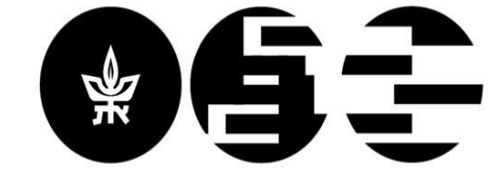

Insert the Project Title here

Project Number: _____

Project Report

Student: K. Doron ID: _____

Student: M. Doron ID: _____

Supervisor: Khen C.

Project Carried Out at: Tel Aviv university

Contents

List of figures

List of tables

Abstract

The main goal of the project is to simulate the Quantum Fourier Transform (QFT) algorithm in a classical configuration on an FPGA component and compare the results to a parallel simulation in Python. This comparison aims to determine if there is potential for performance improvement when simulating on an FPGA compared to using a classical computer by comparing run times for different workloads. These workloads are defined by the number of Qubits on which the algorithm runs. A Qubit is the quantum equivalent of a classical bit, allowing for entanglement, which does not exist in classical bits. For 2 Qubits, the difference can be represented as follows:

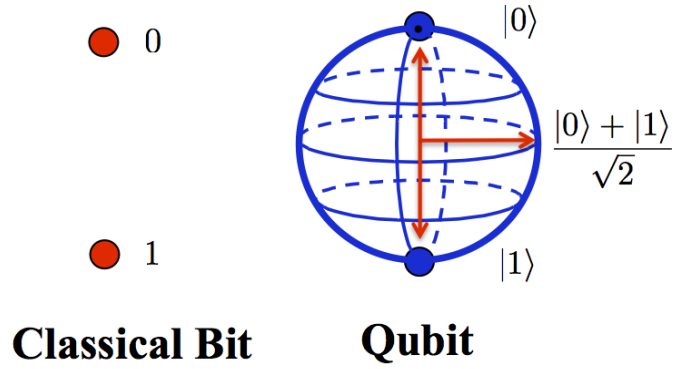

Figure 1: Single qubit representation

The project is divided into two main parts. The theoretical part involves taking the algorithm in its quantum form, breaking it down into the quantum gates that compose it, converting each gate into its corresponding matrix representation, and then translating these into a classical representation based on relevant literature. The specific methods are detailed in the theoretical background section, along with the creation of pseudo-algorithms for running the algorithm with different numbers of Qubits.

The second part of the project is practical. It includes creating Python code to generate the classical representation matrices for all required gates and running the algorithm. Concurrently, equivalent code was written for the FPGA using System Verilog, and a communication platform was established between the FPGA and the connected computer using the UART-USB bridge communication protocol [4][5][6]. All blocks designed for the FPGA operated synchronously at a clock rate of 200 MHz. Detailed descriptions of the communication protocol and the codes are

provided in the implementation section. Additionally, there is an explanation for each part of the code in the project's GitHub repository [3], along with the relevant codes themselves.

The implementation of the algorithm's run block on the FPGA was done using a state machine, which is graphically described here. We will detail each relevant gate and stage further on.

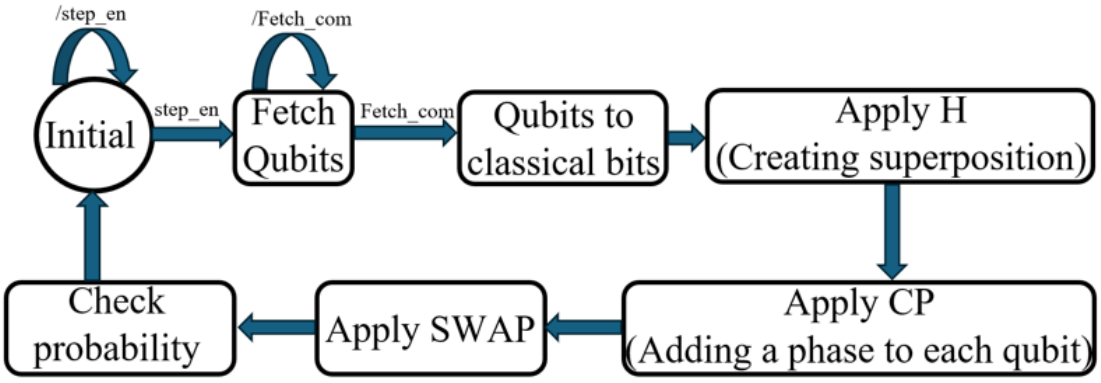

Figure 2: Block diagram

1 Introduction

The project has 2 main goals:

· Testing a new method to simulate quantum circuits, while focusing on the $QFT(Quntaum\ fourier\ algorithm)$, the algorithm is fully explained in the theoretical background part.

· Simulating the QFT algorithm using the new method on an FPGA device, and python using a classic CPU home machine (PC), comparing the two. In addition to comparing the two to the most popular open-source python library used to simulate quantum circuits $Qskit$ [1].

One of the biggest obstacles when it comes to quantum computers today is creating a machine which can run enough $Qubits$ [2] to be able to run complex and practical algorithms for real world usage. Additionally, research focus on classical simulations to quantum circuits of smaller systems to be able to more effectively understanding quantum behavior and therefore better guide quantum hardware and algorithm development [7]. As a result, a lot of quantum computing research is focusing on how and what is the most effective way to simulate such circuit, trying to use and find new methods to do so all the time. One such new direction is using FPGA devices parallel nature to simulate these circuits, as the simulation of such circuit requires a lot of matrices mathematics [8] [9]. All of these motivated us, bases on one such research which claims to have found a new method to represent quantum circuits using classical mathematics [10] to test it and running a basic quantum circuit on an FPGA device.

Our approach to the project was to firstly analyze and understand the new method from the paper [10], afterwards using it to translate each needed gate, and creating a theoretical framework (as seen in the theoretical background chapter in detail) that we can base on to make the simulations and needed software. After wards implementing a python simulation using both our proposed method, and the known method using $Qskit$ python library [1] to check and see if we get the expected results which fit the theory. Lastly a communication platform was built using Verilog so we can send and receive information from the FPGA device and creating system Verilog

simulations and software to run on our FPGA device [11] before comparing all the results, which can be seen in more detail in the results analysis chapter.

2 Theoretical background

The goal of this project is to simulate the QFT algorithm on FPGA component. QFT – Quantum Fourier Transform, is the quantum analogue of the DFT - Discrete Fourier Transform algorithm.

To explain the QFT algorithm, and its usage, we must explain first the DFT, DTFT, FT algorithms, Fourier analysis and the Fourier series

Fourier series[15]:

A Fourier series is an expansion of a periodic function into a sum of trigonometric functions.

