Check for updates

Fast Modular Squaring with AVX512IFMA

1

Nir Drucker and Shay Gueron

1.1 Introduction

The Multiply and Accumulate (MAC) operation consumes three inputs a, b, c, and computes $a = a + b \cdot c$. It is a fundamental step in many floating-point and integer computations. Examples are dot product calculations, matrix multiplications, and modular arithmetic. Modern processors offer instructions for performing MAC over floating-point inputs, e.g., AMD Bulldozer's Fused Multiply-Add (FMA), and Intel's Single Instruction Multiple Data (SIMD)-FMA (starting with the microarchitecture Codename Haswell). Here, we focus on Intel's AVX512IFMA instructions [1] that compute MAC on unsigned integers.

The AVX512IFMA instructions are defined, but are not yet widely available. However, a demonstration of their capabilities is already given in [11], showing a 2x potential speedup for 1024-bit integer multiplication (and more for larger operands). Another example is [6], where we showed a 6x potential speedup over OpenSSL's Montgomery Multiplication (MM). Additional code examples [5, 10] contributed to OpenSSL, include optimized 1024/1536/2048-bit MM. These demonstrations did not optimize modular squaring specifically; rather, they used a multiplication routine for squaring as well. Here, we show how to use the AVX512-IFMA instructions for optimizing modular squaring. Our developments build on top of the AMS optimization of [7] (other squaring methods can be found in [4, 9, 13]).

The paper is organized as follows. Section 1.2 discusses some preliminaries. Section 1.3 deals with implementing the AMS algorithm with the AVX512IFMA instructions. In Sect. 1.4, we propose a potential improvement to the definition of AVX512IFMA. Finally, we show our experimental results in Sect. 1.5, and provide our conclusions in Sect. 1.6.

1.2 Preliminaries and Notation

Hereafter, we use lower case letters to represent scalars (64-bit integers), and upper case letters to represent 512-bit wide register. We denote zero extension of a 64 bits variable x by $\mathrm{ZE}(x)$.

1.2.1 The AVX512IFMA Instructions

Intel's Software Developer Manual [1] introduces two instructions called AVX512IFMA: `VPMADD52LUQ` and `VPMADD52HUQ`. Their functionality is illustrated in Algorithm 1.1. These instructions multiply eight 52-bit unsigned integers residing in wide 512-bit registers, produce the low (`VPMADD52LUQ`) and high (`VPMADD52HUQ`) halves of the 104-bit products, and add the results to 64-bit accumulators (i.e., SIMD elements), placing them in the destination register. They are designed for supporting big number multiplications, when the inputs are stored in a "redundant representation" using radix 2^{52} (as explained in [8]).

The AVX512IFMA instructions build on the existence of other instructions called SIMD-FMA, which are designed to support IEEE standard Floating-Point Arithmetic [12]. The SIMD-FMA instructions handle double-precision floating-point numbers ($x[63:0]$), where the bits are viewed as: (a)

This work was done prior to joining Amazon.

N. Drucker · S. Gueron (✉)
University of Haifa, Haifa, Israel

Amazon Web Services Inc, Seattle, WA, USA
e-mail: shay@math.haifa.ac.il

© Springer Nature Switzerland AG 2019

S. Latifi (ed.), *16th International Conference on Information Technology-New Generations (ITNG 2019)*,
Advances in Intelligent Systems and Computing 800,
https://doi.org/10.1007/978-3-030-14070-0_1

Algorithm 1.1 DST = VPMADD52(A,B,C) [1]

Inputs: A,B,C (512-bit wide registers)
Outputs: DST (a 512-bit wide register)

procedure VPMADD52LUQ(A, B, C)
  for j := 0 to 7 **do**
    i := j × 64
    TMP[127 : 0] := ZE(B[i+51:i]) × ZE(C[i+51:i])
    DST[i+63:i] := A[i+63:i] + ZE(TMP[**51** : **0**])
procedure VPMADD52HUQ(A, B, C)
  for j := 0 to 7 **do**
    i := j × 64
    TMP[127 : 0] := ZE(B[i+51:i]) × ZE(C[i+51:i])
    DST[i+63:i] := A[i+63:i] + ZE(TMP[**103** : **52**])

fraction $x[51:0]$ (53 bits where only 52 bits are explicitly stored); (b) exponent $x[62:52]$; (c) sign bit $x[63]$.

