# The QuDot Virtual Machine: QuDotVM

# 1 Introduction

As we enter the era of Universal Fault-Tolerant Quantum Computation (UFTQC) our programming tools are going to be just as important as our hardware. Suppose you are an AI researcher, materials engineer or data scientist and someone gives you a quantum computer: now what? QuDot is here to show you what to do with a quantum computer. The first step in making UFTQC friendly to developers is to develop a virtual machine architecture where higher level abstractions of UFTQC can execute.

QuDot separates UFTQC into four main components depicted in Fig 1:

1. The **Quantum Front Ends** are the top level of the UFTQC stack. They are tools that allow a developer to incorporate quantum computation in their applications without being a quantum physicists. These front ends can include anything from Python bindings (QiSkit, Cirq), domain specific languages (Q#) and direct framework integration such as integration with Google's OpenFermion framework.

2. The **QuDot SDK** provides high level tools to compile front end application programs to QuDot bytecode for execution on the QuDotVM. Users of the QuDot platform can continue to use their favorite tools to program quantum circuits and the QuDot SDK will allow them to run the same program on the QuDotVM. Once users are comfortable with their results they can start submitting jobs to their hardware providers of choice.

3. **The QuDot Virtual Machine: QuDotVM** can read a qudot bytecode file and perform UFTQC. It is based on the QuDot instruction set explained in section 3. The QuDotVM also knows how to handle ensembles of programs. Given the probabilistic nature of UFTQC quantum programmers are going to run a program multiple times and expect a probability distribution as output instead of a deterministic value.

4. The **Quantum Compute Bridge: QuCB** ties in a probability distribution generated by the QuDotVM and the classical program that called it. UFTQC will be a mix of classical and quantum computing constructs. The QuCB serves as the bridge between the two. As probability distributions are generated by the QuDotVM the QuCB manages the communication of the quantum results with the classical callers.

# 2 The QuDot Virtual Machine

The QuDotVM is a bytecode, register based virtual machine backed by QuDot patent pending technology: Quantum Multiverse Networks (QuMvN). The QuDotVM allows users to simulate their quantum circuits on classical hardware for a fraction of the cost of quantum hardware up to a number of qubits. The QuDotVM has the following components:

1. An instruction set composed of both classical instructions and quantum instructions capable of simulating UFTQC

2. The Feynman Unit that handles the application of all quantum gates

3. The EPR Unit that handles entanglement detection for speeding up quantum computations

4. The Many Worlds Logic Unit (MWLU) which handles arithmetic operations (similar to the ALU) in a many worlds context. For example, in quantum computing your operation may have a carry bit enabled in one world but not in another. Lastly, the MWLU manages the scratch qubits needed in arithmetic operations. Addition requires $2n + 1$ scratch qubits and multiplication requires $n$ scratch qubits.