A Fourier series is a continuous, periodic function, created by the summation of harmonically related sinusoidal functions. By denoting the period – P, and the function - $s_N(x)$, we will get:

$$s_N(x) = A_0 + \sum_{n=1}^{N} \left(A_n \cos\left(2\pi \frac{n}{P} x\right) + B_n \sin\left(2\pi \frac{n}{P} x\right) \right)$$

Where the coefficients can be described as (normally, the integration is over $\left[-\frac{P}{2}, \frac{P}{2}\right]$ or $[0, P]$):

$$A_0 = \frac{1}{P} \int_P s(x) dx\,, \qquad A_{n\geq 1} = \frac{2}{P} \int_P s(x) \cos\left(2\pi \frac{n}{P} x\right) dx\,,$$

$$B_{n\geq 1} = \frac{2}{P} \int_P s(x) \sin\left(2\pi \frac{n}{P} x\right) dx$$

Fourier analysis[23]:

Fourier analysis is the study of the way general functions may be represented (or approximated) by sums of simpler trigonometric functions. Fourier analysis started from the study of Fourier series

DTFT – Discrete Time Fourier Transform [16]:

The DTFT algorithm is a form of Fourier analysis, that is applicable to a sequence of discrete values. DTFT is often used to analyze samples of continuous functions. The term “discrete time” refers to the fact that the transform operates on discrete data. From uniformly spaced samples, we will get a function of the frequency, that is a periodic summation of the continuous Fourier transform of the original continuous

function. Under certain conditions, we can recover the original continuous function, perfectly, from the DTFT

By denoting the function – x(t) we can find the FT:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi i \cdot f \cdot t} dt$$

By replacing x(t) with a discrete sequence of samples x(nT) and replacing dt with T we can replace the integration with summation:

$$X_{\frac{1}{T}}(f) = \sum_{n=-\infty}^{\infty} \underbrace{T \cdot x(nT)}_{x[n]} e^{2\pi i \cdot f \cdot Tn}$$

This is a Fourier series in frequency, with periodicity of $\frac{1}{T}$. By denoting $\omega = 2\pi f T$ (the frequency variable, normalized to units of $\frac{rad}{sample}$), the periodicity is 2π and we get the DTFT:

$$X_{2\pi}(\omega) = X_{\frac{1}{T}}\left(\frac{\omega}{2\pi T}\right) = \sum_{n=-\infty}^{\infty} x[n] e^{-i\omega n}$$

DFT – Discrete Fourier Transform [17]:

The DFT algorithm takes a finite sequence of equally spaced samples of a function, and converts the samples into a (same length) sequence of equally spaced samples of the DTFT – Discrete Time Fourier Transform (a complex valued function of the frequency)

The DFT transforms a sequence of N complex numbers $\{x_N\} = x_0, x_1 \dots x_{N-1}$ into a different sequence of complex numbers $\{X_k\} = X_0, X_1, \dots, X_{N-1}$ using the transformation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{k}{N} n}$$

We can define the inverse transformation:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{2\pi i \frac{k}{N} n}$$

After explaining all the relevant background, we can start talking about QFT:

In quantum computing, QFT is a linear transformation applied on quantum bits (Qbits) and is the quantum analogue of the DFT algorithm. QFT is a part of many

quantum algorithms, such as Shor's algorithm [18] (factoring and computing the discrete logarithm), quantum phase estimation [19] (estimating the eigenvalues of the unitary operator) and more.

The QFT can be performed efficiently on a quantum computer [20]. The quantum circuit can be implemented with only $\mathcal{O}(n^2)$ Hadamard gates and controlled phase gates [20] (n is the number of Qbits). By comparing this algorithm to the classical DFT [17] (which takes $\mathcal{O}(n2^n)$ gates – n is the number of bits), we see that the classical algorithm is exponentially more taxing (gate-wise).

The QFT operates on a quantum state [20] (represented as a vector), and the classical DFT operates on a normal vector [17]. Both vectors can be also written as lists of complex numbers. where in the quantum case, the list consists of amplitudes of all possible outcomes upon measurement.

The best known QFT algorithm requires $\mathcal{O}\big(nlog(n)\big)$ gates to achieve an efficient approximation [20].

The quantum algorithm [20]**:**

Let $N = 2^n$, where n is the number of Qbits. The starting quantum state will be represented as:

$$|x\rangle = \sum_{i=0}^{N-1} x_i |i\rangle$$

And we will get the quantum state after the QFT, which can be represented as:

$$|y\rangle = \sum_{i=0}^{N-1} y_i |i\rangle\,, \qquad y_k = \frac{1}{\sqrt{N}} \sum_{n=1}^{N-1} x_n \omega_N^{nk}\,, \qquad \omega_N = e^{\frac{2\pi i}{N}}, \qquad k \in [0, N-1]$$

We can also get the inverse QFT as:

$$x_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} y_k \omega_N^{-nk}\,, \qquad n \in [0, N-1]$$

In case $|x\rangle$ is a basis state, applying QFT will result in

$$|x\rangle \longmapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{xk} |k\rangle$$

We can also think of QFT as a single gate - F_N:

$$F_N = \frac{1}{\sqrt{N}}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

When we try to implement the quantum circuit for QFT we need to apply 2 main types of quantum gates – Hadamard and Phase (in the algorithm, we use the controlled version of the Phase gate):

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad P(\phi_k) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi_k} \end{bmatrix}, \qquad \phi_k = \frac{2\pi}{2^k}$$

Where Hadamard gate takes a Qbit and transform it into a superposition of the two states:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \qquad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

The controlled version of Phase gate (CP) takes two Qbits and checks the state of the first Qbit. If the first Qbit is 0, nothing happened. If the first Qbit is 1, then we apply the Phase gate on the second Qbit.

The circuit looks will look as follows:

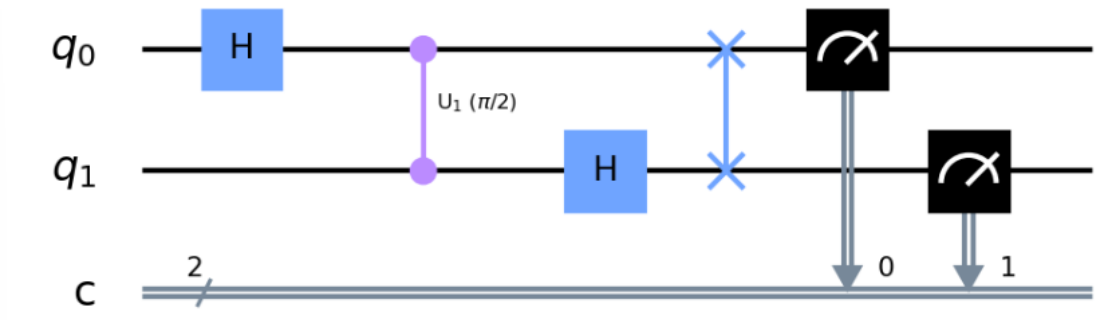

Figure 3: Quantum circuit diagram [12]

In the image, we see that we apply Hadamard gate to the first Qbit, apply Phase gate with $k = 0$, then Hadamard on the second Qbit, apply SWAP gate and then measure the Qbits.