1.2.2 Almost Montgomery Multiplication

MM is an efficient technique for computing modular multiplications [14]. Let t be a positive integer, k an odd modulus and $0 \leq a, b < k$ integers. We denote the MM by $MM(a,b) = a \cdot b \cdot 2^{-t} \pmod{k}$, where 2^t is the Montgomery parameter. A variant of MM, called Almost Montgomery Multiplication (AMM) [7], is defined as follows. Let k and t be defined as above, and $0 \leq a, b < B$ integers, then $AMM(a,b)$ is an integer U that satisfies: (1) $U \pmod{m} = a \cdot b \cdot 2^{-t} \pmod{k}$; (2) $U \leq B$.

The advantage of AMM over MM is that the former does not require a (conditional) "final reduction" step. This allows using the output of one invocation as the input to a subsequent invocation. The relation between AMM and MM is the following. If $0 \leq a, b < B$, $RR = 2^{2t} \pmod{k}$,$a' = AMM(a, RR)$, $b' = AMM(b, RR)$, $u' = AMM(a', b')$ and $u = AMM(u', 1)$, then $u = a \cdot b \pmod{k}$.

1.3 Implementing AMS with AVX512IFMA

One of the common squaring algorithms [4] is the following. Let $A = \sum_{i=0}^{n} B^i a_i$ be an n digits integer in base B, $a_i \geq 0$. Then,

$$\begin{aligned} A^2 &= \sum_{i=0}^{n}\sum_{j=0}^{n} B^{i+j} a_i a_j \\ &= \sum_{i=0}^{n} B^{2i} a_i^2 + 2\sum_{i=0}^{n}\sum_{j=i+1}^{n} B^{i+j} a_i a_j \end{aligned} \tag{1.1}$$

where the last multiplication by 2 can be carried out by a series of left shift operations [9]. This reduces about half of the single-precision multiplications (compared to regular multiplication). Additional improvement is achieved by using vectorization. For example, [8] shows squaring implementations that use Intel's Advanced Vector Extensions (AVX) and AVX2 instructions. In these implementations, integers are stored in a "redundant representation" with radix $B = 2^{28}$ (each of the n digits is placed in a 32-bit container, padded from above with 4 zero bits). Each of the AVX2 256-bit wide registers (ymm) can hold up to eight 32-bit containers. This allows for (left) shifting of 8 digits in parallel, without losing their carry bit.

Algorithm 1.2 describes an implementation of AMS= AMM(a,a) that uses the AVX512IFMA instructions. Let the input (a), the modulus (m) and the result (x) be n-digit integers in radix $B = 2^{52}$, where each digit is placed in a 64-bit container (padded with 12 zero bits from above). Let $z = \lceil n/8 \rceil$ be the total number of wide registers needed for holding an n-digit number, and denote $k_0 = -m^{-1} \pmod{2^{52}}$. The final step of Algorithm 1.2 returns the result to the radix $B = 2^{52}$ format, by rearranging the carry bits. An illustration of a simple AMS flow is given in Fig. 1.1 that shows how ∼20% of the `VPMADD52` calls (left as blank spaces in the figure) are saved, compared to an AMM. The algorithm applies the left shift optimization of [9] to the AVX512IFMA AMM implementation of [11]. This can be done through either Eq. 1.1 (perform all MAC calculations and then shift the result by one), or according to:

Algorithm 1.2 x = AMS52(a, m, k_0)

