A Verification of Brickell's Fast Modular Multiplication Algorithm

Colin D. Walter & Stephen E. Eldridge

Department of Computation
UMIST
PO Box 88
Manchester M60 1QD, UK
www.co.umist.ac.uk

Abstract. This paper refers to the algorithm and its hardware implementation described by Brickell [1] for modular multiplication in $N+10$ clock pulses where N is the number of bits in the binary integers involved. That paper uses a delayed carry representation which consists of two registers of N bits each – one for the uncarried carries. Of course, up to N clocks ticks may eventually be required to assimilate the carries at the end of the computation.

Several sources of possible error are reported here – one in the hardware, one in the specification which the intended hardware satisfies, and one in the definition of the control variables T_1 and T_2. Our main contributions are the supply of further detail to remove such ambiguities, a determination of the minimum number of extra bits required during the calculation, a verification of the more detailed system, and its extension to an integer division procedure.

The existence of a proof enables it to be used reliably for its intended purpose in applications such as cryptography [5], where attempts have already been made to use the algorithm or similar methods in RSA chips [4]. The reduced number of extra bits required also marginally increases the speed.

Concurrent work by J. K. Gibson [2] describes different additional detail to make Brickell's algorithm work. What is supplied in this article we believe to be more natural than Gibson's in that the standard delay-carry addition stands unaltered here, and the minimum number of clock pulses required remains clear.

Key words: RSA algorithm, fast multiplication, verification, cryptography, computer arithmetic.

C.R. Categories: F.2.1, B.7.1, E.3.

1 Introductory Detail

1.1 Type Definitions

We use a slightly extended version of Pascal to describe the main parts of our version of Brickell's algorithm [1]. Let N be the number of bits needed for each input integer. An additional Q+1 (≥ 6) bits of working space are required so that the natural size of binary integers becomes N+Q+1 bits. Brickell used 11 extra bits.

```
type BitIndex           = 0..N+Q ;
     BinaryInteger      = array[BitIndex] of Boolean ;
     DelayCarryInteger  = array[1..2] of BinaryInteger ;
```

Every $D : DelayCarryInteger$ is required to satisfy the data invariant

$$(\text{not } D[2,0]) \quad \& \quad (\text{not}(D[1,I{-}1] \;\&\; D[2,I]) \text{ for } I \text{ in } 1\,..\,N{+}Q)$$

Here $D[2]$ can be considered to contain the unassimilated carries resulting from the addition of two numbers of type $DelayCarryInteger$. The integer represented by D is $\sum_{I\in \mathrm{BitIndex}}(D[1,I]{+}D[2,I]){\times}2^I$ where $True$ and $False$ are interpreted as 1 and 0 respectively. We will not, in fact, distinguish $Booleans$, $BinaryIntegers$ and $DelayCarryIntegers$ from their values as integers unless the context makes it necessary. Maximum values are given by choosing $D[2,I] = 1$ rather than $D[1,I{-}1] = 1$ in the data invariant. So,

Lemma 1 The maximum value of a delay carry integer with L bits is $2^L + 2^{L-1} - 2$.

1.2 The Half Adder

Brickell describes a half adder which takes as input two binary integers (X and Y here) and produces as output two binary integers. We stress the relationship between the output integers by combining them into a delay carry integer D. This must then necessarily satisfy the data invariant above. The procedure HalfAdd(X,Y: BinaryInteger; **var** D: DelayCarryInteger; **var** Overflow: Boolean) achieves

$$\begin{array}{lrcl} & D \quad + Overflow{\times}2^{N+Q+1} & = & X+Y \\ \text{by} & D[2] + Overflow{\times}2^{N+Q+1} & = & 2(X \text{ and } Y) \\ \text{and} & D[1] & = & X \text{ xor } Y \end{array}$$

which are interpreted bitwise. Software or hardware for the above can be executed in one clock pulse by performing the bit operations in parallel. To distinguish different overflows it is sometimes convenient to describe the one here as $D[2, N{+}Q{+}1]$, although the register contains no such bit. Easily,

Lemma 2

i) Suppose that 2^L divides both inputs of *HalfAdd*. Then 2^L divides both the outputs $D[1]$ and $D[2]$.

ii) Suppose that 2^L divides one input of *HalfAdd*. Then 2^{L+1} divides the output $D[2]$.

Lemma 3

i) The outputs $D[2, I{+}1]$ and $D[1, I]$ of *HalfAdd* depend only on the inputs $X[I]$ and $Y[I]$. In particular, the *Overflow* parameter returned only depends on $X[N{+}Q]$ and $Y[N{+}Q]$.

ii) For delay carry integers X, Y and D, the result of the call *HalfAdd*$(X[1], Y[1], D, Overflow)$ satisfies $(D[2]\ \&\ X[2]) = 0$ and $(D[2]\ \&\ Y[2]) = 0$.

