Impact of Intel's New Instruction Sets on Software Implementation of $GF(2)[x]$ Multiplication

Chen Su
School of Software, Tsinghua University
Beijing 100084, PR China
sochat88@gmail.com

Haining Fan
School of Software, Tsinghua University
Beijing 100084, PR China
fhn@mail.tsinghua.edu.cn

Abstract

PCLMULQDQ, a new instruction that supports $GF(2)[x]$ multiplication, was introduced by Intel in 2010. This instruction brings dramatic change to software implementation of multiplication in $GF(2^m)$ fields. In this paper, we present improved Karatsuba formulae for multiplying two small binary polynomials, compare different strategies for PCLMULQDQ-based multiplication in the five $GF(2^m)$ fields recommended by NIST and conclude the best design approaches to software implementation of $GF(2)[x]$ multiplication.

Keywords: $GF(2)[x]$ multiplication, Karatsuba Algorithm, SSE, AVX, PCLMULQDQ

1 Introduction

Multiplication in characteristic-two fields $GF(2^m)$ is a key operation in implementation of many cryptographic applications [1]. Multiplication in $GF(2^m)$ can be divided into two steps: multiplying two binary polynomials and reducing their product to the final result. Traditionally, it is complicated to implement binary polynomial multiplication on x86 processors. The straightforward method is using exclusive OR (XOR) and shift operations (also called shift-and-add method), which is quite a time consuming approach. Another technique is using table-lookup methods to precompute products of two small polynomials. Limited by word size and memory of computers, this approach cannot be efficiently generalized to polynomials of larger degrees. Therefore, accelerating $GF(2)[x]$ multiplication can significantly contribute to achieving high speed secure computing and communication.

In 2010, Intel introduced a new instruction called PCLMULQDQ in their 32nm processor families. The PCLMULQDQ instruction computes the product of two 64-bit operands carrylessly, that is, multiplication of binary polynomials of degree at most 63, producing a polynomial of degree at most 126. According to Agner Fog's manual [2], the instruction can be executed in $8 \sim 14$ clock cycles, which is significantly faster than traditional methods. If we use the instruction as the basic operation in multi-precision polynomial multiplication, we may compute the product of two polynomials in larger size in high speed. In 2011, Intel introduced Advanced Vector Extensions (AVX) instruction set, providing 256-bit wide SIMD registers and new non-destructive instruction syntax, which may bring further speedup.

In a whitepaper from Intel [3], Gueron and Kounavis presented their results of performance comparison between different implementations of AES-GCM, including the one using PCLMULQDQ and AES instructions. The result showed that new instructions offered significant acceleration, but their work did not provide exclusive tests for performance of PCLMULQDQ. More recently, Jonathan Taverne et al. discussed the impact of the instruction on some field arithmetic and elliptic curve scalar multiplication in [4], while a comprehensive analysis for PCLMULQDQ's impact on practical $GF(2)[x]$ multiplication was still missing.

In this work, we discuss strategies for using Intel's new instructions on multiplication of binary polynomials, mainly focusing on polynomials in the five $GF(2^m)$ fields recommended in the FIPS 186-3 standard by National Institute of Standards and Technology (NIST), where $m = 163$, 233, 283, 409 and 571 respectively. We present our improvement to the well-known Karatsuba algorithm. Programming optimization techniques are also discussed systematically. We then compare different implementations of multiplication and give the conclusions on best methods for software implementation.

2 Methods for binary polynomial multiplication

Besides the straightforward method, the two most common methods to improve binary polynomial multiplication are López and Dahab's comb method [5] and Karatsuba algorithm (KA) [6]. The comd method is an improvement to the shift-and-add method, and was considered to be the fastest for fields of practical interest [7]. However, the method cannot take advantage of the PCLMULQDQ instruction, so we will focus on Karatsuba algorithm in this work.

For two binary polynomials $a(x) = a_1(x)x^l + a_0(x)$ and $b(x) = b_1(x)x^l + b_0(x)$ of degrees $m = 2l$, the two-term (basic) Karatsuba formula computes their product as follows:

$$a(x) \cdot b(x) = a_1b_1x^{2l} + [(a_1 + a_0)(b_1 + b_0) + a_1b_1 + a_0b_0]x^l + a_0b_0. \tag{1}$$

This edition is adapted from the original one in [6], since the addition and subtraction are the same in fields of characteristic two. The formula can be applied recursively until reaching some threshold, where a straightforward method (such as shift-and-add or table-lookup) can then be used [7]. The splits usually take place on word boundaries and the threshold is set to word length. In our experiments, we use PCLMULQDQ as the straightforward method, and set the threshold to 64.

Some improved Karatsuba formulae for small non-power-of-2 lengths have been proposed in [8], [9] and [10]. These formulae can be used for fields of practical interest, for example, the five $GF(2^m)$ fields recommended by NIST. When using 64-bit word size, polynomials in these fields can be represented by 3, 4, 5, 7 and 9 words respectively. Based on these Karatsuba formulae, we present the following techniques for further improvement.

2.1 Non-recursive Karatsuba multiplication

Traditionally the four-term KA is done by calling two-term KA recursively twice. We expand the recursion and get the following explicit formula:

$$\begin{aligned}
a(x) \cdot b(x) &= (a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) \\
&= (a_3x + a_2)(b_3x + b_2)x^4 + \{[(a_1 + a_3)x + (a_0 + a_2)][(b_1 + b_3)x + (b_0 + b_2)] \\
&\quad + (a_3x + a_2)(b_3x + b_2) + (a_1x + a_0)(b_1x + b_0)\}x^2 + (a_1x + a_0)(b_1x + b_0) \\
&= m_3x^6 + (m_2 + m_3 + m_5)x^5 + (m_1 + m_2 + m_3 + m_7)x^4 \\
&\quad + (m_0 + m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8)x^3 \\
&\quad + (m_2 + m_1 + m_0 + m_6)x^2 + (m_1 + m_0 + m_4)x + m_0,
\end{aligned} \tag{2}$$

where $m_0 = a_0b_0$, $m_1 = a_1b_1$, $m_2 = a_2b_2$, $m_3 = a_3b_3$, $m_4 = (a_1+a_0)(b_1+b_0)$, $m_5 = (a_3+a_2)(b_3+b_2)$, $m_6 = (a_2+a_0)(b_2+b_0)$, $m_7 = (a_1+a_3)(b_1+b_3)$ and $m_8 = (a_2+a_0+a_1+a_3)(b_2+b_0+b_1+b_3)$. To simplify the formula, we use cubic quadranomials as multiplicands. A general formula can be obtained by replacing x with x^l.

The formula has the same number of multiplications as the recursive edition, but the overhead of recursion is eliminated. Although it needs 56 word additions for multiplication of two 4-word polynomials (more than 40 of the recursive edition), its addition complexity can be further reduced using common subexpression elimination (see Section 2.3).

2.2 Modified formulae

While the two-term KA formula (1) needs 8 word additions for multiplication of two 2-word polynomials, the following trick can reduce the number of additions to 7 [11, 12]:

$$a(x) \cdot b(x) = (a_1 b_1 x^l + a_0 b_0)(x^l + 1) + (a_1 + a_0)(b_1 + b_0)x^l. \tag{3}$$

We generalize it for the three-term formula presented in [13]:

$$\begin{aligned} a(x) \cdot b(x) \;=\; & (a_2 b_2 x^{2l} + a_1 b_1 x^l + a_0 b_0)(x^{2l} + x^l + 1) + (a_1 + a_0)(b_1 + b_0)x^l \\ & + (a_2 + a_0)(b_2 + b_0)x^{2l} + (a_1 + a_2)(b_1 + b_2)x^{3l}. \end{aligned} \tag{4}$$

The formula needs 20 word additions for multiplication of two 3-word polynomials, fewer than 24 in the original one.

Similarly, we also generalize this technique to the four-term formula (2) and five-term formula in [10] to reduce the number of additions:

$$\begin{aligned} a(x) \cdot b(x) \;=\; & (m_3 x^3 + m_2 x^2 + m_1 x + m_0)(x^3 + x^2 + x + 1) + m_8 x^3 \\ & + (m_5 x^2 + m_4)(x^3 + x) + (m_7 x + m_6)(x^3 + x^2) \\ =\; & (m_3 x^3 + m_2 x^2 + m_1 x + m_0)(x + 1)(x^2 + 1) + m_8 x^3 \\ & + (m_5 x^2 + m_4)(x^3 + x) + (m_7 x + m_6)(x^3 + x^2), \end{aligned} \tag{5}$$

where the $m_1, \ldots, m_{13}$ are the same as defined in (2), and

$$\begin{aligned} a(x) \cdot b(x) \;=\; & (m_5 x^3 + m_1)(x + 1)(x^4 + x^2 + 1) + m_{12}(x^3 + x^5) \\ & + (m_4 x^2 + m_3 x + m_2)(x + 1)(x^3 + 1) + (m_{11} x + m_{10})(x^3 + x^4) \\ & + (m_9 x^3 + m_8 x^2 + m_7 x + m_6)(x^4 + x) + m_{13}(x^3 + x^4 + x^5), \end{aligned} \tag{6}$$

where the $m_1, \ldots, m_{13}$ are the same as defined in [10]. The two formulae above need 42 and 85 additions respectively, also fewer than 56 and 102 in the original formulae.

2.3 Common subexpression elimination

Another technique to reduce the number of additions is common subexpression elimination (CSE). However, finding best strategy for eliminating common subexpressions is a hard problem, so we only tried to make such elimination based on observation and optimized code by hand.

The implementation of modified formulae discussed above usually takes more temporary space to store intermediate results. An alternative approach is to split products of parts into two halves and perform CSE on additions of the halves. Roche gave an example for two-term KA in [14]. We now generalize it for three-term formula: let $m_0 = a_0 b_0$, $m_1 = a_1 b_1$, $m_2 = a_2 b_2$, $m_3 = (a_1 + a_0)(b_1 + b_0)$, $m_4 = (a_2 + a_0)(b_2 + b_0)$, $m_5 = (a_1 + a_2)(b_1 + b_2)$, m_{kh} and m_{kl} denote high and low halves of m_k ($k = 0, \ldots, 5$), then we get

$$\begin{aligned} a(x) \cdot b(x) \;=\; & m_2 x^{4l} + (m_2 + m_1 + m_5)x^{3l} + (m_0 + m_1 + m_2 + m_4)x^{2l} \\ & + (m_0 + m_1 + m_3)x^l + m_0 \\ =\; & m_{2h} x^{5l} + (\underline{m_{2l} + m_{1h} + m_{2h}} + m_{5h})x^{4l} \\ & + (m_{0h} + \underline{m_{1h} + m_{2h} + m_{2l}} + m_{4h} + m_{1l} + m_{5l})x^{3l} \\ & + (\mathbf{m_{0l}} + \mathbf{m_{1l}} + \mathbf{m_{0h}} + m_{2l} + m_{4l} + m_{1h} + m_{3h})x^{2l} \\ & + (\mathbf{m_{0l}} + \mathbf{m_{1l}} + \mathbf{m_{0h}} + m_{3l})x^l + m_{0l}. \end{aligned} \tag{7}$$

The common subexpressions in (7) are emphasized. After eliminating them, the formula also costs only 20 additions. Similarly, the number of additions in (2) can be reduced to 38 using this approach.

This method actually changes the granularity of additions form $2l$ to l. Smaller granularity leads to more common subexpressions, thus more additions can be eliminated. For the purpose of comparison, if we perform CSE on the original two-term KA formula, nothing can be eliminated; for the original three-term formula, only 2 additions can be eliminated, with 22 left. However, the difficulty of finding common subexpressions also increases, and the code becomes more complex. Besides, granularity also affects instructions used for additions (see Section 3.1).

