Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures

Shay Gueron[1, 2] and Vlad Krasnov[2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Corporation, Israel Development Center, Haifa, Israel

Abstract. This paper describes an algorithm for computing modular exponentiation using vector (SIMD) instructions. It demonstrates, for the first time, how such a software approach can outperform the classical scalar (ALU) implementations, on the high end x86_64 platforms, if they have a wide SIMD architecture. Here, we target speeding up RSA2048 on Intel's soon-to-arrive platforms that support the AVX2 instruction set. To this end, we applied our algorithm and generated an optimized AVX2-based software implementation of 1024-bit modular exponentiation. This implementation is seamlessly integrated into OpenSSL, by patching over OpenSSL 1.0.1. Our results show that our implementation requires 51% less instructions than the current OpenSSL 1.0.1 implementation. This illustrates the potential significant speedup in the RSA2048 performance, which is expected in the coming (2013) Intel processors. The impact of such speedup on servers is noticeable, especially since migration to RSA2048 is recommended by NIST, starting from 2013.

Keywords: modular arithmetic, modular exponentiation, Montgomery multiplication, RSA, SIMD, AVX, AVX2.

1 Introduction

The cryptographic algorithms that underlie SSL/TLS connections are a critical computational load for the supporting servers. As the major ingredient in the SSL/TLS handshake, RSA is an important factor, and NIST's recommendation for key-lengths [1], makes RSA2048 an important optimization target.

Currently, software implementations of RSA are "scalar code" that use ALU instructions (e.g., *ADD*/*ADC*/*MUL*). Improvements in the performance of *ADD*/*ADC*/*MUL* have made the scalar implementations very efficient on the modern x86_64 processors (see [7], [5] for details).

In this paper, we show that RSA software can gain significant performance by "vectorized" implementations that utilize the modern SIMD (vector) architectures. This is done by implementing a version of the RSAZ algorithm (short for RSA ZARIZ - Hebrew for "quick") [7]. SIMD (aka vector architecture), an acronym for "Single Instruction Multiple Data", is an architecture where a single instruction computes a function of several inputs, simultaneously (see e.g., [9]). These inputs are called "elements" and reside in registers that hold a few of them together. Early SIMD

F. Özbudak and F. Rodríguez-Henríquez (Eds.): WAIFI 2012, LNCS 7369, pp. 119–135, 2012.

architectures used the MMX instructions that operate on *64*-bit SIMD registers. It was followed by the SSE architecture that introduced *128*-bit registers. The Advanced Vector Extensions (AVX) extends the SSE architecture in several respects, e.g., by introducing non-destructive destination and floating point operations over *256*-bit registers [11]. AVX2 is the latest SIMD architecture. It has been recently disclosed, and will be first introduced in the next architecture (Codename "Haswell") in 2013 [12]. AVX2 includes sixteen *256*-bit registers (called YMM's), each one is capable of holding eight *32*-bit elements, or four *64*-bit elements. It also offers new integer instructions that operate on these wide registers.

Many algorithms (often in media processing, e.g., DCT) operate on multiple independent elements, and are therefore inherently suitable for SIMD architectures. However, big-numbers (multi-digit) arithmetic, which is in the heart of RSA computations, is not naturally suitable for vector architectures: the digits of the multi-digit numbers are not independent, due to carry propagation during arithmetic operations such as addition and multiplication.

We offer here an efficient method for using SIMD architecture for big-numbers arithmetic, in particular for modular exponentiation. Some attempts to use SIMD for big-numbers arithmetic (and RSA) have been made. For example, Page and Smart [19] suggested using SIMD architectures to calculate several exponentiations in parallel, and using a "redundant Montgomery representation" (which we call here Non Reduced) to avoid conditional final subtractions in Montgomery Multiplications. Lin [16] implemented a *128*-by-*128* bits integer multiplication function using SIMD instructions on Freescale's e600 *32*-bit processor, and used it as a building block for larger multiplications. Reference [10] suggests converting the big-numbers to numbers that have only *29*-bit digits, and use SIMD operations for multiplying them. Such a method is also used in [2] for prime field ECC. This is and underlying idea in this paper as well, although we do not use it (directly) for integer multiplications.

We use our algorithms for efficiently computing (a variant of) Montgomery Multiplications (and Squaring). The novelty in our approach includes the balancing of the computational workload between the SIMD and the ALU units, in an efficient manner. This resolves the bottlenecks that exist in a purely SIMD or purely ALU implementation.

2 Preliminaries

2.1 The RSA Context

In this paper, we discuss RSA cryptosystem with a $2n$-bit modulus, $N = P \times Q$, where P and Q are n-bit primes. Let the $2n$-bit private exponent be d. Decryption of a $2n$-bit message C requires one $2n$-bit modular exponentiation $C^d \bmod N$. To use the Chinese Remainder Theorem (CRT), the following quantities are pre-computed: $d_1 = d \bmod (P-1)$, $d_2 = d \bmod (Q-1)$, and $Qinv = Q^{-1} \bmod P$. Then, two n-bit modular exponentiations, namely $M_1 = C^{d1} \bmod P$ and $M_2 = C^{d2} \bmod Q$, are computed (M_1, M_2, d_1, d_2 are n-bit integers). The results are recombined by using $C^d \bmod$

$N = M_2+(Qinv \times (M_1-M_2) \bmod P) \times Q$. Using the CRT, the computational cost of a $2n$-bit RSA decryption is well approximated by twice the computational cost of one n-bit modular exponentiations. In our context, we can assume that by construction (of the RSA keys), $2^{n-1} < P, Q < 2^n$.

2.2 The Non Reduced Montgomery Multiplication

For a detailed description of software implementation of modular exponentiation, and the resulting performance, we refer the readers to [7]. In general, the critical building block in modular exponentiation computations is modular multiplication, or an equivalent. Here, we use the Non Reduced Montgomery Multiplication (*NRMM*), as defined in [4] (see also [20], [22]). *NRMM* is a variation of the well known Montgomery Multiplication (MM hereafter; see also [3], [14], [15], [21])

Definition 1. [Montgomery Multiplication] Let M be an odd integer (modulus), a, b be two integers such that $0 \le a, b < M$, and t be a positive integer (hereafter, all the variables are non-negative integers). The Montgomery Multiplication of a by b, modulo M, with respect to t, is defined by $MM(a, b) = a \times b \times 2^{-t} \bmod M$.

