Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Designing a Quantum Algorithm for Real-Valued Addition Using Posit Arithmetic

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

Tim Driebergen
4565320

Delft, The Netherlands
August 2019

BSc thesis APPLIED MATHEMATICS

"Designing a Quantum Algorithm for Real-Valued Addition Using Posit Arithmetic"

TIM DRIEBERGEN

Delft University of Technology

Supervisor

Dr. M. Möller

Other committee members

Prof.dr.ir. C. Vuik

Dr. J. G. Spandaw

August, 2019

Delft

Abstract

Currently there are no efficient quantum algorithms for the addition of real-valued numbers. In classical computers addition is performed by using barrel shifters, a concept proven to be very inefficient as a quantum circuit due to its many garbage outputs when the barrel shifter is made reversible. This thesis aims to design a quantum algorithm able to perform floating-point arithmetic. It uses the new Posit format as its number format so the algorithm can be built on a very small scale, which makes it possible to easily implement the entire algorithm. The designed Quantum Posit Adding Algorithm uses a tablebase approach, examining each number checking if it changes during addition. An optimized version of the algorithm is also designed, removing any unnecessary controls. Finally a method to extend the algorithm is proposed along with an approach to building a similar subtractor, also presenting some non-working ideas.

Preface

This thesis was written as the final part of the Bachelor of Science in Applied Mathematics at Delft University of Technology (TU Delft). I have been engaged in researching the topics of both Posit and quantum computing since the start of 2018, originally conducting the research as part of the Honours Programme Delft. The origin of my interest in these fields of mathematics comes from a lecture Linear Algebra 2 by Matthias Möller, who covered an introduction to quantum mechanics with bra-ket notation. Following his lecture I asked him if he had a project for me on the topic, when he introduced me to the Posit format. After the Honours programme research continued under the guidance of Matthias, who eventually became my supervisor.

I would like to thank Matthias for his excellent support and guidance during this process. Even when he very rarely did not know the answers to my queries, he always provided a new insight and made time to discuss any issues. Special thanks also go to Theodore Omtzigt, who taught me much about the workings of Posit as well as an efficient implementation. Our hour-long discussion about rounding behaviour and barrel shifters followed by his presentation greatly helped in advancing the project. Lastly I would like to thank my parents, whose wise words and encouragement kept me motivated.

I hope you enjoy your reading.

Tim Driebergen
Delft, August 2019

Contents

1 Introduction

Addition is the most simple operation of arithmetic. Together with subtraction, multiplication and division it forms the four basic operations and the concept of counting is believed to be around 20,000 years old. [1]. Nowadays addition is fundamental to how digital computers operate, and therefore its efficiency is critical for the performance of these computers. Many different adders have been designed, the most well known schemes being the Ripple Carry Adder and the Carry Look-ahead Adder [2], but no absolute best design is possible due to the many differences between circuits in power, area and delay. The best adder depends on the application the circuit is used in; higher performance applications demand more area, while low-power applications use less area and power at the cost of more delay [3]. Important to recognize is that adder design is critical, especially now that the world is busy designing a new kind of computer; the quantum computer.

Quantum computers can perform certain calculations significantly faster than classical computers. Quantum computers use quantum-mechanical superposition and entanglement to parallelize arithmetic and solve problems more efficiently. One of the most famous quantum algorithms is Shor's algorithm, which given an integer N finds its prime factors [4]. Addition is and will be a part of many quantum algorithms like Shor's, but the current adding algorithms are not practical on a quantum computer. Presently numbers are saved on a computer via the IEEE 754 Floating-Point format [5]. However, using this format on a quantum computer is not possible at the moment, as current quantum computers are relatively much smaller than classical computers. There are simply not enough qubits (quantum bits) available to run the same algorithm on a quantum computer that is run on a classical computer.

For that reason in this thesis a different, more recently invented number format called Posit will be used to design a new adding algorithm. An algorithm that can handle larger numbers like classical computers, but also works in the quantum computing environment. First the relevant quantum computing knowledge necessary to understand the adder will be discussed, after which an introduction is given into the Posit format. Lastly the adder itself is presented along with optimizations and further extensions.

2 Relevant Basics of Quantum Computing

Classical computers operate with bits. Bits are the units of information in information theory, and in the case of classical computing only have values of 0 and 1. Logic gates are devices that have one or more bits as input and produce a single bit as a binary output value. Multiple logic gates are used to construct entire logic circuits, the basic building blocks of any digital system. An example of a logic gate would be the AND gate, which produces output value 1 if both of its input values are 1. Most of these concepts are easily translated into the field of quantum computing, except that quantum computing has some restrictions in its working due to the nature of quantum physics.

Quantum computers work with quantum bits or 'qubits'. Qubits can represent 0 or 1, but also a superposition of those two states. Essentially the qubits can attain a state in between 0 or 1, but when observed collapse to only 0 or 1. The best representation of a qubit is the Bloch Sphere, seen in Figure 1.

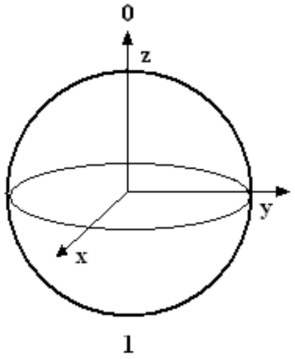


Figure 1: The Bloch Sphere [6]

Two classical bits can be in four different states (00, 01, 10 or 11), but can only be in exactly one of these states at the same time. However, two qubits can be in a superposition of these four states, so together the two qubits are in all four states at the same time. In general a qubitstring of n qubits can be in one superposition of 2^n states simultaneously. This is why quantum computers can be faster than classical computers; a classical computer would have to do 2^n runs to calculate the result of each state, while the quantum computer can perform the calculation on a superposition of all the states, effectively calculating the results of all the states at the same time.

Logic gates are now quantum logic gates (or just quantum gates) and operate on a small number of qubits. Unlike most classical gates, quantum gates need to be reversible and also unitary. Any physical implementation of a quantum computer has to follow the no-go theorems of theoretical physics, the two most important ones being the no-cloning theorem and the no-deleting theorem. The no-cloning theorem states that an arbitrary quantum state cannot be copied [7], while the no-deleting theorem states that when you have two copies of an arbitrary unknown quantum state, it is impossible to delete one of the copies while leaving the other untouched [8]. Any unknown quantum state can not be deleted, so from the output of a quantum gate the input should be deducible, as quantum information cannot be destroyed.

The previously discussed AND gate is not reversible. When the resulting output bit is 1, it is clear that both the two input bits were also 1 due to the definition of the gate. However, if the output bit is 0, the input bits could have either been 0 and 0, 1 and 0, or 0 and 1. Since this information is lost within the operation, the gate is not reversible. In contrast, the NOT-gate is reversible, an output bit of 1 indicates an input bit of 0 and vice versa. No information is lost, and thus the gate is reversible.

At the end of a quantum circuit a measurement is performed. A measurement collapses a qubit in any superposition to either 0 or 1, which one depends on the current superposition of the qubit, or its position on the Bloch Sphere. Important to note is that measurement is not reversible and therefore usually at the end of an algorithm. It removes the quantum mechanical element of the bit and practically makes it a classical bit again. Measurement is commonly thought of as random and indeterministic, as a superposition has a chance to be measured as 0 and a chance to be measured as one.

Although a qubit can take on any state on the Bloch Sphere, the quantum adder proposed by this thesis only uses quantum gates that keep the states 0 and 1 where they are or swap them. The circuit starts with all 0s and all qubits have either value 0 or 1 all throughout the circuit (assuming a perfect quantum environment, so without quantum decoherence) and no superposition is used. The measurement therefore does not affect the qubits in this perfect quantum environment. The adder only uses SWAP-gates, which acts on two qubits and swaps them, and NOT-gates. These gates are controlled by other qubits, meaning the SWAP or NOT gates are only performed when the qubits the gate is controlled by have specific values.

The different gates used in the designed adder with their meaning and circuit representation can be seen in Table 1.

Table 1: Relevant quantum gates and controls.

