Simplifying Quotient Determination in High-Radix Modular Multiplication*

Holger Orup
Computer Science Department
Aarhus University
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, DENMARK
e-mail: orup@daimi.aau.dk

Abstract

Until now the use of high radices to implement modular multiplication has been questioned, because it involves complex determination of quotient digits for the modulo reduction. This paper presents algorithms that are obtained through rewriting of Montgomery's algorithm. The determination of quotients becomes trivial and the cycle time becomes independent of the choice of radix. It is discussed how the critical path in the loop can be reduced to a single shift-and-add operation. This implies that a true speed up is achieved by choosing higher radices.

1 Introduction

Since the introduction of public key crypto systems [1, 12] considerable effort has been directed toward fast hardware implementation of modular multiplication of very large integer operands. A review of techniques for speeding up modular multiplication is included in [4]. It is recognized that *quotient determination*, i.e. determination of the multiple of modulus to subtract at each reduction stage, is the critical operation [4, 13]. This is the reason why, during the last five years, Montgomery's modular multiplication method [7] has been considered the best candidate for faster implementations. Compared to traditional SRT division, the method requires additional pre- and post-processing, but the time for this additional processing becomes negligible when several modular multiplications have to be performed on intermediate results, as is the case when calculating modular exponentials.

In our previous work [11, 10], we have studied the possibilities of speeding up modular multiplication by using higher radices. A single chip modular exponentiation processor using radix 32 multiplication has been successfully implemented [9]. It is based on a traditional division method and is capable of exponentiating 560 bit operands in less than 5.5 ms, corresponding to a throughput of more than 100 Kbit/s, at a clocking frequency of 25 MHz. According to our knowledge, this is the fastest *single chip* implementation for performing modular exponentials. Only one implementation [13] has been reported to be faster. The high radix approach has been criticized [4, 13] for a large hardware depth (meaning a slow clocking frequency), for a large hardware consumption and for having a non-trivial determination of quotient digits. This is also our experience, but the high radix approach gives potential for substantial speed improvements of modular multiplication. In the rest of this paper we will rewrite the original algorithm of Montgomery and show how this leads to a high-radix algorithm, suited for hardware implementation, where the quotient determination becomes trivial, and the obtainable clocking frequency is independent of the choice of radix.

2 High-radix modular multiplication algorithm

In this section we will rewrite Montgomery's algorithm through a series of development steps. We will show how this leads to an algorithm, where the quotient determination is trivial. Each of the presented algorithms is supplied with an invariant, stating the algebraic relation between the stimulus and the intermediate result, and an upper bound for the range of the intermediate result. The range condition of stimulus has been matched such that the response of the modular multiplication algorithm can be used as stimulus for the same algorithm without additional processing. This has implications on the usefulness of the algorithms for e.g. modular exponentiation. All algorithms are expressed in terms of non-redundant radix 2^k digit sets. This is done in order to limit the description, but the ideas apply as well to other digit sets. See [5] for a high-radix version of Montgomery's algorithm using a symmetric re-

*This work has been supported by the Danish Natural Science Research Council, grant no. 5.21.08.02.

dundant digit set. Additional processing, that may have to be performed when Montgomery's method for modular multiplication is used, is discussed in [7, 4, 5]. The first algorithm is a radix 2^k version of the algorithm proposed by Peter L. Montgomery [7] for multiplying two integers modulo M.

Algorithm 1
(Radix 2^k Montgomery Modular Multiplication)

Stimulus:

A modulus $M > 2$ with $\gcd(M,2) = 1$ and positive integers k, n such that $4M < 2^{kn}$.
The integers R^{-1} and M' are given such that $(2^{kn}R^{-1}) \bmod M = 1$ and $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, where $0 \leq A \leq 2M$, and integer multiplier $B = \sum_{i=0}^{n-1}(2^k)^i b_i$, where digit $b_i \in \{0, 1, \ldots, 2^k - 1\}$ and $0 \leq B \leq 2M$.

Response:

An integer S_n such that $S_n \equiv ABR^{-1} \pmod M$ and $0 \leq S_n < 2M$.