SWAP gate switches the Qubits, since the output has the Qubits in the reverse order to the order, we want we will receive the following pseudo algorithm:

Pseudo algorithm for QFT (no measurement):

- Take n Qubits
- K=n-1
- Phase = $\pi/2$
- While $k \neq 0$:
    - Implement H on q_k
    - i = n-1
    - while $i > k - 1$:
        - implement CP on $q_i$using $q_{k-1}$with phase of $\frac{Phase}{2^{i-k}}$
        - i=i-1
- implement H on q_0
- if n is even:
    - k=n
    - while $k \neq \frac{n}{2} - 1$:
        - swap q_{n-k}, q_{k-1}
        - k=k-1
- if n is odd:
    - $k = n$
    - while $k \neq \frac{n-1}{2}$:
        - swap q_{n-k}, q_{k-1}
        - k=k-1

Classical QFT approach [10]:

In the algorithm we rely on, the idea is to take the quantum gate and Qubits, and transform them to a sort of classical version of them – instead of having Qubits, that can be represented as vectors the size of $2X1$: $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$, $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$, we will

represent them in a different way: each Qbit will become a classical bit with a size of $8X1$:

$$\varphi|0\rangle_{quantum} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_{classical} \quad , \quad \varphi|1\rangle_{quantum} = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_{classical}$$

For a general vector:

$$|k\rangle = c_0|0\rangle + c_1|1\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} x_0 + iy_0 \\ x_1 + iy_1 \end{pmatrix} = \vec{x} + i\vec{y}$$

$$\varphi|k\rangle_{quantum} = \frac{1}{8}(\vec{u}_{classical} + \vec{p}_{classical}), \qquad \vec{u} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \qquad \vec{p} = \begin{pmatrix} \vec{x} \\ -\vec{x} \\ \vec{y} \\ -\vec{y} \end{pmatrix}$$

In quantum mechanics, there is a concept called entanglement – by applying a quantum gate on a few Qubits at the same time (such as Control Phase gate), the Qubits are now called entangled, meaning that now we cannot look at each state by itself, and instead the state is composed of several Qubits (for example, non-entangled will look like $|0\rangle_1 \otimes |1\rangle_2$ and entangled will look like $|01\rangle_{1,2}$). In the classical representation we use, the entanglement will look like this:

For 2 Qubits, with classical representations s_1, s_2 we get

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle, \qquad s_{12} = \varphi|\psi\rangle, \qquad s_1 = \varphi|\psi_1\rangle, \qquad s_2 = \varphi|\psi_2\rangle$$

$$s_{12} = \frac{1}{8^2}(\vec{u} \otimes \vec{u} + \vec{p}_1 \otimes \vec{p}_2) = \frac{1}{64}\left(\begin{pmatrix}\vec{u}\\ \vec{u}\\ \vec{u}\\ \vec{u}\\ \vec{u}\\ \vec{u}\\ \vec{u}\\ \vec{u}\end{pmatrix} + \begin{pmatrix}\vec{x}_1 \otimes \begin{pmatrix}\vec{x}_2\\ -\vec{x}_2\\ \vec{y}_2\\ -\vec{y}_2\end{pmatrix}\\ -\vec{x}_1 \otimes \begin{pmatrix}\vec{x}_2\\ -\vec{x}_2\\ \vec{y}_2\\ -\vec{y}_2\end{pmatrix}\\ \vec{y}_1 \otimes \begin{pmatrix}\vec{x}_2\\ -\vec{x}_2\\ \vec{y}_2\\ -\vec{y}_2\end{pmatrix}\\ -\vec{y}_1 \otimes \begin{pmatrix}\vec{x}_2\\ -\vec{x}_2\\ \vec{y}_2\\ -\vec{y}_2\end{pmatrix}\end{pmatrix}\right)$$

For n Qubits:

$$s_{12\dots n} = s_{1,n} = \frac{1}{8^n}\left(\vec{u}^{\otimes n} + \vec{p}_1 \otimes \vec{p}_2 \otimes \dots \otimes \vec{p}_n\right)$$

To represent the matrices, we must look at two important matrices:

$$P_0 = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}, \qquad P_1 = \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}$$

All control gates can be represented using those matrices (we will demonstrate this using CNOT):

$$CNOT = Control\ Not = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix} = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} + \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} \otimes \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

$$CNOT = P_0 \otimes \mathbb{I}_2 + P_1 \otimes NOT$$

To find the classical representation of any gate, we have to apply $M[Gate]$:

$$M[Gate] = \begin{pmatrix}\mathcal{R}e(Gate) & 0 & 0 & \mathcal{I}m(Gate)\\ 0 & \mathcal{R}e(Gate) & \mathcal{I}m(Gate) & 0\\ \mathcal{I}m(Gate) & 0 & \mathcal{R}e(Gate) & 0\\ 0 & \mathcal{I}m(Gate) & 0 & \mathcal{R}e(Gate)\end{pmatrix}$$

And if $Gate = A \otimes B + C$ we get

$$M[Gate] = M[A] \otimes M[B] + M[C]$$

And therefore, in the QFT algorithm, we get:

$$M[H] = \begin{pmatrix} H & 0 & 0 & 0 \\ 0 & H & 0 & 0 \\ 0 & 0 & H & 0 \\ 0 & 0 & 0 & H \end{pmatrix}, \qquad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$P(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}, \qquad \mathcal{Im}\big(P(\phi)\big) = \begin{pmatrix} 0 & 0 \\ 0 & \sin(\phi) \end{pmatrix}, \qquad \mathcal{Re}\big(P(\phi)\big) = \begin{pmatrix} 1 & 0 \\ 0 & \cos(\phi) \end{pmatrix}$$

$$M[P(\phi)] = \begin{pmatrix} \mathcal{Re}\big(P(\phi)\big) & 0 & 0 & \mathcal{Im}\big(P(\phi)\big) \\ 0 & \mathcal{Re}\big(P(\phi)\big) & \mathcal{Im}\big(P(\phi)\big) & 0 \\ \mathcal{Im}\big(P(\phi)\big) & 0 & \mathcal{Re}\big(P(\phi)\big) & 0 \\ 0 & \mathcal{Im}\big(P(\phi)\big) & 0 & \mathcal{Re}\big(P(\phi)\big) \end{pmatrix}$$

$$CP = P_0 \otimes \mathbb{I}_2 + P_1 \otimes P(\phi), \qquad M[CP] = M[P_0] \otimes M[\mathbb{I}_2] + M[P_1] \otimes M[P(\phi)]$$

$$M[CP] = \begin{pmatrix} M[\mathbb{I}_2] & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & M[P(\phi)] & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & M[\mathbb{I}_2] & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & M[P(\phi)] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & M[\mathbb{I}_2] & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & M[P(\phi)] & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & M[\mathbb{I}_2] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & M[P(\phi)] \end{pmatrix}$$

