Efficient and Formally Proven Reduction of Large Integers by Small Moduli

LUC RUTTEN
Delft Development Laboratory, Tivoli, Software Group, IBM
and
MARKO VAN EEKELEN
Institute for Computing and Information Sciences, Radboud University Nijmegen
School of Computer Science, Open University of the Netherlands

On w-bit processors which are much faster at multiplying two w-bit integers than at dividing $2w$-bit integers by w-bit integers, reductions of large integers by moduli M smaller than 2^{w-1} are often implemented sub-optimally, leading applications to take excessive processing time.

We present a modular reduction algorithm implementing division by a modulus through multiplication by a reciprocal of that modulus, a well-known method for moduli larger than 2^{w-1}. We show that application of this method to smaller moduli makes it possible to express certain modular sums and differences without having to compensate for word overflows.

By embedding the algorithm in a loop and applying a few transformations to the loop, we obtain an algorithm for reduction of large integers by moduli up to 2^{w-1}. Implementations of this algorithm can run considerably faster than implementations of similar algorithms that allow for moduli up to 2^w. This is substantiated by measurements on processors with relatively fast multiplication instructions.

It is notoriously hard to specify efficient mathematical algorithms on the level of abstract machine instructions in an error-free manner. In order to eliminate the chance of errors as much as possible, we have created formal correctness proofs of our algorithms, checked by a mechanized proof assistant.

Categories and Subject Descriptors: G.4 [**MATHEMATICAL SOFTWARE**]: Algorithm design and analysis; Efficiency; Verification

General Terms: Algorithms, Performance, Verification

Additional Key Words and Phrases: Computer Arithmetic, Machine Checked Proofs, Modular Reduction, Optimization

1. INTRODUCTION

There are many applications of *modular reductions*, which are computations of *residues* $x \bmod M$ ("x modulo M") where $x \in \mathbb{Z}$ and $M \in \mathbb{Z}^+$. The positive integer M is called the *modulus.* The computation of $x \bmod M$ is called the *reduction of x by M*. The residue $x \bmod M$ is sometimes called the *remainder (after division of x by M).*

Often, applications of modular reductions employ moduli which may be (much) larger than 2^w, where $w \in \mathbb{Z}^+$ is the *word size* of a computer processor. Some applications though only employ moduli $M \leq 2^{w-1}$, which we call *small moduli.* Such applications include small modulus specializations of large integer reduction [GMP www], modular exponentiation, and modular multiplication, for example employed in residue arithmetic and in Garner's algorithm [Knuth 1998]. Implementations of such small-modulus specializations benefit from efficient reduction by small moduli.

Unsigned integer arithmetic sets and operations are presented in section 2. In section 3 we present an efficient small modulus reduction algorithm expressed in terms of these sets and operations. The algorithm – ModRed – can be employed to compute residues $x \bmod M$ for $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$ and $x \in \mathbb{Z}_{2^{\lceil \lg M \rceil + w}}$.

In section 4 we show that ModRed can be embedded in a loop in order to reduce large integers. Some transformations are applied to the loop to obtain a more efficient algorithm – MultiRed – which can be employed to compute residues $x \bmod M$ for $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$ and $x \in \mathbb{Z} \mid 0 \leq x$.

Algorithms ModRed and MultiRed contain low-word multiplications and high-word multiplications but no divisions except by powers of 2. This indicates that they can be relatively efficiently implemented on processors where divison of $2w$-bit words by w-bit words is expensive relative to low-word and high-word multiplications. On a side note, when M is a power of 2, $x \bmod M$ can be even more efficiently computed by evaluating $\sum_{i=0}^{w-1} 2^i(\lfloor \frac{x \bmod 2^w}{2^i} \rfloor \bmod 2)(\lfloor \frac{M-1}{2^i} \rfloor \bmod 2)$ using a single *bitwise-and* instruction.

While the number and kind of operations of an algorithm may give an impression of its efficiency, many other factors affect real life performance. In section 5, we therefore present performance measurements of ModRed and MultiRed implementations. The performance of these implementations is compared with the performance of a similar implementation from [GMP www], whose method for reducing an unsigned double word by an unsigned word is based on an algorithm proposed in [Granlund and Montgomery 1994].

It is notoriously hard to specify mathematical algorithms like ModRed and MultiRed in an error-free manner. In order to obtain a high degree of certainty that ModRed and MultiRed are free of errors, we have created formal correctness proofs of the algorithms, checked by a mechanized proof assistant. This is discussed in section 6.

Related and future work are discussed in sections 7 and 8 while conclusions are presented in section 9.

2. PRELIMINARIES

The algorithms proposed in section 3 and section 4 are expressed in terms of the standard unsigned integer arithmetic sets and operations (on the level of abstract machine instructions) shown in the current section, which may be skimmed by readers familiar with these concepts.

In this text, $\mathbb{Z}^+$ is defined as $\{i \in \mathbb{Z} \mid 0 < i\}$. For each $n \in \mathbb{Z}$, $\mathbb{Z}_n$ is defined as $\{i \in \mathbb{Z} \mid 0 \leq i < n\}$. For each $x \in \mathbb{Z}$ and $M \in \mathbb{Z}^+$, $\lfloor \frac{x}{M} \rfloor$ is defined as the unique integer $q \in \mathbb{Z}$ and $x \bmod M$ is defined as the unique integer $r \in \mathbb{Z}_M$ for which it holds: $x = qM + r$. Expressions $\lfloor \frac{x}{M} \rfloor$ are sometimes written like $\lfloor x/M \rfloor$. We will write mod as a left associative operator with the same precedence as the multiplication operator. For each $x \in \mathbb{Z}^+$, the *binary logarithm of x rounded up to the nearest integer*, $\lceil \lg x \rceil$, is defined as the (unique) nonnegative integer i such that $\lfloor \frac{2^i}{2} \rfloor < x \leq 2^i$. For each $x \in \mathbb{Z}^+$, $\lfloor \lg x \rfloor$ is similarly defined as the (unique) nonnegative integer i such that $2^i \leq x < 2^{i+1}$.

For each word size $w \in \mathbb{Z}^+$, each $x \in \mathbb{Z}_{2^w}$ is called a *w-bit word* in *unsigned w-bit integer arithmetic*. Such a word x may be identified with a tuple of w bits:

$(\lfloor \frac{x}{2^i} \rfloor \bmod 2)_{i=0}^{w-1} \in \mathbb{Z}_2^w$. A *w-bit processor* provides instructions for direct manipulation of representations of such tuples. For $x, y \in \mathbb{Z}_{2^w}$ and $i \in \mathbb{Z}_w$, instructions for computation of $(x + y) \bmod 2^w$ (addition), $(x - y) \bmod 2^w$ (subtraction), $xy \bmod 2^w$ (low-word multiplication), $\lfloor \frac{xy}{2^w} \rfloor$ (high-word multiplication), $2^i x \bmod 2^w$ (left shift), and $\lfloor \frac{x}{2^i} \rfloor$ (right shift) are commonly provided. Each of these expressions usually takes just a single instruction, while some processors are able to compute $(xy \bmod 2^w, \lfloor \frac{xy}{2^w} \rfloor)$ with a single instruction. Conditional operations are also commonly provided, usually taking a comparison instruction and a conditional move or branch instruction. An additional unconditional branch instruction may also be involved.

The greatest integer that can be represented in unsigned w-bit integer arithmetic is $2^w - 1$. Greater integers are said to *overflow* a word, and $\lfloor \frac{x}{2^w} \rfloor$ is called the *nonzero word overflow* of such an integer x.

Tuples of words $(\lfloor \frac{x}{2^{wi}} \rfloor \bmod 2^w)_{i=0}^{k-1} \in \mathbb{Z}_{2^w}^k$ can represent *multi-word integers* $x \in 2^{wk}$, also called *large integers* because they may be much larger than $2^w - 1$. For example, each *double word* $x \in \mathbb{Z}_{2^{2w}}$ can be represented with a *low word* $x \bmod 2^w$ and a *high word* $\lfloor \frac{x}{2^w} \rfloor$. Accessing the words of a multi-word integer and composing multi-word integers from words may require the use of *load* and *store* instructions.

