Long Modular Multiplication for Cryptographic Applications[1]

Laszlo Hars

Seagate Research, 1251 Waterfront Place, Pittsburgh PA 15222, USA
Laszlo@Hars.US

Abstract. A digit-serial, multiplier-accumulator based cryptographic co-processor architecture is proposed, similar to fix-point DSP's with enhancements, supporting long modular arithmetic and general computations. Several new "column-sum" variants of popular quadratic time modular multiplication algorithms are presented (Montgomery and interleaved division-reduction with or without Quisquater scaling), which are faster than the traditional implementations, need no or very little memory beyond the operand storage and perform squaring about twice faster than general multiplications or modular reductions. They provide similar advantages in software for general purpose CPU's.

Keywords: Computer Arithmetic, Cryptography, Modular multiplication, Modular reduction, Montgomery multiplication, Quisquater multiplication, Optimization, Multiply-accumulate architecture, Reciprocal

1 Introduction

Long exponentiation based cryptographic operations are performed infrequently in secure client devices like smart cards or OSD disk drives [15] and it is not economical to include a large piece of HW dedicated only to that task. A DSP-like architecture with minor enhancements for speeding up long modular arithmetic can be used for many other tasks, too, providing an almost free long modular arithmetic engine. A digit-serial, multiplier-accumulator based cryptographic co-processor architecture is proposed, like fix-point DSP's, with inexpensive enhancements for speeding up long modular arithmetic.

Internal fast memory is expensive (storage for eight 2K-bit integers costs about twice as much as the whole arithmetic engine), but only a portion of this memory is needed for processing other than RSA or Diffie-Hellman type operations, so we try to keep this memory small. Several new variants of popular modular multiplication algorithms are presented (Montgomery and interleaved division-reduction with-, or without Quisquater scaling), which either don't need extra memory beyond the parameters or, for extra speed, use one or two additional pre-computed parameters of the size of the modulus. All of these algorithms perform squaring about twice faster than general multiplications and modular reductions. The speed and memory usage advantages of these algorithms are preserved for SW for general purpose CPU's as well.

[1] The full version of the paper is at: http://www.hars.us/Papers/ModMult.pdf

M. Joye and J.-J. Quisquater (Eds.): CHES 2004, LNCS 3156, pp. 45–61, 2004.

1.1 Notations

- Long integers are denoted by $A = \{a_{n-1}\ldots a_1 a_0\} = \sum d^i a_i$ in a d-ary number system, where a_i ($0 \le a_i \le d-1$) are called the **digits**. We consider here $d = 2^{16} = 65{,}536$, that is 16 bits, but our results directly translate to other digit sizes, like $d = 2^{32}$
- $|A|$ or $|A|_d$ denotes the number of digits, the length of a d-ary number
- $[x]$ stands for the integer part of x.
- `D0(Q)` refers to the Least Significant digit (digit 0) of an integer or accumulator `Q`
- `MS(Q)` refers to the content of the accumulator `Q`, shifted to the right by one digit
- LS stands for **L**east **S**ignificant, the low order bit/s or digit/s of a number
- MS stands for **M**ost **S**ignificant, the high order bit/s or digit/s of a number
- (*Grammar*) *School multiplication*: the digit-by-digit-multiplication algorithms, as taught in schools. It has 2 variants in the order the digit-products are calculated:
  - Row-order: $\texttt{for}_{i=0\ldots|a|-1}\ \texttt{for}_{j=0\ldots|b|-1}\ \ldots a_i b_j \ldots$
  - Column-order: $\texttt{for}_{k=0\ldots|a|+|b|-2}\ \texttt{for}_{i,j:\,i+j=k}\ \ldots a_i b_j \ldots$

2 Computing Architecture

Our experimental HW is clocked at 150 MHz. It is designed around a 16×16-bit single cycle **multiplier**. This is the largest, the most expensive part of the HW after the memory (0.13μm CMOS: 16,000 μm^2). Such circuits have been designed for 300 MHz clock rate and beyond in even for CMOS 0.18 μm technology, often used in smart cards, but they are large and expensive. There are much faster or longer multipliers at smaller feature sizes, like the 370 MHz 36×36-bit multipliers in ALTERA FPGA's [1].

We concentrate on algorithms, which accumulate digit-products in column order, although many of the speed-up techniques are applicable to row multiplications, too [17]. The digits of both multiplicands are constantly reloaded to the multiplier: $C = A \cdot B = \{a_0b_0,\ a_1b_0+a_0b_1,\ a_2b_0+a_1b_1+a_0b_2\ldots\}$. The digit-products are summed in columns. High-speed 40-bit CMOS adders have 5 gates delay, can perform 4 additions in a single clock cycle with time for loading and storing the results in RAM. After a digit-multiplication the 32-bit result is added to the accumulator and the multiplier starts working on the next product. When all digit-products are accumulated for the current digit of C the LS digit from the accumulator is shifted out and stored. By feeding the multiplier and accumulator constantly we achieve sustained single cycle digit multiply-accumulate operations.

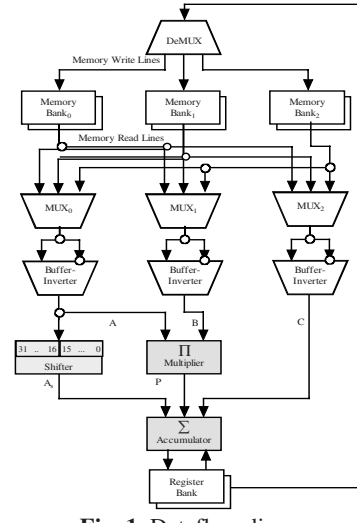

Fig. 1. Dataflow diagram

Memory. To avoid access conflicts separate RAM banks are needed for the operands and the result. Optimal space efficiency requires the ability to change the value of a

digit, i.e., read and write to the same memory location within the same clock cycle. It can be done easily at low speed (smart cards), but faster systems need tricky designs. The algorithms presented here would benefit from this read/write capability, which is only needed in one memory bank, holding the temporary results.

At higher clock frequencies some data buffering, pipelining is necessary to run the algorithms at full speed, causing address *offsets* between the read and write phase of the memory update. Because the presented algorithms access their data *sequentially*, *circular* addressing can be used. The updated digit is written to the location where the next digit is being read from, not where the changed digit was originated from. At each update this causes a shift of loci in data storage. The corresponding circular address offset has to be considered at the next access to the data.

This way, one address decoder for this memory bank is enough, with some registers and a little faster memory cells – still less expensive than any other solution. However, it is a custom design; no off-the-shelf (library) memory blocks can be used.

