Montgomery Arithmetic from a Software Perspective*

Joppe W. Bos[1] and Peter L. Montgomery[2]

[1]NXP Semiconductors
[2]Self

Abstract

This chapter describes Peter L. Montgomery's modular multiplication method and the various improvements to reduce the latency for software implementations on devices which have access to many computational units.

We propose a representation of residue classes so as to speed modular multiplication without affecting the modular addition and subtraction algorithms.

Peter L. Montgomery [55]

1 Introduction

Modular multiplication is a fundamental arithmetic operation, for instance when computing in a finite field or a finite ring, and forms one of the basic operations underlying almost all currently deployed public-key cryptographic protocols. The efficient computation of modular multiplication is an important research area since optimizations, even ones resulting in a constant-factor speedup, have a direct impact on our day-to-day information security life. In this chapter we review the computational aspects of Peter L. Montgomery's modular multiplication method [55] (known as *Montgomery multiplication*) from a software perspective (while the next chapter highlights the hardware perspective).

Throughout this chapter we use the terms *digit* and *word* interchangeably. To be more precise, we typically assume that a b-bit non-negative multi-precision integer X is represented by an array of $n = \lceil b/r \rceil$ computer words as

$$X = \sum_{i=0}^{n-1} x_i r^i$$

(the so-called radix-r representation), where $r = 2^w$ for the word size w of the target computer architecture and $0 \leq x_i < r$. Here x_i is the i-th word of the integer X.

*This material has been published as Chapter 2 in "Topics in Computational Number Theory Inspired by Peter L. Montgomery" edited by Joppe W. Bos and Arjen K. Lenstra and published by Cambridge University Press. See `www.cambridge.org/9781107109353`.

Let N be the positive modulus consisting of n digits while the input values to the modular multiplication method (A and B) are non-negative integers smaller than N and consist of up to n digits. When computing modular multiplication $C = AB \bmod N$, the definitional approach is first to compute the product $P = AB$. Next, a division is computed to obtain $P = NQ + C$ such that both C and Q are non-negative integers less than N. Knuth studies such algorithms for multi-precision non-negative integers [48, Alg. 4.3.1.D]. Counting word-by-word instructions, the method described by Knuth requires $O(n^2)$ multiplications and $O(n)$ divisions when implemented on a computer platform. However, on almost all computer platforms divisions are expensive (relative to the cost of a multiplication). Is it possible to perform modular multiplication without using any division instructions?

If one is allowed to perform some precomputation which only depends on the modulus N, then this question can be answered affirmatively. When computing the division step, the idea is to use only "cheap" divisions and "cheap" modular reductions when computing the modular multiplication in combination with a precomputed constant (the computation of which may require "expensive" divisions). These "cheap" operations are computations which either come for free or at a low cost on computer platforms. Virtually all modern computer architectures internally store and compute on data in binary format using some fixed word-size $r = 2^w$ as above. In practice, this means that all arithmetic operations are implicitly computed modulo 2^w (i.e., for free) and divisions or multiplications by (powers of) 2 can be computed by simply shifting the content of the register which holds this value.

Barrett introduced a modular multiplication approach (known as Barrett multiplication [6]) using this idea. This approach can be seen as a Newton method which uses a precomputed scaled variant of the modulus' reciprocal in order to use only such "cheap" divisions when computing (estimating and adjusting) the division step. After precomputing a single (multi-precision) value, an implementation of Barrett multiplication does not use any division instructions and requires $O(n^2)$ multiplication instructions.

Another and earlier approach based on precomputation is the main topic of this chapter: Montgomery multiplication. This method is the preferred choice in cryptographic applications when the modulus has no "special" form (besides being an odd positive integer) that would allow more efficient modular reduction techniques. See Section 3 on page 11 for applications of Montgomery multiplication in the "special" setting. In practice, Montgomery multiplication is the most efficient method when a generic modulus is used (see e.g., the comparison performed by Bosselaers, Govaerts, and Vandewalle [19]) and has a very regular structure which speeds up the implementation. Moreover, the structure of the algorithm (especially if its single branch, the notorious conditional "subtraction step", can be avoided, cf. page 9 in Section 2.4) has advantages when guarding against certain types of cryptographic attacks (for more information on *differential power analysis attacks* see the seminal paper by Kocher, Jaffe, and Jun [51]). In the next chapter, Montgomery's method is compared with a version of Barrett multiplication in order to be more precise about the computational advantages of the former technique.

As observed by Shand and Vuillemin in [66], Montgomery multiplication can be seen as a generalization of Hensel's odd division [40] for 2-adic numbers. In this chapter we explain the motivation behind Montgomery arithmetic. More specifically, we show how a change of the residue class representatives used improves the performance of modular multiplication. Next, we summarize some of the proposed modifications to Montgomery multiplication which can further speed up the algorithm in certain settings. Finally, we show how to implement Montgomery arithmetic in software. We especially study how to compute a single Montgomery multiplication concurrently using either vector instructions or when many computational units are available which can compute in parallel.

2 Montgomery multiplication

Let N be an odd b-bit integer and P a $2b$-bit integer such that $0 \leq P < N^2$. The idea behind Montgomery multiplication is to change the representatives of the residue classes and change the modular multiplication accordingly. More precisely, we are not going to compute $P \bmod N$ but $2^{-b}P \bmod N$ instead. This explains the requirement that N needs to be odd (otherwise $2^{-b} \bmod N$ does not exist). It turns out that this approach is more efficient (by a constant factor) on computer platforms.

Let us start with a basic example to illustrate the strategy used. A first idea is to reduce the value P one bit at a time and repeat this for b bits such that the result has been reduced from $2b$ to b bits (as required). This can be achieved *without computing any expensive modular divisions* by noting that

$$2^{-1}P \bmod N = \begin{cases} P/2 & \text{if } P \text{ is even,} \\ (P+N)/2 & \text{if } P \text{ is odd.} \end{cases}$$

When P is even, the division by two can be computed with a basic operation on computer architectures: shift the number one position to the right. When P is odd one can not simply compute this division by shifting. A computationally efficient approach to compute this division by two is to make this number P even by adding the odd modulus N, since obviously modulo N this is the same. This allows one to compute $2^{-1}P \bmod N$ at the cost of (at most) a single addition and a single shift.

Let us compute $D < 2N$ and $Q < 2^b$ such that $P = 2^bD - NQ$ since then $D \equiv 2^{-b}P \bmod N$. Initially set D equal to P and Q equal to zero. We denote by q_i the i-th digit when Q is written in binary (radix-2), i.e., $Q = \sum_{i=0}^{b-1} q_i 2^i$. Next perform the following two steps b times starting at $i = 0$ until the last time when $i = b - 1$:

$$\textbf{(Step 1)}\ q_i = D \bmod 2, \qquad \textbf{(Step 2)}\ D = (D + q_i N)/2.$$

This procedure gradually builds the desired Q and at the start of every iteration

$$P = 2^i D - NQ$$

remains invariant. The process is illustrated in the example below.

Example 1: Radix-2 Montgomery reduction

For $N = 7$ (3 bits) and $P = 20 < 7^2$ we compute $D \equiv 2^{-3}P \bmod N$. At the start of the algorithm, set $D = P = 20$ and $Q = 0$.

$$\begin{array}{lll} i = 0, & 20 = 2^0 \cdot 20 - 7 \cdot 0 & \Rightarrow 2^{-0} \cdot 20 \equiv 20 \bmod 7 \\ & \textbf{(Step 1)}\ q_0 = 20 \bmod 2 = 0, & \textbf{(Step 2)}\ D = (20 + 0 \cdot 7)/2 = 10 \\ i = 1, & 20 = 2^1 \cdot 10 - 7 \cdot 0 & \Rightarrow 2^{-1} \cdot 20 \equiv 10 \bmod 7 \\ & \textbf{(Step 1)}\ q_1 = 10 \bmod 2 = 0, & \textbf{(Step 2)}\ D = (10 + 0 \cdot 7)/2 = 5 \\ i = 2, & 20 = 2^2 \cdot 5 - 7 \cdot 0 & \Rightarrow 2^{-2} \cdot 20 \equiv 5 \bmod 7 \\ & \textbf{(Step 1)}\ q_2 = 5 \bmod 2 = 1, & \textbf{(Step 2)}\ D = (5 + 1 \cdot 7)/2 = 6 \end{array}$$

Since $Q = q_0 2^0 + q_1 2^1 + q_2 2^2 = 4$ and $P = 20 = 2^k D - NQ = 2^3 \cdot 6 - 7 \cdot 4$, we have computed $2^{-3} \cdot 20 \equiv 6 \bmod 7$.

Algorithm 1 The Montgomery reduction algorithm. Compute PR^{-1} modulo the odd modulus N given the Montgomery radix $R > N$ and using the precomputed Montgomery constant $\mu = -N^{-1} \bmod R$.

Input: $N, P,$ such that $0 \leq P < N^2$.
Output: $C \equiv PR^{-1} \bmod N$ such that $0 \leq C < N$.
1: $q \leftarrow \mu(P \bmod R) \bmod R$
2: $C \leftarrow (P + Nq)/R$
3: **if** $C \geq N$ **then**
4: $\quad C \leftarrow C - N$
5: **end if**
6: **return** C

The approach behind Montgomery multiplication [55] generalizes this idea. Instead of dividing by two at every iteration the idea is to divide by a Montgomery radix R which needs to be coprime to, and should be larger than, N. By precomputing the value

$$\mu = -N^{-1} \bmod R,$$

adding a specific multiple of the modulus N to the current value P ensures that

$$\begin{aligned} P + N\,(\mu P \bmod R) &\equiv P - N\left(N^{-1}P \bmod R\right) \\ &\equiv P - P \equiv 0 \pmod{R}. \end{aligned} \tag{1}$$

Hence, $P + N\,(P\mu \bmod R)$ is divisible by the Montgomery radix R while P does not change modulo N. Let P be the product of two non-negative integers that are both less than N. After applying Equation (1) and dividing by R, the value P (bounded by N^2) has been reduced to at most $2N$ since

$$0 \leq \frac{P + N(\mu P \bmod R)}{R} < \frac{N^2 + NR}{R} < 2N \tag{2}$$

(since R was chosen larger than N). This approach is summarized in Algorithm 1: given an integer P bounded by N^2, it computes $PR^{-1} \bmod N$, bounded by N, *without* using any "expensive" division instructions when assuming the reductions modulo R and divisions by R can be computed (almost) for free. On most computer platforms, where one chooses R as a power of two, this assumption is indeed true.

