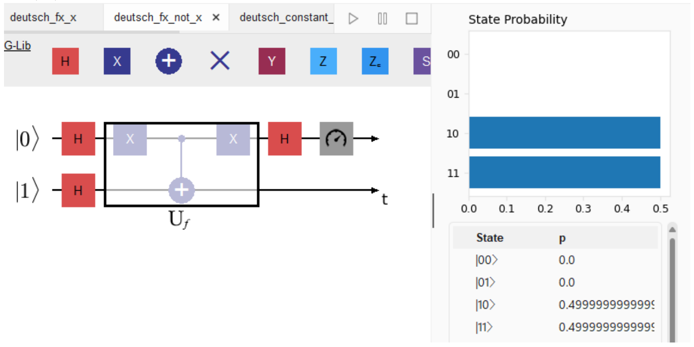
| State | p |
|---|---|
| \|00〉 | 0.49999999999999 |
| \|01〉 | 0.49999999999999 |
| \|10〉 | 0.0 |
| \|11〉 | 0.0 |

Fig. 25: The circuit for the Deutsch algorithm with $U_f$ using $f(x) = 1$



| State | p |
|---|---|
| \|00〉 | 0.0 |
| \|01〉 | 0.0 |
| \|10〉 | 0.499999999999 |
| \|11〉 | 0.499999999999 |

Fig. 26: The circuit for the Deutsch algorithm with $U_f$ using $f(x) = x$



| State | p |
|---|---|
| \|00〉 | 0.0 |
| \|01〉 | 0.0 |
| \|10〉 | 0.499999999999 |
| \|11〉 | 0.499999999999 |

Fig. 27: The circuit for the Deutsch algorithm with $U_f$ using $f(x) = \neg x$

### K. *Deutsch Algorithm $U_f$ Matrices*

This section covers all 4 possible unitary transformation matrices that may be used as the *oracle* $U_f$ in the Deutsch algorithm (Deutsch-Josza with $n = 1$). We note again that we are looking to determine a matrix representation for the following equation:

$$|x, y\rangle \xrightarrow{U_f} |x, y \oplus f(x)\rangle$$

In total we have 4 functions that can be distinguished as operating on 1 input bit:

- *Constant* functions: $f(x) = 0$ and $f(x) = 1$
- *Balanced* functions: $f(x) = x$ and $f(x) = \neg x$

**Balanced function $f(x) = x$:**

For this function we can determine the truth table:

| $x$ | $y$ | $f(x) = x$ | $y \oplus f(x)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

Hence the state maps for the gate $U_f$ would have to map:

$$|0, 0\rangle \xrightarrow{U_f} |0, 0\rangle$$
$$|0, 1\rangle \xrightarrow{U_f} |0, 1\rangle$$
$$|1, 0\rangle \xrightarrow{U_f} |1, 1\rangle$$
$$|1, 1\rangle \xrightarrow{U_f} |1, 0\rangle$$

This corresponds to the following matrix, which is in fact the CNOT matrix (see Appendix E):

$$\begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} =: \mathit{CNOT}$$

**Balanced function $f(x) = \neg x$:**

Corresponding truth table:

| $x$ | $y$ | $f(x) = \neg x$ | $y \oplus f(x)$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

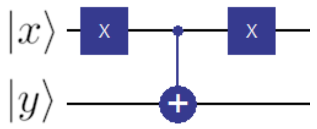
Mappings for corresponding $U_f$:

$$|0, 0\rangle \xrightarrow{U_f} |0, 1\rangle$$
$$|0, 1\rangle \xrightarrow{U_f} |0, 0\rangle$$
$$|1, 0\rangle \xrightarrow{U_f} |1, 0\rangle$$
$$|1, 1\rangle \xrightarrow{U_f} |1, 1\rangle$$

This corresponds to the following matrix:

$$\begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 0 & 1 & 0 & 0 \\ 01 & 1 & 0 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 \\ 11 & 0 & 0 & 0 & 1 \end{array}$$

