

A Design Of FPGA Framework For Quantum Computing Simulation



A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science

Department of Computer Engineering

FACULTY OF ENGINEERING

Chulalongkorn University

Academic Year 2021

Copyright of Chulalongkorn University

การออกแบบเฟรมเวิร์กเอฟพีจีเอสำหรับการจำลองการคำนวณแบบควอนตัม



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2564
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title A Design Of FPGA Framework For Quantum Computing
Simulation
By Miss Yaninee Jungjarassub
Field of Study Computer Science
Thesis Advisor Associate Professor KRERK PIROMSOPA

Accepted by the FACULTY OF ENGINEERING, Chulalongkorn University in
Partial Fulfillment of the Requirement for the Master of Science

THESIS COMMITTEE

----- Dean of the FACULTY OF
ENGINEERING
(Professor SUPOT TEACHAVORASINSKUN)

----- Chairman
(Assistant Professor NATAWUT NUPAIROJ)

----- Thesis Advisor
(Associate Professor KRERK PIROMSOPA)

----- Examiner
(Assistant Professor VEERA MUANGSIN)

----- External Examiner
(Assistant Professor Jittat Fakcharoenphol)

ญานินี จิงจรัสทรัพย์ : การออกแบบเฟรมเวิร์กเอพฟี่จีเอสำหรับการจำลองการคำนวณแบบควอนตัม. (A Design Of FPGA Framework For Quantum Computing Simulation) อ.ที่ปรึกษาหลัก : รศ. ดร.เกริก ภิรมย์โสภา

งานวิจัยนี้ใช้เอพฟี่จีเอเพื่อเพิ่มประสิทธิภาพการจำลองของการคำนวณควอนตัมในสองด้าน ได้แก่ 1. ใช้คำสั่งควบคุมเงื่อนไขแทนการคำนวณผลิตภัณฑ์เทนเซอร์ สิ่งนี้ทำให้ผลิตภัณฑ์เทนเซอร์ของตัวดำเนินการควอนตัมแต่ละตัวถูกสร้างขึ้นในวงจรรนาฬิกาเดียว 2. เก็บค่าที่คำนวณไว้ล่วงหน้าไว้ในรอมซึ่งถูกใช้สำหรับประมาณค่าไซน์และโคไซน์ สิ่งนี้อำนวยความสะดวกในการคำนวณประตูควอนตัมที่เกี่ยวข้องกับมุม เพื่อตรวจสอบงานวิจัยนี้ จึงนำการออกแบบของเรามาใช้ใน VerilogHDL ประสิทธิภาพสามารถประเมินได้โดยใช้โปรแกรมจำลอง FPGA ผลที่ได้แสดงให้เห็นว่าในกระบวนการจำลองด้วยงานของเราดีขึ้นเมื่อเปรียบเทียบกับจำลองบนคอมพิวเตอร์แบบคลาสสิก



สาขาวิชา วิทยาศาสตร์คอมพิวเตอร์
ปีการศึกษา 2564

ลายมือชื่อนิสิต
ลายมือชื่อ อ.ที่ปรึกษาหลัก

6270068621 : MAJOR COMPUTER SCIENCE

KEYWORD: tensor product, quantum gate, FPGA, simulation

Yaninee Jungjarassub : A Design Of FPGA Framework For Quantum Computing Simulation. Advisor: Assoc. Prof. KRERK PIROMSOPA

We use FPGA to optimize the simulation of quantum computing in two aspects. (a) The if-else state is used in place of tensor product calculation. This allows the tensor product of each quantum operator to be generated in a single clock cycle. (b) The pre-calculated lookup ROM is used for estimating the sine and cosine values. This facilitates the computation of quantum gates that are related to angle. To validate our work, we implement our design in VerilogHDL. The performance is evaluated using an FPGA simulator. The result shows a dramatic improvement in the simulation process comparing to those of simulation on classical computers.



Field of Study: Computer Science

Academic Year: 2021

Student's Signature

Advisor's Signature

ACKNOWLEDGEMENTS

Thank you to my advisor, Dr. Kerk Piromsopa, for his patience, guidance, and support. I have benefited greatly from his wealth of knowledge and meticulous editing. I am extremely grateful that he took me on as a student and continued to have faith in me.

Thank you to my committee members, Dr. Nutawut Nupairoj, Dr. Veera Muangsin, Dr. Puchong Uthayopas and Dr. Jittat Fakcharoenphol. Your encouraging words and thoughtful, detailed feedback have been very important to me.

Thank you to Dr. Thiparat Chotibut and Apimuk Sornsang, for all assistance about the quantum computing knowledge that you have provided.

Thank you to my family, Patcharamon Jungjarrasub, Akkarapong Jungjarrasub and Pholparis Jungjarrasub, for your endless support. You have always stood behind me, and this was no exception. Thank you for cheering me up and calming me down on the day when my mind was depressed. Thank you for all of your love and for always reminding me of the end goal.

Thank you to my sweetheart, for always being there for me and for telling me that I should do my thesis even though I'm discouraged.

Thank you to my master's degree friends, Nutchazazum for always listening to me and talking for telling me that I can do it even when I didn't feel that way. Kitsaphon Thitisiriwech and Vitchaya Siripoppohn these friends made my master's degree life more enjoyable.

Thank you to my bachelor's degree friends, for always supporting and listening to me.

Thank you to SPA lab's member, for exchanging knowledge and practicing questioning.

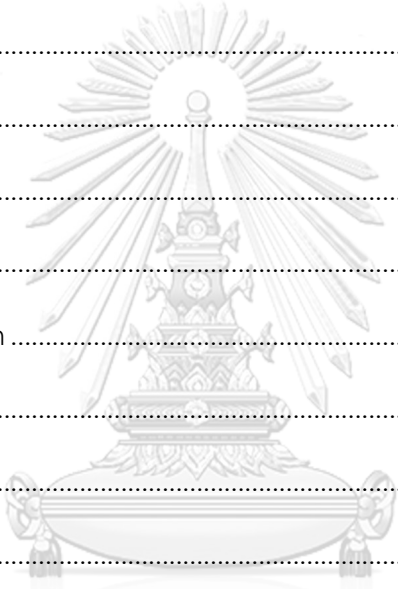
Thank you to Hinata (from Haikyuu) and Naheedo (from Twenty Five Twenty One) for being a model of effort. That's the key to helping me complete this thesis.

Yaninee Jungjarassub



TABLE OF CONTENTS

	Page
.....	iii
ABSTRACT (THAI)	iii
.....	iv
ABSTRACT (ENGLISH)	iv
ACKNOWLEDGEMENTS	v
TABLE OF CONTENTS	vii
LIST OF TABLES	x
LIST OF FIGURES.....	xi
Problem and Motivation	1
Objective.....	2
Scope	2
Background knowledge	2
Quantum Computer.....	2
Superposition.....	2
Entanglement	2
Representing a quantum state in quantum computing.....	3
Representing an operator in quantum computing	3
Basic quantum gate.....	3
Quantum gate in quantum system	5
Definition of unitary matrix.....	6
Transforming state in quantum computing.....	6



Field-programmable gate arrays (FPGAs).....	6
Quantum Computer Architecture.....	7
Qiskit.....	7
Graphic processing units (GPU).....	8
Field programable gate arrays (FPGA).....	8
Propositions.....	15
Instruction set architecture.....	18
Instruction format.....	18
Representation of data in our system.....	20
System design.....	21
Signal.....	21
Processor.....	22
State machine.....	22
Memory.....	25
Generator.....	26
Vector generator.....	26
Matrix generator.....	26
Co-processor.....	27
Vector-Matrix multiplication module.....	27
Measurement module.....	29
Verification of the simulation.....	31
Verification of generator module.....	35
Verification of vector generator module.....	35
Verification of matrix generator module.....	36

Verification of Vector-Matrix Multiplication module	43
Verification of Measurement module	44
Result of propositions.....	45
Result of Proposition I.....	45
Result of Proposition II.....	50
REFERENCES.....	59
VITA	62



LIST OF TABLES

	Page
Table 1 Show the quantum gates.....	5
Table 2 Comparison of CORDIC and ROM.....	14
Table 3 Type of opcode in our system	19
Table 4 Relevant signal of our system.....	21
Table 5 Representation of actual output of 4 modules in our system	34
Table 6 Verification of vector generator module.....	35
Table 7 Verification of 1-qubit gate generator module	39
Table 8 Verification of 2-qubit gate generator module	41
Table 9 Verification of 3-qubit gate generator module	42
Table 10 Verification of Vector-Matrix multiplication module.....	43
Table 11 Verification of Measurement module	44
Table 12 Comparison of runtime between software simulation and our work	49
Table 13 Meaning of all states in our system.....	50
Table 14 Simulation time comparison of our work and software simulation.....	51
Table 15 Clock speed comparison of our work and software simulation	51
Table 16 Raw data of run time of Qiskit.....	58

LIST OF FIGURES

	Page
Figure 1 Quantum circuit of Quantum Fourier Transform (QFT) for 5 qubits.....	11
Figure 2 Algorithm for creating matrix operator of single-qubit gate.....	16
Figure 3 Algorithm for creating matrix operator of multiple-qubit gate	17
Figure 4 Fixed-point number that represent complex number.....	20
Figure 5 System design of our work.....	21
Figure 6 State machine of system module.....	23
Figure 7 State machine of generator matrix module and vector-matrix multiplication module.....	24
Figure 8 State machine of measurement module	25
Figure 9 Vector generator module	26
Figure 10 Matrix generator module	27
Figure 11 Vector-Matrix multiplication module	28
Figure 12 Design of vector-matrix multiplication module.....	29
Figure 13 Design of measurement module.....	30
Figure 14 Simulation result for generating operator step in 1 clock cycle	49
Figure 15 Timing diagram represent state machine when simulation (Zoom in).....	50
Figure 16 Timing diagram represent state of state machine when simulation	50

Introduction

Problem and Motivation

While a quantum computing provider (such as IBM) allows remote access to a real quantum computer, the developers still have to wait in a long line. Consequently, the use of a quantum computing simulator or emulator is an alternative tool that allows initial algorithm development to be tested and validated.

With the use of physical effects like superposition, a quantum state is able to represent multiple states at the same time. In addition, entanglement allows a change in the state of one qubit to immediately change the state of the associated qubit. These properties allow a quantum computer to solve problems that are intractable for a classical computer. Examples include solving the factorization problem for RSA decryption using Shor's algorithm in polynomial time. Because one part of Shor's algorithm comes from a quantum fourier transform that takes advantage of quantum computing.

There exist several software simulations of a quantum computing on classical computers. Unlike the parallel execution in a quantum computer, the inherent classical computer is, however, executed in a sequence. Intuitively, the implementation of a quantum algorithm on the classical computer would take more time to execute comparing to those of the quantum computer. To ease the simulation speed, parallel execution hardware, such as graphics processing units (GPUs) and Field Programmable Gate Arrays (FPGAs), has been employed to achieve faster simulation time.

We aim at using FPGA to improve the speed of quantum computer simulators. This simulator allows developer to test arbitrary quantum algorithm.

Objective

To design a platform for quantum computing simulation using VerilogHDL simulation that can simulate arbitrary quantum algorithms at a faster speed than software simulation.

Scope

1. This research is simulated on VerilogHDL.
2. We need to focus on run time to compare with the baseline (Qiskit).
3. The Qiskit is used as a baseline for proving the correctness of quantum circuits.

Background knowledge

Quantum Computer

A quantum computer is a type of computer that takes the advantage of quantum phenomena, this allows the quantum computer to outperform a classical computer. In general, there are 2 properties: superposition and entanglement.