Definition 2. [Non Reduced Montgomery Multiplication] Let M be an odd integer (modulus), a, b be two integers such that $0 \le a, b < 2M$, and t be a positive integer such that $2^t > 4M$. The Non Reduced Montgomery Multiplication of a by b, modulo M, with respect to t, is defined by

$$NRMM(a,b) = \frac{a \times b + \left(\left(-M^{-1} \times a \times b\right) \bmod 2^t\right) \times M}{2^t} \tag{1}$$

We say that 2^t is the Montgomery parameter. For the Non Reduced Montgomery Square, we denote $NRMM(a, a) = NRMSQR(a)$.

The following lemma shows how *NRMM* can be used, similarly to *MM*, for efficient computations of modular exponentiation.

Lemma 1. Let M be an odd modulus a, b be two integers such that $0 \le a, b < 2M$, and t be a positive integer such that $2^t > 4M$. Then, a) $NRMM(a, b) < 2M$; b) $NRMM(a, 1) < M$.

Proof. To prove part a, we write

$$NRMM(a,b) < \frac{a \times b + 2^t \times M}{2^t} < \frac{2M \times 2M}{2^t} + M < \frac{4M^2}{4M} + M = 2M \tag{2}$$

To prove part b, we write

$$NRMM(a,1) < \frac{a + 2^t \times M}{2^t} < \frac{2M}{2^t} + M < \frac{1}{2} + M \tag{3}$$

The last inequality follows from $2^t > 4M$ and $a < 2M$. Therefore, *NRMM (a, 1)* is fully reduced modulo *M*.

Remark 1. Lemma 1 (part a) shows the "stability" of *NRMM*: the output of one *NRMM* can be used as an input to a subsequent *NRMM*.

Remark 2. Since *NRMM(a, b) mod M = MM(a, b)*, it follows, from the bound in Lemma 1, that *NRMM(a, b)* is either *MM(a, b)* or *MM(a, b)+M*.

Remark 3. Suppose that $0 \leq a, b < 2M$, $c2 = 2^{2t} \bmod M$, $a` = NRMM(a, c2)$, $b` = NRMM(b, c2)$, $u` = NRMM(a`, b`)$, and $u = NRMM(u`, 1)$. Then, Lemma 1 implies that $a`$, $b`$, $u`$ are smaller than $2M$, and $u = a \times b \bmod M$.

These remarks indicate how *NRMM* can be used for computing modular exponentiation in a way that is similar to the way in which *MM* is used. For a given modulus *M*, the constant $c2 = 2^{2t} \bmod M$ can be pre-computed. Then, $a^x \bmod M$ (for $0 \leq a < M$ and some integer x) can be computed by: a) Mapping the base (a) to the (non reduced) Montgomery domain, $a' = NRMM(a, c2)$ b) Using an exponentiation algorithm while replacing modular multiplications with *NRMM*'s c) Mapping the result back to the residues domain, $u = NRMM(u', 1)$.

2.3 The Relevant AVX2 Instructions

The AVX2 vector operations we use in our context are (see [12] for details):

- *VPADDQ* – addition of four *64*-bit integer values, from one YMM register and four *64*-bit values from another YMM register, and storing the result in a third YMM register.
- *VPMULUDQ* – multiplication of four *32*-bit unsigned integer values, from one YMM register, by four *32*-bit values from another YMM register, producing four 64-bit products into a third YMM register.
- *VPBROADCASTQ* – copying a given *64*-bit value, four times, to produce a YMM register with four equal elements (with that value).
- *VPERMQ* – Permutes *64*-bit values inside a YMM register, according to an *8*-bit immediate value.
- *VPSRLQ*/*VPSLLQ* – Shift *64*-bit values inside a YMM register, by an amount specified by an *8*-bit immediate value.

3 Modular Exponentiation with Vector Instructions

SIMD instructions are designed to repeat the same operation on *independent* elements stored in a register. Therefore, it has an inherent difficulty with efficiently handling carry propagation associated with big-numbers arithmetic. As an example, the carry propagation in a (*2* digit) × (*3* digit) multiplication, is illustrated in Fig. 1.

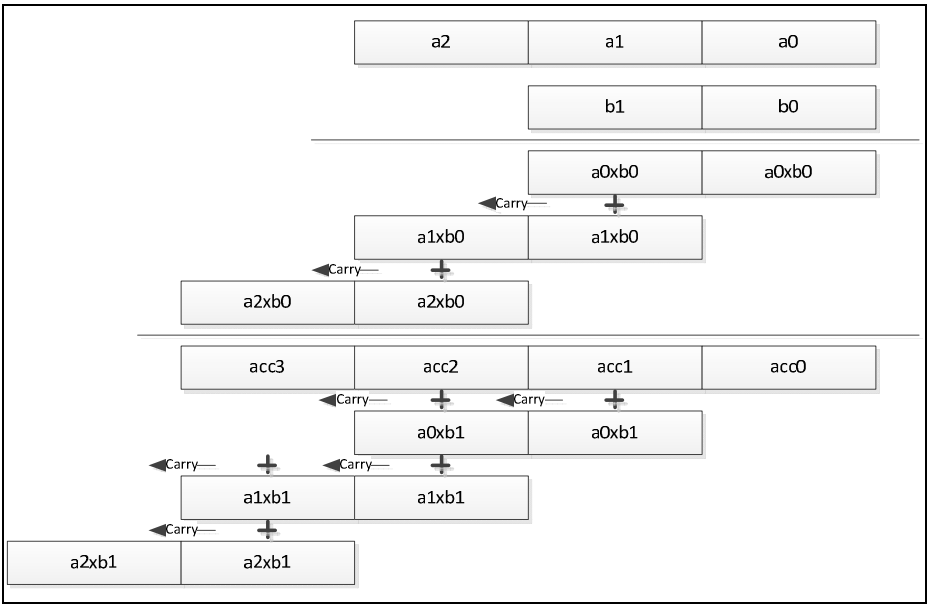

Fig. 1. Illustration of the carry propagation during a multiplication of the two integers A (3 digits) and B (2 digits). The schoolbook method is used. Each digit of A is multiplied by each digit of B, and the appropriate sub-products are aligned and summed accordingly. The digits of the partial sums are not independent of each other, due to the carry propagation, and this is why SIMD architectures would require some cumbersome manipulations to handle such a flow.

3.1 Redundant Representation

Modular exponentiation can be translated to a sequence of *NRMM*/*NRMSQR*'s. We show here how to optimize these operations, using vector instructions. The underlying idea is to always operate on "small" elements (i.e., less than 2^{32}). This allows two big-numbers products to be summed up, without causing an overflow inside the *64*-bit "container" that holds the digits of the accumulator. The cumbersome handling of the carry propagation can therefore be avoided. To this end, we define here an alternative representation of long (multi-digits) integers.