Equation (2) guarantees that the output is bounded by $2N$. Hence, a conditional subtraction needs to be computed at the end of Algorithm 1 to ensure the output is less than N. The process is illustrated in the example below, where P is the product of integers A, B with $0 \leq A, B < N$.

Algorithm 2 The radix-r interleaved Montgomery multiplication algorithm. Compute $(AB)R^{-1}$ modulo the odd modulus N given the Montgomery radix $R = r^n$ and using the precomputed Montgomery constant $\mu = -N^{-1} \bmod r$. The modulus N is such that $r^{n-1} \leq N < r^n$ and r and N are coprime.

Input: $A = \sum_{i=0}^{n-1} a_i r^i, B, N$ such that $0 \leq a_i < r,\ 0 \leq A, B < R$.
Output: $C \equiv (AB)R^{-1} \bmod N$ such that $0 \leq C < N$.

1: $C \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad C \leftarrow C + a_i B$
4: $\quad q \leftarrow \mu C \bmod r$
5: $\quad C \leftarrow (C + Nq)/r$
6: **end for**
7: **if** $C \geq N$ **then**
8: $\quad C \leftarrow C - N$
9: **end if**
10: **return** C

Example 2: Montgomery multiplication

Exact divisions by $10^2 = 100$ are visually convenient when using a decimal system: just shift the number two places to the right (or "erase" the two least significant digits). Assume the following modular reduction approach: use the Montgomery radix $R = 100$ when computing modulo $N = 97$. This example computes the Montgomery product of $A = 42$ with $B = 17$. First, precompute the Montgomery constant

$$\mu = -N^{-1} \bmod R = -97^{-1} \bmod 100 = 67.$$

After computing the product $P = AB = 42 \cdot 17 = 714$, compute the first two steps of Algorithm 1 omitting the division by R:

$$\begin{aligned} P + N(\mu P \bmod R) &= 714 + 97(67 \cdot 714 \bmod 100) \\ &= 714 + 97(67 \cdot 14 \bmod 100) \\ &= 714 + 97(938 \bmod 100) \\ &= 714 + 97 \cdot 38 \\ &= 4400. \end{aligned}$$

Indeed, 4400 is divisible by $R = 100$ and we have computed

$$(AB)R^{-1} \equiv 42 \cdot 17 \cdot 100^{-1} \equiv 44 \pmod{97}$$

without using any "difficult" modular divisions.

2.1 Interleaved Montgomery multiplication

When working with multi-precision integers, integers consisting of n digits of w bits each, it is common to write the Montgomery radix R as

$$R = r^n = 2^{wn},$$

where w is the word-size of the architecture where Montgomery multiplication is implemented. The Montgomery multiplication method (as outlined in Algorithm 1) assumes the multiplication is computed before performing the Montgomery reduction. This has the advantage that one can use asymptotically fast multiplication methods (like e.g., Karatsuba [44], Toom-Cook [74, 25], Schönhage-Strassen [64], or Fürer [30] multiplication). However, this has the disadvantage that the intermediate results can be as large as r^{2n+1}. Or, stated differently, when using a machine word size of w bits the intermediate results are represented using at most $2n + 1$ computer words.

The multi-precision setting was already handled in Montgomery's original paper [55, Section 2] and the reduction and multiplication were meant to be interleaved by design When representing the integers in radix-r representation

$$A = \sum_{i=0}^{n-1} a_i r^i, \quad \text{such that } 0 \leq a_i < r$$

then the radix-r interleaved Montgomery multiplication (see also the work by Dussé and Kaliski Jr. in [29]) ensures the intermediate results never exceed $r+2$ computer words. This approach is presented in Algorithm 2. Note that this interleaves the naive schoolbook multiplication algorithm with the Montgomery reduction and therefore does not make use of any asymptotically faster multiplication algorithm. The idea is that every iteration divides by the value r (instead of dividing once by $R = r^n$ in the "non-interleaved" Montgomery multiplication algorithm). Hence, the value for μ is adjusted accordingly. In [22], Koç, Acar, and Kaliski Jr. compare different approaches to implementing multi-precision Montgomery multiplication. According to this analysis, the interleaved radix-r approach, referred to as coarsely integrated operand scanning in [22], performs best in practice.

2.2 Using Montgomery arithmetic in practice

As we have seen earlier in this section and in Algorithm 1 on page 4 Montgomery multiplication computes $C \equiv PR^{-1} \bmod N$. It follows that, in order to use Montgomery multiplication in practice, one should transform the input operands A and B to $\tilde{A} = AR \bmod N$ and $\tilde{B} = BR \bmod N$: this is called the *Montgomery representation*. The transformed inputs (converted to the Montgomery representation) are used in the Montgomery multiplication algorithm. At the end of a series of modular multiplications the result, in Montgomery representation, is transformed back. This works correctly since Montgomery multiplication $\mathcal{M}(\tilde{A}, \tilde{B}, N)$ computes $(\tilde{A}\tilde{B})R^{-1} \bmod N$ and it is indeed the case that the Montgomery representation $\tilde{C}$ of $C = AB \bmod N$ is computed from the Montgomery representations of A and B since

$$\begin{aligned}
\tilde{C} \equiv \mathcal{M}(\tilde{A}, \tilde{B}, N) &\equiv (\tilde{A}\tilde{B})R^{-1} \\
&\equiv (AR)(BR)R^{-1} \\
&\equiv (AB)R \\
&\equiv CR \pmod{N}.
\end{aligned}$$

Converting an integer A to its Montgomery representation $\tilde{A} = AR \bmod N$ can be performed using Montgomery multiplication with the help of the precomputed constant $R^2 \bmod N$ since

$$\mathcal{M}(A, R^2, N) \equiv (AR^2)R^{-1} \equiv AR \equiv \tilde{A} \pmod{N}.$$

Converting (the result) back from the Montgomery representation to the regular representation is the same as computing a Montgomery multiplication with the integer value one since

$$\mathcal{M}(\tilde{A}, 1, N) \equiv (\tilde{A} \cdot 1)R^{-1} \equiv (AR)R^{-1} \equiv A \pmod{N}.$$

As mentioned earlier, due to the overhead of changing representations, Montgomery arithmetic is best when used to replace a sequence of modular multiplications, since this overhead is amortized. A typical use-case scenario is when computing a modular exponentiation as required in the RSA cryptosystem [63].

As noted in the original paper [55] (see the quote at the start of this chapter) computing with numbers in Montgomery representation does not affect the modular addition and subtraction algorithms. This can be seen from

$$\tilde{A} \pm \tilde{B} \equiv AR \pm BR \equiv (A \pm B)R \equiv \widetilde{A \pm B} \pmod{N}.$$

Computing the *Montgomery inverse* is, however, affected. The Montgomery inverse of a value $\tilde{A}$ in Montgomery representation is $\widetilde{A^{-1}}$. This is different from computing the inverse of $\tilde{A}$ modulo N since $\tilde{A}^{-1} \equiv (AR)^{-1} \equiv A^{-1}R^{-1} \pmod{N}$ is the Montgomery representation of the value $A^{-1}R^{-2}$. One of the correct ways of computing the Montgomery inverse is to invert the number in its Montgomery representation and Montgomery multiply this result by R^3 since

$$\mathcal{M}(\tilde{A}^{-1}, R^3, N) \equiv ((AR)^{-1}R^3)R^{-1} \equiv A^{-1}R \equiv \widetilde{A^{-1}} \pmod{N}.$$

Another approach, which does not require any precomputed constant, is to compute the Montgomery reduction of a Montgomery residue $\tilde{A}$ twice before inverting since

$$\begin{aligned}
\mathcal{M}(\mathcal{M}(\tilde{A}, 1, N), 1, N)^{-1} &\equiv \mathcal{M}((AR)R^{-1}, 1, N)^{-1} \\
&\equiv \mathcal{M}(A, 1, N)^{-1} \\
&\equiv (AR^{-1})^{-1} \\
&\equiv A^{-1}R \\
&\equiv \widetilde{A^{-1}} \pmod{N}.
\end{aligned}$$

2.3 Computing the Montgomery constants μ and R^2

In order to use Montgomery multiplication one has to precompute the Montgomery constant $\mu = -N^{-1} \bmod r$. This can be computed with, for instance, the extended Euclidean algorithm. A particularly efficient algorithm to compute μ when r is a power of two and N is odd, the typical setting used in cryptology, is given by Dussé and Kaliski Jr. and presented in [29]. This approach is recalled in Algorithm 3.

To show that this approach is correct, it suffices to show that at the start of Algorithm 3 and at the end of every iteration we have $Ny_i \equiv 1 \pmod{2^i}$. This can be shown by induction as follows. At the start of the algorithm we set y to one, denote this start setting with y_1, and the condition holds since

Algorithm 3 Compute the Montgomery constant $\mu = -N^{-1} \bmod r$ for odd values N and $r = 2^w$ as presented by Dussé and Kaliski Jr. in [29].

Input: Odd integer N and $r = 2^w$ for $w \geq 1$.
Output: $\mu = -N^{-1} \bmod r$

```
y ← 1
for i = 2 to w do
    if (Ny mod 2^i) ≠ 1 then
        y ← y + 2^(i−1)
    end if
end for
return μ ← r − y
```

N is odd by assumption. Denote with y_i, for $2 \leq i \leq w$, the value of y in Algorithm 3 at the end of iteration i. When $i > 1$, our induction hypothesis is that $Ny_{i-1} = 1 + 2^{i-1}m$ for some positive integer m, at the end of iteration $i - 1$.

We consider two cases

- (m is even) Since $Ny_{i-1} = 1 + \frac{m}{2}2^i \equiv 1 \pmod{2^i}$ we can simply update y_i to y_{i-1} and the condition holds.
- (m is odd) Since $Ny_{i-1} = 1 + 2^{i-1} + \frac{m-1}{2}2^i \equiv 1 + 2^{i-1} \pmod{2^i}$, we update y_i with $y_{i-1} + 2^{i-1}$. We obtain $N(y_{i-1} + 2^{i-1}) = 1 + 2^{i-1}(1 + N) + \frac{m-1}{2}2^i \equiv 1 \pmod{2^i}$ since N is odd.

Hence, after the for loop y_w is such that $Ny_w \equiv 1 \bmod 2^w$ and the returned value $\mu = r - y_w \equiv 2^w - N^{-1} \equiv -N^{-1} \pmod{2^w}$ has been computed correctly.