3. A NEW ALGORITHM FOR SMALL MODULUS REDUCTION

In this section, we present a new algorithm which can be employed to compute residues $x \bmod M$ for $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$ and $x \in \mathbb{Z}_{2^{\lceil \lg M \rceil + w}}$. The algorithm – ModRed – is defined on the level of an abstract machine using instructions introduced in section 2. Some applications of ModRed are mentioned in section 1.

The main idea behind the ModRed algorithm is to compute $x \bmod M$ as $(\lfloor \frac{x}{a} \rfloor a \bmod M + x \bmod a \bmod M) \bmod M$ where $a = 2^{\lceil \lg M \rceil}$, and to approximate $\lfloor \frac{x}{a} \rfloor a \bmod M$ by $d = \lfloor \frac{x}{a} \rfloor a - qM$ where $q = \lfloor \lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{M} \rfloor / b \rfloor$ and $b = 2^w$.

The value of q is only 0 or 1 smaller than $\lfloor \lfloor \frac{x}{a} \rfloor a / M \rfloor$, as

$$\forall y, a, b \in \mathbb{Z}^+ \ \forall x \in \mathbb{Z}_{ab+a} : \lfloor \frac{\lfloor \frac{x}{a} \rfloor a}{y} \rfloor - 1 \leq \lfloor \frac{\lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{y} \rfloor}{b} \rfloor \leq \lfloor \frac{\lfloor \frac{x}{a} \rfloor a}{y} \rfloor \quad (1)$$

Therefore, d can be equal to $\lfloor \frac{x}{a} \rfloor a \bmod M$ or $\lfloor \frac{x}{a} \rfloor a \bmod M + M$. When $M \leq 2^{w-1}$, both values lie in $\mathbb{Z}_{2^w}$ so word overflows do not occur: $\lfloor \frac{d}{2^w} \rfloor = 0$. The value of $\lfloor \frac{x}{a} \rfloor a \bmod M$ is therefore easy to derive from d with a conditional subtraction. To the resulting value $\lfloor \frac{x}{a} \rfloor a \bmod M$, $x \bmod a \bmod M$ can be added modulo M to obtain $x \bmod M$, again using only w-bit integer arithmetic and without word overflows. If M would have been greater than 2^{w-1}, word overflows would have had to be handled with additional operations. Additional operations are also needed if $x \bmod M$ is approximated more directly, using equation 65 of section 7. That can be seen in the `udiv_qrnnd_preinv1` macro of [GMP www].

By choosing $b = 2^w$, the low word of $\lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{M} \rfloor$ is not needed for computation of q. When $\lceil \lg M \rceil < w - 1$, then on processors with separate low-word and high-word multiplication instructions, this saves a low-word multiplication w.r.t. Barrett's algorithm [Barrett 1987], which is reviewed in section 7. Furthermore, when $b = 2^w$, the high word of $\lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{M} \rfloor$ can on most processors be obtained without carrying out any operation after the multiplication because the high word is already present in a machine register after a high-word (or double-word) multiplication. This contrasts with the left shift, right shift, and addition operations which are usually employed to divide a double word by $2^{\lceil \lg M \rceil + 1}$ on a w-bit processor when $\lceil \lg M \rceil < w - 1$.

These advantages of choosing $b = 2^w$ are shared with the algorithm proposed in section 8 of [Granlund and Montgomery 1994]. The low values of M that ModRed applies to – $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$ – give rise to an additional advantage relative to this algorithm, which applies to $M \in \mathbb{Z} \mid 1 \leq M < 2^w$: because ModRed does not apply to moduli $M \in \mathbb{Z} \mid 2^{w-1} < M < 2^w$, it needs fewer operations for taking care of word overflows.

Algorithm 1 ModRed

Inputs: $M \in \mathbb{Z}_{2^w}$, $p, t \in \mathbb{Z}_w$, $M', v, u \in \mathbb{Z}_{2^w}$
Output: $\mathrm{ModRed}(M, p, t, M', v, u) = r'''$
where $s, s', h, h', q, q', y, d, r, r', r'', r''' \in \mathbb{Z}_{2^w}$ are defined with

$$s = \lfloor \frac{u}{2^p} \rfloor \tag{2}$$

$$s' = 2^p s \bmod 2^w \tag{3}$$

$$h = 2^t v \bmod 2^w \tag{4}$$

$$h' = (h + s) \bmod 2^w \tag{5}$$

$$q = \lfloor \frac{h' M'}{2^w} \rfloor \tag{6}$$

$$q' = (q + h') \bmod 2^w \tag{7}$$

$$y = q' M \bmod 2^w \tag{8}$$

$$d = (s' - y) \bmod 2^w \tag{9}$$

$$r = (u - y) \bmod 2^w \tag{10}$$

$$r' = \begin{cases} r & \text{if } d < M \\ (r - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{11}$$

$$r'' = \begin{cases} r' & \text{if } r' < M \\ (r' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{12}$$

$$r''' = \begin{cases} r'' & \text{if } r'' < M \\ (r'' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{13}$$

Let $C \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\} \to \mathbb{Z}_w \times \mathbb{Z}_w \times \mathbb{Z}_{2^w}$ be defined with

$$C(M) = (\lceil \lg M \rceil, w - \lceil \lg M \rceil - \lfloor \frac{1}{M} \rfloor, \lfloor \frac{2^{\lceil \lg M \rceil + w}}{M} \rfloor - 2^w) \tag{14}$$

where $M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\}$. Then, splitting the integer to be reduced (x) in its high and low word as explained in section 2, the correctness of algorithm ModRed is expressed by the following theorem:

Theorem 3.1 Correctness of ModRed.

$$\forall M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\} \; \forall x \in \mathbb{Z}_{2^{\lceil \lg M \rceil + w}} : \quad \text{ModRed}(M, C(M), \lfloor \frac{x}{2^w} \rfloor, x \bmod 2^w) = x \bmod M \quad (15)$$

Efficient computation of the $\lceil \lg M \rceil$ component of $C(M)$ may take place using one of the `count_leading_zeros` implementations of [GMP www], subtracting the number of leading zeros of $M - 1$ from w. The value of the $\lfloor \frac{2^{\lceil \lg M \rceil + w}}{M} \rfloor - 2^w$ component of $C(M)$ may be computed using e.g. the division of nonnegative integers algorithm or the high-precision reciprocal algorithm, both described in [Knuth 1998], or the `udiv_qrnnd` implementation of [GMP www].

Computation of the components of $C(M)$ can be time consuming in comparison with evaluating an application of ModRed. It is therefore recommended to use ModRed only if ModRed(M, $C(M)$, $\lfloor \frac{x}{2^w} \rfloor$, $x \bmod 2^w$) for a single value of M is to be computed for many values of x. A suitable minimum number of such values can be determined with performance measurements. When a compiler uses ModRed to generate better code for $x \bmod M$ where M is known at compile-time, the minimum number may be equal to 1 as the compiler can precompute $C(M)$.

The outcome of one ModRed application may be passed to the next application. This can be seen for example in the algorithm proposed in section 4.

4. REDUCTION OF LARGE INTEGERS BY A SMALL MODULUS

In this section, it will be seen that algorithm ModRed can be placed in a loop to obtain a multi-word reduction algorithm. We will demonstrate how that algorithm can be transformed to a more efficient multi-word reduction algorithm, called MultiRed.

Rationale

Algorithm ModRed can be employed in a loop of k iterations in order to reduce an integer in $\mathbb{Z}_{2^{wk}}$ by a modulus $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$. This is illustrated by the reduction of a 3-word integer $x = x_2 2^{w \cdot 2} + x_1 2^w + x_0$ where $x_2, x_1, x_0 \in \mathbb{Z}_{2^w}$: $x \bmod M = (((0 \cdot 2^w + x_2)2^w + x_1)2^w + x_0) \bmod M = (((((0 \cdot 2^w + x_2) \bmod M)2^w + x_1) \bmod M)2^w + x_0) \bmod M = \text{ModRed}(M, C(M), \text{ModRed}(M, C(M), \text{ModRed}(M, C(M), 0, x_2), x_1), x_0)$. In the first loop iteration, the innermost ModRed application is evaluated, while in the the last loop iteration, the outermost ModRed application is evaluated. Performance measurements of ModRed in a multi-word reduction program (see section 5) show it to be slower than or about as fast as the `udiv_qrnnd_preinv1` macro of [GMP www]. By applying some transformations to the ModRed loop, it is possible obtain a more efficient multi-word reduction program.

