Custom Instruction Design for Quantum Computers: An Experimental Study on Q-ISA Development

Venkatakrishnan Sutharsan[a,1]

[a] *Texas A&M University*

Professor: Sunil P Khatri

Abstract—This report delves into the world of Quantum Instruction Set Architectures (QISAs), the need for QISAs and explores the potential of defining custom instructions within the Quil language. Quil, a popular QISA for superconducting quantum processors, offers a flexible framework for building quantum circuits. By extending Quil with custom instructions, we can potentially enhance the expressiveness and efficiency of quantum programs. This expansion also gives the ability to develop operations that are complex to implement directly in Quil Language. We have added support for ADD (Addition) of 2 qubits using a half-adder circuit implementation and for QFT (Quantum Fourier Transform) which is supported only for 3 qubits(as of now). The investigations also adds to the point that having classical equivalence is possible but doesn't necessarily need to be present as the classical counterpart is easier to implement.

keywords—*quantum superposition, quantum entanglement, quantum instruction set architecture,*

1. Introduction

The late 20th century saw a surge in research on optimizing computer performance, pushing the boundaries of what was thought possible. This led to a look into various avenues of parallel computation like instruction level parallelism, data level parallelism, and thread level parallelism. Similarly, physicists in the mid-1920's started to reason some results about black-body radiation and photoelectric effect which was later found to be not explainable using classical physics. The formulation of Heisenberg's and Schrödinger's wave mechanics marked the birth of "**Modern Quantum Mechanics**" establishing a formal framework for the discipline of Quantum Physics. Therefore, computer scientists inspired by Quantum Mechanics and various quantum principles started to develop "**Quantum Computing**" which relied majorly on 2 quantum principles of Superposition and Entanglement. Quantum Computing has gained a lot of traction in the last decade with more and more demonstrations of a practical quantum computer. Recently, Google has demonstrated a quantum computer with an order of 10^2 qubits and with an average logical error rate of 10^-2. This quantum computing has also proved that algorithms that take exponential time (as size increases) in a classical computer were multi-fold faster in a quantum computer. This performance has inspired computer scientists to develop computers using quantum logic.

2. Basics of Quantum Computing

Quantum computers operate on the principles of quantum mechanics, a world vastly different from our everyday experience. Unlike classical computers that rely on bits, which can be either 0 or 1, quantum computers use qubits. The qubits store values similar to a bit in a classical computer, but this has to abide by the laws of quantum mechanics.

2.1. What is a Qubit?

In quantum computing, a quantum bit, or qubit (sometimes qbit) is a unit of quantum information. It is the fundamental unit of information, similar to how a bit is a basic unit in classical computers. These qubits can exploit 2 important phenomena from Quantum mechanics called Quantum Superposition and Quantum Entanglement.

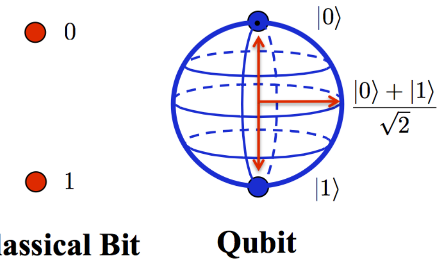

Figure 1. Example representation of a Qubit

In Figure. 1 we can notice that a classical bit can either be in the state of 0 or 1. The corresponding equivalent of 0 and 1 in the quantum domain is $|0\rangle$ and $|1\rangle$ respectively. Also, there can be a state of superposition meaning 0 and 1 can co-exist together in the same qubit.

2.2. Quantum Superposition

Quantum Superposition can be explained from Quantum Mechanics as a co-existence of multiple states (here 0 and 1 simultaneously). Superposition allows a single qubit to explore multiple possibilities at once. This is particularly powerful when dealing with complex problems that have many potential solutions. By harnessing superposition, quantum computers can perform calculations exponentially faster than classical computers for certain problems. A coin spinning in the air is a classical analogy (not a perfect one) for superposition. Until you observe it (by looking or catching it), the coin is both heads and tails with a certain probability of landing on either side. This can be expressed as the following equation:

$$\frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{1}$$

The equation 1 shows the probability of predicting 0 (tails) and 1(heads) with a chance of 50% for each state. A possible application is to factorize a number to break the encryption of a system.

