The Journal of Supercomputing (2025) 81:442
https://doi.org/10.1007/s11227-025-06955-y

Inqasm: InQuIR compiler to NetQASM

Jorge Vázquez-Pérez[1] · F. Javier Cardama[1] · César Piñeiro[3] · Juan C. Pichel[1,3] · Tomás F. Pena[1,3] · Andrés Gómez[2]

Accepted: 14 January 2025
© The Author(s) 2025

Abstract
Quantum computing is a rapidly evolving field, with almost every aspect open to change or improvement. This includes moving from using a single quantum processing unit to interconnecting multiple quantum processing units (or several of them), establishing a new paradigm called distributed quantum computing and increasing the overall computing capability. Some research is already underway in this area to prepare the ground for an eventual architecture with these characteristics. This is the case of InQuIR (Nishio and Wakizaka in arXiv:2302.00267 2023) and NetQASM (Dahlberg et al in QST 7:035023 2022), two languages developed for distributed quantum computing. This paper presents the development of the InQASM compiler with the aim of translating code from the InQuIR language to NetQASM, establishing a compilation stack for the new distributed paradigm. An example of this compilation and a simulation of the compiled code are shown to showcase it.

Keywords NetQASM · InQuIR · Compiler · Distributed · Quantum · QPU · DQC

1 Introduction

Quantum computing and, more specifically, distributed quantum computing (distributed quantum computing (DQC)) [1] brings many problems and difficulties to the software stack. From how to express circuits at a high level of abstraction to how to optimise circuits for a particular architecture or technology, it all remains an area of research with few agreed standards and practices.

✉ Jorge Vázquez-Pérez
jorgevazquez.perez@usc.es

[1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain

[2] Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain

[3] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, Santiago de Compostela, Spain

Published online: 28 January 2025

In the field of quantum computing, discussions often focus on algorithms and their potential advantages in terms of time efficiency, particularly in solving problems that remain intractable to classical computation. One algorithm frequently cited in this context is Shor's algorithm [2], famous for its impact on modern cryptography. However, building the necessary infrastructure to run these algorithms is a substantial undertaking. Questions of input and output of information, storage of results, monitoring of execution, and other logistical aspects are far from trivial. The straightforward solution to these questions lies in classical computing. However, implementing this solution is far from simple.

The apparent paradox of solving quantum software challenges through classical computation is not a contradiction. Classical technologies such as memory and I/O communication dwarf their quantum counterparts because they do not have to comply with the constraints of the no-cloning theorem and can hold information indefinitely, unlike quantum systems, which are prone to decoherence [3]. Although some quantum memory models exist, they are not yet ready to replace classical memory [4–6]. In addition, all quantum computers are controlled and monitored by classical computation. For example, it is common to see field-programmable gate arrays (FPGAs) managing the sending of pulses to the qubits to execute a particular gate [7, 8]. It is, therefore, understandable that quantum computing software is predominantly classical.

Thus, it is interesting to integrate quantum computers into the high-performance computing (HPC) environment and use them to accelerate specific tasks, similar to accelerators such as FPGAs or graphics processing units (GPUs). Although they fundamentally differ from classical accelerators in that they introduce a new computing paradigm that complements the existing heterogeneous HPC landscape [9, 10].

Focusing on the compilation process for quantum systems, it is important to recognise that quantum software, being inherently classical, requires the construction of a classical software stack [11].

While adapting the conventional classical software stack to quantum computing presents significant challenges due to the lack of abstraction available in this domain, it still serves as the most viable option [11]. In this context, the quantum computing *software stack* can be viewed as comprising several layers, analogous to those found in classical computing systems [12, 13].

- The *frontend* [14]: This layer parses high-level quantum programming languages such as Qiskit or Cirq [15, 16]. It performs syntax and semantic analysis to check for errors and generates a IR that simplifies the code structure for further processing.
- The *IR*: An abstract, platform-independent code that bridges high-level languages and low-level machine code. It allows for various optimisations and hardware-independent analysis, facilitating portability and adaptability across different quantum hardware platforms.
- The *backend* [17]: This layer is the result of translating the IR into hardware-specific instructions or machine code. It includes optimisations tailored to the

architecture and constraints of the target quantum hardware to ensure efficient execution of the quantum program.

Additionally, *simulators* and *emulators* replicate the behaviour of quantum hardware and provide a virtual environment for testing, debugging, and validating quantum algorithms [18]. They allow researchers and developers to experiment with quantum programs without the need for physical quantum processors, making them essential for early-stage development and learning.

Transversal to the three mentioned layers is the *compiler*. The compiler is the piece of software responsible for translating the code of each stage to the following (frontend software to IR and IR to backend code) so that it can run in an emulator or an actual quantum computer.

Regarding quantum computing, some aspects of classical compilation are compatible with quantum, while others are not. Thus, both the software stacks for quantum and classical computing have to deal with different types of hardware. The quantum hardware differs in the kind of technology—trapped ions, superconductors, photons, etc.—and in the set of gates supported [19, 20]. This means a quantum circuit has to be translated into an equivalent one using the supported gates before execution in a particular device, just as in classical computing with different instruction set architectures. In this sense, having these two different levels of abstraction is advantageous because, as in the classical counterpart, the compiler designer only has to worry about translating the IR into the gate set supported by the machine in question.

On the other hand, unlike classical models of computation, quantum computing cannot provide a level of abstraction equivalent to the classical one. In the classical realm, layers of abstraction separate code written in a high-level language from assembly language, allowing the programmer to not worry about bit-level operations. In contrast, there are not many abstractions in today's quantum computing, and there is always a need to work at the qubit level. This means that it is difficult to raise the level of abstraction of IR above that of quantum assembly languages in any appreciable way, so it is difficult to distinguish between a quantum IR and a quantum assembly language. A discussion about these issues will be performed in later sections.

In classical computing, Moore's law has enabled a consistent increase in CPU power simply by adding more transistors to the chip. This growth has driven significant performance improvements. However, as transistors become thinner, it is more challenging to limit power consumption and heat generation [21]. As a result, instead of continuously increasing the transistor count on a single core, the industry has shifted towards multicore systems, where multiple processing units work in parallel to enhance performance without relying solely on transistor scaling.

Quantum computing is expected to see a similar phenomenon in the near future [22]. Due to problems such as crosstalk [23–25], single quantum chip architectures are believed to be limited in the number of qubits, with multicore quantum chips being the solution, following a path that could be considered an analogue to the classical one.

Assuming that this is the path quantum computing will follow, two main problems arise. The first is how the work will be distributed, and the second is how the software will be designed to run it. Neither of these problems is trivial, but they are very different in nature. The first one is more related to the actual quantum computation, where you have to guess how to perform the computation in a distributed system without changing the result and, moreover, increasing the efficiency in terms of time and error. On the other hand, the second one is a purely classical problem because, as already mentioned, all quantum software is actually classical. This work will focus entirely on this second part, leaving the distribution of quantum workloads as a topic for future investigation. For more information about the first part and, in general, about the DQC paradigm, an extensive review of state of the art was presented in [1].

This work focuses on developing part of the aforementioned compilation process. In particular, it will focus on the translation from a quantum IR to a backend. Specifically, this work will perform the compilation of InQuIR code [26] to NetQASM code [27]. The project's scope is to provide the first step towards having a complete distributed quantum software stack to run distributed applications in quantum computing similarly to its classical counterpart. It will not be part of the scope of this work to optimise the efficiency of the distributed quantum circuits.

The rest of the paper is organised as follows. Section 2 presents an analysis of related work and background on DQC. After establishing a baseline on state of the art, Sect. 3 describes the InQuIR and NetQASM works in detail in order to justify the decisions taken when building the InQASM compiler. After this, all the necessary elements for building the InQASM compiler are summoned, enabling Sect. 4 to thoroughly describe the intricacies of the compilation and implementation of the software. Next, in Sect. 5, the code produced by the software is proven to work correctly, along with a representative example of DQC. Finally, in Sect. 6, a discussion of various issues presented throughout the work is carried out, along with the conclusions. All the code employed in this paper is open source.[1]

2 Background and related work

As mentioned previously, this work is dedicated to developing a software stack for DQC. The scheme outlined in the Introduction will be followed to perform a structured analysis of the literature. This involves examining the works on the frontend, then on the IR, assessing the backend, and finally providing a brief analysis of emulators. This brief review will focus primarily on the core topics of this paper: the IR and backend layers. It should be noted that a comprehensive review of the state of the art can be found at [1].

[1] It is available in the repository: https://github.com/jorgevazquezperez/InQASM.

2.1 Frontend

Regarding the frontend, not much software has been developed or even designed for the DQC paradigm. This is not surprising since this type of software has always been painfully difficult to model, even in classical distributed computing and especially in parallel computing. Initially, a great deal of effort was put into achieving automatic parallelisation—the so-called holy grail of parallel computing—but this has had only limited success.

