Montgomery Multiplication

By Henry S. Warren, Jr.

July 2012

This note explains the theory and practice of Montgomery multiplication.

Montgomery multiplication is a method for computing $ab \bmod m$ for positive integers *a*, *b*, and *m*.[1] It reduces execution time on a computer when there are a large number of multiplications to be done with the same modulus *m*, and with a small number of multipliers. In particular, it is useful for computing $a^n \bmod m$ for a large value of *n*. The number of multiplications modulo *m* in such a computation can be reduced to a number substantially less than *n* by successively squaring and multiplying according to the pattern of the bits in the binary expression for *n* ("binary decomposition"). But it can still be a large enough number to be worthwhile speeding up if possible. The difficulty is in the reductions modulo *m*, which are, essentially, division operations, which are costly in execution time. If one defers the modulus operation to the end, then the products will grow to very large numbers, which slows down the multiplications and also the final modulus operation.

To use Montgomery multiplication, we must have the multipliers *a* and *b* less than the modulus *m*. We introduce another integer *r* which must be greater than *m*, and we must have gcd(*r*, *m*) = 1. The method, essentially, changes the reduction modulo *m* to a reduction modulo *r*. Usually *r* is chosen to be an integral power of 2, so the reduction modulo *r* is simply a masking operation; that is, retaining the lg(*r*) low-order bits of an intermediate result, and discarding higher order bits. If *r* is a power of 2, we must have *m* odd, to satisfy the gcd requirement. (Any odd value from 3 to $r-1$ is acceptable.)

The method:

1. Find two integers r^{-1} and m' such that

$$rr^{-1} - mm' = 1.$$

   This can be done by the *extended gcd* algorithm. There is a *binary extended gcd* algorithm which does no divisions, and which simplifies substantially when one argument (*r*) is a power of 2 and the other (*m*) is odd. This simplified version of the algorithm is given below (C function xbinGCD).

[1] This note uses "mod" not as an expression of an equivalence relation, but as a binary operator: *a* mod *b* is the remainder upon dividing *a* by *b*.

2. Transform the multipliers to "Montgomery space" by multiplying them by r (a shift left operation if r is a power of 2) and reducing the product modulo m. That is,

$$\overline{a} = ar \bmod m, \text{ and}$$
$$\overline{b} = br \bmod m.$$

These are expensive operations, but they are done only once per multiplier, and they are not done on the intermediate products of a chain of multiplications.

3. Perform the Montgomery multiplication step. This operates on the transformed quantities $\overline{a}$ and $\overline{b}$, giving the product of a and b in Montgomery space. That is, the result is $abr \bmod m$. The direct way to calculate it is $u = abr \bmod m = \overline{a}\overline{b}r^{-1} \bmod m$, but that is expensive (mainly because of the reduction modulo m). The more efficient way to calculate it is

$$\begin{aligned} &t = \overline{a}\overline{b}, \\ &u = (t + (tm' \bmod r)m)/r \\ &\text{if } (u \geq m) \text{ then return } u - m; \text{ else return } u. \end{aligned} \tag{1}$$

This involves three multiplications and some less expensive operations such as addition and shifting right (division by r, assuming r is a power of 2). The multiplication tm' is not too expensive because the mod r implies that only the low-order lg(r) bits of the product need be produced.

If the calculations are performed to some fixed length w bits, with $r = 2^w$, then the other two multiplications are of the form $w \times w \Rightarrow 2w$ bits and the addition is of the form $2w + 2w \Rightarrow 2w + 1$ bits (it can overflow). After division by r (a shift), u is of length $w + 1$ bits.

4. Do the inverse transformation to convert the result to an ordinary integer:

$$ab = ur^{-1} \bmod m.$$

Let us now derive step 3 above. We wish to compute $u = abr \bmod m$.

$$
\begin{aligned}
u &= abr \bmod m = arbrr^{-1} \bmod m = \bar{a}\bar{b}r^{-1} \bmod m \\
&= (\bar{a}\bar{b}rr^{-1} / r) \bmod m \\
&= (\bar{a}\bar{b}(1 + mm') / r) \bmod m && (\text{from } rr^{-1} - mm' = 1) \\
&= ((\bar{a}\bar{b} + \bar{a}\bar{b}mm') / r) \bmod m \\
&= ((\bar{a}\bar{b} + \bar{a}\bar{b}mm') / r + km) \bmod m && \text{for any integer } k \\
&= ((\bar{a}\bar{b} + \bar{a}\bar{b}mm' + kmr) / r) \bmod m \\
&= ((\bar{a}\bar{b} + (\bar{a}\bar{b}m' + kr)m) / r) \bmod m \\
&= ((\bar{a}\bar{b} + (\bar{a}\bar{b}m' \bmod r)m) / r) \bmod m
\end{aligned}
$$

