CEN/CLC/JTC 22/WG 3 "Quantum Computing and Simulation"

Convenor: **Lefebvre Catherine Mme**

CEN-CLC-JTC 22-WG 3_LayerModel_Draft06

Document type	Related content	Document date	Expected action
Project / Draft	Meeting: VIRTUAL 22 Jan 2025	2025-01-20	**COMMENT/REPLY** by 2025-01-28

Description

Dear Members,

Please find attached the new draft (V.6) on Layer Model.

Kind regards

CEN/TC XXX

Date: 20YY-XX

prEN XXXXX:20YY

Secretariat: XXX

JTC 22 WG3 Quantum Computing

Layer Model for gate-based quantum computers

Draft 06, 2025-01-20

CCMC will prepare and attach the official title page.

Contents

Page

European foreword

This document (prEN XXXX:20YY) has been prepared by Technical Committee CEN/TC JTC22/WG3 "Quantum Computing and simulation", the secretariat of which is held by XXX.

This document is currently submitted to the CEN Enquiry/Formal Vote/Vote on TS/Vote on TR.

This document has been prepared under a Standardization Request given to CEN by the European Commission and the European Free Trade Association, and supports essential requirements of EU Directive(s) / Regulation(s).

Introduction

A layer model is an abstract description of a (computing) system via a common stack of layers. The layer model for gate-based quantum computing slices down the overall complexity of quantum computing into two main layer models of addressing the whole system. The lower main layer model addresses mainly hardware, and it is dependent of the physical platform. The upper main layer model addresses mostly software at a higher level of abstraction.

Each of these two main layer models comprises a stack of inner layers. The lower (hardware) main layer model comprises multiple stacks, one for each identified architecture family.

The higher up in the stack the more hardware-agnostic the inner layers of the upper (software) main layer model will gradually be. By agnostic we mean that the same system works for different quantum computing hardware platforms such as solid state quantum computing, ion traps, neutral atoms, optical quantum computing and topological quantum computing.

This structure decouples the software design from the hardware design to some extent, which has clear advantages, such as the reputability of algorithms for different hardware. At the same time the structure does not impose a fully hardware-agnostic upper main layer model to encompass the design of quantum hardware and software in a co-design approach, that is, adapt software to make optimal use of the hardware used and the vice versa. This approach is inevitable for current and near-future quantum computer development, just as it turned out to be vital for classical computers in early stage and current classical computing disciplines, e.g., in microcontroller design.

The first purpose of this document is to define a common language that will be used to describe the features and functional requirements for each layer of the stack of a quantum computer. Another purpose is to analyse and describe the interaction between the layers by means of well-defined interfaces. These are essential steps towards interworking between modules from different origins. The functional description of each layer should offer sufficient guidance on where a desired functionality should be described, and what kind of exchange is needed with other modules through the interfaces. The boundaries between the layers are natural locations for such interfaces. Correctly defining such boundaries requires careful analysis of the interaction between the layers.

1 Scope

This document describes a layer model that covers the entire stack of universal gate-based quantum computers. The group of lower-level (hardware) layers are organized in different hardware stacks tailored to different hardware architectures, while the group of higher-level (software) layers are built on top of these and expected to be common for all quantum computing systems. The higher-up in the stack, the more agnostic it will be from underlying layers. Reducing the dependencies between higher and lower layers is a crucial point for optimized quantum computations. A co-requisite point is to allow for a free but well-defined flow of information up and down the higher and lower layers to allow for co-designing hardware and software.

The scope of this document is limited to a universal gate-based quantum-computing model, also known as a digital or circuit quantum-computing model, on multiple physical systems such as transmon, spin-qubit, ion-trap, neutral-atom, and other. This limitation keeps technologies like the universal adiabatic quantum-computing model, and its heuristic form quantum annealing, as out of scope if they do not correspond to a gate-based quantum circuit. Moreover, this limitation keeps quantum computing models that are not universal, such as quantum simulators and special purposes, as out of scope.

Limiting the scope to a universal gate-based quantum computing model is justified by expected commonalities at the higher layers, mainly above the hardware abstraction layer (HAL), up to the application layer. These commonalities imply a market for software products usable for this wide range of quantum computing technologies.

