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

Convenor: **Paul Alexandra Mme**

CEN-CLC-JTC 22-WG 3_LayerModel_Draft08

Document type	Related content	Document date	Expected action
Project / Draft	Meeting: VIRTUAL 19 Feb 2025	2025-02-19	**COMMENT/REPLY** by 2025-02-26

Description

Dear members,

Please find attached the V.8 of the WI on the Layer Model.

Kind regards,

CEN/TC XXX

Date: 20YY-XX

prEN XXXXX:20YY

Secretariat: XXX

JTC 22 WG3 Quantum Computing

Layer Model for quantum computers

Draft 08, 2025-02-18

CCMC will prepare and attach the official title page.

Contents

Page

European foreword 4
Introduction 5
1 Scope 6
2 Normative references 6
3 Terms and definitions 7
4 Abbreviations 8
5 Overview 9
6 Low level Hardware and Control layers 11
6.1 Cryogenic Solid State 11
6.2 Room Temperature Solid State 13
6.3 Trapped Ions 13
6.4 Neutral Atoms 14
6.5 Photonic quantum computing 14
6.6 Other Architectures 14
7 Hardware Abstraction Layer (HAL) 15
7.1 Organization of qubits 15
7.2 The concept of native gates 15
7.3 Concept of primitive gates 17
7.4 Concept of measurement 17
7.5 Interfacing considerations 17
8 Assembly layer 17
9 Programming layer 19
9.1 Programming Languages and Libraries 19
9.2 Quantum Compilation 19
10 Service Layer 20
11 Communication Unit 21
11.1 Example Information Flow 21
Bibliography 23

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 model for gate-based quantum computing, in scope[1] of this Technical Report, slices down the overall complexity of quantum computing into two main groups of layers, addressing this quantum system. The group of lower layers addresses mainly hardware, and is dependent of the physical platform. The group of upper layers addresses mostly software at a higher level of abstraction.

The group of lower (hardware) layers 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 group of upper layers 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 micro-controller design.

One purpose of this document is to define a common language that can 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 ought to offer sufficient guidance on where a desired functionality is to 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 demand for careful analysis of the interaction between the layers.

[1] This limitation keeps technologies like the universal adiabatic quantum-computing model, the universal photonic one-way quantum computing model and its heuristic form quantum annealing, as out of scope if they do not correspond to a gate-based quantum circuit.

1 Scope

This document defines 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 Technical Report is restricted 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 others. This document does not apply to technologies like the universal adiabatic quantum-computing model and its heuristic form quantum annealing, if they do not correspond to a gate-based quantum circuit. Due to major architecture differences in lower layers, it does not apply either to the universal photonic one-way quantum computing model even though it is fully compatible with gate-based quantum-computing model. Moreover, quantum computing models that are not universal, such as quantum simulators and special purposes, are also 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 service layer. These commonalities imply a market for software products usable for this wide range of quantum computing technologies.

The present Technical Report is focussed on a high-level (functional) description of the layers involved. Additional details of the individual layers are reserved for 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

Codesign

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.

Note 1: This approach allows for hardware-specific optimizations and adaptations to optimize quantum computations.

3.2

Gate-based quantum computing

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.

Note 1: 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.

3.3

ISA (Instruction Set Architecture)

a lower-level method of defining operations on a quantum computer.

Note 1: Instead of defining specific gates, this method defines gates (or other instructions) as operations, using pulses pulsed for a certain time, on specific qubits.

3.4

Universal gate-based quantum computing

a quantum computer being capable of processing an arbitrary quantum circuit.

Note 1: A universal gate-based quantum computer ought to 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.

4 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

5 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 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 and software layers, each having dedicated functionalities. A communication unit connects the stack with the outside world to prevent unauthorized access to the stack. They are described in succeeding chapters. The legend describing each colour can be seen in Figure 1. Each layer aims to be more agnostic to the exact implementation of lower layers.

A module is within the context of this TR something that can be sold and shipped independently from other modules. It can offer the functionality of a single layer, of multiple layers or just a fragment of a layer. In all cases, they require interfaces to let them interwork with other modules. The boundaries between the layers are natural locations for defining standardised interfaces between layers, so modules can take advantage of that. But when the functionality of a module span two or more layers, there is no need to implement the interfaces between the inner layers.

A module may also support different operating modes, such that it complies with different requirements of multiple members and/or multiple architecture families.

