Faster Quantum Number Factoring via Circuit Synthesis

Article *in* Physical review A, Atomic, molecular, and optical physics · January 2013

DOI: 10.1103/PhysRevA.87.012310 · Source: arXiv

2 authors:

Igor L. Markov
University of Michigan

Mehdi Saeedi
AMD

Faster Quantum Number Factoring via Circuit Synthesis

Igor L. Markov[1,*] and Mehdi Saeedi[2]
[1] Department of EECS, University of Michigan, Ann Arbor, MI 48109-2121
[2] Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-2562

A major obstacle to implementing Shor's quantum number-factoring algorithm is the large size of modular-exponentiation circuits. We reduce this bottleneck by customizing reversible circuits for modular multiplication to individual runs of Shor's algorithm. Our circuit-synthesis procedure exploits spectral properties of multiplication operators and constructs optimized circuits from the traces of the execution of an appropriate GCD algorithm. Empirically, gate counts are reduced by 4-5 times, and circuit latency is reduced by larger factors.

PACS numbers: 03.67.Ac, 03.67.Lx, 89.20.Ff

I. INTRODUCTION

Shor's number factoring remains the most striking algorithm for quantum computation as it quickly solves an important task [1] for which no conventional fast algorithms were found in 2,300 years [16]. Today, a scalable implementation of Shor's technique would have dire implications to Internet commerce. Laboratory demonstrations factored $15 = 3{\cdot}5$ circa 2000 [2], but further progress was slow [3–5] as factoring sizable semiprimes requires very large circuits. The bottleneck of Shor's number factoring is in *modular exponentiation* — a reversible circuit computing ($b^z \mod M$) for known coprime integers b and M. This computation is performed as a sequence of conditional modular multiplications ($\mathcal{C}M$) [6] by precomputed powers of a randomly selected base value (b), controlled by the bits of z (Figure 1). In most cases, $b = 2$ or $b = 3$ suffice [7]. Such $\mathcal{C}M$ blocks are assembled from unmodified unconditional modular multiplication blocks ($\mathcal{U}M$) using pre- and post-processing: since multiplication always preserves the integer 0, a $\mathcal{U}M$ block can be "turned off" by conditionally swapping a 0 with its inputs and then restoring the inputs by an identical swap. Conditional swaps can be simplified, and further circuit optimizations focus on $\mathcal{U}M$ blocks. These steps are reviewed in detail in [7].

II. PRIOR WORK

$\mathcal{U}M$ blocks are assembled from modular additions and multiplications by two, scheduled according to the binary expansion of the constant multiplicand and its modular inverse [6] (see a contemporary summary in [7]). In one popular approach, the input value x is copied into a zero-initialized register to obtain (x, x). To compute $13x$, follow the binary expansion $13{=}b1101$: (x, x)-$(2x, x)$-$(3x, x)$-$(6x, x)$-$(12x, x)$-$(13x, x)$. Now the second register must be restored to 0 for the next $\mathcal{U}M$ block to use it. However, this requires dividing $13x$ by 13, i.e., multiplying by the *modular inverse* of 13. For $M = 101113 = 569 \cdot 1777$, the inverse of $C = 13$ is 77778, (10010111111010010_2), requiring a large circuit. In [7], we constructed alternative circuits without computing modular inverses. To accomplish this, we introduced circuit blocks for modular multiplication and division by two that restore their ancillae to 0. We then estimated costs of circuit blocks for modular addition, subtraction, multiplication and division by two, and several others [7, Table 2]. Using these blocks, we found optimal $\mathcal{U}M$ circuits for each C, M up to 15 bits. The same procedure can be used for different cost estimates, but optimal search does not scale well beyond 15 bits. Numerical results demonstrated that traditional circuits based on binary expansion are far from optimal, thus asking for scalable constructions beyond 15 bits.

Researchers optimizing circuits for Shor's algorithm [8, 9] adapted these circuits to use only nearest-neighbor quantum couplings [10] and restructured them to leverage parallel processing [11]. Applying multiple quantum couplings in parallel allows one to finish computation faster. The smaller required lifespan of individual qubits additionally reduces the susceptibility of qubits to decoherence and decreases the overall need for quantum error-correction. The runtime (latency) of parallel quantum computation is estimated by the *depth* of its quantum circuit, i.e., the maximum number of gates on any input-output path. Depth reductions in the literature sharply increase the required number of qubits, e.g., by 50 times or more, making them impractical for modern experimental environments where controlling 50-100 qubits remains a challenge. Vice versa, prior circuits with 1-2 fewer qubits use more gates [12]. Rosenbaum has shown [13] how to adapt unrestricted circuits to nearest-neighbor architecture using teleportation, while asymptotically preserving their depth. For Shor's algorithm, the range of practical interest is currently between several hundred and a thousand logical qubits, where FFT-based multiplication needs more gates than simpler techniques.