Let A be an n-bit integer, written in a radix 2^{64} as an l-digits integer, where $l = \lceil n/64 \rceil$, and where each *64*-bit digit a_i satisfies $0 \le a_i < 2^{64}$. This representation is unique. Consider a positive m such that $1 < m < 64$. We can write A in radix 2^m as $A = \sum_{i=0}^{k-1} x_i \times 2^{m \times i}$. This representation is unique, and requires $k = \lceil n/m \rceil > l$ digits, x_i, satisfying $0 \le x_i < 2^m$, for $i = 0, \ldots, k-1$. See Fig. 2 for an example.

If we relax the requirement $0 \le x_i < 2^m$, and allow the digits to satisfy only the inequality $0 \le x_i < 2^{64}$, we say that A is written in a Redundant-radix-2^m Representation (redundant representation for short). This representation is not unique. Figuratively speaking, the redundant representation is simply "embedding the digits of a number in a big container".

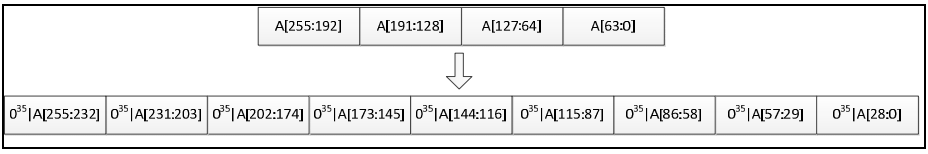

Fig. 2. A *256*-bit integer in radix 2^{64} (*n=256, l=4*) written in radix 2^{29} (*m=29*) using *9* digits (*k = ceil(n/m) = 9*). Each digit can be stored as a *64*-bit "element" in a vector of *k=9* elements.

An integer *A*, written with *k* digits in redundant representation, satisfying $A < 2^{m \times k}$, can be converted to a radix 2^m representation with the *same* number of digits (*k*), as shown in Algorithm 1 (Fig. 3).

```
Algorithm 1: Redundant-to-2^m
Input: U in redundant representation using k digits(assumption: U < 2^(m×k))
Output: U in radix 2^m representation
Flow:
   1.  temp = 0
   2.  For i = 0 to k-1
          a.  temp = temp + u_i
          b.  v_i = temp mod 2^m
          c.  temp = temp/2^m
   End for
   Return V
```

Fig. 3. Redundant-to-2^m conversion (see explanation in the text)

Example 1: take *n=1024* and the *1024*-bit number $A = 2^{1024}-105$. It has $l = 16$ digits in radix 2^{64}. The least significant digit is *0xFFFFFFFFFFFFFF97*, and the other *15* digits are *0xFFFFFFFFFFFFFFFF*. With $m = 28$, A becomes a number with $k = 37$ digits in radix 2^{28} the least significant digit is *0xFFFFF97*, the most significant digit is *0xFFFF*, and the rest are *0xFFFFFFF*. For *m=29*, A becomes a number with *k=36* digits in radix 2^{29} the least significant digit is *0x1FFFFF97* the most significant digit is *0x1FF* and the rest are *0x1FFFFFFF*.

Operations on integers that are given in redundant representation can be "vectorized", as follows:

Let X and Y be two numbers given in redundant representation, such that $X = \sum_{i=0}^{k-1} x_i \times 2^{m \times i}$, $Y = \sum_{i=0}^{k-1} y_i \times 2^{m \times i}$, with $0 \le x_i, y_i < 2^{64}$. Let $t > 0$ be an integer.

Addition: If $x_i + y_i < 2^{64}$ for *i = 0, 1, …, (k-1)*, then the sum $X + Y$ is given, in redundant representation, by $X + Y = \sum_{i=0}^{k-1} z_i \times 2^{m \times i}$ with $z_i = x_i + y_i; 0 \le z_i < 2^{64}$.

Multiplication by constant: If $x_i \times t < 2^{64}$ for *i = 0, 1, …, (k-1)*, then the product $t \times X$ is given by $t \times X = \sum_{i=0}^{k-1} z_i \times 2^{m \times i}$ with $z_i = t \times x_i; 0 \le z_i < 2^{64}$.

To illustrate, we provide the following example.

Example 2: *n=256*, *m=29*.

```
A = fa5401a8593c981b||fd42a2802a750928||e930850d63bc2c5f||da8d4ca9655091ad
B = eb9a100d6e586233||50608103451895a2||5572dfe2de045f13||132ba675e3adb497
A in radix 2^29 =
0000000000fa5401||00000000150b2793||00000000006ff50a||000000001140153a||0000000010
928e93||00000000010a1ac7||000000000f0b17f6||00000000146a654b||00000000055091ad
B in radix 2^29 =
0000000000eb9a10||0000000001adcb0c||0000000008cd4182||000000000081a28c||0000000009
5a2557||0000000005bfc5bc||000000000117c4c4||00000000195d33af||0000000003adb497
A + B in redundant representation =
0000000001e5ee11||0000000016b8f29f||00000000093d368c||0000000011c1b7c6||0000000019
ecb3ea||0000000006c9e083||000000001022dcba||000000002dc798fa||0000000008fe4644
A + B in radix 2^29 =
0000000001e5ee11||0000000016b8f29f||00000000093d368c||0000000011c1b7c6||0000000019
ecb3ea||0000000006c9e083||000000001022dcbb||000000000dc798fa||0000000008fe4644
t = 0x00000001fedcba98
t × A in redundant representation =
01f38b30a4869a98||29fe5dc375b44d48||00df6ab21b1ac1f0||226c89ee9750be70||2112420fb2
ef7548||021306c96a787c28||1e05123ba966f610||28bd901be338a288||0a9b1753885a30b8
t × A in radix 2^29 =
000000000f9c598f||00000000147988b3||000000001cafa2e1||000000000e7f116c||000000001f
e2ceee||000000000387ab9a||000000001aa10e0f||000000000f5376f1||0000000018115d24||00
000000085a30b8
```

In Fig. 5, we illustrate how redundant representation helps delaying the carry propagation to the last stage of the multiplication.

3.2 NRMM

NRMM can be computed in a "word-by-word" style, as shown in Algorithm 2, Fig. 4.