3 Implementation issues and experimental results

Based on the original and improved Karatsuba formulae mentioned above, we have different dividing strategies for computing products of two polynomials in the NIST fields using PCLMULQDQ. The strategies are listed in Table 1. The seven-term formula was generated using moduli polynomials from [10]. We implemented multiplication in the five NIST fields using C programming language.[1]

Although we focus on polynomial multiplication in this paper, in order to compare our results with [4], we also adapted and implemented the algorithms for fast reduction in the five NIST fields from [7] for 64-bit word size. The algorithms were also optimized using the double precision shift instruction, SHLD.

3.1 Programming optimization techniques

Optimization techniques in programming include utilization of Intel SSE and AVX instructions. The PXOR instruction in SSE2 instruction set provides 128-bit exclusive OR operation while requiring the same number of clock cycles as 64-bit general purpose instruction XOR, accelerating additions. Using 64-bit word size, the PCLMULQDQ instruction produces a 128-bit product. We used PXOR to perform additions on the products. Although AVX do not provide a 256-bit integer SIMD instruction for exclusive OR operation, two float-point instructions, VXORPD and VXORPS, can both offer the same functionality. Either of them has the same latency as PXOR, but with larger reciprocal throughput (1 cycle rather than 1/3). We also used VXORPD instead of PXOR in some parts of the code to reduce the number of instructions.

As discussed in Section 2.3, the granularity of additions can be changed. When the granularity is a multiple of 128 bits, additions can be implemented using only 128-bit PXOR instruction. However, 64-bit XOR instruction must be used when the granularity is an odd multiple of words, increasing the total number of addition instructions. As a result, the implementation of four-term formula (2) actually contains fewer addition instructions than the recursive edition. For the same reason, the formulae with halved granularity were not used in some strategies, including 3-1, 4-2, 5-1 and 7-1 in Table 1.

Polynomial multiplication may benefit from another feature of AVX: non-destructive instruction syntax. Traditional Intel x86 instructions have destructive syntax, i.e., one of the source operands is used as the destination operand, with its original value destructed. As a result, compilers often have to insert some extra copy operations to save the original value in other registers, thus the number of registers needed increases. AVX introduce the non-destructive instruction syntax for (and only for) SIMD instructions, which adds one more operand as destination. This instruction syntax can reduce amount of unnecessary copy instructions.

We also analyzed two famous open source libraries which were carefully optimized for multiplication of small binary polynomials: NTL [15] and gf2x [16], and utilized some useful

[1] The source code can be found at the end of this paper.

Table 1: Comparison of Karatsuba multiplication strategies (timings in clock cycles)

Size of polynomials	No.	Strategies	Number of word multiplications	Timings	
				gcc	icc
3 words	3-1	Use three-term formula [13]	6	59	61
		Word-by-word "schoolbook" multiplication	9	87	85
4 words	4-1	Use two-term KA recursively twice	9	95	91
	4-2	Use four-term formula (2)	9	88	90
5 words	5-1	Use five-term formula [10]	13	135	135
	5-2	Use two-term KA and divide the polynomial into two parts: 2 words and 3 words respectively	15	160	159
	5-3	Use three-term formula (4) and divide the polynomial into three parts, 1, 2 and 2 words respectively	16	180	176
7 words	7-1	Use seven-term formula	22	252	265
	7-2	Use two-term KA and divide the polynomial into two parts: 3 words and 4 words respectively	24	255	260
	7-3	Use four-term formula (5) and divide the polynomial into four parts: 1, 2, 2 and 2 words respectively	25	298	291
9 words	9-1	Use two-term KA and divide the polynomial into two parts: 4 words and 5 words respectively. Multiplication of 5-word polynomials is done using five-term formula [10].	35	381	380
	9-2	Use three-term formula [13] recursively twice	36	384	384
	9-3	Use five-term formula (6) and divide the polynomial into five parts: 1, 2, 2, 2 and 2 words respectively	37	430	408

techniques found in these libraries for our implementation, such as interleaved storage of the two polynomials and aligning the product of two words to 128-bit before performing additions.

3.2 Results and comparisons

We ran the code to get running times of our implementations. The compilers used were GNU C Compiler 4.6.0 (gcc) and Intel C++ Compiler 12.0.4 (icc). Compiler intrinsics were used instead of assembly language for accessing special instructions. Compiler optimization level was O3, and word size was 64-bit. The target machine was an Intel Sandy Bridge Core i5 2300 (2.80 GHz) that supports PCLMULQDQ and AVX, running Linux with a 2.6.38 kernel. To obtain more accurate results, the system was run in single-user text mode, with Speedstep Technology and Turbo Boost Technology of the CPU disabled.

Table 1 presents timings of different implementations. Our experiments show that optimized implementations using n-term formulae for polynomials of n words offer best performance, since they cost fewest PCLMULQDQ instructions while the increase of additions is acceptable. For three-word polynomials we also tested an optimized edition of the "schoolbook" method, and it is much slower than Karatsuba algorithm. The implementation using four-term formula (2) is slightly better than the recursive edition. Although compilers would perform inlining for the latter and actually generate non-recursive code, the four-term formula can benefit more from SIMD instructions and compiler optimization. Three strategies (5-3, 7-3 and 9-3) were implemented using 256-bit AVX instructions in order to reduce the number of addition instructions, but their performance is still worst due to large amount of multiplications.

Table 2: Comparison between our results and gf2x 1.0 (timings in clock cycles)

	Fields	$GF(2^{163})$	$GF(2^{233})$	$GF(2^{283})$	$GF(2^{409})$	$GF(2^{571})$
Our fastest	gcc	59	88	135	252	381
	icc	61	90	135	265	380
gf2x [16]	gcc	70	99	159	298	449

The gf2x library (version 1.0) includes routines tailored for multiplying polynomials of sizes 1 ~ 9 words, which are also based on PCLMULQDQ instructions and Karatsuba formulae. The library can only be compiled using gcc. We ran the routines for multiplying 3, 4, 5, 7 and 9-word polynomials, which are suited for multiplication in the five NIST fields, and list timings of the fastest ones for each field in Table 2. Our codes take 11% ~ 16% less time than the library.

Table 3: Comparison of field multiplication between our results and [4] (timings in clock cycles)

	Platforms	Compilers	$GF(2^{233})$	$GF(2^{409})$
Our fastest	Westmere	gcc	126	322
		icc	126	304
	Sandy Bridge	gcc	115	288
		icc	116	300
[4]	Westmere	gcc	128	345
		icc	128	348
	Sandy Bridge	gcc	100	270
		icc	100	273

Table 3 compares best results of field multiplication (polynomial multiplication with reduction) in $GF(2^{233})$ and $GF(2^{409})$ that we implemented with results in [4]. Since the latter were obtained on both a Westmere processor and a Sandy Bridge processor, we also list timings of our code run on an Intel Core i5 650 in the table. It is worth noting that Taverne et al. claimed implementations based on multiple-term Karatsuba formulae had lower performance than applying the two-term Karatsuba formula recursively in [4], but that is contrary to our results. Since no source code or detail of implementation is given in their work, we cannot make further analysis.

Table 4 shows the running time of codes implementing the same algorithms, but using different instruction sets (with 128-bit SSE instructions and with 256-bit AVX instructions). The 256-bit VXORPD instruction does not bring significant speedup, which we believe is caused by relatively large reciprocal throughput of the instruction.

Table 5 shows the running time of codes implementing the same algorithms, both using 128-bit SSE instructions, but compiled with -mavx and -msse4.2 options respectively. The former

Table 4: Timings of codes using different instruction sets (in clock cycles)

Strategies		4-1	7-2	9-1	9-2
with SSE	gcc	97	261	391	392
	icc	91	264	382	391
with AVX	gcc	96	255	381	383
	icc	94	260	380	383

Table 5: Timings of codes using destructive syntax and non-destructive syntax (in clock cycles)

Strategies		3-1	4-2	5-1	7-1	9-1
Non-destructive syntax	gcc	59	88	135	252	381
	icc	61	90	135	260	380
Destructive syntax	gcc	61	96	141	253	390
	icc	63	98	136	267	404

uses non-destructive syntax and must be run on CPU supporting AVX, while the latter uses destructive syntax and can be run on CPU of previous generation. Codes with non-destructive syntax are slightly faster, since only SIMD instructions can benefit from it.

4 Conclusions

The utilization of PCLMULQDQ, with some other SIMD instructions, changes the way to implement $GF(2)[x]$ multiplication in software, making Karatsuba algorithms the best choice for the NIST fields. In this paper, we give non-recursive four-term Karatsuba formula and improved three, four and five-term Karatsuba formulae. We discuss different strategies for implementation and optimization of polynomial multiplication in the five NIST fields using Intel's new instructions and compare running time of codes using the strategies. Experimental results show that our improved multiple-term Karatsuba formulae have better performance, and the fastest implementations are better than previous works in [4] and gf2x library. The results also show that current AVX instructions bring slight speedup to $GF(2)[x]$ multiplication.

Acknowledgments

The authors would like to thank the reviewers for their valuable comments and suggestions that helped to improve the presentation of this paper. The work was supported by NSFC under grant number 60970147 and 973 project No.2010CB328004.

References

[1] R. P. Brent, P. Gaudry, E. Thomé, P. Zimmermann, Faster multiplication in $GF(2)[x]$, in: ANTS-VIII 2008, Vol. 5011 of LNCS, Springer-Verlag, 2008.

[2] A. Fog, Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs (2012).
URL `http://www.agner.org/optimize/instruction_tables.pdf`

[3] S. Gueron, M. Kounavis, Intel® carry-less multiplication instruction and its usage for computing the GCM mode (2010).

[4] J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, J. López, Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction, Journal of Cryptographic Engineering 1 (2011) 187–199.

[5] J. López, R. Dahab, High-speed software multiplication in $\mathbb{F}_{2^m}$, in: INDOCRYPT 2000, Vol. 1977 of LNCS, Springer-Verlag, 2000.

[6] A. Karatsuba, Y. Ofman, Multiplication of multidigit numbers on automata, Soviet Physics Doklady 7 (1963) 595.

[7] D. Hankerson, A. Menezes, S. Vanslone, Guide to Elliptic Curve Cryptography, Springer-Verlag, New York, 2004.

[8] P. L. Montgomery, Five, Six, and Seven-Term Karatsuba-Like Formulae, IEEE Transactions on Computers 54 (2005) 362–369.

[9] H. Fan, M. A. Hasan, Comments on “Five, Six, and Seven-Term Karatsuba-Like Formulae”, IEEE Transactions on Computers 56 (2007) 716–717.

[10] M. Cenk, F. Özbudak, Improved Polynomial Multiplication Formulas over $\mathbb{F}_2$ Using Chinese Remainder Theorem, IEEE Transactions on Computers 58 (2009) 572–576.

[11] R. T. Moenck, Practical fast polynomial multiplication, in: the third ACM symposium on Symbolic and algebraic computation, 1976, pp. 136–148.

[12] D. J. Bernstein, Fast multiplication (2000).
URL http://cr.yp.to/talks.html#2000.08.14

[13] A. Weimerskirch, D. Stebila, S. C. Shantz, Generic GF(2^m) arithmetic in software and its application to ECC, in: ACISP 2003, Vol. 2727 of LNCS, Springer-Verlag, 2003.

[14] D. S. Roche, Space- and time-efficient polynomial multiplication, in: the 2009 international symposium on Symbolic and algebraic computation, ACM, 2009, pp. 295–302.

[15] V. Shoup, NTL: A library for doing number theory, http://www.shoup.net/ntl/ (2009).

