A Software Development Kit and Translation Layer for Executing Intel 8080 Assembler on a Quantum Computer (August 2022)

JAMES FITZJOHN, ADRIAN WINCKLES, GEORGE WILSON, AND DOMENICO VICINANZA
Anglia Ruskin University, Cambridge CB1 1PT, U.K.
Corresponding author: James Fitzjohn (e-mail: mistasparkaru@hotmail.co.uk).

ABSTRACT One of the major obstacles to the adoption of quantum computing is the requirement to define quantum circuits at the quantum gate level. Many programmers are familiar with high-level or low-level programming languages but not quantum gates nor the low-level quantum logic required to derive useful results from quantum computers. The steep learning curve involved when progressing from quantum gates to complex simulations such as Shor’s algorithm has proven too much for many developers. The purpose of this article and the software presented within addresses this challenge by providing a Software Development Kit (SDK), translation layer, emulator, and a framework of techniques for executing Intel 8080/Z80 assembler on a quantum computer, i.e., all salient points of CPU execution, logic, arithmetic, and bitwise manipulation will be executed on the quantum computer using quantum circuits. This provides a novel means of displaying the equivalency and interoperability of quantum and classical computers. Developers and researchers can use the SDK to write code in Intel 8080/Z80 assembler which is executed locally via traditional emulation and remotely on a quantum computer in parallel. The emulator features side-by-side code execution with visibility of the running quantum circuit and reusable/overridable methods. This enables programmers to learn, reuse, and contrast techniques for performing any traditional CPU-based technique/instruction on a quantum computer, e.g., a programmer may know how to multiply and perform checks on a classical CPU but is not able to perform the same tasks in a quantum implementation, and this SDK allows the programmer to pick and choose the methods they would like to use to fulfil their requirements. The SDK makes use of open-source software, specifically Python and Qiskit for the emulation, translation, API calls, and execution of user-supplied code or binaries.

INDEX TERMS Emulation, macro and assembly languages, processors, Qiskit, quantum computing, quantum gates, translation.

I. INTRODUCTION
Quantum computers are strategically used to solve problems which traditional computers may never be able to complete. This is due to developers being able to take advantage of the fundamental differences in data-handling techniques resulting from superposition, entanglement, and the probabilistic nature of quantum computers. This, combined with the huge cost and the complexity of programming has led many to believe that quantum computers are to be treated as a separate entity or a black box: a device only to be interfaced with via API calls if there is a specific need to do so.

The traditional computing and quantum computing fields require a very different mindset to move from the logic-based approach found in traditional computing to a probability-based system found in quantum computing; nevertheless, many traditional logic-based methods are still required in quantum implementations.

The traditional computing field and its immense complexity are all built upon previous generations of code and hardware. Modern programming languages, operating systems, and software all use the same fundamental building blocks provided by the CPU. These building blocks can be traced...
back to the early 1970s when Intel released their 4004 CPU. This CPU set the path for modern x86 CPUs where each generation is backward compatible with the previous. In contrast, the quantum computing field is still in its infancy and suffers from a wide variety of limitations and complexities related to build (the difficulty in cooling LC circuits to near absolute zero), operation (the complex nature of programming quantum computers\(^1\)), capacity (the limited number of qubits available), and a lack of standardization. This article is mindful of the heritage of traditional computing and aims to bring some of the maturity and code reusability present in traditional CPU programming to the quantum computing field.

A. RATIONALE

The newness of this work is delivered in two aspects of the article. The first being a complete real-world/real-time realization of a traditional CPU emulated for execution on a quantum computer. The second is the development of an SDK\(^2\) which includes methods for all traditional CPU bitwise, arithmetic and logic functions executed on a quantum computer. The SDK also includes the emulator, translation layer, and an optional physical CPU interface. The practical side of this article implemented in the SDK aims to deliver some of the reusability present in traditional CPU programming to developers of quantum systems.

The Intel 8080/Z80 was chosen as the subject for this emulation and SDK due to its widespread use, direct lineage to modern CPUs and that its bus size and number of internal registers fit the number of available qubits. As and when systems with more qubits become available, the SDK can be extended to emulate more modern CPUs. This approach is novel in that it allows for Intel 8080/Z80 assembler to be executed on a quantum computer. This is in contrast with other researchers and developers [1], [2] who have concentrated on extending assembler (or other programming languages) to include functions to call quantum computers. The SDK also differs from other SDKs such as Qiskit which provide researchers with access to the quantum computer gates directly, whereas our SDK plugs into an existing workflow and demonstrates how to perform each instruction/task.

The motivation for this article and the SDK it describes is to provide a learning tool and reusable reference framework, a framework that will help bridge the steep learning curve and enable developers to drive useful results from quantum computers. The framework will show developers that quantum computers can do anything a traditional computer can, just in a different manner. The SDK also prepares the groundwork (via the delivery of the emulators and translation layer) for a possible scenario [3, Fig. 3] where the overhead for performing classical computations on a quantum computer is not so large. This is not to say that the value of the SDK is purely theoretical, as even with today’s capability many quantum-based simulations require aspects of traditional logic which are included as reusable methods within the framework.

Emulating a CPU was chosen as a goal for this article as it provides a provable mechanism for delivering every logic/arithmetic/bitwise function needed in a traditional computer. It is not our aim to reduce a quantum computer to a device only capable of traditional logic-based calculations but describe, prove, and demonstrate how to perform those calculations. The logic calculations can be used to form parts of many quantum-based calculations and are found in a framework which developers are familiar with and can readily reuse.