2.3. Quantum Entanglement

Quantum entanglement is an interesting phenomenon in quantum mechanics where two or more particles become linked in a special way resulting in similar behavior. These entangled particles share a single quantum state, regardless of the physical distance separating them, which means these 2 qubits in a system will have a correlation, and thus measuring any one of the qubits will determine the result of the other qubit.

In the same kind of example as above, let us say we toss 2 coins in a classical world which can give us four combinations of results which are (00, 01, 10, 11) where 0 is tails and 1 is heads. But in a quantum world where these 2 qubits are entangled, we will get only (00, 11) if they are positively entangled or (01, 10) if they are negatively entangled. In either cases, we will have only 2 outcomes and not 4 (in classical we have 4 outcomes). We can see that both qubits in this entangled scenario are perfectly correlated. Thus, by measuring the first qubit and knowing about the type of entanglement we can know the second qubit without even measuring.

2.4. Qubit Representation

Information in quantum space is described by a state in a 2-level quantum mechanical system which is formally equivalent to a two-dimensional vector space over the complex numbers. The two basis states (or vectors) are conventionally written as $|0\rangle$ and $|1\rangle$ (pronounced: 'ket 0' and 'ket 1') as this follows the usual bra-ket notation of writing quantum states. Hence a qubit can be thought of as a quantum mechanical version of a classical data bit. A pure qubit state is a linear quantum superposition of those two states. This means that each qubit ($|\psi\rangle$) can be represented as a linear combination of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2}$$

The probability that the qubit will be measured in the state $|0\rangle$ is α^2 and the probability that it will be measured in the state $|1\rangle$ is β^2. Hence the total probability of the system being observed in either state $|0\rangle\, or\, |1\rangle$ is 1. From the generalized expression 2 we can say that the square of the co-efficient of the pure states, $\alpha and \beta$ should add up to one. This can be mathematically expressed as:

$$\alpha^2 + \beta^2 = 1 \tag{3}$$

2.5. Need for Quantum ISA

With the above-mentioned advantages of quantum over classical computing, we require developing an instruction set architecture (ISA) for quantum computing as this is going to be the future for complex problem-solving. We need a structured development so that many complex algorithms can be implemented without the need for knowledge of quantum mechanics (providing an abstract view to the programmer).

3. Available Quantum ISA

Quantum computing is a new technology that has gained traction in the last decade, which limits the number of available Q-ISA. There are a couple of ISAs which are made specifically for Quantum Computing and they are as follows:

- OpenQASM by IBM (currently at v3.0) - used in Qiskit [3],
- eQASM [1] by Delft University of Technology (Netherlands), derived from OpenQASM,
- cQASM by QuTech,
- Quil by Rigetti Computing (Berkeley, USA) [4]

4. Quil: a Quantum Instruction Language

4.1. Introduction to Quil Instruction Language

Quil is an instruction-based language for representing quantum computations with classical feedback and control. It can be written directly for quantum programming, used as an intermediate format within classical programs, or used as a compilation target for quantum programming languages. The Quantum Abstract Machine (QAM) is an abstract representation of a general-purpose quantum computing device. It includes support for manipulating both classical and quantum states. The Quantum Abstract Machine (QAM) is an abstract representation of a general-purpose quantum computing device. It includes support for manipulating both classical and quantum states. The state of the QAM is specified by the following elements:

- A fixed but arbitrary number of qubits N_q indexed from 0 to $N_q - 1$. The k^{th} qubit is written Q_k. The state of these qubits is written $|\psi\rangle$ and is initialized to $|00...0\rangle$.
- A classical memory C of N_c bits, initialized to zero and indexed from 0 to N_c - 1. The k^{th} bit is written C[k].
- A fixed but arbitrary list of static gates G, and a fixed but arbitrary list of parametric gates G'.
- A fixed but arbitrary sequence of Quil instructions P.
- An integer program counter k counting from 0 to mod P indicating the position of the next instruction to execute.