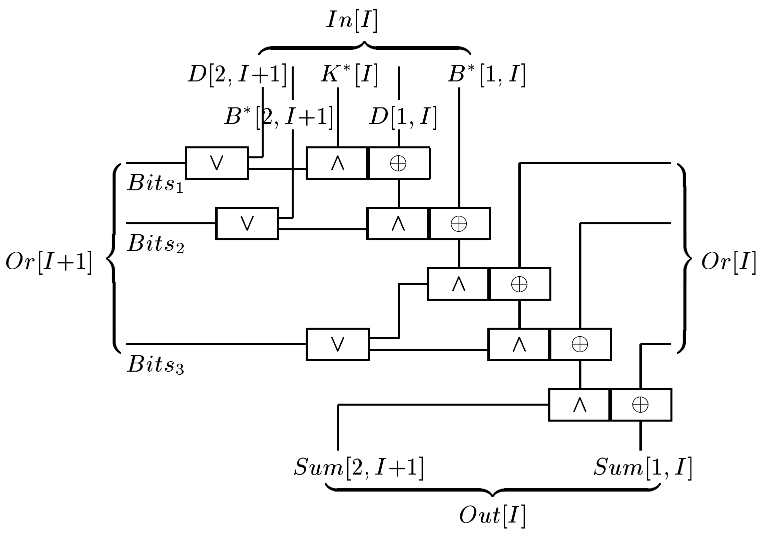

Fig. 1. Definitions of $In[I]$, $Or[I]$ and $Out[I]$.

1.3 Adding Three Numbers

The main step in the modular multiplication algorithm requires the addition of a binary integer and two delay carry integers. The precise specification is messy to give, and so we make use of the code given to produce the results of this section. A hardware implementation is given in Figure 2 of [1] using a cascade of 5 half adders, but with an extra shift. The repeated unit without the shift is given in Figure 1.

```
procedure Add(D,B* : DelayCarryInteger ; K* : BinaryInteger ;
             var Sum: DelayCarryInteger ; var Overflows: Integer);

{ Post-Condition : Sum + Overflows*2^(N+Q+1) = B*+D+K* }

var  Bits1, Bits2, Bits3 : BinaryInteger ;
     R1, R2, R3, R4       : DelayCarryInteger ;
     Overflow1, Overflow2, Overflow3, Overflow4, OverflowSum
                          : Boolean ;
begin { Add }
     HalfAdd(K*, D[1], R1, Overflow1) ;
     HalfAdd(R1[1],B*[1],R2,Overflow2) ;
     Bits1  :=  D[2]  OR  R1[2] ;
     HalfAdd(R2[1],Bits1,R3,Overflow3) ;
     Bits2  :=  B*[2]  OR  R2[2] ;
     HalfAdd(R3[1],Bits2,R4,Overflow4) ;
     Bits3  :=  R3[2]  OR  R4[2] ;
     HalfAdd(R4[1],Bits3,Sum,OverflowSum) ;
     Overflows := Ord(Overflow1)+Ord(Overflow2)+Ord(Overflow3)
                  +Ord(Overflow4)+Ord(OverflowSum)
end ; { Add }
```

Here the OR is interpreted componentwise for each *BitIndex*. For each call of it the ANDed inputs give 0 by Lemma 3(ii), and so the output is their sum. Neglecting the *Overflows* which may be regarded as $[N{+}Q{+}1]$-components, the code of *Add* is now equivalent to 8 assignments each of which just adds two binary integers, and so is easy to verify. Note from the figures that this procedure can be implemented by hardware which operates in a single clock pulse. The following results are straightforward:

Lemma 4
If 2^L divides the inputs $D[1]$, $B^*[1]$ and K^* of *Add*, and 2^{L+1} divides the inputs $D[2]$ and $B^*[2]$, then 2^L divides the output $Sum[1]$ of *Add* and 2^{L+3} divides the output $Sum[2]$. In particular, $Sum[2,0] = Sum[2,1] = Sum[2,2] = 0$.

Lemma 5
i) Suppose 2^L divides each of the five input binary integer components of the parameters D, B^* and K^* of *Add*. Then 2^L divides both the outputs $Sum[1]$ and $Sum[2]$.
ii) Suppose 2^L divides at least four of the five input binary integer components of the parameters D, B^* and K^* of *Add*. Then 2^{L+2} divides the output $Sum[2]$.

Lemma 6
i) The input bits of *Add* which contribute to the output bits $Sum[1, I]$ and $Sum[2, I{+}1]$ are $D[1, J]$, $B^*[1, J]$, $K^*[J]$, for J in $I{-}2\,..\,I$ and $D[2, J]$ and $B^*[2, J]$ for J in $I{-}1\,..\,I$.

ii) The input bits which contribute to overflow in *Add* are $D[1,J]$, $B^*[1,J]$ and $K^*[J]$ for J in $N{+}Q{-}2\,..\,N{+}Q$, and $D[2,J]$, $B^*[2,J]$ for J in $N{+}Q{-}1\,..\,N{+}Q$.

1.4 The Top Bits

Algorithms computing a result modulo C often progress by subtracting a multiple of C when possible. Where numbers are represented by registers of length L, subtraction of C is the same as addition of $K = 2^L - C$, providing that there is a single overflow of 2^L which is neglected. Brickell's algorithm does this, predicting in advance the overflows when the procedure *Add* is executed, as well as the overflow when a subsequent shift is applied.

By Lemma 6 the *Add* overflow is given by looking at the top 3 bits only of the inputs K^*, $B^*[1]$ and $D[1]$, and the top 2 bits only of $B^*[2]$ and $D[2]$. For this reason we look at 3-bit delay carry integers satisfying the usual data invariant. By adding a subscript 3, all our definitions and procedures can be repeated for $BitIndex_3 = 0\,..\,2$ instead of $BitIndex$. Now Add_3 applied to the most significant bits of the input to *Add* will predict the overflows of *Add* correctly. This creates a need for a function $Top3Bits$ which will copy bits with indices in the range $N{+}Q{-}2\,..\,N{+}Q$ to a new type with indices $0\,..\,2$, discarding the rest. The data invariant for the $DelayCarryInteger_3$ output is obtained by forcing the $[2,0]$-component to be $False$. All this applies also with subscript I referring to $BitIndex_I = [0\,..\,I{-}1]$. For example,

Definition. For natural numbers I, the function

$$TopIBits(D : DelayCarryIntger) : DelayCarryInteger_I$$

is specified by $TopIBits(D)[1] = D[1] \text{ div } 2^{N+Q+1-I}$ and $TopIBits(D)[2] = 2{\times}(D[2] \text{ div } 2^{N+Q+2-I})$.