A dual ported memory bank offer simultaneous read/write capabilities, but two single ported RAM banks cost the same and provide double the storage room and more options to optimize algorithms. With 2 banks, data is read from one and the changed data is written to another memory bank, not being accessed by the read logic.

The **accumulator** consists of a couple of 50-bit registers (48 data bits with 2 additional bits for sign and carry/borrow), a barrel shifter and a fast 48-bit adder. It can be laid out as a 32-bit adder and an 18-bit counter for most of our algorithms, performing together up to 3 additions in a clock cycle. One of the inputs is from a 50-bit 2's complement internal register, the other input is selected from the memory banks (can be shifted), from the multiplier or from a (shifted) accumulator register. At the end of the additions, the content of the register can be shifted to the left or (sign extended) to the right and/or the LS digit stored in RAM.

Since only the LS digit is taken out from the accumulator, it can still work on carry propagation while the next addition starts, allowing cheaper designs (0.13μm CMOS: 2,000μm^2, 12.5% of the area of the multiplier). Some algorithms speed up with one or two internal shift-add operations, effectively implementing a fast multiplier with multiplicands of at most 2 nonzero bits (1, 2, 3, 4, 5, 6; 8, 9,10; 12; 17…).

3 Basic Arithmetic

Addition/Subtraction: In our algorithms digits are generated, or read from memory, from *right to left* to handle carry propagation. The sign of the result is not known until all digits are processed; therefore, 2's complement representation is used.

Multiplication: Cryptographic applications don't use negative numbers; therefore our digit-multiplication circuit performs only unsigned multiplications. The products are accumulated (added to a 32…50-bit register) but only single digits are extracted from these registers and stored in memory.

For operand sizes in cryptographic applications the school multiplication is the best, requiring simple control. Some speed improvement can be expected from the more complicated Karatsuba method, but the Toom-Cook 3-way (or beyond) multi-

plication is actually slower for these lengths [6]. An FFT based multiplication takes even longer until much larger operands (in our case about 8 times slower).

Division: The algorithms presented here compute long divisions with the help of short ones (one- or two-digit divisors) performed by multiplications with reciprocals. The reciprocals are calculated with only a small constant number of digit-operations. In our test system linear approximations were followed by 3 or 4 Newton iterations. [9]

4 Traditional Modular Multiplication Algorithms

We assume $m = \{m_{n-1}\, m_{n-2} \ldots m_0\}$ is normalized, that is $\frac{1}{2}d \le m_{n-1} < d$ or $\frac{1}{2}d^{n-1} \le m < d^n$. It is normally the case with RSA moduli. If not, we have to normalize it: replace m with $2^k m$. A modular reduction step (discussed below) fixes the result: having $R_k = a \bmod 2^k m$ calculated, $R \leftarrow R_k - q \cdot m$, where q is computed from the leading digits of R_k and $2^k m$. These de/normalization steps are only performed at the beginning and end of the calculations (in case of an exponentiation chain), so the amortized cost is negligible.

There are a basically 4 algorithms used in memory constrained, digit serial applications (smart card, secure co-processors, consumer electronics, etc.): Interleaved row multiplication and reduction, Montgomery, Barrett and Quisquater multiplications. [1], [3], [12], [13]. We present algorithmic and HW speedups for them, so we have to review their basic versions first.

Montgomery multiplication. It is simple and fast, utilizing right-to-left divisions (sometimes called exact division or odd division [7]). In this direction there are no problems with carries (which propagate away from the processed digits) or with estimating the quotient digit wrong, so no correction steps are necessary. This gives it some 6% speed advantage over Barrett's reduction and more than 20% speed advantage over division based reductions [3]. The traditional Montgomery multiplication calculates the product in "row order", but it still can take advantage of a speedup for squaring. (This is commonly believed not to be the case, see e.g. in [11], Remark 14.40, but the trick of Fig. 7 works here, too.) The main *disadvantage* is that the numbers have to be converted to a special form before the calculations and fixed at the end, that is, significant pre- and post-processing and extra memory is needed.

```
for i = 0 … n-1
    t = x_i·m' mod d
    x = x + t·m·d^i
x = x / d^n
if( x ≥ m )
    x = x – m
```

Fig. 2. Montgomery reduction of x

In Fig. 2 the Montgomery reduction is described. The constant $m' = -m_0^{-1} \bmod d$ is a pre-calculated single digit number. It exists if m_0 is odd, which is the case in cryptography, since m is prime or a product of odd primes.

The rational behind the algorithm is to represent a long integer a, $0 \le a < m$, as aR mod m with the constant $R = d^n$. A modular product of these (aR)(bR) mod m is not in the proper form. To correct it, we have to multiply with R^{-1}, because (aR)(bR)R^{-1} mod m = (ab)R mod m. This

```
x = 0^n
for i = 0 … n-1
    t = (x_0+a_ib_0)m' mod d
    x = (x + a_ib + t·m)/d
if( x ≥ m )
    x = x – m
```

Fig. 3. Montgomery multiplication

$x \to x\mathrm{R}^{-1} \bmod m$ correction step is called ***Montgomery reduction***, taking about the same time as the school multiplication.

The product AB can be calculated interleaved with the reduction, called the ***Montgomery multiplication*** (Fig. 3). It needs squaring-speedup as noted above. The instruction **x = (x+a$_i$b+t·m) /d** is a loop through the digits of B and *m* from right to left, keeping the carry propagating to the left.

Barrett multiplication. It can take advantage of double speed squaring, too, but calculates and stores the quotient $q = [ab/m]$, although it is not needed later. (During the calculations q and the remainder r cannot be stored in the same memory bank.) To avoid slow divisions, Barrett uses μ, a pre-computed reciprocal of m. These result in an extra storage need of $2n$ digits.

```
(A1d^n + A0) ← a b
q ← MS n digits of A1μ
r ← A0 - LS n digits of qm
if r < 0: r ← r + d^(n+1)
while r ≥ m: r ← r - m
```

Fig. 4. Barrett's multiplication

The modular reduction of a product is $ab \bmod m = ab - [ab/m]\, m$. Division by m is slower than multiplication, therefore $\mu = 1/m$ is calculated beforehand to multiply with. It is scaled to make it suitable for integer arithmetic, that is, $\mu = [d^{2n}/m]$ is calculated (μ is 1 bit longer than m). Multiplying with that and keeping the most significant n bits only, the error is at most 2, compared to the larger result of the exact division. The subtraction gives at most n-digit results, i.e. the most significant digits of ab and $[ab/m]\, m$ agree, therefore only the least significant n digits of both are needed. These yield the algorithm given in Fig. 4.

The half products can be calculated in half as many clock cycles as the full products with school multiplication. In practice a few extra bits precision is needed to guarantee that the last "while" cycle does not run too many times. This increase in operand length makes the algorithm slightly slower than Montgomery's multiplication.