Transformations

In the descriptions of the transformations, the loop iterations employed to reduce $x \in \mathbb{Z}_{2^{wn}}$ will be numbered $n-1$, $n-2$, ..., 0. For each $i \in \mathbb{Z}_n$, x_i will be defined as $\lfloor \frac{x}{2^{wi}} \rfloor \bmod 2^w$, so $x = \sum_{i=0}^{n} 2^{wi} x_i$. The ModRed variables of each iteration i will get suffix i. We define r_n''' as 0. The inputs of iteration i are defined as $v_i = r_{i+1}'''$

and $u_i = x_i$. The value of r'''_0 is computed in iteration 0, i.e. the last iteration. The value of r'''_0 equals $x \bmod M$.

At the end of each iteration i, r'''_i is derived from r''_i using a conditional subtraction. The r'''_i value then enters the next loop iteration as v_{i-1} (if $i > 0$). Because $r''_i = r'''_i \vee r''_i = r'''_i + M$ and because $r''_i < 2^{\lceil \lg M \rceil}$, we can let v_{i-1} be defined as r''_i instead of r'''_i. That is because for all $u \in \mathbb{Z}_{2^w}$ and for all $v \in \mathbb{Z}_{2^{\lceil \lg M \rceil}}$ such that $v + M < 2^{\lceil \lg M \rceil}$, it holds $\text{ModRed}(M, C(M), v + M, u) = (2^w(v + M) + u) \bmod M$ $= (2^w v + u) \bmod M = \text{ModRed}(M, C(M), v, u)$. By defining v_{i-1} as r''_i instead of r'''_i, one conditional subtraction is saved per loop iteration (except for the last iteration) as was already suggested at the end of section 3. Therefore, let's define r''_k as 0 and let's define v_i as r''_{i+1}.

After transforming the loop by replacing $v_i = r'''_{i+1}$ with $v_i = r''_{i+1}$, each loop iteration ends with two conditional subtractions. Also, each loop starts with a right shift of x_i, which is preceded by loading x_i from memory. The instruction level parallelism is increased by moving the two conditional subtractions at the end of iteration i to the beginning of the next iteration, $i-1$. After this transformation, the conditional subtractions can in principle be computed in parallel with the right shift (and subsequent left shift).

Now h'_i depends on h_i and s_i, where h_i depends on r''_{i+1}, which depends on r'_{i+1}, which depends on r_{i+1}. On the other hand, s_i just depends on x_i. Therefore, it is likely that s_i will have been computed well before h_i. The conditional subtraction from r'_{i+1} is then replaced by a conditional subtraction from s_i. After this transformation, the two dependency chains of the (new) components of h'_i have equal lengths, increasing instruction level parallelism.

In the second conditional subtraction of the resulting algorithm there is a comparison between r' and M. This can be optimized for some processors which take less time to evaluate r'' if $r' < c$ than if $r' \geq c$, especially for values of M close to $2^{\lceil \lg M \rceil - 1} + 1$. The transformation is to replace M by $c = 2^{\lceil \lg M \rceil}$. This does not cause overflows in the rest of the computation. So, no extra corrections are needed.

After performing the four transformations some extra equations have to be added after the loop to make up for the moved conditional subtractions. This results in the algorithm MultiRed given below.

For expressing the correctness of the algorithm MultiRed we need another auxiliary definition. $C' \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\} \rightarrow \mathbb{Z}_w \times \mathbb{Z}_w \times \mathbb{Z}_{2^w} \times \mathbb{Z}_{2^w} \times \mathbb{Z}_{2^w}$ is defined with

$$C'(M) = (C(M), 2^{\lceil \lg M \rceil}, 2^{w - \lceil \lg M \rceil - \lfloor \frac{1}{M} \rfloor} M \bmod 2^w) \qquad (32)$$

where $M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\}$. Using this definition we formulate the MultiRed correctness theorem:

Theorem 4.1 Correctness of MultiRed.

$$\forall M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\}\ \forall n \in \mathbb{N}\ \forall x \in \mathbb{Z}_{2^{nw}} :$$
$$\text{MultiRed}(M, C'(M), n, x) = x \bmod M \qquad (33)$$

Algorithm 2 MultiRed

Inputs: $M \in \mathbb{Z}_{2^w}$, $p, t \in \mathbb{Z}_w$, $M', c, M'', \in \mathbb{Z}_{2^w}$, $n \in \mathbb{N}$, $x \in \mathbb{Z}_{2^{nw}}$
Output: $\text{MultiRed}(M, p, t, M', c, M'', n, x) = r_0'''$
where $d_n, r_n \in \mathbb{Z}_{2^w}$ are defined with

$$d_n = 0 \tag{16}$$

$$r_n = 0 \tag{17}$$

and where for $i = n-1, n-2, ..., 0$, x_i is defined as $\lfloor \frac{x}{2^{wi}} \rfloor \bmod 2^w$ and $r_i', s_i, s_i', s_i'', h_i, h_i', q_i, q_i', y_i, d_i, r_i \in \mathbb{Z}_{2^w}$ are defined with

$$r_i' = \begin{cases} r_{i+1} & \text{if } d_{i+1} < M \\ (r_{i+1} - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{18}$$

$$s_i = \lfloor \frac{x_i}{2^p} \rfloor \tag{19}$$

$$s_i' = 2^p s_i \bmod 2^w \tag{20}$$

$$s_i'' = \begin{cases} s_i & \text{if } r_i' < c \\ (s_i - M'') \bmod 2^w & \text{otherwise} \end{cases} \tag{21}$$

$$h_i = 2^t r_i' \bmod 2^w \tag{22}$$

$$h_i' = (h_i + s_i'') \bmod 2^w \tag{23}$$

$$q_i = \lfloor \frac{h_i' M'}{2^w} \rfloor \tag{24}$$

$$q_i' = (q_i + h_i') \bmod 2^w \tag{25}$$

$$y_i = q_i' M \bmod 2^w \tag{26}$$

$$d_i = (s_i' - y_i) \bmod 2^w \tag{27}$$

$$r_i = (x_i - y_i) \bmod 2^w \tag{28}$$

and where $r_0', r_0'', r_0''' \in \mathbb{Z}_{2^w}$ are defined with

$$r_0' = \begin{cases} r_0 & \text{if } d_0 < M \\ (r_0 - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{29}$$

$$r_0'' = \begin{cases} r_0' & \text{if } r_0' < c \\ (r_0' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{30}$$

$$r_0''' = \begin{cases} r_0'' & \text{if } r_0'' < M \\ (r_0'' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{31}$$

4.1 Parallellization

It is possible to compute $x \in \mathbb{Z}_{2^{2kw}} \bmod M$ using the formula $x \bmod M = ((\lfloor \frac{x}{2^{kw}} \rfloor \bmod M) \cdot (2^{kw} \bmod M) + (x \bmod 2^{kw}) \bmod M) \bmod M$. The components $\lfloor \frac{x}{2^{kw}} \rfloor$ and $x \bmod 2^{kw}$ of x can be reduced (e.g. with MultiRed) in parallel, independently of each other. The value of $2^{kw} \bmod M$ can be determined with fewer than $2\lceil \lg k \rceil$ low-word multiplications, $2\lceil \lg k \rceil$ high-word multiplications, and $2\lceil \lg k \rceil$ reductions of integers in $\mathbb{Z}_{2^{\lceil \lg M \rceil + w}}$ by M e.g. employing ModRed. When k is large, computation of $2^{kw} \bmod M$ therefore only takes a small fraction of the time to reduce the components of x. Reducing the components in parallel and combining the residues may then take only a bit more than half the (wall-clock) time taken to reduce the components after one another.