Our circuits moderately increase qubit counts to significantly decrease gate counts and circuit depth. Built from standard components, they are readily adapted to nearest-neighbor quantum architectures by optimizing these components to each particular architecture [10].

*Address correspondence to: imarkov@eecs.umich.edu

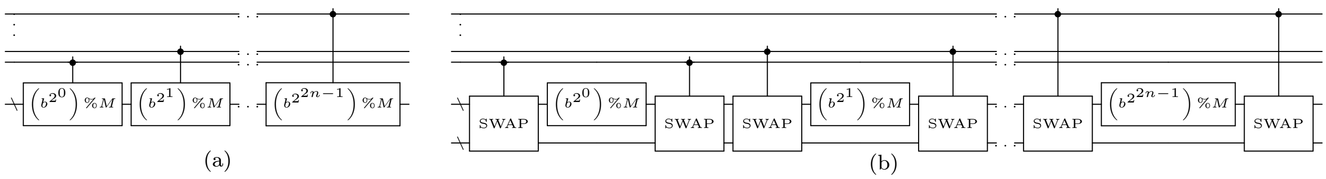

Figure 1: (a) Modular exponentiation using conditional modular multiplications [6]. (b) Conditional multiplications implemented using unmodified unconditional modular multiplication blocks and conditional swaps with a zero register [7].

III. NEW CIRCUITS

We propose two-register $\mathcal{UM}$ circuits to compute ($Cx \mod M$) for coprime C and M, $\mathrm{GCD}(C, M) = 1$. These circuits transform $(x, 0)$ into (x, x) using parallel CNOT gates and then compute $(Cx \mod M, 0)$. Clearing ancillae in the second register allows the next circuit module to use them again. As reversible building blocks, we use *modular addition and subtraction between the two registers* $(a, b) \mapsto (a \pm b \mod M, b)$ and $(a, b) \mapsto (a, a \pm b \mod M)$, as well as circuits from [7] for modular multiplication by two that clear their ancillae $a \mapsto 2a \mod M$.

Our key insight is to use the *coprimality* of C and M (guaranteed in Shor's algorithm) to read off a circuit from the execution trace of an appropriate GCD algorithm. Recall that the Euclidean GCD algorithm for (A, B) proceeds by replacing the larger number A with ($A \mod B$) until the result evaluates to 0. For $C = 13$ and $M = 21$, this produces the Fibonacci sequence $(21, 13)$-$(8, 13)$-$(8, 5)$-$(3, 5)$-$(3, 2)$-$(1, 2)$-$(1, 0)$. For convenience, one may consider the last configuration to be $(1, 1)$, so that each step performs a subtraction — a simpler operation than modular reduction. As a result, we obtain $\mathrm{GCD}(C, M) = 1$. Reversing the order of operations, interpreting each number as a multiple of x starting with $(1x, 1x)$, and mapping each step into a mod-21 addition, we obtain (x, x)-$(2x, x)$-$(2x, 3x)$-$(5x, 3x)$-$(5x, 8x)$-$(13x, 8x)$-$(13x, 21x) = (13x, 0)$. Since $21x \mod 21 = 0$, the second register is restored to 0. This $\mathcal{UM}$ circuit bypasses Bennett's construction based on modular inverses (Section II) and is smaller than prior art [1, 6].

Unfortunately, some modular reductions in the Euclidean GCD algorithm may require a large number of gates. Consider ($11x \mod 21$) and its Euclidean GCD trace $(21, 11)$-$(10, 11)$-$(10, 1)$-$(1, 1)$. Implementing the last mod operation by nine subtractions produces a sequence of nine mod-21 additions (x, x)-$(2x, x)$-$(3x, x)$-$\ldots$-$(10x, x)$. To improve efficiency, we replace the Euclidean GCD algorithm by a binary GCD algorithm that avoids the mod operation and uses a shortcut for the case of odd GCD. Given a pair of odd numbers, the larger one is replaced by their difference, which must be even. Any even number is divided by two, which can be implemented by a controlled bit-shift (as shown in [7]).