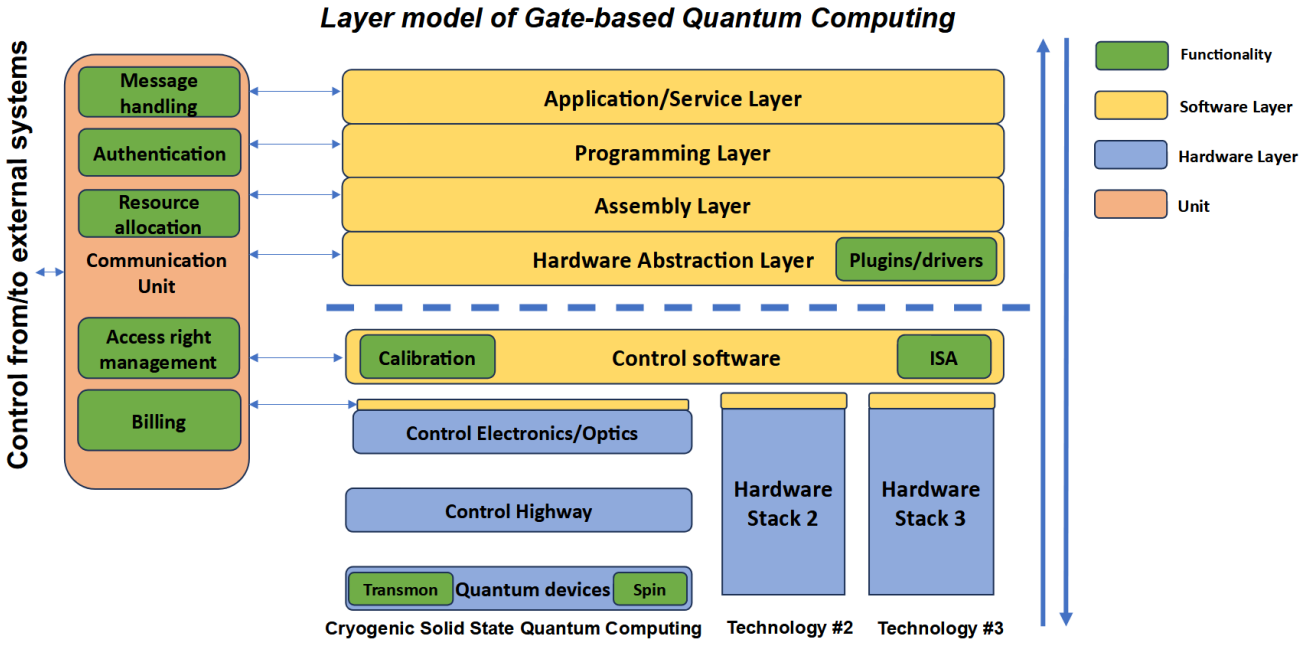

Figure 1 *- Overview of the layer model of quantum computing.*

Figure 1 shows an overview of the proposed layer model. In principle, each layer interacts only with the one below and above it, but it is not excluded that interaction bypasses a layer to interact directly with one deeper or higher. The communication unit can exchange information directly with each layer. The dashed line separates the group of higher layers from the group of lower layers.

A one-size fits all approach may not apply to all these different architectures, and therefore each one may have its own stack. The use of four lower layers have shown to be adequate for serving the needs of cryogenic solid-state based technologies. Other architectures, depicted in this figure as technology #2

and #3, may need another composition of lower layers. Therefore their stack has been drawn as a single box, and their details are left for further study.

It is possible that the layer above the dashed line must account for different interfaces below the line that are dedicated for each hardware stack. The aim of the hardware abstraction layer is to offer a more harmonized and common interface to higher software layers.

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;
- Molecular spins;
- Other architectures that may be identified in the future

These architectures are described in further detail in succeeding chapters.

Within an architecture family, multiple members may exist, like transmons and spin-qubits for cryogenic solid state QC. Small differences in functionalities of the lower layers may therefore occur as well.

6 Low level Hardware and Control layers

6.1 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 needed 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.

6.1.1 Layer 1 – Quantum Devices

The quantum devices in the bottom hardware layer function as modules containing qubits, typically operating at cryogenic temperatures and implemented either as chips or on a PCB. Their quantum states can be manipulated and read out by sending pulses and measuring their response. These devices may also have strict requirements regarding shielding, operating temperature, magnetic conditions, and other environmental factors.