[16] E. Thomé, P. Zimmermmann, P. Gaudry, R. Brent, gf2x is a library for multiplying polynomials over the binary field, https://gforge.inria.fr/projects/gf2x/ (2011).

```
/*///////////////////////////////////////////////////////////////////////////
//
// This is the source code of GF(2)[x] multiplication algorithms
//  in paper "Impact of Intel's New Instruction Sets on Software
//  Implementation of GF(2)[x] Multiplication" by C. Su and H. Fan.
//  The code is licensed under the GNU General Public License:
//  http://www.gnu.org/licenses/gpl-3.0.txt
//
// Author: Chen Su
//
// Platform: Intel Sandy Bridge processors, Linux kernel 2.6.30 or later, 64-bit only
// Usage: 1. Save the code as a source file (eg. mul.cpp);
//      2. Use GNU C Compiler (4.5 or later) or Intel C++ Compiler
//         (11.1 or later) to compile the source:
//       g++ -O3 -mpclmul -mavx -static -o mul mul.cpp
//       icpc -O3 -ip -inline-level=2 -xAVX -mavx -static -o mul mul.cpp
//      3. Run the executable generated.
// You can change some macros and statements to conduct different tests. See the
//   comments in the code for further information.
////////////////////////////////////////////////////////////////////////////*/

//Macros for compiler intrinsics included (for GCC)
#define __SSE3__ 1
#define __SSSE3__ 1
#define __SSE4_1__ 1
#define __PCLMUL__ 1
#define __AVX__ 1

#define _USE_AVX_ 1 //If defined, use 256-bit AVX instructions
/*
 * To run the code on Westmere processors, comment the two macros above out
 * and replace "avx" with "sse4.2" in compiler parameters
 */

#include <x86intrin.h>
#include <wmmintrin.h>
#include <immintrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

//If defined, align on 32 bytes
//#define _ALIGN_32_ 1

const int WordLength = sizeof(unsigned long)*8;
#define MAX(a,b) (((a)>(b))?(a):(b))
#define MIN(a,b) (((a)<(b))?(a):(b))
#define WINDOW 4
#define pow2(n) ((unsigned long)1 << (n))

//Different copy methods
#define MEM_COPY_SET 1
#ifdef PLAIN_COPY_SET
#define _copy(a, b, n) for(int zqa=0; zqa<(n); zqa++) *((a)+zqa) = *((b)+zqa);;
#elif defined MEM_COPY_SET
#define _copy(a, b, n) memcpy(a, b, (n)*sizeof(unsigned long))
#else
#define _copy(a, b, n) __movsq(a, b, n)
#endif

#ifndef __INTEL_COMPILER
#define _PXOR2 1
#endif
#ifdef _PXOR2
#define xor2(a, b, c, n) for(int zqa=0; zqa<(n)-1; zqa+=2) _mm_storeu_si128((__m128i *)((a)+zqa),\
     _mm_xor_si128(_mm_loadu_si128((__m128i *)((b)+zqa)), _mm_loadu_si128((__m128i *)((c)+zqa))));\
    if((n) & 1) *((a)+(n)-1) = *((b)+(n)-1) ^ *((c)+(n)-1);
#else
#define xor2(a, b, c, n) for(int zqa=0; zqa<(n); ++zqa) *((a)+zqa) = *((b)+zqa) ^ *((c)+zqa);;
#endif

//The two intrinsics below are not supported by icc
#ifdef __INTEL_COMPILER
inline __m128i _mm_set_epi64x(long long q1, long long q0)
{
    return _mm_xor_si128(_mm_cvtsi64_si128(q0), _mm_slli_si128(_mm_cvtsi64_si128(q1), 8));
}

inline __m128i _mm_set1_epi64x(long long q)
{
    return _mm_insert_epi64(_mm_cvtsi64_si128(q), q, 1);
```

```
}
#endif

inline void ui64_m128i(__m128i &v, const unsigned long *i)
{
    v = _mm_loadu_si128((__m128i *)i);
}

inline void m128i_ui64(unsigned long *i, const __m128i &v)
{
    _mm_storeu_si128((__m128i *)i, v);
}

inline void xor_128(__m128i &v, const unsigned long *i)
{
    __m128i v1 = _mm_loadu_si128((const __m128i *)i);
    v = _mm_xor_si128(v, v1);
}

inline void xor_128(unsigned long *i, const __m128i &v)
{
    __m128i v1 = _mm_loadu_si128((__m128i *)i);
    v1 = _mm_xor_si128(v1, _mm_loadu_si128(&v));
    _mm_storeu_si128((__m128i *)i, v1);
    /**i ^= _mm_cvtsi128_si64(v);
    *(i+1) ^= _mm_extract_epi64(v, 1);*/
}
#ifdef __AVX__
inline void xor_256(unsigned long *i, const __m256d &v)
{
    __m256d v1 = _mm256_loadu_pd((double *)i);
    v1 = _mm256_xor_pd(v1, v);
    _mm256_storeu_pd((double *)i, v1);
}
#endif

//Multi-precision shift by SHRD
#define shiftright128(c,a,b,n) __asm__("movq %1, %0; shrd %3, %2, %0;" : "=D"(c) : "r"(a), "r"(b), "J"(n))

// 1-word multiplication
inline void ins_mul1_64(__m128i &a, const unsigned long &b, const unsigned long &c)
{
    __m128i v1, v2;
    v1 = _mm_cvtsi64_si128(b);
    v2 = _mm_cvtsi64_si128(c);
    a = _mm_clmulepi64_si128(v1, v2, 0);
}

// 2-word multiplication
#define _SHIFT 1        //Different ways for computing (a0 + a1)(b0 + b1), see below
inline void ins_mul2_64(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    register __m128i v1, v2;
    __m128i t1, t2, t3, t4;

    ui64_m128i(v1, b);
    ui64_m128i(v2, c);
    t1 = _mm_clmulepi64_si128(v1, v2, 0);
    t2 = _mm_clmulepi64_si128(v1, v2, 0x11);
#ifdef _XOR64
    t3 = _mm_cvtsi64_si128(b[0] ^ b[1]);
    t4 = _mm_cvtsi64_si128(c[0] ^ c[1]);
    v1 = _mm_clmulepi64_si128(t3, t4, 0);
#elif defined _XTR
    t3 = _mm_cvtsi64_si128(_mm_cvtsi128_si64(v1) ^ _mm_extract_epi64(v1, 1));
    t4 = _mm_cvtsi64_si128(_mm_cvtsi128_si64(v2) ^ _mm_extract_epi64(v2, 1));
    v1 = _mm_clmulepi64_si128(t3, t4, 0);
#elif defined _SHIFT
    t3 = _mm_xor_si128(v1, _mm_srli_si128(v1, 8));
    t4 = _mm_xor_si128(v2, _mm_srli_si128(v2, 8));
    v1 = _mm_clmulepi64_si128(t3, t4, 0);
#elif defined _SHUF
    t4 = _mm_castpd_si128(_mm_shuffle_pd(_mm_castsi128_pd(v1), _mm_castsi128_pd(v2), 1));
    t3 = _mm_xor_si128(v1, t4);
    t4 = _mm_xor_si128(v2, t4);
    v1 = _mm_clmulepi64_si128(t3, t4, 16);
#elif defined _UPK
    t3 = _mm_unpackhi_epi64(v1, v2);
    t4 = _mm_xor_si128(t3, _mm_unpacklo_epi64(v1, v2));
    v1 = _mm_clmulepi64_si128(t4, t4, 1);
#endif
    v2 = _mm_xor_si128(_mm_xor_si128(v1, t2), t1);
    t1 = _mm_xor_si128(t1, _mm_slli_si128(v2, 8));
    t2 = _mm_xor_si128(t2, _mm_srli_si128(v2, 8));
    m128i_ui64(a, t1);
    m128i_ui64(a + 2, t2);
}

#define xor_2(x,y,z) _mm_storeu_si128((__m128i *)(x), _mm_xor_si128(_mm_loadu_si128((__m128i *)(y)), \
    _mm_loadu_si128((__m128i *)(z))))
#define xor4(a,b) _mm_store_si128(a, _mm_xor_si128(_mm_load_si128(a), _mm_load_si128(b))); \
    _mm_store_si128((a+1), _mm_xor_si128(_mm_load_si128(a+1), _mm_load_si128(b+1)));

#ifdef __AVX__
    #define xor4_avx(x,y,z) _mm256_storeu_pd((double *)(x), \
            _mm256_xor_pd(_mm256_loadu_pd((double *)(y)), _mm256_loadu_pd((double *)(z))))
    #if defined _ALIGN_32_
        #define xor4a_avx(x,y,z) _mm256_store_pd((double *)(x), \
                _mm256_xor_pd(_mm256_load_pd((double *)(y)), _mm256_load_pd((double *)(z))))
    #else
        #define xor4a_avx xor4_avx
    #endif

    inline __m256d ins_mul_128(const __m256d &b, const __m256d &c, const int imm8)
    {
        register __m128i v1, v2;
        unsigned long a[4];
        __m128i t1, t2, t3, t4;

        v1 = _mm256_castsi256_si128(_mm256_castpd_si256(b));
        v2 = _mm256_extractf128_si256(_mm256_castpd_si256(c), 1);
        t1 = _mm_clmulepi64_si128(v1, v2, 0);
        t2 = _mm_clmulepi64_si128(v1, v2, 0x11);
        t3 = _mm_xor_si128(v1, _mm_srli_si128(v1, 8));
        t4 = _mm_xor_si128(v2, _mm_srli_si128(v2, 8));
        m128i_ui64(a, t1);
        m128i_ui64(a + 2, t2);
        ui64_m128i(v1, a + 1);
        m128i_ui64(a + 1, _mm_xor_si128(v1, _mm_xor_si128(_mm_xor_si128(_mm_clmulepi64_si128(t3, t4, 0), t2), t1)));
        return _mm256_load_pd((double *)a);
    }
#endif

// 3-word multiplication
inline void ins_mul3_64(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    __m128i v1, v2;
    __m128i t1, t2, t3;
    __m128i t4, t5, t6, t7, t8;

    t3 = _mm_set_epi64x(c[2], b[2]);
    t7 = _mm_clmulepi64_si128(t3, t3, 1);   //a2b2
    t4 = _mm_loadu_si128((__m128i *)b);
    t5 = _mm_loadu_si128((__m128i *)c);
    t8 = _mm_clmulepi64_si128(t4, t5, 0);       //a0b0
    t6 = _mm_clmulepi64_si128(t4, t5, 0x11);        //a1b1
    v1 = _mm_unpacklo_epi64(t4, t5);
    v2 = _mm_unpackhi_epi64(t4, t5);

    t4 = _mm_xor_si128(v1, v2);
    t1 = _mm_clmulepi64_si128(t4, t4, 1);
    t5 = _mm_xor_si128(t3, v2);
    t2 = _mm_clmulepi64_si128(t5, t5, 1);
    v1 = _mm_xor_si128(v1, t3);
    t3 = _mm_clmulepi64_si128(v1, v1, 1);


    v1 = _mm_xor_si128(t6, t8);
    t1 = _mm_xor_si128(t1, v1);
    t2 = _mm_xor_si128(t2, _mm_xor_si128(t6, t7));
    t3 = _mm_xor_si128(t3, _mm_xor_si128(v1, t7));
    t8 = _mm_xor_si128(t8, _mm_slli_si128(t1, 8));
    t7 = _mm_xor_si128(t7, _mm_srli_si128(t2, 8));
    t3 = _mm_xor_si128(t3, _mm_alignr_epi8(t2, t1, 8));
    m128i_ui64(a, t8);
    m128i_ui64(a+4, t7);
    m128i_ui64(a+2, t3);
}


//3-word multiplication (naive)
void naive163(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    __m128i ab0, ab1, ab2;
    ab0 = _mm_set_epi64x(b[0], a[0]);
    ab1 = _mm_set_epi64x(b[1], a[1]);