Method:

```
$S_0 := 0$;
for $i := 0$ to $n - 1$ do
  $L:\ q_i := (((S_i + b_i A) \bmod 2^k)M') \bmod 2^k$;
      $S_{i+1} := (S_i + q_i M + b_i A) \operatorname{div} 2^k$;
end
```

Where:

The following invariant holds at label L:

$$\begin{aligned} 2^{ki}S_i &= A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} + M \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \ \bigwedge \\ 0 &\leq S_i \\ &< A + M. \end{aligned}$$

Correctness: The condition $\gcd(M,2) = 1$ is sufficient to ensure the existence of R^{-1} and M'. To establish the invariant, note that $q_i \equiv (S_i + b_i A)M' \pmod{2^k}$ so $q_i M \equiv -(S_i + b_i A) \pmod{2^k}$ and, hence, that 2^k divides $S_i + q_i M + b_i A$ in the updating of S_{i+1}. The invariant holds trivially for $i = 0$. Assuming it holds for $i = \ell$, from the updating of $S_{\ell+1}$, $2^k S_{\ell+1} = S_\ell + q_\ell M + b_\ell A$, we then obtain:

$$\begin{aligned} 2^{k(\ell+1)}S_{\ell+1} &= A \cdot \sum_{j=0}^{\ell-1} b_j 2^{kj} + M \cdot \sum_{j=0}^{\ell-1} q_j 2^{kj} \\ &\quad + 2^{k\ell} q_\ell M + 2^{k\ell} b_\ell A \\ &= A \cdot \sum_{j=0}^{\ell} b_j 2^{kj} + M \cdot \sum_{j=0}^{\ell} q_j 2^{kj}. \end{aligned}$$

Hence the first part of the invariant holds for $i = \ell + 1$. The last part follows from $0 \leq q_i \leq 2^k - 1$ and $0 \leq b_i \leq 2^k - 1$:

$$\begin{aligned} 2^{k(\ell+1)}S_{\ell+1} &\leq A \cdot (2^k - 1)\frac{2^{k(\ell+1)} - 1}{2^k - 1} \\ &\quad + M \cdot (2^k - 1)\frac{2^{k(\ell+1)} - 1}{2^k - 1} \\ S_{\ell+1} &< A + M. \end{aligned}$$

By inserting conditions of stimulus we find upon exit from the loop:

$$\begin{aligned} 2^{kn}S_n &= AB + M \cdot \sum_{j=0}^{n-1} q_j 2^{kj} \quad \text{and} \\ 0 &\leq 2^{kn}S_n \\ &< 2M \cdot 2M + M \cdot 2^{kn} \\ &< M \cdot 2^{kn} + M \cdot 2^{kn}. \end{aligned}$$

So $R^{-1}2^{kn}S_n \equiv S_n \pmod M \equiv ABR^{-1} \pmod M$ and $0 \leq S_n < 2M$ which proves the correctness of Algorithm 1. □

Montgomery's method for modular multiplication has been implemented for a standard DSP processor [2] and for a board of field programmable gate arrays [13]. Further, in [3, 17, 4, 6] some suggestions for hardware implementations are described. Apart from the DSP implementation, where the radix is determined from the available instruction set, all proposals for a hardware implementation end up with choosing radix 2 or radix 4.

However, for a given operand size, the number of iterations in Algorithm 1 can be reduced by choosing a larger radix. But it is not obvious that this leads to a smaller computation time. The time for an iteration increases for higher radices. This is mainly due to the quotient determination which requires a k bit addition and a $k \times k$ bit multiplication. In the updating of S_{i+1}, the calculation of multiples and addition can be efficiently performed by constant time adders, e.g. carry save adders, [11, 10]. Still the carry-out from the k least significant bits of $S_i + q_i M + b_i A$ must be computed in each iteration. Because the two statements of the loop are strictly sequential, we are not able to reduce the computation time by overlapping the execution of the statements.