Most attention in recent years has been given to compiler support for technologies such as open multi-processing (OpenMP) [28] or message passing interface (MPI) [29], among others. openMP is an API that enables cross-platform shared memory multi-processing in C, C++ and Fortran, facilitating the development of parallel applications. It provides a simple and flexible interface through compiler directives that parallelise sections of code, making it easier for developers to take advantage of multicore processors without extensive code changes. On the other hand, MPI is a standardised and portable message passing system designed to facilitate communication between processes in parallel computing environments. It enables the development of scalable parallel applications by providing a comprehensive set of libraries for inter-process communication across different computing platforms, including clusters and supercomputers. A contrast can now be carried out by comparing these two technologies with the quantum paradigm.

On the one hand, the concept of threads in classical computing—i.e. different tasks sharing a common memory—does not apply to the quantum analogue. This is mainly because memory in quantum computing cannot be understood in the same way as in classical computing due to decoherence. One could make a point about the relationship between threads and superposition, understanding the qubits as the shared memory that superposition uses to perform different paths of the same computation. But this seems like too much effort to mould a classical concept into the quantum world. And, with some confidence, it can be said that a quantum language model that mimics the openMP behaviour is unlikely.

On the other hand, the MPI proposal is much more compatible with quantum computing. This is mainly because, unlike the concept of a thread, the concept of a message—understood as a piece of information sent from one device (Alice) to another (Bob)—does exist in DQC as it has been shown in [1] with the teledata and telegate protocols. In fact, there have already been some approaches towards a quantum message passing interface (QMPI) [30, 31]. It should be emphasised that QMPI is at a very early stage, its implementation is only at a design stage.

This last fact about QMPI shows how premature the current landscape of frontend software for DQC is. And even more shocking than its prematurity is the fact that there is only one—the QMPI—software tool designed specifically for this area. But, to be fair, this problem extends to the whole field of quantum computing, as it is challenging to find abstraction in this computational model. It just stands out in DQC due to its lower profile in the research sphere.

2.2 Intermediate representation (IR)

Moving on to the next level of abstraction, IRs are the next step. Making a similar exercise as before, and to better understand the concept, a comparison between quantum and classical IRs can help to clarify some of its peculiarities.

In classical computing, and more specifically in classical compiling, IRs are used as a kind of "common denominator". Suppose there are n high-level languages and m different machines, each with its own set of instructions. To be able to compile each language on each machine—i.e. translate the high-level language into the set of instructions required—it would be necessary to program nm different compilers: one for each combination of language and machine. What a IR does is to define a general set of instructions into which any language can be compiled and, more importantly, which can be translated into any specific set of instructions depending on the machine used. Thus, only one compiler is needed for each language to get translated into the IR, and analogously, the IR only needs one compiler to be transformed to each set of instructions. This means that the number of required compilers is $n + m$.

Focusing now on DQC, there has been a proposal for a quantum IR for distributed systems, which has already been mentioned: InQuIR [26]. This distributed quantum IR was born out of the lack of a suitable representation as a compilation target for this kind of system. Authors focused on creating a formal semantic that would allow them to properly express problems such as deadlocks, qubit exhaustion, barriers, and entanglement swapping. All concepts are closely related to the distribution of quantum states[2]. This work will be further analysed in the following sections, where the characteristics and key points will be pointed out, as well as some aspects that were found to be improved during the development of this work. To the best of our knowledge, this is the only IR specifically defined for distributed architectures in quantum computing (NetQASM [27], with its basic set of instructions, is said to work as a kind of IR, but this will be discussed later). Of course, many quantum IRs have been proposed for the monolithic.[3] case, but it is beyond the scope of this paper to explain the differences between them.

Before discussing the backends, it is important to note a specific aspect of IRs for distributed systems. It has been emphasised in the definition of a IR that it must be platform-independent. This is true, but in the distributed case, the IR actually needs to know some information about the system: how many processors are available. This is because if no information about this is given at the IR stage, there is no way of specifying communication directives. This, for simplicity and coherence, will not be considered a platform-specific property but a parameter of the IR. In this sense, the platform-specific characteristics will be the same as for the monolithic case, i.e. the gate set supported, the connectivity of the qubits in each QPU, etc.

[2] All this concepts will not be formally defined in this manuscript; please refer to the InQuIR manuscript [26] for details.

[3] In this text when the term "monolithic case" or "monolithic quantum computing" is used, it refers to quantum computations performed with only one quantum processing unit (QPU). In other words, it refers to the opposite of the DQC.

2.3 Backend

Once the previous two layers have been shelled, the backend becomes the protagonist. This is the last compilation stage, and the part is closest to the machine. In classical compilation, this is the moment when the instructions defined in the IR are translated into the set of instructions understood by the machine in order to execute them.

Again, quantum computation lacks a perfect match for this kind of concept, or perhaps there has been a tendency in the literature to call quantum assembly something that does not quite fit the definition of quantum assembly. The example par excellence is the Quantum Assembly Language (QASM) and all its variants: OpenQASM [32], eQASM [33], cQASM [34], etc. As its name suggests, QASM is inherently defined as an assembly for quantum computing. And the problem with defining QASM as such lies in one of its characteristics: it is platform-independent. This reduces its proximity to the machine and perverts its definition as an assembly language. In fact, if someone takes a quantum assembly and makes it platform-independent, it becomes a quantum IR because what has been done is basically to raise a level of abstraction (exactly where the IRs is found).

This is precisely what happens with NetQASM [27], which can be seen as another variant of QASM, but for distributed systems. To be fair, despite its name, NetQASM is not defined as an assembly language. In fact, it is defined as a *low-level assembler-like language*. But NetQASM solves this problem of platform independence by adding the concept of *flavours*. These are modifications of the instruction set to extract platform-specific advantages. The basic flavour, the universal gate set, is called the vanilla flavour. They even say in this paper that: “the vanilla flavour can be seen as an IR and the translation to a specific flavour as a backend compilation step.”

The lack of standards and guidelines was mentioned previously as part of the motivation for this work. This is the perfect example. Most of the time, quantum compilation processes are called by names that refer to classical stages but do not quite match their nature. Quantum assembly languages are not the only example. For example, the term “transpile” is commonly used in classical computing to translate one language into another at the same level of abstraction. In quantum computing, however, it is commonly used to refer to the translation of a circuit into the gate set of a machine, plus optimisations. Ironically, this is exactly what the term “compile” is used for in classical compilation, as explained earlier in this section. In this sense, the work developed in this paper—i.e. the translation from InQuIR to NetQASM—can be considered as part of the compilation process or as transpilation, depending on whether NetQASM is considered to be at a lower level of abstraction than InQuIR or not. In the conclusions, there will be a discussion treating this topic, taking into account the different characteristics extracted through the work of the two languages.

2.4 Emulators

In this era of quantum computing, the so-called noisy intermediate-scale quantum (NISQ) era, emulators of the behaviour of a quantum computer are of particular interest. Developing strong emulators will allow us to predict better how quantum computers will perform with a given algorithm. They allow the development of new algorithms and techniques while quantum computers are still in the process of becoming viable in a hardware sense.

For DQC, contrary to the monolithic case, there are not so many emulators, but the number is significant compared to the small amount of software that the compilation stack has. There is, of course, an explanation for this. Although the concept of DQC as an analogue to the multicore concept in classical computation is not widely used in practice, quantum networks are. This is why there are emulators for DQC: they were initially designed to simulate a quantum network, not a multicore architecture. Fortunately, the multicore architecture can be treated as a quantum network with special properties, e.g. a small distance between nodes.

A first example of a quantum emulator for distributed computation is the distributed quantum computing simulator (DQCS) [35], a recently developed tool for simulating the behaviour of distributed quantum systems. This emulator is built on top of Qiskit [16] for illustrative purposes rather than efficient simulation. A similar example is the QuNetSim emulator [36], a Python software framework that attempts to provide an easy-to-use platform for testing quantum network protocols. This goal was shared with another emulator called SimulaQron [37]. The SQUANCH emulator [38] is a Python software framework like QuNetSim and has the special feature of modelling noisy quantum channels.

On the other hand, [39] propose a framework design that exploits computational and networking aspects by introducing the concept of an execution manager: a scheduler for networked computers. NetQuil [40] is another example that integrates with the Quil language used by Rigetti's quantum processors, providing a seamless way to simulate quantum networks and distributed algorithms using a common programming language. A more interesting example for the purposes of this work is Interlin-q [41], a simulation platform designed from the outset with the idea of multicore quantum computers in mind rather than simulating quantum networks such as the quantum internet.

There are also several discrete event quantum emulators: QuISP [42], qkdX [43], SeQUeNCe [44] or NetSquid [45]. Discrete event simulation, a well-established method for simulating classical network systems, is a modelling paradigm that advances time by moving through a sequence of events, which is applied to quantum networks in the above emulators. The specific differences between these programs are explained in [45] and will not be described in this work.