Symbol	Meaning
	NOT-gate: maps 0 to 1 and 1 to 0
	SWAP-gate: swaps two qubits
	Flips controlled only if this qubit is **1**
	Flips controlled only if this qubit is **0**
Spacer	Does nothing

Another important phenomenon in quantum computing is quantum entanglement. It essentially lets qubits be directly influenced by other qubits during measurement. For instance, it is possible to create superpositions with entanglement such that if the first of two qubits measures 0 the other measures 1 and vice versa. The adder designed in this thesis does not use quantum entanglement or superpositions to their full effect at all, as the goal is to design an adder that can be used by other quantum algorithms which do use entanglement and superpositions to their full effect.

All relevant quantum computing knowledge necessary to understand the adder has now been covered. It remains to find a solution to the fact that a IEEE 754 Floating-point adder is too large to convert to a quantum algorithm. The next chapter briefly covers the workings of the IEEE 754 Floating Point format after which the new Posit format will be introduced.

3 Posit Arithmetic

This chapter is largely based on the paper *Beating Floating Point at its Own Game: Posit Arithmetic* by John Gustafson and Isaac Yonemoto, and includes both figures and a table from the paper [9].

3.1 The IEEE 754 Floating-Point Format

To thoroughly comprehend how the Posit format works, it is important to understand the workings of the currently used IEEE 754 Floating-point format. Although the Posit format is technically also a floating-point format, from now on both "Floating-point" and "The Floating-point format" will refer to the IEEE 754 Floating-point format. The Floating-point format is 32 bits long and works in base 2, which means every digit is either a 0 or a 1 as is usually the case with bits. As can be seen in Figure 2, the Floating-point format works with a sign, exponent and a fraction.

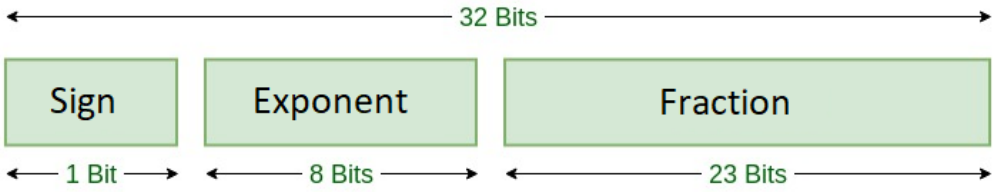


Figure 2: The Floating-point format [10]

A number is represented by 32 bits in total, the first of which is a sign bit. If the sign bit is a 0, the number is positive, and if the sign bit is 1, the number is negative. Next is the exponent, which is a string of 8 bits that multiplies the value of the fraction with a factor of 2^{e-127} where e is the value of the exponent (the 127 is due to the fact that the exponent should be able to be negative as well). For instance, if the exponent was **10000101** (which is 133 in binary) then the fraction would be multiplied with a factor of $2^{133-127} = 2^6$. Last is the fraction, where each bit represents the reciprocal of a factor of 2, so the first bit represents halves, the second bit quarters, etc. The fraction starts with a "hidden 1", which means a fraction is always of the form 1. followed by the actual fraction. For example, a (for illustration purposes 4-bit) fraction of **1010** would indicate the number $1 + 1 \times \frac{1}{2} + 0 \times \frac{1}{4} + 1 \times \frac{1}{8} + 0 \times \frac{1}{16} = 1\frac{5}{8}$. The fraction is always a number between 1 and 2 and is therefore normalized.

Finally one example of a floating-point number will be covered. Take the bitstring **01000001011000000000000000000000**, then the corresponding value would be

$$\begin{array}{lllll} 0 & 10000010 & 11000000000000000000000 & & \\ + & 2^{130-127} & \times (1 + \frac{1}{2} + \frac{1}{4}) & = & \\ + & 2^3 & \times 1\frac{3}{4} & = & 14 \end{array}$$

The Floating-point format has many other properties such as different infinities and NaNs (not-a-number), but these are not relevant to understand when comparing it to the Posit format. Many concepts of the Posit format are similar to those of the Floating-point format, so understanding Floating-point is preferred when learning about the Posit format.

3.2 The Posit Format

As discussed before, there simply are not enough qubits available to build a quantum adder for floating-point numbers with a structure similar to that of a regular adder. Quantum addition for floating-point numbers however remains an exciting prospect. Quantum computers or even simulators with millions of qubits will not be available in the near future, so a possible design for such quantum adder will need to use as little qubits as possible. Posit provides a solution in terms of size. Posit arithmetic is a number system designed to be more flexible and more efficient than Floating-Point Arithmetic, and its flexibility in size enables us to build a much smaller adder specifically designed for quantum addition.

The Posit format is very similar to the Floating-point format as can be seen in Figure 3. The first bit is still a sign bit, and signifies a positive number with a 0, and a negative number with a 1. In blue there are the exponent bits, which contribute a factor of 2^n, where n is the binary number that the bitstring represents (the -127 is gone due to the fact negative exponents are not needed anymore). In black remain the **fraction** bits, where the first fraction bit still represents a factor 0.5, the second a factor 0.25 etc.

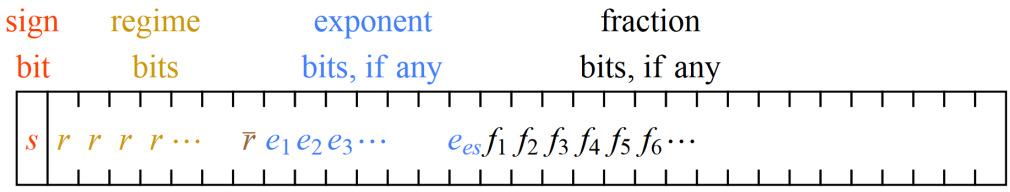


Figure 3: The Posit format [9]

New in contrast to the Floating-point format are the regime bits, whose purpose is to act as a superexponent. The regime bits multiply the number by another factor of the exponent, so if the exponent ranges from 0 to 2^7, then adding 1 to the value of the regime bits multiplies the number by a factor 2^8. Just like with Floating-Point, a two's complement is used when dealing with negative numbers. Usually posit numbers will be shown in their decoded expression (e.g. +0101 instead of **00101**), as this is much more clear and does not make a difference apart from hardware implementation. However, as will be seen later, the quantum adder proposed in this thesis only adds positive posits to other positive posits. In this case the decoded and original bitstrings are identical apart from the preceding 0 of the positive sign.

The regime consists of a binary string with all 0 or 1 bits in a row. This string is then terminated by the opposite bit (which will be color coded brown), or by the end of the string. The numerical meaning k depends on the length of the recurring 0 or 1 bits. This is best illustrated by Table 2.

Table 2: Different regime bitstrings and their numerical meaning [9]

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	−4	−3	−2	−1	0	1	2	3

As seen in the table, if m is the number of 0s or 1s in a row before being terminated by a 1 or 0 respectively, then if the bits are 0, we have $k = -m$, and if the bits are 1, we have $k = m-1$ due to the fact that a numerical value of 0 needs a binary string as well. To understand this let us return to the example of an exponent ranging from 0 to 2^7. The regime now works with factors of 2^8, so a bitstring of 0000 would indicate a factor of $(2^8)^{-4}$ and a bitstring of 110 would indicate a factor of $(2^8)^1$. There are two exceptions to this regime rule. A bit pattern of all **0** bits always has a numerical value of 0,

and it is also the **only** bitstring with value 0 in any posit system. In Floating-point many bitstrings have this value, so by only having one bitstring with value 0 posit allows other those other bitstrings to take on different values and therefore improve its efficiency. Likewise $\pm\infty$ also has one and only one bitstring; all **1** bits. This also means there is no $+\infty$ or $-\infty$, only $\pm\infty$.

An important difference between the Posit format and Floating-point format is the flexibility of the Posit format in the sizing of its fraction. The flexibility of the Posit format allows the user to choose a specific posit for a specific implementation. While single precision Floating-Point will always have length 32 (1 bit sign, 8 bit exponent and 23 bit fraction, as prescribed the IEEE 754 standard) [5], posits of arbitrary length can be implemented. Hence, posits can be considered as a customizable application-specific number format, whereas IEEE-754 is a general-purpose one. The letter n will be used to signify the length of the used posit system. While the length of a posit is fixed once chosen, the amount of regime and fraction bits used depends on the bit pattern. The amount of exponent bits does not do this and is always fixed unless there is no space due to a long regime (e.g. 0001). The letter p will denote the amount of exponent bits used given a certain posit length n, and any computer only needs these two inputs to generate an entire posit system. The system will be denoted by posit $<n,p>$, inspired by Theodore Omtzigt's Stillwater Supercomputing Inc [11].

