Fast Implementations of RSA Cryptography

M. Shand
J. Vuillemin
Digital Equipment Corp.,
Paris Research Laboratory (PRL),
85 Av. Victor Hugo.
92500 Rueil-Malmaison, France.

Abstract

We detail and analyse the critical techniques which may be combined in the design of fast hardware for RSA cryptography: chinese remainders, star chains, Hensel's odd division (a.k.a. Montgomery modular reduction), carry-save representation, quotient pipelining and asynchronous carry completion adders.

A PAM[1] implementation of RSA which combines all of the techniques presented here is fully operational at PRL: it delivers an RSA secret decryption rate over 600Kb/s for 512b keys, and 165Kb/s for 1Kb keys. This is an order of magnitude faster *than any previously reported* running *implementation.*

While our implementation makes full use of the PAM's reconfigurability, we can nevertheless derive from our (multiple PAM designs) implementation a (single) gate-array specification whose size is estimated under 100K gates, and speed over 1Mb/s for RSA 512b keys.

Each speed-up in the hardware *performance of RSA involves a matching gain in* software *performance which we also analyse. In addition to the techniques enumerated above, our best software implementation of RSA involves Karatsuba multiplication and specific squaring. Measured on 512b keys, the method leads to software implementations at 2Kb/s on a VAX 8700, 16Kb/s on a 40MHz DECstation 5000/240, and 56Kb/s on a 150MHz Alpha, which is faster than any RSA hardware commercially available in 1992.*

1 Introduction

In 1977 Rivest, Shamir and Adleman [RSA 78] introduced an important *public key crypto-system* based on computing modular exponentials. The security of RSA cryptography ultimately lies on our inability to effectively factor large integers. As a point in case, [LLMP 92] used hundreds of computers world-wide for a number months in order to explicitly factor the 9-th Fermat number $F_9 = 2^{2^9} + 1$, a 513 bit integer. Admittedly this factorization uses special properties of F_9 and fully general techniques for factoring numbers over 512b are even slower. Nevertheless, RSA implementations with key lengths of $512b$ must be prepared to renew their keys regularly and cannot be used for reliably transmitting any data which must remain secret for more than a few weeks. Longer keys ($768b$ or $1Kb$) appear safe within the current state-of-the-art on integer factoring.

The complexity of RSA encoding a km bit message with a k bit key is m times that of computing a k bit modular exponential, which is sk^3 in software[2] and hk^2 in hardware[3]. This paper deals with techniques to lower the values of the time constants s and h for both software and hardware implementations of RSA.

During the last five years, we have used RSA cryptography as a *benchmark* for evaluating the computing power of the PAMs built at PRL (see [BRV 89] and [BRV 92]). This reconfigurable hardware has allowed us to implement measure and compare over ten successive versions of RSA (see [SBV 91] for details).

Our fastest current hardware implementation of RSA relies on reconfigurability in many ways: we use a different PAM design for RSA encryption and decryption; we generate a different hardware modular multiplier for each (different prime) modulus P (the k coefficients in the binary representation of P are hardwired into the logic equations).

So, deriving an ASIC from our PAM design is not an automatic task, and we account for these facts[4] in

[1] for *Programmable Active Memory*: based on Programmable Gate Array (PGA) technology, a PAM is a universal configurable hardware co-processor closely coupled to a standard host computer. The PAM can speed-up many critical software applications running on the host, by executing part of the computations through a specific hardware PAM configuration.

[2] although algorithms with better asympotic bounds exist, at the bit lengths of interest in RSA, modular multiplication has complexity k^2.

[3] assuming that the area of the hardware is proportional to k.

[4] as well as earlier experimental figures obtained by H. Touati and R. Rudell for general PAM to ASIC compiling

1063-6889/93 $03.00 © 1993 IEEE

the performance prediction regarding that technology given in the abstract.

2 RSA Cryptography

Let us recall the ingredients for RSA cryptography:

- The *public* modulus $M = P \times Q$ is a $k = l_2(M)$ bit integer (here $l_2(M) = \lceil log_2(M+1) \rceil$), obtained by multiplying two suitably generated *secret* prime numbers P and Q.
- The *public* exponent E is a fixed odd number; in order to speed-up *public* encryption, it is chosen to be small: either $E = F_0 = 3$ as in [K 81], or $E = F_4 = 2^{16}+1$ as recommended by the CCITT standard.
- The *secret* exponent D is determined by the relation:

$$E \times D = 1 \pmod{(P-1) \times (Q-1)}.$$

- A $n = km$ bits message is divided into m blocks and each block represents a k bit integer A:

  1. The *public* encryption $\mathcal{P}(A)$ of block A is:

  $$\mathcal{P}(A) = A^E \pmod{M}.$$

  2. The *secret* decryption $\mathcal{S}(A)$ is:

  $$\mathcal{S}(A) = A^D \pmod{M}.$$

  3. Encrypt and decrypt are respective inverses:

  $$\mathcal{S}(\mathcal{P}(A)) = \mathcal{P}(\mathcal{S}(A)) = A,$$

  modulo M and for all $0 \leq A < M$.