The last line here is the same as the first two lines of (1), except it appears here that we must do a modulo m reduction, and thus haven't accomplished anything. However, it can be shown that the quantity u calculated by the last line above is less than $2m$. Therefore, the modulo m reduction can be done by simply comparing u to m, and if $u \geq m$, subtracting m from u. This accounts for the last line in equations (1).

To see that $(\bar{a}\bar{b} + (\bar{a}\bar{b}m' \bmod r)m) / r < 2m$, notice that $\bar{a}\bar{b}m' \bmod r < r$, and that $\bar{a}, \bar{b} < m$, so that the expression is less than $(m^2 + rm) / r$. Because $r > m$, this is less than $2m$.

A 64-bit Implementation

Here we take a close look at an implementation of Montgomery multiplication for arguments up to the computer's word size. For concreteness we take it to be 64 bits. The modulus m can be as large as $2^{64} - 1$, and a and b can be as large as $m - 1$. We take $r = 2^{64}$. This is a 65-bit number, but it can be handled without great difficulty.

Step 1: The GCD Operation

Below is a C function for the binary extended gcd operation, simplified for the case in which its first argument a is a power of 2 and the second argument b is odd. It is a simplification of the algorithm available on the web at:

http://www.ucl.ac.uk/~ucahcjm/combopt/ext_gcd_python_programs.pdf.

The code is also altered so that the first argument is half of what it "should" be. We'll invoke it with $a = 2^{63}$, and it will treat that argument as if it were 2^{64}. It finds u and v such that $u(2a) - vb = 1$.

```
typedef unsigned long long uint64;
typedef long long int64;

void xbinGCD(uint64 a, uint64 b, uint64 *pu, uint64 *pv)
   {
   uint64 alpha, beta, u, v;

   u = 1; v = 0;
   alpha = a; beta = b;         // Note that alpha is
                                // even and beta is odd.

   /* The invariant maintained from here on is:
   2a = u*2*alpha - v*beta. */

   while (a > 0) {
      a = a >> 1;
      if ((u & 1) == 0) {             // Delete a common
         u = u >> 1; v = v >> 1;      // factor of 2 in
      }                               // u and v.
      else {
         /* We want to set u = (u + beta) >> 1, but
         that can overflow, so we use Dietz's method. */
         u = ((u ^ beta) >> 1) + (u & beta);
         v = (v >> 1) + alpha;
      }
   }

   *pu = u;
   *pv = v;
   return;
}
```

Step 2: Transform the Multipliers

We must compute $\bar{a} = ar \bmod m$, and similarly for $\bar{b}$. Because $r = 2^{64}$, there is no multiplication to do. We must form a 128-bit integer that consists of a followed by 64 0-bits, and compute the remainder of division of that quantity by m. Some machines have an instruction for that. For other machines, the C function shown below may be used. This is the “hardware division” algorithm of Hacker’s Delight. Invoke it as follows, where the first two arguments represent ar. All variables are 64-bit unsigned integers.

```
abar = modul64(a, 0, m);
```

```
uint64 modul64(uint64 x, uint64 y, uint64 z) {

   /* Divides (x || y) by z, for 64-bit integers x, y,
   and z, giving the remainder (modulus) as the result.
   Must have x < z (to get a 64-bit result). This is
   checked for. */

   int64 i, t;

   if (x >= z) {
      printf("Bad call to modul64, must have x < z.");
      exit(1);
   }
   for (i = 1; i <= 64; i++) {  // Do 64 times.
      t = (int64)x >> 63;       // All 1's if x(63) = 1.
      x = (x << 1) | (y >> 63); // Shift x || y left
      y = y << 1;               // one bit.
      if ((x | t) >= z) {
         x = x - z;
         y = y + 1;
      }
   }
   return x;                    // Quotient is y.
}
```