    ab2 = _mm_set_epi64x(b[2], a[2]);
    __m128i p, q, r;
    p = _mm_clmulepi64_si128(ab0, ab0, 1);
    _mm_storeu_si128((__m128i *)c, p);
    q = _mm_xor_si128(_mm_clmulepi64_si128(ab1, ab1, 1), _mm_clmulepi64_si128(ab0, ab2, 1));
    r = _mm_xor_si128(q, _mm_clmulepi64_si128(ab0, ab2, 16));
    _mm_storeu_si128((__m128i *)(c+2), r);
    q = _mm_clmulepi64_si128(ab2, ab2, 1);
    _mm_storeu_si128((__m128i *)(c+4), q);
    xor_128(c+1, _mm_xor_si128(_mm_clmulepi64_si128(ab0, ab1, 1), _mm_clmulepi64_si128(ab0, ab1, 16)));
    xor_128(c+3, _mm_xor_si128(_mm_clmulepi64_si128(ab2, ab1, 1), _mm_clmulepi64_si128(ab2, ab1, 16)));
}

// 4-word multiplication (recursive)
void ins_mul4_64(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    unsigned long t1[2], t2[2], t[4];
    ins_mul2_64(a, b, c);
    ins_mul2_64(a + 4, b + 2, c + 2);
    m128i_ui64(t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+2))));
    m128i_ui64(t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)c), _mm_loadu_si128((__m128i *)(c+2))));
    ins_mul2_64(t, t1, t2);
    __m128i *p = (__m128i *)a, *q = (__m128i *)t;
    xor4(q, p);
    p = (__m128i *)(a+4);
    xor4(q, p);
    p = (__m128i *)(a+2);
    xor4(p, q);
}

// 4-word multiplication (non-recursive)
void mul4_64_0(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    register __m128i v1, v2, t1, t2;
    __m128i m[9], bc0, bc1, bc2, bc3;

    ui64_m128i(v1, b);
    ui64_m128i(v2, c);
    bc0 = _mm_unpacklo_epi64(v1, v2);
    bc1 = _mm_unpackhi_epi64(v1, v2);
    ui64_m128i(t1, b+2);
    ui64_m128i(t2, c+2);
    bc2 = _mm_unpacklo_epi64(t1, t2);
    bc3 = _mm_unpackhi_epi64(t1, t2);
    /*
    bc[0] = _mm_set_epi64x(c[0], b[0]);
    bc[1] = _mm_set_epi64x(c[1], b[1]);
    bc[2] = _mm_set_epi64x(c[2], b[2]);
    bc[3] = _mm_set_epi64x(c[3], b[3]);
    */

    m[0] = _mm_clmulepi64_si128(bc0, bc0, 1);
    m[1] = _mm_clmulepi64_si128(bc1, bc1, 1);
    m[2] = _mm_clmulepi64_si128(bc2, bc2, 1);
    m[3] = _mm_clmulepi64_si128(bc3, bc3, 1);
    t1 = _mm_xor_si128(bc0, bc1);
    t2 = _mm_xor_si128(bc2, bc3);
    m[4] = _mm_clmulepi64_si128(t1, t1, 1);
    m[5] = _mm_clmulepi64_si128(t2, t2, 1);
    t1 = _mm_xor_si128(bc0, bc2);
    t2 = _mm_xor_si128(bc1, bc3);
    m[6] = _mm_clmulepi64_si128(t1, t1, 1);
    m[7] = _mm_clmulepi64_si128(t2, t2, 1);
    v1 = _mm_xor_si128(t1, t2);
    m[8] = _mm_clmulepi64_si128(v1, v1, 1);


    t1 = _mm_xor_si128(m[0], m[1]);
    t2 = _mm_xor_si128(m[2], m[6]);
    v1 = _mm_xor_si128(t1, m[4]);
    bc0 = _mm_xor_si128(m[0], _mm_slli_si128(v1, 8));
    bc1 = _mm_xor_si128(t1, t2);
    m128i_ui64(a, bc0);
    t1 = _mm_xor_si128(m[2], m[3]);
    t2 = _mm_xor_si128(m[1], m[7]);
    v2 = _mm_xor_si128(t1, m[5]);
    bc2 = _mm_xor_si128(t1, t2);
    bc3 = _mm_xor_si128(m[3], _mm_srli_si128(v2, 8));
    m128i_ui64(a+6, bc3);
    t1 = _mm_xor_si128(m[6], m[7]);
    t2 = _mm_xor_si128(_mm_xor_si128(v1, v2), _mm_xor_si128(t1, m[8]));
    m128i_ui64(a+2, _mm_xor_si128(bc1, _mm_alignr_epi8(t2, v1, 8)));
    m128i_ui64(a+4, _mm_xor_si128(bc2, _mm_alignr_epi8(v2, t2, 8)));
}

// 4-word multiplication (recursive improved)
void mul4_64_1(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    unsigned long t1[2], t2[2], t[4];
    ins_mul2_64(a, b, c);
    ins_mul2_64(a + 4, b + 2, c + 2);
    m128i_ui64(t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+2))));
    m128i_ui64(t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)c), _mm_loadu_si128((__m128i *)(c+2))));
    ins_mul2_64(t, t1, t2);
    __m128i *p = (__m128i *)(a+2), *q = (__m128i *)(a+4);
    __m128i v = _mm_xor_si128(_mm_loadu_si128(p), _mm_loadu_si128(q));

    __m128i t3, t4;
    ui64_m128i(t3, a);
    ui64_m128i(t4, a+6);
    t3 = _mm_xor_si128(t3, v);
    t4 = _mm_xor_si128(t4, v);
    ui64_m128i(v, t);
    m128i_ui64(a+2, _mm_xor_si128(t3, v));
    ui64_m128i(v, t+2);
    m128i_ui64(a+4, _mm_xor_si128(t4, v));
}

// 4-word multiplication (recursive using 256-bit avx)
#ifdef __AVX__
void mul4_64_avx(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    unsigned long t1[2], t2[2];
#if defined _ALIGN_32_ && defined __INTEL_COMPILER
    __declspec(align(32))
#endif
    unsigned long t[4];
    ins_mul2_64(a, b, c);
    ins_mul2_64(a + 4, b + 2, c + 2);
    m128i_ui64(t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+2))));
    m128i_ui64(t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)c), _mm_loadu_si128((__m128i *)(c+2))));
    ins_mul2_64(t, t1, t2);

    double *p = (double *)a, *q = (double *)t;
    _mm256_store_pd(q, _mm256_xor_pd(_mm256_load_pd(q), _mm256_load_pd(p)));
    p = (double *)(a+4);
    _mm256_store_pd(q, _mm256_xor_pd(_mm256_load_pd(q), _mm256_load_pd(p)));
    p = (double *)(a+2);
    _mm256_storeu_pd(p, _mm256_xor_pd(_mm256_loadu_pd(p), _mm256_load_pd(q)));
}
#endif

// 5-word multiplication (strategy 5-1)
void ins_mul5_64(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    register __m128i v1, v2, t1, t2;
    __m128i m[14], bc[5];

    bc[0] = _mm_set_epi64x(c[0], b[0]);
    bc[1] = _mm_set_epi64x(c[1], b[1]);
    bc[2] = _mm_set_epi64x(c[2], b[2]);
    bc[3] = _mm_set_epi64x(c[3], b[3]);
    bc[4] = _mm_set_epi64x(c[4], b[4]);

    ui64_m128i(v1, b);
    ui64_m128i(v2, c);
    m[1] = _mm_clmulepi64_si128(v1, v2, 0);
    m[2] = _mm_clmulepi64_si128(v1, v2, 0x11);
    ui64_m128i(v1, b+2);
    ui64_m128i(v2, c+2);
    m[3] = _mm_clmulepi64_si128(v1, v2, 0);
    m[4] = _mm_clmulepi64_si128(v1, v2, 0x11);
    m[5] = _mm_clmulepi64_si128(bc[4], bc[4], 1);
    v1 = _mm_xor_si128(bc[0], bc[1]);
    m[6] = _mm_clmulepi64_si128(v1, v1, 1);
    v2 = _mm_xor_si128(bc[0], bc[2]);
    m[7]= _mm_clmulepi64_si128(v2, v2, 1);
    t1 = _mm_xor_si128(bc[4], bc[2]);
    m[8] = _mm_clmulepi64_si128(t1, t1, 1);
    t2 = _mm_xor_si128(bc[3], bc[4]);
    m[9] = _mm_clmulepi64_si128(t2, t2, 1);
    v2 = _mm_xor_si128(v2, bc[3]);
    m[10]= _mm_clmulepi64_si128(v2, v2, 1);
    t1 = _mm_xor_si128(t1, bc[1]);
    m[11] = _mm_clmulepi64_si128(t1, t1, 1);
    v1 = _mm_xor_si128(v1, t2);
    m[12] = _mm_clmulepi64_si128(v1, v1, 1);
    v1 = _mm_xor_si128(v1, bc[2]);
    m[13] = _mm_clmulepi64_si128(v1, v1, 1);

    _mm_store_si128((__m128i *)a, m[1]);


    _mm_store_si128((__m128i *)(a+8), m[5]);
    m[0] = _mm_xor_si128(m[1], m[2]);
    m128i_ui64(a+2, _mm_xor_si128(m[0], _mm_xor_si128(m[7], m[3])));
    v1 = _mm_xor_si128(m[0], m[6]);

    m[0] = _mm_xor_si128(m[4], m[5]);
    m128i_ui64(a+6, _mm_xor_si128(m[0], _mm_xor_si128(m[8], m[3])));
    v2 = _mm_xor_si128(m[0], m[9]);

    register __m128i t;
    t = _mm_xor_si128(m[13], v1);
    t = _mm_xor_si128(t, v2);
    t = _mm_xor_si128(t, m[10]);
    t = _mm_xor_si128(t, m[11]);
    _mm_store_si128((__m128i *)(a+4), t);

    t = _mm_xor_si128(m[13], m[12]);
    m[0] = _mm_xor_si128(_mm_load_si128((__m128i *)(a+2)), t);
    t = _mm_xor_si128(_mm_load_si128((__m128i *)(a+6)), t);
    m[0] = _mm_xor_si128(m[0], m[5]);
    m[0] = _mm_xor_si128(m[0], m[11]);
    t = _mm_xor_si128(t, m[1]);
    t = _mm_xor_si128(t, m[10]);
    xor_128(a+5, m[0]);
    xor_128(a+3, t);
    xor_128(a+1, v1);
    xor_128(a+7, v2);
}

// 5-word multiplication (strategy 5-2)
void mul5_64_1(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    unsigned long t1[3], t2[3], t3[6];
    ins_mul3_64(c, a, b);
    ins_mul2_64(c + 6, a+3, b+3);
    _mm_storeu_si128((__m128i *)t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)a), _mm_loadu_si128((__m128i *)(a+3))));
    t1[2] = a[2];
    _mm_storeu_si128((__m128i *)t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+3))));
    t2[2] = b[2];
    ins_mul3_64(t3, t1, t2);
    xor2(t3, t3, c, 6);
    xor2(t3, t3, c + 6, 4);
    xor2(c + 3, c + 3, t3, 6);
}

// 5-word multiplication (strategy 5-2, modified)
void mul5_64_2(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    unsigned long t1[3], t2[3], t3[6];
    ins_mul3_64(c, a, b);
    ins_mul2_64(c + 6, a+3, b+3);
    _mm_storeu_si128((__m128i *)t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)a), _mm_loadu_si128((__m128i *)(a+3))));
    t1[2] = a[2];
    _mm_storeu_si128((__m128i *)t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+3))));
    t2[2] = b[2];
    ins_mul3_64(t3, t1, t2);

    unsigned long t4[7];
    _copy(t4, c, 6);
    xor2(t4 + 3, t4 + 3, c + 6, 3);
    t4[6] = c[9];
    xor2(c + 3, t4, t3, 6);
    memset(c, 0, 3 * sizeof(unsigned long));
    xor2(c, c, t4, 7);
}

// 5-word multiplication (strategy 5-3)
#ifdef __AVX__
void mul5_128_avx(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    __m128i t;
```

```
#if defined _ALIGN_32_ && defined __INTEL_COMPILER