For even A and B, $(A, B) = (A/2, B/2)$
For even A and odd B, $(A, B) = (A/2, B)$
For odd A and even B, $(A, B) = (A, B/2)$
For odd A and B, if $A < B$, $(A, B) = (A, B - A)$
else $(A, B) = (A - B, B)$

One stops when $A{=}B{=}\mathrm{GCD}{=}1$ (assuming coprime inputs). The sequence of operations performed for our example $(21, 11)$-$(10, 11)$-$(5, 11)$-$(5, 6)$-$(5, 3)$-$(2, 3)$-$(1, 3)$-$(1, 2)$-$(1, 1)$ can be improved by $(2, 3)$-$(2, 1)$-$(1, 1)$. To obtain a circuit, such sequences are reversed and interpreted as modular multiplications by two and modular additions, with the initial state $(1x, 1x)$. Further improvements are obtained by allowing both subtractions and additions, e.g., $15x = 16x - x$ versus $15x = 8x + 4x + 2x + 1x$ (here $16x$, $8x$, etc are computed by doubling).

In a more involved example ($7x \mod 1017$), the addition leading to $(7, 1024)$ is a better first step than the subtraction leading to $(7, 1010)$, because $(7, 1024)$ enables eight successive divisions by two which reduce the values down to $(7, 4)$ faster than subtractions would. Then, subtractions become the best operators: $(3, 4)$-$(3, 1)$-$(2, 1)$-$(1, 1)$. This optimization relies on a three-step lookahead. To select each next operator, we consider all possible irredundant three-step sequences of operators (modular addition, subtraction and division by two), find their final states, and score the remaining circuit according to the trace of the binary GCD algorithm (without lookahead). The cost of each operator/step can be specific to the quantum machine. Taking the best three-step sequence, we commit to its first operator. The remaining two steps are ignored, and the next operator is chosen by a separate round of lookahead. For ($11x \mod 21$), we obtain $(21, 11)$-$(10, 11)$-$(5, 11)$-$(5, 6)$-$(5, 1)$-$(4, 1)$-$(2, 1)$-$(1, 1)$.

IV. EMPIRICAL VALIDATION

Our algorithms for on-demand construction of modular multiplication circuits [17] were embedded into the framework of Figure 1. The number of ancillae in resulting mod-exp circuits was $5n + 2$ (as in [7]), but several optimizations from [7] were not used, and the number of mod-mult blocks was exactly as in [6]. Our software was written in C++ using the GNU MP library (for multi-precision arithmetic) supplied with the GCC 4.6.3 compiler on Linux. We used a workstation with an Intel® Core™ 2 Duo 2.2 GHz CPU and 2 GB Memory.

To evaluate our optimizations of Shor's number-factoring algorithm, we studied all odd n-bit semiprime values of the modulus ($M = pq$) for $7 \leq n \leq 15$, and a subset of n-bit M values for n =16–512 that are products of the 1st and 10th largest $n/2$-bit primes. Circuit sizes for $n < 16$ were averaged over all M-coprime C values. Results for $n = 16$ include all coprime C values for the given M. For 24-, 32-, 48-, and 64-bit M val-

Table I: Circuits produced by our technique and prior art, compared by Toffoli gate counts. Circuit sizes for $n < 16$ are averaged over all M-coprime C values. Results for $n = 16$ include all coprime C values for the given M. For 24-, 32-, 48-, and 64-bit M values, results are averaged over the first 5000 coprime C values. For larger n values in mod-mult circuits, only $C = -1/17$ mod M (e.g., 47679095568306588235294117647058823529411764705882352941176470588235294117647 for $n = 256$) are shown. For modular exponentiation, results include all C values appearing in $\mathcal{UM}$ blocks for $b = 2$. All results reported are circuit sizes (Toffoli gate counts), except for values in the *depth* column. For 'Avg ratio' in mod-mult, we used Ours/[7] and [6]/Ours.

