Quantum Information Processing (2024) 23:329
https://doi.org/10.1007/s11128-024-04524-5

An optimization of traditional CPU emulation techniques for execution on a quantum computer

James Fitzjohn[1] · George Wilson[1] · Domenico Vicinanza[1] · Adrian Winckles[1]

Received: 6 February 2024 / Accepted: 22 August 2024

Abstract
The use and adoption of quantum computers by the wider computing community is diminished by the need to adopt new programming techniques. These techniques involve moving from a high-level language where the programmer can define and manipulate objects, to a quantum model where the programmer defines and configures the circuits at a gate level. Previous work by the authors aimed to ease this transition through the use of a software development kit (Qx86 SDK) that emulates a traditional CPU for execution on a quantum computer, but only delivered a raw capability. The current work now presents a number of new methods that extends and improves the SDK's capability. These methods include optimizing traditional logic gate emulation, multiple gate simplification methods, reducing the number of required qubits and alternative optimized techniques for many CPU instructions. A quantum machine code mapping method is described that enhances the emulation of a traditional/quantum hybrid CPU prototype. While still orders of magnitude slower than the performance of a traditional CPU in terms of arithmetic, logic and bitwise operations, execution speed is shown to be markedly improved (in some cases by more than 1,000%) and without introducing any unrealistic requirements (that is, all execution can be performed utilizing less than 32 qubits). The usefulness of the SDK has now been enhanced as a reference guide, where the programmer/researcher can contrast traditional methods versus multiple quantum methods of execution.

Keywords Emulation · Translation · Qiskit · Quantum computing · Quantum gates · Macro and assembly languages · Processors

✉ James Fitzjohn
jaf191@pgr.aru.ac.uk

[1] Anglia Ruskin University, Cambridge, UK

Published online: 30 September 2024

1 Introduction

This paper extends the capability of our previous work [1] which described an SDK which includes a fully instruction and cycle-accurate Intel 8080/z80 emulator [2]. The emulator executes all salient parts of execution on a quantum computer using quantum circuits and showed how the capabilities of such quantum computers can be extended with traditional logic functions. That approach focused heavily on emulating the traditional CPU as closely as possible without prioritizing optimization. The real-world realization of this implementation strategy was delivered by mimicking the CPU's functionality by constructing quantum circuit-based logic gates and rebuilding the CPU as it would be constructed "in silicon" [3]. In this paper, we will again override the emulated CPU instructions, but this time the main focus will be on improving efficiency and making better use of the capabilities of a quantum computer. These improvements were delivered via additional selectable quantum implementation methods for each CPU instruction. This allows the user to contrast the initial functional methods with the newly delivered optimized methods. The quantum implementation methods now include additional features such as incorporating the entire functionality of the instruction, including the setting of the flags register in the quantum circuit. This allows for an interesting adaptation of the techniques used in a traditional CPU pipeline, where multiple CPU instructions can be combined into a single quantum circuit, realizing some of the benefits of the traditional CPU pipeline but with quantum circuits. This greatly increases the amount of work which can be achieved in a quantum circuit. The paper also describes a machine code to quantum instruction mapping table, enabling an interesting proposal of a hybrid CPU architecture with quantum calls included in the CPU itself. The motivation of this paper aligns to the goals of the previous paper, in that the SDK it describes and provides 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. A different manner which has now been optimized under continual improvement and now warrants an exploration of the improvements.

1.1 Rationale

The SDK will be greatly improved by offering multiple reusable methods for each instruction. These methods can be contrasted and referenced by the programmer, allowing them to initially use a method they are familiar with and then progress to methods that take better advantage of quantum computers.

2 Overview

2.1 The grouping of instructions

To save silicon die space and consequently cost, CPU designers reuse circuits wherever possible [4]. For example, a decrement circuit reuses a full adder circuit with all the secondary input bits set to 1 and has the effect of rolling over the input to itself minus one. This reusability was originally exploited in the creation of reusable methods for each of the CPU instruction groups. Quantum methods for the instruction grouping listed in Table 1 were also created with a focus on reusability, so that they can be combined to construct a full instruction set for any traditional CPU.

Table 1 Instruction grouping for quantum methods

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.

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

3 Instruction methods

The SDK and fully functioning emulator provide a unique platform to explore alternative approaches for overriding and implementing CPU instructions. This work presents some alternatives which have been designed to more effectively utilize the benefits of quantum computing, reduce execution time and improve the educational and accessibility aspects of the SDK.

To this end, each CPU instruction will be implemented with 2 to 4 user selectable methods. Method 1 will implement the instruction on a quantum computer as it would be executed in silicon. Method 2 presents an optimized approach where the entire instruction is executed in the quantum domain. Versions of Methods 1 and 2 were delivered in release 1 of the SDK. Method 3 takes better advantage of the quantum gates and effects of quantum computing, as does Method 4. These additional and alternative methods demonstrate new knowledge/novelty in their usability/reusability allowing the developer to pick and choose which method best fits their requirements. The improvements described in this work will be delivered in the next version release of the SDK. This approach allows for the comparison and use of methods delivered by the authors and the wide variety of highly optimized circuits delivered by external researchers. It is not the intension of the SDK/emulator to provide an exhaustive inclusion of the quantum circuits delivered by the research community, but provide a reusable framework for using/selecting/contrasting quantum methods.