5. The Heisenberg Unit handles all aspects of measurement and generating a probability distribution from a quantum state

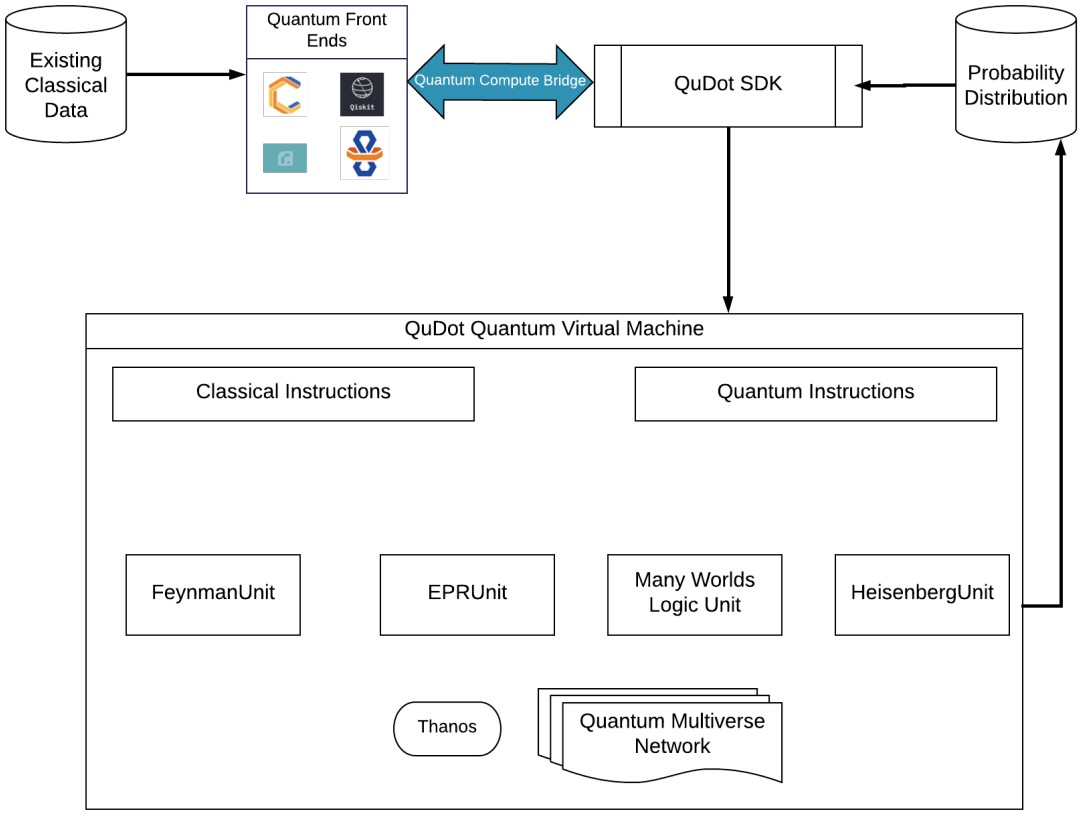


Figure 1: The QuDotVM Architecture

# 3 The QuDot File Format

A QuDotVM implementation knows how to read a program from a *.qudotc* file which is a byte representation of QuDir. We use the below conventions to describe the *.qudotc* byte file:

**bN**: An N byte integer, b1 (1 byte), b4 (4 byte) etc

**b[]**: An array of bytes

**gateInfo**: A sequence of bytes defining a gate

**constPoolInfo**: A sequence of bytes defining a constant pool entry

```
qudot_file {
        b4 VERSION
        b4 numQubits
        b4 ensembleSize
        gateInfo mainGate
        b4 constPoolSize
        constPoolInfo[] constPool
        b[] code
}
```

```
gateInfo {
        b4 nameLength
        b[] name (In US-ASCII Character Set)
        b4 args (number of argument)
        b4 regs (number of registers)
        b4 qubitRegs (number of qubit registers)
        b4 codeAddress (gate address in code[])
}
```

```
constPoolInfo {
        b1 type
        b4 length
        b[] info
}
```

The types supported in the constant pool are defined as:

1. GATE = 1

The last part of the qudot file is a byte array with a sequence of bytecodes to be executed by the QuDotVM. The bytecodes must comply with the QuDot VM Instruction Set detailed in the next section.

# 4 The QuDot VM Instruction Set

The QuDotVM supports the following quantum gate operation in the bytecode specification:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \end{bmatrix}$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & \exp\frac{i\pi}{4} \end{bmatrix} \quad R(k) = \begin{bmatrix} 1 & 0 \\ 0 & \exp\frac{2\pi i}{2^k} \end{bmatrix}$$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad CROT(k) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp\frac{2\pi i}{2^k} \end{bmatrix}$$