The 6-tuple $(|\psi_i\rangle, C, G, G', P, k)$ summarizes the state of the QAM. The QAM executes programs represented in a quantum instruction language called Quil, which has well-defined semantics in the context of the QAM. Most importantly, Quil defines what can be done with the QAM. Quil has support for:

- Applying arbitrary quantum gates,
- Defining quantum gates as optionally parameterized complex matrices,
- Defining quantum circuits as sequences of other gates and circuits, which can be bit and qubit parameterized,
- Expanding quantum circuits,
- Measuring qubits and recording the measurements into classical memory,
- Synchronizing execution of classical and quantum algorithms,
- Branching unconditionally or conditionally on the value of bits in classical memory, and
- Including files as modular Quil libraries such as the standard gate library

A classical implementation of Quil is done on a QVM (Quantum Virtual Machine) and to implement on a real hardware we use QPU (Quantum Processing Unit). In the next subsection, we will look at how the Quil Language can be interpreted and used to define gates for quantum operations.

4.2. Quil Language Syntax and Semantics

4.2.1. Classical Address and Qubits

A qubit is referred to by its integer index. For example, Q_5 is referred to by 5. This can be expressed as:

```
MEASURE 5
```
(4)

A classical memory address is referred to by an integer index in square brackets. For example, the address 7 pointing to the bit C[7] is referred to as [7]. The same example above can be changed to:

```
MEASURE 5 out[0]
```
(5)

the above change means that we can read the Qubit 5 and store the value in classical memory named "out" bit 0. The above examples show 2 types of measurements which are:

- Measurement for Effect - which will affect the qubit and make it to collapse, Expression 4 gives an example of this.
- Measurement for Record - will measure the qubit and store the measured value into a classical memory pointed by the operation. Expression 5 gives an example of this type of measurement.

4.2.2. Static and Parametric Gate

There are two gate-related concepts in the QAM: static and parametric gates. A static gate is an operator in U(2^{N_q}), and a parametric gate is a function C^n -> U(2^{N_q}), where the n complex numbers are called parameters.

A static 2 qubit gate named "GATE" acts on 2 qubits Q_0 and Q_1 which is operated on quantum space $\mathcal{B}_0$ and $\mathcal{B}_1$ is defined as:

```
GATE 0 1
```

A parametric three-qubit gate named "PGATE" with a single parameter $\exp^{-i\pi/7}$ acting on Q_1, Q_2, and Q_3, which is an operator lifted from $\mathcal{B}_1 \oplus \mathcal{B}_2 \oplus \mathcal{B}_3$, is written in Quil as

```
PGATE(0.9009688679-0.43388373911i) 1 2 3
```

Quil Language Example - Bell State

Let us try to create a Bell State Circuit using the static gate definition mentioned above. For creating a Bell State Circuit we need a Hadamard Gate (H) and a Contolled-NOT (CNOT) gate. Let us create a Bell Circuit with Qubit 0 and 1:

```
        DECLARE out BIT[2]
        H 0
        CNOT 0 1
        MEASURE 0 out[0]
        MEASURE 1 out[1]
```

The above code does create a memory called out with 2 bits in classical memory named "out". It connects qubit 0 to Hadamard Gate (H) and Controlled-NOT (CNOT) to qubit 0 and 1

4.2.3. Defining a Custom Gate

A custom gate can be defined in Quil using the DEFGATE directive and followed by the matrix operation performed by that gate. For example, let us create a HADAMARD Gate. With the definition of HADAMARD gate we know the matrix of it. So, it can be defined as:

```
        DEFGATE HADAMARD:
            1/sqrt(2), 1/sqrt(2)
            1/sqrt(2), -1/sqrt(2)
```

With the above example, we can see that we can create custom gate-definition in Quil which is useful to do some complex operations over a qubit or even multiple qubits. This gate is provided in Quil by default as "H" gate.