5. PERFORMANCE

The ModRed and MultiRed algorithms introduced in section 3 and section 4 have been designed with efficiency in mind. To gain some insight in the algorithms' efficiency, we will compare run-times of implementations of the algorithms, each allowing for moduli up to 2^{w-1}, with run-times of the `mpn_mod_1` function of [GMP www], which uses the `udiv_qrnnd_preinv1` macro allowing for moduli between 2^{w-1} and 2^w. A variation of `mpn_mod_1` is also measured.

We benchmarked the reduction of a multi-word integer by several moduli smaller than 2^{w-1}. In particular we took the multi-word integer $\sum_{i=0}^{\lfloor \frac{\lfloor \sqrt{s} \rfloor w}{16} \rfloor - 1} ((16807^i \bmod (2^{31}-1)) \bmod 2^{16}) 2^{16i}$ where s denotes the processor speed in Hz, and where the pseudo-random number generator $n \mapsto 16807n \bmod (2^{31}-1)$ is attributed to [Lewis et al. 1969]. We took moduli $2^{w-1} - 1 - i \lfloor \frac{2^{w-1}}{\lfloor \sqrt{s} \rfloor} \rfloor$ for all $i \in \mathbb{Z}_{\lfloor \sqrt{s} \rfloor}$. Implementations of ModRed and MultiRed were compared with an implementation (denoted as mm1) directly based on the `mpn_mod_1` function from the GNU Multiple Precision Arithmetic Library [GMP www].

The `mpn_mod_1` function uses the `udiv_qrnnd_preinv1` macro which is based on an algorithm of [Granlund and Montgomery 1994]. It can be employed to reduce integers in $\mathbb{Z}_{2^{2w}}$ by moduli $M \in \mathbb{Z} \mid 2^{w-1} \leq M < 2^w$. The following equations reflect that part of the macro which computes $r' = (2^w v + u) \bmod M$. Equations to determine $\lfloor \frac{2^w v + u}{M} \rfloor$ are not included because this quotient is not needed for computing residues. In the pseudocode, M' denotes $\lfloor \frac{2^{2w}-1}{M} \rfloor - 2^w$. The integers Y, Z, and Z' are double words. When $\lfloor \frac{Z}{2^w} \rfloor = 0$, $r = Z \bmod 2^w$ so $\lfloor \frac{Z'}{2^w} \rfloor$ does not really have to be computed.

$$q = \lfloor \frac{vM'}{2^w} \rfloor \tag{a}$$

$$q' = (q + v) \bmod 2^w \tag{b}$$

$$Y = q'M \tag{c}$$

$$Z = (2^w v + u) - Y \tag{d}$$

$$Z' = \begin{cases} Z & \text{if } \lfloor \frac{Z}{2^w} \rfloor = 0 \\ Z - M & \text{otherwise} \end{cases} \tag{e}$$

$$r = \begin{cases} Z' \bmod 2^w & \text{if } \lfloor \frac{Z'}{2^w} \rfloor = 0 \\ Z' \bmod 2^w - M & \text{otherwise} \end{cases} \tag{f}$$

$$r' = \begin{cases} r & \text{if } r < M \\ r - M & \text{otherwise} \end{cases} \tag{g}$$

We also compared MultiRed with a variation mm1' of the mm1 implementation. We made this variation in order to view the effects of some transformations on mm1 that are analogous to the transformations on the ModRed loop to obtain MultiRed. Preliminary performance measurements indicated that MultiRed can be considerably more efficient than ModRed, leading to the expectation that mm1' is more efficient than mm1. The mm1 implementation is transformed to the mm1' implementation by moving the last three equations ((e), (f), and (g)) of the mm1 loop body to the front of the loop, and by putting some operations in front of the loop and after the loop to compensate for this.

Note that mm1 and mm1' implement reduction of multi-word integers by moduli smaller than 2^{w-1}, rather than by moduli between 2^{w-1} and 2^w. To be able to use the `udiv_qrnnd_preinv1` macro, a double word is reduced by $2^{w-\lceil \lg M \rceil}M$ rather than by M in each iteration of the mm1 an mm1' inner loops. The resulting residue in $\mathbb{Z}_{2^w}$ is reduced by M to obtain the final result. While this reduction may take a (costly) division operation, the number of loop iterations is rather large so the time spent by the division operation is rather small in comparison with the time spent by the loop iteration.

The benchmarks were carried out on two 64-bit processors: a 1.6 GHz AMD Turion* 64 X2 ML-50 [AMD www], and a 2 GHz IBM PowerPC† 970FX [IBM www]. 64-Bit processors are often employed in workstations and they are beginning to emerge in personal computers. Even to reduce a multi-word integer by a 32-bit word, it is profitable to use a 64-bit ModRed implementation on a 64-bit processor, as the number of 64-bit ModRed iterations needed to reduce a $64n$-bit multi-word integer is equal to n, while the number of 32-bit ModRed iterations needed to reduce the same $64n$-bit multi-word integer is equal to $2n$.

Each benchmark was compiled with a version of the GNU C compiler [GCC www], using option `-O3` for high optimization: gcc 4.1.2 with the Turion 64 X2 ML-50, and gcc 4.3.0 with the PowerPC 970FX. The benchmarks were carried out on quiet systems, with no other processor, memory, or disk intensive tasks at hand. The benchmarks were repeated in the course of several days and at different times of a day.

The following table lists (average) numbers of clock cycles spent by a single iteration of four implementations for reducing multi-word integers by moduli smaller than 2^{w-1}.

	Turion 64 X2	PowerPC 970FX
mm1	*24.5*	40.6
mm1'	25.0	*38.4*
ModRed	24.0	47.6
MultiRed	*18.0*	*36.2*
$\lfloor \frac{\text{bestmm1}\cdot 100}{\text{bestModRedfamily}} \rfloor/100$	1.36	1.06

The performance measurement of MultiRed shows only a small improvement (1.06) for the PowerPC 970FX processor. In our opinion this is caused by internal scheduling and instruction level parallelism differences between the considered processors. It suggests that a different order of the calculations might give better performance on the PowerPC. After several experiments we found a calculation sequence showing an improvement which is more alike the improvement found with the Turion. The calculation scheme does not only concern changing the order of the instructions. It also encompasses changes in the actual instructions employed and even in the number of instructions. For this reason it is justifiable to give the resulting variant of MultiRed a separate name – MultiRed′ – and to specify and

*AMD Turion is a trademark of Advanced Micro Devices, Inc.

†PowerPC is a trademark of International Business Machines Corporation

verify the algorithm separately. This variation of MultiRed is described in the next section.

5.1 A variation of MultiRed

The lower performance of MultiRed on the PowerPC 970FX suggests that the first conditional subtraction of its loop may be a bottleneck on some processors. It is possible to apply a few transformations to the MultiRed loop in order to obtain an algorithm which runs more efficiently on some of those processors. Surprisingly, this involves the introduction of yet another conditional subtraction.

In each loop iteration, we determine $f_i = x_i \bmod 2^{\lceil \lg M \rceil}$ (e.g. with $(x_i - s'_i) \bmod 2^w$).

At the end of each loop iteration, we determine g_i, which equals r_i if $f_i < M$, and $(r_i - M) \bmod 2^w$ otherwise. Instead of r_i, we pass g_i to the next loop iteration (if any).

Finally, we replace the comparison $d_{i+1} < M$ with $d_{i+1} < c$, which may lead to a subtraction being carried out less often on average when $c = 2^{\lceil \lg M \rceil}$, like described at end the end of section 3.

The correctness theorem of MultiRed′ much resembles the MultiRed correctness theorem, once more using the function C' defined with equation 32.

Theorem 5.1 Correctness of MultiRed'.

$$\forall M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\} \; \forall n \in \mathbb{N} \; \forall x \in \mathbb{Z}_{2^{nw}} : \quad \text{MultiRed}'(M, C'(M), n, x) = x \bmod M \tag{52}$$

Below, we give the performance measurements including the new algorithm MultiRed′.