Since the public exponent (say $E = 2^{16}+1$) is small, the complexity of the *public* encryption procedure is only $17sk^2$ in software and $17hk$ in hardware. On current fast micro-processors[5], this provides an effective RSA public encoding bandwidth of 150Kb/s for $k = 512$ and 90Kb/s for $k = 1024$.

Since the secret exponent D has typically $k = l_2(M)$ bits, RSA *secret* decryption is *slower* than public encryption: 32 times slower for $k = 512b$ and $E = F_4$, and 512 times slower for $k = 1Kb$ and $E = 3$.

[5] DECstation 5000/240 containing a MIPS R3000A @ 40MHz

3 Chinese Remainders

In order to speed-up the RSA secret decryption $A^D \pmod{M}$ we take advantage of the secret knowledge of the prime decomposition $M = P \times Q$ [QC 82], and use chinese remainders $\pmod{P}$ and $\pmod{Q}$: [6]

Algorithm 1 (RSA decrypt) *In order to compute* $\mathcal{S}(A) = A^E \pmod{M = P \times Q}$*:*

1. *Compute*
$$A_p = A \pmod{P},$$
$$A_q = A \pmod{Q}.$$

2. *Compute*
$$B_p = A_p^{D_p} \pmod{P},$$
$$B_q = A_q^{D_q} \pmod{Q},$$
with (precomputed) exponents:
$$D_p = D \pmod{P-1},$$
$$D_q = D \pmod{Q-1}.$$

3. *Compute*
$$\mathcal{S}_p = B_p \times C_p \pmod{M},$$
$$\mathcal{S}_q = B_q \times C_q \pmod{M},$$
with (precomputed chinese) coefficients:
$$C_p = Q^{P-1} \pmod{M},$$
$$C_q = P^{Q-1} \pmod{M}.$$

4. *Compute*
$\mathcal{S} = \mathcal{S}_p + \mathcal{S}_q$,
$\mathcal{S}(A) =$ *if* $\mathcal{S} \geq M$ *then* $\mathcal{S} - M$ *else* $\mathcal{S}$.

With chinese remainders, the software complexity of RSA decryption goes from sk^3 down to $\frac{s}{4}k^3 + 4sk^2$; a speed-up factor of $4 - \epsilon(k)$, with $\epsilon(k) = \frac{4}{k} < 1/100$ for $k > 400$. The factor $\epsilon(k)$ accounts for the initial modulo reductions and final chinese recombination.

With a single k bit multiplier, the hardware speed-up from chinese remainders is only $2 - \epsilon'(k)$, with $\epsilon'(k) = 4/k$ accounting for the initial and final overhead.

A better way to use the *same* silicon area is to operate *two* $\frac{k}{2}$ bit multipliers in parallel, one for each factor of M as in fig. 1. The exponentiation time becomes $\frac{h}{4}k^3$ for a speed-up near 4, provided that we can perform the final chinese recombination at the same rate. Taking full advantage of the reconfigurability of the PAM, we have implemented three solutions to this problem:

1. Compute the chinese recombination *in software*: a 40 MIPS host performs this operation at a rate of more than 600Kb/s on a pair of 512b primes, which easily accommodates our hardware exponentiation rate for 1Kb RSA keys.
2. Assist a slower host computer with a fast enough hardware multiplier, running in parallel with the two exponentiators [SBV 91].

[6] The knowledge of P and Q is equivalent to that of D, as we can efficiently factor $M = P \times Q$ once we know M, E and D. This justifies our use of chinese remainders.

3. Design the two $\frac{k}{2}$ bit modulo P and Q multipliers so that they can be reconfigurated quickly enough into one single k bit multiplier modulo M, which performs the final chinese combination.

4 Modular Exponential

4.1 Binary Methods

Considering the binary representation of the exponent $E = [e_{k-1} \cdots e_0]_2$ with $e_{k-1} = 1$, there are *two* ways to reduce the computation of $B = A^E \pmod M$ to a sequence of squares and modular products. Each is determined by the order in which it processes exponent E, from low bits to high bits in algorithm L, and conversely in algorithm H[7].