Inputs: a,m (n-digit unsigned integers), k_0 (52-bit unsigned integer)
Outputs: x (n-digit unsigned integers)

procedure MULA[L/H]PART(i)
  X_i := VPMADD52[L/H]UQ(X_i, A_{curr}, A_i)
  for $j := i+1$ to z **do**
    T := VPMADD52[L/H]UQ($ZERO$, A_{curr}, A_j)
    $X_j := X_j + (T \ll 1)$

procedure AMS52(a, m, k_0)
  load a into $A_0 \ldots A_z$ and m into $M_0 \ldots M_z$
  zero($X_0 \ldots X_z$, $ZERO$)
  for $i := 0$ to z **do**
    for $j := 0$ to min$\{8, n-(8 \cdot i)\}$ **do**
      A_{curr} = broadcast($a[8 \cdot i + j]$)
      MulALPart(i)
      $y[127:0] := k_0 \cdot X_0[63:0]$
      Y := broadcast($y[52:0]$)
      for $l := 0$ to z **do**
        X_l := VPMADD52LUQ(X_l, M_l, Y)
      $x_0 := X_0[63:0] \gg 52$
      $X := X \gg 64$
      $X_0[63:0] = X_0[63:0] + x_0$
      MulAHPart(i)
      for $l := 0$ to z **do**
        X_l := VPMADD52HUQ(X_l, M_l, Y)
      FixRedundantRepresentation(X)
      return X