Step 3: Montgomery Multiplication

This step deals with 128-bit integers, but no more than that. The computation $t = \overline{a}\overline{b}$ is multiplying two 64-bit unsigned integers, giving a 128-bit product. Some machines have an instruction for that. For other machines, the C function below may be used.

```
void mulul64(uint64 u, uint64 v, uint64 *whi, uint64 *wlo)
   {
   uint64 u0, u1, v0, v1, k, t;
   uint64 w0, w1, w2;

   u1 = u >> 32; u0 = u & 0xFFFFFFFF;
   v1 = v >> 32; v0 = v & 0xFFFFFFFF;

   t = u0*v0;
   w0 = t & 0xFFFFFFFF;
   k = t >> 32;

   t = u1*v0 + k;
   w1 = t & 0xFFFFFFFF;
```

```
   w2 = t >> 32;

   t = u0*v1 + w1;
   k = t >> 32;

   *wlo = (t << 32) + w0;
   *whi = u1*v1 + w2 + k;

   return;
}
```

Next, the following expression must be evaluated:

$$u = (t + (tm' \bmod r)m) / r.$$

Variable t is a 128-bit unsigned integer, and m' is a 64-bit unsigned integer. Because of the "mod r," only the low-order 64 bits of the product tm' is needed. This means that the high-order half of t can be ignored, and $64 \times 64 \Rightarrow$ 64-bit multiplication can be used. The subsequent multiplication by m must be $64 \times 64 \Rightarrow$ 128-bit multiplication. The addition of t must be $128 + 128 \Rightarrow$ 129-bit addition. This can be done with $128 + 128 \Rightarrow$ 128-bit addition and separately computing the carry, as shown in the code below (variable `ov`).

This sum always ends in 64 0-bits, so the low-order part of the sum is computed only to produce a carry bit. Incidentally, if the low-order halves of the summands were known to be both nonzero, then the carry would be 1, resulting in a simplification. However, the summands can be 0 if either a or b is 0.

Lastly (for step 3), we must perform the computation:

$$\text{if } (u \geq m) \text{ then return } u - m; \text{ else return } u.$$

Variable u is a 65-bit integer, in effect, because of the overflow mentioned above. But the final result of the calculation is a 64-bit integer. If the addition of t overflowed, then certainly $u > m$. Otherwise, u and m may be compared as 64-bit integers. The subtraction can be a 64-bit operation, because it is known that after the subtraction, the 65^{th} bit of the difference will be 0.

A C function for these computations follows.

```
uint64 montmul(uint64 abar, uint64 bbar, uint64 m,
               uint64 mprime) {

   uint64 thi, tlo, tm, tmmhi, tmmlo, uhi, ulo, ov;

   mulul64(abar, bbar, &thi, &tlo);  // t = abar*bbar.

   /* Now compute u = (t + ((t*mprime) & mask)*m) >> 64.
```

```
   The mask is fixed at 2**64-1. Because it is a 64-bit
   quantity, it suffices to compute the low-order 64
   bits of t*mprime, which means we can ignore thi. */

   tm = tlo*mprime;

   mulul64(tm, m, &tmmhi, &tmmlo);   // tmm = tm*m.

   ulo = tlo + tmmlo;                // Add t to tmm
   uhi = thi + tmmhi;                // (128-bit add).
   if (ulo < tlo) uhi = uhi + 1;     // Allow for a carry.

   // The above addition can overflow. Detect that here.

   ov = (uhi < thi) | ((uhi == thi) & (ulo < tlo));

   ulo = uhi;                   // Shift u right
   uhi = 0;                     // 64 bit positions.

   if (ov > 0 || ulo >= m)      // If u >= m,
      ulo = ulo - m;            // subtract m from u.

   return ulo;
}
```

The conditional subtraction of `m` may be done as follows, which avoids two branches if the compiled code evaluates the predicate `ulow >= m` without a branch.

```
   ulo = ulo - (m & -(ov | (ulo >= m)));
```

Step 4: The Inverse Transformation

We must compute $ur^{-1} \bmod m$, which is the product of a and b modulo m as ordinary integers. All variables are 64-bit unsigned integers. The multiplication must be done using $64 \times 64 \Rightarrow$ 128-bit multiplication, and the modulo operation must be done using $128 / 64 \Rightarrow$ 64-bit division (actually remaindering). Using the functions given above, in C this can be done by

```
   mulul64(p, rinv, &phi, &plo);
   p = modul64(phi, plo, m);
```

where `p` is the result of the Montgomery multiplication step, and also the final result `a*b`.

Reference

[PM] Montgomery, Peter L. “Modular Multiplication Without Trial Division.” *Mathematics of Computation 4A*, 170 (April 1985), 519-521. Available at http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf