High-Performance Modular Multiplication on the Cell Processor

Joppe W. Bos

Laboratory for Cryptologic Algorithms
EPFL, Station 14, CH-1015 Lausanne, Switzerland

Abstract. This paper presents software implementation speed records for modular multiplication arithmetic on the synergistic processing elements of the Cell broadband engine (Cell) architecture. The focus is on moduli which are of special interest in elliptic curve cryptography, that is, moduli of bit-lengths ranging from 192- to 521-bit. Finite field arithmetic using primes which allow particularly fast reduction is compared to Montgomery multiplication. The special primes considered are the five recommended NIST primes, as specified in the FIPS 186-3 standard, and the prime used in the elliptic curve *curve25519*. While presented and benchmarked on the Cell architecture, the proposed techniques to efficiently implement the modular multiplication algorithms are suited to run on any architecture which is able to compute multiple computations concurrently; e.g. graphics processing units.

Keywords: Cell Broadband Engine, Curve25519, Elliptic Curve Cryptography (ECC), Montgomery Multiplication, NIST primes.

1 Introduction

Elliptic curve cryptography (ECC) [20,24] is an approach to public-key cryptography which enjoys increasing popularity since its invention in the mid 1980s. The attractiveness of small key-sizes [22] has placed this public-key cryptosystem as the preferred alternative to the widely used RSA public-key cryptosystem [30]. This is emphasized by the current migration away from 80-bit to 112-bit security where, for instance, the United States' National Security Agency restricts the use of public key cryptography in "Suite B" [27] to ECC.

In this paper we present performance results for one of the key operations in ECC: modular multiplication. The performance results are obtained when running on the heterogeneous, multi-core, single instruction, multiple data (SIMD) Cell broadband engine (Cell) architecture. As far as we know, our performance results set new speed records for generic moduli, using interleaved Montgomery multiplication [25], and *special* modular multiplication for moduli ranging from 192 to 521 bits. This range covers the current standardized parameters for ECC cryptosystems as specified by National Institute of Standards (NIST) [34].

The special primes considered in this work are the recommended primes of special form by NIST [34] and the prime used in *curve25519* as proposed by

M.A. Hasan and T. Helleseth (Eds.): WAIFI 2010, LNCS 6087, pp. 7–24, 2010.

Bernstein [2]. These special primes are used to enhance the performance of ECC-based schemes in practice by exploiting the special form of the primes to construct a fast reduction step. Typically, the multiplication and special reduction are performed sequentially. For the separated multiplication step we consider schoolbook and Karatsuba multiplication [18] techniques. We use the straight-forward methods to implement the fast reduction for the NIST recommended primes (see [32]). For the special prime in *curve25519* we use a different approach in order to compare with the proposed fast reduction from [2].

The performance results are obtained by using the features of SIMD architectures. The implementations are optimized for the Cell and take both the advantages (e.g., the rich instruction set and large register file) and disadvantages (e.g., the "small" $16 \times 16 \rightarrow 32$-bit multiplier) of this architecture into account. Furthermore, multiple streams of computations are interleaved to increase throughput. Multi-stream modular multiplication computations are useful in both a cryptanalytic and cryptographic setting. For instance, one could use multi-stream modular multiplication routines, either the generic or special variant, to speedup batch decryption for ECC-based schemes. Additionally, this work shows the practical benefit of using the special over generic prime moduli on the Cell.

The paper is organized as follows. Section 2 introduces the Cell broadband engine architecture. Section 3 recalls some basic facts about elliptic curves, Montgomery multiplication and discusses the special primes used in this work. Section 4 describes the cryptographic and cryptanalytic applications where multi-stream modular multiplications can be used. Section 5 describes how the different modular multiplication methods can be combined into a multi-stream high-performance implementation on the Cell. Section 6 presents and discusses our performance results and compares them to implementations by others on the Cell. Section 7 concludes the paper.

2 The Cell Broadband Engine

The Cell architecture [15], jointly developed by Sony, Toshiba, and IBM, is equipped with one dual-threaded, 64-bit in-order "Power Processing Element" (PPE), which can offload work to the eight "Synergistic Processing Elements" (SPEs) [33]. The SPEs are the workhorses of the Cell processor which can be found in the PlayStation 3 (PS3) game console. Each SPE, running at 3.2 GHz in the PS3, consists of a Synergistic Processing Unit (SPU), 256 kilobyte of private memory called Local Store (LS) and a Memory Flow Controller.

Most SPU instructions are 128-bit wide single instruction, multiple data (SIMD) operations performing sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit computations in parallel. Each SPU is equipped with a large register file containing 128 registers of 128 bits each, providing space for unrolling and software pipelining of loops, hiding the relatively long latencies of its instructions. Unlike the processor in the PPE, the SPUs are asymmetric processors, having two pipelines (denoted by the odd and the even pipeline) which are designed to

execute two disjoint sets of instructions (denoted by odd and even instructions). In the ideal case, two instructions (one odd and one even) can be dispatched per cycle. The SPUs are in-order processors and have no hardware branch-prediction. Instead, the programmer (or compiler) can tell the instruction fetch unit in advance where a (single) branch instruction will jump to.

Each SPE has access to a rich instruction set which operates simultaneously on 8-, 16- or 32-bit words. Instructions of particular interest are `shuffle` (odd instruction) and `select` (even instruction). The $d = \texttt{shuffle}(a, b, c)$ instruction uses the pattern given in c to shuffle 16 of the 32 bytes of a and b to the output d, in such a way that the jth byte of c determines the jth byte of d, either as a copy of a byte of a or b or as one of the constants {`0x00, 0xFF, 0x80`}, and where duplicate copies are allowed. The $d = \texttt{select}(a, b, p)$ instruction acts as a 2-way multiplexer; depending on the input pattern p the corresponding bit from either a or b is selected as the output bit in d. The SPEs are equipped with a 4-way SIMD multiplier (even instruction) which can compute four 16-bit integer multiplications simultaneously per clock cycle. In addition, an even 4-way SIMD multiply-and-add instruction, which performs a $16 \times 16 \rightarrow 32$-bit unsigned multiplication and an addition of a 32-bit unsigned operand to the 32-bit product, is available and has the same latency as a multiplication without the addition. Note that carries are not generated for this instruction.

3 Preliminaries

In this section the required background about elliptic curves, the various (modular) multiplication techniques and the special primes are recalled. We want to compute the product $C \equiv A \cdot B \bmod M$, by either first applying schoolbook or Karatsuba multiplication and next a fast reduction, or $C \equiv A \cdot B \cdot r^{-n} \bmod M$ using Montgomery multiplication, with $A, B, C, r, n, M \in \mathbb{Z}$. Here, M is an n-word, odd modulus such that $r^{n-1} \leq M < r^n$. In practice $r = 2^w$ with w the bit-length of a word, for the algorithms implemented for the SPE we either use $w = 32$ or $w = 16$ (cf. Section 5).

Elliptic Curves. Let $p > 3$ be a prime, then any $a, b \in \mathbb{F}_p$ such that $4a^3 + 27b^2 \neq 0$ define an elliptic curve $E_{a,b}$ over $\mathbb{F}_p$. The zero point, the so-called point at infinity, together with the set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ which satisfy the shortened affine Weierstrass equation $y^2 = x^3 + ax + b$, form an Abelian group $E_{a,b}(\mathbb{F}_p)$ [31] (usually written additively). Repeated point addition is called scalar multiplication and a single instance of point addition can be computed using multiple operations in $\mathbb{F}_p$. Besides the affine Weierstrass representation one can use a whole range of different representations. An overview of the costs, expressed in arithmetic operations in the underlying field, is given by Bernstein and Lange in [5].

Montgomery Multiplication. The Montgomery modular multiplication method is introduced in [25] and can be used to replace the conventional modular multiplication. In order to be used, the operands need to be converted: given an

Algorithm 1. Schoolbook (left), Karatsuba (middle) and interleaved Montgomery (right) multiplication algorithms.

Input: $\begin{cases} A = \sum_{i=0}^{n-1} a_i r^i, \\ B = \sum_{i=0}^{n-1} b_i r^i \end{cases}$
Output: $\begin{cases} C = A \cdot B \\ \quad = \sum_{i=0}^{2n-1} c_i r^i \end{cases}$
1. $C = A \cdot b_0$
2. **for** $i = 1$ to $n-1$ **do**
3. $\quad C = C + r^i (A \cdot b_i)$
4. **return** C

Input: $\begin{cases} A = \sum_{i=0}^{n-1} a_i r^i, \\ B = \sum_{i=0}^{n-1} b_i r^i, \\ T: \text{ some threshold for} \\ \text{switching to schoolbook} \\ \text{multiplication.} \\ \text{Let } \widetilde{r} = r^{\lceil n/2 \rceil}. \end{cases}$
Output: $\begin{cases} C = A \cdot B \\ \quad = \sum_{i=0}^{2n-1} c_i r^i \end{cases}$
1. **if** $n < T$ **then**
2. $\quad$ **return** $C = \text{schoolbook}(A, B)$
3. $A = A_0 + A_1 \widetilde{r}, \quad 0 \le A_0, A_1 < \widetilde{r}$
4. $B = B_0 + B_1 \widetilde{r}, \quad 0 \le B_0, B_1 < \widetilde{r}$
5. $T_0 = \text{Karatsuba}(A_0,\ B_0)$
6. $T_1 = \text{Karatsuba}(A_1,\ B_1)$
7. $T_2 = \text{Karatsuba}(A_0 + A_1,\ B_0 + B_1) - T_0 - T_1$
8. **return** $C = (T_0 + T_2 \cdot \widetilde{r} + T_1 \cdot \widetilde{r}^2)$

Input: $\begin{cases} A = \sum_{i=0}^{n-1} a_i r^i, B, \\ M, \mu \text{ such that} \\ 0 \le A, B < r^n, \\ r^{n-1} \le M < r^n, \\ 2 \nmid M, \quad \gcd(r, M) = 1 \\ \mu = -M^{-1} \bmod r, \end{cases}$
Output: $\begin{cases} C \equiv A \cdot B \cdot r^{-n} \bmod M \\ \text{ such that } 0 \le C < r^n \end{cases}$
1. $C = 0$
2. **for** $i = 0$ to $n-1$ **do**
3. $\quad C = C + a_i \cdot B$
4. $\quad q = \mu \cdot C \bmod r$
5. $\quad C = (C + q \cdot M)/r$
6. **if** $C \ge r^n$ **then**
7. $\quad C = C - M$
8. **return** C

integer X, the Montgomery residue of this integer is defined as $\widetilde{X} = X \cdot r^n \bmod M$ with $r^{n-1} \le M < r^n$. The constant r^n is the Montgomery radix such that $\gcd(r^n, M) = 1$. The Montgomery product is defined as $\widetilde{X} \cdot \widetilde{Y} \cdot r^{-n} \bmod M$, addition and subtraction remain unchanged. Since converting to and from Montgomery form requires computational effort, the Montgomery multiplication is mostly used in settings where the computation of a sequence of modular operations is required. See Algorithm 1 for a high-level description of the interleaved Montgomery multiplication method.

Fast Reduction. One way to speed up elliptic curve arithmetic is to enhance the performance of the finite field arithmetic by using a prime of a special form. The structure of such a prime is exploited by constructing a fast reduction method, applicable to this prime only. Typically, the multiplication and reduction are in two sequential phases. For the multiplication phase we consider the so-called schoolbook, or textbook, multiplication and the asymptotically faster Karatsuba multiplication techniques (see Algorithm 1 for a high-level description).

NIST Primes. In the FIPS 186-3 standard [34] NIST recommends the use of five prime fields when using the elliptic curve digital signature algorithm. These primes allow fast reduction, see Appendix A for the algorithms optimized for a machine word (limb) size of 32 bits, based on the work by Solinas [32]. The five recommended primes are

$$p_{192} = 2^{192} - 2^{64} - 1, \qquad p_{224} = 2^{224} - 2^{96} + 1,$$
$$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, \qquad p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1,$$
$$p_{521} = 2^{521} - 1.$$

An extensive study of a software implementation of the NIST-recommended elliptic curves over prime fields on the x86 architecture is given by Brown et al. [8].

Curve25519. The elliptic curve *curve25519* is proposed by Bernstein in [2]. Besides offering high-speed arithmetic, a list of other advantages can be found

in the original article [2]. This curve is over $\mathbb{F}_{p_{255}}$ with $p_{255} = 2^{255} - 19$, an element $x \in \mathbb{F}_{p_{255}}$ can be represented as $x = \sum_{i=0}^{9} x_i 2^{\lceil 25.5i \rceil}$. Bernstein proposes to implement the arithmetic using floating point instructions and therefore representation inside a CPU is achieved by using floating-point registers. The original article gives performance data obtained on a Pentium M.