5. Methodology

One of the main features of this project is to create custom instructions using Quil Language as the building block. This can be accomplished in Python programming language as Quil has support in python with a package called as pyQuil. This python package will actually create the Quil equivalent of your indented code by definition of gates at python level. Therefore, a person doesn't need to understand Quil to program as this python package can help with creating that.

The installation of pyQuil is not a simple process as it has dependencies with QVM (Quantum Virtual Machine) and QUILC(QUIL Compiler) which has a very long process to define. To make life easier we have moved towards the use of docker-based environment provided by Rigetti Computing. Inc.

We are already familiar with certain operations in the Quantum domain like superposing gate (H -gate) and Toffoli Gate (CCNOT -gate) and have also seen how to create a custom gate in Quil. With that being said, lets dive into the main part of the project - to get some custom instructions implemented using pyQuil and test them out.

We are defining a function called "**custom_instruction**" which has the function prototype as the following:

```
custom_instruction(<program_quil_object>,
<instruction_type>, * parameters_for_instruction)
```

This function will accept normal gates like Hadamard, Controlled-NOT, Toffoli and even NOT(Negate). These set of gates form a Universet Gate set which are useful to form any combination. Apart from these basic gates, we have also implemented custom instructions like "ADD" using half-adder which adds 2 qubits and gives sum and a carry. This is one of the instructions that we have developed and is not a part of Quil Language. We have also added support for 3-Qubit Quantum Fourier Transform (QFT) which is also an added feature of this project and is not a part of Quil Language by default.

The next section (6) will provide insights into the realization of these operations and how they are implemented in our program.

6. Implementation

The pyQuil library facilitates the development of quantum programs using Quil. Understanding pyQuil package is easy and we will show an example to demonstrate the implementation of a Pauli-X gate for qubit flipping, serving as a foundational example for understanding both the library and the Quil language itself.

```
1  from pyquil import Program, get_qc
2  from pyquil.gates import *
3
4  def view_graph(p):
5      print(p)
6
7  p = Program()
8  ro = p.declare(’ro’, ’BIT’, 1)
9  p += X(0)
10 p += MEASURE(0, ro[0])
11
12 view_graph(p)
13
14 qc = get_qc(’1q-qvm’)  # You can make any ’nq-qvm’ this way for any reasonable ’n’
15 executable = qc.compile(p)
16 result = qc.run(executable)
17 bitstrings = result.get_register_map().get(’ro’)
18 print(bitstrings)
```

Code 1. Basic Quantum-NOT gate operation using Pauli-X gate using pyQuil

The above code is to demonstrate the Qubit flipping using Pauli-X gate. Lines 1-2 import the required packages (mainly pyQuil library). Lines 4-5 define a function to print the dynamic graph that is created on quantum program. Line 7 defines the Program object from pyQuil to use as the main program. Line 8-10 define the memory declaration, operator X (Pauli-X gate) on qubit 0 and to measure the qubit 0 into the memory we defined. Lines 14-18 are to setup the simulation environment and simulate to print the results. The output of the above program can be seen below.

```
DECLARE ro BIT[1]
X 0
MEASURE 0 ro[0]

[[1]]
```

Code 2. Output of Pauli-X gate

We can see that the output list has the value 1 meaning it has changed the value $|0\rangle$ to $|1\rangle$. Now, we will implement our definition of the **custom_instruction** function which will perform as we have discussed in 5.