Lastly, the rounding behaviour in Posit depends on the type of bit the final bit of the bitstring is. When rounding exponent or regime bits the 'nearest' number is the number with the smallest ratio and when rounding fraction bit the 'nearest' number is the number with the smallest difference. In the case of a tie the number is rounded to the nearest even number, which always corresponds to the number ending in a 0. Furthermore, posits do not underflow to 0 or overflow to infinity like floating-point numbers do. This will be important later in deciding the size of our quantum posit adder. To give some examples, in a posit $<5,1>$ environment (refer to Figure 7) the largest number is 64, so the calculation $64 + 64 = 128$ would result in 64, as 128 is rounded down, and the calculation $3 + \frac{1}{2} = 3\frac{1}{2}$ would result in 4, as 4 ends in a 0 in this environment while 3 ends in a 1.

3.3 The Posit Circle

The real numbers are usually presented on a number line. With the Posit format however, it is more intuitive to represent the real numbers on a circle due to its "one infinity". The circle is normalized as can be seen in Figure 4, with 0 on the bottom, $\pm\infty$ on the top, -1 one the left and 1 on the right of the circle. Negative numbers will be on the left side of the circle while positive numbers will be on the right side. The smallest possible posit, posit (<2,0>), has now been constructed and its posit circle can be seen in Figure 4. The goal is to extend this circle step by step to a posit <5,1>, the posit the designed quantum adder will be based on.

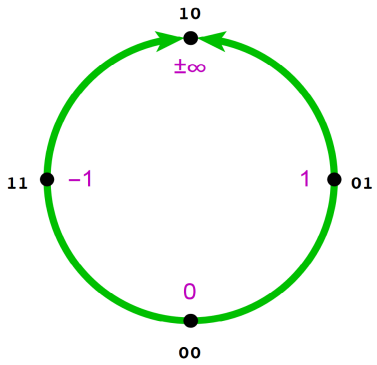


Figure 4: The posit <2,0> circle [9]

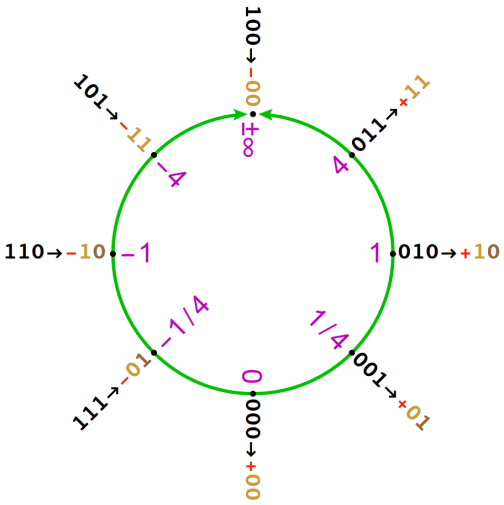


Figure 5: The posit <3,0> circle [9]

To extend the circle first the regime factor is added as seen before, remember that with 3 exponent bits the exponent could handle factors up to 2^7 and thus the regime worked with factors 2^8. Now, as the goal is a posit<5,1>, there is only one exponent bit available, so the regime works with factors of 4. The <3,0> posit circle is therefore as seen in Figure 5.

Note that during this first extension (and also with any further extensions) the values which were also in the previous circle have the same bit pattern with a **0** added to them.

Next, between 4 and $\pm\infty$ and between 0 and $\frac{1}{4}$ the same approach is followed as before; another regime bit is added here and the values 16 and $\frac{1}{16}$ are added. Between 1 and 4 the new bit pattern 101 is added. The regime of 1 is terminated by the 0 and thus our newly added 1 is an exponent bit. This contributes a factor 2 and so the value added to the circle between 1 and 4 is also 2. Repeating this process on the rest of the circle produces the next posit<4,1> circle as seen in Figure 6. Note the change to <4,1> instead of <4,0> due to the addition of the exponent bit.

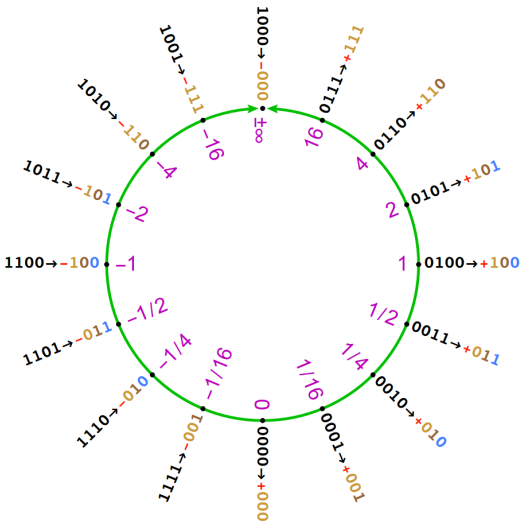


Figure 6: The posit <4,1> circle [9]

It remains to add fraction bits. Between 16 and $\pm\infty$ the value 64 is added because of the addition of another regime bit, and between 4 and 16 the value 8 is added as a result of the added exponent bit. Between 1 and 2 the bitstring 1001 is added. The regime of 1 is terminated by the opposing 0, and the exponent bit is also 0, therefore the last remaining bit will be a fraction bit. Since this is the first fraction bit in the bitstring, it will contribute a factor 0.5 and the number between 1 and 2 will be 1.5. In general, when adding a fraction bit the new number is always halfway in between the previous two. Repeating this process results in the posit <5,1> circle, as can be seen in Figure 7.

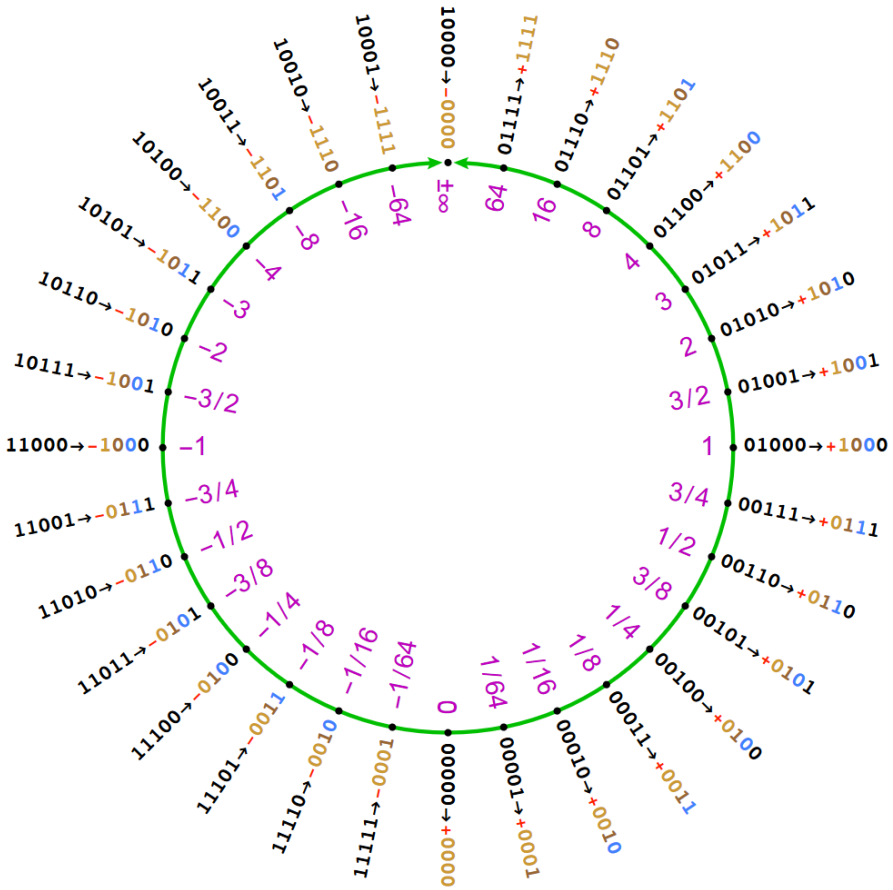


Figure 7: The posit <5,1> circle [9]

3.4 Posit Example

Lastly, in this section one example of posit decoding will be covered. Consider the bitstring **0000110111011101**, which is color coded in Figure 8 :

