Journal of Computational Electronics (2019) 18:329–342
https://doi.org/10.1007/s10825-018-1287-5

An FPGA-based real quantum computer emulator

Jakub Pilch[1] · Jacek Długopolski[1]

Published online: 15 December 2018

Abstract
While we cannot efficiently emulate quantum algorithms on classical architectures, we can move the weight of complexity from time to hardware resources. This paper describes a proposition of a universal and scalable quantum computer emulator, in which the FPGA hardware emulates the behavior of a real quantum system, capable of running quantum algorithms while maintaining their natural time complexity. The article also shows the proposed quantum emulator architecture, exposing a standard programming interface, and working results of an implementation of an exemplary quantum algorithm.

Keywords Quantum emulator · Qubit · FPGA · Parallel

1 Introduction

It has been over 30 years since the idea of using quantum circuits to perform computational tasks was first proposed. In spite of the rapid growth of our knowledge in quantum physics, a fully workable quantum computer of a desired scale still remains outside our reach. Many different approaches have been taken, from nano-scale superconducting circuits to ion-traps. Yet, we are still far from running most of the already developed quantum algorithms in a useful scale. What cannot be denied, however, is the great potential and power of quantum computing. This is what drives researchers around the world to develop new algorithms for machines that we cannot be sure will ever come to existence.

The praised quantum speedup of many famous algorithms comes from massive parallelism in quantum computation. However, as it was suggested by Richard Feynman and stated by the Quantum Strong Church–Turing Thesis, only quantum machines are capable of efficient emulation of quantum circuits [1]. In other words—there is no physical way to achieve the quantum speedup on classical, sequential machines. Emulation of any quantum algorithm on a standard computer will often require exponentially more time than it would on a quantum machine. While we cannot bypass that need for resources when emulating quantum circuits, we can shift the weight from time to hardware complexity. This is where field-programmable gate arrays, or FPGAs, come to help us.

This paper describes an approach to build a very scalable, easily parametrized and programmable universal quantum computer emulator, reflecting natural behaviors of real quantum circuits. We designed the hardware to physically emulate qubits, with quantum manipulation methods provided by unitary matrices and a randomizer to introduce the uncertainty of quantum systems (as seen in FPGA section in Fig. 1).

Our primary objectives included:

- Natural parallelization—every gate can be applied to the state in a single operation/clock-tick, regardless of the number of emulated qubits
- Universality—rather than pre-implementing gates, our design can run any gates sent by the user to the processor
- Code-level scalability—modifying a single parameter in our code is enough to change the emulated qubits count, numerical precision and other processor parameters.

Our implementation can be viewed as a reproduction of a physical, universal quantum computer, where our hardware qubits replace ion traps or particle spins, and manipulation matrices imitate precise lasers or magnetic field generators. The created system is capable of running any quantum algorithm completely in hardware (only limited by available

✉ Jakub Pilch
jfpilch@gmail.com

Jacek Długopolski
dlugopol@agh.edu.pl

1 Department of Computer Science, Faculty of Computer Science, Electronics and Telecommunications, AGH University of Science and Technology, al. A. Mickiewicza 30, 30-059 Kraków, Poland

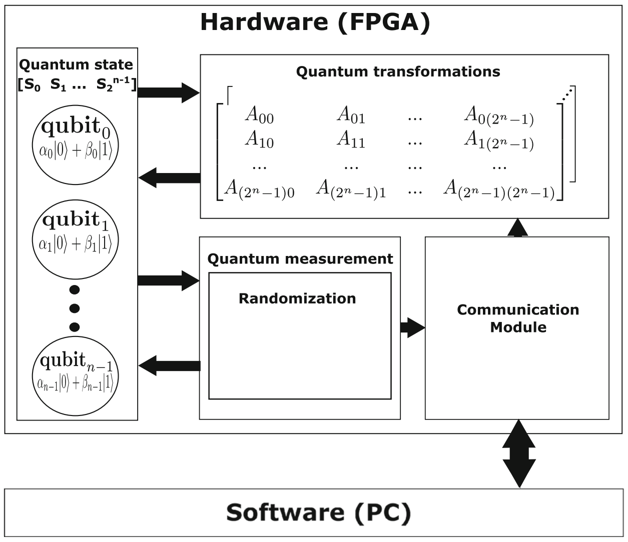

Fig. 1 Quantum system overview

FPGA resources), while maintaining time complexity and reflecting natural behaviors of quantum circuits, including parallelism. The processor exposes a standard programming interface, allowing the user to design and use any quantum gates from the software level, without any hardware modification.

2 Motivation

Today, we are at the very beginning of a quantum revolution. Companies such as IBM [2], Rigetti [3] or D-Wave [4] are investing in building physical quantum computers and exposing programmable interfaces to users around the world through cloud solutions. At the same time, real quantum machines still have a long way to go before they become truly universal, stable and scalable. Many of today's architectures are specifically tailored to run specialized algorithms, such as quantum annealing [5]. Furthermore, spontaneous decoherence, state preparation and measurement faults, or the engineering challenge of creating entangled states, are serious problems that we are learning to solve in the real world. For these reasons, quantum emulators are still being designed and built to help on different frontiers of the quantum revolution.

The goal of our work described in the article was to build a quantum computer emulator, which from user's perspective would behave like a real quantum computer. The primary focus was to ensure that every operation is naturally parallelized, and therefore every transformation on the n-qubit state is completed in a single clock-tick.

However, it was not our intention to simulate actual physical phenomena that would take place in a physical quantum machine. Instead, we wanted to provide users with the comfort of running a universal emulator that delivers results practically as fast as a quantum machine would, but without having to pay attention to error correction and other undesired side-effects. Therefore, the current design of our system emulates a quantum computer composed of perfect qubits.

Thanks to this approach, we hoped to enable users to design, test and iterate on quantum algorithms without the need to own, run and maintain an actual quantum computer. With this in mind, we considered spontaneous decoherence, state setup uncertainty and measurement faults to be imperfections of today's real quantum systems and, therefore, did not implement them in our design.

Because of the complexity of the problem, as well as hardware limitations (small available FPGA chips), we had to compromise on some of the important implementation details. In the end we were only able to synthesize a two-qubit system. One of the most significant settlements was to use fixed-point number representation (based on [28]), which allowed us to dramatically reduce the need for FPGA resources, while limiting precision to a point that may be problematic for certain algorithms. We have not described this problem extensively, however, as the modular design allowed us to abstract the implementation of number representations, and we plan on replacing it with a much more precise floating-point engine in the future. The synthesized 10-bit fixed-point number representation was sufficient for the tested Deutsch Algorithm, as described in Sect. 8.

Another implementation choice driven by the need for simplicity in the first iteration was to implement quantum state processing as matrix–vector multiplications. The utilized approach is correct and universal, reflecting a traditional mathematical way of representing a quantum system's evolution. It is also easy to scale from code level, as described in Sect. 6.4, which was one of our main objectives. However, much more efficient computing methods for quantum state evolution, including those proposed by Nikahd et al. [26] or Viamontes et al. [27], already exist. Our choice allowed us to achieve our initial goals in the first iteration, and we plan to research and implement a more efficient method in the future versions, while keeping the universality and code-level scalability of the current solution.

