

Special applications of 64-bit arithmetic: Acceleration on the Apple G5*

S. Noble[†] and J. Papadopoulos[‡]

26 May 2006

Abstract: We exhibit fundamental number-theoretical routines optimized for Apple's 64-bit PowerPC G5 platform. Additionally we provide performance comparisons against 32-bit scenarios. Special attention is given to widely used core operations such as $a \cdot b \bmod c$, which often support sophisticated multiprecision software.

*Although the techniques described here are indeed optimized for the G5, most of the techniques herein can be applied to any other architecture having 64-bit integer arithmetic.

[†]Advanced Computation Group, Apple Computer

[‡]Apple Cooperative Researcher

1 Motivation

The arrival of 64-bit processors in commodity hardware, beginning with the Apple G5, provides opportunities for accelerating numerous mathematical applications. A great deal has been written on the topic of 64-bit transitions, but for the most part the discussion has been constrained primarily to issues involving safely recompiling existing programs in a 64-bit environment or taking advantage of ever larger quantities of RAM. There are, in fact, cases where larger integer registers provide significant direct computational benefits, and we address such issues herein. In this sense, the present document stands as a kind of complement to essays, descriptions, and documentation in regard to memory.

Historically, obtaining results for operations on 64-bit quantities (with general purpose 32-bit processors) has required the use of multi-precision arithmetic on 16-bit chunks of the larger 64-bit operands. One might point out that in some instances the impact of the extra work can be partially mitigated through the use of special vector processing instructions, like those found in the PowerPC Velocity Engine [7]. On the G5, however, we can do better for several important special cases by taking advantage of new 64-bit integer instructions not available on the the G4 or any other 32-bit platform.

Multiplication, as an inherently $O(n^2)$ operation for n -bit operands in “grammar-school” mode, sees the greatest advantage from increased register size. On 32-bit platforms, like the Apple G4, a 64-bit-by-64-bit multiplication requires 4 full 32-bit-by-32-bit multiplications which translates into 8 multiplication instructions on the G4’s 32-bit PowerPC processor. Once the smaller multiplications have been performed, eight more addition instructions are necessary to complete the calculation. Figure 1 shows how the G4 computes the product of two 64-bit integers. On a G5, not only can the operation be performed in one full 64-bit-by-64-bit operation (two instructions on the G5), but the two integer processing units on the 64-bit PowerPC 970 can be used in concert to compute the low and high¹ halves of the multiplication simultaneously.

Let $a, b \in [0, 2^{64} - 1]$, $W = \lfloor a/2^{32} \rfloor$, $X = a \bmod 2^{32}$, $Y = \lfloor b/2^{32} \rfloor$, $Z = b \bmod 2^{32}$

G4	W	X
\times	Y	Z
<hr/>		
	HI ($Z \cdot X$)	LO ($Z \cdot X$)
	HI ($Z \cdot W$)	LO ($Z \cdot W$)
	HI ($Y \cdot X$)	LO ($Y \cdot X$)
$+$	HI ($Y \cdot W$)	LO ($Y \cdot W$)
<hr/>		

Figure 1: The G4 performing a full 64-bit multiplication

¹For $r = a \cdot b$, the $2k$ -bit result of a k -bit-by- k -bit multiplication, we say that the *low* half of the multiplication is $r \bmod 2^k$, and the *high* half is $\lfloor \frac{r}{2^k} \rfloor$

2 Conventions and caveats

C source code has been generated by **gcc** 4 as included in MacOS X 10.4. Experiments have been run on MacOS X 10.4 on a 2 GHz G5 at the “Highest Performance” *Energy Saver* Setting. Failure to adjust the *Energy Saver* settings may result in a substantial detriment to the performance of many of these routines, simply because they are so short—longer programs will cause the system to automatically make this transition. Where indicated, relevant experiments were also run on a 1.5 GHz G4 with a PPC 7447A processor.