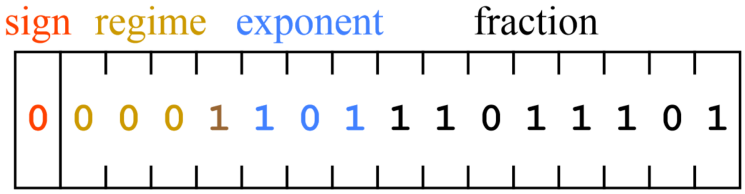


Figure 8: An example of a posit bitstring [9]

Note that the posit bit string has length 16 with 3 exponent bits, so this is a posit <16,3>. The leading 0 means the number is positive. Next there are three 0s and a 1 terminating the regime, so 0001 means a k value of -3, this contributes a factor of 256^{-3}. The 3-bit (fixed size) exponent of 101 translates in binary to the value 5, so the exponent is 2^5. Finally, the remaining bits are the binary fraction, which translate to the decimal value 221, so our fraction is $(1 + \frac{221}{256})$ (256 is due to the fact that of the 16 bits, 8 remain for the fraction, which means a the last bit adds $\frac{1}{256}$). Thus the value works out to:

$$+\ 256^{-3} \times 2^5 \times (1 + \frac{221}{256}) = 477 \times 2^{-27} \approx 3.55393 \times 10^{-6}$$

All properties of the Posit format important for the quantum adder have now been covered, but this topic is far from being fully explored. Further explanation and reading on posits, like its comparison to floating-point in terms of performance, can be found both on posithub.org [12] and in the paper mentioned in the beginning of the chapter. An efficient implementation of posits is available at the universal library of Stillwater Supercomputing [13]. Next the adder itself will be discussed, going into detail about both its workings and uses, and how it can be optimized for optimal performance.

4 Quantum Posit Adding Algorithm

4.1 Motivation

One of the most well known quantum algorithms, Shor's algorithm, is a quantum algorithm for integer factorization [4]. Classical computers can not factorize large integers yet if they are the product of two or three 300 digit primes, but a quantum computer can solve this problem by using Shor's algorithm. A part of Shor's algorithm is adding two different integers together (during the period-finding subroutine). Addition on a quantum computer has already been explored for more than two decades now, and many different quantum integer adders have been designed, like the Draper adder [14], Cuccaro adder [15] and Muños-Coreas Adder [16]. However, to use these algorithms the quantum computer will need to initialize the original 600 digit number in binary, as all of these algorithms operate in a base-2 number system. The number 10^{600} can be represented in binary with $\lceil \log_2(10^{600}) + 1 \rceil = 1995$ bits, though as seen before this many qubits can not be realized in the near future. For that reason a different quantum adder based on floating-point arithmetic (not necessarily IEEE 754) will need to be designed to drastically decrease the number of qubits needed to perform basic operations.

As most of the current adding algorithms are based on integer adding, it seems logical to think about quantum addition for real-valued numbers as well. As seen before addition of real-valued numbers on an ordinary computer is done with floating-point arithmetic and uses barrel shifters [17], which is currently not a realistic option on quantum computers as they are based on multiplexers which are not reversible. Reversible versions of multiplexers do exist, but these are accompanied by many undesirable garbage output qubits and a huge increase in size of the algorithm. For these reasons the Quantum Posit Adding Algorithm (QPAA) was designed, a quantum adding algorithm based on the Posit format which can perform addition of real-valued numbers, while also being able to handle larger integers with much fewer bits. Important to note is that the addition of real-valued numbers in this algorithm is done in a classical way. The goal was not to design an addition algorithm that uses quantum mechanics to improve a classical addition algorithm, but to design an addition algorithm that works on a quantum computer, so that larger addition can be used in actual quantum algorithms like Shor's algorithm. Classical posit adders also work with barrel shifters, which is why this algorithm uses a tablebase approach instead to successfully determine the outcome of a calculation, as will be shown in this

chapter.

4.2 Unoptimized QPAA Design

The QPAA has a total of 8 input bits, which amounts to two <5,1> posits, since sign bits are omitted as only positive numbers are added. Note that the top bit in the QPAA corresponds to the least significant bit of the bitstring. During the operation the first four bits are added to the second four bits, and the result is stored in the first four bits again. Like any quantum circuit, the QPAA needs to be reversible because the corresponding matrix has to be unitary. If the 8 output bits would be **11111111** it would be unclear what number was added to 1111, because the number did not change at all. Adding any number to 1111 results in 1111, as there is no rounding to $\pm\infty$ as discussed earlier. To achieve this, an extra four ancilla bits are needed to store information "thrown away" by the operation.

The QPAA has two components, an initialization and the actual calculation, and is ended by a measurement on every qubit. The full algorithm is seen in its most high-level form in Figure 9.

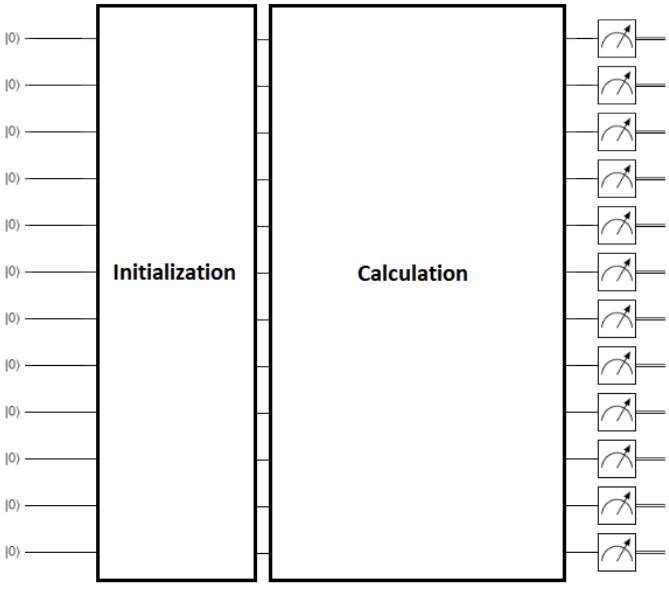


Figure 9: The Quantum Posit Adding Algorithm for posit <5,1>

4.2.1 Initialization

Posit <5,1> has a total of $16^2 = 256$ possible combinations of numbers to add. Most of these additions are trivial and do not require any computation at all. In 215 of these combinations the result is the largest of the two added numbers. For that reason during initialization the smallest of the two numbers is swapped into the ancilla qubits. This empties the qubits where the result will be stored, and next the larger number is copied into the result bits using controlled-NOT gates. The full initialization circuit can be seen in Figure 10. This small circuit already executes $\frac{215}{256} \approx 84\%$ of calculations correctly. After the initialization both the top 4 and the middle 4 qubits contain the larger number, and the bottom 4 qubits contain the smaller number.

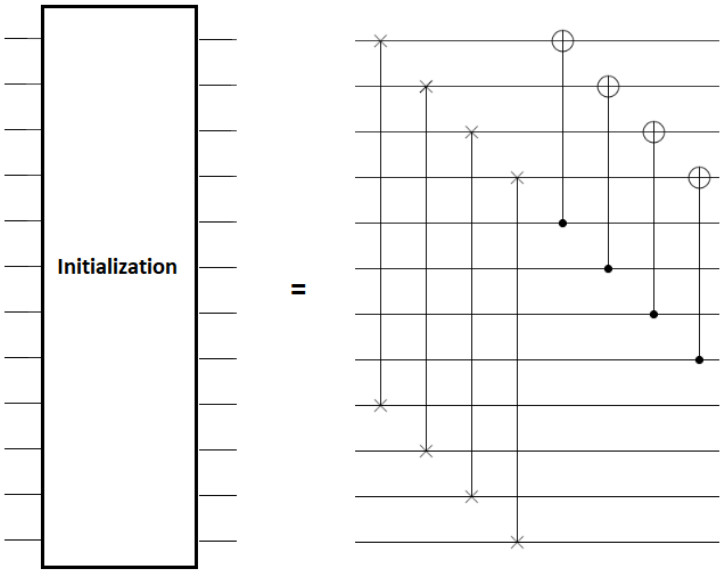


Figure 10: Initialization circuit of the QPAA for posit <5,1>

After the larger number is copied into the target qubits only the remaining 41 cases where the larger number is not the correct solution have to be accounted for. This is done in the remaining 8 blocks. Each block accounts for a regime-exponent combination (e.g. 1101 or 010) and uses the gates from Table 1 to transform the largest number into the correct solution. There are a total of 12 regime-exponent combinations, but only 8 blocks, as some regime-exponent combinations never see a change when a number equal to or smaller than that number is added to it, so no block is needed for these combinations:

- 1111 or 64; there is no rounding up to $\pm\infty$, so any number added to 64 is still 64.
- 1110 or 16; as adding 16 to 16 gives 32 which is again rounded down to 16. Therefore any number added to 16 will be rounded down to 16 as well, so no block is necessary for this regime-exponent combination.
- 0001 or $\frac{1}{64}$; the only two numbers smaller than or equal to $\frac{1}{64}$ are $\frac{1}{64}$ itself and 0. Adding $\frac{1}{64}$ to $\frac{1}{64}$ gives $\frac{1}{32}$, which rounds down to $\frac{1}{64}$ again, and adding 0 to $\frac{1}{64}$ is still $\frac{1}{64}$
- 0000 or 0; as the only positive number smaller than or equal to 0 is 0, and $0 + 0 = 0$.

