YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

MASTER THESIS

ACCELERATED MODULAR INVERSE ALGORITHM FOR MULTIDIGIT INTEGERS

PAKİZE ŞANAL

THESIS ADVISOR: ASST. PROF. HÜSEYİN HIŞIL

COMPUTER ENGINEERING

PRESENTATION DATE: 25.07.2019

BORNOVA / İZMİR
JULY 2019

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Jury Members: **Signature:**

Asst. Prof. Serap ŞAHİN, Ph.D.
İzmir Institute of Technology

Asst. Prof. İbrahim ZİNCİR, Ph.D.
Yaşar University

Asst. Prof. Hüseyin HIŞIL, Ph.D.
Yaşar University

Prof. Cüneyt GÜZELİŞ, Ph.D.
Director of the Graduate School

ABSTRACT

ACCELERATED MODULAR INVERSE ALGORITHM FOR MULTIDIGIT INTEGERS

Şanal, Pakize
Msc, Computer Engineering
Advisor: Asst. Prof. Hüseyin HIŞIL

July 2019

In this thesis, a multi-digit modular multiplicative inverse algorithm has been aimed to SIMD parallelized by utilizing AVX2 instructions which are commonly encountered on new generation Intel processors. Euclid’s extended GCD approach is an well known method which also computes modular inverse and GCD together. Binary XGCD algorithms based upon this technique are quite fast in computer architecture since they only use shifting operations instead of multiplication. Generalized version of binary XGCD algorithm was firstly introduced by Lehmer. It reduces the numbers in digit level instead of bits, from left to right which makes the algorithm fast for large numbers. The accelerated GCD algorithm proposed by Jebelean and Weber also realized the same operation in reverse direction; from right to left. Their method has been improved by some other researchers, and eventually became more efficient. In all of these algorithms process Euclid's invariant equations the distinct data in similar way and by same operation, naturally convenient for SIMD parallelization. In this thesis, the modular multiplicative inverse version of this algorithm is developed. The fundamental part of this algorithm has been SIMD parallelized successfully and the sub-functions have been parallelized partially.

Key Words: Greatest Common Divisor (GCD), modular multiplicative inverse, accelerated GCD, Lehmer algorithm, Jebelean-Weber algorithm, multi-digit GCD, Single Instruction Multiple Data (SIMD), Intel Intrinsic, Intel's Advanced Vector Extensions 2 (AVX2).

ÖZ

ÇOK BASAMAKLI SAYILAR İÇİN HIZLANDIRILMIŞ MODÜLER TERS ALMA ALGORİTMASI

Şanal, Pakize
Yüksek Lisans Tezi, Bilgisayar Mühendisliği
Danışman: Yrd.Doç. Dr. Hüseyin HIŞIL, Ph.D.
Temmuz 2019

Bu tez, yeni model Intel işlemciler üzerinde bulunan AVX2 yönergeleri kullanılarak sağdan sola çok basamaklı küçültme yöntemiyle uygulanan modüler çarpımsal ters alma hesaplamasını SIMD paralel şekilde geliştirilmesini amaçlamaktadır. Euclid in genişletilmiş GCD metodu hem GCD yi hem de modüler ters almayı hesaplayan iyi bilinen bir yöntemdir. Bu yöntemle yazılan binary XGCD algoritmaları, çarpma operasyonu yerine kaydırma operasyonu kullandığı için bilgisayar mimarisinde hızlı algoritmalardır. Binary XGCD algoritmasının genelleştirilmiş hali, ilk kez Lehmer tarafından yazılmıştır. Bu algoritma, sayıları bit seyivesi yerine soldan sağa basamak seviyesinde küçültür, bu da algoritmayı büyük sayılar için hızlı bir yöntem haline getirir. Jebelean ve Weber tarafından sunulan genelleştirilmiş GCD algoritması da aynı işlemi tersten sağdan sola gerçekleştirmektedir. Bu method ise zaman içerisinde farklı araştırmacılar tarafından geliştirilmiş ve sonunda daha etkili hale getirilmiştir. Tüm bu algoritmalar, Euclid in invaryant denklemlerini birbirinden bağımsız ama benzer şekilde ve aynı operasyonlarla işlemektedir, bu da SIMD paralelleştirme için oldukça uygundur. Bu tezde, bu algoritmanın modular çarpımsal ters alma versiyonu geliştirildi. Bu algoritmanın ana döngüsü başarılı bir şekilde SIMD paralel hale getirildi ve alt fonksiyonlar kısmen paralelleştirildi.

Anahtar Kelimeler: En büyük ortak bölen (GCD), modüler çarpımsal ters, hızlandırılmış GCD, Lehmer algoritması, Jebelean-Weber algoritması, çok basamaklı GCD, (Tek komut çoklu veri) SIMD, Intel Intrinsic, İntel’in Gelişmiş Vektör Uzantıları 2 (AVX2).

ACKNOWLEDGEMENTS

Firstly, I would like to state my profound appreciation to my advisor, Dr. Hüseyin Hışıl for his precious and constructive ideas during the formation and progress of this research work. He has been encouraging since the days I began as an undergraduate student. He has always been an understanding and patient supervisor for both my academic and personal life. By his true leading and guidance, I overcame numerous challenges during this study. Without his back-up and mentorship, it would not be possible for me to accomplish my goal.

I am thankful to association committees of ECC2017, ECC2018, CHES2018, Summer School on Real-World Crypto and Privacy. They supported me with scholarships to attend their events. Their assistance is irrevocably significant for the way that I would like to run through. I'm also grateful to Peter Schwabe who led me to participate in these nice events and provide me good opportunities to meet new students and professionals in this area coming from around the world.

My sincere thanks go to my office mates at Yaşar University for creating such a great working environment, for the sleepless nights we worked together before deadlines, and for all the fun we have had in the last two years, especially on birthdays.

A lot of acknowledgments to Özge Erten, who was a true friend ever since we started high school, for almost 12 years.

Last but not least, I would like to thank my family in particular to my mother who always supported me while writing this thesis with her high motivation and with her pioneering ideas in my life.

Pakize Şanal

İzmir, 2019

TEXT OF OATH

I declare and honestly confirm that my study, titled "ACCELERATED MODULAR INVERSE ALGORITHM FOR MULTIDIGIT INTEGERS" and presented as a Master's Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Pakize Şanal

July 25, 2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODES

CHAPTER 1

INTRODUCTION

Several number theoretic constructions makes frequent reference to greatest common divisors (GCD) or related primitives such as Bezout identity or modular inverses as subroutine. Typical examples include,

1. Number theoretic functions: Basis of a two dimensional lattice, finite fields, Groebner basis theory.
2. Cryptographic functions: Elliptic curve cryptography, lattice based cryptography, post quantum cryptography.
3. Cryptanalytic functions: Number field sieve algorithm, index calculus algorithm, Pollard's rho algorithm, Shank's baby step giant step algorithm.

Since all of these subroutines are computed on binary computers a typical question is to optimize GCD related computations.

As the clock speed of modern processors got close to its foreseeable physical limit on the current semi-conductor based transistors, a rather old hardware trend started to gain more attraction from hardware vendors i.e. manufacturing single instruction multiple data (SIMD) instruction sets. New processors are devoting a larger die area for these type of instruction sets. This is a limited yet powerful way of parallel processing. For example, `vpmuludq` instruction can accommodate four 32x32→64 bit unsigned integer multiplications. The same processor can do only a single 64x64→128 bit multiplication on its `amd64` integer circuit. The computational capabilities of such an instruction set can be highly exploited in software if the underlying computation is suitable for SIMD processing.

This thesis is a study of reviewing existing k-ary GCD based algorithms and investigate their suitability to AVX2 programming. In particular, we concentrate on a variant which was developed with accumulative results by Jebelean (Jebelean, 1993), Weber (Weber, 1995), Sorenson (Sorenson, 2004), and Sedjelmaci (Sedjelmaci, 2007). We call this algorithm as the JWSS algorithm in this work.

1.1 MOTIVATION

While developing and implementing a number theoretic function, oftentimes there are two main concerns in mind,

i the function can be computed in finite time and memory.

ii having **i** satisfied, it would be very beneficial to compute efficiently.

One motivation of this thesis comes from computing GCD sequences and other related operations such as modular inverses in above mentioned fashion. Another motivation comes from low level parallelization of such computations to utilize the underlying hardware at its peak. In particular, single instruction multiple data (SIMD) support is an important feature of modern microprocessors and is preferable in some implementations of cryptographic primitives such as Montgomery and Genus-1&2 Kummer ladders, cf. (Bernstein, 2006), (Chou, 2015), (Bernstein et al., 2014), and (Karati and Sarkar, 2017). Such implementations produce higher throughput in comparison to alternative implementations using the 64 bit integer circuit. The key feature of the success behind these implementations comes from the fact that ladder formulas can be put in SIMD friendly form. A similar situation seems to be satisfied in k-ary GCD algorithms given in Chapters 2 and 3. However, it is not clear whether a SIMD implementation of these algorithms can provide any practical speed-up. It is not even clear whether these algorithms can be realized at all in a SIMD fashion. For instance, some GCD algorithms require integer division instruction but not all SIMD platforms provide such an option. The most widely available SIMD circuit, Intel's AVX2, is one example of this class. Therefore, a SIMD implementer has to overcome such inabilities. Yet, the linear transformation phase of some other algorithms seems to be SIMD friendly. On the other hand, no publicly available implementation is known to date in this context. These unknowns also provide motivation to this thesis.

Intel introduced SIMD extension, MMX in the Pentium processor 1993, SSE in Pentium III 1999 and then AVX in Sandy Bridge 2008. Intel AVX is 256 bit instruction set extension, twice the number of data elements that SSE can process with a single instruction and four times that of MMX, has enhanced performance with longer vectors, new extensible syntax, and rich functionality. It is later extension, AVX2 was released in 2013 as a superior of AVX. Recently, Intel AVX-512 was announced that available on the latest Xeon and i9 processors. This fast development shows that the progress on SIMD is inevitable. Furthermore, demanding technological development on Intel Intrinsics is easy to implement suitable parallel algorithms in C language syntax. Having these in mind, SIMD instructions have potential to boost the performance. This is mostly related with how much the algorithm is suitable for SIMD parallelization. There are cases where such a parallelization is not even possible. Therefore, a deeper research is needed to test k-ary GCD algorithms in this context.

