Montgomery Multiplication on the Cell

École Polytechnique Fédérale de Lausanne (EPFL),
CH-1015 Lausanne, Switzerland
{joppe.bos,marcelo.kaihara}@epfl.ch

Abstract. A technique to speed up Montgomery multiplication targeted at the Synergistic Processor Elements (SPE) of the Cell Broadband Engine is proposed. The technique consists of splitting a number into four consecutive parts. These parts are placed one by one in each of the four element positions of a vector, representing columns in a 4-SIMD organization. This representation enables arithmetic to be performed in a 4-SIMD fashion. An implementation of the Montgomery multiplication using this technique is up to 2.47 times faster compared to an unrolled implementation of Montgomery multiplication, which is part of the IBM multi-precision math library, for odd moduli of length 160 to 2048 bits. The presented technique can also be applied to speed up Montgomery multiplication on other SIMD-architectures.

Keywords: Cell Broadband Engine, Cryptology, Computer Arithmetic, Montgomery Multiplication, Single Instruction Multiple Data (SIMD).

1 Introduction

Modular multiplication is one of the basic operations in almost all modern public-key cryptographic applications. For example, cryptographic operations in RSA [1], using practical security parameters, requires a sequence of modular multiplications using a composite modulus ranging from 1024 to 2048 bits. In elliptic curve cryptography (ECC) [2,3], the efficiency of elliptic curve arithmetic over large prime fields relies on the performance of modular multiplication. In ECC, the length of most commonly used (prime) moduli ranges from 160 to 512 bits.

Among several approaches to speed up modular multiplication that have been proposed in the literature, a widely used choice is the Montgomery modular multiplication algorithm [4]. In the current work, we study Montgomery modular multiplication on the Cell Broadband Engine (Cell). The Cell is an heterogeneous processor and has been used as a cryptographic accelerator [5,6,7,8] as well as for cryptanalysis [9,10].

In this article, a technique to speed up Montgomery multiplication that exploits the capabilities of the SPE architecture is presented. This technique consists of splitting a number into four consecutive parts which are placed one by one in each of the four element positions of a vector. These parts can be seen as four columns in a 4-SIMD organization. This representation benefits from

R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 477–485, 2010.

the features of the Cell, e.g. it enables the use of the 4-SIMD multiply-and-add instruction and makes use of the large register file. The division by a power of 2, required in one iteration of Montgomery reduction, can be performed by a vector shift and an inexpensive circular change of the indices of the vectors that accumulate the partial products. Our experimental results show that an implementation of Montgomery multiplication, for moduli of sizes between 160 and 2048 bits, based on our newly proposed representation is up to 2.47 times faster compared to an unrolled implementation of Montgomery multiplication in the IBM's Multi-Precision Math (MPM) Library [11].

The article is organized as follows. In Section 2, a brief explanation of the Cell broadband engine architecture is given. In Section 3, we describe the Montgomery multiplication algorithm. In Section 4, our new technique for Montgomery multiplication is presented. In Section 5 performance results of different implementations are given. Section 6 concludes this paper.

2 Cell Broadband Engine Architecture

The Cell architecture [12], developed by Sony, Toshiba and IBM, has as a main processing unit, a dual-threaded 64-bit Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The SPEs are the workhorses of the Cell and the main interest in this article. The SPE consist of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). Every SPU has access to a register file of 128 entries, called vectors or quad-words of 128-bit length, and a 256 kilobyte Local Store (LS) with room for instructions and data. The main memory can be accessed through explicit direct memory access requests to the MFC. The SPUs have a 128-bit SIMD organization allowing sixteen 8-bit, eight 16-bit or four 32-bit integer computations in parallel. The SPUs are asymmetric processors that have two pipelines denoted as even and odd pipelines. This means that two instructions can be dispatched every clock cycle. Most of the arithmetic instructions are executed on the even pipeline and most of the memory instructions are executed on the odd pipeline. It is a challenge to fully utilize both pipelines always at the same time. The SPEs have no hardware branch-prediction. Instead, hints can be provided by the programmer (or the compiler) to the instruction fetch unit that specifies where a branch instruction will jump to.

