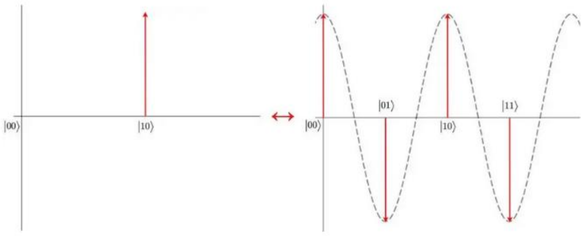
INTRODUCTION:

The Fourier transform occurs in different versions throughout classical computing, in areas ranging from signal processing to data compression to complexity theory. The quantum Fourier transform is the quantum implementation of the discrete Fourier transform over the amplitudes of wavefunction. It is a part of many quantum algorithms, like Shor's algorithm and quantum phase estimation.

The quantum Fourier transform can be performed efficiently on a quantum computer with a decomposition into the product of simpler unitary matrices. The discrete Fourier transform on 2^n amplitudes can be implemented as a quantum circuit consisting of only O(2^n) Hadamard gates and controlled phase shift gates, where n is the number of qubits

The quantum Fourier transform acts on a quantum state vector (a quantum register), and the classical Fourier transform acts on a vector. Both types of vectors can be written as lists of complex numbers. In the quantum case it is a sequence of probability amplitudes for all the possible outcomes upon measurement (called *basis states*, or *eigenstates*). Because measurement collapses the quantum state to a single basis state, not every task that uses the classical Fourier transform can take advantage of the quantum Fourier transform's exponential speedup.



MATHEMATICAL ANALYSIS:

A) Introduction to QFT

A Fourier transform is a mathematical transform that decomposes functions into frequency components which are represented by the output of the transform as a function of frequency

The discrete Fourier transform acts on a vector (x0, ………, xn-1) and maps it to the vector (y0,…….. ,yn-1) according to the formula given below

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk} \quad \text{where} \quad \omega_N^{jk} = e^{2\pi i \frac{jk}{N}}.$$

Similarly, the quantum Fourier transform acts on a quantum state $|X\rangle = \sum_{j=0}^{N-1} x_j |j\rangle$ and maps it to the quantum state $|Y\rangle = \sum_{k=0}^{N-1} y_k |k\rangle$ according to the formula given above.

From the above we can infer that only the amplitudes of the state were affected by thus transformation. This can also be expressed as the map:

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle \quad \text{or the unitary matrix:} \quad U_{QFT} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle\langle j|$$

B) INTUITION:

The quantum Fourier transform transforms between two bases, the computational (Z) basis, and the Fourier's basis. The H- gate is the single qubit QFT, and it transforms between the Z-basis states |0> and |1> to the X- basis states |+> and |->. In the same way, all multi – qubit states in the computational basis have corresponding basis have corresponding states in the Fourier basis. The QFT is simple the function transforms between these bases.

$$|\text{State in Computational Basis}\rangle \xrightarrow{\text{QFT}} |\text{State in Fourier Basis}\rangle$$
$$\text{QFT}|x\rangle = |\tilde{x}\rangle$$

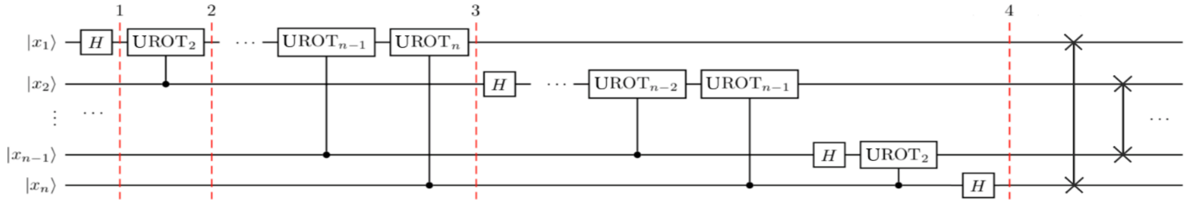
C) The Circuit that implements the QFT:

The action of H on the single qubit states |xk> is $H|x_k\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(\frac{2\pi i}{2}x_k\right)|1\rangle\right)$

The second part is a two -qubit controlled rotation CROT given in block – diagonal format as

$$CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix}$$ where $$UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{2\pi i}{2^k}\right) \end{bmatrix}$$

Given these two gates, a circuit that implements an n- qubit QFT is shown below.



The circuit operates as follows, We start with an n -qubit input state |x1x2.xn>|x1x2….xn>

Algorithm for the circuit given above:

1. After the first Hadamard gate on qubit 1, the state is transformed from the input state to

$$H_1|x_1x_2\ldots x_n\rangle = \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

2. After the $UROT_2$ gate on qubit 1 controlled by qubit 2, the state is transformed to

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

3. After the application of the last $UROT_n$ gate on qubit 1 controlled by qubit n, the state becomes

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x_n + \frac{2\pi i}{2^{n-1}}x_{n-1} + \ldots + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

Noting that

$$x = 2^{n-1}x_1 + 2^{n-2}x_2 + \ldots + 2^1x_{n-1} + 2^0x_n$$

we can write the above state as

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

4. After the application of a similar sequence of gates for qubits $2\ldots n$, we find the final state to be:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^{n-1}}x\right)|1\rangle\right] \otimes \ldots \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^1}x\right)|1\rangle\right]$$

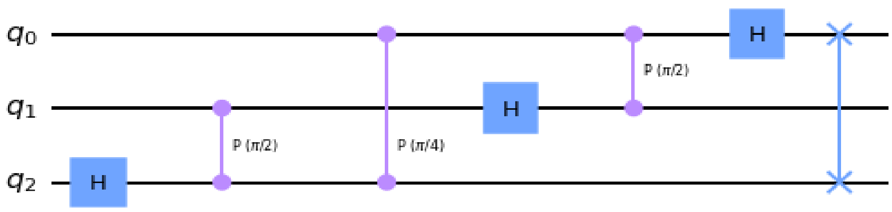
which is exactly the QFT of the input state as derived above with the caveat that the order of the qubits is reversed in the output state.

EXPERIMENTAL IMPLEMENTATION:

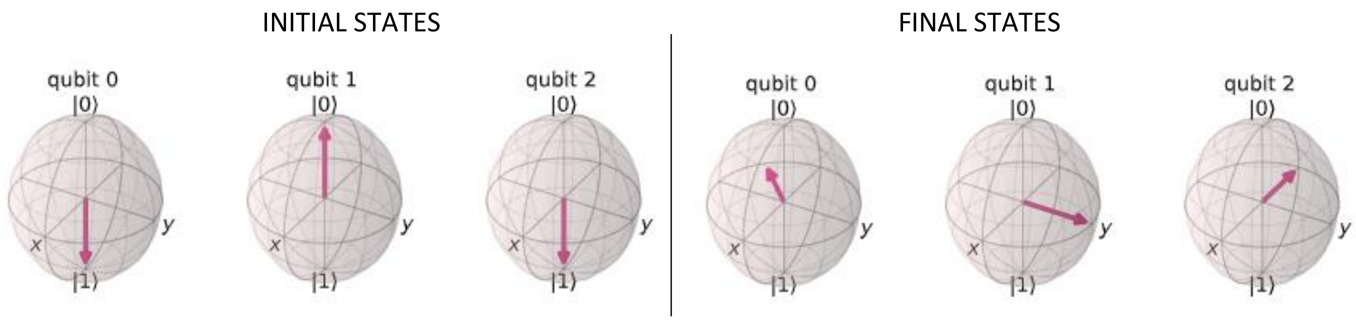
A) SOFTWARE IMPLEMENTATION:

