

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

---

SCUOLA DI SCIENZE  
Corso di Laurea in Informatica

# Static Analysis of Resources in QASM: Estimation of the Number of Qubits

Relatore:  
Chiar.mo Prof.  
Ugo Dal Lago

Presentata da:  
Damiano Scevola

Sessione I  
Anno Accademico 2020/2021

*A nonno Francesco*

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Quantum Computing</b>	<b>6</b>
2.1	Quantum states . . . . .	6
2.1.1	Deterministic systems . . . . .	6
2.1.2	Probabilistic systems . . . . .	7
2.1.3	Quantum systems . . . . .	7
2.2	Qubits and quantum registers . . . . .	8
2.2.1	Bits and qubits . . . . .	8
2.2.2	Measurement . . . . .	9
2.2.3	Quantum registers . . . . .	10
2.2.4	Entanglement . . . . .	10
2.3	Quantum Gates . . . . .	11
2.3.1	Hadamard gate . . . . .	11
2.3.2	Controlled NOT gate . . . . .	12
2.3.3	Phase shift gate . . . . .	12
2.4	Quantum Circuits . . . . .	13
2.4.1	Deutsch algorithm . . . . .	13
2.4.2	Deutsch-Jozsa algorithm . . . . .	15
<b>3</b>	<b>OpenQASM</b>	<b>17</b>
3.1	Header . . . . .	17
3.2	Lexer rules . . . . .	18
3.3	Types . . . . .	19
3.3.1	Quantum types . . . . .	19
3.3.2	Classical types . . . . .	19
3.4	Index identifiers . . . . .	21
3.5	Global statements . . . . .	21
3.5.1	Subroutine definition . . . . .	22
3.5.2	Quantum gate definition . . . . .	23
3.5.3	Quantum declaration statements . . . . .	23

3.6	Statements . . . . .	23
3.6.1	Expression statements . . . . .	24
3.6.2	Assignment statements . . . . .	26
3.6.3	Classical declarations . . . . .	27
3.6.4	Branching statements . . . . .	28
3.6.5	Loop statements . . . . .	28
3.6.6	Quantum statements . . . . .	28
<b>4</b>	<b>Static Analysis and Symbolic Execution</b>	<b>30</b>
<b>5</b>	<b>Estimation of the Number of Qubits</b>	<b>31</b>

# Chapter 1

## Introduction

The idea of quantum computing was conceived in the 1970s, when researchers of the caliber of Feynman, Manin and Deutsch started to theorize that the properties of quantum mechanics might prove useful for performing computations differently from the classical paradigm. The main concept that lays at the core of this new model is the superposition of states, that is the simultaneous co-existence of multiple computational states at the same moment, with the possibility to perform the same operation to all of those states with a single computational step. Physical devices that are capable of such power are called “quantum computers”, and they constitute a polynomial-time implementation of non-determinism, thus achieving an exponential-order level of parallelism. This fact invalidates the assumption that some problems (NP-complete ones, for instance) require too much time to be solved compared to human life duration, hence it opens the doors to a new variety of applications, and it also tears down the ones which rely on this assumption (like RSA cryptography). Literature usually refers to this scenario as “Quantum Supremacy”.

Quantum computers operate through qubits, which are the quantum equivalent of bits, and can have not only 0s and 1s as values, but also combinations of them. Qubits are sequenced together to form quantum registers, which hold the actual computational power of quantum computers. For instance, a 3-qubit register can be in a superposition of the states  $|000\rangle, |001\rangle, |010\rangle, \dots, |111\rangle$ , that are  $2^3 = 8$  base states. Generalizing the previous result, we notice that an  $n$ -qubit register can represent up to  $2^n$  states simultaneously, thus showing where the exponential speedup resides.

Some companies like Google and IBM have already built fully-functional quantum computers. However, all of them have a limited number of qubits due to the elevate cost of implementing them physically. Therefore, it is crucial that quantum programmers are thrifty with the usage of qubits while designing quantum circuits. The aim of this document is to show a simple tool which is able to analyze programs written in QASM language (Quantum Assembly) and check whether the number of qubits that are *effectively* used is bounded by a certain expression. We start by introducing some concepts of

quantum computing, then we describe the QASM language and we provide details about the features that are taken into account by the analyzer. Then we proceed by going through the concepts of static analysis and symbolic execution, and finally we describe the tool that performs the estimation of the number of qubits.