This document is limited to a high-level (functional) description of the layers involved. Additional details of the individual layers will be described in other future CEN/Trs.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply:

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp/
- IEC Electropedia: available at https://www.electropedia.org/

3.1 Terminology

Codesign - A design approach where (software) modules query lower layers for identifying the (hardware) capabilities and limitations of a system and subsequently tailor their behaviour to these capabilities and limitation. This approach allows for hardware-specific optimizations and adaptations to optimize quantum computations.

Gate-based quantum computing - A gate-based quantum computer processes a sequence of instructions (called a quantum circuit) to change the state of a quantum register with many qubits before the resulting state is queried by measurements. The instructions may comprise gates, mid-circuit measurements and state preparations. Gates are unitary operations acting on a set of qubits. A gate-based quantum computer can be characterized by a gate set, wherein the gate set is composed of gates which can be performed by the quantum computer.

ISA – An "Instruction set architecture" is a lower-level method of defining operations on a quantum computer. Instead of defining specific gates, this method defines gates (or other instructions) as operations, using pulses pulsed for a certain time, on specific qubits.

Universal gate-based quantum computing - A universal quantum computer is defined as a quantum computer being capable of processing an arbitrary quantum circuit. A universal gate-based quantum computer must have a gate set which is universal. A gate set is said to be universal if any unitary operation may be approximated to arbitrary accuracy by a quantum circuit involving only those gates [2]. The definition also comprises non-fault-tolerant universal quantum computers, which can process an arbitrary quantum circuit reliably only up to a certain length, size or gate count.

3.2 Abbreviations

API - Application Programming Interface

SDK – Software Development Kit

ISA – Instruction Set Architecture

PCB – Printed Circuit Board

SDK – Software Development Kits

QEC – Quantum Error Correction

HAL – Hardware Abstraction Layer

RF – Radio Frequency

DC – Direct Current

AWG – Arbitrary Waveform Generator

NV center - Nitrogen-Vacancy center

4 Overview

Quantum computing is an area covering many different implementations. A convenient way of specifying its requirements is via a stack of layers, as shown in Figure 4.1. The layers are chosen in such a manner that the functionality of each layer can be described in an independent manner. This causes that the interworking between these layers can be described through well-defined interfaces at the boundaries of these layers. Note that such an interface can be virtual (hidden internally within the implementation of the same origin) or real (between implementations of different origin).

The stack covers hardware layers, software layers, modules and functionalities. The legend describing each colour can be seen in Figure 4.1. While layers can be defined with well-defined boundaries, a module is an implementation that may be constructed from (smaller) modules and components, which could offer the functionality of a single layer, of multiple layers, or just of a fragment of a layer. A module may also support different operating modes, such that it complies with the requirements of multiple members and/or multiple architecture families. As such, the functionality of a module may cover multiple layers and/or families and/or members. Functionalities can be described as features or optional components a specific layer may possess. Each layer aims to be more agnostic to the exact implementation of lower layers. The layers are described in further detail in succeeding chapters.