II. OVERVIEW

This article proposes an interesting engineering effort to begin exploring the integration of classical processors with quantum coprocessors with a unique twist. We provide an educational implementation of quantum arithmetic, logic, and bitwise routines and their equivalency to classical universal gates. This is demonstrated via the creation of a classical CPU emulator which is executed on a quantum computer, where every classical gate and circuit that makes up a traditional CPU is rewritten using quantum gates and circuits. It

\(^{1}\)Note the inherent difficulty results from the need to describe exponentially large state spaces and complex interactions such as entanglement, not an immaturity of the SDKs.

\(^{2}\)Due to intellectual property reasons the SDK will be publicly released via GitHub alongside the publication of this document. Prepublication demonstrations are available upon request.
is not our intention to limit the use of quantum computers to purely classical logic but to provide a real-world implementation of any classical function and describe how it is performed. This will allow the developer to pick and choose which instructions to reuse within the emulator or in their own quantum circuits. The SDK proposed in this article extends ideas published by Sohele et al. [4] and Vlatko et al. [5] by constructing all the remaining circuits (adder, subtractor, increment, decrement, latch, etc.). These quantum circuits are combined to form and emulate all CPU instructions including but not limited to LOAD (latch), Arithmetic (adder, etc.), Branch (C-not controlled) and bitwise logic (quantum-based logic gates) needed to implement a full instruction set. The quantum circuit-based instructions are then combined to create a fully functioning Intel 8080/Z80 Emulator [6].

The SDK contains three main components the first being an Intel 8080/Z80 emulator with pseudo RAM\(^3\), register\(^4\), visibility, and distinct methods for each instruction group. The second part contains an additional Python class which overrides the Intel 8080/Z80 instruction methods with new versions (see Fig. 1). The new methods perform the same instruction but with the salient function overridden for execution on a quantum computer via the creation of quantum circuits. The third part is a translation layer which works with the emulator to reroute execution of emulated instructions to IBMs quantum computer.

\(^3\)The term Pseudo RAM is used to describe a 2-D array of 64k × 8 boolean variables which is used as RAM for the emulation class.

\(^4\)A register is a small amount of internal (to the CPU) RAM, i.e., the Accumulator stores the result of the previous instruction and is a CPU register.
The emulator can execute the machine code directly from the Python control script or run binaries supplied at the command line. As execution progresses through the program, the instruction is described which includes a diagram of the quantum circuits needed to mirror the functionality and how the quantum gates affect the qubit input and output. This makes the SDK/Emulator a useful tool for teaching quantum computing or referencing how to perform a particular task.

The Python control script is used to orchestrate the emulation of the CPUs. The emulator translates calls to IBM's quantum computer while the control scripts handles the optional CPU interface which includes separate 64 k arrays of Booleans. The arrays are used by the traditional emulation, emulation performed on a quantum computer and an optional physical CPU interface as pseudo RAM. When the script is executed, the contents of all three RAM arrays will be loaded with a local test program or external binary. Once loading is complete, the arrays will be treated as separate entities to ensure the physical and emulated CPUs do not interfere with each other's operation.

The emulation code is contained in an external Python class (see Fig. 1), this class is initiated when the control script is executed and communicated with via the same method as a physical CPU, i.e., using a (virtual) address, data, clock, pins, etc. This method was chosen to mimic the physical CPUs connectivity as closely as possible, ensuring all three pillars have the same IO characteristics and can be compared directly. Like the physical CPU, the emulation classes will only perform their functions when they receive a clock signal on their (virtual) clock pin. The control script will ensure both the physical CPU and emulated CPUs receive a clock signal at the same time. The emulated CPUs will follow the fetch-decode-execute cycle as closely as possible when receiving clock signals (note, some instructions do not precisely follow this architecture). When the CPU has completed the execution of an instruction, it will increment its program counter and set its read pin to False to signify it is ready for another instruction and the cycle repeats.

III. USAGE
The SDK aims to enable developers to use a similar code-build-execute workflow they are used to, allowing developers to run traditional assembler/machine code on a quantum computer.

The developer first creates a sample Intel 8080/Z80 assembler program which can then be built with Vasm [7], a widely used portable assembler. Vasm accepts user-supplied assembler code written in a text file and builds it into a binary executable. The developer then executes the SDK control/emulator script with the binary output from Vasm as a command line option. This is a novel approach as most quantum computers are accessed via API/JSON, but in our approach the SDK abstracts this access via the emulator and translation layer which allows the developer to use the traditional workflow/tool chain they are familiar with. The control script will then execute the binary on the locally connected physical CPU (if available) in traditional software-based emulation and via the creation of quantum circuits which are executed on IBM's quantum computer.

When the code is executed, the script displays the execution flow, RAM, registers, and flags much like other emulators but also shows the quantum circuits created and executed on IBM's quantum computer in line with the execution (see Fig. 2). The control script demonstrates to the programmer how to perform any traditional logic task on a quantum computer and provides reusable code/circuits which they can use in other applications.

IV. EXAMPLE EXECUTION
This section describes how a developer can use the SDK to construct a simple Intel 8080/Z80 application and have it executed on IBM's quantum computer, including how it is translated and executed via quantum circuits. It is important to appreciate that the portion of the program executed on the quantum computer comprises the computational parts of the CPU execution i.e., its logic, arithmetic, and bitwise functions. To clarify the SDK provides the CPU emulator whilst all other components such as input/output, operating system, BIOS, and video output fall out of the scope of the SDK and are not emulated.