Compilation

A special note is in order regarding compilation of 64-bit functions – by default, **gcc** will generate 32-bit code on many platforms. For 32-bit and 64-bit compilation respectively, our experiments utilized the following compiler flags:²

```
gcc -fast -m32 -c <filename>.c
gcc -fast -m64 -c <filename>.c
```

Another point to keep in mind – when moving from different operating environments, the size of intrinsic datatypes can change. To combat this, we make use of the types defined in **stdint.h**, a part of any C99-complaint compiler implementation. **stdint.h** defines **uint64_t**, which is guaranteed to be 64 bits wide on any platform.

High-resolution timing

All performance optimization is an iterative process; one should verify by direct experiment that performance-sensitive code runs as fast as expected. This in turn requires finding out, to accuracy as high as possible, the amount of runtime taken by a code snippet. Measuring execution runtime is not the main goal of this paper; nonetheless, we’ll need a simple tool that allows low-overhead measurement of runtime.

One option that has the advantage of portability across different unix implementations is the **gettimeofday()** function. This returns a value that changes every microsecond, and is suitable for coarse runtime measurements. A better alternative on PowerPC is the **mftb** (move from time base) instruction, which reads the value of the PowerPC time base register. The value in this register increments at a rate that is processor-specific. On the target G5 workstation, an **mftb** based timing routine is about thirty three times more precise than **gettimeofday()**. Given that the target platform is MacOS X, there is a third option available. The **mach_absolute_time()** function is guaranteed to provide the most accurate method of measuring time on any Macintosh (PowerPC or Intel).

```
#include <mach/mach_time.h>
double currentTime(void)
{
```

²for 32-bit applications that want to use 64-bit instructions, one may use `gcc -fast -mpowerpc64 -c <filename>.c`

```

static double scale = 0.0;
if (0.0 == scale) {
    mach_timebase_info_data_t info;
    mach_timebase_info(&info);
    scale = info.numer / info.denom * 1e-9;
}
return mach_absolute_time() * scale;
}

```

High word of 128-bit products

In general, the product of two 64-bit integers requires 128 bits. There is no standard “C” intrinsic to retrieve the high half of the result – in general, the low half can be retrieved by the standard `long long` multiplication operator. As of the current release, there is not even a **gcc** intrinsic to get at the high 128 bits. Using inline assembly, the following (**gcc**-specific) insertion suffices to retrieve the high half of a full 64-bit multiplication on the PowerPC.

```

#include <stdint.h>
static inline uint64_t __mulhdu (uint64_t a, uint64_t b) __attribute__((always_inline));

static inline uint64_t __mulhdu (uint64_t a, uint64_t b)
{
    uint64_t result;
    __asm__ ("mulhdu %0, %1, %2"
            /* outputs: */ : "=r" (result)
            /* inputs: */ : "r" (a), "r"(b));
    return result;
}

```

3 An important operation: $a \cdot y + b$

Let $y \in [0, 2^{64} - 1]$, $a \in [0, 2^{64} - 1]^n$, and $b \in [0, 2^{64} - 1]^{n+1}$. We wish to compute $a \cdot y + b$.

Motivation

Consider an alternate formulation of the problem. Given a multiple precision number expressed as an array of (64-bit) machine words, the task is to add the product of another such multiple-precision quantity and a 1-word multiplier. This **addmul** primitive is the key inner loop for multiple precision multiplication, division, and exponentiation operations, and is used for cryptographic primitives such as RSA, Diffie-Hellman, and DSA, for prime number generation, primality testing, and integer factorization. In particular, E-commerce literally reduces to **addmul** operations! We note that different microprocessors can achieve vastly different **addmul** speeds. In this section, we will show that the G5 is no slouch.

Schematically, an `addmul` operation $b = a \cdot y + b$ can be expressed as follows:

$$\begin{array}{cccccc}
 & \text{LO}(y \cdot a_0) & \text{HI}(y \cdot a_0) & & & \\
 & & \text{LO}(y \cdot a_1) & \text{HI}(y \cdot a_1) & & \\
 & & & \text{LO}(y \cdot a_2) & \text{HI}(y \cdot a_2) & \\
 & & & & \text{LO}(y \cdot a_3) & \text{HI}(y \cdot a_3) \quad \dots \\
 + & b_0 & b_1 & b_2 & b_3 & b_4 \quad \dots \\
 \hline
 & b_0 & b_1 & b_2 & b_3 & b_4 \quad \dots \quad b_n
 \end{array}$$

$$a_j \in [0, 2^{64} - 1], j < n$$

$$\begin{aligned}
 \text{HI}(y \cdot a_j) &:= \lfloor y \cdot a_j / 2^{64} \rfloor, \\
 \text{LO}(y \cdot a_j) &:= y \cdot a_j \bmod 2^{64}
 \end{aligned}$$