There is an analogous definition for $TopIBits$ applied to a variable of type $BinaryInteger$. Now, by Lemma 6,

Lemma 7

i) If $I \geq 3$ then, for the same input,
$Add(D, B^*, K^*, Sum, Overflows)$
and
$Add_I(TopIBits(D), TopIBits(B^*), TopIBits(K^*), Sum_I, Overflows_I)$
will satisfy $Overflows = Overflows_I$.

ii) If $I \geq 4$ then, for the same input,
$Add(D, B^*, K^*, Sum, Overflows)$
and
$Add_I(TopIBits(D), TopIBits(B^*), TopIBits(K^*), Sum_I, Overflows_I)$
will satisfy $Sum[1, N{+}Q] = Sum_I[1, I{-}1]$, $Sum[2, N{+}Q] = Sum_I[2, I{-}1]$, and $Overflows = Overflows_I$.

A numerical characterisation of overflows is useful. Working with the numbers the bits represent rather than with the bits themselves is more elegant, and our version of Brickell's algorithm is written in this way. However, Gibson [2] deals directly with the overflow bits and, although it is not stated, we presume Brickell also meant to refer to the overflow bits of the delay carry integers rather than overflows in their numerical equivalents.

Lemma 8

i) The value of the overflows in *Add* is

$$[Top3Bits(D)+Top3Bits(K^*)+Top3Bits(B^*)] \operatorname{div} 8$$

ii) The sum of the output bits $Sum[1, N+Q]$ and $Sum[2, N+Q]$ of *Add* is

$$[Top4Bits(D)+Top4Bits(K^*)+Top4Bits(B^*)] \operatorname{div} 8$$

$$-2(\ [Top3Bits(D)+Top3Bits(K^*)+Top3Bits(B^*)] \operatorname{div} 8\ )$$

By Lemma 7(i), the proof of (i) can be simplified by first noting that it suffices to look at the corresponding call to Add_3. From the analogue to Lemma 4 for Add_3 (just take $N+Q = 2$ there), we have the output Sum_3 of Add_3 satisfying $Sum_3[2,0] = Sum_3[2,1] = Sum_3[2,2] = 0$, i.e. $Sum_3[2] = 0$. So $Sum_3 = Sum_3[1]$ and the maximum value of Sum_3 is therefore that of $Sum_3[1]$, namely 7. Thus $Sum_3 = Sum_3 \bmod 8$ and by the post-condition for *Add* reduced modulo 8, $Sum_3 = (Top3Bits(D)+Top3Bits(K^*)+Top3Bits(B^*)) \bmod 8$. The post-condition for *Add* now taken div 8 gives the number of overflows as that stated since $Sum_3 \operatorname{div} 8 = 0$.

Now consider part (ii). Suppose we ignore the topmost bits, i.e. those of index $N+Q$, so that $Top3Bits$ extracts the bits of index $N+Q-3 \,..\, N+Q-1$. Then the expression of (i) would be the "overflow" from the $N+Q-1$st place rather than the $N+Q$th. The same "overflow" is obtained by applying $Top4Bits$ properly instead of $Top3Bits$, if we ignore the input bits of index $N+Q$. This new expression, which is the first term in (ii), therefore represents all bits flowing into the $N+Q$th place, both those coming from direct input and those passing up the adder from lower positions. Some of these bits combine in pairs to give the overflow bits from the $N+Q$th position, whilst the remainder give the output from the $N+Q$th position. Hence this last output is given by subtracting twice the number of overflow bits (as in (i)) from the number of bits flowing into the $N+Q$th place.

Lemma 9

a) If $24 > Top4Bits(D)+Top4Bits(K^*)+Top4Bits(B^*) \geq 16$ then exclusively either
   i) there is one overflow from *Add*, or
   ii) the two output bits from *Add* with index $N+Q$ are 1;

b) If $16 > Top4Bits(D)+Top4Bits(K^*)+Top4Bits(B^*) \geq 8$ then
   i) there is no overflow from *Add*, and

ii) one output bit from *Add* with index $N+Q$ is 1, the other 0;

c) If $8 > Top4Bits(D)+Top4Bits(K^*)+Top4Bits(B^*)$ then

i) there is no overflow from *Add*, and

ii) neither output bit from *Add* with index $N+Q$ is 1.

The proof of this depends on the previous lemma. Because in general $Top4Bits(S) \geq 2\times Top3Bits(S)$, it is clear from Lemma 8(i) that in cases (b) and (c) there can be no overflows, and in case (a) at most 1 overflow. As remarked in the proof of Lemma 8(ii), the expression

$$[Top4Bits(D)+Top4Bits(K^*)+Top4Bits(B^*)] \operatorname{div} 8$$

gives the sum of the output bits with index $N+Q$ and the overflow bits. In case (a) this amounts to 2, which can be split only as expressed in (i) or (ii). In cases (b) and (c) it amounts to 1 and 0 respectively, so that there is no overflow, as stated in the parts (i), and the output bits must be as stated in the parts (ii).