Quisquater's multiplication is a variant of Barrett's algorithm. It multiplies the modulus with a number S, resulting in many leading 1 bits. This makes μ unnecessary, and the corresponding MS-half division becomes trivial. The conversion steps and the calculation with longer modulus could offset the time savings, but in many cases this algorithm is faster. [5]

Interleaved row-multiplication and reduction Long division (modular reduction) steps can be interleaved with the multiplication steps. The advantage is that we don't need to store $2n$-digit full products before the division starts. Furthermore, as the digit-products are calculated, we can keep them in short (32-bit) registers and subtract the digits calculated for the modular reduction, making storing and re-loading unnecessary. During accumulation of the partial products if an intermediate result becomes too large we subtract an appropriate multiple of the modulus.

```
C = 0  // long integer
for i = n-1 … 0
    C = C·d + A·b_i
    q = MS( MS(C)·μ )
    C = C-q·M
    if( C ≥ M )
        C = C-M
```

Fig. 5. Interleaved row multiplication & modular reduction

- Multiply one of the operands, A, by the digits of the other one, from *left to right*: $C_{n-1} = Ab_{n-1}$, $C_{n-2} = Ab_{n-2} \ldots C_0 = Ab_0$. (Here each C_i can be $n+1$-digit long, implicitly padded with i zeros to the right.)
- $C \leftarrow C_{n-1} - q_{n-1}m$, with such a q_{n-1}, that reduces the length of C to n digits. (Multiply the MS digit of C with the pre-computed $\mu = d^2/m_{n-1}$ to get the ratio. A few extra guard bits ensure an error of q_{n-1} to be at most 1.)
- For $i = n-2 \ldots 0$: add C_i to C, shifted to the right by one digit. Again, C can become $n+1$ digits long (excluded the trailing 0's), so $C \leftarrow C - q_i m$, with such a q_i, that reduces the length of C to n digits. If the implicit padding is taken into consideration, we actually subtracted $d^{i-1}qm$.

If the reduction $C \leftarrow C - q_i m$ is not sufficient to reduce the length of C to n digits (q is off by 1), we need a subtraction of m (equivalent to setting $q_i \leftarrow q_i + 1$). With guard bits it is very rare, so the average time for modular multiplications is only twice longer than what it takes to normally multiply 2 numbers. The result is AB mod m, and the digits q_i form the quotient $q = \sum d^i q_i = [AB/m]$. (When not needed, don't store them.) It is a left-right version of the Montgomery multiplication. The products $\mathbf{A \cdot b_i}$ and $\mathbf{q \cdot M}$ are calculated with loops collecting digit-products in an accumulator, storing the LS digit of the accumulator in memory shown in Fig. 6.

```
Q = 0 //32-bit accu
for k = 0 … n-1
    Q = MS(Q) + a_k b_i
    c_k = D0(Q)
c_n = MS(Q)
```

Fig. 6. Row products $\mathbf{C = A \cdot b_i}$

Memory access. If the number of read/write operations is a concern, the MS digits of $\mathbf{C \cdot d + A \cdot b_i}$ could be calculated first, giving a good estimate for q. A single loop calculates $\mathbf{C' + A \cdot b_i - q \cdot M}$, there is no need to store and retrieve the digits of C in between.

```
Q = 0 //33-bit accu
for k = 0 … i-1
    Q = MS(Q) + 2a_k a_i
    c_k = D0(Q)
Q = MS(Q) + a_i^2
c_{i,i+1} = Q
```

Fig. 7. Row-squaring

Squaring. In row i the product $a_k a_i$ is calculated, which has the weight d^{k+i}. The same product is calculated in row k, too. Instead of repeating the work, we accumulate $2a_k a_i$ in row i. In order to keep track of which product is calculated when, in row i we consider only $a_k a_i$ with $k \le i$. Because of the multiplier 2 the product is accumulated 1 bit shifted, but the result is still at most $i+1$ digits. (The worst case if all the digits are $d-1$ gives $d^{i+2} - d^i - 2d + 2 < d^{i+2}$.) We can accumulate half products faster: $a_k a_i$ and $a_k^2/2$, and double the final result. When a_0 was odd we accumulate $(a_0^2 - 1)/2$, with a final correction step [17].

5 New Multiplication Algorithms

Below new variants of popular modular multiplication algorithms are presented, tailored to the proposed HW, which was in turn designed, to allow speeding up these algorithms. Even as microprocessor *software* these algorithms are faster than their traditional counterparts. Their running time comes from two parts, the multiplication and the modular reduction. On the multiply-accumulate HW the products are calculated in n^2 clock cycles for the general case, and $n(n+1)/2$ clock cycles for squaring (loop j in Fig. 10 ignores repeated products, double the result and adds the square term). The modular reduction phase differentiates the algorithms, taking $(1+s)\,n^2 + t \cdot n + constant$ clock cycles.

5.1 Row- or Column-Multiplication

At modular reduction a multiple of m: $q \cdot m$ is subtracted from the partial results. q is computed from the MS digits. At row-multiplication they get calculated last, which forces us to save/load partial results, or process the MS digits out of order. It slows down the algorithms and causes errors in the estimation of q, to be fixed with extra processing time. Calculating the product-digits in columns, the MS digits are available when they are needed, but with the help of longer or bundled accumulators. Both techniques lead to algorithms of comparable complexities. Here we deal with ***column multiplications***, while the row multiplication results are presented in [17].

5.2 Left-Right-Left Multiplication with Interleaved Modular Reduction

The family of these algorithms is called LRL (or military-step) algorithms. They don't need special representation of numbers and need only a handful of bytes extra memory.

Short reciprocal. The LRL algorithms replace division with multiplication with the reciprocal. For approximate 2-digit reciprocals, $\mu \approx d^{n+2}/2m$, only the MS digits are used, making the calculation fast. Since m is normalized, $\frac{1}{2} d^n \le m < d^n$, the exact reciprocal $\frac{1}{2} d^2 < [d^{n+2}/2m] \le d^2$ is 2 digits, except at the minimum of $m = \frac{1}{2} d^n$. Decreasing the overflowed values, to make $\mu \le d^2 - 1$, does not affect the overall accuracy of the approximation. Most algorithms work well with this approximate reciprocal, calculated with a small constant number of digit operations (around 20 with Newton iteration). If the *exact* 2-digit reciprocal is needed, compare $d^{n+2}/2$ and μm. If the approximation is done according to Note R below, adding m if needed makes the error $0 \le err < m$. When this addition was necessary, $\mu+1$ is the exact reciprocal, and it can be calculated in $2n + \mathrm{O}(1)$ clock cycles, in the average ($4n + const$ worst case).