Among all these emulators, NetSquid has been selected for this work mainly because it accepts NetQASM instructions as a valid set. This also happens in SimulaQron, but NetSquid was ultimately preferred due to its deeper design and scope. This simulator is designed for quantum networks and, therefore, does not have a multicore structure-centred scope like Interlin-q. However, as mentioned above, multicore architectures can be treated as quantum networks with specific characteristics.

3 InQuIR and NetQASM

This section starts with a description of InQuIR. The aspects and intricacies of both the manuscript and the actual code will be highlighted, pointing out what had to be changed to make the compiler work. Secondly, the same effort is made with NetQASM. Although NetQASM is deeper than InQuIR in content (because they developed a whole framework along with the assembly-like language), only the aspects that affect this work will be explored, along with a general explanation of the software and the work.

3.1 InQuIR

Delving into the paper proposing InQuIR [26], the authors explain that their primary motivation was the lack of IR for distributed quantum systems. In fact, they noticed the effort of some works in trying to map the qubits in an optimal way to the different distributed architectures. Still, they called themselves "compilers" only by doing the optimisation stage of the compilation process, which is just an operation on the backend, not the whole compiler.

So, with this motivation in mind, they claimed the following contributions:

- *Definition of formal semantics.*
- *Examples of use.*
- *Resource estimation software tool.*
- *Roadmap for the introduction of static analysis.*

Formal semantics in programming languages is the precise mathematical study of what programs mean. It provides clear, unambiguous definitions of how programs behave and run. This helps in designing languages, verifying the correctness of programs, building reliable compilers, and improving our understanding of programming concepts. Different approaches, such as operational, denotational and axiomatic semantics, provide different ways of describing and reasoning about the meaning of programs. In this case, InQuIR chooses operational semantics to be able to explain how the system changes during the execution of the program. In fact, they defined the runtime state R as

$$R = [\rho, Q, E, P, H],$$

where ρ is the density matrix of the state (they do not specify, but it can be inferred that it is of the whole system), Q is the set of data qubits, E is the set of communication qubits, P is the system currently being evaluated, and H is the heap for classical communication. With this definition, the operational semantics can be described by analysing how a runtime state R changes into another R'.

Speaking of *examples to use*, this is probably the most innovative feature presented in the InQuIR work. Almost all the work around quantum distribution focuses on optimising the mapping of the qubits and minimising classical communication

between nodes. But none of them focuses on the problem that can cause a deadlock.[4] With the tools used to construct the operational semantics, they present four important examples: entanglement swapping, barriers, qubit exhaustion and deadlocks. As of this work, these four examples will not be employed, but they are a significant aspect of InQuIR.

About the *resource estimation software tool*, they introduce several metrics: E-count and E-depth, C-count and C-depth, estimated time and number of remaining operations by processors at each time. With E-count and E-depth, they analyse the number of generated entanglement pairs and the depth of the critical path, respectively, considering only the dependencies of the entanglement generation. C-count and C-depth, on the other hand, focus on classical communication. They used theoretical values to calculate the time cost of each operation, so the present estimate serves only as an indication of performance. Still, it could be improved by adapting the time cost of each operation to that performed by a real machine. However, this should be done when compiling InQuIR for the specific machine and not before because, as clearly stated in this manuscript, the IRs must be platform-independent.

Finally, the *roadmap for introducing static analysis* is just a brief explanation of how static analysis could help improve the verification of quantum programs. Static analysis of a quantum program involves examining its structure and properties without running it to detect problems and verify correctness. This is an increasingly common practice in monolithic quantum computing, where it is increasingly seen as an alternative to classical debugging. This is because it is impossible to read quantum states without affecting the actual state, so the only way to verify that the program is correct is to check certain aspects of the circuit before execution.

3.1.1 Benefits and drawbacks

It is now necessary to carry out a corresponding analysis to understand the choices made in the implementation of our compiler. Starting with the strengths of this work, it is worth noting the establishment of a syntax and operational semantics. Doing this to theoretically explain and support the creation of a language—in fact of an IR—is a good practice to reduce inconsistencies and errors. The introduction of a resource usage estimator is also worth mentioning. From a compiler point of view, this could be used to improve the IR code, a common practice in classical compilation when dealing with IRs.

Regarding the drawbacks, it is necessary to talk about the InQuIR software available in the repository.[5] Along with the language, InQuIR developers have included a toy compiler that, given a QASM code and the connectivity of the QPU, transforms the QASM code into InQuIR code. This toy compiler chooses to do automatic distribution of the quantum workload, which is probably not the best idea considering the problems of autoparallelisation studied in classical computing. But since it is a

[4] A deadlock in programming is a situation where two or more processes or threads are stuck waiting for each other to release resources, causing them all to stop indefinitely.

[5] The repository is located in https://github.com/team-InQuIR/InQuIR.

toy compiler and not a real compiler, it is understandable that it does not look for the most efficient compilation.

The real problem is that this compiler does not follow the defined syntax. The grammar defined in [26] does not match the grammar implemented by the actual compiler, resulting in a completely different language. The actual grammar implemented by the compiler is shown in Fig. 1. There is little interest in comparing the actual grammar with the theoretical one. However, it is interesting to analyse the problems of the actual grammar, some of them being the following.

- Qubits and classical data are intertwined under the same variable, called *value*.
- There is no differentiation between the communication and data qubits.
- The system definition only allows two system concatenated.
- The format employed does not follow any specific formal syntax definition format.

To solve these inconveniences, we have developed a modified version of InQuIR in order to use with our compiler. The inconveniences pointed out were modified with a appropriated justification. It is important to mention that the language allowed by this new syntax will be exactly the same as the one defined by the actual grammar. This is just a formal precision which, although irrelevant from a practical point of view, will make the InQuIR grammar much clearer and, thus, the understanding of this work.

3.2 NetQASM

Next the NetQASM [27] language will be explained. However, it is first necessary to understand how quantum networks are viewed and their execution model. After that, a schematic explanation of the NetQASM language will be presented. In this case, the syntax will not be revealed since it is the target of the compilation and not the source, which means there is no need to read instructions (only to generate correct

Participants $\ni$ p   Sessions $\ni$ s   Labels $\ni$ l   DataQubits $\ni$ q   CommQubits $\ni$ $\overline{q}$

$$
\begin{array}{rl}
\text{(System)} & S ::= [P]_p \mid (S_1 \mid S_2) \\
\text{(Process)} & P ::= s = \mathsf{open}[p_1, \dots, p_n];\ P \mid v = \mathsf{init}();\ P \mid \mathsf{free}\ v;\ P \mid \\
& \quad U(e_1, \dots, e_n)[e]\ v;\ P \mid v = \mathsf{M}\ v_1, \dots, v_n;\ P \mid \\
& \quad v = \mathsf{genEnt}[p](l);\ P \mid (v_1, v_2) = \mathsf{entSwap}(e_1, e_2);\ P \mid \\
& \quad v = \mathsf{qrecv}[p](s, l, e_2);\ P \mid \mathsf{qsend}[p](s, l, e_1, e_2);\ P \mid \\
& \quad \mathsf{recv}(s, l, e_1);\ P \mid \mathsf{send}[p](s, l : e_1);\ P \mid \\
& \quad \mathsf{rcxc}[p](s, l, e_1, e_2);\ P \mid \mathsf{rcxt}[p](s, l, e_1, e_2);\ P \\
\text{(Expression)} & e ::= v \mid e_1 \wedge e_2 \mid e_1 \oplus e_2 \mid \dots \\
\text{(Value)} & v ::= 0 \mid 1 \mid q \mid \overline{q} \mid x \mid \dots \\
\text{(Gate)} & U ::= \mathsf{X} \mid \mathsf{Z} \mid \mathsf{H} \mid \mathsf{T} \mid \mathsf{CX} \mid \dots
\end{array}
$$

Fig. 1 Actual syntax of the InQuIR grammar

ones from the InQuIR code). Finally, some brief remarks about NetQASM will be made to clarify and reinforce some ideas.

The first thing that needs to be explained about NetQASM, even before its language, is the abstract model of the hardware and software architecture. It can be seen in Fig. 2. This model is composed of two main parts, the *application layer* and the *quantum network processing unit (QNPU)*, both classically connected by a shared memory and the sending and receiving of NetQASM code or, as the authors call it, *NetQASM subroutine*. This represents a node in a quantum network, and for this work, it will be considered a QPU.

It has already been said that NetQASM is constructed and considered for quantum networks. Perhaps taking into account the context of this work, i.e. a quantum multicore structure, having an application layer on top of the QNPU might be a bit cumbersome. However, as this is at an early stage and the ultimate goal is to create a link between two software—NetQASM and InQuIR—this is left as a future improvement. Also, the intricacies of the QNPU are left to the reader; for this endeavour, it is sufficient to understand that the NetQASM subroutines are sent to the QNPU to be executed and that the application layer has access to a memory shared with the QNPU. This is illustrated in Sect. 5 with the example employed.