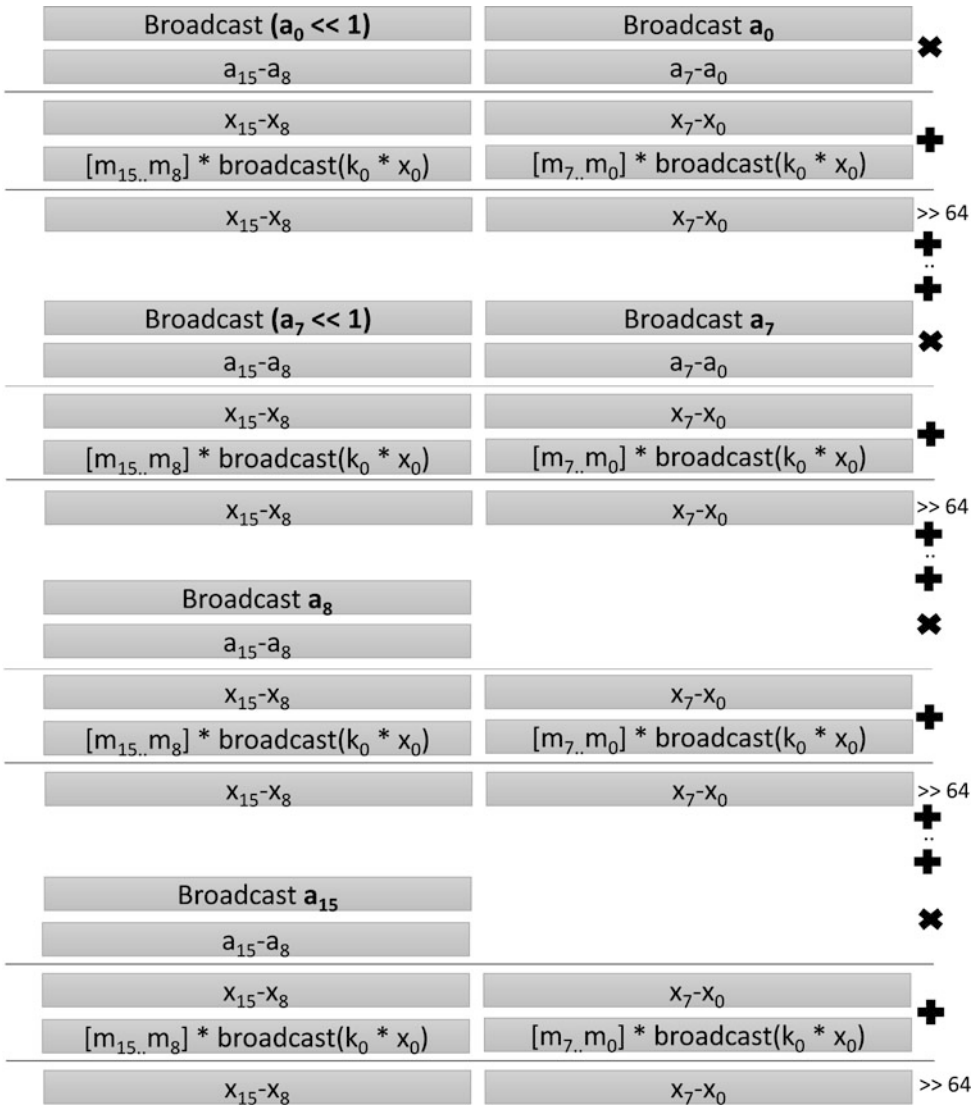

Fig. 1.1 Flow illustration of $x = \mathrm{SQR}(a, m, k_0)$, where a, m and x are 16-digit operands, each one is accommodated in two *zmm* registers

$$A^2 = \sum_{i=0}^{n} B^{2i} a_i^2 + \sum_{i=0}^{n} \sum_{j=i+1}^{n} B^{i+j} a_i a'_j \qquad (1.2)$$

where $a' = a \ll 1$. An efficient implementation of the first approach requires to accommodate a, m, and x in wide registers (not in memory), while an implementation of the second approach requires accommodating a' in wide registers as well. Consequently, the AVX512, which has only 32 wide registers, can hold n-digit integers up to $n \leq 85$ with the first approach, or up to $n \leq 64$ with the second approach. For example, 4096-bit modular squaring (part of a 4096-bit exponentiation, e.g., for Paillier encryption) has $n = 80$-digits operands (written in radix $B = 2^{52}$). It requires 40 wide registers with the second approach (but there are not enough). With the first approach, only 30 wide registers are needed (there are 32). This situation seems better, but in practice, it is not good enough.

Performing left shifting of an n-digit number requires some extra wide registers. These are not necessarily available for use with the above two approaches. Thus, we propose Algorithm 1.2, that is based on the following identity:

$$A^2 = \sum_{i=0}^{n} B^{2i} a_i^2 + \sum_{i=0}^{n} \sum_{j=i+1}^{n} 2(B^{i+j} a_i a_j) \qquad (1.3)$$

Here, the left shifts are performed on-the-fly, and free some wide registers for supporting other operations.

Identifying an additional bottleneck On-the-fly left shifting can be implemented in three ways, but unfortunately, all three do not go along well with the AVX512IFMA architecture. The first alternative is to multiply, accumulate and shift the result. This may double shift some of the previously accumulated data. The second alternative is to shift one of the

VPMADD52's input operands. This may lead to a set carry bit in position 53, which would be (erroneously) ignored during the multiplication (see Algorithm 1.1). The third alternative splits the MAC operation, to inject the shift between. This is not feasible with the atomic operation of VPMADD52, but can be resolved by performing the Multiply-Shift-Accumulate operation in two steps, with an extra temporary (zeroed) wide register. Indeed, Algorithm 1.2, MulA[L/H]Part (steps 4, 5) executes this flow.

1.4 Is Using Radix 2^{51} Better?

In this section, we discuss the selection of the radix. The AVX512IFMA instructions leverage hardware that is needed anyhow, for the FMA unit (floating-point operations need 53-bit multiplication for a 106-bit mantissa). Obviously, given AVX512IFMA, it is natural to work with radix $B = 2^{52}$. Using a larger radix (e.g., $B = 2^{58}$) could be better in theory, but will incur too many costly conversions to allow for using VPMADD52. We also note that no native SIMD instructions for a larger radix are available. A smaller radix (e.g., 2^{51}) is, however, possible to choose. This allows to cut about half of the serialized instructions in steps 3–5 of $MulA[L/H]Part$, by left shifting one of the operands before the multiplication.