4 Applications

To increase throughput the 4-way SIMD instructions of the SPE are used to implement a modular multiplication routine which operates on 4 streams, or a small multiple of 4 by interleaving these streams, in parallel. When a sequence of multiplications has to be computed, for instance in elliptic curve scalar multiplication, the algorithm performs the same operations in SIMD-mode on all inputs. When the scalar multipliers are different, a square-and-multiply algorithm needs to perform a different sequence of point additions and doublings, since this depends on the binary expansion of the scalar multiplier. Performing the same computations on multiple streams concurrently, when multiplying with different scalars, in a SIMD fashion might be suboptimal since all streams which are being processed in parallel need to perform the same computations. In this section we present some applications in cryptography and cryptanalysis where SIMD modular multiplication algorithms can be beneficial; i.e., where the same multiplier is used in multiple independent instances.

Cryptography. Cryptographic schemes often need to perform exponentiations with a randomly selected exponent, or scalar multiplications when using the additive group law as in the elliptic curve setting. If this exponent is used several times, in independent calculations, these operations can be performed in parallel in a SIMD fashion. For instance, in elliptic curve public-key schemes the ability to process multiple streams of modular multiplication computations can be used to speedup batch decryption. Examples of such schemes are the elliptic curve integrated encryption scheme (ECIES), proposed by Bellare and Rogaway [1] and standardized in [9], and the provably secure encryption curve scheme (PSEC), based on the work by Fujisaki and Okamoto [13] and standardized in [17]. The decryption of a message consist of multiplying an elliptic curve point, as specified by the ciphertext, by the private key d in PSEC or by $h \cdot d$ in the case of ECIES, where h is the cofactor of the group order and is constant for a given private key. When many messages need to be decrypted, using the same private key, SIMD algorithms as described in this article can be used to speedup computations.

In other settings, where the bitsize of the modulus is usually larger compared to the ECC setting, multi-stream modular multiplication computations can be useful as well. ElGamal encryption schemes [12] require two exponentiations with the same random exponent. Other related methods perform more exponentiations with the same exponent. The double base variant of ElGamal by Damgård, often referred to as Damgård ElGamal [11], performs three exponentiations. The "double" hybrid Damgård ElGamal, as proposed by Kiltz et al. [19], requires four exponentiations with the same exponent in every encryption.

Cryptanalysis. In cryptanalysis, multi-stream modular multiplication computations, for moduli sizes as considered in this article, can be used to enhance the performance of the Pollard rho discrete logarithm algorithm [29], a method to solve the elliptic curve discrete logarithm problem (ECDLP) which is essential to the security of ECC. In practice, modular inversions in the Pollard rho algorithm are traded for modular multiplications, to increase speed, by using the Montgomery simultaneous inversion technique [26]. This technique allows one to trade, when running N computations in parallel, N inversions for roughly $3N$ modular multiplications and one inversion. For example, this technique is used in [7] to solve a 112-bit ECDLP on the SPE architecture by working concurrently on 400 computations. Here, 70 percent of the total run-time is spent on the computation of modular multiplications [7].

Another cryptanalytic application is factoring integers. The integer factorization problem is essential to cryptographic algorithms as RSA. The fastest known method to factor integers is the number field sieve [28,21]. This method can use the elliptic curve factorization method (ECM) [23] in a co-factorization phase. Performing elliptic curve arithmetic on multiple points allows the use of multi-stream modular multiplication methods. Related work by Bernstein et al. [4] gives performance details of a high-performance multi-stream implementation of modular arithmetic in the ECM on graphics cards.

5 Multiplication on the SPE Architecture

The (modular) multiplication operations in this work are designed to operate on relatively small ($\leq$ 521 bits) integers. On the widely available x86 and x86-64 architectures the threshold for switching from schoolbook multiplication to methods with a lower asymptotic run-time complexity (e.g. Karatsuba multiplication) is $>$ 800 bits [14]. On these architectures the size of the operands on which the multiplication and addition instructions work is typically the same (either 32 or 64 bits).

On the Cell "only" a $16 \times 16 \rightarrow 32$ bits multiplication instruction is available, performing four multiplications in parallel, while the size of the 4-way SIMD operands to the addition instruction is 32 bits. Unlike the x86 architecture an integer multiply-and-add instruction is available. This allows the addition of two extra 16-bit values to a result of a 16-bit multiplication without generating a carry, since if $0 \leq a, b, c, d < 2^{16}$, then $a \cdot b + c + d < 2^{32}$. We consider both the schoolbook and Karatsuba multiplication for the special modular multiplication routines.

Integer Representation on the Cell. For a high-performance implementation of arithmetic algorithms on the Cell, vectorization techniques are applied and data are represented using the 4-way SIMD organization of the SPEs. Using m 128-bit registers $x[0], x[1], \ldots, x[m-1]$ a four-tuple (x_1, x_2, x_3, x_4) of integers is represented. Each x_i is a wm-bit integer, where w is either 16 or 32 depending on the setting; typically we use $w = 16$ for multiplication and $w = 32$ for

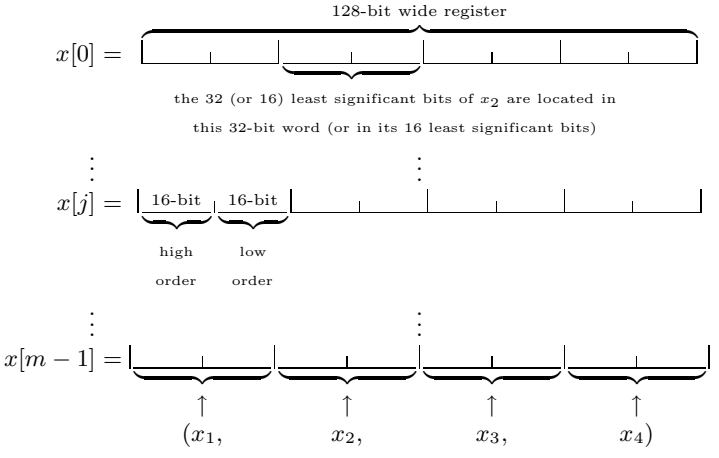

Fig. 1. A four-tuple (x_1, x_2, x_3, x_4) of $32m$-bit (or $16m$-bit) integers arranged in m 128-bit registers

addition and subtraction to match the bit-lengths of the corresponding 4-way SIMD instructions. Every element of the four-tuple is represented in a radix-2^w system:

$$x_i = \sum_{j=0}^{m-1} x[j]_i 2^{wj},$$

for $i = 1, 2, 3, 4$. The four 32-bit words of the 128-bit register $x[j]$ are denoted by $x[j]_i$. The representation of such a four-tuple (x_1, x_2, x_3, x_4) is depicted in Figure 1.

