A Fast Quantum Computer Simulator Based on Register Reordering***,****

Masaki NAKANISHI†(a), Member, Miki MATSUYAMA†*, and Yumi YOKOO†**, Nonmembers

SUMMARY Quantum computer simulators play an important role when we evaluate quantum algorithms. Quantum computation can be regarded as parallel computation in some sense, and thus, it is suitable to implement a simulator on hardware that can process a lot of operations in parallel. In this paper, we propose a hardware quantum computer simulator. The proposed simulator is based on the register reordering method that shifts and swaps registers containing probability amplitudes so that the probability amplitudes of target basis states can be quickly selected. This reduces the number of large multiplexers and improves clock frequency. We implemented the simulator on an FPGA. Experiments show that the proposed simulator has scalability in terms of the number of quantum bits, and can simulate quantum algorithms faster than software simulators.

key words: quantum computer simulator, hardware implementation, register reordering

1. Introduction

Development of quantum algorithms is a difficult task and sometimes needs analysis based on simulation as well as theoretical analysis. For this purpose, simulation of quantum computers is highly demanded. Simulation of quantum computers is a time consuming task, and many simulation methods have been investigated intensively [1]–[16]. Some of them implement the simulator on an LSI [1], [5], [8], [11]. Hardware implementation is considered to be suitable for tasks that run in parallel. Quantum computers can process many computation paths in parallel, which is called quantum parallelism. Thus, it is expected that hardware implementation of quantum computer simulators has an advantage. Quantum algorithms are often represented by quantum circuits. In [8], an FPGA-based quantum circuit emulator is proposed. For each quantum gate, the corresponding classical sub-circuit that emulates it is implemented on the emulator. By reconfiguring the classical sub-circuit according to the quantum circuit given as an input, they constructed an efficient emulator circuit. However, the area needed to implement all the quantum gates increases in proportion to the size of the given quantum circuit. Therefore, the size of quantum circuits that can be simulated is limited. Indeed, in their experiments, the number of quantum bits of the target quantum circuits is 3. In [1], it is shown that by adding several quantum circuit primitives to those in [8], a lot of quantum algorithms can be simulated quickly and the size of the emulator circuit can be decreased. However, the number of quantum bits and quantum gates that can be simulated are still limited. In [3], [4], a space efficient quantum computer simulator is proposed. This is based on the fact that BQP is included in PSPACE, where BQP is the class of decision problems that can be solved by quantum Turing machines in polynomial time with bounded error, and PSPACE is the class of decision problems that can be solved by classical Turing machines in polynomial space. The software implementation of the simulator is evaluated and it is shown that their method decreases memory usage. Although FPGA implementation approaches are also discussed, the actual implementation remained as future work. In [5], an FPGA-based quantum computer emulator is proposed. This emulator is specific to Grover’s algorithm, which is also called a quantum search algorithm. The emulator does not trace the quantum algorithm, but executes binary search instead. This makes the emulator faster. However, the time evolution of quantum state vector cannot be traced with the emulator. In [11], fractal inter-connection networks are proposed. When a quantum circuit simulator is implemented on a hardware device, inter-connection networks often become a bottleneck. The fractal inter-connection networks are used to avoid such a bottleneck and to improve performance.

In this paper, we propose a hardware quantum computer simulator architecture. The size of logic circuits of the proposed architecture does not depend on the size of the quantum circuit to be simulated.**** Thus, the proposed architecture is scalable in contrast to the hardware quantum circuit simulators in [1], [8], [11]. Also, the proposed architecture does not use complex inter-connection networks, but uses register reordering in order to select necessary register values. Register reordering can be done by iterating shift-and-swap operations of register values, and thus, can be executed at a high clock frequency. Additionally, this has another advantage; the proposed architecture does not need massive multiplexers to select operands, and can improve the size of data memory depends on the number of quantum bits (i.e., the width of a quantum circuit to be simulated). This is inevitable since the size of the description of a quantum state increases in terms of the number of quantum bits.