Algorithm 2: Word-by-word computation of NRMM (WW-NRMM)
Input:
M, an odd modulus such that $2^{n-1} < M < 2^n$ (M has n bits in binary form)
Integer $1 < m < 64$, such that if $k = \lceil n/m \rceil$, then $k \times m > n+2$.
A, B < $2M < 2^{n+1}$ given in radix 2^m (a_i, b_i denote their radix 2^m digits)
Pre-computed: $k_0 = -M^{-1} \bmod 2^m$
Output: X = NRMM (A, B)
Flow:
1. X = 0
2. For i = 0 to k-1
   2.1. $X = X + A \times b_i$
   2.2. $x_0 = X \bmod 2^m$
   2.3. $y = x_0 \times k_0 \bmod 2^m$
   2.4. $X = X + M \times y$
   2.5. $X = X/2^m$
3. Output X

Fig. 4. Word-by-word computation of NRMM

Proof of correctness: note that $X = (A \times B + ((-M^{-1} \times A \times B) \bmod 2^{k \times m}) \times M)/2^{k \times m}$. Therefore, (a) follows immediately and (b) follows from Lemma 1. Note also that if $B = 1$, then $X \bmod M = A \times B \times 2^{-l \times m}$ exactly. In step 2.5, X is divisible by 2^m due to steps 2.4-2.4 and the definition of k_0.The number of digits in the final result remains unchanged (k), because the result $X < 2^{n+1}$, and $k \times m > n+2$.

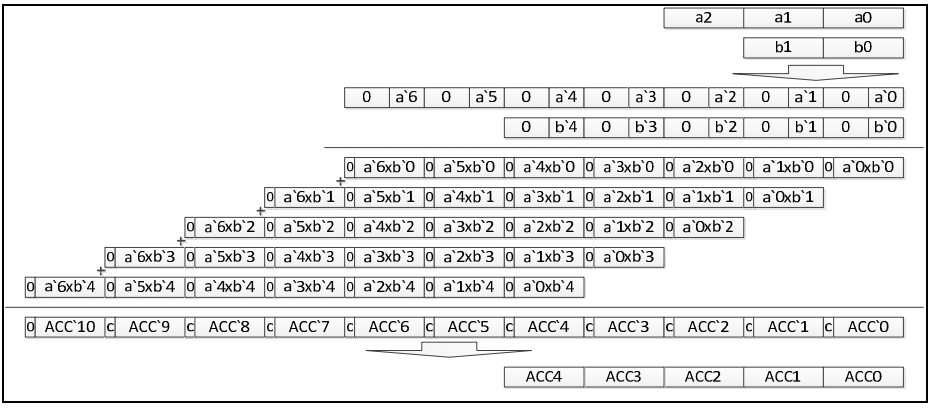

Fig. 5. Illustration of carry accumulation during a multiplication of two integers, A and B, using vectorized computations (compare to Fig. 1). Suppose that $m=29$. A (*3* digits) and B (*2* digits) are first converted into redundant representation in radix 2^{29}. In this representation, they have *7* and *5* digits, where each digit is smaller than 2^{29}, and is stored in a *64*-bit "container" (SIMD element). Then, the sub-products are accumulated, while the carry bits spill into the "empty space" inside the *64*-bit container (which is initially *0*). In the end, the result is "normalized" back to a standard 2^m representation, according to Algorithm 1. This result can be fed to a consecutive multiplication, or transformed back to radix 2^{64}.

Remark 4. In the redundant representation, steps 2.1 and 2.4 can be computed efficiently using vector instructions (*VPMULUDQ*). Steps 2.2 and 2.3 can be computed efficiently using scalar instructions, because they operate on a single digit.

The above remark indicates that Algorithm 2 is useful for computing *NRMM* via a mix of scalar and vector instructions. Indeed, our implementation uses scalar instructions to compute the low digits of *X*, and vector instructions for the other computations, described in Algorithm 3.

Algorithm 3, for computing *NRMM* (with the parameter *m*) assumes the inputs *A*, *B* and *M* are represented in redundant form, with each digit is strictly smaller than 2^m (we call it "normalized redundant representation").

Since we want to integrate such an *NRMM* implementation into a "standard" implementation, we need to transform our input/output from/to a regular (radix 2^{64}) representation. The cost of such transformation is only few tens of cycles, but if done for every *NRMM*, it can add up to a noticeable overhead. For efficiency, we transform the inputs to a redundant form in the beginning of the exponentiation; carry out all the operations in the redundant form, during the entire exponentiation; in the end, transform the result back to the standard representation. This makes the overhead of the to-and-from transformation negligible, while keeping a standard interface for the exponentiation function (transparent to the user).

Algorithm 3 [VNRMM]: Vectorized implementation of NRMM(A,B)
Input: A, B and M, in radix 2^m
Pre-computed: $k_0 = -M^{-1} \bmod M$
Output: X = NRMM (A, B)
Flow:
1. $x_0 = 0$, X_q, …,$X_0 = 0$
2. $a_0 = A \bmod 2^m$ (i.e., digit 0 of A)
3. $m_0 = M \bmod 2^m$ (i.e., digit 0 of M)
4. load digits 1, 2, …, (k-1) of A into SIMD registers $A_1..A_q$ (q as required)
5. load digits 1, 2, …, (k-1) of M into SIMD registers $M_1..M_q$ (q as required)
6. addCounter = 0
7. for i = 0 to k-1
   7.1. $x_0 = x_0 + a_0 \times b_i$
   7.2. T = Broadcast b_i
   7.3. for j = 1 to q
        7.3.1. $X_j = X_j + A_j \times T$
   7.4. $y_0 = x_0 \times k_0 \bmod 2^m$
   7.5. $x_0 = x_0 + m_0 \times y_0$
   7.6. T = Broadcast y_0
   7.7. for j = 1 to q
        7.7.1. $X_j = X_j + M_j \times T$
   7.8. $x_0 = x_0 >> m$
   7.9. $x_0 = x_0 + X_1[0]$
   7.10. $X_q,\ldots,X_1 = X_q,\ldots,X_1 >> 64$
   7.11. addCounter = addCounter + 2
   7.12. if addCounter ≥ (2^{64-2m})
        7.12.1. perform X "cleanup"
        7.12.2. addCounter = 0
End for
8. Convert $X_q,..,X_1$, x_0: from redundant representation to radix 2^m (using Algorithm 1)

Fig. 6. Computing NRMM using combination of vector and scalar operations. In order to avoid carry overflow, a "cleanup" procedure may be initiated, converting X to normalized 2^m representation.

3.3 Modular Exponentiation Using VNRMM

We note that a windowed method modular exponentiation requires judicious preparation and use of tables (see [7] for details). In the vectorized implementations, these tables are larger than the tables that are used for the scalar implementation (that uses *64*-bit digits), because the redundant representation requires more than twice as many digits. However, since the top *64-m* bits of each digit are zeroed in the end of each NRMM call, and therefore do not need to be stored, we can decrease the size of the required tables, to some extent.