An additional advantage of the SPE architecture is the availability of a rich instruction set. With a single instruction, four 16-bit integer multiplication can be executed in parallel. An additional performance improvement may be achieved with the multiply-and-add instruction which performs a 16×16-bit unsigned multiplication and an addition of the 32-bit unsigned operand to the 32-bit multiplication result. This instruction has the same latency as a single 16×16-bit multiplication and requires the 16-bit operands to be placed in the higher positions of the 32-bit sections (carries are not generated for this instruction).

One of the first applications of the Cell processor was to serve as the heart of Sony's PS3 video game console. The Cell contains eight SPEs, and in the

Input: $M : r^{n-1} \leq M < r^n, 2 \nmid M$
$X, Y : 0 \leq X, Y < M$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$

$Z = 0$
for $i = 0$ to $n - 1$ **do**
  $Z = Z + y_i \cdot X$
  $q = (-Z \cdot M^{-1}) \bmod r$
  $Z = (Z + q \cdot M)/r$
end for
if $Z \geq M$ **then**
  $Z = Z - M$
end if

Algorithm 1. Radix-r Montgomery Multiplication[4]

PS3 one of them is disabled. Another SPEs is reserved by Sony's hypervisor (a software layer which is used to virtualize devices and other resources in order to provide a virtual machine environment to, e.g., Linux OS). In the end, six SPEs can be accessed when running Linux OS on the PS3.

3 Montgomery Modular Multiplication

The Montgomery modular multiplication method introduced in [4] consists of transforming each of the operands into a Montgomery representation and carry out the computation by replacing the conventional modular multiplications by Montgomery multiplications. This is suitable to speed up, for example, modular exponentiations which can be decomposed as a sequence of several modular multiplications. One of the advantages of this method is that the computational complexity is usually better compared to the classical method by a constant factor.

Given an n-word odd modulus M, such that $r^{n-1} \leq M < r^n$, where $r = 2^w$ is the radix of the system, and w is the bit length of a word, and an integer $X = \sum_{i=0}^{n-1} x_i \cdot 2^{w \cdot i}$, then the Montgomery residue of this integer is defined as $\widetilde{X} = X \cdot R \bmod M$. The Montgomery radix R, is a constant such that $\gcd(R, M) = 1$ and $R > M$. For efficiency reasons, this is usually adjusted to $R = r^n$. The Montgomery product of two integers is defined as $M(\widetilde{X}, \widetilde{Y}) = \widetilde{X} \cdot \widetilde{Y} \cdot R^{-1} \bmod M$. If $\widetilde{X} = X \cdot R \bmod M$ and $\widetilde{Y} = Y \cdot R \bmod M$ are Montgomery residues of X and Y, then $\widetilde{Z} = M(\widetilde{X}, \widetilde{Y}) = X \cdot Y \cdot R \bmod M$ is a Montgomery residue of $X \cdot Y \bmod M$. Algorithm 1 describes the radix-r interleaved Montgomery algorithm.

The conversion between the ordinary representation of an integer X to the Montgomery representation $\widetilde{X}$ can be performed using the Montgomery algorithm by computing $\widetilde{X} = M(X, R^2)$, provided that the constant $R^2 \bmod M$ is pre-computed. The conversion back from the Montgomery representation to the ordinary representation can be done by applying the Montgomery algorithm to the result and the number 1, i.e. $Z = M(\widetilde{Z}, 1)$.

In cryptologic applications, where modular products are usually performed succeedingly, the final conditional subtraction, which is costly, is not needed until the end of a series of modular multiplications [13]. In order to avoid the last conditional subtraction (lines 7 to 9 of Algorithm 1), R is chosen such that $4M < R$ and inputs and output are represented as elements of $\mathbb{Z}/2M\mathbb{Z}$ instead of $\mathbb{Z}/M\mathbb{Z}$, that is, operations are carried out in a redundant representation. It can be shown that throughout the series of modular multiplications, outputs from multiplications can be reused as inputs and these values remain bounded. This technique does not only speed-up modular multiplications but also prevents the success of timing attacks [14] as operations are data independent [13].