	Turion 64 X2	PowerPC 970FX
mm1	*24.5*	40.6
mm1'	25.0	*38.4*
ModRed	24.0	47.6
MultiRed	*18.0*	36.2
MultiRed′	21.0	*30.8*
$\lfloor \frac{\text{bestmm1} \cdot 100}{\text{bestModRedfamily}} \rfloor / 100$	1.36	1.24

The table shows that for the PowerPC MultiRed′ indeed improves over MultiRed as expected. For the Turion however, its performance is slightly worse than that of MultiRed.

5.2 Evaluation of performance measurements

The benchmark results indicate that implementation of MultiRed and MultiRed′ can be profitable on processors of different kinds. It can also be seen that mm1' is not much faster than mm1. The PowerPC 970FX has no instructions to divide 128-bit integers by 64-bit integers. The Turion 64 X2 has a 128-bit by 64-bit unsigned integer division instruction but its instruction for multiplying two unsigned 64-bit integers to obtain a 128-bit result is much faster: the MultiRed implementation ran more than four times as fast as an implementation using the 128-bit integer by 64-bit integer division instruction. For these reasons, benchmarks using double-word

Algorithm 3 MultiRed′

Inputs: $M \in \mathbb{Z}_{2^w}$, $p, t \in \mathbb{Z}_w$, $M', c, M'', \in \mathbb{Z}_{2^w}$, $n \in \mathbb{N}$, $x \in \mathbb{Z}_{2^{nw}}$
Output: $\mathrm{MultiRed}'(M, p, t, M', c, M'', n, x) = r_0'''$
where $d_n, g_n \in \mathbb{Z}_{2^w}$ are defined with

$$d_n = 0 \tag{34}$$

$$g_n = 0 \tag{35}$$

and where for $i = n-1, n-2, ..., 0$, x_i is defined as $\lfloor \frac{x}{2^{wi}} \rfloor \bmod 2^w$ and $r_i', s_i, s_i', f_i, s_i'', h_i, h_i', q_i, q_i', y_i, d_i, r_i, g_i \in \mathbb{Z}_{2^w}$ are defined with

$$r_i' = \begin{cases} g_{i+1} & \text{if } d_{i+1} < c \\ (g_{i+1} - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{36}$$

$$s_i = \lfloor \frac{x_i}{2^p} \rfloor \tag{37}$$

$$s_i' = 2^p s_i \bmod 2^w \tag{38}$$

$$f_i = (x_i - s_i') \bmod 2^w \tag{39}$$

$$s_i'' = \begin{cases} s_i & \text{if } r_i' < c \\ (s_i - M'') \bmod 2^w & \text{otherwise} \end{cases} \tag{40}$$

$$h_i = 2^t r_i' \bmod 2^w \tag{41}$$

$$h_i' = (h_i + s_i'') \bmod 2^w \tag{42}$$

$$q_i = \lfloor \frac{h_i' M'}{2^w} \rfloor \tag{43}$$

$$q_i' = (q_i + h_i') \bmod 2^w \tag{44}$$

$$y_i = q_i' M \bmod 2^w \tag{45}$$

$$d_i = (s_i' - y_i) \bmod 2^w \tag{46}$$

$$r_i = (x_i - y_i) \bmod 2^w \tag{47}$$

$$g_i = \begin{cases} r_i & \text{if } f_i < M \\ (r_i - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{48}$$

and where $r_0', r_0'', r_0''' \in \mathbb{Z}_{2^w}$ are defined with

$$r_0' = \begin{cases} g_0 & \text{if } d_0 < M \\ (g_0 - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{49}$$

$$r_0'' = \begin{cases} r_0' & \text{if } r_0' < c \\ (r_0' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{50}$$

$$r_0''' = \begin{cases} r_0'' & \text{if } r_0'' < M \\ (r_0'' - M) \bmod 2^w & \text{otherwise} \end{cases} \tag{51}$$

by single-word divisions have not been listed in the table.

The PowerPC has no *predicative* instructions, which are (non-branch) instructions which are only executed when a certain condition code is true. The conditional subtractions are implemented with conditional branch instructions. A misprediction of the branch target may incur a large performance penalty due to an instruction pipeline flush. The first condition in the MultiRed core is hard to predict. Some measurements showed that the condition is true 29% of the time and false 71% of the time. The first conditional subtraction of MultiRed may therefore negatively impact MultiRed performance on the PowerPC. In contrast, the first condition of MultiRed′ is true 9% of the time and false 91% of the time. The associated conditional branch is more predictable, which may explain some of the performance advantage of MultiRed′ on the PowerPC.

The Turion has a "conditional move" instruction, which can be viewed as a predicative move instruction. In the Turion code generated for the cores of both the MultiRed and the MultiRed′ algorithms, each conditional subtraction is represented with a move instruction, a subtraction, a comparison, and a conditional move instruction. While dependencies on the target register of each of these instructions may lead to pipeline bubbles, pipeline flushes need not occur. This may party explain the rather small number of cycles that the Turion needs to execute the cores of MultiRed as well as MultiRed′.

Besides on the number and kind of operations of an algorithm, the execution time of an implementation of the algorithm depends on factors like instruction arities, number of registers, handling of constants, pipelining, the programming language, compiler, and compiler options. Because the efficiency aspects of these factors can hardly be formalized, benchmarks can be used to measure an implementation's execution time. Beware though that benchmark results depend on uncalculated factors. In order to determine the relative efficiency of ModRed, MultiRed, and MultiRed′ on processors of other kinds than the ones mentioned in this section, or using compilers other than the ones mentioned, the benchmarks should be carried out with those processors and compilers.

6. CORRECTNESS PROOFS

Writing an algorithm conform to a given specification is an error-prone task. Some errors may be found by testing an implementation of the algorithm but typically some errors go unnoticed because they have only a very small chance of showing up in tests. Algorithms involving integer arithmetic are no exception; to the contrary, they may contain errors which show up in an extremely small fraction of all possible tests. Such errors can be avoided by *formally proving* the correctness of algorithms. It is possible to obtain a high degree of assurance that a "correctness proof" is not in error itself by constructing or checking it with a computer.

6.1 The choice of proof assistant

Several computerized proof assistants are available, e.g. Coq, PVS, and Isabelle. For a comparison see [Wiedijk 2003].

We have chosen to use the Coq proof assistant [Coq www] here because it produces explicit proof terms which can be checked independently with a relatively simple proof checker. This satisfies the *de Bruijn criterion*, named after the Dutch mathematician N.G. de Bruijn, who is considered to be the principal founder of machine verification of formalized proofs. He emphasized the following criterion [de Bruijn 1970] for reliable automated proof-checkers: *their programs must be small, so small that a human can (easily) verify the code by hand.* We feel that the trust which is required for mathematical algorithms like ModRed is best obtained by using a proof checker that satisfies this criterion. Of course this does not give 100% certainty of correctness since to a certain degree, the correctness of the theorems proved with Coq depends on the correctness of the Coq implementation and of the correct operation of the computer which runs Coq.

6.2 The overall proof methodology

The correctness of the ModRed, MultiRed, and MultiRed′ algorithms, defined in this text as theorem 3.1, 4.1 and 5.1, has been formally verified with a computer [Rutten www], using the Coq proof assistant [Coq www].

Before giving the essential parts of the proofs in section 6.3, we will first explain the methodology with which we maintained the correspondence between the proofs in this paper and the Coq proofs. After that we will shortly discuss the overall structure of the proofs.

Guaranteeing the connection between computerized proof and paper proof

A computerized proof has a very high degree of reliability. It may however occur that the properties that are proven do not fully correspond to the properties that have to be proven. When this happens, the proof is not wrong but it is the wrong proof. There are many syntactical differences between the Coq level and a general mathematical description. An error is easily made. Therefore we pay extra attention to guaranteeing the correspondence between the two levels. Below we explain an approach to avoid discrepancies between the Coq proof and the mathematical proof.

As an attempt to reduce the number of errors in the numbered definitions and theorems in this text, we automated the translation from Coq definitions and theorems to numbered definitions and theorems in this text. The translation is performed by a straightforward Python [Martelli 2006] script.