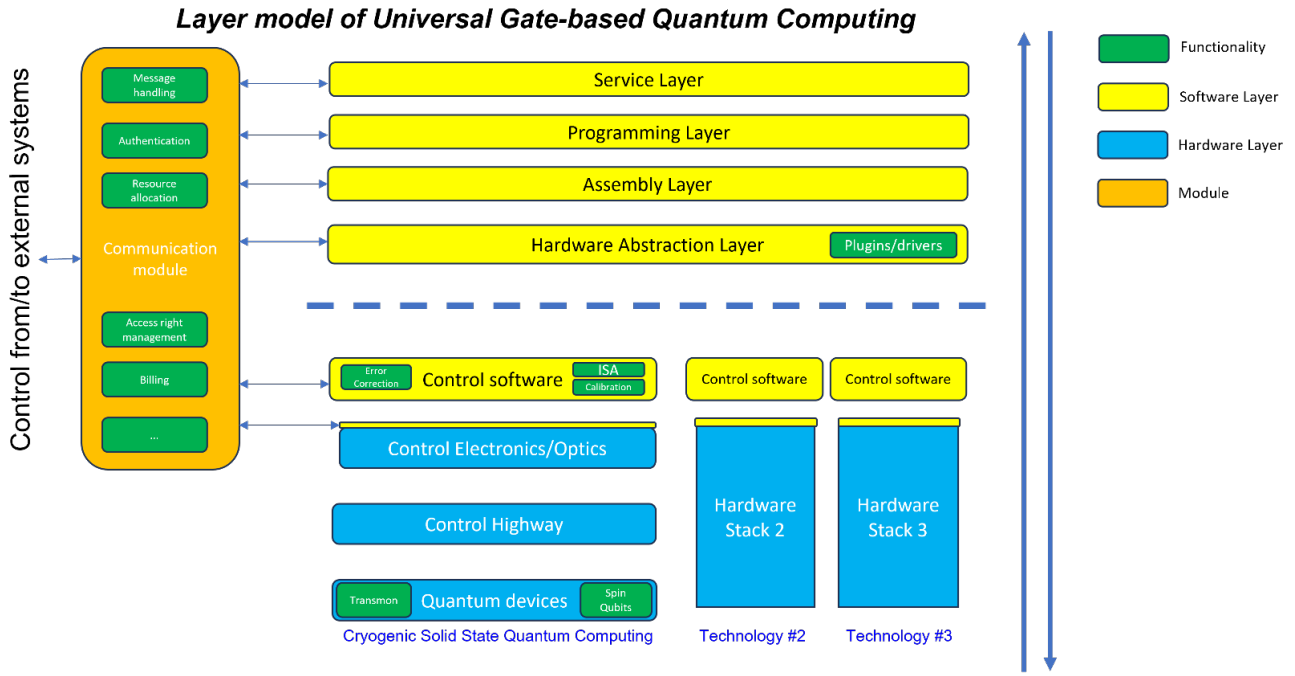

Figure 4.1 *- Overview of the layer model of quantum computing. It supports multiple hardware implementations in the hardware layers*

The layered approach allows for using different hardware stacks for specifying the requirements of different architecture families. Each architecture family can have multiple members (Cryogenic Solid State Quantum Computing, Technology #2, Technology #3) and the description of its hardware layers may account for differences between these members. The diagram in Figure 4.1 has illustrated this symbolically by drawing a non-specific blue square in place of the hardware layers for different members of the same family. The dashed blue line signifies the generalization of knowledge required for the backend.

In Figure 4.1, the cryogenic solid state quantum computer is shown in more detail, however different quantum hardware may have different hardware layer stacks as seen in technology #2 & #3. So far, the following quantum architecture families have been identified (in arbitrary order):

- Cryogenic solid-state based;
- Room temperature solid-state based;
- Trapped ions;
- Neutral atoms;
- Photonic quantum computing;
- Other architectures that may be identified in the future

These architectures are described in further detail in succeeding chapters.

5 Low level Hardware layers

5.1 Cryogenic Solid State

The members of this architecture family have in common that they all make use of a cryostat, where the quantum devices in a holder are controlled from outside the fridge by room-temperature electronics. Consequently, a huge amount of control channels is required to interconnect those two, especially when many qubits are to be controlled in a single fridge.

The following members have been identified within this architecture family:

- Transmons;
- Flux qubits;
- Semiconductor spin qubits;
- Topological qubits;
- Artificial atoms in solids.

Four hardware layers have been identified for this architecture family.

5.1.1 Layer 1 – Quantum Devices

The quantum devices in hardware layer 1 are modules with qubits that are typically operating at cryogenic temperatures and may be implemented as chip and/or on PCB. They may have though requirements on shielding, operating temperature, magnetic aspects, etc.

5.1.2 Layer 2 – Control Highway

Hardware layer 2 covers all infrastructure needed for transporting microwave, light wave, RF and DC signals (via electrical and/or optical means) between the control electronics at room temperature and the quantum devices at cryogenic temperatures. It is usually a mix of transmission lines, filtering, attenuation, amplification, (de)multiplexing, as well as means for proper thermalization. A huge number of control channels are required to control many qubits in a single fridge (which clarifies the name) and this can easily become very bulky. It could have tough requirements on aspects like heat-flow, thermal noise and vacuum properties.