A. CREATE A SAMPLE ASSEMBLER PROGRAM
This example loads the CPU’s A register with 04h, increments the A register, loads the result into memory address 000f and then halts. The developer should copy the code in Listing 1 and save it to a text file named inc.asm

B. BUILD THE PROGRAM WITH VASM
The developer can now run the command shown in Listing 2 which uses the Vasm executable vasm80_oldstyle to build the supplied assembler file inc.asm into a binary named a.out.

```
ld a, $04
inc a
ld ($000f).a
halt
```

```
~/vasm $ ./vasm80_oldstyle -Fbin -dopt inc.asm
```

LISTING 1. A sample Intel 8080/Z80 assembler program with the increment instruction.

LISTING 2. Building the sample binary with Vasm.

5 A wiring diagram and component list is included as part of the SDK.
TABLE 1 Assembler to Machine Code Mapping

| Memory address | Machine code | Assembler |
|----------------|--------------|-----------|
| 000000         | 3E           | ld a      |
| 000001         | 04           | (With) $04 |
| 000002         | 3C           | inc a     |
| 000003         | 32           | ld        |
| 000004         | 0F           | ($000F),a |
| 000005         | 00           |           |
| 000016         | 76           | halt      |

Note: A mapping of the assembler code written in Listing 1 to the machine code stored in (pseudo) RAM for execution by the emulator class. The machine code shown in this table is visible in the 3 execution pillars of Fig. 2.

~ $ Python3 cpu3.py vasm/a.out

LISTING 3. Running the binary file a.out with the cpu3.py control script/emulator.

C. EXECUTE THE BINARY WITH THE CONTROL SCRIPT/EMULATOR

The developer now executes the command shown in Listing 3 to run Python3 which executes the emulator/control script cpu3.py with the user supplied binary a.out created in part B.

D. COMPARISON

Fig. 2 shows the sample binary being executed on all three pillars in parallel, where each pillar is displayed side by side for easy comparison. Note the machine code in RAM directly aligns to the assembler code written in step A and is listed in Table 1 for reference.

The screenshot in Fig. 2 shows the three pillars of execution, the physical CPU, the traditionally emulated CPU, and the CPU emulated for execution on a quantum computer. Each pillar has its own dedicated 64 k of pseudo RAM where the supplied binary is loaded. The pillars with the emulated CPUs include additional details such as register values. Note these are not directly visible on the physical CPU as displaying them would require customizing the user supplied code. A small number of quality-of-life methods and controls have been included, e.g., the ability to log all execution, scroll through RAM, and alter execution speed. These enhancements were included to aid the programmer’s ability to check how a particular instruction is executed. A portion of the stack, the internal registers and the quantum circuit currently being executed on the quantum computer can also be observed. The provided example shows a quantum AND circuit being executed as part of a larger increment circuit.

V. METHOD

A. METHOD OVERVIEW

A CPU is very good at accurately executing simple logic and mathematic functions extremely quickly, but it is important to remember that even the most complex task is performed by breaking it down until we reach the core logic gates of the CPU [8]. This layer of abstraction hides the complexity of modern computing, even a low-level language such as assembler does not directly interact at the gate level. However, when programming with quantum computers the standard practice for programming is at the quantum gate level, this is due to the availability of qubits being so low and means their use must be efficiently managed.

An interesting comparison can be drawn in that a traditional CPU executes all tasks at the gate level, but users have no visibility of this. This is in contrast with traditional low-level emulation where all gate-based logic must be implemented in software and users have full visibility of the logic gates. To emulate the CPU on a quantum computer these techniques must be merged and extended, as the programming typically uses a high-level language to implement low level quantum gates. Whilst some comparisons can be made, the quantum circuit model is not one-to-one compatible with a classical Instruction Set Architecture. One of the goals of this article is to show how this incompatibility can be alleviated via the translation layer and emulator.

B. TRADITIONAL EMULATION

The emulation of the CPU is handled in an external Python class to allow for the segregation of code, easier troubleshooting, and future code reuse. Where possible the emulation follows the fetch-decode-execute CPU flow.

Normal operation is fully instruction (documented) and cycle accurate [9]. The process involves setting the read pin to False (to mimic the active low/0v method used by the physical CPU) to signify that the CPU is ready to read or “Fetch” the next instruction. The emulated CPU sets the address bus bits to the value of the program counter, this tells the pseudo RAM to place the byte signified by the address bus on the data bus. The CPU then reads the byte on the data bus which is “Decoded” into the next instruction and is “Executed.” Any instruction which requires writing to RAM sets the read bit to True, the write bit to False and the address bus to the memory location to be written. This signifies to the Pseudo RAM that the CPU wants to write to the location on the address bus with the data on the data bus.

To aid code reuse and testing, emulation reuses (where possible and practical) methods for each instruction group, i.e., All ADD instructions will use the same method but accept and return specified registers as variables.

C. TRANSLATION OF EMULATION FOR EXECUTION ON A QUANTUM COMPUTER

The Execution of the CPU instruction on a quantum computer is performed in line with the traditional emulation to aid comparison. Each instruction is broken down to the logic
def myINC(self, register):
    x = register
    x = x + 1
    if x == 256:
        x = 0

    self.delay = 4  # The delay variable is used by the emulation class to mimic the number of clock ticks the physical CPU needs to complete the instruction.