We briefly mention that in order to be side-channel protected, our implementation operates in "constant time": the memory access patterns (and timing) do not depend on the secret exponent. This is achieved (among other factors) by holding the tables in a way that a portion of each entry lies in a portion of each cache-line that is used by the table. Details on choosing the optimal table size and on implementing side-channel protected table access, is provided in [7].

Algorithm 4: w-ary modular exponentiation using VNRMM
Input: A, X and M – n-bit integers, in radix 2^{64} representation
Pre-computed: $k_0 = -M^{-1} \bmod 2^{64}$, $RR = 2^{2n} \bmod M$, w – window size
Output: $C = A^X \bmod M$
Flow:

1. Let m be the largest integer such that $2^{64-2m} > 2 \times \lceil n/m \rceil$
2. Let A`, RR` and M` be A, RR and M converted to normalized radix 2^m
3. Let X be $x0 + x1 \times 2^w + \ldots + xj \times 2^{jw}$, where $0 \le x0, x1 \ldots xj < 2^w$
4. Let $k_0` = k_0 \bmod 2^m = -M^{-1} \bmod 2^m$
5. C2 = VNRMM(RR`, RR`) (congruent to $2^{4n-k \times m} \bmod M$)
6. C2 = VNRMM(C2, 4k×m – 4n) (congruent to $2^{2k \times m} \bmod M$)
7. Table[0] = VNRMM(C2, 1)
8. Table[1] = VNRMM(C2, A`)
9. For i = 2,..,2^{w-1}-1 do
   - 9.1. Table[i×2]=VNRMM(M[i], M[i])
   - 9.2. Table[i×2]=VNRMM(M[i×2], M[1])

   End for
10. h = m[xj]
11. For i=j-1,…0 do
    - 11.1. For l = 1,…,w
      - 11.1.1. h = VNRMM(h,h)

      End for
    - 11.2. h = VNRMM(h, xi)

    End for
12. h = VNRMM(h, 1)
13. hh = radix-2^m-to-radix-2^{64}(h)
14. Return hh

- The access to the Table is side channel protected

Fig. 7. The w-ary modular exponentiation, using VNRMM

4 Implementation, Choice of Parameters and Optimizations

4.1 Choice of Parameters

For our usages, we are mainly interested in *n=512*, *1024*, *2048*, (for RSA1024, RSA2048, RSA4096, respectively). We choose *m=29* if *n=512* or *n=1024*, and *m=28* when *n=2048*.

To explain this choice of parameters, we first point out that *m = 29* for *n = 1024* is larger than the value of *m* that is specified in Step 1 of Algorithm 4 (namely *m=28*). Indeed, the correctness of Algorithm 4 can be maintained with different choices of *m*, as long as the "cleanup" steps are properly applied, to prevent any overflows beyond the range allowed by the *64* bits container. The tradeoff is clear: a large *m* decreases the number of digits of the operands – which improves the efficiency of the computations. On the other hand, it requires a more frequent "cleanup", because a fewer "spare" bits are left for accumulation.

In our case, where *n=1024*, choosing *m=29* leads to *36* digits operands, which results in *58*-bit products, and leaves only *6* "spare" bits for carry-accumulation. Therefore, cleanup is required after $2^6=64$ accumulations, that is, every *32* iterations of the loop (Step 7) in Algorithm 3. For *n=1024*, this loop repeats *36* iterations, so the cleanup is required only once. With *m=28*, we have *37* digits operands, and *8* "spare"

bits, therefore the cleanup is required every *128* iterations of the loop, allowing exponents of up to *3584* bits without any cleanup.

In our implementation, we optimize the cleanup step (shaving off only the necessary number of bits), and make $m=29$ the preferable parameter choice.

4.2 Why Is the AVX2 Architecture Sufficient for an Efficient Vectorized Implementation?

We explain here why AVX2 is the first SIMD architecture that can support vectorized *NRMM* implementation that can outperform the scalar implementation.

For simplicity, we consider a schoolbook scenario, and count the operations and the tradeoffs. Computing *NRMM* in redundant representation requires $2\times[n/m]^2$ single precision multiplications. Similarly, the scalar implementation (in radix 2^{64}) requires $2\times[n/64]^2$ single precision multiplications. However, *NRMM* in redundant representation requires only one single precision addition per multiplication, whereas the scalar implementation requires three single precision add-with-carry operations.

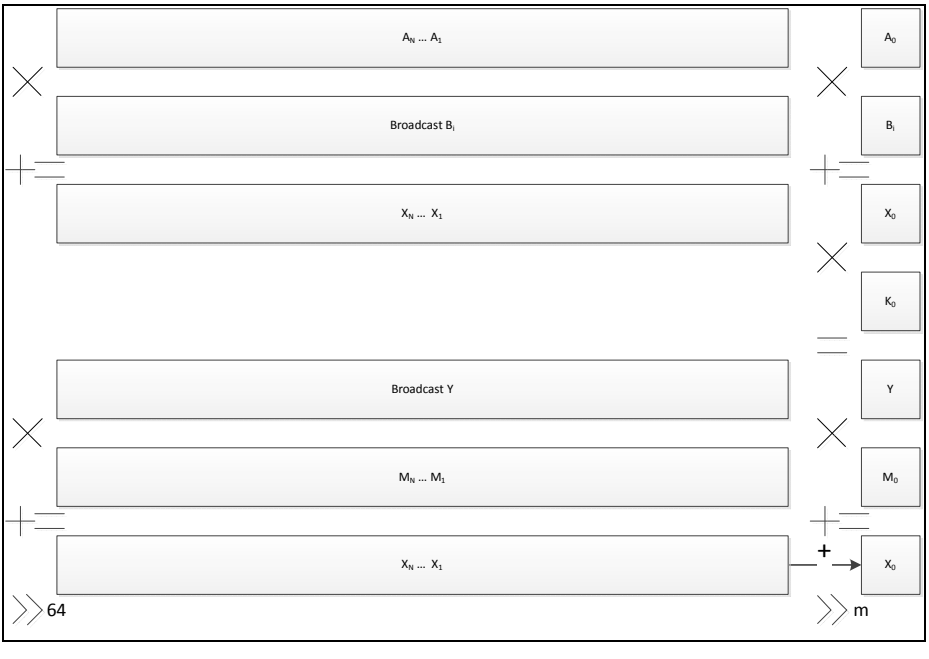

Fig. 8. Illustration of VNRMM as described in Fig. 6, as a flow of vector instructions (on the left) and scalar (ALU) instructions (on the right)

For example, the *1024*-bit *NRMM* using scalar implementation, requires *~512* multiplications and *~1536* additions. The redundant implementation with $m=29$ requires *~2592* multiplications and *~2592* additions. The total number of multiplications and additions for the scalar implementation with radix 2^{64} is *~2048* and for the *NRMM* the instruction total is *~5184* instructions.