6.1.2 Layer 2 – Control Highway

The control highway covers all hardware needed for transporting microwave, lightwave, 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.

Downstream *signals* require attenuation at cryogenic temperatures to keep most of the thermal noise away from the qubits. Overall loss values of 50 dB or more are not uncommon. Additional filtering up to IR frequencies can reduce unwanted out-of-band noise even further. Since attenuators heat up by dissipating attenuated signals, they produce more thermal noise than desired. Thermalization is therefore required to keep attenuators cool and to drain away most of the heat flow from room temperature nodes through the transmission lines. Superconducting sections can offer additional thermal isolation to prevent that qubits heat up.

Upstream *signals* require low-noise amplification, making it essential to minimize signal loss between the qubit and the first amplifier. When TWPAs (Traveling-Wave Parametric Amplifier) are used as the first amplification stage, the control highway should transport pump signals as well.

As the number of qubits in a single quantum computer grows rapidly, managing an extensive number of channels within a single cryostat becomes increasingly complex. The size of a control highway can easily become very bulky, making it more challenging to keep crosstalk under certain thresholds. Outgassing is also an issue that must be kept minimal since the control highway has to operate under demanding vacuum conditions. Moreover, it should be designed such that vibrations in the cryostat do not induce unwanted signals into the qubits. Therefore, a control highway is more than just a collection of cables; it is a carefully designed subsystem essential for efficient operation. A convenient

implementation of a control highway is a module offered as a top or side loader for insertion into a cryostat, having all thermalization on board.

6.1.3 Layer 3 – Control Electronics

Hardware layer 3 covers all room temperature electronics for generating, receiving, and processing microwave, RF, and DC *signals*. Some implementations make use of routing/switching and/or multiplexing of control *signals*. It receives *commands* from higher layers to fire baseband and modulated pulses, generate pump signals, and to perform a measurement of qubit responses.

If these *commands* are standardized, the control electronics can easily be replaced by similar electronics from other brands. This capability usually requires a simple translation of standardized commands into proprietary hardware commands for storing samples in the memory of an AWG (Arbitrary Waveform Generator) or firing a selected pulse.

Figure 1 illustrates that this translation can be accomplished through a thin software wrapper layered directly above the hardware, serving as an integral component of layer 3. It can be offered as firmware built into the electronics or as an external piece of (driver) software.

6.1.4 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. It plays a crucial role in translating higher-level quantum assembly *instructions* into *commands* that can be handled by the control electronics. This layer may include an instruction set architecture (ISA), error correction and calibration functionalities (as shown in Figure 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.

6.1.4.1 Functionality of an ISA

The aim of an instruction set architecture (ISA) is to convert a sequence of machine-specific instructions from higher layers into commands for the control electronics to control individual qubits. As such, the ISA has full awareness of the underlying quantum hardware and its topology. Due to the ISA's knowledge of the quantum hardware, it also has the responsibility of handling the execution timings and scheduling of individual instructions, such that higher layers should only know their sequence.

On input, the ISA may receive instructions from higher layers, for instance, to change the quantum state of qubits (gate instructions), to read out qubits (measurement instructions) or any other instructions to

interact with all available qubits. These instructions can be provided to the ISA in a specific format, such as binary machine instructions, ASCII human-readable instructions, or function calls. Instructions intended for controlling one or two qubits may be fed one by one to the ISA, but it is more efficient if an ISA can handle many of them in parallel as a "vector" of instructions to interact with an ensemble of qubits simultaneously.

Higher layers can either push these instructions into a buffer within the ISA whenever the ISA signals readiness, or the ISA can poll instructions from a buffer within higher layers after completing the execution of a previous instruction group. This process also includes polling requests and instructions from multiple users. In all cases, it requires a well-defined interface (API) with the above layer(s), as well as a well-defined instruction set language (such as OpenPulse [7] or Pulser [8]).

An ISA may handle gate-level instructions as well as pulse-level instructions. Both may be mixed in a single compilation pass for bypassing specific gate instructions, which can be parsed in the Software Development Kit (SDK) by the user. Gate-level instructions are considered to be any set of operations that can be parsed onto universal gate-based quant`um computers regardless of the hardware, while pulse-level instructions are operations that are heavily dependent on the system's physical architecture. The ISA will thus support instructions to specify the exact waveform of a pulse to be fired to a specific qubit, as well as an ensemble of pulses where each pulse has its own waveform and relative delay.