# Chapter 2

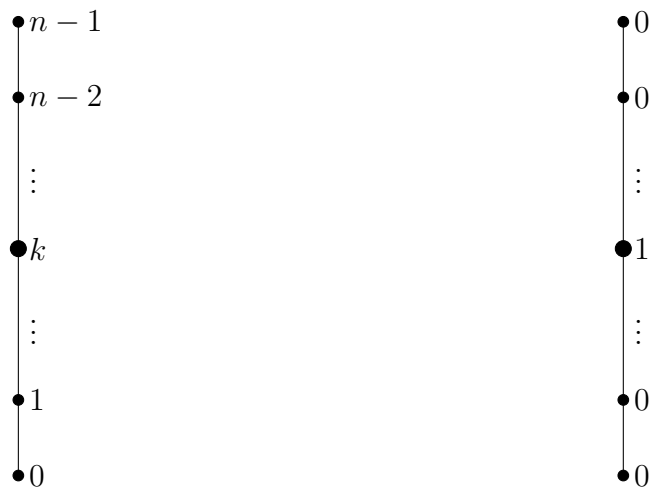
## Quantum Computing

### 2.1 Quantum states

#### 2.1.1 Deterministic systems

In classical systems, when we need to encode the state of an object, we often use numbers. For example, consider a simple case where we have a skyscraper with  $n$  floors (counting the ground floor also), and we want to represent on which one an employee is located. Trivially, this can be done using an integer number  $k$ , and if we assume there are no underground floors, then we can state  $k \in \{0, 1, \dots, n-1\}$ .

Another option for representing such a state would be using a *state vector*, that is a vector  $\mathbf{v} \in \{0, 1\}^n$  whose entries are all zeros except for the  $k$ -th, which is a 1.



Such an approach seems exaggerated for this simple case, but it is crucial to understand how quantum systems work. Let's take a look at another scenario.

### 2.1.2 Probabilistic systems

Imagine now that the floor the employee is on is not known for sure, but we are only given probabilities of her being on each floor. The state vector now looks something like:

$$\mathbf{v} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} \in [0, 1]^n$$

We now have a generalized version of our previously defined state vector, because each entry  $p_i$  is a real number between 0 and 1. Since we are talking about probabilities, we set the constraint that all the entries of  $\mathbf{v}$  must sum up to 1.

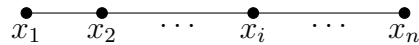
$$\sum_{i=0}^{n-1} p_i = 1$$

It should be noticed that there is no way to represent probabilistic states with a single number like we do for deterministic ones. It is clear, however, that in case we know the exact location of the employee, the vector would collapse to the previous case. In fact, if we are certain that the employee is on the  $k$ -th floor, then  $p_k = 1$ , and  $\forall i \in \{0, 1, \dots, n-1\} \setminus \{k\}$ ,  $p_i = 0$ .

Now it is time to define state vectors for quantum systems.

### 2.1.3 Quantum systems

Consider a quantum object (e.g. an electron or a photon), and assume its position can be measured in the domain  $\{x_1, x_2, \dots, x_n\}$ .



A horizontal line with points labeled  $x_1, x_2, \dots, x_i, \dots, x_n$ . Each label is positioned below a point on the line, and there are ellipses between  $x_2$  and  $x_i$ , and between  $x_i$  and  $x_n$ .

To each position  $x_i$ , it corresponds a basic vector having as entries all zeros and a 1 as the  $i$ -th entry.

$$|x_1\rangle = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, |x_2\rangle = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, |x_n\rangle = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

**Definition 2.1.1** (Quantum State). A quantum state  $|\psi\rangle$  is a linear combination of the basic vectors  $|x_1\rangle, |x_2\rangle, \dots, |x_n\rangle$  having complex weights  $c_1, c_2, \dots, c_n \in \mathbb{C}$ .

$$|\psi\rangle = c_1 |x_1\rangle + c_2 |x_2\rangle + \dots + c_n |x_n\rangle$$



Therefore,  $|\psi\rangle$  can be represented as

$$|\psi\rangle = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} \in \mathbb{C}^n$$

where

$$\sum_{i=0}^{n-1} |c_i|^2 = 1$$

Since it is a weighted sum of basic states, we say that  $|\psi\rangle$  is a *superposition* of the basic states. The square norm of  $c_i$  tells us the probability of observing the particle in the position  $x_i$ , and since the particle is indeed in one of those positions, the sum of square norms must add up to 1.

The reason why we need complex numbers to represent quantum states is that quantum objects (like photons or electrons) behave both like particles and waves, so they can interfere either constructively or destructively. By using complex numbers, we can sum waves and capture the fact that they could interfere destructively, since the square norm of the sum of two complex number is not necessarily greater than both of the square norms of the addends.

## 2.2 Qubits and quantum registers

### 2.2.1 Bits and qubits

In order to understand what qubits are, we should remark the meaning of classical bits and then make a generalization.

**Definition 2.2.1** (Bit). A bit is an elementary unit of information describing the state of the simplest classical system.

As we know, a bit can only assume values in the domain  $\{0, 1\}$ .



By following the approach of the previous section, we can also describe bits using a two-dimensional state vector, where the first element stands for 0, and the last for 1:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

If we take  $|0\rangle$  and  $|1\rangle$  as basic states, we can define a quantum system by considering all normalized linear combinations of those states. Such a quantum system is called *qubit*.