Superposition

It is the principle of quantum superposition states that allows a quantum state to represent more than one (classical) state at the same time. In another word, we can represent arbitrary states by using the combination of all possible states in quantum bits.

Entanglement

Quantum entanglement is a physical phenomenon that happens when we have 2 qubits, which can be entangled. Change in the state of one qubit will immediately change the state of another one. The change of that qubit is easy to foretell. This phenomenon has been true although these qubits are far away from each together.

[1]

Representing a quantum state in quantum computing

In the classical machine, a bit is used for describing the information of the classical system. The classical bit is either 0 or 1. On another hand, the quantum computer has a unit that can represent multiple states at the same time. it's called a qubit. [2] A quantum bit or a qubit is a unit of information describing a two-dimensional quantum system. A qubit can be represented by a 2^n -by-1 matrix with complex numbers.

$$|q\rangle = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \begin{matrix} 0 \\ 1 \end{matrix} \quad (1)$$

where n is the number of qubits and $|c_0|^2$ and $|c_1|^2$ are the probabilities amplitude of qubit, where the measurement will result in $|0\rangle$ and $|1\rangle$ respectively. Thus, $|c_0|^2 + |c_1|^2 = 1$. [1]

Representing an operator in quantum computing

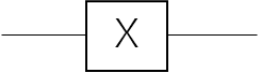

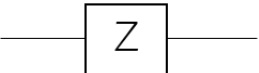
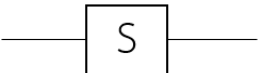

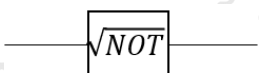
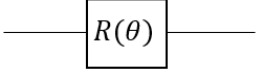
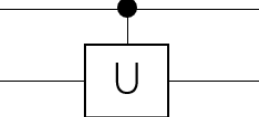
Basic quantum gate

In classical computer, logic gates are ways of managing bits. If the input bits are passed through a gate, we will get the result bits. So, the calculation relies on types of each gate. In the same way, quantum computer qubits are manipulated by quantum gate.

A quantum gate is simply an operator that acts on qubits. Such operators can be represented by unitary matrices. We can express the gate in the form of 2^n -by- 2^n matrix (2). Where n is number of qubits.

$$U = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad (2)$$

There are many types of quantum gates that are shown with their unitary matrix in Table 1.[3]

Operator	Gate	Matrix
Pauli-X (X)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
S		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$T \left(\frac{\pi}{8}\right)$		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Square root of NOT gate		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$
Phase shift		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$
Controlled U		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix}$ When U is an arbitrary single quantum gate.

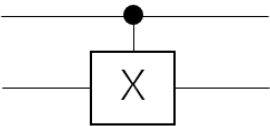
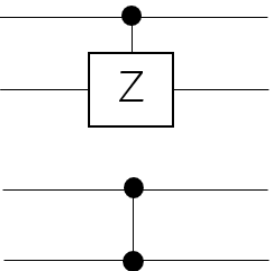
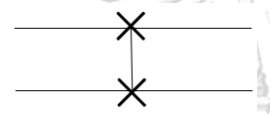
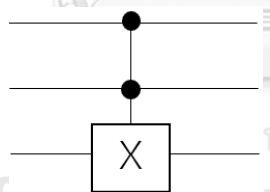
Controlled NOT (CX, C-NOT)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli gate (CCX, CCNOT, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Table 1 Show the quantum gates

Quantum gate in quantum system

Each qubit in a quantum system interacts with others. To model them, a tensor product is the mathematical expression from multiple qubits transformation. The tensor operation on any pair of 1-qubit transformations is illustrated as follows.[1]

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \otimes \begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \end{bmatrix} \\
 &= \begin{bmatrix} a_0 b_0 & a_0 b_1 & a_1 b_0 & a_1 b_1 \\ a_0 b_2 & a_0 b_3 & a_1 b_2 & a_1 b_3 \\ a_2 b_0 & a_2 b_1 & a_3 b_0 & a_3 b_1 \\ a_2 b_2 & a_2 b_3 & a_3 b_2 & a_3 b_3 \end{bmatrix}
 \end{aligned} \tag{3}$$

Definition of unitary matrix

When we have an n-by-n matrix U, it will be unitary matrix if and only if

$$U \cdot U^\dagger = U^\dagger \cdot U = I_n \tag{4}$$

when U^\dagger is conjugate and transpose of U.

Transforming state in quantum computing

If we want to apply a quantum gate (operator) on a quantum state, a matrix-vector multiplication can be used to describe this transformation. The example is shown in equation 5.

$$U|q\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} a \cdot c_0 + b \cdot c_1 \\ c \cdot c_0 + d \cdot c_1 \end{bmatrix} \tag{5}$$

Field-programmable gate arrays (FPGAs)

Field-programmable gate arrays (FPGAs) are digital devices that rely on digital logic. There are many general-purposed logic gates on FPGA that can be used for design the circuit using drawing logic gates or using the Verilog (or another) hardware description language. [4]

Related Work

There are many ways to simulate or emulate quantum computing. In this chapter, we will present the proposition of the quantum computer architecture and the previous quantum computing simulator or emulator by software, GPUs and FPGA respectively.

Quantum Computer Architecture

The concept and implementation of a quantum computer architecture to allow creating a new computational device as a quantum computer accelerator are presented by Koen Bertels et al [6]. They present the idea of a quantum accelerator that contains the full-stack of the layers of an accelerator. The highest level is an algorithm or application, and the next layer is quantum logic that can represent the algorithm. This logic is translated to a common assembly language called cQASM by OpenQL. Then, the compiler can convert cQASM to eQASM to generate an executable on the specific device.

Moreover, this article mentions the use of a full-stack quantum accelerator. There are 2 ways of using quantum accelerators, so it depends on the type of the lowest layer device. If the device is quantum chip material, it is used to improve the quality of qubits. If the device is a quantum simulator, such as a GPU or FPGA, it is used to develop quantum algorithms.

Qiskit

Qiskit [quiss-kit] is an open-source SDK for working with quantum computers at the level of pulses, circuits, and application modules. [5]

However, the nature of the classical computer, which processes in sequence, causes it to take exponential time and resources, and is not suitable for simulating the parallel processing inherent in the quantum computer. Accordingly, the usage of devices that can process in parallel, such as GPUs and FPGAs, is interesting.

Graphic processing units (GPU)

Graphic processing units (GPU) is a device that can process in parallel. Smith A. and Khavari K. [6] implemented quantum fourier transform (QFT) which is the heart of many other quantum algorithms using Compute Unified Device Architecture (CUDA) GPUs. They proposed optimization for GPU in two ways. First, Algebraic manipulations include combining consecutive phase shift gates into one that improves 3.8x speedup by choosing $\cos\theta$ and $\sqrt{1-\cos 2\theta}$ instead of $\sin\theta$ and $\cos\theta$ (by the knowledge that $\sin\theta = \sqrt{1-\cos 2\theta}$ in calculations). Second, Combining Kernels and Shared Memory when they realize these methods together can achieve further speed up.

Oumarou et al. [7] use CuPy, which is the NumPy equivalent library that supports CUDA enabled GPUs, a general-purpose library (linear algebra) developed specifically for CUDA-based GPUs, to simulate quantum circuits. Within the Python ecosystem, they have to pay attention to usability, implementation, and maintainability. They benchmarked the performance of CuPy using two types of circuits: supremacy and arithmetic. When compared to state-of-the-art C++-based simulators, the speedup for supremacy circuits is around 2x, and for quantum multipliers it is nearly 22x.

Field programable gate arrays (FPGA)

Field-programmable gate arrays (FPGAs) are digital devices that rely on digital logic. There are many general-purposed logic gates on FPGA that can be used for designing the circuit using drawing logic gates, VerilogHDL or other hardware description languages.[4] Moreover, FPGA which has intrinsic parallelism is a good choice to mimic quantum computer behavior because of its properties in performing bit-level parallelism. From the previous studies, there are 3 ways to simulate quantum computer using FPGA.

From the architectural point of view, C. Conceição and R. Reis [8] proposed a processor architecture based on single instruction multiple data (SIMD) which is capable of efficiently emulate quantum circuit. Lee et al. [9] presented serial-parallel architecture with efficient resource utilization. Their work has two advantages: serial architecture uses fewer resource comparing to those of others and the pipeline architecture yield higher throughput. This work can achieve a linear reduction of resource utilization when compared to pipeline architecture. Furthermore, this work chooses a suitable number of fixed-point representations for balancing between precision error and resource utilization.

From the circuit point of view, N. Mahmud, E. El-Araby, and D. Caliga [10] inspected the approach for emulating quantum computing instead of emulating gate-based quantum circuit. Scalability was their goal. There were 2 models of emulation depending on the type of matrix representing the quantum algorithm. The first model is suitable for the dense quantum algorithm matrix. Reducing the quantum circuit to arithmetic operation (complex multiplication-and-accumulation CMAC) is a key of this model. Moreover, they can optimize this model by combining two types of computation consisting of lookup and dynamic generation. Lookup needs to pre-compute and store value in memory for speed optimization. Dynamic generation causes space optimization. The second model is appropriate for the sparse quantum algorithm matrix. The core operations of that matrix are extracted as a kernel. It is then applied iteratively across all groups of input states.

From the coprocessor point of view, A. U. Khalid et. al. [11] represents the approach for emulating the quantum circuit using the ability of FPGA to emulate the parallelism of quantum computing and to resolve the bottleneck in software simulation. Besides, the code-generation, which is a capability of VHDL language, constructs the n-input gates from 1-input gates (from software quantum circuit package) instead of keeping the multiple-input gate in large matrix form. Regarding the evolution, the quantum circuit have quantum state register (QSR) for holding

amplitude of every state after each transformation. For the quantum measurement, simulator from the software part is used. The universal and scalable quantum emulator using the FPGA to emulate the behavior of a real quantum system is proposed in [12]. This emulator was user, so it focuses on the ease of use and reflects on the behaviors of real-quantum computer by using advantage of FPGA that can operate the instruction in one clock tick. Hence, there are parts of software and hardware that can communicate together. Moreover, it can run the entire quantum algorithm, contains state initialization, transformation, and measurement. However, the resource optimization is not their goal. Hence, this work can emulate only two qubits, which is not enough for general quantum algorithms.



Design and Methodology

In this chapter, we will describe the system design and methodology of our simulation. To understanding more, let's start with the example. If we want to run Quantum Fourier transform (QFT) for 5 qubits, we begin the process by generating quantum state or vector of complex number and quantum circuit that is the matrix of operator. Next, to transforming quantum state the vector is multiplied with that matrix. The process is repeated until all operators have been used. Lastly, the measurement step is performed on a final result of quantum state. The details of this process are described following this.

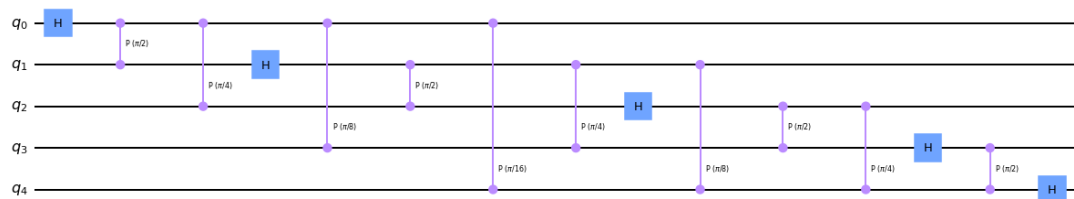


Figure 1 Quantum circuit of Quantum Fourier Transform (QFT) for 5 qubits.