Now, it is necessary to describe the NetQASM language. First, although it can be (and in this work has been) classified as a variant of the QASM language, it is quite different in form and intention. Apart from the obvious fact that, unlike QASM, NetQASM is designed for quantum networks, it is also striking that NetQASM is almost more like a classical assembly language than like QASM. This is mainly due to the virtualisation of the qubits. This is done by an artefact called *unit module*. A unit module defines the topology of the available qubits by specifying which qubits are connected, i.e. on which qubit pairs a two-qubit gate can be executed. It also contains additional information about each qubit. These unit modules contain virtual qubit IDs—similar to classical computers with registers—and, unnoticed by the application layer, the QNPU maintains a mapping of these virtual qubit IDs to the physical qubits. In Fig. 3a, this virtualisation can be seen, for example, in line 4. This line, `set Q0 0`, does not set the qubit number zero to the state $|0\rangle$. What it does is to assign the ID 0 to the variable `Q0` and then, when operations are applied

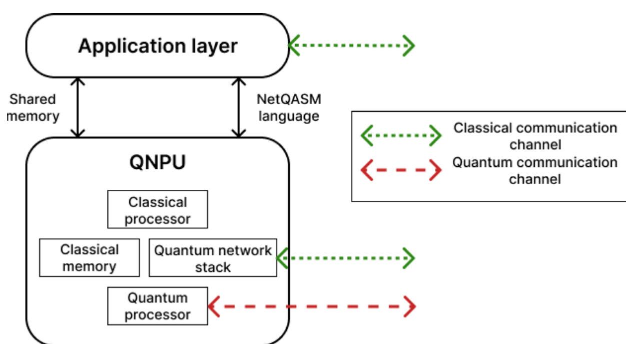


Fig. 2 Abstract model considered in the NetQASM work [27]

```
# NETQASM 1.0
# APPID 0
array 1 @2
set Q0 0
set Q1 1
cnot Q0 Q1
h Q0
meas Q0 M0
qfree Q0
store M0 @2[0]
ret_arr @2
```

(a) NetQASM protosubroutine example

```
# NETQASM 1.0
# APPID 0
set R0 1
array R0 @2
set Q0 0
set Q1 1
cnot Q0 Q1
h Q0
meas Q0 M0
qfree Q0
set R0 0
store M0 @2[R0]
ret_arr @2
```

(b) NetQASM subroutine example

Fig. 3 NetQASM language example in different versions

to this qubit, the QNPU will determine—with its mapping—which physical qubit will be the target of such operations. This common practice in classical assembly language finds its equivalent here, in the DQC field, in NetQASM.

Furthermore, something not even mentioned in [27] but present in the developed code is the distinction between *protosubroutines* and *subroutines*. These represent the human-readable and binary versions of the NetQASM language, respectively. A difference between these two representations is that the protosubroutines have jump variables and constant values, and, on the contrary, the subroutines translate these values to registers. These can be observed in Fig. 3, where all the constant values employed out of a `set` directive are substituted for a register (for instance, `R0` in Fig. 3b).

Finally, the Python software development kit (SDK) developed by the authors to create NetQASM programs with a higher level of abstraction has to be mentioned. This may seem contradictory to what was exposed in Sect. 2, when the lack of frontend languages was mentioned. But it is not. The abstraction this Python SDK provides is not at a quantum level. It is an abstraction at the classical level because there is still the need to work with gates and, therefore, bit operations—been the bits, in this case, qubits. Nevertheless, this SDK is of interest because it will be used to insert the compiled subroutines into the NetSquid simulator and test their correctness. Or, in other words, this Python SDK will represent the application layer, being the one communicating with the simulated QNPUs.

3.2.1 Benefits and drawbacks

In terms of benefits, first of all, virtualising the qubits and abstracting the application layer and the QNPU is a very powerful way of designing the model. It frees the compilation process from dealing with the quantum resources of the QNPU and removes tasks from the quantum part of the node to speed up the processes, perfectly separating the quantum and classical workloads. As far as the software infrastructure is concerned, the amount of tools and options is quite impressive. The easy integration with the NetSquid simulator gives this work a step up in quality, as it is possible to test real use cases to verify the functioning of the theoretical design.

As for the drawbacks of this work, theoretically, there are not too many. The proposed scheme is an interesting way of modelling the problem of generating low-level instructions for a QNPU. From a language point of view, there are no theoretical flaws or, as happened with InQuIR, any mismatch between what is theoretically exposed and what was actually implemented. The main drawback found in this work is the compilation part, where many unnecessary instructions were generated when working with the Python SDK and not corrected when transforming the protosubroutine to the subroutine. This will be pictured in Sect. 5 when executing the compiler.

4 The InQASM compiler

As has already been mentioned, the focus of this endeavour was to develop a tool capable of translating InQuIR files into NetQASM files. There are two main reasons for doing this translation. The most important one is to create a link between these two languages, which allows making a seamless connection between them and, in the process, extracting useful information about quantum IRs for DQC. Secondly, this connection opens up the possibility of testing the InQuIR code in an emulator since, as already mentioned, NetSquid accepts NetQASM as a valid instruction set.

Is in this section where the aforementioned objective of this work is carried to fruition. In this section, a detailed description of the InQuIR to NetQASM compiler will be provided. This starts by explaining the syntax implemented for InQuIR, which defines the grammar, using the extended Backus-Naur form (EBNF) [46]. After this, an explanation about how the translation was performed, i.e. how each instruction was transformed into a NetQASM instruction, will be conducted. Finally, a brief explanation of how the resulting code was tested is given. As an implementation note, the languages used in this endeavour are C++, Python and Java, although Java is used indirectly, as it will be shown.

4.1 Grammar definition

As noted in Sect. 3.1.1, the biggest drawback of InQuIR was the inconsistencies in the grammar. In addition, although it was not mentioned, the form used to express the syntax—usually referred to as a meta-syntax—is not as powerful and rigorous as, for example, the EBNF. In fact, explaining the grammar in this way will allow a seamless implementation of the grammar using ANother Tool for Language Recognition (ANTLR) [47]. Of course, as the name suggests, many language recognition tools exist. ANTLR was chosen because of its excellent integration with C++, which is the core language of the compiler, but other tools could have been used.

In Fig. 4, the implemented grammar is displayed. Some remarks have to be made to justify some decisions made.

- *Use of EBNF*. As already noted, instead of using a self-created style to express the grammar, EBNF was used. This narrows the gap between the theoretical

```
        system ::=   { process }
                 |   "(" system "|" system { "|" system } ")"
                 ;
       process ::=   participant "(" { line } ")" ;
          line ::=   function ";"
                 |   q_inst ";"
                 ;
      function ::=   session "=" "open" "(" participant { "," participant } ")"
                 |   "close" "(" session ")"
                 |   data_qubit "=" "init" "(" ")"
                 |   comm_qubit "=" "genEnt" "[" participant "]" "(" label ")"
                 |   "(" comm_qubit "," comm_qubit ")" "="
                         "entSwap" "(" expression "," expression ")"
                 |   "recv" "(" session "," label ":" value ")"
                 |   "send" "[" participant "]" "(" session "," label ":" clbit ")"
                 |   "RCXC" "[" participant "]" "(" session "," label ","
                         expression "," expression ")"
                 |   "RCXT" "[" participant "]" "(" session "," label ","
                         expression "," expression ")"
                 ;
        q_inst ::=   gate [ "(" real { ( "," real ) } ")" ] qubit [qubit]
                 |   gate [ "(" real { ( "," real ) } ")" ] "[" expression "]" qubit [qubit]
                 ;
    expression::=    clbit
                 |   expression "AND" expression
                 |   expression "OR" expression
                 |   expression "XOR" expression
                 |   "(" expression ")"
                 ;
          gate ::=   "X" | "Y" | "Z" | "H" | "CX" | "I" | "S" | "Rz" ;
         qubit ::=   data_qubit
                 |   comm_qubit
                 ;
         clbit ::=   "0" | "1" ;
    data_qubit ::=   id ;
    comm_qubit ::=   id ;
         label ::=   id ;
          real ::=   digit { digit } ["."] { digit } ;
            id ::=   letter , { letter | digit | "_" } ;
```

Fig. 4 Implemented syntax of InQuIR using EBNF. Note that the digits and letters specification is not done to simplify the figure

explanation of the grammar and the implementation, making the work easier and, even more important, making the understanding of the code much better.