4.2.2 Calculation

The calculation is split into 8 different blocks, where each block accounts for a certain regime-exponent combination. The blocks are separated by spacers (except for the last two blocks), which are no-ops introduced to improve readability of the circuit. For a given regime-exponent combination usually only 1-4 numbers smaller than it will affect the number. The block is a small quantum circuit that deals with these specific calculations and transforms the larger number to the correct solution using the gates seen in Table 1. The circuit checks if the smaller number is of a specific form of **0**s and **1**s, and if this is the case it changes the result bits with NOT-gates so that the solution will be registered there. A combination is only effected by exactly 1 or 0 blocks. If the combination is not changed at all we are dealing with one of the four combinations mentioned above, otherwise exactly 1 block checks all possible calculations containing that combination. The full calculation circuit can be seen in Figure 11 (and in larger scale in Appendix A).

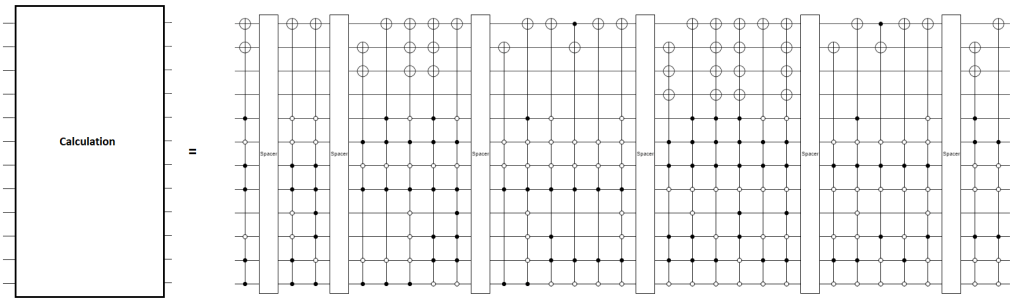


Figure 11: Calculation circuit of the QPAA for posit <5,1>

To understand how a single block works, the second block seen in Figure 12 will be explained below. This second block covers the combination 1100, which is the number 4. The only numbers that will produce a result that differs from 4 when added to 4 are 3 and 4 itself, as $4+3 = 7$ which is rounded up to 8 and $4 + 4 = 8$. Adding any number smaller than 3 to 4 results in 4, so these calculations are already accounted for by the initialization.

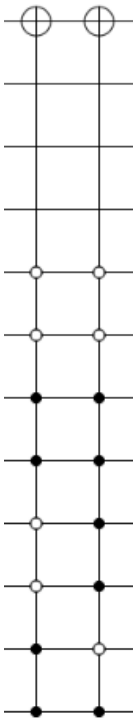


Figure 12: Block 2 of the QPAA for posit <5,1>

When 4 (1100) is added to 4 (1100) and 3 (1011), the 12 bits after initialization are **0011 0011 0011** or **0011 0011 1101** respectively. Remember that in the algorithm the last bit of the bitstring is the topmost qubit. In the circuit shown in Figure 12, first it is checked if both the middle 4 qubits (the larger number) and the final 4 qubits (the smaller number) are **0011** (4), if this is the case, a NOT-gate is performed on the first qubit, which changes the first 4 qubits from **0011** to **1011** or 8, as intended. Similarly, the second column checks if the middle 4 qubits are **0011** (4) and the final 4 qubits are **1101** (3), if this is the case again a NOT-gate is performed on the first qubit, which changes the first 4 qubits from **0011** to **1011** or 8.

The other blocks work in a analogous way, changing the top 4 qubits dependent on the bottom 8. As expected the blocks which cover regime-exponent combinations around 1 are larger than the blocks which cover regime-exponent combinations further away from 1, as the relative size between numbers is smaller. This means more numbers have a result different than the larger number and so more cases need to be accounted for. However, this algorithm is far from optimal, as it contains many multi-controlled gates which take long to execute. Any form of optimization matters in these algorithms, and the next section will discuss how and why this algorithm can be optimized both "externally" (by removing controls in the algorithm) and "internally" (by changing the way certain gates are implemented when running the algorithm).

4.3 Optimization

4.3.1 External Optimization

Much of the system can be optimized by removing certain conditions. Even removing one control in a very large multi-controlled-NOT can have a significant impact on the speed of the algorithm. The reason for this is that any unitary n-controlled gate U can be constructed using two (n-1)-controlled-NOT gates and an (n-1)-controlled version of "square root version" of U, as can be seen in Figure 13 [18].

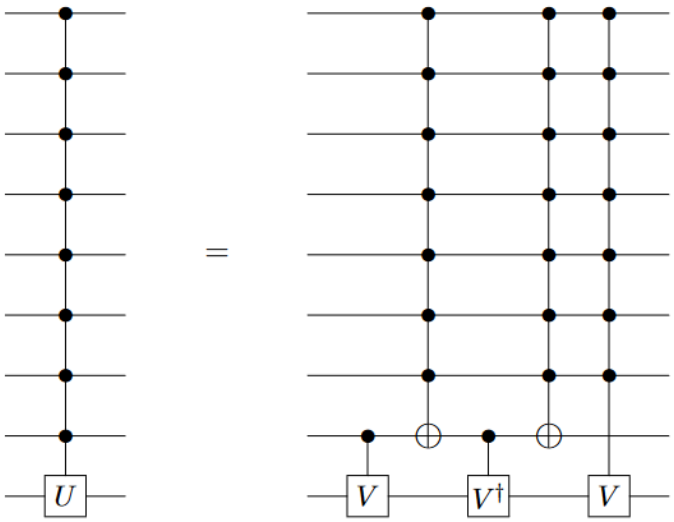


Figure 13: An n-controlled gate written as (n-1)-controlled gates

On a quantum computer all gates have to be implemented as a sequence of smaller gates that operate on a maximum of two qubits. A large multi-controlled-NOT thus has to be written as a circuit of many smaller gates, first writing the n-controlled gate as multiple (n-1)-controlled gates, then writing each (n-1)-controlled gate as multiple (n-2)-controlled gates, etc. Therefore removing the need for extra controls speeds up the execution significantly. The goal of this external optimization is to remove as many unnecessary controls as possible from the calculation circuit shown in Figure 11.

All of these removals will use the fact that it is known which of the two added numbers is larger and that this number is located in the middle 4 qubits when the algorithm is executed. When executing a certain block different controls are used to check the bottom qubits, but some of these checks now redundant and can be removed. To best show this procedure, let us recall block 2:

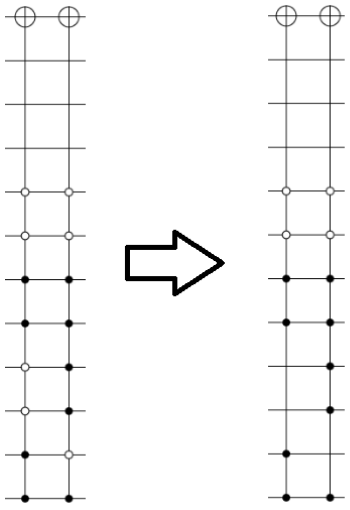


Figure 14: Block 2 of the QPAA for posit <5,1> optimized

The only numbers that were checked were 4 (**1100**) and 3 (**1011**). However, checking for the **0** in **1011** is not necessary, as it can not be a **1**. If it was a **1**, the number on the bottom would be **1111** (16) which is larger than 4. Yet it was assumed at the start of the algorithm the larger number would be in the middle 4 qubits, so this is not possible. Similarly, checking for the two **0**s at the end of **1100** is not needed, as they can not be **1**s. If one of them or both were **1**s, then the number represented by the bitstring would be larger than 4.