2 The Modular Multiplication Algorithm

The modular multiplication algorithm M can now be described. Several minor changes have been introduced which make it different from Brickell's description: the loop is more naturally entered at a different point, the number of extra bits required beyond the N of the inputs is made variable ($Q+1$ instead of 11), and the Booleans T_1 and T_2 have been accumulated into a delay carry integer T which almost gives the result of integer division. Of course, if the integer quotient is not required then T may be discarded.

One further possible difference between this algorithm and that given by Brickell is worth noting. It concerns the definition of the Booleans $T[1, J-1]$ and $T[2, J]$ – Brickell's T_1 and T_2. Our definition using the function $Top4Bits$ ignores the bit which might have formed the $[2,0]$-component in the result, but Brickell seems to include it. As it does not contribute to any overflows it must be ignored if a numerical characterisation of the different cases is used as here. We must assume therefore that when Brickell mentions overflows he is referring to overflow in a delay carry representation not a binary integer representation. He is not explicit on this distinction and so readers could be misled to an incorrect definition of the Booleans T_1 and T_2.

The idea is first to multiply the most significant bits of A by B to obtain the most significant bits of the product. Then, when the production of less significant bits results in little or no change to the highest bits, multiples of C are subtracted. In fact, since only addition is allowed, rather than subtracting C, the algorithm must add a multiple of $K = 2^N - C$. This is chosen to ensure that the overflows match the multiple added. For example, adding $KQ1 = 2^{Q+1}\times(2^N - C)$ is done when an overflow of exactly $2^{N+Q+1}$is guaranteed, in order that the result is equivalent to subtracting $2^{Q+1}\times C$. These overflows, (which arise from either of the calls to *Add* or *Shift* in the algorithm below), are predicted by Lemma 9.

The specification given at the head of the procedure will show that it is not precisely a modular multiplication algorithm: the residue may differ from the least non-negative one by the addition of C, so that the integer quotient is 1 less than it should be. Brickell makes no comment that the residue may not be in the expected range. Examples which illustrate this are easy to find.

Finally note that our proof includes very naturally the limit case of $C = 2^{N-1}$. That it works for such values of C is not as accidental as Brickell's treatment suggests.

2.1 Code for the Algorithm

We introduce first one more function and a few definitions before giving the code of the algorithm. Multiplication by a power of 2 whether to a positive or negative exponent is achieved by a (hardwired) shift up or down. For this we use a function

$$Shift(D : DelayCarryInteger; E : Integer) : DelayCarryInteger$$

to shift D by E places. Possible overflow is important, but not underflow in this context. Shifting in either direction will not change the invariant property for delay carry integers D since it is accompanied within the function by finally setting $D[2, 0] = 0$. Thus a possibly significant bit may be lost.

The code is documented with pre- and post- conditions, and a three part loop invariant. The proof and correct action of the algorithm depend upon the previous properties established for a particular implementation of *Add* and on Q being taken large enough: $Q \geq 5$ is proved here and included as a pre-condition. Incorrect action can be demonstrated for $Q \leq 4$.

```
procedure M(A,B : DelayCarryInteger ; C : BinaryInteger ;
            var D,T : DelayCarryInteger ) ;

{ Write Only     :  D, T                          }

{ Pre-Conditions :  A[1]<2^N, A[2]<2^N,
                    B[1]<2^N, B[2]<2^N,
                    2^(N-1) ≤C<2^N,
                    Q ≥ 5.                         }

{ Post-Conditions:  A*B = C*T + D and 0 ≤ D < 7C/4 }

var     K, KQ, KQ1, K* : BinaryInteger ;
        B*             : DelayCarryInteger ;
        J              : BitIndex ;
        Overflows      : Integer ;

begin { M }
    K   := 2^N - C ;
    KQ  := Shift(K,Q) ;
    KQ1 := Shift(K,Q+1) ;
    D   := 0 ;
    T   := 0 ;
```

```
    for J := N+Q downto 1 do
    begin
        {1: Top4Bits(D) + Top4Bits(KQ1) ≤ 19             }
        {2: Shift(A,Q-J)*2*B  =  Shift(T,-J)*2^(Q+1)*C + D}
        {3: If J ≤ Q then 2^(Q+1-J) divides D[1] and
                          2^(Q+4-J) divides D[2]         }

        if J-Q > 0 then
            B*  := A[1,J-Q-1]*B + A[2,J-Q])*Shift(B,1)
        else
            B*  :=  0 ;
        T[2,J]  := (Top4Bits(D)+Top4Bits(KQ1) >= 16) ;
        T[1,J-1]:= (not T[2,J]) and
                   (Top4Bits(D)+Top4Bits(KQ)  >=  8) ;
        K*      := T[1,J-1]*KQ  +  T[2,J]*KQ1 ;
        Add(D,B*,K*,D,Overflows) ;
        D  := Shift(D,1)
    end  ;
    D  := Shift(D,-Q-1)
end ; { M }
```

2.2 Remarks on the Hardware

As the data invariant clearly holds for A and T, the calculations of B^* and K^* involve a choice rather than an addition, and so only one clock cycle is needed for each iteration of the loop. If we neglect the time spent to initialise the parameters, during which the first five assignments can also be made, or to emit the result, during which the last shift can be done, the algorithm performs the work in just $N{+}Q$ cycles. The least time is achieved with the minimal value of $Q = 5$, as opposed to Brickell's unexplained choice of $Q = 10$.

The topmost nonzero bit of B^* will have index at most N and its top Q bits are therefore all 0. Feeding this property into the hardware allows the top parts of the second and fourth half adders to be removed, as in Figure 2, because they no longer perform a non-trivial function. The naming of the output bits in Brickell's Figure 3 suggests that the top part of the fifth half adder might also be removed to leave only two half adders. That this is not the case in general is clear from trying to patch together the top section with the middle one. This is the possible source of error in the hardware referred to in the abstract. The lower end of the adder can also be simplified in an obvious way using the information given in Lemma 4 and the data invariant restriction on the $[2, 0]$ bit.

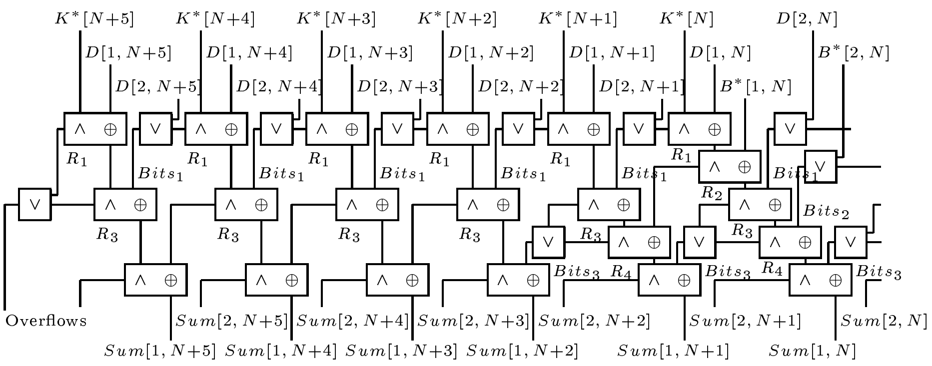

Fig. 2. Special delayed carry adder: Upper end for $Q = 5$.

2.3 The First Loop Invariant

The crux of our proof of the algorithm is the verification of the loop invariants. A little preparation is necessary; to study the flow of bits up the adder it is convenient to start with a definition, illustrated in Figure 1.

Definition. Using the notation of the code for *Add*,

i) let $Or[I]$ denote $Bits_1[I]{+}Bits_2[I]{+}Bits_3[I]$;

ii) let $In[I]$ denote $2{\times}D[2,I{+}1] + D[1,I] + 2{\times}B^*[2,I{+}1] + B^*[1,I] + K^*[I]$;

iii) let $Out[I]$ denote $2{\times}Sum[2,I{+}1] + Sum[1,I]$.

These quantities relate to the input and output from the single cascade of five components of the half adders representing 2^I, taking into account the natural pairing of bits under the data invariant. The bits are weighted to give 1 for an index I, and 2 for an index $I{+}1$. Equating input and output bits clearly yields the first two of the following lemmas:

Lemma 10 With the above notation,

$$In[I]{+}Or[I] = 2{\times}Or[I{+}1] + Out[I]$$

Lemma 11 Execution of the code of *Add* makes its parameters satisfy

$$\begin{aligned} &Top4Bits(D) + Top4Bits(B^*) + Top4Bits(K^*) + Or[N{+}Q{-}3] \\ &\quad = \quad Top4Bits(Sum) + Overflows{\times}2^4 \end{aligned}$$

Lemma 12 The call of *Add* in M satisfies

i) If $Q \geq 4$ then $2Or[N{+}Q{-}3] + Out[N{+}Q{-}4] \leq 6 + K^*[N{+}Q{-}4]$;
ii) If $Q \geq 5$ then $2Or[N{+}Q{-}3] + Out[N{+}Q{-}4] \leq 5 + K^*[N{+}Q{-}4]$;
iii) If $Q \geq 5$ and $K^* = 0$ then $2Or[N{+}Q{-}3] + Out[N{+}Q{-}4] \leq 4$;
iv) If $Q \geq 6$ then $2Or[N{+}Q{-}3] + Out[N{+}Q{-}4] \leq 4 + K^*[N{+}Q{-}4]$;
v) If $Q \geq 7$ and $K^* = 0$ then $2Or[N{+}Q{-}3] + Out[N{+}Q{-}4] \leq 3$.

Lemma 10 allows the properties claimed in Lemma 12 to be translated into upper bounds on $In[N{+}Q{-}4]{+}Or[N{+}Q{-}4]$. In the first case, from the definition of In, $B^*[2, N{+}Q{-}3] = 0$ gives $In[N{+}Q{-}4] \leq 3 + K^*[N{+}Q{-}4]$, and $Or[I] \leq 3$ completes the proof. In subsequent cases, $B^*[2, N{+}Q{-}3] = B^*[1, N{+}Q{-}4] = 0$ gives $In[N{+}Q{-}4] \leq 2{+}K^*[N{+}Q{-}4]$, and the proofs are completed by bounding $Or[N{+}Q{-}4]$. In the second case, as for (i), $Or[I] \leq 3$ suffices. In the third case it remains to obtain $Or[N{+}Q{-}4] \leq 2$. This last is apparent from the hypothesis $K^* = 0$, $B^*[2, N{+}Q{-}4] = 0$ and

$$\begin{aligned}
& Bits_1[N{+}Q{-}4] + Bits_2[N{+}Q{-}4] \\
&= D[2, N{+}Q{-}4] + R_1[2, N{+}Q{-}4] + B^*[2, N{+}Q{-}4] + R_2[2, N{+}Q{-}4] \\
&= D[2, N{+}Q{-}4] + K^*[N{+}Q{-}5] \times D[1, N{+}Q{-}5] \\
&\qquad + (K^*[N{+}Q{-}5] \neq D[1, N{+}Q{-}5]) \times B^*[1, N{+}Q{-}5] \\
&= D[2, N{+}Q{-}4] + D[1, N{+}Q{-}5] \times B^*[1, N{+}Q{-}5] \\
&\leq D[2, N{+}Q{-}4] + D[1, N{+}Q{-}5] \\
&\leq 1
\end{aligned}$$