Consequently, to make the vectorized code outperform the scalar implementation, it has to run on a SIMD architecture that can execute a factor of *2.53* more single precision operations than the scalar implementation. In other words, we need a SIMD architecture that accommodates *2.53* digits (of *64* bits), implying registers of at least *162* bits. The soon-to-appear AVX2 architecture has registers of *256* bits (*4* digits) with the appropriate integer instructions, and is therefore the first SIMD generation with the potential to support fast vectorized implementations. We demonstrate here how this potential can be realized.

By plain counting, one might suspect that the existing (narrower) SIMD architectures, could also support a vectorized implementation (of modular exponentiation) that outperforms the scalar implementation. We argue here that this is not the case on platforms with an efficient *64*-bit *mul* and *adc* performance, because the overhead associated with the vectorized implementation is too high. To illustrate, we consider the SSE3 instructions set, with SSSE3 extensions. We point out that:

- The SSE architectures have only *16* xmm registers. This allows for storing only *2048* bit of data at a time, which is not enough to keep the accumulators in registers. This adds overhead for memory traffic.
- The SSE architectures operate only on *16*-byte aligned memory operands, which hinders the "shift-save" optimization.
- The lack of a "broadcast" instruction adds an overhead for loading the digits of the operands.

4.3 Optimizing the Implementation

Implementing *NRMM* as in Algorithm 3 is rather straightforward, as illustrated in Fig. 8. However, we identify two bottlenecks in that implementation:

- The expensive right shifting of a vector (of digits) across several registers.
- The latency between the computation of y_0 (in step 7.4 of Algorithm 3), followed by broadcasting to a SIMD register, and the point in time where the multiplications in step 7.7 can start.

To address these bottlenecks, we use the following optimizations. We first note that instead of right shifting X, we can keep the values $A_q..A_1$ and $M_q..M_1$ in memory, and use "unaligned" *VPMULUDQ* operations with the proper address offset. To do this correctly, the operands A and M are padded with zeroes. For the second bottleneck, we preemptively calculate a few digits, using ALU instructions, to achieve a better pipelining of the ALU and SIMD units.

These optimizations are illustrated, schematically, in Fig. 10. Furthermore, they are implemented in real code, in the form of an OpenSSL patch, which the readers can download from [8] and examine.

4.4 Vectorized Redundant Montgomery Square

Modular exponentiation involves *NRMM,* majority of which are of the form *NRMM(A,A)*. We therefore add dedicated optimization for this case and call it

NRMSQR. Unlike *NRMM*, where we interleave the operations, the function *NRMSQR(A)* starts with calculating A^2, followed by a Montgomery Reduction. Our implementation employs the big-numbers squaring method published in [6]: first creating a copy of *A*, left shifted by *1*. In the redundant representation, this operation is simple: merely left shifting by *1*, of each element. Subsequently, the elements of *A*, and "*A<<1*" are multiplied, as described in Fig. 9.

Algorithm 5 [VNRMSQR]: Vectorized implementation of NRMSQR(A)
Input: A and M, in radix 2^m
Pre-computed: $k_0 = -M^{-1} \bmod M$
Output: X, such that X mod M = $A^2 \times 2^{-k \times m} \bmod M$ and $X < 2^{n+1}$
Flow:
1. Let A` = A×2 (i.e. $a`_i = a_i$ << 1)
2. Let s be the number of 64-bit elements in a SIMD register
3. $X_{2q+1}, \ldots, X_0 = 0$
//First stage – perform the square
4. load digits 0, 2, …, (k-1) of A into SIMD registers $A_0..A_q$ load digits 0, 2, …, (k-1) of A` into SIMD registers $A`_0..A`_q$
5. for i = 0 to $\lceil k-1/s \rceil$
   5.1. for j = 0 to s-1
      5.1.1. T = Broadcast $a_{i \times s+j}$
      5.1.2. $X_{2q+1},\ldots,X_i = X_{2q+1},\ldots,X_i + (A`_q,\ldots,A`_{i+1},A_i * T << 2^{64 \times (i \times s+j)})$
//Second stage – perform word-by-word reduction
6. $m_0 = M \bmod 2^m$ (i.e., digit 0 of M)
7. $x_0 = X_0[0]$
8. load digits 1, 2, …, (k-1) of M into SIMD registers $M_0..M_q$
9. $X_q,\ldots,X_0 = X_q,\ldots,X_0 >> 64$
10. for i = 0 to k-1
   10.1. $y_0 = x_0 \times k_0 \bmod 2^m$
   10.2. $x_0 = x_0 + m_0 \times y_0$
   10.3. T = Broadcast y_0
   10.4. $X_q,\ldots,X_0 = X_q,\ldots,X_0 + (M_0,\ldots,M_0 * T)$
   10.5. $x_0 = x_0 >> m$
   10.6. $x_0 = x_0 + X_0[0]$
   10.7. $X_q,\ldots,X_0 = X_q,\ldots,X_0 >> 64$
   End for
11. for j = 1 to q
   11.1. $X_j = X_j + X_{j+q+1}$
12. Convert $X_q,..,X_1$, x_0 from redundant representation to radix 2^m according to Algorithm 1.

Fig. 9. Optimized NRMSQR: using combination of vector and scalar instructions

5 Results

To assess our algorithm, we implemented an optimized *1024*-bit modular exponentiation code, using the described mix of vector (AVX2) and scalar instructions. We integrated this code into OpenSSL, in the form of a fully functional OpenSSL patch, which accelerates RSA2048. We call this implementation "RSAZ-AVX2" (the "Z" is for "Zariz" - "fast" in Hebrew). The patch is available at [8].