Multiplication. Algorithm 2 depicts schoolbook multiplication designed to run on SIMD architectures and is optimized for architectures with a native multiply-and-add instruction. After trivially unrolling the for-loops the algorithm is branch-free. Algorithm 2 splits the operands in 16-bit words, to take advantage of the 16-bit multiplier on the Cell, but this can be modified to work with any other word size on different architectures. Hence, on the SPE, Algorithm 2 operates on four-tuples of inputs simultaneously using the data representation from Fig. 1.

After the multiply-and-add, and a possible extra addition of one 16-bit word, the 32-bit result z is split into the 16 most and 16 least significant bits, x and y respectively. This is denoted by $\texttt{split}(z) = (\lfloor \frac{z}{2^{16}} \rfloor, z \bmod 2^{16})$. On the SPE this splitting can be implemented in different ways, i.e. by using two odd `shuffle` instructions, or one even `and` and one odd `shuffle` instruction, or two even `and` instructions. The appropriate `split`ting implementation is chosen to balance the number of odd and even instructions, reducing the total number of required cycles. Note that when $i = 1$ the extra addition of d_{i+1} can be omitted. Hence, Algorithm 2 requires $n^2 \times \texttt{split}$, $n^2 \times \texttt{muladd}$ and $n(n-2) \times \texttt{add}$ (when multiplying two $16n$-bit integers); this can be computed in $2n(n - \frac{3}{4})$ cycles,

Algorithm 2. Radix-2^{16} schoolbook multiplication algorithm.

Input: $\begin{cases} \text{Integer } a = (a_{n-1}, \ldots, a_1, a_0), & \text{each } a_i \text{ is a 16-bit word.} \\ \text{Integer } b = (b_{n-1}, \ldots, b_1, b_0), & \text{each } b_i \text{ is a 16-bit word.} \end{cases}$
Output: Integer $c = (c_{2n-1}, \ldots, c_1, c_0) = a \cdot b$, each c_i is a 16-bit word.
1. $d_i = 0, \quad i \in [1, n]$
2. **for** $j = 0$ to $n - 1$ **do**
3. $\quad (e_0, D_j) = \texttt{split}(a_0 \cdot b_j + d_1)$
4. $\quad$ **for** $i = 1$ to $n - 1$ **do**
5. $\quad\quad (e_i, d_i) = \texttt{split}(a_i \cdot b_j + e_{i-1} + d_{i+1})$
6. $\quad d_n = e_{n-1}$
7. **return** $(c = (d_n, d_{n-1}, \ldots, d_1, D_{n-1}, D_{n-2}, \ldots, D_0))$

optimistically assuming all odd and even pairs can be dispatched simultaneously. Furthermore, this approximation ignores the function-call overhead and loading and storing the in- and output from the local store. This leads to an optimistic approximation for the computation of a single $16n \times 16n \rightarrow 32n$-bit schoolbook multiplication in $\frac{n}{2}\left(n - \frac{3}{4}\right)$ cycles (when processing 4 streams in parallel).

A branch-free (when unrolled) Karatsuba multiplication algorithm optimized for vector architectures is given in Algorithm 3. This algorithm works on 32-bit words, which is the word size of the even 4-way SIMD addition and subtraction instructions on the SPE. Just as with the schoolbook multiplication this word size can trivially be modified. Algorithm 3 assumes that the bitsize of the input values is a multiple of 64 to split the operands evenly in two 32-bit multiples. These parts are multiplied using another multiplication routine `mul`, which is either a schoolbook or Karatsuba multiplication, which operates on inputs of half the size.

The $2m$-bit multiplication is split into two $m \times m$-bit and one $(m+1) \times (m+1)$-bit multiplications (see Alg. 1). In order to avoid the use of a probably more expensive multiplication by an extra limb, three $m \times m$-bit multiplications are used. The correct result, for the $(m+1) \times (m+1)$-bit multiplication, is computed by creating select-masks from the most significant bit of each of the two operands. These are used to select the appropriate value (one of the inputs) or zero, which is added to the result of the $m \times m$-bit multiplication. Note that the initial borrow values, in line 21, are (counterintuitively) set to one. An extra subtraction of one is performed when the borrow is zero and no subtraction is performed when the borrow is one on the SPE.

Special Reduction. The special reduction algorithms, see Appendix A, do not fully reduce the input to the range $[0, p\rangle$ but to $[0, t \cdot p\rangle$, where p is the prime modulus used and t a small positive integer. In order to reduce a four-tuple of integers simultaneously using SIMD instructions, different approaches can be applied. Obviously the reduction algorithm can be applied again. A most likely faster approach, when t is sufficiently small, is to subtract p repeatedly until the result is in the desired range. The repeated subtracting is done by masking the value appropriately before subtracting, which needs to be performed up to $t - 1$ times since multiple integer values are processed in parallel.

Algorithm 3. Radix-2^{32} Karatsuba multiplication algorithm for architectures which support vector instructions, n is even.

Input: $\begin{cases} \text{Integer } X = (x_{n-1}, \ldots, x_0), \text{ each } x_i \text{ is a 32-bit word.} \\ \text{Integer } Y = (y_{n-1}, \ldots, y_0), \text{ each } y_i \text{ is a 32-bit word.} \end{cases}$

Output: Integer $Z = (z_{2n-1}, \ldots, z_0) = X \cdot Y$, each z_i is a 32-bit word.