First, we will explain generating quantum state step. This step produces the vector of complex number size 2^n -by-1 is show as in (6) when n is a number of qubits and 2^n is all possible quantum state in our system.

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ \vdots \\ 100 \\ 101 \\ 110 \\ 111 (2^n) \end{matrix} \quad (6)$$

After that, generating quantum circuit as the matrix size 2^n -by- 2^n is performed. For the circuit is shown as in figure 1, all matrix operator that represent the circuit at

each layer can be creating using directly calculating the tensor product. Its layer is presented by equation (7) – (21)

$$1^{st} \text{ layer} \quad I \otimes I \otimes I \otimes I \otimes H \quad (7)$$

$$2^{nd} \text{ layer} \quad I \otimes I \otimes I \otimes CR\left(\frac{\pi}{2}\right) \quad (8)$$

$$3^{rd} \text{ layer} \quad I \otimes I \otimes SWAP_{(2,1)} \otimes I \\ \cdot I \otimes I \otimes I \otimes CR\left(\frac{\pi}{4}\right) \\ \cdot I \otimes I \otimes SWAP_{(1,2)} \otimes I \quad (9)$$

$$4^{th} \text{ layer} \quad I \otimes SWAP_{(3,2)} \otimes I \otimes I \\ \cdot I \otimes I \otimes SWAP_{(2,1)} \otimes I \\ \cdot I \otimes I \otimes I \otimes CR\left(\frac{\pi}{8}\right) \\ \cdot I \otimes I \otimes SWAP_{(1,2)} \otimes I \\ \cdot I \otimes SWAP_{(2,3)} \otimes I \otimes I \quad (10)$$

$$5^{th} \text{ layer} \quad SWAP_{(4,3)} \otimes I \otimes I \otimes I \\ \cdot I \otimes SWAP_{(3,2)} \otimes I \otimes I \\ \cdot I \otimes I \otimes SWAP_{(2,1)} \otimes I \\ \cdot I \otimes I \otimes I \otimes CR\left(\frac{\pi}{16}\right) \\ \cdot I \otimes I \otimes SWAP_{(1,2)} \otimes I \\ \cdot I \otimes SWAP_{(2,3)} \otimes I \otimes I \\ \cdot SWAP_{(3,4)} \otimes I \otimes I \otimes I \quad (11)$$

$$6^{th} \text{ layer} \quad I \otimes I \otimes I \otimes H \otimes I \quad (12)$$

$$7^{th} \text{ layer} \quad I \otimes I \otimes CR\left(\frac{\pi}{2}\right) \otimes I \quad (13)$$

$$8^{th} \text{ layer} \quad I \otimes SWAP_{(3,2)} \otimes I \otimes I \\ \cdot I \otimes I \otimes CR\left(\frac{\pi}{4}\right) \otimes I \\ \cdot I \otimes SWAP_{(2,3)} \otimes I \otimes I \quad (14)$$

$$\begin{aligned}
9^{\text{th}} \text{ layer} \quad & SWAP_{(4,3)} \otimes I \otimes I \otimes I \\
& \cdot I \otimes SWAP_{(3,2)} \otimes I \otimes I \\
& \cdot I \otimes I \otimes CR\left(\frac{\pi}{8}\right) \otimes I \\
& \cdot I \otimes SWAP_{(2,3)} \otimes I \otimes I \\
& \cdot SWAP_{(3,4)} \otimes I \otimes I \otimes I
\end{aligned} \tag{15}$$

$$10^{\text{th}} \text{ layer} \quad I \otimes I \otimes H \otimes I \otimes I \tag{16}$$

$$11^{\text{th}} \text{ layer} \quad I \otimes CR\left(\frac{\pi}{2}\right) \otimes I \otimes I \tag{17}$$

$$\begin{aligned}
12^{\text{th}} \text{ layer} \quad & SWAP_{(4,3)} \otimes I \otimes I \otimes I \\
& \cdot I \otimes CR\left(\frac{\pi}{4}\right) \otimes I \otimes I \\
& \cdot SWAP_{(3,4)} \otimes I \otimes I \otimes I
\end{aligned} \tag{18}$$

$$13^{\text{th}} \text{ layer} \quad I \otimes H \otimes I \otimes I \otimes I \tag{19}$$

$$14^{\text{th}} \text{ layer} \quad CR\left(\frac{\pi}{2}\right) \otimes I \otimes I \otimes I \tag{20}$$

$$15^{\text{th}} \text{ layer} \quad H \otimes I \otimes I \otimes I \otimes I \tag{21}$$

From these equations, we notice that in case of equation (7), (8), (12), (13), (16), (17) (19), (20), (21). We can directly find the tensor product of these by operating quantum gate for target qubit and identity gate (I) for others. On the other case for the rest equations, Since the CR(theta) gate must be acted on 2-closed-qubits, therefore these layers are consisted of SWAP gate before and after a desired gate CR(theta) for switching 2-target-qubit to close each other.

Next, matrix-vector multiplication, which is the executing of each layer operator on qubits, is performed for transforming quantum state. Finally, if all operators have been used, we will measure a final vector from this process.

As an above example, in the latter case of generating quantum circuit by using tensor product at each layer uses a lot of operations of switching qubits.

Moreover, if the 2-target-qubit are more far, the number of operations for switching are larger. Therefore, we propose an optimization proposition for calculating tensor product step to improve the speed of quantum computing simulation with an implementation using FPGA. We will explain it in the next section.

Moreover, for the quantum gate that involved with angle such as R gate, CR gate. It must use sine and cosine value in calculating tensor product step. CORDIC algorithm was introduced for the computation of Trigonometric functions, Multiplication, Division, Data type conversion, Square Root and Logarithms in FPGA as mention in [13].

Criteria	CORDIC	ROM
Time		✓
Space	✓	
Accuracy		✓

Table 2 Comparison of CORDIC and ROM

In table 2, we compared 2 methods that are used to compute sine and cosine values in 3 aspects.

First, in the time aspect, ROM can overcome CORDIC. Because ROM uses the pre-compute of sine and cos values, this allows the ROM to directly use the sine and cos values. While CORDIC is an iterative algorithm, this can compute trigonometric values. For this reason, it spends more time computing each iteration than ROM.

Next, in view of space, CORDIC can overcome ROM. Because CORDIC collect a group of initial values that is used to compute other values. On another hand, the space for collecting value in ROM is depend on the number of values that we want to use. For example, suppose that we want to use sine in range 0 – 90 degrees. If we use CORDIC, it will spend 30 units to collect initial value then it will use that value to compute others. But, if we use ROM, it will spend 91 unit to collect all values.

Finally, in terms of accuracy, ROM can outperform CORDIC. Because ROM outputs the exact values, whereas CORDIC outputs the approximate values by using the group of initial values. Furthermore, the CORDIC requires more iteration for calculating sine and cosine value to achieve higher accuracy.

For the reasons stated above, ROM can outperform CORDIC in terms of speed and accuracy. As a result, the ROM is chosen for collecting sine and cosine values rather than directly calculating them (CORDIC).

Propositions

From the example above, there are two steps to simulate quantum computing using the matrix method: calculating the tensor product of quantum operators, which creates operators that act on all qubits, and matrix-vector multiplication, which is the operation of operator on qubits. In this work, we propose an optimization proposition for calculating tensor product step to improve the speed of quantum computing simulation with an implementation using VerilogHDL simulation.

Proposition 1: Describes the relationship within the operator's matrix rather than directly calculating the tensor product

From usage this proposition, the matrix operator can be created without the need for any calculation operations. Because this method simply compares the desired qubit's index in the matrix using a comparison operation instead of calculation. Therefore, this method uses (2^{2n}) operations for generating the operator. The algorithm for use of this method is shown in algorithm 1 for single-qubit gate and 2 for multiple-qubits gate. In these algorithms, the initial state is the row index in the binary form of the matrix, and the final state is the column index in the binary form of the matrix. And each qubit is equivalent to a binary index bit.

Algorithm 1 Proposition 1 for single-qubit gate

if The state of other qubits, which are not the target qubit, at the initial is not equal to the state at the final. **then**
 $c[\text{address}] \leftarrow 0$
else
 Assign a value to that address in accordance with the definition of a quantum gate.
if The state of the target qubit at initial and final are both equal to 0 **then**
 $c[\text{address}] \leftarrow a$
else if The state of the target qubit at initial is equal to 0 and final is equal to 1 **then**
 $c[\text{address}] \leftarrow b$
else if The state of the target qubit at initial is equal to 1 and final is equal to 0 **then**
 $c[\text{address}] \leftarrow c$
else if The state of the target qubit at initial and final are both equal to 1 **then**
 $c[\text{address}] \leftarrow d$
end if
end if

Figure 2 Algorithm for creating matrix operator of single-qubit gate



Algorithm 2 Proposition 1 for multiple-qubit gate

```

if The state of the control qubit at the initial is not equal to
the state at the final then
   $c[address] \leftarrow 0$ 
else
  if The state of other qubits, which are not the target qubit,
at the initial is not equal to the state at the final. then
     $c[address] \leftarrow 0$ 
  else
    if The control qubit is equal to 0. then
       $c[address] \leftarrow 1$ 
    else
      Assign a value to that address in accordance with
the definition of a quantum gate.
      if The state of the target qubit at initial and final are
both equal to 0 then
         $c[address] \leftarrow a$ 
      else if The state of the target qubit at initial is equal
to 0 and final is equal to 1 then
         $c[address] \leftarrow b$ 
      else if The state of the target qubit at initial is equal
to 1 and final is equal to 0 then
         $c[address] \leftarrow c$ 
      else if The state of the target qubit at initial and final
are both equal to 1 then
         $c[address] \leftarrow d$ 
      end if
    end if
  end if
end if

```

Figure 3 Algorithm for creating matrix operator of multiple-qubit gate

Proposition II: Use a lookup table to collect sine and cosine values instead of calculating them directly

By implementing our propositions on FPGAs, we hope to speed up the quantum computing simulation in the calculation the tensor product step.

Instruction set architecture

In this section, we will explain about an instruction format that is used to command the system. Moreover, the representation of data, which is quantum state and quantum gate, and involved signal of our system are also described.

Instruction format

In our system, we designed the instruction format have a length of 32 bits. It contains the following data that is required to simulate a quantum circuit:

1. Opcode is interpreted as the type of operator that a user wants to simulate.

There are 7 types of opcodes as shown in Table 3.

Type	Opcode	Operation
Gen_vec	101000	Generate quantum state as vector of complex number by gen_vec module.
1Q-type 1 target qubit gate	.000xxx	Generate quantum gate by using opcode and target as input
	000000	X gate
	000001	Y gate
	000010	Z gate
	000110	H gate
1Q1C-type 1 target qubit with 1 constant (an angle).	001000	Generate P gate by using opcode, target and angle as input.
2Q-type 1 target qubit and 1 control qubit.	010xxx	Generate quantum gate by using opcode, target and control as input
	010000	CX gate

	010001	CY gate
	010010	CZ gate
2Q1C-type	011000	Generate CP gate by using opcode, target control and angle as input.
3Q-type	100000	Generate CCX gate by using opcode, target and 2 control as input

Table 3 Type of opcode in our system

2. Target is a position of target qubit that a user wants to perform quantum gate on it.
3. Ctrl0 and Ctrl1 are the position of control qubit. The Ctrl0 is used in case that quantum gate what we perform has 1 control qubit but in case it has 2 control qubits the both Ctrl0 and Ctrl1 are used.
4. Angle is interpreted as the angle that is used to rotate the qubit for the quantum gate that is involved with angle. In our work, it can use angle from 0 to 90 degree because it has 7 bits which represents this part.
5. N is a number of qubits of the quantum circuit that user want to simulate.