Note that this matrix inverses the behaviour for CNOT in a certain sense. For this matrix we flip the second qubit if the first qubit is not set.
This can be simulated by the following gates:



Here, we first flip $|x\rangle$, then apply CNOT based on this flipped value, and reset the flip once that is done.
Alternatively, $U_f$ is given by:

$$(X \otimes I)(CNOT)(X \otimes I)$$

$$= \left(\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right) \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix} \left(\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right)$$

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

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

**Constant function $f(x) = 0$:**

For this function we can determine the truth table:

| $x$ | $y$ | $f(x) = 0$ | $y \oplus f(x)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

Mappings for corresponding $U_f$:

$$|0,0\rangle \xrightarrow{U_f} |0,0\rangle$$
$$|0,1\rangle \xrightarrow{U_f} |0,1\rangle$$
$$|1,0\rangle \xrightarrow{U_f} |1,0\rangle$$
$$|1,1\rangle \xrightarrow{U_f} |1,1\rangle$$

We note that $y$ will be unchanged. This is the $4 \times 4$ identity matrix:

$$\begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 1 & 0 & 0 & 0 \\ 01 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 \\ 11 & 0 & 0 & 0 & 1 \end{array}$$

Hence in this case $U_f$ has no effect. This corresponds to not using any gates/no change.

**Constant function $f(x) = 1$:**

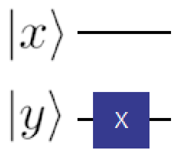
For this function we can determine the truth table:

| $x$ | $y$ | $f(x) = 1$ | $y \oplus f(x)$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

Mappings for corresponding $U_f$:

$$|0,0\rangle \xrightarrow{U_f} |0,1\rangle$$
$$|0,1\rangle \xrightarrow{U_f} |0,0\rangle$$
$$|1,0\rangle \xrightarrow{U_f} |1,1\rangle$$
$$|1,1\rangle \xrightarrow{U_f} |1,0\rangle$$

We note that $y$ is always getting inverted by $U_f$. This is the same as applying an "unconditional CNOT" aka simply the X-gate to qubit $y$:



Corresponding to matrix:

$$(I \otimes X) = \begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 0 & 1 & 0 & 0 \\ 01 & 1 & 0 & 0 & 0 \\ 10 & 0 & 0 & 0 & 1 \\ 11 & 0 & 0 & 1 & 0 \end{array}$$

**Computing two H-gates being applied to $|01\rangle$**
We compute (annotated with state-vector indices):

$$(H \otimes H)\,|01\rangle$$

$$= \left(\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\right) \begin{array}{c|c} 0 & 00 \\ 1 & 01 \\ 0 & 10 \\ 0 & 11 \end{array}$$

$$= \frac{1}{2}\begin{bmatrix}1&1&1&1\\1&-1&1&-1\\1&1&-1&-1\\1&-1&-1&1\end{bmatrix} \begin{array}{c|c} 0 & 00 \\ 1 & 01 \\ 0 & 10 \\ 0 & 11 \end{array}$$

$$= \frac{1}{2}\begin{array}{c}\begin{bmatrix}1\\-1\\1\\-1\end{bmatrix}\end{array}\begin{array}{c}\scriptstyle 00\\ \scriptstyle 01\\ \scriptstyle 10\\ \scriptstyle 11\end{array} = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \quad \text{(A.1)}$$

This gives us the result of Equation 8.2.

### *L. Deutsch's Algorithm: Balanced Functions Proof*

This solution is based on [38].
To prove the expected result for the two *balanced* cases, we note that if $f$ is a *balanced* function, then $f(0) \neq f(1)$. This also means that $1 \oplus f(0) = f(1)$ and $1 \oplus f(1) = f(0)$, as we are working with a single bit input and output. Resuming from Equation 8.2, we apply either of the two possible $U_f$:

$$\begin{aligned}
&U_f\frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)\\
&= \frac{1}{2}(|0, 0 \oplus f(0)\rangle - |0, 1 \oplus f(0)\rangle + |1, 0 \oplus f(1)\rangle - |1, 1 \oplus f(1)\rangle)\\
&= \frac{1}{2}(|0, f(0)\rangle - |0, f(1)\rangle + |1, f(1)\rangle - |1, f(0)\rangle)\\
&= \frac{1}{2}((|0\rangle - |1\rangle) \otimes |f(0)\rangle - (|0\rangle - |1\rangle) \otimes |f(1)\rangle)\\
&= \frac{1}{2}(|0\rangle - |1\rangle) \otimes (|f(0)\rangle - |f(1)\rangle)\\
&= \frac{1}{\sqrt{2}}|-\rangle \otimes (|f(0)\rangle - |f(1)\rangle) \quad \text{(A.2)}
\end{aligned}$$

We now know that $q_0$ is in state $|-\rangle$ (which was mentioned in Section VII-A). If we then rewrite as follows:

$$\frac{1}{\sqrt{2}}|-\rangle \otimes (|f(0)\rangle - |f(1)\rangle) = |-\rangle \otimes \frac{1}{\sqrt{2}}(|f(0)\rangle - |f(1)\rangle)$$

Then we can observe that: $\frac{1}{\sqrt{2}}(|f(0)\rangle - |f(1)\rangle) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$, if $f(x) = x$.
And $\frac{1}{\sqrt{2}}(|f(0)\rangle - |f(1)\rangle) = \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = -|-\rangle$, if $f(x) = \neg x$.
Hence, in either balanced case, we get $\pm|-\rangle$.
Continuing from Equation A.2, this gives us:

$$\frac{1}{\sqrt{2}}|-\rangle \otimes (|f(0)\rangle - |f(1)\rangle) = \pm|-\rangle|-\rangle \quad \text{(A.3)}$$

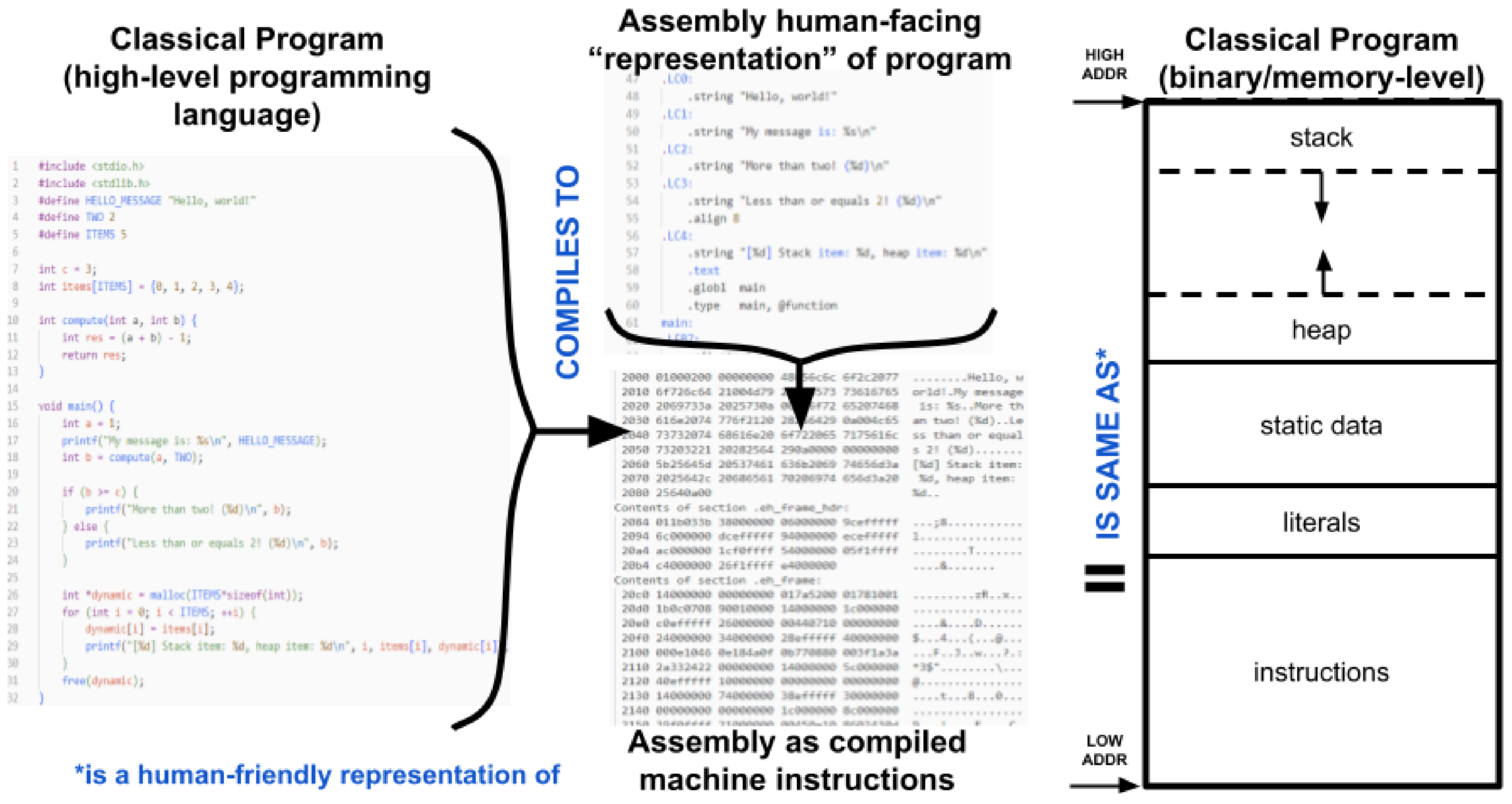
If we now apply a H-gate to $q_0$, continuing from Equation A.3, and we get:

$$(H \otimes I)(\pm|-\rangle|-\rangle) = \pm\frac{1}{2\sqrt{2}}\begin{bmatrix}1 & 0 & 1 & 0\\0 & 1 & 0 & 1\\1 & 0 & -1 & 0\\0 & 1 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\-1\\-1\\1\end{bmatrix}$$

$$= \pm\frac{1}{2\sqrt{2}}\begin{bmatrix}0\\0\\2\\-2\end{bmatrix} = \pm\frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\1\\-1\end{bmatrix}\begin{array}{c}\scriptstyle 00\\ \scriptstyle 01\\ \scriptstyle 10\\ \scriptstyle 11\end{array} = \pm|1\rangle|-\rangle$$

This confirms what was shown in our UI, that for both *balanced* cases, the first qubit $q_0$ will always be 1.

### M. Classical Program vs Quantum Program Visualised

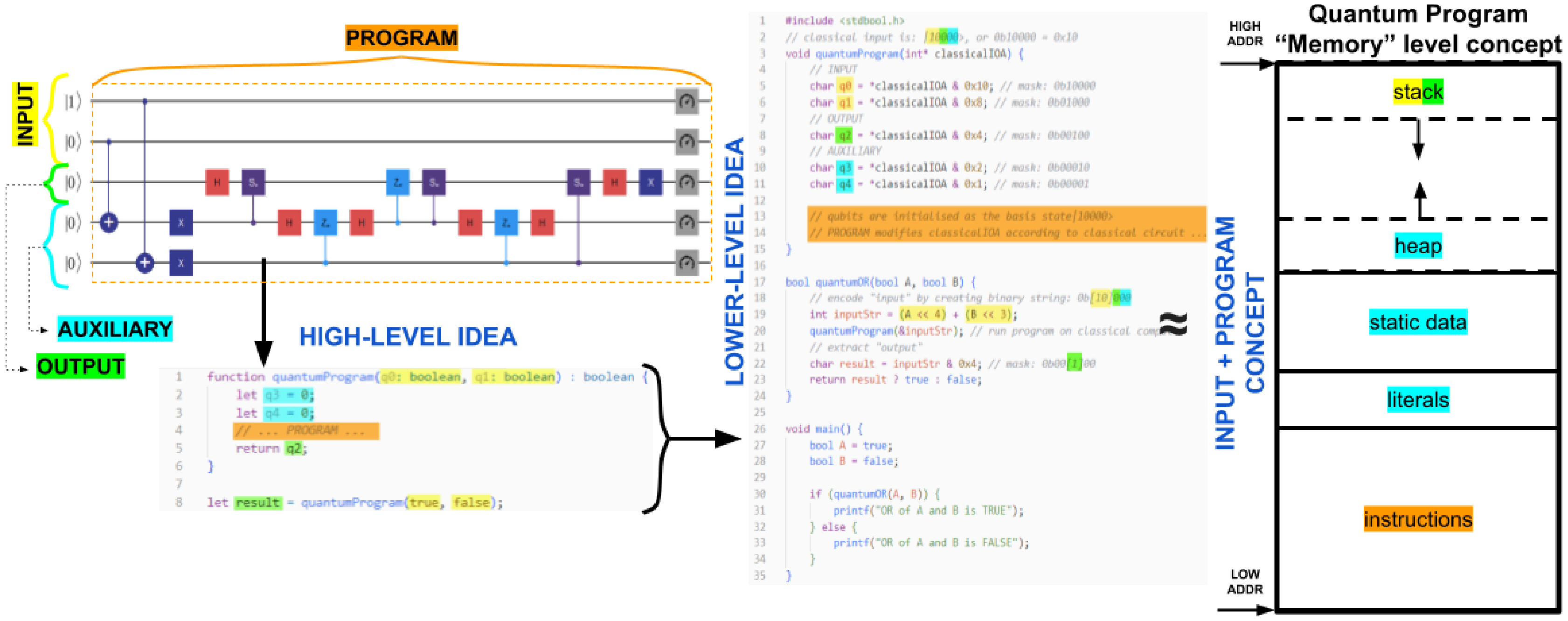

Classical Program (high-level programming language)
*is a human-friendly representation of
COMPILES TO
Assembly human-facing “representation” of program
Assembly as compiled machine instructions
= IS SAME AS*
HIGH ADDR
LOW ADDR
Classical Program (binary/memory-level)
stack
heap
static data
literals
instructions

```c
#include <stdio.h>
#include <stdlib.h>
#define HELLO_MESSAGE "Hello, world!"
#define TWO 2
#define ITEMS 5

int c = 3;
int items[ITEMS] = {0, 1, 2, 3, 4};

int compute(int a, int b) {
    int res = (a + b) - 1;
    return res;
}

void main() {
    int a = 1;
    printf("My message is: %s\n", HELLO_MESSAGE);
    int b = compute(a, TWO);

    if (b >= c) {
        printf("More than two! (%d)\n", b);
    } else {
        printf("Less than or equals 2! (%d)\n", b);
    }

    int *dynamic = malloc(ITEMS*sizeof(int));
    for (int i = 0; i < ITEMS; ++i) {
        dynamic[i] = items[i];
        printf("[%d] Stack item: %d, heap item: %d\n", i, items[i], dynamic[i]);
    }
    free(dynamic);
}
```