5.1.3 Layer 3 – Control Electronics

Hardware layer 3 covers all electronics for generating, receiving, and processing microwave, RF and DC signals. Some implementations make use of routing/switching and/or multiplexing of control signals at room temperatures. It may have some firmware on board to guide the signal generation and signal processing.

As shown in Figure 4.1, this layer includes a small software layer in order to translate a unified way to instruct the control hardware into implementation-specific (proprietary) commands tailored to the electronics. An example is the translation of wave pulse shapes, defined as an array of samples, into proprietary commands for storing them into the memory of an AWG (Arbitrary Waveform Generator).

5.1.4 Layer 4 – Control Software

The control software refers to the software systems and tools designed to manage, coordinate and optimize operations dictated by higher level languages. Thus, the software plays a crucial role in translating higher-level quantum assembly instructions into executable instructions that can be processed by quantum processors.

This layer may include an instruction set architecture (ISA), error correction and calibration functionalities (as shown in Figure 4.1).

- **ISA** (Instruction set architecture) refers to a lower-level method of defining operations on a quantum computer. Instead of defining specific gates, this layer defines gates (or other instructions) as operations, using pulses pulsed for a certain time, on specific qubits. An example of an instruction set architecture is pulse level programming where a user can specify wave pulses on qubits instead of gates. This requires knowledge of the system's control equipment as well as the topology and qubit nature.
- **Error correction** refers to all low-level techniques to enable error-robust physical operations. Error correction as a whole is a functionality distributed over various (higher) layers. The control software handles only low-level techniques, such as detection or simple corrections, partly autonomously and partly controlled from higher layers.
- **Calibration** refers to low-level methods to stabilize the hardware by continuous monitoring of hardware performance to maintain optimal operation.

5.2 Room Temperature Solid State

The members of this architecture family have in common that solid-state qubits are all operating at room temperatures. Examples of members in this architecture family are:

- Artificial atoms in solids, such as NV centres;
- Optical quantum dots.

The description of this architecture family and associated low-level layers is to be developed in future.

5.3 Trapped Ions

The members of this architecture family can operate either at room temperature or at cryogenic temperatures (e.g. 4K). Quantum devices are controlled by electronics operating either at room temperature or under cryogenic conditions. For a larger number of qubits, the required amount of routing signals becomes bulky, and efficient thermal management, low-noise electrical and magnetic components are required.

Room temperature architectures that are identified are

- Optical qubits;
- Raman qubits;
- Spin (microwave) qubits;

Cryogenic (4K) architectures that are identified are

- Optical qubits;
- Raman qubits;
- Spin (microwave) qubits

The description of this architecture family and associated low-level layers is to be developed in future.

5.4 Neutral Atoms

Systems of individually-controlled neutral atoms, interacting with each other when excited to Rydberg states, have emerged as a possible platform for quantum information processing. The two main examples are ensembles of individual atoms trapped in optical lattices or in arrays of microscopic dipole traps separated by a few micrometres. In these platforms, the atoms are almost fully controllable by optical addressing techniques.

The description of this architecture family and associated low-level layers is to be developed in future.

5.5 Photonic quantum computing

These architectures have in common that the quantum information during computing is encoded into photonic properties. We can divide different families of photonic quantum computers in two categories, universal and non-universal quantum computers. Non-universal quantum computers cannot perform every task but can at least perform one task.

Non-universal photonic quantum computing families that are identified are:

- Boson sampling;
- Gaussian boson sampling.

Universal families that are identified are:

- Knill-Laflamme-Milburn scheme (This was a theoretical proof-of-principle, but not practically feasible);
- Measurement based quantum computing using cluster states;
- Continuous variable quantum computing.

The description of this architecture family and associated low-level layers is to be developed in future.

5.6 Other Architectures

When other architectures are identified in future, they will be added to this list.

6 Hardware Abstraction Layer (HAL)

The aim of the Hardware Abstraction Layer for universal gate-based quantum computers is to inform higher layers with capabilities and limitations supported by the underlying hardware. Layers above the HAL can use this information to hide many implementation-specific details to higher layers by offering a more unified interface. Layers above may also use this information to provide higher-level commands to programmers or programs allowing for implementing hardware-specific optimizations and adaptations.

