Evolving quantum circuits and an FPGA-based Quantum Computing Emulator

GORAN NEGOVETIC, MAREK PERKOWSKI, MARTIN LUKAC, ANDRZEJ BULLER*

Portland Quantum Logic Group

*Department of Electrical and Computer Engineering, Portland State University, Portland, Oregon 97207-0751, 1900 SW Fourth Ave, mperkows@ee.pdx.edu, * Human Information Science Laboratories, Advanced Telecommunications Research Institute International, 2-2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan, buller@atr.co.jp*

1.0 The goal of this research

The goal of the PQLG group is to develop complete methodologies, software tools and circuits for quantum logic. Our interests are mainly in logic synthesis for quantum circuits and quantum system design [10]. Emulation of quantum circuits using standard reconfigurable FPGA technology and FPGA-based Evolvable Quantum Hardware, proposed here, are research areas not yet dealt with by other research groups. A parallel software simulator was presented in [13].

1.1 Quantum Logic and Gates

Unlike classical bits that are realized as electrical voltages or currents present on a wire, quantum logic operations manipulate qubits [12]. Qubits are microscopic entities such as a photon or atomic spin. Boolean quantities of 0 and 1 are represented by a pair of distinguishable different states of a qubit. These states can be a photon's horizontal or vertical polarization denoted by |1> or |2>, or an elementary partice's spin denoted by |#> or |∃> for spin up and spin down, respectively. After encoding these distinguishable quantities into Boolean constants, a common notation for qubit states is |0> and |1>.

Qubits exist in a linear superposition of states, and are characterized by a wavefunction ψ. As an example, it is possible to have light polarizations other than purely horizontal or vertical, such as slant 45° corresponding to the linear superposition of $\psi=½[\sqrt{2}|0>+\sqrt{2}|1>]$. In general, the notation for this superposition is $\alpha|0>+\beta|1>$. These intermediate states cannot be distinguished, rather a measurement will yield that the qubit is in one of the basis states, |0> or |1>. The probability that a measurement of a qubit yields state |0> is $|\alpha|^2$, and the probability is $|\beta|^2$ for state |1>. The absolute values are required since, in general, α and β are complex quantities.

Pairs of qubits are capable of representing four distinct Boolean states, |00>, |01>, |10> and |11>, as well as all possible superposition of the states. This property is known as "entanglement", and may be mathematically described using the Kronecker product operation ⊗ [5]. As an example, consider two qubits with $\psi_1=\alpha_1|0>+\beta_1|1>$ and $\psi_2=\alpha_2|0>+\beta_2|1>$. When the two qubits are considered to represent a state, that state ψ_{12} is the superposition of all possible combinations of the original qubit, where

$$\psi_{12}= \psi_1\otimes\psi_2 = \alpha_1\alpha_2|00> + \alpha_1\beta_2|01> + \alpha_2\beta_1|10> + \beta_1\beta_2|11>. \quad (1)$$

This property permits qubit states to grow dimensionly much faster than classical bits. In a classical system, *N* bits represents 2^N distinct states, whereas N qubits corresponds to a **superposition** of 2^N states.

In terms of logic operations, anything that changes a vector of qubit states can be considered as an operator. We can model this phenomena using the analogy of a "quantum circuit", where wires do not transmit signals corresponding to Boolean constants, but transmit pairs of complex values, α and β. Logic gates with this quantum circuit transform the complex values on their inputs to new set of values on their outputs. Quantum logic gates can be modeled as matrix operations. Typically the notation |0> and |1> are not present in the matrix formulation of the equations, only the probability amplitudes (i.e. α and β) are included; however, there are kept in Equation (2) for illustrative purposes.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} \alpha|0> \\ \beta|1> \end{bmatrix} = \begin{bmatrix} a\alpha|0> + b\beta|1> \\ c\alpha|0> + d\beta|1> \end{bmatrix} \quad (2)$$