***A preliminary version of this paper was presented at the 22nd IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2014).

****The size of data memory depends on the number of quantum bits (i.e., the width of a quantum circuit to be simulated). This is inevitable since the size of the description of a quantum state increases in terms of the number of quantum bits.

Manuscript received July 2, 2015.
Manuscript revised October 16, 2015.
Manuscript publicized November 19, 2015.
The authors are with the Faculty of Education, Art and Science, Yamagata University, Yamagata-shi, 990–8560 Japan.
Presently with Sakata City.
**Presently with Kirayaka Bank Ltd.
†A preliminary version of this paper was presented at the 22nd IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2014).
†The work was supported by JSPS KAKENHI Grant Numbers 24500003 and 24106009, and also by the Asahi Glass Foundation.
E-mail: masaki@cs.e.yamagata-u.ac.jp
DOI: 10.1587/transinf.2015EDP7260

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers
area efficiency.

The proposed simulator executes quantum circuit simulation by tracing time evolution of a state vector. Thus, the output of the simulator is the complete information of the final state vector. The proposed simulator can simulate a universal quantum gate set, and thus, can simulate any quantum algorithm. These are advantages over [5].

The rest of the paper is organized as follows. In Sect. 2, we give several basic notions of quantum computing. In Sect. 3, we define quantum circuit simulations. In Sect. 4, we describe the proposed simulator architecture. In Sect. 5, we describe the algorithm of register reordering. In Sect. 6, we discuss experimental results, and Sect. 7 concludes the paper.

2. Quantum Computing

2.1 Quantum Bits

In quantum computing, information is stored in quantum bits (or qubits), which are quantum analogue of classical bits. The qubit whose value is 0 (resp. 1) is described by a two-dimensional vector $|0\rangle = (1, 0)^T$ (resp. $|1\rangle = (0, 1)^T$). In general, a qubit can be in a superposition of $|0\rangle$ and $|1\rangle$, which is described by a two-dimensional complex vector $(a, b)^T$ whose norm is one, i.e., $|a|^2 + |b|^2 = 1$. For a qubit in a state $(a, b)^T$, we say that the qubit is in a superposition of $|0\rangle$ and $|1\rangle$ with probability amplitudes $a$ and $b$, respectively. For an $n$-qubit system, we consider $2^n$ orthonormal basis vectors, $|00\rangle, \ldots, |11\rangle$, each of which corresponds to one of $n$-bit values from 00 to 11. Then, a superposition of an $n$-qubit system is described by a $2^n$-dimensional complex vector, $a_{00,0} |00\rangle + \ldots + a_{11,1} |11\rangle = (a_{00,0}, \ldots, a_{11,1})^T$ whose norm is one. The vector is called a state vector. We number the bits from right to left starting from 0 to $n - 1$, i.e., the LSB is the 0-th bit and the MSB is the $(n - 1)$-th bit.

Note that the size of state vectors increases exponentially in terms of the number of qubits. This is why simulation of quantum computers is a time consuming task.

2.2 Quantum Gate Operations

A one-qubit gate operation is described by a $2 \times 2$ unitary matrix. Examples of one-qubit gate operations are the following:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

where $X$ and $H$ are called a NOT-operation and an Hadamard-operation, respectively. Applying a gate operation to a qubit is formulated by applying the corresponding unitary matrix to the state vector. For example, the resulting state vector after applying a NOT-operation to $|0\rangle$ is obtained by

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle,$$

which inverts the value of the qubit.