Not all quantum computers make use of the same paradigm. Annealing quantum computers behave differently from gate-based quantum computers, and therefore their HALs might be different as well. The HAL can therefore provide information about the underlying architecture, such as for instance being “gate-based”, “annealing” or “simulation”.

A gate-based quantum computer processes a sequence of instructions to change the state of a quantum register with many qubits before the resulting state is queried by measurements. A convenient graphic representation of such a sequence has the appearance of a circuit where the elements seem to operate on one or more qubits simultaneously. Due to this convenient graphic representation, these instructions are called gates.

6.1 Organization of qubits

Quantum register: A quantum register is a system comprising multiple qubits. The HAL supports instructions to operate on such a register, for initializing, changing, and querying its state.

Width: The HAL can specify the number of available qubits and how they are organized in these registers. It can also specify if all qubits are part of a single quantum register or if are they are allocated to multiple (smaller) registers. The use of multiple registers may occur when using modular hardware architectures.

Depth: The HAL can specify the maximum depth for circuits of gates that can be executed before the calculated result becomes unreliable. This value is related to coherence time of the implementation and other imperfections of underlying hardware.

Connections: The HAL can also provide an "adjacency matrix" for each quantum register, to indicate which qubits are edge-connected. For instance, when a register has N qubits, then this adjacency matrix C has size NxN. The default of each element in this matrix is false, but if qx and qy are the indices of two adjacent qubits then C(qx,qy)=C(qy,qx)=true. Matrix C is therefore a symmetric matrix, since C(k,r)=C(r,k).

The HAL can provide additional information about the underlying architecture.

6.2 The concept of native gates

The HAL can specify a list of "native gates" supported by the underlying hardware. The name "native gate" refers to an operation for changing the quantum state of a register by means of a "single" physical action on one or more qubits simultaneously. An example is a single pulse composition that cannot be broken down into two or more shorter pulse compositions. In other words, if a gate can be divided into two or more shorter independent sequential physical actions, it is not native.

As a result, a native gate can be executed in the minimum amount of execution time. Knowledge about which gates are native is relevant information for compilers that try to optimize a circuit with respect to execution time.

Gates that can only be implemented by a sequence of two or more native gates are called "compound" gates.

The boxed example in figure 6.1 illustrates for a specific case that the single qubit gates X, Y, Rx(a), Ry(b) are all native for that implementation, while the gates Z and Rz(c) are compound gates. A similar example can be elaborated with dual qubit gates. For a specific implementation, a gate like CNOT may turn out to be compound as well when it cannot be implemented with one native dual qubit gate.

Example

The concept of native gates can be explained by the following example. Assume that a specific hardware implementation supports a mechanism to rotate a qubit via a "single" pulse composition that can be controlled with two real parameters "a" and "b". Assume that the definition of this rotation function equals:

```
RN(a,b) = [ cos(a/2),              -j*exp(-j*b)*sin(a/2)]
          [-j*exp(j*b)*sin(a/2),                cos(a/2)]
```

Then some of the well known gates can be implemented via:

```
Rx(a) = [   cos(a/2), -j*sin(a/2)] = RN(a,0)
        [-j*sin(a/2),    cos(a/2)]
Ry(b) = [   cos(b/2),   -sin(b/2)] = RN(a,pi/2)
        [   sin(b/2),    cos(b/2)]
Rz(c) = [exp(-j*c/2),       0    ] = RN(pi,0) * RN(pi,-c/2)       * exp(j*pi)
        [          0,  exp(j*c/2)]


X     = [ 0,  1] = Rx(pi) * exp(j*pi/2) = RN(pi,0)                * exp(j*pi/2)
        [ 1,  0]
Y     = [ 0, -j] = Ry(pi) * exp(j*pi/2) = RN(pi,pi/2)             * exp(j*pi/2)
        [+j,  0]
Z     = [ 1,  0] = Rz(pi) * exp(j*pi/2) = RN(pi,pi) * RN(pi,pi/2) * exp(-j*pi/2)
        [ 0, -1]
```

In this hardware implementation, Rx(a), Ry(b), X, Y can be considered as native gates. The gates Rz(c) and Z are to be combined from two sequential native gates, so they are compound. Knowledge about which gates are native is relevant for quantum algorithms that try to find an optimal circuit representation in terms of execution time.