| Instruction | Code | Args | Example | Summary |
|---|---|---|---|---|
| halt | 0 | | halt | Ends the program |
| paths | 1 | | paths | Prints all possible states (tomography). This operation can be exponential in the number of qubits |
| x | 2 | | x | Apply the $X$ Gate on the entire circuit |
| y | 3 | | y | Apply the $Y$ Gate on the entire circuit |
| z | 4 | | z | Apply the $Z$ Gate on the entire circuit |
| s | 5 | | s | Apply the $S$ Gate to the entire circuit |
| sdag | 63 | | sdag | Apply the $S^\dagger$ Gate to the entire circuit |
| t | 6 | | t | Apply the $T$ Gate to the entire circuit |
| tdag | 65 | | tdag | Apply the $T^\dagger$ Gate to the entire circuit |
| phi | 7 | REG | phi r5 | Apply the $R(k)$ Gate to the entire circuit where $k$ is stored in the register operand |
| phidag | 62 | | phidag r5 | Apply the $R^\dagger(k)$ Gate to the entire circuit |
| h | 8 | | h | Apply the H Gate to the entire circuit |
| swap | 9 | | swap | Apply the swap operation to the entire circuit |
| swap_ab | 10 | QUREG, QUREG | swap_ab q3, q7 | swap two qubits |
| measure | 11 | | measure | Perform a measurement |
| cnot | 12 | QUREG, QUREG | cnot q2, q5 | Perform a control-Not where first argument is the control and the second argument is the target |
| crot | 13 | REG, QUREG, QUREG | crot q2, q5 | Perform a control-R(k) operation where k is in REG |
| xon | 16 | QUREG | xon q3 | Apply $X$ to qubit register |
| yon | 17 | QUREG | yon q3 | Apply $Y$ to qubit register |
| zon | 18 | QUREG | zon q3 | Apply $Z$ to qubit register |
| son | 19 | QUREG | son q3 | Apply $S$ to qubit register |
| sdagon | 64 | QUREG | sdagon q3 | Apply $S^\dagger$ to qubit register |
| ton | 20 | QUREG | ton q3 | Apply $T$ to qubit register |
| tdagon | 66 | QUREG | tdagon q3 | Apply $T^\dagger$ to qubit register |
| phion | 21 | REG, QUREG | phion r6, q3 | Apply $R(k)$ to specified register where k is in REG |
| phidagon | 62 | QUREG | phidagon q3 | Apply $R^\dagger(k)$ to qubit register |
| hon | 22 | QUREG | hon q3 | Apply $H$ to qubit register |
| mon | 23 | QUREG | mon q3 | Measure qubit register |
| qload | 25 | QUREG, INT | qload q7, 24 | Load the qubit index into the specified QUREG |
| qload_array | 26 | QUREG, INT, INT, INT, ... | qload_array q1, 3, 2, 7, 9 | Load an array of qubits into a qubit register |
| iadd | 27 | REG, REG, REG | iad r3, r1, r2 | Integer addition $r3 = r1 + r2$ |