4 Montgomery Multiplication on the Cell

In this section, we present an implementation of the Montgomery multiplication algorithm that takes advantage of the features of the SPE; e.g. the SIMD architecture, the large register file and the rich instruction set. The multiplication algorithm implemented in the MPM library uses a radix $r = 2^{128}$ to represent large numbers. Each of these 128-bit words is in turn represented using a vector of four consecutive 32-bit words in a SIMD fashion. One drawback of this representation is that operands whose sizes are slightly larger than a power of 2^{128} require an entire extra 128-bit word to be processed, waisting computational resources. Another drawback is that the addition of the resulting 32-bit product to a 32-bit value might produce a carry which is not detected. In contrast to the addition instruction, there is no carry generation instruction for the multiply-and-add operation and is not used in the Montgomery multiplication of the MPM library.

The technique we present uses a radix $r = 2^{16}$ which enables a better division of large numbers into words that match the input sizes of the 4-SIMD multipliers of the Cell. This choice enables the use of the following property, and hence the multiply-and-add instruction: If $a, b, c, d \in \mathbb{Z}$ and $0 \leq a, b, c, d < r$, then $a \cdot b + c + d < r^2$. Specifically, when $r = 2^{16}$, this property enables the addition of a 16-bit word to the result of a 16×16-bit product (used for the multi-precision multiplication and accumulation) and an extra addition of 16-bit word (used for 16-carry propagation) so that the result is smaller than $r^2 = 2^{32}$ and no overflow can occur. We will assume hereafter that the radix $r = 2^{16}$.

Given an odd $16n$-bit modulus M, i.e. $r^{(n-1)} \leq M < r^n$, a Montgomery residue X, such that $0 \leq X < 2M < r^{(n+1)}$, is represented using $s = \lceil \frac{n+1}{4} \rceil$ vectors of 128 bits. The extra 16-bit word is considered in the implementation because the intermediate accumulating result of Montgomery multiplication can be up to $2M$. The Montgomery residue X is represented using a radix r system, i.e. $X = \sum_{i=0}^{n} x_i \cdot r^i$. On the implementation level the 16-bit words x_i are stored column-wise in the s 128-bit vectors X_j, where $j \in [0, s-1]$. The four 32-bit parts of such a vector are denoted by $X_j = \{X_j[0], X_j[1], X_j[2], X_j[3]\}$, where $X_j[0]$ and $X_j[3]$ contain the least and most significant 32-bits of X_j respectively. Each of the $(n+1)$ 16-bit words x_i of X is stored in the most significant 16 bits

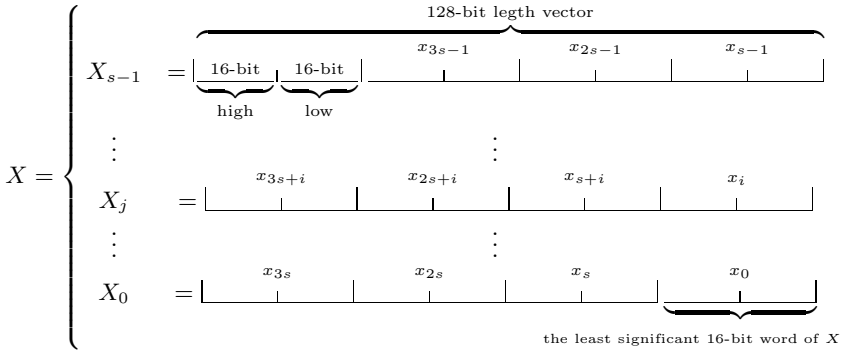

Fig. 1. The 16-bit words x_i of a $16(n+1)$-bit positive integer $X = \sum_{i=0}^{n} x_i \cdot r^i < 2M$ are stored column-wise using $s = \lceil \frac{n+1}{4} \rceil$ 128-bit vectors X_j on the SPE architecture

of $X_{i \bmod s}[\lfloor \frac{i}{s} \rfloor]$. A graphical representation of this arrangement is provided in Figure 1. We follow hereafter the same notation to represent the long integer numbers used.