In qiskit the implementation of the CROT gate used in the discussion above is a controlled phase rotation gate. This gate is defined in OpenQASM the mapping from the CROT gate in the above mathematical analysis into the CP gate is found from the equation $\theta = 2\pi/2^k = \pi/2^{k-1}$

With reference from the above-mentioned algorithm, we implement the circuit below using qiskit tool on the IBM quantum simulator (For implementation code refer Appendix D of Supplementary material)



On implementation of the circuit, let us find the initial states i.e input states for the given circuit using the aer_simulator . After u get to know the initial states let us input the initial states the QFT circuit and analyse the results obtained. (Refer Appendix B of supplementary material for general circuit function for n – qubit QFT)

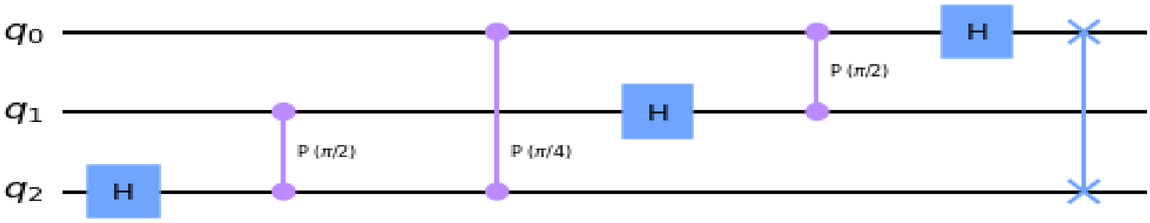


On analysis of the final states Obtained we can see that the QFT function and the implemented circuit has worked properly. Compared the state |˜0⟩=|+++⟩|0~⟩=|+++⟩ , Qubit 0 has been rotated by 5/8 of a full turn , qubit 1 by 10/8 full turns (equivalent to ¼ of a full turn) , and qubit 2 by 20/8 full turns (equivalent to ½ of a full turn)

B) HARDWARE IMPLEMENTATION:

ZedBoard is a low-cost development board for the Xilinx Zynq®-7000 SoC. This board contains everything necessary to create a Linux, Android, Windows® or other OS/RTOS-based design. Additionally, several expansion connectors expose the processing system and programmable logic I/Os for easy user access.

Qiskit implemented circuit:



Hardware implementation and plan

"To mimic the functionality of the QFT circuit implemented using qiskit on the zedboard ."

To implement the QFT circuit we divide the circuit into 7 stages. Each on the seven stages represents the manipulation that the input qubit undergoes.

Algorithm of Implementation on Zedboard :

STAGE 1: Apply Hadamard gate on q2 (third qubit) .

STAGE 2: Check if q1 (|1>) is high , we have a predefined matrix of qubit 1 state and we do matrix matching to check for equivalence . If this is true we do a tensor product of q1 and q2 . Then we apply CROT on the tensor product, if q1 is not high then we print no tensor product and no CROT.

STAGE3: We check if q0 is high, we do a tensor product of q0 and q2 and then apply CROT on the tensor product at P (pie/4), else no tensor product and no CROT.

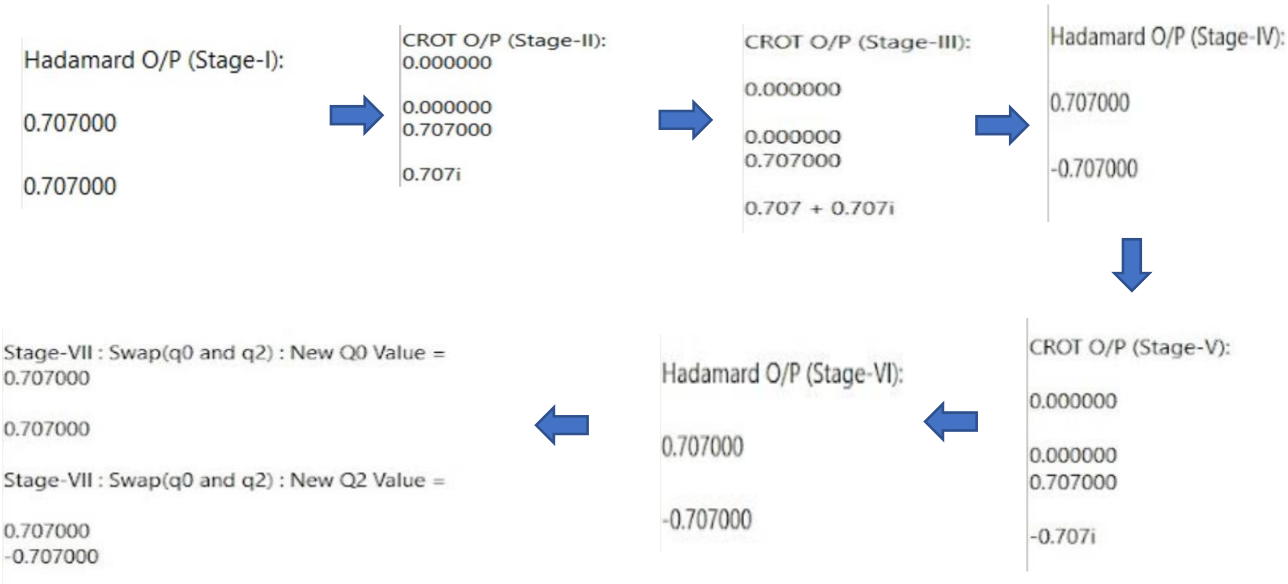
STAGE 4: We apply Hadamard on the second qubit.

STAGE 5: We again check if q0 is high, if true do a tensor product of q0 and q1 then apply CROT on the tensor product of q0 and q1 at P(pie/2).

STAGE 6: Apply Hadamard on the first qubit.

STAGE 7: Swap the contents of q0 and q2.

The below is output for test input case q0 ,q1 and q2 are in state |1>



The QFT implementation on Zedboard executes calculations for each stage while all the inputs are in qubit |1> state. With the help of this test case, we can demonstrate the theocratic presumption that if we supply all high states to the algorithm as input, it should execute all 7 phases. The code and additional output analysis data can be obtained from the supplementary section Appendix – F.

CONCLUSION:

The QFT (quantum Fourier transform) circuit was successfully implemented on qiskit using the IBM quantum simulator. The circuit was also tested on a real IBM quantum computer and satisfactory results were obtained.

On the Zedboard, the QFT circuit was successfully developed and tested for a variety of test scenarios, with positive outcomes.

REFERENCES:

[1] D. Camps, R. V. Beeumen and C. Yang, Quantum fourier transform revisited (2020), 2003.03011.

[2] V. Vorobyov, S. Zaiser, N. Abt, J. Meinel, D. Dasari, P. Neumann and J. Wrachtrup, Quantum fourier transform for quantum sensing (2020), 2008.09716.

[3] M. Mastriani, Fourier's quantum information processing (2020), 2008.07914.

[4] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press (2000).