    # Set flags
    Parity = self.getParity(x)
    Nflag = self.getNFlag(x)
    Carry = self.getCFFlag(x)
    HalfCarry = self.getHalfCarry(x)
    Zero = self.getZFlag(x)
    Signed = self.getSignedFlag(x)

    self.setFlags(Signed, Zero, HalfCarry, Parity, Nflag, Carry)
    return x

Listing 4. Pseudocode/Python method of emulating the Increment instruction.

We will now create the quantum circuits for a XOR gate equivalent by adapting techniques detailed by CP Williams in his publication “Explorations in Quantum Computing” [12], being mindful of the no deletion theorem [10] and ensuring the logic output matches that of a traditional XOR gate as shown in Table V.

Listing 5 shows the Pseudo Code Required to Implement the Quantum (Fig. 4) XOR Gate and Call IBM’s Quantum Computer for execution.

It is important to highlight that the circuit must be executed multiple times. Each “shot” or program execution yields a single execution outcome from the prepared probability distribution. It is therefore necessary to average the results of many execution shots in order to accurately compute the programs average values from the prepared distribution. This approach is also required to reduce the impact of noise (measurement/gate error/coherence). The code in Listing 5 achieves this by performing 2000 shots of the quantum circuit and then creates an integer average of the results. Increasing the number of shots will reduce the variance of the estimated outcome but comes at the cost of additional program executions. About 2000 shots was chosen as a good compromise between execution time and the diminishing returns in the emulated instruction’s accuracy.

This compromise was based on the results (see Fig. 5) of a simple experiment which set all qubits to 0, measured them, then set them to 1, and measured them again. The resultant percentage of error showed little improvement in accuracy after 2000 shots.

Returning to the increment example (see Fig. 3), we can see that we also require a quantum-based implementation of an AND gate. This quantum AND gate is subject to the same constraints as the XOR gate and is detailed in Fig. 6.

Listing 6 shows the pseudocode required to implement the quantum AND gate and call IBM’s quantum computer for execution.

gate level and is then built up using the equivalent quantum circuits. The quantum circuits have the same output as the traditional logic gates (see Table 5) but are constrained by the no deletion theorem [10]. The quantum logic circuits are combined [11] to construct larger and more complex circuits such as full adders, subtractors, latch, increment, etc. This process is demonstrated in listings 4 onward where the traditionally emulated instruction INC (increment) adds 1 to the input, sets the relevant CPU flags, and returns the output. The methods are then overridden utilizing the quantum circuit methods myXor(inputA,inputB) and myAnd(inputA,inputB) to construct a quantum based increment circuit.

It is important to note that individually each CPU instruction, quantum circuit, logic circuit, etc., is relatively simple, but breaking the execution down to this level is required to map the instructions to Python to the quantum circuits needed to implement the CPU. Despite being a relatively simple 8-bit/16-bit CPU the emulation required over 1300 instructions to be written and overridden. The INC instruction is illustrated to represent an instruction that is relatively easy to follow as opposed to many more complex instructions that have been implemented using a similar process.

The first step in the process is to create a method that emulates the increment instruction, which is detailed in the pseudocode in Listing 4. Note incrementing the x variable only takes 2 lines of code, the remaining code is required to handle timing and the setting of flags. As we can see adding 1 to a number is a simple process but ensuring it is instruction and cycle accurate is much more complex.

The next step is to reference the standard increment circuit shown in Fig. 3 to identify the logic circuits required to override the method for execution on a quantum computer. In this example we can see that we need to implement quantum equivalent XOR and AND gates.
Fitzjohn et al.: SOFTWARE DEVELOPMENT KIT AND TRANSLATION LAYER FOR EXECUTING INTEL 8080 ASSEMBLER

def myXor(self,inputA,inputB):
    circuit = q.QuantumCircuit(2,2)
    circuit.ry(inputA * math.pi,0)
    circuit.ry(inputB * math.pi,1)
    circuit.cx(0,1)
    circuit.measure(0,0)
    circuit.measure(1,1)
    circuitText = str(circuit)
    self.statusLine = "Executing a Quantum XOR circuit on " + str(backend) + " with the input " + str(inputA) + " and " + str(inputB)
    job = q.execute(circuit, backend=backend, shots=2000, memory=True)
    result = job.result()
    memory = result.get_memory(circuit)
    qubit0 = 0
    qubit1 = 0
    print("#Run through each shot of the quantum circuit creating an integer average and returning a 0 or 1 for result in memory:")
    qubit0 = qubit0 + int(result[0])
    qubit1 = qubit1 + int(result[1])
    qubit0 = qubit0/2000 #Number of shots
    qubit1 = qubit1/2000
    return int(round(qubit1))

LISTING 5. Pseudocode/Python implementation of a XOR gate in a quantum circuit.

def myAnd(self,inputA,inputB):
    circuit = q.QuantumCircuit(3,3)
    circuit.ry(inputA * math.pi,0)
    circuit.ry(inputB * math.pi,1)
    circuit.ccx(0,1,2)
    circuit.measure(0,0)
    circuit.measure(1,1)
    circuit.measure(2,2)
    circuitText = str(circuit)
    self.statusLine = "Executing a Quantum AND circuit on " + str(backend) + " with the input " + str(inputA) + " and " + str(inputB)
    job = q.execute(circuit, backend=backend, shots=2000, memory=True)
    result = job.result()
    memory = result.get_memory(circuit)
    qubit0 = 0
    qubit1 = 0
    qubit2 = 0
    print("#Run through each shot of the quantum circuit creating an integer average and returning a 0 or 1 for result in memory:")
    qubit0 = qubit0 + int(result[0])
    qubit1 = qubit1 + int(result[1])
    qubit2 = qubit2 + int(result[2])
    qubit0 = qubit0/2000 #Number of shots
    qubit1 = qubit1/2000
    qubit2 = qubit2/2000
    return int(round(qubit2))

LISTING 6. Pseudocode/Python implementation of a AND gate in a quantum circuit.

