3
Hardware Aspects of Montgomery Modular Multiplication[*]

Colin D. Walter
Royal Holloway, University of London,
Egham, Surrey, TW20 0EX, United Kingdom

Abstract

This chapter compares Peter Montgomery's modular multiplication method with traditional techniques for suitability on hardware platforms. It also covers systolic array implementations and side channel leakage.

3.1 Introduction and Summary

This chapter looks at the hardware implementation of Peter Montgomery's *Modular Multiplication without Trial Division* [39]. Such dedicated hardware is used primarily for arithmetic over the rational integers $\mathbb{Z}$ with a very large modulus N, including the prime field $GF(p)$ case when $N = p$ is prime. For simplicity, it is assumed that all the arithmetic here is over the integers. There are increasingly important applications over large finite fields. However, apart from simpler carry propagation when the field characteristic is very small, the main issues covered here have similar solutions. The interested reader might start by consulting [28, 46, 3, 1, 35] for this case.

Because of the overhead of translating to and from the Montgomery domain ([5], §2.2), use of Montgomery's method is, for the most part, in cryptography where exponentiation is a central operation and the cost of the translation can be amortised over all the modular multiplications in the exponentiation. Diffie-Hellman key exchange, RSA, DSA, ElGamal encryption and their elliptic curve equivalents are chief among the applications [11, 45, 40, 14, 38, 27, 21]. This justifies the concentration on instances over $\mathbb{Z}$ where the modulus N is a large integer, typically with 1024 or more bits. A consequence of this is that, with representations in radix r, processing a regular digit position in the long number arithmetic should be as efficient as possible and, in the notation of [5], Alg$^{\mathrm{m}}$. 2,

[*]This material has been published as Chapter 3 in [4]. It makes a number of references to Chapter 2, which is also available as [5].

the determination of the multiple q of N which has to be added to, or subtracted from, the running total[1] C should not slow down such processing.

Although not used exclusively, Montgomery's method should be the predominant one for modular multiplication in the applications above. We will consider the reasons for its already widespread adoption by comparing it with classical techniques, demonstrating how it fits in well with standard word sizes whereas standard methods suffer from widespread contamination by over-large digits[2]. The chapter is structured to provide an overview of the main acceleration techniques for hardware modular multiplication and for each of these we deduce that Montgomery multiplication is better than, or at least as good as, classical techniques.

Many of the implementations of interest to us occur in smart cards where side channel leakage is a significant threat. Consequently, the final part of this chapter is a study of the security issues that arise from the use of Montgomery's method and how they can be mitigated.

At the other end of a secure transaction involving a smart card there is probably a server performing a large number of simultaneous cryptographic processes for many secure transactions, and so requiring the capability of very high throughput. This can be achieved using a systolic array and, for such a context, Montgomery's algorithm is really the only sensible choice.

The target hardware could be one of a wide variety of different possibilities. Among these there are small smart-card cryptographic co-processors with a single 8- or 16-bit multiplier, ARM-based processors, single core and multi-core Single Instruction Multiple Data (SIMD) processors with pipelined multipliers, systolic arrays with substantial processing power in each processing element, and Application Specific Integrated Circuits (ASICs) which may process all bits of the modulus in parallel. The variety means that allowance must be made for all possible ways of implementing the basic arithmetic operations which appear in a description of Montgomery's modular multiplication method, particularly addition and scalar multiplication, and indeed including consideration of number representations (such as redundant ones) which are alternatives to the usual binary. The low level programming of these architectures was covered in the previous chapter [5]. Here we consider the hardware itself, particularly the size of the digit$\times$digit multiplier, how to increase clock speed, and the effects of bus width and communications.

[1]As in Alg$^{\mathrm{m}}$. 0, P is used here in this chapter rather than the C of [5].

[2] Where classical methods are still used, the reason is often ascribed to the difficulty of translating inputs to the Montgomery domain, i.e. mapping A to $\tilde{A} \equiv AR \bmod N$ where R is the Montgomery radix – see [5], §2.2. This requires computing and storing $R^2 \bmod N$. However, if the public exponent E is known and the exponentiation is to the power D where $DE \equiv 1 \bmod \phi(N)$, then this reason is spurious. Specifically, start by computing 1^{E-1} using Montgomery multiplication. Then $U \equiv R^{2-E} \bmod N$ is obtained if $E > 2$. This generally requires only a small number of multiplications as E is typically a small Fermat prime. Next Montgomery-multiply A and U to obtain $AR^{1-E} \bmod N$. Raising this to the power $D > 1$ using Montgomery multiplication yields $(AR^{1-E})^D R^{1-D} \equiv A^D R^{D(1-E)+(1-D)} \equiv A^D R \bmod N$. The usual post-processing of Montgomery-multiplying by 1 then yields $A^D \bmod N$, as required, and Montgomery representations have been avoided.

3.2 Historical Remarks

By the late 1980s, RSA encryption had been in the public domain for a dozen or so years and was regarded as more secure for encryption than DES because academic mathematicians had failed to make significant progress in factoring large numbers despite considerable effort and there were mathematically unfounded suspicions than IBM had built some weaknesses into DES that would allow the NSA to crack ciphertexts more easily than could the general public. Consequently, there was great interest in implementing RSA not just for military purposes but also to provide better security in the banking sector where card payment at point of sale was already very well established using hand-written signatures for verification but only plain text data transmission. There was an expanding market which made it commercially viable to develop point of sale terminals containing RSA for secure data transmission, and which was shortly going to expand to smart cards. Then, "smart" cards were used only in prepaid telephone applications and contained essentially no security. The market for them was yet to develop.

At that time RISC-based processors such as the ARM2 with its 32-bit multiplier did not yet yield acceptable performance for software implementations of RSA. ASIC cryptographic co-processors with a full battery of hardware acceleration techniques and the most efficient algorithms were required to yield the necessary encryption speeds of at most several seconds. (A fraction of a second is required nowadays.) Peter Montgomery's then recently published work [39] on modular multiplication played, and continues to play, a significant role in achieving user-acceptable encryption and signing speeds, perhaps now more than ever with the proliferation of RFID tags and embedded cryptographic devices in an increasingly security-conscious world.

3.3 Montgomery's Novel Modular Multiplication Algorithm

Modular Multiplication without Trial Division was published in 1985 [39] and provided a more efficient way of implementing the necessary arithmetic historically at exactly the right time. Indeed, it arose out of the need to speed up RSA. The details are given in Algorithms 1 and 2 of the previous chapter [5], and the reader is encouraged to renew familiarity with the methods and the notation provided there. For convenience, the second of these algorithms is reproduced here as Algorithm 0.

As the title of the paper suggests, Montgomery's technique avoids the delay caused by the usual trial-and-error method of determining which multiple q of the modulus N needs to be subtracted from partial products P, replacing that trial division with an exact digit$\times$digit multiplication. Readers will be familiar with the schoolbook method of long division in which the first few decimal digits of N and P are used in a trial-and-error manner to determine the first digit q of the quotient P/N. The first estimate q' is used to compute $P-q'N$ and it is

Algorithm 0 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 pre-computed Montgomery constant $\mu = -N^{-1} \bmod r$. The modulus N is such that $r^{n-1} \le N < r^n$ and r and N are co-prime.

```
Input: $A = \sum_{i=0}^{n-1} a_i r^i, B, N$ such that $0 \le a_i < r,\ 0 \le A, B < R$.
Output: $P \equiv (AB)R^{-1} \bmod N$ such that $0 \le P < N$.
 1: $P \leftarrow 0$
 2: for $i = 0$ to $n-1$ do
 3:     $P \leftarrow P + a_i B$
 4:     $q \leftarrow \mu P \bmod r$
 5:     $P \leftarrow (P + qN)/r$
 6: end for
 7: if $P \ge N$ then
 8:     $P \leftarrow P - N$
 9: end if
10: return $P$
```

then refined to the correct value q depending on whether the result is less than 0 or $\ge N$. Similarly, with hardware rather than human processors, the base is a power of 2 such as 2^{16} or 2^{32} rather than 10, but the problem is otherwise identical for the multi-digit numbers appearing in RSA. As explained in the previous chapter [5], Peter Montgomery's method replaces this usual hit-or-miss technique with an exact calculation which depends only on the lowest digit of the partial product and the modulus – see Alg$^{\mathrm{m}}$. 0. Although there is the penalty of scaling inputs and outputs to or from their Montgomery representations in the Montgomery domain ([5], §2.2), there are other advantages. For example, the exact process at each digit iteration means much more straightforward firmware which is both easier to verify and takes up less ROM. Furthermore, subtraction can ususally be completely eliminated. This has the significant commercial value of much less likelihood of company critical errors in the implementation as well as shorter time-to-market and smaller die size.

3.4 Standard Acceleration Techniques

The reason for emphasising the commercial and implementation advantages of Peter Montgomery's algorithm is that the mathematical advantages are often over-stated in the literature. Indeed, with the use of appropriate implementation techniques, the overall complexity seems to be asymptotically exactly the same as for classical modular reduction techniques when the argument size increases. The main disadvantage is that, unlike Montgomery's algorithm, traditional techniques are ill-matched to standard hardware word sizes, bus widths and multiplier sizes. However, before reaching that conclusion, let us review all the normal hardware acceleration techniques applicable in the traditional computation of $A{\cdot}B \bmod N$ and consider their analogues in the Montgomery computation of

ABR^{-1} mod N. Most of these were enumerated in [13] and they are covered in detail in the following sub-sections. This comparison will show where the main differences are in the complexity of implementing the arithmetic components. Chief among these is the determination and use of the quotient digit q which is covered in §3.7. There are usually significant differences between the two algorithms for this aspect when the hardware multiplier is already provided rather than custom-built: most digit-level multipliers are ill-adapted to processing the slightly larger digits encountered in classical algorithms.

At the cutting edge of the fastest or most efficient implementations, it is the fine detail which becomes important, such as the critical path length, the number of load and write operations, loop unrolling, the types of register and memory used, and the area devoted to wiring. The interested reader is referred to the vast research literature such as [36] for this more detailed level of coverage. Further, we omit consideration of enhancements which make modular squaring faster than modular multiplication. Instead, we confine ourselves to what is sufficient in a normal commercial setting, ignoring also minor costs such as those for loop establishment and index calculations. Several of the acceleration techniques are already covered in the previous chapter [5], there being no clear boundary between what is hardware and what is software. Some techniques involved the choice of moduli with particular properties such as sparseness (of non-zero bits) – see [5], §3. In this chapter it is assumed that the hardware needs to process *any* modulus, and so inputs may need to be transformed first in order to benefit from the previously mentioned techniques. Such transformations are described in detail.

Although "efficiency" is measured here in terms of *Time* or *Area*× *Time*, in many portable and some RFID devices the predominant issue is energy consumption. For cryptographic applications, the choice of cryptosystem combined with the use of dedicated rather than general purpose hardware is critical in reducing the energy used. At the hardware level the use of fields with small characteristic saves power in a multiplier by reducing the switching activity due to propagating carries. Secondly, careful algorithm design to reduce memory access is also very helpful, and is applicable to time efficiency as well. Thirdly, one can employ fast multiplication techniques such as Karatsuba-Ofman [24, 18, 25]. However, for the most efficient use of the space available in this chapter, the discussion of energy efficiency is limited to this paragraph!

3.5 Shifting the Modulus N

The Classical Algorithm

When using the traditional algorithm (see Alg. 1) it is advisable to shift the modulus N up so that its most significant bit always has the same position in the hardware (normally the top bit of a register). This left alignment allows moduli with different numbers of bits to be processed in a more standard way, reducing combinational logic and ROM code. Identical adjusting shifts are made

Algorithm 1 A radix-r interleaved classical modular multiplication algorithm to compute $A \cdot B \bmod N$.

```
Input: $A = \sum_{i=0}^{n-1} a_i r^i, B, N$ such that $0 \le a_i < r,\ 0 \le A, B < N < r^n$.
Output: $P \equiv A \cdot B \bmod N$ such that $0 \le P < N$.
 1: $P \leftarrow 0$
 2: for $i = n-1$ downto 0 do
 3:     $P \leftarrow rP + a_i \cdot B$
 4:     $q \approx \lfloor P/N \rfloor$      (a lower approx$^{\rm n}$ to the greatest integer $\le P/N$)
 5:     $P \leftarrow P - q \cdot N$
 6: end for
 7: while $P \ge N$ do
 8:     $P \leftarrow P - N$
 9: end while
10: return $P$
```

to A or B and reversed in the output. If S is the power of 2 corresponding to the number of bit positions in the shift then the new values to use for A and N are AS and NS. So the correct result is obtained by computing

$$(A \cdot B) \bmod N = S^{-1}((AS \cdot B) \bmod NS)$$

in which S^{-1} is the shift back down.

For efficiency, the quotient digit q in line 4 of Alg. 1 is generally computed from only the top bits of P and N. So, if n' is the value of the top bits of the register containing N which are used for this, then the shift up means that n' has a more limited range of values. In particular, the top bit of n' being non-zero keeps its value well away from zero and so provides an upper limit to q, whose definition includes a division by n' or $n'{+}1$. We look at the definition and computation of q in more detail in §3.7.

Montgomery

For completeness, it is worth observing that there is an analogue of this classical technique for Montgomery multiplication which is slightly more than just the right alignment of the modulus and operands. It makes the Montgomery method *generally* applicable instead of only in circumstances where the modulus is prime to the radix r of the number representation.

If we take N to be a general modulus as in the classical case, then any common factor S with the computation base r needs to be removed before applying Montgomery's reduction technique. Replacing N by $N' = S^{-1}N$ is the analogous shifting down process. With binary representations, S is a power of 2, N' is odd, and the lowest non-zero bit of N' is the bottom one in its register. The Montgomery multiplication is then done modulo N'.