Algorithm 2 (L Modular Exponential) *Compute* $B = A^E \pmod M$ *by:*

```
B[0] = 1; P[0] = A;
for i<k-1 do
 { P[i+1] = P[i] * P[i] (mod M);
   B[i+1] =  if e{i}=1
               then P[i] * B[i] (mod M)
               else B[i] };
B =  P[k-1] * B[k-1] (mod M).
```

Algorithm 3 (H Modular Exponential) *Compute* $P = A^E \pmod M$ *by:*

```
B[0] = A;
for i<k-1 do
 {   P[i] = B[i] * B[i] (mod M);
   B[i+1] = if e{k-i-2}=1
              then P[i] * A (mod M)
              else P[i] };
B = B[k-1].
```

Both algorithms involve $k-1 = l_2(E)-1$ squares and $\nu_2(E)$ modular products, where $\nu_2(E) = \sum_{i<k} e_i$ is the number of one bits in the binary representation of the exponent E; clearly, $0 < \nu_2(E) \leq k$ and the average value of $\nu_2(E)$ is $\frac{k}{2}$.

In *software*, the choice of either L or H makes *no difference* with regard to the computation time whose average is $\frac{3s}{2}k^3$; here s is the time required per bit of modular product.

In *hardware*, algorithm L requires *two* storage registers (for B and P) while algorithm H gets away with only *one* storage register (for both B and P).

It is natural (see [OK 91]) to implement algorithm L with *two* logically distinct k bit modular multipliers; one for squaring B, the other for multiplying B and P as in fig. 2[8]. The number of cycles required for computing a k bit modular exponential in this way is sk^2, where s is the cycles required per bit of modular product. In [OK 91] these two multipliers are time-multiplexed on the same physical multiplier, even though the multiplication of B and P happens on average in only half of the cycles allocated to it. This multiplexed implementation is chosen because data dependencies in the inner loop of the modular multiplication algorithm make it impossible to commence the next step on the next cycle.

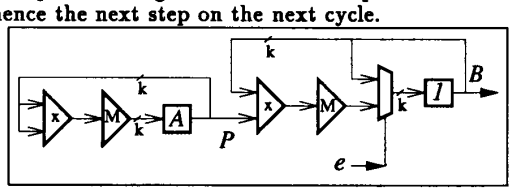

Figure 2.

When the underlying modular multiplications can be implemented free from such pipeline bubbles, it is natural to implement Algorithm H with *only one* k bit modular multiplier as in fig. 3, which is used for both squaring B and for multiplying B by P. The number of cycles for computing a k bit modular exponential in this way is $sk(k+\nu_2(k))$, where s is the cycle time per bit of modular product. On the average, this is only 1.5 times slower than the two multipliers design based on algorithm L; with only *half* the hardware. This is actually 1.33 times faster on the average than an implementation of algorithm L with a single time-multiplexed hardware multiplier as in [OK 91], because every cycle is productive.

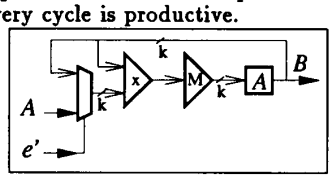

Figure 3.

4.2 Star Chains

It is well known that the binary methods are not optimal. The most efficient known techniques for evaluating powers involve *addition chains*, in which each multiply takes as input the values of two previous multiplies, starting from the initial values A and 1 [Y 91]. Both binary methods are special cases of addition chains. A *star chain* (see [K 81]) is an addition chain where one of the operands is the result of the previous multiply operation. This restriction is desirable in

[7] the index of all our for loops is initialized to zero, and stepped by one.

[8] In our schemas, two triangles labelled x and M denote a multiplier modulo M. A square denotes a register, with initial value indicated inside. A trapezoid represents a multiplexer, controlled by its vertical input. It is the bit-serial exponent e, from low to high bits in fig. 2, and e' from high to low bits in fig. 3.

hardware structures because it allows the hardwiring of one of the inputs to the multiplier. Despite this restriction, star-chains are known (see [K 81] again) to be almost as efficient as general addition chains.

Asymptotically, an optimal star chain requires only k multiplies, against $k+\nu_2(k)$ for the binary methods. The sequence of multiplies only depends upon the exponent E and can thus be computed off-line. However, there is no efficient algorithm known to compute the optimal star chain sequence.