3.1 Method 1—copying a traditional silicon-based approach with quantum-based logic circuits

The baseline of the SDK is implemented with quantum circuits [5] that are used to represent the traditional logic gates. These quantum logic gates are joined to construct larger circuits such as adders, subtractors and latches. This process mimics how the instructions would be constructed as part of a CPU in silicon. For example, the quantum circuits in Figs. 1 and 2 can be combined to construct an increment circuit (Fig. 3) which in turn can be used to emulate the INC instruction.

Although a useful guide and interesting proof of concept for performing bitwise, logic and arithmetic functions on a quantum computer, this technique does not take advantage of the underlying benefits of quantum computers such as being able to handle multiple states in one qubit.

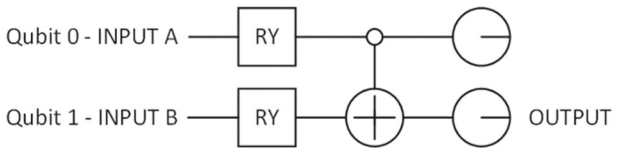

Fig. 1 A quantum XOR gate implementation using two Ry gates and a C-Not

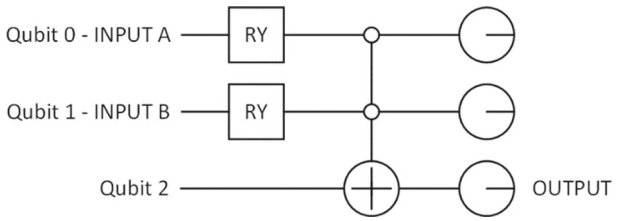

Fig. 2 A quantum AND gate implementation using two Ry gates and a Toffoli gate

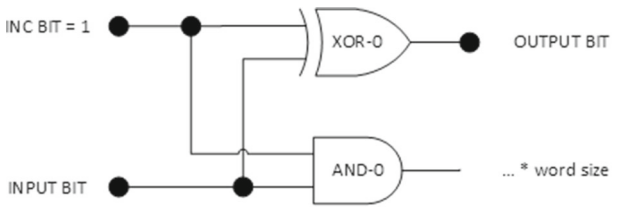

Fig. 3 A traditional increment circuit using AND and XOR gates

3.2 Method 2—executing the circuit entirely in the quantum domain

Previous work that presented the Qx86 SDK showed that the baseline approach is logically accurate but extremely slow [1]. This shortfall is mostly due to the API-based method of accessing IBMs' quantum computers [6] as each circuit/logic gate must join a queue for execution. This is compounded by some larger circuits such as a 16-bit subtractor requiring over 100 logic gates, and passing information back and forth for each of these gates has no practical benefit (other than for process clarification). To mitigate this, shortfall quantum circuits can be created that combine the required logic gates into a single circuit. For the practical implementation of larger 8- or 16-bit instructions, multiple 4-bit variants were created where noise or the number of qubits available would not allow for a full implementation. The scalability of larger instructions was assessed on a case by case basis and was primarily dependent on the number of gates/qubits required to implement the instruction. For example, a fictious a 16-bit XOR circuit requires less qubits and gates than a 4-bit subtractor. An example of this approach is shown in Fig. 8 where the XOR and AND gates are combined to implement an 8-bit increment instruction. This method has vastly improved execution time as the majority of instructions (excluding repetitive instructions such as CPIR, LDIR, etc.) can be completed utilizing one or two circuits. This approach requires more qubits but care was taken to ensure that no unreasonable overheads existed. For this reason, the qubit requirements were kept below 32 for all circuits to ensure they could be performed on a Falcon r5.11H class quantum computer [7]. The accuracy of each of the 1300 + CPU instructions was assessed and only deemed satisfactory if the circuit was executed accurately over 2000 shots.

3.3 Methods 3 & 4—new and experimental methods for implementing instructions

This section presents a number of examples that describe the methods and quantum circuits required to create the new alternative (improved) instruction methods.

1. Utilizing the probabilistic nature of qubits. This method is suited to mathematics functions such as the ADD, SUB, INC and DEC instructions. The input numbers of arithmetic based instructions can be encoded into the probability of two Quantum Ry gate rotations. Both Ry gates are then applied to one qubit (Fig. 4). Each input number is scaled to a fraction of an arbitrary word size (in this example using a 4-bit 0–15 word size) and this fraction is then scaled from 0–π which represents the rotation angle of the Ry gate. Both Ry rotations are performed on one qubit and the resulting qubit probability is descaled to equal the sum of the original inputs.

Figure 5 shows how the input number 9 scales to a predictable qubit probability when run through a Ry gate.

As an example consider adding two input numbers, 4 and 5. These input numbers can be encoded with the Ry rotations of 0.84 and 1.05 that will result in the qubit probability of 34.78%. The resultant probability can be descaled back into the sum by referencing the chart in Fig. 5 or running the code in Listing 1. It was important to

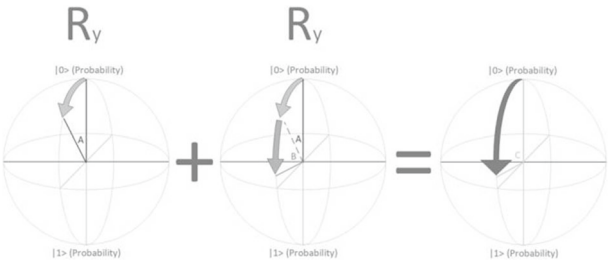

Fig. 4 Adding the numbers, A and B by scaling them into Ry rotations and decoding the sum form the resultant probability