In the case of the SWAP gate, we cannot use the approach we took with the other gates (it is not a control gate, or a $2X2$ gate). However, we know that $SWAP = CNOT_{12} \cdot CNOT_{21} \cdot CNOT_{12}$ (meaning we apply a controlled NOT (1 is the control and 2 is the controlled), then a controlled NOT (2 is the control and 1 is the controlled) and then a controlled NOT (1 is the control and 2 is the controlled) again). We only need to find $M[CNOT_{12}], M[CNOT_{21}]$, in order to find $M[SWAP] = M[CNOT_{12}] \cdot M[CNOT_{21}] \cdot M[CNOT_{12}]$:

$$CNOT_{12} = P_0 \otimes \mathbb{I}_2 + P_1 \otimes NOT, \qquad CNOT_{21} = \mathbb{I}_2 \otimes P_0 + NOT \otimes P_1$$

$$M[CNOT_{12}] = \begin{pmatrix} M[\mathbb{I}_2] & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & M[NOT] & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & M[\mathbb{I}_2] & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & M[NOT] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & M[\mathbb{I}_2] & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & M[NOT] & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & M[\mathbb{I}_2] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & M[NOT] \end{pmatrix}$$

$$M[CNOT_{21}] = \begin{pmatrix} M[P_0] & M[P_1] & 0 & 0 & 0 & 0 & 0 & 0 \\ M[P_1] & M[P_0] & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & M[P_0] & M[P_1] & 0 & 0 & 0 & 0 \\ 0 & 0 & M[P_1] & M[P_0] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & M[P_0] & M[P_1] & 0 & 0 \\ 0 & 0 & 0 & 0 & M[P_1] & M[P_0] & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & M[P_0] & M[P_1] \\ 0 & 0 & 0 & 0 & 0 & 0 & M[P_1] & M[P_0] \end{pmatrix}$$

When we try to find the classical representation of the matrices for higher number of Qubits, we must find the Quantum representation first. The Quantum representation for control gates (when the control Qbit is the first one and the Controlled Qbit is the last one):

$$Cgate_{1,n} = P_0 \otimes \mathbb{I}_{2^{n-1},2^{n-1}} + P_1 \otimes \mathbb{I}_{2^{n-2},2^{n-2}} \otimes gate$$

$$Cgate_{n,1} = \mathbb{I}_{2^{n-1},2^{n-1}} \otimes P_0 + \mathbb{I}_{2^{n-2},2^{n-2}} \otimes gate \otimes P_1$$

And therefore:

$$M\left[Cgate_{1,n}\right] = M[P_0] \otimes M\left[\mathbb{I}_{2^{n-1},2^{n-1}}\right] + M[P_1] \otimes M\left[\mathbb{I}_{2^{n-2},2^{n-2}}\right] \otimes M[gate]$$

$$M\left[Cgate_{n,1}\right] = M\left[\mathbb{I}_{2^{n-1},2^{n-1}}\right] \otimes M[P_0] + M\left[\mathbb{I}_{2^{n-2},2^{n-2}}\right] \otimes M[gate] \otimes M[P_1]$$

And when we have n Qubits, we can denote the control Qbit is j, the controlled Qbit k and we get (the control Qbit is the first one and the controlled Qbit is the second one):

$$j < k: \;\; Cgate_{1,j,k,n} = \mathbb{I}_{2^{j-1},2^{j-1}} \otimes Cgate_{1,k-j+1} \otimes \mathbb{I}_{2^{n-k},2^{n-k}}$$

$$j > k: \;\; Cgate_{i,j,k,n} = \mathbb{I}_{2^{j-1},2^{j-1}} \otimes Cgate_{j-k+1,1} \otimes \mathbb{I}_{2^{n-k},2^{n-k}}$$

And therefore:

$$M\left[Cgate_{1,j,k,n}\right] = M\left[\mathbb{I}_{2^{j-1},2^{j-1}}\right] \otimes M\left[Cgate_{1,k-j+1}\right] \otimes M\left[\mathbb{I}_{2^{n-k},2^{n-k}}\right]$$

Example for 4 Qubits, Control Phase gate and Control NOT gate:

$$CP\left(\frac{\pi}{8}\right)_{1,4}^{4Qbits} = P_0 \otimes \mathbb{I}_{8x8} + P_1 \otimes \mathbb{I}_{4x4} \otimes P\left(\frac{\pi}{8}\right)$$

$$CNOT_{1,2,3,4}^{4Qbits} = \mathbb{I}_{2x2} \otimes (P_0 \otimes \mathbb{I}_{2x2} + P_1 \otimes NOT) \otimes \mathbb{I}_{2x2}$$

In order to calculate the probabilities of different states from the final vector we will get, we must denote the quantum state as: $|a_n a_{n-1} \dots a_2 a_1 a_0\rangle$

Then, the probabilities are denoted as [10]:

$$P(|a_n a_{n-1} \dots a_2 a_1 a_0\rangle = state) = \left(1 - 8^n s_{1,n}[x]\right)^2 + \left(1 - 8^n s_{1,n}[x+4]\right)^2$$

Where

$$x = a_0 \cdot 8^0 + a_1 \cdot 8^1 + a_2 \cdot 8^2 + \cdots$$

In our code, we ignore the normalization by 8^n throughout the entire code, in order to retain as much information as possibly (in large number of Qubits, we will see a very small normalization factor. Meaning we might not see any information in the final state vector) – therefore we get the probabilities:

$$P(|a_n a_{n-1} \dots a_2 a_1 a_0\rangle = state) = \left(1 - s_{1,n}[x]\right)^2 + \left(1 - s_{1,n}[x+4]\right)^2$$

Example:

$$P(\langle 011|a_2 a_1 a_0\rangle) = \left(1 - s_{1,n}[9]\right)^2 + \left(1 - s_{1,n}[13]\right)^2$$

Some necessary background for the simulation (binary representation):

Two's complement [21] – instead of using all bits, to represent numbers from 0 to $2^n - 1$, we use the MSB to represent the sign (0 means positive and 1 means negative), and the rest of the bits are used normally. In this way, we can represent numbers from -2^{n-1} to $2^{n-1} - 1$.