2.1 Avoiding multiplication in quotient determination

In the case of radix 2 , i.e. $k = 1$, the multiplication operation in the quotient determination in Algorithm 1 is avoided. Because $M' \bmod 2 = 1$, the statement reduces to $q_i := (S_i + b_i A) \bmod 2$. For all values of modulus, having the property $M' \bmod 2^k = 1$, the multiplication operation is avoided in the general case of radix 2^k. This observation leads us to transform modulus M to a new value $\widetilde{M}$ that possesses the wanted property. The transformation is simple, $\widetilde{M} = (M' \bmod 2^k)M$, and only has to be performed once. The resulting algorithm is:

Algorithm 2
(Avoiding Multiplication in Quotient Determination)

Stimulus:

A modulus $M > 2$ *with* $\gcd(M,2) = 1$ *and positive integers* k, n *such that* $4\widetilde{M} < 2^{kn}$, *where* $\widetilde{M}$ *is given by* $\widetilde{M} = (M' \bmod 2^k)M$.
The integers R^{-1} *and* M' *are given such that* $(2^{kn}R^{-1}) \bmod M = 1$ *and* $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, *where* $0 \le A \le 2\widetilde{M}$, *and integer multiplier* $B = \sum_{i=0}^{n-1}(2^k)^i b_i$, *where digit* $b_i \in \{0, 1, \ldots, 2^k - 1\}$ *and* $0 \le B \le 2\widetilde{M}$.

Response:

An integer S_n *such that* $S_n \equiv ABR^{-1} \pmod{M}$ *and* $0 \le S_n < 2\widetilde{M}$.

Method:

```
$S_0 := 0$;
for $i := 0$ to $n - 1$ do
  $L$ : $q_i$     := $(S_i + b_i A) \bmod 2^k$;
        $S_{i+1}$ := $(S_i + q_i\widetilde{M} + b_i A) \operatorname{div} 2^k$;
end
```

Where:

The following invariant holds at label L:

$$\begin{aligned} 2^{ki}S_i &= A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \quad \wedge \\ 0 &\le S_i \\ &< A + \widetilde{M}. \end{aligned}$$

Correctness: Algorithm 2 is verified by using $\widetilde{M} = (M' \bmod 2^k)M$ and $\widetilde{M} \equiv -1 \pmod{2^k}$. □

Transforming M into $\widetilde{M}$ corresponds to moving the common factor $M' \pmod{2^k}$ from the quotient determination to the updating of S_{i+1}. Hereby, a single initial multiplication replaces a multiplication in each iteration. The penalty of using this algorithm is a larger range of the resulting S_n and a value of n that has increased by at most one.

2.2 Avoiding addition in quotient determination

We can further reduce the quotient determination complexity by replacing A by $2^k A$. This technique is also used in [4] and [5]. Since the expression $(S_i + b_i A) \bmod 2^k$ in Algorithm 2 then reduces to $S_i \bmod 2^k$, we have avoided the addition operation. In the update of S_{i+1} we replace $(S_i + q_i\widetilde{M} + b_i A) \operatorname{div} 2^k$ by $(S_i + q_i\widetilde{M}) \operatorname{div} 2^k + b_i A$. Compared to Algorithm 2 the number of iterations is increased by one to compensate for the extra factor 2^k:

Algorithm 3
(Avoiding Multiplication and Addition in Quotient Determination)

Stimulus:

A modulus $M > 2$ *with* $\gcd(M,2) = 1$ *and positive integers* k, n *such that* $4\widetilde{M} < 2^{kn}$, *where* $\widetilde{M}$ *is given by* $\widetilde{M} = (M' \bmod 2^k)M$.
The integers R^{-1} *and* M' *are given such that* $(2^{kn}R^{-1}) \bmod M = 1$ *and* $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, *where* $0 \le A \le 2\widetilde{M}$, *and integer multiplier* $B = \sum_{i=0}^{n}(2^k)^i b_i$, *where digit* $b_n = 0$, $b_i \in \{0, 1, \ldots, 2^k - 1\}$ *for* $0 \le i < n$ *and* $0 \le B \le 2\widetilde{M}$.

Response:

An integer S_{n+1} *where* $S_{n+1} \equiv ABR^{-1} \pmod{M}$ *and* $0 \le S_{n+1} < 2\widetilde{M}$.