Equivalently, we may describe a gate operation by specifying the transformation of each basis state such as:

$$X: \begin{cases} |0\rangle \rightarrow |1\rangle \\ |1\rangle \rightarrow |0\rangle \end{cases}, \quad H: \begin{cases} |0\rangle \rightarrow \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \\ |1\rangle \rightarrow \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \end{cases}. $$

An $m$-qubit gate operation is described by a $2^m \times 2^m$ unitary matrix. An example of a two-qubit gate is a Controlled-NOT (CNOT) gate. A CNOT gate has a control bit and a target bit, and the target bit is inverted if and only if the control bit is 1. The formal definition of a CNOT gate is as follows.

$$\text{CNOT}: \begin{cases} |00\rangle \rightarrow |00\rangle \\ |01\rangle \rightarrow |01\rangle \\ |10\rangle \rightarrow |11\rangle \\ |11\rangle \rightarrow |10\rangle \end{cases}$$

where the control bit is on the 1st bit and the target bit is on the 0th bit. It is known that the set of all one-qubit gates and a CNOT gate is universal [17]. A generalization of a CNOT gate is a controlled-unitary gate. A controlled-unitary gate has an associated unitary matrix, which is applied to the target bit if and only if the control bit is 1. The formal definition of a controlled-unitary gate is as follows.

$$\text{Controlled-U}: \begin{cases} |00\rangle \rightarrow |00\rangle \\ |01\rangle \rightarrow |01\rangle \\ |10\rangle \rightarrow a|10\rangle + c|11\rangle \\ |11\rangle \rightarrow b|10\rangle + d|11\rangle \end{cases}$$

where the control bit is on the 1st bit, the target bit is on the 0th bit, and the associated unitary matrix is $U = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

Another example of a two-qubit gate is a SWAP gate. A SWAP gate swaps the values of the two qubits:

$$\text{SWAP}: \begin{cases} |00\rangle \rightarrow |00\rangle \\ |01\rangle \rightarrow |10\rangle \\ |10\rangle \rightarrow |01\rangle \\ |11\rangle \rightarrow |11\rangle \end{cases}$$

2.3 Applying Gate Operations to a Larger System

Note that an $n$-qubit system is represented by a vector of size $2^n$, and an $m$-qubit gate is represented by a $2^m \times 2^m$ unitary matrix. Thus, when an $m$-qubit gate is applied to an $n$-qubit system ($m < n$), the corresponding unitary matrix must be extended from $2^m \times 2^m$ to $2^n \times 2^n$ in order to fit to the size of the state vector. Formally, this can be done by tensoring with identity matrices. Readers should refer to [18] for more information. In the following, we describe how the above mentioned gates can be extended so that they can be applied to a larger system.

- **Case: One-qubit Gate**

  We consider the case that a one-qubit gate, whose matrix representation is $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is applied to the $t$-th
We consider the case that a SWAP gate is applied to the $i$-th and the $j$-th qubits of an $n$-qubit system. In this case, each basis state is transformed as follows:

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_10x_{t-1}\ldots x_0\rangle &\rightarrow a|x_{n-1}x_{n-2}\ldots x_10x_{t-1}\ldots x_0\rangle \\
&\quad + c|x_{n-1}x_{n-2}\ldots x_11x_{t-1}\ldots x_0\rangle
\end{align*}
$$

for any assignment to $x_i$'s ($l \neq t$). (1)

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle &\rightarrow b|x_{n-1}x_{n-2}\ldots x_{s+1}0x_{t-1}\ldots x_0\rangle \\
&\quad + d|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle
\end{align*}
$$

for any assignment to $x_i$'s ($l \neq t$). (2)

- **Case: Controlled-Unitary**

We consider the case of a controlled-unitary gate, whose control-bit and target-bit are the $s$-th and the $t$-th respectively, and whose associated unitary matrix is 

$$
\begin{pmatrix}
    a & b \\
    c & d
\end{pmatrix}
$$

In this case, each basis state is transformed as follows:

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}0x_{t-1}\ldots x_0\rangle &\rightarrow |x_{n-1}x_{n-2}\ldots x_{s+1}0x_{t-1}\ldots x_0\rangle \\
&\quad \text{for any assignment to } x_i \text{'s (} l \neq s, l \neq t \text{).} (3)
\end{align*}
$$

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle &\rightarrow a|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle \\
&\quad + c|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle \\
&\quad \text{for any assignment to } x_i \text{'s (} l \neq s, l \neq t \text{).} (4)
\end{align*}
$$