- *Branch 'value' to 'clbit' and 'qubit'.* Something particularly problematic about the InQuIR grammar is the fact that the nonterminal element 'value' included both classical bits and qubits. This does not make sense, as this grammar would permit classical operations to be executed over qubits and vice-versa. Separating classical information and quantum information in the grammar facilitates the following processing and, even better, improves the readability of the grammar.
- *Change the 'system' definition.* In the original definition, a single InQuIR file could just accept two systems concatenated, while common sense indicates that it could be convenient to have an arbitrary number of systems concatenated to create a queue of systems.
- *Definition of 'process' simplified.* In the original grammar, the nonterminal variable 'process' agglutinates all the instructions when, looking at an InQuIR file, the 'process' looks like it has to be the whole circuit of a single QPU. This way,

each line can be treated as a nonterminal of two types: quantum instruction and function.

- *Differentiation of data and communication qubits.* Although it has no implications in terms of physical implementation—a qubit is a qubit—, it is a nice practice to, given the fact that they can be differentiated in the syntax, separate the concept of communication qubit and data qubit. The earlier their functionalities are separated, the easier will be to treat them later.
- *Consideration of conditional gates.* Finally, the InQuIR grammar did not consider a pivotal case in its language: conditional gates. The conditional gates are exactly the same as the usual gates, but its application on the qubits is determined by a classical value. They are also referred to in the literature as intermediate measurements because, commonly, the classical value conditioning the application of the gate results from a measurement mid-execution. It must be said that in the actual grammar—the one exposed in Fig. 1 and that is implemented in their compiler—this case is considered, but it is important to remark that in the grammar here implemented, it is also considered

As already mentioned, this new grammar is implemented using ANTLR. The process involves generating a tree for each file containing InQuIR code, with the compiler walking through it, successively converting each sentence into a NetQASM instruction. One advantage of using ANTLR is that it automatically generates the necessary methods to create and walk the tree from the files, so the developer can concentrate on the actual task: transforming each piece of InQuIR code into NetQASM's.

4.2 Transforming InQuIR into NetQASM

Now, it is time to actually transform the code. Many details and peculiarities have been noted, but if any has been missed, it will be pointed out here. The code that InQuIR is transformed into is the protosubroutine code, i.e. the human-readable version of the NetQASM. This is important to mention because one of the possible improvements that this work could have is the direct translation of InQuIR into binary subroutines.

First, Table 1 shows the equivalence of the instructions. There is a lot to say about the contents of this table. At first sight, it highlights the fact that every InQuIR instruction is equivalent to one or more NetQASM instructions, except in a few cases where they lead to no instructions at all. The latter case is, however, quite logical: these instructions correspond to operations that NetQASM identifies as the application layer's responsibility. Going back to Fig. 2, the application layer was responsible for classical communication with the application layer of another node. This way of handling classical communication is probably one of the most, if not the most, detrimental aspects of NetQASM when used in the context of multicore architectures. Having to rely on SDK to perform classical communication drastically reduces efficiency due to the need to add extra layers of software that just send and receive information. Again, as will be pointed

Table 1 Equivalence between InQuIR and NetQASM instructions

InQuIR instructions	NetQASM instructions
world = open[0,1,2,...]	NONE
q0 = init()	set Q0 0
	qalloc Q0
	init Q0
_cq0 = genEnt[1](l0)	array 10 @0
	array 1 @1
	store 0 @1[0]
	array 20 @2
	store 0 @2[0]
	store 1 @2[1]
	create_epr(1,0) 1 2 0
	wait_all @0[0:10]
(_cq0, _cq1) = entSwap(0, 1)	Equivalent to genEnt
U q0 _cq0	U Q0 Q1
U[_m1] q0	Division in two subroutines
send[1](world, l1:_m0)	NONE
recv(world, l1_2:_m1)	NONE
RCXC[1](world, l0, q0, _cq0)	Example in Chapter 5
RCXT[0](world, l1, q1, _cq1)	Example in Chapter 5
c0 = measure q0	array 1 @3
	meas Q0
	qfree Q0
	store M0 @3[0]

out in the discussion below, this is a good indicator that the level of abstraction achieved by InQuIR is higher than that of NetQASM. In fact, InQuIR guarantees a much wider range of architecture types.

Moreover, this table also allows us to see the qubit virtualisation of the NetQASM model. It is common sense that the registers starting with "Q" are actually qubits on the back. But looking at the actual lines, it would be impossible to affirm that this is code for quantum computation without any knowledge of the context of this assembler code.

Another notable case is that of conditional gates. The table shows that this type of instruction causes the subroutine to be split into two parts. This is due to how the NetQASM model was conceived. Only the application layer can communicate with other nodes, which means that the NetQASM subroutines cannot have classical communication with other nodes. Therefore, at the application layer, when a conditional gate is encountered, two subroutines have to be compiled: the one with the gate applied and the one without the gate. This way, when the classical bit is received or measured inside the own QNPU, the correct subroutine is sent, and the other one is discarded. This will be illustrated in Sect. 5 with the example examined. This is quite inefficient for the multicore architecture this work is

designed for due to the same reasons explained before when talking about the instructions that were translated to nothing.

Furthermore, the RCXC and RCXT cases will be analysed in the next section since the example treated is a remote CX gate. It will be shown what circuit these gates produce when converted to NetQASM. It should be noted that although they are included in the grammar, the actual RCXC and RCXT instructions are not written in the example. Instead, the entire circuits equivalent to these gates are expressed as an illustrative way of applying the entire table of equivalences rather than just using these instructions.

Finally, the entanglement swap, i.e. the 'entSwap' operation, is treated as equivalent to a 'genEnt', i.e. a normal entanglement generation. This is the opposite case of conditional gates, in the sense that the application layer does not know how the quantum connections are (it only knows the number of nodes) and, therefore, the QNPUs automatically performs the necessary entanglement swaps without the user above noticing. This is a feature that InQuIR should probably adopt to add a bit more abstraction. Again, this will be considered in the discussion of Sect. 6.

As a final remark, one question that may arise is whether the compiled NetQASM code inherits certain properties from the original InQuIR code. Specifically, if an InQuIR program guarantees, for example, freedom from deadlocks or qubit exhaustion, does the compiled NetQASM code maintain these guarantees? Inheritance is a deep question in this field of compilation and often depends on the specific scenario. In this case, deadlocks may or may not occur in a compiled NetQASM program because they fall under the responsibility of the application layer, which NetQASM does not manage. On the other hand, qubit exhaustion is tied to the quantum aspects of the program, which are indeed encoded by NetQASM subroutines. Therefore, qubit exhaustion guarantees are preserved. As this example illustrates, quantum characteristics are likely to be inherited when using the InQASM compiler, whereas classical aspects are not, due to NetQASM subroutines' inability to handle classical functionalities.

4.3 Testing the resulting code

So, as the code transformation has been performed, it is time to test it to see if the translated code has actually been translated correctly. This exercise checks two aspects: first, the correctness of the translated code, and second, the quality of the translation compared to the NetQASM code generator. The NetQASM code generator to be used is the one implemented by the previously mentioned Python SDK. It has to be said that the compiler efficiency will not be tested in terms of time performance. This is a first stage compiler with no optimisations added to it. This means that the operations performed are tokenisation, parsing and walking the abstract syntax tree (AST) in order to generate the NetQASM code. This is a very efficient process and, even more important, it mainly depends on the ANTLR implementation.

Basically, the code will be tested using two Python programs: one generating the NetQASM code from the Python directives and the other taking the NetQASM generated from the InQuIR code as input. Both will execute the subroutines in the

NetSquid simulator using the SquidASM library to compare the results and see if they match.

Examples following these steps will be shown in Sect. 5, allowing for a final and thorough analysis that will set up all for a final discussion about this work, InQuIR and NetQASM, their most significant problems and virtues.

5 Results

In this chapter, the distributed CX will be the operation used, as already noted in Table 1. This operation is chosen mainly because it is the baseline for DQC. The ability to correctly execute a distributed CX allows the execution of any other controlled gate between different QPUs simply by changing the X for the gate of interest.

The circuit in question is shown in Fig. 5. This figure shows why this operation, the distributed CX, is the baseline for DQC. This circuit is a well-known case in the DQC literature, usually called *telegate*, composed of two parts, the cat-entangler and the cat-disentangler. The communication qubit of QPU_1 is set to the same state as $|\psi\rangle$ by the effect of the first block, the cat-entangler, and then the effect of the gate is kicked back to the QPU_0 by the effect of the second, the cat-disentangler. This allows this qubit to act as the control qubit for the local controlled operations in QPU_1. There are other methods, such as the *teledata*, also known as *teleport*. The reader is referred to Barral et al. [1] for more details.

To carry out a thorough analysis, the InQuIR code for implementing this gate is first shown. Then, the NetQASM generated from this code is shown. For comparison purposes, as explained in Sect. 4.3, the code generated by the Python SDK is depicted. Both codes are compared to show differences and possible improvements. Finally, both codes are simulated and the outputs after the simulation will be compared to show that the generated code behaves as is should.