[5] U. Skosana and M. Tame, Demonstration of shor's factoring algorithm for n=21 on ibm quantum processors (2021), 2103.13855.

[6] D. Kopczyk, Quantum machine learning for data scientists (2018), 1804.10068.

[7] K. Hietala, R. Rand, S.-H. Hung, X. Wu and M. Hicks, A verified optimizer for quantum circuits (2019), 1912.02250.

[8] R. Rand, J. Paykin and S. Zdancewic, Qwire practice: Formal verification of quantum circuits in coq (2018), 1803.00699.

[9] Y. S. Weinstein, S. Lloyd and D. G. Cory, Implementation of the quantum fourier transform (1999), quant-ph/9906059.

[10] Qiskit: An open-source framework for quantum computing, oi:10.5281/zenodo.2573505 (2021).

[11] P. Young, The quantum fourier transform and a comparison with the fast fourier trans-form (2020), 2003.03011

[12] https://community.qiskit.org/textbook/ch-states/single-qubit-gates.html (Last Accessed 22/12/2022 : 11:12 AM)

[13] https://www.quantiki.org/wiki/tensor-product (Last Accessed 22/12/2022 : 2:17 PM)

[14] https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html (Last Accessed 21/12/2022 : 6:37 PM)

SUPPLEMENTARY MATERIAL:

APPENDIX A :EXAMPLE 1:1 QUBIT QFT

Consider how the QFT operator as defined above acts on a single qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. In this case, $x_0 = \alpha$, $x_1 = \beta$, and $N = 2$. Then,

$$y_0 = \frac{1}{\sqrt{2}}\left(\alpha \exp\left(2\pi i \frac{0\times 0}{2}\right) + \beta \exp\left(2\pi i \frac{1\times 0}{2}\right)\right) = \frac{1}{\sqrt{2}}(\alpha+\beta)$$

and

$$y_1 = \frac{1}{\sqrt{2}}\left(\alpha \exp\left(2\pi i \frac{0\times 1}{2}\right) + \beta \exp\left(2\pi i \frac{1\times 1}{2}\right)\right) = \frac{1}{\sqrt{2}}(\alpha-\beta)$$

such that the final result is the state

$$U_{QFT}|\psi\rangle = \frac{1}{\sqrt{2}}(\alpha+\beta)|0\rangle + \frac{1}{\sqrt{2}}(\alpha-\beta)|1\rangle$$

This operation is exactly the result of applying the Hadamard operator (H) on the qubit:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

If we apply the H operator to the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we obtain the new state:

$$\frac{1}{\sqrt{2}}(\alpha+\beta)|0\rangle + \frac{1}{\sqrt{2}}(\alpha-\beta)|1\rangle \equiv \tilde{\alpha}|0\rangle + \tilde{\beta}|1\rangle$$

Notice how the Hadamard gate performs the discrete Fourier transform for $N = 2$ on the amplitudes of the state.

APPENDIX B : EXAMPLE 2 : 3 QUBIT QFT

The steps to creating the circuit for $|y_3y_2y_1\rangle = QFT_8|x_3x_2x_1\rangle$ would be:

1. Apply a Hadamard gate to $|x_1\rangle$

$$|\psi_1\rangle = |x_3\rangle \otimes |x_2\rangle \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

2. Apply a $UROT_2$ gate to $|x_1\rangle$ depending on $|x_2\rangle$

$$|\psi_2\rangle = |x_3\rangle \otimes |x_2\rangle \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

3. Apply a $UROT_3$ gate to $|x_1\rangle$ depending on $|x_3\rangle$

$$|\psi_3\rangle = |x_3\rangle \otimes |x_2\rangle \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^3}x_3 + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

4. Apply a Hadamard gate to $|x_2\rangle$

$$|\psi_4\rangle = |x_3\rangle \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_2\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^3}x_3 + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

5. Apply a $UROT_2$ gate to $|x_2\rangle$ depending on $|x_3\rangle$

$$|\psi_5\rangle = |x_3\rangle \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_3 + \frac{2\pi i}{2}x_2\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^3}x_3 + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

6. Apply a Hadamard gate to $|x_3\rangle$

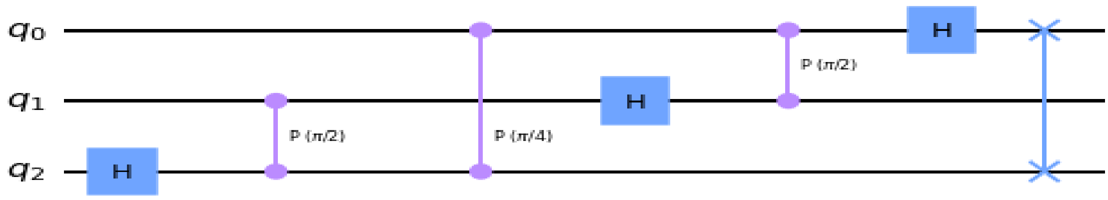
$$|\psi_6\rangle = \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_3\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_3 + \frac{2\pi i}{2}x_2\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^3}x_3 + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right]$$

7. Keep in mind the reverse order of the output state relative to the desired QFT. Therefore, we must reverse the order of the qubits (in this case swap y_1 and y_3).

APPENDIX C : CODE SNIPPET FOR CIRCUIT:

```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()

import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()


qc = QuantumCircuit(3)
qc.h(2)
qc.cp(pi/2, 1, 2) # CROT from qubit 1 to qubit 2
qc.cp(pi/4, 0, 2) # CROT from qubit 2 to qubit 0
qc.h(1)
qc.cp(pi/2, 0, 1) # CROT from qubit 0 to qubit 1
qc.h(0)
qc.swap(0,2)
qc.draw()
```

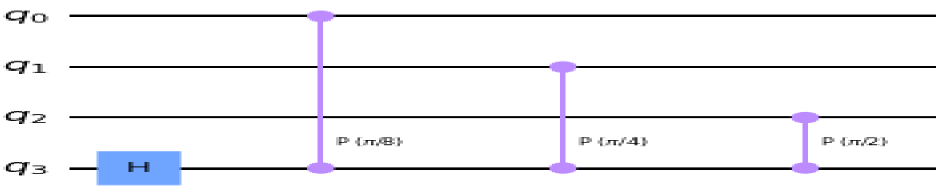
OUTPUT:



APPENDIX D : GENERAL QFT FUNCTION:

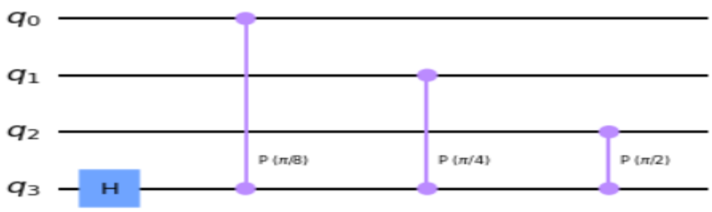
```
def qft_rotations(circuit, n):
    if n == 0: # Exit function if circuit is empty
        return circuit
    n -= 1 # Indexes start from 0
    circuit.h(n) # Apply the H-gate to the most significant qubit
    for qubit in range(n):
        # For each less significant qubit, we need to do a
        # smaller-angled controlled rotation:
        circuit.cp(pi/2**(n-qubit), qubit, n)


qc = QuantumCircuit(4)
qft_rotations(qc,4)
qc.draw()
```