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle &\rightarrow b|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle \\
&\quad + d|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{t-1}\ldots x_0\rangle \\
&\quad \text{for any assignment to } x_i \text{'s (} l \neq s, l \neq t \text{).} (5)
\end{align*}
$$

- **Case: SWAP**

We consider the case that a SWAP gate is applied to the $i$-th and the $j$-th qubits of an $n$-qubit system. In this case, each basis state is transformed as follows:

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}0x_{j-1}\ldots x_0\rangle &\rightarrow |x_{n-1}x_{n-2}\ldots x_{s+1}0x_{j-1}\ldots x_0\rangle \\
&\quad \text{for any assignment to } x_i \text{'s (} l \neq i, l \neq j \text{).} (6)
\end{align*}
$$

$$
\begin{align*}
|x_{n-1}x_{n-2}\ldots x_{s+1}1x_{j-1}\ldots x_0\rangle &\rightarrow |x_{n-1}x_{n-2}\ldots x_{s+1}1x_{j-1}\ldots x_0\rangle \\
&\quad \text{for any assignment to } x_i \text{'s (} l \neq i, l \neq j \text{).} (7)
\end{align*}
$$

In summary, we show examples of gate operations in Fig. 1.

3. Quantum Circuit Simulation

A quantum circuit consists of a sequence of quantum gate operations, which is a program for a quantum computer. An example of a quantum circuit is illustrated in Fig. 2. Each line represents a qubit, and quantum gates are placed on those qubits. Time goes from left to right. According to the quantum circuit in Fig. 2, quantum gates are applied in the following order: (1) the one-qubit gate, (2) the controlled-unitary gate, (3) the SWAP gate. An input of the proposed simulator is a description of a quantum circuit (i.e., a program for a quantum computer) together with the initial state of qubits (i.e., a complex vector of size $2^n$ for an $n$-qubit system). The goal of a quantum circuit simulation is to output the final state vector that can be obtained after applying all the gate operations. We allow any one-qubit gates, any controlled-unitary gates and any swap gates as primitives, which forms a universal gate set [17]. We call one-qubit gates and controlled-unitary gates operation gates. We assume that each of the one-qubit gates and each of the target-
bits of controlled-unitary gates are placed between the 0-th qubit and the \((k-1)\)-th qubit, where \(k\) is a design parameter given in advance, and \(2^k\) represents the size of the register array implemented in the simulator. Note that this assumption does not lose generality since we can move any one-qubit gate from the \(i\)-th qubit (\(i > k\)) to the \(j\)-th qubit (\(j \leq k\)) by inserting a swap gate. Figure 3 illustrates an example for \(k = 0\). The two quantum circuits in Fig. 3 are equivalent. In the right-side circuit, each of the one-qubit gates and the target bits of controlled-unitary gates is placed on the bottom line, or the 0-th qubit.

4. The Architecture of the Quantum Computer Simulator

The proposed quantum computer simulator can simulate quantum algorithms that are given as quantum circuits. The proposed simulator architecture consists of three portions; a register array, operation units, and a memory module (see Fig. 4). Each entry of the state vector is stored in the data memory. A description of a quantum circuit (i.e., a sequence of quantum gates) is stored in the instruction memory. The simulator fetches the description of a quantum gate one by one from the instruction memory. According to the description of the quantum circuit, vector entries are loaded from the data memory to the register array. Then, the register reordering module permutes vector entries, and the register values are passed to the operation units, in which sum of products operations corresponding to the gate operation are executed.

We describe how quantum gates are simulated in the following.

4.1 Simulation of SWAP Gates

Note that a swap gate just swaps the positions of the specified two qubits. In order to simulate swap gates, the simulator memorizes the current order of qubits. When a swap gate is applied, the simulator updates the order of qubits by swapping the positions of the two qubits specified by the swap gate. For example, let \((b_3, b_2, b_1, b_0)\) be the current order of qubits, where each \(b_i\) \((0 \leq i \leq 3)\) is the identifier of the qubit. If we apply a swap gate between the 0-th and the 1-st qubits, then the resulting order of the qubits will be \((b_3, b_2, b_0, b_1)\). Memorizing the order is enough to continue the simulation.