Method:

```
$S_0 := 0$;
for $i := 0$ to $n$ do
  $L_1$ : $q_i$     := $S_i \bmod 2^k$;
  $L_2$ : $S_{i+1}$ := $(S_i + q_i\widetilde{M}) \operatorname{div} 2^k + b_i A$;
end
```

Where:

The following invariant holds at label L_1:

$$\begin{aligned} 2^{ki}S_i &= 2^k A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \quad \wedge \\ 0 &\le S_i \\ &< 2^k A + \widetilde{M}. \end{aligned}$$

Correctness: To verify the response, we note that $q_0 = 0$ and $b_n = 0$, hence upon exit from the loop we get:

$$\begin{aligned} 2^{k(n+1)}S_{n+1} &= 2^k A \cdot \sum_{j=0}^{n} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n} q_j 2^{kj} \\ 2^{kn}S_{n+1} &= A \cdot \sum_{j=0}^{n-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n-1} q_{j+1} 2^{kj}. \end{aligned}$$

So $S_{n+1} \equiv ABR^{-1} \pmod{M}$ and $0 \le S_{n+1} < 2\widetilde{M}$. □

Noting that $q_i = S_i \bmod 2^k$ and that $\widetilde{M}+1$ is divisible by 2^k, we can rewrite the statement at label L_2 in the loop:

$$\begin{aligned}
&(S_i + q_i\widetilde{M}) \operatorname{div} 2^k + b_iA \\
&= S_i \operatorname{div} 2^k + (q_i\widetilde{M} + S_i \bmod 2^k) \operatorname{div} 2^k + b_iA \\
&= S_i \operatorname{div} 2^k + (q_i(\widetilde{M}+1)) \operatorname{div} 2^k + b_iA \\
&= S_i \operatorname{div} 2^k + q_i((\widetilde{M}+1) \operatorname{div} 2^k) + b_iA.
\end{aligned}$$

The same approach is used for a radix 2 version of Montgomery modular multiplication in [6]. By calculating $(\widetilde{M}+1) \operatorname{div} 2^k$ once for each new value of M, this is a simplification of the statement at label L_2. Now we do not have to calculate the carry-out from the k least significant bits of $S_i + q_i\widetilde{M}$ in the updating statement.

2.3 Utilizing quotient pipelining in modular multiplication

In [13] Montgomery's modular multiplication algorithm has been modified by applying *quotient pipelining*. The idea is to delay the use of quotient digit q_{i-d}, determined from information available in iteration $i-d$, by d iterations. The effect is that d iterations are now available for determining a quotient. In [13] the penalty is d extra iterations and an increased quotient digit range, $q_{i-d} \in \{0,1,\ldots,2^{k(d+1)-1}\}$. In Algorithm 4 we have pipelined the quotient determination of Algorithm 3 without increasing the quotient digit range but instead increasing the range of the result:

Algorithm 4
(Modular Multiplication with Quotient Pipelining)

Stimulus:

A modulus $M > 2$ with $\gcd(M,2) = 1$ and positive integers k,n such that $4\widetilde{M} < 2^{kn}$, where $\widetilde{M}$ is given by $\widetilde{M} = (M' \bmod 2^{k(d+1)})M$ and integer $d \geq 0$ is a delay parameter.
Integer R^{-1}, where $(2^{kn}R^{-1}) \bmod M = 1$, and integer M', where $(-MM') \bmod 2^{k(d+1)} = 1$, are given.
Integer multiplicand A, where $0 \leq A \leq 2\widetilde{M}$, and integer multiplier $B = \sum_{i=0}^{n+d}(2^k)^i b_i$, where digit $b_i \in \{0,1,\ldots,2^k-1\}$ for $0 \leq i < n$, $b_i = 0$ for $i \geq n$ and $0 \leq B \leq 2\widetilde{M}$.

Response:

Integer S_{n+d+2} where $S_{n+d+2} \equiv ABR^{-1} \pmod M$ and $0 \leq S_{n+d+2} < 2\widetilde{M}$.