The Montgomery multiplication algorithm using this representation is described in Algorithm 2. The algorithm takes as inputs the modulus M of n words and the operands X and Y of $(n+1)$ words. The division by r, which is a shift by 16 bits of the accumulating partial product U, is implemented as a logical right shift by 32 bits of the vector that contains the least significant position of U and a change of the indices. That is, during each iteration, the indices of the vectors that contain the accumulating partial product U change circularly among the s registers without physical movement of data. In Algorithm 2, each 16-bit word of the inputs X, Y and M and the output Z is stored in the upper part (the most significant 16 bits) of each of the four 32-bit words in a 128-bit vector. The vector μ stores the replicated values of $(-M)^{-1} \bmod r$ in the lower 16-bit positions of the words. The temporary vector K stores in the most significant 16-bit positions the replicated values of y_i, i.e. each of the parsed coefficients of the multiplier Y corresponding to the i-th iteration of the main loop. The operation $A \leftarrow \mathbf{muladd}(B,\ c,\ D)$, which is a single instruction on the SPE, represents the operation of multiplying the vector B (where data are stored in the higher 16-bit positions of 32 bit words) by a vector with replicated 16-bit values of c across all higher positions of the 32-bit words. This product is added to D and the overall result is placed into A. The temporary vector V stores the replicated values of u_0 in the least significant 16-bit words. This u_0 refers to the least significant 16-bit word of the updated value of U, i.e. $U = \sum_{j=0}^{n} u_j \cdot r^j$ represented by s vectors of 128-bit $U_{i \bmod s}, U_{i+1 \bmod s}, \ldots, U_{i+n \bmod s}$ following the above explained notation (i refers to the index of the main loop). The vector Q is computed as an element-wise logical left shift by 16 bits of the 4-SIMD product of vectors V and μ.

The propagation of the higher 16-bit carries of $U_{(i+j) \bmod s}$ described in lines 9 and 15 consist of extracting the higher 16-bit words of these vectors and placing

Input:
$\begin{cases} M \text{ represented by } s \text{ 128-bit vectors: } M_{s-1}, \ldots, M_0, \text{ such that} \\ \quad r^{n-1} \leq M < r^n, \quad 2 \nmid M, \quad r = 2^{16} \\ X \text{ represented by } s \text{ 128-bit vectors: } X_{s-1}, \ldots, X_0, \\ Y \text{ represented by } s \text{ 128-bit vectors: } Y_{s-1}, \ldots, Y_0, \text{ such that } 0 \leq X, Y < 2M \\ \mu: \text{ a 128-bit vector containing } (-M)^{-1} \bmod r \text{ replicated in all 4 elements.} \end{cases}$

Output: $\begin{cases} Z \text{ represented by } s \text{ 128-bit vectors: } Z_{s-1}, \ldots, Z_0, \text{ such that} \\ \quad Z \equiv X \cdot Y \cdot r^{-(n+1)} \bmod M, \quad 0 \leq Z < 2M \end{cases}$

1: **for** $j = 0$ to $s - 1$ **do**
2: $\quad U_j = 0$
3: **end for**
4: **for** $i = 0$ to n **do**
5: $\quad K = \{y_i,\ y_i,\ y_i,\ y_i\}$
6: $\quad$ **for** $j = 0$ to $s - 1$ **do**
7: $\quad\quad U_{(i+j) \bmod s} = \mathbf{muladd}(X_j,\ K,\ U_{(i+j) \bmod s})$
8: $\quad$ **end for**
9: $\quad$ *Perform 16-carry propagation on* $U_{(i+j) \bmod s}$ *for* $j = 0, \ldots, s - 1$
10: $\quad V = \{u_0,\ u_0,\ u_0,\ u_0\}$
11: $\quad Q =$**shiftleft**(**mul**(V, μ), 16) /* $Q = V \cdot \mu \bmod r$ */
12: $\quad$ **for** $j = 0$ to $s - 1$ **do**
13: $\quad\quad U_{(i+j) \bmod s} = \mathbf{muladd}(M_j,\ Q,\ U_{(i+j) \bmod s})$
14: $\quad$ **end for**
15: $\quad$ *Perform 16-carry propagation on* $U_{(i+j) \bmod s}$ *for* $j = 0, \ldots, s - 1$
16: $\quad U_{i \bmod s} =$**vshiftright**$(U_{i \bmod s}, 32)$ /* *Vector logical right shift by 32 bits*/
17: **end for**
18: *Perform carry propagation on* $U_{i \bmod s}$ *for* $i = n + 1, \ldots, 2n + 1$
19: **for** $j = 0$ to $s - 1$ **do**
20: $\quad Z_j = U_{(n+j+1) \bmod s}$ /* *Place results in higher 16-bit positions*/
21: **end for**