The instruction format of our system can be represented as below.

gen_vec-type

Opcode (6 bits)	Reserved (22 bits)	N (4 bits)
-----------------	--------------------	------------

1Q-type

Opcode (6 bits)	Target (5 bits)	Reserved (17 bits)	N (4 bits)
-----------------	-----------------	--------------------	------------

1Q1C-type

Opcode (6 bits)	Target (5 bits)	Reserved (10 bits)	Angle (7 bits)	N (4 bits)
-----------------	-----------------	--------------------	----------------	------------

2Q-type

Opcode (6 bits)	Target (5 bits)	Ctrl0 (5 bits)	Reserved (12 bits)	N (4 bits)
-----------------	-----------------	----------------	--------------------	------------

2Q1C-type

Opcode (6 bits)	Target (5 bits)	Ctrl0 (5 bits)	Reserved (5 bits)	Angle (7 bits)	N (4 bits)
-----------------	-----------------	----------------	-------------------	----------------	------------

3Q-type

Opcode (6 bits)	Target (5 bits)	Ctrl0 (5 bits)	Ctrl1 (5 bits)	Reserved (7 bits)	N (4 bits)
-----------------	-----------------	----------------	----------------	-------------------	------------

Representation of data in our system

In general, data representation of a quantum state is represented by vector of complex number and a quantum gate is represented by matrix of complex number. However, in hardware it cannot represent data as 2 dimensions and complex number. Hence, these data must be represented by 1 dimension of fixed-point number of real part and imaginary part stick together as in figure 4.

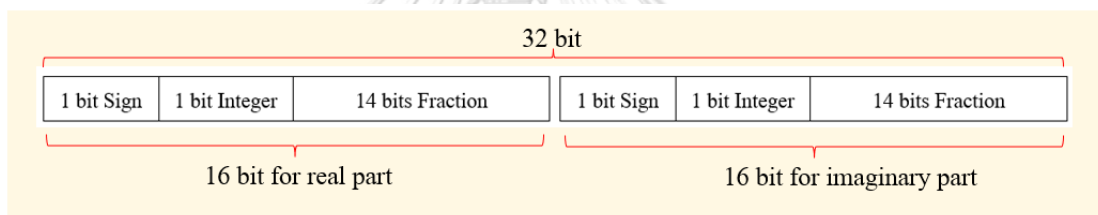


Figure 4 Fixed-point number that represent complex number

In fixed-point number, 1 bit is used to represent sign of the value and 1 bit is used to represent an integer because the amplitude of quantum state and the value in quantum gate is not over 1. And the rest of fixed-point (14 bits) are enough for using to represent the fraction part. So, the 16 bits fixed-point number is sufficient to represent the value of each part.

System design

Our system design contains 3 main parts that is processor, generator and co-processor. Overall, of our system can be shown as figure 5.

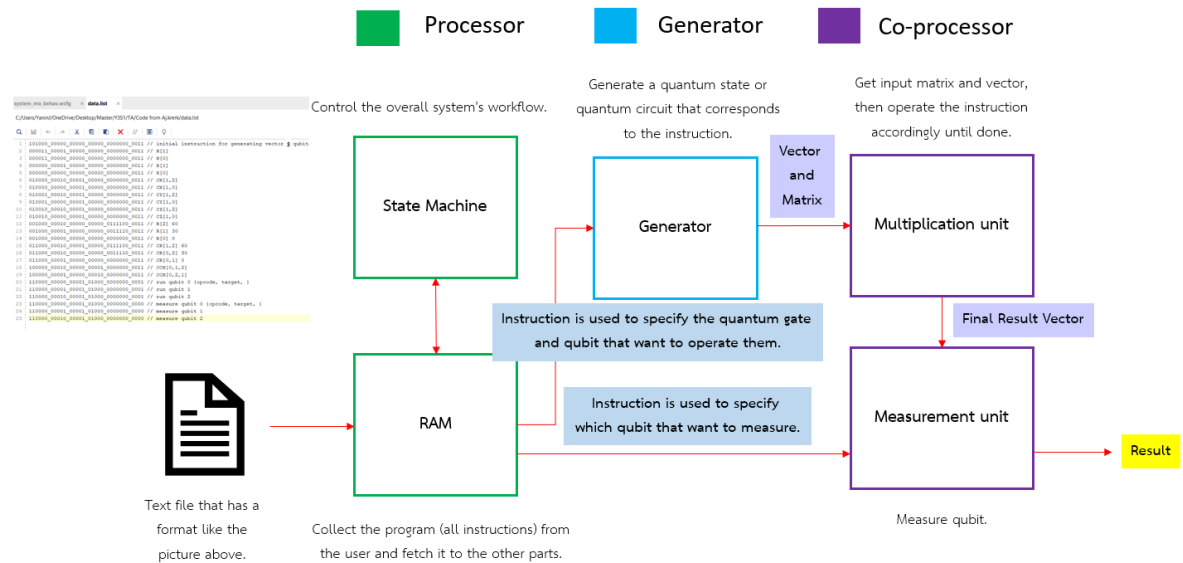


Figure 5 System design of our work

Signal

Before we get into the details of each module, these modules have the relevant signals that are used commonly for controlling it. As shown in this table 4.

Signal	Function
clock	Determine the timing of each module.
nreset	Determine the non-reset (nreset=1) or reset (nreset=0) of each module.
start_flag	Determine the starting of each module.
done_flag	Determine the completion of each module.

Table 4 Relevant signal of our system

Processor

Processor is a part that put the state machine and memory together. It controls and manages overall of system. Function of each part can be delineated as follows.

State machine

State machine is a heart for controlling workflow of our system. In our work, there is a state machine that is different type depends on the functionality of each module in our system.

For the first module is a system module is shown as in figure 6. It has seven states: "IDLE", "FETCH", "GENVEC", "MEASURE", "GENMAT", "CALCULATE" and "DONE". If this module receives a system-start signal from user, it will change its state from "IDLE" to "FETCH." After that, an instruction from memory is read and checked its opcode. If it equal to $6'b101100$ a next state is assigned as "GENVEC" then the generator vector module will start. But if the opcode is $6'b110000$ the next state is assigned as "MEASURE" then the measure module will start. Otherwise, If the opcode equal to other values the next state is assigned as "GENMAT" then the generator matrix module will start. Afterwards, when it's done, the next state is changed to "CALCULATE" for starting vector-matrix multiplication. Finally, when "GENVEC" or "MEASURE" or "CALCULATE" are finished, it sends a signal to the system's state machine telling it to change the state to "done," and then it returns to "idle."

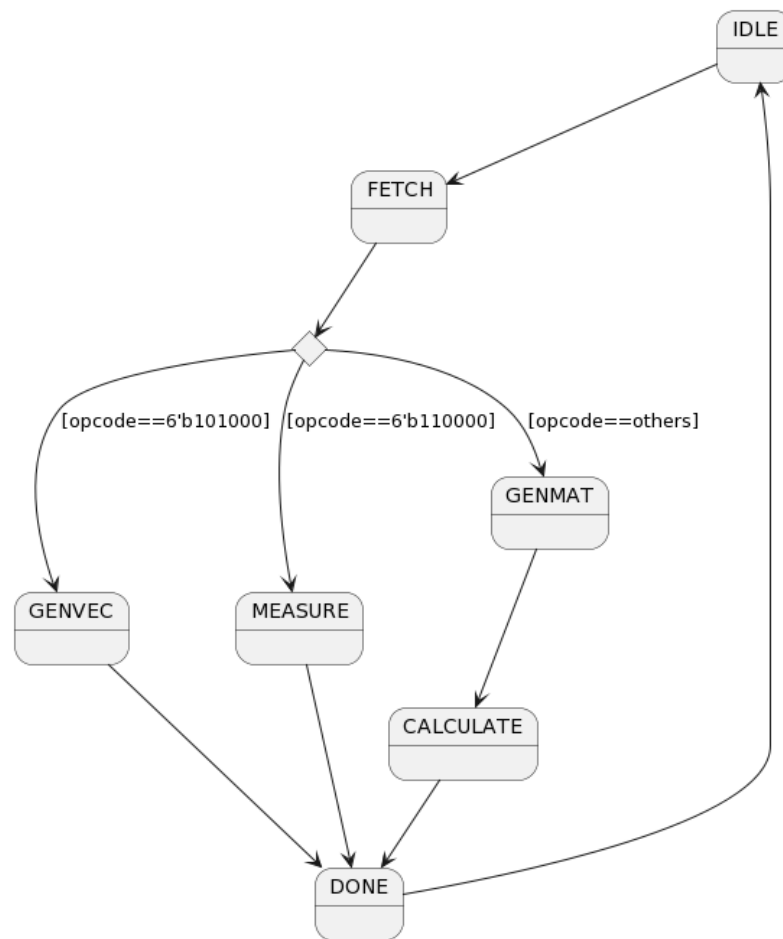


Figure 6 State machine of system module

Second module is a group of generator matrix module. The word “group” refers to, there are five types that depend on the input of these modules. These differences are discussed in the section of instruction format. Although these modules have differences, they have the same workflow. Hence, there are the same state machine. It has three states:” idle,”” busy,” and” done” that is shown as in figure 7. If this module receives a system-start signal, it will change its state from” idle” to” busy.” This module generating matrix follows proposition I when the state is” busy.” When it is finished, it sends a signal to the state machine telling it to change the state to” done,” and then it returns to” idle.”

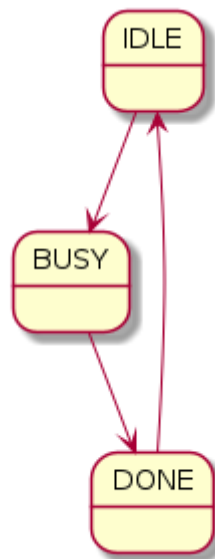


Figure 7 State machine of generator matrix module and vector-matrix multiplication module

Next module is a vector-matrix multiplication module. It has three states is same as the state machine of generator matrix module. For working of this state machine looks as follows. If this module receives a system-start signal, it will change its state from "idle" to "busy." Then, this module starts to multiply matrix and vector input. When it has finished, it sends a signal to the state machine to change the state to "done," and then it returns to "idle."

Last module is measurement module as shown in figure 8. It has four states: "idle", "cal_prob", "clear" and "done". Normally, the state is "idle" after that if the start-to-measure signal comes to this module, the state will be changed to "cal_prob" then vector input is calculated probability. Then, when it has finish calculating, the state is changed to "clear" to clear the value of all registers and changed to "done" and then it returns to "idle."

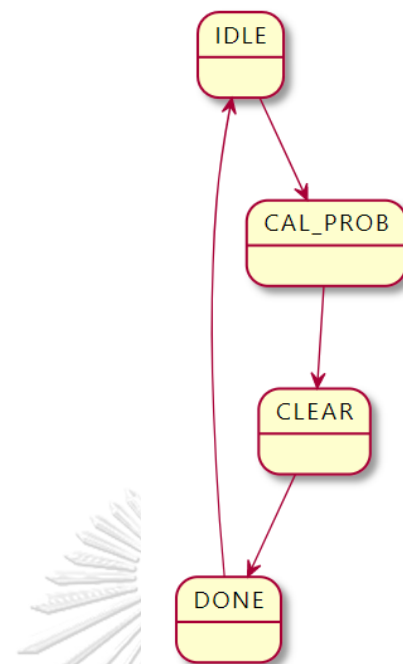


Figure 8 State machine of measurement module

Memory

Memory is used for writing an instruction from user to our system and read or fetch instruction to the system. Thus, random access memory (RAM) that can be read and written data is used as memory in our work.