The common way of sending instructions to the ISA is via higher level layers such as the assembly or programming layer. Alternatively, a user may be allowed, via the communication init, access to the control software layer directly or via the hardware abstraction layer by supplying ISA-readable instructions directly.

On output, the ISA issues commands to the control hardware, such as triggering pulses to qubits or reading their responses through measurement. This requires the ISA to have full control over the timing of all these commands. If a pulse is to be applied to a qubit, the ISA may calculate its characteristics in real-time, such as waveform / pulse-shape and magnitude. Alternatively, it may also read predefined characteristics from a library created by other software units, stored either in the control software layer or in the control electronics layer. In all cases, it requires a well-defined interface (API) with the control electronics as well as a well-defined command set.

6.2 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.

6.3 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 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.

6.4 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.

6.5 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 manipulate directly qubits and execute quantum circuits but provides more specialized computing primitives.

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 using post-selection schema;
- Measurement based quantum computing using cluster states.

The description of these architectures and associated low-level layers is to be developed in future.

6.6 Other Architectures

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

7 Hardware Abstraction Layer (HAL)

The aim of the Hardware Abstraction Layer for 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.

Moreover, the HAL facilitates task scheduling by creating a queue for tasks submitted concurrently by multiple users. Furthermore, the scheduling can be optimized for time or resource efficiency.

Another functionality is that the HAL can select between multiple quantum architectures and even partition a single task into multiple queues to perform calculations in parallel on multiple architectures.

7.1 Organization of qubits

The HAL can report to higher layers how qubits are organised in one or more quantum registers. This is a system comprising multiple qubits, each with its own index. All qubits can be members of a single register, or be spread out over multiple (smaller) registers. The HAL supports instructions to operate on such registers for initializing, changing, and querying the state of the qubits. The HAL can report the properties of each register by means of the following parameters:

- *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.

7.2 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 2 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 2 *- Example of a specific hardware implementation*

7.3 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 ought to 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.

7.4 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.

7.5 Interfacing considerations

A preferred way of communicating with the HAL is by means of binary instructions, preferably common for all quantum computing implementations. Therefore, a list of binary commands is needed 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.

8 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.

9 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.

9.1 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, one being Quipper [6].

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.

The most advanced SDKs support device architectures from multiple providers, provided that the quantum computational model is the same. Examples are Qiskit [9] and Cirq [10], concerning the quantum circuit model.

9.2 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 3, in which a quantum circuit is compiled into another quantum circuit by considering the connectivity constraints of the target quantum computer. 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”.

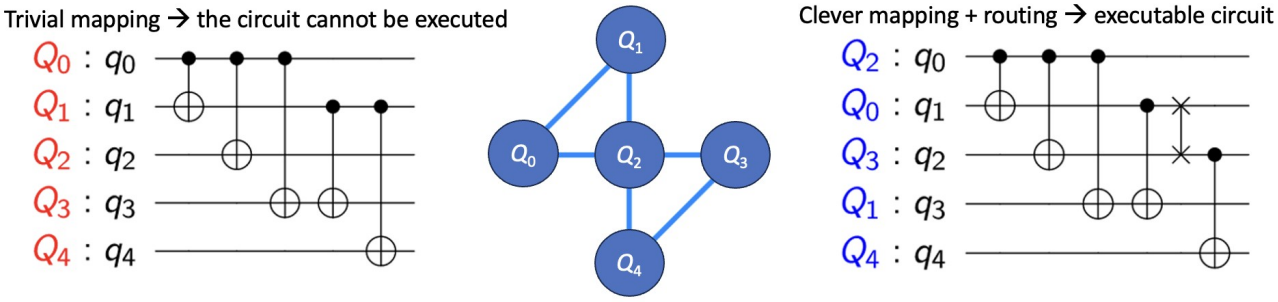

Figure 3 *– Example of compiling a generic circuit into an executable circuit*

10 Service Layer

This layer contains the user-side where a task, or subset of a task, exists that requires execution. Quantum computers can help execute this task and the user can then start programming algorithms to obtain the sought-after answer. Depending on the service used, 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.

11 Communication Unit

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 unit. The communication unit 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, including:

- Handshaking - a protocol to start communicating between remote nodes
- Message handling - means to exchange information between two nodes
- Authentication – to verify if a user is allowed to get access,
- Access right management - to what layers does this user have access
- Resource allocation - reserve memory, time slots, priorities etc.
- Billing - counting how much resources have been used

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.