The last two cases are concluded by showing $Or[N{+}Q{-}4] \leq 2$ and $Or[N{+}Q{-}4] \leq 1$ respectively. They are left as exercises for the reader and involve showing $Bits_2[N{+}Q{-}4] = 0$ for both cases and that $Bits_3[N{+}Q{-}4]$ and $Bits_1[N{+}Q{-}4]$ are not both 1 in the last case. The next lemma is also an exercise.

Lemma 13 If $I < N{+}Q$ then $Top4Bits(KQ1) \leq 8 - K^*[I]$.

In the first loop invariant the aim is to bound the value which the partial product D may have: it should satisfy

$$\mathbf{1} : \; Top4Bits(D) + Top4Bits(KQ1) \leq 19 \,.$$

This is clearly the case when the loop is entered because $D = 0$, and the restriction $C \geq 2^{N-1}$ forces $Top4Bits(KQ1) \leq 8$. An inductive argument is used to deduce that (1) holds at the start of a general iteration. We make essential use of the consequence $Top4Bits(B^*) = 0$ of choosing $Q \geq 4$. This leads to a simplification in Lemma 9 which forms the basis of the proof. Note that Gibson [2] achieves an essentially similar loop invariant by his normalisation process which produces numbers for which the invariant cannot be false.

The induction step in the proof is divided into three cases according to the values of $T[2, J]$ and $T[1, J{-}1]$. Initially, suppose that $T[2, J]$ holds. This with the induction hypothesis (1) provides

$$16 \leq Top4Bits(D) + Top4Bits(KQ1) \leq 19.$$

By Lemmas 9(a) and 11 with $Top4Bits(B^*) = 0$, we have, after the call which adds D, $KQ1$ and B^*, that $Top4Bits(D) \leq 19 + Or[N{+}Q{-}3] - 16d$ for $d = 1$ or 0 according as *Add* overflows or not respectively. Using

$$Top5Bits(Sum) = 2{\times}Top4Bits(Sum) + Out[N{+}Q{-}4],$$

which is immediate from the definition of Out, we have

$$Top5Bits(D) \leq 2{\times}(19{+}Or[N{+}Q{-}3]{-}16d) + Out[N{+}Q{-}4].$$

By Lemma 9(a) *Shift* overflows if and only if $d = 0$. So, after the *Shift* call,

$$\begin{aligned} Top4Bits(D) &\leq 2{\times}(19{+}Or[N{+}Q{-}3]{-}16d) + Out[N{+}Q{-}4] - 2{\times}16(1{-}d) \\ &= 6 + 2{\times}Or[N{+}Q{-}3] + Out[N{+}Q{-}4]. \end{aligned}$$

By Lemma 12(ii), $Top4Bits(D) \leq 6{+}5{+}K^*[N{+}Q{-}4]$ at the end of the iteration, and this with Lemma 13 yields

$$\begin{aligned} Top4Bits(D) + Top4Bits(KQ1) &\leq 11 + Top4Bits(KQ1) + K^*[N{+}Q{-}4] \\ &\leq 11 + 8 \\ &= 19 \end{aligned}$$

for the start of the next iteration.

The second case is when $T[2, J]$ is false, but $T[1, J{-}1]$ holds. Now

$$8 \leq Top4Bits(D) + Top4Bits(KQ) \leq Top4Bits(D) + Top4Bits(KQ1) \leq 15$$

before the *Add* call, which adds D, B^*, and $K^* = KQ$. Lemmas 9(b) and 11 make it possible to deduce that after the *Add* call D satisfies

$$Top4Bits(D) - Top4Bits(KQ) + Top4Bits(KQ1) \quad \leq \quad 15 + Or[N{+}Q{-}3].$$

Just as in the previous case,

$$\begin{aligned} Top5Bits(D) \leq 2{\times}(&Top4Bits(KQ) - Top4Bits(KQ1) \\ &+15 + Or[N{+}Q{-}3]) + Out[N{+}Q{-}4], \end{aligned}$$

and the use of Lemma 12(ii) produces

$$Top5Bits(D) \leq 30{+}2{\times}(Top3Bits(KQ1){-}Top4Bits(KQ1)){+}5{+}K^*[N{+}Q{-}4].$$

After the *Shift*, which by Lemma 9(b) with our initial hypothesis must overflow once,

$$\begin{aligned} &Top4Bits(D) \\ &\leq \quad 30 - 16 + 5 + 2{\times}(Top3Bits(KQ1){-}Top4Bits(KQ1)) + KQ1[N{+}Q{-}3] \end{aligned}$$

Thus,

$$\begin{aligned} &Top4Bits(D) + Top4Bits(KQ1) \\ &\quad \leq \quad 19 + 2{\times}Top3Bits(KQ1) - Top4Bits(KQ1) + KQ1[N{+}Q{-}3] \\ &\quad = \quad 19 \end{aligned}$$

at the start of the next iteration.