Method:

```
$S_0 := 0$; $q_{-d} := 0$; $q_{-d+1} := 0$; $\ldots$; $q_{-1} := 0$;
for $i := 0$ to $n+d$ do
    $L_1 : q_i := S_i \bmod 2^k$;
    $L_2 : S_{i+1} := S_i \operatorname{div} 2^k + q_{i-d}((\widetilde{M}+1) \operatorname{div} 2^{k(d+1)}) + b_iA$;
end;
$S_{n+d+2} := 2^{kd}S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1}2^{kj}$;
```

Where:

The following invariant holds at label L_1:

$$\begin{aligned}
2^{ki}S_i + \sum_{j=i-d}^{i-1} q_j2^{kj}
&= 2^kA\cdot\sum_{j=0}^{i-1} b_j2^{kj} + \widetilde{M}\cdot\sum_{j=0}^{i-d-1} q_j2^{kj} \;\wedge \\
0 &\leq S_i \\
&< 2^k(A+M).
\end{aligned}$$

Correctness: To establish the invariant, note that $\widetilde{M} \equiv -1 \pmod{2^{k(d+1)}}$ so $2^{k(d+1)}$ divides $\widetilde{M}+1$ and note that $2^k(S_i \operatorname{div} 2^k) = S_i - q_i$. The invariant holds trivially for $i = 0$. Assuming it holds for $i = \ell$, from the updating of $S_{\ell+1}$ at label L_2, we then obtain:

$$\begin{aligned}
&2^{k(\ell+1)}S_{\ell+1} \\
&= 2^{k(\ell+1)}(S_\ell \operatorname{div} 2^k) \\
&\quad + 2^{k(\ell+1)}q_{\ell-d}((\widetilde{M}+1) \operatorname{div} 2^{k(d+1)}) \\
&\quad + 2^{k(\ell+1)}b_\ell A \\
&= 2^{k\ell}(S_\ell - q_\ell) + 2^{k(\ell-d)}q_{\ell-d}(\widetilde{M}+1) + 2^{k(\ell+1)}b_\ell A \\
&= 2^kA\cdot\sum_{j=0}^{\ell-1} b_j2^{kj} + \widetilde{M}\cdot\sum_{j=0}^{\ell-d-1} q_j2^{kj} \\
&\quad - \sum_{j=\ell-d}^{\ell-1} q_j2^{kj} - 2^{k\ell}q_\ell + 2^{k(\ell-d)}q_{\ell-d} \\
&\quad + 2^{k(\ell-d)}q_{\ell-d}\widetilde{M} + 2^{k(\ell+1)}b_\ell A \\
&= 2^kA\cdot\sum_{j=0}^{\ell} b_j2^{kj} + \widetilde{M}\cdot\sum_{j=0}^{\ell-d} q_j2^{kj} - \sum_{j=\ell+1-d}^{\ell} q_j2^{kj}.
\end{aligned}$$

Hence the first part of the invariant holds for $i = \ell+1$. The last part is established by noting that,

$$\widetilde{M}+1 \leq (2^{k(d+1)}-1)M < 2^{k(d+1)}M.$$

So $(\widetilde{M}+1) \operatorname{div} 2^{k(d+1)} < M$. The updating of $S_{\ell+1}$ at label L_2 then gives:

$$\begin{aligned} S_{\ell+1} &< (2^k(A+M)) \operatorname{div} 2^k \\ &\quad + (2^k-1)M + (2^k-1)A \\ &\le 2^k(A+M). \end{aligned}$$

Hence the last part of the invariant holds for $i = \ell + 1$. To verify the response, we note that $q_0 = 0$ and $b_j = 0$ for $j \ge n$, hence upon exit of the loop we get:

$$\begin{aligned} & 2^{k(n+d+1)} S_{n+d+1} + \sum_{j=n+1}^{n+d} q_j 2^{kj} \\ &= 2^k A \cdot \sum_{j=0}^{n+d} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n} q_j 2^{kj} \\ & 2^{kn}(2^{kd} S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1} 2^{kj}) \\ &= A \cdot \sum_{j=0}^{n-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n-1} q_{j+1} 2^{kj}. \end{aligned}$$

So after the last statement of the algorithm we obtain $S_{n+d+2} \equiv ABR^{-1} \pmod{M}$ and $0 \le S_{n+d+2} < 2\widetilde{M}$. □