1. $(B_{n-1}, \ldots, B_0) = \text{mul}((x_{n-1}, \ldots, x_{n/2}),\ (y_{n-1}, y_{n/2}))$
2. $(C_{n-1}, \ldots, C_0) = \text{mul}((x_{n/2-1}, \ldots, x_0),\ (y_{n/2-1}, \ldots, y_0))$
3. $zero = \text{carry}_1 = \text{carry}_2 = \{0\}$
4. **for** $i = 0$ to $n/2 - 1$ **do**
5. $\quad X_i = \text{add\_extended}(x_{n/2+i},\ x_i,\ \text{carry}_1)$
6. $\quad Y_i = \text{add\_extended}(y_{n/2+i},\ y_i,\ \text{carry}_2)$
7. $\quad \text{carry}_1 = \text{gen\_carry\_extended}(x_{n/2+i},\ x_i,\ \text{carry}_1)$
8. $\quad \text{carry}_2 = \text{gen\_carry\_extended}(y_{n/2+i},\ y_i,\ \text{carry}_2)$
9. $\text{mask}_1 = \text{cmpgt}(\text{carry}_1, 0),\ \text{mask}_2 = \text{cmpgt}(\text{carry}_2, 0)$
10. **for** $i = 0$ to $n/2 - 1$ **do**
11. $\quad s_i = \text{select}(\text{zero}, Y_i, \text{mask}_1),\ t_i = \text{select}(\text{zero}, X_i, \text{mask}_2)$
12. $c_1 = \text{select}(\text{zero}, \text{carry}_1, \text{mask}_2)$
13. $(z_{n-1}, \ldots, z_{n/2}, A_{n/2-1}, \ldots, A_0) = \text{mul}((X_{n/2-1}, \ldots, X_0),\ (Y_{n/2-1}, \ldots, Y_0))$
14. $\text{carry}_1 = \text{carry}_2\{0\}$
15. **for** $i = n/2$ to $n - 1$ **do**
16. $\quad T = \text{add\_extended}(z_i,\ s_{i-n/2},\ \text{carry}_1)$
17. $\quad A_i = \text{add\_extended}(T,\ t_{i-n/2},\ \text{carry}_2)$
18. $\quad \text{carry}_1 = \text{gen\_carry\_extended}(z_i,\ s_{i-n/2},\ \text{carry}_1)$
19. $\quad \text{carry}_2 = \text{gen\_carry\_extended}(T,\ t_{i-n/2},\ \text{carry}_2)$
20. $A_n = \text{add\_extended}(\text{carry}_1,\ \text{carry}_2,\ c_1)$
21. $\text{borrow}_1 = \text{borrow}_2 = \{1\}$
22. **for** $i = 0$ to $n - 1$ **do**
23. $\quad T = \text{sub\_extended}(A_i,\ B_i,\ \text{borrow}_1)$
24. $\quad E_i = \text{sub\_extended}(T,\ C_i,\ \text{borrow}_2)$
25. $\quad \text{borrow}_1 = \text{gen\_borrow\_extended}(A_i,\ B_i,\ \text{borrow}_1)$
26. $\quad \text{borrow}_2 = \text{gen\_borrow\_extended}(T,\ C_i,\ \text{borrow}_2)$
27. $E_n = \text{sub}(A_n,\ zero,\ \text{borrow}_1),\ E_n = \text{sub}(A_n,\ zero,\ \text{borrow}_2)$
28. $\text{carry}_1 = 0$
29. **for** $i = n/2$ to $n - 1$ **do**
30. $\quad Z_i = \text{add\_extended}(C_i,\ E_{i-n/2},\ \text{carry}_1)$
31. $\quad \text{carry}_1 = \text{gen\_carry\_extended}(C_i,\ E_{i-n/2},\ \text{carry}_1)$
32. **for** $i = n$ to $n + n/2 - 1$ **do**
33. $\quad Z_i = \text{add\_extended}(B_{i-n},\ E_{i-n/2},\ \text{carry}_1)$
34. $\quad \text{carry}_1 = \text{gen\_carry\_extended}(B_{i-n},\ E_{i-n/2},\ \text{carry}_1)$
35. $Z_{n+n/2} = \text{add\_extended}(B_{n/2},\ E_n,\ \text{carry}_1)$
36. $\text{carry}_1 = \text{gen\_carry\_extended}(B_{n/2},\ E_n,\ \text{carry}_1)$
37. **for** $i = n + n/2 + 1$ to $2n - 1$ **do**
38. $\quad Z_i = \text{add}(B_{i-n},\ \text{carry}_1)$
39. $\quad \text{carry}_1 = \text{gen\_carry}(B_{i-n},\ \text{carry}_1)$
40. **return** $Z = (Z_{2n-1}, \ldots, Z_{n/2}, C_{n/2-1}, \ldots, C_0)$

Table 1. The values of the 32-bit unsigned limbs of $t \cdot p_{224}$, c_7 and c_0 are the most and least significant limb respectively. In order to avoid using a look-up table the value $t \cdot p_{224}$ can be computed efficiently. Given t, $c_0 = t$, $c_1 = c_2 = 0$, $c_3 = 0 - t$, the values for c_4, c_5, c_6, c_7 can be constructed (using the `select` instruction) depending on t.

t	$t \cdot p_{224} = \{c_7, \ldots, c_0\}$								
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	0	0	0	0	0	0
1	0	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	0	0	1
2	1	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-2$	0	0	2
3	2	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-3$	0	0	3
4	3	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-4$	0	0	4

An additional performance gain is possible when the modulus is constant. Select the desired multiple of p, which needs to be subtracted, from a look-up table and perform a single subtraction. This can be achieved, when operating on multiple integer values in parallel, using the `select` instruction. If reduction to $[0, 2^m\rangle$, for an m-bit modulus p, is allowed, the most significant word, containing the possible carry, has to be inspected only to determine the multiple of p to subtract. Note that an extra single subtraction might be needed in the unlikely situation that the result after the subtraction is $> 2^m$. This rare case is implemented by a branch which is hinted to be false to reduce the branch-overhead. The partially reduced numbers can be used as input to the same modular multiplication routines and if reduction to $[0, p\rangle$ is required this can be achieved at the cost of a single multi-limb comparison and subtraction.

For the moduli of special form more instructions can be saved. For example consider the modulus $p_{224} = 2^{224} - 2^{96} + 1$. As described in Algorithm 6, in Appendix A, the algorithm returns with $(s_1 + s_2 + s_3 - s_4 - s_5)$, where all the s_i are 224-bit integers. At the implementation level we work with unsigned integers and prefer not to work with negative numbers. This is achieved by subtracting $s_4 + s_5$ from $2p_{224}$. We can bound the return value d by $d = s_1 + s_2 + s_3 + (2p_{224} - s_4 - s_5) < 5p_{224}$. To reduce d to $[0, 2^{224}\rangle$ the value $t \cdot p_{224}$, for some $t \in [0, 5\rangle$, must be subtracted for four possibly different values of t in parallel after inspection of the most significant word. As can be seen from the representation in Table 1, when using a 2^{32} radix system, selecting the correct value for the different limbs is computationally easy. This allows the computation of $t \cdot p_{224}$ on-the-fly without the need to use and load from a look-up table. The reductions for the other special NIST primes can be done in a similar fashion.

We propose a different approach for calculating the reduction step for the special prime $p_{255} = 2^{255} - 19$ compared to the floating point approach from [2] (see Section 3). This approach is similar to the special reduction technique applied to the 112-bit prime modulus in [6, Appendix A]. A redundant representation modulo $\widetilde{P}_{255} = 2 \cdot p_{255} = 2^{256} - 38$ is used. Let $R = r_h \cdot 2^{256} + r_l$ be the 512-bit result after multiplication. Next, the first reduction step is performed by computing $S = r_l + 38 \cdot r_h \equiv R \bmod \widetilde{P}_{255}$; note that $S < 2^{262}$. Next, the same computation

Algorithm 4. Radix-2^{16} Montgomery Multiplication Algorithm.