4.2 Simulation of Operation Gates

We use the linearity of matrix-vector multiplication to simulate operation gates efficiently, which is commonly used in quantum circuit simulations. Let \(v = (\alpha_0, \ldots, \alpha_{2^k-1})^T\) be a state vector of an \(n\)-qubit system. We divide \(v\) into \(v_i\)'s as follows: \(v_i = (0, 0, \ldots, 0, \alpha_i, 2^i, \ldots, \alpha_{i+j}2^i-1, 0, \ldots, 0)^T\) for \(0 \leq i < 2^n/2^k\), where \(k\) is the design parameter mentioned in Sect. 3. In other words, each \(v_i\) is the projection of \(v\) onto the \(2^k\)-dimensional subspace. Then, it is obvious that \(\sum v_i = v\).

We consider a sequence of operation gates, \((g_1, g_2, \ldots, g_l)\). We identify each of operation gates \(g_i\) with the corresponding transformation. Then by linearity, the following holds:

\[
g_1 \circ \cdots \circ g_2 \circ g_1(v_i) = \sum g_1 \circ \cdots \circ g_2 \circ g_1(v_i),
\]

where \(g_j \circ g_i\) is a composite transformation of \(g_i\) and \(g_j\). This means that we may apply gate operations to each \(v_i\) independently, and then sum up the resulting vectors \(v'_i = g_1 \circ \cdots \circ g_2 \circ g_1(v_i)\) \((0 \leq i \leq 2^n/2^k)\) to obtain the whole result.

Note that each of one-qubit gates (or each of target-bits of controlled-unitary gates) is placed between the 0-th qubit and the \((k-1)\)-th qubit as mentioned in Sect. 3. Also note that \(v_i\) has vector entries from \(\alpha_{i2^k} \rightleftharpoons \alpha_{i+i2^k-1}\) (from \(\alpha_{x_1x_2 \ldots x_{2^k}00\ldots0}\) to \(\alpha_{x_1x_2 \ldots x_{2^k}11\ldots1}\)) in binary notation of indices where \((x_{n-1}x_{n-2} \ldots x_k) = i\). Thus, each \(v'_i\) is in the same subspace as \(v_i\) by Eqs. (1) to (5).

For a sequence of operation gates, \(g_1, \ldots, g_l\), the simulator runs as follows:

**Step 1** A block of entries of the state vector (a set of \(2^k\) entries out of \(2^n\) entries) is loaded to the register array from the data memory. These entries form \(v_i\) mentioned...
The description of a gate to be processed is loaded from the instruction memory. Then, the simulator permutes register values so that the vector entries to be operated (i.e., the operands for each operation unit) can be placed next to each other.

**Step 3** Register values are updated by executing sum-of-products operations in parallel. This corresponds to applying a unitary matrix to \( v_t \).

**Step 4** The simulator repeats Steps 2 and 3 for each of \( g_1, \ldots, g_l \).

**Step 5** Register values are written back to the data memory. Then, the next block of vector entries, which corresponds to \( v_{t+1} \), is loaded to the register array. The above procedure is repeated again for \( g_1, \ldots, g_l \) until \( i = 2^n / 2^k - 1 \).

We adopted a double-buffer architecture for loading and storing data between the register array and the data memory so that memory access overhead can be reduced.

Any quantum circuit can be represented by iterations of a sequence of swap gates followed by a sequence of operation gates.\(^1\) Figure 5 illustrates an example. Register values are loaded (resp. written back) from (resp. to) the data memory at the start (resp. the end) of a sequence of operation gates as explained in the above. If the sequence of operation gates is long, we have enough time to load (resp. store) the next (resp. the previous) block of vector entries while processing the sequence. Note that we adopted the double-buffer architecture. Thus, we can load (resp. store) the next (resp. the previous) block of vector entries while processing the current block of vector entries. Therefore, long sequences of operation gates can hide memory access overheads.