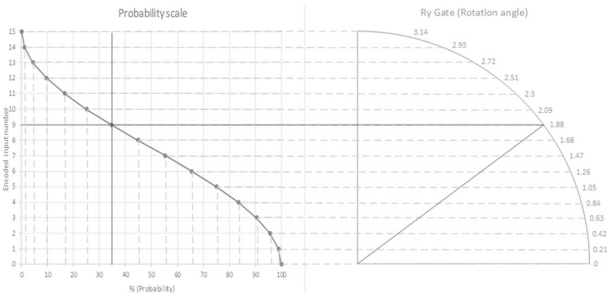

Fig. 5 A cross reference of the mapping between input number, the required Ry rotation and output probability

fine tune the word size versus noise to allow enough margin of error so that all results were correct. In this example, a 4-bit word size was found to be the largest number that could be encoded into a Ry rotation before errors were detected.

```
def numberToRyRotation(self,x):
#A helper function which scales a number to an Ry
rotation
    output = x * (math.pi /15)
    return round(output,2)

def probabilityToNumber(x):
#A helper function which converts qubit probability
to output (SUM) number
    output = math.acos((x-50)/100 * 2)
    output = round((output/math.pi) * 15,0)
    return output


inputA = 4 #The first number to add
inputB = 5 #The second number to add

#We now convert the terms inputA and inputB to an
Ry rotation
aRy = self.numberToRyRotation(inputA)
bRy = self.numberToRyRotation(inputB)

circuit.ry(aRy,0)
circuit.ry(bRy,0)

…
#Execute a quantum circuit with an Ry gate rotated
to aRy plus bRy and store the result as a float
based average of the qubit values in a variable
named probability.
…


total = probabilityToNumber(probability)

print( str(total) ) #The result = 9 i.e. 4 + 5
```

Listing 1 A pseudo code demonstration of converting input numbers to Ry rotation and decoding the resultant qubit probability from the total

This approach has the advantage of reducing the required number of qubits to 1, and the size of the numbers being added (4-bit, 8-bit, etc.) is limited only by the noise inherent in the system. However, it requires the orchestrating computer to perform more tasks (such as the encoding of the input rotation, decoding of the sum from the probability and handling of overflows).

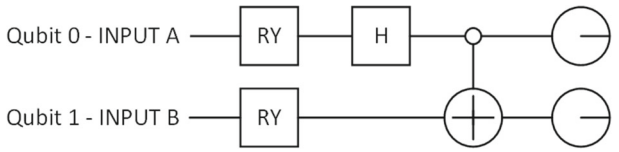

Fig. 6 A quantum circuit which entangles qubits 0 and 1 via the a Hadamard and a controlled not gates

2. Using entanglement via a Hadamard gate as a latch circuit for BIT, SET, RES instructions and their combination into LOAD instructions.

As described in our previous paper, qubit entanglement using a Hadamard (Fig. 6) gate [1] offers an interesting parallel with a latch circuit.

The SET, RES and by proxy all load instructions utilize latch circuits. If the logic table of a latch circuit (Table 2) is compared against a simple Hadamard entanglement circuit (Table 3), the similarities can be seen and exploited.

Using Fig. 6 and Table 3 as a guide, the "Set" qubit rotation can be set to 0 and the "Reset" qubit rotation to π and will measure a 50% probability of the qubits collapsing to 0 and 1 and a 50% probability of the qubits collapsing to 1 and 0. Conversely if the "Set" qubit rotation is set to π and the "Reset" qubit rotation to 0 then 00 and 11 will each be measured 50% of the time. Previously, the method achieved the same results as a latch circuit by performing a logical XOR (Listing 2) on the measured qubits.

Table 2 Latch circuit truth table

Set–Reset	Q–$\underline{Q}$
0–0	N/A
0–1	0–1
1–0	1–0
1–1	N/A

The only valid inputs for a latch circuit that stores logical 0 or 1 are the combinations Set 0, Reset 1 and Set 1, Reset 0

Table 3 A 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–$\boldsymbol{\pi}$	(50%) 01 and (50%) 10
$\boldsymbol{\pi}$–0	(50%) 00 and (50%) 11
$\boldsymbol{\pi}$–$\boldsymbol{\pi}$	(50%) 01 and (50%) 10

A measurement of the entanglement quantum circuit is detailed in Fig. 6

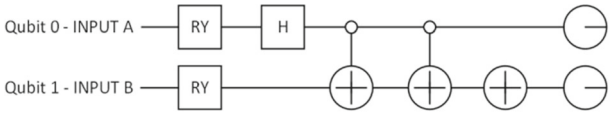

Fig. 7 An extended entanglement circuit with additional XOR functionality

```
notQ = xor(measuredBit0,measuredBit1)
q = not(notQ)
```

Listing 2 Retrieving latch equivalent results from a Hadamard circuit

The instruction method can be improved by keeping the entire function in the quantum domain via the inclusion of the XOR calculation in the quantum circuit. This can be achieved with the extended quantum circuit shown in Fig. 7. These improvements can again be used in all BIT/SET/RES instructions and combined with multiple copies to form an 8-bit or 16-bit LOAD instruction.

This approach further simplifies the quantum circuit required to implement a latch and all subsequent LOAD instructions. The Hadamard-based latch circuit also reduces the number of qubits needed (compared to mimicking the "in silicon" approach) to perform an 8-bit LOAD from 28 to 16 and the number of gates from 122 to 32.