Algorithm 1.3 is a modification of Algorithm 1.2, operating in radix 2^{51}. While it avoids the shift operations before the VPMADD52LUQ, it still needs to perform the shifting before the VPMADD52HUQ instruction. For example, Let a, b, c_1, c_2 be 51-bit integers. After performing $c_1 = \texttt{VPMADD52LUQ}(0, a, b) = (a \times b)[51 : 0]$ and $c_2 = \texttt{VPMADD52HUQ}(0, a, b) = (a \times b)[102 : 52]$, c_1 and c_2 are no longer in (pure) radix 2^{51}. Propagating the carry bit in c_1 can be delayed to step 22 of Algorithm 1.3. In contrary, c_2 must be shifted prior to the accumulation step. As we show in Sect. 1.5, Algorithm 1.3 does not lead to faster squaring, with the current architecture. This suggests a possible improvement to the architectural definition.

1.4.1 A Possible Improvement for AVX512IFMA

Algorithm 1.3 offers better parallelization compared to Algorithm 1.2, but still includes serialized steps (e.g., the function MulHighPart). A completely parallelized algorithm requires hardware support. To this end, we suggest a new instruction that we call Fused Multiply-Shift-Add (FMSA), and describe in Algorithm 1.4. It shifts the multiplication result by an immediate value ($imm8$) before accumulating it. This instruction can be based on the same hardware that supports FMA (just as AVX512IFMA). Note that when $imm8 = 0$ then this instruction is exactly VPMADD52HUQ.

Algorithm 1.3 x = AMS51(a, m, k_0)

Inputs: a,m (n-digit unsigned integers), k_0 (52-bit unsigned integer)
Outputs: x (n-digit unsigned integers)

1: **procedure** MULHIGHPART(SRC1, SRC2, DEST)
2: TMP := VPMADD52HUQ(ZERO, SRC1, SRC2)
3: DEST := DEST + (TMP $\ll$ 1)

1: **procedure** AMS51(a, m, k_0)
2: load a into $A_0 \ldots A_z$ and m into $M_0 \ldots M_z$
3: zero($X_0 \ldots X_z$, ZERO)
4: **for** i := 0 to z **do**
5: **for** j := 0 to min{8, $n - (8 \cdot i)$} **do**
6: A_{curr} = broadcast($a[8 \cdot i + j]$)
7: $A_{shifted} = A_{curr} \ll 1$
8: X_i := VPMADD52LUQ(X_i, A_{curr}, A_i)
9: **for** $j := i + 1$ to z **do**
10: X_i := VPMADD52LUQ(X_i, $A_{shifted}$, A_i)
11: $y[127 : 0] := k_0 \cdot X_0[63 : 0]$
12: Y := broadcast($y[51 : 0]$)
13: **for** l := 0 to z **do**
14: X_l := VPMADD52LUQ(X_l, M_l, Y)
15: $x_0 := X_0[63 : 0] \gg 51$
16: $X := X \gg 64$
17: $X_0[63 : 0] = X_0[63 : 0] + x_0$
18: MulAHighPart(X_i, A_{curr}, A_i)
19: **for** $l := i + 1$ to z **do**
20: MulAHighPart(X_l, $A_{shifted}$, A_l)
21: **for** l := 0 to z **do**
22: MulAHighPart(X_l, M_l, Y)
23: FixRedundantRepresentation(X)
24: **return** X

Algorithm 1.4 DST=FMSA(A,B,C,imm8)

1: **for** j := 0 to 7 **do**
2: i := j*64
3: TMP[127 : 0] := ZE(B[i+51:i]) $\times$ ZE(C[i+51:i])
4: DST[i+63:i] := A[i+63:i] + ZE(**TMP[103 : 52]** $\ll$ **imm8**)

1.5 Results

1.5.1 Results for the Current Architecture

This section provides our performance results. For this study, we wrote new optimized code for all the algorithms discussed above, and measured them with the following methodology.

Currently, there are only a limited series of processors with VPMADD52, which we currently don't have. Therefore, to predict the potential improvement on future Intel architectures we used the Intel Software Developer Emulator

(SDE) [3]. This tool allows us to count the number of instructions executed during each of the tested functions. We marked the start/end boundaries of each function with "SSC marks" 1 and 2, respectively. This is done by executing "`movl ssc_mark, %ebx; .byte 0x64, 0x67, 0x90`" and invoking the SDE with the flags "-start_ssc_mark 1 -stop_ssc_mark 2 -mix -cnl". The rationale is that a reduced number of instructions typically indicates improved performance that will be observed on a real processor (although the exact relation between the instructions count and the eventual cycles count is not known in advanced).