If $T[2, J]$ and $T[1, J{-}1]$ are both false, then $Top4Bits(D){+}Top4Bits(KQ) \leq 7$ for the initial value of D and $K^* = 0$. Lemma 9(c) applies and states that neither of the calls to *Add* nor *Shift* will overflow. So, using Lemma 11 as before, after the *Add* call

$$Top4Bits(D) + Top4Bits(KQ) \le 7 + Or[N{+}Q{-}3],$$

which gives

$$Top5Bits(D) \le 2{\times}(7 - Top4Bits(KQ) + Or[N{+}Q{-}3]) + Out[N{+}Q{-}4].$$

Directly Lemma 12(iii) yields

$$Top5Bits(D) \le 14 - 2{\times}Top4Bits(KQ) + 4.$$

After the *Shift* therefore

$$Top4Bits(D) \le 18 - 2{\times}Top3Bits(KQ1).$$

Hence the next iteration starts with

$$\begin{aligned} Top4Bits(D) + Top4Bits(KQ1) &\le 18 + Top4Bits(KQ1) - 2{\times}Top3Bits(KQ1) \\ &= 18 + KQ1[N{+}Q{-}3] \\ &\le 19, \end{aligned}$$

as required.

So the first loop invariant holds by induction before and after each iteration of the loop.

2.4 The Second Loop Invariant

The second loop invariant in the code for procedure M relates the partial quotient T and intermediate residue D to the part of the product $A{\times}B$ computed so far.

$$\mathbf{2}:\ Shift(A, Q{-}J){\times}2{\times}B = Shift(T, -J){\times}2^{Q+1}{\times}C + D\ .$$

This holds when the loop is entered as the right side of the equality is zero, and shifting A down N bits removes every non-trivial bit so that the left side also is zero.

Now suppose the invariant holds at the beginning of iteration J. From the first loop invariant and Lemma 9(a), when $T[2,J] = 1$ either *Add* has an overflow of 2^{N+Q+1} or there are two overflows from the subsequent *Shift*. Similarly, when $T[1, J{-}1] = 1$, the *Shift* has exactly one overflow. So, assuming bits of A with negative indices are zero, at the end of that iteration,

$$\begin{aligned} D_{\rm out} &= 2{\times}(D_{\rm in}{+}B^*{+}K^*) - T[2,J]{\times}2^{N+Q+2} - T[1,J{-}1]{\times}2^{N+Q+1} \\ &= 2{\times}(Shift(A, Q{-}J){\times}2{\times}B - Shift(T,-J){\times}2^{Q+1}{\times}C \\ &\qquad +A[1, J{-}Q{-}1]{\times}B + A[2, J{-}Q]{\times}(2{\times}B) \\ &\qquad +T[1, J{-}1]{\times}KQ + T[2,J]{\times}KQ1) \\ &\qquad -T[2,J]{\times}2^{N+Q+2} - T[1, J{-}1]{\times}2^{N+Q+1} \\ &= 2{\times}(2{\times}Shift(A, Q{-}J) + A[1, J{-}Q{-}1] + 2{\times}A[2, J{-}Q]){\times}B \\ &\qquad -(2{\times}Shift(T, -J) + T[1, J{-}1] + 2{\times}T[2,J]) \times 2^{Q+1} \times C \\ &= 2{\times}Shift(A, Q{+}1{-}J){\times}B - Shift(T, 1{-}J) \times 2^{Q+1} \times C \end{aligned}$$

This is clearly the loop invariant again, with $J-1$ in place of J. Hence decrementing J to start the next loop iteration will establish the loop invariant correctly again. Thus by induction, the loop invariant will always hold.

2.5 The Third Loop Invariant

The last of the three loop invariants, namely

$$\mathbf{3:} \text{ If } J \leq Q \text{ then } 2^{Q+1-J} \text{ divides } D[1] \text{ and } 2^{Q+4-J} \text{ divides } D[2],$$

holds for $J = Q$ by virtue of the case $L = 0$ of Lemma 4 and the *Shift* at the end of the previous iteration. In general, suppose it holds at the start of an iteration with $J \leq Q$. Clearly $B^* = 0$ and 2^Q divides K^*. So all three [1]-component inputs to *Add* are divisible by 2^{Q+1-J} and both the [2]-component inputs are divisible by 2^{Q+4-J}. Now Lemma 4 forces the outputs $D[1]$ and $D[2]$ of *Add* also to be divisible by 2^{Q+1-J} and 2^{Q+4-J} respectively. The *Shift* of the iteration then adds 1 to the exponent of those powers of 2 dividing each component of D, yielding the loop invariant for the start of the next iteration.

2.6 The Post-Condition

For $Q \geq 5$ each of the three parts of the loop invariant has been established also to hold at the end of the last iteration. Thus we may set $J = 0$ to obtain a post-condition for the loop. With appropriate simplifications, this becomes:

1: $Top4Bits(D) + Top4Bits(KQ1) \leq 19$
2: $A{\times}B{\times}2^{Q+1} = T{\times}2^{Q+1}{\times}C + D$
3: 2^{Q+1} divides $D[1]$ and 2^{Q+4} divides $D[2]$.

Because of the third assertion, it is possible to divide D by 2^{Q+1} using *Shift* without losing significant bits. Hence, after the final *Shift* down of D by $Q{+}1$ places, (2) yields

$$A{\times}B = C{\times}T + D$$

which is most of the claimed post-condition.