Lemma 1 The 2-digit approximate reciprocal calculated from the 2 MS digits of m $\mu = [d^4/(2dm_{n-1}+2m_{n-2})]$ has an error $-2 < \gamma < 1$ from to the true ratio $d^{n+2}/2m$.

Proof $d^{n+2}/2m = d^4/(2dm_{n-1}+2m_{n-2}+2m')$ with $0 \le m' < 1$, m' representing the omitted digits. $d^4 - \mu \cdot 2(dm_{n-1}+m_{n-2}) = r$, the remainder of the division. $0 \le r < 2(dm_{n-1}+m_{n-2})$. From $d^4/(2dm_{n-1}+2m_{n-2}+2m') = \mu + \gamma$ we get $d^4 = 2(dm_{n-1}+m_{n-2}+m')(\mu + \gamma)$, or

$$\gamma = (r/2 - \mu m')/(dm_{n-1}+m_{n-2}+m'). \tag{1}$$

It is decreased by setting $r = 0$, giving a negative expression, which is decreasing with m'. Putting $m' = 1$, larger than its maximum, gives $-\mu/(dm_{n-1}+m_{n-2}+1) < \gamma$. μ is largest when the denominator is the smallest, resulting in $-2 < -(d^4/d^2)/(d^2/2+1) < \gamma$.

The other side of the inequality is proved by setting r to $2(dm_{n-1}+m_{n-2})$ larger than its maximum. If the expression in (1) is positive, it is decreasing with m', so setting it to its minimum $m' = 0$ gives the maximum: $(dm_{n-1}+m_{n-2}-\mu \cdot 0)/(dm_{n-1}+m_{n-2}+0) = 1 > \gamma$. []

Let's denote the 2-digit reciprocals by $\mu = [d^{n+2}/2m]$, $\mu^{(0)} = [d^4/(2dm_{n-1}+2m_{n-2})]$, and $\mu^{(1)} = [d^4/(2dm_{n-1}+2m_{n-2}+2)]$ (subtracting 1 if needed to make them exactly 2-digit).

From the definition: $\mu^{(1)} \le \mu \le \mu^{(0)}$.

Lemma 2 a) If $(dm_{n-1}+m_{n-2}) \cdot (dm_{n-1}+m_{n-2}+1) \le \frac{1}{2} d^4$: $\mu^{(0)} - \mu^{(1)} = 1$ or 2 and
b) if $(dm_{n-1}+m_{n-2}) \cdot (dm_{n-1}+m_{n-2}+1) \ge \frac{1}{2} d^4$: $\mu^{(0)} - \mu^{(1)} = 0$ or 1.

Proof $d^4/(2dm_{n-1}+2m_{n-2}) - d^4/(2dm_{n-1}+2m_{n-2}+2) = \frac{1}{2} d^4/\big((dm_{n-1}+m_{n-2}) \cdot (dm_{n-1}+m_{n-2}+1)\big)$.

In case a) the difference is less than 2, so the integer parts are at most 2 apart. In case b) the difference is less than 1, making the integer part differ at most by 1. []

Corollary $\mu = [d^4/2m]$, the 2-digit reciprocal of m can be calculated from its 2 MS digits with an error of ≤ 2, with desired sign of the error, knowing $\mu^{(1)} \leq \mu \leq \mu^{(0)}$.

Lemma 3 If $d = 2^{16}$ and $m \geq$ 0xB504 C417… $> d^n/\sqrt{2}$, the error $\mu^{(0)} - \mu^{(1)} = 0$ or 1.

Proof Easily verified by computing the values not covered by Lemma 2, case b). []

Corollary If $d = 2^{16}$ and $m > d^n/\sqrt{2} =$ 0xB504 F333…, the error $\mu^{(0)} - \mu^{(1)} = 0$ or 1. (Similar results hold for other practical digit sizes, like 24, 32, 48 or 64 bits.)

Note R. In Lemma 2 Case a) we can look at the next bit (MS bit of m_{n-3}). If it is 0, $\mu^{(0)}$ is closer to μ than $\mu^{(1)}$, so an approximation with an error at most 1 is provided by $\mu^{(1)}+1 \leq \mu \leq \mu^{(0)}$. If the MS bit of m_{n-3} is 1, $\mu^{(1)}$ is closer to μ, then $\mu^{(1)} \leq \mu \leq \mu^{(0)}-1$ provides an error of at most 1.

Note T. Dropping some LS bits, the truncated $\mu^{(1)}$ is never too large and has an error at most 1 in the LS bit kept. This approximation of the reciprocal can be calculated with only a small constant number of digit operations. Adding 1 in the last bit-position of the truncated reciprocal yields another approximate reciprocal, which is never too small and has an error of at most one in this bit-position.

Algorithm LRL4

- Calculate $a_{n-1}b_{n-1}$, padded with $n-1$ zeros to the right. Subtract a multiple of the modulus m, such that the result becomes n digits long (it was at most $n+1$ digits long): Multiply the MS digit with twice the pre-computed $\mu = [d^{n+2}/2m]$ to get the ratio q_{n-1}. (A few extra guard bits ensure an error of at most 1.)
- Calculate the next MS digit from the product: $a_{n-1}b_{n-2} + a_{n-2}b_{n-1}$, pad, and add to the n-digit partial remainder calculated before, giving at most $n+1$ digits. A multiplication with μ of the MS digit(s) gives q_{n-2}, such that subtracting $q_{n-2}m$ reduces the partial result again to n digits. Multiplication the digits of m (taken from right-to-left) with the single digit q_{n-2} and their subtraction from the partial remainder is done in parallel (pipelined).

```
R_{n-1…n-3} = a_{na-1}b_{nb-1}d + a_{na-1}b_{nb-2} + a_{na-2}b_{nb-1}
for k = na+nb-4…n-3   // products downward
    R_{n…n-4} += Σ_{i+j==k} a_i b_j
    if (overflow) R -= m
    q = (R_{n-1}μ_1 d^2 + R_{n-1}μ_0 d + R_{n-2}μ_1 d + R_{n-2}μ_0)/d^3 · 2
    R = (R−q·m)d
for k = 0 … n-4          // LS product-digits
    R_{n…k} += Σ_{i+j==k} a_i b_j
while( R_n > 0 ) R -= m // overflow
```

Fig. 8. The LRL4 modular multiplication algorithm

- Continue this way until the new digit-products summed in columns do not increase the length of the partial product (the LS $n-2$ digits of the product). The rest is calculated from *right-to-left* and added normally to the product, with a possible final reduction step.