Thanks to code-level scalability and modularity, the proposed architecture is designed to be easily extendable and flexible, allowing for simple iterative improvements. As larger FPGA chips become available, changing single parameters in our code enables growing the number of emulated qubits or improving numerical precision. Examples of easily scalable code include state transformation parametrized by emulated qubit count (as described in Sect. 6.4) or numerical type definition abstracted and defined in a single place (as described in Sect. 6.2). Furthermore, any of the existing modules, such as those responsible for state transformation or measurement, can be easily adjusted or replaced through modifying single, isolated pieces of code. It should be emphasized that we have described the first iteration of our approach, which was focused on building and verifying the architecture in practice, and thanks to our design it can be progressively improved in future iterations.

While our emulator utilizes the massive parallelism of an FPGA chip, it does so to emulate natural quantum parallelism, rather than to achieve the speed-up itself. Because every transformation in our design takes a single clock-tick, we can speed the design up by increasing the clock frequency as far as our FPGA hardware allows us. With every additional emulated qubit, if we can maintain the clock-frequency, our design gains an exponential advantage over sequential software emulators, thanks to an asymptotically faster hardware architecture. At the same time, we have not analysed the emulation speed, as it was not one of our initial objectives in itself.

To keep the design simple, universal and cross-compatible (including for future FPGA chips), we resigned from direct use of specialistic (and quite often custom) computational and memory modules, such as DSP or BRAM, available in today's FPGAs. Of course, it does not mean the modules are not used in our synthesized hardware—we left it for the compiler and synthesizer to decide where and when such improvements will be applied, using various available resources of the target chip. This way our architecture is completely independent of the hardware and truly universal.

3 Existing work

There are many approaches to emulate or simulate quantum computers. Many of them are built around GPU processors, multiprocessor systems or even supercomputers and focused on reducing the time necessary to emulate quantum algorithms on classical architectures [6–9]. While these solutions often provide useful tools to simulate outcomes of running quantum algorithms, they do not offer the time-complexity obtained by using real quantum computers.

FPGA technology provides an attractive opportunity to leverage its massive parallelism to completely emulate the time-speedup of quantum machines. There are many proposed emulator architectures based on FPGA circuits, but none of them are fully focused on simple, naturally parallel emulation of quantum circuits.

VHDL library proposed by Khalid et al. [10] puts emphasis on efficient computation of quantum circuits through analyzing code and ensuring it is implemented in the most efficient way on the FPGA. In their solution, the authors decided to model quantum circuits from pre-implemented gates, that are designed for fast execution on FPGAs and provided as a VHDL library. Therefore, the quantum circuit to be emulated must be known before synthesis. In our approach, hardware is built to run any quantum gates that can be represented by complex number matrices. Transformations are dynamically loaded from the software level, while the processor itself is gate-agnostic and universal. Furthermore, in our implementation, quantum measurement is fully parallelized and performed in hardware, as opposed to a software solution utilized by the authors of [10]. Finally, our design is built to emulate a quantum processor connected to a classical machine, allowing to run complete quantum algorithms in hardware, consisting of any number of gates, including state preparation, computation and measurement. This stands in contrast to using VHDL to synthesize hardware to speed-up parts of quantum algorithms, as described in [10].

Goto and Fujishima [11] designed a solution making use of unitary macro-operations, allowing memory-efficient simulation of quantum circuits on FPGA and corresponding to classical processors' behavior. Their approach was concentrated on decomposing macro-operations in software to some pre-designed hardware operations, and then running them in parallel on hardware, rather than trying to emulate physical, universal quantum computers.

Lee et al. [12] conducted extensive research on existing solutions and prepared a software–hardware system to emulate quantum computation with high precision and efficiency. The authors took an innovative approach of mixing parallel and serial processing on FPGA, which allowed them to achieve desired speedups without hardware resources' exponential growth. While this design provides great tools for quantum algorithm analysis, it was not focused on reflecting natural, fully parallelized behaviors in hardware. Every quantum operation emulated in our system is completely parallelized, which contributes to exponential growth of required hardware resources with every emulated qubit, but provides a closer reflection of natural quantum systems.

Problems with exponential growth of space and time complexity during simulation of larger quantum circuits were also addressed by Franka et al. [13]. They designed and conducted empirical complexity measurements of a working software prototype of a quantum computer simulator avoiding excessive space requirements. Rather than to emulate quantum systems in hardware, its purpose was to provide a space-efficient model for running quantum algorithms, with agreement for some space–time tradeoffs.

Negovetic et al. [18] has proposed a software–hardware system for emulating quantum circuits on FPGAs. They presented two approaches: one where a software preprocessor converts quantum netlists into HDL that can be then synthesized and ran on FPGA, and an evolvable one, where software generates a netlist satisfying problem constraints, and then gets it translated into hardware as in the first case. The authors also considered moving the netlist generation entirely to hardware. While both of the described solutions utilize FPGA parallelism to speed-up quantum computation emulation, they require re-synthesizing hardware according to generated HDL for every algorithm. This stands in contrast to our design, where software dynamically loads gate matrices to hardware to emulate any quantum algorithm without resynthesis. Furthermore, our architecture allows quantum gate matrices of any size, only limited by the number of emulated qubits, compared to only 1- and 2-qubit gates proposed in [18].

Fujishima et al. [19,21] proposed a very interesting approach of utilizing FPGAs for high-speed quantum computing emulations with small memory requirements. The described architecture was designed to solve search-based problems. Utilizing the fact that initial state amplitudes would be always either 0 or $\frac{1}{\sqrt{m}}$, where m is the number of possible solution candidates, the proposed logic quantum processor represents initial amplitudes with single bits, rather than complex numbers, contributing to large memory savings and computation speed. The emulator also included stochastic bit error simulation to help emulate quantum systems' behavior. Fujishima et al. [20,22] proposed an improved design, where rather than storing the entire quantum state vector, the architecture includes a quantum index processor, which only keeps track of the indices of bits set to 1 in the state. This resulted in even greater memory savings, which allowed for synthesis of a massive 75-qubit emulator. While the proposed architectures have proven to be very fast, the emulated quantum operations are restricted to Walsh–Hadamard and C-NOT gates, and a non-quantum INQUIRY operation is added to enable emulation of certain algorithms (like Shor's factorization algorithm [25].) The architecture proposed in our paper is less optimized for fast emulations, but is fully universal and capable of running any quantum gates and algorithms, limited only by the number of emulated qubits.

Aminian et al. [23] has described a universal and efficient method of emulating quantum circuits on FPGAs. The authors proposed an efficient way to emulate a universal set of quantum gates on FPGA hardware and tested multiple algorithms constructed from gates in the set. This interesting and scalable approach focuses on emulating particular gates in hardware and using them to emulate quantum algorithms. However, it does not allow direct usage of any desired quantum gate in hardware without increasing circuit depth, as opposed to our solution, where any gate matrix can be dynamically loaded into hardware without resynthesis.