The modular multiplier in fig. 3 can be modified so as to exponentiate along any star chain, provided that input A is able to take one of its operands from a memory in which intermediate results have been saved. For $k = 512b$ the required memory is less than $64kB$.

An alternative to star chains which requires less storage and is easier to compute is to use the $\beta - ary$ method of exponentiation. We pre-compute a table of the powers $A^k \pmod M$ for all small $k < \beta$, ($\beta = 2^p$). The exponential A^E is then obtained by a repeated sequence of p squarings followed by a multiplication by the appropriate power of A. If we allow the multiply to occur early in the squaring sequence we only need to store the odd powers of A. This is a simple generalization of algorithm H to radix β, with $E = [e_{\frac{k}{p}-1} \cdots e_0]_\beta$ and $0 \leq e_i < \beta$ for $i < \frac{k}{p}$. The storage required is $kp/2$ bits; as the squaring sequence need not start until the second most significant digit, the expected number of modular products is (assuming that k is a multiple of p):

$$\frac{\beta}{2} + k - p + \frac{k}{p}\frac{\beta - 1}{\beta}.$$

In software, for $k = 256$ (corresponding to a public modulus of 512b) the optimal choice of p is 5 ($\beta = 32$) and the average number of multiplies $1.24k$; for $k = 512$ the optimal p is 6 and the average number of multiplies is $1.22k$.

With a fast host, the computation of small powers can be done in software while the hardware completes the previous exponentiation thus eliminating the $\beta/2$ term corresponding to the number of products required for building the table.

5 Hensel's Odd Division

There are two dual ways to divide a $k+p$ bit integer

$$N = [n_{k+p-1} \cdots n_0]_2$$

by a k bit integer M, so as to compute $N \pmod M$.

1. The euclidian binary division proceeds from *high to low* bits in N:

   Algorithm 4 (Euclid's division) *Starting from* $R[0] = [n_{k+p-1} \cdots n_{p+1}]_2$, *compute:*

```
for i<=p do
  {  S[i] = 2 R[i] + n{p-i};
     q{i} = if S[i]<M then 0 else 1;
     R[i+1]  = S[i] - q{i}M };
 R = R[p+1].
```

   Upon termination, we obtain Euclid's relation:

$$N = MQ + R,\ R < M,$$

   with quotient $Q = [q_0 \cdots q_{p-1}]_2$. The corresponding hardware scheme is:

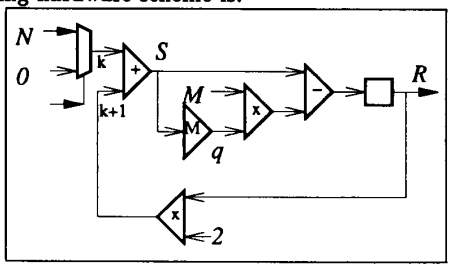

Figure 4.

2. Hensel introduced the odd division around 1900, for computing the inverses of odd 2adic numbers. This implies that the modulus $M = 1+2M'$ must be odd.

   Algorithm 5 (Hensel's division) *Starting from* $R[0] = [n_{k+p-1} \cdots n_0]_2$, *compute:*

```
for i<p do
  {  q{i} = R[i] mod 2;
     R[i+1]  = ( R[i] + q{i}M ) div 2 };
 R = R[p].
```

   Upon termination, we obtain Hensel's relation:

$$N = -MQ + 2^p R,\ R < 2M,$$

   with quotient $Q = [q_{p-1} \cdots q_0]_2$. The corresponding hardware scheme is:

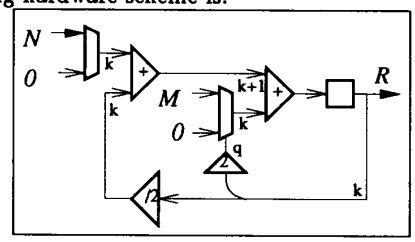

Figure 5.

Hensel's division computes $N2^{-p} \pmod M$ rather than Euclid's result $N \pmod M$ making it not appropriate for all applications; however, in hardware terms, Hensel's division has a decisive advantage over Euclid's: it *does not require* to implement *a quotient unit*. The quotient q_i in Hensel's division is simply the current low bit of the R register. Euclid's division requires a *full k bits comparison* between M and the current value of $S[i]$ in order to compute the corresponding quotient bit q_i.