    __declspec(align(32))
#endif
    unsigned long t1[4], t2[4], t3[4];
    ins_mul2_64(c + 4, a + 2, b + 2);
    ins_mul1_64(t, a[4], b[4]);
    m128i_ui64(c + 8, t);
    xor_2(c, a, a+2);
    xor_2(c+2, b, b+2);
    ins_mul2_64(t1, c, c+2);
    _mm_store_si128((__m128i *)c, _mm_xor_si128(_mm_loadu_si128((__m128i *)(b+2)), _mm_cvtsi64_si128(b[4])));
    _mm_store_si128((__m128i *)(c+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+2)), _mm_cvtsi64_si128(a[4])));
    ins_mul2_64(t2, c, c+2);
    _mm_store_si128((__m128i *)c, _mm_xor_si128(_mm_loadu_si128((__m128i *)(b)), _mm_cvtsi64_si128(b[4])));
    _mm_store_si128((__m128i *)(c+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a)), _mm_cvtsi64_si128(a[4])));
    ins_mul2_64(t3, c, c+2);
    ins_mul2_64(c, b, a);

    xor4a_avx(t1, t1, c);
    xor4a_avx(t1, t1, c+4);
    xor_2(t2, t2, c+8);
    xor4a_avx(t2, t2, c+4);
    xor4a_avx(t3, t3, c);
    xor_2(t3, t3, c+8);
    xor4a_avx(c+4, c+4, t3);
    xor4_avx(c+2, c+2, t1);
    xor4_avx(c+6, c+6, t2);
}
#endif

// 7-word multiplication (strategy 7-1)
void mul7_64_0(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    __m128i m[22], bc[7];
    __m128i v1, v2;
    __m128i t1, t2, t3, t4, t5, t6, t7, t8;

    bc[0] = _mm_set_epi64x(c[0], b[0]);
    bc[1] = _mm_set_epi64x(c[1], b[1]);
    bc[2] = _mm_set_epi64x(c[2], b[2]);
    bc[3] = _mm_set_epi64x(c[3], b[3]);
    bc[4] = _mm_set_epi64x(c[4], b[4]);
    bc[5] = _mm_set_epi64x(c[5], b[5]);
    bc[6] = _mm_set_epi64x(c[6], b[6]);

    ui64_m128i(v1, b);
    ui64_m128i(v2, c);
    m[1] = _mm_clmulepi64_si128(v1, v2, 0); //p0
    m[2] = _mm_clmulepi64_si128(v1, v2, 0x11);      //p1
    t1 = _mm_xor_si128(v1, _mm_xor_si128(_mm_slli_si128(v1, 8), _mm_cvtsi64_si128(b[2])));
    t2 = _mm_xor_si128(v2, _mm_xor_si128(_mm_slli_si128(v2, 8), _mm_cvtsi64_si128(c[2])));
    m[4] = _mm_clmulepi64_si128(t1, t2, 0); //p02
    m[3] = _mm_clmulepi64_si128(t1, t2, 0x11);      //p01
    ui64_m128i(v1, b+4);
    ui64_m128i(v2, c+4);
    m[5] = _mm_clmulepi64_si128(v1, v2, 0); //p4
    m[6] = _mm_clmulepi64_si128(v1, v2, 0x11);      //p5
    t3 = _mm_xor_si128(v1, _mm_set1_epi64x(b[6]));
    t4 = _mm_xor_si128(v2, _mm_set1_epi64x(c[6]));
    m[7] = _mm_clmulepi64_si128(t3, t4, 0); //p46
    m[8] = _mm_clmulepi64_si128(t3, t4, 0x11);      //p56
    v1 = _mm_xor_si128(t1, t3);
    v2 = _mm_xor_si128(t2, t4);
    m[9] = _mm_clmulepi64_si128(v1, v2, 0);  //p0246
    t5 = _mm_xor_si128(_mm_xor_si128(bc[1], bc[3]), bc[5]);
    m[16] = _mm_clmulepi64_si128(t5, t5, 1);    //p135
    t6 = _mm_xor_si128(t5, bc[4]);
    m[17] = _mm_clmulepi64_si128(t6, t6, 1);    //p1345
    m[18] = _mm_clmulepi64_si128(_mm_xor_si128(v1, t5), _mm_xor_si128(v2, _mm_srli_si128(t5, 8)), 0);   //p0123456
    v1 = _mm_xor_si128(_mm_srli_si128(t1, 8), t3);
    v2 = _mm_xor_si128(_mm_srli_si128(t2, 8), t4);
    t7 = _mm_xor_si128(_mm_insert_epi64(v1, 0, 1), _mm_slli_si128(v2, 8));
    m[15] = _mm_clmulepi64_si128(t7, t7, 1);    //p0146
    t8 = _mm_xor_si128(t7, bc[3]);
    m[10] = _mm_clmulepi64_si128(t8, t8, 1);  //p01346
    t7 = _mm_xor_si128(bc[3], bc[0]);
    t7 = _mm_xor_si128(t7, _mm_srli_si128(t3, 8));
    t7 = _mm_xor_si128(t7, _mm_insert_epi64(t4, 0, 0));
    m[11] = _mm_clmulepi64_si128(t7, t7, 1);   //p0356
    t7 = _mm_xor_si128(t7, bc[2]);
    m[12] = _mm_clmulepi64_si128(t7, t7, 1);   //p02356
    m[13] = _mm_clmulepi64_si128(bc[2], bc[2], 1);     //p2
    m[14] = _mm_clmulepi64_si128(bc[6], bc[6], 1);     //p6
    t5 = _mm_xor_si128(_mm_xor_si128(bc[2], bc[4]), bc[5]);
    t6 = _mm_xor_si128(t5, bc[1]);
    m[19] = _mm_clmulepi64_si128(t6, t6, 1);   //p1245
    t7 = _mm_xor_si128(t5, bc[6]);
    m[20] = _mm_clmulepi64_si128(t7, t7, 1);   //p2456
    t5 = _mm_xor_si128(bc[2], bc[3]);
    t6 = _mm_xor_si128(t5, _mm_xor_si128(bc[0], bc[4]));
    m[21] = _mm_clmulepi64_si128(t6, t6, 1);     //p0234
    t7 = _mm_xor_si128(t5, _mm_xor_si128(bc[1], bc[6]));
    m[0] = _mm_clmulepi64_si128(t7, t7, 1);         //p1236

    _mm_store_si128((__m128i *)a, m[1]);
    _mm_store_si128((__m128i *)(a+12), m[14]);
    t1 = _mm_xor_si128(m[1], m[2]);
    t2 = _mm_xor_si128(m[4], m[13]);
    t8 = _mm_xor_si128(t1, t2);
    _mm_store_si128((__m128i *)(a+2), t8);
    _mm_store_si128((__m128i *)(a+4), t8);
    t8 = _mm_xor_si128(t1, m[3]);
    xor_128(a+1, t8);

    t7 = _mm_xor_si128(t2, m[3]);
    _mm_store_si128((__m128i *)(a+6), t7);
    _mm_store_si128((__m128i *)(a+8), t7);

    t5 = _mm_xor_si128(m[6], m[14]);
    t6 = _mm_xor_si128(m[5], m[7]);
    t4 = _mm_xor_si128(t5, t6);
    _mm_store_si128((__m128i *)(a+10), t4);
    xor_128(a+11, _mm_xor_si128(t5, m[8]));
    t3 = _mm_xor_si128(m[8], m[14]);
    t1 = _mm_xor_si128(m[15], m[17]);
    t5 = _mm_xor_si128(m[16], m[9]);
    v1 = _mm_xor_si128(m[8], m[6]);
    xor_128(a+7, _mm_xor_si128(_mm_xor_si128(t7, t5), _mm_xor_si128(_mm_xor_si128(t6, t3),
        _mm_xor_si128(m[20], t1))));
    t1 = _mm_xor_si128(m[12], t1);
    v2 = _mm_xor_si128(m[0], _mm_xor_si128(m[19], m[21]));
    xor_128(a+5, _mm_xor_si128(_mm_xor_si128(t1, v2), _mm_xor_si128(_mm_xor_si128(t6, t3),
        _mm_xor_si128(m[2], m[3]))));
    xor_128(a+6, _mm_xor_si128(_mm_xor_si128(v1, v2), _mm_xor_si128(m[10], m[20])));
    t3 = _mm_xor_si128(m[10], _mm_xor_si128(m[18], m[21]));
    xor_128(a+8, _mm_xor_si128(_mm_xor_si128(_mm_xor_si128(m[1], m[6]), t6),
        _mm_xor_si128(m[0], _mm_xor_si128(t1, t3))));
    xor_128(a+4, _mm_xor_si128(_mm_xor_si128(t1, t5), _mm_xor_si128(_mm_xor_si128(m[18], m[20]),
        _mm_xor_si128(v1, m[19]))));
    xor_128(a+9, _mm_xor_si128(_mm_xor_si128(_mm_xor_si128(m[11], m[17]), _mm_xor_si128(m[2], m[19])),
        _mm_xor_si128(_mm_xor_si128(t4, t2), _mm_xor_si128(t3, t5))));
    v1 = _mm_xor_si128(_mm_xor_si128(m[11], m[17]), m[20]);
    v2 = _mm_xor_si128(_mm_xor_si128(m[0], m[10]), m[12]);
    xor_128(a+3, _mm_xor_si128(_mm_xor_si128(t6, m[8]), _mm_xor_si128(_mm_xor_si128(t5, t8),
        _mm_xor_si128(v1, v2))));
}

// 7-word multiplication (strategy 7-2)
void mul7_64_1(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    unsigned long t1[4], t2[4], t3[8];
    /*ins_mul4_64*/mul4_64_0(c, a, b);
    ins_mul3_64(c+8, a+4, b+4);
    m128i_ui64(t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)a), _mm_loadu_si128((__m128i *)(a+4))));
    m128i_ui64(t1+2, _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+2)), _mm_cvtsi64_si128(*(a+6))));
    m128i_ui64(t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+4))));
    m128i_ui64(t2+2, _mm_xor_si128(_mm_loadu_si128((__m128i *)(b+2)), _mm_cvtsi64_si128(*(b+6))));
    /*ins_mul4_64*/mul4_64_0(t3, t1, t2);
#ifndef _USE_AVX_
    xor2(t3, t3, c, 8);
    xor2(t3, t3, c + 8, 6);
    xor2(c + 4, c + 4, t3, 8);
#else
    _mm256_store_pd((double *)t3, _mm256_xor_pd(*(__m256d *)t3, _mm256_loadu_pd((double *)c)));
    _mm256_store_pd((double *)(t3+4), _mm256_xor_pd(*(__m256d *)(t3+4), _mm256_loadu_pd((double *)(c+4))));
    _mm256_store_pd((double *)t3, _mm256_xor_pd(*(__m256d *)t3, _mm256_loadu_pd((double *)(c+8))));
    _mm_store_si128((__m128i *)(t3+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+4)),
        _mm_loadu_si128((__m128i *)(c+12))));
    _mm256_storeu_pd((double *)(c+4), _mm256_xor_pd(_mm256_loadu_pd((double *)(c+4)), *(__m256d *)(t3)));
    _mm256_storeu_pd((double *)(c+8), _mm256_xor_pd(_mm256_loadu_pd((double *)(c+8)), *(__m256d *)(t3+4)));
#endif
}

// 7-word multiplication (strategy 7-3, without using 256-bit avx)
```

```
void mul7_64_2(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    unsigned long t1[4], t2[4], t3[4], t[4];

    ins_mul2_64(a + 4, b+2, c + 2);
    ins_mul2_64(a + 8, b + 4, c + 4);
    m128i_ui64(a, _mm_xor_si128(_mm_loadu_si128((__m128i *)b), _mm_loadu_si128((__m128i *)(b+2))));
    m128i_ui64(a+2, _mm_xor_si128(_mm_loadu_si128((__m128i *)c), _mm_loadu_si128((__m128i *)(c+2))));
    ins_mul2_64(t1, a, a+2);
    m128i_ui64(a, _mm_xor_si128(_mm_loadu_si128((__m128i *)(b+2)), _mm_loadu_si128((__m128i *)(b+4))));
    m128i_ui64(a+2, _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+2)), _mm_loadu_si128((__m128i *)(c+4))));
    ins_mul2_64(t2, a, a+2);
    m128i_ui64(a, _mm_xor_si128(_mm_loadu_si128((__m128i *)(b+4)), _mm_loadu_si128((__m128i *)(b))));
    m128i_ui64(a+2, _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+4)), _mm_loadu_si128((__m128i *)(c))));
    ins_mul2_64(t3, a, a+2);
    ins_mul2_64(a, b, c);

    xor_2(t+2, a+8, a+6);
    xor_2(t+2, t+2, a+10);
    xor_2(t, a, a+2);
    xor_2(t, t, a+4);
    xor_2(t3, t3, a+6);