For RSA and ECC analogues, N is odd and certainly prime to any conceivable computing base. Hence N does not generally require to be shifted down in this way, and the normal right alignment corresponds to no shift, i.e. $S = 1$. Nevertheless, with this action Montgomery can be used for general moduli.

To complete the computation after the shift adjustment requires application of the Chinese Remainder Theorem (CRT) for the co-prime moduli N' and S. This is best done after exiting from the Montgomery domain ([5], §2.2). Using Garner's formula [17] we have

$$(A{\cdot}B) \bmod N = (A{\cdot}B) \bmod N' + N'T$$

where

$$T = (N'^{-1} \bmod S)((A{\cdot}B) \bmod S - (A{\cdot}B) \bmod N') \bmod S$$

Of course, here T is probably calculated very easily even in the most general setting so that the overhead is low. This is because mod S selects the lowest bits of a number to base r. In particular, if S divides the computation base r, then T is a single digit computed from the lowest digit of each of A, B and N'. A similar process of adjusting the modulus can be applied to any modular arithmetic, not just to modular multiplication, thereby enabling the Montgomery modular reduction technique to be used just as easily in a general setting.

From here onwards it is assumed that the shifting described in this section is performed. Consequently, the highest or lowest bit of N, as appropriate, is assumed to be 1 in all future discussion.

3.6 Interleaving Multiplication Steps with Modular Reduction

As already noted in [5], §2.1, instead of computing the product $A{\cdot}B$ first and then performing the reduction modulo N, it is advisable to interleave modular subtractions at each shift-and-add step during the normal calculation of the product. This means the partial product stays roughly the size of the modulus N rather than being as large as $A{\cdot}B$. This saves register space. The term *integrated* is often used for this technique, as opposed to *separated*, which is when the modular reduction is performed after the multiplication.

However, the product can be computed in two main ways. The easier to organise and more uniform is the *operand scanning* technique in which the partial product P is increased by a_iB, with i being decremented or incremented at each iteration step according to whether the classical or Montgomery algorithm is being used: $P \leftarrow P{+}a_iB$. This is how it is computed in the two algorithms 0 and 1 above. Alternatively, the *product scanning* technique accumulates all the digit products a_jb_{i-j} ($j{=}0, 1, 2, \ldots$) at the ith step, again with i decremented or incremented at each step: $P \leftarrow P{+}\sum_j a_jb_{i-j}r^i$ for digit base r [12, 19].

For both these alternatives, there is the choice of whether to alternate between the multiplication steps and the reduction steps $P \leftarrow P \pm qN$ at a digit level, or at a limb level – the level of the outermost loop over multiplier digits. These are referred to as the *finely integrated* and *coarsely integrated* operand or product scanning techniques respectively (FIOS and CIOS for operand scanning). The details of the different possibilities have been much studied by Ç. Koç and others [29, 36][3]. The reader is referred to those publications for further details. Here, as in Algorithms 0 and 1, we concentrate on CIOS where the multiplication steps $P \leftarrow P{+}a_iB$ alternate with reduction steps $P \leftarrow P \pm qN$. (C is used in [5] for the partial product rather than the P here.) However, other processing orders can be beneficial in processing carries and preventing pipeline stalls on certain hardware architectures with SIMD operations [48].

For product scanning, the successive quotient digits q have to be stored. This makes its area requirements greater than for operand scanning. Consequently, interleaved multiplication using operand scanning is generally preferred and that is what is described in this chapter. On the other hand, Liu and Großschädl [36] note that product scanning requires fewer store instructions on an Atmega128 with 32 registers, and they use it for speed.

Readers who wish to perform the multiplication independently in advance of the reduction, or perform just a Montgomery reduction, can still use the contents of this chapter almost verbatim. As with hardware multiplication, a library modular multiplication routine is often best presented as a modular multiply-accumulate operation since the initial partial product is almost as easily set to the value of the accumulate argument as to zero. That can be done by a very minor modification of Algorithm 0. So, for a modular reduction only, readers just need to omit the step $P \leftarrow P{+}a_iB$ (i.e. set $A = 0$ or $B = 0$) after initialising P to the product value which, in this case, may initially have many more digits than N.

3.7 Accepting Inaccuracy in Quotient Digits

This section takes a close look at the definition and computation of the quotient digit q in order to highlight the main difference between the traditional and Montgomery algorithms. It is primarily this difference which makes Montgomery the preferred method in the majority of high-end commercial cryptographic applications, whether in hardware or software. Readers are invited to skim or skip the technical details of the following sub-section on the traditional modular multiplication algorithm and head straight for the sub-section §3.7.3 on Montgomery's method unless and until they have the need to check the mathematics or perform similar calculations themselves! A final summary in §3.7.4 covers what is necessary to appreciate the value of Peter's contribution.

[3][29] claims that FIOS requires more digit level additions and reading/writing operations, making it slower than CIOS. This occurs in their algorithm because of an extra digit addition to incorporate the upper word from the first multiply-accumulate digit operation (MAC). However, this extra addition can be avoided by having two carry digit variables instead of one and incorporating one carry into each of the two multiply-accumulate operations as occurs in the MACs of their CIOS version. Thus, the algorithms should use the same number of each type of digit operation.

3.7.1 Traditional

The aim of the acceleration technique in this section is to simplify the hardware logic for calculating q so that it is much faster. This is achieved in two ways. One is to replace the division by N in line 4 of the classical Algorithm 1 with a multiplication by a pre-calculated N^{-1}. Multiplication is faster than division, so this improves the speed. The other aid is to approximate the long numbers P and N^{-1} by using only their most significant digits. The shortened multiplication will also speed up the calculation of q. Recognition of the use of such an approximation process is given by writing "$q \approx \ldots$" in the algorithm. The resulting method is generally known either as Barrett reduction [2] or as Quisquater's method [23], depending on the details of the approximations. The more general version here appeared in [54] and later in [8, 9, 10, 26], and it allows more control over how large P might become.

These simplifications lead to an occasionally slightly inaccurate value for $\lfloor P/N \rfloor$. However, with care they can be used in every multiplication step without any correcting adjustment until after the main loop terminates. Clearly, if too low a multiple of N is subtracted on one iteration, a compensating subtraction needs to be made at some future point. Hence the "digit" q may have to be larger than the natural bound of the base r which is being used for the number representations. The larger digit then compensates for the inaccuracy of the previous choice of quotient digit. At the end of Algorithm 1, conditional subtractions have been added in case P has grown to be larger than N.

Let us denote by p' and n'_{inv} the approximations to P and N^{-1} which will be used in defining q. They will turn out to be just two or three bits longer than a radix-r digit. For convenience, assume r is a power of 2 and let us aim at k-bit approximations for some small k that we have yet to determine. Define Z (for "Zwei") as the power of 2 such that

$$2^{k-1}Z < N \le 2^k Z \tag{1}$$

Then the most significant few bits of the inverse N^{-1} are given by

$$n'_{inv} = \lfloor 2^{2k}Z/N \rfloor \tag{2}$$

that is, the greatest integer equal to or less than $2^{2k}Z/N$. Bounds on n'_{inv} are easily deduced from these definitions: $n'_{inv} \le 2^{2k}Z/N < 2^{2k}Z/2^{k-1}Z = 2^{k+1}$ and $n'_{inv}{+}1 > 2^{2k}Z/N \ge 2^{2k}Z/2^kZ = 2^k$, so that

$$2^k \le n'_{inv} < 2^{k+1} \tag{3}$$

Thus n'_{inv} has exactly $k{+}1$ bits, the leading bit being 1.

Next, let us write P_{mid} for the value of P which is used in the calculation of q (Algorithm 1, line 4). To reduce P_{mid} modulo N, the bits we need from P for the approximation p' are those above the position of the top bit of N plus one or two more to make q accurate enough. This is fewer bits than given by $\lfloor P_{mid}/Z \rfloor$, so let

$$p' = \lfloor P_{mid}/zZ \rfloor \tag{4}$$

where z is a (small) power of 2 which will be determined shortly.

The new, approximate definition for q is then the following:

$$q = \lfloor p' \cdot n'_{inv} z / 2^{2k} \rfloor \tag{5}$$

Note the symmetry here with the definition (10) of q for Montgomery's algorithm: a product of the top bits of P and N^{-1} is taken rather than a product of the bottom bits of these quantities. Note also that the integer division by 2^{2k} in (5) is trivial in hardware logic so that, as in Montgomery's algorithm, q is essentially determined just by a multiplication. Thirdly, note that, as in computing n_0^{-1}, the cost of computing n'_{inv} is almost immaterial since n'_{inv} will only be computed once for each fresh modulus N. First, n'_{inv} can be approximated using just the top few digits of N, and then that value incremented until (2) holds.

From (5), $q \leq p' \cdot n'_{inv} zZ/2^{2k}Z \leq P_{mid} n'_{inv}/2^{2k}Z \leq P_{mid}/N$, so the partial product P does not become negative in line 5 of Alg. 1. This establishes 0 as a lower bound on P throughout execution of the algorithm. As each of these comparisons could be equalities, this is in some sense the best approximation we can employ to maintain this lower bound[4].

3.7.2 Bounding the Partial Product

The next task is to establish a common upper bound $\mathcal{B}$ on the value P_{end} of the partial product at the beginning and end of each iteration. This bound depends on the number of input bits which are used in calculating q. The more bits are taken, the more accurate q is, and so the lower the value of $\mathcal{B}$ can be made.

As the bound $\mathcal{B}$ must be preserved from one iteration to the next, we require

$$P_{end} \leq \mathcal{B} \;\Rightarrow\; rP_{end} + a_i B - qN \leq \mathcal{B} \tag{6}$$

for each digit a_i in A. To determine a suitable value $\mathcal{B}$ easily, we now switch from discrete, integer arithmetic and allow continuous real-valued quantities for the variable P. As will be clear from the argument below, the value of $\mathcal{B}$ obtained in that context will certainly work for the integer case.

The most difficult case to satisfy (6) is when the approximation q is smallest compared with the correct value, $\mathcal{B}$ is the least upper bound, and P_{end} and $a_i B$ are maximal. With P_{mid} the value used to calculate q as in §3.7.1, this would occur for the following values at the point when q is evaluated: $P_{end} = \mathcal{B}$, $q = (p' n'_{inv} z - 2^{2k} + 1)/2^{2k}$, $a_i = r-1$, $B = N-1$, $P_{mid} = p'zZ + zZ - 1$ and $(n'_{inv}+1)N = 2^{2k}Z$. Note that the graph of $P_{mid} - qN$ is like the teeth of a saw, increasing at the same rate as P_{mid}, but stepping down each time q increases by 1. The restriction on the form of P_{mid} is to ensure we select a value at the

[4]Signed numbers are usually avoided in modular arithmetic and they are avoided here. However, if we wanted the output to be the residue of least absolute value instead of the least non-negative, we might take the nearest integer approximation in the definitions rather than greatest integer below. This would sometimes cause P to be negative and a new negative lower bound would need to be established in the same way as is done next for the upper bound.

top of one of the teeth as these are the worst cases for satisfying (6). The values of these maxima increase with P because q is defined using the upper bound $2^{2k}Z/n'_{inv}$ for N instead of N itself. So under these conditions the maximum value for P at the start of the loop will lead to the maximum output value at the end of the loop. Hence, the selection of $\mathcal{B}$ as the least upper bound means that $\mathcal{B}$ will also be the value at the end of the iteration. Substituting in these values at the beginning and end of the iteration,

$$\begin{array}{rcl} P_{mid} - qN & = & p'zZ{+}zZ{-}1 - (p'n'_{inv}z{-}2^{2k}{+}1)N/2^{2k} \;=\; \mathcal{B} \\ P_{mid} & = & p'zZ{+}zZ{-}1 \;=\; r\mathcal{B}{+}(r{-}1)N \end{array} \tag{7}$$

Eliminating $\mathcal{B}$ and N from the equations (7) and ignoring the relatively small terms which are not multiples of Z yields

$$p'z = \frac{(r{-}1)(n'_{inv}{+}1)z + (2r{-}1)2^{2k} - r}{n'_{inv}{+}1{-}r} \tag{8}$$

As the numerator is positive, and p' is non-negative, the denominator must be positive. Hence $n'_{inv} \geq r$, which is expected because n'_{inv} must provide q with at least as many bits of accuracy as in a digit.

The more bits chosen for n'_{inv}, the better the approximation for q and so the lower the bound $\mathcal{B}$. A good choice is to take n'_{inv} with three more bits than a digit, i.e. $4r \leq n'_{inv} < 8r$ so that $2^k = 4r$ by (3). Then plugging the above value for $p'z$ into the second equation of (7) and using $(n'_{inv}{+}1)N = 2^{2k}Z$ to eliminate k yields an upper bound on P_{end} of

$$\frac{(r{-}1)zZ{-}Z{+}(2{-}r^{-1})(n'_{inv}{+}1)N}{n'_{inv}{+}1{-}r} + zZ - (1{-}r^{-1})N$$

which is easily determined to be less than

$$\frac{(r{-}1)zZ{-}Z{+}(2r{-}1)N}{3r{+}1} + zZ + N \;=\; \frac{5r}{3r{+}1}N + \frac{4rz{-}1}{3r{+}1}Z \tag{9}$$

Using the property $2rZ < N$ which holds for this case, this in turn is less than $2N$ for $z = r$ in the case of $r = 2$, and always less than $2N$ for $z = \frac{1}{2}r$. Thus $2N$ is always an upper bound on the output of each iteration in the modular multiplication for some z. This yields $P_{mid} < 2rN{+}(r{-}1)(N{-}1) < 3rN$ as a bound on the size of register needed for P. Then the property $q \leq P_{mid}/N$ means $q < 3r$ and so q has at most two more bits than a digit. Moreover, $N < 4rZ$ yields $P_{mid} < 12r^2Z = 12rzZ$ or $24rzZ$ for $z = r$ or $z = \frac{1}{2}r$ respectively. So p' will be 4 or 5 more bits than a digit.

Appropriate adjustments to these calculations can easily be made for different scenarios, such as i) if B is not fully reduced but has another upper bound than N (such as R or $2N$), ii) if some of the quantities have carry-save representations that lead to different bounds on their values, iii) if the multiplication is completed before any reduction, or iv) N has a special form or fixed, known value.

Clearly, once derived, the classical modular multiplication algorithm has a straight-forward formula for the multiple q of N which needs to be subtracted. It is only mildly tedious to determine in the above manner i) the best number of bits to use in the calculation and ii) an upper bound on the partial product to ensure sufficient register space is made available. However, the fact that q and the inputs to its calculation are generally larger by several bits than the radix r means that the built-in hardware multiplier and bus width may be too small for computing q and qN in the most obvious, convenient way.

3.7.3 Montgomery

By contrast, the determination, calculation, storage and use of q is much easier in Montgomery's Algorithm 0: the value of q is simply

$$q = -p_0/n_0 \bmod r \tag{10}$$

Unlike the classical case, all the operations here involve quantities which are within the normal digit range $[0, r{-}1]$. So fewer bits are required from N^{-1} and P for computing q than in the classical case, and a standard digit$\times$digit multiplier suffices for computing qN. As with the traditional algorithm, the definition of q using short division is replaced by one involving multiplication:

$$q = p_0{\cdot}n_{inv} \quad \text{where } n_{inv} = -{n_0}^{-1} \bmod r \tag{11}$$

It is clear by induction that $2N$ is an upper bound on the value of P at the end of each iteration and so also at the end of the loop when $B < N$ and the digits of A are from a non-redundant representation, i.e. in the range $[0, r{-}1]$. However, for several reasons, we need to look at this bound again later because the inputs A and B are more likely to be bounded above by $2N$ rather than N.

3.7.4 Summary

This section has highlighted one of the key problems with the classical algorithm that makes Montgomery's algorithm so much easier to implement: checking the details of the classical algorithm and implementing it are more complex. In particular, the value of q is typically two bits larger than a normal digit and so requires a larger hardware multiplier and perhaps more clock cycles or wider buses than normal. On the other hand, in Montgomery's algorithm bounds are easily established, q has a normal digit size, and standard multipliers and buses can be used.

Although the computation of q and the cost of the scalar multiplication $q{\cdot}N$ are slightly more expensive for the classical algorithm, this needs to be compared with the cost of scaling the inputs and output from Montgomery's algorithm. In the case of RSA exponentiation, these costs are fairly similar if the multiplier can be chosen accordingly. However, with a fixed digit-sized multiplier, the extra bits in q make the classical algorithm more expensive to design, more complex to implement, hungrier in its firmware area, slower to execute, more prone to implementation errors, and therefore overall less profitable and generally less desirable to choose.

3.8 Using Redundant Representations

The use of redundant representations enables digit calculations to be done in parallel. Typically this employs a carry-save representation in order to avoid the problems of carry propagation exhibited by the decimal addition of 1 to 999...9 or the subtraction of 1 from 100...0. This is the problem that forces all our schoolbook arithmetic to be done from the least significant digit to the most significant. However, with a carry-save representation the carries (or borrows) are absorbed by the next digit up as part of the next operation instead of the current one. Then carries only need to be propagated at the end of all the arithmetic.

An addition of two base r digits x and y in the range $[0, r{-}1]$ creates a result in the range $[0, 2r{-}2]$ which is stored as a base r digit s (the save part) and an overflow bit c (the carry part) such that $x{+}y = s{+}rc$. Fortunately, this addition also has space to incorporate a carry bit c' from a previous addition in the position below: $x{+}y{+}c'$ is in the range $[0, 2r{-}1]$ and so it too can be split into a one bit carry and a save digit. This means long number additions can easily be done in any order, not just from least to most significant digit but also, for example, from most to least significant digit or all together in parallel, or a number of digit additions in parallel using whatever resources are available. In an ASIC the extra combinational logic and wiring for one extra bit per digit is not too significant.

Similarly, a multiplication of two base r digits x and y creates a result in the range $[0, (r{-}1)^2]$ which is split into two digits s and c such that $x{\cdot}y = s{+}rc$. More generally, the operation is usually implemented in hardware as a multiply-accumulate (MAC) operation which will add in one or two further digits: $MAC(x, y, d, e) = x{\cdot}y{+}d{+}e$ generates an output in the range $[0, r^2{-}1]$ and can therefore also be represented in a two-digit carry-save form (s, c) whose value is $s{+}rc$. Thus, the carry from one MAC of the form $x{\cdot}y{+}z$ can be accumulated by the next digit position up when it performs a similar MAC operation on the next clock cycle. As with additions, this means long number scalar (limb) multiplications such as $P \leftarrow P{+}a_i{\cdot}B$ can easily be done from least to most significant digit, but also that a succession of long number scalar multiplications $P \leftarrow P{+}a_i{\cdot}B, i = 0, 1, \ldots$ can have their digits processed in parallel, with each scalar multiplication absorbing the carries from the previous such operation.

The use of parallel digit operations implies significant hardware resources for long integer operations, with adders, multipliers, memory and registers required for a number of digit positions. These are available to a limited extent in multi-core processors and more widely on FPGA boards and ASICs. However, the usual count of logic gates is insufficient as a measure of area complexity. This is because there can be significant wiring overheads to consider as well since some data needs to be broadcast simultaneously to all digit positions which are performing operations. For example, when modular multiplication digits are computed in parallel, the multiplier and quotient digits a_i and q need to be broadcast to each digit position. Whereas the digits a_i are known in advance and can be queued ready for use, the digits q may only be known on the previous cycle and require immediate broadcast to all computing elements with consequent heavy wiring cost.

Traditional

In the classical algorithm with a standard number representation, if we want to avoid computing the top digits twice the determination of q can only be easily made once the carries have been fully propagated. This limits the faster computation of the modular product even if additional computing elements are available. However, with a carry-save representation, additional multipliers can be used for parallel digit operations or digits can be processed most significant first provided an approximation is used for q that doesn't require the carry-up to be known. In section §3.7 we saw how approximate values can be used satisfactorily. As carries from lower positions have almost no impact on the computation of the top digit or so of P, and only one or two more bits than the top digit are used in order to choose q, the adjustments to the argument in §3.7 to accommodate a redundant representation are minimal, with only a minor increase in the maximum value of P if no other action is taken.

Montgomery

In Montgomery's algorithm q does not depend on the completion of any carry propagation. Hence multiple computing elements can be used in parallel with ease if a carry-save representation is used. However, there is still one significant carry to take care of, and this can lead to bubbles (which are when computing elements are left with nothing to do). Specifically, if one takes the iteration boundary to be when q is calculated, then one loop iteration of the interleaved Algorithm 0 computes first q and then $P \leftarrow (P+q\cdot N)r^{-1}+a_i\cdot B$. Thus, as well as the lowest digit of $a_i\cdot B$, the next value of q depends on the digits with index 1 from $P+q\cdot N$ plus any carry from its digits of index 0.

In both algorithms we see that latency − the time from input to output − can be reduced by employing redundancy when more hardware resources are available. The carry propagation need only be done after the final operation of a cryptographic function, not during or after every constituent arithmetic operation although, as we will see later, executing some limited intermediate carry propagation can be very useful to reduce the size of carries. However, as q is on the critical path, even with additional resources both methods are potentially held up while q is computed and distributed to all the computing elements.

3.9 Changing the Size of the Hardware Multiplier

Suppose first that there is no choice over the available integer multiplier because the hardware platform or multiplier design is fixed. Typically, the multiplier will perform a multiply-accumulate (MAC) operation $a\cdot b+d$ or perhaps $a\cdot b+d+e$

where a, b, d and e are single words. Then the multiplier may be used more efficiently by supplying the modular multiplication arguments A and B divided into digits which don't adhere to the word boundaries. In particular, if digits have fewer bits than the word size, carries may be incorporated within a word without any overflow. This is very helpful for a carry-save representation and a standard square multiplier may be able to process these values and the over-sized q from the traditional algorithm in one go.

As noted in the previous section, the accumulate part conveniently accommodates carries from long integer multiplications. However, the multiplier may not be square (i.e. the above MAC inputs a and b could have different lengths), and may indeed allow several extra bits above the word size for each argument in order to deal with overflows more efficiently or to enable better rounding when used for floating point operations. When a carry-save representation is used, any available extra bits in multiplier arguments can be valuable in processing carries.

There may be several multipliers which can be operated in parallel on the chip using a SIMD architecture. Combined in this way, they should operate as a single non-square multiplier enabling several words/digits of long integer arguments to be processed simultaneously in a single MAC operation $a{\cdot}b{+}d$ where a and b now have different numbers of bits but which is actually split over the multipliers. Consequently, in order to make the best use of the multiplier(s), we need the flexibility to consider the arguments of $(A{\cdot}B) \bmod N$ to have representations with different bases which may or may not match the word size or multiples of it [22, 43].

However, if there is a choice of multiplier, then one has to balance costs such as power and floor area against any advantages of the various choices as well as noting that, while a larger multiplier reduces the number of clock cycles, it may increase the depth of the hardware that has to be driven so that each clock tick is longer. Hardware synthesis tools, such as those provided by Synthesys, include Intellectual Property (IP) Blocks for parametrisable multipliers, thereby enabling ASIC designers to choose their own MAC operation circuitry without having to design it themselves. A $v{\times}w$-bit multiplier will have area approximately proportional to vw and critical path depth proportional to $\log(vw)$ plus the register delay and set-up and hold times.

As observed in §3.8, having a second argument that can be accumulated in a MAC operation is useful. If the multiplier is non-square, i.e. has multiplier and multiplicand inputs a and b with different numbers v and w of bits respectively, then its maximum output for $a{\cdot}b$ is $(2^v{-}1)(2^w{-}1) = (2^v2^w{-}1) - (2^v{-}1) - (2^w{-}1)$. So it is possible to accumulate arguments d and e of v and w bits respectively without overflowing the $v{+}w$ bits necessary for the output. When performing a scalar multiplication $a_i{\cdot}B$ where a_i has v bits and B is partitioned into w-bit digits, each use of the multiplier performs a v- by w-bit multiplication, and accumulates both the save value of w bits from the previous operation at that position and the carry of v bits from the MAC on the previous position.

At the extreme, an $n{\times}1$-bit multiplier is just an adder; dispensing with a multiplier is a possible optimisation. Without a multiplier but registers that can hold all of A, B and N, the clock speed in a cryptographic co-processor can usually be increased substantially. In the early days of ASICs for RSA cryptography, the fastest chips often used base 4 for A and, rather than employing a multiplier, just added in B and $2B$ to the partial product as required by the two-bit digits of A. However, besides initialisation, multiplication, additions and subtractions, the modular multiplication algorithms include reading and writing to various types of memory and other movement of data as well as comparisons, incrementing of counters and instruction processing. Some of these rather than the multiplier may limit clock speed, or even dominate overall performance by using many clock cycles.

If Montgomery's modular reduction is used then the multiplication requirements are marginally less than for the traditional algorithm. Specifically, q has around two fewer bits, which may make the scalar multiplication $q{\cdot}B$ less expensive (see §3.7). Multipliers may require more area if the number of bits in the arguments is increased, but their throughput and depth of combinational logic should be logarithmic in the number of bits. Consequently, adding two bits to an argument (and to the bus) to accommodate the classical algorithm may not alter the time a cryptographic process takes. However, the power and area requirements will be marginally increased.

3.10 Shifting an Operand

In this section we start to make progress on the earlier computation of q so that its calculation ceases to be a bottleneck. Specifically, we want to ensure that none of the hardware devoted to the long number operations $P \leftarrow P{+}a_i{\cdot}B$ and $P \leftarrow P{\pm}q{\cdot}N$ is lying idle while q is computed and broadcast to where it is needed.

Traditional

In the classical Algorithm 1, the determination of q is on the critical path. The operation $P \leftarrow P{-}q{\cdot}N$ cannot start until q is known. Therefore, any simplification in computing q has the potential to improve the latency of the modular multiplication.

Let us shift up the multiplicand A and the modulus N by several digits before the modular multiplication starts, and perform a compensating shift down at the end. So we calculate

$$((AS{\cdot}B) \bmod (SN))/S$$

where S is the power of r corresponding to the shift. The output is still equal to $(A{\cdot}B) \bmod N$ but the inputs to the calculation of q now come from several digits higher up in the registers. Using $as_i, i = 0, 1, \ldots$, to denote the digits of AS, the shift must be sufficient to the move the position of these input bits to above the most significant bit of each as_iB. Then B is small in comparison with the new modulus SN and addition of the digit multiple as_iB in $P \leftarrow rP{+}as_i{\cdot}B$ will only affect the topmost bits used to compute q if there is a carry which propagates as far as the relevant top bits of P. We saw earlier that q is only computed as an approximation to the correct value, and so it can allow for ignoring that propagated carry. In fact, the relatively smaller value of B means that the formula for q and the bound $\mathcal{B}$ in §3.7 can be improved[5]. The determination of q can now be advanced to make it available without delaying the reduction step: it can be computed after the assignment $P \leftarrow P{-}q{\cdot}N$ in the previous loop iteration without waiting for the addition $P \leftarrow rP{+}as_i{\cdot}B$.

Surprisingly, the computational complexity is little changed although the shift means more iterations – one more for each digit of shift. For simplicity, let us assume that the shift is by a whole number of digits. Indeed this must be the case if, as in §3.5, we are aligning the most significant bit of the modulus with the top bit of a digit. Because the number of non-zero digits in AS is the same as in A, the number of digit×digit multiplications $a_i{\cdot}b_j$ is unchanged by the shift if the program code can omit the steps for which the bottom digits of AS are known always to be zero. Thus the scalar multiplication steps $P \leftarrow rP{+}as_i{\cdot}B$ should have the same computational complexity as before.

Now consider the cost of the reduction steps. First note that the sequence of values q form a long integer Q such that $Q \cdot SN$ is the quantity subtracted from P by all executions of Line 5 in Algorithm 1, while $AS{\cdot}B$ is the quantity added to P by all executions of Line 3. So $AS{\cdot}B - Q \cdot SN$ is the output of the main loop. Incrementing Q as necessary to account for any subtraction arising from the final lines 7 to 9 yields $Q = \lfloor AS{\cdot}B/SN \rfloor$. This integer quotient is unchanged by the shift since $\lfloor AS{\cdot}B/SN \rfloor = \lfloor A{\cdot}B/N \rfloor$. So the *value* of Q is the same in both cases. However, the two *representations* of Q may be different as a result of the shift. From §3.7.2, a typical definition of q ensures $q_i < 3r$ for each i. Hence, both representations might differ from the standard radix-r representation by a borrow of up to 2 being transferred from each digit to the one below it. So there is almost no scope for changing the number of digits in Q by the shift (unless r is very small). Thus, as program code should ensure the same number of digit×digit multiplications are performed calculating $q{\cdot}SN$ as $q{\cdot}N$, the shifted and unshifted modular multiplications can normally be expected to require the same number of digit×digit multiplications in the modular reduction steps.

Before concluding, let us review the minor differences in computational complexity that might arise in the modular reduction steps. Since B is smaller compared to the modulus in the shifted case, the upper bound on P is a little smaller, making the digits q_i slightly smaller on average in that case. For certain parameter choices, the lower bound on these digits could reduce the number of bits they require, leading to small but real hardware and power reductions in digit×digit multiplications. The smaller bound on P could also translate to slightly fewer iterations of the final conditional subtraction in lines 7 to 9 of Algorithm 1. On the other hand, the lower bound on digits q_i may lead to the representation of Q sometimes using one more digit than in the non-shifted case[6].

[5]$\mathcal{B}$ is smaller by about $\frac{1}{3}N$ for the parameter choice in §3.7.2 and a one digit shift.

[6]Note, however, that the maximum value for the leading digit is atypical as it depends closely on the upper bounds for A and B. If both are bounded by $2N$, then $Q < 4N$ and its

So, overall, the main cost of this acceleration technique is just some increased data movement caused by the shifting, an increased register length, not forgetting, of course, the increased program code area to avoid the known multiplications by zero. However, the arithmetic digit operations have very similar costs.

How and where does this technique improve performance? Recall that the purpose of shifting A and N up is to enable the digits q to be calculated earlier, and specifically without having to await the addition in line 3 of Algorithm 1. The value of the technique arises when several digit$\times$digit multipliers (or equivalent) are available to operate in parallel. One scenario might be using a number of processing elements (PEs) in an FPGA with a pipeline of digits in memory awaiting processing. Another would be a multi-core processor. For simplicity, assume that the digit$\times$digit multiplier can accept q as one of its inputs[7]. Assume also that one multiplication is enough to calculate q and that in total k multipliers are available. Then, without the shift, for each of the n loop iterations there can be some multipliers idle while the last digits of $P \leftarrow rP{+}a_i{\cdot}B$ are determined and then there are $k{-}1$ multipliers that are idle during each calculation of a quotient digit. However, with the shift and a suitable instruction set, $P \leftarrow rP{+}a_i{\cdot}B$ can commence while one multiplier is computing q, so that q is ready when there are multipliers available for starting on $P \leftarrow P{-}q{\cdot}N$. Then no multipliers are idle until the end of the last loop iteration when at most $k{-}1$ multiplier cycles may be lost rather than at least $n(k{-}1)$. Thus, at least $k{-}1$ idle PE cycles will have been saved on every iteration except maybe the last.

Remarks. i) An alternative view of this shifting process is that the reductions are simply delayed until q is available and, in the meanwhile, the processing of the multiplication steps continues.

ii) Since a carry-save representation is used to enable parallel processing of k digit positions, we could avoid performing the shift and still calculate q in time. Thus, the scheduler could select the topmost digits of line 3 to be computed first, then compute q using the next free multiplier cycles, and then complete the execution of line 3. This way, q would be available for use in starting line 5 if there were spare multiplier cycles when finishing the execution of line 3. However, executing line 4 within line 3 is bound to be rather messy for both code and data movement.

Montgomery

In Montgomery's method, the corresponding shifting technique requires making the determination of q independent of the lowest digit of B which is used in the step $P \leftarrow P{+}a_iB$. B can be shifted up to achieve this by computing the

top digit position could only have the value 0, 1, 2 or 3.

[7]This property of the multiplier is desirable in an ASIC because half of all digit products involve the quotient "digit" q, but recall that q may exceed $r{-}1$.

Montgomery product

$$P \leftarrow (A \odot SB) \bmod N$$

where the notation $\odot$ is a convenience to avoid explicitly writing in the introduced power of r which depends on the number of loop iterations in Algorithm 0. This number is discussed in the next paragraph. So B is shifted rather than A and N which were shifted in the traditional schoolbook method above. Once more, assume the shift is by a whole number of digit places since otherwise there are needless complications.

As in the classical case, some extra loop iterations are required to process the larger arguments. Thus, if the original Montgomery computation requires n iterations (the number of digits in A and B) and B is shifted up by s digits, i.e. $S = r^s$, then there should be $n{+}s$ iterations in $P \leftarrow (A \odot SB) \bmod N$. The extra s iterations process s extra digits $a_n, a_{n+1}, \ldots, a_{n+s-1}$ of A which are all zero. This means $(A{\cdot}Br^{-n}) \bmod N$ is computed in the unshifted case and $(A{\cdot}r^sBr^{-n-s}) \bmod N$ in the case of a shift by s digits. Hence the residue class modulo N of the output is unchanged by the shift and corresponding extra iterations. The value of the partial product P at the end of the last iteration is easily seen to be less than $(A{\cdot}BS + r^{n+s}{\cdot}N)r^{-n-s} = (A{\cdot}B + r^n{\cdot}N)r^{-n} < 2N$, which is the same upper bound as in the unshifted case. Hence the output of the shifted algorithm satisfies all the post-conditions of the unshifted algorithm, and is therefore acceptable as is. In fact, a more careful analysis would reveal that the arithmetic is entirely identical, so that the outputs are the same. As noted for the classical algorithm shift in Remark (i) above, the shift simply delays the evaluation of q and addition of $q{\cdot}N$ relative to the addition of $a_i{\cdot}B$.

To appreciate the potential value of the shift, suppose again that there are k hardware multipliers that can simultaneously process k digits of the long number operations $P \leftarrow P{+}a_i{\cdot}B$ or $P \leftarrow (P{+}q{\cdot}N)/r$. If the three steps of each loop iteration are performed sequentially using the multipliers as they become available, then up to $k{-}1$ may again be idle waiting while one multiplier computes q before $P \leftarrow (P{+}q{\cdot}N)/r$ can commence. The solution to this bottleneck requires earlier scheduling of the computation of q. Without the shift, it would have to take place somewhere in the middle of performing $P \leftarrow P{+}a_i{\cdot}B$ once the lowest digits have been obtained. But, as with Remark (ii) above for the classical algorithm, this would be messy to implement. It is far cleaner to perform a shift so that the determination of q can precede $P \leftarrow P{+}a_i{\cdot}B$.

Such a solution can also cater for the case when the hardware processes all digit positions of $P \leftarrow P{+}a_i{\cdot}SB$ and $P \leftarrow (P{+}q{\cdot}N)/r$ in parallel and separate circuitry is used to find q. Then, to process one loop iteration requires a depth of hardware equal to two multipliers in each digit position above the shift. However, below the shift position only the depth of a single digit$\times$digit multiplier is required since the corresponding digits of SB are zero. This gives the time needed to compute q without delaying the rest of the hardware [13]. Moreover, each further shift by one digit provides additional time to calculate and broadcast the quotient digits.

In one sense the overall computational complexity is essentially the same with the shift as without: the same digit$\times$digit products need to be formed – the extra such products involve an operand digit which is known to be zero and so can be programmed out. The advantage in hardware is that it is much easier to avoid resources being idle while quotient digits are computed. Thus time is reduced. However, as in the case of shifting for the classical algorithm, the number of data movements and other minor operations has increased slightly. Thus the shifting technique applies equally well to the two algorithms.

Remarks. i) Whilst a shift of about one digit position seems sufficient for both the classical and Montgomery algorithms, a larger shift enables the computation of q to start even earlier and results in more time being available for its calculation and broadcasting to all digit positions when digit operations are done in parallel.

ii) For large moduli and large numbers of multipliers, the wiring, multiplexers and power for distributing the value of q can become a significant cost that might take several clock cycles and needs to be factored into any measure of circuit complexity. Such wiring may lead to design and implementation problems arising from routing issues and noise from crosstalk. However, shifting inputs as above is free of any need for further pre- and post-processing of I/O.

3.11 Pre-computing Multiples of B and N

A much-valued and widely used method for increasing the speed of many computations is to pre-compute and store frequently used values [13, 42]. Including tables of pre-computed values is a space-time trade-off. It reduces algorithmic time by computing repeatedly used data once beforehand instead of every time it is required. The cost is the increased area taken up by the memory used for the data. Here, if we have some control over the design of the multiplier, which may be the case with an ASIC, the digit multiplier can be personalised into essentially an adder of a subset of pre-computed values aB and qN. This adapted multiplier should then execute faster than a general multiplier. When memory for the pre-computed values is very cheap or available and otherwise unused and access to it is fast, this makes good sense.

If only a very small number of bits of A are processed in one clock tick, then every possible value of aB and qN or $aB{\pm}qN$ could be pre-computed and stored in a look-up table (LUT). As the same digits a and q will turn up frequently, the repeated re-computation of aB and qN would be completely avoided. Then each iteration of either modular multiplication algorithm would require just one or two additions and no multiplier. However, in practice a and q can be only at most two or three bits in size before the space requirement becomes prohibitive.

For larger radices $r = 2^w$, simply storing the w shifted values 2^iB and 2^iN, $i = 0, \ldots, w{-}1$ is not so expensive and may save valuable computing time and power when shifting itself takes time. If the necessary shifts cost virtually nothing (such as when hard-wired into an ASIC) then storing a small number of other combinations, such as B, $3B$, N, and $3N$, can be worthwhile. Then, for example, $P+a_i{\cdot}B$ can be formed by adding in shifted copies of B and $3B$ as necessary for every pair of bits in a_i.

The advantages of using a LUT depend critically on the extent to which the technique shortens the critical path length or reduces the power consumption, and these will vary enormously between different implementations. Selecting and loading the required multiples of B and N itself takes time. There are also many different types of memory, some fast but expensive such as register space and others cheap but slow.

These techniques apply equally well and in exactly the same way to both classical and Montgomery modular multiplication, but may or may not be applicable in a particular context.

3.12 Propagating Carries & Carry-Save Inputs

For a modular multiplication we will always assume the modulus input N is in a standard non-redundant form, i.e. its digits are in the range $[0, r{-}1]$ for some radix r. This is because its conversion to such a form will be cheaper than the extra processing involved in the modular multiplication using a redundant carry-save form, as described in §3.8. However, redundancy may be desirable to allow in one or both of the other inputs, i.e. in A or B, because such a form enables digit values to be processed independently.

When modular multiplication is used in the context of a modular exponentiation, the output from one modular multiplication is an input to a subsequent such operation. If only one or at most two digit multipliers are available, then the modular multiplication algorithm should process the partial product P sequentially from least to most significant digit propagating carries on the way and outputting a result in non-redundant form. This is not a problem for inputting to the next modular multiplication. However, since the use of carry-save representations is helpful when a greater number of digit multipliers is available, carry propagation may be necessary between modular multiplications unless the modular multiplier allows inputs in redundant form.

As the hardware modular multiplier may be used for modular squaring, one might expect that both or neither of the arguments should allow for redundant inputs. However, the digit products in a given $a_i{\cdot}B$ may be performed in parallel, and the scalar products $a_i{\cdot}B$ ($i = 0, 1, \ldots, n{-}1$ or $i = n{-}1, n{-}2, \ldots, 1, 0$) generated sequentially. This means the digits of operand B are required in parallel, but the digits of A are consumed sequentially. Thus there could be time to convert operand A to a standard form before use but not operand B. So sometimes only one argument may need to be allowed a redundant form, even for modular squarings.

Clearly, a carry-save representation takes up valuable resources, adding to the cost of reading, processing, writing and storing the output. Consequently, although processing the carries takes time, carries should always be propagated when there is minimal cost, such as by using any spare argument in a MAC operation, as noted earlier. Otherwise, the propagation cost is that of one (full length) addition when digits are processed sequentially. In practice, this is unlikely to be much less than the cost of a scalar multiplication $a{\cdot}B$ or $q{\cdot}N$. However, even a full-length, digit-parallel addition would reduce each carry-save pair of digits to a single digit and a one bit carry. When performed on the output from one modular multiplication, it reduces the work for the next modular multiplication: with a full two-digit carry-save representation for B and single digit for a_i, the scalar multiplication $a_i{\cdot}B$ will require $2n$ digit$\times$digit multiplications whereas reducing each digit position of B to a digit and a carry bit cuts the work to only n digit$\times$digit multiplications and an n-digit addition. Indeed, the hardware multiplier might be large enough to accommodate the carry bit as well as the digit, §3.9.

Unfortunately, in the classical algorithm, the digits of input A are consumed from most to least significant. So, if the argument A is not already in a non-redundant form, it is not easy to convert it fully on a digit-by-digit basis before use if time is an issue. However, A can still be converted from a two-digit carry-save representation to a digit and carry bit representation by processing the digits from most significant to least significant, thereby reducing the work of each modular multiplication in the same way as when improving the representation of B. There may be sufficient time and resources to do this. Then the digits a_i will have a range less than that of q and a multiplier which is large enough to compute the products $q{\cdot}n_i$ may also be sufficient for computing the $a_i{\cdot}b_j$. On the other hand, in Montgomery's algorithm the digits of A are consumed one-by-one, least significant first. So, if the argument A were not already in non-redundant form, it would be easy to fully convert it just before use. This would make the formation of $a_i{\cdot}B$ cheaper and easier even than in the traditional algorithm.

Now consider evaluating $P \leftarrow P{+}a_i{\cdot}B$ using parallel digit operations with B in a carry-bit representation and a_i in standard binary. The products $a_i{\cdot}b_j$ contribute a maximum of $r^2{-}r$ to each digit position if the bit carries in each b_j are shifted up to contribute 1 to the next position. So the necessary redundancy in P will give a maximum of at least r^2 in each position. This is beyond the full extent of the carry-save representation and does not even leave room to accumulate any carry up from the previous operation without expanding into a third digit. Consequently, even in the Montgomery case, it is necessary either to have a larger digit multiplier[8] or to insert an extra addition into each loop iteration in order to deal with carries when parallel digit operations are to be performed. The situation is slightly worse for the classical algorithm where the digit representation of a_i may also have a carry bit. This emphasises the desirability of processing digits sequentially from right to left.

Lastly, in Montgomery's algorithm the value of q depends on the lowest digit of the partial product and this does not require carry propagation except from the digit position deleted in the division by r. On the other hand, for the classical algorithm the value of q depends on the top bits of the partial product,

[8]"Larger" is relative to the radix r. One can decrease r if the multiplier size is fixed. For example, in a SIMD software context, Intel [22] uses 29 bits for r rather than the more natural 32.

and its accuracy is reduced, thereby causing extra processing, if carries are not propagated. Once again, Montgomery is more efficient for the resources that are typically available.

Overall, this section has illustrated the cost of allowing redundant representations internally and for the I/O of the modular multiplier. It is greater for the traditional algorithm but is, nevertheless, still a significant issue if some parallel processing of digits is envisaged for Montgomery's algorithm.

3.13 Scaling the Modulus

The computation of q is a stumbling block for speed in the traditional algorithm, as well as perhaps slowing down Montgomery's. As seen in §3.10, in both cases operand scaling by shifting reduces the complexity of determining the quotient and so has the potential to speed up the modular multiplication. On the other hand, as observed in [5], §3.2, special moduli might be chosen to simplify quotient digit selection. For general moduli the same efficiency gains as there can be achieved through scaling the modulus[9] [53].

We have already noted in §3.10 that for both the traditional and Montgomery algorithms shifting A and N relative to B makes it possible to determine q before the addition $P \leftarrow P+a_i \cdot B$, and hence reduce any delay in broadcasting q to the positions where it is needed first for use by a bank of multipliers. However, this delay can be further reduced if the computation of the quotient is made easier by scaling the modulus.

So, specifically, we wish to remove, or at least simplify, the multiplications (or divisions) which occur in (5), (10) and (11). This is done by scaling N so that the top digit or so is all one bits in the case of the classical algorithm, and so that the bottom digit or so is all zero bits except for the lowest bit in the case of Montgomery. To be precise (except for the possible shift in the classical case to place the top bit at the top of a digit and in the Montgomery case to make N odd), replace the modulus N by its multiple

$$N^* = n_{inv} N \tag{12}$$

where n_{inv} is the quantity defined as n'_{inv} in (2) or as n_{inv} in (11), as appropriate. As well as this pre-processing, some post-processing is required. The final output needs to be reduced modulo N, but all intervening modular arithmetic will now use N^*.

For most choices of moduli, N^* will now have more digits than N and so qN^*, A, B and the partial products will be longer (by the same amount) and more time will have to be spent computing them. The increase is about one digit in length, and so about two more digit$\times$digit multiplications are required on each iteration. However, the special, simple form of an end digit of N^* should enable the extra digit$\times$digit multiplication in $q \cdot N^*$ to be avoided. So the cost

[9]Sparseness in N does not make any difference to the cost of quotient digit determination. So we do not require scaled moduli for which most digits are zero.

is one digit$\times$digit multiplication, which is exactly what has been saved from computing q. Nevertheless, this simplification of q can speed up the hardware when more than one digit position needs q immediately or multipliers are lying idle.

Let us consider in detail the properties of the new modulus N^* and how it will be used. With Montgomery's technique, n_{inv} is an odd, but standard, digit in the range $[1, r{-}1]$ and so scaling by it will provide a modulus $N^* = n_{inv}N \equiv n_0^* \equiv n_0{\cdot}n_{inv} \equiv -1 \bmod r$ which is at most one digit longer than N. So the lowest digit of N^* satisfies $n_{inv}^* \equiv 1 \bmod r$ and, by(10), the reduction then uses

$$q^* = p_0$$

No processing at all is required to obtain q^*.

On the other hand, in the revised schoolbook method of Algorithm 1, suppose we choose n'_{inv} to have two or three bits more than a digit, as in §3.7.1. The scaled modulus $n'_{inv}N$ then needs shifting (up or down) to ensure its top bit is at the top of a digit boundary. So the length of N is increased by probably one or two digits, but perhaps by none or three. In the exceptional case of N being a 2-power, this shift cancels the multiplication by $n'_{inv} = 2^k$, so $N = N^* = 2^kZ$ has an unchanged number of digits and the quotient formula (5) yields

$$q^* = \lfloor p'z/2^k \rfloor \tag{13}$$

Again, as in the Montgomery case, almost no processing is required to obtain q^*.

We will now show this formula also holds for the classical algorithm for all other N^*, thereby always removing the multiplication from (5). Without loss of generality, assume (12) holds as written, with any further shift to place the top bit on a word boundary delayed until later. Since the length of N^* is normally different from that of N, the application of (1) to N^* provides a new power Z^* of 2 defined by

$$2^{k-1}Z^* < N^* < 2^kZ^* \tag{14}$$

The maximality of n'_{inv} in (2) gives a tighter bound on N^*, namely

$$2^{2k}Z - N < n'_{inv}N < 2^{2k}Z \tag{15}$$

So $Z^* = 2^kZ$, N^* is exactly k bits longer than N, and the leading k bits of N^* are all equal to 1. Also from definition (2), n'^*_{inv} is the unique integer satisfying

$$2^{2k}Z^* - N^* < n'^*_{inv}N^* < 2^{2k}Z^* \tag{16}$$

Hence, as with the special case of a 2-power for N, $n'^*_{inv} = 2^k$ because $2^{2k}Z^* - N^* = 2^{3k}Z - n'_{inv}N \leq 2^k(2^{2k}Z - N) < 2^kn'_{inv}N = 2^kN^* < 2^{2k}Z^*$ where first inequality holds by (3), the second by(15), and the third by (14). Thus (13) holds for all cases of scaling in the classical case.

For both the classical and the Montgomery algorithms, operand scaling is therefore an easy means of removing the multiplication from the definition of q and replacing it with a simple bit selection. Note, however, that with the traditional algorithm, international standards for RSA cryptography already place the top bit of N on a word boundary. The above scaling adds k bits to N, making N^* just more than one digit longer, and causing two more digits of processing at many points. On the other hand, scaling in the Montgomery case only adds one digit to the modulus length. Therefore, at least in the classical case, modulus scaling may make more sense only as the number of digits increases.

3.14 Systolic Arrays

A systolic array consists of a number of processing elements (PEs) for repeatedly computing some function which itself consists of the repetition of a small number of different tasks. The PEs are arranged in a regular manner, communicate data only locally in the direction of computation, and each performs just one of the component tasks of the function. The input data for the function is fed in at one end of the array whenever the function has to be evaluated, and the result is eventually output at the opposite end. The array acts like a pipeline in the sense that it contains several instances of the function evaluation at various stages of completion. Such arrays can be particularly useful on servers with high volumes of connections or for other data intensive processing.

One of the advantages of using the architecture of a systolic array is the potential to avoid the cost of simultaneous data broadcasting (delays, wiring and multiplexers) to a set of PEs operating in parallel and thereby to improve Area$\times$Time efficiency. Although overall latency may not be improved by using a systolic array rather than a parallel architecture, throughput can be substantially increased.

There are many papers on the use of systolic arrays in cryptography. [55] is the first to enable modular multiplication in a fully systolic way, i.e. using PEs with only local connections so that there is no need for the simultaneous broadcast of data to many PEs. Savaş *et al.* [46] provide a wider view, integrating the integer and characteristic 2 field cases, considering scalability issues and providing simulation timings as well as a good bibliography of prior work. In [3] Bertoni *et al.* apply the concept to finite field extensions $GF(p^m)/GF(p)$.

Since hardware always involves a fixed number of PEs and there are rarely sufficient of them to process all digit positions without re-use, it is necessary to store various intermediate data in queues of digits. This aspect is covered briefly after describing the case where there are sufficient PEs. Freking and Parhi [15] discuss this and the data dependencies in a little more detail for the 2-dimensional version of the array, as well as providing more possibilities for arranging the cells.

For convenience, we will assume the PEs take a single clock cycle to perform their operation, including reading any required data, forwarding results to the next PE and storing data sent on by the previous PEs. We suppose also that a PE can perform its operation on every clock cycle. In reality, there could be a three or more stage fetch-decode-execute pipeline and some code to perform the PE's operation as a number of simpler instructions over many clock cycles. This might be the case, for example, when using a Field-Programmable Gate Array (FPGA).

3.14.1 A Systolic Array for $A{\times}B$

An easy example of a 2-dimensional systolic array is one for multiplying $A = \sum_{i=0}^{n-1} a_i r^i$ and $B = \sum_{j=0}^{n-1} b_j r^j$ using an $n{\times}n$ array of PEs. The example follows multiplication by hand, starting with the least significant digits of both A and B, adding one $a_i{\cdot}B$ at a time, and propagating carries upwards every time a digit product $a_i{\cdot}b_j$ is added to the partial product. It outputs the digits of the final product as they become known from least to most significant with each addition of an $a_i{\cdot}B$ and, for convenience, it shifts down the rest of the partial product at the same time. So, the $(i,j)^{\text{th}}$ processing element $\text{PE}_{i,j}$ $(i,j{\geq}0)$ contributes to the calculation of the product digit p_{i+j} by computing the carry-save value

$$c_{i,j+1}r{+}s_{i+1,j-1} \;\leftarrow\; a_i{\cdot}b_j{+}s_{i,j}{+}c_{i,j} \tag{17}$$

and then forwarding $c_{i,j+1}$ to $\text{PE}_{i,j+1}$, $s_{i+1,j-1}$ to $\text{PE}_{i+1,j-1}$, a_i to $\text{PE}_{i,j+1}$ and b_j to $\text{PE}_{i+1,j}$ all at time $T = 2i{+}j{+}1$ relative to the first inputs to the multiplication at $T = 0$. It is easy to see that the digit values a_i, b_j, $s_{i,j}$ and $c_{i,j}$ are then all received by $\text{PE}_{i,j}$ no later than at time $2i{+}j$, which is in time for their use in the above calculation. No other data need to be stored by the PE except that two digits of B are queued ready for use by the PE rather than one. If one ignores the digits of Q and N, this is just what happens in Fig. 1 ([55], Fig. 1), where the black dots in the diagram represent latches to delay the forwarding of digits of B (and N) by one clock tick, so they arrive exactly when required. The diagonal dashed lines in the data dependency diagram indicate the times at which each PE executes its operation for this multiplication. It is clear that the data flow is always in the direction of increasing time.

Now consider the periphery of the array. The input data from B and initial zeros for the save digits $s_{0,j}$ trickle in through the top row of the array. On the other hand, digits from A and initial zeros for the carry digits $c_{i,0}$ come through initialising the right hand column. If there were a left hand column with index n, $\text{PE}_{i,n}$ would use $b_n{=}0$ and $s_{i,n+1}{=}0$. Consequently $\text{PE}_{i,n}$ would simply find $c_{i,n+1}{=}0$ and forward $s_{i+1,n-1}{=}c_{i,n}$ to $\text{PE}_{i+1,n-1}$. As this column does no computation, it can easily be absorbed into the neighbouring column of index $n{-}1$, to yield an $n{\times}n$ array, thereby saving some hardware. So, if the number of digits in B is always bounded above by the number of PEs in a row, it can be assumed that the actual left hand column (of index $n{-}1$) has been slightly modified in this way. In the last row with index $n{-}1$, each $\text{PE}_{n-1,j}$ performs its task as normal, outputting the save digit, which is part of the final result, and forwarding the carry digit to the next PE in the row. The digits p_k of the output $P = A{\cdot}B$ appear at intervals of one or two clock ticks. First, n save values $p_i{=}s_{i+1,-1}$ exit the right hand column processors $\text{PE}_{i,0}$ two clock ticks apart at times $2i{+}1, i{=}0,1,2,\ldots,n{-}1$. As is clear from (17), $s_{i+1,-1}$ does indeed represent a digit coefficient of r^i. Then the remaining n product digits $p_{n-1+j}{=}s_{n,j-1}$ exit the last row processors $\mathrm{PE}_{n-1,j}$ one clock tick apart at times $2n{-}1{+}j, j{=}1,2,\ldots,n$, (except for $p_{2n-1} = c_{n-1,n}$ at time $3n{-}2$ if column n is absorbed into column $n{-}1$) and they are the coefficients of r^{n-1+j}.

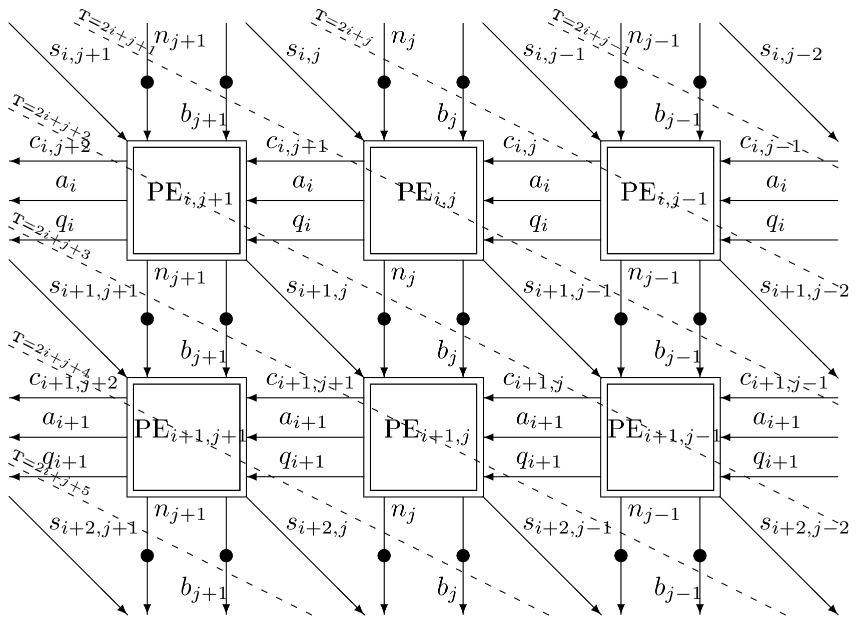

Figure 1: Data Flow between Modular Mult$^{\mathrm{n}}$ Cells ([55], Fig. 1)

It should be reasonably clear that if each PE uses its data at the correct time and the output is collected at the appropriate time, then the array does indeed compute $A{\cdot}B$. Thus, for a given k, each carry digit $c_{i,k-i}$, save digit $s_{i,k-i}$ and product $a_i{\cdot}b_{k-i}$ is processed by $\mathrm{PE}_{i,k-i}$ and contributes to the coefficient p_k of r^k in the product P. The carry it generates represents a multiple of r^{k+1} and is sent to a PE that deals with r^{k+1}. The save it generates represents a multiple of r^k and is sent to the next PE that deals with r^k. So the diagonal sequence of elements $\mathrm{PE}_{0,k}$, $\mathrm{PE}_{1,k-1}$, $\mathrm{PE}_{2,k-2}$, $\ldots$, $\mathrm{PE}_{k,0}$ computes and outputs the final digit p_k of the product, having added in all the necessary digit products from A and B, propagated carries as necessary, and consumed the carries it has been sent. An alternative row-by-row view is given by noting that the equation (17) for a fixed i is just that for a carry-save computation of $a_i{\cdot}B{+}Pr^{-1}$ where P is the output received from the previous row, namely the result of accumulating the inputs to rows 0 to $i{-}1$. The movement of the save digits by one PE diagonally rightwards means P should be weighted by r^{-1}. Thus row $n{-}1$ would compute $\sum_{i=0}^{n-1} a_i r^i B = A{\times}B$ if there were further columns to the right. As those extra columns would use zeros for the digits of B with negative indices, they would not do any computing, and instead just forward on the save digits, which we decided to collect earlier when they left the column of index 0. All carries have been propagated by the time the last save digit exits the array.

Note that one multiplication is performed like a wave travelling from top right to bottom left in the array, with only a diagonal of PEs (those along a dashed line in the figure) being busy performing the multiplication at any specific clock time. The other PEs are free and so can be used for further multiplications while the first one is still being computed. As another multiplication can start being fed into $\mathrm{PE}_{0,0}$ at every clock tick, and it takes $3n-2$ clock ticks before the last digit is output, the array could be performing $3n-2$ multiplications simultaneously, each starting, progressing and being output one clock tick behind the one in front.

3.14.2 Scalability

Still with the multiplication example, it is important to consider its scalability. Typically the hardware resources will be fixed but they must be able to deal with variable sized inputs. So, suppose the input arguments have n digits but the array is of size $s{\times}t$, perhaps with $s \neq t$. If $s, t \geq n$ then there is no problem: A and B are simply extended to s and t digits respectively by adding zero digits at the most significant ends, the product is computed as above, and then the first $2n$ digits of output taken as the product. As before, PEs which have zero digits from the most significant end of A or B simply find themselves with zero for the carries and forward the incoming save digit, which is eventually output. If appropriate, extra control could be put in to extract the result directly from the nth row rather than wait until it exits row s.

However, if $s < n$ but $t \geq n$ then there are insufficient rows and the data output from the last row has only computed $P_s = \sum_{i=0}^{s-1} a_i r^i B$. This is simply fed back into the top row as it appears digit by digit from the bottom row, and the next set of s digits from A is used to add the next set of s products $a_i{\cdot}B$, yielding the output $P_{2s} = \sum_{i=0}^{2s-1} a_i r^i B$. This is repeated until all the digits of A have been used. Thus the array works like a tube by effectively having the top and bottom edges joined, and the data just cycles round and round the tube until all n digits of A have been processed.

The next case is when $t < n$ but $s \geq n$. This works in a similar way by effectively connecting the left and right edges of the array so that the save digits which exit the right side are fed back into the left side and the carry and other digits which flow out of the left side are fed back into the right side. This time, each cycle across the array extends by t digits of B the proportion of each $a_i r^i{\cdot}B$ that has been calculated. For each iteration round the rows, the next set of t digits of B is fed one digit at a time into the top of the array, ready for use when the PEs need them.

In both these cases for each circuit of data round the array only one PE of each row and one PE in each pair of adjacent columns is operating on the multiplication. Consequently, when the data exits from a row or column all the PEs in that row or column are free to be used in the next cycle with the next set of s or t digits.

Things become more complicated in the fourth case, when both $s < n$ and $t < n$. Again, the array needs to be viewed as having its opposite edges joined together, this time joining both pairs of edges. The data cycles round the rows and round the columns as before, but there is the potential for PEs to be busy when needed. In particular, data may travel from $\mathrm{PE}_{0,t-1}$ back to $\mathrm{PE}_{0,0}$ at the same time as data wants to go from $\mathrm{PE}_{s-1,0}$ back to $\mathrm{PE}_{0,0}$. So $\mathrm{PE}_{0,0}$ is in demand from two parties. The straight-forward solution is to queue the data exiting from one edge in a shift register until processors are free to continue the calculation. As memory for holding digits is much cheaper than PEs, and digits in the queue can easily be recovered sufficiently in advance not to hold up the processing, this solution is very cost effective. One just has to be careful that memory access if fast enough to keep up with the demands of the array.

3.14.3 A Linear Systolic Array

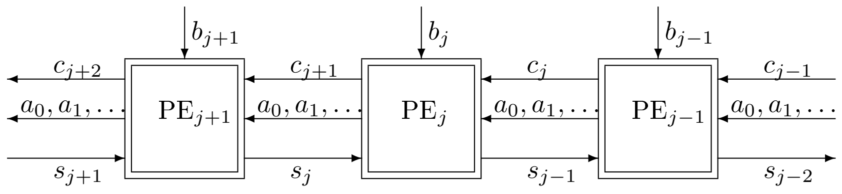

Figure 2: Part of a Linear Systolic Array for Multiplication

A special case is when the multiplication array is only one dimensional, i.e. $s = 1$ or $t = 1$. Assume $s = 1$ and $n \leq t$, so that there is a single row of PEs, as in Fig. 2, and enough PEs to process every digit of B. As described before, the save digits coming out from the last row are fed back into the top row, but at one position to the right. This simply means the save digits go back to the previous PE and it corresponds to a shifting up of the partial product as each new digit a_i is processed. PE_j processes digit b_j as previously for column j, and its computation is

$$c_{j+1}r+s_{j-1} \;\leftarrow\; a_i{\cdot}b_j+s_j+c_j$$

at time $2i+j+1$. This is a contribution to the digit of index $i+j$. Every two clock ticks it needs the next digit from A; hence the stream of digits $a_0, a_1, a_2, \ldots$ which are fed into the right hand end on alternate cycles. At the next clock tick, PE_{j-1} receives s_{j-1}, which is a coefficient of r^{i+j}, and adds in it and the contribution from $a_{i+1}{\cdot}b_{j-1}$, also of weight r^{i+j}. Also at that next clock tick, PE_{j+1} receives c_{j+1}, which is a coefficient of r^{i+j+1}, and adds in both it the product $a_i{\cdot}b_{j+1}$, also of weight r^{i+j+1}. Thus the digit$\times$digit products, the carries and the saves are all added into the correct total for each digit position. Eventually the digits of A run out so that the carries all become zero, and the save digits are passed rightwards for output. At time $2i+1$, $i = 0, 1, \ldots, 2n-1$, PE$_0$ ejects the save digit s_{-1} of weight r^i. As all relevant contributions from $A{\times}B$ have been added to it, this is the value of the digit p_i of the product P.

Because of pressure on chip resources, a likely scenario is that there are many fewer PEs than digits of B, i.e. $t < n$, If the array is indeed too small to hold all the digits of B then, as above, it can compute $A{\times}B_j$ where B_j is a t-digit number. With some simple adjustments, it can be used iteratively to compute $P = A{\times}B = \sum_j A{\times}B_j r^{tj}$ where B_j is a radix r^t digit of B, i.e. t radix-r digits of B. Each iteration uses t more digits from B to generate t more digits of output for the product P, and a carry C of n digits to be used in the next iteration, as follows:

$$Cr^t + P_j \ \leftarrow\ C + A{\times}B_j$$

for $j = 0, 1, \ldots$ This requires initialising the array to the currently incomplete top part C of the product $A{\times}B$. C is just the first n carries which exit the left end of the array on alternate cycles after the first t cycles, and P_j is given by the first t save values exiting the right end on alternate cycles. The n digits of C need to be fed back in in parallel with re-inputting the digits of A, and this replaces the initialisation to zero of the right hand carry-in which is performed only for the first iteration. This detail was subsumed in the discussion above for the two dimensional array.

With this set-up, the leftmost PEs are inactive at the end of a multiplication while the t save digits of the output are passed rightwards down the array. These digits are unchanged by the PEs because t further zero digits are supplied on the A pipeline. However, the save digits could be extracted from the lower edge once the final term involving a_{n-1} has been added. This would allow the following part of the multiplication (that involving the next B_j) or another multiplication to commence immediately, resulting in each PEs being fully used on every alternate cycle.

Of course, with each PE operating only on alternate clock ticks the array can perform a second (independent) multiplication in the other clock cycles, enabling the full use of its computing power.

3.14.4 A Systolic Array for Modular Multiplication

Could the array of §3.14.1 be adapted to perform an interleaved modular multiplication rather than just multiplication? For the classical algorithms the answer is clearly "no" because the first product digits are already exiting the array by the time any top digits are available for the determination of the multiple q of the modulus N which should be subtracted. However, with Montgomery's algorithm the multiple q is determined initially and then the addition of qN can progress hand in hand with the addition of a_iB and the propagation of carries. Figure 1 illustrates how the typical PEs would work. PE$_{i,j}$ now calculates

$$c_{i,j+1}r{+}s_{i+1,j-1} \ \leftarrow\ a_i{\cdot}b_j{+}q_i{\cdot}n_j{+}s_{i,j}{+}c_{i,j} \tag{18}$$

at time $T = 2i{+}j{+}1$. By analogy with the multiplication case, this clearly computes $P = A{\times}B + Q{\times}N$ where $Q = \sum_{i=0}^{n-1} q_i r^i$. If Q has been chosen correctly, P will be a multiple of r^n and the lowest n output digits can be ignored to leave the normal Montgomery modular product.

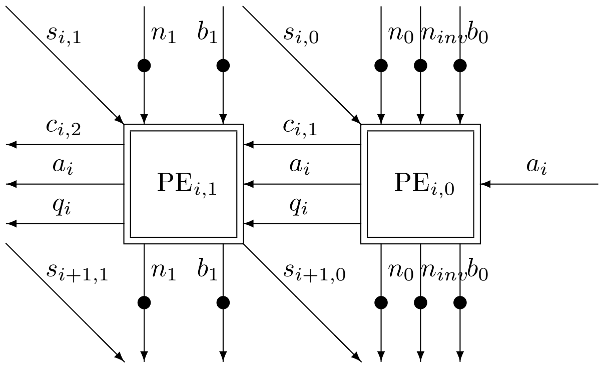

Figure 3: The Rightmost Cells.

The rightmost column of cells needs to compute the digits q_i which cause the partial product to be divisible by r when q_iN is added. So PE$_{i,0}$ must also determine q_i such that its save output $s_{i+1,-1}$ is 0:

$$\begin{aligned} p_{i,0} &\leftarrow a_i{\cdot}b_0{+}s_{i,0} \\ q_i &\leftarrow p_{i,0}{\cdot}n_{inv} \bmod r \\ c_{i,1}r &\leftarrow p_{i,0}{+}q_i{\cdot}n_0 \end{aligned} \tag{19}$$

at time $T = 2i{+}1$, where n_{inv} is the pre-computed inverse of n_0 as defined in (11). Note that q_i is determined after a delay of two multiplications and so should take no longer to generate than the carry output in a standard cell of the array. Although it looks as if $c_{i,1}$ takes longer to compute, a number of methods for simplifying this were discussed earlier in the chapter, including shifting B up so that the first multiplication in (19) is eliminated (§3.10) or scaling N so that the second and third multiplications are removed (§3.13). In an FPGA these would reduce the time taken by PE$_{i,0}$ to equal that of the other cells. Alternatively, in an ASIC systolic array, there are considerable simplifications arising i) from only having to compute the less significant digit of $p_{i,0}$ for input to q_i, ii) from the cancellations due to $n_0{\cdot}n_{inv} = -1 \bmod r$ when calculating $n_0{\cdot}(p_{i,0}{\cdot}n_{inv} \bmod r)$ for input to $c_{i,1}r$, and iii) from only having to compute the more significant digit of $p_{i,0}{+}q_i{\cdot}n_0$.