```
.LC0:
    .string "Hello, world!"
.LC1:
    .string "My message is: %s\n"
.LC2:
    .string "More than two! (%d)\n"
.LC3:
    .string "Less than or equals 2! (%d)\n"
    .align 8
.LC4:
    .string "[%d] Stack item: %d, heap item: %d\n"
    .text
    .globl  main
    .type   main, @function
main:
```

```
 2000 01000200 00000000 48656c6c 6f2c2077  ........Hello, w
 2010 6f726c64 21004d79 [illegible] 73616765  orld!.My message
 2020 2069733a 2025730a [illegible] 65207468   is: %s..More th
 2030 616e2074 776f2120 [illegible] 0a004c65  an two! (%d)..Le
 2040 73732074 68616e20 6f722065 7175616c  ss than or equal
 2050 73203221 20282564 290a0000 00000000  s 2! (%d).......
 2060 5b25645d 20537461 636b2069 74656d3a  [%d] Stack item:
 2070 2025642c 20686561 70206974 656d3a20   %d, heap item:
 2080 25640a00                             %d..
Contents of section .eh_frame_hdr:
 2084 011b033b 38000000 06000000 9cefffff  ...;8...........
 2094 6c000000 dcefffff 94000000 ecefffff  l...............
 20a4 ac000000 1cf0ffff 54000000 05f1ffff  ........T.......
 20b4 c4000000 26f1ffff e4000000           ....&.......
Contents of section .eh_frame:
 20c0 14000000 00000000 017a5200 01781001  .........zR..x..
 20d0 1b0c0708 90010000 14000000 1c000000  ................
 20e0 c0efffff 26000000 00440710 00000000  ....&....D......
 20f0 24000000 34000000 28efffff 40000000  $...4...(...@...
 2100 000e1046 0e184a0f 0b770880 003f1a3a  ...F..J..w...?.:
 2110 2a332422 00000000 14000000 5c000000  *3$"........\...
 2120 40efffff 10000000 00000000 00000000  @...............
 2130 14000000 74000000 38efffff 30000000  ....t...8...0...
 2140 00000000 00000000 1c000000 8c000000  ................
 2150 [illegible]
```