Most previously reported hardware implementations of RSA deal with modular reduction in Euclid's way: they avoid carrying out the full-compare quotient by using an approximate quotient computation which only involves a *small* number of high bits in S. The resulting *redundant* quotient has *at least* one more bit than Euclid strictly demands. Examples of such methods are found in [PV 90] which uses radix 4, [OK 91][9] which uses radix 32, and [IWSD 92]. An alternative is given by [T 91] which uses base 4 quotient digits in a redundant 3b per digit quotient system. As a consequence, the modular multiplier in [T 91] has an area which is (at least) 1.5 larger than the Hensel based implementation which follows.

One last advantage of Hensel's division is to allow for quotient pipelining, as shown in section 8.2. We are not aware of any similar pipelining technique for Euclid's division.

6 Modular Product

There are *many* ways to compute modular products

$$P = A \times B \pmod M.$$

As shown in the previous section, one has to first choose the order in which to process the multiplier.

Hensel's scheme (from low bits to high bits) has been used in software implementations of RSA, and [DK 91] present some of the benefits. Even [E90] presents a hardware design for modular exponentiation based on Hensel's scheme. Ours appears to be the first reported *working hardware* implementation of RSA to operate in this manner. The following is a generalization of an original algorithm in [Mo 85]:

Algorithm 6 (Modular Product) *Let $A, B, M \in \mathbf{N}$ be three integers, each presented by n radix $\beta = 2^p$ digits*

$$A = [a_{n-1} \cdots a_0]_\beta,$$
$$B = [b_{n-1} \cdots b_0]_\beta,$$
$$M = [m_{n-1} \cdots m_0]_\beta,$$

with the modulus M relatively prime to β, that is $gcd(m_0, \beta) = 1$. We compute a product $P = P[n]$ such that

$$P = \frac{A \times B}{\beta^n} \pmod M, \qquad (1)$$

by letting $P[0] = 0$ and evaluating, for $t = 0, \cdots, n-1$:

$$P[t+1] = \frac{1}{\beta}(A \times b_t + P[t] + q_t \times M). \qquad (2)$$

At each step (2) the quotient digit $q_t \in [0 \cdot \cdot \beta - 1]$ is chosen so that $P[t+1]$ is an integer:

$$A \times b_t + P[t] + q_t \times M = 0 \pmod \beta.$$

This is achieved by letting

$$q_t = \mu \times (a_0 \times b_t + p_0(t)) \pmod \beta, \qquad (3)$$

where number $\mu = -M^{-1} \pmod \beta$ is pre-computed so that $m_0 \times \mu + 1 = 0 \pmod \beta$, and $p_0(t) = P[t] \pmod \beta$ denotes the least significant digit of $P[t]$.

It is easily checked, by induction on t, that

$$\beta^t \times P[t] = A \times [b_{t-1} \cdots b_0]_\beta + M \times [q_{t-1} \cdots q_0]_\beta$$

hence (1) follows for $t = n$. As a numerical example, let us set

$$\beta = 2,\ n = 5,$$
$$A = 26 = [11010]_2,$$
$$B = 11 = [01011]_2,$$
$$M = 19 = [10011]_2$$

and use Algorithm 1 to compute

t	0	1	2	3	4
$P[t+1]$	13	29	24	25	22
q_t	0	1	1	0	1

thus establishing the diophantine equation:

$$2^5 \times 22 = 26 \times 11 + 22 \times 19.$$

Observe that the values of $P[t]$ remain bounded: indeed, a simple induction on t establishes that

$$0 \le P[t] < A + M \qquad (4)$$

is an invariant of Algorithm 1. It follows that n digits plus *one bit* are sufficient to represent $P[t] < 2\beta^n$, for all $t \ge 0$. Equation (4) also shows that, assuming $0 \le A < M$, we can compute $P = \frac{A \times B}{\beta^n} \pmod M$ with $0 \le P < M$ with just one conditional subtraction following Algorithm 1:

$$P = \text{if } P[n] \ge M \text{ then } P[n] - M \text{ else } P[n].$$

For modular exponential, we avoid performing this reduction after each modular product by letting all intermediate results have *two extra bits* of precision;

[9] [OK 91] present their quotient unit as a *parallel exhaustive search*.

all operands can nevertheless be represented with n digits since:

$$\text{if } M < \frac{1}{4}\beta^n \text{ and } A, B \leq 2M \quad \text{then} \quad P[n] < 2M \quad (5)$$

Indeed, since $P[n-1] < A + M$ by (4) and (5) implies $b_{n-1} \leq \frac{\beta}{2} - 1$, we have:

$$\begin{aligned} P[n] &= \frac{1}{\beta}(P[n-1] + b_{n-1}A + q_{n-1}M) \\ &< \frac{1}{\beta}(A + M + (\frac{\beta}{2} - 1)A + (\beta - 1)M) \\ &< \frac{A}{2} + M \leq 2M. \end{aligned}$$

6.1 Radix Choice

Algorithm 6 requires to choose the radix $\beta = 2^p$ in which to decompose the $n = \frac{k}{p}$ digits multiplicand

$$B = [b_{n-1} \cdots b_0]_\beta.$$

Let us analyse the impact of the choice of radix β on a k bits keys RSA implementation.