All numbered (and some unnumbered) definitions and theorems in this text correspond with definitions and theorems written in Gallina, the specification language of Coq. The definition of the ModRed algorithm is entirely included with equation 2 to equation 13. Similarly, the entire definitions of MultiRed and MultiRed′ are included through equations.

In the correspondences, the set $\mathbb{Z}$ is identified with the Coq set `Z`, while sets $\mathbb{Z}_n$ are identified with Coq sets `Z_ n`. To illustrate the correspondences, let us recall the ModRed correctness theorem 3.1, expressed by equation 15:

$$\forall M \in \{M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}\}\ \forall x \in \mathbb{Z}_{2^{\lceil \lg M \rceil + w}} :$$
$$\mathrm{ModRed}(M, C(M), \lfloor \frac{x}{2^w} \rfloor, x \bmod 2^w) = x \bmod M$$

In Coq, this theorem looks like

```
Theorem ModRed_eq : forall (M : Mset)
                           (x : Z_ (2 ^ (Zlog_sup M + w))),
  ModRed M
         (C M)
         (exist (in_Z_ (2 ^ w)) (x / 2 ^ w)   (xhex M x))
         (exist (in_Z_ (2 ^ w)) (x mod 2 ^ w) (xlex x))     =
  x mod M.
```

Each expression `exist (in_Z_ (2 ^ w))` z p represents a value $z \in \mathbb{Z}$ and a proof p of the fact that z is an element of $\mathbb{Z}_{2^w}$.

Structure of the proofs

The structure of the proofs closely corresponds to the structure of the algorithms. The goal of the first proof is to show that after the last step of the ModRed algorithm, the value of the result variable r''' is equal to $x \bmod M$, where $x = 2^w v + u$. For each step the proof introduces a lemma that captures an essential property of the variable defined at that step. For instance, for equation 2 in the algorithm:

$$s = \lfloor \frac{u}{2^p} \rfloor$$

we create the lemma expressed by equation 53 below.

Using such lemmas the ModRed correctness theorem is proved below in a bottom-up fashion.

6.3 Proof of the algorithm ModRed, theorem 3.1

In the rest of this section, the following assumptions hold for $M, v, u \in \mathbb{Z}_{2^w}$ and $p \in \mathbb{Z}_w$: $1 \leq M$, $M \leq 2^{w-1}$, $p = \lceil \lg M \rceil$, $v < 2^p$, and $x = 2^w v + u$. The variables s to r''' are defined with equation 2 to equation 13.

At the beginning of the proof, the definition of s (i.e. equation 2) immediately leads to the following identity.

$$s = \lfloor \frac{x \bmod 2^w}{2^p} \rfloor \tag{53}$$

By substituting the right hand side of equation 53 in $s' = 2^p s \bmod 2^w$, we get $s' = 2^p \lfloor \frac{x \bmod 2^w}{2^p} \rfloor \bmod 2^w$. Because $2^p \lfloor \frac{x \bmod 2^w}{2^p} \rfloor < 2^w$, $s' = 2^p \lfloor \frac{x \bmod 2^w}{2^p} \rfloor$. Therefore,

$$s' = x \bmod 2^w - x \bmod 2^p \tag{54}$$

The definition of h (i.e. equation 4) and the inequality $\lfloor \frac{x}{2^w} \rfloor < 2^p$ lead to the identity

$$h = \lfloor \frac{x}{2^w} \rfloor 2^{w-p} \tag{55}$$

By substituting the right hand sides of equations 55 and 53 in $h' = (h + s) \bmod 2^w$, we get $h' = (\lfloor \frac{x}{2^w} \rfloor 2^{w-p} + \lfloor \frac{x \bmod 2^w}{2^p} \rfloor) \bmod 2^w$. Therefore, $h' = (\lfloor \frac{\lfloor \frac{x}{2^w} \rfloor 2^{w-p} 2^p}{2^p} \rfloor + \lfloor \frac{x \bmod 2^w}{2^p} \rfloor) \bmod 2^w = (\lfloor \frac{\lfloor \frac{x}{2^w} \rfloor 2^w + x \bmod 2^w}{2^p} \rfloor) \bmod 2^w = (\lfloor \frac{x - x \bmod 2^w + x \bmod 2^w}{2^p} \rfloor) \bmod 2^w = \lfloor \frac{x}{2^p} \rfloor \bmod 2^w$. Because $x < 2^{p+w}$, this leads to

$$h' = \lfloor \frac{x}{2^p} \rfloor \tag{56}$$

By substituting the right hand side of this equation in the definition of q (i.e. equation 6), we get

$$q = \lfloor \frac{\lfloor \frac{x}{2^p} \rfloor (\lfloor \frac{2^{p+w}}{M} \rfloor \bmod 2^w)}{2^w} \rfloor \tag{57}$$

The right hand sides of equations 57 and 56 can be substituted in $q' = (q + h') \bmod 2^w$, leading to $q' = (\lfloor \frac{\lfloor \frac{x}{2^p} \rfloor (\lfloor \frac{2^{p+w}}{M} \rfloor \bmod 2^w)}{2^w} \rfloor + \lfloor \frac{x}{2^p} \rfloor) \bmod 2^w$. Because $\lfloor \frac{2^{p+w}}{M} \rfloor = 2^w + (\lfloor \frac{2^{p+w}}{M} \rfloor \bmod 2^w)$, we have $q' = (\lfloor \frac{\lfloor \frac{x}{2^p} \rfloor (\lfloor \frac{2^{p+w}}{M} \rfloor - 2^w)}{2^w} \rfloor + \lfloor \frac{x}{2^p} \rfloor) \bmod 2^w$, so $q' =$

$\lfloor\frac{\lfloor\frac{x}{2^p}\rfloor\lfloor\frac{2^{p+w}}{M}\rfloor}{2^w}\rfloor \bmod 2^w$. Central to the proof is to show that q' has only two possible values:

$$q' = \lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor \bmod 2^w \vee q' = (\lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor - 1) \bmod 2^w \tag{58}$$

This is easy to prove using equation 1. By substituting the right hand sides of equation 58 in $y = q'M \bmod 2^w$, we get

$$y = \lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor M \bmod 2^w \vee y = (\lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor M - M) \bmod 2^w \tag{59}$$

The right hand side of equation 54 and the right hand sides of equation 59 can then be substituted in $d = (s' - y) \bmod 2^w$, obtaining $d = (x \bmod 2^w - x \bmod 2^p - (\lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor M(-M)) \bmod 2^w) \bmod 2^w = (x - x \bmod 2^p - \lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor M(+M)) \bmod 2^w$. Therefore, $d = (\lfloor\frac{x}{2^p}\rfloor 2^p - \lfloor\frac{\lfloor\frac{x}{2^p}\rfloor 2^p}{M}\rfloor M(+M)) \bmod 2^w = (\lfloor\frac{x}{2^p}\rfloor \bmod M(+M)) \bmod 2^w$. Because $M \leq 2^{w-1}$, this leads to

$$d = \lfloor\frac{x}{2^p}\rfloor 2^p \bmod M \vee d = \lfloor\frac{x}{2^p}\rfloor 2^p \bmod M + M \tag{60}$$

Now equation 54 can be employed to derive $r = (x \bmod 2^w - y) \bmod 2^w = (s' + x \bmod 2^p - y) \bmod 2^w = ((s' - y) \bmod 2^w + x \bmod 2^p) \bmod 2^w$, so

$$r = (d + x \bmod 2^p) \bmod 2^w \tag{61}$$

In case $d < M$, the first right hand side of equation 60 can be substituted in equation 61, and in case $d \geq M$, the second right hand side of equation 60 can be substituted in equation 61. In the first case it holds $r' = r$, while in the second case it holds $r' = (r - M) \bmod 2^w$. Both cases lead to $r' = (\lfloor\frac{x}{2^p}\rfloor 2^p \bmod M + x \bmod 2^p) \bmod 2^w$, and this immediately leads to

$$r' = \lfloor\frac{x}{2^p}\rfloor 2^p \bmod M + x \bmod 2^p \tag{62}$$

If $r' < M$, then $r' = (\lfloor\frac{x}{2^p}\rfloor 2^p \bmod M + x \bmod 2^p) \bmod M = (\lfloor\frac{x}{2^p}\rfloor 2^p + x \bmod 2^p) \bmod M = x \bmod M$. If $M \leq r' < 2M$, then $r' = (\lfloor\frac{x}{2^p}\rfloor 2^p \bmod M + x \bmod 2^p) \bmod M + M = x \bmod M + M$. If $2M \leq r'$, then $r' = x \bmod M + 2M$. Using these three cases and using $r'' = r'$ if $r' < M$ and $r'' = (r' - M) \bmod 2^w$ if $r' \geq M$, we get

$$r'' = x \bmod M \vee r'' = x \bmod M + M \tag{63}$$

Using $r''' = r''$ if $r'' < M$ and $r''' = (r'' - M) \bmod 2^w$ if $r'' \geq M$, this leads to

$$r''' = x \bmod M \tag{64}$$

Equation 64 immediately leads to the ModRed correctness theorem, expressed by equation 15.