Fig. 28: Classical program visualised in C-code, compiled x86 assembly, and with its final memory layout once it is converted into machine code



```
function quantumProgram(q0: boolean, q1: boolean) : boolean {
    let q3 = 0;
    let q4 = 0;
    // ... PROGRAM ...
    return q2;
}

let result = quantumProgram(true, false);
```

```
#include <stdbool.h>
// classical input is: |10000>, or 0b10000 = 0x10
void quantumProgram(int* classicalIOA) {
    // INPUT
    char q0 = *classicalIOA & 0x10; // mask: 0b10000
    char q1 = *classicalIOA & 0x8; // mask: 0b01000
    // OUTPUT
    char q2 = *classicalIOA & 0x4; // mask: 0b00100
    // AUXILIARY
    char q3 = *classicalIOA & 0x2; // mask: 0b00010
    char q4 = *classicalIOA & 0x1; // mask: 0b00001

    // qubits are initialised as the basis state|10000>
    // PROGRAM modifies classicalIOA according to classical circuit ...
}

bool quantumOR(bool A, bool B) {
    // encode "input" by creating binary string: 0b[10]000
    int inputStr = (A << 4) + (B << 3);
    quantumProgram(&inputStr); // run program on classical comp
    // extract "output"
    char result = inputStr & 0x4; // mask: 0b00[1]00
    return result ? true : false;
}

void main() {
    bool A = true;
    bool B = false;

    if (quantumOR(A, B)) {
        printf("OR of A and B is TRUE");
    } else {
        printf("OR of A and B is FALSE");
    }
}
```