The reduction step is better delayed until the MS 3 digits of the product are calculated. This way adding the next product-digit can cause a carry at the MS bit at most 1: the possible maximum is $n(d-1)^2 << d^3$. The overflowed digit need not be stored, the situation can be fixed immediately by subtracting m. This overflow happens if the new digit is large, and the first digit of the partial result is larger than $d-n$. This digit is the result of the modular reduction, so it is very close to uniformly distributed random. The probability of an overflow is less than n/d, about 1 in 1,000 at 16-bit digits, practically 0 for 32-bit digits.

```
c = 0 // 1 digit temp
store
Q = 0 // 33-bit
accumulator
for k = 0 … n-1
    Q = MS(Q) + c - q·m_k
    c = r_k
    r_k = D0(Q)
```

Fig. 9. Inner loop of modular reduction `R =(R-q·m)d`

This way we need not store $2n$-digit-products, the partial remainders only grow to n digits (plus an occasional over/under-flow bit). Although the used column-sums of digit-products add only to the 3 MS digits of the partial product (no long carry propagation), the subtraction of $q_i m$ still needs n-digit arithmetic (the LS digits further to the right are implicit 0's). All together a handful (t) steps are needed to calculate an approximate quotient q_i, which makes $n^2 + tn$ clock cycles for the modular reduction. $\{\mu_1, \mu_0\}$ is a 2-digit pre-computed reciprocal of m. The implementation details are shown in Fig. 9 and Fig. 10. If the quotient $[ab/m]$ is needed, too, we store the sequence of q_i's. There is a minor difficulty with the occasional $q \leftarrow q+1$ correction steps, which might cause repeated carry propagation. Since it happens rarely, it does not slow down the process noticeably, but we could store the locations of the correction steps (a few bytes storage) and do a final right-to-left scan of the quotient, to add one to the digits where the correction occurred.

```
Q = 0              // 50bit accumulator
for k = 0 … n-4
    Q = MS(Q) + r_k
    for j = max(0,k+1-
n_a)…min(k+1,n_b)
        Q += a_{k-j}b_j
    r_k = D0(Q)
for i = n-3 … n  // storing digits
    Q = MS(Q) + r_i
    r_i = D0(d)
```

Fig. 10. Calculation the digits of $a \cdot b$ `for k = 0 … n-4: R_{n…k} += Σ_{i+j==k} a_i b_j`

Time complexity $\mu = [d^{n+2}/2m]$ the 2-digit reciprocal is at most 1 smaller than the true ratio $d^{n+2}/2m$. The quotient q is calculated from the 2 MS digits of the dividend x and the reciprocal μ:

$$q = [2(\mu_1 x_n d^2 + (\mu_1 x_{n-1} + \mu_0 x_n)d + \mu_0 x_{n-1}) / d^3]. \qquad \text{(LRL4}q\text{)}$$

Proposition: q is equal to or at most 1 less than the integer part of the true quotient.

Proof: At the reduction steps the true ratio was $q' = [(x_n d^n + x_{n-1} d^{n-1} + \xi d^{n-1}) / m]$ with some $0 \le \xi < 1$, representing the rest of the digits of x. The approximate reciprocal μ has an error γ, at most 1 from the rational $d^{n+2}/2m$, so our estimated ratio

$$q = [2(x_n d + x_{n-1})\cdot(d^{n+2}/2m - \gamma)\,/\,d^3]$$
$$= [(x_n d^n + x_{n-1} d^{n-1} + \xi d^{n-1})\,/\,m - \xi d^{n-1}/m - 2(x_n d + x_{n-1})\,\gamma\,/d^3]$$

The subtracted error term, $\xi d^{n-1}/m + 2(x_n d + x_{n-1})\,\gamma/d^3$ is increased if we put $\gamma = 1$, $\xi = 1$, x_n, $x_{n-1} = d - 1$, and m to its smallest possible value, $\frac{1}{2}\, d^n$. The error is smaller than the resulting $2d^{n-1}/d^n + 2(d^2 - 1)/d^3 = (4d^2 - 2)/d^3$, which is less than 1 if $d \geq 4$. []

If the reciprocal was calculated from the 2 MS digits of m (error $-2 < \gamma < 1$), the partial remainder could become negative, having an error of at most roughly $\pm 4/d$. Most of the time the added new digit-products fix the partial results, but sometimes an *add-back* correction step is necessary. The running time of the modular reduction in SW is less than $\boxed{1.0001n^2 + 4n}$, in the average; $\boxed{n^2 + 4n}$ with extra HW.

LRL3 From the calculation of q we can drop $\mu_0 x_{n-1}$. It causes an additional error of at most $2(d-1)^2/d^3 < 2/d$. The approximate quotient digit is calculated with

$$q = [2(\mu_1 x_n d + \mu_1 x_{n-1} + \mu_0 x_n)\,/\,d^2]. \qquad \text{(LRL3}q\text{)}$$

It takes 3 clock cycles with our multiply-accumulate HW, with a 1 bit left-shift and extracting the MS digit. In pure software implementation the occasional addition-subtraction steps add $0.0001n^2$ to the running time, in the average. Alternatively, the overflow can be handled at the next reduction step, where the new quotient will be larger, $q = d + q'$, with $q' < d$. In the proposed HW the extra shifted additions of the modulus digits can be done in parallel to the multiplications, so the worst case running time is $\boxed{n^2 + 3n}$ for the modular reduction, in SW we need $\boxed{1.0001n^2 + 3n}$ steps.

Note If the reduction step fails to reduce the length of the remainder, the second MS digit is very small (0…±6). In this case, adding the subsequent digit of the product (the next column-sum) cannot cause an overflow, so the reduction can be safely delayed after the next digit is calculated and added. It simplifies the algorithm but does not change its running time.

LRL2 Shifting μ a few bits to the right, does not help. E.g. 8 bits give $\mu_1 \approx \sqrt{d}$ which still does not make any of the terms in calculation of q small, like $\mu_1 x_{n-1} < 2d^{3/2}$. Dropping it anyway makes the resulting

$$q = [(\mu_1 x_n d + \mu_0 x_n)\,/\,d^{3/2}] \qquad \text{(LRL2}q\text{)}$$

not very good, giving often an error of 1, sometimes 2. The average running time of the SW version is large, $\boxed{2n^2 + 2n}$; but with extensive use of HW it is $\boxed{n^2 + 2n}$.

LRL1 Instead of dropping terms from (LRL3q), we can use shorter reciprocals and employ shift operations to calculate some of the products. Without first shifting the modulus to make the reciprocal normalized we get $\mu = [d^3/(dm_{n-1} + m_{n-2})]$, with $\mu_1 = 1$, which reduces some multiplications to additions. Unfortunately, at least 1 more bit is needed to the right of μ_0. Its effects can be accommodated by a shift-addition with the content of the accumulator. This leads to LRL1, the fastest of the LRL algorithms.