| | | | | |
|---|---|---|---|---|
| isub | 28 | REG, REG, REG | isub r3, r1, r2 | Integer subtraction $r3 = r1 - r2$ |
| imul | 29 | REG, REG, REG | imul r3, r1, r2 | Integer multiplication $r3 = r1 * r2$ |
| ilt | 30 | REG, REG, REG | ilt r3, r1, r2 | r3 has the boolean value of $r1 < r2$ |
| ieq | 31 | REG, REG, REG | ieq r3, r1, r2 | r3 has the boolean value of $r1 == r2$ |
| incr | 32 | REG | incr r67 | incr the INT in r67 by 1 |
| br | 33 | INT | br 132 | Branch to code address 132 |
| brt | 34 | REG, INT | brt r4, 132 | Branch to code address 132 if r4 is True |
| brf | 35 | REG, INT | brf r4, 132 | Branch to code address 132 if r4 is false |
| iload | 36 | REG, INT | iload r4, 8 | Load the integer 8 into register r4 |
| ret | 37 | | ret | returns from a gate call |
| move | 38 | REG, REG | move r2, r3 | $r3 = r2$ |
| null | 39 | REG | null r5 | $r5 = null$ |
| call | 40 | GATE LABEL | call $bell(), r0$ | Calls a gate. First argument is a register. If $r0$ Then no arguments otherwise arguments are continuous registers from parameter |
| printr | 41 | REG | printr r5 | Prints the contents of r5 |
| qload_seq | 42 | QUREG, INT, INT | qload_seq q7, 1, 5 | Load the continuous sequence 1,2,3,4,5 into qubit register q7 |
| breq | 43 | REG, REG, INT | breq r6, r7, 27 | Branch to code address 27 if $r6 == r7$ |
| brgez | 44 | REG, INT | brgez r15, 69 | Branch to code address 69 if $r15 >= 0$ |
| brgtz | 45 | REG, INT | brgtz r17, 96 | Branch to code address 96 if $r17 > 0$ |
| brlez | 46 | REG, INT | brlez r1024, 75 | Branch to code address 75 if $r1024 <= 0$ |
| brltz | 47 | REG, INT | brltz r24, 1245 | Branch to code address 1245 if $r24 < 0$ |
| brneq | 48 | REG, INT | brneq r51, r65, 33 | Branch to code address 33 if $r51! = r65$ |
| qloadr | 49 | QUREG, REG | qloadr q5, r7 | Load value of r7 into q5 of |
| idiv | 50 | REG, REG, REG | idiv r5, r4, r2 | Integer division $r5 = r4/r2$ |
| decr | 51 | REG | decr r5 | Decrement integer contents of r5 by 1 |

| | | | | |
|---|---|---|---|---|
| toff | 52 | QUREG, QUREG | toff q1, q2 | The multi-control Toffoli Gate. q1 is the target qubit and q2 is a list of control qubits |
| iquadd | 53 | REG | iquadd r5 | Add integer contents of r5 to the quantum state |
| qft | 54 | QUREG, QUREG | qft q2, q3 | Quantum Fourier Transform intrinsic from qubit q2 to qubit q3 |
| qft_inv | 55 | QUREG, QUREG | qft q2, q3 | Inverse Quantum Fourier Transform intrinsic from qubit q2 to qubit q3 |
| iquadd_mod | 56 | REG, REG | iquadd_mod r5, r6 | Add integer contents of r5 modulo integer contents of r6 to the quantum state |
| iquadd_mul | 57 | REG, REG | iquadd_mul r5, r6 | Multiply integer contents of r5 modulo integer contents of r6 to the quantum state |
| ciquadd_mod | 58 | REG, REG, QUREG, QUREG, QUREG | ciquadd_mod r5, r6, q1, q2, q3 | Add integer contents of r5 modulo integer contents of r6 to qubits q1 through q2 controlled by q3 |
| ciquadd_mul | 59 | REG, REG, QUREG, QUREG, QUREG | ciquadd_mul r5, r6, q1, q2, q3 | Multiply integer contents of r5 modulo integer contents of r6 to qubits q1 through q2 controlled by q3 |

# 5 Examples

Listing 1: bell state

```
.qudot qubits=2, ensemble=1000000

.gate bell: args=0, regs=0, qubit_regs=2
  qload q0, 1
  qload q1, 2
  hon q0
  paths
  cnot q0, q1
  ret

.gate main: args=0, regs=0, qubit_regs=0
  printr r0
  call bell(), r0
  printr r0
  measure
  paths
  halt
```

Listing 2: GHZ state

```
.qudot qubits=20, ensemble=10000

.gate main: args=0, regs=2, qubit_regs=0
    iload r1, 1
    move  r2, r0
    call bell_n(), r1
    paths
    halt

.gate bell_n: args=2, regs=2, qubit_regs=3
    qloadr q0, r1
    move r3, r1
    hon q0
    iload r4, 1

    ghz:
      breq r3, r2, end
      qloadr q1, r3
      iadd r3, r3, r4
      qloadr q2, r3
      cnot q1, q2
      br ghz

    end:
      ret
```