$\beta = 2^k$ In [SBV 91], we compute modular products through a long integer hardware multiplier, which is forcing the largest base $\beta = 2^k$ upon us. The modular product is realized by a sequence of three full-length $k \times k \mapsto 2k$ integer products:

1. Compute the $2k$ bit product $C = A \times B$; let $C_0 = C \pmod{2^k}$ and $C_1 = C \div 2^k$, so that $C = C_0 + 2^k C_1$.
2. Compute the $2k$ bit product $Q = C_0 \times -M^-$, where M^- is the precomputed inverse of $M \pmod{2^k}$. Keep the k low order bits as hensel quotient $q = Q \pmod{2^k}$.
3. Compute the $2k$ bit product $D = q \times M$, and add: $P = C + D$; number P is such that $P = \frac{A \times B}{2^k} \pmod{M}$ with $P \leq 2M$.

The resulting complexity is high: $6hk^2$, and we see that using large radixes is not efficient.

$\beta = 2^{32}$ Let $\mathcal{M}(p)$ represent the cost of a $p \times p \rightarrow 2p$ *bit* multiply. In step 2 of algorithm 6, the products $A \times b_t$ and $q_t \times M$ each cost $\frac{k}{p}\mathcal{M}(p)$, and the computation of q_t costs $\mathcal{M}(p)$. The multiply cost of Algorithm 6 is thus:

$$2k^2\frac{\mathcal{M}(p)}{p^2} + k\frac{\mathcal{M}(p)}{p} \quad (6)$$

In software at the moderate bit lengths of RSA, the cost of multiplication grows quadratically with the length of the operands. Thus the first term of (6) is constant while the second term grows with p. So, software should choose as small a radix β as possible: typically one machine word.

$\beta = 2^2$ In a dedicated hardware, equation (6) shows that p should also be small. The choice $\beta = 2^2$ allows for a trivial calculation of q_t, and permits to use Booth recoded multiplication: this doubles the multiplier's performance compared to $\beta = 2$, at a modest increase (about 1.5) in hardware area. Higher radixes which offer better multiply performance had to be dismissed, since they involve too much hardware and the computation of the quotient digits is no longer trivial.

7 Software Implementations

7.1 Karatsuba Multiplication

In software the choice of radix β is determined by the most efficient multiply primitive available, which is not always the machine word size. On most current microprocessors, integer multiplication is relatively slow, compared to adds which get executed in one clock cycle.

On the MIPS R3000A our most efficient multiply primitive is a $64 \times 64 \rightarrow 128$ *bits*[10]. This is implemented in 51 cycles using three integer multiplications $32 \times 32 \rightarrow 64$ *bit* and the Karatsuba algorithm [K 81]. The multiply primitive on a MIPS takes 16 cycles, and all other operations are overlapped. Careful coding can hide all but three cycles of the Karatsuba overhead into the (otherwise wasted) cycles during which the integer multiply unit is busy. Trying to contruct larger multiplies by applying Karatsuba's algorithm recursively does not result in a performance improvement: there are not enough registers to hold intermediate results, and all of the idle cycles are already absorbed by the first level of Karatsuba.

The argument following equation (6) suggests that we should choose a radix equal to the word size. However, with Karatsuba, the optimal radix choice increases. For example, on a MIPS R3000A the conjunction of radix 2^{64} and Karatsuba multiplication brings a speed-up of 1.22.

On some computers, it is faster to forget about integer multiplies and Karatsuba altogether, and use floating point multiplications[11]. However the lower result precision of floating point implies a smaller digit size (typically 24 bits); moving digits between the floating point and the integer unit also introduces significant overheads.

7.2 Optimized Squaring

A further optimization is to treat squaring specially. By rearranging the order of operations in Algorithm 6 we can perform $A \times B$ before the modular reduction.

[10] likewise on the Alpha AXP we use $128 \times 128 \rightarrow 256$ *bits* as our multiply primitive

[11] even more so on vector supercomputers [BW 89].