The precomputed constant $R^2 \bmod N$ is required when converting a residue modulo N from its regular to its Montgomery representation (see Section 2.2 on page 6). When $R = r^n$ is a power of two, which in practice is typically the case since $r = 2^w$, then this precomputed value $R^2 \bmod N$ can also be computed efficiently. For convenience, assume $R = r^n = 2^{wn}$ and $2^{wn-1} \leq N < 2^{wn}$ (but this approach can easily be adapted when N is smaller than 2^{wn-1}). Commence by setting the initial $c_0 = 2^{wn-1} < N$. Next, start at $i = 1$ and compute

$$c_i \equiv c_{i-1} + c_{i-1} \bmod N$$

and increase i until $i = wn + 1$. The final value

$$c_{wn+1} \equiv 2^{wn+1}c_0 \equiv 2^{wn+1}2^{wn-1} \equiv 2^{2wn} \equiv (2^{wn})^2 \equiv R^2 \bmod N$$

as required and can be computed with $wn + 1$ modular additions.

2.4 On the final conditional subtraction

It is possible to alter or even completely remove the conditional subtraction from lines 3–4 in Algorithm 1 on page 4. This is often motivated by either performance considerations or turning the (software) implementation into straight-line code that requires no conditional branches. This is one of the basic requirements for cryptographic implementations which need to protect themselves against a variety of (simple) side-channel attacks as introduced by Kocher, Jaffe, and Jun [51] (those attacks which use physical information, such as elapsed time, obtained when executing an implementation, to deduce information about the secret key material used). Ensuring constant running-time is a first

step in achieving this goal. In order to change or remove this final conditional subtraction the general idea is to bound the input and output of the Montgomery multiplication in such a way that they can be re-used in a subsequent Montgomery multiplication computation. This means using a redundant representation, in which the representation of the residues used is not unique and can be larger than N.

2.4.1 Subtraction-less Montgomery multiplication

The conditional subtraction can be omitted when the size of the modulus N is appropriately selected with respect to the Montgomery radix R. (This is a result presented by Shand and Vuillemin in [66] but see also the sequence of papers by Walter, Hachez, and Quisquater [77, 36, 78].) The idea is to select the modulus N such that $4N < R$ and to use a redundant representation for the input and output values of the algorithm. More specifically, we assume $A, B \in \mathbb{Z}/2N\mathbb{Z}$ (residues modulo $2N$), where $0 \leq A, B < 2N$, since then the outputs of Algorithm 1 on page 4 and Algorithm 2 on page 5 are bounded by

$$0 \leq \frac{AB + N(\mu AB \bmod R)}{R} < \frac{(2N)^2 + NR}{R} < \frac{NR + NR}{R} = 2N. \tag{3}$$

Hence, the result can be reused as input to the same Montgomery multiplication algorithm. This avoids the need for the conditional subtraction except in a final correction step (after having computed a sequence of Montgomery multiplications) where one reduces the value to a unique representation with a single conditional subtraction.

In practice, this might reduce the number of arithmetic operations whenever the modulus can be chosen beforehand and, moreover, simplifies the code. However, in the popular use-cases in cryptography, e.g., in the setting of computing modular exponentiations when using schemes based on RSA [63] where the bit-length of the modulus must be a power of two due to compliance with cryptographic standards, the condition $4N < R$ results in a Montgomery-radix R which is represented using one additional computer word (compared to the number of words needed to represent the modulus N). Hence, in this setting, such a multi-precision implementation *without* a conditional subtraction needs one more iteration (when using the interleaved Montgomery multiplication algorithm) to compute the result compared to a version which computes the conditional subtraction.

2.4.2 Montgomery multiplication with a simpler final comparison

Another approach is not to remove the subtraction but make this operation computationally cheaper. See the analysis by Walter and Thompson in [79, Section 2.2] which is introduced again by Pu and Zhao in [62]. In practice, the Montgomery-radix $R = r^n$ is often chosen as a multiple of the word-size of the computer architecture used (e.g., $r = 2^w$ for $w \in \{8, 16, 32, 64\}$). The idea is to reduce the output of the Montgomery multiplication to $\{0, 1, \ldots, R - 1\}$. instead of to the smaller range $\{0, 1, \ldots, N - 1\}$. Just as above, this is a redundant representation but working with residues from $\mathbb{Z}/R\mathbb{Z}$. This representation does not need more computer words to represent the result and therefore does not increase the number of iterations one needs to compute; something which might be the case when the Montgomery radix is increased to remove the conditional subtraction. Computing the comparison if an integer $x = \sum_{i=0}^{n} x_i r^i$ is at least $R = r^n$ can be done efficiently by verifying if the most significant word x_n is non-zero. This is significantly more efficient compared with computing a full multi-precision comparison.

This approach is correct since if the input values A and B are bounded by R then the output of the

Montgomery multiplication, before the conditional subtraction, is bounded by

$$0 \leq \frac{B + N(\mu AB \bmod R)}{R} < \frac{R^2 + NR}{R} = R + N. \tag{4}$$

Subtracting N whenever the result is at least R ensures that the output is also less than R. Hence, one still requires to evaluate the condition for subtraction in every Montgomery multiplication. However, the greater-or-equal comparison becomes significantly cheaper and the number of iterations required to compute the interleaved Montgomery multiplication algorithm remains unchanged. In the setting where a constant running-time is required this approach does not seem to bring a significant advantage (see the security analysis by Walter and Thompson in [79, Section 2.2] for more details). A simple constant-running time solution is to compute the subtraction and select this result if no borrow occurred. However, when constant running-time is not an issue this approach (using a cheaper comparison) can speed up the Montgomery multiplication algorithm.

2.5 Montgomery multiplication in $\mathbb{F}_{2^k}$

The idea behind Montgomery multiplication carries over to finite fields of cardinality 2^k as well. Such finite fields are known as *binary fields* or *characteristic-two finite fields*. The application of Montgomery multiplication to this setting is outlined by Koç and Acar in [50]. Let $n(x)$ be an irreducible polynomial of degree k. Then an element $a(x) \in \mathbb{F}_{2^k} \cong \mathbb{F}_2[x]/(n(x))$ can be represented in the polynomial-basis representation by a polynomial of degree at most $k - 1$

$$a(x) = \sum_{i=0}^{k-1} a_i x^i \quad \text{where } a_i \in \mathbb{F}_2.$$

The equivalent of the Montgomery-radix is the polynomial $r(x) \in \mathbb{F}_2[x]/(n(x))$ which in practice is chosen as $r(x) = x^k$. Since $n(x)$ is irreducible this ensures that the inverse $r^{-1}(x) \bmod n(x)$ exists and the Montgomery multiplication

$$a(x)b(x)r^{-1}(x) \bmod n(x)$$

is well-defined.

Let $a(x), b(x) \in \mathbb{F}_2[x]/(n(x))$ and their product $p(x) = a(x)b(x)$ of degree at most $2(k-1)$. Computing the Montgomery reduction $p(x)r^{-1}(x) \bmod n(x)$ of $p(x)$ can be done using the same steps as presented in Algorithm 1 on page 4 given the precomputed Montgomery constant $\mu(x) = -n(x)^{-1} \bmod r(x)$. Hence, one computes

$$\begin{aligned} q(x) &= p(x)\mu(x) \bmod r(x) \\ c(x) &= (p(x) + q(x)n(x))/r(x). \end{aligned}$$

Note that the final conditional subtraction step is not required since

$$\deg(c(x)) \leq \max(2(k-1), k-1+k) - k = k - 1,$$

(because $r(x)$ is a degree k polynomial). A large characteristic version of this approach using the interleaved Montgomery multiplication for finite fields of large prime characteristic from Section 2.1 on page 6, works here as well.

3 Using primes of a special form

In some settings in cryptography, most notably in elliptic curve cryptography (introduced independently by Miller and Koblitz in [54, 49]), the (prime) modulus can be chosen freely and is fixed for a large number of modular arithmetic operations. In order to gain a constant factor speedup when computing the modular multiplication, Solinas suggested [68] a specific set of special primes which were subsequently included in the FIPS 186 standard [75] used in public-key cryptography. More recently, prime moduli of the form $2^s \pm \delta$ have gained popularity where $s, \delta \in \mathbb{Z}_{>0}$ and $\delta < 2^s$ such that δ is a (very) small integer. More precisely, the constant δ is small compared to the typical word-size of computer architectures used (e.g., less than 2^{32}) and often is chosen as the smallest integer such that one of $2^s \pm \delta$ is prime. One should be aware that the usage of such primes of a special form not only accelerates the cryptographic implementations, the cryptanalytic methods benefit as well. See, for instance, the work by this chapter's authors, Kleinjung, and Lenstra related to efficient arithmetic to factor Mersenne numbers (numbers of the form $2^M - 1$) in [15]. An example of one of the primes, suggested by Solinas, is $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ where the exponents are selected to be a multiple of 32 to speed up implementations on 32-bit platforms (but see for instance the work by Käsper how to implement such primes efficiently on 64-bit platforms [45]). A more recent example proposed by Bernstein [7] is to use the prime $2^{255} - 19$ to implement efficient modular arithmetic in the setting of elliptic curve cryptography.

3.1 Faster modular reduction with primes of a special form

We use the Mersenne prime $2^{127} - 1$ as an example to illustrate the various modular reduction techniques in this section. Given two integers a and b, such that $0 \leq a, b < 2^{127} - 1$, the modular product $ab \bmod 2^{127} - 1$ can be computed efficiently as follows. (We follow the description given by the first author of this chapter, Costello, Hisil, and Lauter from [10]). First compute the product with one's preferred multiplication algorithm and write the result in radix-2^{128} representation

$$c = ab = c_1 2^{128} + c_0, \qquad \text{where } 0 \leq c_0 < 2^{128} \text{ and } 0 \leq c_1 < 2^{126}.$$

This product can be almost fully reduced by subtracting $2c_1$ times the modulus since

$$c \equiv c_1 2^{128} + c_0 - 2c_1(2^{127} - 1) \equiv c_0 + 2c_1 \pmod{2^{127} - 1}.$$

Moreover, we can subtract $2^{127} - 1$ one more time if the bit $\lfloor c_0/2^{127} \rfloor$ (the most significant bit of c_0) is set. When combining these two observations a first reduction step can be computed as

$$c' = (c_0 \bmod 2^{127}) + 2c_1 + \lfloor c_0/2^{127} \rfloor \equiv c \pmod{2^{127} - 1} \tag{5}$$

This already ensures that the result $0 \leq c' < 2^{128}$ since

$$c' \leq 2^{127} - 1 + 2(2^{126} - 1) + 1 < 2^{128}.$$

One can then reduce c' further using conditional subtractions. Reduction modulo $2^{127} - 1$ can therefore be computed without using any multiplications or expensive divisions by taking advantage of the form of the modulus.