The custom_instruction function is defined as follows:

```
def custom_instruction(p, instr, *param):
    # print(param)
    if (instr == "superposition"):
        """ Instruction to do superposition on a qubit mentioned in the call.
        Usage: custom_instruction(p, "superposition", "h0", 0)
        """
        p += Declare(param[0],"BIT", 1)
        p += H(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "rotate"):
        """ Instruction to do rotation on a qubit mentioned by a certain angle.
        Usage: custom_instruction(p, "rotate", "X",
        """
        if (param[0] == "X" or param[0] == "x"):
            p += RX(param[1], param[2])
        elif (param[0] == "Y" or param[0] == "y"):
            p += RY(param[1], param[2])
        elif (param[0] == "Z" or param[0] == "z"):
            p += RZ(param[1], param[2])
    elif (instr == "NEG" or instr == "neg"):
        """ Instruction to do negation of a Qubit specified by the user/compiler
        Usage: custom_instruction(p, "neg", 1)
        """
        p += Declare(param[0],"BIT", 1)
        p += X(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "ADD" or instr == "add"):
        """ Instruction to addition of 2 bits.
        Usage : custom_instruction(p, "add", "add_qubit", 0, 1, 2)
```

```python
            Param[0] - Name of memory you want to instantiate for this instruction result to be stored.
            Param[1] - The Qubit Number you want to use as input 1.
            Param[2] - The Qubit Number you want to use as input 2.
        """
        p += Declare(param[0],"BIT", 2)
        p += CCNOT(param[1], param[2], param[3])
        p += CNOT(param[1], param[2])
        p += MEASURE(param[2], (param[0], 0))  # Sum
        p += MEASURE(param[3], (param[0], 1))  # Carry
    elif (instr == "QFT" or instr == "qft"):
        """ Instruction to perform Quantum Fourier Transform on given number of bits.
        NOTE: Use only with 3-Qubits..!
        Usage: custom_instruction(p, "QFT", "qft_result", [1, 2, 3])
        """
        num_qubits = len(param[1])
        q0 = param[1][0]
        q1 = param[1][1]
        q2 = param[1][2]
        p += Declare(param[0],"BIT", num_qubits)
        # Apply Hadamard to all qubits
        p += H(param[1][0])
        p += H(param[1][1])
        p += H(param[1][2])

        # Apply controlled phase rotations
        for i in range(2, 0, -1):
            for j in range(i):
                control_qubit = param[1][j]
                target_qubit = param[1][j+1]
                angle = pi / (2**i)
                p += CPHASE(angle, control_qubit, target_qubit)
        p += MEASURE(param[1][0], (param[0], 0))
        p += MEASURE(param[1][1], (param[0], 1))
        p += MEASURE(param[1][2], (param[0], 2))
        # Apply SWAP gate for bit-reversal order (optional)
        # p += SWAP(q0, q2)
    elif (instr == "MOV" or instr == "mov"):
        """ Instruction to move value from one qubit to another qubit
        Usage: custom_instruction(p, "MOV", 1, 2)
        """
        # Can also be done using 3-CNOT gate
        p += Declare(param[0],"BIT", 2)
        p += SWAP(param[1], param[2])
        p += MEASURE(param[1], (param[0], 0))
        p += MEASURE(param[2], (param[0], 1))
```

Code 3. Python Implementation of Custom Instruction Function

The function has multiple instructions being implemented and we can see that there are instructions like ADD (add 2 qubits using half adder), QFT (quantum fourier transform operating on 3 qubits), MOV (move value similar to MOV instruction in x86), NEG (negate a qubit), rotate (rotation gate which is a parametric gate and the user can input the axis and the angle to apply to the rotation). In the next section 7 we will now look at how these functions in Python can be used in applications.

7. Custom Instruction Examples

In this section, we will see how we can use the implemented **custom_instruction** function to develop complex logics that we want to work on. We will now see a demonstration of "ADD" instruction which adds 2 qubits (qubits 0 and 1 in this demonstration).