Bits	# of semiprimes	Modular Multiplication (circuit size)								Modular Exponentiation			
		Optimal [7]		Ours		Beckman [6]		Avg ratio		Ours		Beckman	Avg ratio
n	[smallest, largest]	max	avg	max	avg	max	avg	/[7]	[6]/	#Gates	Depth	[6]	[6]/Ours
7	7 in [65, 119]	182	**134.3**	210	**138.3**	1240	**852**	**1.03**	**6.16**	**1292**	**1083**	11154	**8.63**
8	16 in [133, 253]	257	**194.3**	292	**202.8**	1700	**1162**	**1.04**	**5.73**	**2288**	**2120**	17415	**7.61**
9	34 in [259, 511]	326	**258.0**	386	**271.3**	2232	**1520**	**1.05**	**5.60**	**3731**	**3059**	25670	**6.88**
10	72 in [515, 1007]	418	**327.3**	481	**347.6**	2836	**1926**	**1.06**	**5.54**	**5286**	**3885**	36195	**6.84**
11	152 in [1027, 2047]	518	**405.0**	626	**434.5**	3512	**2380**	**1.07**	**5.48**	**7447**	**4959**	49266	**6.61**
12	299 in [2051, 4087]	635	**488.8**	765	**523.8**	4260	**2882**	**1.07**	**5.50**	**10002**	**6075**	65159	**6.51**
13	621 in [4097, 8189]	750	**580.3**	930	**627.3**	5080	**3432**	**1.08**	**5.47**	**13364**	**7472**	84150	**6.29**
14	1212 in [8197, 16379]	882	**678.6**	1120	**738.9**	5972	**4030**	**1.08**	**5.45**	**16854**	**8617**	106515	**6.32**
15	2429 in [16387, 32765]	-	-	1340	**868.0**	6936	**4676**	-	**5.39**	**21523**	**9985**	132530	**6.16**
16	$M = (2^8 - 5) \times (2^8 - 59)$	-	-	1425	**990.3**	7972	**5370**	-	**5.42**	**28581**	**15884**	162471	**5.68**
24	$M = (2^{12} - 3) \times (2^{12} - 77)$	-	-	4237	**2705.9**	18852	**12650**	-	**4.67**	**109405**	**39671**	576455	**5.27**
32	$M = (2^{16} - 15) \times (2^{16} - 123)$	-	-	6023	**5024.0**	34340	**23002**	-	**4.57**	**268387**	**86110**	1400679	**5.21**
48	$M = (2^{24} - 3) \times (2^{24} - 167)$	-	-	13447	**11852.4**	79140	**52922**	-	**4.46**	**954662**	**201065**	4845118	**5.07**
64	$M = (2^{32} - 5) \times (2^{32} - 267)$	-	-	24028	**21354.8**	142372	**95130**	-	**4.45**	**2358531**	**422070**	11626238	**4.92**
96	$M = (2^{48} - 59) \times (2^{48} - 257)$	-	-		**46400**	324132	**216410**	-	**4.66**	**8.15e6**	**1.00e6**	3.97e7	**4.87**
128	$M = (2^{64} - 59) \times (2^{64} - 363)$	-	-		**82207**	579620	**386842**	-	**4.71**	**1.97e7**	**2.03e6**	9.47e7	**4.81**
192	$M = (2^{96} - 17) \times (2^{96} - 347)$	-	-		**189327**	1311780	**875162**	-	**4.62**	**6.91e7**	**4.63e6**	3.22e8	**4.65**
256	$M = (2^{128} - 159) \times (2^{128} - 1193)$	-	-		**331126**	2338852	**1560090**	-	**4.71**	**1.65e8**	**9.25e6**	7.64e8	**4.63**
384	$M = (2^{192} - 237) \times (2^{192} - 1143)$	-	-		**746212**	5277732	**3519770**	-	**4.71**	**5.64e8**	**2.09e7**	2.59e9	**4.58**
512	$M = (2^{256} - 189) \times (2^{256} - 1883)$	-	-		**1324289**	9396260	**6265882**	-	**4.73**	**1.34e9**	**4.12e7**	6.15e9	**4.56**

ues, results were averaged over the first 5000 coprime C values. For larger n values in modular multiplication circuits, only $C = -1/17 \mod M$ are shown. Results for modular exponentiation include all C values appearing in unconditional modular multiplication blocks for $b = 2$ (Figure 1). These are $C = b^{2^0}\%M$, $C = b^{2^1}\%M$, $\cdots$, and $C = b^{2^{2n-1}}\%M$. For $n \leq 15$, Table I shows that circuits found by our heuristic are closer to optimal circuits [7] than to scalable circuits from [6]. Beyond the reach of optimal techniques ($n \geq 24$), Figure 2 shows that our circuits are at least 4.5 times smaller and retain their advantage as n increases. Our runtimes ranged from negligible ($n \leq 32$) to 30 min for one 512-bit (M, C) pair.