Example for 4 bits:

$$0000_2 \to 0, \quad 0001_2 \to 1, \quad 0010_2 \to 2, \quad 0011_2 \to 3, \quad 0100_2 \to 4, \quad 0101_2 \to 5$$

$$0110_2 \to 6, \quad 0111_2 \to 7, \quad 1000_2 \to -8, \quad 1001_2 \to -7, \quad 1010_2 \to -6,$$

$$1011_2 \to -5 \quad 1100_2 \to -4, \quad 1101_2 \to -3, \quad 1110_2 \to -2, \quad 1111_2 \to -1$$

Floating point [22] – instead of using all bits, to represent numbers from 0 to $2^n - 1$, we use some of the bits to represent an integer, and some to represent a fraction. Then we combine them together to get numbers such as $\frac{1}{\sqrt{2}}$ (which we use almost in every quantum algorithm).

Example for 8 bits (4 bits for the integer and 4 bits for the fraction):

$$10001000_2 \to 1000.1000_2 \to 8.5, \quad 10100001_2 \to 1010.0001_2 \to 10.0625$$

3 Simulation

During the simulation we worked with vectors, in two's complement [21] and floating point [22] representation. Each Qbit was represented as vector of 8 elements, with each element being composed of 24 bits – 1 bit for sign, 4 bits for integer representation, and 19 bits for fraction representation. If we want to represent $\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}$ for example, we will get:

$$00000101101010000010011 1_2 = 00000.10110101000001001 11_2 \approx \frac{1}{\sqrt{2}}$$

$$11111010010101111101100 1_2 = 11111.01001010111110110 01_2 \approx -\frac{1}{\sqrt{2}}$$

after writing a simulation for the case of 2 Qubits, we checked the probabilities to be in each of the 4 states (in the quantum notation - $|00\rangle,\ |01\rangle,\ |10\rangle,\ |11\rangle$), and got the probabilities below:

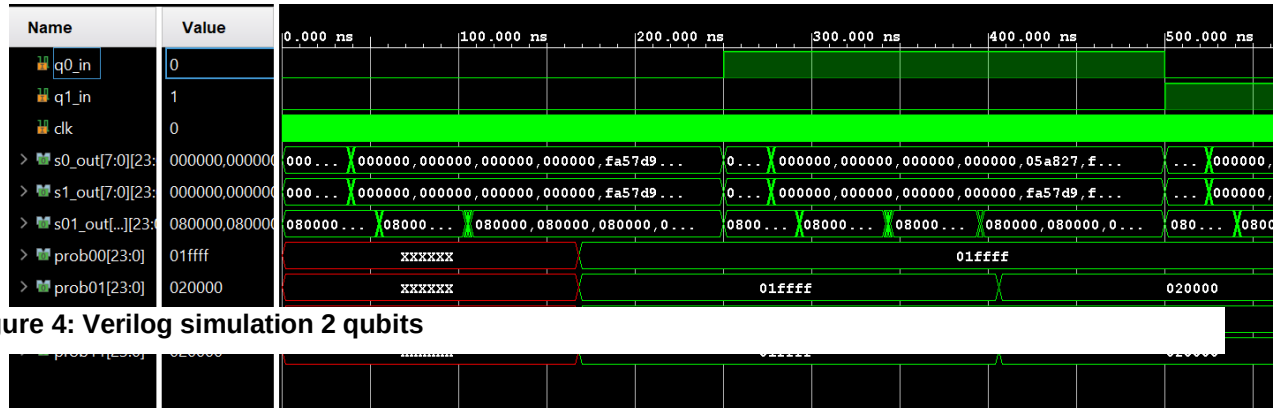

Figure 4: Verilog simulation 2 qubits

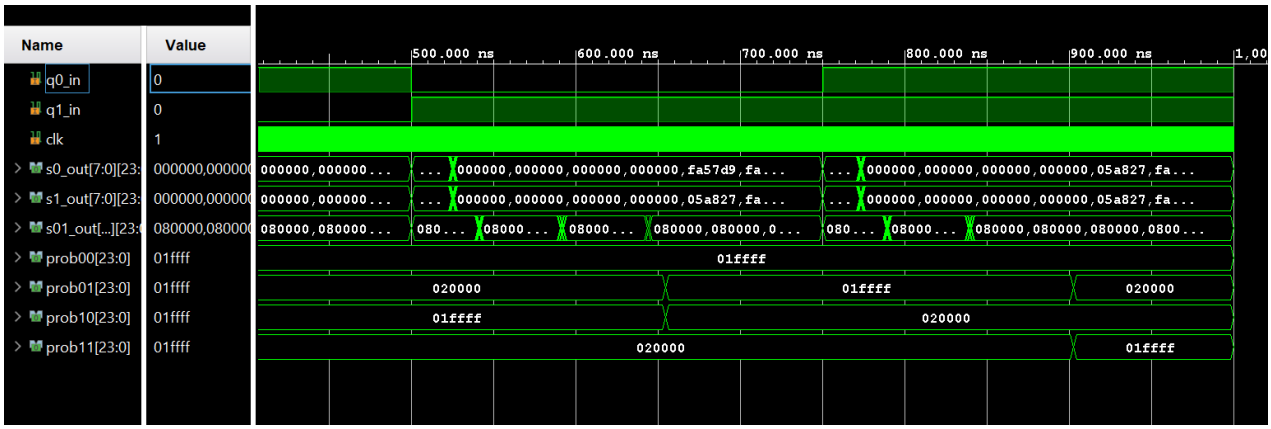

Figure 5: Verilog simulation 2 qubits

we changed the input Qubits every $250\ ns$ (the clock cycle was $0.4\ ns$), and we can see that after about $170\ ns$ (from the start of each input), we get the probabilities. As we can see we got the exact set of probabilities we expected (apart from the obvious

error that stems from the finite number of bits we used to represent each number, where we have numbers like $\frac{1}{\sqrt{2}}$ that requires infinite number of bits to represent correctly):

$$020000_{16} = 00000010000000000000000_2 = 00000.010000000000000000_2$$
$$= 0.25_{10}$$
$$01ffff_{16} = 00000001111111111111111_2 = 00000.001111111111111111_2$$
$$= 0.249998_{10}$$
$$\Rightarrow 01ffff_{16} \approx 020000_{16}$$

Additionally, we have made 2 python simulations codes, one of which is using the qiskit library [5] to simulate the QFT algorithm [20], and a second one simulating using the same method to simulate as explained in the theoretical background [10], giving us the ability to compare run times between the FPGA device, and python and between the two different simulation methods. In the python scripts we use the library time to follow the runtime for each case, while in the FPGA device we use the total clock cycle needed while the clock speed is known to us [11]. Full documentation and the codes are added to the GitHub of the project [3]. All the simulations were ran using the same PC, to make the comparison as fair as possible.