**Definition 2.2.2** (Qubit). A qubit is a quantum system having  $|0\rangle$  and  $|1\rangle$  as basic states, and therefore it can be represented as an element of  $\mathbb{C}^2$ .

$$|q\rangle = c_0 |0\rangle + c_1 |1\rangle = c_0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \in \mathbb{C}^2$$

Keep in mind that the sum of the square norms must be equal to 1. In other words, the state must be *normalized*.

$$|c_0|^2 + |c_1|^2 = 1$$

**Example 2.2.1.** Here is an example of a qubit:

$$|q\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

In fact, if we check the sum of the square norms of the coefficients, we see that they add up to 1.

$$\left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} + \frac{1}{2} = 1$$

Of course, the following are also qubits:

$$|q_0\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \quad |q_1\rangle = \frac{|1\rangle - |0\rangle}{\sqrt{2}} = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

We shall notice that  $|q\rangle$ ,  $|q_0\rangle$  and  $|q_1\rangle$  are all superpositions of the basic states  $|0\rangle$  and  $|1\rangle$ .

## 2.2.2 Measurement

When we measure a qubit, the possible outcomes are only 0 and 1, and the probability of observing the one or the other is dictated by the square norms of the coefficients  $c_0$  and  $c_1$ . In addition to this, the measurement affects the internal state of the qubit permanently, making it collapse to the value that has just been measured.

**Example 2.2.2.** Consider a qubit whose state is

$$|\psi\rangle = \frac{\sqrt{3}}{2} |0\rangle + \frac{1}{2} |1\rangle$$

If we measure it, we have  $\left(\frac{\sqrt{3}}{2}\right)^2 = \frac{3}{4} = 75\%$  of probability to observe the outcome being 0, and  $\left(\frac{1}{2}\right)^2 = \frac{1}{4} = 25\%$  of probability of it being 1.

Let's say we obtained 0 from the measurement. The state of the qubit is not  $|\psi\rangle$  anymore, but it has become  $|\psi'\rangle = |0\rangle$  because the measurement made it collapse as a side effect, so if we measure it again we have 100% of chances to observe the value 0 again.

This fact is at the core of quantum mechanics, and it just tells us that in order to observe something we must inevitably interact with it and alter its internal state.

### 2.2.3 Quantum registers

Quantum computers with a single qubit are not very interesting, so we need a way to combine qubits and form registers. In order to combine quantum systems, we need to use the tensor product. For example, let's say we wanted to represent the two-qubits state  $|01\rangle$ . We would then need to compute the tensor product  $|0\rangle \otimes |1\rangle$  as follows:

$$|0\rangle \otimes |1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ 0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{bmatrix}$$

If we observe the result carefully, we can notice that the normalization constraint still holds, and the only 1 that appears in the final result corresponds to  $c_{01}$ , which is right the coefficient labelled with our state  $|01\rangle$ . If we try to do the same thing with the state  $|101\rangle$ , we will obtain:

$$|101\rangle = |1\rangle \otimes |0\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} c_{000} \\ c_{001} \\ c_{010} \\ c_{011} \\ c_{100} \\ c_{101} \\ c_{110} \\ c_{111} \end{bmatrix}$$

Since the result of the tensor product enumerates all possible states with  $n$  qubits, the dimension of the output is as large as  $2^n$ .

Of course, considering the fact that quantum computers allow superpositions of many states, we shall clarify that the previous examples have only taken into account basic states. Hence we can also have a 3-qubit register in the state

$$|\psi\rangle = \frac{|000\rangle + |010\rangle - |011\rangle - |100\rangle + |110\rangle}{\sqrt{5}}$$

### 2.2.4 Entanglement

When multiple qubits are involved to form a quantum register, it can sometimes have the property that if we measure one of the qubits, then we automatically know the value of the others. We say, then, that those qubits are *entangled*.

**Example 2.2.3.** Consider a quantum register with 2 qubits and assume its state is

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Then, if we measure the first qubit and we observe the value 1, we automatically know that the other qubit also has value 1 by just noticing that the only basic state that appears in  $|\psi\rangle$  and has a  $|1\rangle$  at the first position is  $|11\rangle$ , so we know for sure that the other one is  $|1\rangle$  too.

In this case, the measurement not only affects the state of the measured qubit making it collapse to  $|1\rangle$ , but it also collapses the state of the entangled one.

## 2.3 Quantum Gates

So far we have discussed how to represent qubits and quantum registers and what their properties are, but in order to perform a computation we need a way to manipulate them according to an algorithm and exploit their computational power.

Just like classical computers manipulate bits using logic gates, quantum computers use quantum gates to achieve their objective.

The physical implementation of quantum gates is as complicated as that of qubits, but we can represent what these gates do by using mathematical objects that operate on state vectors and keep the output normalized so that the sum of the square norms of the coefficients of the output vector is always 1. These objects are unitary matrices.

There are infinite possible quantum gates. Here we describe the set of universal ones  $\{H, CNOT, R(\theta)\}$ , which includes three gates that can be combined together to simulate an arbitrary gate.