To compare our circuits with latency(depth)-optimized constructions in [11], we note that the most accurate data in [11] are given for $n = 128$. Our smallest 128-bit mod-exp circuits use 1.97×10^7 Toffoli gates with 642 ancillae. To reduce the latency of our circuits, we replaced linear-depth Cuccaro adders with $\log n$-depth adders from [14] also used in [11]. Accordingly, circuit depth is reduced to 2.03×10^6 Toffoli gates with ~ 900 ancillae. This process is outlined in the next section, but here we summarize the results. A circuit with 660 ancillae [11, Algorithm G, Table II] exhibits latency 1.50×10^7 Toffoli gates. The best circuit in [11, Algorithm E, Table II] has latency 1.71×10^5 Toffoli gates but uses 12657 ancillae, which is far less practical with technology under development today. Circuit depths of our modular exponentiation circuits for all attempted n values are reported in Table I. A quantum machine with only some limited form of parallelism may still benefit from our techniques, given strong results for both parallel and sequential cases.

V. REDUCING CIRCUIT LATENCY

Our circuits can be adapted to quantum architectures with high degree of parallelism by replacing building blocks by parallelized variants. Circuit-size calculations in Table I are based on the costs of circuit modules (addition, subtraction, modular multiplication by 2, etc) from [7, Table 2]. Cuccaro adders used in [7] are small, but exhibit linear latency. To optimize latency for comparisons to [11], we replaced Cuccaro adders with QCLA adders from [14] (also used in [11]) whose depth is $(4\log_2 n + 3,\ 4,\ 2)$ in terms of (T,C,N) gates. As in [11], we measure latency in Toffoli gates. This results in circuit latency (depth) $4\log_2 n + 3$ for additive operators (`~1,~2,+1,+2,-1,-2`) in [7, Table 2]. The oper-

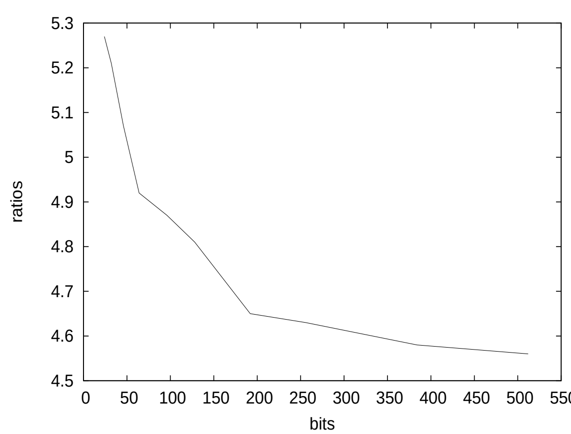

Figure 2: Asymptotic behavior of circuit-size ratios between Beckman et al [6] and our constructions.

ators that perform modular multiplication (d1,d2) and division by two (h1,h2) exhibit latency $6\log_2 n + 12$. To count the number of ancillae in our modular exponentiation circuits, note that QCLA adders from [14] need $2n - \log n - 2$ ancillae (vs. 1 for Cuccaro adders). Given that QCLA adders clear all ancillae, the number of ancillae in our mod-exp circuits grows to $\leq 7n$.

We also restructure n one-bit controlled-SWAP gates with shared control to reduce latency from n to $\log_2 n$. The control bit is temporarily copied to n zero-initialized ancillae (with $\log_2 n$ latency) [15]. We use n parallel one-bit controlled-SWAP gates, and then clear the n ancilla (also with $\log_2 n$ latency). Because these ancillae are cleared immediately, we can share them with the QCLA ancillae. Thus, the overhead is $2\log_2 n$ latency and $2n$ CNOT gates used to copy/clear ancillae.

VI. CONCLUSIONS

The n-bit multiplication circuits developed in this work significantly simplify the implementation of Shor's algorithm, but use $\Theta(n^2)$ gates, as do traditional circuits. Circuit sizes are improved by large constant factors. These factors appear exaggerated for *small qubit arrays* because, for $b = 2$, our construction implements a non-trivial fraction of modular multiplications using $\Theta(n)$ gates using circuit blocks from [7]. In contrast, prior work typically uses generic $\Theta(n^2)$ circuits regardless of b. We have experimented with several enhancements to our technique, but the resulting improvement was not justified by the increased runtime and programming difficulty.