In summary, the main motivation of this thesis is to determine whether Intel's AVX2 instruction sets can be preferable in the implementation of these algorithms over the 64 bit integer circuit. The expected outcome is to determine whether one can obtain a better throughput in modular inverse computations based on GCD sequences.

1.2 AIMS & OUTCOMES

The main objective of this thesis is to do research on SIMD implementation of JWSS algorithm and its variants for computing multidigit modular multiplicative inverse. The target hardware is widely available AVX2 on i3, i5, and i7 series The programming environment is built on C language and Intel Intrinsics library.

Parallel implementation of the selected sequential algorithm is not a simple task. Because it requires investigating the best combination of instructions by considering many concerns simultaneously. This can be provided by maximizing the range of options, reveal the necessary actions and reducing the bad choices to achieve the best performance. In order to achieve this goal we determined the following aims for this work:

- Perform a literature review on available algorithms to compute GCD and modular multiplicative inverse.
- Modify JWSS algorithm to produce an extended GCD sequence. The extended GCD sequence can then be simplified to produce Bezout's identity, modular inverse or simply GCD.
- Identify parts of JWSS algorithm that are suitable for SIMD programming.
- Determine hard-to-parallelize parts and develop efficient solutions/variations.
- Define the representation of the multidigit data with having in mind the limitations of Intel AVX2 instructions.
- Implement the selected algorithm(s) with a high level programming language. This language is Magma in our case.

- Implement SIMD version of the algorithm on AVX2 platforms reflecting the Magma implementation.

- Measure the performance of several trials made. And then make a comparison to determine the best strategy.

We obtained the following outcomes for this work:

- The k-ary style GCD algorithms are understood to be SIMD friendly leaving a very small room for 2-ary algorithms on very small inputs. k-ary GCD algorithms make some processing on a small portion of inputs and then perform linear transformations to get rid of several bits at once. Two classic approaches are left-to-right and right-to-left elimination of digits. We selected the JWSS algorithm, a right-to-left method, to implement after suitable modifications. Details are given in Chapters 2 and 3.

- A magma code is developed to satisfy the JWSS algorithm and our modifications to be explained.

- A C/assembly SIMD implementation of the JWSS algorithms is developed. This code showed that computation of GCD sequences can efficiently benefit from widely available AVX2 SIMD instruction sets.

These aims and outcomes brought us to implementation oriented contributions which are provided in Section 1.3.

1.3 CONTRIBUTIONS

Building on the aforementioned aims and outcomes, this work makes the following contributions:

- Extended GCD adaptation of JWSS algorithm is proposed with minor modifications for SIMD friendly implementation.

- The data permutation is costly on both AVX2 platforms. We show how to eliminate all permutations despite the fact that SIMD lanes need intercommunication. This allows a faster SIMD implementation of the extended JWSS algorithm and of its variants. We provide a discussion of how to represent data in order to get optimal performance.

- We provide the first AVX2 implementations of the variable-time modular inversion algorithm based on our extended JWSS algorithm.

These contributions will provide implementers a wide angle of decision alternatives when implementing a k-ary GCD algorithm in a SIMD platform. Our reported experiences are expected to be very useful if the trend in SIMD hardware support continues its progression.

1.4 OUTLINE

This master of science thesis is organized as follows. Chapter 2 provides a literature review of selected algorithms in the context of aims of thesis work. This chapter also provides extended GCD adaptations of both Lehmer and JWSS algorithms. Chapter 3 provides the modular inverse variant of the extended GCD algorithm and provides modifications tailored towards SIMD implementation. Chapter 4 provides details on Magma and C/assembly implementations of JWSS algorithms. Conclusions and future research directions are given in Chapter 5.

CHAPTER 2

BACKGROUND ON K-ARY GCD ALGORITHMS

There are several algorithms to compute the GCD of two inputs. These inputs can be integers, polynomials over integers, or elements of some Euclidean domain. This thesis focuses on integer inputs. On the other hand, one method developed for integer inputs can oftentimes be applied analogously for other mathematical objects.

The classical Euclidean algorithm with division step has quadratic (Knuth, 2014) time complexity. This algorithm can be applied on processors with integer division instruction efficiently. The bits are processed from left to right in Euclidean algorithm. Another approach is Stein's algorithm. This algorithm processes the bits from right to left and the complexity of the algorithm is again of quadratic time, (Stein, 1967). Asymptotically faster GCD algorithms exist. For instance, see (Knuth, 1971), (Schönhage, 1971), (Stehlé and Zimmermann, 2004), and (Möller, 2008). However, the take over input sizes for such algorithms are not in the context of this thesis work and thus omitted hereafter.

Both Euclid and Stein type algorithms underwent several modifications allowing faster software and hardware realizations. Historically most important achievements can noted as Lehmer's and Sorenson's generalizations.

Lehmer's algorithm, which is in the left-to-right category of GCD algorithms, simulates the consecutive division steps of Euclidean GCD on most significant part of the inputs and then jumps the intermediate steps with the help of a linear transformation step. This linear transformation can be implemented very efficiently with a fast signed integer multiplier. Most modern processors support this feature. The main loop of Lehmer's algorithm eliminates roughly one word of each input in every iteration. Lehmer's algorithm is therefore very suitable for processors with fast multiplication and division circuits.

Sorenson's k-ary algorithm (Sorenson, 1994) can be viewed as the right-to-left adaptation of Lehmer's approach. This algorithm can also be viewed as the generalization of Stein's binary GCD algorithm. Sorenson described how jumps from right to left can be achieved via linear transformations but did not give an explicit algorithm explaining how to compute auxiliary constants needed by the linear transformation. Sorenson proves that such constants exist and suggest to look up from a table. Jebelean (Jebelean, 1993) and Weber (Weber, 1995) independently found how to compute the missing auxiliary constants via an Euclidean type algorithm. Jebelean and Weber's variant was implemented and used for a long time in GMP library. One drawback of Jebelean and Weber's variant is that the linear transformations have a potential to introduce spurious factor in the results. Such spurious factor can be eliminated with a final fast GCD step. Sorenson later showed how to prevent such spurious factors with a closer analogy to Lehmer's method. In 2007, Sedjelmaci provide an explicit algorithm for computing GCD using Sorenson's approach, see (Sedjelmaci, 2007). We call Sedjelmaci's variant as JWSS k-ary GCD algorithm.

The latest developments on GCD sequences concentrated more on developing a constant-time yet efficient GCD sequence. The first attempt based on Kaliski's variant was proposed by Bos (Bos, 2014). Very recently, possibly a case closing solution came from Bernstein and Yang in (Bernstein and Yang, 2019). Bernstein and Yang developed a new rule set for the computing a left-to-right k-ary GCD sequence which eliminates several irregularities suffered in both Lehmer and JWSS type variants, which are long right shifts, long zero checks, long divisions, and long conditional swaps at the expense of doing more iterations on the outer loop. We refer to (Bernstein and Yang, 2019) for BY algorithm.

The following sections briefly summarize Lehmer and JWSS variants. The section provides more details than the original ones appeared in the literature. In particular, the presented work in this thesis extends these algorithms in the context of extended GCD algorithms so that outputs satisfies invariant equations throughout the computation, coming from Bezout's identity.

2.1 LEHMER'S LEFT TO RIGHT K-ARY GCD SEQUENCE

Lehmer's algorithm is an alternative approach to Euclid's algorithm which eliminate expensive long divisions (Lehmer, 1938). At each iteration of the main loop, the algorithm produces four auxiliary single digit signed integer values with respect to the the high-order digit of x, y where y could be 0 but not x, see (Katz et al., 1996). These auxiliary values are then used to jump several steps through the classical Euclidean algorithm. In particular, the auxiliary values are used to apply linear transformations as given Algorithms 2 to reduce the size of x and y from left to right. If the least significant digit of the smaller number is zero, the algorithm makes a larger jump through long division.

Algorithm 1: `AuxiliaryCoefficients`

input : Integers $\bar{x}$ and $\bar{y}$ with $\bar{x}$ has β bits.

output: Auxiliary values for Algorithm 2

$A, B, C, D \longleftarrow 1, 0, 0, 1$
while $(\bar{y} + C) \neq 0$ *and* $(\bar{y} + D) \neq 0$ **do**
    $q, q^{'} \longleftarrow \lfloor (\bar{x} + A)/(\bar{y} + C) \rfloor, \lfloor (\bar{x} + B)/(\bar{y} + D) \rfloor$
    if $q \neq q^{'}$ **then**
        Return A, B, C, D
    else
        $A, C \longleftarrow C, A - qC$
        $B, D \longleftarrow D, B - qD$
        $\bar{x}, \bar{y} \longleftarrow \bar{y}, \bar{x} - q\bar{y}$
    end
end
Return A, B, C, D

The original algorithm of Lehmer computes GCD only. We provide an extended version in Algorithm 2.

Algorithm 2: Lehmer's Algorithm

input : two positive integers x and y in radix β representation, with $x \geq y$.

output: $\gcd(x, y), x', y'$ satisfying $x \cdot x' + y \cdot y' = \gcd(x, y)$.

$x^{'} = 1$, $y^{'} = 0$, $x^{''} = 0$, $y^{''} = 1$
while $y > 0$ **do**
    Set $\bar{x}$, $\bar{y}$ to be the high-order digit of x, y, respectively (y could be 0).
    $A, B, C, D \leftarrow$ `AuxiliaryCoefficients`$(\bar{x}, \bar{y})$
    if $B = 0$ **then**
        $q \leftarrow x/y$
        $x, y \leftarrow y, x - q \cdot y$
        $x^{'}, y' \leftarrow y', x^{'} - q \cdot y'$
        $x^{''}, y'' \leftarrow y'', x^{''} - q \cdot y''$
    else
        $x, y \leftarrow A \cdot x + B \cdot y, C \cdot x + D \cdot y$
        $x^{'}, y' \leftarrow A \cdot x^{'} + B \cdot y^{'}, C \cdot x^{'} + D \cdot y^{'}$
        $x^{''}, y'' \leftarrow A \cdot x^{''} + B \cdot y^{''}, C \cdot x^{''} + D \cdot y^{''}$
    end
end
return x, x', y'.