Keep the last results in the 50-bit accumulator $Q = c d^3 + x_n d^2 + x_{n-1} d + x_{n-2}$ (3 digits + carry). Use 1 digit + 2 bits reciprocals in the form $\mu = \frac{1}{2}[2d^{n+1}/m] = d + \mu_0 + \delta$, with $\delta = 0, \frac{1}{2}$. The multiplication with d needs no action when manipulating the con-

tent of the accumulator. Multiplication with $\delta > 0$ is also done inside the accumulator parallel to the multiplication with μ_0: add the content of the accumulator to itself, shifted right by 17 bits.

What remains is to calculate $\mu_0(cd^2 + x_n d + x_{n-1})$. If c is 0 or 1 then $\mu_0 c d^2$ is computed by a shift/add, parallel to the $\mu_0 x_n$ multiplication. We drop the term $\mu_0 x_{n-1}/d^2$ from the approximation of q. The LRL1 algorithm still works, using only one digit-multiplication ($\mu_0 x_n$) for the quotient estimate with two shift-adds, $\mu_0 c d$ and the accumulator times δ/d. All aditions are done in the LS 32 bits of the accumulator.

$$q = \left[\left(\mathrm{Q} + \mathrm{Q}\,\delta/d \right)/d^2 + \mu_0 c + \boldsymbol{\mu_0 x_n}/d \right] \qquad \text{(LRL1}q\text{)}$$

Similar reasoning as at LRL4 assures that the conditions $c = 0$ or 1, and $0 \le q < 3d$ is maintained during the calculations. The probability of the corrections is $\approx$¼, giving an average SW running time $\boxed{1.25n^2 + n}$; in the accumulator-shift HW $\boxed{n^2 + n}$.

Computer experiments. A C program simulating the algorithm was tested against the results of GMP (GNU Multi Precision library [6]). 55 million modular multiplications of random 8192×8192 mod 8192 bits were tried, in addition to several billion shorter ones. The digit $n+1$ of the partial results in the modular reduction was always 0 or 1 and $0 \le q < 3d$ remained true.

Note. The last overflow correction (after the right-to-left multiplication phase) can be omitted, if we are in an exponentiation chain. At overflow the MS digit is 1, the second digit is small, and so at the beginning of the next modular multiplication there is no more overflow (the MS digit stays as 1), and it can be handled the normal way.

5.3 Modulus Scaling

The last algorithm uses only one digit-multiplication to get the approximate quotient for the modular reduction steps. We can get away with even fewer (0), but with a cost in preprocessing. It increases the length of the modulus. To avoid increasing the length of the result, the final modular reduction step in the modular multiplications is done with the original modulus. Preprocessing the modulus is simple and fast, so the algorithms below are competitive to the standard division based algorithms even for a single call. If a chain of calculations is performed with the same modulus (exponentiation) the preprocessing becomes negligible compared to the overall computation time.

The **main idea** is that the calculation of q becomes easier if the MS digit is 1 and the next digit is 0. Almost the same good is if all the bits of the MS digit are 1. In these cases ***no*** multiplication is needed for finding the approximate q [5], [15].

We can convert the modulus to this form by multiplying it with an appropriate 1-digit scaling factor. This causes one-time extra costs at converting the modulus in the beginning of the calculation chain, but the last modular reduction is done with the original modulus, and so the results are correct at the end of the algorithms (no post-processing). The modified modulus is one digit longer, which could require extra multiply-accumulate steps at each reduction sweep, unless SW or HW changes take advantage of the special MS digits. These modified longer reduction sweeps are performed 1 fewer times than the original modulus would require. The final reduction with the original modulus makes the number of reduction sweeps the same as before.

There are two choices for the scaled modulus, $\{m_{n+1}, m_n\} = \{1, 0\}$ and $m_n = d-1$. The corresponding modular reduction algorithms are denoted by S10 and S0F. In both cases $q = r_{n+1}$ gives the estimate. Both algorithms need to store and process only the LS n digits of the scaled modulus, the processing of the MS digits can be hardwired.

Algorithm S0F The initial conversion step changed the modulus, such that $|m| = n+1$ and $m_n = d-1$. The current remainder R is shifted up (logically, that is only the indexing is changed) to make it n+2 digits long, such that $R = \{r_{n+1}, r_n\ldots, r_1, 0\}$.

Proposition: $q = r_{n+1}$ is never too large, sometimes it is 1 too small.
Proof: (similar to the proof under S10 below) []

Correction steps q is *often* off by 1. It results in an overflow, i.e., the bit is not cleared from above the MS digit of the partial results. We have to correct them immediately (by subtracting m), otherwise the next q will be 17 bits long, and the error of the ratio estimation is doubled, so there could be a still larger overflow next.

These frequent correction steps cannot be avoided with one digit scaling. It corresponds to a division of single digits and they don't provide good quotient estimates. The S10 algorithm below uses an extra bit, so it is better in this sense.

Algorithm S10: The initial conversion step changed the modulus, such that $|m| = n+2$ and $m_{n+1} = 1$, $m_n = 0$. The current remainder R is shifted to the left (logically, that is only the indexing is changed) to make it also n+2 digits long, such that $R = \pm\{r_{n+1}, r_n,\ldots, r_1, 0\}$. We use signed remainders and have to handle overflows, that is, a sign and an overflow bit could temporarily precede the digits of R. The algorithm will add/subtract m to fix the overflow if it occurs.

Now there are $|d|_2$+1 bits of the modulus fixed, and the signed R is providing also $|d|_2$+1 bits of information for q. With them correction steps will rarely be needed.
Proposition: $q = r_{n+1}$ for $R \geq 0$, and $q = d-1-r_{n+1}$ for $R < 0$ assignment gives an estimate of the quotient, which is never too small, sometimes 1 too large.
Proof: When $R \geq 0$ the extreme cases are:

1. Minimum/maximum: $R=\{r_{n+1},0,\ldots 0\}$, and $m=\{1, 0, d-1,\ldots d-1\}=d^{n+1}+d^n-1$. The true quotient is $[r_{n+1}d^{n+1}/(d^{n+1}+d^n-1)] = [r_{n+1} - r_{n+1}(d^n-1)/(d^{n+1}+d^n-1)] \geq [r_{n+1} - (d-1)(d^n-1)/(d^{n+1}+d^n-1)] = r_{n+1}-1$, because r_{n+1} is integer and the subtracted term inside the integer part function is positive and less than 1: $(d^{n+1} - d^n - d + 1)/(d^{n+1}+d^n-1) < 1$.
2. Maximum/minimum: $R = \{r_{n+1}, d-1, d-1,\ldots d-1\} = r_{n+1}d^{n+1}+d^{n+1}-1$, and $m = \{1, 0, 0,\ldots 0\} = d^{n+1}$ give $[(r_{n+1}d^{n+1}+d^{n+1}-1)/d^{n+1}] = [r_{n+1} + (d^{n+1}-1)/d^{n+1}] = r_{n+1}$. These show that the estimated q is never too small.