Connections between number factoring and GCD computation were known to Euclid around 300 B.C.E. Today the two problems play similar roles in their respective complexity classes. Number-factoring is in NP (problems whose solutions can be checked in polynomial time), not known to be in P (problems solvable in polynomial time), but is not believed to be NP-complete (most difficult problems in NP). GCD is in P, not known to be in NC (problems that can be solved very efficiently when many parallel processors are available), but is not believed to be P-complete (inherently sequential). Unlike provably-hard problems (such as Boolean satisfiability), or problems for which fast serial and parallel algorithms are known (such as sorting), number factoring and GCD appear to be good candidates for demonstrations of physics-based computing that exploits parallelism.

Our use of GCD algorithms to speed up modular exponentiation and number factoring incurs only small overhead. All the invocations of our GCD-based circuit construction for one run of Shor's algorithm can run in parallel because they are independent. Thus, the classical-computing overhead of our technique for one run of Shor's algorithm is limited to 1-2 GCD-based circuit constructions. This overhead is acceptable because Shor's algorithm performs multiple GCD-like computations after quantum measurement.

Acknowledgments. IM's work was sponsored in part by the Air Force Research Laboratory under agreement FA8750-11-2-0043. Héctor J. García helped with Fig. 1.

---

[1] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information.* Cambridge Univ. Press, 2000.

[2] L.M.K. Vandersypen et al. Experimental realization of an order-finding algorithm with an NMR quantum computer. *Phys. Rev. Lett.*, 85:5452–5455, Dec 2000.

[3] B. P. Lanyon et al. Experimental demonstration of a compiled version of Shor's algorithm with quantum entanglement. *Phys. Rev. Lett.*, 99:250505, Dec 2007.

[4] C.-Y. Lu, D. E. Browne, T. Yang, and J.-W. Pan. Demonstration of a compiled version of Shor's quantum factoring algorithm using photonic qubits. *Phys. Rev. Lett.*, 99:250504, Dec 2007.

[5] A. Politi, J. C. F. Matthews, and J. L. O'Brien. Shor's quantum factoring algorithm on a photonic chip. *Science*, 325(5945):1221, 2009.

[6] D. Beckman, A. N. Chari, S. Devabhaktuni, and J. Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54:1034–1063, Aug 1996.

[7] I. L. Markov, M. Saeedi. Constant-optimized quantum circuits for modular multiplication and exponentiation. *Quantum Info. Comput.*, 12(5-6):361–394, May 2012.

[8] E. Knill. On Shor's quantum factor finding algorithm: Increasing the probability of success and tradeoffs involving the fourier transform modulus. *Tech. Report LAUR-95-3350, Los Alamos Natl. Lab*, Aug 1995.

[9] D. McAnally. A refinement of Shor's algorithm. *arXiv:quant-ph/0112055*, 2002.

[10] A. G. Fowler, S. J. Devitt, and L. C. L. Hollenberg. Implementation of Shor's algorithm on a linear nearest neighbour qubit array. *Quantum Info. Comput.*, 4(4):237–251, July 2004.

[11] R. Van Meter and K. M. Itoh. Fast quantum modular exponentiation. *Phys. Rev. A*, 71:052320, May 2005.

[12] Y. Takahashi, S. Tani, and N. Kunihiro. Quantum addition circuits and unbounded fan-out. *Quant. Inf. Comput.*, 10(9&10):872–890, 2010.

[13] D. Rosenbaum. Optimal quantum circuits for nearest-neighbor architecture. *arXiv:1205.0036*, 2012.

[14] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Quantum Info. Comput.*, 6(4):351–369, July 2006.

[15] C. Moore and M. Nilsson. Parallel quantum computation and quantum codes. *SIAM J. Comput.*, 31(3):799–815, 2002.

[16] Euclid studied number factorization circa 300 B.C.E. as a way to compute GCD, which is required to add fractions. Failing to find a fast algorithm, he developed what is now known as Euclid's GCD algorithm.

[17] A small fraction of C values are positive or negative modular powers of two, or their modular negations. These rare cases are enumerated directly for each M, so that our GCD-based algorithm can skip them.