The last statement in Algorithm 4 is just a left shift of S_{n+d+1} where the d last quotient digits are shifted in from the right. Algorithm 4 is clearly an improvement of the quotient pipelined version in [13]: There is *no* calculation involved in the quotient determination. The quotient digit range has *not* increased by a factor of 2^d, otherwise implying an increased complexity in the calculation of multiples $q_{i-d}((\widetilde{M}+1) \operatorname{div} 2^{k(d+1)})$, where $(\widetilde{M}+1) \operatorname{div} 2^{k(d+1)}$ is a pre-calculated integer. However, compared to the quotient pipelined algorithm in [13], Algorithm 4 has an increased range of the result. In [13] the pipelining technique was applied in order to perform overlapping calculations of quotient digits, hereby reducing the hardware depth for a radix 4 modular multiplication algorithm. In Algorithm 4 it could seem meaningless to apply pipelining because of the trivial quotient determination. The calculation of multiples becomes more time consuming for higher radices, but we use the pipeline technique for performing overlapping calculations of the multiples $q_{i-d}((\widetilde{M}+1) \operatorname{div} 2^{k(d+1)})$ and $b_i A$. All of these operands are available after iteration $i-d$, and the resulting multiples are not needed before iteration i. If convenient, we could even perform an addition of the multiples before iteration i. Then we have reduced the time for an iteration to the time for a shift and an addition of two words, $S_{i+1} := S_i \operatorname{div} 2^k + T_{i-d}$. Because of the increased range of the result and the increased number of iterations, we should choose d to be as small as possible. The calculation of the multiples can be performed in about $\log_2 k$ steps by two pipelined adder-trees if a redundant representation of the resulting multiples is allowed [16].

3 Example hardware architecture

To get an impression of the speed potential of Algorithm 4 we will discuss a hardware architecture for executing the algorithm. Figure 1 shows an example architecture where $k = 8$ and $d = 3$, i.e. the radix is 2^8 and the architecture has 3 pipeline stages. The architecture includes registers

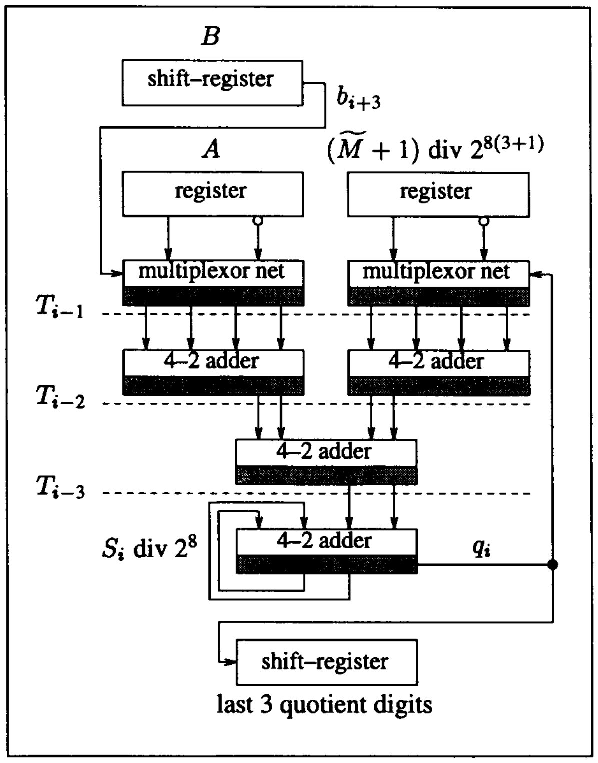

Figure 1: Hardware architecture using radix 2^8 and pipelined into 3 stages.

for the operands A, B and $(\widetilde{M}+1) \operatorname{div} 2^{8(3+1)}$. A small register for holding the last 3 quotient digit (q_{i-1}, q_{i-2} and q_{i-3}) is also depicted. This register is used in the last statement of Algorithm 4. All intermediate results are redundant represented and an 4–2 adder is used for adding two redundant represented integers. As described in [10], a multiple of A or $(\widetilde{M}+1) \operatorname{div} 2^{8(3+1)}$ can be calculated by feeding the binary representation through a multiplexor network. Then the multiple is represented as the sum of a number of integers. In this case, where the radix is 2^8, the number of integers that represents a multiple is four. These four integers can then be compressed into two integers by using an 4–2 adder. Hence, a multiple is redundant represented as two integers. In the figure, the upper pairs of multiplexor networks and 4–2 adders are used for calculating the multiples of A and of $(\widetilde{M}+1) \operatorname{div} 2^{8(3+1)}$. After this calculation, the multiples are added by a third 4–2 adder giving $T_{i-3} = b_i A + q_{i-3}(\widetilde{M} + 1)$ div $2^{8(3+1)}$. Finally, T_{i-3} is added to S_i div 2^8 by the 4–2 adder shown in the lowest part of the figure. All of the multiplexor networks and 4–2 adders are latched. This implements the pipeline.