3.2 Faster Montgomery reduction with primes of a special form

Along the same lines one can select moduli which speed up the operations when computing Montgomery reduction. Such special moduli have been proposed many times in the literature to reduce the number of arithmetic operations (see for instance the work by Lenstra [52], Acar and Shumow [1], Knezevic, Vercauteren, and Verbauwhede [47], Hamburg [37], and the first author of this chapter, Costello, Hisil, and Lauter [10, 11]). They are sometimes referred to as Montgomery-friendly moduli or Montgomery-friendly primes. Techniques to scale an existing modulus such that this scaled modulus has a special shape which reduces the number of arithmetic operations, using the same techniques as for the Montgomery-friendly moduli, are also known and called Montgomery tail tailoring by Hars in [39]. Following the description in the book by Brent and Zimmermann [20], this can be seen as a form of preconditioning as suggested by Svoboda in [70] in the setting of division.

When one is free to select the modulus N beforehand, then the number of arithmetic operations can be reduced if the modulus is selected such that

$$\mu = -N^{-1} \bmod r = \pm 1$$

in the setting of interleaved Montgomery multiplication (as also used by Dixon and Lenstra in [28]). This ensures that the multiplication by μ can be avoided (since $\mu = \pm 1$) in every iteration of the interleaved Montgomery multiplication algorithm. This puts a first requirement on the modulus, namely $N \equiv \mp 1 \bmod r$. In practice, $r = 2^w$ where w is the word-size of the computer architecture. Hence, this requirement puts a restriction on the least significant word of the modulus (which equals either 1 or $-1 \equiv 2^w - 1 \bmod 2^w$).

Combining lines 4 and 5 from the interleaved Montgomery multiplication (Algorithm 2 on page 5) we see that one has to compute $\frac{C+N(\mu C \bmod r)}{r}$. Besides the multiplication by μ one has to compute a multi-word multiplication with the (fixed) modulus N. In the same vein as the techniques from Section 3.1 above, one can require N to have a special shape such that this multiplication can be computed faster in practice. This can be achieved, for instance, when one of the computer words of the modulus is small or has some special shape while the remainder of the digits are zero except for the most significant word (e.g., when $\mu = 1$). Along these lines the first author of this chapter, Costello, Longa, and Naehrig select primes for usage in elliptic curve cryptography where

$$N = 2^{\alpha}(2^{\beta} - \gamma) \pm 1 \tag{6}$$

where α, β, and γ are integers such that $\gamma < 2^{\beta} \leq r$.

The final requirement on the modulus is to ensure that $4N < R = r^n$ since this avoids the final conditional subtraction (as shown on page 9 in Section 2.4). Examples of such Montgomery-friendly moduli include

$$2^{252} - 2^{232} - 1 = 2^{192}(2^{60} - 2^{40}) - 1 = 2^{224}(2^{28} - 2^8) - 1$$

(written in different form to show the usage on different architectures which can compute with β-bit integers) proposed by Hamburg in [37] and the modulus

$$2^{240}(2^{14} - 127) - 1 = 2^{222}(2^{32} - 2^{18} \cdot 127) - 1$$

proposed by the first author of this chapter, Costello, Longa, and Naehrig in [12]. The approach is illustrated in the example below. Other examples of Montgomery-friendly moduli are given in [32, Table 4] based on Chung-Hasan arithmetic [24].

Example 3: Montgomery-friendly reduction modulo $2^{127} - 1$

Let us consider Montgomery reduction modulo $2^{127} - 1$ on a 64-bit computer architecture ($w = 64$). This means we have $\alpha = 64$, $\beta = 63$, and $\gamma = 0$ in Equation (6) on page 12 ($2^{127} - 1 = 2^{64}(2^{63} - 0) - 1$). Multiplication by μ can be avoided since $\mu = -(2^{127} - 1)^{-1} \bmod 2^{64} = 1$. Furthermore, due to the special form of the modulus the multiplication by $2^{127} - 1$ can be simplified. The computation of $\frac{C+N(\mu C \bmod r)}{r}$, which needs to be done for each computer word (twice in this setting), can be simplified when using the Montgomery interleaved multiplication algorithm.

Write $C = c_2 2^{128} + c_1 2^{64} + c_0$ (see line 3 in Algorithm 2 on page 5) with $0 \leq c_2, c_1, c_0 < 2^{64}$, then

$$\begin{aligned}
\frac{C + N(\mu C \bmod r)}{r} &= \frac{C + (2^{127} - 1)(C \bmod 2^{64})}{2^{64}} \\
&= \frac{(c_2 2^{128} + c_1 2^{64} + c_0) + (2^{127} - 1)c_0}{2^{64}} \\
&= \frac{c_2 2^{128} + c_1 2^{64} + c_0 2^{127}}{2^{64}} \\
&= c_2 2^{64} + c_1 + c_0 2^{63}.
\end{aligned}$$

Hence, only two addition and two shift operations are needed in this computation.

4 Concurrent computing of Montgomery multiplication

Since the seminal paper by the second author introducing modular multiplication without trial division, people have studied ways to obtain better performance on different computer architectures. Many of these techniques are specifically tailored towards a (family) of platforms motivated by the desire to enhance the practical performance of public-key cryptosystems.

One approach focuses on reducing the latency of the Montgomery multiplication operation. This might be achieved by computing the Montgomery product using many computational units in parallel. One example is to use the single instruction, multiple data (SIMD) programming paradigm. In this setting a single vector instruction applies to multiple data elements in parallel. Many modern computer architectures have access to vector instruction set extensions to perform SIMD operations. Example platforms include the popular high-end x86 architecture as well as the embedded ARM platform which can be found in the majority of modern smartphones and tablets. To highlight the potential, Gueron and Krasnov were the first to show in [35] that the computation of Montgomery multiplication on the 256-bit wide vector instruction set AVX2 is faster than the same computation on the classical arithmetic logic unit (ALU) on the x86_64 platform.

In Section 4.1 below we outline the approach by the authors of this chapter, Shumow, and Zaverucha from [17] for computing a single Montgomery multiplication using vector instruction set extensions which support 2-way SIMD operations (i.e., perform the same operation on two data points simultaneously). This approach allows one to split the computation of the interleaved Montgomery multiplication into two parts which can be computed in parallel. Note that in a follow-up work [65] by Seo, Liu, Großschädl, and Choi it is shown how to improve the performance on 2-way SIMD architectures even further. Instead of computing the two multiplications concurrently, as is presented in Section 4.1, they compute every multiplication using 2-way SIMD instructions. By careful schedul-

ing of the instructions they manage to significantly reduce the read-after-write dependencies which reduces the number of bubbles (execution delays in the instruction pipeline). This results in a software implementation which outperforms the one presented in [17]. It would be interesting to see if these two approaches (from [17] and [65]) can be merged on platforms which support efficient 4-way SIMD instructions.

In Section 4.3 on page 21 we show how to compute Montgomery multiplication when integers are represented in a residue number system. This approach can be used to compute Montgomery arithmetic efficiently on highly parallel computer architectures which have hundreds of computational cores or more and when large moduli are used (such as in the RSA cryptosystem).

4.0.1 Related work on concurrent computing of Montgomery multiplication

A parallel software approach describing systolic Montgomery multiplication is described by Dixon and Lenstra in [28], by Iwamura, Matsumoto, and Imai in [42], and Walter in [76]. See Chapter 3 for more information about systolic Montgomery multiplication. Another approach is to use the SIMD vector instructions to compute *multiple* Montgomery multiplications in parallel. This can be useful in applications where many computations need to be processed in parallel such as batch-RSA or cryptanalysis. This approach is studied by Page and Smart in [58] using the SSE2 vector instructions on a Pentium 4 and by the first author of this chapter in [9] on the Cell Broadband Engine (see Section 4.2.1 on page 18 for more details about this platform).

An approach based on Montgomery multiplication which allows one to split the operand into two parts, which can then be processed in parallel, is called bipartite modular multiplication and was introduced by Kaihara and Takagi in [43]. The idea is to use a Montgomery radix $R = r^{\alpha n}$ where α is a rational number such that αn is an integer and $0 < \alpha n < n$: hence, the radix R is smaller than the modulus N. For example, one can choose α such that $\alpha n = \lfloor \frac{n}{2} \rfloor$. In order to compute $xyr^{-\alpha n} \bmod N$ (where $0 \leq x, y < N$) write $y = y_1 r^{\alpha n} + y_0$ and compute

$$xy_1 \bmod N \quad \text{and} \quad xy_0 r^{-\alpha n} \bmod N$$

in parallel using a regular interleaved modular multiplication algorithm (see, e.g., the work by Brickell [21]) and the interleaved Montgomery multiplication algorithm, respectively. The sum of the two products gives the correct Montgomery product of x and y since

$$\begin{aligned}(xy_1 \bmod N) + (xy_0 r^{-\alpha n} \bmod N) &\equiv x(y_1 r^{\alpha n} + y_0) r^{-\alpha n} \\ &\equiv xyr^{-\alpha n} \pmod N.\end{aligned}$$

4.1 Montgomery multiplication using SIMD extensions

This section is an extended version of the description of the idea presented by this chapter's authors, Shumow, and Zaverucha in [17] where an algorithm is presented to compute the interleaved Montgomery multiplication using two threads running in parallel performing identical arithmetic steps. Hence, this algorithm runs efficiently when using 2-way SIMD vector instructions as frequently found on modern computer architectures. For illustrative purposes we assume a radix-2^{32} system, but this can be adjusted accordingly to any other radix system. Note that for efficiency considerations the choice of the radix system depends on the vector instructions available.

Algorithm 2 on page 5 outlines the interleaved Montgomery multiplication algorithm and computes two $1 \times n \to (n+1)$ computer word multiplications, namely $a_i B$ and qN, and a single $1 \times 1 \to 1$

computer word multiplication ($\mu C \bmod 2^{32}$) every iteration. Unfortunately, these three multiplications depend on each other and therefore can not be computed in parallel. Every iteration computes (see Algorithm 2)

1. $C \leftarrow C + a_i B$
2. $q \leftarrow \mu C \bmod 2^{32}$
3. $C \leftarrow (C + qN)/2^{32}$

In order to reduce latency, we would like to compute the two $1 \times n \rightarrow (n+1)$ word multiplications in parallel using vector instructions. This can be achieved by removing the dependency between these two multi-word multiplications by computing the value of q first. The first word c_0 of $C = \sum_{i=0}^{n-1} c_i 2^{32i}$ is computed twice: once for the computation of q (in $\mu C \bmod 2^{32}$) and then again in the parallel computation of $C + a_i B$. This is a relatively minor penalty of one additional one-word multiplication and addition per iteration to make these two multi-word multiplications independent of each other. This means an iteration i can be computed as

1. $q \leftarrow \mu(c_0 + a_i b_0) \bmod 2^{32}$
2. $C \leftarrow C + a_i B$
3. $C \leftarrow (C + qN)/2^{32}$

and the $1 \times n \rightarrow (n+1)$ word multiplications in steps 2 and 3 ($a_i B$ and qN) can be computed in parallel using, for instance, 2-way SIMD vector instructions.