### 2.3.1 Hadamard gate

The Hadamard gate is encoded by the following matrix:

$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The Hadamard matrix is the transition matrix from the canonical basis  $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$

to the Hadamard basis  $\left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \right\}$ , which is used to express arbitrary states in terms of superpositions of basic states, and this allows calculations to be performed on a superposition, hence on all of the basic states it is composed of with a single computational step.

Since  $H$  is unitary, by applying it twice we go back to the canonical basis, so after calculations are finished, Hadamard gate can be applied again to translate the results back into the canonical basis, ready to be measured by the observer.

**Example 2.3.1.** By applying Hadamard gate to the qubits  $|0\rangle$  and  $|1\rangle$  we get:

$$H|0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle$$

$$H|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

It is easy to verify that applying Hadamard on  $|+\rangle$  and  $|-\rangle$  we go back respectively to  $|0\rangle$  and  $|1\rangle$ .

We will represent the Hadamard gate as a box labelled with ‘H’

$$|0\rangle \text{ --- } \boxed{H} \text{ --- } \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

### 2.3.2 Controlled NOT gate

While Hadamard gate affects a single qubit, the controlled NOT gate (CNOT or CX for short) needs two of them. One is the control qubit, and the other is the target.

$$\begin{array}{c} |x\rangle \text{ --- } \bullet \text{ --- } |x\rangle \\ |y\rangle \text{ --- } \oplus \text{ --- } |x \oplus y\rangle \end{array}$$

Here  $\oplus$  denotes the binary exclusive OR operation. That is, if  $|x\rangle = |0\rangle$  then the bottom output will be left unaltered, otherwise if  $|x\rangle = |1\rangle$ , then the bottom qubit is flipped. The matrix that corresponds to the CNOT gate is:

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

### 2.3.3 Phase shift gate

Phase shift gate is a parametric gate that affects a single qubit by rotating it in the Bloch sphere, adjusting its phase. We will not go over the details of Bloch sphere representation, but we just give the matrix which encodes the phase shift here.

$$R(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

Now that we have discussed quantum gates, we can move to designing quantum circuits (or algorithms).

## 2.4 Quantum Circuits

Quantum circuits are the equivalent of algorithms for classical computing. By combining quantum gates, we can build a circuit that performs a certain computation and exploits the power of superpositions to drastically reduce the time of execution compared to a classical program.

**Definition 2.4.1** (Quantum circuit). A quantum circuit is a computational routine consisting of coherent quantum operations on quantum data, such as qubits, and concurrent real-time classical computation. It is an ordered sequence of quantum gates, measurements and resets, all of which may be conditioned on and use data from the real-time classical computation.

In order to understand how quantum circuits work, we will discuss the Deutsch algorithm for balanced and constant function classification.

### 2.4.1 Deutsch algorithm

Before describing the Deutsch algorithm, let's define what balanced and constant functions are.

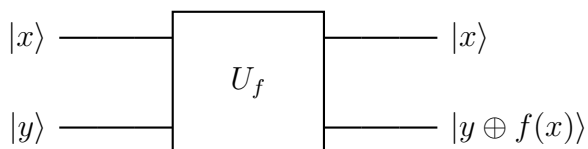
**Definition 2.4.2** (Balanced and constant functions). A function  $f : \{0, 1\} \rightarrow \{0, 1\}$  is *balanced* if  $f(0) \neq f(1)$ , otherwise it is *constant*.

The Deutsch algorithm solves the following problem:

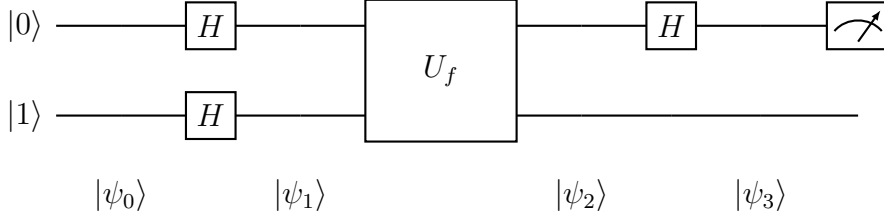
**Problem 2.4.1** (Deutsch-Jozsa). Suppose we have a function  $f : \{0, 1\} \rightarrow \{0, 1\}$  which we can evaluate, but we do not know its exact definition. Determine whether  $f$  is balanced or constant.

If we wanted to use the classical approach, we would need to evaluate  $f$  twice and compare the results. With the Deutsch algorithm, we will see that only one evaluation is sufficient.