The cycle time of Algorithm 4 (the time for computing a single iteration of the loop) is determined by the critical path, i.e. the circuitry with the longest delay between two latches. Since the delay of the multiplexor network is approximately the delay of a single 4–1 multiplexor plus the set-up delay of a latch, the cycle time of Algorithm 4 is seen to be equal to the (longer) delay of an 4–2 adder. In modern CMOS technologies the delay of the 4–2 adders (including the latch set-up delay) is, by a conservative estimate, less than 5 ns. As an example, we will perform modular multiplication with 512 bit operands. If n is set to 69 the stimulus conditions of Algorithm 4 are fulfilled. Then the number of cycles in the loop is $69 + 3 + 1 = 73$. Since the first multiple of A, $b_0 A$, is delayed by 3 stages in the pipeline, the first cycle of the loop can begin after 3 clock periods. So, from the input are available to the result is present 76 clock periods are elapsed. With a 5 ns clock period this is 380 ns. Note that the result is redundant represented and that A and B, in this example, are assumed to be non–redundant represented. This means that the result has to be converted to non–redundant representation before it can be used as input for a new multiplication. The conversion must be performed by a carry completion adder. According to Algorithm 4 the result can be up to $8 \cdot 69 - 1 = 551$ bit wide. The fastest carry completion adders for these very large operands have a delay proportional to $\log_2 551$ but they are quit large in comparison with a carry ripple adder. In [13] a carry ripple adder with an asynchronous carry completion detection circuit is proposed. It is utilized that the average carry propagation length is the logarithm of the operand bit length, i.e. $\log_2 551 < 10$ for our example. If we estimate the average time for a conversion from redundant to non–redundant representation to be 10 clock periods a total of 83 clock periods, or 415 ns, is used for the multiplication.

In the above estimate for the computing time it turns out that about 15% of the time is used for conversion into non-redundant representation. As described in [10] and in [14] it is also possible to perform multiplications where the inputs are in redundant representation. Regarding the multiplier B, this does not imply serious trouble: The multiplier is scanned digit by digit from the least significant end, so a conversion into non-redundant representation may be done on-the-fly. The required circuitry for a register capable of holding a redundant represented operand will be about double the circuitry for holding a non-redundant represented operand. Regarding the multiplicand A, the penalty for using a redundant representation is larger: The required circuitry for computation of the multiples $b_i A$ will expand and the depth of the adder-tree will increase. This means that the delay parameter d must be increased. So, it is possible to obtain a further improvement of the computation time at the cost of additional circuitry.

In the computation of modular exponentials, also with 512 bit operands, the average number of required multiplications is 768 for a sequential algorithm. This can be done in $768 \cdot 415\text{ns} \approx 319\mu s$, which corresponds to a throughput of more than 1.6 Mbit/s. If a parallel algorithm is applied, see [10], the computing time is equal to the time for performing 512 multiplications, 212 μs, and a throughput of more than 2.4 Mbit/s is achieved. This is four times faster than the fastest known implementation [13]. Furthermore, if the modulus is a composite, and the prime factorization is known, it possible to speed up the computation by using the Chinese Remainder Theorem. This technique is applied in [13] to improve the time by about a factor of four.

The hardware consumption for the example architecture in Figure 1 is two multiplexor networks, four latched 4–2 adders and registers for the input operands. Each multiplexor network consists of four rows of latched 4–1 multiplexors and each 4–2 adder consists of two rows of fulladders where one of these is latched. Compared to the exponentiation processor in [9] this is an increase in circuitry of two rows of latched fulladders, two rows of latched 4–1 multiplexors plus the cost for modifying six rows of 4–1 multiplexors into latched 4–1 multiplexors. In [9] a large quotient determination unit and a carry completion adder are also included. There is no longer a need for the quotient determination unit. The circuitry cost of the primitive hardware components used in the exponentiation processor is analyzed in [8]. The transistor count for the complete exponentiation processor is 304,000. We estimate that the architecture in Figure 4, capable of multiplying 512 bit operands, can be implemented by not more than 300,000 transistors.