In order to rewrite the remaining operations, besides the multiplication, the authors of [17] suggest inverting the sign of the Montgomery constant μ, i.e., instead of using $-N^{-1} \bmod 2^{32}$ use $\mu = N^{-1} \bmod 2^{32}$. When computing the Montgomery product $C = AB2^{-32n} \bmod N$, one can compute D (which contains the sum of the products $a_i B$) and E (which contains the sum of the products qN), separately and in parallel using the same arithmetic operations. Due to the modified choice of the Montgomery constant μ we have $C = D - E \equiv AB2^{-32n} \pmod{M}$, where $0 \leq D, E < N$: the maximum values of both D and E fit in an n-word integer. This approach is presented in Algorithm 4 on the next page.

At the start of every iteration in the for-loop iterating with j, the two separate computational streams running in parallel need to communicate information to compute the value of q. More precisely, this requires the knowledge of both d_0 and e_0, the least significant words of D and E respectively. Once the values of both d_0 and e_0 are known to one of the computational streams, the updated value of q can be computed as

$$\begin{aligned} q &= ((\mu a_0) b_j + \mu(d_0 - e_0)) \bmod 2^{32} \\ &= \mu(c_0 + b_j a_0) \bmod 2^{32} \end{aligned}$$

since $c_0 = d_0 - e_0$.

Except for this communication cost between the two streams, to compute the value of q, all arithmetic computations performed by computation 1 and computation 2 in the outer for-loop are identical but work on different data. This makes this approach suitable for computation using 2-way 32-bit SIMD vector instructions. This technique benefits from 2-way SIMD $32 \times 32 \rightarrow 64$-bit multiplication and matches exactly the 128-bit wide vector instructions as present in many modern computer

Algorithm 4 A parallel radix-2^{32} interleaved Montgomery multiplication algorithm. Except for the computation of q, the arithmetic steps in the outer for-loop, performed by computation 1 and computation 2, are identical. This approach is suitable for 32-bit 2-way SIMD vector instruction units.

Input: $\begin{cases} A, B, M, \mu \text{ such that} \\ \quad A = \sum_{i=0}^{n-1} a_i 2^{32i}, B = \sum_{i=0}^{n-1} b_i 2^{32i}, M = \sum_{i=0}^{n-1} m_i 2^{32i}, \\ \quad 0 \leq a_i, b_i < 2^{32},\ 0 \leq A, B < M,\ 2^{32(n-1)} \leq M < 2^{32n}, \\ \quad 2 \nmid M,\ \mu = M^{-1} \bmod 2^{32}, \end{cases}$

Output: $C \equiv AB2^{-32n} \bmod M$ such that $0 \leq C < M$

Computation 1	**Computation 2**
$d_i = 0$ for $0 \leq i < n$	$e_i = 0$ for $0 \leq i < n$
for $j = 0$ to $n-1$ **do**	**for** $j = 0$ to $n-1$ **do**
	$q \leftarrow ((\mu b_0)a_j + \mu(d_0 - e_0)) \bmod 2^{32}$
$t_0 \leftarrow a_j b_0 + d_0$	$t_1 \leftarrow qm_0 + e_0$ // *where* $t_0 \equiv t_1 \pmod{2^{32}}$
$t_0 \leftarrow \left\lfloor \frac{t_0}{2^{32}} \right\rfloor$	$t_1 \leftarrow \left\lfloor \frac{t_1}{2^{32}} \right\rfloor$
for $i = 1$ to $n-1$ **do**	**for** $i = 1$ to $n-1$ **do**
$p_0 \leftarrow a_j b_i + t_0 + d_i$	$p_1 \leftarrow qm_i + t_1 + e_i$
$t_0 \leftarrow \left\lfloor \frac{p_0}{2^{32}} \right\rfloor$	$t_1 \leftarrow \left\lfloor \frac{p_1}{2^{32}} \right\rfloor$
$d_{i-1} \leftarrow p_0 \bmod 2^{32}$	$e_{i-1} \leftarrow p_1 \bmod 2^{32}$
end for	**end for**
$d_{n-1} \leftarrow t_0$	$e_{n-1} \leftarrow t_1$
end for	**end for**

$$\searrow \qquad \swarrow$$

$$C \leftarrow D - E \quad \text{// where } D = \sum_{i=0}^{n-1} d_i 2^{32i},\ E = \sum_{i=0}^{n-1} e_i 2^{32i}$$

if $C < 0$ **do** $C \leftarrow C + M$ **end if**

return C

architectures. By changing the radix used in Algorithm 4, larger or smaller vector instructions can be supported.

Note that as opposed to a conditional subtraction in Algorithm 1 on page 4 and Algorithm 2 on page 5, Algorithm 4 computes a conditional addition because of the inverted sign of the precomputed Montgomery constant μ. This condition is based on the fact that if $D - E$ is negative (produces a borrow), then the modulus is added to make the result positive.

4.1.1 Expected performance

We follow the analysis of the expected performance from [17], which just considers execution time. The idea is to perform an analysis of the number of arithmetic instructions as an indicator of the expected performance when using a 2-way SIMD implementation instead of a regular (non-SIMD) implementation for the classical ALU. We assume the 2-way SIMD implementation works on pairs of 32-bit words in parallel and has access to a 2-way SIMD $32 \times 32 \rightarrow 64$-bit multiplication instruction. A comparison to a regular implementation is not straightforward since the word-size can be different, the platform might be able to compute multiple instructions in parallel (on different ALUs) and the number of instructions per arithmetic operation might differ. This is why we present

Table 1: A simplified comparison, only stating *the number of word-level instructions required*, to compute the Montgomery multiplication when using a $32n$-bit modulus for a positive even integer n. Two approaches are shown, a sequential one on the classical ALU (Algorithm 2 on page 5) and a parallel one using 2-way SIMD instructions (performing two operations in parallel, cf. Algorithm 4 on the previous page).

Instruction	Classical		2-way SIMD
	32-bit	64-bit	32-bit
`add`	-	-	n
`sub`	-	-	n
`shortmul`	n	$\frac{n}{2}$	$2n$
`muladd`	$2n$	n	-
`muladdadd`	$2n(n-1)$	$n(\frac{n}{2}-1)$	-
`SIMD muladd`	-	-	n
`SIMD muladdadd`	-	-	$n(n-1)$

a simplified comparison based on the number of arithmetic operations when computing Montgomery multiplication using a $32n$-bit modulus for a positive even integer n. We denote by $\texttt{muladd}_w(e, a, b, c)$ and $\texttt{muladdadd}_w(e, a, b, c, d)$ the computation of $e = ab + c$ and $e = ab + c + d$, respectively, for $0 \leq a, b, c, d < 2^w$ (and thus $0 \leq e < 2^{2w}$). These are basic operations on a computer architecture which works on w-bit words. Some platforms have these operations as a single instruction (e.g., on some ARM architectures) while others implement this using separate multiplication and addition instructions (as on the x86 platform). Furthermore, let $\texttt{shortmul}_w(e, a, b)$ denote $e = ab \bmod 2^w$: this $w \times w \to w$-bit multiplication only computes the least significant w bits of the result and is faster than computing a full double-word product on most modern computer platforms.

Table 1 summarizes the expected performance of Algorithm 2 on page 5 and Algorithm 4 on the preceding page in terms of arithmetic operations only. The shifting and masking operations are omitted for simplicity as well as the operations required to compute the final conditional subtraction or addition. When just taking the `muladd` and `muladdadd` instructions into account it becomes clear from Table 1 that the SIMD approach uses exactly half the number of instructions compared to the 32-bit classical implementation and almost twice as many operations compared to the classical 64-bit implementations. However, the SIMD approach requires more operations to compute the value of q every iteration and has various other overheads (e.g., inserting and extracting values from the large vector registers). Hence, when assuming that all the characteristics of the SIMD and classical (non-SIMD) instructions are identical, which is most likely not the case on most platforms, then we expect Algorithm 4 running on a 2-way 32-bit SIMD unit to outperform a classical 32-bit implementation using Algorithm 2 by at most a factor of two while being roughly half as fast as a classical 64-bit implementation.

4.2 A column-wise SIMD approach

A different approach, suitable for computing Montgomery multiplication on architectures supporting 4-way SIMD instructions is outlined by the first chapter author and Kaihara in [13]. This approach is particularly efficient on the Cell Broadband Engine (see a brief introduction to this architecture below), since it was designed for usage on this platform, but can be used on any platform supporting the SIMD instructions used in this approach. This approach differs from the one described in the previous section in that it uses the SIMD instructions to compute the multi-precision arithmetic in parallel, so it works

on a lower level, while the approach from Section 4.1 above computes the arithmetic operations itself sequentially but divides the work into two steps which can be computed concurrently.

4.2.1 The Cell broadband engine

The Cell Broadband Engine (cf. the introductions given by Hofstee [41] and Gschwind [34]), denoted by "Cell" and jointly developed by Sony, Toshiba, and IBM, is a powerful heterogeneous multiprocessor which was released in 2005. The Cell contains a *Power Processing Element*, a dual-threaded Power architecture-based 64-bit processor with access to a 128-bit AltiVec/VMX single instruction, multiple data (SIMD) unit (which is not considered in this chapter). Its main processing power, however, comes from eight *Synergistic Processing Elements* (SPEs). For an introduction to the circuit design see the work by Takahashi et al. [73]. Each SPE consists of a Synergistic Processing Unit (SPU), 256 KB of private memory called Local Store (LS), and a Memory Flow Controller (MFC). To avoid the complexity of sending explicit direct memory access requests to the MFC, all code and data must fit within the LS.

Each SPU runs independently from the others at 3.192GHz and is equipped with a large register file containing 128 registers of 128 bits each. Most SPU instructions work on 128-bit operands denoted as *quadwords*. The instruction set is partitioned into two sets: one set consists of (mainly) 4- and 8-way SIMD arithmetic instructions on 32-bit and 16-bit operands respectively, while the other set consists of instructions operating on the whole quadword (including the load and store instructions) in a single instruction, single data (SISD) manner. The SPU is an asymmetric processor; each of these two sets of instructions is executed in a separate pipeline, denoted by the *even* and *odd* pipeline for the SIMD and SISD instructions, respectively. For instance, the $\{4, 8\}$-way SIMD left-rotate instruction is an even instruction, while the instruction left-rotating the full quadword is dispatched into the odd pipeline. When dependencies are avoided, a single pair consisting of one odd and one even instruction can be dispatched every clock cycle.