This approach is not without its drawbacks, as consideration must be made to the physical layout of the underlying qubits and hence, it is not sufficient to rely on the circuit/intermediate representations only. Intermediate representations abstract the physical implementation and can utilize additional nonuser visible gates at build/optimization time. IBM incorporates either a heavy hex or square lattice approach to qubit layout. This results in many qubits not being directly connected and thus swap gates are required to pass quantum information across noncontiguous qubits in the chain. This is generally not an issue on smaller circuits where the qubit location can be easily optimized. However, circuits which utilize more qubits inherently compound the noise problem due to the difficulty in arranging multiple entangled qubits. The impact of these challenges are limited with this method, as each bit of the load uses adjacent/directly connected qubit pairs for the source and destination, therefore the drawbacks will not be realized. Larger more complex instructions such as the SUB(tract) instruction can take advantage of recent advancements in the qiskit SDK and IBMs quantum offering, which allow for in circuit measurement and thus "in circuit" quantum teleportation. This offers another useful tool for reducing errors across larger circuits where the use of contiguous qubits is not achievable. However, this was not required for the realization of the emulator in this SDK, as noise levels where never high enough to impact the results of any instruction while using 2000 shots.

3. An implementation of the LOAD/EXX instruction using quantum swap gates.

   In contrast to the original LOAD implementation that uses 8/16 latch circuits in parallel, quantum swap gates can be used to move the bits from the source to the destination. Care must be taken only to use this technique where the LOAD

instruction has a transitory source value, otherwise there is a risk of overriding the source with the destination data. However, the EXX (exchange) instruction has no such constraints and is well suited to the use of quantum swap gates.

4. An implementation of the ROTATE instruction using quantum swap gates.

   The rotate instructions RL, RLC, SLA, SLC, RR, RRC all perform variations of rotating the bits in a register left or right in combination with the C flag. These instructions make excellent candidates for the use of quantum swap gates as bits can be swapped from one qubit to the next. This approach is much more efficient than the silicon method of multiple in-line latch circuits.

5. An implementation of the JP/JR instructions using quantum swap gates.

   This example is similar to example 3 in that instead of loading a destination with a source, the program counter is now loaded with a 16-bit number. The JP/JR routines can be more complex as they often depend on the status of a flag, e.g., JP C jumps to a location if the C flag is set. This is not problematic as the checking of the C flag can be prepended with a quantum c-not gate, the subsequent steps then follow the same process as in example 3, again being careful only to use swap gates if the source is transitory.

6. The CP instruction.

   The "in silicon" approach for performing the CP (compare) instruction on the registers A and B involves subtracting B from A and checking if the result equals 0. This approach is sensible where silicon die space is restricted as the SUB instruction can be reused. However, this is an inefficient slow process for the quantum computer as the subtractor circuit requires multiple XOR, NOT and AND gates to be performed on each bit in turn. The process can be optimized by performing the quantum implementation of a XOR circuit (Fig. 2) on each bit in parallel and then checking if the result equals 0.

4 Improving the emulation

This section focuses on the logistics of improving the CPU emulation. The F register is a register that stores status "Flags" and can be considered a hot spot for read/write activities, as many instructions set the value of the bits in the F register depending on the result of their execution [2]. It is therefore prudent to optimize the writing to and reading from the F register as it is accessed so often.

We can optimize the setting and reading from the F register (which contains the flags: signed, zero, half-carry, parity, overflow, negative and carry) utilizing a single quantum circuit with the approach detailed in the following example.

Consider the quantum implementation of an increment circuit shown in Fig. 8. This circuit implements the salient points of execution, i.e., the increment computation. In order to implement all points of execution the quantum circuit must be extended to include the relevant flags in the F register, being mindful not to alter the output qubits as this may impact the result. The first 5 of 8 flags are relatively simple to compute as

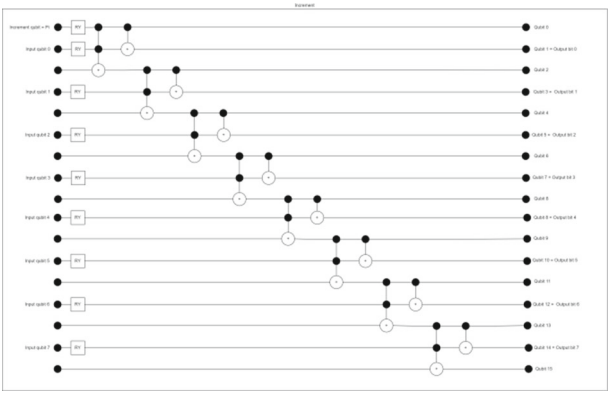

Fig. 8 A quantum increment circuit

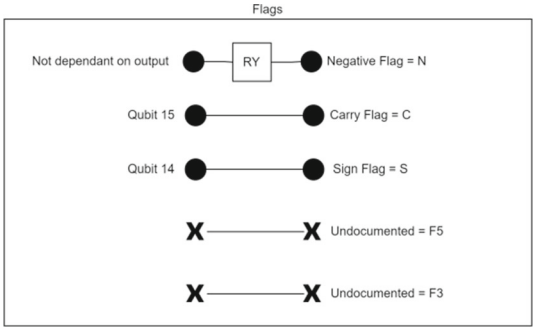

Fig. 9 Reading the F register flags, N, C, S, F3 and F5 from the increment circuit

the N flag is set programmatically, while C and S are directly available in the circuit and the F3/F5 flags are not documented (Fig. 9).