OUTPUT:




```
from qiskit_textbook.widgets import scalable_circuit
scalable_circuit(qft_rotations)
```

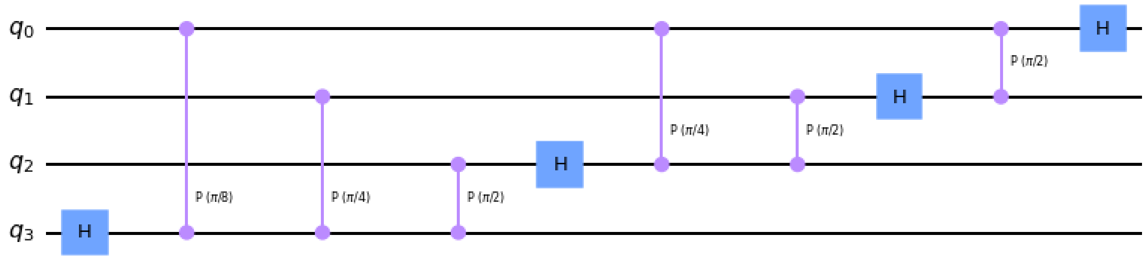
OUTPUT:




```
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circuit (without swaps)"""
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the same function again on
```

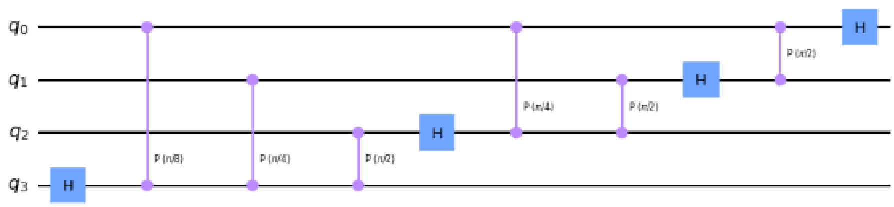
```
    # the next qubits (we reduced n by one earlier in the function)
    qft_rotations(circuit, n)

# Let's see how it looks:
qc = QuantumCircuit(4)
qft_rotations(qc,4)
qc.draw()
```

OUTPUT:



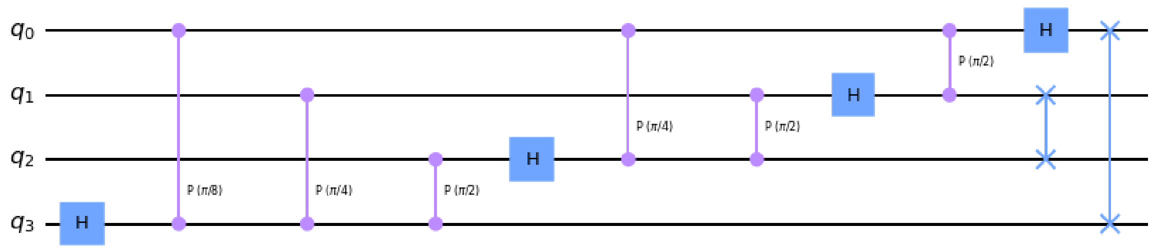

```
scalable_circuit(qft_rotations)
```

OUTPUT:



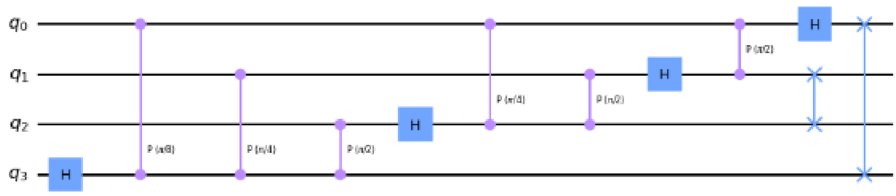

```
def swap_registers(circuit, n):

    for qubit in range(n//2):
        circuit.swap(qubit, n-qubit-1)
    return circuit

def qft(circuit, n):
    """QFT on the first n qubits in circuit"""
    qft_rotations(circuit, n)
    swap_registers(circuit, n)
    return circuit

# Let's see how it looks:
qc = QuantumCircuit(4)
qft(qc,4)
qc.draw()
```

OUTPUT:



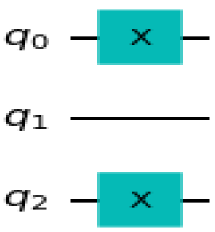

```
scalable_circuit(qft)
```

OUTPUT:



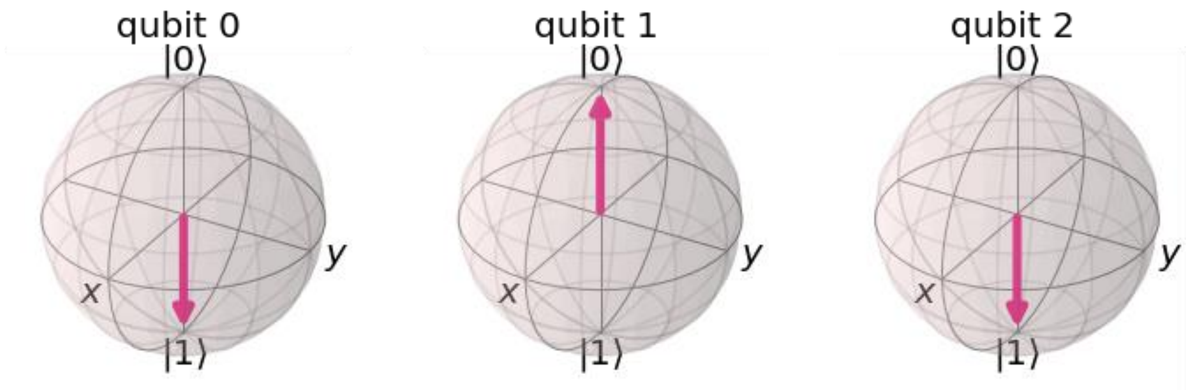

```


# Create the circuit
qc = QuantumCircuit(3)

# Encode the state 5
qc.x(0)
qc.x(2)
qc.draw()
```

OUTPUT:



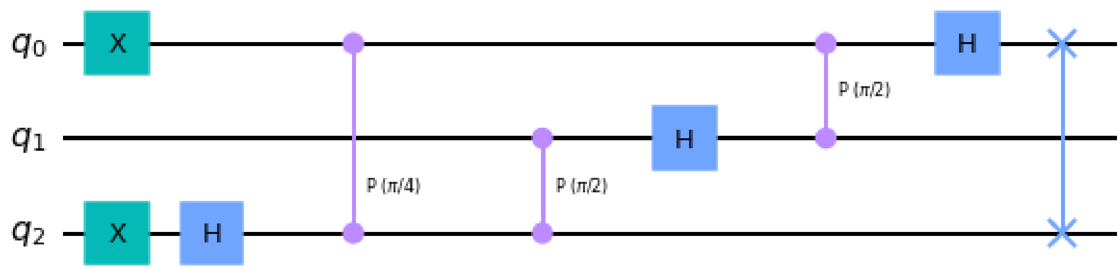

```
sim = Aer.get_backend("aer_simulator")
qc_init = qc.copy()
qc_init.save_statevector()
statevector = sim.run(qc_init).result().get_statevector()
plot_bloch_multivector(statevector)
```

OUTPUT:



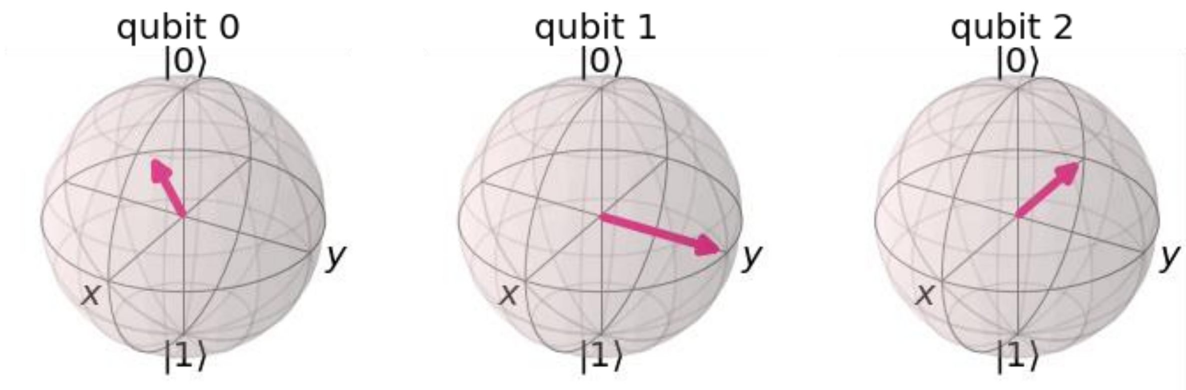

```
qft(qc,3)
qc.draw()
```

OUTPUT:



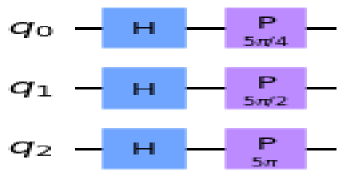

```
qc.save_statevector()
statevector = sim.run(qc).result().get_statevector()
plot_bloch_multivector(statevector)
```

OUTPUT:



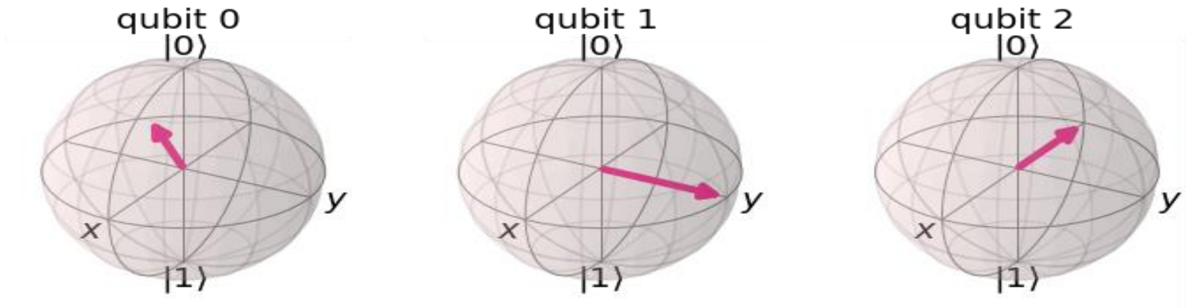
```
def inverse_qft(circuit, n):
    """Does the inverse QFT on the first n qubits in circuit"""
    # First we create a QFT circuit of the correct size:
    qft_circ = qft(QuantumCircuit(n), n)
    # Then we take the inverse of this circuit
    invqft_circ = qft_circ.inverse()
    # And add it to the first n qubits in our existing circuit
    circuit.append(invqft_circ, circuit.qubits[:n])
    return circuit.decompose() # .decompose() allows us to see the indi
vidual gates


nqubits = 3
number = 5
qc = QuantumCircuit(nqubits)
for qubit in range(nqubits):
    qc.h(qubit)
qc.p(number*pi/4,0)
qc.p(number*pi/2,1)
qc.p(number*pi,2)

qc.draw()
```

OUTPUT:



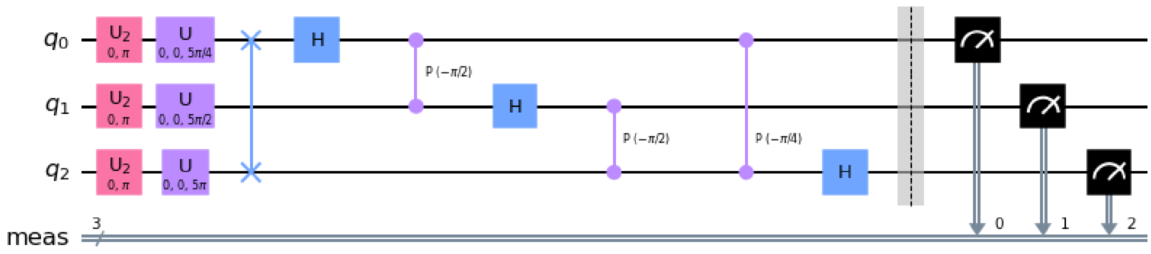

```
qc_init = qc.copy()
qc_init.save_statevector()
sim = Aer.get_backend("aer_simulator")
statevector = sim.run(qc_init).result().get_statevector()
plot_bloch_multivector(statevector)
```

OUTPUT:



```
qc = inverse_qft(qc, nqubits)
qc.measure_all()
qc.draw()
```

OUTPUT:



APPENDIX E : RUNNING QFT ON A REAL QUANTUM DEVICE:

```
# Load our saved IBMQ accounts and get the least busy backend device wi
th less than or equal to nqubits
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuratio
n().n_qubits >= nqubits
                                        and not x.configuration().simula
tor
                                        and x.status().operational==True
))
print("least busy backend: ", backend)
```

OUTPUT :

```
least busy backend:  ibmq_quito
```

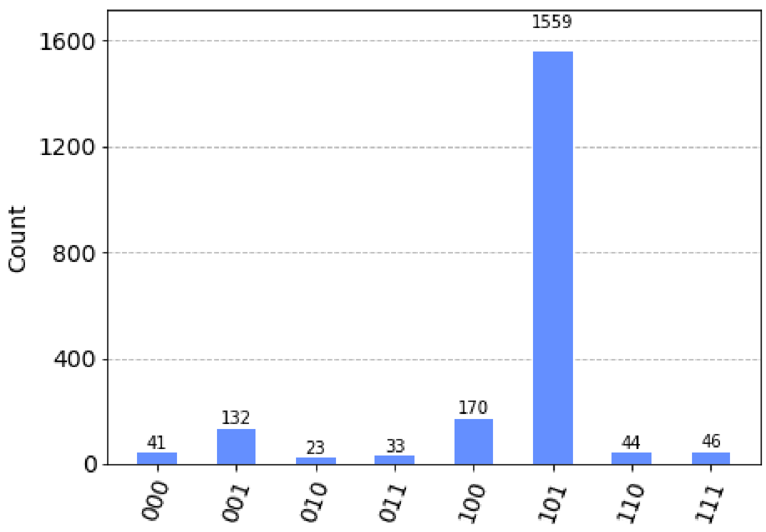
```
shots = 2048
transpiled_qc = transpile(qc, backend, optimization_level=3)
job = backend.run(transpiled_qc, shots=shots)
job_monitor(job)
```

OUTPUT:

```
Job Status: job has successfully run
```

```
counts = job.result().get_counts()
plot_histogram(counts)
```