We can now override the traditional INC method detailed in listing 4 using the quantum logic gate methods described in listings 5 and 6.

The flexible nature of an object orientated programming language like Python allows for the overriding of methods in multiple classes, which in turn allows us to easily implement valued peer feedback. For example, Listing 7 is useful for quantum computers with very few qubits with the drawback that it prevents the user from exploring entanglement across multiple qubits. Another drawback of this approach is that it adds execution delays each time data is passed to and from the quantum computer. A quantum computer with more qubits (24) could keep the entire calculation in the quantum domain, reducing the execution delays to a minimum. Fig. 7 and Listing 8 show how it is possible to override the INC instruction, but this time the entire calculation is kept in the quantum domain.

Using Table 2 as a high-level reference, we have rebuilt each of the 1300+ instructions in the instruction set via the use of the 6 base quantum logic gate circuits which are combined into larger quantum (combination) circuits [11]. We used the same method implemented in the increment circuit,
def myINC(self,register):
    #Construct an increment quantum circuit from the
    Quantum logic gates
    output0 = self.myXor[1,inputA[0]]
    temp = self.myAnd[1,inputA[1]]
    output1 = self.myXor[temp,inputA[1]]
    temp = self.myAnd(temp,inputA[2])
    output2 = self.myXor(temp,inputA[2])
    temp = self.myAnd(temp,inputA[3])
    output3 = self.myXor(temp,inputA[3])
    temp = self.myAnd(temp,inputA[4])
    output4 = self.myXor(temp,inputA[4])
    temp = self.myAnd(temp,inputA[5])
    output5 = self.myXor(temp,inputA[5])
    temp = self.myAnd(temp,inputA[6])
    output6 = self.myXor(temp,inputA[6])
    temp = self.myAnd(temp,inputA[7])
    output7 = self.myXor(temp,inputA[7])
    x = self.setRegister(output7, output6, output5, output4,
                         output3, output2, output1, output0)

    self.delay = 4  # The delay variable is used by the
    emulation class to mimic the number of clock ticks the
    physical CPU needs to complete the instruction. In this
    method an INC instruction takes 4 clock ticks to complete.

    # Set flags
    Parity = self.GetParity(x)
    Nflag = self.getNflag(self.F)
    Carry = self.getCflag(self.F)
    HalfCarry = self.GetHalfCarry(x)
    Zero = self.getZflag(x)
    Signed = self.getSignedflag(x)

    self.setFlags(Signed, Zero, Carry, Parity, Nflag, Carry)
    return x

LISTING 7. Pseudocode/Python method which Overrides the Increment
instruction using the equivalent quantum circuits, XOR and AND.

def myINC(self, input7, input6,..., input0):
    # A secondary implementation of an increment circuit
    executed in the quantum domain. Note the reversed input
    order to make working from LSB to MSB easier.
    incrementbit = 1 * math.pi
    input0 = input6 * math.pi
    #.... Repeat for all input bits
    input7 = input7 * math.pi

    circuit = q.QuantumCircuit(25,25)
    circuit.ry(incrementbit,0)
    circuit.ry(input0,1)
    circuit.ex(0,1)
    circuit.ry(input0,2)
    circuit.ry(incrementbit,3)
    circuit.cxx(2,3,4)
    circuit.barrier(0...x)
    circuit.ry(input1,5)
    circuit.cx(4,5)
    circuit.ry(input1,6)
    circuit.cxx(6,4,7)
    circuit.barrier(0...x)
    #.... Repeat for all input bits
    circuit.ry(input7,23)
    circuit.cx(22,23)
    circuit.ry(input7,24)
    circuit.cxx(24,22,25)
    circuit.barrier(0...x)
    circuit.measure(0,0)
    #.... Repeat for all input bits
    circuit.measure(24,24)

    job = q.execute(circuit, backend=backend, shots=2000,
                    memory=True)
    result = job.result()

    memory = result.get_memory(circuit)
    qubit0 = 0
    #.... Repeat for all input bits
    qubit24 = 0

    for result in memory:
        qubit0 = qubit0 + int(result[0])
        #.... Repeat for all input bits
        Qubit24 = qubit24 + int(result[24])

    qubit0 = qubit0/2000 # Number of shots
    #.... Repeat for all input bits
    qubit24 = qubit24/2000

    return str(int(qubit24)) + '.... Repeat ' + str(int(qubit1))

LISTING 8. Secondary pseudocode/Python method, which overrides the
Increment instruction entirely in the quantum domain.

---
i.e., calling the quantum logic gate methods to mimic how
the instruction would be implemented “in silicon” as part of a
real CPU. Extra effort was made to implement reusable compo-
"nents where possible; for example, the PUSH instruction
reuses a 16-bit variation of the DEC instruction to reduce
errors, aid reliability, and promote reuse.