The remaining flags, Z (zero), H (half-carry) and P/V (parity/overflow) will all use variants of the quantum circuit listed in Fig. 10. The circuit in Fig. 10 configures a C-not on each output qubit which is fed into a NOT gate and checked via three Toffoli gates. The Toffoli gates then switch the desired output flag. The Z (zero) flag reuses the C-not chain across all output qubits to check if the result = 0. The quantum circuit required for the H (half-carry) flag checks if the four least significant bits are 0. The lower four bits being 0 indicates that the last increment instruction carried over and a half-carry occurred. The overflow bit repeats the circuit over the first seven least significant bits. The P (parity) flag is not required for the increment instruction but is shown here for reference. The parity flag is computed with a chain of controlled not gates that mimics the traditional XOR gate method of computing parity. Note this approach is reusable but will require modification as each instruction grouping sets the F register in slightly different ways. For example, the half-carry flag method shown here works for an increment instruction as we know if the four least significant bits are 0 (so reflecting the half-carry). Conversely, the four least significant bits in an ADD instruction may be less than the four least significant bits started with, therefore

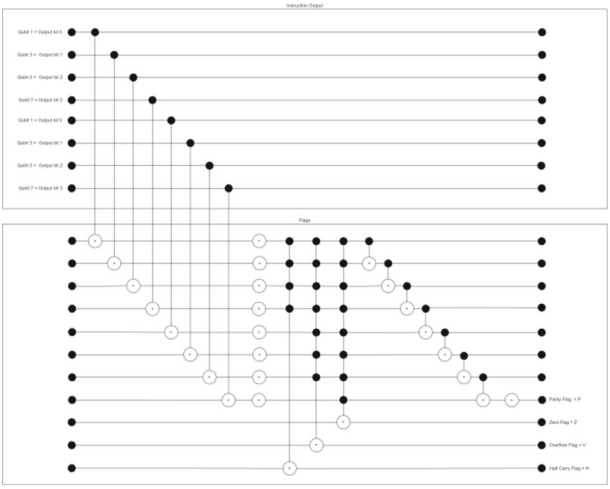

Fig. 10 Calculating the parity, zero, overflow and half-carry flags from the increment circuit

the half-carry bit calculation must add the four least significant bits of both inputs and check if the result is greater than 15. The computation of the traditional CPUs parity bit offers an interesting and reusable technique for managing fault tolerance. The in (quantum) circuit implementation of the parity bit enables the detection and subsequent correction of errors in circuit, at the cost of using more qubits. This is not to suggest it will replace more advanced forms of fault tolerance but rather a fortunate ability to reuse what is already required to achieve accurate emulation.

The next optimization we describe takes advantage of the programmatic nature of creating quantum circuits. In silicon (excluding FPGA devices) once the circuit design is finalized and etched no changes can be made, but with quantum circuits the circuits can be altered as required. It is therefore beneficial to perform as many instructions as possible in one circuit. However, various constraints to the number and type of instructions that can be incorporated into a single circuit have been identified, including:

4.1 The use of internal registers only

Instructions can only be grouped into 1 circuit if they do not access external RAM or devices. For example consider the instructions in Listing 3.

```
ADD A,B
INC A
```

Listing 3 Instructions that can be combined in one quantum circuit as all computations are performed on internal registers

The instructions in Listing 3 could be grouped into one circuit as all operations are executed within the CPU.

```
LD BC,$ffff
LD (ffff),$aa
LD A,(BC)
```

Listing 4 Instructions that should not be combined in a quantum circuit

Conversely the instructions of Listing 4 would be unsafe to group into one circuit as exclusive control of the RAM cannot be guaranteed, i.e., the real CPU could have been halted during a DMA while another device makes changes to the memory address referenced.

4.2 A move to complete emulation

To group instructions, the move from executing the salient points of execution to a full implementation, including the emulation of the F register must be made. Consider the program shown in Listing 5, which loads the A register with 5h, adds FFh, then adds 1h with the carry bit.

```
LD a,$05
ADD a,$FF
ADC a,$01
```

Listing 5 Loading, adding and adding with carry

If the instructions in listing 5 were grouped the result would be incorrect. This is because execution could not be passed back to the traditional computer to perform ancillary tasks, such as the setting of the carry flags. This would result in the A register being incorrect as the carry flag would be missed in the addition in the ADC instruction. Therefore, without a move to complete emulation (detailed in Sect. 4), the type of instructions that can be grouped in one circuit would be restricted to those which do not affect the F register.

4.3 The number of instructions being combined

The constructed quantum circuit must not be so large that decoherence or other errors alter the results. A simple experiment was conducted on a Falcon r5.11H quantum computer where an XOR equivalent quantum instruction was repeated (Fig. 11) until the result from the XOR was no longer accurate.