When R is negative, the modular reduction step is done with adding $q{\cdot}m$ to R. The above calculations can be repeated with the absolute value of R. However, the estimated (negative) quotient would be sometimes 1 too small, so we increase its value by one. This is also necessary to keep q a single-digit number. []

Correction steps. Overflow requiring extra correction steps, almost never occurs. Computer experiments showed that, during over 10 million random 8192×8192 mod 8192-bit modular multiplications there were only 5 subtractive corrections step performed, no underflow was detected. During over 10^{10} random 1024×1024 mod 1024-bit modular multiplications there were 2 overflows and no underflow. Accordingly, the use of only software corrections does not slow down S10.

```
R_{n-1…n-3} = a_{n_a-1}b_{n_b-1}d + a_{n_a-1}b_{n_b-2} + a_{n_a-2}b_{n_b-1}
for k = n_a+n_b-4 … n-3          // products down
    R_{n…n-4} += Σ_{i+j==k} a_i b_j
    if ( R_n > 0 ) R -= m        // overflow
    if ( R_n <-1 ) R += m        // underflow
    if ( R_n == 0 )              // positive rem
        q = R_{n-1}
        R =(R - q·m)d
    else                         // nagative rem
        q = d-1-R_{n-1}
        R =(R + q·m)d
for k = 0…n-4                    // LS digits
    R_{n…k} += Σ_{i+j==k} a_i b_j
while( R_n > 0 ) R -= m          // overflow
while( R_n <-1 ) R += m          // underflow
```

Fig. 11. Basic structure of the S10 modular multiplication algorithm

The remainder is rarely longer than $n+1$, because at the start, there is no overflow: the MS digits of the products, even together with all the others don't give longer than $|A|+|B|$ -digit results. After a modular reduction

- if the estimated quotient q was correct, the remainder R is less than mS, that is, either $|R| = n+1$, or R = {1, 0, r_{n-1}, …}, where the third digit is smaller than that of mS. This last case is very unlikely (less than $1/(4d)$).
- if the estimated quotient q was one too large, the remainder R changes sign. If the correct quotient $q-1$ would reduce R to a number of smaller magnitude than {0, 0, r_{n-1}, …}, with the third digit at most as large as that of mS, then the actual q could cause an overflow. Again, it is a very unlikely combination of events.

Implementation. The MS 3 digits and the sign/overflow bits of the partial results can be kept in the accumulator. Because the multiplier is not used with the special MS digits of the scaled modulus, they can be manipulated together in the accumulator. Adding the next product-digit is done also in the accumulator in a single cycle.

Proposition: There is no need for longer than 50-bit accumulator.

Proof. The added product-digit could cause a carry less than n to the $n+1^{st}$ digit. If there was already an overflow, this digit is 0, so there will be no further carry propagation. If the $n+1^{st}$ digit was large, we know, the $n+2^{nd}$ digit of R was zero. These show that the MS digit of R can be −2, −1, 0, 1 — exactly what can be represented by the carry and sign, as a 2-bit 2's complement number. []

Computing the scaling factor. S Let n, n_a, n_b denote the length of m, A and B respectively. Let us calculate $S = [(d-1)d/m_{n-1}]$ for S0F, and $S = [d^2/m_{n-1}]$ for S10. With normalized m: $\frac{1}{2}d \le m_{n-1} < d$, so we get S one bit and one digit long. If the short division estimated the long one inaccurately, we may need to add or subtract m, but at most a couple of times suffice, providing an O(n) process. Adding/subtracting m changes the MS digit by at most 1, so it gets the desired value at the end.

5.4 Montgomery Multiplication with Multiply-Accumulate Structure

We can generate the product-digits of AB from right to left and keep updating R, an $n+1$-digit partial remainder, initialized to 0. The final result will be also in R. When the next digit of AB has been generated with its carry (all necessary digit-products accumulated), it is added to R, followed by adding an appropriate multiple of the modulus $t \cdot m$, such that the ***last digit*** is cleared: $t = r_0 \cdot m'$ with $m' = -m_0^{-1} \bmod d$. It can be done in a single sweep over R. The result gets stored in R shifted by 1 digit to the right (suppressing the trailing 0). The new product-digit, when calculated in the next iteration, is added again to the LS position of R.

```
R = 0^(n+1)
for i = 0 … 2n-1
    Q = r_0
    for_(j,k) 0 ≤ (j,k) < n, j+k == i
        Q += a_j b_k
    t = Q·m' mod d
    Q+= t·m_0
    for j = 1 … n-1
        Q = MS(Q) + r_j + t·m_j
        r_(j-1) = D0(Q)
    r_(n,n-1) = MS(Q)
if( R ≥ m )
    R = R - m
```

Fig. 12. Montgomery multiplication with multiply-accumulate structure

Proposition after each reduction sweep (loop i) the remainder is $R \le m + n(d-1)$.

This fact ensures, that the intermediate results do not grow larger than $n+1$ digits, and a single final correction step is enough to reduce the result R into $[0, m)$. If $n \ge 3$ and any of the MS digits $m_k < d-1$, $k = 2, 3, 4\ldots$, then R remains always n-digit long. (The case where each but the 2 LS digits of m is $d-1$ is trivial to handle.)

Proof by induction. At start it is true. At a later iteration c_i is the new product-digit, $R \leftarrow [(R + c_i + t \cdot m)/d] \le \left[(m + n(d-1) + (d-1)^2 \cdot n + (d-1) \cdot m)/d\right] = m + n(d-1)$. []

We don't need correction steps or processing the accumulator, so some HW costs can be saved. The final reduction step can be omitted during a calculation chain [5], since the next modular reduction produces a result $R \le m + n(d-1)$ from n-digit inputs, anyway. Only the final result has to be fixed. Also, double-speed squaring can be easily done in the loop (j,k).

Montgomery-T (Tail Tailoring) The Montgomery reduction needs one multiplication to compute the reduction factor t. Using Quisquater's scaling, this time to transform the LS, instead of the MS digit to a special value, we can get rid of this multiplication. The last modular reduction step is performed with the original modulus m to reduce the result below m.

If $m' = 1$, the calculation of $t = \mathrm{LS}(r_0 \cdot m')$ becomes trivial: $t = r_0$. It is the case if $m_0 = d - 1$, what we want to achieve with scaling.