In the above procedure, Step 2 could be a hardware demanding portion of the simulator since possible interconnections of registers that store vector entries are huge (see Fig. 6). In order to avoid complex interconnection networks, we developed a hardware algorithm that reorders register values so that the vector entries to be operated can be placed next to each other. In the next section, we describe the details of the register reordering method.

Step 3 is executed by a linear array of operation units, which is directly connected to the register array. Each operation unit is a sum-of-products unit that consists of two complex multipliers and an adder.

**5. Register Reordering and Parallel Execution of Calculations of Vector Entries**

In the proposed simulator architecture, the simulator selects operands for each operation unit. If we did it by using multiplexers, it would result in massive inter-connection networks. This is why we developed a new hardware algorithm that selects operands for each operation unit by reordering register values. Our algorithm just iterates shifts and swaps of register values, and does not need massive multiplexers. Thus, this has advantages such as high clock frequency and area efficiency. Let \( v = (\alpha_0, 0, \ldots, \alpha_{11\ldots 1})^T \) and \( v' = (\alpha'^0_0, 0, \ldots, \alpha'^{11\ldots 1}_1)^T \) be the current state vector and the state vector after applying a gate operation respectively. We focus on a one-qubit gate operation in the following, and the case of a controlled-unitary gate follows immediately. According to Eqs. (1) and (2), we have the following equations when a quantum gate is applied to the \( t \)-th qubit.

\[
\alpha'^{a_0a_1\ldots a_{2^{k-1}}}_{a_2a_3\ldots a_0}(0_{a_1\ldots a_0}) = a \cdot \alpha_{a_2a_3\ldots a_0}0_{a_1\ldots a_0} + c \cdot \alpha_{a_2a_3\ldots a_0}1_{a_1\ldots a_0}, \quad (6)
\]

\[
\alpha'^{b_0b_1\ldots b_{2^{k-1}}}_{b_2b_3\ldots b_0}(1_{b_1\ldots b_0}) = b \cdot \alpha_{b_2b_3\ldots b_0}0_{b_1\ldots b_0} + d \cdot \alpha_{b_2b_3\ldots b_0}1_{b_1\ldots b_0}. \quad (7)
\]

Thus, it is necessary to make up pairs of vector entries \( (\alpha_{a_2a_3\ldots a_0}0_{a_1\ldots a_0}, \alpha_{a_2a_3\ldots a_0}1_{a_1\ldots a_0}) \) as inputs for operation units. As mentioned in the previous section, a multiplexer-based implementation causes complex interconnection networks as illustrated in Fig. 6. We developed the register reordering algorithm, as shown in Fig. 7, to avoid such complex interconnection networks. In Fig. 7, the indices are expressed in decimal notation, such as \( a_0, a_1, \ldots, a_{2^{k-1}} \), rather than in binary notation. For simplicity, we assume that vector entries of \( a_0, a_1, \ldots, a_{2^{k-1}} \) are loaded to the register array (i.e., the first block is loaded). Figures 8 and 9 illustrates how the register reordering works. In the figures, the target bit is the second bit.\(^n\)

---

\(^1\) A sequence of gates may consists of a single gate.

\(^n\) Note that we defined the least significant bit as the 0-th bit.
The execution time for reordering the register array and the subsequent multiply-add operations is given as the following equation:

\[
t_{all} = c \cdot n + t_{mult} + 3t_{add},
\]

where \(c\) is a constant, \(t_{mult}\) and \(t_{add}\) are execution time of an integer multiplication and an integer addition, respectively. Note that \(c \cdot n\) represents the execution time of register reordering and \(t_{mult} + 3t_{add}\) represents the execution time of the subsequent multiply-add operations.

On the other hand, if multiply-add operations were done by sequentially, the execution time would be the following:

\[
t_{all} = c \cdot n + 6n \cdot t_{mult} + 12n \cdot t_{add}
\]

when implementing a complex multiplier with three integer multipliers,

\[
t_{all} = c \cdot n + 8n \cdot t_{mult} + 6n \cdot t_{add}
\]

when implementing a complex multiplier with four integer multipliers.