The same logic can be applied to every single block. Controlling for a **1** cannot be removed, as a **0** instead of a **1** would result in a smaller number, but controls for **0**s can be removed if replacing them by a **1** results in a larger number than the regime-exponent combination of the block. Doing this results in the optimized calculation circuit shown in Figure 15 (also visible in larger scale in Appendix B.

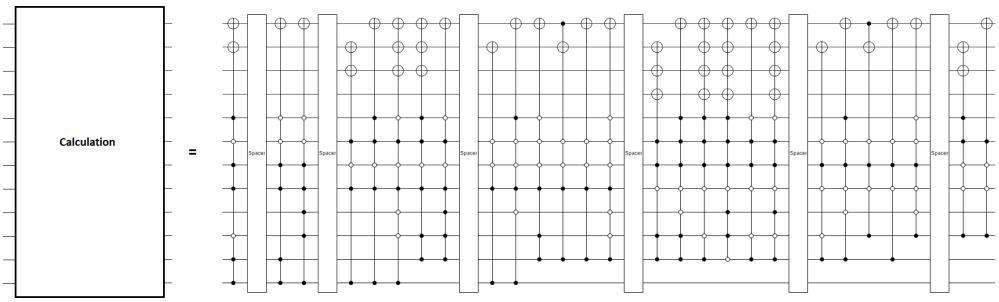


Figure 15: Optimized calculation circuit of the QPAA for posit <5,1>

4.3.2 Internal Optimization

The algorithm can also be optimized "internally", meaning that the different implementations of the gates in the algorithm can be optimized when writing them in their full form. The multi-controlled gates in the algorithm can be written out and then optimized by removing unnecessary controls as done before with external optimization, so that the overall algorithm runs faster. Usually this happens because certain controls cancel each other out which makes them redundant.

The problem with this kind of optimization in this algorithm is that the order in which the operations need to be executed for optimal performance is extremely unclear. All the blocks and even all the columns within the blocks can be run in arbitrary order and the results would be the same. The current order of the blocks and columns is chosen so that the algorithm is easily readable and understandable, but for an effective implementation of the algorithm this is not important at all. Different orders will need to be experimented with to find the order in which this algorithm is best executed.

5 Experimental Validation

To show that the QPAA indeed produces the correct result, two examples of calculations will be shown here, first calculating the correct result by hand and then comparing it to the result produced by the algorithm.

Example 1
In this first example the calculation 2 (**1010**) + $1\frac{1}{2}$ (**1001**) will be performed. The result should be $2 + 1\frac{1}{2} = 3\frac{1}{2}$ which gets rounded to 4. In the algorithm 2 (**1010**) and $1\frac{1}{2}$ (**1001**) are both loaded in back to front with the larger number first, so together with the last 4 qubits which start as 0, the input will be **1001 0101 0000**.

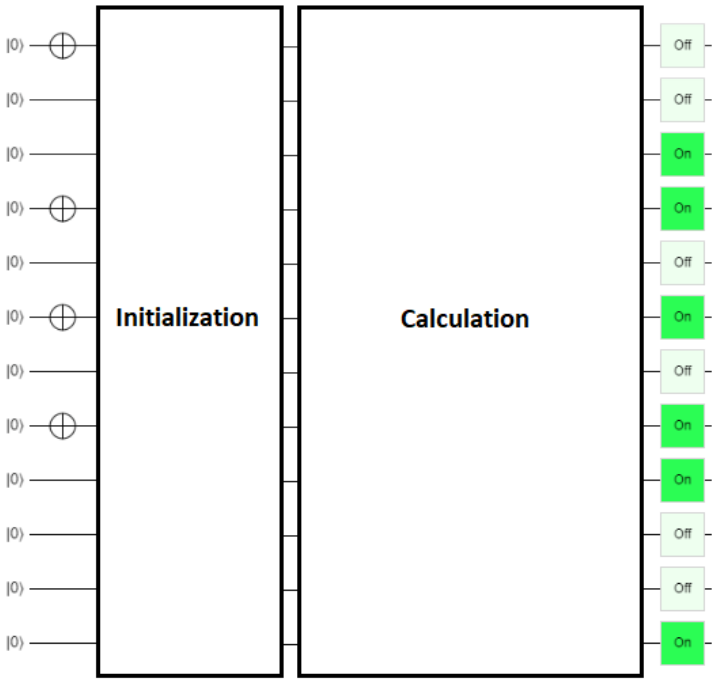


Figure 16: Calculation example 1

The output can be seen in Figure 16, where a ON signifies a 1 while a OFF signifies a 0. The first four qubits of the output are the result of the calculation, the second four the larger number, and the third four the smaller number. An output of **0011 0101 1001** therefore means the result of the calculation was **1100** or 4, as expected.

Example 2
In this second example the calculation $\frac{1}{2}$ (**0110**) + $\frac{3}{8}$ (**0101**) will be performed. By hand the result should be $\frac{1}{2} + \frac{3}{8} = \frac{7}{8}$ which gets rounded to 1. The input of the algorithm will be **1010 0110 0000**.

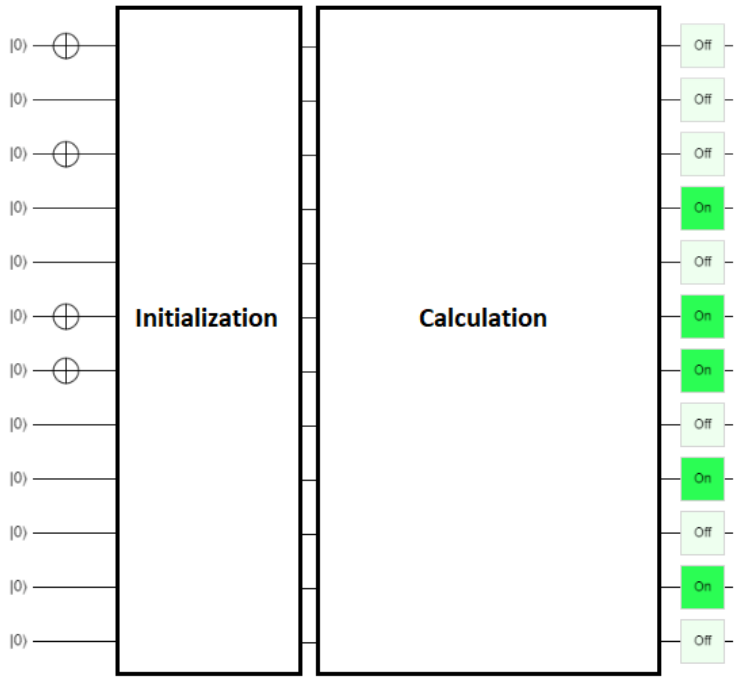


Figure 17: Calculation example 2

The output of this calculation can be seen in Figure 17. An output of **0001 0110 1010** indicates the result of the calculation was **1000** or 1, as expected.

6 Conclusions

The Posit format provides an environment where quantum addition is possible on a larger scale. While it does not use superpositions or quantum entanglement, the Quantum Posit Adding Algorithm allows larger numbers to be added with fewer qubits than before. The adder operates correctly in a perfect quantum environment, but the question remains how the adder operates when quantum decoherence is a factor.

The algorithm has been optimized significantly by removing controls, only leaving it to be optimized even more through implementations of the different gates used by the algorithm, with the implementation of the multi-controlled gates being the most important. The multi-controlled gates can be realized in multiple ways, the most efficient still uncertain, and experimenting with the order of logic gates in the algorithm depending on this implementation will be the key to optimizing the algorithm.

7 Extensions and non-working ideas

In this chapter extensions of the algorithm will be discussed, as it is clear this smaller algorithm cannot operate on larger posits yet. Ideas for constructing a larger adding algorithm, but also a subtractor will be presented, along with different concepts which were interesting ideas, but proved not to work when carefully considered.