In cryptographic applications, a and b are typically the same size and represent 512-bit or 1024-bit integers. If a and b have n words each, $y \cdot a$ yields an $(n+1)$ -word product; the extra most significant word is usually treated specially. Execution flows from left to right in the diagram above, with carry bits propagated from word to word of b .

Basic scalar implementation

The following routine performs an `addmul` which uses 32-bit or 64-bit arithmetic for, respectively, the G4 or the G5.

```

#if defined(__ppc64__)
typedef uint64_t word_t;
#define MUL(prodhi, prodlo, a, y) \
    prodlo = (a) * (y); \
    asm("mulhdu %0, %1, %2": "=r"(prodhi): "r"(a), "r"(y));
#else
typedef uint32_t word_t;
#define MUL(prodhi, prodlo, a, y) { \
    uint64_t prod = (uint64_t)(a) * (uint64_t)(y); \
    prodhi = (word_t)(prod >> 32); \
    prodlo = (word_t)(prod); \
}
#endif

#define NUM_WORDS (1024 / (8 * sizeof(word_t)))

word_t addmul(word_t *B, word_t *A, word_t Y) {
    word_t lo, hi, carry = 0;
    int i;

    for (i = 0; i < NUM_WORDS; i++) {
        MUL(hi, lo, A[i], Y);
        lo += carry;
    }
}

```

```

        hi += (lo < carry);
        B[i] += lo;
        carry = hi + (B[i] < lo);
    }
    return carry;
}

```

As one might reasonably expect, the basic `addmul` implementation runs about twice as quickly when using 64-bit words instead of 32-bit words. The G5 performance of the 64-bit version was about 287 cycles/`addmul` while the 32-bit version was measured to be about 543 cycles/`addmul`.

Optimized implementation

An optimal `addmul` implementation requires a significant amount of platform-specific knowledge (including comfortability with the architecture’s assembly language) and access to good performance analysis tools. In particular, **simg5**, the 64-bit PowerPC simulator, has proved to be indispensable. Information about the G5’s PowerPC 970 processor can be found in [10]. Some of the information critical to a good `addmul` routine:

1. The PowerPC 970 has two integer execution units. Each can perform a multiply.
2. No other integer instruction may be executing in an integer unit currently performing a multiply.
3. Addition operations finish in two cycles but a throughput of one addition (per unit!) can be achieved per cycle.
4. Multiplication operations finish in 7 cycles but a throughput of one multiply (per unit) can be achieved every 6 cycles.

In addition, taking full advantage of this knowledge requires even more knowledge about the optimization tricks performed by compilers. This kind of low-level optimization, while incredibly instructive, is beyond the scope of this paper. For those interested in the process of finely tuning a routine to run on the G5, we refer you to Apple’s *G5 Performance Programming* web page [3]. Those who might be curious about how the language of mathematics can map to our slightly obscure implementation in PowerPC assembly language are encouraged to follow up.

The highly-optimized 64-bit `addmul` routine can be found in the appendix as **addmul3asm.S**. On the G5, the final `addmul` runs at 8.5 cycles per 64-bit word or 136 clock cycles for the 1024-bit integers used in the previous section.

Altivec

It’s worth pointing out that the `addmul` operation can be implemented via multi-precision arithmetic using vector instructions. Such an implementation has the desirable property of compatibility with 32-bit processors as well as the potential for being very fast. The `vecLib` framework (available on MacOS X) provides optimized primitives (defined in `vBigNum.h`) for operating on 512-bit multi-precision integers.