There is also an interesting proposition of QuIDE software solution [14], providing a quantum computer simulator with an integrated development environment. It has been designed to create quantum programs with code (C# QuIDE library), as well as with a graphical circuit designer. The authors are working on a bridge between the IBM-Q environment [2] and their own QuIDE simulator. Software tools such as QuIDE are focused on providing tools for easier quantum algorithm design and not on any form of emulating quantum speedup. However, there are also many other quantum computer emulators for PCs, including QDD emulation library for C++ [24], focusing on optimal use of memory and processing power for the fastest emulation of quantum circuits on classical, sequential architectures.

In general, existing approaches can be divided into three categories:

- Efficient emulation of quantum algorithms from a prebuilt set of operations focused on time and/or hardware resources used, rather than reflecting physical behavior and universality of a quantum computer. Proposed designs included HDL libraries, often together with preprocessing software, as well as CPU-like solutions
- Emulating behavior of chosen physical quantum circuits. Those solutions were mostly focused on reflecting physics of a selected group of synthesized circuits, built from tools provided in HDL libraries, rather than constructing a processing unit with a universal set of instructions and able to execute any quantum algorithm
- Tools for designing and running quantum algorithms on classical architectures. Proposed tools are focused on providing the user the ability to write and run quantum algorithms on their classical machines, without focusing on emulating natural massive parallelism characteristic for quantum computers.

In our approach, we decided to design a universal architecture, providing a standard programming interface (set of instructions), while emulating physical quantum circuits in hardware. Our design for FPGA consists of modules responsible for emulation of parts of quantum circuits, such as quantum state, gates and measurement hardware, while being easily programmable by code sent from a connected PC (as shown in Fig. 1). Although in some cases it may not be the most efficient in terms of FPGA resources or emulator-PC data transfer time, proposed solution fully reflects mathematical representation of a quantum computer's behavior, is capable of running any quantum gates loaded from software while maintaining a computation's natural time complexity and is usable in a way similar to classical CPUs.

While many hardware–software systems for quantum computing emulation already exist, they are mostly focused on using software to utilize hardware more efficiently (as in [10–12,18]). In contrast, our approach decouples hardware from software, and the latter is only used to provide programming abstraction for the former. Our processor is designed to run entire quantum algorithms, including state preparation, evolution and measurement, completely in hardware, while software allows the programmer to focus on the algorithm, rather than the use of machine interfaces.

4 Theoretical background

To emulate quantum algorithms using classical hardware, we needed to decide which quantum computer phenomena we need to implement and how to do so. The goal of this section is to briefly introduce some of the most important ideas behind quantum computing, before describing our approach to reflect them in our solution.

4.1 Quantum versus classical information

The fundamental difference between quantum and classical computation lies in information representation. Classical information science uses bits to describe the world, where each bit represents a single value: 0 or 1. Deterministic information processing is, in essence, a manipulation of such binary values. Probabilistic computation introduces probabilistic bits, which return 0 or 1 with some probabilities p_0 and p_1, respectively. Classical probabilistic computing can be viewed as manipulating those probabilities for every output bit. Quantum information is stored in quantum bits, or qubits, which can make use of some unique phenomena only encountered in quantum physics. State of a single qubit can be described by a pair of complex numbers, as quantum state descriptions belong to some vector space over complex numbers with inner product of vectors (usually called Hilbert space).

4.2 Quantum bits

There are three main phenomena that are unique for quantum bits:

– Superposition—every qubit can be represented as a mixture of two base states, $|0\rangle$ and $|1\rangle$, with certain complex amplitudes. In other words, a qubit can be both 0 and 1 at the same time, but only one of those values will be returned when the qubit's value is checked (during an operation called measurement).
– Entanglement—multiple qubits can be entangled, forming a single coherent quantum state that can only be interfered with as a whole. Entangled qubits react to changes together, even if we interfere with just a single qubit and the whole system is spread across great distances (this is what Albert Einstein described as ”spooky action at a distance”).
– Interference—because entangled qubits form a coherent system, changes made to one of them causes shifts in amplitudes of all the others.

Utilizing all three of these in quantum information representation and processing allows us to achieve extreme, even exponential, speedups in computation.

4.3 Quantum circuit model

While a Turing machine is probably the most popular model of computation, another commonly used one is the circuit model. Information processing can be viewed as a series of operations performed by a set of gates on a group of parallel binary inputs, flowing through some paths (wires). A widely used model for quantum computing is an analogy to the latter and is called a quantum circuit (or quantum gate) model of computation. In this case, we envision algorithms as series of unitary transformations performed by quantum gates on some quantum state register, rather than qubits flowing through the gates via some paths (which would hardly reflect physical possibilities).

It should be noted that one of the requirements in quantum computing is reversibility of computation (which comes from laws of quantum mechanics). Because all quantum gates represent unitary operations, this is naturally fulfilled, so that for an output of any quantum gate we can determine what the input was [15].

In mathematical notation, we may represent a quantum state with a vector of complex values (amplitudes of all possible states), and quantum gates as unitary matrices of such. Therefore, computation may be viewed as series of matrix–vector multiplications.

5 Emulating quantum circuits

According to the Quantum Strong Church–Turing Thesis, only quantum computers can effectively simulate themselves. As every qubit can represent both 0 and 1 at the same time, n qubits are capable of carrying a state of 2^n numbers. This leads to an easy observation that emulating quantum systems using classical information representation requires exponentially more resources than it would on a quantum computer. Apart from clear memory restrictions (storing state of a system consisting of a few tens of qubits quickly becomes impractical, if not impossible), processing such a state is problematic itself.

Software emulators simply unroll quantum parallel transformations and perform them sequentially, in single steps. For every possible quantum state (and for n emulated qubits there are 2^n possible states), an amplitude must be computed. Of course, time complexity of such an approach makes it hardly possible to emulate transformation of as little as 30 qubits (which corresponds to series of multiplications of square matrices sized 2^{30} by 2^{30} and a complex vector of size 1 by 2^{30}). FPGA circuits allow us to parallelize this computation, which leads to massive performance improvements. However, we cannot run away from the Quantum Strong Church–Turing Thesis, and this speedup comes at a cost of exponential hardware resources.

Depending on the approach, there are designs parallelizing the whole process or just parts of it, which results in different balances between time and resource requirements. An often-overlooked part is the implementation of quantum measurement, which in reality also relates to exponential amount of operations performed simultaneously, as measurement of a single qubit may affect all other qubits in the state. Many designs chose to perform it sequentially to save limited hardware resources.

In our solution, every operation of the emulated quantum computer is represented by a single instruction in the processor, fully parallelized and performed in one clock-tick. The natural parallelism of quantum operations is, therefore, reflected in our design, at the cost of exponential hardware resources used. Because of the FPGA's size limitations, such a model will never compete with physical quantum computing systems. However, it might be very useful in analysing quantum algorithms, as well as understanding the quantum model of computation in general.