Suppose we have a unitary matrix  $U_f$  that encodes  $f$ :



Let's now build a circuit that uses  $U_f$  and solves our problem:



In terms of matrices, the circuit corresponds to

$$(H \otimes I)U_f(H \otimes H) |0, 1\rangle$$

The algorithm starts with the state

$$|\psi_0\rangle = |0, 1\rangle$$

Then it proceeds by applying Hadamard to both qubits, putting them in a superposition:

$$|\psi_1\rangle = |+, -\rangle = \left[ \frac{|0\rangle + |1\rangle}{\sqrt{2}} \right] \left[ \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] = \frac{|0, 0\rangle - |0, 1\rangle + |1, 0\rangle - |1, 1\rangle}{2}$$

Now let's pause for a moment and ponder about what happens if we apply  $U_f$  to a generic state  $|x, -\rangle$ .

$$U_f |x, -\rangle = U_f(|x\rangle \left[ \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right]) = U_f \left[ \frac{|x, 0\rangle - |x, 1\rangle}{\sqrt{2}} \right] = |x\rangle \left[ \frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right]$$

Which simplifies to

$$U_f |x, -\rangle = |x\rangle \left[ \frac{|f(x)\rangle - |\overline{f(x)}\rangle}{\sqrt{2}} \right]$$

Where we denoted the opposite of  $f(x)$  with  $\overline{f(x)}$ . So we have

$$U_f |x, -\rangle = \begin{cases} |x\rangle \left[ \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] & \text{if } f(x) = 0 \\ |x\rangle \left[ \frac{|1\rangle - |0\rangle}{\sqrt{2}} \right] & \text{if } f(x) = 1 \end{cases}$$

We can write the last expression as

$$U_f |x, -\rangle = (-1)^{f(x)} |x\rangle \left[ \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] = (-1)^{f(x)} |x\rangle |-\rangle$$

Going back to our circuit, this means that, after applying  $U_f$  to  $|\psi_1\rangle$ , we get

$$\begin{aligned} |\psi_2\rangle &= U_f |+, -\rangle = \frac{U_f |0, -\rangle + U_f |1, -\rangle}{\sqrt{2}} \\ &= \frac{(-1)^{f(0)} |0\rangle + (-1)^{f(1)} |1\rangle}{\sqrt{2}} |-\rangle \end{aligned}$$

Now let's see what happens in the two cases where  $f$  is balanced or it is constant. If  $f$  is balanced  $f(0) \neq f(1)$ , then the top qubit becomes either  $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$  or  $\frac{-|0\rangle + |1\rangle}{\sqrt{2}}$  depending on which way it is balanced, but in either case we can write

$$|\psi_2\rangle = (\pm 1) \frac{|0\rangle - |1\rangle}{\sqrt{2}} |-\rangle = (\pm 1) |-, -\rangle$$

Since the coefficient  $(\pm 1)$  is global, it can be ignored due to the normalization constraint, so we can say  $|\psi_2\rangle = |-, -\rangle$  if  $f$  is balanced.

On the other hand, if  $f$  is constant  $f(0) = f(1)$ , then the top qubit becomes either  $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$  or  $\frac{-|0\rangle - |1\rangle}{\sqrt{2}}$ , but in either case we can write

$$|\psi_2\rangle = (\pm 1) \frac{|0\rangle + |1\rangle}{\sqrt{2}} |-\rangle = (\pm 1) |+, -\rangle$$

Here the argument for  $(\pm 1)$  still holds, so we can say  $|\psi_2\rangle = |+, -\rangle$  if  $f$  is constant. Summing up, we have

$$|\psi_2\rangle = \begin{cases} |+, -\rangle & \text{if } f \text{ is constant} \\ |-, -\rangle & \text{if } f \text{ is balanced} \end{cases}$$

Finally, the last Hadamard gate brings our state back to the canonical basis, and so

$$|\psi_3\rangle = \begin{cases} |0, -\rangle & \text{if } f \text{ is constant} \\ |1, -\rangle & \text{if } f \text{ is balanced} \end{cases}$$

Now we can measure the top qubit, and we can be sure about the function being balanced or constant by just looking at the result.

Notice that the quantum algorithm only evaluated the function once, namely on the superposition  $|+, -\rangle$ . The generalized version of this algorithm, the Deutsch-Jozsa algorithm, shows the speedup in the computation even more.

## 2.4.2 Deutsch-Jozsa algorithm

Although we are not going to illustrate all the analysis of the Deutsch-Jozsa algorithm, we will still define the problem, show the circuit that solves it and state some facts about its computational complexity compared to the classical case. Let's generalize the definition of balanced and constant functions.

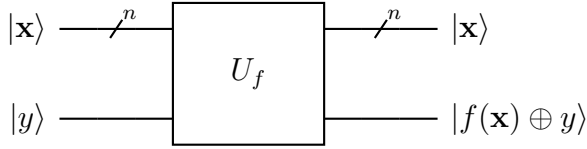


**Definition 2.4.3** (Balanced and constant functions). A function  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  is *balanced* if exactly half of the inputs go to 0 (and the other half go to 1), while it is *constant* if all of the inputs go to the same output (either 0 or 1).