Because the qubit probabilities must be preserved at the output of the quantum gate, it is noted that all matrices representing them are unitary, i.e. $U^*U=I$. An important unitary matrix property is that of full rank. This property implies that quantum gate matrix rows and columns are orthonormal. Therefore, past results from spectral methods for classic digital logic are directly applicable to quantum logic synthesis. Furthermore, since quantum logic gates are represented by unitary orthonormal matrices, they represent logically reversible gates. These observations mean that the single input/output quantum logic gates as represented in Equation (2) are rotation matrices characterized by some particular rotation angle θ, where, for example, $a=\cos\theta$, $b=\sin\theta$, $c=-\sin\theta$ and $d=\cos\theta$. With this viewpoint, it can be seen that there are, in fact, an infinite number of single input/output qubit gates. However, three

elementary gates can be used to generate any rotation [1]. These are the R, S, and T gates described in matrix notation by

$$R=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix} \qquad S=\begin{bmatrix}1 & 0\\ 0 & e^{j\theta}\end{bmatrix} \qquad T=\begin{bmatrix}e^{j\theta} & 0\\ 0 & e^{-j\theta}\end{bmatrix} \quad . \tag{3}$$

A quantum gate that is often mentioned in the literature is the quantum XOR gate. This gate allows inputs of |00> and |01> to appear unchanged at the outputs, but interchanges the pairs |10> and |11>. For example, consider the quantum XOR gate's operation for an input |10>.

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}0\\ 1\\ 1\\ 0\end{bmatrix}=\begin{bmatrix}0\\ 1\\ 0\\ 1\end{bmatrix} \tag{4}$$

In this example, the input is |10>=(0)|0>+(1)|1>+(1)|0>+(0)|1>, and the input vector is represented by the coefficients shown in parentheses. It is a significant fact that the unitary gates described by Equations (3) and (4) can realize any quantum logic function [1] (including standard binary). There are several strong similarities of quantum logic and classic digital circuit design using AND/XOR logic which are used in our research.

2. Standard Hardware Implementation of Quantum Circuit Models

2. 1. Advantages of Modern FPGAs

Since truly quantum computers are for very small functions at present and many research groups cannot afford even those, verification of most of the research in this area is done now using software-based quantum circuit simulators. Several such simulators have been developed and are available either in public domain or commercially. However, because of massive parallelism of quantum algorithms, only small circuits can be simulated because of an excessive time. We believe therefore that it will be beneficial for the research community to develop a fast hardware simulator (emulator) of quantum circuits. Our approach proposed here uses electrically reconfigurable Field Programmable Gate Arrays (FPGAs). In our preliminary research we developed a small FPGA-based quantum hardware simulator. Our goal is a further development of simulator technology and employing it to practical quantum circuits such as Quantum: Search, Factorization, Associative Memories, Spectral Memories and Neural Nets [4,6,7,9,11,14,15,16]. Below we will explain in a simplified way the main ideas of our simulator.

Quantum calculations obey the laws of quantum mechanics - quantum computation performs exponential amount of calculation in a polynomial amount of space and time. As observed by Feynman, this is the reason why simulating in current (classical) computer technologies even limited size circuits requires exponential amount of memory and processing time. Physicists need thus supercomputers to simulate quantum world. However, quantum circuit operation exhibits a large amount of regularity and micro-scale parallelism, which suggests that classical supercomputers may be not the best medium for quantum mechanics simulation, which is the same as the simulation of quantum logic circuits. This led us to an idea of using FPGA-based hardware for quantum circuit simulation. It is important to realize that this circuit can be only of a finite size to meet the hardware resources. However, today's FPGA offer very large amount of memory and flip- flops, as well as hard-wired arithmetic operations such as multipliers. Below we present work done so far and we discuss how it can be expanded. One of possible approaches is to build a special board with powerful new generation Xilinx chips. Other approach is to use the CBM FPGA-based computer of ATR and adopt the software developed for it [12].

2.2. Emulating Quantum Circuits in standard binary hardware

Quantum circuits are composed of quantum logic gates. There exist in theory an infinite number of elementary quantum gates, many such gates have been proposed; however, only a subset of these are required to implement an arbitrary size quantum circuit. Penalty for smaller set of gates is paid with a larger overall circuit.

As presented in previous section, the basic storage unit in a quantum computer is a qubit. A qubit can take value of zero, one, and a superposition of |0> and |1>. In the superposition state, complex amplitudes are used to represent probabilities of the qubit being in one of states, |0> and |1>. For a register of N bits, there are superpositions of 2^N states. This is the root of parallelism, since all possible states are calculated in parallel; this is also a fundamental difference of standard and quantum computing and is reflected in our simulator design. When the state of a qubit is observed, its probability collapses to either 0 or 1, hence the superposition is destroyed. This is another quantum peculiarity which calls for massive probabilistic circuits in a simulator. Also, since only finite precision can be employed for either software or hardware quantum circuit simulation, there is accumulated error that propagates through the system.

Quantum gate operations can be expressed as matrix operations. Each gate operation is expressed as multiplying 2^N-dimensional vector by the 2^N x 2^N transformation matrix. This makes a next requirement for the simulator – it is a matrix manipulator in hardware. Thus, techniques borrowed from DSP, image processing and other classical areas can be borrowed. Going from M x 2^N-dimensional Quantum vectors to one 2^{M+N} vector, we employ the Kronecker product (tensor product) matrix operation. Also, since in our prototype hardware simulator we will use only gates that operate on a small number of qubits (N = 1 or 2), these matrices are 2x2 or 4x4 matrices, respectively, replicated many times.

To simplify our presentation, most of the following text is based on an example. A simple three-gate quantum circuit is presented. Different issues and aspects of the hardware implementation are discussed subsequently. At the end, algorithms for conversion of quantum netlist to HDL description of hardware circuit are briefly outlined and open research and development issues are presented.

2.3. Example

2.3.1. Basics

Figure 1 below presents a simple quantum circuit with unitary quantum gates, inverter δ_x , and

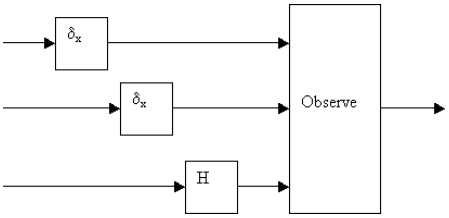

Fig.1

Hadamard H.

In matrix notation,

$$\delta_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \text{ and } H = 1/\sqrt{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The whole circuit before the observation gate can be represented in a matrix transform notation. The transform is calculated by Kronecker tensor product. The transform matrix QC is then: QC = $\delta_x \otimes \delta_x \otimes H$, where $\otimes$ denotes the Kronecker product.

Let

$$r = 1/\sqrt{2}$$

then

$$QC = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & r & r \\ 0 & 0 & 0 & 0 & 0 & 0 & r & -r \\ 0 & 0 & 0 & 0 & r & r & 0 & 0 \\ 0 & 0 & 0 & 0 & r & -r & 0 & 0 \\ 0 & 0 & r & r & 0 & 0 & 0 & 0 \\ 0 & 0 & r & -r & 0 & 0 & 0 & 0 \\ r & r & 0 & 0 & 0 & 0 & 0 & 0 \\ r & -r & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

How are matrix operations implemented in standard hardware?

This type of matrix operations is done in hardware via butterflies. Butterflies are known from DSP and image processing. However, in contrast, for quantum simulation, the butterfly structure is always the same, but operations used in the butterflies will differ. Elementary generalized butterfly structure is shown in Figure 2.

2.3.2. Model of Quantum Circuit as a network of butterflies

Single δx gate can be implemented as in Figure 3.

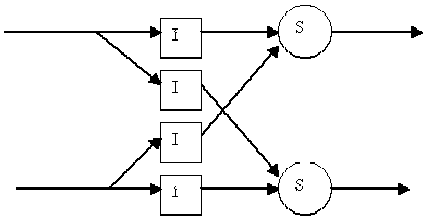

Fig.2

A single qubit requires two registers; one to store value α for |0> and another to store value β for |1>. Thus, δx requires four registers. It is a permuter. All quantum circuits that are also circuits of classical reversible (binary) logic are permuters. This simplifies the logic of the simulator. What about realizing in standard binary hardware the quantum circuit from Figure 4 ?

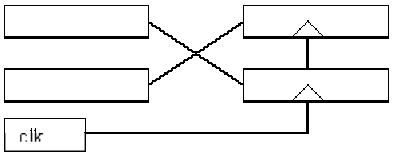

Fig.3.

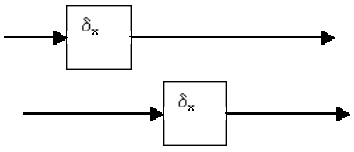

Fig.4.