Let $x_0, y_0, x, y, x^{'}, y^{'}, x^{''}, y^{''} \in \mathbb{Z}$ satisfy the invariant equations

$$x_0 x^{'} + y_0 y^{'} = x \text{ and } x_0 x^{''} + y_0 y^{''} = y.$$

These equations are still satisfied after every linear transformations on x, y, $x^{'}$, $y^{'}$, $x^{''}$, $y^{''}$;

$$\begin{aligned} x &\leftarrow ax + by, & y &\leftarrow cx + cy, \\ x^{'} &\leftarrow ax^{'} + bx^{''}, & y^{'} &\leftarrow ay^{'} + by^{''}, \\ x^{''} &\leftarrow cx^{'} + dx^{''}, & y^{''} &\leftarrow cy^{'} + dy^{''}. \end{aligned}$$

To see this, observe that the initial values $x^{'} = 1$, $y^{'} = 0$, $x^{''} = 0$, $y^{''} = 1$ trivially satisfy the equations above. Now, for arbitrary values of $a, b, c, d, x^{'}, y^{'}, x^{''}, y^{''}$ in the sequence of Algorithm 2, we get

$$
\begin{aligned}
ax_0x^{'} + ay_0y^{'} &= ax, \\
bx_0x^{''} + by_0y^{''} &= by.
\end{aligned}
$$

which can be rewritten as

$$
\begin{aligned}
x_0(ax^{'} + bx^{''}) + y_0(ay^{'} + by^{''}) &= ax + by \\
x_0(cx^{'} + dx^{''}) + y_0(cy^{'} + dy^{''}) &= cx + dy.
\end{aligned}
$$

It is possible to write a complete proof based on induction from this observation. We recover the invariant equation once the updates on $x, y, x^{'}, y^{'}, x^{''}, y^{''}$ are performed. Similarly, we rewrite for the special case $B = 0$,

$$
\begin{aligned}
x_0x^{'} + y_0y^{'} &= x, \\
qx_0x^{''} + qy_0y^{''} &= qy.
\end{aligned}
$$

in the form

$$
\begin{aligned}
x_0(x^{'} - qx^{''}) + y_0(y^{'} - qy^{''}) &= x - qy, \\
x_0(x^{''}) + y_0(y^{''}) &= y
\end{aligned}
$$

recover the invariant equation once more.

Figure 2.1 depicts the extended GCD sequence computed with extended Lehmer sequence using Algorithm 2. For comparison, Figure 2.2 repeats the same for identical inputs with extended Euclidean algorithm, see Appendix A. It can be observed that every line in Figure 2 appears in at some place in Figure 2.2, while Lehmer is noticeably shorter. The speedup gained with Lehmer's approach (over Euclidean GCD) is constant. On the other hand, Lehmer's algorithm is still of quadratic time complexity.

Figure 2.1 Lehmer's k-ary GCD illustration for $k = 2^8$

x		y	
18914144994474109809	1	20860527183790487785	0
252325405442301828	-43	45026087805066295	75
8467276359221531	1404	896568669652880	-2315
398158332345611	22239	100252004961658	-46793
2849687501021	-209411	512942425923	7282592
227967054517	43904963	57008316889	-80527334
56942103850	285486965	66213039	-366014299
66213039	-366014299	65103349	314691769806
741329	18588043247896	368361	-18903101032001
368361	-18903101032001	4607	56394245311898
4607	56394245311898	4408	-4474048480671943
199	4530442725983841	30	-104143788452316445
1	-7652331427004922736	1	13208195756785565049
1	13208195756785565049	0	-20860527183790487785
	$x^{'}$		$y^{'}$

It can be noted that larger values of k makes the sequence even shorter. The optimal choice for k depends heavily on the target hardware. For instance, a typical choice for k on an 64-bit processor is 62. One bit is preserved for sign management and another for possible carry bit generated by the addition part of the linear transformations in Algorithm 2.

Figure 2.2 Extended Euclidean GCD illustration

x		y	
18914144994474109809	1	20860527183790487785	0
20860527183790487785	0	18914144994474109809	1
18914144994474109809	1	1946382189316377976	0
1946382189316377976	-1	1396705290626708025	1
1396705290626708025	10	549676898689669951	-9
549676898689669951	-11	297351493247368123	10
297351493247368123	32	252325405442301828	-29
252325405442301828	-43	45026087805066295	39
45026087805066295	75	27194966416970353	-68
27194966416970353	-418	17831121388095942	379
17831121388095942	493	9363845028874411	-447
9363845028874411	-911	8467276359221531	826
8467276359221531	1404	896568669652880	-1273
896568669652880	-2315	398158332345611	2099
398158332345611	22239	100252004961658	-20164
100252004961658	-46793	97402317460637	42427
97402317460637	162618	2849687501021	-147445
2849687501021	-209411	512942425923	189872
512942425923	7282592	284975371406	-6603093
284975371406	-36622371	227967054517	33205337
227967054517	43904963	57008316889	-39808430
57008316889	-80527334	56942103850	73013767
56942103850	285486965	66213039	-258849731
66213039	-366014299	65103349	331863498
65103349	314691769806	1109690	-285329594513
1109690	-315057784105	741329	285661458011
741329	18588043247896	368361	-16853694159151
368361	-18903101032001	4607	17139355617162
4607	56394245311898	4408	-51132405393475
4408	-4474048480671943	199	4056599381701687
199	4530442725983841	30	-4107731787095162
30	-104143788452316445	19	94426698697795251
19	629393173439882511	11	-570667923973866668
11	-733536961892198956	8	665094622671661919
8	1362930135332081467	3	-1235762546645528587
3	-2096467097224280423	2	1900857169317190506
2	5555864329780642313	1	-5037476885279909599
1	-7652331427004922736	0	6938334054597100105
	x'		y'

The linear transformations in the case $B \neq 0$ seems to be SIMD friendly since all multiplications can be computed in parallel. Unfortunately, the case $B = 0$ is not. There is no obvious way of making the long integer division SIMD compatible. Even worse, eliminating the case $B = 0$ does not seems to be possible. Therefore, our conclusion is that Lehmer's algorithm cannot put nicely into SIMD parallel form. An implementer can of course insist on using SIMD features in implementing the algorithm by using non-SIMD instructions for the rare case $B = 0$. On the other hand, this would make the code hard develop and sacrifice the code readability.

The next section discusses a right-to-left method which has a similar disadvantage as in Lehmer's algorithm. On the other hand, the situation can be remedied by removing the long division step, namely `dmod`. The details are provided in the following section.

2.2 JWSS RIGHT TO LEFT K-ARY GCD SEQUENCE

The algorithm of Lehmer is oftenly used in GCD calculation of large numbers which is also encountered in older versions of GNU-GMP library. While Lehmer's algorithm works in left to right fashion, JWSS method works in opposite direction. The first explicit algorithm in this direction was proposed Jebelean and

Weber independently in early 1990s (Jebelean, 1993), (Weber, 1995). Jebelean proposed the mathematical background of this problem whereas Weber handled this matter in a programmatic way. The main loop of Jebelean and Weber's algorithm has potential to produce spurious factors which are handled separately. Sorenson (Sorenson, 2004) describes a modification that prevents spurious factors from appearing. Sedjelmaci (Sedjelmaci, 2007) contributed to Sorenson's idea by decreasing the running time of the algorithm and by making complexity analysis easier. This is the reason that we call Sedjelmaci's version as JWSS method. All these aforementioned papers made significant contribution to the basis of this thesis work. Considering that the original algorithm proposed by Weber represents the idea in a more generic way, his notation will be used in the following parts. Algorithm 3 recalls Weber's version.

Algorithm 3: Accelerated GCD Algorithm

input : $u_0, v_0 > 0$, with $\ell_\beta(u_0) \geq \ell_\beta(v_0)$ and $\texttt{gcd}(u_0, \beta) = \texttt{gcd}(v_0, \beta) = 1$.

output: $\texttt{gcd}(u_0, v_0)$

$u \leftarrow u_0$, $v \leftarrow v_0$
while $v \neq 0$ **do**
    if $\ell_\beta(u) - \ell_\beta(v) > s(v)$ **then**
        $u \leftarrow \texttt{dmod}(u, v, \beta)$
    else
        $(n, d) \leftarrow \texttt{ReducedRatMod}(u, v, \beta^{2t(v)})$
        $u \leftarrow |nv - du| / \beta^{2t(v)}$
    end
    $\texttt{RemoveFactors}(u, \beta)$
    $\texttt{swap}(u, v)$
end
$x \leftarrow \texttt{gcd}(\texttt{dmod}(v_0, u, \beta), u)$
return $\texttt{gcd}(\texttt{dmod}(u_0, x, \beta), x)$.

A toy example is provided below for t equals 2^{16}. Let $u = 230073838367939094855$ and $v = 152188744061051876535$. Writing the numbers in radix 2^{16} we have,

$$u = 12 \cdot (2^{16})^4 + 30954 \cdot (2^{16})^3 + 30979 \cdot (2^{16})^2 + 8101 \cdot (2^{16})^1 + 59719 \cdot (2^{16})^0$$
$$v = 8 \cdot (2^{16})^4 + 16395 \cdot (2^{16})^3 + 2148 \cdot (2^{16})^2 + 39894 \cdot (2^{16})^1 + 13495 \cdot (2^{16})^0$$

which can be succinctly summarized with the following sequences,

$$U = [12, 30954, 30979, 8101, 59719],$$
$$V = [8, 16395, 2148, 39894, 13495]$$

In the first iteration, `ReducedRatMod` operation is calculated with two the least significant digits of the numbers u and v,

$$[n, d] = [40267, 27899] \leftarrow \texttt{ReducedRatMod}([8101, 59719], [39894, 13495])$$

which satisfy the equality $nv - du \equiv 0 \pmod{2^{2\times 16}}$, and thus,

$$40267 \cdot (39894 \cdot 2^{16} + 13495) - 27899 \cdot (8101 \cdot 2^{16} + 59719) \equiv 0 \pmod{2^{32}}$$

Then, u is assigned the value $nv - du$ which clears away at least one lower digit of the updated u, by construction. Now also clearing away factors of 2 from u we get,

$$U = [7877, 63688, 26415].$$

The values appearing in this step together with the other steps are enumerated in Table 1.