Figure 6.1 *- Example of a specific hardware implementation, where Rx, Ry, X, Y are native gates and Rz, Z are compound gates*

6.3 Concept of primitive gates

A compiler or interpreter does not always know how to convert well-known gates into a smart combination of native gates for any possible set of native gates. In those cases, a fall-back situation should be supported by the HAL in terms of predefined solutions for well-known gates like Rx(a), Ry(b), Rz(c), X, Y, Z, H, S, T, CNOT, etc.

Therefore, the HAL can specify a list of "primitive gates" that it can emulate by a sequence of one or more native gates.

6.4 Concept of measurement

The HAL supports instructions to query the state of one or more qubits in a quantum register by means of a measurement. The answer will be returned as a binary string stored in a dedicated register. Note that the state will be collapsed after such a query.

The HAL also supports instructions to read out the bits in this register and/or to use these bits for instructing controlled gates.

If the hardware supports it, the HAL can also offer instructions to specify the basis for these measurements.

6.5 Interfacing considerations

A preferred way of communicating with the HAL is by means of binary instructions, preferably common for all quantum computing implementations. Therefore, it requires a list of binary commands for letting the HAL report capabilities and limitations of the underlying hardware, and for executing all aforementioned instructions.

Such an interface may also offer a convenient format for instructing a simulator that emulates a quantum computer with a limited set of qubits.

7 Assembly layer

This layer concerns quantum assembly languages, such as OpenQASM [3], that describe quantum computations according to one specific model (e.g., circuit model, measurement-based model, quantum annealing model), with a per-architecture instruction set.

There will not be a single quantum assembly language and the syntax may also differ among various implementations. Languages for gate-based quantum computing have in common that they can describe universal circuits with single qubit gates, and entangled gates such as CNOT. Due to the huge diversity of quantum computing architectures, it is not likely that a unique, widely accepted, quantum assembly language would emerge and later become a standard.

8 Programming layer

The specification of quantum algorithms using register-level representation languages is not easy for programmers. Indeed, quantum assembly programs are usually generated by a software library, from a piece of code written in a common programming language, such as Python.

In general, the Programming Layer includes all the languages, libraries, and software development facilities for coding quantum algorithms or high-level applications that use predefined quantum algorithms as subroutines.

8.1 Programming Languages and Libraries

In the quantum computing domain, Python is the most used high-level programming language. It is a general-purpose imperative language, as it allows developers to write code that specifies the steps the computer must take to accomplish the goal. Other imperative languages have been designed on purpose for quantum computing, such as Q# [4] and Silq [5].

Alternative to imperative programming is functional programming, where programs are constructed by applying and composing functions. In the quantum computing domain, there are a few functional programming languages, such as QPL [6] and Quipper [7].

Writing a program in a high-level language implies using software development kits (SDKs) that include application programming interfaces (APIs) for coding quantum algorithms from scratch, but also collections of ready-to-use quantum algorithms. The APIs may be very different, depending on the quantum computational model (quantum circuit model, quantum annealing, measurement-based quantum computation, etc.) and specific application domain (quantum optimisation, quantum machine learning, etc.).

For Python programmers, there are several advanced SDKs. Some of them are bound to proprietary hardware platforms. Other SDKs are general-purpose and support device architectures from multiple providers.

8.2 Quantum Compilation

Being high-level programs hardware-agnostic, quantum compilers are necessary to translate abstract quantum algorithms into the most efficient equivalents of themselves, considering the constraints and features exposed by the Register-level representation layer.

The input to the quantum compiler is a quantum circuit including single or multi-qubit gates. Usually, the input circuit is the simplest (and most elegant) representation of a quantum algorithm (e.g., the Quantum Fourier Transform). Such a representation does not consider the constraints that may characterise the target quantum computer, such as the available gate set and the connectivity constraints between which qubits a two-qubit gate is natively allowed.

The quantum compiler leverages information provided by the Register-level representation layer to translate the input circuit into an equivalent circuit that fits the target device.

An example is provided in figure 8.1, in which a quantum circuit is compiled into another quantum circuit by considering the connectivity constraints of the target quantum computer.