This module has clock, address, instruction (when writing) as input and done flag, instruction (when reading) as output.

Generator

Generator part is a part that generates vector and matrix corresponding with an instruction. These vector and matrix are used for input of multiplication unit to transform quantum state. There are 2 types of this generator:

Vector generator

For the first line of every set of instruction must be the instruction declare a number of qubits and command our system to generate initial vector of complex number size 2^n . Vector generator is a module that is used for this task as shown in figure 9.

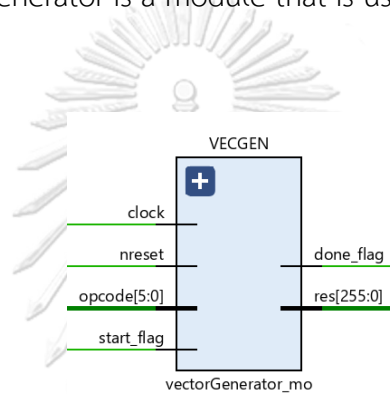


Figure 9 Vector generator module

The input of this module is clock, nreset (non-reset), opcode and start_flag and the output are done_flag and res (result) that is the vector of complex number.

Matrix generator

Matrix generator module is used to generate a matrix of quantum operator in our system. This module can be divided this into 5 types according to instruction format. In each type, there are differences input that is shown in figure 10. All modules have similar output that is done_flag and res (result). The result represents the matrix of complex number.

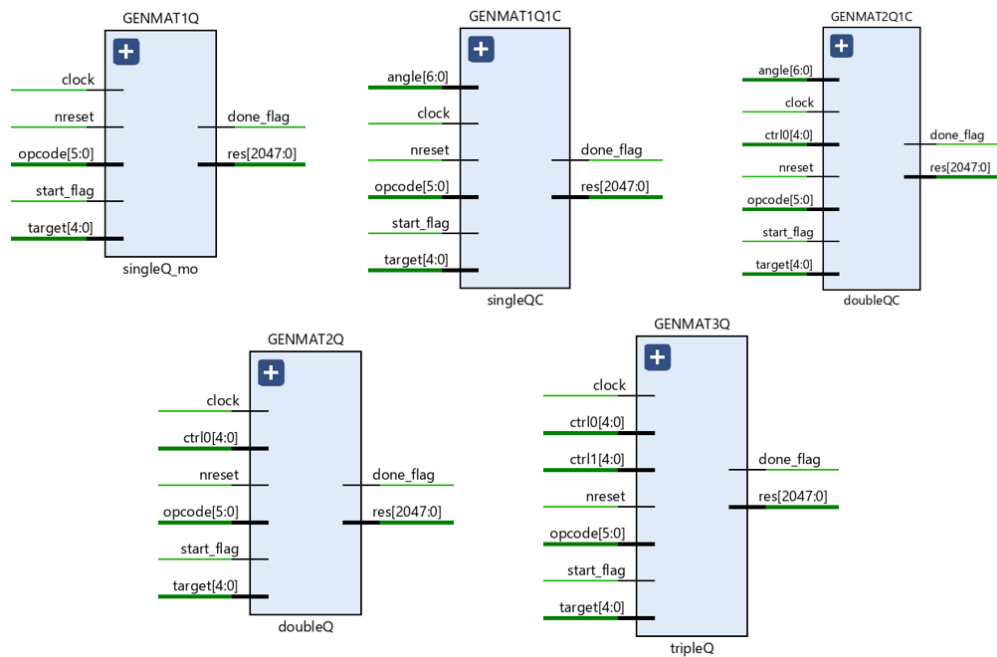


Figure 10 Matrix generator module

Co-processor

The main function of co-processor is about arithmetic operation. This part includes a vector-matrix multiplication module and a measurement module.

Vector-Matrix multiplication module

Vector-Matrix multiplication module is used for multiplication vector of quantum state and matrix of quantum operator. In addition, this multiplication is a complex number that is represented in fixed-point number format. This number is between $[-1,1]$ because the probability of quantum state no more than 1. As illustrated in the figure 11 it has vector size 2^n -by-1 and matrix size 2^n -by- 2^n be as an input and result size 2^n -by-1 be as an output.

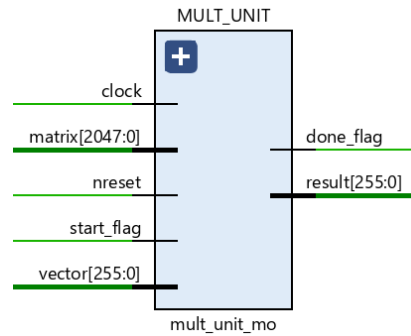


Figure 11 Vector-Matrix multiplication module

Inside this module, it has 4 QMULT and 4 QADD for using in multiplication and addition between each element of vector and matrix as shown in figure 12. To describe how it works, we assumed that it has one vector v , which has size 2-by-1 in equation 22, and one matrix m , which has size 2-by-2 in equation 23, are input of our module.

$$v = \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} v_0^{re} + jv_0^{im} \\ v_1^{re} + jv_1^{im} \end{bmatrix} \quad (22)$$

$$m = \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} = \begin{bmatrix} m_{00}^{re} + jm_{00}^{im} & m_{01}^{re} + jm_{01}^{im} \\ m_{10}^{re} + jm_{10}^{im} & m_{11}^{re} + jm_{11}^{im} \end{bmatrix} \quad (23)$$

If we calculate $m \cdot v$, we will get an output vector o , which has size 2-by-1 by using calculation as in equation 24.

$$o = m \cdot v = \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} m_{00}v_0 + m_{01}v_1 \\ m_{10}v_0 + m_{11}v_1 \end{bmatrix} \quad (24)$$

From figure 12, the QADD number 3 and 4 are used for accumulating the result from multiplication of each row ($m_{00}v_0, m_{01}v_1$ in first row and $m_{10}v_0, m_{11}v_1$ in second row) in real part and imaginary part respectively. The RTL_REG_SYNC is used for collecting the temporary out output from QADD 3 and 4 when it has not finished adding. Next, we focused in multiplication between a multiplier and a multiplicand of each element such as $m_{00}v_0$ that refers to complex number multiplication as $(m_{00}^{re} + jm_{00}^{im}) \cdot (v_0^{re} + jv_0^{im})$. It has multiplication 2 times for real part of complex number that is $m_{00}^{re} \cdot v_0^{re}$, $m_{00}^{im} \cdot v_0^{im}$, and others for imaginary part that is $m_{00}^{re} \cdot v_0^{im}$,

$m_{00}^{im} \cdot v_0^{re}$. Thus, QMULT number 1 and 2 are used for multiplication this real part and QMULT number 3 and 4 are used for the rest part. In addition, the QMULT has an i input it will assign to 1 when the multiplier and the multiplicand are both imaginary then flip a sign of output of it (from + to - or from - to +). For others case, i is set to 0 and the sign is not flipped. The output of QMULT number 1 and 2, which is real part, and number 3 and 4, which is imaginary part, are added by QADD number 1 and 2 respectively. Finally, the output of that QADD is accumulated by QADD number 3 and 4.

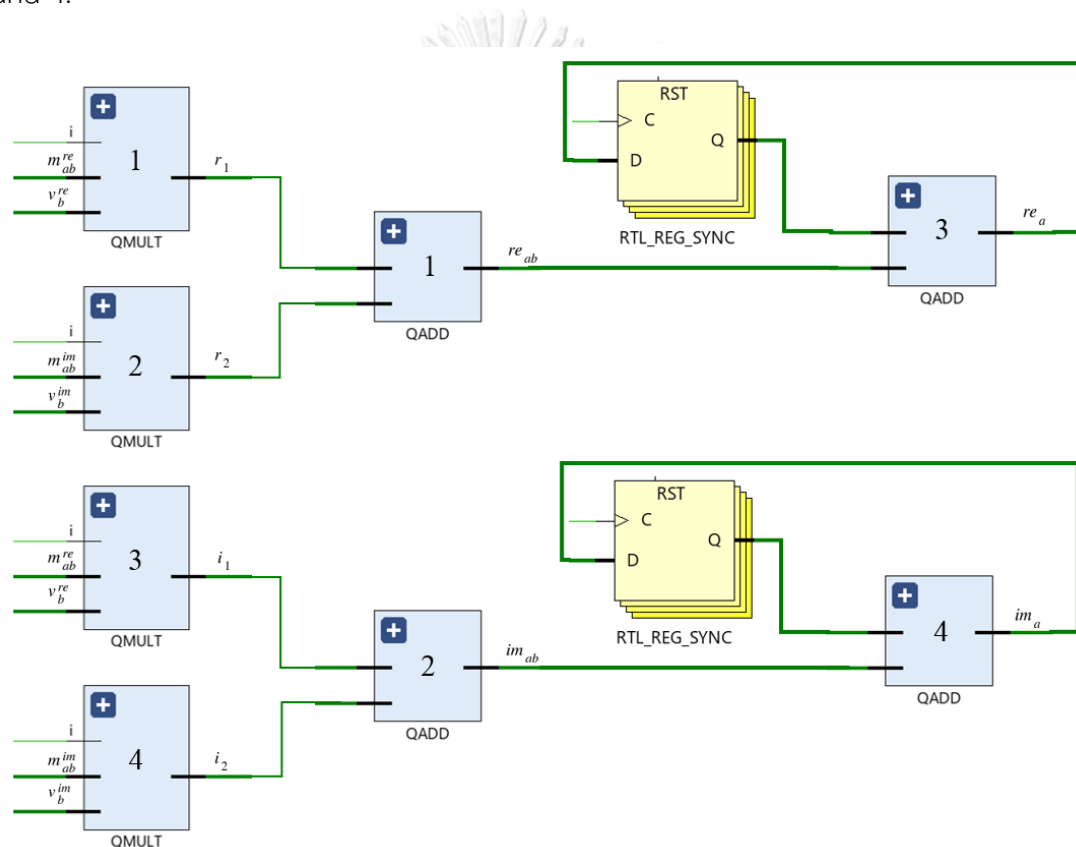


Figure 12 Design of vector-matrix multiplication module

Measurement module

Measurement module is used for measuring the final-quantum state, which is matrix of vector state that are transformed by quantum circuit. As shown in figure 13, this module has clock, nreset, startp_flag and vector as input. The startp_flag is used to trigger calculating probability and the vector is the final-quantum state.

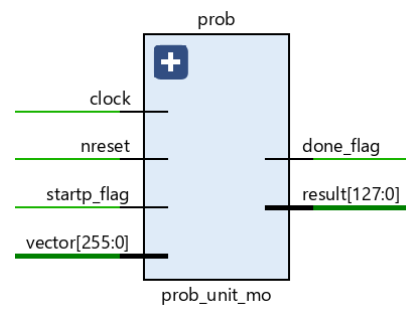


Figure 13 Design of measurement module



Results

In this chapter, we will show the result of our simulation. By starting with the verification of our system. Next, the result of each proposition is represented. Finally, the runtime of the simulation is compared with the baseline.

Verification of the simulation

In this part, the verification of our modules in system are shown. There are three parts that must be checked for the correctness of the work.