7.1 Extensions

7.1.1 Extending to a <n,p> posit

The optimized calculation circuit seen in Figure 15 operates on two <5,1> posits, but designing an algorithm in a similar way that can operate on larger posits is definitely feasible. Since the algorithm is designed with a tablebase in mind, the same tablebase approach can be used to form a larger adding algorithm for any $<n,p>$ posit. The fact that most additions do not change the larger number remains, and only a certain amount of numbers smaller than the larger number need to be checked. To illustrate this, a small start is now made to the algorithm that adds two <7,1> posits. The posit <7,1> circle can be seen in Figure 18 (and in larger scale in Appendix C).

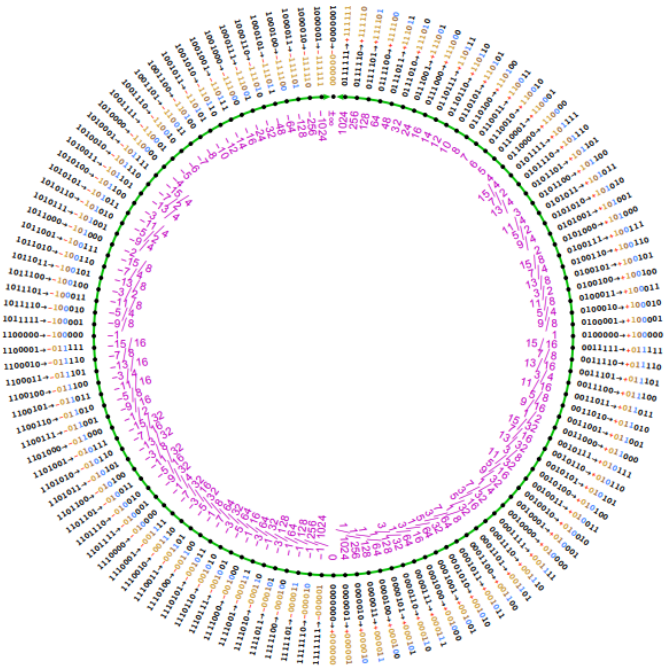


Figure 18: The posit <7,1> circle [9]

The algorithm starts with the same initialization as before. Next the largest number in the posit <7,1> circle is 1024, but adding anything to 1024 does not change the number. This means nothing needs to be added to the algorithm, and the same reasoning applies to the next number; 256. The first addition which gates are needed for is $128 + 128 = 256$, so the first gates will be two controlled NOT-gates on the first two bits to transform 128 (**1011110**) into 256 (**0111110**) (note that the bit order is reversed in the adder). The rest of the algorithm can be constructed in a similar way.

In fact, a program can be written to construct this algorithm. Given a posit <n,p> circle, a program can determine exactly which numbers change the larger number and then construct the logic circuit needed to achieve the correct change. However, this program will most likely not be able to find the most efficient logic circuit doing so, as sometimes a smaller logic circuit will be able to perform multiple calculations at the same time which the program will not be able to recognize. It will still be an effective way to design a larger adder that works though, the question is if the algorithm designed by the program will be significantly faster than a directly implemented IEEE Floating-Point quantum adder using reversible barrel shifters.

7.1.2 Designing a subtractor

A posit <5,1> subtraction algorithm could work very much alike the adder algorithm, but it does come with some added challenges. The current QPAA can add positive posits to positive posits, but also negative posits to negative posits. As long as the sign of the two numbers are the same the QPAA can correctly determine the outcome, because it just copies the sign of the larger number into the result. When designing a subtraction algorithm the sign should be taken into account, first checking which number is larger and then copying the sign of that larger number into the result sign through a controlled NOT gate.

An issue is the fact that many more calculations need to be performed by the subtractor. With the adding algorithm $\frac{215}{256} \approx 84\%$ of the calculations could be done instantly with the initialization, but with a subtraction algorithm this number would be much smaller, as more numbers change the larger number. For example, returning to the posit <5,1> circle shown in Figure 7, only two numbers change the number 4 when added to it (namely 3 and 4 itself) but six numbers change the number 4 when subtracted from it ($\frac{3}{4}$, 1, $1\frac{1}{2}$, 2, 3 and 4). Other numbers act the same way, every number has at least as many numbers that change the number when subtracted from it than change the number when added to it.

This difference results in a significantly longer algorithm, and it brings the question if tablebase is still the right approach for this subtractor. Maybe subtractors should have some kind of inherent calculation or should be combined with the adder instead of designing separate algorithms. One thing is clear, which is that the classical way of subtracting does not work. In classical computers Floating-Point adders work with a two's complement, which basically turns the subtraction into an addition so that the overflow makes the number smaller again. However, overflow does not exist in the Posit format, so any form of adding cannot make the number smaller than it already is. Furthermore, in the Floating-Point format the exponent is always at the same place which makes it easy to decide where the two's complement needs to begin, but in the Posit format the place of the last digit of the exponent differs, meaning two's complements of different numbers have different meaning at the same places in the two's complement. For these reasons the idea of two's complement seems impractical, although the tablebase approach also might not prove efficient for a quantum subtraction algorithm.

7.2 Non-Working Ideas

7.2.1 Superpositions as new values

Currently superpositions have no specific use in the QPAA, but what if a superposition could be used as some sort of "in-between value". For instance, in the posit <4,1> environment (seen in Figure 6) 2+1 = 3 would be rounded to 4, but the actual value of the number would be exactly between **0101** (2) and **0110** (4). The first three qubits are exactly the same in both answers, only the last qubit differs. Now what if the algorithm could be designed so that the last qubit is in a superposition between 0 and 1, exactly in the middle as 3 is exactly between 2 and 4. Achieving this superposition is possible with multiple different gates, most notably the Hadamard gate or simply a $\frac{\pi}{2}$ rotation gate around the x-axis or y-axis.

The problem with assigning new values like this is that the qubit is not the new in-between value 3, but actually 50% of the time measured as 2 and 50% of the time measured as 4. To give an example why this does not work, adding $\frac{1}{2}$ to this in between value should result in $3 + \frac{1}{2} = 3\frac{1}{2}$ which always gets rounded to 4, but 50% of the time the actual result of the calculation will be $2 + \frac{1}{2} = 2\frac{1}{2}$ which gets rounded back down to 2, and 50% of the time the result will be $4 + \frac{1}{2} = 4\frac{1}{2}$ which gets rounded back down to 4. The result of the calculation is therefore the superposition we started with; the in-between value 3, instead of the correct result of 4.

To solve this two new qubits could be added to the algorithm to save information on the other superpositions, which would make the superpositions as in-between values work. However, those new qubits could also be used to extend a $<n,p>$ posit to a $<n+1,p>$ or a $<n+1,p+1>$ posit. The Posit format is specifically designed to maximize both precision and range, and therefore if the extra qubits were available it would be more beneficial to extend the posit environment than to introduce in-between values using superpositions.

7.2.2 Rotating information into a single qubit

Rotating in powers of 2
Classical bits only have values of 0 or 1, but qubits can have values 0, 1, and a superposition of the two. Using the fact that at the start of the quantum algorithm all qubits are initialized as 0 or 1, is it possible to take a bitstring of 0s and 1s and save all that information into a single qubit?

The idea was to perform certain rotations on the single qubit depending on if the value of the classical bit was a 1 or not. If the nth bit was a 1, a rotation around the x-axis of $\frac{\pi}{2^{n+1}}$ (so the maximal rotation would be $\frac{\pi}{2}$) would be performed. This process would be repeated for a second number on the same qubit, and to add them the rotations would simply be added. The reason this does not work is the fact that this operation is not reversible. For example, consider the two identical bit strings 01 and 01. As prescribed by the method above, a turn of $\frac{\pi}{8}$ would be performed on a qubit twice, resulting in a qubit rotated $\frac{\pi}{4}$ around the x-axis. However, the same result would be obtained if 10 and 00 were added. This shows the operation is not reversible, though there seems to be a fix for the problem.

Rotating using reciprocals of primes
If instead of rotating around the x-axis with turns of $\frac{\pi}{2^n+1}$, turns depending on the primes would be performed, the problem would seemingly be solved, as from any rotation the previous numbers would be deducible. Sadly, the sum of the reciprocals of the primes diverges [19], meaning rotating depending on them would not be possible in a reversible way. To show another way why the idea of turning information into a single qubit does not work, consider the operation as a matrix. The information of 0s and 1s is transferred out of the (in this example 3) qubits containing those values and into the single target qubit, so the operation has the matrix form seen in Figure 19.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

Figure 19: Matrix for rotating classical information into a qubit

Remember that quantum gates are always represented by unitary matrices, and clearly the matrix in Figure 19 is never unitary, effectively dismissing the idea instantly.