The SDK and fully functioning emulator provide a unique
platform to explore alternative approaches for overriding and
implementing the instructions. Every instruction has been
 TABLE 2 Instruction Grouping

| Instruction | Description |
|-------------|-------------|
| INC         | Increment a value (add 1) using an increment circuit |
| DEC         | Decrement a value (subtract 1) using a decrement circuit |
| ADD         | Add two numbers using a (full) adder circuit |
| SUB/CP      | Subtract/Compare two numbers using a subtractor circuit (CP checks if the result equals 0) |
| AND         | Logical AND on the bits of two numbers |
| OR          | Logical OR on the bits of two numbers |
| XOR         | Logical XOR on the bits of two numbers |
| LOAD        | Load a register or memory address with another using multiple latch circuits |
| EX          | Exchange registers for their shadow versions using multiple load instruction |
| PUSH        | Save a register to the stack using an INC and LOAD instruction |
| POP         | Load a register from the stack using a DEC and LOAD instruction |
| NOP         | Do nothing |
| JUMP/CALL   | Move the program counter to a point of execution by LOADing the program counter with a value |
| JUMP IF     | JUMP dependent on an IF condition using a JUMP instruction inline with conditional IF |
| RET         | Return from routine by LOADing the program counter |
| RST         | ROM calls |
| ROT/SLA     | Logical BIT rotation using multiple latch circuits |
| SET/RES     | Set/Reset a bit in memory or register using latch circuits |
| NEG         | Subtracts A from 0 using a SUB instruction |
| BIT         | LOADs the opposite value of the bit to the Z flag |
| Misc        | Multiple miscellaneous instructions such as SCP, DAA, etc |

Note: Every variation of these instructions has been implemented in traditional emulation and for execution on a quantum computer within the SDK.

TABLE 3 Latch Circuit Truth Table

| Set – Reset | Q – Ω |
|-------------|-------|
| 0 – 0       | N/A   |
| 0 – 1       | 0 – 1 |
| 1 – 0       | 1 – 0 |
| 1 – 1       | N/A   |

Note: Valid inputs for a latch circuit are the combinations Set 0, Reset 1 and Set 1, Reset 0 to store a logical 0 or 1.

FIGURE 8. Quantum circuit which entangles qubits 0 and 1 via a Hadamard and a controlled not gate.

TABLE 4 Hadamard Based (Entanglement) Quantum Circuit Truth Table

| qubit0 (Set) – qubit1 (Reset) | Measured bit 0 and bit 1 |
|-----------------------------|--------------------------|
| 0 – 0                       | (50%) 00 and (50%) 11    |
| 0 – π                       | (50%) 01 and (50%) 10    |
| π – 0                       | (50%) 01 and (50%) 11    |
| π – π                       | (50%) 01 and (50%) 10    |

Note: A measurement of the entanglement quantum circuit detailed in Fig.8.

LISTING 10. Retrieving latch equivalent results from a Hadamard circuit.

```python
def quantumPUSH(self, register):
    my16BitDEC(StackPointer)
    my16BitDEC(StackPointer)
    quantumLOAD(StackPointer, register)
```

To derive the same logic output as the traditional latch circuit we can label qubit0 as “Set,” qubit1 as “Reset,” and xor the measured bits (see Table 4, highlighted in gray).

This approach simplifies the quantum circuit required to implement a latch and all subsequent SET, RES, and LOAD instructions. However, this approach requires a change from integer averaging the results from the quantum circuit to recording the two most common results, which is not seen as detrimental.

VI. RESULTS

A. TRADITIONAL EMULATION

Frank Cringles Z80 instruction exerciser [13] was used to test all variations of every documented instruction. Each test in the instruction exerciser runs through a group of similar instructions, i.e., all 16-bit “ADD” instructions or all variants of “INC.” The tests will set registers to a known value, execute every variation of the instruction, and then compare the results to a known CRC value. Additional tests were performed where the expected CRC value was changed to an arbitrary...
value, this ensured the logic would catch an incorrect CRC match. The Emulator ran through a full test suite of 67 test groupings successfully, being instruction and cycle accurate. The test suite confirmed that the emulated CPUs matched the physical CPU and expected CRC value in every test.

B. EMULATION EXECUTED ON A QUANTUM COMPUTER

The translation layer and execution of emulation on a quantum computer was performed in November 2021 on IBM’s ibmq_BelkorFalconr4T, a 5-qubit, quantum computer with a quantum volume of 16. This quantum computer was chosen to represent a “worst case scenario” for noise and capability [14] and to ensure no special requirement exists [15] should researchers wish to recreate the results. Subsequent methods (Listing 8) which keep the entire calculation in the quantum domain use more qubits and require at least a Falcon r8 class quantum computer.

Table 5 shows the average value of the output qubit after 2000 shots/measurements of the individual quantum logic gates which we used to build up the Intel 8080/Z80 instruction set. The results show how error prone working with quantum computers can be, in that a single qubit measurement will add a circa 3% error rate, a rate which is directly proportional to the number of gates in the circuit. It can however be seen that rounding up or down the average measurement results in the correct classical value and thus matches the corresponding truth table. The pseudo/Python code used to create the quantum logic gates in Section V returns an integer-based average of 2000 shots, which provides sufficient accuracy to ensure no logical errors were present in any of the higher-level instructions during testing.