Table 1 Example of Accelerated GCD Algorithm

Step	u	v	$[n, d]$
1	$[12, 30954, 30979, 8101, 59719]$	$[8, 16395, 2148, 39894, 13495]$	$[40267, 27899]$
2	$[8, 16395, 2148, 39894, 13495]$	$[7877, 63688, 26415]$	$[34141, 40069]$
3	$[7877, 63688, 26415]$	$[20660, 65261, 2609]$	$[43805, -7421]$
4	$[20660, 65261, 2609]$	$[7351, 3539]$	$[9520, 19344]$
5	$[7351, 3539]$	$[6098, 26707]$	$[34324, 60436]$
6	$[6098, 26707]$	$[3585]$	$[12389, -20633]$
7	$[3585]$	$[15]$	$[239, 1]$
-	$[15]$	$[0]$	

In each step in the Table 1, new value of u is calculated, factors of 2 are removed, and result is swapped with v. In the last step, when the number represented in the v variable is 0, the GCD value is the number represented by the u variable. The value of the GCD for the given example is 15.

The main part of Weber's study is shown in Algorithm 3 and he named his work as "Accelerated GCD Algorithm". Besides, the algorithm has two auxiliary parts, namely `dmod` and `reducedRatMod`.

The following conditions must be satisfied in order to utilize this algorithm and kept during the loop,

1. u and v must be positive.

2. u must be greater then v.

3. v and β must be relatively prime.

The initial value of u being relatively prime with β, is a result of condition 2 and 3 written above.

Else condition is the most significant part of this work which reduces the number u fairly quickly with respect to other algorithms. Herein, special (n, d) values are produced by `reducedRatMod` function. The updated u with at least two least significant digits 0, is obtained by special (n, d) values. Even though the cropping operation has been realized in least two significant digits, the size of the operands are trimmed around t bits.

An "if condition" is used when the difference between u and v is large and it decreases the distance between operands by using the so called `dmod` function. It ensures that `reducedRatMod` algorithm works successfully by providing $2s(v) < t(v) - 1$ so that u and v variables can be swapped without searching any conditions. This function reduces the number u more efficiently in two manners: it does one multiplication rather then two and it does not lead spurious factors.

The condition $\gcd(v, \beta) = 1$ is satisfied by `RemoveFactors` and `swap` operations. At the end of the loop $u = \gcd(u_0, v_0)$ may not be realized. This is due to possible spurious factors occurred in `reducedRatMod` by the subtraction of $nv - du$. Spurious factors problem has been solved by using `dmod` & `gcd` functions two times in a row.

Algorithm 4: General ReducedRatMod algorithm

input : $x, y > 0, m > 1$, with $\gcd(x,m) = \gcd(y,m) = 1$.
output: (n, d) such that $0 < n, |d| < \sqrt{m}$ and $ny \equiv xd \pmod{m}$

$c \leftarrow x/y \bmod m$
$f_1 = (n_1, d_1) \leftarrow (m, 0)$
$f_2 = (n_2, d_2) \leftarrow (c, 1)$
while $n_2 \geq \sqrt{m}$ **do**
    $f_1 \leftarrow f_1 - \left\lfloor \frac{n_1}{n_2} \right\rfloor f_2$
    $swap(f_1, f_2)$
end
Return f_2.

Theorem 2.2.1. *(Weber, 1995) The output from the general* `reducedRatMod` *algorithm satisfies*

$$ny \equiv nx \pmod{m} \text{ and } 0 < n, |d| < \sqrt{m}$$

This is an Euclidean step. Define $n_1^{'}$, $n_2^{'}$ with initial values n_1 and n_2, respectively. The initial values of e_1, e_2, d_1, d_2 are set as 0, 1, 0, 1, respectively. Then, it is straight forward to show that the invariant equations

$$\begin{aligned} n_1^{'}e_2 + n_2^{'}d_1 &= n_1 \\ n_1^{'}e_1 + n_2^{'}d_2 &= n_2 \end{aligned}$$

are satified in every iteration.

The equation is $n_1 v - d_1 u \equiv 0 \pmod{2^{2t}}$ where the initial values are $n_1 = 2^{2t}$ and $n_2 = 0$. After these successful linear transformation steps, the invariant equations are still satisfied. The values e_1 and e_2 are not part of the computation in Algorithm 4. They are rather auxiliary numbers to inspect through the algorithm.

Algorithm 5: `dmod` operation

input : $u_0, v_0, \beta > 0$, with $\gcd(v_0, \beta) = 1$.

output: $|u_0 - (u_0/v_0 \bmod \beta^{\ell_\beta(u_0)-\ell_\beta(v_0)+1})v_0|/\beta^{\ell_\beta(u_0)-\ell_\beta(v_0)+1}$

$u \leftarrow u_0$
while $\ell_\beta(u) \geq \ell_\beta(v_0) + W$ **do**
  if $u \not\equiv 0 \pmod{\beta^W}$ **then**
    $u \leftarrow |u - (u/v_0 \bmod \beta^W)v_0|$
    $u \leftarrow u/\beta^W$
  end
end
$d \leftarrow \ell_\beta(u) - \ell_\beta(v_0)$
if $u \not\equiv 0 \pmod{\beta^{d+1}}$ **then**
  $u \leftarrow |u - (u/v_0 \bmod \beta^{d+1})v_0|$
end
Return u/β^{d+1}.

If the difference between the size of the operands gets too large, there is an long division operation to make large jumps through the Euclidean steps. Weber achieves this by using `dmod` (digit modulus) operation (Weber, 1995). In contrast, Sedjelmaci makes this operation by using `mod` operation (Sedjelmaci, 2007). Weber's version is given in Algorithm 5.

2.2.1 ELIMINATING SPURIOUS FACTORS

Despite the fact that spurious factors might occur, Jebelean-Weber algorithm is a fast alternative to the classical Euclidean gcd algorithm. In our aim to compute modular inverses however, these spurious factors are disasterous. One needs to prevent them from happening even before attempting to compute the extended gcd sequence with a Jebelean-Weber variant.

Sorenson (Sorenson, 2004) decribes how to prevent spurious factors from appearing. Later on, Sedjelmaci (Sedjelmaci, 2007) uses Sorenson's description to provide an explicit k-ary gcd sequence. The core ideas are provided in the following theorem.

Theorem 2.2.2 ((Sorenson, 2004))**.** *Let a, b, c, $d \in \mathbb{Z}$ satisfy $ad-bc = 1$. Then,*

$$\gcd(u, v) = \gcd(av - bu, cv - du).$$

Proof. We will show that $\gcd(u, v) \mid \gcd(av - bu, cv - du)$ and $\gcd(av - bu, cv - du) \mid \gcd(u, v)$.

Suppose $k = \gcd(av - bu, cv - du)$ where $ad - bc = 1$ for a, b, c, $d \in \mathbb{Z}$. Then, $k \mid (av - bu)$ and $k \mid (cv - du)$. Therefore, we have $k\alpha = av - bu$ and $k\beta = cv - du$ for some α, $\beta \in \mathbb{Z}$. We multiply these equations with c and a respectively and get $kc\alpha = acv - bcu$ and $ka\beta = acv - adu$. Now with subtraction we get $kc\alpha - ka\beta = k(c\alpha - a\beta) = (ad - bc)u = u$. Similarly multiplying with d and b, we get $kd\alpha = adv - bdu$ and $kb\beta = bcv - bdu$. And so, $k(d\alpha - b\beta) = (ad - bc)v = v$. Therefore, $k \mid u$ and $k \mid v$. This implies $\gcd(av - bu, cv - du) \mid \gcd(u, v)$.

Now, assume $t = \gcd(u, v)$. By definition, $u = t\alpha^{'}$ and $v = t\beta^{'}$ for some $\alpha^{'}$, $\beta^{'} \in \mathbb{Z}$. Now, $av - bu = at\beta^{'} - bt\alpha^{'}$ and $cv - du = ct\beta^{'} - dt\alpha^{'}$, and we get $\gcd(u, v) \mid \gcd(av - bu, cv - du)$.

In conclusion, $\gcd(u, v) = \gcd(av - bu, cv - du)$. $\square$

The solution, is therefore, requires a computation of two linear transformation rather than one and operate on both operands. Building on this observation, Chapter 3 presents an extended version of the JWSS algorithm. Our variant does not use the long divison step which occurs rarely for practical values of t (e.g. t=32-2=30 or t=64-2=62).

CHAPTER 3

MODULAR INVERSE BASED ON JWSS METHOD

In this chapter, we show how to use the JWSS method to compute the modular inverse

$$V^{-1} \mod U$$

for given two positive integers $U > V > 0$ with U is odd and $GCD(U, V) = 1$. The algorithm is essentially an extended GCD version of the JWSS algorithm. We then analyze the algorithm and provide a proof of its correctness. We also provide a Magma implementation at the end of this chapter.

3.1 EXTENDED JWSS METHOD AND MODULAR INVERSE COMPUTATION

Since the notation and background on the JWSS algorithm has already been provided in Chapters 2 and 3, it is suitable to give the extended version without further discussion, in Algorithm 6. The computations regarding y' and y'' are redundant. In other words, those computations can be removed from the modular inverse computations. The algorithm is given in full detail to prevent repetition.