6 Hardware representation of qubits

As mentioned earlier, a single qubit can be represented by a pair of complex numbers. This corresponds to the fact that we need to store amplitudes for all possible base states of any given qubit. In quantum computing, we are usually interested in two base states—$|0\rangle$ and $|1\rangle$. Therefore, when emulating n qubits, we have 2^n possible base states of the system, which requires us to store $f2^n$ complex numbers.

6.1 Synthesizable real numbers

In our solution, we started by designing our own synthesizable definition of complex numbers in VHDL hardware description language. We decided to build it on top of IEEE VHDL Fixed-point Package [28]. The choice of fixed, rather than floating-point arithmetic came from two important factors:

- Both real and imaginary parts of a quantum state's amplitude are real numbers in the range [0, 1], so we only require one bit for the integer part
- A fixed-point arithmetic operations' hardware implementation is simple, compared to floating-point, and therefore much more resource efficient

Table 1 Quantum state representation in VHDL

$\alpha\|0\rangle + \beta\|1\rangle$
$\alpha, \beta \in \mathbb{C}$
	`subtype QReal is`
	`sfixed(INT_BIT downto FRAC_BIT);`
	
	`type QComplex is record`
	`  RE, IM : QReal;`
	`end record;`
	
	`type state_vector is`
	`array(QUBIT_INDEX downto 0)`
	`  of QComplex;`

Using the standard package, we defined real numbers as fixed-point numbers of parametrized constant size. To prevent additional growth of required hardware resources, we decided to fix the size of every real number represented in the system. Every arithmetic operation on one or more real numbers represented by n bits returns an n-bit result, rather than one resized to hold all possible outputs. While this approach might lead to precision loss, especially during multiplication, it is the only simple scalable option. Enabling scale adjustment was one of our main focuses, and resizing results would lead to gigantic number representations after a series of multiplications.

Our design is easy to adjust—a single parameter defines the number of bits used to represent every real number. Moreover, because of a modular approach, the entire system is based on interfaces. Therefore changing the representation of real numbers only requires code modifications in one place.

6.2 Complex numbers and quantum state representation

Complex numbers are defined as VHDL records, containing real and imaginary parts, both represented by real numbers of our implementation. All necessary operations, including, but not limited to, addition, multiplication, division, absolute value and square-root, were designed and implemented to suit our needs.

Quantum state is represented by arrays of complex numbers of our implementation. Similarly, quantum gates are defined as two-dimensional arrays of complex numbers of our implementation. The VHDL code for these types is presented in Table 1.

Like every part of our design, the size of state and gates is easily adjusted by modifying a single parameter representing the number of emulated qubits. All structures will be changed to specified sizes at compilation time. Quantum state is represented by arrays of 2^n complex numbers, while quantum gates are stored as arrays of $2^n \times 2^n$ complex numbers, where n is the parameter defining the number of emulated qubits.

6.3 Preparing initial state

Every quantum algorithm begins with an initial quantum state vector. Our solution enables this in two ways:

- Loading the desired initial state directly from software, as a vector of complex numbers
- Loading the initial state of all qubits set to $|0\rangle$ and then using quantum gates to prepare the desired state

The latter may sound more feasible in physical quantum computers, but we wanted to leave the possibility of representing quantum algorithms in our hardware like they are often described in theory - with some assumptions about the initial state. It should also be noted that in reality preparing a definite quantum state is really hard. Most of the time we cannot be entirely sure that measurement of a just-prepared qubit would return an expected value. We decided to remove that uncertainty from our system and simplify the initialization process.

6.4 Transforming states and entanglement

In order to avoid complex addressing while operating on selected qubits, in our approach every transformation is properly modified and applied to the entire state. This corresponds to the fact that leaving a qubit untouched is equivalent to transforming it with an identity gate. Therefore, for every transformation on any selected qubit or qubits, we can define a transformation for the whole state, such that all unaffected qubits are transformed through identity. We utilize this approach in our solution, and every transformation is performed as a multiplication of a matrix representing a gate for all qubits and the entire state vector.

The described method also allows us to easily achieve entanglement of any qubits in the represented state. Any entangling transformation will modify the whole state to reflect entanglement of desired qubits. For example, we can achieve a pair of entangled qubits within a 3-qubit state by using 2-qubit Hadamard (H) and Identity (I) gates, and a 3-qubit CNOT gate as presented in Eq. (1).

$$
\begin{aligned}
&|000\rangle \cdot I \otimes H \otimes I \rightarrow \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|010\rangle \\
&\quad \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|010\rangle \cdot I \otimes CNOT \rightarrow \\
&\quad \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|011\rangle. \qquad (1)
\end{aligned}
$$

In our processor, the same transformation would be represented by a series of matrix–vector multiplications, as shown in Eq. (2).

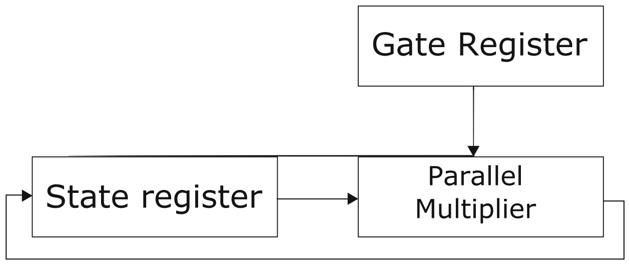

Fig. 2 Transformation data flow

$$
\begin{bmatrix}
\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\
\frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\
0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}}
\end{bmatrix}
\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\rightarrow
\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\rightarrow
\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \qquad (2)
$$

Transformations (or quantum gates) are applied to the state through multiplying selected gate's matrix and the state vector. The result replaces the input state vector in its register, to reflect physical operations on a quantum register (as shown in Fig. 2). The transformation module performs matrix multiplication as a single operation, in one clock-tick. Amplitudes of all possible states are considered and recomputed simultaneously, which mimics quantum parallelism.

For the transformation module to be easily scalable, we wanted a single parameter to change the entire structure. The hardware is, therefore, defined with nested VHDL parallel "for" loops (example shown in Table 2).

It is sufficient to change a constant defining the number of emulated qubits for the whole code to adapt.

6.5 State measurement

An often-overlooked problem in quantum circuits emulation is measurement complexity. Measuring a single qubit affects amplitudes of all possible states in a quantum system. In reality, the entire state is affected instantly, which requires

Table 2 Quantum state transformation in VHDL

```
for row in 0 to QUBIT_INDEX loop
  for col in 0 to QUBIT_INDEX loop
    temp_value(row) := temp_value(row) +
      (gate_in(row, col) * state_in(col));
  end loop;
  state_out(row) <= temp_value(row);
end loop;
```

exponential resources to emulate with classical devices (as we need to process 2^n amplitudes for n qubits.)