It is important to note that only one example has been compared with the Python SKD code, which is clearly insufficient to fully prove the compiler's functionality.

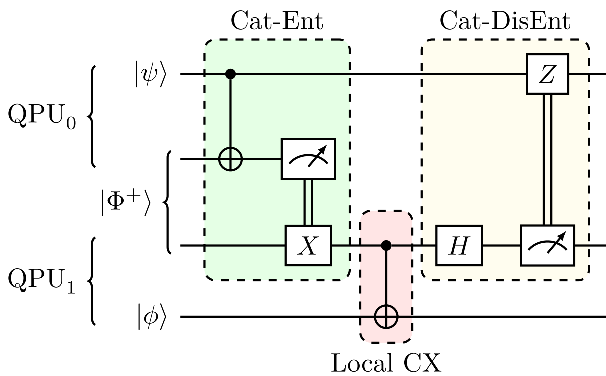


Fig. 5 Distributed CX

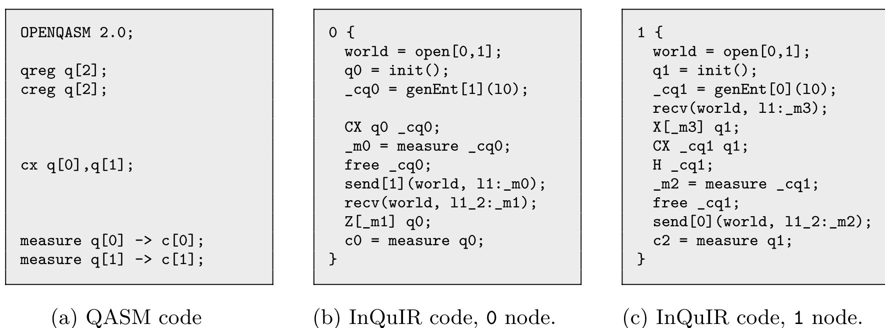
The purpose of this example is not to demonstrate flawless performance but rather to illustrate how the compiler operates and how its results compare to those produced by the NetQASM Python SDK. The Python SDK is designed for writing programs using Python directives, rather than injecting external NetQASM subroutines, which further complicates the construction of more direct comparisons. However, to better illustrate the compiler's functionality, additional and more complex examples are available in the repository for readers to explore. It should be noted that these examples cannot currently be tested using the NetQASM Python SDK, as the SDK does not support the injection of externally generated subroutines, making it impossible to execute them in certain cases. Consequently, these examples are not included in this document. Moreover, in order to reproduce the results here presented, the versions of the software employed are specified in Appendix C.

5.1 InQuIR code

For this example, the InQuIR code could be hand-written. Instead, the InQuIR software introduced in Sect. 3.1.1 was used. The input to generate the InQuIR code was a simple QASM code with a CX and an architecture of two QPUs. This code and the corresponding InQuIR code can be seen side by side in Fig. 6. Obviously, the number of instructions introduced in the InQuIR code is significant due to the need to establish the distributed connection.

5.2 NetQASM code of the translation

Following the equivalences in Table 1, the NetQASM code is now generated by the compiler. In Fig. 7, you can see the resulting code for QPU_0 and in Fig. 8 the same for QPU_1. First, the effect of conditional gates, as explained in Sect. 4.2, can be seen at first glance by the fact that there is more than one subroutine for each QPU. One might think that this could lead to an exponential increase in the number of NetQASM files for each InQuIR file, but there are at most $2n + 1$ files in each QPU,



(a) QASM code (b) InQuIR code, 0 node. (c) InQuIR code, 1 node.

Fig. 6 QASM code and its respective InQuIR's generated one

where n is the number of conditional gates applied. Otherwise, the result obtained is just the consequence of applying the equivalence table to each row.

5.3 NetQASM code of the python SDK

Now, in Figs. 9 and 10, you can see the code produced by the Python SDK for each QPU. The code shown in these chunks is quite similar to that produced by this work's compiler, but there are some significant differences. First of all, in QPU_0 in Fig. 9a, the instruction `set Q0 1` is repeated almost continuously, which is obviously something completely inefficient. This can also be seen in Fig. 10b, c with the instruction `set Q0 0`. But probably the most outstanding result is the one observed in Fig. 10a, where all the highlighted code is completely unnecessary. The reasons why the Python SDK generated this piece of code are unknown, but after trying the subroutines with and without this piece of code, the results were equivalent. This means that this code's generation is unnecessary and could—and should—be eliminated in a first simple round of optimisations. This shows a pretty big mistake compared to the previous instruction repetition because not only are classical operations repeated, but quantum ones are as well. Every quantum operation introduces errors into the qubits, so it is always helpful to minimise the use of unnecessary ones. These two inaccuracies show a problem of the NetQASM software: the lack of optimisation when generating the code of the subroutine. This represents a possible improvement in future works.

5.4 Comparison of results

Finally, after all the subroutines are obtained, the code is executed in the NetSquid simulator. Instead of executing and comparing the measurements (which would require a serious statistical analysis), NetSquid allows obtaining the density matrix of the joint state of the control qubit in QPU_0 and the target qubit in QPU_1 (thanks to the SquidASM library). This allows for a robust comparison of the results, because instead of having to mathematically ensure that both distributions are equal, a direct comparison between density matrix gives us the exact answer.

In order to compare the result between both circuits, several states have to be tested. In order to do so, two rotation operations—one in the X axis and one in the Y axis—for the control and the target qubits are added to both the compiled and the generated by the Python SDK subroutines. This makes us able to generate arbitrary states for the control and the target qubits, obtaining an actual exam on whether the compilation files are obtaining the same state as the Python SDK files. Twenty different angles were selected, equally spaced along the unity circle—i.e. the multiples of $\frac{\pi}{10}$ until reaching 2π. In all the cases, the results were exactly the same excepting approximations.

This allows us to conclude that the translation of InQuIR code to NetQASM subroutines is being correctly performed for this example, creating a compilation pipeline between these two pieces of software.

6 Conclusion

This section starts by discussing all the different aspects of the InQuIR and NetQASM languages, along with the key aspects of the simple compiler developed in this work. After this, a summary of the work and ideas will be made to extract some solid conclusions. Finally, a discussion of future work will conclude this manuscript.

6.1 Discussion

This work has portrayed a translation from the InQuIR language to the NetQASM language. This has been referred to as a compiler, i.e. as a tool that translates code to another language that represents an inferior level of abstraction. The first point to discuss is whether this characterisation is accurate. The correct answer is that it depends on the context. While InQuIR is explicitly defined as an IR for interconnected quantum computers, NetQASM has not been clearly established as an assembly for quantum interconnects. As already mentioned in Sect. 2.3, NetQASM is defined as a *low-level assembler*, which does not precisely stress the assembly nature of NetQASM. In fact, when the concept of "flavours" was introduced, it was mentioned that with the vanilla flavour, i.e. the basic one, NetQASM behaves similarly to a IR. Because of this, considering the software developed in this work a transpiler instead of a compiler would be completely correct.

Focusing on the various aspects of the languages, several points can be highlighted. Starting with InQuIR, some aspects where found problematic along the development. First of all, the fact that InQuIR needs to know about the connectivity of the underlying architecture is quite a low-level characteristic to an IR. Even though it is not critical, it forces InQuIR to implement the entanglement swap operations, which is something related to qubit routing in DQC architectures. An IR should just indicate that two processes have to share an entanglement pair and, when generating the code for a specific machine with its specific architecture, the entanglement swap operations should be performed to allow this sharing. This could be compared to the classical case when an expression like $a = b + c$ is performed. The user at IR level has nothing to do with the transportation of the different registers involved with the arithmetic logic unit (ALU). Also, the grammar in this work is poorly defined due to the lack of a standardised definition using a meta-syntax language. This has been corrected in this work by using EBNF, as already shown in Fig. 4.

```
# NETQASM 1.0
# APPID 0
set Q0 0
qalloc Q0
init Q0
set Q1 1
qalloc Q1
init Q1
array 10 @0
array 1 @1
array 20 @2
store 1 @1[0]
store 0 @2[0]
store 1 @2[1]
create_epr(1, 0) 1 2 0
wait_all @0[0:10]
cnot Q0 Q1
meas Q1 M0
qfree Q1
array 1 @3
store M0 @3[0]
ret_arr @0
ret_arr @1
ret_arr @2
ret_arr @3
```

(a) 1st protosubroutine

```
# NETQASM 1.0
# APPID 0
set Q0 0
z Q0
```

(b) 2nd protosubroutine

Fig. 7 NetQASM translated for QPU_0

```
# NETQASM 1.0
# APPID 0
set Q0 1
qalloc Q0
init Q0
array 10 @0
array 1 @1
store 0 @1[0]
recv_epr(0,0) 1 0
wait_all @0[0:10]
ret_arr @0
ret_arr @1
```