Our measurements show that the overall number of instructions in our AMM and AMS implementations (in radix 2^{52}) is almost identical. However, the number of occurrences per instruction varies between the two algorithms. The most noticeable change was for the VPADDQ, VPMADD52, VPSLLQ, and VPXORQ instructions. Let u_{AMS}/u_{AMM} be the number of occurrences of the instruction u in AMS/AMM code, and let t_{AMM} be the total number of instructions in the AMM code. We write $r_u = (u_{AMS} - u_{AMM})/t_{AMM}$. Table 1.1 compares the r_u values for different u and operands sizes. It shows that reducing the number of VPMADD52 instructions is achieved through increasing the number of other instruction (e.g., VPADDQ, VPSLLQ, and VPXORQ).

To assess the impact of the above trade-off, we note that the latency of VPADDQ, VPSLLQ, and VPXORQ is 1 cycle, the throughput of VPADDQ and VPXORQ is 0.33 cycles, and the throughput of VPSLLQ is 1 cycle [2]. By comparison, we can expect that the latency/throughput of a future VPMADD52 would be similar to VPMADDWD (i.e., 5/1), or to VFMA$\star$ (i.e., 4/0.5). It appears that trading one VPMADD52 for 4 other instructions (which is worse than the trade-off we have to our AMS implementation) could still be faster than the AMM implementation.

To study the effects at the higher scale of the modular exponentiation code, we define the following notation. Let $u_{ModExpAMS}/u_{ModExp}$ be the number of occurrences of the instruction u in the modular exponentiation code, with and without AMS, respectively, and let t_{ModExp} be the overall number of instructions in this code (w/o AMS). We write $s_u = (u_{ModExpAMS} - u_{ModExp})/t_{ModExp}$. Table 1.2 shows the values s_u.

Table 1.1 Values of r_u for different u instructions and different operands sizes

Size	VPADDQ	VPMADD52	VPSLLQ	VPXORQ
1024	0.06	−0.05	0.06	0.06
1536	0.13	−0.07	0.07	0.07
2048	0.05	−0.09	0.08	0.08
3072	0.05	−0.13	0.15	0.15
4096	0.12	−0.12	0.12	0.12

Table 1.2 Values of s_u for different instructions (u) and different operands sizes

Size	VPADDQ	VPMADD52	VPSLLQ	VPXORQ
1024	0.01	−0.01	0.02	0.01
1536	0.02	−0.01	0.02	0.02
2048	0.02	−0.03	0.02	0.02
3072	0.04	−0.04	0.04	0.04

Table 1.3 Values of w_u^{AMM}, w_u^{AMS}, and w_u^{ModExp} for different instructions (u) and different operands sizes

Function name	VPADDQ	VPMADD52	VPSLLQ	VPXORQ
AMM3072	0.32	0.01	0.32	0.32
AMM4080	0.28	0.01	0.28	0.28
AMM4096	0.28	0.01	0.28	0.27
AMS3072	0.07	0.00	0.09	0.07
AMS4080	0.07	0.00	0.10	0.07
AMS4096	0.08	0.01	0.10	0.06
ModExp3072	0.05	0.00	0.06	0.05

We use the following notation for evaluating the radix 2^{51} technique. Let $u_{AMS51}/u_{AMM51}/u_{ModExpAMS51}$ be the number of occurrences of the instruction u in radix 2^{51} code. We write

$$w_u^{AMM} = (u_{AMM} - u_{AMM51})/t_{AMM}$$
$$w_u^{AMS} = (u_{AMS} - u_{AMS51})/t_{AMS}$$
$$w_u^{ModExp} = (u_{ModExpAMS} - u_{ModExpAMS51})/t_{ModExp}$$

Table 1.3 shows the values w_u^{AMM}, w_u^{AMS}, and w_u^{ModExp}. Here, we see that the number of VPMADD52 instructions is almost unchanged, but the number of VPADDQ, VPXORQ, and VPSLLQ was increased. Therefore, we predict that implementations with operands in radix 2^{51} will be slower than those in radix 2^{52}.