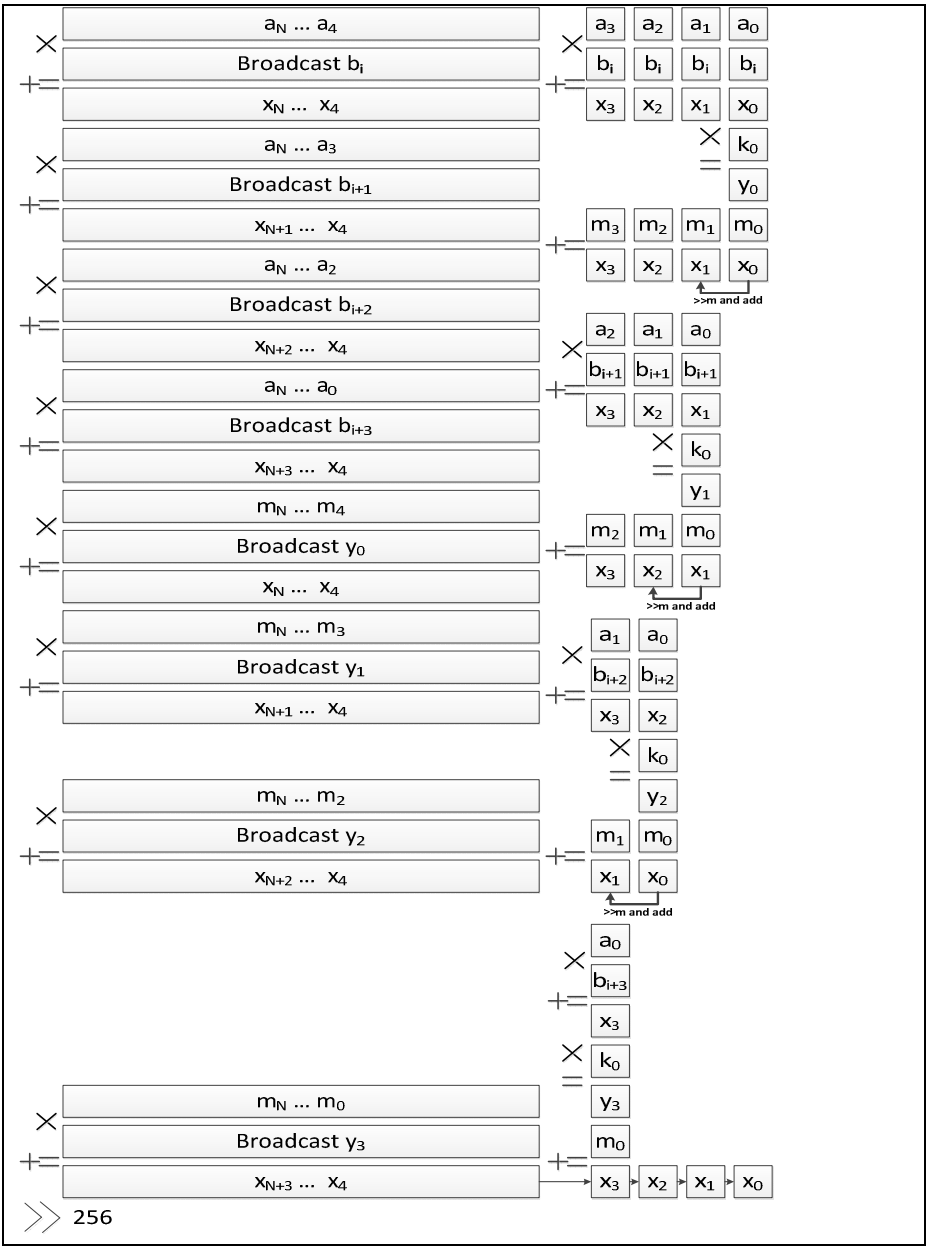

Fig. 10. Optimized VNRMM computation, via a flow of vector instructions (on the left) and scalar instructions (on the right). To reduce bottlenecks, four redundant digits are loaded into GPRs, and handled using ALU instructions, while the rest is handled via SIMD instructions. Vector shifting is performed only once per four digits.

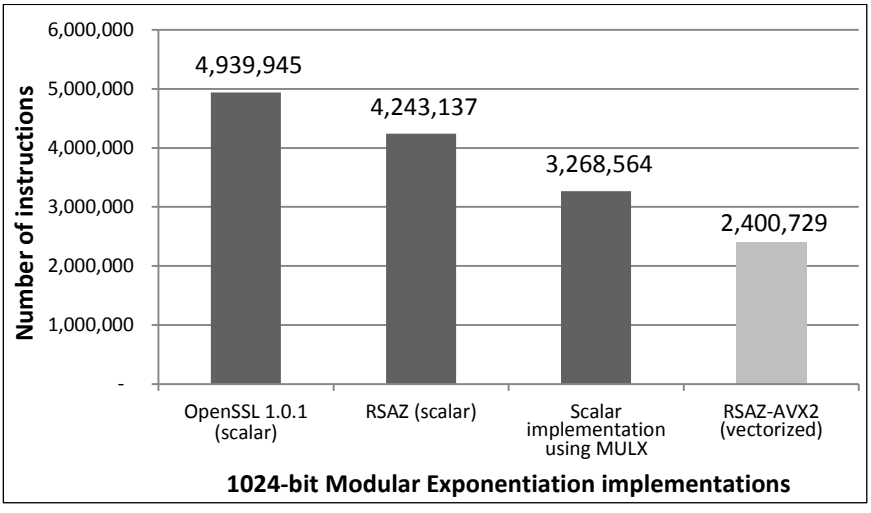

Fig. 11. The number of instructions in scalar and vector implementations of 1024-bit modular exponentiation (see explanation in the text)

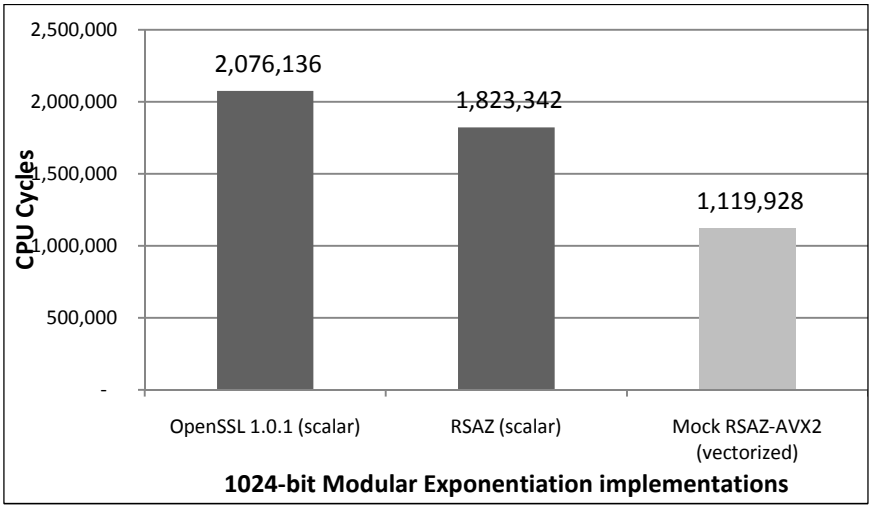

Fig. 12. Cycles count for a *1024*-bit modular exponentiation: Mock RSAZ-AVX2 (see explanation in the text) compared to other (real) scalar codes. The performance was measured on an Intel® Core™ i7-2600K processor.