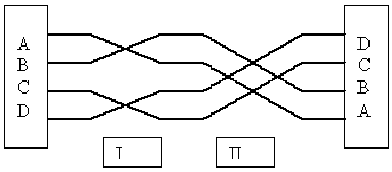

Fig.5

This piece of circuit means that there are four possible states the circuit can be in (2^N, N=2). Unlike bits in normal circuits, qubits are entangled (using matrices, this entanglement is achieved by taking the Kronecker product). This circuit is implemented using the δx butterflies, as presented in Figure 5.

Observation: Stages I and II are identical; they just operate on different numbers of channels (bits or groups of bits).

The following question arises: Is it possible to make a general butterfly, regardless of number of channels?

2.3.3. Generalized butterfly element

Two-by-two δx quantum gate is described as follows. For any input vector $[A, B]^T$ (where T is the matrix transpose), the output vector is $[B, A]^T$. By analogy, N-by-N δx quantum gate is described; for input vector $[(a_1, a_2,\ldots, a_N), (b_1, b_2,\ldots, b_N)]^T$, the output vector is $[(b_1, b_2,\ldots, b_N), (a_1, a_2,\ldots, a_N)]^T$, where 'N' is the number of channels of the δx gate. This gate can look as follows, Figure 6:

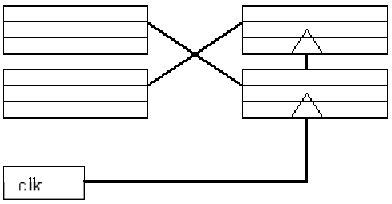

Fig.6.

The two-gate circuit can be build using only these types of δx gate. On the first level, though, only two channels are needed, and N-2 channels are wasted (where N is total number of channels). Direct hardware implementation would be very inefficient, wasting many registers (in the case of δx gate only registers, but in others, like Hadamard, multipliers and adders as well). However, consider the whole FPGA quantum simulator design flow: after the quantum netlist is transformed (synthesized) to the regular circuit design (HDL file), the subsequent steps are exactly the same as for any other circuit: logic synthesis, technology mapping, place and route, and target device programming. It is possible to remove unutilized resources (unused channels of the δx gate) in the next processing stages.

2.3.4. HDL Experiment

The following HDL Verilog file, Figure 7, is tested. The source represents a 2-channel δx gate. The first module is the gate itself, and the second module contains the gate module instantiation. Because one channel is unused, two inputs are replaced with "8'hx" which in Verilog means 8-bit wide wire with constant value of "unknown". The two outputs are 8-bit wires "tmp". These wires do not appear in the module input-output list (next to keyword "module"). The project is synthesized, and the log file is created. Synthesis tool used by us in our preliminary research was Synplify by Simplicity, (for the lack of spaces, several details will be not presented). On this log file, the two warnings appear in bold type. Synthesis tool realized the unused combinatorial logic (registers in this case) and removed it from the implementation. This is an important result. Even if the source code is very resource inefficient, if parts of codes are instantiated properly (with knowing what the tool will do), the tool will remove unused parts of the design leading to an efficient implementation.

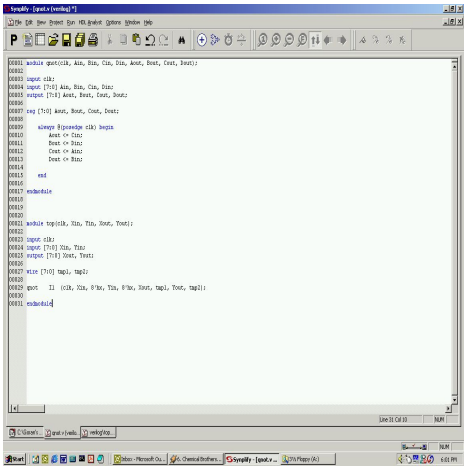

Fig.7

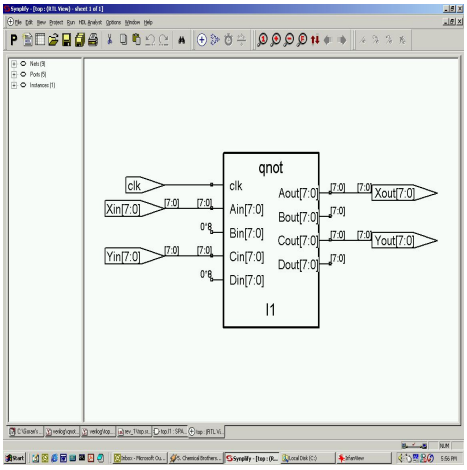

Fig.8

The following figures, Figure 8, Figure 9, show parts of the hardware implementation of the HDL source.