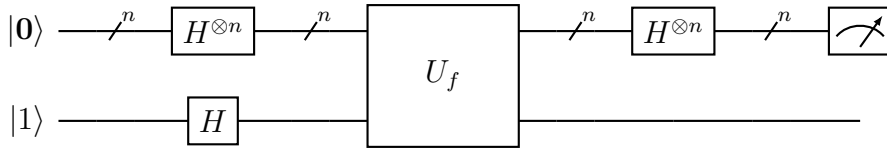
The Deutsch-Jozsa problem is the same as the Deutsch one, with the assumptions that  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  and that  $f$  is guaranteed to be either constant or balanced. With the classical approach, in order to determine whether  $f$  is balanced or constant we need to evaluate it at most on one plus half of the inputs. This is because as soon as we detect two different output values, we can conclude that the function is balanced, but if  $f$  is constant, we cannot be sure of it until we check that more than half of its outputs are the same. So the worst case scenario requires  $1 + \frac{2^n}{2} = 1 + 2^{n-1}$  evaluations, which is exponential with respect to the size of the input.

Let's now delve into the quantum world and contemplate its power.

Since we are given the possibility to evaluate  $f$ , assume there is a unitary matrix  $U_f$  that fulfills the task:



Here we denoted the top input  $|\mathbf{x}\rangle = |x_0 x_1 \dots x_{n-1}\rangle$  in bold and the wire mark  $\text{---}^n$  indicates that it is a bit sting with length  $n$ . The circuit that solves the problem is the following:



We can notice that it is very similar to the Deutsch circuit, with the only difference that there are  $n$  qubits as the top input, and we denoted the fact that we are applying Hadamard to each one of them with the gate  $H^{\otimes n}$ , that is the tensor product of  $n$  Hadamard matrices.

This circuit is capable to tell us if  $f$  is balanced or constant in a single run. In other words, we have an exponential speedup compared to the classical algorithm for the same problem. There are many other problems that benefit of the quantum speedup, and some of them are of a practical nature. This is why quantum computing is so powerful. In order to design and execute quantum algorithms, we need a language that allows us to describe them in terms of gates and classical instructions. In the next chapter we will explore OpenQASM, which is a powerful and compact language that achieves the goal.

# Chapter 3

## OpenQASM

OpenQASM is an imperative programming language designed for quantum algorithms and applications. It has a dual nature as an assembly language and as a hardware description language. There are multiple versions of OpenQASM, but we will describe a simplified version of OpenQASM 3.0.

The formal grammar of the language is presented using the ANTLRv4 syntax, and starts with:

```
program
    : header (globalStatement | statement)*
    ;
```

So we first describe the header and then we proceed to discuss the other aspects of the language, which also include statements. In each section we will present the formal grammar and comment it providing informal semantic and context.

### 3.1 Header

```
header
    : version? include*
    ;

version
    : 'OPENQASM' (Integer | RealNumber) SEMICOLON
    ;

include
    : 'include' StringLiteral SEMICOLON
    ;
```

The header consists in an optional version string followed by zero or more ‘include’ statements, that are used to import gates and functions from other files at compile time. A typical example of what the header looks like is:

```
// Header
OPENQASM 3.0;
/*
comment
*/
include "stdgates.qasm";
```

We can see here that comments are C-like, and string literals are wrapped by double quotes.

From now on we will proceed in a bottom-up fashion, describing all the features of the language that are used in more complex rules, starting from lexicon, types and index identifiers up to global and non-global statements.

## 3.2 Lexer rules

```
LBRACKET: '[';
RBRACKET: ']';
```

```
LBRACE: '{';
RBRACE: '}';
```

```
LPAREN: '(';
RPAREN: ')';
```

```
COLON: ':';
SEMICOLON: ';';
```

```
DOT: '.';
COMMA: ',';
```

```
EQUALS: '=';
ARROW: '->';
```

```
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
```

```

Constant: ( 'pi' | 'tau' | 'euler' );

Whitespace: [ \t]+ -> skip;
Newline: [ \r\n]+ -> skip;

fragment Digit: [0-9];
Integer: Digit+;

fragment ValidUnicode: [ \p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}];
fragment Letter: [A-Za-z];
fragment FirstIdCharacter: '_' | '$' | ValidUnicode | Letter;
fragment GeneralIdCharacter: FirstIdCharacter | Integer;

Identifier: FirstIdCharacter GeneralIdCharacter*;

fragment SciNotation: [eE];
fragment PlusMinus: PLUS | MINUS;
fragment Float: Digit+ DOT Digit*;
RealNumber: Float (SciNotation PlusMinus? Integer)?;

LineComment: '//' ~[\r\n]* -> skip;
BlockComment: '/*' .*? '*/' -> skip;

StringLiteral
: '"' ~["\r\t\n]+? '"'
| '\'' ~['\r\t\n]+? '\''
;

```

## 3.3 Types

In OpenQASM, types are divided in two categories: quantum types and classical types.

### 3.3.1 Quantum types

```

quantumType
: 'qubit'
| 'qreg'
;

```

We will describe how to make a quantum declaration in the ‘Global statements’ section.