Before we consider our verification results, we need to understand the actual data format of each part's output. The output is binary (fixed-point representation) as described in section of data representation. For example, if we want to create quantum circuit that has 3 qubits (0, 1, 2) and operate the H gate on 0th qubit. The process will be in the following 4 steps. First, the vector generator will create vector of complex number with the size of 2^3 -by-1. Second, the matrix generator will create matrix of complex number with the size 2^3 -by- 2^3 . Afterwards, the vector and the matrix will be multiplied by using vector-matrix multiplication module that get the vector of complex number with the size 2^3 -by-1. Finally, for measurement step the measurement module will find the probability of each quantum state (each element of the final-state vector) and then get the output as a vector of real number with the size of 2^3 -by-1. Hence, the actual output from each module at each step can be shown as in Table 5.

Step	Module	Input	Output																																
1 st Create initial vector of complex number	Vector generator	Instruction from memory (32 bits)	<p>Initial vector state of complex number in fixed-point representation.</p> <p>Output in hexadecimal form is short than binary.</p> <table border="1" data-bbox="544 504 930 683" style="margin-left: auto; margin-right: auto;"> <tr><td>00000000</td><td>7</td></tr> <tr><td>00000000</td><td>6</td></tr> <tr><td>00000000</td><td>5</td></tr> <tr><td>00000000</td><td>4</td></tr> <tr><td>00000000</td><td>3</td></tr> <tr><td>00000000</td><td>2</td></tr> <tr><td>00000000</td><td>1</td></tr> <tr><td>40000000</td><td>0</td></tr> </table> <p>Output in binary form is easy to under interpret when compare with fixed-point representation format.</p> <table border="1" data-bbox="1038 338 1382 846" style="margin-left: auto; margin-right: auto;"> <tr><td>00000000000000000000000000000000</td><td>7</td></tr> <tr><td>00000000000000000000000000000000</td><td>6</td></tr> <tr><td>00000000000000000000000000000000</td><td>5</td></tr> <tr><td>00000000000000000000000000000000</td><td>4</td></tr> <tr><td>00000000000000000000000000000000</td><td>3</td></tr> <tr><td>00000000000000000000000000000000</td><td>2</td></tr> <tr><td>00000000000000000000000000000000</td><td>1</td></tr> <tr><td>01000000000000000000000000000000</td><td>0</td></tr> </table>	00000000	7	00000000	6	00000000	5	00000000	4	00000000	3	00000000	2	00000000	1	40000000	0	00000000000000000000000000000000	7	00000000000000000000000000000000	6	00000000000000000000000000000000	5	00000000000000000000000000000000	4	00000000000000000000000000000000	3	00000000000000000000000000000000	2	00000000000000000000000000000000	1	01000000000000000000000000000000	0
00000000	7																																		
00000000	6																																		
00000000	5																																		
00000000	4																																		
00000000	3																																		
00000000	2																																		
00000000	1																																		
40000000	0																																		
00000000000000000000000000000000	7																																		
00000000000000000000000000000000	6																																		
00000000000000000000000000000000	5																																		
00000000000000000000000000000000	4																																		
00000000000000000000000000000000	3																																		
00000000000000000000000000000000	2																																		
00000000000000000000000000000000	1																																		
01000000000000000000000000000000	0																																		

<p>2nd Create matrix of complex number H[0]</p>	<p>Matrix generator</p>	<p>Instruction from memory (32 bits)</p>	<p>Matrix of complex number in fixed-point representation.</p> $\begin{bmatrix} ad410000 & 2d410000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 \\ 2d410000 & 2d410000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 \\ 00000000 & 00000000 & ad410000 & 2d410000 & 00000000 & 00000000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 2d410000 & 2d410000 & 00000000 & 00000000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 00000000 & 00000000 & ad410000 & 2d410000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 00000000 & 00000000 & 2d410000 & 2d410000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & ad410000 & 2d410000 \\ 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 2d410000 & 2d410000 \end{bmatrix}$
<p>3rd Multiply matrix with vector</p>	<p>Matrix-Vector Multiplication</p>	<p>Vector of state and Matrix of gate</p>	<p>Vector of complex number in fixed-point representation that passed the H[0]</p> $\begin{bmatrix} 00000000 \\ 00000000 \\ 00000000 \\ 00000000 \\ 00000000 \\ 00000000 \\ 2d410000 \\ 2d410000 \end{bmatrix}$

4 th Measure probability	Measurement	Vector of final state	Vector of real number in fixed-point representation
			$\begin{bmatrix} 0000 & 7 \\ 0000 & 6 \\ 0000 & 5 \\ 0000 & 4 \\ 0000 & 3 \\ 0000 & 2 \\ 1fff & 1 \\ 1fff & 0 \end{bmatrix}$

Table 5 Representation of actual output of 4 modules in our system

From the example above, we notice that outputs are hexadecimal representation. Moreover, the position of each element in the output is sorted in descending order from left to right and top to bottom. Therefore, for the rest of the results, we convert all outputs in this form to a new form that is a decimal representation and reverse the order of elements for ease of reading the result.

Verification of generator module

The verification result of generator module is described in this section. It contains 2 parts of generator module. The result of vector generator module is shown as in Table 6 and the result of matrix generator modules are shown as in Table 7-9.

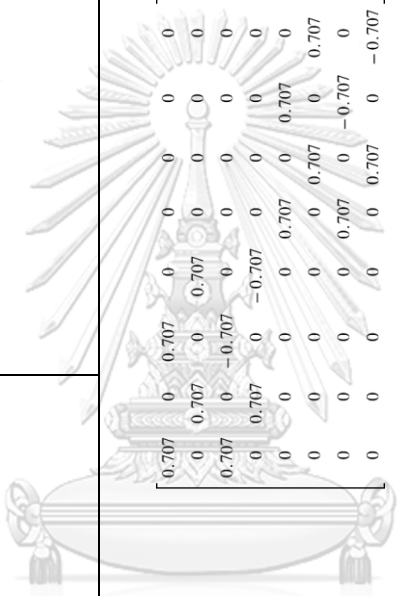
Verification of vector generator module

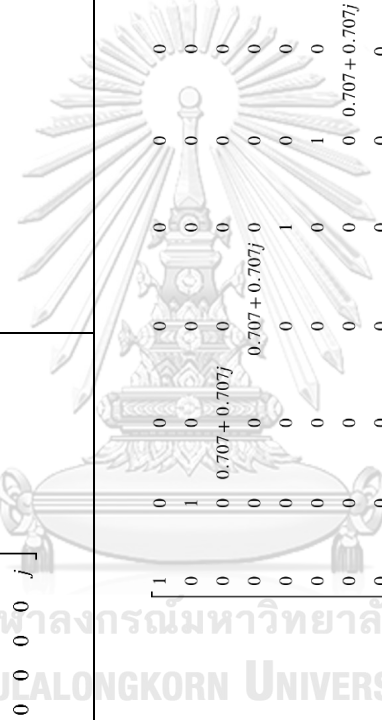
Number of qubits	Expected result	Simulation result
3	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} (1 + 0j) \\ 0j \\ 0j \\ 0j \\ 0j \\ 0j \\ 0j \\ 0j \end{bmatrix}$
4	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} (1 + 0j) \\ 0j \end{bmatrix}$

Table 6 Verification of vector generator module

Verification of matrix generator module

Quantum gate	Expected result	Simulation result
X[1]	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \end{bmatrix}$
Y[1]	$\begin{bmatrix} 0 & 0 & -j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0 \\ j & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j \\ 0 & 0 & 0 & 0 & j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & j & 0 \end{bmatrix}$	$\begin{bmatrix} 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \end{bmatrix}$

<p>Z[1]</p>	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (-1+0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (-1+0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (-1+0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (-1+0j) \end{bmatrix}$
<p>H[1]</p>	<p>Expected result</p> $\begin{bmatrix} 0.707 & 0 & 0.707 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.707 & 0 & 0.707 & 0 & 0 & 0 & 0 \\ 0.707 & 0 & -0.707 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.707 & 0 & -0.707 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.707 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.707 & 0 & 0.707 \\ 0 & 0 & 0 & 0 & 0 & 0.707 & 0 & -0.707 \\ 0 & 0 & 0 & 0 & 0 & 0.707 & 0 & -0.707 \end{bmatrix}$	
<p>Simulation result</p>	$\begin{bmatrix} (0.70709228515625 + 0j) & 0j & (0.70709228515625 + 0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & (0.70709228515625 + 0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ (0.70709228515625 + 0j) & 0j & (-0.70709228515625 + 0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (-0.70709228515625 + 0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (0.70709228515625 + 0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (0.70709228515625 + 0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (-0.70709228515625 + 0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (-0.70709228515625 + 0j) \end{bmatrix}$	$\begin{bmatrix} 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (0.70709228515625 + 0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (-0.70709228515625 + 0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (0.70709228515625 + 0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (0.70709228515625 + 0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (-0.70709228515625 + 0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (-0.70709228515625 + 0j) \end{bmatrix}$

<p>S[1]</p>	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix}$	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 1j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 1j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 1j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & 1j \end{bmatrix}$
<p>T[1]</p>	<p>Expected result</p> $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.707+0.707j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.707+0.707j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.707+0.707j \end{bmatrix}$	
<p>Simulation result</p>	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (0.70709228515625 + 0.70709228515625j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (0.70709228515625 + 0.70709228515625j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (0.70709228515625 + 0.70709228515625j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (0.70709228515625 + 0.70709228515625j) \end{bmatrix}$	

<p>R[1] 45</p>	<p>Expected result</p> $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.707+0.707j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.707+0.707j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.707+0.707j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.707+0.707j \end{bmatrix}$
<p>Simulation result</p> $\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (0.70709228515625+0.70709228515625j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (0.70709228515625+0.70709228515625j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (0.70709228515625+0.70709228515625j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (0.70709228515625+0.70709228515625j) \end{bmatrix}$	

Table 7 Verification of 1-qubit gate generator module

Quantum gate	Expected result	Simulation result
CX[2,0]	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) \end{bmatrix}$
CY[2,0]	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -j & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix}$	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & -1j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) \end{bmatrix}$

Quantum gate	Expected result	Simulation result
CCX[0,2,1]	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) & 0j \\ 0j & 0j & 0j & 0j & 0j & 0j & 0j & (1+0j) \end{bmatrix}$

Table 9 Verification of 3-qubit gate generator module



Verification of Vector-Matrix Multiplication module

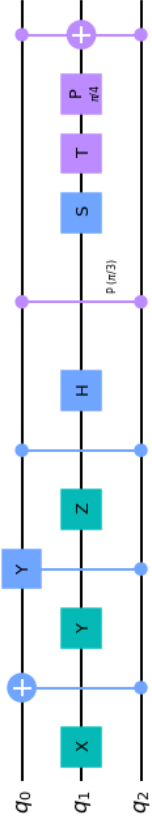
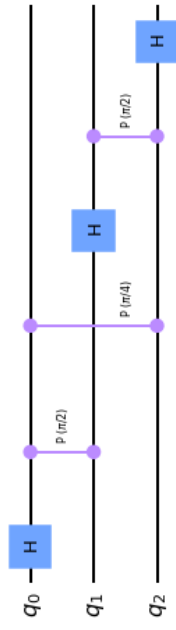
Quantum circuit	Expected result	Simulation result
<p style="text-align: center;">All-gate</p> 	$\begin{bmatrix} -0.707j \\ 0 \\ 5.55 \times 10^{-15} + 0.707j \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -0.70709228515625j \\ 0j \\ 0.7069091796875j \\ 0j \\ 0j \\ 0j \\ 0j \\ 0j \end{bmatrix}$
<p style="text-align: center;">Quantum Fourier Transform of 3 qubits (QFT 3)</p> 	$\begin{bmatrix} 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \end{bmatrix}$	$\begin{bmatrix} (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \end{bmatrix}$

Table 10 Verification of Vector-Matrix multiplication module

we got the result that didn't match with the baseline. Therefore, the expected result is recalculated by hand. To check if the results from our simulations are correct or not. As a result, our result matched with this calculation. Note that this result is shown in appendix section. Thus, we concluded that our system can simulate quantum computing correctly.