(a) 1st protosub.

```
# NETQASM 1.0
# APPID 0
set Q0 0
set Q1 1
x Q0
cnot Q0 Q1
h Q0
array 1 @2
meas Q0 M0
qfree Q0
store M0 @2[0]
ret_arr @2
```

(b) 2nd protosub. (version 1)

```
# NETQASM 1.0
# APPID 0
set Q0 0
set Q1 1
cnot Q0 Q1
h Q0
array 1 @2
meas Q0 M0
qfree Q0
store M0 @2[0]
ret_arr @2
```

(c) 2nd protosub. (version 2)

Fig. 8 NetQASM translated for QPU_1

Now, talking about NetQASM, the fact that, as explained since the beginning, this language is thought for quantum networks brings unnecessary layers to the language and the model regarding the scope of multicore architectures adopted in this work. This can be perfectly observed in, for instance, the need to send and receive the subroutines between the QNPU and the application layer. This could be performed exclusively by a small classical unit inside the QNPU to fit in the multicore model. Moreover, this is related to how intermediate measurements and, therefore, conditional gates are treated. In fact, this is a strong point to argue that NetQASM

```
# NETQASM 1.0
# APPID 0
array 10 @0
array 1 @1
store 0 @1[0]
array 20 @2
store 0 @2[0]
store 1 @2[1]
array 1 @3
create_epr(1,0) 1 2 0
wait_all @0[0:10]
set Q0 1
qalloc Q0
init Q0
set Q0 1
set Q1 0
cnot Q0 Q1
set Q0 0
meas Q0 M0
qfree Q0
store M0 @3[0]
ret_arr @0
ret_arr @1
ret_arr @2
ret_arr @3
```

(a) QASM code

```
# NETQASM 1.0
# APPID 0
set Q0 1
z Q0
```

(b) InQuIR code

Fig. 9 NetQASM code for the QPU_0

cannot be an IR because it should treat conditional gates as an implemented operation and leave to the lower stages of the code the way of implementing that type of operation. The way employed of getting the classical bit to the application layer and then sending the correct subroutine seems like too much specification for an IR. Even more if one thinks about the fact that, in the future, an architecture focused on a distributed model could be created with conditional gate as an operation supported as native. Something that has to be pointed out as a positive feature of NetQASM is the fact that it does not implement entanglement swaps, leaving that responsibility to the QNPU, just as it was suggested to the InQuIR language in the previous paragraph.

Now, looking at the software infrastructure surrounding both languages, InQuIR and NetQASM, a couple of notes have to be made. In the case of InQuIR, it has a simplified compiler associated, as already noted. Using this compiler provides a useful way to observe how the connectivity is employed, as mentioned before, but it is intended only as a support tool rather than a fully developed software solution. Regarding the NetQASM software, the code generation does not perform the most basic optimisation techniques as it is to check if an instruction is duplicated. This provokes that, as shown in the codes of Appendix B with the highlighted lines, several code lines end up being duplicated. Even more, it even generates a ton of unnecessary instructions, as depicted in Fig. 10a with, again, the highlighted lines.

6.2 Summary

So, to sum up, a correct translation has been portrayed from InQuIR to NetQASM code. This has shown different aspects of both languages and, even more, of both works. InQuIR and NetQASM have shown different benefits and flaws for the distributed compilation stack, which will be taken into account in future works in order to build a strong backbone for DQC.

In fact, a possible conclusion that can be extracted is the fact that a refined IR for distributed architectures is still to be constructed. This is a surprising conclusion if one considers that this work was born with the intent of linking an assembly and an IR for DQC. But, again, as it has been shown in this work, the boundaries among stack levels are diffuse and can reveal themselves as almost nonexistent sometimes, as has happened with InQuIR and NetQASM.

6.3 Future work

As future work, it has already been hinted that an IR is to be defined for DQC. Moreover, with this new IR this compiler could be modified to directly translate to NetQASM's different flavours (that represent the actual low level of the NetQASM software compound) or directly to the different simulators exposed in Sect. 2.4. Moreover, a lot of techniques could be introduced to generate higher-quality code. For example, static checks about qubit exhaustion and deadlocks can be implemented for the IR, as presented for the InQuIR. Another example is the implementation of optimisations for each backend in order to increase the performance or even optimisations to the IR that will reduce communications or other desired metrics (T-count, depth, size, etc.).

NetQASM code of the translation

In this section, the NetQASM code generated from the compilation—referred in the title as translation—is displayed. Figures 7 and 8 show the code obtained from the compiled code of Figs. 6b, c.

NetQASM code of the python SDK

In this section, on the contrary, the NetQASM code displayed is the one automatically generated by the Python SDK. In Figs. 9 and 10 is shown the code obtained from compiled the code of Fig. 6b, c, the same piece of InQuIR code as before.

Software version

The software used in this manuscript is the *INQASM v0.1.0* release, available on the project's GitHub repository. All package versions required by the software are listed in the release notes for reproducibility.

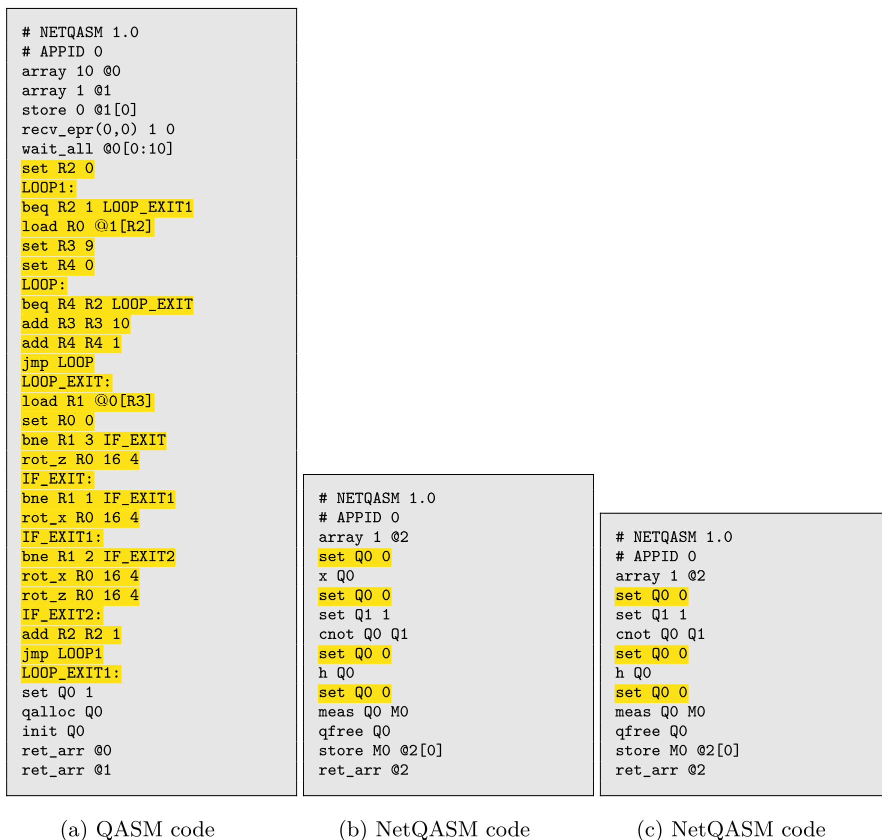


Fig. 10 NetQASM code for the QPU_1

Author contributions Jorge Vázquez-Pérez wrote de main de manuscript text. All authors reviewed the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by MICINN through the European Union NextGenerationEU recovery plan (PRTR-C17.I1) and by the Galician Regional Government through the “Planes Complementarios de I+D+I con las Comunidades Autónomas” in Quantum Communication. Simulations on this work were performed using the Finisterrae III Supercomputer, funded by the project CESGA-01 FINISTERRAE III. This work was also supported by the Ministry of Economy and Competitiveness, Government of Spain (Grants Number PID2019-104834GB-I00, PID2022-141623NB-I00 and PID2022-137061OB-C22), Consellería de Cultura, Educación e Ordenación Universitaria (accreditations ED431C 2022/16 and ED431G-2019/04), and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Center in Intelligent Technologies of the University of Santiago de Compostela as a Research Center of the Galician University System.

Data availability Specific data are available in the GitHub repository: https://github.com/jorgevazquezperez/InQASM.