Algorithm 2. Montgomery Multiplication Algorithm for the Cell

Table 1. Performance results, expressed in nanoseconds, for the computation of one Montgomery multiplication for different bit-sizes on a single SPE on a PlayStation 3

Bit-size of moduli	This work (ns)	MPM (ns)	Ratio	Unrolled MPM (ns)	Ratio
192	174	369	2.12	277	1.59
224	200	369	1.85	277	1.39
256	228	369	1.62	277	1.21
384	339	652	1.92	534	1.58
512	495	1,023	2.07	872	1.76
1024	1,798	3,385	1.88	3,040	1.69
2048	7,158	12,317	1.72	11,286	1.58

them into the lower 16-bit positions of temporary vectors. These vectors are then added to $U_{(i+j+1) \bmod s}$ correspondingly. The operation is carried out for the vectors with indices $j \in [0, s-2]$. For $j = s - 1$, the temporary vector that contains

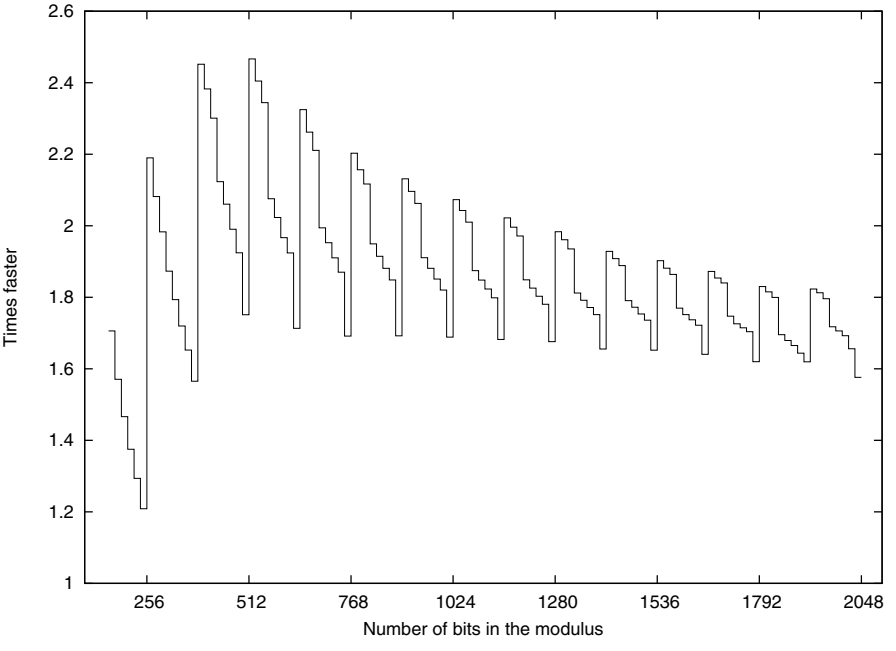

Fig. 2. The speed-up of the new Montgomery multiplication compared to the unrolled Montgomery multiplication from MPM

the words is logically shifted 32 bits to the left and added to the vector $U_{i \bmod s}$. Similarly, the carry propagation of the higher words of $U_{(i+j) \bmod s}$ described in line 18 is performed with 16-bit word extraction and addition, but requires a sequential parsing over the $(n+1)$ 16-bit words. Note that U is represented with vectors whose values are placed in the lower 16-bit word positions.

5 Experimental Results

In this section, performance comparison of different software implementations of the Montgomery multiplication algorithm running on a single SPE of a PS3 using moduli of varying bit lengths is presented. The approach described in Section 3 is compared to the implementation in the Multi-Precision Math (MPM) Library [11]. The MPM library is developed by IBM and part of the software development kit [15] for the Cell. MPM consists of a set of routines that perform arithmetic on unsigned integers of a large number of bits. According to [11]: "All multi-precision numbers are expressed as an array of unsigned integer vectors (vector unsigned int) of user specified length (in quadwords). The numbers are assumed to be big endian ordered".