### 3.3.2 Classical types

```

classicalType

```

```

        : singleDesignatorType designator
        | doubleDesignatorType doubleDesignator
        | noDesignatorType
        | bitType designator?
        ;

bitType
    : 'bit'
    | 'creg'
    ;

singleDesignatorType
    : 'int'
    | 'uint'
    | 'float'
    | 'angle'
    ;

doubleDesignatorType
    : 'fixed'
    ;

noDesignatorType
    : 'bool'
    ;

designator
    : LBRACKET expression RBRACKET
    ;

doubleDesignator
    : LBRACKET expression COMMA expression RBRACKET
    ;

```

A **designator** is simply a parameter for the type, for example `int[16]` indicates a 16-bit integer, where 16 is the designator. The single designator for the `float` and `angle` types is for specifying the precision. Another example could be `fixed[7, 24]`, which indicates a  $24 + 7 + 1 = 32$ -bit fixed point signed number having 7 integer bits and 24 fractional bits.

`bitType` has a single designator that stands for the number of bits of the array (or register, equivalently), while only the boolean type has no designator.

## 3.4 Index identifiers

```
indexIdentifier
  : Identifier rangeDefinition
  | Identifier (LBRACKET expressionList RBRACKET)?
  | indexIdentifier '||' indexIdentifier
  ;

indexIdentifierList
  : (indexIdentifier COMMA)* indexIdentifier
  ;

indexEqualsAssignmentList
  : (indexIdentifier equalsExpression COMMA)* indexIdentifier
    equalsExpression
  ;

rangeDefinition
  : LBRACKET expression? COLON expression? (COLON expression)?
    RBRACKET
  ;
```

These rules are used to refer to specific bits or qubits in both classical and quantum registers. For better understanding, we provide some examples given two registers **x** and **y**, each having 8 bits:

- **x || y**: the 16-bit concatenation of **x** and **y**
- **x[0]**: the first bit of **x**
- **y[-1]**: the last bit of **y**
- **x[0,3,5]**: bits 0, 3 and 5 of **x**
- **x[0:6]**: the first 6 bits of **x**
- **y[0:2:7]**: every second bit of **y** from 0 to 7 (stops at 6)
- **y[-4:-1]**: the last three bits of **y**

## 3.5 Global statements

After the header, the program consists in a sequence of statements and global statements. Global statements are defined by the rule:

```

globalStatement
  : subroutineDefinition
  | quantumGateDefinition
  | quantumDeclarationStatement
  ;

```

In this section we will discuss each type of global statement.

### 3.5.1 Subroutine definition

```

subroutineDefinition
  : 'def' Identifier ( LPAREN classicalArgumentList? RPAREN )?
    quantumArgumentList returnSignature? subroutineBlock
  ;

```

```

classicalArgumentList
  : (classicalArgument COMMA)* classicalArgument
  ;

```

```

classicalArgument
  : classicalType association
  ;

```

```

quantumArgumentList
  : (quantumArgument COMMA)* quantumArgument
  ;

```

```

quantumArgument
  : quantumType designator? association
  ;

```

```

association
  : COLON Identifier
  ;

```

```

returnStatement: 'return' statement;

```

```

returnSignature
  : ARROW classicalType
  ;

```

```

subroutineBlock
  : LBRACE statement* returnStatement? RBRACE

```

;

Subroutines have two sets of arguments: classical arguments and quantum arguments. They also have an optional return signature, which is used to specify the return type of the function. Here is an example of a subroutine definition:

```
def f3 (int[5]:x, bit[4]:b) qreg[5]:q -> int[8] {
    bit c[5] = "0110";
    c[2,3] = b[3,1];
    if (c[2] == 1 && c[3] == 0) {
        x += 2;
    } else {
        if (c >= 3) {
            x += 3;
        } else {
            x -= 4;
        }
    }
    return x;
}
```

### 3.5.2 Quantum gate definition

We will not delve into the actual grammar of quantum gate definitions, since the analysis that we are going to discuss in the next chapters does not take them into account. In fact, our analyzer treats them as black-boxes that entangle all the input qubits together.

### 3.5.3 Quantum declaration statements

```
quantumDeclarationStatement: quantumDeclaration SEMICOLON;

quantumDeclaration
    : quantumType indexIdentifierList
    ;
```

As we can notice, a quantum declaration is simply the declaration of a qubit (or an array of them). It must be done globally, since qubits cannot be allocated dynamically.

## 3.6 Statements

```
statement
    : expressionStatement
    | assignmentStatement
    | classicalDeclarationStatement
```



```

| branchingStatement
| loopStatement
| quantumStatement
;

```

In our simplified version of OpenQASM we consider six types of statement: expressions, assignments, classical declarations, branching if-then-elses, for and while loops and quantum statements. We now go over these statement types one by one, and we will be commenting only those that require further clarification, since most of them have similar syntax and semantic to other well-known programming languages.