Input:
- Integer $a = (a_{n-1}, \ldots, a_1, a_0)$, each a_i is a 16-bit word.
- Integer $b = (b_{n-1}, \ldots, b_1, b_0)$, each b_i is a 16-bit word.
- Integer $M = (M_{n-1}, \ldots, M_1, M_0)$, each M_i is a 16-bit word and M is odd.
- An 16-bit integer $\widetilde{m} = -M^{-1} \bmod 2^{16}$.

Output: Integer $c = (c_{n-1}, \ldots, c_1, c_0) \equiv a \cdot b \cdot 2^{-16n} \bmod M$.

1. $d_i = 0, \quad i \in [0, n]$
2. **for** $i = 0$ to $n - 1$ **do**
3. $\quad (e_0, d_0) = \texttt{split}(a_0 \cdot b_i + d_0)$
4. $\quad$ **for** $j = 1$ to $n - 1$ **do**
5. $\quad\quad (e_j, d_j) = \texttt{split}(a_j \cdot b_i + d_j + e_{j-1})$
6. $\quad d_n = d_n + e_{n-1}$
7. $\quad (*, q) = \texttt{split}(d_0 \cdot \widetilde{m})$
8. $\quad (e_0, d_0) = \texttt{split}(M_0 \cdot q + d_0)$
9. $\quad$ **for** $j = 1$ to $n - 1$ **do**
10. $\quad\quad (e_j, d_{j-1}) = \texttt{split}(M_j \cdot q + d_j + e_{j-1})$
11. $\quad (d_n, d_{n-1}) = \texttt{split}(d_n + e_{n-1})$
12. **if** $d_n > 0$ **then**
13. $\quad (d_{n-1}, \ldots, d_1, d_0) = (d_n, d_{n-1}, \ldots, d_1, d_0) - (M_{n-1}, \ldots, M_1, M_0)$
14. **return** $(c = (d_{n-1}, \ldots, d_1, d_0))$

is repeated on $S = s_h \cdot 2^{256} + s_l$: $T = s_l + 38 \cdot s_h \equiv S \equiv R \bmod \widetilde{P}_{255}$. This is computationally faster since $s_h < 2^6$, note that the resulting $T < 2^{257}$. Similar techniques as described for the NIST primes are used to reduce the result to $[0, 2^{256}\rangle$.

Montgomery Multiplication. The interleaved Montgomery multiplication, optimized for the use on vector architectures, is given in Algorithm 4. As presented, it uses 16-bit limbs and on the Cell four-tuples of inputs are processed concurrently (but Alg. 4 can trivially be modified to operate on any radix size). A conditional subtraction step is needed at the end of the algorithm to ensure that the result is $< 2^{16n}$, for $16n$-bit inputs. This conditional subtraction is replaced by a comparison which creates a select mask, using this mask the value zero or the value of the modulus is selected and subtracted. This eliminates a branch which is to be avoided when processing multiple integer values in a SIMD fashion. For efficiency, the integer representation is switched to a 2^{32} radix system when doing the final masking and subtraction.

The same notation for the split function is used as in Section 5. Hence, Algorithm 4 requires $2n(n+1) \times$ `split`, $2n(n+1) \times$ `muladd` (when counting the multiplication in line 8 as an multiply-and-add) and $2n(n-1) \times$ `add` since the addition of d_j in line 5 when $j = 1$ can be omitted. For the conditional subtraction we first convert the integer representation to a 2^{32} radix system using $\lceil \frac{n}{2} \rceil$ `shuffle` instructions. Next we compare the carry (one `cmpgt` instruction) and mask the value which we are going to subtract using $\lceil \frac{n}{2} \rceil$ `and` instructions. The subtraction requires $\lceil \frac{n}{2} \rceil$ (extended) subtraction instructions and $\lceil \frac{n}{2} \rceil - 1$ (extended) generate borrow instructions.

Counting the number of instructions required in Algorithm 4 gives $4n^2 + 3\lceil \frac{n}{2} \rceil$ even and $\lceil \frac{n}{2} \rceil$ odd instructions plus $2n(n+1)$ times the split function. Hence, an optimistic estimate of the number of cycles, ignoring overhead and assuming perfect scheduling, for a single computation of Montgomery multiplication on $16n$-bit inputs, when computing four computations in parallel, using Algorithm 4 on a single SPE is $n^2 + \frac{9n}{16}$ cycles.

6 Results

We implemented the proposed generic and special modular multiplication algorithms using the C-programming language for the SPEs on the Cell architecture. Four, or a small multiple of four, computations are processed in parallel. The performance benchmarks are performed on a single SPE in the PlayStation 3 game console. We summarize these results, together with other (single and multi-stream computation) modular multiplication results,7 obtained from the literature, in Table 2. The metric of our performance results is the number of cycles for a single modular multiplication computation. Our performance results are obtained by averaging over long sequences, hundreds of millions, of different modular multiplications and include the timing benchmark overhead, the function call overhead, loading and storing the in- and output from the local store and possibly converting the in- and output from the different integer representations (from radix-2^{32} to radix-2^{16} and vice-versa).

Performance Comparison. Performance results obtained with the Multi-Precision Math (MPM) Library [16], provided by IBM in the example API for the Cell, are given in Table 2 for different bit-sizes. The MPM library implements a single-stream Montgomery multiplication computation. In order to obtain a faster implementation for specific bit-lengths (to make a fair comparison) we unrolled the various loops inside the MPM library. These unrolled versions are significantly faster compared to the standard MPM implementation; e.g., the unrolled 256-bit Montgomery multiplication is 1.4 times faster compared to the unmodified MPM implementation. Our multi-stream implementations have a higher latency compared to the unrolled MPM library but process multiple streams resulting in fewer cycles per single multiplication. For instance, in the setting of 256-bit moduli the unrolled MPM requires 877 cycles for a single multiplication while our implementation requires 1188 cycles to compute four multiplications in parallel. This is a speedup of almost a factor of three per single multiplication.

In [10] Costigan and Schwabe implement elliptic curve arithmetic aimed at *curve25519* on the SPE architecture. The representation used differs slightly, but is based on, the one proposed in [2]; an element $x \in \mathbb{F}_{p_{255}}$ is represented as $x = \sum_{i=0}^{19} x_i 2^{\lceil 12.75i \rceil}$. A multi-stream version working on four streams in parallel is implemented and hand-optimized in assembly and "perfectly" scheduled with the surrounding code in a larger function implementing elliptic curve arithmetic. This multi-stream implementation is estimated to compute a single modular multiplication in around 168 cycles [10], this does not include any overhead for

Table 2. Performance results of Montgomery multiplication or modular multiplication modulo the special prime p_i. The latter uses a separate multiplication (schoolbook (S) or Karatsuba (K)) and a fast reduction phase. The benchmarks are performed on a single SPE on a Cell in the PlayStation 3 game console. The stated number of cycles are for a single modular multiplication (when processing the reported number of streams in parallel) and the optimistic estimates are from the formulas from Section 5 and do not include the special reduction cost.