Listing 3: Looping and branching

```
.qudot qubits=3, ensemble=1

.gate main: args=0, regs=5, qubit_regs=0
    iload r1, 0
    iload r2, 12
    call while_test(), r1
    call while_test_2(), r1
    iload r3, -5
    call if_else_test(), r3
    halt

.gate while_test: args=2, regs=3, qubit_regs=0
    iload r3, 1
    iload r4, 0
    loop:
        iadd r1, r1, r2
        isub r2, r2, r3
        ieq  r5, r2, r4
        brf r5, loop

    printr r1
    ret

// condensed version of above for loop using fewer registers and commands
.gate while_test_2: args=2, regs=1, qubit_regs=0
    iload r3, 1
    loop2:
        iadd r1, r1, r2
        isub r2, r2, r3
        brgtz r2, loop2

    printr r1
    ret

.gate if_else_test: args=1, regs=4, qubit_regs=0
    iload r2, 0
    ilt r3, r1, r2
    printr r3
    brf r3, else

    isub r4, r2, r1
    printr r4
    iload r5, 1
    iadd r4, r4, r5
    br next
    else:
        iload r5, 1
        iadd r4, r1, r5
    next:
        printr r4
        ret
```

Listing 4: Shor's Algorithm to factor 77 if we pick a random number 69

```
.qudot qubits=20, ensemble=100000
// 77 is a 7 bit number, we need 2n+1=15 scratch qubits
// these 15 qubits are allocated and handled by the MWLU
.gate main: args=0, regs=9, qubit_regs=7
    // control qubits start/end
    iload r1, 1
    iload r2, 13
    // modulo multiplication qubits start/end
    iload r3, 14
    iload r4, 20

    qload_seq q0, 1, 13
    qloadr q1, r3
    qloadr q2, r4

    // initialize state
    hon q0
    xon q2

    // number to factor
    iload r5, 77
    // random number
    iload r6, 69

    // setup loop variable
    move r7, r2
    iload r9, 0
    // modular exponatiation
    ModExp:
        brlez r7, doneModExp
        modpow r8, r6, r9, r5
        //printr r8
        qloadr q3, r7
        ciqumul_mod r8, r5, q1, q2, q3
        decr r7
        incr r9
        br ModExp

    doneModExp:
        // measure the second register
        qload_seq q4, 14, 20
        mon q4

        qloadr q5, r1
        qloadr q6, r2
        qft_inv q5, q6

        halt
```

Listing 5: Quantum Fourier Transform (note that this entire listing can be replaced by the qft bytecode)

```
.qudot qubits=4, ensemble=400000

.gate main: args=0, regs=2, qubit_regs=1
    qload_array q0, 2, 2, 4
    hon q0
    iload r1, 1
    iload r2, 4
    call qft(), r1
    halt

// two arguments:
//    r1: start qubit
//    r2: end qubit
.gate qft: args=2, regs=11, qubit_regs=3
    iload r3, 1
    // r4 -> q
    move r4, r1
    // r5 -> end qubit + 1
    iadd r5, 2, r3

    for1:
        breq r4, r5, donefor1
        qloadr q3, r4
        hon q3
        // i = q + 1
        iadd r6, r4, r3
        // r=2
        iload r7, 2
        while:
            breq r6, r5, donewhile
            qloadr q0, r4
            qloadr q1, r6
            crot r7, q0, q1
            // i+=1
            iadd r6, r6, r3
            // r+=1
            iadd r7, r7, r3
            br while

        donewhile:
            // q += 1
            iadd r4, r4, r3
            br for1

    donefor1:
        iload r8, 2
        // swap point + 1
        idiv r9, r2, r8
        iadd r9, r9, r3
        // q
        iload r10, 1
    for2:
        breq r10, r9, donefor2
```

```
        qloadr q0, r10
        isub r11, r2, r10
        iadd r11, r11, r3
        qloadr q1, r11
        swap_ab q0, q1
        iadd r10, r10, r3
        br for2

    donefor2:
        ret
```