As states' amplitudes correspond to probabilities of measuring them, another important factor is random value generation, allowing to emulate qubit's behavior during measurement. To reflect nature, we provided a hardware circuit, which sets measured qubit's state to either 0 or 1 with regard to corresponding states' amplitudes.

Qubit measurement in our design is implemented as the Von Neumann measurement. The following procedure describes computational steps preformed by the hardware:

1. Probability of measuring 0 is computed based on the entire state
2. A pseudo-random real number of our implementation is generated
3. If the number from step 2. is greater than the probability from step 1., qubit's measured value is set to 1. Otherwise, the qubit's measured value is set to 0
4. Amplitudes of all impossible states (ones where selected qubit's value is different than measured) are set to 0
5. All amplitudes are normalized so that $\sum_i (amplitude_i)^2 = 1$

Just like transformations, this procedure is designed as a single operation, performed in one clock-tick. An example measurement for a 3-qubit system is shown in Fig. 3.

As measuring a single qubit changes values of amplitudes for the entire system, some information that was stored before measurement gets destroyed. Amplitudes of all states for which the value of the measured qubit was different from actually read are set to 0. Therefore, information stored in those amplitudes gets erased. This behavior represents real quantum system measurement, which also involves destruction of unread states.

Because of the randomized factor, the nature of computation in our system is probabilistic and separate measurements of the same quantum state may bring different results. The desired output may be destroyed during measurement, and actually read bit-sequence may be useless. Therefore, just like with an actual quantum machine, to get accurate information from quantum algorithm's result, it may be necessary

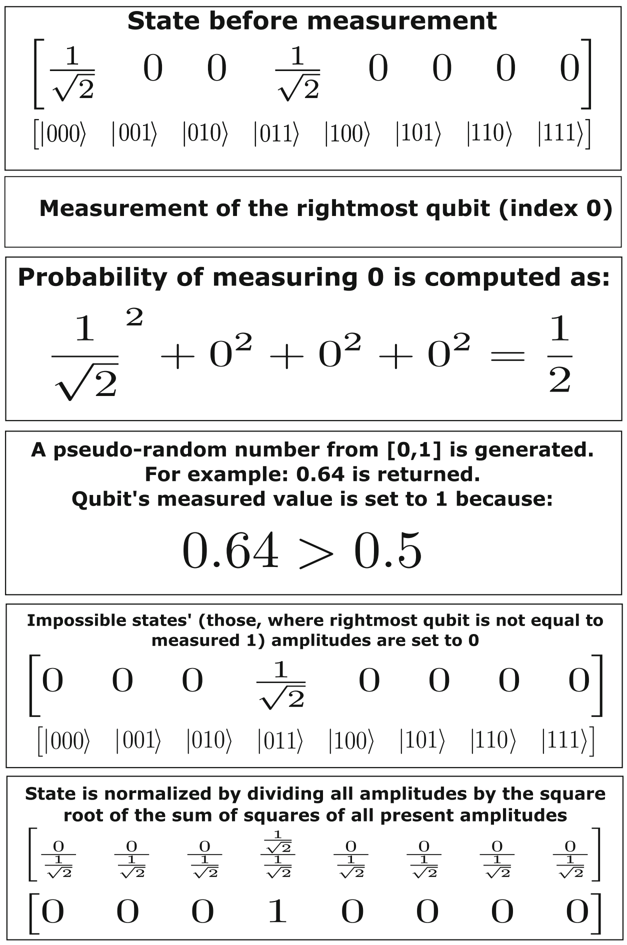
State before measurement
$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \end{bmatrix}$
$[|000\rangle \; |001\rangle \; |010\rangle \; |011\rangle \; |100\rangle \; |101\rangle \; |110\rangle \; |111\rangle]$
Measurement of the rightmost qubit (index 0)
Probability of measuring 0 is computed as:
$\frac{1}{\sqrt{2}}^2 + 0^2 + 0^2 + 0^2 = \frac{1}{2}$
A pseudo-random number from [0,1] is generated.
For example: 0.64 is returned.
Qubit's measured value is set to 1 because:
$0.64 > 0.5$
Impossible states' (those, where rightmost qubit is not equal to measured 1) amplitudes are set to 0
$\begin{bmatrix} 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \end{bmatrix}$
$[|000\rangle \; |001\rangle \; |010\rangle \; |011\rangle \; |100\rangle \; |101\rangle \; |110\rangle \; |111\rangle]$
State is normalized by dividing all amplitudes by the square root of the sum of squares of all present amplitudes
$\begin{bmatrix} \frac{0}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} & \frac{\frac{1}{\sqrt{2}}}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} & \frac{0}{\frac{1}{\sqrt{2}}} \end{bmatrix}$
$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$

Fig. 3 Steps of quantum measurement implementation, 3-qubit example

to run it multiple times and compare the returned values later.

7 Quantum system architecture

Our system is divided into hardware and software parts. Hardware, specified in VHDL, is designed to serve as a universal quantum computer, emulating a set of qubits, providing methods to manipulate the quantum state and exposing an instruction interface. Software, developed with C#, provides an abstraction layer, allowing an easy and efficient interaction with the quantum emulator.

The hardware part is designed to be very easily parametrized and scalable from code-level. As mentioned before, changing single parameters in VHDL can modify all hardware to emulate a desired number of qubits, or be more or less precise in number representation. Software is prepared to be aware of that setup, so that scaling hardware does not require any modifications in software, and any special tuning should be done through extension (creating new

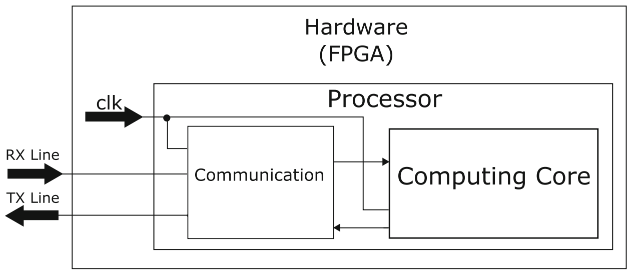

Fig. 4 Quantum processor emulator architecture

gates or result interpretation methods), rather than modification.

7.1 Architecture of the quantum processor emulator

In our VHDL CPU design, we decided to specify three separate blocks: computing core, communication and processor (as shown in Fig. 4).

The first contains hardware for emulating quantum processes, including computing and state measurement. The second one implements communication modules, allowing software running on PC to access hardware implemented on FPGA through a specified port. Last block provides an instruction interface to former blocks, allowing the programmer to interact with hardware in a standard way.

7.1.1 Computing core

The core consists of a state register, a transformation unit, gate memory and a measurement module. During initialization, the starting state is set in the state register and gates are loaded to the gate memory. Once that process is finished, the core is ready to operate. In a single cycle, a gate is loaded into the transformation unit, which then uses it to transform the state register. An instruction counter keeps track of which gate to use. Once all transformations from the memory have been applied to the state, the instruction counter sets a flag, marking that the computation has been finished and the result is ready to be read from the register.