Suppose, as usual, that n is the number of digits in A, B and N, and the array is large enough for our purposes. Then the leftmost column of cells may need to have an index larger than $n{-}1$ because the intermediate and final values of the product P are bounded above by $2N$, *cf* (21). Moreover, as noted earlier, this leftmost column may forward its carry directly to the next row to avoid having an extra column of PEs for which the normal processing of a cell is trivial.

For the result of the modular multiplication, recall that the first, i.e. lowest, n digits that the multiplication array produced are now ignored. Indeed, they are now zero after the modular reduction, and the output lines for them have been removed. The required digits $p_j = s_{n,j}$ $(j{=}0,1,\ldots)$ of the Montgomery modular product follow at times $2n{-}1{+}j$ $(j{=}0,1,\ldots)$ if the least significant input digits were multiplied at time 1 and the product digits are output, as before, from the nth row of the array. If the array does not have n rows, then adjustments are made in the same way as in §3.14.2 for the multiplication array.

As in the case of the multiplication array, further modular multiplications can be fed serially into the array, each starting one clock cycle after the previous one until every processing element is busy. With the first digit being output at time 1 and the last at time $3n{-}2$, there is the capacity for $3n{-}2$ simultaneous modular multiplications taking place in an $n{\times}n$ array at any one time. When, as is usually the case, there are fewer rows or columns than digits in A or B then, as before, any unused diagonals in the array can be allocated to the next part of a modular product computation. More detail is given in the references cited at the start of this section.

For a system-on-a-chip (SoC), different applications compete for space and layout can be a problem. As die sizes increase, it becomes more and more feasible to use a systolic array with many PEs for modular multiplication, but area is always going to be an issue. A very useful observation to reduce area is simply to implement one rather than two multipliers in each $\mathrm{PE}_{i,j}$ and to split its function over two clock cycles ([57], eqn. 3) using an intermediate double digit variable $d_{i,j}$ which is computed at time $T = 2i{+}j$:

$$\begin{array}{rcl} d_{i,j} & \leftarrow & a_i{\cdot}b_j \\ c_{i,j+1}r{+}s_{i+1,j-1} & \leftarrow & d_{i,j}{+}q_i{\cdot}n_j{+}s_{i,j}{+}c_{i,j} \end{array} \qquad (20)$$

This just requires digits a_i and b_j to be input one clock cycle earlier. The main carry and save digits are processed as before at time $2i{+}j{+}1$. The multiplier must now be able to add in an extra variable for which more register space is required, and an extra control bit is needed to distinguish between the two operations of the PE but, apart from that, the area is reduced by a factor of almost 2. This makes it much easier to make full use of all the multipliers on every cycle in the linear version of the array. In particular, when performing an exponentiation in the linear (i.e. one dimensional) version of the array, the squarings and any necessary multiplications can be performed sequentially with no PE being idle. Moreover, the layout of a linear array can be much more easily adapted to any odd-shaped area on the chip. [57] provides more detail and more options for such arrays, including a discussion of applications to elliptic curve cryptography.

There are alternative formulations of the normal schoolbook method of multiplying two numbers, and some can be adapted to interleaved modular reductions. For example, Kornerup [34] adapts [55] to a multiplier design in which two digit×digit multiplications of $A{\cdot}B$ are performed by each PE. However, their extra complexity does not seem to provide an improved measure of Area×Time.

3.15 Side Channel Concerns and Solutions

One of the advantages of listening to digital rather analogue radio stations is that switching on a light or a nearby thunder storm no longer interferes with the quality of reception. However, this electro-magnetic phenomenon, when an electric current is suddenly switched on or off, led to the discovery of radio waves and the invention of radio communication at the end of the 19th century.

During the cold war it was well-appreciated that current variation in valves and cathode ray tube (CRT) monitors led to radio emanations which leaked information from electrical apparatus such as computers, teletypes and telephones. This led to Tempest shielding [41, 7, 52]. To a large extent, this requires putting everything in a metal box or at least within a metal lattice − including cables − but it also concerns input current variation, and even sound[10]. Considerable understanding and expertise was developed by government bodies for analysing this "side channel" leakage, undoubtedly making use of advanced statistical methods, very powerful probing tools, and unlimited computer facilities which are still well beyond the funding means of university researchers.

Of course, the transistors used in chips are just switching devices and therefore radiate energy during operation just as valves do. This was also well-known from their invention. The need for counter-measures to this lower level of leakage has been known for many years [51]. However, the earliest unclassified published demonstrations of successful attacks on cryptographic systems using side channel leakage are due to Paul Kocher [32, 33]. He used timing measurements of operations deduced from power variations, and the profile of power use during clock cycles to determine the secret keys within a device running algorithms for public key and symmetric key cryptography. Without any countermeasures, this just required running the cryptographic operation many times with the same data and averaging the results to reduce the noise sufficiently to distinguish between properties of different secret keys. The main difficulty in performing a similar attack using EMR measurements (electro-magnetic radiation) was the manufacture of a sufficiently small antenna − but these are readily available in the heads used to read hard disks, for example. Success in this field of EMR was demonstrated shortly after Kocher's publications by Quisquater, Gemplus and others [44, 16].

The laboratory equipment required to perform such timing, power or EMR measurements is not expensive, and consists mainly of an oscilloscope and a probe. Moreover, understanding of the cause of the leakage is not always necessary. For such attacks it suffices to find a correlation between the secret key and any parts of the recorded phenomena. Averaging many oscilloscope traces is generally important to improve the signal-to-noise ratio (SNR), but it is essential to select the traces to amplify the signal rather than reduce it. Some of the more sophisticated methods for trace selection require some knowledge

[10]As well as the sounds made by different keys on a keyboard or key pad, CPUs running at low clock speeds used to make the case of a PC vibrate sufficiently to be able to hear when it was performing RSA cryptography even without any listening equipment beyond the human ear.

of how the cryptographic algorithms are implemented in order to target times when leakage may occur, such as when key-dependent material is moved along a bus, and remove parts of the traces with no key-related information.

In his first paper [32], Kocher identified conditional subtractions in modular multiplications as a primary source of timing variations which revealed the secret key. Bit-by-bit he reconstructed the secret exponent in RSA by observing the frequency of the subtractions and choosing the next bit to match the observed frequency. Kocher used known data inputs. So the immediate counter-measure was to blind the data fed into RSA [6, 32, 37]. However, this does not solve the problem since the average behaviour of squarings and multiplications enables them to be distinguished without knowledge of any inputs, and the sequence of squarings and multiplications is directly related to the exponent bits when the usual square-and-multiply exponentiation algorithm is used.

$A \setminus B$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	**1**	**3**
3	0	3	**1**	**4**	**2**
4	0	4	**3**	**2**	**1**

Table 1: Products of residues A, B mod 5.

The theoretical basis for this was first published by Schindler [47] and Walter & Thompson [61] and is easily illustrated by looking at a small modulus such as $N = 5$, as in Fig. 3.15 where the reduced products are in bold. If the reduction is done for results equal to, or above, N the ratio of subtractions for multiplications to squarings is $\frac{8}{25} : \frac{2}{5} = \frac{4}{5}$ in the example when we assume the N^2 possible products occur with equal frequency, and the N possible squares also appear with equal frequency. Simply put, the average value for a square is greater than the average value for a product and so the probability of a subtraction is greater for a square.

For the large moduli of cryptographic applications, the exact frequencies of the final conditional subtraction in line 8 of Algorithm 0 can be determined straight-forwardly as in [60]. It requires a precise bounding interval for the output P of the main loop of Montgomery's algorithm. This is given by

$$ABR^{-1} \leq P < N{+}ABR^{-1} \tag{21}$$

which is easily verified by showing that line 3 of Algorithm 0 contributes ABR^{-1} to P and line 5 contributes less than N [59]. When A and B are less than N and $N < R$, this is a sub-interval of $[0, 2N]$ with length N so that at most one subtraction is required. In cryptographic applications, it is reasonable to assume N is large and prime or almost prime (i.e. all or almost all natural numbers less than N are prime to N), that A and B are uniformly distributed modulo N, and hence that P is uniformly distributed modulo N, or almost so, and so is effectively uniformly distributed over the given interval. In that case, the probability of the conditional subtraction is proportional to the size of the interval $[N, N+ABR^{-1}]$, i.e. the probability is $ABR^{-1}N^{-1}$. Integrating this with respect to A and B over the range $[0, N]$ then gives a very accurate value for the probability of the conditional subtraction for a multiplication, namely $\frac{1}{4}NR^{-1}$. Identifying A and B and integrating over A gives the probability for a squaring as $\frac{1}{3}NR^{-1}$. Fixing A and letting B occur with uniform distribution gives the probability of the subtraction for a constant multiplication by A, namely $\frac{1}{2}AR^{-1}$. Similar distinguishing probabilities hold for alternative implementations in which the condition for subtracting N is $P \geq r^n$ rather than $P \geq N$ and inputs and outputs are bounded above by r^n rather than N.

So the frequency of conditional subtractions is indeed measurably greater for squarings than for multiplications when sufficient observations are made. Constant multiplications can also be identified from each other and from squarings and general multiplications by the frequencies for them. Such frequencies reveal the sequence of multiplications, squarings and constant multiplications in any exponentiation which uses a standard algorithm, such as the binary and m-ary left-to-right and right-to-left algorithms. This leads to the discovery of the secret key [60].

Consequently, where side channel attacks might be a concern it is necessary to adopt a version of Montgomery's algorithm which does not involve a conditional subtraction. This was presented in [5], §2.4. The solution is to set a bound greater than N, say $\mathcal{B}$, such that all inputs to the algorithm will be less than $\mathcal{B}$ and simply perform a large enough fixed number of iterations of the main loop to ensure the output is also less than $\mathcal{B}$. Since R is increased by a factor of r by each extra iteration, it is possible to ensure

$$R \;\geq\; \frac{\mathcal{B}^2}{\mathcal{B}-N} \tag{22}$$

and then the main loop output P is bounded by $\mathcal{B}$ because of (21). This can then be used safely in any future modular multiplication. A typical choice is $\mathcal{B} = 2N$. This requires $R \geq 4N$ and therefore just one more iteration of the main loop in most situations[11]. At the end of an exponentiation it also means at most one conditional subtraction of N. However, as is readily verified by setting $A = 1$ in (21), this subtraction can never occur if a modular multiplication by 1 is performed to retrieve the result from the Montgomery domain [56, 20, 59]. Thus, there is no need to implement subtraction in any part of a modular exponentiation using Montgomery's algorithm, and that means hardware savings on an ASIC.

Given that execution time should be independent of secret data in order to decrease side channel leakage, it should also be noted that compilers may optimise ROM code to eliminate unnecessary multiplications by zero and additions

[11]So it would be more efficient and secure if standard key and word sizes led to $R \geq 4N > \frac{1}{2}R$, giving the smallest number of iterations such that $R > N$, but the world is not always ideal.

of zero. Especially in the case of small radices r, this might enable sufficient instances of $q = 0$ to be detected and used to recover a secret key if the input message has not been blinded.

For comparison, there are no published claims of comparable leakage from a classical modular multiplication algorithm if the inputs are blinded, but it is necessary, of course, to implement subtraction. If the inputs are not blinded in this traditional case, the exponent can be re-created bit-by-bit by reproducing the exponentiation and choosing bits to match the observed leakage, as Kocher did [32].

Besides the removal of conditional subtractions from a modular multiplication algorithm, it should be clear that there is a need to protect other aspects of cryptographic exponentiation implementations from critical side channel leakage. In particular, squarings should ideally be made to behave in the same way as multiplications. This may mean fetching an argument from memory for a squaring even if it is unused or already present in a register, in order to match behaviour for a multiplication. It may also mean taking action to ensure the two arguments of the squaring are suitably modified so that they appear to be as different as in a multiplication when passing along a bus or when used in a multiplier. Such counter-measures are advisable since single exponentiations can be attacked [58] – it may be unnecessary to perform many exponentiations in order to have sufficient sections of leakage trace to average and achieve a good signal-to-noise ratio. Often a sufficient solution is to randomise the exponent so that averaging over many uses of the secret key removes rather than reveals data dependency in the observed signal.

Side channel leakage, together with low power, is one of the major concerns for implementers of hardware for cryptography. It is a vast subject. We have merely touched on how Montgomery modular multiplication is affected and provided a counter-measure to one point of leakage, namely the conditional subtraction. With a modification to remove that data-dependent subtraction, other attacks on modular arithmetic in cryptographic hardware, such as active ones involving fault injection, seem not to be specific to the choice of modular multiplication algorithm. Counter-measures to them are then typically generic and applicable to any choice of modular multiplication algorithm.