Thus, by parallel executions of multiplications and additions, we can reduce the execution time by \((6n - 1) \cdot t_{mult} + (12n - 3) \cdot t_{add}\) or \((8n - 1) \cdot t_{mult} + (6n - 3) \cdot t_{add}\), from which the acceleration is derived.

6. Experimental Evaluation

In order to evaluate the performance of the proposed simulator, we synthesized the simulator and evaluated resource usages and the maximum clock frequency. The design environment is as follows:

| Language      | Verilog-HDL |
|---------------|-------------|
| EDA tool      | Xilinx ISE 12.4 |
| Target device | Xilinx Vertex 5 (XC5VTX240T) |

We represent a vector entry as a fixed point number whose word size is 32 bits for each of the real and the imaginary part.\(^1\) We set the design parameter \(k = 5\) (i.e., the size of the register array is \(2^5 = 32\) words). We implement complex multipliers using DSP slices and also implement the data memory using block RAMs.

Figure 10 shows resource usages (look-up tables, DSP slices, and Block RAMs) and the maximum clock frequency. The experimental results show that resource usages for the register array and operation units are almost independent of the number of qubits, and so is the maximum clock frequency. In other words, only the size of the data memory

\(^1\)Xilinx Core Generator, which is an IP core generator, has an option of implementing a complex multiplier with three multipliers which trades off one multiplier with three adders. A naive implementation of a complex multiplier needs four multipliers and two adders.

\(^{11}\)Note that vector entries are complex values. Thus, each vector entry needs \(32 + 32 = 64\) bits of memory.
increases depending on the number of qubits. In this sense, our architecture is scalable in terms of the size of quantum circuits.

By simulating the implemented circuit on the EDA tool, we evaluated the number of clock cycles needed to complete simulations of quantum gate operations. Then, based on the maximum clock frequencies in Fig. 10, we calculated the throughputs of the simulator in different settings. The throughput of the simulator depends on the length of each sequence of operation gates as mentioned in Sect. 4. Figure 11 shows throughputs of the simulator for each of the lengths of sequences of operation gates.

The throughput saturates when the length of the sequence of operation gates is more than or equal to 5. This is because the next block of vector entries are completely loaded to the buffers within processing the first five gates of the sequence, and thus, memory access overhead is completely eliminated if the length of the sequence of operation gates is at least 5.

Finally, we compare the performance of the proposed hardware simulator with a software quantum computer simulator. To do this, we developed a software quantum computer simulator. We have several options of architectures on which the software simulator runs, such as multi-core CPUs, GPGPUs, etc. Many-core architectures have a disadvantage of data communication overhead. Simulation of a quantum computer has almost no locality of memory access. Thus, many processing elements communicate with each other to exchange vector entries, and this can be a bottleneck. In fact, one of the author pointed out this bottleneck in [12]. Therefore, we used a four-core CPU rather than many-core accelerators such as GPGPUs.

The experimental environment for the software simulator is as follows: OS: Linux 2.6.18, CPU: Intel Core2 Quad Q9650 3.0GHz, Memory: 4GB, Compiler: g++ (GCC) 4.4.7. The simulator is designed to run on four cores and compiled with OpenMP and with optimization option ‘-O3’. The experimental results are shown in Fig. 11.