Once a session is initiated, the communication unit 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 unit, which in turn can send messages with those results to the client application outside the quantum stack.

The communication unit 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. 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.

11.1 Example Information Flow

11.1.1 Single user accessing the full quantum stack

Assume a session has been initiated by an external user or client application. The communication unit will then start handling incoming messages and communicates them, for instance, with the service layer to load and run a task. The associated program will be compiled or interpreted into a sequence of instructions that are then sent to the HAL. Most of these instructions are passed over with minimal conversion effort to the ISA within the control software layer. The ISA generates in turn low-level commands to the control electronics to let it generate a variety of (analogue) signals, such as baseband or modulated pulses. These signals are transported through the control highway to reach the qubits.

Once an instruction is executed for measuring a qubit response, the measured result is reported back to the calling program in higher layers. The results from this program are communicated back to the communication unit, which in turn sends them as messages to the external user or client application.

11.1.2 Multiple users accessing the full quantum stack

Assume multiple users with similar privileges have initialized their sessions simultaneously and each of them has access to the stack. The communication unit is responsible for processing incoming messages, verifying access permissions, checking user privileges, allocating resources for each session and keeping all sessions separated from each other. The users submit their respective tasks and the communication unit may ask a compiler to compile each task to store the results into an allocated thread within the HAL. The HAL offers scheduling and priority of each thread based on user specific information flagged by the communication unit. The scheduled instructions in the output queue are sequentially fed to the control software layer for further processing as previously described.

Once the circuit is executed, the measured result is communicated back to the HAL, which in turn communicates this to the calling program.

11.1.3 User accessing lower layer

Assume a super-user with adequate access rights wants to upgrade low-level software and submit it directly to a lower layer. For instance an upgrade of the ISA functionality within the control software layer. The super-user sends this upgraded software to the communication unit, including information about the target layer. The communication unit sends “wait” requests to all layers, pauses until full confirmation, upgrades the low-level software, sends “success” or “failure” messages back to the super-user, and reports “done” to all layers. After that, the full stack can proceed as usual.

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 - Cross, A., Javadi-Abhari, A., Alexander, T., De Beaudrap, N., Bishop, L. S., Heidel, S., … Johnson, B. R. (2022). OpenQASM 3: A Broader and Deeper Quantum Assembly Language. ACM Transactions on Quantum Computing, 3(3), 1–50. doi:10.1145/3505636

[4] Q# or Q-sharp - Tapia, H., Bravo, S. L., & Ben, B. (n.d.). Introduction to the quantum programming language Q# - azure quantum. Introduction to the Quantum Programming Language Q# - Azure Quantum | Microsoft Learn. https://learn.microsoft.com/en-us/azure/quantum/qsharp-overview

[5] Silq - Bezganovic, V., Lewis, M., Soudjani, S., & Zuliani, P. (2024). High-level quantum algorithm programming using Silq. arXiv [Quant-Ph]. Retrieved from http://arxiv.org/abs/2409.10231

[6] Quiper - Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., & Valiron, B. (2013). Quipper: a scalable quantum programming language. ACM SIGPLAN Notices, 48(6), 333–342. doi:10.1145/2499370.2462177

[7] Open Pulse - McKay, D.C., Alexander, T., Bello, L., Biercuk, M.J., Bishop, L., Chen, J., Chow, J.M., C´orcoles, A.D., Egger, D., Filipp, S., Gomez, J., Hush, M., Javadi-Abhari, A., Moreda, D., Nation, P., Paulovicks, B., Winston, E., Wood, C.J., Wootton, J., Gambetta, J.M.: Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments (2018). https://arxiv.org/abs/1809.03452

[8] Pulser - Silverio, H., Grijalva, S., Dalyac, C., Leclerc, L., Karalekas, P.J., Shammah, N., Beji, M., Henry, L.-P., Henriet, L.: Pulser: An open-source package for the design of pulse sequences in programmable neutral-atom arrays. Quantum 6, 629 (2022) https://doi.org/10.22331/q-2022-01-24-62

[9] Qiskit - Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum computing with Qiskit (2024). https://arxiv.org/abs/2405.08810

[10] Cirq - Developers, C.: Cirq. Zenodo, (2024). https://doi.org/10.5281/ZENODO, https://zenodo.org/doi/10.5281/zenodo.4062499