The quantum logic gates presented in Table 5 make up the base building blocks for every instruction and were successfully combined to match the traditional emulation (over 1300 instructions with multiple use-cases each). No logical or averaging errors were present in a complete run through of all instructions in the quantum override class. The results show that both the traditional emulation and emulation overridden for execution on a quantum computer are fully accurate, match the physical CPU, and can pass a full test suite.

VI. LIMITATIONS AND EVALUATION OF THE METHODOLOGY

One drawback of this framework is that due to the probabilistic nature of quantum computers having to perform multiple (1000+) shots [16] for each circuit makes these methods orders of magnitude slower than traditional logic circuits. It is also acknowledged that using quantum computers as the implementation for classical logical gates is highly inefficient for performance, cost/resource and sampling reasons and is unlikely to yield practical improvements. However, keeping the entire circuit in the quantum domain (Listing 8) helps reduce the performance penalty while adding value to the educational aspects of the SDK.

It can be argued that averaging digital logic from a quantum computer is not where the value of quantum computing lies. It is however evident [17] that the quantum device performance and the challenging methods of interacting with quantum computers have had limited real-world results [18] and the useability/programming requires expansion, hence the production of this SDK.

The view that abstracting access to the quantum computer via a translation layer limits the quantum computer to only performing functions that the traditional CPU can already do but in a slower manner is not prescribed or enforced; rather, we are showing developers “how” to perform these actions with a tool set they are familiar with, with methods that can be reused in their own circuits which do take advantage of the quantum functions they may wish to use. Adapting an existing emulator such as YAZE-AG or QEMU was not chosen as most existing emulators are written in C++ for speed of execution, while most quantum computer providers expose APIs.
with documentation in Python. Overriding every instruction and the associated coding of a new emulator was worth the effort to keep the resultant code in one language and simple to follow.

These valid criticisms are not seen as shortfalls or impact on the value of this work as its purpose was not to discuss performance tuning or benchmarking but rather offer a new SDK and framework of techniques which will aid developers in their move to quantum computing.

VIII. CONCLUSION

We have met our goal and the purpose of this article via the implementation of the methods in the SDK. The methods developed in the SDK and their use in the supplied emulator demonstrate to developers how any logic function or traditional assembler code can be executed on a quantum computer.

The use of this SDK and its demonstration via side-by-side comparison of physical, emulation, and emulation executed on a quantum computer (see Fig. 2) successfully provides developers with a set of building blocks to use and adapt to their own use-cases. The SDK eases the steep learning curve involved when moving to quantum computing as developers can reference methods for performing any task they are familiar with. When the developer becomes more accustomed to the methods and complexities, they can progress to develop their own quantum circuits leveraging the benefits of entanglement and superposition, referring to the methods presented in the SDK when traditional logic is required. Developers can also leverage existing programming skills via the translation of assembler into quantum circuits which allows for the testing and comparison of traditional computing to a subset of quantum computing.

Future versions of the SDK will expand the Intel 8080 capability into true x86 compatibility. This is not realistic at present due to the lack of qubits and noise levels, however as the availability of qubits increases the emulator will be expanded.

We could also move from an instruction-by-instruction method of execution to a model where the entire binary is translated into one quantum circuit. This would depend heavily on the length of the binary (decoherence) and type of instructions contained within, but it does open an interesting possibility for vastly increasing execution speed. There is also the possibility to work with other teams such as the OpenQASM [19] project to implement their quantum-based assembler routines within the emulator/translation layer.

Many proven uses of quantum computers have multiple classical components, i.e., Shor’s algorithm contains addition (ADD), multiplication (ADD loop with a JUMP), compare (CP), etc., all of which are implemented and described in the SDK. The current work is not forcing quantum computers to only be used for classical computations but rather showing how those calculations can be performed in a manner developers are familiar with.

For many programmers a complex quantum algorithm could be too daunting to implement and experiment with, so the issue of knowledge transfer is important. The reusable code presented within this SDK allows the developer or researcher to use it straight away, thus removing another blocker to delivering useful results.

Although performance\(^6\) is currently much lower than with a traditional CPU, it is appropriate to develop and enhance the toolset for performing traditional logic-based code on quantum computers as the SDK provides a useful learning tool and a valuable method of implementing aspects of quantum code.

ACKNOWLEDGMENT

The authors would like to thank IBM for the use of their quantum computers under the IBM Quantum Experience program [20], without which this SDK would have been much more difficult to realize.