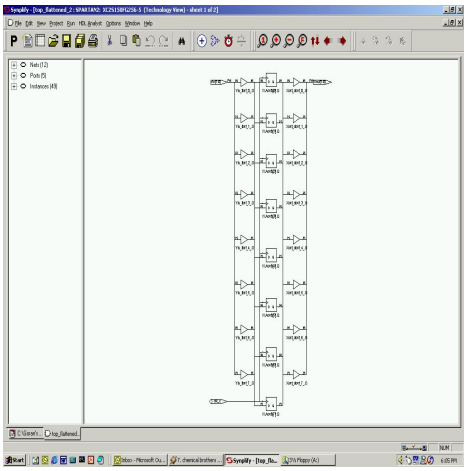

Fig.9

Next, the example circuit will be implemented with these findings in mind.

2.3.5 Example circuit implementation

The example circuit consists of two quantum inverters (δx gates) and one Hadamard gate (Figure 4). Direct matrix multiplication looks as follows:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & r & r \\ 0 & 0 & 0 & 0 & 0 & 0 & r & -r \\ 0 & 0 & 0 & 0 & r & r & 0 & 0 \\ 0 & 0 & 0 & 0 & r & -r & 0 & 0 \\ 0 & 0 & r & r & 0 & 0 & 0 & 0 \\ 0 & 0 & r & -r & 0 & 0 & 0 & 0 \\ r & r & 0 & 0 & 0 & 0 & 0 & 0 \\ r & -r & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = r * \begin{bmatrix} g+h \\ g-h \\ e+f \\ e-f \\ c+d \\ c-d \\ a+b \\ a-b \end{bmatrix}$$

where $r = 1/\sqrt{2}$

In general, a, b,…,h are complex numbers. The schematics of the hardware circuit is shown in Figure 10. Aside from needing gates of sufficient number of inputs and outputs, it is also required to mix the bits between the gates. This mixing is always ordered, for example bit reversal or logic shift of bits. Bit shifting is already used for many algorithms, like Fast Fourier Transform. On the other hand, can this be simplified? Using the knowledge about unused logic removal, consider the two-gate circuit seen before. It is implemented in hardware and presented as a file, similarly as in Figure 13. Even though the circuit looks wasteful on the schematics, its hardware realization is efficient. This is the main principle we will use for the FPGA based quantum circuit simulation.

2.3.6. Observation Gate

In the quantum circuit, states of the bits are superposed. This means that the circuit is in many states at the same time. However, once the circuit state is observed, the uncertainty is destroyed, and all bits collapse to the familiar '0' and '1' state (this and other quantum circuit properties are coming from the rules of quantum physics). In the simulation circuit, there must be a gate to simulate this behavior: from all superimposed circuit states, only one will be valid after this gate; the state is chosen based on probabilities. Output state is then: $|\alpha_1> A_1 + |\alpha_2> A_2 + \ldots + |\alpha_2{}^n> A_2{}^n$, where A_1, A_2, … are n-bit binary states, α_1, α_2, … are square roots of probabilities of the output being in that state, and from probability theory, magnitude of probability is equal to one.

In order to build a hardware circuit to simulate the "observe" gate, a random number generator is needed. A circuit often used for pseudo-random number generation in hardware is LFSR (Linear Feedback Shift Register) circuit. We can use LFSR theory and tables to calculate such circuits very efficiently and insert their generators to our CAD software package for designing automatically the reconfigurable quantum simulators (for lack of space this topic is not presented).

How to set bits based on probabilities? LFSR provides ½ probability. If two LFSR are uncorrelated, ANDing them will give ¼ probability. Hence, logic AND multiples probabilities: A ⊗ B = A * B (where ⊕ is logic AND, and * arithmetic operator). Similarly, logic OR and XOR operators give following probabilistic (arithmetic) operations: A '+' B = A + B – A*B (logic OR), and A ⊕ B = A + B – 2*A*B (logic XOR). Using these operators on the probabilities available, all probabilities can be derived. However, this approach is not feasible to implement for large number of probabilities. Larger set of probabilities can be achieved either by some other hardware method, or offline, in software.