7.2.3 Projecting the posit circle on a single qubit

The thought of projecting a posit circle onto the Bloch Sphere does not seem like an unreasonable concept. Essentially the idea would be to let different superpositions on the Bloch sphere represent different numbers on a posit circle. By rotating the single qubit the value it represents would change, depending on the distance to the poles of the qubit. If possible this would even allow two different posit values to be saved into a single qubit by rotating around the x-axis for one posit and around the y-axis for another. The benefit of this would be the amount of qubits needed to represent a value. In the optimized QPAA the same amount of qubits as classical bits is needed, but only a single qubit would be needed with this concept.

However, this idea directly violates the no-teleportation theorem [20]. The idea would allow classical information (i.e. the bitstring of the posit) to be extracted directly from an arbitrary state of a qubit, because you would want the entire circle projected onto the Bloch sphere. In contrast, it is possible to convert classical information into quantum information and then back, but to do this you need qubits equal to the amount of classical bits you want to convert, yet the entire concept was to reduce the number of qubits needed.

References

[1] Brooks, A. S., & Smith, C. C. (1987). Ishango revisited: new age determinations and cultural interpretations. *The African Archaeological Review*, *5*, 65–78.

[2] Ercegovac, M. D., & Lang, T. (2004). Two-Operand Addition. *Digital Arithmetic*, 50âĂŞ135. https://doi.org/10.1016/B978-155860798-9/50004-X

[3] Uma, R., Vijayan, V., Mohanapriya, M., & Paul, S. (2012). Area, delay and power comparison of adder topologies. *International Journal of VLSI Design & Communication Systems*, *3*(1), 153.

[4] Shor, P. W. (1994, November). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (pp. 124-134). IEEE.

[5] IEEE 754-2019 - IEEE Approved Draft Standard for Floating-Point Arithmetic. (n.d.). Retrieved August 13, 2019, from https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html

[6] The Bloch Sphere. (2012, April 26). Retrieved August 13, 2019, from http://quantumcomputing101.blogspot.com/2012/04/bloch-sphere.html

[7] Park, J. L. (1970). The concept of transition in quantum mechanics. *Foundations of Physics*, *1*(1), 23–33. https://doi.org/10.1007/BF00708652

[8] Pati, A. K., & Braunstein, S. L. (2000). Impossibility of deleting an unknown quantum state. *Nature*, *404*(6774), 164.

[9] Gustafson, J. L., & Yonemoto, I. T. (2017). Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, *4*(2), 71-86.

[10] IEEE Standard 754 Floating Point Numbers - GeeksforGeeks. (2019, January 31). Retrieved August 13, 2019, from https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

[11] Stillwater | Knowledge Processing Platform. (n.d.). Retrieved August 13, 2019, from http://www.stillwater-sc.com/

[12] Unum & Posit- Next Generation Arithmetic. (n.d.-b). Retrieved August 13, 2019, from https://posithub.org/

[13] stillwater-sc/universal. (n.d.). Retrieved August 13, 2019, from https://github.com/stillwater-sc/universal

[14] Draper, T. G. (2000). Addition on a quantum computer. *arXiv preprint quant-ph/0008033*.

[15] Cuccaro, S. A., Draper, T. G., Kutin, S. A., & Moulton, D. P. (2004). A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184*.

[16] MuÃśoz-Coreas, E., & Thapliyal, H. (2017). T-count optimized design of quantum integer multiplication. *arXiv preprint arXiv:1706.05113*.

[17] Barrel-shifter (8 bit). (n.d.). Retrieved August 14, 2019, from https://tams.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/60-barrel/shifter8.html

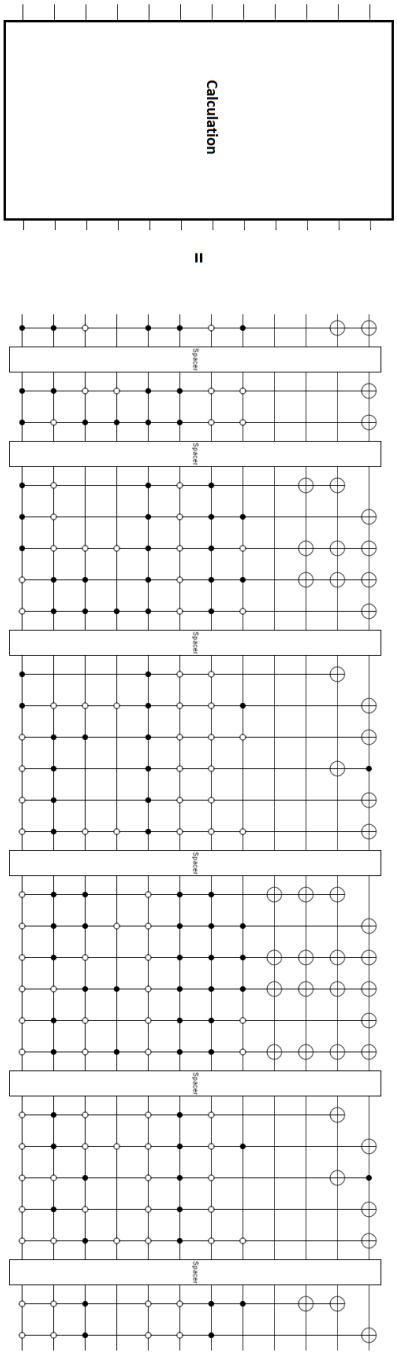
[18] Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., âĂę Weinfurter, H. (1995). Elementary gates for quantum computation. *Physical Review A*, *52*(5), 3457–3467. Lemma 7.5. https://doi.org/10.1103/PhysRevA.52.3457

[19] Euler, L. (1737). Variae observationes circa series infinitas. *Commentarii academiae scientiarum imperialis Petropolitanae*, *9*(1737), 160-188.

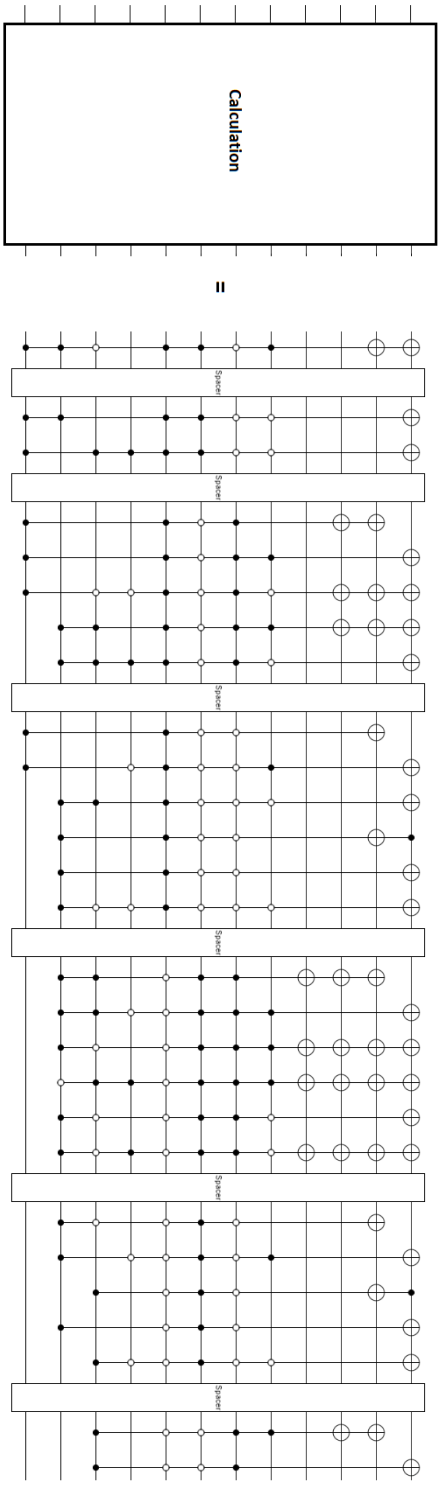
[20] Gruska, J., & Imai, I. (2001). Power, Puzzles and Properties of Entanglment. *Machines, Computations, and Universality: Third International Conference*, p. 41.

Appendices

Appendix A: Figure 11 in larger scale



Appendix B: Figure 15 in larger scale



Appendix C: Figure 18 in larger scale