Declarations

Conflict of interest The authors have no Conflict of interest that may have affected the content of this work.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Barral D, Cardama FJ, Díaz G, Faílde D, Llovo IF, Juane MM, Vá¡zquez-Pérez J, Villasuso J, Piñeiro C, Costas N, Pichel JC, Pena TF, Gómez A (2024) Review of distributed quantum computing. From single QPU to high performance quantum computing
2. Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput 26(5):1484–1509. https://doi.org/10.1137/s0097539795293172
3. Gyongyosi L, Imre S (2019) A survey on quantum computing technology. Comput Sci Rev 31:51–71. https://doi.org/10.1016/j.cosrev.2018.11.002
4. Giovannetti V, Lloyd S, Maccone L (2008) Quantum random access memory. Phys Rev Lett 100:160501. https://doi.org/10.1103/PhysRevLett.100.160501
5. Jaques S, Rattew AG (2023) QRAM: a survey and critique . arXiv:2305.10310
6. Phalak K, Chatterjee A, Ghosh S (2023) Quantum random access memory for dummies. Sensors 23(17):7462. https://doi.org/10.3390/s23177462
7. Ryan CA, Johnson BR, Risté D, Donovan B, Ohki TA (2017) Hardware for dynamic quantum computing. Rev Sci Instrum 88(10):104703. https://doi.org/10.1063/1.5006525
8. Qin X, Zhang W, Wang L, Zhao Y, Tong Y, Rong X, Du J (2020) An fpga-based hardware platform for the control of spin-based quantum systems. IEEE Trans Instrum Meas 69(4):1127–1139. https://doi.org/10.1109/TIM.2019.2910921
9. Vázquez-Pérez J, Piñeiro C, Pichel JC, Pena TF, Gómez A (2024) Qpu integration in opencl for heterogeneous programming. J Supercomput 80(8):11682–11703. https://doi.org/10.1007/s11227-023-05879-9
10. Saurabh N, Jha S, Luckow A (2023) A conceptual architecture for a quantum-HPC middleware . arXiv:2308.06608
11. Serrano MA, Cruz-Lemus JA, Perez-Castillo R, Piattini M (2022) Quantum software components and platforms: overview and quality assessment. ACM Comput Surv 55(8):1–31. https://doi.org/10.1145/3548679
12. Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley Longman Publishing Co. Inc, USA
13. Svore KM, Aho AV, Cross AW, Chuang I, Markov IL (2006) A layered software architecture for quantum computing design tools. Computer 39(1):74–83. https://doi.org/10.1109/MC.2006.4
14. Heim B, Soeken M, Marshall S, Granade C, Roetteler M, Geller A, Troyer M, Svore K (2020) Quantum programming languages. Nat Rev Phys 2(12):709–722. https://doi.org/10.1038/s42254-020-00245-7
15. Developers C (2024) Cirq Zenodo. https://doi.org/10.5281/zenodo.11398048
16. Contributors Q (2023) Qiskit: an open-source framework for quantum computing . https://doi.org/10.5281/zenodo.2573505

17. Schmale T, Temesi B, Baishya A, Pulido-Mateo N, Krinner L, Dubielzig T, Ospelkaus C, Weimer H, Borcherding D (2022) Backend compiler phases for trapped-ion quantum computers. In: 2022 IEEE International Conference on Quantum Software (QSW), pp. 32–37 . https://doi.org/10.1109/QSW55613.2022.00020
18. Häner T, Steiger DS, Smelyanskiy M, Troyer M (2016) High performance emulation of quantum circuits. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16. IEEE Press, Salt Lake City, Utah
19. DiVincenzo DP (2000) The physical implementation of quantum computation. Fortschr Phys 48(9–11):771–783. https://doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E
20. Lu F (2021) Several ways to implement qubits in physics. J Phys: Conf Ser 1865(2):022007. https://doi.org/10.1088/1742-6596/1865/2/022007
21. Chowdhary KR (2021) Software-hardware evolution and birth of multicore processors. arXiv:2112.06436
22. Jnane H, Undseth B, Cai Z, Benjamin SC, Koczor B (2022) Multicore quantum computing. Phys Rev Appl 18(4):044064. https://doi.org/10.1103/physrevapplied.18.044064
23. Rudinger K, Proctor T, Langharst D, Sarovar M, Young K, Blume-Kohout R (2019) Probing context-dependent errors in quantum processors. Phys Rev X 9:021045. https://doi.org/10.1103/PhysRevX.9.021045
24. Sheldon S, Magesan E, Chow JM, Gambetta JM (2016) Procedure for systematically tuning up cross-talk in the cross-resonance gate. Phys Rev A 93:060302. https://doi.org/10.1103/PhysRevA.93.060302
25. Piltz C, Sriarunothai T, Varón AF, Wunderlich C (2014) A trapped-ion-based quantum byte with 10-5 next-neighbour cross-talk. Nat Commun 5:1 5, 1–10 https://doi.org/10.1038/ncomms5679
26. Nishio S, Wakizaka R (2023) Inquir: intermediate representation for interconnected quantum computers. https://doi.org/10.48550/arXiv.2302.00267
27. Dahlberg A, Vecht BVD, Donne CD, Skrzypczyk M, Raa IT, Kozlowski W, Wehner S (2022) Netqasm-a low-level instruction set architecture for hybrid quantum-classical programs in a quantum internet. Quantum Sci Technol 7:035023. https://doi.org/10.1088/2058-9565/AC753F
28. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in openMP. Morgan Kaufmann Publishers Inc., San Francisco
29. Message passing interface forum: MPI: a message-passing interface standard version 4.1. (2023). https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
30. Häner T, Steiger DS, Hoefler T, Troyer M (2021) Distributed quantum computing with qmpi. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3458817.3476172
31. Shi Y, Nguyen T, Stein S, Stavenger T, Warner M, Roetteler M, Hoefler T, Li A (2023) A reference implementation for a quantum message passing interface. In: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 1420–1425. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3624062.3624212
32. Cross AW, Bishop LS, Smolin JA, Gambetta JM (2017) Open quantum assembly language
33. Fu X, Riesebos L, Rol MA, Straten J, Someren J, Khammassi N, Ashraf I, Vermeulen RFL, Newsum V, Loh KKL, Sterke JC, Vlothuizen WJ, Schouten RN, Almudever CG, DiCarlo L, Bertels K (2019) eQASM: an executable quantum instruction set architecture
34. Khammassi N, Guerreschi GG, Ashraf I, Hogaboam JW, Almudever CG, Bertels K (2018) cQASM v1.0: towards a common quantum assembly language
35. Muralidharan S (2024) The simulation of distributed quantum algorithms. arXiv:2402.10745
36. Diadamo S, Nötzel J, Zanger B, Beşe MM (2021) Qunetsim: a software framework for quantum networks. IEEE Trans Quantum Eng 2:1–12. https://doi.org/10.1109/TQE.2021.3092395
37. Dahlberg A, Wehner S (2018) Simulaqron- simulator for developing quantum internet software. Quantum Sci Technol 4:015001. https://doi.org/10.1088/2058-9565/AAD56E
38. Bartlett B (2018) A distributed simulation framework for quantum networks and channels. arXiv:1808.07047
39. Ferrari D, Amoretti M (2023) A design framework for the simulation of distributed quantum computing. arXiv:2306.11539
40. Radzihovsky M, Espinosa Z (2020) netquil: a quantum playground for distributed quantum computing simulations. Bull Am Phys Soc **65**

41. Parekh R, Ricciardi A, Darwish A, Diadamo S (2021) Quantum algorithms and simulation for parallel and distributed quantum computing. Proceedings of QCS 2021: 2nd International Workshop on Quantum Computing Software, Held in Conjunction with SC 2021: The International Conference for High Performance Computing, Networking, Storage and Analysis, 9–19 https://doi.org/10.1109/QCS54837.2021.00005
42. Matsuo T (2019) Simulation of a dynamic, ruleset-based quantum network. arXiv:1908.10758
43. Mailloux LO, Morris JD, Grimaila MR, Hodson DD, Jacques DR, Colombi JM, McLaughlin CV, Holes JA (2015) A modeling framework for studying quantum key distribution system implementation nonidealities. IEEE Access 3:110–130. https://doi.org/10.1109/ACCESS.2015.2399101
44. Wu X, Kolar A, Chung J, Jin D, Zhong T, Kettimuthu R, Suchara M (2020) Sequence: a customizable discrete-event simulator of quantum networks. Quantum Sci Technol 6:045027. https://doi.org/10.1088/2058-9565/ac22f6
45. Coopmans T, Knegjens R, Dahlberg A, Maier D, Nijsten L, Oliveira Filho J, Papendrecht M, Rabbie J, Rozpędek F, Skrzypczyk M, Wubben L, Jong W, Podareanu D, Torres-Knoop A, Elkouss D, Wehner S (2021) Netsquid, a network simulator for quantum information using discrete events. Commun Phys 4:1 4, 1–15 https://doi.org/10.1038/s42005-021-00647-8
46. Scowen RS (1993) Generic base standards. In: proceedings 1993 software engineering standards symposium, pp. 25–34 . https://doi.org/10.1109/SESS.1993.263968
47. Parr T (2013) The definitive ANTLR 4 reference, 2nd edn. Pragmatic Bookshelf, Raleigh

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.