One way to implement the quantum observation gate would be by using RAM memory and LFSR's. For N Q-bits, there are 2^N possible quantum states the observed quantum wires (registers) could be in at a time. If there is a RAM memory with 2^N locations, N address lines are needed to encode all memory locations. Filling the RAM with appropriate values, randomly generated address would choose the output value of the observation gate. For instance, out example circuit has 3 input/outputs. There $2^3 = 8$ possible output states (000, 001, 010… 111), each associated with a certain probability (based on input

values and circuit content). Lets say our input was [010]. In vector notation, input vector is [00100000]; this can be interpreted as probability of zero for [000], zero for [001], one for [010], zero for [011], etc.

Matlab function represents the example quantum circuit in matrix notation

```
function OutVector = qcir(InVector)  %Gate definitions
r = 1/sqrt(2);

H = [r -r; r r];
 inv = [0 1;1 0];
[sizeX, sizeY] = size(InVector);
 if(([sizeX, sizeY] == [1 8]) ~= [0 1]),
  error('Invalid Input Vector Size!'); end
%Circuit definitions
Transform = kron(H, kron(inv, inv));
OutVector = (Transform * InVector').^2;
% expressed as probability
```

Running the function with the input vector, we get output vector: [0, 0.5, 0, 0, 0, 0.5, 0, 0] – expressed in probability. It means that output [001] and [101] both occur with ½ probability (50% of time).

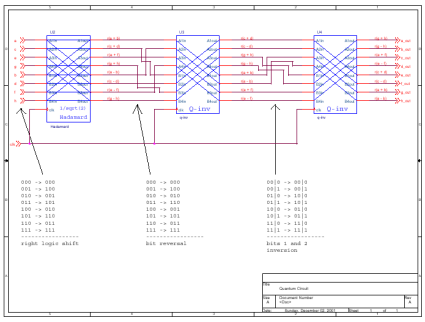

Fig.10

How to implement this in hardware? We need a RAM with 8 addresses (to account for each possible state). It has to be filled with states according to the probability. Figure 11. (Note: ordering of the RAM content is not important, since it is chosen randomly.) Address is randomly generated (with equal probability of all states). For this case, three bits are needed to encode 8 addresses; so three LFSR outputs are needed (these should be uncorrelated). The RAM content is based on the inputs, so it has to be filled dynamically (meaning, it cannot be pre-compiled). However, the structure of the RAM content is preserved regardless on the inputs. For example, for the circuit examined, all possible inputs would give only two possible outputs with ½ probability each. This observation can lead to somewhat simpler hardware implementation – Fig. 12.

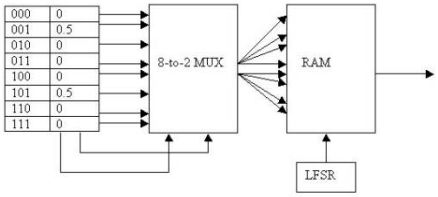

Fig.12

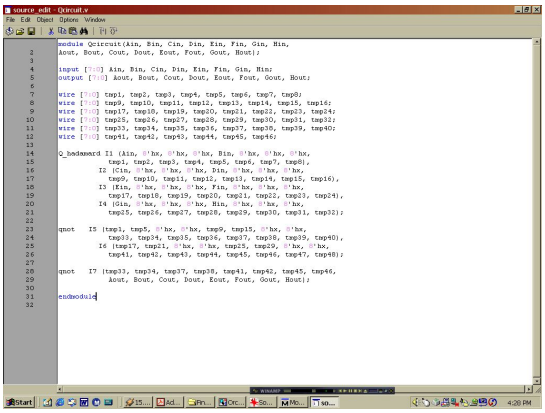

Fig 13

There are many formats for net-list representation.

Address	State
3 1	Output 001
2	001
3	001
4	001
5	101
6	101
7	101
8	101

Figure 11

Most often, a netlist first defines all components in the design, naming them with unique reference designators, and than lists all nets with their unique names and lists of component and pin names they go

to. For a quantum circuit, in addition to this, there can be a time slice where a gate or sub-circuit belongs. This is true because in a quantum circuit operation on qubits are done in parallel, such that in one time slice the qubits are updated even if a particular qubit does not go trough a gate. The reason for this is that qubits have values of probabilities of being a particular state, so changing a probability of one qubit, affects probabilities of other qubit states (due to probability theory).