Algorithm 6: Modular Inverse Algorithm Based On JWSS Method

input : $u_0 > v_0 > 0$ integers with u is odd and $\gcd(u_0, v_0) = 1$.

output: $(v_0)^{-1} \mod u_0$

$u \leftarrow u_0$, $v \leftarrow v_0$
$x' \leftarrow 0$, $x'' \leftarrow 1$
$y' \leftarrow 0$, $y'' \leftarrow 1$
$E \leftarrow 0$
$v, x', y', E \leftarrow$ `MakeOdd`(v, x', y', E)
while $v \neq 0$ **do**
  $a, b, c, d \leftarrow$ `ReducedRatMod`$(u \bmod 2^{2t}, v \bmod 2^{2t}, 2^{2t})$
  $u, v \leftarrow$ `LinearTransform`(u, v, a, b, c, d)
  $x', x'' \leftarrow$ `LinearTransform`(x', x'', a, b, c, d)
  $y', y'' \leftarrow$ `LinearTransform`(y, y'', a, b, c, d)
  $u \leftarrow$ `RemoveDigits`$(u, 2t)$
  $v \leftarrow$ `RemoveDigits`$(v, 2t)$
  $u, x'', y'', E \leftarrow$ `MakeOdd`(u, x'', y'', E)
  $v, x', y', E \leftarrow$ `MakeOdd`(v, x', y', E)
  $u, x', y' \leftarrow$ `MakePositive`$(u, x', y', u < 0)$
  $v, x'', y'' \leftarrow$ `MakePositive`$(v, x'', y'', v < 0)$
  $u, v, x', x'', y', y'' \leftarrow$ `Swap`$(u, v, x', x'', y', y'', v > u)$
  $E \leftarrow E + 2t$
end
Return `Modinv2e`(x', u_0, E)

We subdivided basic tasks within the algorithm into auxiliary functions for easier treatment. We start by detailing these functions all of which are self-explanatory.

Algorithm 7: Swap

input : $x, y, a, b, c, d \in \mathbb{Z}$ and a boolean value k

if $k = \mathit{true}$ **then**
  Return y, x, b, a, d, c
else
  Return x, y, a, b, c, d
end

Algorithm 8: Remove Digits

input : $x, t \in \mathbb{Z}$ where $x \equiv 0 \pmod{2^{2t}}$

Return $x/2^{2t}$

Algorithm 9: Make Positive

input : $x, a, b \in \mathbb{Z}$ and a boolean value k.

if $k = \mathit{true}$ **then**
  Return $-x, -a, -b$
else
  Return x, a, b
end

Algorithm 10: Make Odd

input : $x, a, b, E \in \mathbb{Z}$

while $x = 0 \pmod 2$ *and* $x \neq 0$ **do**
  $x, a, b, E \leftarrow x/2, a \cdot 2, b \cdot 2, E + 1$
end
Return x, y, E.

Algorithm 10 here needs some extra care. It is used to make the number odd by clearing the multiples of 2. However, since we need to maintain the invariant equations

$$v_0 x'' + u_0 y' = v * 2^E \tag{3.1}$$

$$v_0 x' + u_0 y'' = u * 2^E \tag{3.2}$$

whose coefficients may not be divisible by 2, we keep track of those missing divisions in counter variable E. This is necessary because if v is not an odd number, the modular inverse operation within the `reducedRatMod` function will not work.

Algorithm 11: Linear Transformation

input : $a, b, c, d, x, y \in \mathbb{Z}$.

Return $bx - ay, dx - cy$

Algorithm 11 provides the linear transformations to produce new values of u, v, x and y using the numbers a, b, c, d generated from `reducedRatMod` function. As seen in Theorem 3.2.2, linear transformations with these four updated values does not violate the invariant equations.

Algorithm 12: Modinv2e

input : $x, u_0, E \in \mathbb{Z}$ where $2^E \bmod u_0$.
output: $x(2^E)^{-1} \bmod u_0$

for $i \leftarrow 1$ **to** k **do**
    if $x = 0 \pmod 2$ **then**
        $x \leftarrow x + u_0$
    else
        $x \leftarrow x/2$
    end
end
Return x.

The algorithm 12 is used to perform missing divisions by 2 which are delayed with the help of the counter E.

3.2 CORRECTNESS AND ANALYSIS

We prove that our algorithm (i.e. Algorithm 6) is correct by introducing the following three theorems: Theorem 3.2.1 bounds the intermediate values in Algorithm 4, Theorem 3.2.2 asserts the invariant equations in Algorithm 6 and Theorem 3.2.3 shows that the algorithm is correct.

Theorem 3.2.1. *For the intermediate values u and v in Algorithm 6, let a, b be the output of* `ReducedRatMod` *algorithm for the inputs u and v. Then $0 < |av - bu| < 2^{(t+1)}u$.*

Proof. By Theorem 2.2.1, we have $0 < a, |b| < 2^t$ and it can be written as $0 < a < 2^t$ and $0 < |b| < 2^t$ respectively. Then,

$$|av - bu| \leq |au| + |bv| < (|a| + |b|)u < (2^t + 2^t)u = 2^{(t+1)}u.$$

It is known that $|av - bu|$ is divided by 2^{2t}, then the size of u is decreased by almost 2^t in every iteration. □

Theorem 3.2.2. *In the Algorithm 6, let u_0, v_0 be the input values, and u, v, x', x'', E be the intermediate values in while loop. Then the following equations hold:*

$$vx' - ux'' = 0 \mod u_0 \quad (3.3)$$

$$v_0x' = u2^E \mod u_0 \quad (3.4)$$

$$v_0x'' = v2^E \mod u_0 \quad (3.5)$$

Proof. We will prove by induction. Note that the equations hold for the initial values $(u, v, x', x'', E) \leftarrow (u_0, v_0, 0, 1, 0)$. Now, by assuming the equations hold after some number of iterations for some intermediate values u, v, x', x'' and E, it is enough to show the equations still hold in the next iteration. For the next iteration, denote the updated intermediate values by $u_{\text{new}}, v_{\text{new}}, x'_{\text{new}}, x''_{\text{new}}$ and E_{new}.

In steps 8-9, the new intermediate values are set as $u_{\text{new}} \leftarrow av - bu, v_{\text{new}} \leftarrow cv - du, x'_{\text{new}} \leftarrow ax'' - bx'$ and $x''_{\text{new}} \leftarrow cx'' - dx'$ using a, b, c, d obtained in step 7 and performing Linear Transformations later. Then

$$\begin{aligned} v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}} &= (cv - du)(ax'' - bx') - (av - bu)(cx'' - dx') && (3.6)\\ &= -bcvx' - adux'' + advx' + bcux'' && (3.7)\\ &= (ad - bc)(vx' - ux'') && (3.8) \end{aligned}$$

Since $vx' - ux'' = 0 \mod u_0$ by our assumption, we have $v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}} = 0 \mod u_0$. Moreover, since $v_0x' - u2^E = 0 \mod u_0$ and $v_0x'' - v2^E = 0 \mod u_0$, we have

$$\begin{aligned} v_0x'_{\text{new}} - u_{\text{new}}2^{E_{\text{new}}} &= v_0(ax'' - bx') - (av - bu)2^{E_{\text{new}}} && (3.9)\\ &= a(v_0x'' - v2^E) - b(v_0x' - u2^E) && (3.10)\\ &= 0 \mod u_0 && (3.11) \end{aligned}$$

and

$$\begin{aligned} v_0x''_{\text{new}} - v_{\text{new}}2^{E_{\text{new}}} &= v_0(cx'' - dx') - (cv - du)2^{E_{\text{new}}} && (3.12)\\ &= c(v_0x'' - v2^E) - d(v_0x' - u2^E) && (3.13)\\ &= 0 \mod u_0. && (3.14) \end{aligned}$$

In the steps 15 and 16, it is clearly seen that the equations do still hold even the signs of the intermediate values are changed after MakePositive functions, because

$$\begin{aligned} (-v_{\text{new}})x'_{\text{new}} - u_{\text{new}}(-x''_{\text{new}}) &= -[v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}}] = 0 \mod u_0, && (3.15)\\ v_{\text{new}}(-x'_{\text{new}}) - u_{\text{new}}(-x''_{\text{new}}) &= -[v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}}] = 0 \mod u_0, && (3.16)\\ v_0(-x'_{\text{new}}) - (-u_{\text{new}})2^{E_{\text{new}}} &= -\left[v_0x'_{\text{new}} - u_{\text{new}}2^{E_{\text{new}}}\right] = 0 \mod u_0, && (3.17)\\ v_0(-x''_{\text{new}}) - (-v_{\text{new}})2^{E_{\text{new}}} &= -\left[v_0x''_{\text{new}} - v_{\text{new}}2^{E_{\text{new}}}\right] = 0 \mod u_0. && (3.18) \end{aligned}$$

Without loss of generality, assume the signs of the intermediate values are suitably changed for the next steps, if it is necessary.

In steps 11, 12 and 18, the intermediate values are updated as $u_{\text{new}} \leftarrow u_{\text{new}}/2^{2t}, v_{\text{new}} \leftarrow v_{\text{new}}/2^{2t}, E_{\text{new}} \leftarrow E_{\text{new}} + 2t$. Then

$$\left(\frac{v_{\text{new}}}{2^{2t}}\right) x'_{\text{new}} - \left(\frac{u_{\text{new}}}{2^{2t}}\right) x''_{\text{new}} = \frac{v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}}}{2^{2t}} = 0 \mod u_0$$

since $v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}}$ is divisible by 2^{2t} and u_0 is odd. Moreover,

$$\left(\frac{u_{\text{new}}}{2^{2t}}\right) 2^{E_{\text{new}}+2t} = u_{\text{new}}2^{E_{\text{new}}},$$

$$\left(\frac{v_{\text{new}}}{2^{2t}}\right) 2^{E_{\text{new}}+2t} = v_{\text{new}}2^{E_{\text{new}}}.$$

Thus, the equations do still hold. Continue to the next steps with updated intermediate values.

In steps 13 and 14, $u_{\text{new}} \leftarrow u_{\text{new}}/2, x''_{\text{new}} \leftarrow 2x''_{\text{new}}, E_{\text{new}} \leftarrow E_{\text{new}} + 1$ incrementally until u_{new} is odd, and similarly $v_{\text{new}} \leftarrow v_{\text{new}}/2, x'_{\text{new}} \leftarrow 2x'_{\text{new}}, E_{\text{new}} \leftarrow E_{\text{new}} + 1$ incrementally until v_{new} is odd. Similar to the proof done for steps 11, 12 and 18, the equations do still hold. Continue to the next steps with updated intermediate values.