7.1. Quantum Addition

```python
# A fully connected QVM
number_of_qubits = 6
qc = get_qc(f"{number_of_qubits}q-qvm")
# Initialize a program
p = Program().wrap_in_numshots_loop(10)

def custom_instruction(p, instr, *param):
    # print(param)
    if (instr == "superposition"):
        """ Instruction to do superposition on a qubit mentioned in the call.
        Usage: custom_instruction(p, "superposition", "h0", 0)
        """
        p += Declare(param[0],"BIT", 1)
        p += H(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "rotate"):
        """ Instruction to do rotation on a qubit mentioned by a certain angle.
        Usage: custom_instruction(p, "rotate", "X",
        """
        if (param[0] == "X" or param[0] == "x"):
            p += RX(param[1], param[2])
        elif (param[0] == "Y" or param[0] == "y"):
            p += RY(param[1], param[2])
        elif (param[0] == "Z" or param[0] == "z"):
            p += RZ(param[1], param[2])
    elif (instr == "NEG" or instr == "neg"):
        """ Instruction to do negation of a Qubit specified by the user/compiler
        Usage: custom_instruction(p, "neg", 1)
        """
        p += Declare(param[0],"BIT", 1)
        p += X(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "ADD" or instr == "add"):
        """ Instruction to addition of 2 bits.
        Usage : custom_instruction(p, "add", "add_qubit", 0, 1, 2)
            Param[0] - Name of memory you want to instantiate for this instruction result to be stored.
            Param[1] - The Qubit Number you want to use as input 1.
            Param[2] - The Qubit Number you want to use as input 2.
        """
        p += Declare(param[0],"BIT", 2)
        p += CCNOT(param[1], param[2], param[3])
        p += CNOT(param[1], param[2])
        p += MEASURE(param[2], (param[0], 0))  # Sum
        p += MEASURE(param[3], (param[0], 1))  # Carry
    elif (instr == "QFT" or instr == "qft"):
        """ Instruction to perform Quantum Fourier Transform on given number of bits.
        NOTE: Use only with 3-Qubits..!
        Usage: custom_instruction(p, "QFT", "qft_result", [1, 2, 3])
        """
        num_qubits = len(param[1])
        q0 = param[1][0]
        q1 = param[1][1]
        q2 = param[1][2]
        p += Declare(param[0],"BIT", num_qubits)
        # Apply Hadamard to all qubits
        p += H(param[1][0])
        p += H(param[1][1])
        p += H(param[1][2])

        # Apply controlled phase rotations
        for i in range(2, 0, -1):
            for j in range(i):
                control_qubit = param[1][j]
                target_qubit = param[1][j+1]
                angle = pi / (2**i)
                p += CPHASE(angle, control_qubit, target_qubit)
        p += MEASURE(param[1][0], (param[0], 0))
        p += MEASURE(param[1][1], (param[0], 1))
        p += MEASURE(param[1][2], (param[0], 2))
        # Apply SWAP gate for bit-reversal order (optional)
        # p += SWAP(q0, q2)
    elif (instr == "MOV" or instr == "mov"):
        """ Instruction to move value from one qubit to another qubit
        Usage: custom_instruction(p, "MOV", 1, 2)
        """
        # Can also be done using 3-CNOT gate
        p += Declare(param[0],"BIT", 2)
        p += SWAP(param[1], param[2])
        p += MEASURE(param[1], (param[0], 0))
        p += MEASURE(param[2], (param[0], 1))

#  Applying Negate to Qubit 0 and 1
custom_instruction(p, "NEG", "neg_result1", 0)
# Applying Add operation to Qubit 0, 1 with Qubit 2 as 0 to get carry directly.
custom_instruction(p, "ADD", "add_result1", 0, 1, 2)
```

```

view_graph(p)

result = qc.run(qc.compile(p)).get_register_map()
wavefunc = wf_sim.wavefunction(p)
print(wavefunc.amplitudes)
print(len(wavefunc.amplitudes))

print(result)
```

Code 4. A code for quantum addition using our custom_instruction function

The above code can be explained as we create a 6-qubit system for quantum simulation, then we negate qubit 0 using our custom_instruction which will give us basis 1 ($|1\rangle$) and then applying addition to qubit 0 and 1 giving us the output as follows:

```
DECLARE neg_result1 BIT[1]
DECLARE add_result1 BIT[2]
X 0
MEASURE 0 neg_result1[0]
CCNOT 0 1 2
CNOT 0 1
MEASURE 1 add_result1[0]
MEASURE 2 add_result1[1]

[0.+0.j 0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
8

{'add_result1': array([[1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0],
       [1, 0]]), 'neg_result1': array([[1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1]])}
```