There are many technical question that we skip here; quantum circuit netlist representation, quantum netlist to Verilog conversion, and others. We developed CAD tools for them which will be further extended and improved, possibly using the CBM tools developed in ATR Japan [2]. The technical questions solved and to be solved are similar as in standard CAD tools for translation from high level languages (Verilog, VHDL and higher) to logic descriptions, using libraries and optimizing designs on logic and register-transfer levels. Verilog source created with our current (not totally optimized) algorithm is shown in Figure 13. Concluding, we can tell that our tool is a ***dynamic preprocessor*** to convert quantum netlist to a standard Verilog FPGA-based design system.

3. Extrinsic and Intrinsic Evolvable Quantum Hardware (EQHW)

There are different views on what is evolvable hardware (EHW). For instance, A. Hirst defines EHW as *"applications of evolutionary techniques to circuit synthesis."* This is a very broad definition that includes any evolutionary software applied to any kind of synthesis or optimization in circuit design. Using this definition in quantum domain we obtain the following definition ***"Evolvable Quantum Hardware (EQHW) is application of evolutionary techniques to quantum circuit synthesis".*** Example of such an approach realized in software (extrinsic approach) is presented in Ref. [10]. T. Higuchi et al define EHW as *"hardware that is capable of on-line adaptation through reconfiguring its architecture dynamically and autonomously".* In quantum domain this definition would correspond to any kind of learning and adapting quantum circuits [4,7,9,11,14,16], but evolutionary algorithms would be not required. Finally, Hugo De Garis defines EHW *as "a Genetic Algorithm realized in hardware".* This narrowest definition is called Intrinsic Evolvable Hardware, in contrast to extrinsic EHW in which the entire learning is performed in software. Following this definition, ***"Evolvable Quantum Hardware is a Genetic Algorithm realized in quantum hardware."***

Observe first, that with present technology Evolvable Quantum Hardware is still a speculation since programmable quantum hardware is available. It is possible, however, to develop evolvable technology for ***FPGA models*** of quantum circuits, as presented in the previous sections. Using this understanding, the combination of our evolutionary software for designing automatically quantum circuit netlists [10] and the presented above method of building FPGA accelerators for quantum netlists, leads to the realization of a complete ***quantum-system-modeled-in-a-binary-system*** evolvable hardware, based on the available standard FPGA technology. In this proposed by us approach, the GA-based software program will generate a new netlist satisfying problem constraints. Next this netlist is translated to Verilog as above, and finally Verilog-based CAD tools are used for mapping to FPGA hardware. The difference between extrinsic and intrinsic hardware is whether the fitness function is calculated in software or in hardware. If calculated in software, then the approach would be extrinsic. If calculated in hardware and the process of GA-based net-list generation is repeated for populations, this would be an example of software-hardware extrinsic-intrinsic approach to quantum evolvable hardware. Another approach would be to realize the QC-generating GA entirely in hardware, where it would directly configure RAM bits that program logic and connections of the model. This works well with reconfigurable hardware such as FPGAs from Xilinx "http://www.xilinx.com/". One argument for the direct approach is to exploit emulator hardware resources for unconstrained hardware evolution. Such work is as an attempt towards adaptive hardware, not necessarily following the human-provided circuit description (like unitary matrix of a truth table) as a design specification for GA [10]. The difference would be here the same as between the binary logic function synthesis and data mining. In logic synthesis the truth table must be strictly followed and a solution candidate with even one minterm not verified in fitness function is bad. In contrast, in data mining the fitness function attempts to satisfy predominantly the Occam's Razor and to minimize the learning error so that solution that differ from the specification are acceptable. Such adaptive quantum learning will explore larger design spaces and thus may be able to discover novel designs for which unitary matrices are even not known but some other constraints are given. It will not assume a priori knowledge and thus can be applied to various domains and obtain feedback from the environment rather than from the human designer. By not requiring exact problem specifications complex systems can be possibly designed, which cannot be

handled by conventional specification-based evolutionary and non-evolutionary (such as backtracking or A* search [17,18]) design approaches to quantum circuit synthesis.