The measurement module is independent of the transformation unit, as we decided it should not be obligatory to read the state once the computation is finished (state may just as well be omitted or destroyed). In our implementation, the processor module waits for the ready flag to be set by the computing core, to then run the measurement. A single qubit's measurement, which modifies amplitudes of the entire state, is also fully parallelized and takes one clock-tick to complete.

For reasons mentioned in previous sections, we need to generate some pseudo-random values to emulate qubit measurement. We used a simple linear-feedback shift register, which shifts with every clock-tick, rather than on request. This provides slightly better randomness, as returned values vary depending on time elapsed from system startup. Thanks to modular architecture, this can be easily replaced with any generator adjusted to return real numbers of our implementation.

7.1.2 Communication

In order to make our hardware usable on a higher level, we designed it to be used with software running on a connected PC. The communication module is responsible for receiving and sending data between hardware and software parts.

To keep our design simple, we implemented communications on top of simple UART transmitter/receiver elements. On the lowest level, data are sent byte-by-byte, with little-endian ordering. Using those procedures, we built abstraction layers allowing hardware to send and receive real and complex numbers of our implementation, as well as entire matrices representing gates or input state configurations.

The connection was built using USB on the PC side and two GPIO with one GND pins (for RX, TX and Zero reference) on the FPGA board. Because of speed limitations of serial port communications, in some cases the process of loading information to the core might take a significant part of the whole operating time. This was not our concern though, as we were focused on making the computation behave like one on a real quantum computer, rather than the entire system to be used in fast emulations.

7.1.3 Processor

The last hardware module encloses previous modules and drives all computation. It is designed as a finite state machine, working in a simple cycle:

1. Receive an instruction from the PC using the communication module
2. Recognize the instruction and send a proper code, confirming execution of a given instruction or informing that it was invalid
3. Run the desired operation using the computing core (for ex. load initial state, load gates, compute and measure results etc.)
4. Send results, if any, back to the PC
5. Send "operation complete" confirmation back to the PC

At the end of every instruction, a confirmation is sent to the software part. This allows for unified use of all instructions, regardless of their returning any results (such as computation and measurement) or not (like setting initial state). Software always waits for the final confirmation before taking next actions.

Table 3 Processor instructions

Code	Instruction	Description
0000	OK	Used for confirmation of various requests from the processor
0001	RECEIVE GATE	Signals the intention to load quantum gate matrices. The processor will expect gate data after receiving this instruction Every gate is received as a series of complex numbers, describing matrix values left-right, top-bottom. As gates should process the entire state (as described in Sect. 6.4), the processor expects a fixed size of the gate equal to 2^n by 2^n, where n is the number of emulated qubits After receiving a single gate, the processor sends it back to the user, and expects a confirmation the data is correct in order to minimize the risk of a transmission error. Confirmation is expressed with an instruction OK
0010	GATES FINISHED	Signals that no more gates should be expected after this instruction
0011	RECEIVE STATE	Signals the intention to load initial quantum state. Just like the gates, it is received as a series of complex numbers of our implementation, and sent back to verify its correctness
0100	COMPUTE	Signals the processor to execute the program stored in its memory. The gates will be applied to the state one-by-one, in the order in which they were received. The finished computation is signalled to the user with an OK response
0101	MEASURE	Signals the intention to receive the index of the qubit to be measured. The state is then measured, causing the chosen qubit to take one of the base values, 0 or 1 (which, of course, can affect the entire state)
0110	SEND RESULTS	Signals the processor to send the measured state back to the user. The state will be sent as a sequence of bits, one for every qubit measured. If MEASURE instruction was not received first, SEND_RESULTS will measure all qubits in the state before returning the result
0111	SEND SPECS	Signals the processor to send back its specifications. This includes number of emulated qubits, number precision in bits and the gate memory size
1000	RESET	Signals the processor to clear the gate memory, reset all qubits and get to idle mode
1111	ERROR	Signals the processor that there has been an error in transmission, or an unexpected operation has been performed. This will result in halting the current operation and returning to idle mode

7.2 Interacting with the processor

From the user's point of view, hardware can be treated as a quantum computer exposing a programming interface. There are nine instructions recognized by the processor, represented by 4-bit integers. The list of available instructions is listed in Table 3.

A computation could be, therefore, viewed as sending and receiving signals to and from the processor. An exemplary simple algorithm could be executed as follows:

1. Send RESET signal to guarantee that we operate on a fresh state.
2. Send SEND_SPECS signal to receive information about available resources. Based on this information we know how to prepare our gates.
3. Send RECEIVE_GATES signal.
4. Send data describing the gates we want to use. For example, if we wanted to use a 1-qubit gate for a single-qubit state, we would have to send 4 complex numbers corresponding to matrix indices [0, 0], [0, 1], [1, 0] and [1, 1].
5. Receive the data back from the processor, and verify it is equal to what we sent. If it is, send OK signal and continue. Otherwise, send ERROR signal and go back to step 3.
6. Send GATES_FINISHED signal to the processor.
7. If we want to set a specific initial state, we should send RECEIVE_STATE signal to the processor, and follow by sending appropriate data and confirmation.
8. Send COMPUTE signal. Await for OK response from the processor.
9. Send SEND_RESULTS signal to the processor to measure the entire state. Receive a string of bits equal in length to the number of qubits, representing the result of our computation.

The current implementation provides a communication module, described in Sect. 7.1.2, which allows for interaction with the processor through sending and receiving 8-bit integer data packages.

However, it should be noted that it is not obligatory to use our processor through UART. Thanks to the modular architecture, we can interact directly with the CPU, using its 4-bit instruction-set and appropriate interfaces for gate and state data input and output.

Therefore, our architecture is not only easy to interact with using any computer with UART communication capability, but also to use directly as a processor in other designs.

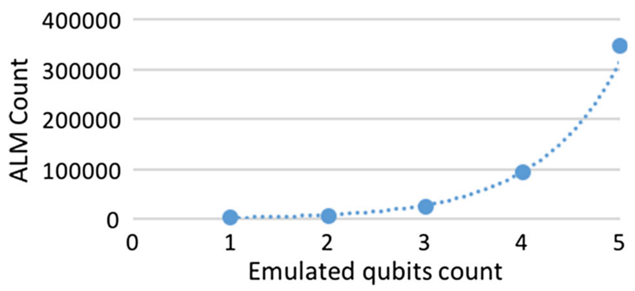

Fig. 5 Relation between the number of emulated qubits and required ALM count

7.3 Software part

While any software capable of sending integers through UART can be used to interact with our processor, we have designed a solution to interact with the core in a seamless way. Our goal was to allow the programmer to think in terms of quantum gates and states, rather than processor codes.

During initialization, our software requests information about emulated qubit count and precision of real numbers from the core, using the SEND_SPECS instruction (as described in Table 3). Based on those constants, the PC adjusts the information sent to the core.

Gates in our software are designed to scale to different qubit counts and number precisions, providing a useful abstraction that can be used regardless of the constants with which the core has been synthesized. The programmer can design desired gates by implementing our interfaces, and then run their algorithms on our processor directly from their code. A library of ready methods, including ones for preparing state, applying gates, measurement and interpreting results, allows for easy, high-level use of the powerful hardware architecture.