From	Bitsize of the modulus	Method	Streams	Performance (cycles)	Estimate (cycles)
This article	192	p_{192} (K)	8	105	
This article	192	p_{192} (S)	8	126	68
This article	192	Montgomery	8	176	151
Bernstein et al. [3]	195	Montgomery	6	189	
This article	224	p_{224} (K)	8	139	
This article	224	p_{224} (S)	8	143	93
This article	224	Montgomery	4	234	204
Costigan and Schwabe [10]	255	p_{255} (S)	4	168[1]	
This article	255	p_{255} (K)	8	175	
This article	255	p_{255} (S)	8	182	122
This article	256	p_{256} (S)	8	192	122
This article	256	p_{256} (K)	4	193	
This article	256	Montgomery	4	297	265
MPM unrolled [16]	256	Montgomery	1	877	
MPM [16]	256	Montgomery	1	1188	
This article	384	p_{384} (K)	4	389	
This article	384	p_{384} (S)	4	391	279
This article	384	Montgomery	4	665	590
MPM unrolled [16]	384	Montgomery	1	1610	
MPM [16]	384	Montgomery	1	2092	
This article	521	p_{521} (S)	4	622	500
This article	521	p_{521} (K)	4	723	
This article	512	Montgomery	4	1393	1042
MPM unrolled [16]	512	Montgomery	1	2700	
MPM [16]	512	Montgomery	1	3275	

saving and storing the in- and output registers to and from the local store, function call overhead and overhead due to benchmarking. In comparison, our implementation requires 175 cycles for a single modular multiplication using a

[1] This is the required number of cycles for an in-register implementation, no loading of input and storing of output from the local store is performed, and excludes benchmark and function call overhead.

different approach for the special reduction (see Section 5). This includes loading and storing the in- and output, function call and benchmarking overhead and additional latencies because not all code can be scheduled perfectly (especially at the beginning and end of the function where stalls occur). Comparing the performance of the two different approaches for the reduction step is difficult since the reported performance results of two versions are in different settings; ours is a stand-alone multiplication function while the implementation from [10] is an inline version working on registers only. In [10] it is estimated that the time to load and store the in- and output requires 56 cycles in the setting of a single modular multiplication. When considering this cost our approach using the redundant representation looks preferable (since $175 < 168 + 56$), especially since we did not use any fine-tuned assembly code to achieve these results.

Improved multi-stream modular multiplication computations results, compared to [4], are given by Bernstein et al. in [3]. Here, not only results for GPUs are reported but also for the Cell architecture as used in the PlayStation 3. In this setting Montgomery multiplication is implemented and optimized for one bit size: a 195-bit generic modulus. A radix-2^{13} system is used to represent 195-bit integers using 15 limbs, this has the advantage of accumulating multiple carries before an overflow occurs (on the SPE architecture) compared to a radix-2^{16} system but requires more limbs to represent the integers. When quadratically scaling our 192-bit performance result, in a similar fashion as done in [3], this leads to an estimate of $176 \cdot (\frac{195}{192})^2 = 182$ cycles; this is slightly faster compared to the 189 required cycles reported in [3].

Discussion. The performance data from Table 2 show that the modular multiplication using the special primes are in almost all cases, with the exception of p_{256} and p_{521}, roughly 1.7 times faster compared to the Montgomery multiplication implementations targeting the same bit-lengths. Our results show that p_{256} is 1.55 times faster than 256-bit Montgomery multiplication while p_{521} is 2.2 times faster compared to 512-bit Montgomery multiplication. This can be partially explained by the relatively complicated and easy structure of p_{192} and p_{521} respectively.

For p_{192} the version using Karatsuba multiplication is significantly (20 percent) faster compared to the version using schoolbook multiplication. For p_{224}, p_{255}, p_{256} and p_{384} the performance is roughly the same while for p_{521} schoolbook multiplication is 16 percent faster. These differences can be explained due to extra load and store operations from and to the local store. For the smaller bitsizes almost all operations can be performed, after the initial loading from the inputs, on registers. For the larger values the available 128 registers are not sufficient and extra load and store instructions, leading to more instructions and possibly extra stalls, are required. This also explains why processing four streams instead of eight gives a higher performance for p_{384} and p_{521}.

The number of cycles required for the Montgomery multiplication is 12 to 17 percent higher compared to the estimations for all special primes except p_{521}. This overhead is mainly caused by extra load and stores and due to the fact

that the estimates are too optimistic (not every cycle a pair of instructions can be dispatched due to instruction dependencies). For the special prime p_{521} more than 33 percent of the estimated number of cycles is needed. After compiling our code to assembly inspection shows that the significant overhead is as expected due to the extra loads and stores. Note that loading the two input values in registers (after conversion to radix-2^{16}) requires 66 registers which is already more than half of the available register space.

7 Conclusions

In this paper we presented techniques to efficiently implement modular multiplication algorithms to SIMD architectures (such as the Cell or GPUs). We considered Montgomery multiplication and various special reduction routines which are of interest for elliptic curve cryptography. The modular multiplication implementations, which use these faster reduction schemes, are at least 1.5 times faster compared to general purpose Montgomery multiplication for the same bitsize. The performance results of our multi-stream modular multiplication implementations for the synergistic processing elements of the Cell broadband engine architecture set new performance records for moduli of bit-length in the range $[192, 521]$ on this platform. These high-performing modular multiplication, generic or special, implementations can be used to speed up public-key cryptography; e.g. in batch elliptic curve decryption.