Result of propositions

In this part, the result of our propositions is shown. Before we consider our proposition results, we need to know about the experiment setup. The propositions are modelled in VerilogHDL using a clock speed of 2.00 GHz. The Qiskit, a quantum simulation software library, was chosen as a baseline. This library runs on Colab with clock speed 2.20 GHz.

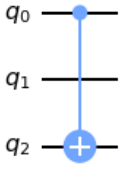
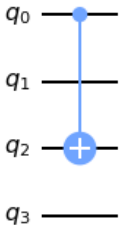
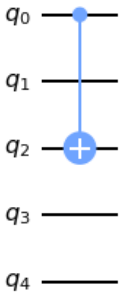
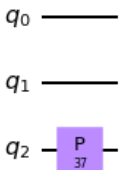
For our baseline, it didn't spend the same amount of runtime each time. Therefore, its runtime, as shown in the result, is an average of run time 10 times by cutting out the min and max value.

Result of Proposition I: Describes the relationship within the operator's matrix rather than directly calculating the tensor product

In this section, we design an experiment to compare how much time the software simulation with our work spends building the tensor product of each quantum gate.

To reduce the overlap measurement of propositions I and II, there are four different types of quantum gates in the experiment. The single qubit gate and the control single gate, which are not involved with angles (sine and cosine values), are used to demonstrate the result of proposition I. The others are used to demonstrate the result of proposition II.

Type of gate	Number of qubits	Qiskit (s)	Our work (s)	Speedup
Single qubit gate H(2)	3 q_0 ——— q_1 ——— q_2 — H —	2.97×10^{-4}	5.13×10^{-8}	5.79×10^3
	4 q_0 ——— q_1 ——— q_2 — H — q_3 ———	3.09×10^{-4}	1.47×10^{-7}	2.10×10^3
	5 q_0 ——— q_1 ——— q_2 — H — q_3 ——— q_4 ———	3.10×10^{-4}	5.31×10^{-7}	5.84×10^2
Control single	3	2.72×10^{-4}	5.13×10^{-8}	5.30×10^3

qubit gate CX(0,2)				
	4 	2.69×10^{-4}	1.47×10^{-7}	1.83×10^3
	5 	2.32×10^{-4}	5.31×10^{-7}	4.37×10^2
Phase gate P(37,2)	3 	2.87×10^{-4}	5.13×10^{-8}	5.59×10^3
	4	2.62×10^{-4}	1.47×10^{-7}	1.78×10^3

	<p>5</p>	2.86×10^{-4}	5.31×10^{-7}	5.39×10^2
Control phase gate CP(60, 0, 2)	<p>3</p>	2.77×10^{-4}	5.13×10^{-8}	5.40×10^3
	<p>4</p>	2.83×10^{-4}	1.47×10^{-7}	1.93×10^3
	<p>5</p>	2.76×10^{-4}	5.31×10^{-7}	5.20×10^2

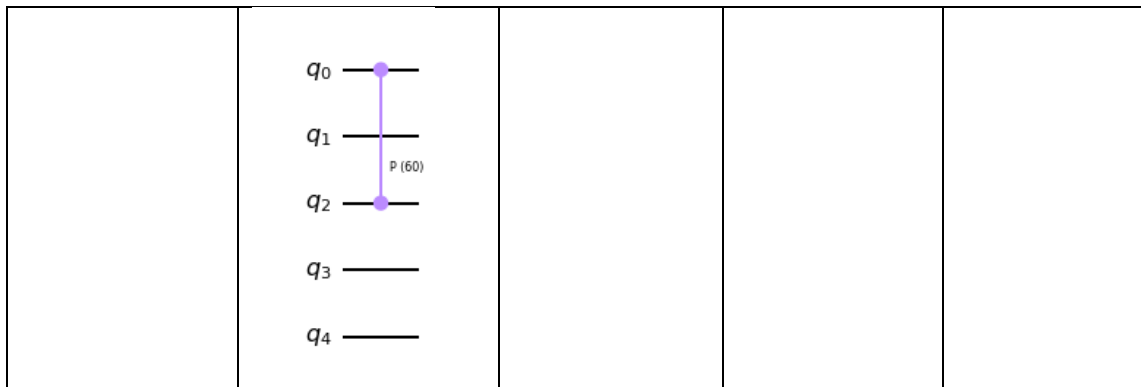


Table 12 Comparison of runtime between software simulation and our work

In table 12, our propose method takes less time to run than those of software simulation. As a result of proposition I, we can generate a matrix operator by assignment through the if-else statement method rather than calculating the tensor product directly. This allows the generation of a matrix step in one clock cycle from 15.750 ns to 16.250 ns as shown in Fig 14. The resources required is 2^{2n} , where n is the number of qubits in the circuit.

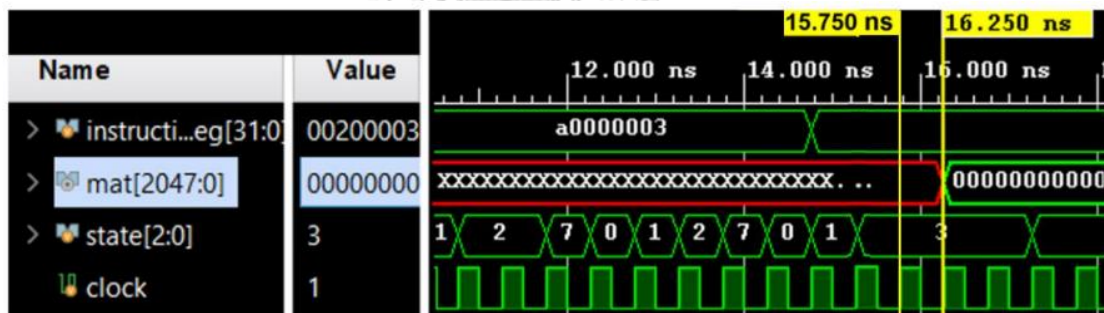


Figure 14 Simulation result for generating operator step in 1 clock cycle

Table 13 represents the meanings of all the states in our system. Fig. 15 shows the timing diagram of the state machine of our system when it was simulating that it contains states that follow the instruction. If we zoom out this timing diagram as Fig. 16, we will find that almost all time is spent on state 4, or the matrix-vector multiplication step.

Number	State
0	IDLE
1	FETCH

2	GENVEC (generate vector)
3	GENMAT (generate matrix)
4	CALCULATE (matrix-vector multiplication)
5	GENRAND (generate random number)
6	MEASURE (find probability of final state)
7	DONE

Table 13 Meaning of all states in our system

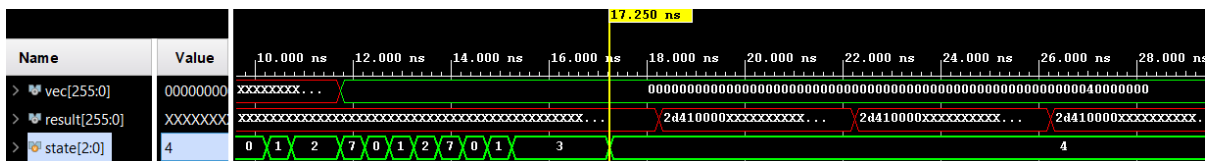


Figure 15 Timing diagram represent state machine when simulation (Zoom in)

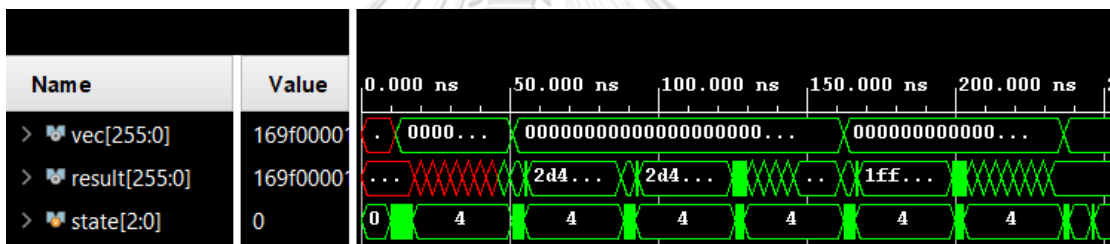


Figure 16 Timing diagram represent state of state machine when simulation (Zoom out)

Result of Proposition II: Use a lookup table to collect sine and cosine values instead of calculating them directly

Proposition II is yet another reason why a matrix step can be generated in a single clock cycle. Because it reduces the time it takes to access sine and cosine values.

To compare the simulation time of our work with software simulation when simulate Quantum Fourier Transform (QFT) at 3, 5, 7 qubits that is a quantum algorithm. Table 14 is shown the simulation time of both.

Quantum circuit	Qiskit run on colab @2.2 GHz (s)	Our work @2 GHz (s)	Speedup
QFT 3 qubits	3.18×10^{-4}	2.24×10^{-7}	1.42×10^3
QFT 5 qubits	4.83×10^{-4}	7.79×10^{-6}	6.20×10
QFT 7 qubits	3.99×10^{-4}	2.30×10^{-4}	1.73

Table 14 Simulation time comparison of our work and software simulation

If we simulate these circuit by our work with the same time spent as qiskit's, we will use the clock speed in each case as shown in table 15.

Quantum circuit	Simulation time (s)	Qiskit	Our work
QFT 3 qubits	3.18×10^{-4}	2.2 GHz	1.41 MHz
QFT 5 qubits	4.83×10^{-4}	2.2 GHz	32.2 MHz
QFT 7 qubits	3.99×10^{-4}	2.2 GHz	1.15 GHz

Table 15 Clock speed comparison of our work and software simulation

The clock speed in this table can be calculated from (22). From the result in table 15. Our work takes less clock speed than qiskit at the same time.

$$\text{clock speed (Hz)} = \frac{\text{simulation time of our work (s)} \times \text{old clock speed of our work (Hz)}}{\text{simulation time of qiskit (s)}} \quad (22)$$

Conclusion and Future Work

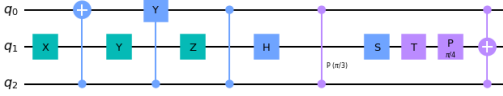
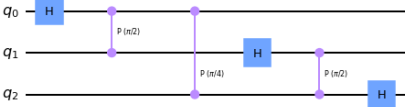
Based on these results, we conclude that optimizing the construction of tensor products using our proposed if-else method can significantly reduce the execution time of quantum computing simulation. However, the number of qubits and quantum gates in the circuit will increase the runtime. This is due to the fact that the multiplication vector-matrix step is still not optimized in this experiment.

Moreover, due to hardware resource constraints, we intend to modify this module to support the generation of quantum operators for larger quantum circuits. Furthermore, the matrix-vector multiplication step should also be redesigned to gain better performance.

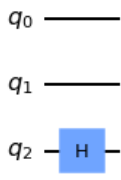
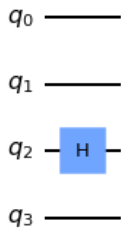

We plan to modify our system by increasing the number of matrix-vector multiplication module in order to multiply vector and matrix in parallel. This will increase the speed of our simulation. In addition, the matrix-vector multiplication module and matrix generator module should be concurrent working for reduce the space that is used to collect the output of matrix generator module to increase the space for supporting larger circuit.