The correctness of the core of the MultiRed algorithm (eventually leading to theorem 4.1) is largely proved in the same way as the correctness of the ModRed algorithm. The MultiRed algorithm applies this core within a loop. The correctness of the loop is proved with natural induction using an induction step expressed as follows, where $\text{fst}(a, b) = a$, $\text{snd}(a, b) = b$, $x_i = \lfloor\frac{x}{2^{wi}}\rfloor \bmod 2^w$, and $f'((d_{i+1}, r_{i+1}), x_i) = (d_i, r_i)$, the latter definition assuming that M and the components of $C'(M)$ are given as inputs.

$$\forall d, r \in Z_{2^w} \forall n \in \mathbb{N} : P(M, p, d, r, \lfloor\frac{x}{2^{(n+2)\cdot w}}\rfloor, x_{n+1}) \implies$$
$$P(M, p, \mathrm{fst}(f'((d, r), x_n)), \mathrm{snd}(f'((d, r), x_n)), \lfloor\frac{x}{2^{(n+1)\cdot w}}\rfloor, x_n)$$

The definition of this induction step is similar to a loop invariant. It uses a predicate P expressing the required relations between its arguments such that at the end of the loop the required property holds. At the end of the loop, values of d and r have been obtained such that $P(M,\ p,\ d,\ r,\ \lfloor\frac{x}{2^w}\rfloor,\ x_0)$ holds. Because of this property of d and r, $x \bmod M$ is easy to derive from d and r. The predicate P is defined as follows.

$$\begin{array}{rll}
P(M, p, d, r, h, l) = (& d & = \lfloor\frac{(h \bmod M)2^w+l}{2^p}\rfloor 2^p \bmod M \\
\vee & d & = \lfloor\frac{(h \bmod M)2^w+l}{2^p}\rfloor 2^p \bmod M + M \\
\vee & d & = \lfloor\frac{(h \bmod M+M)2^w+l}{2^p}\rfloor 2^p \bmod M \\
\vee & d & = \lfloor\frac{(h \bmod M+M)2^w+l}{2^p}\rfloor 2^p \bmod M + M \\
) & & \\
\wedge & r & = (d + l \bmod 2^p) \bmod 2^w
\end{array}$$

The induction step infers a property of n from a property of $n + 1$. Proofs by induction usually employ a step where a property of $n+1$ is inferred from a property of n, but here the order of n and $n+1$ is reversed, starting with the highest nonzero word of x and iterating through the words of x until its lowest word is reached.

With the MultiRed$'$ correctness proof (theorem 5.1), a predicate slightly different from P is employed in an induction step. Instead of $r = (d + l \bmod 2^p) \bmod 2^w$ it has $r = (d + l \bmod 2^p \bmod M) \bmod 2^w$.

With the MultiRed correctness proof, a constant $M'' = (2^{w-p}M) \bmod 2^w = 2^{w-p}M$ is involved. The constant does not appear in the ModRed correctness proof. The conditional subtraction of M'' from s_i, obtaining s''_i, and the addition of s''_i to $2^{w-p}r'_i$ has replaced the equivalent conditional subtraction of M from r'_i and the addition of the result, times 2^{w-p}, to s_i.

The MultiRed correctness proof involves a subtraction of a multiple of M depending on the condition $r'_i < 2^p$ instead of $r'_i < M$. This does not cause overflows, and it relates to the fact that ModRed can reduce integers in $\mathbb{Z}_{2^p 2^w}$ rather than just integers in $\mathbb{Z}_{M\cdot 2^w}$. Also, the conditional subtraction of a multiple of M does not affect the final result $x \bmod M$. That can also be said of each conditional subtraction in MultiRed$'$.

6.4 Evaluation of proof methodology

In our proof scripts, 272 theorems and lemmas are found at the time of this writing. The proof scripts are comprised of 2800 lines, amounting to 140403 characters. The proofs employ large distributed Coq libraries. We made numerous revisions and variations of our proofs, and in the course of time, due to an increased experience in proving, the size of the proofs, and the time to construct them, shrank considerably. The lemmas and theorems were first proved in mind, then on paper, and finally, using Coq, on a computer. The proofs in mind made implicit use of many properties of integers. The proofs on paper required these properties to be explicated. We estimate that the total effort of producing all revisions and variations of the proofs was about 6 weeks full-time work. This includes about 2 weeks to create a number of w-bit integer arithmetic definitions and lemmas. It would take about 3 days to make a proof of an entirely new variant on a few sheets of paper, and it would take about 6 days to convert such a proof to Coq.

During the proof process no errors in the algorithms were found. The Coq proofs provide the best possible trust one can have in the correctness of these algorithms.

Many definitions and small lemmas that were needed for the proofs can also be used in other proofs concerning computer arithmetic. To facilitate that, we placed the proofs and the more general definitions and lemmas they rely on in separate files on the Coq site [Rutten www]. This may serve as the first step in creating a large computer arithmetic library for formal proofs.

7. RELATED WORK

With Barrett's division-free modular reduction algorithm [Barrett 1987], $x \bmod M$ is computed by first approximating $\lfloor \frac{x}{M} \rfloor$ with $q = \lfloor \lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{M} \rfloor / b \rfloor$, for $a = d^{\lfloor \log_d M \rfloor}$ and $b = d^{\lfloor \log_d M \rfloor + 2}$, where $d \in \mathbb{Z} \mid d \geq 4$. As can be seen in e.g. [Bosselaers et al. 1994], and in equations 1 and 65 of this text, it is also possible to choose $d = 2$. The value of $q = \lfloor \lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{M} \rfloor / b \rfloor$ can be $\lfloor \frac{x}{M} \rfloor$, $\lfloor \frac{x}{M} \rfloor - 1$, or $\lfloor \frac{x}{M} \rfloor - 2$, as

$$\forall y, b \in \mathbb{Z}^+ \ \forall a \in \mathbb{Z}_y \ \forall x \in \mathbb{Z}_{ab+a} : \lfloor \frac{x}{y} \rfloor - 2 \leq \lfloor \frac{\lfloor \frac{x}{a} \rfloor \lfloor \frac{ab}{y} \rfloor}{b} \rfloor \leq \lfloor \frac{x}{y} \rfloor \qquad (65)$$

From the 3 possible values of q, it immediately follows that $x - qM$ can be equal to $x \bmod M$, $x \bmod M + M$, or $x \bmod M + 2M$. If $M > \lfloor \frac{2^w}{3} \rfloor$, the value of $x - qM$ may be greater than or equal to 2^w, which means there may be a nonzero word overflow $\lfloor \frac{x - qM}{2^w} \rfloor$. In that case, $x \bmod M$ cannot be derived from $(x - qM) \bmod 2^w$ just using at most two comparisons and two subtractions. In contrast, with equation 1 a similar overflow can only occur when $M > \lfloor \frac{2^w}{2} \rfloor$.