One of the first applications of the Cell processor was to serve as the brain of Sony's PlayStation 3 game console. Due to the wide-spread availability of this game console and the fact that one could install and run one's own software this platform has been used to accelerate cryptographic operations [27, 26, 23, 57, 18, 9] as well as cryptanalytic algorithms [69, 16, 14].

4.2.2 Montgomery multiplication on the Cell broadband engine

In this section we outline the approach presented by the first author of this chapter and Kaihara tailored towards the instruction set of the Cell Broadband Engine. Most notably, the presented techniques rely on an efficient 4-way SIMD instruction to multiply two 16-bit integers and add another 16-bit integer to the 32-bit result, and a large register file. Therefore, the approach described here uses a radix $r = 2^{16}$ to divide the large numbers into words that match the input sizes of the 4-SIMD multipliers of the Cell. This can easily be adapted to any other radix size for different platforms with different SIMD instructions.

The idea is to represent integers X in a radix-2^{16} system, i.e., $X = \sum_{i=0}^{n} x_i 2^{16i}$ where $0 \leq x_i < 2^{16}$. However, in order to use the 4-way SIMD instructions of this platform efficiently these 16-bit digits x_i are stored in a 32-bit datatype. The usage of this 32-bit space is to ensure that intermediate values of the form $ab + c$ do not produce any carries since when $0 \leq a, b, c < 2^{16}$ then $0 \leq ab + c < 2^{32}$. Hence, given an odd $16n$-bit modulus N, then a Montgomery residue X, such that $0 \leq X < 2N < 2^{16(n+1)}$, is represented using $s = \left\lceil \frac{n+1}{4} \right\rceil$ vectors of 128 bits. Note that this representation uses roughly twice the number of bits when compared to storing it in a "normal" radix-representation. The single additional

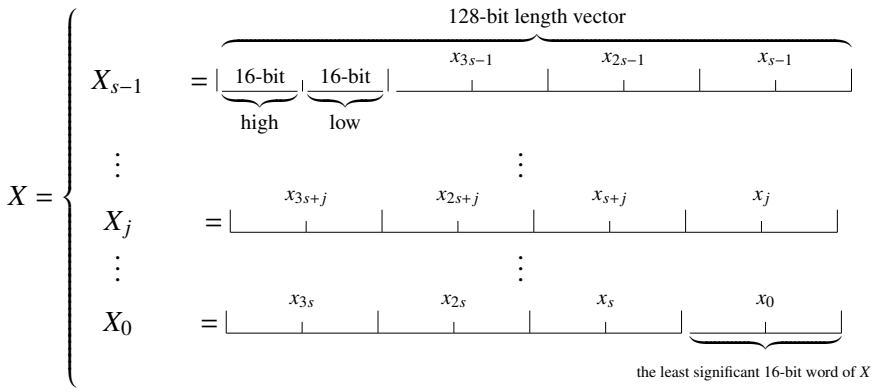

Figure 1: *The 16-bit words x_i of a $16(n+1)$-bit positive integer $X = \sum_{i=0}^{n} x_i 2^{16i} < 2N$ are stored column-wise using $s = \left\lceil \frac{n+1}{4} \right\rceil$ 128-bit vectors X_j on the SPE architecture.*

16-bit word is required because the intermediate accumulating result of Montgomery multiplication can be almost as large as $2N$ (see page 9 in Section 2.4).

The 16-bit digits x_i are placed *column-wise* in the four 32-bit datatypes of the 128-bit vectors. This representation is illustrated in Figure 1. The four 32-bit parts of the j-th 128-bit vector X_j are denoted by

$$X_j = \{X_j[3], X_j[2], X_j[1], X_j[0]\}.$$

Each of the $(n+1)$ 16-bit words x_i of X is stored in the most significant 16 bits of $X_{i \bmod s}[\lfloor \frac{i}{s} \rfloor]$. The motivation for using this column-wise representation is that a division by 2^{16} can be computed efficiently: simply move the digits in vector X_0 "one position to the right", which in practice means a logical 32-bit right shift, and relabeling of the indices such that X_j becomes X_{j-1}, for $1 \leq j < s-1$ and the modified vector X_0 becomes the new X_{s-1}. Algorithm 5 on the next page computes Montgomery multiplication using such a 4-way column-wise SIMD representation.

In each iteration, the indices of the vectors that contain the accumulating partial product U change cyclically among the s registers. In Algorithm 5, each 16-bit word of the inputs X, Y and N 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 contains the replicated values of $-N^{-1} \bmod 2^{16}$ in the lower 16-bit positions of the four 32-bit words. In its most significant 16-bit positions, the temporary vector K stores 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 $A \leftarrow$ **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 (in 4-way SIMD manner) 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, where $U = \sum_{j=0}^{n} u_j 2^{16j}$ and is stored as s vectors of 128-bit $U_{i \bmod s}, U_{i+1 \bmod s}, \ldots, U_{i+n \bmod s}$ (where 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 $U_{(i+j) \bmod s}$ as stated in lines 10 and 18 of Algorithm 5 consists of extracting the higher 16-bit words of these vectors and placing them into the lower

Algorithm 5 Montgomery multiplication algorithm for the Cell

Input:
- N represented by s 128-bit vectors: $N_{s-1}, \ldots, N_0$, such that $2^{16(n-1)} \leq N < 2^{16n}, \quad 2 \nmid N$,
- X, Y each represented by s 128-bit vectors: $X_{s-1}, \ldots, X_0$, and $Y_{s-1}, \ldots, Y_0$, such that $0 \leq X, Y < 2N$,
- μ : a 128-bit vector containing $(-N)^{-1} \pmod{2^{16}}$ replicated in all 4 elements.

Output:
- Z represented by s 128-bit vectors: $Z_{s-1}, \ldots, Z_0$, such that $Z \equiv XY2^{-16(n+1)} \bmod N, \quad 0 \leq Z < 2N$.

1: **for** $j = 0$ to $s - 1$ **do**
2: $\quad U_j = 0$
3: **end for**
4: **for** $i = 0$ to n **do**
5: $\quad$ /* lines 6-9 compute $U = y_i X + U$ */
6: $\quad K = \{y_i,\ y_i,\ y_i,\ y_i\}$
7: $\quad$ **for** $j = 0$ to $s - 1$ **do**
8: $\quad\quad U_{(i+j) \bmod s} = \textbf{muladd}(X_j,\ K,\ U_{(i+j) \bmod s})$
9: $\quad$ **end for**
10: $\quad$ *Carry propagation on* $U_{(i+j) \bmod s}$ *for* $j = 0, \ldots, s - 1$ *(see text)*
11: $\quad$ /* lines 12-13 compute $Q = \mu V \bmod 2^{16}$ */
12: $\quad V = \{u_0,\ u_0,\ u_0,\ u_0\}$
13: $\quad Q = \textbf{shiftleft}(\textbf{mul}(V, \mu), 16)$
14: $\quad$ /* lines 15-17 compute $U = NQ + U$ */
15: $\quad$ **for** $j = 0$ to $s - 1$ **do**
16: $\quad\quad U_{(i+j) \bmod s} = \textbf{muladd}(N_j,\ Q,\ U_{(i+j) \bmod s})$
17: $\quad$ **end for**
18: $\quad$ *Carry propagation on* $U_{(i+j) \bmod s}$ *for* $j = 0, \ldots, s - 1$ *(see text)*
19: $\quad$ /* line 20 computes the division by 2^{16} */
20: $\quad U_{i \bmod s} = \textbf{vshiftright}(U_{i \bmod s}, 32)$
21: **end for**
22: *Carry propagation on* $U_{i \bmod s}$ *for* $i = n + 1, \ldots, 2n + 1$ *(see text)*
23: **for** $j = 0$ to $s - 1$ **do**
24: $\quad Z_j = U_{(n+j+1) \bmod s}$
25: **end for**

16-bit positions of temporary vectors. These vectors are then added to the "next" vector $U_{(i+j+1) \bmod s}$ correspondingly. The operation is carried out for the vectors with indices $j \in \{0, 1, \ldots, s - 2\}$. For $j = s - 1$, the last index, the temporary vector that contains 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 $U_{(i+j) \bmod s}$ in line 22 of Algorithm 5 is performed with 16-bit word extraction and addition, but requires a sequential parsing over the $(n + 1)$ 16-bit words.

Hence, the approach outlined in Algorithm 5 computes Montgomery multiplication by computing the multi-word multiplications using SIMD instructions and representing the integers using a column-wise approach (see Figure 1 on the preceding page). This approach comes at the cost that a single $16n$-bit integer is represented by $128\lceil \frac{n+1}{4} \rceil$ bits: requiring slightly over twice the amount of storage.

Note, however, that an implementation of this technique outperforms the native multi-precision big-number library on the Cell processor by a factor of about 2.5, as summarized in [13].

4.3 Montgomery multiplication using the residue number system representation

The residue number system (RNS) as introduced by Garner [31] and Merrill [53] is an approach, based on the Chinese remainder theorem, to represent an integer as a number of residues modulo smaller (coprime) integers. The advantage of RNS is that additions, subtractions and multiplication can be performed independently and concurrently on these smaller residues. Given an *RNS basis* $\beta_n = \{r_1, r_2, \ldots, r_n\}$, where $\gcd(r_i, r_j) = 1$ for $i \neq j$, the *RNS modulus* is defined as $R = \prod_{i=1}^{n} r_i$. Given an integer $x \in \mathbb{Z}/R\mathbb{Z}$ and the RNS basis β_n, this integer x is represented as an n-tuple $\vec{x} = (x_1, x_2, \ldots, x_n)$ where $x_i = x \bmod r_i$ for $1 \leq i \leq n$. In order to convert an n-tuple back to its integer value one can apply the Chinese remainder theorem (CRT)

$$x = \left(\sum_{i=1}^{n} x_i \left(\left(\frac{R}{r_i} \right)^{-1} \bmod r_i \right) \frac{R}{r_i} \right) \bmod R. \tag{7}$$

Modular multiplication using Montgomery multiplication in the RNS setting has been studied, for instance, by Posch and Posch in [61] and by Bajard, Didier, and Kornerup in [4] and subsequent work. In this section we outline how to achieve this. First note for the application in which we are interested, we can not use the modulus N as the RNS modulus since N is either prime (in the setting of elliptic curve cryptography) or a product of two large primes (when using RSA). When computing the Montgomery reduction one has to perform arithmetic modulo the Montgomery radix. One possible approach is to use the RNS modulus $R = \prod_{i=1}^{n} r_i$ as the Montgomery radix. This has the advantage that whenever one computes with integers represented in this residue number system they are reduced modulo R implicitly. However, since we are performing arithmetic in the ring $\mathbb{Z}/R\mathbb{Z}$ this means that division by R, as required in the Montgomery reduction, is not well-defined.