In step 17, if the intermediate values are necessarily swapped as $u_{\text{new}}, v_{\text{new}}, x'_{\text{new}}, x''_{\text{new}} \leftarrow v_{\text{new}}, u_{\text{new}}, x''_{\text{new}}, x'_{\text{new}}$, we have

$$\begin{aligned} u_{\text{new}}x''_{\text{new}} - v_{\text{new}}x'_{\text{new}} &= -\left[v_{\text{new}}x'_{\text{new}} - u_{\text{new}}x''_{\text{new}}\right] = 0 \mod u_0, && (3.19)\\ v_0x''_{\text{new}} &= v_{\text{new}}2^{E_{\text{new}}} \mod u_0 && (3.20)\\ v_0x'_{\text{new}} &= u_{\text{new}}2^{E_{\text{new}}} \mod u_0. && (3.21) \end{aligned}$$

In the end of the while loop, we see that the equations still hold. □

Theorem 3.2.3. *For two odd integers $u_0 > v_0 > 0$ with $\gcd(u_0, v_0) = 1$, Algorithm 6 returns $v_0^{-1} \bmod u_0$.*

Proof. Recall the second equation in Theorem 3.2.2, i.e.

$$v_0x' = u2^E \mod u_0.$$

Note that, after the last iteration, $v = 0$ and $u = 1$. Therefore,

$$v_0x' = 2^E \mod u_0$$

where x' here is the final value after the while loop. Then,

$$v_0x'(2^E)^{-1} = 1 \mod u_0$$

in other words $x'(2^E)^{-1} \mod u_0$ is the desired solution which is obtained by Algorithm 12. $\square$

3.3 MAGMA CODES

This section provides Magma implementations of the algorithms provided in Section 3.1. These codes are then used to implement the same in C language.

```
Swap := function(x,y,a,b,c,d,k)
  if k eq true then
    return y,x,b,a,d,c;
  end if;
  return x,y,a,b,c,d;
end function;
```

Code 3.1: Magma Code for Swap

```
MakePositive := function(x,a,b,k)
  if k eq true then
    return -x, -a, -b;
  else
    return x, a, b;
  end if;
end function;
```

Code 3.2: Magma Code for Make Positive

```
LinearTransform := function(x,y,a,b,c,d)
  return b*x-a*y, d*x-c*y;
end function;
```

Code 3.3: Magma Code for Linear Transform

```
MakeOdd := function(x,a,b,E)
  while IsEven(x) and (x ne 0) do
    x := x div 2; a *:= 2; b *:= 2; E +:= 1;
  end while;
  return x,a,b,E;
end function;
```

Code 3.4: Magma Code for Make Digits Odd

```
Modinv2e := function(xdd,ud,k)
  for i:=1 to k do
    if IsOdd(xdd) then
      xdd := xdd+ud;
    end if;
    xdd := xdd div 2;
  end for;
  return xdd;
end function;
```

Code 3.5: Magma Code for Modinv2e

```
RemoveDigits := function(x, _2t)
  return x div 2^_2t;
end function;
```

Code 3.6: Magma Code for Remove Digits

```
swapt := function(a,b)
  return b, a;
end function;

ReducedRatMod := function(u, v, _2t)
  c := (u * Modinv(v, 2^_2t)) mod 2^_2t;
  a := 2^_2t;
  b,d := copy2(0,1);
  while a ge Sqrt(2^_2t) do
    assert (A*e2 + B*b) eq a;
    assert (A*e1 + B*d) eq c;

    q := a div c;
    a := a - q*c;
    a, c := swapt(a,c);

    b := b - q*d;
    b, d := swapt(b,d);
  end while;
  return a,b,c,d;
end function;
```

Code 3.7: Magma Code for ReducedRatMod

```
accelModinv := function(u, v, base, s, t, W)
  xdd,xd,ud,vd := copy4(1,0,u,v);
  ydd,yd := copy2(0,1);
  E := 0;
  v,xd,yd,E := MakeOdd(v,xd,yd,E);
  while (v ne 0) do
    assert vd*xd + ud*yd eq u*2^E;
    assert vd*xdd + ud*ydd eq v*2^E;
    a,b,c,d := ReducedRatMod(u mod 2^(2*t), v mod 2^(2*t), 2*t);
    u, v := LinearTransform(u,v,a,b,c,d);
    xd, xdd := LinearTransform(xd,xdd,a,b,c,d);
    yd, ydd := LinearTransform(yd,ydd,a,b,c,d);
    u := RemoveDigits(u, 2*t);
    v := RemoveDigits(v, 2*t);
    u,xdd,ydd,E := MakeOdd(u,xdd,ydd,E);
    v,xd,yd,E := MakeOdd(v,xd,yd,E);
    u,xd,yd := MakePositive(u,xd,yd,u lt 0);
    v,xdd,ydd := MakePositive(v,xdd,ydd,v lt 0);
    u,v,xd,xdd,yd,ydd := Swap(u,v,xd,xdd,yd,ydd,v gt u);
    E +:= 2*t;
  end while;
  return Modinv2e(xd,ud,E);
end function;
```

Code 3.8: Magma Code for k-ary Modular Inverse

CHAPTER 4

SIMD IMPLEMENTATION

Intel's AVX2 instruction set is currently the most accessible high-end processing platform since it is available in and after every Haswell processors including other popular processor families like Skylake and Kabylake. Therefore, it is reasonable to investigate the performance of Algorithm 6. AVX2 provides 16×256-bit `ymm` registers. The amount of data that can be kept in these registers is over 4 times more than the data that be accommodated in the 16×64-bit integer registers. Therefore, inputs of Algorithm 6 has potential to be processed faster on AVX2. This section investigates this possibility.

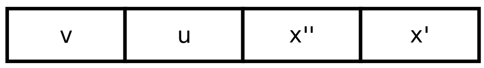
AVX2 feature is extremely important where time consuming operations are in question. AVX2 instructions are capable of processing a large set of numbers at a time, rather than processing them individually and so that enhance the application performance. These large numbers are placed into AVX2 vectors such that, they can enlarge up to 256 bits. AVX2 features can be accessed via `immitrin.h` header file through Intel intrinsics.

In implementing Algorithm 6 over AVX2 circuit, the first question that arises is how to represent large integers. There is a vast number of possibilities at this phase. It is our experience that the representation choice tends to make a huge difference in the overall performance. We summarize a few below and explain the best choice out of them together with the reasoning.

4.1 HIGH LEVEL REPRESENTATION OF DATA

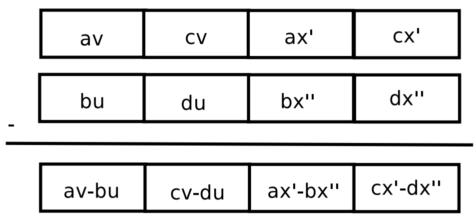
One approach could be working over the four 64-bit lanes where the lanes are dedicated to v, u, x'', and x'. Such an approach look very simple, cf. Figure 4.2. This approach leads to very poor utilization of the underlying hardware since u and v tends to decrease where x' and x'' tends to increase in size. However, when keeping then side by side in vector form, the implementer is forced to allocate equal amount of memory for all. And then, several digits will be dummily processed. Other problems do exist. For instance, one can easily compute bu, av, bx', and ax'' but then one has to permute inside 128 bit lanes in order to compute $bu - av$ and $bx' - ax''$. Similar comments applies to linear transforms with c and d. Finally, the implementation will require extra permutations for packing data back in aforementioned v, u, x'', and x' form horizontally aligned in vector form.

Figure 4.1 4-way Representation, a first attempt



Another approach which solves some of these problems is to separate vector variables for u & v and x' & x''. In this version, the 64 bit lanes in a vector contains repeated data in the form u, u, and v, v. Yet another variable contains x', x' and x'', x''. In this form u and v can share equal number of digits from start to the end of computation. Similar applies to x' and x''. This approach partially solves the digit count problem in the first approach. However, permutations are still not eliminates. For instance, Figure 4.2 depicts linear transformation phase in such a situation.

Figure 4.2 4-way Representation, a second attempt



The output $av - bu$ is now need to be copied over the first two lanes of v. Similar applies to $cv - du$, $ax' - bx''$, $cx' - dx''$. The programmer should prevent such permutations as much as possible in order to obtain a high throughput.

A third approach could be place limbs of each variable vertically. Figure 4.3 summarizes this situation. The main problem here is the maintenance of carries between limbs. For instance, carries from a_2 to a_3 would require a sizeable amount of extra code which will not only cost time but also sacrifice code readability and easy maintenance.

Figure 4.3 4-way Representation, a third attempt

vec_1	a_0	a_3	a_6	a_9	$v[0][0]$	$v[0][1]$	$v[0][2]$	$v[0][3]$
vec_2	a_1	a_4	a_7	a_{10}	$v[1][0]$	$v[1][1]$	$v[1][2]$	$v[1][3]$
vec_3	a_2	a_5	a_8	a_{11}	$v[2][0]$	$v[2][1]$	$v[2][2]$	$v[2][3]$
		$\vdots$				$\vdots$		

Up to now, it seems that any alternative comes with a huge disadvantage. Nevertheless, we were able to find the following fine grain solution.

The representation that we use separates all variables in to distinct vector arrays and places the limbs of a variable first in horizontal fashion in 64 bit lanes of a vector and then vertically over elements of the vector array. This approach is depicted in Figure 4.4.