Given the tremendous ease with which `vecLib` can be used to implement routines requiring multi-precision arithmetic, it is entirely within the scope of this paper to investigate the performance achievable with a straightforward 512x512-bit multiply. Precisely, we compare `vU512FullMultiply()`, as used in **`addmulv2.c`** with the following small routine:

```
typedef uint64_t word_t;
extern word_t addmul(word_t *B, word_t *A, word_t Y, word_t size);
#define NUM_WORDS (512/64)

void kilomul(word_t *prod, word_t *a, word_t *b) {
    int i;
    memset(prod, 0, NUM_WORDS * sizeof(word_t));

    for (i = 0; i < NUM_WORDS; i++)
        prod[NUM_WORDS + i] = addmul(prod + i, a, b[i], NUM_WORDS);
}
```

On our G5, the `addmul` operation developed above allows a 512-bit-by-512-bit multiply to be performed more than twice as fast as the vector implementation using `vecLib`. That's 561 cycles per `kilomul` and 1176 cycles per `vU512FullMultiply()`.

Note that **`addmul3asm.S`** requires that the number of words be a multiple of four. The routine in **`addmul4asm.S`** removes this requirement in exchange for additional overhead which becomes less significant with larger operands. For the 512-bit multiply above of the previous paragraph, the more flexible final general purpose `addmul` implementation requires 711 clock cycles. As operand size increases, the per-word runtime converges to its asymptotic value of about 9 cycles per 64-bit word.

For the sake of completeness, we observe that a general purpose multi-precision multiply using 64-bit words and the `addmul` routine developed above can be easily implemented as follows.

```
void bigmul(word_t *prod, word_t *a, word_t *b, word_t nwords) {
    int i;
    memset(prod, 0, nwords * sizeof(word_t));

    for (i = 0; i < nwords; i++)
        prod[nwords + i] = addmul(prod + i, a, b[i], nwords);
}
```

4 A profoundly important operation: $a \cdot b \bmod c$

Motivation

From cryptography to computational number theory to high-precision algebra to large-integer arithmetic generally, the operation $a \cdot b \bmod c$ is ubiquitous, and often the main time-sink above all other arithmetic in a given software run. For example, in factoring of integers, *all* of the high-power methods use this operation in one guise or another. (Typically, almost all arithmetic proceeds modulo N , with N being the mystery number to be factored.) It is reasonable to wonder how a high-precision modulus relates to 64-bit (or larger) registers. In fact, there are important algorithms that underlie powerful packages, and some of these need operations such as a large-polynomial multiply in an algebraic ring, say Z_c , and that can in turn be done via a so-called binary-segmentation multiply [9], which in turn itself relies on $a \cdot b \bmod c$ heavily.

If $a, b \in [0, 2^{64} - 1]$ then $a \cdot b$ is potentially a 128-bit quantity. The G5 can compute the full 128-bit product of two 64-bit integers, but the processor can not directly divide the (potentially) 128-bit integer result. While any general purpose implementation of multi-precision arithmetic (such as [1, 2]) can perform this operation, the overhead associated is not optimal where arguments are all expressible as 64-bit quantities. As this is a common operation among number-theorists, we endeavor to develop a reasonably efficient routine to perform this computation.

Standard implementation with divide

A standard implementation of a general purpose `modmul` routine will almost certainly follow Knuth in [11]. In fact, division operations on the C `long long` data type frequently use a refined implementation of Knuth's Algorithm D. This routine is freely available and we have made minimal modifications to adapt it to the 128-bit-by-64-bit modulo operation necessary for computing $a \cdot b \bmod c$. Using `qdivrem`, modified from the version of Algorithm D as distributed by the OpenBSD project, to implement the hard part of the `modmul` operation results in a routine which completes in 223 cycles for $a, b \geq 2^{32}$ and $a \cdot b \geq 2^{96}$.

Reciprocal-based approaches

In many important cases where the `modmul` operation is found, the divisor is constant or changes relatively infrequently. Examples include factoring, chinese remainder reconstruction, testing integers for primality, and multiplying very large integers using error-free transform-based convolution methods [9]. The entire `modmul` operation can be sped up significantly if a reciprocal of the form $2^k/c$ (where k is carefully chosen) is computed in advance. In this way one may repeatedly effect a modulo operation without resorting to explicit division each time.