To enhance the performance and avoid expensive conditional branches, a code generator has been designed. This generator takes as input the bit-size

of the modulus and outputs the Montgomery multiplication code in the C-programming language that uses the SPU-intrinsics language extension. Such a generator has been made for both our approach as well as for the implementation of Montgomery multiplication in the MPM library. We refer to this faster version of MPM as unrolled MPM. The computational times, in nanoseconds, of a single Montgomery multiplication for cryptographically interesting bit sizes, are given in Table 1. Our Montgomery implementation is the subtraction-less variant (suitable for cryptographic applications) while the MPM version is the original Montgomery multiplication including the final subtraction. A subtraction-less variant of the MPM implementation is significantly slower since it requires the processing of an entire 128-bit vector for one extra bit needed to represent the operands in the interval of values $[0, 2M)$. Thus, it is not considered in our comparison.

The performance results include the overhead of benchmarking, function calls, loading and storing the inputs and output back and forth between the register file and the local store. The stated ratios are the (unrolled) MPM results versus the new results. Figure 2 presents the overall speed-up ratios, compared to the unrolled implementation of MPM. Every 128 bits a performance peak can be observed in the figure, for moduli of $128i+16$ bits. This is because the (unrolled) MPM implementations work in radix $r = 2^{128}$ and requires an entire 128-bit vector to be processed for these extra 16 bits. This peak is more considerable for smaller bit-lengths because of the significant relative extra work compared to the amount of extra bits. This effect becomes less noticeable for larger moduli. The drop in the performance ratio of our algorithm, that occurs every multiple of 64 bits in Figure 2, can be explained by the fact that moduli of these bit sizes require an extra vector to hold the number in redundant representation. Despite this fact our implementation outperforms the unrolled MPM (which is faster compared to generic MPM) because it only iterates over the 16-bit words that contain data. The only bit sizes where the unrolled MPM version is faster compared to our approach are in the range $[112, 128]$. This is due to a small constant overhead built in our approach and becomes negligible for larger bit-lengths. The maximal speed-up obtainable from our approach is 2.47 times compared to the unrolled Montgomery multiplication of the MPM library and occurs for moduli of size 528 bit.

6 Conclusions

We have presented a technique to speed up Montgomery multiplication on the SPEs of the Cell processor. The technique consist of representing the integers by grouping consecutive parts. These parts are placed one by one in each of the four element positions of a vector, representing columns in a 4-SIMD organization. In practice, this leads to a performance speed-up of up to 2.47 compared to an unrolled Montgomery multiplication implementation as in the MPM library for moduli of sizes 160 to 2048 bits which are of particular interest for public-key cryptography. Although presented for the Cell architecture, the proposed techniques can also be applied to other SIMD architectures.

References

1. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120–126 (1978)
2. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48, 203–209 (1987)
3. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
4. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
5. Costigan, N., Scott, M.: Accelerating SSL using the vector processors in IBM's Cell broadband engine for Sony's playstation 3. Cryptology ePrint Archive, Report 2007/061 (2007), `http://eprint.iacr.org/`
6. Bos, J.W., Casati, N., Osvik, D.A.: Multi-stream hashing on the PlayStation 3. In: PARA 2008 (2008) (to appear)
7. Bos, J.W., Osvik, D.A., Stefan, D.: Fast implementations of AES on various platforms. Cryptology ePrint Archive, Report 2009/501 (2009), `http://eprint.iacr.org/`
8. Costigan, N., Schwabe, P.: Fast elliptic-curve cryptography on the Cell broadband engine. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 368–385. Springer, Heidelberg (2009)
9. Bos, J.W., Kaihara, M.E., Montgomery, P.L.: Pollard rho on the PlayStation 3. In: SHARCS 2009, pp. 35–50 (2009)
10. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D.A., de Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In: Halevi, S. (ed.) Advances in Cryptology - CRYPTO 2009. LNCS, vol. 5677, pp. 55–69. Springer, Heidelberg (2009)
11. IBM: Multi-precision math library. Example Library API Reference, `https://www.ibm.com/developerworks/power/cell/documents.html`
12. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA 2005. IEEE Computer Society, Los Alamitos (2005)
13. Walter, C.D.: Montgomery exponentiation needs no final subtractions. Electronics Letters 35(21), 1831–1832 (1999)
14. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
15. IBM: Software Development Kit (SDK) 3.1 (2007), `http://www.ibm.com/developerworks/power/cell/documents.html`