Code 5. The output for our quantum addition using custom_instruction function

The output is inferred as there are 10 trails as we mentioned about the wrap_in_numshots_loop(10) meaning to run 10 trails. So we have 10 outputs in the result for each value we measure. Also, since we run the code with values of $|0\rangle$ and $|1\rangle$ thus we are sure to get the sum as 1 and carry as 0 shown as

`array[1, 0]`

We can provide inputs that are in superposition like the output of a Hadamard Gate and then feed to this circuit with that so we can compute the

7.2. Quantum Fourier Transform

Quantum Fourier Transform is one of the most powerful applications of a classical discrete Fourier transform but in the quantum domain, as it is easier to compute in this domain. It decomposes a signal into its constituent frequencies.

```
# A fully connected QVM
number_of_qubits = 6
qc = get_qc(f"{number_of_qubits}q-qvm")
# Initialize a program
p = Program().wrap_in_numshots_loop(10)

def custom_instruction(p, instr, *param):
    # print(param)
    if (instr == "superposition"):
        """ Instruction to do superposition on a qubit mentioned in the call.
        Usage: custom_instruction(p, "superposition", "h0", 0)
        """
        p += Declare(param[0],"BIT", 1)
        p += H(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "rotate"):
        """ Instruction to do rotation on a qubit mentioned by a certain angle.
        Usage: custom_instruction(p, "rotate", "X",
        """
        if (param[0] == "X" or param[0] == "x"):
            p += RX(param[1], param[2])
        elif (param[0] == "Y" or param[0] == "y"):
            p += RY(param[1], param[2])
        elif (param[0] == "Z" or param[0] == "z"):
            p += RZ(param[1], param[2])
    elif (instr == "NEG" or instr == "neg"):
        """ Instruction to do negation of a Qubit specified by the user/compiler
        Usage: custom_instruction(p, "neg", 1)
        """
        p += Declare(param[0],"BIT", 1)
        p += X(param[1])
        p += MEASURE(param[1], (param[0], 0))
    elif (instr == "ADD" or instr == "add"):
        """ Instruction to addition of 2 bits.
        Usage : custom_instruction(p, "add", "add_qubit", 0, 1, 2)
            Param[0] - Name of memory you want to instantiate for this instruction result to be stored.
            Param[1] - The Qubit Number you want to use as input 1.
            Param[2] - The Qubit Number you want to use as input 2.
        """
        p += Declare(param[0],"BIT", 2)
        p += CCNOT(param[1], param[2], param[3])
        p += CNOT(param[1], param[2])
        p += MEASURE(param[2], (param[0], 0))  # Sum
        p += MEASURE(param[3], (param[0], 1))  # Carry
    elif (instr == "QFT" or instr == "qft"):
        """ Instruction to perform Quantum Fourier Transform on given number of bits.
        NOTE: Use only with 3-Qubits..!
        Usage: custom_instruction(p, "QFT", "qft_result", [1, 2, 3])
        """
        num_qubits = len(param[1])
        q0 = param[1][0]
        q1 = param[1][1]
        q2 = param[1][2]
        p += Declare(param[0],"BIT", num_qubits)
        # Apply Hadamard to all qubits
        p += H(param[1][0])
        p += H(param[1][1])
        p += H(param[1][2])

        # Apply controlled phase rotations
        for i in range(2, 0, -1):
            for j in range(i):
                control_qubit = param[1][j]
                target_qubit = param[1][j+1]
                angle = pi / (2**i)
                p += CPHASE(angle, control_qubit, target_qubit)
        p += MEASURE(param[1][0], (param[0], 0))
        p += MEASURE(param[1][1], (param[0], 1))
        p += MEASURE(param[1][2], (param[0], 2))
        # Apply SWAP gate for bit-reversal order (optional)
        # p += SWAP(q0, q2)
    elif (instr == "MOV" or instr == "mov"):
        """ Instruction to move value from one qubit to another qubit
        Usage: custom_instruction(p, "MOV", 1, 2)
        """
        # Can also be done using 3-CNOT gate
        p += Declare(param[0],"BIT", 2)
        p += SWAP(param[1], param[2])
        p += MEASURE(param[1], (param[0], 0))
        p += MEASURE(param[2], (param[0], 1))

# Negating Qubit 3 to get 1 basis and fo QFT on Qubits 3,4,5
custom_instruction(p, "NEG", "neg_result", 3)
custom_instruction(p, "QFT", "qft_result", [3, 4, 5])

view_graph(p)