The results in Table 4 show a clear degradation in accuracy as more instructions are incorporated in the circuit. The resultant probability varies from the perfect result of 1.0 to a background noise level at 0.5. The results indicate that a maximum of

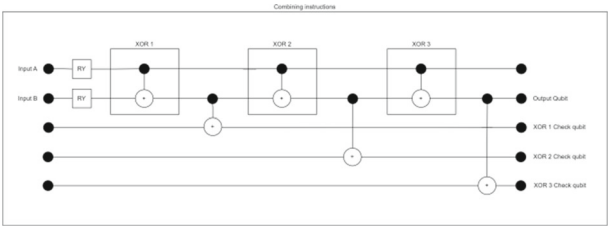

Fig. 11 A quantum circuit executing multiple XOR instructions in an effort to measure the rate of error added by each instruction

Table 4 Qubit accuracy of executing n instructions against number of Shots

Instruction/shots:	1000	2000	3000
One instruction	0.905✓	0.882✓	0.894✓
Two instructions	0.702✓	0.682✓	0.693✓
Three instructions	0.528✗	0.487✗	0.516✗
Four instructions	0.517✗	0.558✗	0.555✗
Five instructions	0.486✗	0.464✗	0.504✗

Results are averaged from the quantum circuit in Fig. 11. The ✓ and ✗ are used to highlight where the result of the instruction can be rounded (1/3 weighted) to the correct value

two instructions per circuit can be reasonably performed. This halves the number of queues required, doubles execution speed, reduces the number of shots and the amount of passing of information/execution back and forth between the traditional and quantum computer. It could be argued that using a C-not to "copy" the result of the XOR instruction to another qubit introduces additional error, and so a swap gate should have been used. This approach was considered but not chosen as it would result in swapping in a clean qubit for each instruction which would skew the outcome in favor of cleaner results.

5 Creating a machine code mapping to call quantum opcodes

Cross et al. [8] propose assembler-based routines for calling quantum computers. The work presented here provides a basis on which such a prototype can be implemented. With the translation layer in place OpenQasm [8] calls can be directly implemented in the emulator/translation layer, making the experience much closer to that of a physical CPU but with quantum capabilities. This can be achieved by implementing the "Machine Target Code Generation" component (Fig. 12) of OpenQasm [8]. This can be undertaken by modifying the VASM [9] assembler to map OpenQasm instructions to machine code, and the machine code will then be interpreted by the emulator/translation layer into QisKit [6] calls to IBMs quantum computer.

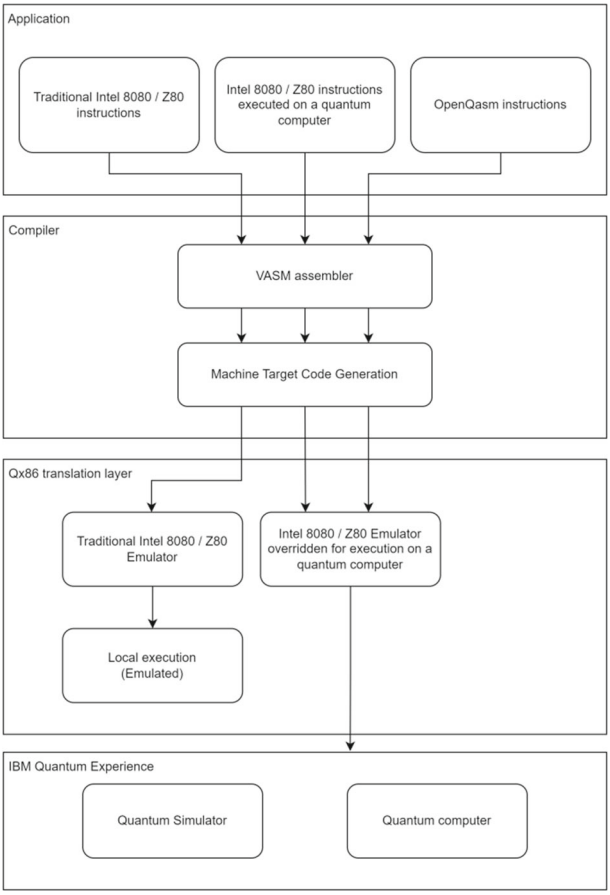

Fig. 12 An implementation of Cross, A Et [8] all's OpenQasm

5.1 Machine code mapping method

Many CPUs contain special opcodes [10] which act as a modifier for the next opcode, for example, the machine code "2C" increments the l register, however prepending the "2C" with "DD" alters the instruction to increment the ixl register instead. This was originally implemented to allow more unique instructions to fit into a smaller data bus width, but this design feature can be advantageously utilized by selecting an unused machine code value to create an opcode modifier. This allows a mapping of machine code to OpenQasm instructions. To make the development of OpenQasm-based quantum circuits easier the source code for VASM can be edited with new mnemonics added to the quantum circuit gates (Listing 6).

Table 5 Assembler to OpenQasm machine code mapping table (abridged)

Machine code	0	1	2	3
0	Ry, a	Rx, a	Rz, a	H, x
1	Ry, b	Rx, b	Rz, b	H, x
2	Ry, c	Rx, c	Rz, c	H, x
3	Ry, d	Rx, d	Rz, d	H, x
4	Ry, e	Rx, e	Rz, e	H, x

```
"RY", { OP_REG_A }, { TYPE_QUANTUM,
0xEDFF01, CPU_ALL, 0 }, /* RY, A */
```

Listing 6 Pseudo code to add a quantum assembler instruction to the VASM assembler. The proposed machine code, 0xEDFF01 maps to an RY gate rotated by the value in the A register

Listing 6 will allow for the assembler instruction RY,A to be translated into the machine code "ED,FF,01." This process enables a complete assembler to machine code mapping table (Table 5) to be constructed for the opcode modifiers "ED,FF.**."

With the new mnemonics in place the assembler code for a Hadamard gate on Qubit x (H,x) can be built into the machine code "ED,FF,03,01." For this case, the final two bytes of the machine code are an operand which represents the qubit to apply the Hadamard gate to. The emulator can now be edited to catch all machine code values which start with "ED,FF" and interpret them to the corresponding calls to IBMs quantum computer.