    xor_2(t3, t3, a+8);
    xor_2(t3+2, t3+2, a+2);
    xor_2(t3+2, t3+2, a+4);
    m128i_ui64((a+2), _mm_load_si128((__m128i *)t));
    m128i_ui64((a+4), _mm_load_si128((__m128i *)t));
    m128i_ui64((a+6), _mm_load_si128((__m128i *)(t+2)));
    m128i_ui64((a+8), _mm_load_si128((__m128i *)(t+2)));
    xor4((__m128i *)(a+4), (__m128i *)t3);
    xor4((__m128i *)(a+2), (__m128i *)t1);
    xor4((__m128i *)(a+6), (__m128i *)t2);

    __m128i v1, v2, bc6, bb, cc;
    ins_mul1_64(v1, b[6], c[6]);
    m128i_ui64(a+12, v1);

    bc6 = _mm_set_epi64x(c[6], b[6]);
    ui64_m128i(bb, b);
    ui64_m128i(cc, c);

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x10);
    v2 = _mm_clmulepi64_si128(cc, bc6, 0);
    m128i_ui64((a+6), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+6)), _mm_xor_si128(v1, v2)));

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x11);
    v2 = _mm_clmulepi64_si128(cc, bc6, 1);
    m128i_ui64((a+7), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+7)), _mm_xor_si128(v1, v2)));
    ui64_m128i(bb, b+2);
    ui64_m128i(cc, c+2);

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x10);
    v2 = _mm_clmulepi64_si128(cc, bc6, 0);
    m128i_ui64((a+8), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+8)), _mm_xor_si128(v1, v2)));

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x11);
    v2 = _mm_clmulepi64_si128(cc, bc6, 1);
    m128i_ui64((a+9), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+9)), _mm_xor_si128(v1, v2)));
    ui64_m128i(bb, b+4);
    ui64_m128i(cc, c+4);

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x10);
    v2 = _mm_clmulepi64_si128(cc, bc6, 0);
    m128i_ui64((a+10), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+10)), _mm_xor_si128(v1, v2)));

    v1 = _mm_clmulepi64_si128(bb, bc6, 0x11);
    v2 = _mm_clmulepi64_si128(cc, bc6, 1);
    m128i_ui64((a+11), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+11)), _mm_xor_si128(v1, v2)));

}

// 7-word multiplication (strategy 7-3, using 256-bit avx)
#ifdef __AVX__
void mul7_128_avx(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    __m256d v1, v2, t1, t2;
    __m256d m[9], bc[4];

    bc[0] = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(
            _mm_loadu_si128((__m128i *)b)), _mm_loadu_si128((__m128i *)c), 1));
    bc[1] = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(
            _mm_loadu_si128((__m128i *)(b+2))), _mm_loadu_si128((__m128i *)(c+2)), 1));
    bc[2] = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(
            _mm_loadu_si128((__m128i *)(b+4))), _mm_loadu_si128((__m128i *)(c+4)), 1));
    bc[3] = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(
            _mm_cvtsi64_si128(b[6])), _mm_cvtsi64_si128(c[6]), 1));

    t1 = _mm256_xor_pd(bc[0], bc[1]);
    t2 = _mm256_xor_pd(bc[2], bc[3]);
    ins_mul2_64((unsigned long *)m, (unsigned long *)bc, (unsigned long *)bc + 2);
    ins_mul2_64((unsigned long *)(m+1), (unsigned long *)(bc+1), (unsigned long *)(bc+1) + 2);
    ins_mul2_64((unsigned long *)(m+2), (unsigned long *)(bc+2), (unsigned long *)(bc+2) + 2);
    ins_mul2_64((unsigned long *)(m+3), (unsigned long *)(bc+3), (unsigned long *)(bc+3) + 2);
    ins_mul2_64((unsigned long *)(m+4), (unsigned long *)(&t1), (unsigned long *)(&t1) + 2);
    ins_mul2_64((unsigned long *)(m+5), (unsigned long *)(&t2), (unsigned long *)(&t2) + 2);

    t1 = _mm256_xor_pd(bc[0], bc[2]);
    t2 = _mm256_xor_pd(bc[1], bc[3]);
    v1 = _mm256_xor_pd(t1, t2);
    ins_mul2_64((unsigned long *)(m+6), (unsigned long *)(&t1), (unsigned long *)(&t1) + 2);
    ins_mul2_64((unsigned long *)(m+7), (unsigned long *)(&t2), (unsigned long *)(&t2) + 2);
    ins_mul2_64((unsigned long *)(m+8), (unsigned long *)(&v1), (unsigned long *)(&v1) + 2);

    _mm_store_si128((__m128i *)a, _mm256_castsi256_si128(_mm256_castpd_si256(m[0])));
    _mm_store_si128((__m128i *)a + 1, _mm_xor_si128(_mm256_extractf128_si256(_mm256_castpd_si256(m[0]), 1),
        _mm256_castsi256_si128(_mm256_castpd_si256(m[1]))));
    _mm_store_si128((__m128i *)a + 2, _mm_xor_si128(_mm256_extractf128_si256(_mm256_castpd_si256(m[1]), 1),
        _mm256_castsi256_si128(_mm256_castpd_si256(m[2]))));
    _mm_store_si128((__m128i *)a + 3, _mm_xor_si128(_mm256_extractf128_si256(_mm256_castpd_si256(m[2]), 1),
        _mm256_castsi256_si128(_mm256_castpd_si256(m[3]))));
    _mm_store_si128((__m128i *)a + 4, _mm256_extractf128_si256(_mm256_castpd_si256(m[3]), 1));
    __m128i v = _mm_xor_si128(_mm_load_si128((__m128i *)a + 3), _mm_load_si128((__m128i *)a + 4));
    _mm_store_si128((__m128i *)a + 6, v);
    _mm_store_si128((__m128i *)a + 5, _mm_xor_si128(_mm_load_si128((__m128i *)a + 2), v));
    __m128i t = _mm_xor_si128(_mm_load_si128((__m128i *)a + 1), _mm_load_si128((__m128i *)a + 2));
    _mm_store_si128((__m128i *)a + 4, _mm_xor_si128(t, v));
    xor_128(a + 6, _mm_xor_si128(_mm_load_si128((__m128i *)a), t));
    _mm_store_si128((__m128i *)a + 2, _mm_xor_si128(t, _mm_load_si128((__m128i *)a)));
    xor_128(a + 2, _mm_load_si128((__m128i *)a));
    t = _mm_xor_si128(_mm256_extractf128_si256(_mm256_castpd_si256(m[6]), 1),
        _mm256_castsi256_si128(_mm256_castpd_si256(m[7])));
    v = _mm256_extractf128_si256(_mm256_castpd_si256(m[7]), 1);
    v1 = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(_mm_xor_si128(t,
        _mm256_castsi256_si128(_mm256_castpd_si256(m[6])))), _mm_xor_si128(t, v), 1));
    v2 = _mm256_xor_pd(_mm256_xor_pd(v1, m[8]), _mm256_xor_pd(m[4], m[5]));
    t1 = _mm256_xor_pd(m[5], _mm256_castsi256_pd(_mm256_castsi128_si256(v)));
    t2 = _mm256_xor_pd(m[4], _mm256_permute2f128_pd(m[6], _mm256_setzero_pd(), 2));
    _mm256_storeu_pd((double *)a+2, _mm256_xor_pd(_mm256_loadu_pd((double *)a+2), t2));
    _mm256_storeu_pd((double *)a+6, _mm256_xor_pd(_mm256_loadu_pd((double *)a+6), v2));
    _mm256_storeu_pd((double *)a+10, _mm256_xor_pd(_mm256_loadu_pd((double *)a+10), t1));

    /* Original code (using original 4-term formula)
    _mm256_storeu_pd((double *)a, m[0]);
    _mm_storeu_pd((double *)a+12, _mm256_extractf128_pd(m[3], 0));
    t1 = _mm256_xor_pd(m[0], m[1]);
    t2 = _mm256_xor_pd(m[2], m[6]);
    _mm256_storeu_pd((double *)a+4, _mm256_xor_pd(t1, t2));
    v1 = _mm256_xor_pd(t1, m[4]);
    _mm256_storeu_pd((double *)a+2, _mm256_xor_pd(_mm256_loadu_pd((double *)a+2), v1));
    t1 = _mm256_xor_pd(m[2], m[3]);
    t2 = _mm256_xor_pd(m[1], m[7]);
    _mm256_storeu_pd((double *)a+8, _mm256_xor_pd(t1, t2));
    v2 = _mm256_xor_pd(t1, m[5]);
    _mm256_storeu_pd((double *)a+10, _mm256_xor_pd(_mm256_loadu_pd((double *)a+10), v2));
    t1 = _mm256_xor_pd(m[6], m[7]);
    _mm256_storeu_pd((double *)a+6, _mm256_xor_pd(_mm256_loadu_pd((double *)a+6),
        _mm256_xor_pd(_mm256_xor_pd(v1, v2), _mm256_xor_pd(t1, m[8]))));
*/}
#endif

// 9-word multiplication (strategy 9-1)
void mul9_64_1(unsigned long *c, const unsigned long *a, const unsigned long *b)
{
    unsigned long t1[5], t2[5];
    /*__declspec(align(32))*/ unsigned long t3[10];
    ins_mul5_64(c, a, b);
    mul4_64_0(c+10, a+5, b+5);
#ifndef _USE_AVX_
    _mm_storeu_si128((__m128i *)t1, _mm_xor_si128(_mm_loadu_si128((__m128i *)a),
        _mm_loadu_si128((__m128i *)(a+5))));
    _mm_storeu_si128((__m128i *)(t1+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+2)),
        _mm_loadu_si128((__m128i *)(a+7))));
    _mm_storeu_si128((__m128i *)t2, _mm_xor_si128(_mm_loadu_si128((__m128i *)b),
        _mm_loadu_si128((__m128i *)(b+5))));
    _mm_storeu_si128((__m128i *)(t2+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(b+2)),
        _mm_loadu_si128((__m128i *)(b+7))));
#else
    _mm256_store_pd((double *)t1, _mm256_xor_pd(_mm256_loadu_pd((double *)a), _mm256_loadu_pd((double *)(a+5))));
    _mm256_storeu_pd((double *)t2, _mm256_xor_pd(_mm256_loadu_pd((double *)b), _mm256_loadu_pd((double *)(b+5))));
#endif
    t1[4] = a[4];
    t2[4] = b[4];
    ins_mul5_64(t3, t1, t2);
#define pxor_assign(x, y) _mm_storeu_si128((__m128i *)(x), _mm_xor_si128(_mm_loadu_si128((__m128i *)(x)), \
    _mm_loadu_si128((__m128i *)(y))))
#ifndef _USE_AVX_
    pxor_assign(t3+8, c+8);
    _mm_storeu_si128((__m128i *)(t3), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3)),
        _mm_xor_si128(_mm_loadu_si128((__m128i *)(c)), _mm_loadu_si128((__m128i *)(c+10)))));
    _mm_storeu_si128((__m128i *)(t3+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+2)),
        _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+2)), _mm_loadu_si128((__m128i *)(c+12)))));
    _mm_storeu_si128((__m128i *)(t3+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+4)),