Simulation for the 4 $Qbits$ case using python and Verilog was made as well, working as expected probabilities wise, but as result of lack of time we didn't manage to make the program to burn and run it on the FPGA device as planned.

for the 4 Qubits case for the Verilog simulation (we chose to take 4 arbitrary states, to show the results of the algorithm - $|0000\rangle, |1100\rangle, |0110\rangle, |1111\rangle$): we changed the input Qubits every $250\ ns$ (the clock cycle for the simulation was

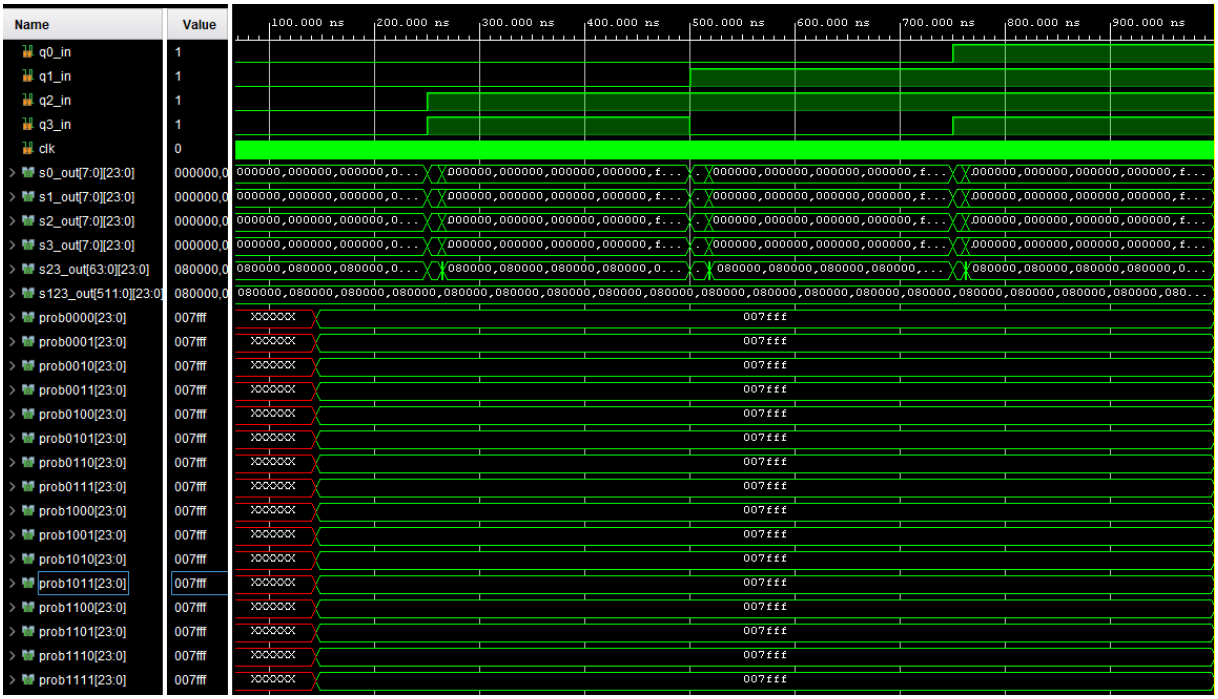

Figure 6: Verilog simulation 4 qubits

$5\ ps$), and we can see that after about $120\ ns$ (from the start of each input), we get the probabilities. As we can see we got the exact set of probabilities we expected (apart from the obvious error that stems from the finite number of bits we used to represent each number, where we have numbers like $\frac{1}{\sqrt{2}}$ that requires infinite number of bits to represent correctly):

$$007fff_{16} = 00000.000011111111111111_2 = 0.06246_{10} \approx 0.0625_{10}$$

4 Implementation

In this chapter, we will describe the implementation of a Quantum Fourier Transform (QFT) [20] algorithm using both software simulation and hardware acceleration on an FPGA device. The project aims to translate the QFT algorithm into a classical representation using a novel method as described in the theoretical background.

The implementation consists of two main components: a Python-based simulation and an FPGA-based hardware acceleration. These components work together to demonstrate the feasibility of simulating quantum algorithms using classical probabilistic bits and circuits, as well as to compare the performance between software and hardware implementations.

A general block diagram of the project implementation is as seen in figure 1.

The Python simulation serves as a reference implementation and a tool for algorithm development. The FPGA implementation accelerates the computation and demonstrates the feasibility of hardware-based quantum algorithm simulation. The PC-FPGA communication allows for data transfer and performance comparison between the two implementations.

4.1 Hardware Description[1]

The hardware component of this project centers around the Xilinx Virtex VC709 FPGA evaluation board [11]. This high-performance FPGA platform provides the necessary resources and interfaces for implementing our quantum algorithm simulation.

Key components of the hardware implementation include:

1. Xilinx Virtex-7 XC7VX690T FPGA [11]: This FPGA chip is the core of the VC709 board, providing a large number of configurable logic blocks, DSP slices, and memory resources necessary for implementing complex algorithms.
2. UART-to-USB Bridge [4] [5] [6]: The VC709 board includes a UART-to-USB bridge, which facilitates communication between the FPGA and a PC. This interface is crucial for transferring data and control signals between the hardware and software components of our system.

3. On-board Memory [11]: The VC709 includes various memory types (e.g., DDR3 SDRAM, QDRII+ SRAM) that can be utilized for storing intermediate results and lookup tables required by our algorithm.
4. Clock Generation and Management [11]: The board provides flexible clock management capabilities, allowing us to generate the necessary clock signals for our design.
5. LED Indicators [11]: While not critical to the core functionality, the on-board LEDs can be used for debugging and status indication during development and operation.

The hardware implementation is designed to take advantage of the FPGA's parallel processing capabilities, allowing for efficient computation of the quantum algorithm simulation. The UART interface serves as the primary means of communication with the host PC, enabling data transfer and control.

4.2 Software Description[2]

The software component of this project consists of two main parts: the Python-based simulation and the FPGA communication interface.

1. Python Simulation (2qbit_simulation.py): This script implements the classical representation of the QFT algorithm for a 2-qubit system. Key features include:
    - Definition of quantum gates (NOT, SWAP, Hadamard, Phase) using both dense and sparse matrix representations.
    - Implementation of helper functions for matrix operations and conversions.
    - Simulation of a 2-qubit quantum algorithm using the translated classical representation.
    - Performance comparison between dense and sparse matrix operations.

The simulation serves as a reference implementation and a tool for algorithm development and verification. It uses libraries such as NumPy [13] for efficient matrix operations and provides functions to convert matrices into formats suitable for FPGA implementation.

2. FPGA Communication Interface (SerialRead.py): This script reads the information arriving from the FPGA to the USB connection via the UART:
    - Configuring and managing the serial port connection.
    - Reading data sent from the FPGA.
    - Processing the received data into 24-bit words.
    - Displaying the processed data in both binary and hexadecimal formats.