### 3.6.1 Expression statements

```

expressionStatement
: expression SEMICOLON
;

expressionList
: (expression COMMA)* expression
;

expression
: expressionTerminator
| unaryExpression
| xOrExpression
| expression '|' xOrExpression
;

xOrExpression
: bitAndExpression
| xOrExpression '^' bitAndExpression
;

bitAndExpression
: bitShiftExpression
| bitAndExpression '&' bitShiftExpression
;

bitShiftExpression
: additiveExpression
| bitShiftExpression ('<<' | '>>') additiveExpression
;

```

```

additiveExpression
    : multiplicativeExpression
    | additiveExpression (PLUS | MINUS) multiplicativeExpression
    ;

multiplicativeExpression
    : expressionTerminator
    | unaryExpression
    | multiplicativeExpression (MUL | DIV | MOD) (
        expressionTerminator unaryExpression)
    ;

unaryExpression
    : unaryOperator expressionTerminator
    ;

expressionTerminator
    : Constant
    | Integer
    | RealNumber
    | Identifier
    | StringLiteral
    | MINUS expressionTerminator
    | LPAREN expression RPAREN
    | expressionTerminator LBRACKET expression RBRACKET
    | expressionTerminator incrementor
    ;

unaryOperator
    : '~' | '!'
    ;

incrementor
    : '++'
    | '--'
    ;

```

The grammar for expressions is defined hierarchically to implement the conventional evaluation order of operators.

Expression statements do not affect the memory unless incrementors are present, in which case the respective variables are updated accordingly to the incrementor sign.

Even if they are not included in the `expressionStatement` rule, we show the grammar for boolean expressions here, because they will be used in other statement types from

now on.

```
booleanExpression
  : membershipTest
  | comparisonExpression
  | booleanExpression logicalOperator comparisonExpression
  ;
```

```
comparisonExpression
  : expression // if (expression)
  | expression relationalOperator expression
  ;
```

```
relationalOperator
  : '>'
  | '<'
  | '>='
  | '<='
  | '=='
  | '!='
  ;
```

```
logicalOperator
  : '&&'
  | '||'
  ;
```

```
membershipTest
  : Identifier 'in' setDeclaration
  ;
```

```
setDeclaration
  : LBRACE expressionList RBRACE
  | rangeDefinition
  | Identifier
  ;
```

### 3.6.2 Assignment statements

```
assignmentStatement: (classicalAssignment |
  quantumMeasurementAssignment) SEMICOLON;
```

```
classicalAssignment
```

```

    : indexIdentifier assignmentOperator (expression |
      indexIdentifier)
    ;

assignmentOperator
  : EQUALS
  | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '~=' | '^=' | '
    '<=>' | '>=>'
  ;

quantumMeasurement
  : 'measure' indexIdentifierList
  ;

quantumMeasurementAssignment
  : quantumMeasurement (ARROW indexIdentifierList)?
  | indexIdentifierList EQUALS quantumMeasurement
  ;

```

Assignments are defined in the usual way, with the addition of quantum measurements, that simply tell the hardware to perform a measurement on certain qubits and store the result in classical variables. As we explained in the previous chapter, we remark that after being measured, qubits collapse to the observed state, so measurements are usually done at the end of the algorithm.

### 3.6.3 Classical declarations

```

classicalDeclarationStatement
  : (classicalDeclaration | constantDeclaration) SEMICOLON
  ;

classicalDeclaration
  : singleDesignatorDeclaration
  | doubleDesignatorDeclaration
  | noDesignatorDeclaration
  | bitDeclaration
  ;

singleDesignatorDeclaration
  : singleDesignatorType designator (identifierList |
    equalsAssignmentList)
  ;

```

```

doubleDesignatorDeclaration
    : doubleDesignatorType doubleDesignator (identifierList |
        equalsAssignmentList)
    ;

noDesignatorDeclaration
    : noDesignatorType (identifierList | equalsAssignmentList)
    ;

bitDeclaration
    : bitType (indexIdentifierList | indexEqualsAssignmentList)
    ;

constantDeclaration
    : 'const' equalsAssignmentList
    ;

```

### 3.6.4 Branching statements

```

programBlock
    : statement
    | LBACE statement* RBACE
    ;

branchingStatement
    : 'if' LPAREN booleanExpression RPAREN programBlock ('else'
        programBlock)?
    ;

```

### 3.6.5 Loop statements

```

loopStatement: loopSignature programBlock;

loopSignature
    : 'for' membershipTest
    | 'while' LPAREN booleanExpression RPAREN
    ;

```

### 3.6.6 Quantum statements

```

quantumStatement
    : quantumInstruction SEMICOLON
    ;

```

```

quantumInstruction
    : quantumGateCall
    | quantumMeasurement
    ;

quantumGateCall
    : quantumGateName (LPAREN expressionList? RPAREN)?
      indexIdentifierList
    ;

quantumGateName
    : 'CX'
    | 'U'
    | 'reset'
    | Identifier
    ;

```

Here it is worth mentioning that quantum gates can be called on multiple qubits, and this is the only possible cause of entanglement.

## Chapter 4

# Static Analysis and Symbolic Execution

## Chapter 5

# Estimation of the Number of Qubits