6 Results

The largest gains in performance were achieved in the move from executing individual logic gates to executing the entire circuit in the quantum domain. When dealing with the variable execution time of running quantum circuits, i.e., the queuing mechanism, it was important to benchmark any potential improvements against a fair playing field and not a favorable queue length. With this in mind, an adaptation of algorithmic complexity was used to assess performance increases. Given that the input size n, is fixed at 8-bit or 16-bit, there is no need to assess infinities and all resultant complexity will be constant (ignoring recursive instructions which will have a linear complexity). This does not reduce the value of assessing the capability but rather simplifies it, in that it is possible to substitute high-level instructions for quantum gates, or more practically quantum circuits. With this tactic in place each of the quantum implementation methods were compared with the following results. Note that Fig. 13 shows and abridged dataset as the goal of quantum implementation methods 3 and 4 was to include more functionality or explore the benefits of the quantum computer, not necessarily reduce the number of circuits.

The upper field shows the number of quantum circuits required for the original gate by gate quantum implementation method, whereas the lower field shows the number

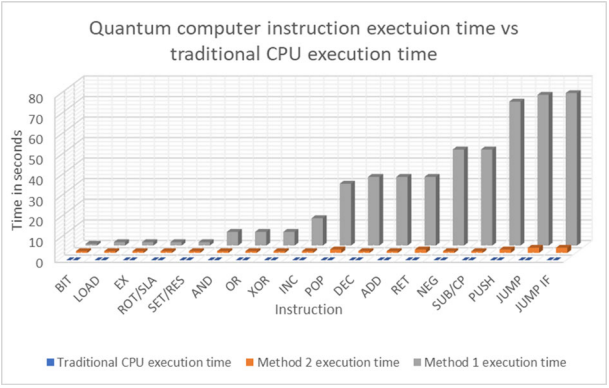

Fig. 13 An overview of the improvements in efficiency made by the new instruction implementations

of quantum circuits required for the optimized (quantum) method, i.e., method 2, 3 or 4. The optimized method shows a clear reduction in the number of quantum circuits required to perform an instruction and hence execution time is vastly reduced.

7 Limitations and evaluation of the methodology

Although orders of magnitude slower than the performance of traditional CPUs in arithmetic, logic and bitwise operations, this new work substantially improved execution speed of an emulated CPU on a quantum computer (in some cases by more than 1,000 percent) without introducing any unrealistic requirements, i.e., all execution can be performed utilizing less than 32 qubits. This is a marked improvement in efficiency compared to the "in silicon" method described in the author's previous paper [1]. Overall performance is still heavily bound by the number of qubits available, the delays involved in the remote API-based approach and the queuing method of accessing IBM's quantum computers. The main improvements are in usability, where each instruction can be executed in circa 30s and the breadth of methods available to the researcher to contrast.

Evaluating the scale of progress in the development of quantum computers versus traditional computers regarding logic operations is difficult because of their different architectures and goals. However, some interesting statistics can be used to compare the linear growth of traditional CPU performance [11] and the exponential growth in quantum computing performance [12]. One approach which provides a generalized appreciation of the trend (and arguably highlights the timeliness of the work presented in this paper) is to predict the number of transistors used in a CPU compared to the number of qubits available in a quantum computer. On the one hand using the last 34 years [11] of Intel CPUs as a baseline to forecast the number of transistors used (normalized by dividing by 6 as it takes 6 transistors to store 1 bit in a CPU [13]), and on the other hand using the limited data set of 3 years of quantum computer development and IBM's roadmap [12], a simplified performance growth prediction of number of

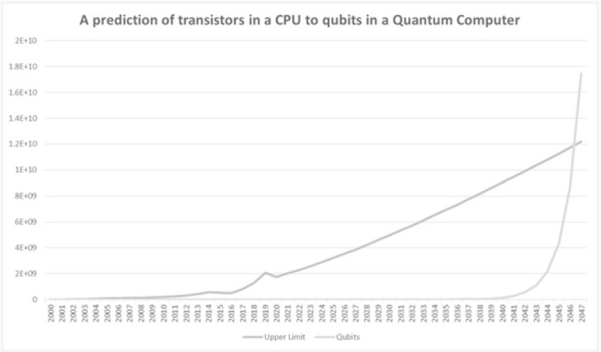

Fig. 14 A prediction of the number of available qubits (exponential) versus the number of transistors in a CPU

transistors versus qubits can be made. This suggests that the number of qubits will overtake the number of normalized transistors in the next 25 years, around the middle of the twenty-first century (Fig. 14). This raises the possibility that the work presented here may have greater practical relevance to the wider industry in the near to medium future [15].

8 Conclusion

Alternative methods of performing traditional CPU instructions on a quantum computer have been demonstrated. These new methods will be included in the next version of the authors' Qx86 SDK. This work enables the programmer and/or researcher to select a method they are familiar with and contrast a traditional method with a more efficient quantum method(s) of execution. The multiple method approach improves the accessibility/usability of quantum computers when transitioning from traditional computing [16].

Future work will include adding full support for a full × 86 instruction set [17]. This is not currently practical due to the requirement of handling multiple concatenated 32-bit registers in one instruction, but will become possible as more advanced quantum computers are developed and released (e.g., the IBM Osprey class of quantum computers [7]).

Acknowledgements The authors would like to thank IBM for the use of their quantum computers under the IBM Quantum Experience program [18], without which the authors' SDK would have been much more difficult to realize.