The first step in each modular reduction sweep, $Q = r_0 + t\,m_0$, has to be performed even though we now the LS digit of the result (0), because the carry, the MS digit is needed. If $m_0 = d - 1$, we know the result without calculation: $Q = r_0 + t\,m_0 = r_0 + r_0\,(d-1) = r_0\,d$. Accordingly, we can start the loop 1 digit to the left, saving one multiplication there, too. Therefore, the modular reduction step is not longer than what it was with the sorter modulus.

To make $m_0 = d - 1$ we multiply (scale) m with an appropriate one-digit factor S, which happens to be the familiar constant $m' = -m_0^{-1} \bmod d$, because $m_0 \mathrm{S} = m_0\,m' \equiv -m_0\,m_0^{-1} \bmod d = d - 1$. Multiplying m with S increases its length by 1 digit, but as we have just seen, the extra trailing digit does not increase the number of digit-multiplications during modular reduction.

The modular reduction performs $n - 1$ iterations with the modulus $m\mathrm{S}$: $n\,(n-1)$ digit-multiplications, and one iteration with the modulus m: $n + 1$ digit-multiplications. We saved $n - 1$ digit-multiplications during the modular reduction steps. The price is the need to calculate S, $m\mathrm{S}$ and store both m and $m\mathrm{S}$.

Proof of correctness: One Montgomery reduction step calculates $\mathrm{A}_1 = (\mathrm{A} + k \cdot m\mathrm{S}) / d$, with such a k, that ensures no remainder of the division. Taking this equation modulo m, we get $\mathrm{A}_1 \equiv \mathrm{A} \cdot d^{-1} \bmod m\mathrm{S}$. After $n - 1$ iterations the result is $\mathrm{A}_{n-1} \equiv \mathrm{A} \cdot d^{-(n-1)} \bmod m\mathrm{S}$. The final step is a reduction mod m: $\mathrm{A}_n \equiv \mathrm{A}_{n-1} \cdot d^{-1} \bmod m$. In general $(a \bmod b \cdot c) \bmod c = a \bmod c$, (for integers a, b and c), so we get $\mathrm{A}_n = \mathrm{A} \cdot d^{-n} \bmod m$, or m larger than that (the result is of the same length as m). This is the result of the original Montgomery reduction. []

6 Summary

There is no single "best" modular multiplication algorithm, for all circumstances.

- In *software* for general purpose microprocessors
  - For very long operands sub-quadratic multiplications are the best, like Karatsuba, Toom-Cook or FFFT-based methods [6]
  - For cryptographic applications
    - If memory is limited (or auxiliary data does not fit to a memory cache): LRL3 or LRL1 (dependent on the instruction timing)
    - If memory is of no concern: S10 or Montgomery-T (needs pre/post-process)
- If some *HW* enhancements can be designed:
  - If there is enough memory: S10 (simpler)
  - Otherwise: LRL1.

The table below summarizes the running time of the *modular reduction phase* of our modular multiplication algorithms (without the n^2 steps of the multiplication phase).

Algorithm	Storage beyond operands	Pre-processing	Post-processing	#Digit-products + fixes in SW	Extra HW	#Digit-products with extra HW
Barrett	$2n$	$O(n^2)$	–	n^2+5n	–	n^2+5n
LRL4	–	–	–	$1.0001n^2+4n$	Shifter	n^2+4n
LRL3	–	–	–	$1.0001n^2+3n$	Shifter	n^2+3n
LRL2	–	–	–	$2n^2+2n$	Shifter	n^2+2n
LRL1	–	–	–	$1.25n^2$	Shifter Accu-adder	n^2+n
S0F	n	n	–	$1.25n^2$	Shifter, Accu-adder	n^2
S10	n	n	–	$(1+\varepsilon)\,n^2$ (signed)	Shifter Accu-adder	n^2
S10-2	n	n	–	$(1+\varepsilon)n^2$ (signed)	Accu-adder	n^2 + εn^2 adds
Montgomery	–	$O(n^2)$	$O(n^2)$	n^2+n	–	n^2+n
Montgomery-T	n	$O(n^2)$	$O(n^2)$	n^2	–	n^2

References

1. ALTERA Literature: Stratix II Devices http://www.altera.com/literature/lit-stx2.jsp
2. P.D.Barrett, *Implementing the Rivest Shamir Adleman public key encryption algorithm on standard digital signal processor*, In Advances in Cryptology-Crypto'86, Springer, 1987, pp.311-323.
3. Bosselaers, R. Govaerts and J. Vandewalle, *Comparison of three modular reduction functions*, In Advances in Cryptology-Crypto'93, LNCS 773, Springer-Verlag, 1994, pp.175-186.
4. E. F. Brickell. *A Survey of Hardware Implementations of RSA*. Proceedings of Crypto'89, Lecture Notes in Computer Science, Springer-Verlag, 1990.
5. J.-F. Dhem, J.-J. Quisquater, *Recent results on modular multiplications for smart cards*, Proceedings of CARDIS 1998, Volume 1820 of Lecture Notes in Computer Security, pp 350-366, Springer-Verlag, 2000
6. GNU Multiple Precision Arithmetic Library http://www.swox.com/gmp/gmp-man-4.1.2.pdf
7. K. Hensel, *Theorie der algebraische Zahlen*. Leipzig, 1908
8. J. Jedwab and C. J. Mitchell. *Minimum weight modified signed-digit representations and fast exponentiation*. Electronics Letters, 25(17):1171-1172, 17. August 1989.
9. D. E. Knuth. *The Art of Computer Programming*. Volume 2. Seminumerical Algorithms. Addison-Wesley, 1981. Algorithm 4.3.3R
10. W. Krandick, J. R. Johnson, *Efficient Multiprecision Floating Point Multiplication with Exact Rounding*, Tech. Rep. 93-76, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.
11. A.Menezes, P.van Oorschot, S.Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
12. P.L. Montgomery, *Modular Multiplication without Trial Division*, Mathematics of Computation, Vol. 44, No. 170, 1985, pp. 519-521.
13. J.-J. Quisquater, *Presentation at the rump session of Eurocrypt'90*.
14. R. L. Rivest; A. Shamir, and L. Adleman. 1978. *A method for obtaining digital signatures and public key cryptosystems*. Communications of the ACM 21(2):120--126

15. SNIA OSD Technical Work Group http://www.snia.org/tech_activities/workgroups/osd/
16. C. D. Walter, *Faster modular multiplication by operand scaling*, Advances in Cryptology, Proc. Crypto'91, LNCS 576, J. Feigenbaum, Ed., Springer-Verlag, 1992, pp. 313—323
17. L. Hars, *manuscript*, 2003.