References

1. Bellare, M., Rogaway, P.: Minimizing the use of random oracles in authenticated encryption schemes. In: Han, Y., Quing, S. (eds.) ICICS 1997. LNCS, vol. 1334, pp. 1–16. Springer, Heidelberg (1997)
2. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006)
3. Bernstein, D.-J., Chen, H.-C., Chen, M.-S., Cheng, C.-M., Hsiao, C.-H., Lange, T., Lin, Z.-C., Yang, B.-Y.: The Billion-Mulmod-Per-Second PC. In: SHARCS 2009, pp. 131–144 (2009)
4. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2010)
5. Bernstein, D.J., Lange, T.: Analysis and optimization of elliptic-curve single-scalar multiplication. In: Finite Fields and Applications. Contemporary Mathematics Series, vol. 461, pp. 1–19 (2008)
6. Bos, J.W., Kaihara, M.E., Kleinjung, T., Lenstra, A.K., Montgomery, P.L.: On the Security of 1024-bit RSA and 160-bit Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2009/389 (2009), `http://eprint.iacr.org/`
7. Bos, J.W., Kaihara, M.E., Montgomery, P.L.: Pollard rho on the PlayStation. In: SHARCS 2009, vol. 3, pp. 35–50 (2009)

8. Brown, M., Hankerson, D., López, J., Menezes, A.: Software implementation of the NIST elliptic curves over prime fields. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 250–265. Springer, Heidelberg (2001)
9. Certicom Research: Standards for Efficient Cryptography 1: Elliptic Curve Cryptography. Standard SEC1, Certicom (2000)
10. 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)
11. Damgård, I.: Towards practical public key systems secure against chosen ciphertext attacks. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 445–456. Springer, Heidelberg (1992)
12. El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1985)
13. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (1999)
14. Granlund, T.: GMP small operands optimization. In: SPEED 2007 (2007)
15. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA 2005, pp. 258–262 (2005)
16. IBM: Multi-precision math library, Example Library API Reference, `https://www.ibm.com/developerworks/power/cell/documents.html`
17. ISO/IEC 18033-2: Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers (2006)
18. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. In: Proceedings of the USSR Academy of Science, vol. 145, pp. 293–294 (1962)
19. Kiltz, E., Pietrzak, K., Stam, M., Yung, M.: A new randomness extraction paradigm for hybrid encryption. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 590–609. Springer, Heidelberg (2010)
20. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48, 203–209 (1987)
21. Lenstra, A.K., Lenstra Jr., H.W.: The Development of the Number Field Sieve. Lecture Notes in Mathematics, vol. 1554. Springer, Heidelberg (1993)
22. Lenstra, A.K., Verheul, E.R.: Selecting cryptographic key sizes. Journal of Cryptology 14(4), 255–293 (2001)
23. Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics 126, 649–673 (1987)
24. 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)
25. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
26. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48, 243–264 (1987)
27. National Security Agency: Fact sheet NSA Suite B Cryptography (2009), `http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml`
28. Pollard, J.M.: Factoring with cubic integers. In: [21], pp. 4–10
29. Pollard, J.M.: Monte Carlo methods for index computation (mod p). Mathematics of Computation 32, 918–924 (1978)
30. 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)

31. Silverman, J.H.: The Arithmetic of Elliptic Curves. In: Gradute Texts in Mathematics. Springer, Heidelberg (1986)
32. Solinas, J.A.: Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo (1999)
33. Takahashi, O., Cook, R., Cottier, S., Dhong, S.H., Flachs, B., Hirairi, K., Kawasumi, A., Murakami, H., Noro, H., Oh, H., Onish, S., Pille, J., Silberman, J.: The circuit design of the synergistic processor element of a Cell processor. In: ICCAD 2005, pp. 111–117. IEEE Computer Society, Los Alamitos (2005)
34. U.S. Department of Commerce and National Institute of Standards and Technology: Digital Signature Standard (DSS) (2009), `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`

A NIST Reduction

Algorithm 5. Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$.

Input: Integer $c = (c_{11}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{192}^2$.
Output: Integer $d \equiv c \bmod p_{192}$.
Define 192-bit integers:
$s_1 = (c_5, c_4, c_3, c_2, c_1, c_0)$, $s_2 = (0, 0, c_7, c_6, c_7, c_6)$,
$s_3 = (c_9, c_8, c_9, c_8, 0, 0)$, $s_4 = (c_{11}, c_{10}, c_{11}, c_{10}, c_{11}, c_{10})$;
return $(\mathrm{d} = s_1 + s_2 + s_3 + s_4)$;

Algorithm 6. Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$.

Input: Integer $c = (c_{13}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{224}^2$.
Output: Integer $d \equiv c \bmod p_{224}$.
Define 224-bit integers:
$s_1 = (\ c_6,\ c_5,\ c_4,\ c_3,\ c_2,\ c_1,\ c_0)$, $s_2 = (\ c_{10},\ c_9,\ c_8,\ c_7,\ 0,\ 0,\ 0)$,
$s_3 = (\ 0,\ c_{13}, c_{12}, c_{11},\ 0,\ 0,\ 0)$, $s_4 = (\ c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7)$
$s_5 = (\ 0,\ 0,\ 0,\ 0, c_{13}, c_{12}, c_{11})$;
return $(\mathrm{d} = s_1 + s_2 + s_3 - s_4 - s_5)$;

Algorithm 7. Fast reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Input: Integer $c = (c_{15}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{256}^2$.
Output: Integer $d \equiv c \bmod p_{256}$.
Define 256-bit integers:
$s_1 = (\ c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$, $s_2 = (\ c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,
$s_3 = (\ 0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$, $s_4 = (\ c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$,
$s_5 = (\ c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$, $s_6 = (\ c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$,
$s_7 = (\ c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$, $s_8 = (\ c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$,
$s_9 = (\ c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$;
return $(d = s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9)$;

Algorithm 8. Fast reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$.

Input: Integer $c = (c_{23}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{384}^2$.
Output: Integer $d \equiv c \bmod p_{384}$.
Define 384-bit integers:
$s_1 = (\ c_{11}, c_{10}, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
$s_2 = (\ 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, 0, 0)$,
$s_3 = (\ c_{23}, c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12})$,
$s_4 = (\ c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23}, c_{22}, c_{21})$,
$s_5 = (\ c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{20}, 0, c_{23}, 0)$,
$s_6 = (\ 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0, 0, 0, 0)$,
$s_7 = (\ 0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, c_{20})$,
$s_8 = (\ c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23})$,
$s_9 = (\ 0, 0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0)$,
$s_{10} = (\ 0, 0, 0, 0, 0, 0, 0, c_{23}, c_{23}, 0, 0, 0)$;
return $(d = s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10})$;

Algorithm 9. Fast reduction modulo $p_{521} = 2^{521} - 1$.

Input: Integer $c = (c_{33}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{521}^2$.
Output: Integer $d \equiv c \bmod p_{521}$.
Define 521-bit integers:
$s_1 = (c_{16}, \ldots, c_1, c_0)$, $\quad s_2 = (c_{32}, \ldots, c_{17}, c_{16})$;
return $(d = s_1 \bmod 2^{521} + \frac{s_2}{2^{23}})$;