Reciprocal calculation

The following sections make use of precalculated reciprocals. For any of these cases, the reciprocal calculation will dominate the computation time for the operation involved. Clearly these routines are most useful when the divisor is constant across operations and the cost of calculating the reciprocal can be amortized over many modulo operations.

A very simple way to calculate any of these reciprocals (of the form $\lfloor 2^k/c \rfloor$) using vecLib is as follows:

```
#include <vecLib/vBigNum.h>
void reciprocal(uint64_t c, unsigned char k, vU256 *r) {
    vU256 a, b, remainder;

    a.v[0] = a.v[1] = b.v[0] = b.v[1] = (vUInt32)(0);
    a.s.LSW = 1;

    b.s.LSW = (uint32_t)c&0xffffffff;
    b.s.d7 = (uint32_t)(c>>32);

    vLL256Shift( &a, k, &a );
    vU256Divide( &a, &b, r, &remainder );
}
```

For the important cases where $1 < c < ab < \min\{c^2, 2^{64}\}$, an interesting integer-arithmetic reciprocal algorithm free of explicit division can be found in section 9.2 of [9]. This is essentially the same technique used in [7], where it is implemented using the AltiVec instruction set.

Special-purpose modulo routine

Given that computations of $a \cdot b \bmod c$ frequently do not require the full 64-bits available in the G5 registers, we will address a few additional approaches which take advantage of reciprocal pre-calculation. For the following examples, the reciprocal calculated with the `reciprocal()` routine must be called where k is chosen as follows – if c is the divisor, having b bits, then R is defined as $\lfloor 4^{b-1}/c \rfloor$, and k is therefore the quantity $2(b-1)$. Further, R is the least significant 64 (or 32) bits returned by the `reciprocal()` routine.

For instance, let us suppose that $c < a \cdot b < \min\{c^2, 2^{64}\}$. A *generalized reciprocal* R can be stored in the same number of bits as c and the “division” can be performed with Barrett’s technique as described in [9, 8]. Not including the cost of precalculating a reciprocal, the Barrett-based `modmul` can be implemented in approximately 35.7 G5 cycles –

```
uint64_t barrettmodmul64( uint64_t a, uint64_t b, uint64_t c, uint64_t R )
{
    unsigned int s = 2*((64-__builtin_clzll( R ))-1);
    uint64_t dhi, dlo, d, mod, ab = a*b;
    dhi = __mulhdu(ab, R);
    dlo = (uint64_t)ab*R;
    dlo >>= s;
    if (s >= 64) {
        d = dhi >> (s-64);
    } else {
        dhi <<= (64-s);
        d = dlo | dhi;
    }
}
```

```

}
mod = ab - (uint64_t)(d * c);
mod -= (mod >= c)?c:0;
return mod;
}

```

What if the goal is to use 64-bit arithmetic to speed up 32-bit calculations rather than simply using integer arithmetic on *larger* numbers? For the case where $c < a \cdot b < \min\{2^{32}, c^2\}$, the same technique can be performed with fewer instructions than on a 32-bit PowerPC, which requires a software implementation of 64-bit integer arithmetic.

```

uint32_t barrettmul32(uint32_t a, uint32_t b, uint32_t c, uint32_t R) {
    unsigned int s = 2*(32-__builtin_clz( R )-1);
    uint32_t div, mod, ab = a*b;
    div = (uint32_t)(((uint64_t)ab*R) >> s);
    mod = ab - (uint32_t)(div * c);
    mod -= (mod >= c)?c:0;
    return mod;
}

```

Other interesting variations on this technique exist and depend on the unique constraints of any given situation.

Double-precision floating point

If, instead of using integer arithmetic, we attempt to implement a `modmul` routine using the double precision floating point instructions, we can take advantage of the two floating point units in the G5 and produce code which is capable of running on a G4.³ The only major drawback to this approach is that it effectively limits the divisor to the range $[0, 2^{50} - 1]$ and $a, b \in [0, 2^{53} - 1]$.