The script uses the pyserial library [14] for serial communication and is designed to handle the specific data format used in our FPGA implementation.

3. 4qbit_simulation
    - Like the 2qbit_simulation.py scripts, only for the 4 qubits case.
    - Additionally, has a helper function which converts a sparse matrix to a Verilog-compatible format, including information about non-zero elements to help in creating the firmware for the FPGA device.

Similarly to the 2qbit_simulation.py it uses libraries such as NumPy [13].

4. nqbits_gates.py
    - This script’s purpose is to help create n-qubits control gates and saving them in a format fitting for use in the FPGA firmware.

Similarly to the 2qbit_simulation.py it uses libraries such as NumPy [13].

FPGA Firmware Details:

1. matrix_nwm.sv - Top-level Module

Purpose: This module serves as the top-level interface, managing data flow between the AXI UART interface and the matrix operation module [4] [5] [6].

State Machine:

- INIT: Initializes the module and waits for the matrix to be ready.
- SET_DATA: Prepares data from the matrix for transmission.
- UPDATE_POS: Updates the position counter for data transmission.
- POS_BUFFER: Buffers the updated position and data.
- SEND_DATA: Transmits data when the UART is ready, breaking down the words to 8-bit parts, as it's the word size limit we have from the UART buffer word size.
- WAIT_4_READY: Waits for the UART to be ready for the next data transmission.
- HALT: Halts operation, waiting for reset to reinitialize.

Operation: The state machine controls the flow of data from the matrix operations to the UART interface. It manages the sequencing of data preparation, position updating, and transmission timing. This ensures that the results of the quantum algorithm simulation are properly communicated to the host PC.

2. state_wm.sv - AXI UART State Management Module

Purpose: This module manages the state transitions required for data transfer via the AXI UART Lite interface [4] [5] [6].

State Machine:

- INIT: Waits for valid data to initiate a transfer.
- SET_ADD: Sets the AXI write address.
- SET_DATA: Prepares the data to be written.
- SET_VALID: Asserts the AXI valid signals.
- AWAIT_READY: Waits for the AXI interface to be ready.
- WAIT_FOR_TX_FIFO: Waits for the TX FIFO to have space available.
- HALT: Halts the state machine and signals readiness for the next operation.

Operation: This state machine handles the intricacies of the AXI UART protocol. It ensures that data is properly formatted and timed for transmission over the UART interface. The module interacts with the axi_uart_wait_full_tx module to manage FIFO status and prevent data overflow.

3. axi_uart_wait_full_tx.v - AXI UART TX Buffer Management Module

Purpose: This module reads data from the AXI UART interface and monitors the transmission buffer status [4] [5] [6].

State Machine:

- INIT: Initial state, waiting to start a read transaction.
- SET_ADDRESS: Sets the read address to the UART status register.
- SET_VALID: Asserts the read address valid signal.
- AWAIT_READY: Waits for the AXI interface to acknowledge the read address.
- AWAIT_VALID_DATA: Waits for valid read data from the AXI interface.
- READ_DATA: Reads the data and checks if the transmission buffer is full.
- SET_READ_READY: Asserts the read data ready signal.
- HALT: Determines the next action based on the buffer status.

Operation: This module is crucial for preventing data loss due to buffer overflow. It continuously monitors the UART status register to determine if the transmission buffer is full. When the buffer is not full, it signals that data transmission can continue, ensuring smooth and error-free communication with the host PC.

4. twobytwo_sv_timing.sv - Quantum Algorithm Simulation Module

Purpose: This module implements the core functionality of transforming qubits into classical bits and applying quantum gates as part of the QFT algorithm [20].

State Machine:

- INIT: Initializes the module and checks for changes in qubit inputs.
- HADAMARD: Applies the Hadamard gate using temporary registers.
- TEMP_MOVE: Moves Hadamard results to output registers.
- JOIN01: Combines transformed states of qubits.
- CP: Applies the Controlled-Phase (CPHASE) gate.
- TEMP_01: Moves CPHASE results to output registers.
- CN12: Applies the first Controlled-NOT (CNOT12) gate.
- TEMP_12: Moves CNOT12 results to output registers.
- CN21: Applies the CNOT21 gate.

- TEMP_21: Moves CNOT21 results to output registers.
- CN12_2: Applies the second CNOT12 gate.
- TEMP_f: Performs final cleanup operations.
- HALT: Halts the state machine when operations are complete.

Operation: This module is the heart of the quantum algorithm simulation. It sequentially applies various quantum gates (Hadamard, CPHASE, CNOT) to the input qubits, transforming them into classical representations. The state machine ensures that each operation is performed in the correct order and that the results are properly stored and propagated through the system. The data is sent in the format of 24-bit words, the specific length was chosen to give us a good enough approximation for our calculation while reserving as much resources as possible.

5. fourxfour_QFT_sparse.sv
    - contains the System Verilog module fourXfour_QFT_sparse, which implements a 4x4 Quantum Fourier Transform (QFT) using sparse matrix representation. This module is designed to perform the QFT operation on a set of quantum bits (qubits) and output the resulting quantum state probabilities.

Operation wise works similarly to the twobytwo_sv_timing.sv.

Overall Top Design:
The overall design creates a complete system for simulating a quantum algorithm using classical hardware:

1. The twobytwo_sv_timing module performs the core quantum computations, simulating the behavior of quantum gates on classical bits.
2. The results from these computations are then passed to the matrix_nwm module, which manages the overall data flow and prepares the results for transmission.
3. The state_wm module interfaces with the AXI UART, handling the protocol-specific details of data transmission.

4. The axi_uart_wait_full_tx module ensures that data is transmitted efficiently without buffer overflow.
5. Finally, the data is sent via the UART-USB bridge to the host PC, where it can be received and processed by the SerialRead.py script.

This design allows for efficient simulation of quantum algorithms on classical hardware, with the FPGA providing acceleration for the computationally intensive parts of the simulation. The modular structure enables easy modification and expansion of the system, such as implementing more complex quantum algorithms or increasing the number of qubits in the simulation.

Full explanation for all the software made can be found in the GitHub [3] of the project.

5 Analysis of results

The results consist of the simulations time run, $qbits$ amount of the run and the maximum amount simulated/ran for each method. The methods we compared were our simulations written in python, system verilog using sparse matrices and to the classical method to simulate quantum circuit by the open python library $Qskit$.