The throughput of the proposed hardware simulator is from 1.39 to 1.76 times faster than the software simulator when the length of the sequence of operation gates is 5, and from 1.42 to 1.85 times faster when the length of the sequence of operation gates is 12. Thus, the proposed simulator has an advantage over the software simulator. Note that the simulator does not need to transfer the initial data to the FPGA device since the initial state vector is fixed to (1, 0, . . . , 0).\(^1\) Thus, we do not need to consider the time to

\(^1\)We can assume that every quantum algorithm starts with the
Fig. 12 Quantum Fourier transform circuit

Fig. 13 Scheduled QFT circuit

Table 1 QFT benchmark

|                  | Throughput (gates/sec) | Execution time (ms) |
|------------------|------------------------|---------------------|
| The proposed simulator | 31164                  | 2.50                |
| Software simulators | 18182                  | 4.29                |

7. Using an External Memory

The implemented simulator is a prototype that uses embedded block RAMs. If we use an external memory, the simulator can cope with a larger number of qubits. Note that, even in this case, the size of the simulator core is almost the initial state vector \((1, 0, \ldots, 0)\) without loss of generality.

We also evaluated our simulator using the 12-qubit quantum Fourier transform circuit as benchmark. The quantum Fourier transform circuit is illustrated in Fig. 12. For efficient simulation, we scheduled the quantum gates preserving gate dependency as illustrated in Fig. 13. The resulting circuit computes the same function as the original circuit. We set the design parameter \(k = 5\). Then, we divided the quantum Fourier transform circuit, which is a sequence of operation gates, into three subsequences of operation gates whose lengths are 33, 35, and 10, respectively. The target bits are placed between the 0th and the 3rd qubits for the first subsequence, between the 4th and the 8th for the second subsequence, and between the 9th and the 11th for the third subsequence. Note that we can insert swap gates between the subsequences so that each target bit can be placed between the 0th and the 4th qubits. Thus, the proposed simulator can simulate the circuit. The results are shown in Table 1, which shows that the proposed simulator is 1.71 times faster than the software simulator.

On the other hand, using a DRAM makes the access time slower. In order to hide the slower memory access, it is needed to make the sequence of operation gates longer. Figure 14 shows how long the sequence of operation gates need to be to completely hide the memory access overhead. The results are represented in terms of memory throughput ranging from 20MB/s to 60MB/s. The results are obtained based on the implementation of the simulator core in the previous section. We assume that the simulator core runs at 220MHz.

The results show that, for memory throughput of 20MB/s, 40MB/s, and 60MB/s, if the length of a sequence of operation gates is longer than or equal to 48, 24, and 16, respectively, the memory access overhead can be hidden. For example, 25-qubit quantum Fourier transform circuit can be divided into five sequences of operation gates whose lengths are 115, 90, 65, 40, and 15, respectively. Thus, we can almost hide the memory access overhead if throughput of the external memory is 60MB/s. Even if the memory throughput is 40MB/s, only the last sequence suffers from the memory access overhead.

8. Conclusion

We proposed a hardware quantum computer simulator architecture. The proposed architecture is based on the register reordering method, and thus, has advantages of high clock frequency as well as small area consumption. We also demonstrated that the proposed hardware simulator runs faster than software quantum circuit simulators. The performance of the proposed simulator depends on the lengths of sequences of operation gates in the given quantum circuit. We can make the lengths of the sequences longer by scheduling gate operations as shown in the QFT benchmark. Optimization of such scheduling remains as a future work.

References

[1] M. Aminian, M. Saeedi, M. Zamani, and M. Sedighi, “Fpga-based circuit model emulation of quantum algorithms,” IEEE Computer Society Annual Symposium on VLSI, ISVLSI ’08, pp.399–404, April 2008.
Masaki Nakanishi received the B.E., M.E. and Ph.D. degrees from Osaka University, Japan, in 1996, 1998 and 2002 respectively. He joined the Graduate School of Information Science, Nara Institute of Science and Technology as Research Associate in 2000. He is currently with Faculty of Education, Art and Science, Yamagata University as Associate Professor. His current interests include VLSI design, quantum computing and design of combinatorial algorithms.

Miki Matsuyama received the Bachelor of Arts and Sciences degree from Yamagata University, Japan, in 2014. She currently works for Sakata City. Her current interests include VLSI design and quantum computing.

Yumi Yokoo received the Bachelor of Arts and Sciences degree from Yamagata University, Japan, in 2014. She currently works for Kirayaka Bank Ltd. Her current interests include VLSI design and quantum computing.