Figure 4.4 4-way Representation, the selected approach

vec_1	a_0	a_1	a_2	a_3	$v[0][0]$	$v[0][1]$	$v[0][2]$	$v[0][3]$
vec_2	a_4	a_5	a_6	a_7	$v[1][0]$	$v[1][1]$	$v[1][2]$	$v[1][3]$
vec_3	a_8	a_9	a_{10}	a_{11}	$v[2][0]$	$v[2][1]$	$v[2][2]$	$v[2][3]$
		$\vdots$				$\vdots$		

This final approach has its pros and cons. On the positive side, every variable is maintained separately so that if not needed the limb access can be limited. In addition, no permutation is needed between lanes. Moreover, the code readability is fairly better in comparison with other alternatives. However, handling the carries and right shifts seems to be problematic at the first glance. But we found a programmatic way of minimizing the speed penalties referenced from this representation. Our solution is as follows. We concentrate on Figure 4.4 for simplicity. For instance, carries that needs to be transferred from a_3 to a_4 can be handled by slow permutation operation. However, we want to eliminate all such permutations. At this stage, one can define a vector pointer whose starting address is a_1. Then, the vector pointer acts as 64 bit right shifted array on the whole number. This greatly simplifies doing the carries and the make odd routine without causing untolerable speed penalties as in the other approaches. In addition, the code reads much simpler and shorter. One obstacle is that, `gcc`

`-avx2 -O0` mode does not work properly for detailed debugging. Therefore, the code is developed in `gcc -avx2 -O3` mode and the debugging was performed with screen outputs.

4.2 LOW LEVEL REPRESENTATION OF DATA

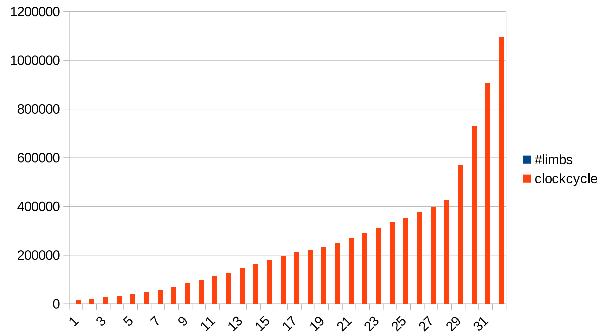
AVX2 multipliers can handle $32\times32\rightarrow64$-bit vector-vector integer multiplication. Algorithm 6 operates on signed integers. Therefore, sign management is necessary in our implementation. Therefore, we could use signed radix 31 representation. However, it is more beneficial to use signed radix 30 since we can delay carries in linear transforms which requires a singed subtraction. When the limbs are kept in 30 bits, the maximum value after linear transformation is calculated as: $(2^{30}-1)\cdot(2^{30}-1) = 2^{60}-2\cdot2^{30}+1$. So that, numbers can be kept in 64 bit registers easily including sign bit. After this operation, the numbers must be reduced to 30 bits in order to perform the following iteration.

Our implementation works for arbitrary sizes of u. Table 1 and Figure 2.2 provides cycle counts on inputs of different sizes.

Table 2 Cycle Counts on Haswell i7-5500U using AVX2 circuit for Algorithm 6

# of limbs	# of bits	cycle counts
3	360	13 680
5	600	26 064
7	840	41 568
10	1200	66 924
15	1800	146 688
20	2400	222 264
25	3000	310 440
30	3600	425 940
40	4800	731 112
45	5400	905 832
50	6000	1 094 868

Figure 4.5 Graphical Illustration of Table 2



Larger inputs benefit more from AVX2 features since `ReducedRatMod` operation generates coefficients a, b, c, d only once for each iteration and once the vectors $[a, b, a, b]$ and $[c, d, c, d]$ are ready to be used in the linear transformation, they are reused for each limb of the numbers. Therefore, the cost of `ReducedRatMod` are less dominant for larger inputs, which is handled with the 64-bit circuit in the classic way using `signed long` data type.

We also experienced using AVX2-only intrinsics for the `ReducedRatMod` operation but this option seems to be slightly slower.

CHAPTER 5

CONCLUSION

In this thesis, well-known quadratic time k-ary GCD algorithms which exist in literature are examined. An extended version of a right-to-left GCD variant, namely JWSS method, is provided. A modular inverse algorithm was derived from the extended sequence and implemented. We conclude that SIMD implementations of the modular inverse algorithm based on JWSS method is very efficient on AVX2 circuit. Even better speeds are likely to be possible on the new AVX-512 supported processors.

Bernstein and Yang have proposed a new k-ary gcd variant which allows fast and constant-time implementation of gcd and modular inverses. Their algorithm solves several irregularities of existing approaches and nicely optimizes the gcd routine. It would be very interesting to investigate their algorithm on AVX platforms in the context of this thesis. Because implementing their algorithm would require an update on the `ReducedRatMod` function and completely deleting subroutines `MakeOdd`, `Swap`, and `IsZero`. However, their algorithm came only very recently (May 2019) towards the finishing of this thesis work. Therefore, this investigation has been left as a future work.

REFERENCES

Bernstein, D. (2006). Curve25519: New Diffie-Hellman speed records. In Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings, pages 207–228.

Bernstein, D. and Yang, B.-Y. (2019). Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3):340–398.

Bernstein, D. J., Chuengsatiansup, C., Lange, T., and Schwabe, P. (2014). Kummer strikes back: New DH speed records. In Sarkar, P. and Iwata, T., editors, Advances in Cryptology - ASIACRYPT 2014, volume 8873 of LNCS, pages 317–337. Springer Berlin Heidelberg.

Bos, J. W. (2014). Constant time modular inversion. Journal of Cryptographic Engineering, 4:275–281.

Chou, T. (2015). Sandy2x: New Curve25519 speed records. In Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers, pages 145–160.

Jebelean, T. (1993). A generalization of the binary GCD algorithm. In Bronstein, M., editor, Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation, ISSAC '93, pages 111–116. ACM.

Karati, S. and Sarkar, P. (2017). Kummer for genus one over prime order fields. In Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II, pages 3–32.

Katz, J., Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). Handbook of applied cryptography. CRC press.

Knuth, D. E. (1971). The analysis of algorithms. Actes du Congrés International des Mathématiciens, 4:269–274.

Knuth, D. E. (2014). Art of computer programming, volume 2: Seminumerical algorithms. Addison-Wesley Professional.

Lehmer, D. H. (1938). Euclid's algorithm for large numbers. The American Mathematical Monthly, 45(4):227–233.

Möller, N. (2008). On Schönhage's algorithm and subquadratic integer gcd computation. Mathematics of Computation, 77(261):589–607.

Schönhage, A. (1971). Schnelle berechnung von kettenbruchentwicklungen. Acta Inf., 1:139–144.

Sedjelmaci, S. M. (2007). Jebelean-Weber's algorithm without spurious factors. Information Processing Letters, 102(6):247–252.

Sorenson, J. P. (1994). Two fast GCD algorithms. Journal of Algorithms, 16(1):110–144.

Sorenson, J. P. (2004). An analysis of the generalized binary GCD algorithm. High Primes and Misdemeanors: Lectures in Honour of the 60th Birthday of Hugh Cowie Williams, pages 327–340.

Stehlé, D. and Zimmermann, P. (2004). A binary recursive gcd algorithm. volume 3076, pages 411–425.

Stein, J. (1967). Computational problems associated with Racah algebra. Journal of Computational Physics, 1(3):397–405.

Weber, K. (1995). The accelerated integer GCD algorithm. ACM Transactions on Mathematical Software (TOMS), 21(1):111–122.

APPENDIX A

CLASSICAL GCD ALGORITHMS

Algorithm 13: Naive Euclid's GCD

input : $a, b > 0$ and $a \geq b$.
output: $\gcd(a, b)$.
while $b \neq 0$ **do**
    $(a, b) \leftarrow (max(b, a-b), min(b, a-b))$
end
return a.

Algorithm 14: Euclid's GCD

input : $a, b > 0$ and $a \geq b$.
output: $\gcd(a, b)$.
while $b \neq 0$ **do**
    $(a, b) \leftarrow (b, a \bmod b)$
end
return a.

The validity of algorithms above is related with the property of gcd;

$$\gcd(a, b) = \gcd(a - qv)$$

Algorithm 15: Binary GCD

input : $a, b > 0$ and $a \geq b$.
output: $\gcd(a, b)$.
$g \leftarrow 1$
while $a \bmod 2 = b \bmod 2 = 0$ **do**
    $(g, a, b) \leftarrow (2g, a/2, b/2)$
end
while $x \neq 0$ **do**
    while $a \bmod 2 = 0$ **do**
        $a \leftarrow a/2$
    end
    while $b \bmod 2 = 0$ **do**
        $b \leftarrow b/2$
    end
    $t \leftarrow |a - b|/2$
    if $x \geq y$ **then**
        $x \leftarrow t$
    else
        $y \leftarrow t$
    end
end
return $(g \cdot a)$.

The algorithm simply consist of successively reducing odd values by using the following familiar properties of gcd function:

1. If a and b are both even, $\gcd(a, b) = 2\gcd(a/2, b/2)$.

2. If a is even and b is odd, $\gcd(a, b) = \gcd(a/2, b)$.

3. If a is odd and b is even, $\gcd(a, b) = \gcd(a, b/2)$.

4. If a and b is both odd, $\gcd(a, b) = \gcd(|a - b|/2, \min(a, b))$.

Using the idea that division by 2 is only requires shift operation and so, it is a better algorithm than Euclid's, though its worst case running time is also $0(n^2)$, where $n = \log_2 n$. Another difference from Euclid's GCD is that it reduces the least significant bits first. This algorithm is used in the GMP library for the small size inputs.

Algorithm 16: Extended Binary GCD

input : $a, b > 0$.
output: integers a, b and v such that
$ax + by = v$, where $v = gcd(x, y)$.

```
g ← 1
while x mod 2 = y mod 2 = 0 do
    (g, x, y) ← (2g, x/2, y/2)
end
(u, v) ← (x, y)
(A, B, C, D) ← (1, 0, 0, 1)
while u ≠ 0 do
    while u mod 2 = 0 do
        u ← u/2
        if A mod 2 = B mod 2 = 0 then
            (A, B) ← (A/2, B/2)
        else
            (A, B) ← ((A+y)/2, (B−x)/2)
        end
    end
    while v mod 2 = 0 do
        v ← v/2
        if C mod 2 = d mod 2 = 0 then
            (C, D) ← (C/2, D/2)
        else
            (C, D) ← ((C+y)/2, (D−x)/2)
        end
    end
    if u ≥ v then
        (u, A, B) ← (u − v, A − C, B − D)
    else
        (v, C, D) ← (v − u, C − A, D − B0)
    end
end
return (a, b, g · v).
```

Idea of calculation Extended GCD is mostly used to find modular multiplicative inverse by solving the following problem: given two integers x and y, with at least one of them is nonzero, it computes $d = gcd(x, y)$. Then, there exits integers A, B s.t. $Ax + By = d$. The equation $Ax + By = d$ is called *Bezout equation* and A, B is called *Bezout's coefficients*. In particular, if x and y are relatively prime (i.e. $gcd(x, y) = 1$), then $Ax + By = 1$.