OUTPUT:



APPENDIX F: HARDWARE IMPLEMENTATION CODE AND OUTPUT

```
/*********************************************************************
*******
*
* Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.
*
* Permission is hereby granted, free of charge, to any person obtaining
 a copy
* of this software and associated documentation files (the "Software"),
 to deal
* in the Software without restriction, including without limitation the
 rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be includ
ed in
* all copies or substantial portions of the Software.
*
* Use of the Software is limited solely to applications:
* (a) running on a Xilinx device, or
* (b) that interact with a Xilinx device through a bus or interconnect.
```

```
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRE
SS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILI
TY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHA
LL
* XILINX  BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OU
T OF
* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN TH
E
* SOFTWARE.
*
* Except as contained in this notice, the name of the Xilinx shall not
be used
* in advertising or otherwise to promote the sale, use or other dealing
s in
* this Software without prior written authorization from Xilinx.
*
***********************************************************************
*******/

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

int main() {
    {
        init_platform();
    int i, j, k;
    int equal,qual,ual;
```

```
    float pop[2][1],C[4][1],D[4][1],E[4][1],pop2[2][1],pop3[2][1],temp[
2][1];
    int b[2][1],z[2][1],m[2][1],res[10][10],res2[10][10],res3[10][10];
// (b=Q0),(Z=Q1),(M=Q2)
    int a[2][2] = {{1,1},{1,-1}}; // Hadamard Gate
    int e[2][1] = {{0},{1}}; // Qubit |1> State
    int crot9[4][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,100}}; // C
ROT P(Pi/2)
    int crot4[4][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,200}}; // C
ROT P(Pi/4)
    float tensor[4][1] = {{0},{0},{0},{0}};  // Tensor Product Matrices
    float tensor2[4][1] = {{0},{0},{0},{0}}; // Tensor Product Matrices
    float tensor3[4][1] = {{0},{0},{0},{0}};
        printf("QFT-Implementation : EC203,EC222\n");
        printf("Enter The Input Qubit (Q0):\n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1 ; j++) {
                scanf("%d", & b[i][j]); // Scan User Input for Q0
            }
        }
        printf("Enter The Input Qubit (Q1):\n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1 ; j++) {
                scanf("%d", & z[i][j]); // Scan User Input for Q1
            }
        }
        printf("Enter The Input Qubit (Q2):\n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1 ; j++) {
                scanf("%d", & m[i][j]); // Scan User Input for Q2
            }
        }
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++) {
                res[i][j] = 0;
                for (k = 0; k < 2; k++) {
                    res[i][j] += a[i][k] * m[k][j];
                }
            }
        }

        printf("Hadamard O/P (Stage-I): \n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                printf("%f\t", 0.707 * res[i][j]);
                printf("\n");
        }
```

```
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                pop[i][j] = 0.707 * res[i][j];
        }

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 1; j++) {
            if (e[i][j] != z[i][j]) {
                equal = 0;}
            else{
                equal = 1;
                }

        }
    }

    if(equal == 1){
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            tensor[i*2 + j][0] = z[i][0] * pop[j][0];
        }
    }
    for (int j = 0; j < 4; j++) {
        C[0][j] = 0;
        for (int k = 0; k < 4; k++) {
            C[0][j] +=  crot9[k][j] * tensor[0][k];
        }
    }
    printf(" ");
    printf("\n");
    printf("CROT O/P (Stage-II): ");
    printf("\n");
    for (int j = 0; j < 4; j++){

        if(C[0][j] >= 60){
           printf("0.707i");
           printf("\n");
        }
        else{
           printf("%f\t",C[0][j]);
           printf("\n");
        }
    }


    }
```

```c
    else{
        printf("\n");
        printf("Stage-II : No Tensor");
        printf("\n");
    }

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 1; j++) {
            if (e[i][j] != b[i][j]) {
                qual = 0;}
            else{
                qual = 1;
                }


        }
    }

    if(qual == 1){
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            tensor2[i*2 + j][0] = b[i][0] * pop[j][0];
        }
    }
    for (int j = 0; j < 4; j++) {
        D[0][j] = 0;
        for (int k = 0; k < 4; k++) {
            D[0][j] +=  crot4[k][j] * tensor2[0][k];
        }

    }
    printf(" ");
    printf("\n");
    printf("CROT O/P (Stage-III): ");
    printf("\n");
    for (int j = 0; j < 4; j++){

        if(D[0][j] >= 60){
           printf("0.707 + 0.707i");
           printf("\n");
        }
        else{
           printf("%f\t",D[0][j]);
           printf("\n");
        }
    }


    }
```

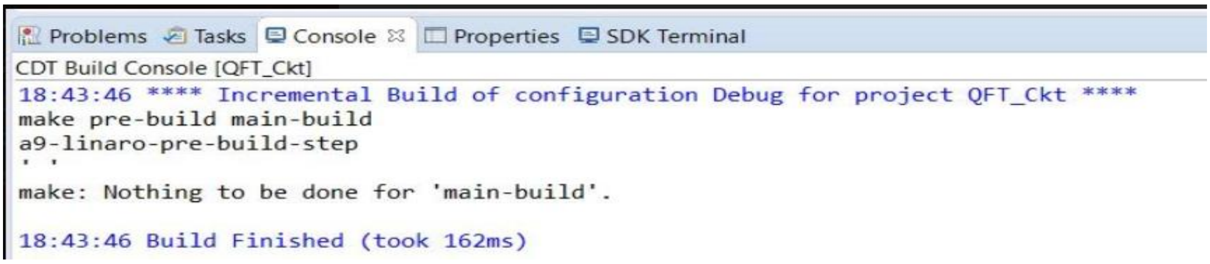
```
    else{
        printf("\n");
        printf("Stage-III : No Tensor");
        printf("\n");
    }
printf("\n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++) {
                res2[i][j] = 0;
                for (k = 0; k < 2; k++) {
                    res2[i][j] += a[i][k] * z[k][j];
                }
            }
        }
printf("Hadamard O/P (Stage-IV): \n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                printf("%f\t", 0.707 * res2[i][j]);
                printf("\n");
        }

        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                pop2[i][j] = 0.707 * res2[i][j];
        }

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 1; j++) {
            if (e[i][j] != b[i][j]) {
                ual = 0;}
            else{
                ual = 1;
                }

        }
    }

    if(ual == 1){
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            tensor3[i*2 + j][0] = z[i][0] * pop2[j][0];
        }
    }

    for (int j = 0; j < 4; j++) {
        E[0][j] = 0;
        for (int k = 0; k < 4; k++) {
            E[0][j] +=  crot9[k][j] * tensor3[0][k];
```

```
        }
    }
    printf(" ");
    printf("\n");
    printf("CROT O/P (Stage-V): ");
    printf("\n");
    for (int j = 0; j < 4; j++){

        if(E[0][j] < 0){
           printf("-0.707i");
           printf("\n");
        }
        else{
           printf("%f\t",E[0][j]);
           printf("\n");
        }
    }


    }

    else{
        printf("\n");
        printf("Stage-V : No Tensor");
        printf("\n");
    }

    for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++) {
                res3[i][j] = 0;
                for (k = 0; k < 2; k++) {
                    res3[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        printf("\n");
        printf("Hadamard O/P (Stage-VI): \n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                printf("%f\t", 0.707 * res3[i][j]);
                printf("\n");
        }

        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                pop3[i][j] = 0.707 * res[i][j];
        }
```