With the use of star chains, or the precomputation of small powers in over 75% of the modular product operations $A = B$. For $k = 512$ and $p = 64$ (using Karatsuba's algorithm), squaring may be implemented 1.77 times more efficiently than general multiplication. This yields an overall speed-up of 1.29.

In hardware this would require extra storage for the length $2k$ intermediate result, but in software memory is cheap.

8 Hardware Implementations

8.1 Asynchronous Carry Completion

In our hardware implementation of RSA we represent the partial sums $P[t]$ in carry-save form. Upon completion of each modular product, the result $P[n]$ must be converted back to non-redundant binary form, so as to be used as input to the next modular product.

Takagi [T 91] avoids this problem by keeping all the operands in carry-save form; the size of the resulting multiplier is nearly *double* that of a radix 4 non-redundant multiplier.

Instead we observe that although in the worst the carry must propagate through all k bits of the result, on average it will only need to propagate through $l_2(k)$ bits before all carries have disappeared. Thus we have implemented an *asynchronous carry completion* detection circuit and clock the final result for as many cycles as needed to fully propagate all carries. This circuit is a very wide OR-gate that collects together all the non-zero carries. This OR-gate need not run at full circuit speed since the carry propagation circuit that the OR-gate monitors is idempotent once all the carries have been eliminated and may be safely clocked for extra cycles without changing the result. The OR-gate provides an asynchronous input to the controller indicating that the datapath is ready to commence the next modular product. Measurements from the implemented hardware show that the average number of carry propagation cycles is indeed very close to $l_2(k)$, as predicted in [K 81]. This technique provides a valuable saving in multiplier area, for small increase in the *average* numbers of cycles per full modular product.

8.2 Quotient Pipelining

Recall the basic steps of Algorithm 6:

$$P[t+1] = \frac{1}{\beta}(A \times b_t + P[t] + q_t \times M), \quad (7)$$

$$q_t = \mu \times (a_0 \times b_t + p_0(t)) \pmod{\beta}. \quad (8)$$

The direct implementation of this recurrence suffers from the dependency between q_t and the current value of $P[t]$: this resulting combinatorial path directly affects the minimum cycle time of the data-path.

In order to speed-up the clock cycle through pipelining, let us define r_t and R_t by:

$$r_t = P[t] + A \times b_t \pmod{\beta}$$
$$R_t = P[t] + A \times b_t - r_t$$

Equation (7) becomes:

$$P[t+1] = \frac{R_t}{\beta} + \frac{(r_t + q_t \times M)}{\beta}.$$

We introduce d levels of pipeline by choosing a quotient digit q'_t such that:

$$r_t + q'_t \times M = 0 \pmod{\beta^{d+1}},$$

with $0 \le q'_t < \beta^{d+1}$. We obtain the modified recurrence:

$$P'[t+1] = \frac{R_t}{\beta} + \frac{(r_{t-d} + q'_{t-d} \times M)}{\beta^{d+1}}$$

This value is related to the old recurrence by:

$$P[t+1] \equiv P'[t+1] + \frac{r_t}{\beta} + \cdots + \frac{r_{t-d+1}}{\beta^d} \pmod{\beta^n}$$

Now,

$$P[n] = \frac{A \times B}{\beta^n} \pmod{\beta^n}.$$

Letting the datapath run for d more iterations gives:

$$\begin{aligned} P[n+d] &= P'[n+d] + \frac{r_{n+d-1}}{\beta} + \cdots \\ &\quad + \frac{r_n}{\beta^d} \pmod{\beta^n} \\ &= \frac{A \times B}{\beta^{n+d}} \pmod{\beta^n}. \end{aligned}$$

Multiplying this result by β^d, which is equivalent to shifting left by d bits, we obtain:

$$\begin{aligned} P[n+d] \times \beta^d &= P'[n+d] \times \beta^d + r_{n+d-1} \times \beta^{d-1} \\ &\quad + \cdots + r_n \pmod{\beta^n} \\ &= \frac{A \times B}{\beta^n} \pmod{\beta^n} \end{aligned}$$

The r_t's produced after the nth iteration are appended to the result in $P'[n+d]$ in order to produce the final modular product.

In this pipelined version, the data-path requires d extra bits of precision, and the iteration has $n + d$ cycles, which is d cycles (plus a few additional cycles to realign the final result) longer than the initial one. This cost is more than compensated for by the faster cycle time which pipelining now allows.