Fig. 29: Quantum program visualised in terms of classical concepts

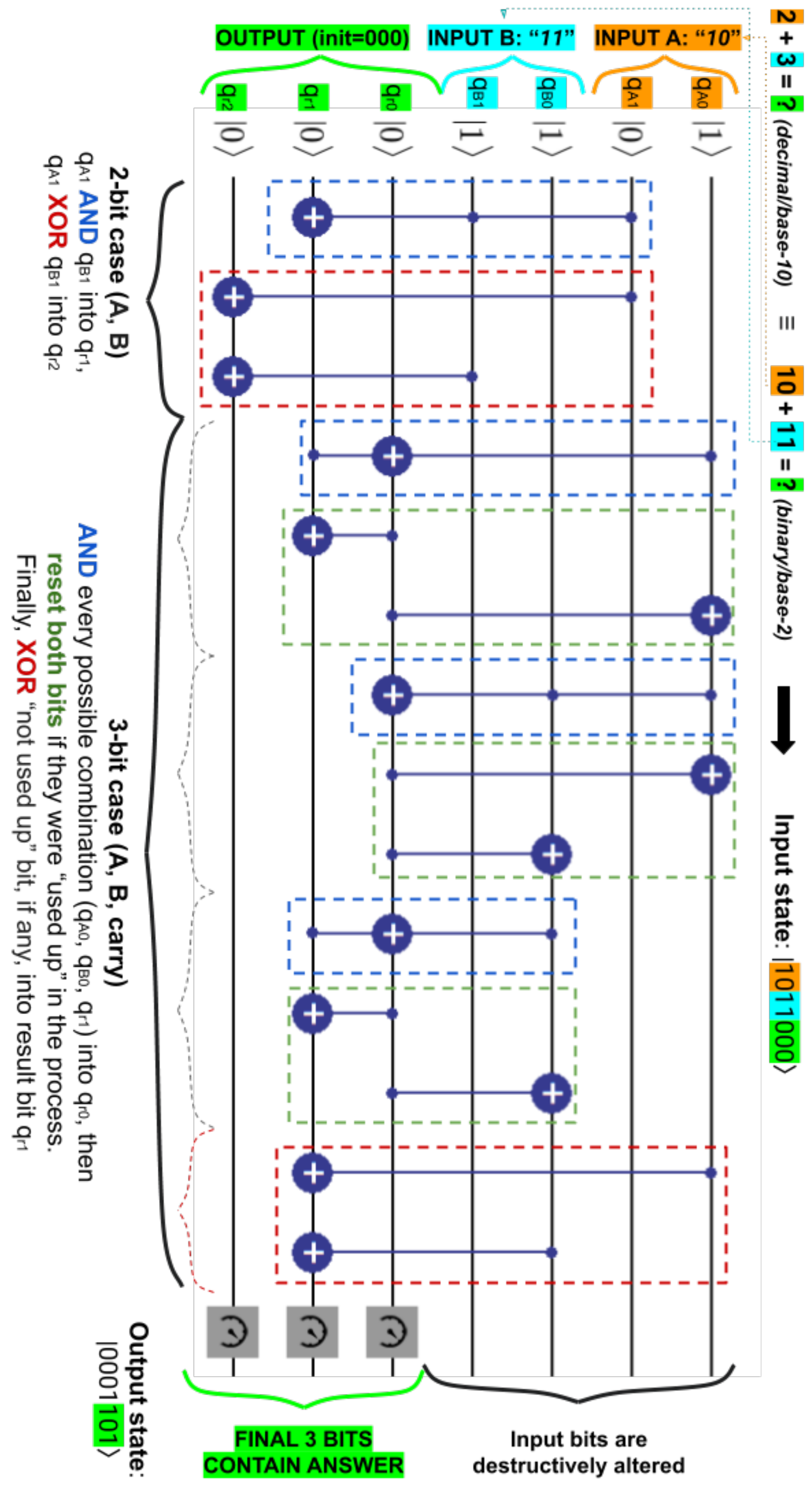

2 + 3 = ? (decimal/base-10)
≡
10 + 11 = ? (binary/base-2)
Input state: |1011000⟩
INPUT A: "10"
INPUT B: "11"
OUTPUT (init=000)
$q_{A0}$ $|1\rangle$
$q_{A1}$ $|0\rangle$
$q_{B0}$ $|1\rangle$
$q_{B1}$ $|1\rangle$
$q_{r0}$ $|0\rangle$
$q_{r1}$ $|0\rangle$
$q_{r2}$ $|0\rangle$
2-bit case (A, B)
$q_{A1}$ AND $q_{B1}$ into $q_{r1}$,
$q_{A1}$ XOR $q_{B1}$ into $q_{r2}$
3-bit case (A, B, carry)
AND every possible combination ($q_{A0}$, $q_{B0}$, $q_{r1}$) into $q_{r0}$, then
reset both bits if they were "used up" in the process.
Finally, XOR "not used up" bit, if any, into result bit $q_{r1}$
Output state: |0001101⟩
FINAL 3 BITS CONTAIN ANSWER
Input bits are destructively altered

Fig. 30: Quantum 2-bit adder. Computes any 2-bit A + B = (3-bit) C, deterministically. Binary inputs here are A=01, B=11. (Highlight colours do not correspond to the ones used in Figure 29!)