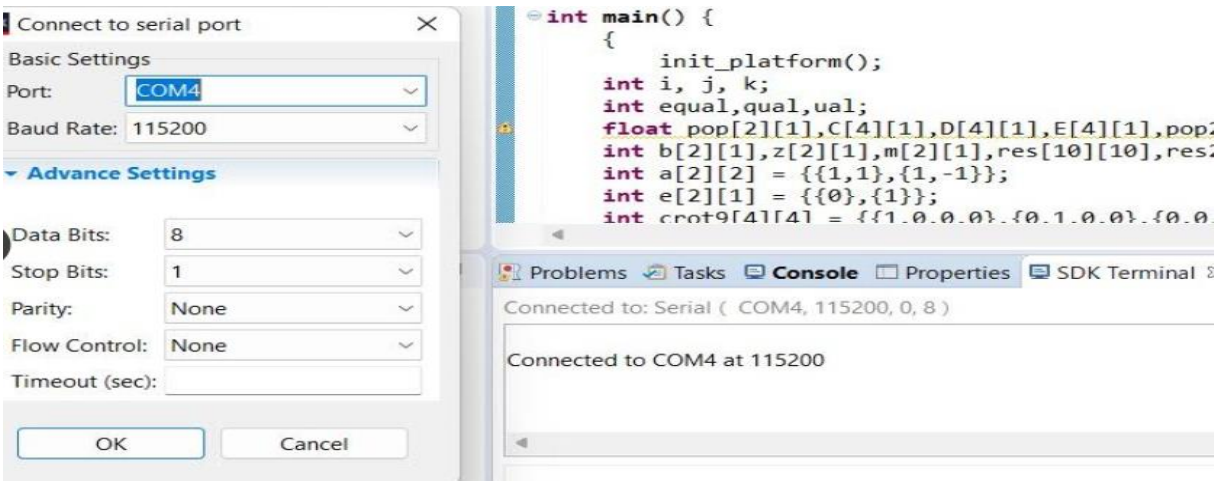
```
        printf("\n");
        printf("Stage-VII : Swap(q0 and q2) : New Q0 Value = \n");
            for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++) {
                printf("%f\t", 0.707 * res[i][j]);
                printf("\n");
        }
            }

        printf("Stage-VII : Swap(q0 and q2) : New Q2 Value = \n");
        for (i = 0; i < 2; i++) {
            for (j = 0; j < 1; j++)
                printf("%f\t", 0.707 * res3[i][j]);
                printf("\n");

        }

         }
      cleanup_platform();
      return 0;
}
```

OUTPUTS:

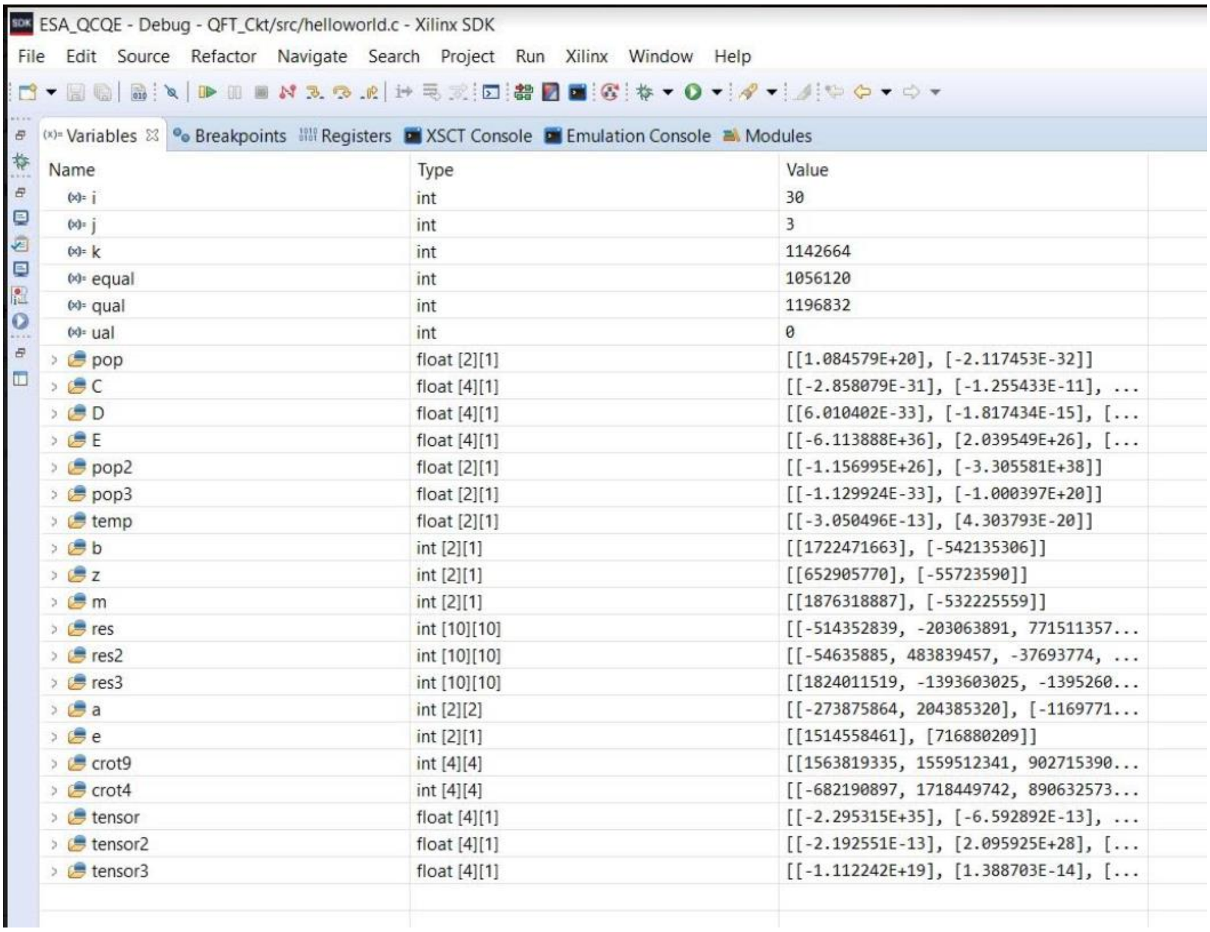
1) CDT BUILD CONSOLE:



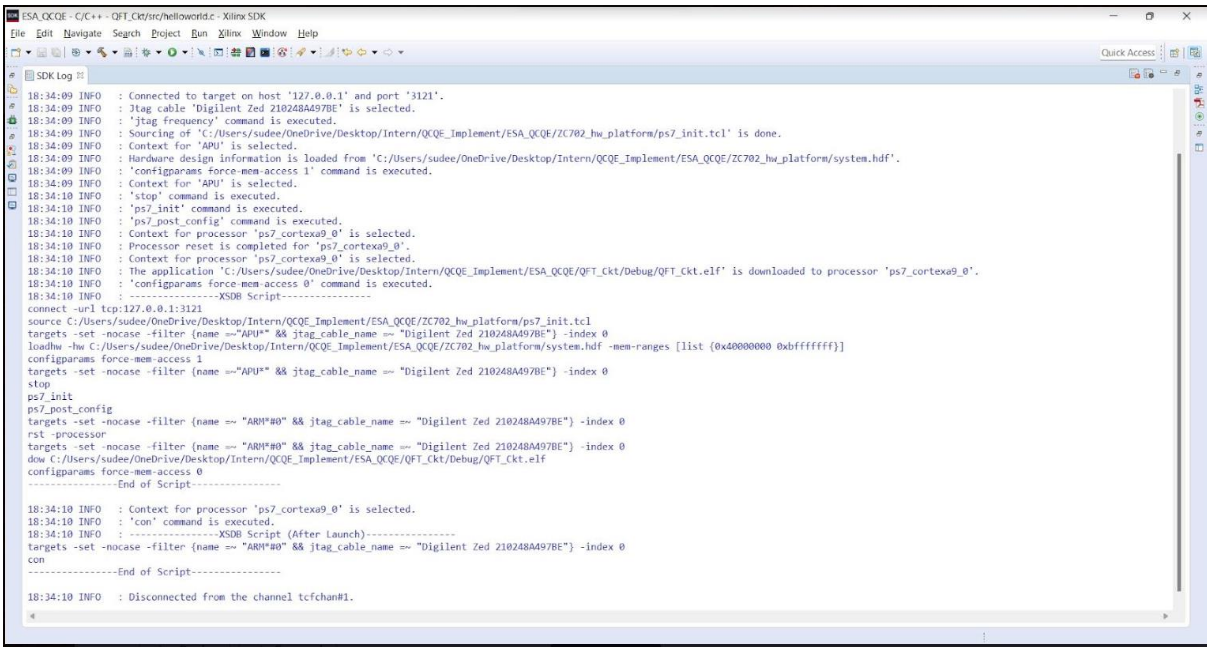
2) ZEDBOARD TO PC CONNECTION DETAILS:



3) VARIABLE DETAILS:



4) SDK LOG FILES:



5) OUTPUT FOR TEST CASE 1: (q0 = |0> state , q1 = |0> state , q2 = |0> state)

Hadamard O/P (Stage-I):

0.707000

0.707000

Stage-II : No Tensor

Stage-III : No Tensor

Hadamard O/P (Stage-IV):

0.707000

0.707000

Stage-V : No Tensor

Hadamard O/P (Stage-VI):

0.707000

0.707000

Stage-VII : Swap(q0 and q2) : New Q0 Value =
0.707000

0.707000

6) CPU UTILIZATION AND CPU IPC DETAILS:

7) REGISTER VALUES DETAILS:

Variables | Breakpoints | Registers | XSCT Console | Emulation Console | Modules

Name	Hex	Decimal	Description	Mnemonic
r0	00000000	0		
r1	00000000	0		
r2	00000001	1		
r3	00000000	0		
r4	00000003	3		
r5	0000001e	30		
r6	0000ffff	65535		
r7	f8f00000	4176478208		
r8	00000000	0		
r9	ffffffff	4294967295		
r10	00000000	0		
r11	0012432c	1196844		
r12	00124320	1196832		
sp	00123cf0	1195248		
lr	00101bb0	1055664		
pc	001005c8	1050056		
> cpsr	600000df	1610612959		
> usr				
> fiq				
> irq				
> abt				
> und				
> svc				
> mon				
> vfp				
> cp15				
> Jazelle				
> gpv_qos301_cpu			AMBA Quality of Service for CPU-to-DDR	
> gpv_qos301_dmac			AMBA Quality of Service for DMAC	
> gpv_qos301_iou			AMBA Quality of Service for IOU	
> gpv_trustzone			AMBA NIC301 TrustZone	
> l2cache			L2 cache PL310	
> mpcore			Application Processing Unit	

8)XSCT PROCESS DETAILS:

```
XSCT Process
Setting PC to Program Start Address 0x00100000
Successfully downloaded C:/Users/sudee/OneDrive/Desktop/Intern/QCQE_Implement/ESA_QCQE/QFT_Ckt/Debug/QFT_Ckt.elf
Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: tcfchan#13 closed
xsct% Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x116914 (Suspended)
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x0 (Vector Catch)

Downloading Program -- C:/Users/sudee/OneDrive/Desktop/Intern/QCQE_Implement/ESA_QCQE/QFT_Ckt/Debug/QFT_Ckt.elf
    section, .text: 0x00100000 - 0x00116f7b
    section, .init: 0x00116f7c - 0x00116f93
    section, .fini: 0x00116f94 - 0x00116fab
    section, .rodata: 0x00116fb0 - 0x0011ad5b
    section, .data: 0x0011ad60 - 0x0011b76b
    section, .eh_frame: 0x0011b76c - 0x0011b76f
    section, .mmu_tbl: 0x0011c000 - 0x0011ffff
    section, .ARM.exidx: 0x00120000 - 0x00120007
    section, .init_array: 0x00120008 - 0x0012000b
    section, .fini_array: 0x0012000c - 0x0012000f
    section, .bss: 0x00120010 - 0x0012032f
    section, .heap: 0x00120330 - 0x0012232f
    section, .stack: 0x00122330 - 0x00125b2f

  0%    0MB   0.0MB/s  ??:?? ETA
100%    0MB   0.4MB/s  00:00

Setting PC to Program Start Address 0x00100000
Successfully downloaded C:/Users/sudee/OneDrive/Desktop/Intern/QCQE_Implement/ESA_QCQE/QFT_Ckt/Debug/QFT_Ckt.elf
Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x116918 (Suspended)
XUartPs_RecvByte() at xuartps_hw.c: 104
104:    while (!XUartPs_IsReceiveData(BaseAddress)) {
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x0 (Vector Catch)

Downloading Program -- C:/Users/sudee/OneDrive/Desktop/Intern/QCQE_Implement/ESA_QCQE/QFT_Ckt/Debug/QFT_Ckt.elf
    section, .text: 0x00100000 - 0x00116f7b
    section, .init: 0x00116f7c - 0x00116f93
    section, .fini: 0x00116f94 - 0x00116fab
    section, .rodata: 0x00116fb0 - 0x0011ad5b
    section, .data: 0x0011ad60 - 0x0011b76b
    section, .eh_frame: 0x0011b76c - 0x0011b76f
    section, .mmu_tbl: 0x0011c000 - 0x0011ffff
    section, .ARM.exidx: 0x00120000 - 0x00120007
    section, .init_array: 0x00120008 - 0x0012000b
    section, .fini_array: 0x0012000c - 0x0012000f
```

9) OUTPUT FOR TEST CASE 2: (q0 = |1> state , q1 = |1> state , q2 = |0> state)

```
Hadamard O/P (Stage-I):|
0.707000

0.707000


CROT O/P (Stage-II):

0.000000

0.000000

0.707000
0.707i


CROT O/P (Stage-III):

0.000000
0.000000
0.707000

0.707 + 0.707i

Hadamard O/P (Stage-IV):

0.707000

-0.707000


CROT O/P (Stage-V):
0.000000

0.000000

0.707000

-0.707i


Hadamard O/P (Stage-VI):

0.707000

-0.707000


Stage-VII : Swap(q0 and q2) : New Q0 Value =

0.707000

0.707000
Stage-VII : Swap(q0 and q2) : New Q2 Value =

0.707000

-0.707000
```