        _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+4)), _mm_loadu_si128((__m128i *)(c+14)))));
    _mm_storeu_si128((__m128i *)(t3+6), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+6)),
        _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+6)), _mm_loadu_si128((__m128i *)(c+16)))));
    pxor_assign(c+5, t3);
    pxor_assign(c+7, t3+2);
    pxor_assign(c+9, t3+4);
    pxor_assign(c+11, t3+6);
    pxor_assign(c+13, t3+8);
#else
    __m256d d1 = _mm256_loadu_pd((double *)(t3));
    d1 = _mm256_xor_pd(d1, _mm256_loadu_pd((double *)(c)));
    d1 = _mm256_xor_pd(d1, _mm256_loadu_pd((double *)(c+10)));

    __m256d d2 = _mm256_loadu_pd((double *)(t3+4));
    d2 = _mm256_xor_pd(d2, _mm256_loadu_pd((double *)(c+4)));
    d2 = _mm256_xor_pd(d2, _mm256_loadu_pd((double *)(c+14)));

    _mm_storeu_si128((__m128i *)(c+13), _mm_xor_si128(_mm_loadu_si128((__m128i *)(c+13)),
            _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+8)), _mm_loadu_si128((__m128i *)(c+8)))));

    _mm256_storeu_pd((double *)(c+5), _mm256_xor_pd(_mm256_loadu_pd((double *)(c+5)), d1));
    _mm256_storeu_pd((double *)(c+9), _mm256_xor_pd(_mm256_loadu_pd((double *)(c+9)), d2));
#endif
}

#define xor3(a, b, c) _mm_storeu_si128((__m128i *)(a), _mm_xor_si128(_mm_loadu_si128((__m128i *)(b)), \
    _mm_loadu_si128((__m128i *)(c)))); *(a+2) = *(b+2) ^ *(c+2);
// 9-word multiplication (strategy 9-2)
void mul9_64_2(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    unsigned long t1[6], t2[6], t3[6], t[6];

    ins_mul3_64(a + 6, b+3, c + 3);
    ins_mul3_64(a + 12, b + 6, c + 6);
    xor3(a, b, b+3);
    xor3(a+3, c, c+3);
    ins_mul3_64(t1, a, a+3);
    xor3(a, b+3, b+6);
    xor3(a+3, c+3, c+6);
    ins_mul3_64(t2, a, a+3);
    xor3(a, b, b+6);
    xor3(a+3, c, c+6);
    ins_mul3_64(t3, a, a+3);
    ins_mul3_64(a, b, c);

    /* Original code (using original 3-term formula)
    xor3(t+3, a + 12, a+9);
    xor3(t+3, t+3, a+15);
    xor3(t, a, a+3);
    xor3(t, t, a+6);
    xor3(t3, t3, a+9);
    xor3(t3, t3, a+12);
    xor3(t3+3, t3+3, a+3);
    xor3(t3, t3, a+6);
    _copy(a+3, t, 3);
    _copy(a+6, t, 3);
    _copy(a+9, t+3, 3);
    _copy(a+12, t+3, 3);
    */
#ifndef _USE_AVX_
    __m128i v1, v2, v3;
    v1 = _mm_loadu_si128((__m128i *)(a));
    v2 = _mm_loadu_si128((__m128i *)(a+6));
    v3 = _mm_loadu_si128((__m128i *)(a+12));
    _mm_storeu_si128((__m128i *)(t1), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t1)), _mm_xor_si128(v1, v2)));
    _mm_storeu_si128((__m128i *)(t2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t2)), _mm_xor_si128(v3, v2)));
    _mm_storeu_si128((__m128i *)(t3), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3)), _mm_xor_si128(v1, v3)));
    v1 = _mm_loadu_si128((__m128i *)(a+2));
    v2 = _mm_loadu_si128((__m128i *)(a+8));
    v3 = _mm_loadu_si128((__m128i *)(a+14));
    _mm_storeu_si128((__m128i *)(t1+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t1+2)), _mm_xor_si128(v1, v2)));
    _mm_storeu_si128((__m128i *)(t2+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t2+2)), _mm_xor_si128(v3, v2)));
    _mm_storeu_si128((__m128i *)(t3+2), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+2)), _mm_xor_si128(v1, v3)));
    v1 = _mm_loadu_si128((__m128i *)(a+4));
    v2 = _mm_loadu_si128((__m128i *)(a+10));
    v3 = _mm_loadu_si128((__m128i *)(a+14));
    _mm_storeu_si128((__m128i *)(t1+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t1+4)), _mm_xor_si128(v1, v2)));
    _mm_storeu_si128((__m128i *)(t2+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t2+4)), _mm_xor_si128(v3, v2)));
    _mm_storeu_si128((__m128i *)(t3+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+4)), _mm_xor_si128(v1, v3)));

    xor2(a+6, a+6, t3, 6);
    xor2(a+3, a+3, t1, 6);
    xor2(a+9, a+9, t2, 6);
#else
    __m256d d1 = _mm256_loadu_pd((double *)(t1));
    d1 = _mm256_xor_pd(d1, _mm256_loadu_pd((double *)(a)));
    d1 = _mm256_xor_pd(d1, _mm256_loadu_pd((double *)(a+6)));
    _mm_storeu_si128((__m128i *)(t1+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t1+4)),
            _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+4)), _mm_loadu_si128((__m128i *)(a+10)))));

    __m256d d2 = _mm256_loadu_pd((double *)(t2));
    d2 = _mm256_xor_pd(d2, _mm256_loadu_pd((double *)(a+12)));
    d2 = _mm256_xor_pd(d2, _mm256_loadu_pd((double *)(a+6)));
    _mm_storeu_si128((__m128i *)(t2+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t2+4)),
            _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+16)), _mm_loadu_si128((__m128i *)(a+10)))));

    __m256d d3 = _mm256_loadu_pd((double *)(t3));
    d3 = _mm256_xor_pd(d3, _mm256_loadu_pd((double *)(a)));
    d3 = _mm256_xor_pd(d3, _mm256_loadu_pd((double *)(a+12)));
    _mm_storeu_si128((__m128i *)(t3+4), _mm_xor_si128(_mm_loadu_si128((__m128i *)(t3+4)),
            _mm_xor_si128(_mm_loadu_si128((__m128i *)(a+4)), _mm_loadu_si128((__m128i *)(a+16)))));

    _mm256_storeu_pd((double *)(a+6), _mm256_xor_pd(_mm256_loadu_pd((double *)(a+6)), d3));
    xor_2(a+10, a+10, t3+4);
    _mm256_storeu_pd((double *)(a+3), _mm256_xor_pd(_mm256_loadu_pd((double *)(a+3)), d1));
    xor_2(a+7, a+7, t1+4);
    _mm256_storeu_pd((double *)(a+9), _mm256_xor_pd(_mm256_loadu_pd((double *)(a+9)), d2));
    xor_2(a+13, a+13, t2+4);
#endif
}

// 9-word multiplication (strategy 9-3)
#define mul2_128(x,y) ins_mul2_64((unsigned long *)(x), (unsigned long *)(y), (unsigned long *)(y) + 2)
#define set_128(y,x) _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(\
    _mm_loadu_si128((__m128i *)(x))), _mm_loadu_si128((__m128i *)(y)), 1))
#ifdef __AVX__
void mul9_128_avx(unsigned long *a, const unsigned long *b, const unsigned long *c)
{
    __m256d v1, v2, t1, t2;
    __m256d m[14], bc[5];

    bc[0] = set_128(c, b);
    bc[1] = set_128(c+2, b+2);
    bc[2] = set_128(c+4, b+4);
    bc[3] = set_128(c+6, b+6);
    bc[4] = _mm256_castsi256_pd(_mm256_insertf128_si256(_mm256_castsi128_si256(_mm_cvtsi64_si128(b[8])),
        _mm_cvtsi64_si128(c[8]), 1));

    mul2_128(m+1, bc+0);
    mul2_128(m+2, bc+1);
    mul2_128(m+3, bc+2);
    mul2_128(m+4, bc+3);
    mul2_128(m+5, bc+4);
    v1 = _mm256_xor_pd(bc[0], bc[1]);
    mul2_128(m+6, &v1);
    v2 = _mm256_xor_pd(bc[0], bc[2]);
    mul2_128(m+7, &v2);
    t1 = _mm256_xor_pd(bc[4], bc[2]);
    mul2_128(m+8, &t1);
    t2 = _mm256_xor_pd(bc[3], bc[4]);
    mul2_128(m+9, &t2);
    v2 = _mm256_xor_pd(v2, bc[3]);
    mul2_128(m+10, &v2);
    t1 = _mm256_xor_pd(t1, bc[1]);
    mul2_128(m+11, &t1);
    v1 = _mm256_xor_pd(v1, t2);
    mul2_128(m+12, &v1);
    v1 = _mm256_xor_pd(v1, bc[2]);
    mul2_128(m+13, &v1);

    _mm256_storeu_pd((double *)a, m[1]);
    m128i_ui64(a+16, _mm256_castsi256_si128(_mm256_castpd_si256(m[5])));
    m[0] = _mm256_xor_pd(m[1], m[2]);
    _mm256_storeu_pd((double *)(a+4), _mm256_xor_pd(m[0], _mm256_xor_pd(m[7], m[3])));
    v1 = _mm256_xor_pd(m[0], m[6]);
    m[0] = _mm256_xor_pd(m[4], m[5]);
    _mm256_storeu_pd((double *)(a+12), _mm256_xor_pd(m[0], _mm256_xor_pd(m[8], m[3])));
    v2 = _mm256_xor_pd(m[0], m[9]);

    __m256d t;
    t = _mm256_xor_pd(m[13], v1);
    t = _mm256_xor_pd(t, v2);
    t = _mm256_xor_pd(t, m[10]);
    t = _mm256_xor_pd(t, m[11]);
    _mm256_storeu_pd((double *)(a+8), t);

    t = _mm256_xor_pd(m[13], m[12]);
    m[0] = _mm256_xor_pd(_mm256_loadu_pd((double *)(a+4)), t);
    t = _mm256_xor_pd(_mm256_loadu_pd((double *)(a+12)), t);
    m[0] = _mm256_xor_pd(m[0], m[5]);
    m[0] = _mm256_xor_pd(m[0], m[11]);
    t = _mm256_xor_pd(t, m[1]);
    t = _mm256_xor_pd(t, m[10]);
    xor_256(a+10, m[0]);
    xor_256(a+6, t);
    xor_256(a+2, v1);
    xor_256(a+14, v2);
}
#endif

//Reduction on GF(2^163)
void reduce163(unsigned long *a, const unsigned long *c)
{
    unsigned long s, t, u;

    m128i_ui64(a, _mm_loadu_si128((__m128i *)c));
    a[2] = c[2] & (pow2(35) - 1);
    t = c[5];
    unsigned long q, r;
    for(int i = 4; i >= 3; --i)
    {
        s = t;
        t = c[i];
        shiftright128(r, t, s, 28);
        q = r;
        shiftright128(r, t, s, 29);
        q ^= r;
        shiftright128(r, t, s, 32);
        q ^= r;
        shiftright128(r, t, s, 35);
        q ^= r;
        a[i-2] ^= q;
    }
    s = t;
    t = c[2] & ~(pow2(35) - 1);
    shiftright128(r, t, s, 28);
    q = r;
    shiftright128(r, t, s, 29);
    q ^= r;
    shiftright128(r, t, s, 32);
    q ^= r;
    shiftright128(r, t, s, 35);
    q ^= r;
    a[0] ^= q;
    t = a[2] >> 35;
    a[0] ^= t ^ (t << 3) ^ (t << 6) ^ (t << 7);
    a[2] &= pow2(35) - 1;
}

//Reduction on GF(2^233)