By using the obvious bounds on the least significant bits of D and $KQ1$, the first assertion yields $(D-2^{N+Q-2}+2)+(KQ1-2^{N+Q-3}+1) \leq 19{\times}2^{N+Q-3}$ at the end of the loop, and therefore $D-2^{N-3} + 2^N-C-2^{N-4} < 19{\times}2^{N-4}$ at the end of the algorithm, i.e. $D < C+6{\times}2^{N-4}$. By making use of $2^{N-1} \leq C$ one obtains the required post-condition easily from $6{\times}2^{N-4} \leq 3C/4$. Clearly, the stronger post-condition $D < C+6{\times}2^{N-4}$ might be preferable in some circumstances.

3 Different Values of Q

Firstly, observe that with the example $N = 8$, $Q = 4$, $C = 129$, $A[1]/A[2] = 64/126$, and $B[1]/B[2] = 160/190$, the wrong answer is obtained as Q is too small.

To obtain bounds on the incorrect behaviour of the algorithm for small values of Q, note that if the algorithm fails for $Q = Q_0$ then it also fails for all $Q < Q_0$. To see this, suppose (N, Q_0, A, B, C) is a failing instance for finding $A \times B \bmod C$ with $Q = Q_0$. Then $(N+(Q_0-Q), Q, A\times 2^{Q_0-Q}, B, C\times 2^{Q_0-Q})$ is a failing instance for $Q \le Q_0$ because exactly the same bits are processed in both cases during the loop of the modular multiplier, with the resulting residue D being shifted fewer times to give 2^{Q_0-Q} times what it was before. Furthermore, if the algorithm fails for $N = N_0$ then it also fails for all $N \ge N_0$. Thus, if (N_0, Q, A, B, C) is a failing instance, then so is $(N, Q, A, B\times 2^{N-N_0}, C\times 2^{N-N_0})$ for the same reasons as before: the same bits are processed after an initial $N-N_0$ loop iterations in which D remains 0, but with an additional $N-N_0$ zero bits at the lower end of all the registers. Hence,

Theorem 14. If the algorithm is incorrect for the pair (N_0, Q_0) of values of (N, Q) then it is incorrect for all pairs (N, Q) with $N \ge N_0$ and $Q \le Q_0$. It is incorrect for $(N, Q) = (8, 4)$.

The benefits of larger values of Q are not substantial. Taking $Q \ge 7$ eliminates completely the effect of the most significant bits of B^* on the overflows during one iteration of the loop. Repeating the proof of Section 2 using Lemma 12 (iv,v) for such Q provides:

Theorem 15 If $Q \ge 7$ then the algorithm works as stated but with 18 in place of 19 in the first loop invariant, and $D < 13C/8$ in the post-condition (or $D < C+5\times 2^{N-4}$).

This slightly increases the likelihood of the residue being less than C and therefore makes it more likely to find $D < C$ by checking just the top few bits (instead of accumulating the carries). However, the advantage in those few cases may not outweigh the extra loop cycles involved. Gibson [2], by choosing to "normalise" his partial answer D on each iteration achieves the slightly better result $D < 3C/2$, but the cost seems to be a longer clock cycle time to incorporate the normalisation.

4 Conclusions

Our results have already been summarised in the abstract at the start of the paper. Several omissions which could lead to errors are indicated, illustrating the care with which it is necessary to check such algorithms for the completeness and correctness of definitions, specifications, and implementations. In particular, note that our definition of "$Top4Bits$" (see Section 1.4 before Lemma 12) is not the one which Brickell may seem to choose. Indeed, care has to be taken over whether "overflow" applies to a binary or delay carry representation. The concept of the algorithm is simple enough but its proof is complicated by having so many bits coming up from lower indices which tighten the inequalities that must be established.

P. Riess and J. Shawe-Taylor [3] remark that all relevant input to Brickell's algorithm had been tested and found to be correct. This pre-supposes certain details and assumptions which we failed to find explicitly in the original paper [1], but have now supplied.

The work described here was implemented in hardware in early 1988 and has now run without fault in systems for many months. Our completion of the details of Brickell's work differs from that described by Gibson [2] in several ways. The most important of these is that we preferred to keep the repetitive nature of the adding hardware so that there was greater flexibility to use it for inputs which did not have N bits. This might enable results to be obtained not just from the top end of the registers. The price of the natural addition is, however, a more complicated proof of correctness, but the benefits are flexibility, simplicity, and probably a shorter cycle time.

References

1. Brickell, Ernest F., *A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography*, Advances in Cryptology (Proceedings of CRYPTO 82) edited by Chaum *et al.*, Plenum, 1983, pp 51-60.
2. Gibson, J. K., *A generalisation of Brickell's algorithm for fast modular multiplication*, BIT **28** (1988) pp 755-763.
3. Riess, P. and Shawe-Taylor, J., *The RSA public key cryptosystem*, Dept. of Stats. & Comp. Sci. Notes, RHNBC, Egham Hill, Egham, Surrey, England, TW20 0EX.
4. Rivest, Ronald L., *RSA Chips (Past / Present / Future)*, Advances in Cryptology, (Proceedings of EUROCRYPT 84), LNCS **209**, Springer Verlag, 1985, pp 159-165.
5. Rivest, R.L., Shamir, A. and Adleman, L., *A method of obtaining digital signatures and public key cryptosystems*, Comm. ACM, **21** (1978), pp.120-126.