The secret for implementing the technique described in the preceding paragraph is that the PowerPC standard requires that floating point multiply-add operations have only one rounding error. This means that we can get access to the entire 106-bit product of 53-bit integers using two floating point operations. One floating-point multiply operation computes the upper 53 bits of the product, and then a floating point multiply-subtract computes the 106-bit product again, subtracts off the high half we found before, and converts the low 53 bits into a floating point value. The ability to compute double-size products very quickly leads to a high-speed `modmul` routine:

```

#include <ppc_intrinsics.h>

fast_modmul(double a, double b, double n,
            double recip_n, double magic) {
    double q, r, nq_hi, nq_lo;

```

³Note that the floating point techniques of this section rely on a fused multiply-add instruction which is not present in the processors descended from the x86 line.

```

q = a * b;                /* compute high 53 bits of a * b */
q = __fmadd(q, recip_n, magic); /* multiply by the reciprocal */
q = q - magic;             /* round to nearest integer */

nq_hi = n * q;
nq_lo = __fmsub(n, q, nq_hi); /* nq_hi + nq_lo = n * q exactly */

r = __fmsub(a, b, nq_hi);
r = r - nq_lo;             /* compute a * b - n * q */

q = r + n;
return __fsel(r, r, q);    /* return r, or r+n if r < 0 */
}

```

The constant `magic` = $3 \cdot 2^{51}$. Note that in this case, `recip_n` is computed externally as `recip_n = 1.0/n`

General-purpose 64-bit modulo routine

Again, for the purposes of completeness we present a general purpose routine capable of computing $a \cdot b \bmod c$ for full 64-bit operands a, b, c . Using a 128-bit reciprocal $R = \lfloor 2^{128}/c \rfloor$, the remainder can be calculated in the following manner. The first step is to compute the value, via a 128-by-128-bit multiplication, $d = \lfloor (ab \cdot R) / 2^{128} \rfloor$. This requires seven independent multiply instructions, which execute 2-at-a-time and can complete in six cycles per multiply, and eight addition instructions to handle carries. Next a preliminary value for the remainder is computed by taking the (possibly) 128-bit divisor d , multiplying it by c and subtracting it from ab : $mod = ab - c \cdot d$. Because of the relationship between c and R , this 64-bit (c) by 128-bit (d) multiply is guaranteed to produce a 128-bit-or-less result. This step is comprised of three multiply instructions and one addition. Finally the 128-bit subtraction requires two subtraction instructions (but because one depends on the other, they can not execute in parallel).

Unfortunately, the mod value produced above is not necessarily correct. Because we used an approximation to the reciprocal of c in our multiplication, the calculated remainder can be off by ab .

Let $r = \lfloor 2^{128}/c \rfloor = 2^{128}/c + \theta_0$, for $\theta_0 < 1$, also let $ab \in \{0, \dots, 2^{128} - 1\}$.

So $d = \lfloor (r \cdot ab) / 2^{128} \rfloor = (ab/c) + (ab \cdot \theta_0) / 2^{128} + \theta_1$ for $\theta_1 < 1$. Observe that $ab \cdot \theta_0 < 2^{128}$, so $d \leq ab/c + \theta'_0 + \theta_1 < ab/c + 2$

The result is that $mod = ab - cd$ is either the true remainder or “the true remainder” + ab . Note that all the reciprocal-approximation techniques we discuss in this paper suffer from this same liability.

When carefully tuned, this technique can produce the result of $a \cdot b \bmod c$ in approximately 61 cycles on the G5. A sample implementation has been included as **modmul64asm_2.S**. The astute reader will note that this is roughly the same as the cost of a single 64-bit by 64-bit division on the G5, which has no instruction for dividing a 128-bit dividend by a 64-bit divisor.

Altivec

As in the case of the `addmul` routine, this `modmul` computation can be performed with minimal effort on our part by using the PowerPC Velocity Engine and Apple's `vecLib` Framework (which is compatible with the G4). A comparable algorithm using the `vecLib` Framework calls the routines `vU256Divide()` to compute the reciprocal and `vU128FullMultiply()` for the mod operation. For more details, the reader is referred to the header files `vBasicOps.h` and `vBigNum.h`.

5 Closing remarks

In the process of preparing this paper, we were asked by a prominent mathematician for basic tricks for achieving parallelism on the G5. Obviously, implementation in low-level PowerPC assembler is not always the right approach for everyone. While a great deal has been written on optimization strategies for this processor [3, 4, 5, 6], the authors have found that there are several relatively easy ways to increase the performance of our computations. One straightforward technique is to write routines which process arrays of data instead of single operands – combined with loop unrolling, this allows the processor to operate on multiple sets of data at the same time. For long dependent sections of calculations, this offers the possibility of considerable compiler optimization. For instance, the function `fast_modmul()` above, can be called as follows (where the size of the arrays is assumed to be a multiple of 8):

```
void array_modmul(double *a, double *b, double n,
                  double recip, double magic, int size) {
    int i;

    for (i = 0; i < size; i += 8) {
        a[i+0] = fast_modmul(a[i+0], b[i+0], n, recip, magic);
        a[i+1] = fast_modmul(a[i+1], b[i+1], n, recip, magic);
        a[i+2] = fast_modmul(a[i+2], b[i+2], n, recip, magic);
        a[i+3] = fast_modmul(a[i+3], b[i+3], n, recip, magic);
        a[i+4] = fast_modmul(a[i+4], b[i+4], n, recip, magic);
        a[i+5] = fast_modmul(a[i+5], b[i+5], n, recip, magic);
        a[i+6] = fast_modmul(a[i+6], b[i+6], n, recip, magic);
        a[i+7] = fast_modmul(a[i+7], b[i+7], n, recip, magic);
    }
}
```

When the compiler can inline⁴ the `fast_modmul()` routine, the entire operation becomes highly pipelined and executes very quickly. The code above can be even faster when the loading and storing into the arrays is reordered. Note that a very similar optimization can be applied to the `barrettmodmul32()` function.

⁴Two ways to easily enable function inlining are (1) directly including a copy of the critical function in the same source file as the loop, or (2) passing both files to the compiler at the same time.

6 Acknowledgments

The authors wish thank J. Buhler, P. Carlisle, R. Crandall, J. Klivington, I. Ollmann, E. Prabhakar, A. Sazegari, B. Tribble, and G. Warner for their support of and contributions to this work.

References

- [1] Apple Computer. *Hardware – Vector Libraries*.
http://developer.apple.com/hardware/ve/vector_libraries.html
- [2] Apple Computer. *An implementation of arbitrary precision arithmetic using the PowerPC Velocity-Engine (G4) vector instructions*. <http://developer.apple.com/samplecode/VelEngMultiprecision/VelEngMultiprecision.html>. January 2003.
- [3] Apple Computer. *Hardware - G5 Performance Programming*.
<http://developer.apple.com/hardware/ve/g5.html>
- [4] Apple Computer. *Optimizing for the Power Mac G5*.
<http://developer.apple.com/performance/g5optimization.html>. April 2005.
- [5] Apple Computer. *PowerPC G5 Performance Primer*.
<http://developer.apple.com/technotes/tn/tn2087.html>. September 2003.
- [6] Apple Computer. *TN2086: Tuning for G5: A Practical Guide*.
<http://developer.apple.com/technotes/tn/tn2086.html>. October 2003.
- [7] Crandall, R. and Klivington, J. *Vector Implementation of Multiprecision Arithmetic*.
<http://www.apple.com/acg>. October 1999.
- [8] Crandall, R. and Papadopoulos, J. *On the implementation of AKS-class primality tests*.
<http://www.apple.com/acg>. March 2003.
- [9] Crandall, R. and Pomerance, C. *Prime Numbers: A Computational Perspective*. Springer-Verlag. New York. 2002
- [10] International Business Machines, Inc. *IBM PowerPC 970FX RISC Microprocessor User's Manual*. v. 1.41. November 2004.
- [11] Knuth, D. *The Art of Computer Programming, Volume 2, Second Edition*. Addison-Wesley. Massachusetts. 1981.
- [12] Markstein, P. *Software Division and Square Root Using Goldshmidt's Algorithms*. Proc. Sixth Intl. Conf. on Real Numbers and Computing, pp. 146–157, November 2004.