REFERENCES

[1] Q. O. S. Foundation, “List of open quantum projects,” 2020. [Online]. Available: https://qosf.org/project_list
[2] R. S. Smith, M. J. Curtis, and W. J. Zeng, “A practical quantum instruction set architecture,” 2016, arXiv:1608.03355, doi: 10.48550/arXiv.1608.03355.
[3] J. Shalf, “The future of computing beyond Moore’s law,” Philos. Trans. Roy. Soc. A, vol. 378, no. 2166, 2020, Art. no. 20190061, doi: 10.1098/rsta.2019.0061.
[4] M. A. Sohnel, N. Zia, M. Akef Ali, and N. Zia, “Quantum computing based implementation of full adder,” in Proc. IEEE Int. Conf. Innov. Technol., Nov. 2020, pp. 1–4, doi: 10.1109/ICONIN539.2020.9298394.
[5] V. Vedral et al., “Quantum networks for elementary arithmetic operations,” Amer. Phys. Soc., vol. 54, 1996, Art. no. 147, doi: 10.1103/PhysRevA.54.147.
[6] S. Mazor, “Intel 8080 CPU chip development,” IEEE Ann. Hist. Comput., vol. 29, no. 2, pp. 70–73, Apr.–Jun. 2007, doi: 10.1109/MAHC.2007.25.
[7] V. Barthelmann and F. Wille. Accessed Apr. 2, 2020, [Online]. Available: http://sun.hasenbraten.de/vasm/
[8] T. Wojtowicz, “Visualizing CPU microarchitecture,” Schedae Informaticae, vol. 24, pp. 197–210, 2015, doi: 10.4467/20838476SI.16.017.4358.
[9] N. D. Zilog, Z80 CPU User Manual. [Online]. Available: http://www.zilog.com/docs/z80/um0080.pdf
[10] M. Horodeci, R. Horodecki, A. S. De, and U. Sen, “No-deleting and no-cloning principles as consequences of conservation of quantum information,” 2003, arXiv: quant-ph/0306044, doi: 10.48550/arXiv. quant-ph/0306044.
[11] J. N. D. Shann, “Arithmetic functions and circuits,” [Online]. Available: http://owc.ncu.edu.tw/course/digital%20design/Logic%20DesignCh05.pdf
[12] C. P. Williams, “Quantum gates,” in Explorations in Quantum Computing, London, U.K.: Springer, 2011, pp. 51–122, doi: 10.1007/978-1-84628-887-6_2.
[13] F. Cringle, “Z80 instruction exerciser,” [Online]. Available: https://mddf.s.net/Software/Z80Exerciser/
[14] IBM, “IBM’s roadmap for scaling quantum Technol-ogy,” 2020, [Online]. Available: https://research.ibm.com/blog/ibm-quantum-roadmap
[15] Qiskit, “Installing qiskit,” 2021. [Online]. Available: https://qiskit.org/documentation/stable/0.24/install.html
[16] F. Zickert, “Simulating a quantum computer with qiskit,” 2021. [Online]. Available: https://towardsdatascience.com/simulating-a-quantum-computer-with-qiskit-4eb57378394
[17] Scientific American, “The problem with quantum computers,” [Online]. Available: https://blogs.scientificamerican.com/observations/the-problem-with-quantum-computers

\(^6\)Classical and quantum systems are different models, and the comparison of performance is extremely difficult.
S. Aaronson, “What makes quantum computing so hard to explain?,” 2021. [Online]. Available: https://www.quantamagazine.org/why-is-quantum-computing-so-hard-to-explain-20210608

A. Cross et al., “OpenQASM 3: A broader and deeper quantum assembly language,” *ACM Trans. Quantum Comput.*, vol. 3, pp. 1–20, 2021, doi: 10.1145/3505636.

IBM Quantum Experience. Accessed: Mar. 31, 2022. [Online]. Available: https://quantumcomputing.ibm.com

James Fitzjohn was born in Cambridgeshire, U.K., in 1978. He received the B.Sc. (Hons.) degree (Open) from the Open University, Milton Keynes, U.K., in 2019. He is currently working toward the Ph.D. degree in quantum computing with Anglia Ruskin University, Cambridge, U.K.

He is employed as a Solution Architect with Aldermore Bank, Peterborough, U.K. His research interests include quantum computing, low-level programming languages, and CPU architecture.

Mr. Fitzjohn was the recipient of the Cisco CCNP in 2012, Cisco CCNA in 2011, and HPE Master ASE in StorageWorks in 2011.

Adrian Winckles received the B.Eng. (Hons.) degree in electrical systems engineering from the University of Essex, Colchester, U.K., in 1991, and the M.Sc. degree in network management and PGSC in learning and teaching from the Anglia Ruskin University, Cambridge, U.K.

He is a Senior Lecturer with the Faculty of Science and Engineering, Anglia Ruskin University. He is a Chartered Engineer and Chartered Information Technology Professional of the British Computer Society. He specializes in network and internet security.

Mr. Winckles is a member of many notable editorial boards and professional bodies including MBCS, AMIEEE, a Chartered Engineer, CEng, CSIP, CSIS and has many prestigious publications including eight with the IEEE.

George Wilson received the B.Sc. (Hons.) degree in geology from Aberystwyth University, Aberystwyth, U.K., the M.Sc. degree in computer science from the Anglia Ruskin University, Cambridge, U.K., the M.Sc. degree in geochemistry from the University of Leeds, Leeds, U.K. and the Ph.D. degree from the university of Bath, Bath, U.K.

He is an Associate Professor at the Anglia Ruskin University where he teaches many aspects of applied computer programming. He has extensive research experience in distributed computing, spatial data manipulation, and search semantics.

Dr. Wilson is Chair of the SCIS Research Ethics Panel and Research Coordinator for External Income. He has also served as organizing committee member with track chair responsibility for two international IEEE conferences (ISIE2017, INDIN2015).

Domenico Vicinanza received the M.Sc. degree in physics (laurea cum laude) and the Ph.D. degree in ultrarelativistic heavy-ion physics from the University of Salerno, Salerno, Italy, in 1998 and 2002, respectively.

He is a Senior Lecturer with the Anglia Ruskin University, Cambridge, U.K., where he teaches the modules of microprocessor systems design, and audio technology. He has published more than 130 times, contributing knowledge to many fields including the search for the Higgs boson.

Dr. Vicinanza is a member of the British Science Association and Association of International Project marketing and management boards. He has been the recipient of many industry awards.