Author contributions JF drafted the manuscript. All authors reviewed and contributed to the manuscript.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Fitzjohn, J., Winckles, A., Wilson, G., Vicinanza, D.: A Software Development Kit and translation layer for executing Intel 8080 assembler on a Quantum Computer, August 2022. IEEE Trans. Quantum Eng. **3**, 1–12 (2022)
2. Zilog, N.D.: Z80 CPU User Manual [Online]. Available: http://www.zilog.com/docs/z80/um0080.pdf
3. Mazor, S.: Intel 8080 CPU Chip Development. IEEE Ann. Hist. Comput. **29**(2), 70–73 (2007). https://doi.org/10.1109/MAHC.2007.25.A
4. Mastorakis, N.: Priority encoding based 8-bit incrementer/decrementer module [Online]. https://www.researchgate.net/figure/Priority-encoding-based-8-bit-incrementer-decrementer-module-3-4_fig1_265684748
5. Williams, C.P.: Quantum gates. Explor. Quantum Comput. 52–94 (2011). https://doi.org/10.1007/978-1-84628-887-6_2
6. QisKit: Installing Qiskit [Online] (2021). Available: https://qiskit.org/documentation/stable/0.24/install.html
7. IBM: IBM Quantum Processor types [Online] (ND). https://quantum-computing.ibm.com/composer/docs/iqx/manage/systems/processors
8. IBM: The IBM quantum heavy hex lattice [Online] (2021). Available: https://www.ibm.com/quantum/blog/heavy-hex-lattice
9. Cross, A., Javadi-Abhari, A., Alexander, T., de Beaudrap, N., Bishop, L.S., Heidel, S., Ryan, C.A., Sivarajah, P., Smolin, J., Gambetta, J.M., Johnson, B.R.: OpenQASM 3: A broader and deeper quantum assembly language. ACM Trans. Quantum Comput. **3**, 1–50 (2021)
10. Barthelmann, V., Wille, F.: Accessed April 2, 2020 [Online]. Available: http://sun.hasenbraten.de/vasm/
11. Clr Home.: Z80 Opcode Table [Online] (ND). https://clrhome.org/table/
12. Sun, Y., Agostini, N.B., Dong, S. and Kaeli, D.: Summarizing CPU and GPU design trends with product data (2019). Preprint at arXiv arXiv:1911.11313
13. IBM: IBM's Roadmap For Scaling Quantum Technol-ogy [Online] (2020). Available: https://research.ibm.com/blog/ibm-quantum-roadmap
14. Imperial College London.: Lecture 12—memory and computer architecture [Online] (2018). http://www.ee.ic.ac.uk/pcheung/teaching/de1_ee/Lectures/Lecture%2012%20-%20Memory%20and%20Computer%20Architecture%20(x2).pdf
15. Shalf, J.: The future of computing beyond Moore's Law. Phil. Trans. R. Soc. A **378**(2166), 20190061 (2020)
16. Aaronson, SF.: What makes quantum computing so hard to explain? [Online] (2021). Available: https://www.quantamagazine.org/why-is-quantum-computing-so-hard-to-explain-20210608
17. Degenbaev, U.: Formal specification of the x86 instruction set architecture (2012)
18. IBM Quantum Experience.: Accessed 31 Mar 2022 [Online]. Available: https://quantumcomputing.ibm.com

James Fitzjohn was born in Cambridgeshire, United Kingdom in 1978. He received the B.Sc. (Hons) degree (Open) from the Open University, Milton Keynes, United Kingdom, in 2019. His research interests include quantum computing, Low level programming languages and CPU architecture. He is currently working toward the Ph.D. degree in quantum computing with the Anglia Ruskin University. He is employed as a SOLUTION ARCHITECT for a large Bank in the United Kingdom. Mr. Fitzjohn has been published in IEEE transactions on Quantum Engineering and has been awarded a Cisco CCNP in 2012, Cisco CCNA in 2011 and HPE Master ASE in StorageWorks in 2011.

George Wilson Received the Ph.D. degree in Isotope Geochemistry at the University of Bath, UK in 1986. He also has an MSc in Geochemistry and an MSc in Computer Science). He is an ASSOCIATE PROFESSOR at the Anglia Ruskin University (ARU) where he teaches many aspects of computer science. He has extensive research experience in distributed computing, spatial data manipulation and search semantics. Dr. Wilson is Chair of the ARU computing Research Ethics Panel, and a 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 Ph.D. degree in Ultrarelativistic Heavy-ion Physics at the University of Salerno, Italy in 2002. A MSc degree in Physics (Laurea cum Laude) 100% with distinction at the University of Salerno, Italy in 1998. He is a SENIOR LECTURER at the Anglia Ruskin University where he teaches the modules of Microprocessor Systems Design and Audio Technology. Dr. Vicinanza is a member of the British Science Association and Association of International Project marketing and management boards. He has received many industry awards and has been published over 130 times, contributing knowledge to many fields including the search for the Higgs boson.

Adrian Winckles received the MSc degree in network Management in 2000 and a PGSC in learning and teaching from the Anglia Ruskin University, Cambridge, United Kingdom in 2006. He also received the BEng (hons) degree in Electrical Systems Engineering from the University of Essex in 1991 He is a SENIOR LECTURER at the Anglia Ruskin University in the Faculty of Science and Engineering. 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, AMIEE, a Chartered Engineer, CSTP, CSIS and has many prestigious publications including 8 with the IEEE.