If x_0, y_0 are the initial values and x, y is the next values,the following invariants keep at the start of each iteration and after the loop: $Ax_0 + By_0 = x$ and $Cx_0 + Dy_0 = y$.

In the last case, B is called the modular multiplicative inverse of a wrt y since $By = 1 \bmod x$. We then simply run Algorithm 16, the equation ends with $Ax + By = 1 \bmod x$, and it is equal to $y^{-1} = B \bmod x$. Since the value x is not needed in this calculation, we can simply ignore computing redundant A and C values for modular inverse operation.

Figure A.1 Extended Binary GCD illustration

x		y	
18914144994474109809	1	20860527183790487785	0
18914144994474109809	1	19463821893163779776	-1
18670847220809562562	-23642681243092637726	24329777366454724247	26075658979738109973
9092125836740234034	-23642681243092637726	24329777366454724247	26075658979738109973
4302765144705569770	-23642681243092637726	24329777366454724247	26075658979738109973
1908084798688237638	-23642681243092637726	24329777366454724247	26075658979738109973
710744625679571572	-23642681243092637726	24329777366454724247	26075658979738109973
177686156419892893	11267215279911334945	6561161724465435 4	-12426681232531442919
144880347797565716	-38234648572813874 32	3280580862232717 7	4216922975629522433
34142783270642 52	-3823464857281387432	3280580862232717 7	4216922975629522433
853569581766063	6472876646973653180	3195223904056111 4	-7138975581024979554
853569581766063	13532779827741867202	1512254993851449 4	-14925386347166991764
853569581766063	17062731418125974213	670770538749118 4	-18818591730237997869
434337995047864	-4844249597140285927	419231586718199	5342752761794652566
54292249380983	2930066264176849587	364939337337216	-3231588156484261531
51441160808036	-14606017876474687668	2851088572947	16109067211240422103
10009201629062	-14606017876474687668	2851088572947	16109067211240422103
2153512241584	-14606017876474687668	2851088572947	16109067211240422103
134594515099	-9822500898075790732	2716494057848	10833296829276719652
134594515099	-5901368131097631810	204967242132	6508655313591131398
83352704566	-10932414530011462857	51241810533	12057427420293026742
41676352283	-4549790808580690913	9565458250	5017992347562444995
36893623158	-11731967901527400361	4782729125	12939259765676466390
13664082454	-11731967901527400361	4782729125	12939259765676466390
2049312102	-11731967901527400361	4782729125	12939259765676466390
1024656051	-4649734980020183101	3758073074	5128221390735374951
1024656051	-4699707065739929195	854380486	5183335912321839929
597465808	-11806926030107019502	427190243	13021931548056163857
37341613	-9737951225085243416	389848630	10740046473402044910
37341613	-2800000807520845622	157582702	3088138162046903508
37341613	6689744012613532 75	41449738	-7378159936306671 93
16616744	-9122585296606378267	20724869	10061355595079910296
2077093	1817054013491865091	18647776	-2004040079735920651
1494350	-11173490652547381915	582743	12323311762883854602
164432	-11173490652547381915	582743	12323311762883854602
10277	-10105940579551181070	572466	11145904202335863024
10277	-3985420216779389690	275956	4395544540619939934
10277	-9385877478435701482	58712	10351742166502237298
2938	-15358843430660045042	7339	16939363158655645501
1469	-13047417612254912236	5870	14390077365814228570
1469	-4212282987722323312	1466	4645752890065697354
736	-2106141493861161656	733	2322876445032848677
23	8539480200672524210	710	-9418245387968318460
23	14915361794869947971	332	-16450244526985326367
23	1888533973955905 02	60	-2082875769046328 90
8	-9409859147888157279	15	10378191697669085670
1	2565951791988852425	14	-2830004059116479379
1	3801714338634381012	6	-4192934194448560846
1	13876668109194200210	2	-15304662854009845472
1	18914144994474109809	0	-20860527183790487785
	x'		y'

APPENDIX B

SUPPLEMENTARY C CODES

```
typedef signed long si;
#define T 30
#define LIMB 3
#define MLIMB (LIMB+1)
#define vec __m256i
#define VMUL _mm256_mul_epi32
#define VSUB _mm256_sub_epi64
#define VADD _mm256_add_epi64
#define VSHR _mm256_srli_epi64
#define VSLR _mm256_slli_epi64
#define VBLD _mm256_blend_epi32
#define VSFL _mm256_shuffle_epi32
#define VPER _mm256_permute4x64_epi64
const vec ZERO = { 0UL, 0UL, 0UL, 0UL };
const vec ANDMASK = { (1UL << T) - 1, (1UL << T) - 1, (1UL << T) - 1, (1UL << T)
    - 1 };
const vec POSMASK[LIMB] = { { 1UL << (2 * T + 1), 1UL << (2 * T + 1), 1UL
    << (2 * T + 1), 1UL << (2 * T + 1) }, { 1UL << (2 * T + 1), 1UL
    << (2 * T + 1), 1UL << (2 * T + 1), 1UL << (2 * T + 1) }, { 1UL
    << (2 * T + 1), 0UL, 0UL, 0UL } };

const vec NEGMASK[LIMB] = { { 1UL << (T + 1), 1UL << (T + 1), 1UL << (T + 1),
    1UL << (T + 1) }, { 1UL << (T + 1), 1UL << (T + 1), 1UL << (T + 1), 1UL
    << (T + 1) }, { 1UL << (T + 1), 0UL, 0UL, 0UL } };

vec RANDMASK = { 0x00007FFFUL, 0UL, 0UL, 0UL };
void myrand(vec *z, int l) {
  int i, j;
  for (i = 0; i < l; i++) {
    for (j = 0; j < 4; j++) {
      z[i][j] = ((unsigned long) random()) & ((1UL << T) - 1);
    }
  }
  z[0][0] |= 1;
  z[2] &= RANDMASK;
}
```

Code B.1: C Header Code for ModInvAVX2

```
while (IsZero(g)) {
  //reducedRatMod
  //linear transform
  for (i = 0; i < LIMB; i++) {
    f[i] = VADD(f[i], POSMASK[i]);
    g[i] = VADD(g[i], POSMASK[i]);
  }

  for (i = 0; i < LIMB; i++) {
    tf[i] = VSHR(f[i], T);
    tg[i] = VSHR(g[i], T);
  }

  for (i = 0; i < LIMB; i++) {
    f[i] &= ANDMASK;
    g[i] &= ANDMASK;
  }
  //makeodd
  for (i = 0; i < LIMB; i++) {
    f[i] = VSUB(tf[i], NEGMASKF[i]);
    g[i] = VSUB(tg[i], NEGMASKG[i]);
  }
  //swap
}
```

Code B.2: C Main Code for ModInvAVX2

```
for (i = 0; i < LIMB; i++) {
  uf[i] = VMUL(cc, g[i]);
  vg[i] = VMUL(dd, f[i]);
  qf[i] = VMUL(aa, g[i]);
  rg[i] = VMUL(bb, f[i]);
  f[i] = VSUB(uf[i], vg[i]);
  g[i] = VSUB(qf[i], rg[i]);
}
```

Code B.3: C Code for LinearTransform

```
void reducedRatMod(si* a, si* b, si* c, si* d, si u, si v, const
      si tt) {
  si nn = tt;
  si n = 1 << tt;
  si q, r, temp, Ud[1];
  modinv_2e(Ud, v, nn);
  r = (*Ud * u) & (n - 1);
  set4(a, b, c, d, n, 0, r, 1);
  si sqrtn = 1 << (nn / 2);
  while (*a >= (sqrtn)) {
    q = *a / *c;
    a[0] -= q * (*c);
    b[0] -= q * (*d);
    temp = *a; *a = *c; *c = temp;
    temp = *b; *b = *d; *d = temp;
  }
}
```

Code B.4: C Code for reducedRatMod

```
void makeodd(vec* a) {
  int i;
  vec af[MLIMB];
  vec *as;
  for (i = 0; i < MLIMB; i++) {
    af[i] = ZERO;
  }
  int cnt = 0;
  si tmp = a[0][0];
  while (!(tmp & 1)) {
    tmp = tmp >> 1;
    cnt++;
  }
  if (cnt != 0) {
    vec CNTMASK = { (1UL << cnt) - 1, (1UL << cnt) - 1, (1UL <<
         cnt)
       - 1, (1UL << cnt) - 1 };
    for (i = 0; i < LIMB; i++) {
      NEGMASK[i] = VSHR(NEGMASK[i], cnt);
      af[i] = VSLR(a[i] & CNTMASK, T - cnt);
      a[i] = VSHR(a[i], cnt);
    }
    as = (vec *) &af[0][1];
    for (i = 0; i < LIMB; i++) {
      a[i] = VADD(a[i], as[i]);
    }
  }
}
```

Code B.5: C Code for MakeOdd

```
void swap(vec* a, vec* b) {
  vec af[MLIMB], bf[MLIMB];
  int c = 0;
  int i, k;
  for (i = 0; i < MLIMB; i++) {
    af[j] = bf[j] = ZERO;
  }
  for (i = LIMB; i >= 0; i--) {
    af[i] = _mm256_abs_epi32(a[i]);
    bf[i] = _mm256_abs_epi32(b[i]);
    for (k = 3; k >= 0; k--) {
      if (af[i][k] < bf[i][k]) {
        c = 1; k = 0; i = 0;
      } else {
        c = 0; k = 0; i = 0;
      }
    }
  }

  if (c == 1) {
    for (i = 0; i < LIMB; i++) {
      af[i] = a[i]; a[i] = b[i]; b[i] = af[i];
    }
  }
}
```

Code B.6: C Code for Swap