This code can be compiled and tested for correctness using the existing public tools [23], [13]. However, we cannot report real performance figures, in cycles, at the time this paper is published: the Intel processor (Codename "Haswell") that supports AVX2 will be available only in 2013 [12].

To demonstrate the power of our method, we overcome this difficulty by approximating the speedup via counting the number of instructions in different

modular exponentiation implementations. The instruction can be counted using the Intel SDE tool, with the "-mix" flag, as described in [13]. As the baseline, we used three implementations. The first two are public and can be run on the existing processors: the current (official) OpenSSL 1.0.1 [18], and the best known implementation (which we call RSAZ) [5]. In addition, to make sure that we compare to the best-future-implementation, we also generated a new scalar implementation that uses the new scalar instruction *MULX*, which is expected to be faster than the existing scalar implementations (the *MULX* instruction will appear, together with the AVX2 instructions, in the coming "Haswell" processor [12]). The instructions count for each implementation is provided in Fig. 11.

Fig. 11 shows that our method requires less than half of the number required by the current OpenSSL 1.0.1 implementation, *~43%* fewer instructions than the currently best known scalar implementation, and *~26%* fewer instructions than a future scalar code that uses the (coming) *MULX* instruction. This clearly indicates that our implementation outperforms the alternatives by a significant margin.

We point out that the instructions count is only an approximation, and cannot be translated directly to CPU cycles, in an out-of-order architecture. To this end, we add another comparative approximation via a "Mock implementation". We took our new modular exponentiation code and replaced all of the AVX2 instructions with AVX1 instructions (the rationale is that most of these instructions are merely a wider version of their AVX1 counterparts). This allowed us to measure the performance (of the mock modular exponentiation) on an existing processor. Performance wise, this gives us yet another *hint* to the expected performance (although the output is functionally incorrect). The numbers are presented in Fig. 12, showing that the mock RSAZ-AVX2 implementation is *1.85* times faster than OpenSSL 1.0.1, and *1.63* times faster than the best known scalar implementation (RSAZ). This improvement is quite consistent with the improvement shown in Fig. 11

6 Conclusion

We introduced here a new method and implementation for computing modular exponentiation, thus accelerating software performance of RSA on modern high end processors. Our algorithm utilized the coming AVX2 architecture, and a special balance between scalar and vector operations. The demonstrated results show that a significant performance gain (up to *40%* over current implementations) will be available together with the release of the new Intel® Architecture Codename Haswell.

Our vectorization method is scalable, and can gain performance from any wide (and wider) SIMD architectures. Finally, we point out that our method can also be used on AVX/SSE architectures. This achieves a significant performance gain for (low end) processors that have AVX/SSE instructions, and only a *32*-bit ALU unit, such as the current Atom processors.

References

1. Barker, E., Roginsky, A.: Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, p. 5. NIST Special Publication 800-131A (2011) `http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf`

2. Bernstein, J.D.: Curve25519: New Diffie-Hellman speed records (2006)
3. Brent, R., Zimmermann, P.: Modern Computer Arithmetic. Cambridge University Press (2010), http://www.loria.fr/~zimmerma/mca/pub226.html (retrieved)
4. Gueron, S.: Enhanced Montgomery Multiplication. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 46–56. Springer, Heidelberg (2003)
5. Gueron, S., Krasnov, V.: Efficient and side channel analysis resistant 512-bit and 1024-bit modular exponentiation for optimizing RSA1024 and RSA2048 on x86_64 platforms, OpenSSL #2582 patch (posted August 2011), http://rt.openssl.org/Ticket/Display.html?id=2582&user=guest&pass=guest
6. Gueron, S., Krasnov, V.: Speeding up Big-numbers Squaring. In: IEEE Proceedings of 9th International Conference on Information Technology: New Generations (ITNG 2012), pp. 821–823 (2012)
7. Gueron, S.: Efficient Software Implementations of Modular Exponentiation. Journal of Cryptographic Engineering 2, 31–43 (2012),
8. Gueron, S., Krasnov, V.: Efficient, and side channel analysis resistant 1024-bit modular exponentiation, for optimizing RSA2048 on AVX2 capable x86_64 platforms, OpenSSL patch (posted June 2012), http://rt.openssl.org/
9. Hassaballah, M., Omran, S., Mahdy, Y.B.: A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications. The Computer Journal 51(6), 630–649 (2007)
10. Intel: Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications (2006)
11. Intel: Intel Advanced Vector Extensions Programming Reference, http://software.intel.com/file/36945
12. Buxton, M. (Intel): Haswell New Instruction Descriptions Now Available!, http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/
13. Intel: Software Development Emulator (SDE), http://software.intel.com/en-us/articles/intel-software-development-emulator/
14. Koc, Ç.K., Kaliski, B.S.: Analyzing and Comparing Montgomery Multiplication Algorithms. Micro 16(3), 26–33 (1996), http://islab.oregonstate.edu/papers/j37acmon.pdf
15. Koç, Ç.K., Walter, C.D.: Montgomery Arithmetic. In: van Tilborg, H. (ed.) Encyclopedia of Cryptography and Security, pp. 394–398. Springer (2005)
16. Lin, B.: Solving Sequential Problems in Parallel. Application Note, Freescale Semiconductor (2006)
17. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography, 5th printing. CRC Press (2001)
18. OpenSSL: The Open Source toolkit for SSL/TLS, http://www.openssl.org/
19. Page, D., Smart, P.: Parallel Cryptographic Arithmetic Using a Redundant Montgomery Representation. IEEE Transactions on Computers 53(11), 1474–1482 (2004)
20. Walter, C.D.: Montgomery exponentiation needs no final subtractions. Electron. Lett. 35, 1831–1832 (1999)
21. Walter, C.D.: Montgomery's Multiplication Technique: How to Make It Smaller and Faster. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 80–93. Springer, Heidelberg (1999)
22. Walter, C.D.: Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 30–39. Springer, Heidelberg (2002)
23. YASM: The YASM Modular Assembler Project, http://yasm.tortall.net/