1.5.2 A "what if" Question: The Potential of FMSA

Table 1.4 is similar to Table 1.3, where we replace the instructions in the MulHighPart with only one VPMADD52HUQ instruction, emulating our new FMSA instruction. Here, the added number of VPADDQ, VPSLLQ, and VPXORQ instructions is no longer needed, and the full power of our AMS can be seen.

Table 1.4 Values of w_u^{AMM}, w_u^{AMS}, and w_u^{ModExp}, when using the FMSA instruction, for different instructions (u) and different operands sizes

Function name	VPADDQ	VPMADD52	VPSLLQ	VPXORQ
AMM3072	0.00	0.01	0.00	0.00
AMM4080	0.00	0.01	0.00	0.00
AMM4096	0.00	0.01	0.00	−0.01
AMS3072	−0.03	0.00	−0.09	−0.10
AMS4080	−0.10	0.00	−0.08	−0.10
AMS4096	−0.10	0.01	−0.08	−0.11
ModExp3072	−0.04	0.00	−0.03	−0.04

1.6 Conclusion

This paper showed a method to use Intel's new AVX512-IFMA instructions, for optimizing software that computes AMS on modern processor architectures. Section 1.5 motivates our prediction that the proposed implementation would further improve the implementations of modular exponentiation described in [7]. As a future research we are aiming on measuring our code on a real processor once it will be widely available.

In addition, we analyzed the hypothetical benefit of using a different radix: 2^{51} instead of 2^{52}. This can significantly improve the AMS algorithm (only) if a new instruction, which we call FMSA, is also added to the architecture. We note that FMSA requires only a small tweak over the current AVX512IFMA, and no new hardware.

Acknowledgements This research was supported by: The Israel Science Foundation (grant No. 1018/16); The Center for Cyber Law and Policy at the University of Haifa in conjunction with the Israel National Cyber Directorate in the Prime Ministers Office.

References

1. Intel ® 64 and IA-32 Architectures Software Developers Manual, Sept 2015
2. Intel ® 64 and IA-32 Architectures Optimization Reference Manual, June 2016
3. Intel® Software Development Emulator, version 8.12.0. https://software.intel.com/en-us/articles/intel-software-development-emulator, Jan 2017
4. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic, vol. 18. Cambridge University Press, Leiden (2010)
5. Drucker, N., Gueron, S.: [openssl patch] Fast 1536-bit modular exponentiation with the new VPMADD52 instructions. http://openssl.6102.n7.nabble.com/openssl-org-4032-PATCH-Fast-1536-bit-modular-exponentiation-with-the-new-VPMADD52-instructions-td60082.html, Sept 2015
6. Drucker, N., Gueron, S.: Paillier-encrypted databases with fast aggregated queries. In: 2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC), Las Vegas, pp. 848–853, Jan 2017
7. Gueron, S.: Efficient software implementations of modular exponentiation. J. Cryptogr. Eng. **2**(1), 31–43 (2012)
8. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. WAIFI **12**, 119–135 (2012)
9. Gueron, S., Krasnov, V.: Speeding up big-numbers squaring. In: 2012 Ninth International Conference on Information Technology: New Generations (ITNG), Las Vegas, pp. 821–823. IEEE (2012)
10. Gueron, S., Krasnov, V.: [openssl patch] Fast modular exponentiation with the new VPMADD52 instructions. https://rt.openssl.org/Ticket/Display.html?id=3590, Nov 2014
11. Gueron, S., Krasnov, V.: Accelerating big integer arithmetic using intel IFMA extensions. In: 2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH), Silicon Valley, pp. 32–38. IEEE (2016)
12. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754–1985 (1985)
13. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton/London/New York (1996)
14. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)