3.16 Logic Gate Technology

Finally, remember that using static CMOS gates is not the only choice for building circuits. Apart from other considerations, power and side channel leakage through power variation and electromagnetic radiation (EMR) may be reduced by using alternative technologies. Research into this has not identified any particular style as solving such problems fully. For example, Pass-Transistor logic (PT) might be used to reduce power. However, at a minimum, a complete solution to side channel leakage requires removing the possibility of hardware glitches (which are data dependent and cause power surges), balancing the amount and delay of charging (loading capacitance) and equalising gate switching (transition counts) to make them independent of input values at every level of the circuit. Even if it were possible, this is clearly wildly inconsistent with any desire for area efficiency or low power. A lower target is to balance the total energy used over a clock cycle to make it nominally the same for all inputs of a given program instruction. The most promising attempts at addressing this problem use a Dual-Rail Pre-charge (DRP) logic style, such as Sense Amplifier Based Logic (SABL) [49] or Wave Dynamic Differential Logic (WDDL) [50]. So far, results show these to be expensive and only partially effective for mitigating the level of leakage.

Overall, the clearest leakage comes from data sent along the bus, and next is probably any data which is broadcast widely at the same time to different processing elements through multiplexers. Generally, depending on the logic, it is the Hamming weight of the data or the Hamming weight of the difference between successive data values which can be determined most easily. Thus, choice of logic gate technology depends not just on the digit multiplier but on wider considerations.

3.17 Conclusion

Peter Montgomery's *Modular Multiplication without Trial Division* [39] has had a significant effect on the design and efficiency of hardware for arithmetic-based cryptography as well as providing commercial advantages arising from simpler implementation when compared with traditional methods. In particular, quotient digits stay within the normal range for non-redundant representations and carry propagation does not need to occur before quotient digit selection. Furthermore, the direction of carry propagation away from the locus of quotient digit calculation means that Montgomery's algorithm is a natural choice for systolic arrays which can perform very efficiently the highest volumes of decryption and digital signing needed on SSL servers.

References

[1] H. Aigner, H. Bock, M. Hütter, and J. Wolkerstorfer, *A low-cost ECC coprocessor for smartcards*, in CHES 2004, M. Joye and J.-J. Quisquater, eds., vol. 3156 of LNCS, Cambridge, Massachusetts, USA, Aug. 11–13, 2004, Springer, Berlin, Germany, pp. 107–118.

[2] P. Barrett, *Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor*, in CRYPTO'86, A. M. Odlyzko, ed., vol. 263 of LNCS, Santa Barbara, CA, USA, Aug. 1986, Springer, Berlin, Germany, pp. 311–323.

[3] G. Bertoni, J. Guajardo, and G. Orlando, *Systolic and scalable architectures for digit-serial multiplication in fields GF(p^m)*, in INDOCRYPT 2003, T. Johansson and S. Maitra, eds., vol. 2904 of LNCS,

New Delhi, India, Dec. 8–10, 2003, Springer, Berlin, Germany, pp. 349–362.

[4] J. W. Bos and A. K. Lenstra, eds., *Topics in Computational Number Theory inspired by Peter L. Montgomery*, Cambridge University Press, Cambridge, UK, 2017.

[5] J. W. Bos and P. L. Montgomery, *Montgomery Arithmetic from a Software Perspective, (Cryptology ePrint Archive Report 2017/1057)*, in Bos and Lenstra [4], 2017, ch. 2.

[6] D. Chaum, *Blind signatures for untraceable payments*, in CRYPTO'82, D. Chaum, R. L. Rivest, and A. T. Sherman, eds., Santa Barbara, CA, USA, 1982, Plenum Press, New York, USA, pp. 199–203.

[7] R. L. Dennis, *Security in the Computing Environment*, Tech. Report SP2440/000/01, System Development Corporation, August 18 1966. (page 16).

[8] J. Dhem, *Modified version of the Barrett algorithm*, tech. report, DICE, Université Catholique de Louvain, 1994.

[9] ——, *Design of an efficient public-key cryptographic library for RISC-based smart cards*, PhD thesis, Université Catholique de Louvain, 1998.

[10] J. Dhem and J. Quisquater, *Recent Results on Modular Multiplications for Smart Cards*, in Smart Card Research and Applications, CARDIS '98, vol. 1820 of LNCS, Springer-Verlag, 1998, pp. 336–352.

[11] W. Diffie and M. E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, 22 (1976), pp. 644–654.

[12] S. R. Dussé and B. S. Kaliski Jr., *A cryptographic library for the Motorola DSP56000*, in EUROCRYPT'90, I. Damgård, ed., vol. 473 of LNCS, Aarhus, Denmark, May 21–24, 1990, Springer, Berlin, Germany, pp. 230–244.

[13] S. E. Eldridge and C. D. Walter, *Hardware Implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Transactions on Computers, 42 (1993), pp. 693–699.

[14] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory, 31 (1985), pp. 469–472.

[15] W. L. Freking and K. K. Parhi, *Performance-Scalable Array Architectures for Modular Multiplication*, Journal of VLSI Signal Processing, 31 (2002), pp. 101–116.

[16] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic analysis: Concrete results*, in Koç et al. [30], pp. 251–261.

[17] H. L. Garner, *The Residue Number System*, in Papers Presented at the March 3-5, 1959, Western Joint Computer Conference, IRE-AIEE-ACM '59 (Western), New York, NY, USA, 1959, ACM, pp. 146–153.

[18] J. Großschädl, R. M. Avanzi, E. Savas, and S. Tillich, *Energy-efficient software implementation of long integer modular arithmetic*, in CHES 2005, J. R. Rao and B. Sunar, eds., vol. 3659 of LNCS, Edinburgh,

UK, Aug. 29 – Sept. 1, 2005, Springer, Berlin, Germany, pp. 75–90.

[19] J. Großschädl and G.-A. Kamendje, *Architectural enhancements for Montgomery multiplication on embedded RISC processors*, in ACNS 03, J. Zhou, M. Yung, and Y. Han, eds., vol. 2846 of LNCS, Kunming, China, Oct. 16–19, 2003, Springer, Berlin, Germany, pp. 418–434.

[20] G. Hachez and J.-J. Quisquater, *Montgomery exponentiation with no final subtractions: Improved results*, in Koç and Paar [31], pp. 293–301.

[21] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[22] Intel Corporation, *Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications, Version 2.0*, Tech. Report AP-941, Intel, 2000. http://software.intel.com/sites/default/files/14/4f/24960.

[23] M. Joye, *On Quisquater's Multiplication Algorithm*, in Cryptography and Security: From Theory to Applications, D. Naccache, ed., vol. 6805 of LNCS, Springer-Verlag, 2012, pp. 3–7.

[24] A. Karatsuba and Y. Ofman, *Multiplication of Many-Digital Numbers by Automatic Computers*, Doklady Akad. Nauk SSSR, 145 (1962), pp. 293–294. Translation in Physics-Doklady **7**, pp. 595–596, 1963.

[25] P. S. Kasat, D. S. Bilaye, H. V. Dixit, R. Balwaik, and A. Jeyakumar, *Multiplication Algorithms for VLSI – A Review*, International Journal on Computer Science and Engineering (IJCSE), 4 (2012), pp. 1761–1765.

[26] M. Knežević, F. Vercauteren, and I. Verbauwhede, *Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods*, IEEE Transactions on Computers, 59 (2010), pp. 1715–1721.

[27] N. Koblitz, *Elliptic Curve Cryptosystems*, Mathematics of Computation, 48 (1987), pp. 203–209.

[28] Ç. K. Koç and T. Acar, *Montgomery Multiplication in GF(2^k)*, Designs, Codes and Cryptography, 14 (1998), pp. 57–69.

[29] Ç. K. Koç, T. Acar, and B. S. Kaliski, Jr., *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, 16 (1996), pp. 26–33.

[30] Çetin Kaya. Koç, D. Naccache, and C. Paar, eds., *CHES 2001*, vol. 2162 of LNCS, Paris, France, May 14–16, 2001, Springer, Berlin, Germany.

[31] Çetin Kaya. Koç and C. Paar, eds., *CHES 2000*, vol. 1965 of LNCS, Worcester, Massachusetts, USA, Aug. 17–18, 2000, Springer, Berlin, Germany.

[32] P. C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, in CRYPTO'96, N. Koblitz, ed., vol. 1109 of LNCS, Santa Barbara, CA, USA, Aug. 18–22, 1996, Springer, Berlin, Germany, pp. 104–113.

[33] P. C. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, in CRYPTO'99, M. J. Wiener, ed., vol. 1666 of LNCS, Santa Barbara, CA, USA, Aug. 15–19, 1999, Springer, Berlin, Germany, pp. 388–397.

[34] P. Kornerup, *A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms*, IEEE Transactions on Computers, 43 (1994), pp. 892–898.

[35] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer, *Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller*, in CHES 2006, L. Goubin and M. Matsui, eds., vol. 4249 of LNCS, Yokohama, Japan, Oct. 10–13, 2006, Springer, Berlin, Germany, pp. 430–444.

[36] Z. Liu and J. Großschädl, *New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers*, in AFRICACRYPT 14, D. Pointcheval and D. Vergnaud, eds., vol. 8469 of LNCS, Marrakesh, Morocco, May 28–30, 2014, Springer, Berlin, Germany, pp. 215–234.

[37] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, *Power analysis attacks of modular exponentiation in smartcards*, in CHES'99, Çetin Kaya. Koç and C. Paar, eds., vol. 1717 of LNCS, Worcester, Massachusetts, USA, Aug. 12–13, 1999, Springer, Berlin, Germany, pp. 144–157.

[38] V. S. Miller, *Use of elliptic curves in cryptography*, in CRYPTO'85, H. C. Williams, ed., vol. 218 of LNCS, Santa Barbara, CA, USA, Aug. 18–22, 1985, Springer, Berlin, Germany, pp. 417–426.

[39] P. L. Montgomery, *Modular Multiplication Without Trial Division*, Mathematics of Computation, 44 (1985), pp. 519–521.

[40] National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS)*, Tech. Report FIPS Publication 186-4, July 2013.

[41] National Security Agency (NSA), *Compromising Emanations Laboratory Test Requirements, Electromagnetics (U)*, Tech. Report National COMSEC Information Memorandum (NACSIM) 5100A, NSA, 1981. (Classified).

[42] G. Orlando and C. Paar, *A scalable GF(p) elliptic curve processor architecture for programmable hardware*, in Koç et al. [30], pp. 348–363.

[43] K. Pabbuleti, D. Mane, A. Desai, C. Albert, and P. Schaumont, *SIMD Acceleration of Modular Arithmetic on Contemporary Embedded Platforms*, in IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2013, pp. 1–6.

[44] J. Quisquater and D. Samyde, *ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards*, in Smart Card Programming and Security, E-smart 2001, Cannes, France, September 19-21, 2001, I. Attali and T. Jensen, eds., vol. 2140 of LNCS, Springer-Verlag, 2001, pp. 200–210.

[45] 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 (1978), pp. 120–126.

[46] E. Savas, A. F. Tenca, and Çetin Kaya. Koç, *A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m)*, in Koç and Paar [31], pp. 277–292.

[47] W. Schindler, *A timing attack against RSA with the chinese remainder*

theorem, in Koç and Paar [31], pp. 109–124.

[48] H. Seo, Z. Liu, J. Grossschädl, J. Choi, and H. Kim, *Montgomery Modular Multiplication on ARM-NEON Revisited*, in ICISC 2014, vol. 8949 of LNCS, Springer-Verlag, 2015, pp. 328–342.

[49] K. Tiri, M. Akmal, and I. Verbauwhede, *A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards*, in European Solid-State Circuits Conference – ESSCIRC 2002, Florence, 24–26 Sept. 2002, Università di Bologna, 2002, pp. 403–406.

[50] K. Tiri and I. Verbauwhede, *A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation*, in Design, Automation and Test in Europe Conference and Exposition – (DATE 2004), Paris, 16–20 February 2004, IEEE, 2004, pp. 246–251.

[51] M. Ugon, *Portable data carrier including a microprocessor.* US Patent and Trademark Office, July 8 1980.

[52] W. van Eck, *Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?*, Computers and Security, 4 (1985), pp. 269–286.

[53] C. D. Walter, *Faster modular multiplication by operand scaling*, in CRYPTO'91, J. Feigenbaum, ed., vol. 576 of LNCS, Santa Barbara, CA, USA, Aug. 11–15, 1991, Springer, Berlin, Germany, pp. 313–323.

[54] C. D. Walter, *Fast Modular Multiplication using 2-Power Radix*, International J. Computer Mathematics, 39 (1991), pp. 21–28.

[55] ——, *Systolic Modular Multiplication*, IEEE Transactions on Computers, 42 (1993), pp. 376–378.

[56] ——, *Montgomery Exponentiation Needs No Final Subtractions*, Electronics Letters, 35 (1999), pp. 1831–1832.

[57] ——, *An Improved Linear Systolic Array for Fast Modular Exponentiation*, IEE Computers and Digital Techniques, 147 (2000), pp. 323–328.

[58] C. D. Walter, *Sliding windows succumbs to Big Mac attack*, in Koç et al. [30], pp. 286–299.

[59] ——, *Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli*, in CT-RSA 2002, B. Preneel, ed., vol. 2271 of LNCS, San Jose, CA, USA, Feb. 18–22, 2002, Springer, Berlin, Germany, pp. 30–39.

[60] ——, *Longer keys may facilitate side channel attacks*, in SAC 2003, M. Matsui and R. J. Zuccherato, eds., vol. 3006 of LNCS, Ottawa, Ontario, Canada, Aug. 14–15, 2003, Springer, Berlin, Germany, pp. 42–57.

[61] C. D. Walter and S. Thompson, *Distinguishing exponent digits by observing modular subtractions*, in CT-RSA 2001, D. Naccache, ed., vol. 2020 of LNCS, San Francisco, CA, USA, Apr. 8–12, 2001, Springer, Berlin, Germany, pp. 192–207.

Subject Index