One way this problem can be circumvented is by introducing an auxiliary basis $\beta'_n = \{r'_1, r'_2, \ldots, r'_n\}$ with auxiliary RNS modulus $R' = \prod_{i=1}^{n} r'_i$ such that

$$\gcd(R', R) = \gcd(R, N) = \gcd(R', N) = 1$$

(and both R and R' are larger than $4N$). The idea is to convert the intermediate result represented in β_n to the auxiliary basis β'_n and perform the division by R here (since R and R' are coprime this inverse exists).

The concept of base-extension, converting the representation from one base to another, is by Szabo and Tanaka in [71] (but see also the work by Gregory and Matula in [33]). Methods are either based on the CRT as used by Shenoy and Kumaresan [67], Posch and Posch [60], and Kawamura, Koike, Sano, and Shimbo [46] or on an intermediate representation denoted by a *mixed radix system* as presented in Szabo and Tanaka in [71]. Carefully selected RNS bases can significantly impact the performance in practice as shown by Bajard, Kaihara, and Plantard in [3] and Bigou and Tisserand in [8]. Another RNS approach is presented by Phillips, Kong, and Lim [59].

With these two RNS bases defined we can compute the Montgomery product modulo N. Let $R = \prod_{i=1}^{n} r_i$ and $R' = \prod_{i=1}^{n} r'_i$ be the two RNS moduli for RNS basis β_n and β'_n respectively. The representations of N in β_n and β'_n are denoted by $\vec{N}$ and $\vec{N'}$. Let $A, B \in \mathbb{Z}/N\mathbb{Z}$ be represented in both RNS bases: β_n ($\vec{a}$ and $\vec{b}$) and β'_n ($\vec{a'}$ and $\vec{b'}$). Then we can compute the Montgomery product in both these RNS bases using the following steps.

1. Compute the product of A and B in both RNS bases β_n and β'_n:

$$\vec{d} = \vec{a} \cdot \vec{b}, \quad \text{where} \quad d_i = a_i b_i \bmod r_i, \text{ for } 1 \le i \le n,$$
$$\vec{d'} = \vec{a'} \cdot \vec{b'} \quad \text{where} \quad d'_i = a'_i b'_i \bmod r'_i \text{ for } 1 \le i \le n.$$

2. Compute $((ab)(-N^{-1} \bmod R) \bmod R)$. This is realized by computing $\vec{q} = -\vec{N}^{-1} \cdot \vec{d}$ in basis β_n as $q_i = -N_i^{-1} d_i \bmod r_i$ for $1 \le i \le n$.

3. Convert $\vec{q}$ in basis β_n to $\vec{q'}$ in basis β'_n (for instance using Equation (7)).

4. Compute the final part $\vec{c'} = (\vec{d'} + \vec{q'} \cdot \vec{N'}) \cdot \vec{R}^{-1}$ of the Montgomery multiplication (including the division by R) in basis β'_n by computing $c'_i = (d'_i + q'_i N'_i) r_i^{-1} \bmod r'_i$ for $1 \le i \le n$.

5. Convert $\vec{c'}$ in basis β'_n to $\vec{c}$ in basis β_n (for instance using Equation (7)).

After step 5 we have $\vec{c} = \{c_1, c_2, \ldots, c_n\}$ and $\vec{c'} = \{c'_1, c'_2, \ldots, c'_n\}$ such that

$$c_i \equiv \left(abR^{-1} \bmod N\right) \bmod r_i \quad \text{and} \quad c'_i \equiv \left(abR^{-1} \bmod N\right) \bmod r'_i.$$

This approach has been used to implement asymmetric cryptography on highly parallel computer architectures like graphics processing units (e.g., as in [5, 56, 72, 38, 2]). The results presented in these papers show that when *multiple* Montgomery multiplications are computed concurrently using RNS the latency can be reduced significantly while the throughput is increased (compared to computation on a multi-core CPU) when computing with thousands of threads on the hundreds of cores on a graphics processing unit. This highlights the potential of using graphics cards as cryptographic accelerators when large batches of work require processing (and a low latency is required). The process is illustrated in the example below.

Example 4: RNS Montgomery multiplication

Let $A = 42, B = 17$ and $N = 67$. In this example we show how to compute the Montgomery product $ABR^{-1} \bmod N$ using a residue number system. Let us first define the two coprime RNS bases

$$\beta_3 = \{3, 7, 13\} \quad \text{with RNS modulus} \quad R = 3 \cdot 7 \cdot 13 = 273,$$
$$\beta'_3 = \{5, 11, 17\} \quad \text{with RNS modulus} \quad R' = 5 \cdot 11 \cdot 17 = 935,$$

such that both RNS moduli are larger than $4N$. Recall that R plays the role of both the RNS modulus as well as of the Montgomery radix. First, we need to represent the inputs A and B in both RNS bases: $A = 42$ is represented as

- $\vec{a} = \{42 \bmod 3, 42 \bmod 7, 42 \bmod 13\} = \{0, 0, 3\}$ in basis β_3,
- $\vec{a'} = \{42 \bmod 5, 42 \bmod 11, 42 \bmod 17\} = \{2, 9, 8\}$ in basis β'_3,

and $B = 17$ is represented as

- $\vec{b} = \{17 \bmod 3, 17 \bmod 7, 17 \bmod 13\} = \{2, 3, 4\}$ in basis β_3,
- $\vec{b'} = \{17 \bmod 5, 17 \bmod 11, 17 \bmod 17\} = \{2, 6, 0\}$ in basis β'_3.

Furthermore, we need to represent the precomputed Montgomery constant $\mu = -N^{-1} \bmod R = -67^{-1} \bmod 273 = 110$ in basis β_3

$$\vec{\mu} = \{110 \bmod 3, 110 \bmod 7, 110 \bmod 13\} = \{2, 5, 6\},$$

as well as the modulus N in basis β_3'

$$\vec{N'} = \{67 \bmod 5, 67 \bmod 11, 67 \bmod 17\} = \{2, 1, 16\}.$$

Compute the first step: the product of these two numbers in both bases. This can be done in parallel for all the individual moduli

- $\vec{d} = \vec{a} \cdot \vec{b} = \{0 \cdot 2 \bmod 3, 0 \cdot 3 \bmod 7, 3 \cdot 4 \bmod 13\} = \{0, 0, 12\}$,
- $\vec{d'} = \vec{a'} \cdot \vec{b'} = \{2 \cdot 2 \bmod 5, 9 \cdot 6 \bmod 11, 8 \cdot 0 \bmod 17\} = \{4, 10, 0\}$.

Next, compute $\vec{q} = \vec{d} \cdot \vec{\mu}$ in basis β_3

$$\vec{q} = \vec{d} \cdot \vec{\mu} = \{0 \cdot 2 \bmod 3, 0 \cdot 5 \bmod 7, 12 \cdot 6 \bmod 13\} = \{0, 0, 7\}.$$

Change the representation of q: convert $\vec{q} = \{0, 0, 7\}$, which is represented in basis β_3, to $\vec{q'}$ which is represented in basis β_3'. This can be done by first converting $\vec{q}$ back to its integer representation following Equation (7) on page 21

$$\begin{aligned} q = \; & \left(0 \cdot \left(\left(\tfrac{273}{3}\right)^{-1} \bmod 3\right) \tfrac{273}{3} + \right. \\ & 0 \cdot \left(\left(\tfrac{273}{7}\right)^{-1} \bmod 7\right) \tfrac{273}{7} + \\ & \left. 7 \cdot \left(\left(\tfrac{273}{13}\right)^{-1} \bmod 13\right) \tfrac{273}{13}\right) \bmod 273 \\ = \; & 0 + 0 + 7 \cdot 5 \cdot 21 \bmod 273 = 189. \end{aligned}$$

From q obtain $\vec{q'} = \{189 \bmod 5, 189 \bmod 11, 189 \bmod 17\} = \{4, 2, 2\}$. The final step computes the result c in basis β_3' as $\vec{c'} = (\vec{d'} + \vec{q'} \cdot \vec{N'}) \cdot \vec{R}^{-1}$:

$$\begin{aligned} \vec{c'} \; & = (\{4, 10, 0\} + \{4, 2, 2\} \cdot \{2, 1, 16\}) \cdot \{2, 5, 1\} \\ & = \{4, 5, 15\}. \end{aligned}$$

When converting this to the integer representation we obtain

$$\begin{aligned} c = \; & \left(4 \cdot \left(\left(\tfrac{935}{5}\right)^{-1} \bmod 5\right) \tfrac{935}{5} + \right. \\ & 5 \cdot \left(\left(\tfrac{935}{11}\right)^{-1} \bmod 11\right) \tfrac{935}{11} + \\ & \left. 15 \cdot \left(\left(\tfrac{935}{17}\right)^{-1} \bmod 17\right) \tfrac{935}{17}\right) \bmod 935 \\ = \; & 374 + 170 + 440 \bmod 935 = 49. \end{aligned}$$

This is indeed correct since

$$\begin{aligned} c &= ABR^{-1} \bmod N \\ &= 42 \cdot 17 \cdot (3 \cdot 7 \cdot 13)^{-1} \bmod 67 \\ &= 49. \end{aligned}$$

Acknowledgements

The authors want to thank Robert Granger, Arjen K. Lenstra, Paul Leyland, and Colin D. Walter for feedback and suggestions. Peter Montgomery's authorship is *honoris causa*; all errors and inconsistencies in this chapter are the sole responsibility of the first author.

References

[1] T. Acar and D. Shumow. Modular reduction without pre-computation for special moduli. Technical report, Microsoft Research, 2010. (Cited on page 12.)

[2] S. Antão, J.-C. Bajard, and L. Sousa. RNS-based elliptic curve point multiplication for massive parallel architectures. *The Computer Journal*, 55(5):629–647, 2012. (Cited on page 22.)