8 Example results

Our hardware performs quantum algorithms with the same time complexity as a real quantum computer, at the cost of exponential hardware resources. For that reason, every additional emulated qubit causes rapid growth of required FPGA size.

Emulating two qubits with our solution requires 8k adaptive logic modules (ALMs) after synthesis with Altera Quartus II software. However, to emulate three qubits, over 25k ALMs would be necessary. With an Altera Cyclone V FPGA chip used for testing purposes, we were capped at 18k ALMs and could only implement a 2-qubit variant in hardware (as shown in Fig. 5).

To test our solution, we decided to implement and run the Deutsch algorithm, using the entire created system. Software part was launched on a PC computer connected through serial port to an FPGA board, with our code synthesized to emulate a 2-qubit quantum computer.

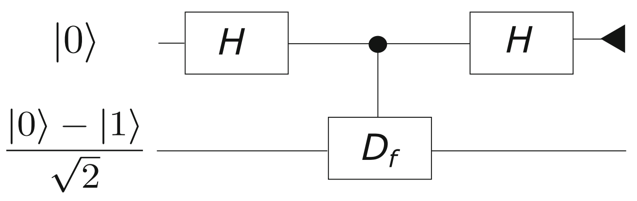

Fig. 6 Deutsch algorithm's quantum circuit

The Deutsch algorithm is a procedure proposed by David Deutsch in 1985 [16]. It is designed to check if some function $f(x)$ defined on $\{0, 1\} \rightarrow \{0, 1\}$ is constant (always returns 0 or always returns 1) or balanced (returns 0 for half of input arguments and 1 for the other half). This computation is, of course, easy to conduct on classical computers, as it requires just two evaluations of $f(x)$ (for 0 and 1) to provide the answer. However, Deutsch quantum algorithm needs to evaluate the tested function just once, giving a deterministic answer.

While the speedup may not be too impressive, the algorithm does show the potential of quantum computing and its parallel power, allowing us to check multiple inputs at once. In fact, the algorithm was improved by Deutsch and Jozsa in 1992 [17] to answer the same question but for $f(x)$ defined on $\{0, 1\}^n \rightarrow \{0, 1\}$. In that variant, we observe a superpolynomial speedup, compared to the classical solution, as the quantum algorithm requires just a single evaluation of $f(x)$ to produce the answer, compared to $\theta(2^n)$ evaluations needed by the classical deterministic algorithm.

Unfortunately, to run the Deutsch-Jozsa algorithm we would need to emulate at least three qubits, which was not possible due to our test hardware limitations.

The quantum routine from Deutsch algorithm can be presented by a quantum circuit shown in Fig. 6.
In the presented circuit, H (Hadamard gate) and D_f (Deutsch algorithm gate) represent quantum gates, which can be described by the unitary operators shown in Eq. (3).

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ H|1\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ D_f(|x\rangle|y\rangle) &\rightarrow |x\rangle|y \oplus f(x)\rangle \end{aligned} \tag{3}$$

where $\oplus$ represents exclusive alternative (XOR) operation. The triangle symbol at the end of the top circuit line in Fig. 6 symbolizes qubit measurement. It can be proven that for the inputs shown in Fig. 6, the measurement of the control qubit will always return 0 if $f(x)$ is constant, or 1 if $f(x)$ is balanced.

In our system, the Deutsch algorithm's circuit for $f(x) = 1$ is represented and computed as a multiplication of the gate matrices and the state vector, as shown in Eq. (4).

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}$$
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} \rightarrow \begin{bmatrix} -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \tag{4}$$
$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \rightarrow \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \end{bmatrix}$$

The multiplication results in the state vector are presented in Eq. (5):

$$\begin{bmatrix} \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$$
$$\frac{-1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle. \tag{5}$$

Only the states where control qubit (one to be measured) is set to 0 have non-zero amplitudes. Therefore, we can be sure that the measurement of the control qubit will always return 0.

To run the computation described above, in our software environment we prepared classes representing necessary transformations (2-qubit Hadamard⊗Idenity gate and Deutsch algorithm gate) and result interpreters. The code responsible for sending and receiving data through serial port is provided out-of-the-box. The procedure of running an algorithm in our system consists of the following steps:

1. A connection with the hardware core is established.
2. Hadamard⊗Identity and Deutsch gates are instantiated, and the initial state of $\frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|01\rangle$ (like shown in Fig. 6) is created as a byte array. Both gates and the initial state are sent to the core.
   (a) Every instruction sent to the core is confirmed with an OK reply code from the hardware.
   (b) All data sent to the core are afterwards sent back to the software part, which then confirms correctness or flags transmission error and retries the operation.
3. Instructions to perform the computation and send back the results are sent to the core. Received result is processed and saved.
4. Step 3. is repeated 20 times. Results are interpreted and returned to the user.

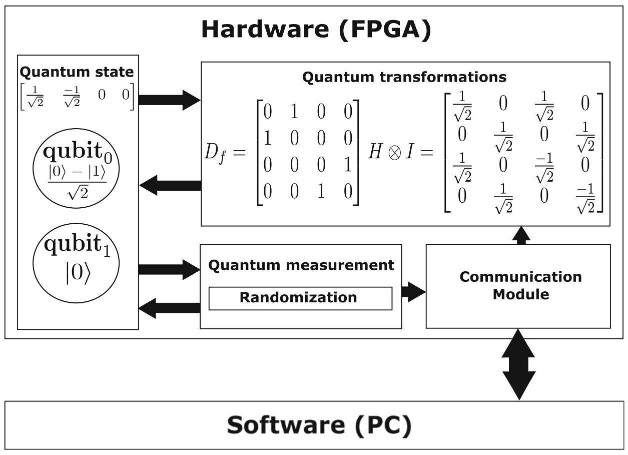
Hardware (FPGA)
Quantum state
$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$
qubit_0 $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$
qubit_1 $|0\rangle$
Quantum transformations
$D_f = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$H \otimes I = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{\sqrt{2}} \end{bmatrix}$
Quantum measurement
Randomization
Communication Module
Software (PC)

Fig. 7 Test-case quantum system overview for the Deutsch algorithm with $f(x) = 1$

Table 4 Results of 20 measurements for the Deutsch algorithm run for $f_1(x) = 1$ and $f_2(x) = x$

Measurement no.	$f_1(x) = 1$	$f_2(x) = x$
1	1(01b)	2(10b)
2	1(01b)	3(11b)
3	1(01b)	2(10b)
4	0(00b)	2(10b)
5	1(01b)	3(11b)
6	0(00b)	2(10b)
7	0(00b)	3(11b)
8	1(01b)	3(11b)
9	1(01b)	2(10b)
10	1(01b)	3(11b)
11	1(01b)	2(10b)
12	0(00b)	3(11b)
13	1(01b)	3(11b)
14	1(01b)	2(10b)
15	1(01b)	3(11b)
16	1(01b)	2(10b)
17	0(00b)	2(10b)
18	1(01b)	2(10b)
19	0(00b)	3(11b)
20	1(01b)	2(10b)

Figure 7 shows the overview of the synthesized test-case system just before the computation (after initialization).