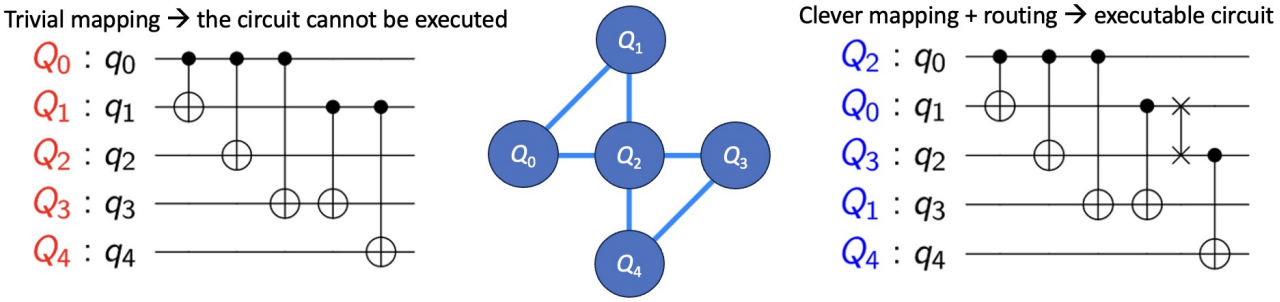

Figure 8.1 *– The circuit on the left does not fit the connectivity constraints of the target device, which are described by the graph in the middle of the figure. The circuit on the right is the compiled version of the circuit on the left, i.e., functionally equivalent but fitting the target device. To produce the output circuit, the compiler chose a different mapping for the input circuit's qubits to the device qubits and inserted a SWAP gate before the last CNOT gate.*

The description format of the output circuit may be different from the description format of the input circuit. If the input and output circuits have the same description format, the compiler is often denoted as "transpiler".

9 Service Layer

This layer contains the user-side where a task, or subset of a task, exists that requires execution. Quantum computers can help executing this task and the user can then start programming algorithms to obtain the sought for answer. Depending on the used service, users may perform tasks locally on a quantum computer. An alternative is that tasks run mainly remotely on a classical computer and use quantum computing as a service (QCaaS) to run specific tasks on a dedicated quantum computer.

10 Communication Module

Currently, commercial quantum computers are built for cloud-based computing, or at least offer access to different end-users. This means that users wanting to execute algorithms on a gate-based quantum computer from outside the stack must place a request to get access to one or more (software) layers inside the stack. For this purpose, it is crucial that each software layer can be reached by the communication module.

The communication module can exchange messages with client applications that run outside the quantum stack, for instance on a nearby computer or on a remote server somewhere in the cloud. It can handle all messages that are needed for starting a quantum computing session (for instance handshaking, authentication, resource allocation, billing, rights-management, etc.). A quantum computing session offers an application the experience as if it has its own resources and as if it is fully protected from other applications. Figure 4.1 shows a few of these functionalities the communication module may possess.

Once a session is initiated, the communication module can start handling incoming messages for instructing the upper layers in the stack. For instance, to load and run a quantum assembly task. Results can be passed back to the communication module, which in turn can send messages with those results to the client application outside the quantum stack.

The communication module can also communicate directly with the lower layers of the quantum stack, provided that the client application is allowed to according to allocated usage rights. For instance, to send low-level commands directly to the control electronics for firing a specific pulse to a qubit. And again, detected results from the control electronics can also be passed back to the communication layer, which in turn can send messages with those results to the client application outside the quantum stack.

Bibliography

[1] IEEE/Open Group 1003.1-2017, Standard for Information Technology - Portable Operating *System Interface (POSIX(TM)) Base Specifications, Issue 7.*

[2] Nielsen & Chuang, Quantum Computation and Quantum Information, Cambridge University Press 2000

[3] OpenQASM, https://en.wikipedia.org/wiki/OpenQASM

[4] Q# or Q-sharp, https://en.wikipedia.org/wiki/Q_Sharp

[5] Silq, https://en.wikipedia.org/wiki/Quantum_programming#Silq

[6] QPL

[7] Quiper, https://en.wikipedia.org/wiki/Quantum_programming#Quipper

Editor's note: Please help identifying a meaningful reference to the above assembly/programming languages. They are referred to in section 8.1. If no adequate reference has been identified, the alternative is to remove the associated example in section 8.1