The above example architecture illustrates the potential of Algorithm 4. We could achieve even higher speeds by choosing higher radices and adding more circuitry to the architecture. A radix 2^{16} version could be implemented by adding a level in the trees for calculating multiples. This would increase the number of pipeline stages by one, increase the number of latched 4–2 adders by four and double the circuitry for the multiplexor networks. The number of clock periods for a 512 bit operand multiplication would then be $4 + 38 + 5 = 47$ for producing a redundant represented result and 57 for producing a non–redundant represented result. This is about 31% reduction of the total computing time for the radix 2^8 version. A 512 bit exponentiation would have a throughput of 2.3 Mbit/s if the sequential exponentiation algorithm is applied or 3.5 Mbit/s if the parallel algorithm is applied.

4 Summary

In this paper we have rewritten a high-radix version of Montgomery's modular multiplication algorithm in order to obtain a trivial quotient determination, where the multiplication and addition operation is avoided by a simple transformation of modulus. The result is a quotient determination that is reduced to a trivial extraction of the least significant digit of the partial modular product, $S_i \bmod 2^k$. By applying a pipeline technique we have enabled overlapping computations. This implies that the critical computation path can be reduced to a shift-and-add operation that is efficiently implemented by a constant time adder. We have achieved a modular multiplication algorithm, where *the critical hardware path is independent of the choice of radix.* For a fixed high radix, the cost of the proposed algorithms, over Montgomery's algorithm, is a few extra iteration cycles, additional pre-processing for the transformation of modulus and a wider range of the final result. When several modular multiplications have to be performed on intermediate results, this cost is more than compensated by the faster time for an iteration and the possibility to reduce the number of iterations through the choice of radix. By pipelining the formations of the multiples, the only limitation to the choice of radix is the size of the circuitry, not the cycle time.

References

[1] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[2] Stephen R. Dussé and Burton S. Kaliski Jr. A Cryptograhpic Library for the Motorola DSP56000. In Ivan B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, Berlin, 1991.

[3] Stephen E. Eldridge. A Faster Modular Multiplication Algorithm. *International Journal of Computer Mathematics*, 40:63–68, 1991.

[4] Stephen E. Eldridge and Colin D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers*, C-42(6):693–699, June 1993.

[5] Peter Kornerup. High-Radix Modular Multiplication for Cryptosystems. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic*, pages 277–283. IEEE Computer Society Press, Los Alamitos, California, 1993.

[6] Peter Kornerup. A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms. *IEEE Transactions on Computers*, C-43(8):892–898, August 1994.

[7] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[8] Holger Orup. Area Reduction for Bit–Sliced Layouts using a Commercial Development System. This article is not published. It is available from the author.

[9] Holger Orup. A 100Kbit/s Single Chip Modular Exponentiation Processor. In *HOT Chips VI, Symposium Record*, pages 53–59. Stanford University, 1994. Only the slides from the presentation at HOT Chips VI are printed in the Symposium Record. An abstract is available from the author.

[10] Holger Orup and Peter Kornerup. A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 51–56. IEEE Computer Society Press, Los Alamitos, California, 1991.

[11] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an Efficient RSA Hardware Implementation. In Ivan B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 245–252. Springer-Verlag, Berlin, 1991.

[12] R. L. Rivest, A. Shamir, and L.Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[13] M. Shand and J. Vuillemin. Fast Implementations of RSA Cryptography. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic*, pages 252–259. IEEE Computer Society Press, Los Alamitos, California, 1993.

[14] Naofumi Takagi. A Radix-4 Modular Multiplication Hardware Algorithm Efficient for Iterative Modular Multiplications. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 35–42. IEEE Computer Society Press, Los Alamitos, California, 1991. This article also appears as [15].

[15] Naofumi Takagi. A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation. *IEEE Transactions on Computers*, C-41(8):949–956, August 1992.

[16] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, February 1964.

[17] Colin D. Walter. Systolic Modular Multiplication. *IEEE Transactions on Computers*, C-42(3):376–378, March 1993.