4. Conclusions and future work

We presented ***two new ideas***: (1) design of an FPGA-based hardware accelerator for quantum computing, (2) design of an FPGA-based hardware evolver of quantum computers. These approaches can work together. We did not discuss the sizes of quantum circuits that can be simulated, because the reported work was for single Xilinx chip only. In future, we will design the board with many chips and this board will be linked to a PC development workstation. The problem of building the board with chips, selection of best chips, how many boards - are all technical questions that will be addressed in future research. It is possible that after more detailed analysis we will come to the conclusion that instead of building the entire simulator from scratch we should purchase one of existing FPGA-based programmable emulators and only develop a new software for it, or that we should use CBM.

Although quantum computers are still in early research phase, the quantum accelerator and the quantum evolver, when build, can be directly used for practical tasks; such as simulation of quantum search, cryptography or robot control. They can be also used in research on quantum computing and for simulation of tough problems in quantum mechanics.

Acknowledgment. We appreciate help of Mitch Thornton, Jerry Bruce, Jong-Hwan Kim, Igor Markov, Alan Mishchenko, George Lendaris, and Mikhail Pivtoraiko.

BIBLIOGRAPHY

1. A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Schor, T. Sleator, J. Smolin and H. Weinfurter, Elementary Gates for Quantum Computation, *Physical Rev* (A), no. 52, pp. 3457-3467, March 1995.
2. H. de Garis, A. Buller, T. Dob, J. Honlet, P. Guttikonda, and D. Decesare, Building Multimodule Systems with Unlimited Evolvable Capacities from Modules with Limited Evolvable Capacities (MECs), *Proceedings of The Second NASA / DoD Workshop on Evolvable Hardware*, July 13-15, Palo Alto, California, pp. 225-234, 2000.
3. E. Fredkin and T. Toffoli, Conservative Logic, *Intern. J. Theor.l Physics,* vol. 21, Nos. 3-4, pp. 219-253, 1982.
4. Y. Z. Ge, L. T. Watson, and E. G. Collins. Genetic algorithms for optimization on a quantum computer. In *Unconventional Models of Computation*, pp. 218-227.
5. A. Graham, *Kronecker Products and Matrix Calculus With Applications*, Ellis Horwood Limited, Chichester, U.K., 1981.
6. L.K. Grover, A Framework for Fast Quantum Mechanical Algorithms, *ACM Symposium on Theory of Computing (STOC),* 1998.
7. Kuk-Hyun Han, Kui-Hong Park, Ci-Ho Lee, and Jong-Hwan Kim, Parallel quantum-inspired genetic algorithm for combinatorial optimization problems, *Proc.2001 Congress on Evolutionary Computation,* volume 2, pp. 1422-1429, 2001.
8. J. P. Hayes and I. Markov, Principle Investigators, Quantum Approaches to Logic Circuit Synthesis and Testing, http://vlsicad.eecs.umich.edu/Quantum/summary.html
9. T. Hogg, Solving Highly Constrained Search Problems with Quantum Computers*, J. Artificial Intelligence Research,* Vol. 10, 1999, pp. 39–66; http://www.jair.org/ abstracts/hogg99a.html.
10. M. Lukac and M. Perkowski, Evolving Quantum Circuits Using Genetic Algorithm, *Evolvable Hardware* 2002, accepted.
11. A. Narayanan and M. Moore, Quantum-Inspired Genetic Algorithms, *Proc. IEEE Int'l Conf. Evolutionary Computing,* IEEE Press, Piscataway, N.J., 1996, pp. 61–66.
12. M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge Univ. Press, September 2000.
13. K.M. Obanland, and A.M. Despain, *A Parallel Quantum Computer Simulator.*
14. M. Perus, Neuro-Quantum Parallelism in Brain-Mind and Computers*, Informatica* 20, pp. 173-183, 1996.
15. P.W. Shor, Algorithms for Quantum Computation: Discrete Logarithms and Factoring, *Proc. 35th Symp. Found. of Comp. Sci.,* IEEE Computer Soc. Press, Los Alamitos, Calif., pp. 124–134, 1994.
16. D. Ventura, Implementing Competitive Learning in a Quantum System, *Proc. Int'l Joint Conf. Neural Networks*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1999.
17. A. Khlopotine, M. Perkowski and P. Kerntopf, Reversible Logic Synthesis by Gate Composition, accepted to *IWLS 2002.*
18. V.V. Shende, A. K. Prasad, I.L. Markov, and J.P. Hayes, Synthesis of Optimal Reversible Logic Circuits, accepted to *IWLS 2002*.