The results can be summarized in the following table:

Qubits amount	**Method used**	**Runtime[sec]**
2	Paper method python	0.008976
2	FPGA device	0.06144
2	Python Qiskit	0.006490
4	Paper method python	8.035774
4	Python Qiskit	0.007016

Table 1: Summary of results

As we can see from the results, even though using the paper method we receive the ability to 'run' on the classical plane, allowing us to represent everything using real numbers, and running the simulation using FPGA devices in addition to a theoretically easy way to expand our control and quantum gates to the many qubits easily, the 8^n complexity results in slower runtimes compared to the existing methods and higher memory requirements from our hardware, meaning lower possible maximum amount of qubits simulated while running the algorithm. Maxing out at $39Qubits$ for the $Qskit$[1] algorithm using our home machine, while being managing to simulate a max of $4\ Qubits$ using the paper method, while the theoretical limit for our FPGA device, considering the memory size limitations was $6Qubits$ even while using the sparse matrices. For our python simulation we have found it's possible to run gates for the $8\ Qubits$ case using our home machine, but the runtime is marginally bigger than the $4Qubits$ case which as it's is much longer than the alternative, but as result of time limitations we didn't manage to create a python simulation the case and as said before as a result of the high memory requirements it's impossible to run using the FPGA device. It's important to note that while using the FPGA device we have ran the programs using the default clock of for simplicity sake which run $200[MHz]$, where it's possible to use with the software

other internal clocks which can go up to $810[MHz]$ [11] which will result a faster runtime, but not to the level where it can compete with the $Qskit$ simulation.

6 Conclusions and further work

As we saw in the results, the method which we chose to implement on the FPGA device compared to the existing method is lacking when it comes to runtime and hardware requirement. While the method is unique, new, and has interesting aspects with her idea on representing the quantum world using classical methodology the added size complexity renders it impractical as it stands. given more time we will add the needed software to the FPGA device so we can use faster clock speeds, in addition to cleaning up the code and calculation methods as much as possible to better the efficiency of the code, and as a result the efficiency of the runtime. Additionally, by nature the paper method [10] gates have a lot of recurring values while using it to run the QFT algorithm [20], which better studying and understanding them can result in 'skipping' part of the calculations and saving much needed memory space and runtime. Future possibilities might be to take the known existing popular methods, such as used in $Qiskit$ library [1] and trying to run it using FPGAs devices to see if we can see an improvement compared to a classic CPU machine such as home computers. In addition to all of these we would like to finish the 4 qubits implementation for the FPGA device we have started to see if for higher qubits we get some improvements in the result in contrast to the python simulation.
Future work might include testing the method and alternative methods using an FPGA device using different quantum circuit to implement different algorithms, to see if there's a difference in effect for different algorithm/circuit tested, and testing the use of GPUs devices while comparing them to the FPGA, as they too have parallel calculation nature.

7 Project Documentation

[4] D. M. Doron Ketchker, "github.com/Doronke," 2024. [Online]. Available: https://github.com/Doronke/FPGA-QFT.

8 References

[1] IBM, "IBM Qiskit python library," [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.QFT.

[2] J. M. S. P. O. Alan Ho, "The Promise and Challenges of Quantum Computing for Energy Storage," *Joule,* vol. 2, no. 5, pp. 810-813, 2018.

[3] D. M. Doron Ketchker, "github.com/Doronke," 2024. [Online]. Available: https://github.com/Doronke/FPGA-QFT.

[4] Xilinx, "Xilinx, UG761 AXI Reference Guide," 15 November 2012. [Online]. Available: https://www.xilinx.com/support/documents/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

[5] Xilinx, "AXI UART Lite v2.0 LogiCORE IP Product Guide (PG142)," 5 April 2017. [Online]. Available: https://hthreads.github.io/modules/eecs-4114/data-sheets/pg142-axi-uartlite.pdf.

[6] Xilinx, "Xilinx DS741 LogiCORE IP AXI UART Lite (v1.02a), data sheet," 25 July 2012. [Online]. Available: https://docs.amd.com/v/u/en-US/axi_uartlite_ds741.

[7] M. A. &. C. C. Andrea D'Urbano, "The Significance of Classical Simulations in the Adoption of Quantum Technologies for Software Development," in *International Conference on Product-Focused Software Process Improvement*, Dornbirn, Austria, 2023.

[8] J. P. &. J. Długopolski, "An FPGA-based real quantum computer emulator," *Journal of Computational Electronics,* vol. 18, pp. 329-342, 2018.

[9] E. E.-A. D. C. Naveed Mahmud, "Scaling reconfigurable emulation of quantum algorithms at high precision and high throughput," *Quantum Engineering,* vol. 1, no. 2, 2019.

[10] A. Y. D. D. Yavuz, "Simulation of quantum algorithms using classical probabilistic bits and circuits," 26 July 2023. [Online]. Available: https://arxiv.org/abs/2307.14452.

[11] Xilinx, "VC709 Evaluation Board for the Virtex-7 FPGA User Guide," 11 March 2019. [Online]. Available: https://www.mouser.com/datasheet/2/903/ug887-vc709-eval-board-v7-fpga-1596461.pdf.

[12] quantumcomputinguk, "quantum-fourier-transform-in-qiskit," quantumcomputinguk, [Online]. Available: https://quantumcomputinguk.org/tutorials/quantum-fourier-transform-in-qiskit.

[13] NumPy, "NumPy documentation," NumPy Developers, [Online]. Available: https://numpy.org/doc/stable/index.html.

[14] L. Chris, "pyserial," 2020. [Online]. Available: https://pyserial.readthedocs.io/en/latest/pyserial.html.

[15] A. Pinkus and S. Zafrany, "Fourier series," in *Fourier Series and Integral Transforms*, Cambridge, UK, Cambridge University Press, 1997, pp. 42-44.

[16] P. Prandoni and M. Vetterli, " Discrete-time signal processing," in *Signal Processing for Communications(1 ed.)*, Boca Raton, FL, CRC Press, 2008, pp. 72-76.

[17] A. V. Oppenheim, R. W. Schafer and J. R. Buck, "Discrete Fourier transform," in *Discrete-time signal processing (2nd ed.)*, Upper Saddle River, N.J., Prentice Hall, 1999, p. 571.

[18] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA, 1994.

[19] M. A. &. I. L. C. Nielsen, Quantum computation and quantum information (Repr. ed.), Cambridge Univ. Press, 2001.

[20] L. Ruiz-Perez and G.-E. Juan Carlos, "Quantum arithmetic with the quantum Fourier transform," *Quantum Inf Process,* vol. 16, no. 152, 2017.

[21] A. Bergel, D. Cassou, S. Ducasse and J. Laval, "Deep into Pharo," Square Bracket Associates, Switzerland, August 2013. [Online]. Available: https://files.pharo.org/books-pdfs/deep-into-pharo/2013-DeepIntoPharo-EN.pdf.

[22] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé and S. Torres, Handbook of Floating-Point Arithmetic (1st ed.), Birkhäuser Boston, MA, 2010.

[23] W. Rudin, Fourier Analysis on Groups, Wiley-Interscience, 1990.