```

```
result = qc.run(qc.compile(p)).get_register_map()
wavefunc = wf_sim.wavefunction(p)
print(wavefunc.amplitudes)
print(len(wavefunc.amplitudes))

print(result)
```

Code 6. A example program for Quantum Fourier Transform using our custom_instruction function

The below snippet shows the result of the Quantum Fourier Transform which we have developed in out custom_instruction function.

```
DECLARE neg_result BIT[1]
DECLARE qft_result BIT[3]
X 3
MEASURE 3 neg_result[0]
H 3
H 4
H 5
CPHASE(0.7853981633974483) 3 4
CPHASE(0.7853981633974483) 4 5
CPHASE(1.5707963267948966) 3 4
MEASURE 3 qft_result[0]
MEASURE 4 qft_result[1]
MEASURE 5 qft_result[2]

[0.+0.j 0.+0.j 1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.
     j]
8

{'qft_result': array([[1, 0, 0],
        [1, 0, 1],
        [0, 0, 0],
        [1, 1, 1],
        [0, 1, 1],
        [0, 0, 1],
        [1, 0, 1],
        [0, 0, 1],
        [0, 1, 0],
        [0, 0, 0]]), 'neg_result': array([[1],
        [1],
        [1],
        [1],
        [1],
        [1],
        [1],
        [1],
        [1],
        [1]])}
```

Code 7. Result for out QFT instruction

Therefore, we can say that we have some basic building blocks for a quantum instruction set architecture and by extending this we can develop a more robust Q-ISA which is capable of handling multi-qubit and complex circuits.

8. Limitations of Q-ISA

- Quantum Instruction Set Architecture proves to give advantages to certain operations and/or tasks which can be useful. However, the construction and implementation of quantum logic which can manipulate multi-qubits are difficult.
- Logics also require circuits to hold information over which provide and store qubits in a non-volatile memory, which is not available at large for quantum systems.
- Classical operations are much easier to execute in a classical computer (like memory copy) and therefore Quantum Computing will be most efficient when used as an extension in the classical ISA.

Therefore, I would suggest that we have a quantum extension rather than creating an extensive quantum architecture which makes things difficult. Also for example, vector processing is an extension in RISC-V named as "V" extension, similarly, we can have n-32 Qubit in RISC-V as an extension which can do wonders that are not possible in classical systems.

9. Future Work

This project can be used and extended to create a compiler for a quantum computer. **This can be used as a backend for a compiler** which constructs the necessary instructions based on the logic of the user program.

Also, with constraints based on time, we were able to implement a subset of operations while this **can be extended to various other operations** and make it more generalizable.

References

[1] X. Fu et al. *eQASM: An Executable Quantum Instruction Set Architecture*. 2019. arXiv: 1808.02449 [cs.AR].

[2] Cupjin Huang et al. "Quantum Instruction Set Design for Performance". In: *Physical Review Letters* 130.7 (Feb. 2023). ISSN: 1079-7114. DOI: 10.1103/physrevlett.130.070601. URL: http://dx.doi.org/10.1103/PhysRevLett.130.070601.

[3] Naoki Kanazawa et al. "Qiskit Experiments: A Python package to characterize and calibrate quantum computers". In: *Journal of Open Source Software* 8.84 (2023), p. 5329. DOI: 10.21105/joss.05329. URL: https://doi.org/10.21105/joss.05329.

[4] Robert S. Smith, Michael J. Curtis, and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. 2017. arXiv: 1608.03355 [quant-ph].