We ran our algorithm for two functions: constant $f_1(x) = 1$ and balanced $f_2(x) = x$. The results returned by our system are presented in Table 4.

For Deutsch gate testing function $f_1(x) = 1$, the number 1 was returned 14 times and 0 was returned six times (meaning we measured states $|01\rangle$ and $|00\rangle$ 14 and six times, respectively). The software results interpreter correctly logged constant function. For Deutsch gate testing function $f_2(x) = x$, the number 2 was returned 11 times and 3 was returned nine times (meaning we measured states $|10\rangle$ and $|11\rangle$ 11 and nine times, respectively). The software results interpreter correctly logged balanced function.

The results also reveal randomness in the returned values, based on the pseudo-random number generator used in the module. This is an important observation, as randomness is one of the intrinsic traits of quantum computing.

9 Conclusions and future work

We have proposed, designed and implemented an easily scalable universal quantum computer emulator, focused on reflecting natural quantum processes in hardware, while maintaining the time complexity of quantum algorithms and exposing an instruction-set interface. As an exemplary use-case result, we have created a hardware–software system capable of running and correctly interpreting results of the Deutsch quantum algorithm.

The next steps for our solution include optimizing code, where possible, so that required hardware size would decrease by some constant, without affecting the clarity and scalability of the current design. One of the main areas of focus will be implementing a more efficient way of computation for quantum state evolution, perhaps utilizing proposals described in [26] or [27]. We also want to synthesize the code on a larger FPGA chip to emulate more qubits and run more sophisticated algorithms. More hardware resources would also allow us to change the number implementation from fixed to floating-point, which would benefit heavy numerical computations' accuracy.

To increase efficiency of a particular implementation of our design, at the cost of making it specific for a chosen FPGA, it is worth considering using DSPs explicitly to speed-up the computation.

Finally, supporting mixed states and implementing some state preparation and measurement imperfections could contribute towards the tool being more useful for real-world quantum computation emulations.

References

1. Kaye, P., Laflamme, R., Mosca, M.: An Introduction to Quantum Computing. Oxford University Press, Oxford (2006)
2. IBM Quantum Experience. https://quantumexperience.ng.bluemix.net/
3. Rigetti Quantum Forest. https://www.rigetti.com/
4. D-Wave Systems. https://www.dwavesys.com/
5. Finnila, A.B., Gomez, M.A., Sebenik, C., Stenson, C., Doll, J.D.: Quantum annealing: a new method for minimizing multidimensional functions. Chem. Phys. Lett. **219**(56), 343–348 (1994)
6. Gutierrez, E., Romero, S., Trenas, M., Zapata, E.: Quantum computer simulation using the cuda programming model. Comput. Phys. Commun. **181**(2), 283–300 (2010)
7. de Avila, A.B., Reiser, R.H.S., Yamin, A.C., Pilla, M.L.: Scalable quantum simulation by reductions and decompositions through the Id-operator. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16), ACM, pp. 1255–1257 (2016)
8. Sawerwain, M.: GPU-based parallel algorithms for transformations of quantum states expressed as vectors and density matrices. In: PPAM'11 Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics, vol. part I, pp. 215–224 (2011)
9. Avila, A., Maron, A., Reiser, R., Pilla, M., Yamin, A.: GPU-aware distributed quantum simulation. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14), ACM, pp. 860–865 (2014)
10. Khalid, A.U., Zilic, Z., Radecka, K.: FPGA emulation of quantum circuits. In: Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '04), pp. 310315. IEEE (2004)
11. Goto, Y., Fujishima, M.: Efficient quantum computing emulation system with unitary macro-operations. Jpn. J. Appl. Phys. **46**(4), 22782282 (2007)
12. Lee, Y.H., Khalil-Hani, M., Marsono, M.N.: An FPGA-based quantum computing emulation framework based on serial-parallel architecture. Int. J. Reconfigurable Comput. 2016, Article ID 5718124 (2016)
13. Frank, M.P., Oniciuc, L., Meyer-Baese, U.H., Chiorescu, I.: A space-efficient quantum computer simulator suitable for high-speed FPGA implementation. In: Proceedings of SPIE 7342, Quantum Information and Computation VII, 734203 (2009)
14. Patrzyk, J., Patrzyk, B., Rycerz, K., Bubak, M.: Towards a novel environment for simulation of quantum computing. Comput. Sci. **16**(1), 103–129 (2015)
15. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information, 10 Anniversary edn. Cambridge University Press, Cambridge (2010)
16. Deutsch, D.: Quantum theory, the church-turing principle and the universal quantum computer. Proc. R. Soc. London **A. 400**, 97 (1985)
17. Deutsch, D., Jozsa, R.: Rapid solutions of problems by quantum computation. Proc. R. Soc. London **A. 439**, 553 (1992)
18. Negovetic, G., Perkowski, M., Lukac, M., Buller, A.: Evolving quantum circuits and an FPGA based quantum computing emulator. In: Proceedings of International Workshop on Boolean Problems (2002)
19. Fujishima, M., Saito, K., Onouchi, M., Hoh, K.: High-speed processor for quantum-computing emulation and its applications. In: IEEE International Symposium on Circuits and Systems, pp. IV-884–IV-887 (2003)
20. Fujishima, M.: FPGA-based high-speed emulator of quantum computing. In: IEEE International Conference on Field-Programmable Technology (FPT), pp. 21–26 (2003)

21. Fujishima, M., Satio, K., Hoh, K.: 16-qubit quantum-computing emulation based on high-speed hardware architecture. Jpn. J. Appl. Phys. **42**, 2182–2184 (2003)
22. Fujishima, M., Inai, K., Kitasho, T., Hoh, K.: 75-qubit quantum computing emulator. In: Extended Abstracts of the 2003 International Conference on Solid State Devices and Materials, pp. 406–407 (2003)
23. Aminian, M., Saeedi, M., Zamani, M.S., Sedighi, M.: FPGA-based circuit model emulation of quantum algorithms. In: IEEE Computer Society Annual Symposium on VLSI, pp. 399–404 (2008)
24. Greve, D.: QDD—a Quantum Computer Emulation Library (2007). http://thegreves.com/david/QDD/qdd.html
25. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of 35th Annual Symposium on Foundations of Computer Science, pp. 124–134 (1994)
26. Nikahd, E., Houshmand, M., Zamani, M.S., Sedighi, M.: One-way quantum computer simulation. Int. J. Microprocess. Microsyst. **39**(3), 210–222 (2015)
27. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Improving gate-level simulation of quantum circuits. Quantum Inf. Process. **2**, 347–380 (2003)
28. Bishop, D.: VHDL Fixed-point Package in VHDL-2008 Support Library. https://github.com/FPHDL/fphdl/