void reduce233(unsigned long *a, const unsigned long *c)
{
    unsigned long t, a4;
    unsigned long q, r, s;

    s = c[7];
    m128i_ui64(a, _mm_loadu_si128((__m128i *)c));
    a[2] = c[2];
    a[3] = c[3] & (pow2(41) - 1);

    q = 0;
    t = c[6];
    shiftright128(r, t, s, 31);
    a4 = r;
    shiftright128(q, t, s, 41);
    for (int i = 5; i >= 4; --i)
    {
        s = t;
        t = c[i];
        shiftright128(r, t, s, 31);
        a[i-2] ^= r ^ q;
        shiftright128(q, t, s, 41);
    }
    s = t;
    t = c[3] & ~(pow2(41) - 1);
    shiftright128(r, t, s, 31);
    a[1] ^= r ^ q;
    shiftright128(r, t, s, 41);
    a[0] ^= r;

    t = a4;
    a[1] ^= (t >> 41);
    a[2] ^= (t >> 31);

    t = a[3] & ~(pow2(41) - 1);
    shiftright128(r, t, a4, 41);
    a[0] ^= r;
    shiftright128(r, t, a4, 31);
    a[1] ^= r;
    a[3] &= (pow2(41) - 1);
}

//Reduction on GF(2^283)
void reduce283(unsigned long *a, const unsigned long *c)
{
    unsigned long t;
    m128i_ui64(a, _mm_loadu_si128((__m128i *)c));
    m128i_ui64(a+2, _mm_loadu_si128((__m128i *)(c+2)));
    a[4] = c[4] & (pow2(27) - 1);
    t = c[8];
    a[4] ^= (t >> 27) ^ (t >> 22) ^ (t >> 20) ^ (t >> 15);
    unsigned long q, r, s;
    for(int i = 7; i >= 5; --i)
    {
        s = t;
        t = c[i];
        shiftright128(r, t, s, 27);
        q = r;
        shiftright128(r, t, s, 22);
        q ^= r;
        shiftright128(r, t, s, 20);
        q ^= r;
        shiftright128(r, t, s, 15);
        q ^= r;
        a[i-4] ^= q;
    }
    s = t;
    t = c[4] & ~(pow2(27) - 1);
    shiftright128(r, t, s, 27);
    q = r;
    shiftright128(r, t, s, 22);
    q ^= r;
    shiftright128(r, t, s, 20);
    q ^= r;
    shiftright128(r, t, s, 15);
    q ^= r;
    a[0] ^= q;
    t = a[4] >> 27;
    a[0] ^= t ^ (t << 5) ^ (t << 7) ^ (t << 12);
    a[4] &= pow2(27) - 1;
}

//Reduction on GF(2^409)
void reduce409(unsigned long *a, const unsigned long *c)
{
    unsigned long t, a7;
    unsigned long q, r, s;

    t = c[12];
    m128i_ui64(a, _mm_loadu_si128((__m128i *)c));
    m128i_ui64(a+2, _mm_loadu_si128((__m128i *)(c+2)));
    m128i_ui64(a+4, _mm_loadu_si128((__m128i *)(c+4)));
    a[6] = c[6] & (pow2(25) - 1);
    a[6] ^= (t >> 25);
    a7 = (t >> 2);
    q = 0;
    for (int i = 11; i >= 7; --i)
    {
        s = t;
        t = c[i];
        shiftright128(r, t, s, 2);
        a[i-5] ^= r ^ q;
        shiftright128(q, t, s, 25);
    }
    s = t;
    t = c[6] & ~(pow2(25) - 1);
    shiftright128(r, t, s, 25);
    a[0] ^= r;
    shiftright128(r, t, s, 2);
    a[1] ^= r ^ q;

    t = a7;
    a[1] ^= (t >> 25);
    a[2] ^= (t >> 2);

    t = a[6] & ~(pow2(25) - 1);
    shiftright128(r, t, a7, 25);
    a[0] ^= r;
    shiftright128(r, t, a7, 2);
    a[1] ^= r;
    a[6] &= (pow2(25) - 1);
}

//Reduction on GF(2^571)
void reduce571(unsigned long *a, const unsigned long *c)
{
    unsigned long t, a9;

    memcpy(a, c, 9 * sizeof(unsigned long));
    a[8] &= pow2(59) - 1;
    t = c[17];
    a9 = (t >> 59) ^ (t >> 57) ^ (t >> 54) ^ (t >> 49);

    unsigned long q, r, s;
    for(int i = 16; i >= 9; --i)
    {
        s = t;
        t = c[i];
        shiftright128(r, t, s, 59);
        q = r;
        shiftright128(r, t, s, 57);
        q ^= r;
        shiftright128(r, t, s, 54);
        q ^= r;
        shiftright128(r, t, s, 49);
        q ^= r;
        a[i-8] ^= q;
    }
    s = t;
    t = c[8] & ~(pow2(59) - 1);
    shiftright128(r, t, s, 59);
    q = r;
    shiftright128(r, t, s, 57);
    q ^= r;
    shiftright128(r, t, s, 54);
    q ^= r;
    shiftright128(r, t, s, 49);
    q ^= r;
    a[0] ^= q;

    s = a9;
    t = a[8] & ~(pow2(59) - 1);
    shiftright128(r, t, s, 59);
    q = r;
    shiftright128(r, t, s, 57);
    q ^= r;
    shiftright128(r, t, s, 54);
    q ^= r;
    shiftright128(r, t, s, 49);
    q ^= r;
    a[0] ^= q;
    a[8] &= pow2(59) - 1;
```

```
}

//Data structure for timing
struct timing
{
    struct timespec t1, t2;
    long ta, tb;
    long sums[32];
    int count;
    unsigned int aux;

public:
    timing()
    {
        count = 0;
        for (int i = 0; i < 32; i++)
        {
            sums[i] = 0;
        }
    }
};

//Timing
#define _addTime(t, x)  t.ta = __rdtscp(&t.aux); x; t.tb = __rdtscp(&t.aux); \
    t.sums[t.count] += t.tb - t.ta; t.count++;

/*
 * If defined, repeat calling a funtion REP times continuously;
 * otherwise, the outer loop will be repeated REP times
 */
#define _REPEAT_ 1

#define REP 10000
#define ntimes(ti, y, m) _addTime(ti, for(int iz = 0; iz < m; ++iz) { y;})
#ifdef _REPEAT_
#define addTime(t, x) ntimes(t, x, REP)
#else
#define addTime(t, x) _addTime(t, x)
#endif

void init(unsigned long **x, unsigned int n)
{
    int length = n / WordLength + (n % WordLength ? 1 : 0);
#ifdef _ALIGN_32_
    *x = (unsigned long *)_mm_malloc(length * sizeof(unsigned long), 32);
#else
    *x = (unsigned long *)malloc(length * sizeof(unsigned long));
#endif
    memset(*x, 0, length * sizeof(unsigned long));
}

void clear(unsigned long *val)
{
#ifdef _ALIGN_32_
        _mm_free(val);
#else
        free(val);
#endif
}

void random(unsigned long *x, unsigned int n)
{
    int r = (n+1) % WordLength;
    unsigned long *q = x + (n+1) / WordLength;
    //srand(time(NULL));
    int l = WordLength / 16;
    for(unsigned long *p = x; p < q; ++p)
    {
        *p = 0;
        for(int i = 0; i < l; i++)
            *p ^= (unsigned long)rand() << 15 * i;
        *p ^= (unsigned long)(rand() % (1 << l)) << 15 * l;
    }
    if(r)
    {
        *q = 0;
        for(int i = 0; i < l; i++)
            *q ^= (unsigned long)rand() << 15 * i;
        *q ^= (unsigned long)(rand() % (1 << l)) << 15 * l;
        *q &= pow2(r) - 1;
    }
}
```

```
void test_mul(int n, const unsigned long *f)
{
    timing ti;
    unsigned long *x, *y, *z;
    unsigned long *t, *w;
    init(&x, n);
    init(&y, n);
    init(&z, n);
    int double_len = 2 * (n / WordLength + (n % WordLength ? 1 : 0)) * WordLength;
    init(&t, double_len);
    init(&w, double_len);
    printf("%d\n", n);
#ifndef _REPEAT_
    for (int i = 0; i < REP; ++i)
#endif
    {
        ti.count = 0;
        random(x, n-1);
        random(y, n-1);

        addTime(ti, );  //Empty statement, for reducing errors

        /*
         * You can change the functions to be tested below. Reduction functions
         * should follow multiplication functions. When testing routines using AVX
         * instructions, define the related macros.
         */
        switch(n)
        {
        case 163:
            addTime(ti, ins_mul3_64(t, x, y));
            addTime(ti, naive163(w, x, y));
            addTime(ti, reduce163(z, t));
            break;
        case 233:
            addTime(ti, mul4_64_0(t, x, y));
            addTime(ti, mul4_64_1(w, x, y));
            addTime(ti, reduce233(z, t));
            break;
        case 283:
            addTime(ti, ins_mul5_64(t, x, y));
            addTime(ti, mul5_128_avx(w, x, y));
            addTime(ti, reduce283(z, t));
            break;
        case 409:
            addTime(ti, mul7_64_0(t, x, y));
            addTime(ti, mul7_64_1(w, x, y));
            addTime(ti, reduce409(z, t));
            break;
        case 571:
            addTime(ti, mul9_64_1(t, x, y));
            addTime(ti, mul9_64_2(w, x, y));
            addTime(ti, reduce571(z, t));
        }
        //For checking
        for (int j = 0; j <= 2 * n / WordLength; ++j)
         {
            if(t[j] != w[j])
                printf("%ld\n", t[j]);
         }
    }

    for (int j = 1; j < ti.count; ++j)
    {
        printf("%ld\t", (ti.sums[j] - ti.sums[0]) / REP);
    }
    printf("\n");
    clear(x);
    clear(y);
    clear(z);
    clear(t);
    clear(w);
}

void SetCoeff(unsigned long *x, unsigned int n, int coeff)
{
    int p = n % WordLength;

    if(coeff != 0)
    {
        x[n / WordLength] |= pow2(p);
    }
    else
    {
        x[n / WordLength] &= ~(pow2(p));
    }
}

void testMuls()
{
    unsigned long *f163, *f233, *f283, *f409, *f571;
    init(&f163, 163);   init(&f233, 233);   init(&f283, 283);
    init(&f409, 409);   init(&f571, 571);
    SetCoeff(f233, 233, 1);
    f233[0] = 1;
    f233[1] = 1 << 10;
    SetCoeff(f283, 283, 1);
    f283[0] = 0x10a1;
    SetCoeff(f571, 571, 1);
    f571[0] = 0x425;
    SetCoeff(f409, 409, 1);
    f409[0] = 1;
    f409[1] = 1 << 23;
    SetCoeff(f163, 163, 1);
    f163[0] = 0xc9;

    test_mul(163, f163);
    test_mul(233, f233);
    test_mul(283, f283);
    test_mul(409, f409);
    test_mul(571, f571);
}

int main()
{
    srand(time(NULL));
    testMuls();
    return 0;
}
```