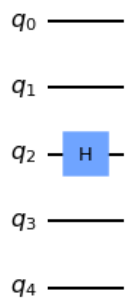
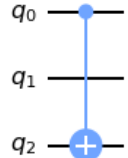
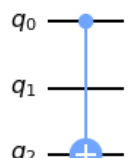
Appendix

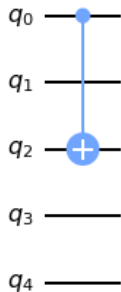
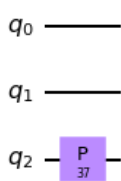
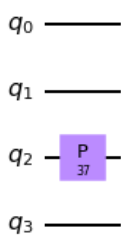
This table represent the calculation of each algorithm by hand.

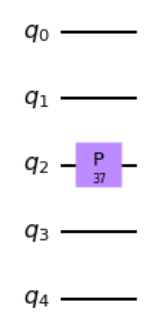
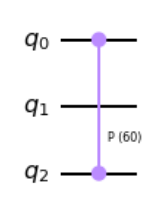
Quantum circuit	Calculation by hand	Simulation result
<p style="text-align: center;">All-gate</p> 	$\begin{bmatrix} -0.707j \\ 0 \\ 0.707j \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -0.70709228515625j \\ 0j \\ 0.7069091796875j \\ 0j \\ 0j \\ 0j \\ 0j \\ 0j \end{bmatrix}$
<p style="text-align: center;">Quantum Fourier Transform of 3 qubits (QFT 3)</p> 	$\begin{bmatrix} 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \\ 0.35355339 \end{bmatrix}$	$\begin{bmatrix} (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \\ (0.35345458984375 + 0j) \end{bmatrix}$

This table represent the raw data of run time of Qiskit.

Type of gate	Number of qubits	Round	Run time of Qiskit (s)
Single qubit gate H(2)	3 	1	4.24×10^{-4}
		2	3.60×10^{-4}
		3	3.00×10^{-4}
		4	3.10×10^{-4}
		5	2.90×10^{-4}
		6	2.89×10^{-4}
		7	2.88×10^{-4}
		8	2.66×10^{-4}
		9	2.71×10^{-4}
		10	2.66×10^{-4}
Single qubit gate H(2)	4 	1	4.04×10^{-4}
		2	4.04×10^{-4}
		3	2.93×10^{-4}
		4	2.95×10^{-4}
		5	2.84×10^{-4}
		6	2.66×10^{-4}
		7	2.70×10^{-4}
		8	3.05×10^{-4}
		9	2.96×10^{-4}
		10	3.27×10^{-4}
Single qubit gate H(2)	5 	1	4.40×10^{-4}
		2	3.95×10^{-4}
		3	3.02×10^{-4}
		4	2.80×10^{-4}
		5	2.94×10^{-4}

		6	3.06×10^{-4}	
		7	2.98×10^{-4}	
		8	2.85×10^{-4}	
		9	2.77×10^{-4}	
		10	3.23×10^{-4}	
Control single qubit gate CX(0,2)	3		1	3.74×10^{-4}
		2	3.91×10^{-4}	
		3	3.90×10^{-4}	
		4	5.55×10^{-4}	
		5	2.42×10^{-4}	
		6	2.04×10^{-4}	
		7	1.93×10^{-4}	
		8	1.93×10^{-4}	
		9	1.88×10^{-4}	
		10	1.86×10^{-4}	
	4		1	3.73×10^{-4}
		2	3.18×10^{-4}	
		3	2.99×10^{-4}	
		4	2.58×10^{-4}	
		5	2.34×10^{-4}	
		6	2.58×10^{-4}	
		7	2.56×10^{-4}	
		8	2.54×10^{-4}	
		9	2.53×10^{-4}	
		10	2.56×10^{-4}	

	5		1	2.79×10^{-4}
			2	2.27×10^{-4}
			3	1.99×10^{-4}
			4	1.96×10^{-4}
			5	1.93×10^{-4}
			6	2.86×10^{-4}
			7	2.35×10^{-4}
			8	2.08×10^{-4}
			9	2.97×10^{-4}
			10	2.29×10^{-4}
Phase gate P(37,2)	3		1	3.69×10^{-4}
			2	3.11×10^{-4}
			3	2.76×10^{-4}
			4	2.83×10^{-4}
			5	2.79×10^{-4}
			6	2.55×10^{-4}
			7	2.54×10^{-4}
			8	2.57×10^{-4}
			9	2.71×10^{-4}
			10	5.05×10^{-4}
	4		1	2.95×10^{-4}
			2	3.38×10^{-4}
			3	2.77×10^{-4}
			4	2.62×10^{-4}
			5	2.55×10^{-4}
			6	3.53×10^{-4}
			7	2.41×10^{-4}
			8	2.07×10^{-4}

		9	2.06×10^{-4}
		10	2.23×10^{-4}
		1	3.82×10^{-4}
		2	3.76×10^{-4}
		3	3.91×10^{-4}
		4	2.70×10^{-4}
		5	2.68×10^{-4}
		6	2.56×10^{-4}
		7	2.45×10^{-4}
		8	2.44×10^{-4}
Control phase gate CP(60, 0, 2)		1	4.01×10^{-4}
		2	3.46×10^{-4}
		3	3.24×10^{-4}
		4	2.84×10^{-4}
		5	3.45×10^{-4}
		6	2.35×10^{-4}
		7	2.34×10^{-4}
		8	2.22×10^{-4}
		9	2.15×10^{-4}
		10	2.28×10^{-4}
	4	1	4.05×10^{-4}
		2	3.27×10^{-4}
		3	3.02×10^{-4}
		4	2.91×10^{-4}
		5	2.79×10^{-4}
		6	2.63×10^{-4}

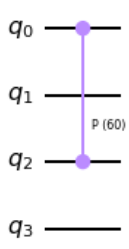
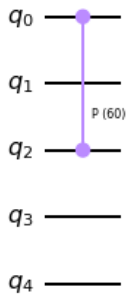
	7	2.63×10^{-4}	
	8	2.69×10^{-4}	
	9	2.68×10^{-4}	
	10	2.65×10^{-4}	
5		1	3.92×10^{-4}
2		2.95×10^{-4}	
3		2.75×10^{-4}	
4		2.93×10^{-4}	
5		2.91×10^{-4}	
6		2.60×10^{-4}	
7		2.64×10^{-4}	
8		3.01×10^{-4}	
9		2.25×10^{-4}	
10		2.31×10^{-4}	

Table 16 Raw data of run time of Qiskit.

REFERENCES

- [1] N. S. Yanofsky and M. A. Mannucci, “Quantum Computing for Computer Scientists,” p. 402.
- [2] E. Gibney, “Hello quantum world! Google publishes landmark quantum supremacy claim,” *Nature*, vol. 574, no. 7779, pp. 461–462, Oct. 2019, doi: 10.1038/d41586-019-03213-z.
- [3] “Chapter 3 - Quantum Circuits and Quantum Information Processing Fundamentals | Elsevier Enhanced Reader.”
<https://reader.elsevier.com/reader/sd/pii/B9780123854919000034?token=5C7553DF50BCFF2C186BB6BE668CCBDCDFBBE60A56B212BB05AC5C09E887EC7EC3CFCCA0814D39AF3273A867AACB46DB&originRegion=eu-west-1&originCreation=20220708160904> (accessed Jul. 08, 2022).
- [4] S. Monk, *Programming FPGAs getting started with Verilog*. 2017.
- [5] “Qiskit.” <https://qiskit.org/> (accessed Jun. 01, 2022).
- [6] “QFT_report.pdf.” Accessed: Jun. 01, 2022. [Online]. Available: https://www.eecg.utoronto.ca/~moshovos/CUDA08/arx/QFT_report.pdf
- [7] O. Oumarou, A. Paler, and R. Basmadjian, “Fast quantum circuit simulation using hardware accelerated general purpose libraries.” arXiv, Jun. 26, 2021. Accessed: Jun. 21, 2022. [Online]. Available: <http://arxiv.org/abs/2106.13995>
- [8] C. Conceição and R. Reis, “Efficient emulation of quantum circuits on classical hardware,” in *2015 IEEE 6th Latin American Symposium on Circuits & Systems (LASCAS)*, Feb. 2015, pp. 1–4. doi: 10.1109/LASCAS.2015.7250404.
- [9] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono, “An FPGA-Based Quantum Computing Emulation Framework Based on Serial-Parallel Architecture,” *Int. J.*

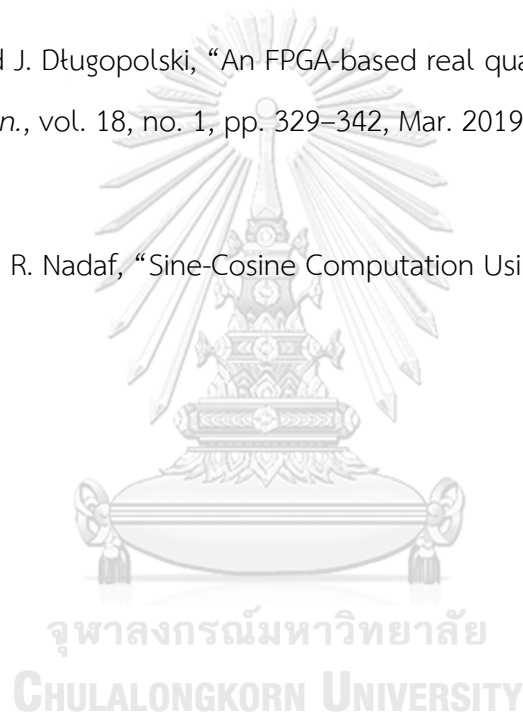
Reconfigurable Comput., vol. 2016, pp. 1–18, 2016, doi: 10.1155/2016/5718124.

[10] N. Mahmud, E. El-Araby, and D. Caliga, “Scaling reconfigurable emulation of quantum algorithms at high precision and high throughput,” *Quantum Eng.*, vol. 1, no. 2, p. e19, 2019, doi: 10.1002/que2.19.

[11] A. U. Khalid, Z. Zilic, and K. Radecka, “FPGA emulation of quantum circuits,” in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, Oct. 2004, pp. 310–315. doi: 10.1109/ICCD.2004.1347938.

[12] J. Pilch and J. Długopolski, “An FPGA-based real quantum computer emulator,” *J. Comput. Electron.*, vol. 18, no. 1, pp. 329–342, Mar. 2019, doi: 10.1007/s10825-018-1287-5.

[13] R. Naik and R. Nadaf, “Sine-Cosine Computation Using CORDIC Algorithm,” vol. 4, no. 9, p. 5.





จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

VITA

NAME	Yaninee Jungjarassub
DATE OF BIRTH	2 February 1997
PLACE OF BIRTH	Chonburi, Thailand
INSTITUTIONS ATTENDED	<ul style="list-style-type: none"> - Bachelor’s Degree at Chulalongkorn University Faculty of Science, Department of Physics, Bangkok, Thailand (3.22) - High School at Bothongwongchanwittaya school Science – Mathematic Program, Chonburi, Thailand (3.94) - Elementary school at Anubanbothong school Chonburi, Thailand
PUBLICATION	<p>Y. Jungjarassub and K. Piromsopa, “A Performance Optimization of Quantum Computing Simulation using FPGA,” in 2022 19th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), May 2022, pp. 1–4. doi: 10.1109/ECTI-CON54298.2022.9795571.</p>
AWARD RECEIVED	Achieved the Academic Year 2016 Outstanding Achievement Development Award