[3] J. Bajard, M. E. Kaihara, and T. Plantard. Selected RNS bases for modular multiplication. In J. D. Bruguera, M. Cornea, D. D. Sarma, and J. Harrison, editors, *19th IEEE Symposium on Computer Arithmetic – ARITH 2009*, pages 25–32. IEEE Computer Society, 2009. (Cited on page 21.)

[4] J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS montgomery modular multiplication algorithm. *IEEE Trans. Computers*, 47(7):766–776, 1998. (Cited on page 21.)

[5] J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 53(6):769–774, June 2004. (Cited on page 22.)

[6] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, Heidelberg, Aug. 1987. (Cited on page 2.)

[7] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, Heidelberg, Apr. 2006. (Cited on page 11.)

[8] K. Bigou and A. Tisserand. Single base modular multiplication for efficient hardware RNS implementations of ECC. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 123–140. Springer, Heidelberg, Sept. 2015. (Cited on page 21.)

[9] J. W. Bos. High-performance modular multiplication on the Cell processor. In M. A. Hasan and T. Helleseth, editors, *Workshop on the Arithmetic of Finite Fields – WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 7–24. Springer, 2010. (Cited on pages 14 and 18.)

[10] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 194–210. Springer, Heidelberg, May 2013. (Cited on pages 11 and 12.)

[11] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 331–348. Springer, Heidelberg, Aug. 2013. (Cited on page 12.)

[12] J. W. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *J. Cryptographic Engineering*, 6(4):259–286, 2016. (Cited on page 12.)

[13] J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics – PPAM 2009*, volume 6067 of *Lecture Notes in Computer Science*, pages 477–485. Springer, Heidelberg, 2010. (Cited on pages 17 and 21.)

[14] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012. (Cited on page 18.)

[15] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In E. Antelo, D. Hough, and P. Ienne, editors, *IEEE Symposium on Computer Arithmetic – ARITH-20*, pages 213–221. IEEE Computer Society, 2011. (Cited on page 11.)

[16] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on cell CPUs. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT 10: 3rd International Conference on Cryptology in Africa*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer, Heidelberg, May 2010. (Cited on page 18.)

[17] J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 471–489. Springer, Heidelberg, Aug. 2014. (Cited on pages 13, 14, 15, and 16.)

[18] J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 279–293. Springer, Heidelberg, Aug. 2010. (Cited on page 18.)

[19] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer, Heidelberg, Aug. 1994. (Cited on page 2.)

[20] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010. (Cited on page 12.)

[21] E. F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology – CRYPTO'82*, pages 51–60. Plenum Press, New York, USA, 1982. (Cited on page 14.)

[22] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996. (Cited on page 6.)

[23] H.-C. Chen, C.-M. Cheng, S.-H. Hung, and Z.-C. Lin. Integer number crunching on the Cell processor. *International Conference on Parallel Processing*, pages 508–515, 2010. (Cited on page 18.)

[24] J. Chung and M. A. Hasan. Montgomery reduction algorithm for modular multiplication using low-weight polynomial form integers. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 230–239. IEEE Computer Society, 2007. (Cited on page 12.)

[25] S. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966. (Cited on page 6.)

[26] N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In B. Preneel, editor, *AFRICACRYPT 09: 2nd International Conference on Cryptology in Africa*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, Heidelberg, June 2009. (Cited on page 18.)

[27] N. Costigan and M. Scott. 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/2007/061`. (Cited on page 18.)

[28] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT'92*, volume 658 of *Lecture Notes in Computer Science*, pages 183–193. Springer, Heidelberg, May 1993. (Cited on pages 12 and 14.)

[29] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. Damgård, editor, *Advances in Cryptology – EUROCRYPT'90*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer, Heidelberg, May 1991. (Cited on pages 6, 7, and 8.)

[30] M. Fürer. Faster integer multiplication. In D. S. Johnson and U. Feige, editors, *39th Annual ACM Symposium on Theory of Computing*, pages 57–66. ACM Press, June 2007. (Cited on page 6.)

[31] H. L. Garner. The residue number system. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 146–153, New York, NY, USA, 1959. ACM. (Cited on page 21.)

[32] R. Granger and A. Moss. Generalised Mersenne numbers revisited. *Math. Comput.*, 82(284):2389–2420, 2013. (Cited on page 12.)

[33] R. T. Gregory and D. W. Matula. Base conversion in residue number systems. In T. R. N. Rao and D. W. Matula, editors, *3rd IEEE Symposium on Computer Arithmetic – ARITH 1975*, pages 117–125. IEEE Computer Society, 1975. (Cited on page 21.)

[34] M. Gschwind. The Cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35:233–262, 2007. (Cited on page 18.)

[35] S. Gueron and V. Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015. (Cited on page 13.)

[36] G. Hachez and J.-J. Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 293–301. Springer, Heidelberg, Aug. 2000. (Cited on page 9.)

[37] M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. `http://eprint.iacr.org/2012/309`. (Cited on page 12.)

[38] O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In B. Preneel, editor, *AFRICACRYPT 09: 2nd International Conference on Cryptology in Africa*, volume 5580 of *Lecture Notes in Computer Science*, pages 350–367. Springer, Heidelberg, June 2009. (Cited on page 22.)

[39] L. Hars. Long modular multiplication for cryptographic applications. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 45–61. Springer, Heidelberg, Aug. 2004. (Cited on page 12.)

[40] K. Hensel. *Theorie der algebraischen Zahlen*. Tuebner, Leipzig, 1908. (Cited on page 2.)

[41] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *High-Performance Computer Architecture – HPCA 2005*, pages 258–262. IEEE, 2005. (Cited on page 18.)

[42] K. Iwamura, T. Matsumoto, and H. Imai. Systolic-arrays for modular exponentiation using Montgomery method (extended abstract) (rump session). In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT'92*, volume 658 of *Lecture Notes in Computer Science*, pages 477–481. Springer, Heidelberg, May 1993. (Cited on page 14.)

[43] M. E. Kaihara and N. Takagi. Bipartite modular multiplication. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 201–210. Springer, Heidelberg, Aug. / Sept. 2005. (Cited on page 14.)

[44] A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145(2):293–294, 1962. Translation in Physics-Doklady **7**, pp. 595–596, 1963. (Cited on page 6.)

[45] E. Käsper. Fast elliptic curve cryptography in OpenSSL. In G. Danezis, S. Dietrich, and K. Sako, editors, *FC 2011 Workshops*, volume 7126 of *Lecture Notes in Computer Science*, pages 27–39. Springer, Heidelberg, Feb. / Mar. 2012. (Cited on page 11.)

[46] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower architecture for fast parallel Montgomery multiplication. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*,

volume 1807 of *Lecture Notes in Computer Science*, pages 523–538. Springer, Heidelberg, May 2000. (Cited on page 21.)

[47] M. Knežević, F. Vercauteren, and I. Verbauwhede. Speeding up bipartite modular multiplication. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields – WAIFI*, volume 6087 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 2010. (Cited on page 12.)

[48] D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997. (Cited on page 2.)

[49] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. (Cited on page 11.)

[50] Çetin Kaya. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, 1998. (Cited on page 10.)

[51] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, Heidelberg, Aug. 1999. (Cited on pages 2 and 8.)

[52] A. K. Lenstra. Generating RSA moduli with a predetermined portion. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 1–10. Springer, Heidelberg, Oct. 1998. (Cited on page 12.)

[53] R. D. Merrill. Improving digital computer performance using residue number theory. *Electronic Computers, IEEE Transactions on*, EC-13(2):93–101, April 1964. (Cited on page 21.)

[54] V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – CRYPTO'85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, Aug. 1986. (Cited on page 11.)

[55] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985. (Cited on pages 1, 4, 6, and 7.)

[56] A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In S. D. Galbraith, editor, *11th IMA International Conference on Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 364–383. Springer, Heidelberg, Dec. 2007. (Cited on page 22.)

[57] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In S. Hong and T. Iwata, editors, *Fast Software Encryption – FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, Heidelberg, Feb. 2010. (Cited on page 18.)

[58] D. Page and N. P. Smart. Parallel cryptographic arithmetic using a redundant Montgomery representation. *IEEE Trans. Computers*, 53(11):1474–1482, 2004. (Cited on page 14.)

[59] B. J. Phillips, Y. Kong, and Z. Lim. Highly parallel modular multiplication in the residue number system using sum of residues reduction. *Appl. Algebra Eng. Commun. Comput.*, 21(3):249–255, 2010. (Cited on page 21.)

[60] K. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50(2):93–104, 1993. (Cited on page 21.)

[61] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):449–454, 1995. (Cited on page 21.)

[62] Q. Pu and X. Zhao. Montgomery exponentiation with no final comparisons: Improved results. In *Pacific-Asia Conference on Circuits, Communications and Systems*, pages 614–616. IEEE, 2009. (Cited on page 9.)

[63] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978. (Cited on pages 7 and 9.)

[64] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971. (Cited on page 6.)

[65] H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim. Montgomery modular multiplication on ARM-NEON revisited. In J. Lee and J. Kim, editors, *Information Security and Cryptology - ICISC 2014*, volume 8949 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2015. (Cited on pages 13 and 14.)

[66] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In E. E. S. Jr., M. J. Irwin, and G. A. Jullien, editors, *11th Symposium on Computer Arithmetic*, pages 252–259. IEEE Computer Society, 1993. (Cited on pages 2 and 9.)

[67] A. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *Computers, IEEE Transactions on*, 38(2):292–297, 1989. (Cited on page 21.)

[68] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99–39, Centre for Applied Cryptographic Research, University of Waterloo, 1999. (Cited on page 11.)

[69] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, Heidelberg, Aug. 2009. (Cited on page 18.)

[70] A. Svoboda. An algorithm for division. *Information processing machines*, 9(25-34):28, 1963. (Cited on page 12.)

[71] N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967. (Cited on page 21.)

[72] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer, Heidelberg, Aug. 2008. (Cited on page 22.)

[73] O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, S. Onish, J. Pille, and J. Silberman. The circuit design of the synergistic processor element of a Cell processor. In *International conference on Computer-aided design – ICCAD 2005*, pages 111–117. IEEE Computer Society, 2005. (Cited on page 18.)

[74] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3(4):714–716, 1963. (Cited on page 6.)

[75] U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`. (Cited on page 11.)

[76] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, Mar. 1993. (Cited on page 14.)

[77] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, Oct. 1999. (Cited on page 9.)

[78] C. D. Walter. Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli. In B. Preneel, editor, *Topics in Cryptology – CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 30–39. Springer, Heidelberg, Feb. 2002. (Cited on page 9.)

[79] C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In D. Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 192–207. Springer, Heidelberg, Apr. 2001. (Cited on pages 9 and 10.)