While Barrett's algorithm with $d = 2$ is based on equation 65, just like ModRed, it uses $a = 2^{\lfloor \lg M \rfloor}$ and $b = 2^{\lfloor \lg M \rfloor + 2}$ instead of $a = 2^{\lceil \lg M \rceil}$ and $b = 2^w$. Barrett's algorithm is usually employed in multi-word integer arithmetic, which is why it does not need to take special measurements to avoid word overflows. Only the upper half of the product $\lfloor \frac{x}{2^{\lfloor \lg M \rfloor}} \rfloor \lfloor \frac{2^{2(\lfloor \lg M \rfloor + 1)}}{M} \rfloor$ needs to be calculated in principle though. In the context of small moduli, the `udiv_qrnnd_preinv1` macro of (version 4.1.4 of) the GNU Multiple Precision Arithmetic Library [GMP www] shows that the low word of $\lfloor \frac{x}{2^{\lceil \lg M \rceil}} \rfloor \lfloor \frac{2^{\lceil \lg M \rceil + w}}{M} \rfloor$ need not be computed at all for $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$. A similar thing can also be seen in our ModRed algorithm.

One of the algorithms from [Granlund and Montgomery 1994] is for division and reduction of integers in $\mathbb{Z}_{2^{2w}}$ by positive integers in $\mathbb{Z}_{2^w}$. When the divisor is restricted to the smaller domain $\{M \in \mathbb{Z} \mid 2^{w-1} \leq M < 2^w\}$, some operations of this algorithm need not be carried out. This leads to the `udiv_qrnnd_preinv1` macro of [GMP www]. The macro yields $\lfloor \frac{x}{M} \rfloor$ and $x \bmod M$ for $x \in \mathbb{Z}_{2^{2w}}$ and $M \in \mathbb{Z} \mid 2^{w-1} \leq M < 2^w$. The algorithm and macro employ $\lfloor \frac{2^{\lfloor \lg M \rfloor + 1 + w}}{M} \rfloor \bmod 2^w$ as a predefined value, in contrast with ModRed, which uses $\lfloor \frac{2^{\lceil \lg M \rceil + w}}{M} \rfloor - 2^w$, and in contrast with Barrett's algorithm, which employs $\lfloor \frac{2^{2(\lfloor \lg M \rfloor + 1)}}{M} \rfloor$.

The algorithms passed in review above are for reduction of certain double words, including products of two integers $m, n \in \mathbb{Z}_M$. That is because $\forall x, y \in \mathbb{Z}_M : 0 \leq xy < M^2 \leq 2^{\lceil \lg M \rceil} 2^{w-1} < 2^{\lceil \lg M \rceil + w}$. Schrage's algorithm [Bratley et al. 1987] is designed to perform modular multiplications $mn \bmod M$, having inputs M and $m, n \in \mathbb{Z}_M$. It contains divisions rather than high-word multiplications. When high-word and double-word multiplication instructions are not available (and thus have to be emulated in software), Schrage's algorithm may evaluate modular multiplications more efficiently than with the previously mentioned modular reduction algorithms.

Some algorithms perform modular multiplications with very special moduli, for example with $M = 2^{31} - 1$ [Payne et al. 1969]. Such algorithms are much less generally applicable than general modular reduction algorithms but their implementations can be much more efficient.

8. FUTURE WORK

Further transformations may be of interest for investigation in the future. It may be possible to replace comparisons with M by comparisons with $2^{\lceil \lg M \rceil}$ in more places. This may require additional corrections at other places in order to maintain correctness. These corrections might influence performance negatively. Different variants will have to be considered, correctness will have to be proven, and performance measurements will have to be done.

It is also possible to derive assembly code from the algorithms without using an intermediate language. By formalizing (parts of) the models of certain processors in Coq, one would be able to directly prove the correctness of the assembly code for those processors.

The proof scripts are currently expressed in terms of a single opaque definition of $w \in \mathbb{Z}^+$. It may be possible to use a Coq mechanism which keeps the scripts independent of any particular value of w but which also allows one to have different instantiations of w (like 32 and 64) in order to perform unsigned w-bit integer arithmetic computations in Coq.

Recently, ideas for a new reduction method have been communicated by Peter Montgomery on the GMP site ([GMP www]). For many cases these ideas seem to incorporate an improvement. It is less clear yet whether they will also be an improvement for reducing relatively small large integers with moduli that are different for each call. As future work it seems worthwhile to define and implement an algorithm using these ideas as a starting point, to prove its correctness with a proof assistant, and to compare its performance with the performance of the algorithms proposed in this paper.

9. CONCLUSIONS

We have proposed an algorithm – ModRed – for reduction of integers in $\mathbb{Z}_{2^{\lceil \lg M \rceil + w}}$ by a modulus M on w-bit processors, where $w \in \mathbb{Z}^+$ and $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$. We have also proposed algorithms MultiRed and MultiRed′ – based on ModRed – for reduction of multi-word integers by moduli $M \in \mathbb{Z} \mid 1 \leq M \leq 2^{w-1}$.

With measurements on processors which provide relatively slow divisions of double words by single words, we have shown that implementations of MultiRed can sometimes be over 30% more efficient than comparable implementations based on the algorithm proposed by Granlund and Montgomery for division and reduction of an unsigned double word by an unsigned word. That algorithm applies to larger moduli $M \in \mathbb{Z} \mid 2^{w-1} \leq M < 2^w$ as well, so with respect to the algorithm, ModRed trades some generality for some efficiency.

The formal correctness of the algorithms has been proved with the aid of the Coq proof assistant. This gives a very high degree of trust in the correctness of these algorithms that are expressed on the level of abstract machine instructions.

ACKNOWLEDGMENT

The authors would like to thank John van Groningen for insight and support relevant to this research.

REFERENCES : WEBSITES

AMD. Advanced Micro Devices, AMD - Processor Homepage. `www.amd.com`.

Coq. The Coq proof assistant. `pauillac.inria.fr/coq`.

GCC. GCC Home Page - GNU Project - Free Software Foundation (FSF). `gcc.gnu.org`.

GMP. The GNU MP Bignum Library. `www.swox.com/gmp`.

IBM. IBM Power Architecture. `www.ibm.com/chips/power`.

Rutten, L. Fast Modular Reduction Proof Scripts in Coq. `coq.inria.fr/contribs/ModRed.html`.

REFERENCES : ACADEMIC

Barrett, P. 1987. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology, Proc. Crypto '86*, A. Odlyzko, Ed. Lecture Notes in Computer Science, vol. 263. Springer-Verlag, 311–323.

Bosselaers, A., Govaerts, R., and Vandewalle, J. 1994. Comparison of three modular reduction functions. In *Advances in Cryptology, Proc. Crypto '93*. Lecture Notes in Computer Science, vol. 773. Springer-Verlag, 175–186.

Bratley, P., Fox, B., and Schrage, L. 1987. *A Guide to Simulation*, second ed. Springer-Verlag New York Inc.

de Bruijn, N. G. 1970. The mathematical language automath, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*. Lecture Notes in Mathematics, vol. 125. INRIA, Versailles, 1968, Springer, Berlin, 29–61.

Granlund, T. and Montgomery, P. 1994. Division by Invariant Integers using Multiplication. *SIGPLAN Notices 29,* 6, 61–72.

Knuth, D. 1998. *The Art of Computer Programming*, third ed. Vol. 2 – Seminumerical Algorithms. Addison Wesley Longman.

Lewis, P.A.W., Goodman, A.S., and Miller, J.M. 1969. A pseudo-random number generator for the System/360. *IBM Systems Journal 8,* 2, 136–146.

Martelli, A. 2006. *Python in a Nutshell*, second ed. O'Reilly.

Payne, W., Rabung, J., and Bogyo, T. 1969. Coding the Lehmer Pseudo-random Number Generator. *Communications of the ACM 12,* 2, 85–86.

Wiedijk, F. 2003. Comparing Mathematical Provers. In *Mathematical Knowledge Management: Second International Conference, Proceedings*, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds. Lecture Notes in Computer Science, vol. 2594. Springer, 188–202.

Received January 2008; revised June 2009; accepted October 2009