The choice of d is technology dependent. Making it unnecessarily large consumes cycles and area[12], but it should be sufficiently large that each step in the distribution of q_t's through the datapath is no greater than other critical paths of the datapath. In our implementation on the PAM the pipelined data-path can be clocked at 25ns. In a non-pipelined version the combinatorial distribution of q_t takes over 100ns. Thus in this particular technology quotient pipelining gives a speed-up of 4.

9 Summary of Speedups

In the following table we recall the various techniques applied to implementing RSA and quantify the speedup achieved for $1Kb$ keys.

TECHNIQUE	Software	Hardware
Chinese remainders	$4-\epsilon$	4
Precompute small powers	1.2	1.25
Hensel's odd division	1.05	1.5
Karatsuba multiplication	1.22	—
Squaring optimization	1.29	—
Carry completion adder	—	$2-l_2(k)/k$
Quotient pipelining	—	4

10 Bibliography

[**Br 89**] Ernest F. Brickell *A Survey of Hardware Implementations of RSA,* in Gilles Brassard, editor, Advances in Cryptology – Crypto '89 , pp 368-370, Springer-Verlag, 1990.

[**BRV 89**] P. Bertin, D. Roncin , J. Vuillemin *Introduction to Programmable Active Memories,* in Systolic Array Processors edited by J. McCanny, J. McWhirter and E. Swartzlander , Prentice Hall, pp 301-309, 1989. Also available as PRL report 3 , Digital Equipment Corp., Paris Research Laboratory, 85, Av. Victor Hugo. 92563 Rueil-Malmaison Cedex, France.

[**BRV 92**] P. Bertin, D. Roncin, J. Vuillemin: *Programmable Active Memories: a Performance Assessment,* report in preparation, Digital Equipment Corp., Paris Research Laboratory, 85, Av. Victor Hugo. 92563 Rueil-Malmaison Cedex, France, 1992.

[**BW 89**] D. A. Buell, R. L. Ward, *A Multiprecise Integer Arithmetic Package,* The journal of Supercomputing 3, pp 89-107; Kluwer Academic Publishers, Boston 1989.

[**DK 91**] S. R. Dussé, B. S. Kaliski Jr.: *A Cryptographic Library for the Motorola DSP 56000,* Proceedings of EUROCRYPT '90, Springer LNCS 473, 1991.

[12] due to the larger intermediate results.

[**E90**] S. Even: *Systolic Modular Multiplication,* Proceedings of Crypto '90 pp 619-624, Springer-Verlag, 1990.

[**IWSD 92**] P. A. Ivey, S. N. Walker, J. M. Stern and S. Davidson: *An ultra-high speed public key encryption processor,* Proceedings of the IEEE 1992 custom integrated circuits conference, Boston, Massachusetts, paper 19.6, 1992.

[**K 81**] D. E. Knuth, The Art of Computer Programming, vol. 2, Seminumerical Algorithms, Addison Wesley, 1981.

[**LLMP 92**] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, J. Pollard: *The factorization of the ninth Fermat number,* Mathematics of Computation to appear, 1992.

[**Mo 85**] P. L. Montgomery *Modular multiplication without trial division,* Mathematics of Computation 44(170):519-521, 1985.

[**PV 90**] J. Vuillemin, F.P. Preparata *Practical Cellular Dividers,* IEEE Trans. on Computers, 39(5):605-614, 1990.

[**OK 91**] H. Orup, P. Kornerup: *A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N,* 10-th IEEE symposium on COMPUTER ARITHMETIC, pp 51-57, 1991.

[**QC 82**] J-J. Quisquater, C. Couvreur *Fast Decipherment Algorithm for RSA Public-key Cryptosystem,* Electronics Letters, 18(21):905-907, 1982.

[**RSA 78**] R. L. Rivest, A. Shamir, L. Adleman *Public key cryptography,* CACM 21, 120-126, 1978.

[**SBV 91**] M. Shand, P. Bertin and J. Vuillemin: *Hardware Speedups in Long Integer Multiplication,* Computer Architecture News, 19(1):106-114, 1991.

[**T 91**] N. Takagi *A Radix-4 Modular Multiplication Hardware Algorithm Efficient for Iterative Modular Multiplications,* 10-th IEEE symposium on COMPUTER ARITHMETIC, pp 35-42, 1991.

[**X**] Xilinx *The Programmable Gate Array Data Book,* Product Briefs, Xilinx, Inc., 1987-1992.

[**Y 91**] Y. Yacobi *Exponentiating Faster with Addition Chains,* Proceedings of EUROCRYPT '90, Springer LNCS 473, 1991.

11 Acknowledgments

P.Bertin, F.Morain, R. Razdan and the anonymous referee.