64-Bit Architecture Speeds RSA By 4x

MIPS Technologies, Inc.
June 2002

Public-key cryptography, and RSA in particular, is increasingly important to e-commerce transactions. Many digital consumer appliances (e.g. set-top boxes and smart cards) are now expected to send and receive RSA-encoded messages. Decoding these messages is processing intensive and the ability to do it efficiently offers enormous value. This paper describes how MIPS Technologies achieved a 4x speedup on the RSA algorithm—without modifying the processor hardware.

Why RSA Needs High Processing Speed

Public-key cryptography is the only known means by which two remote parties can share information without transmitting a code that a would-be eavesdropper could use to intercept the conversation. As a result, public-key cryptography has become fundamental to the operation of e-commerce and virtual private networks (VPNs).

The RSA algorithm (named after Rivest, Shamir and Adleman, the second team to invent the algorithm but the first to publicize and patent it) is the most widely used public-key cryptography system today.

One problem facing public-key cryptography is that it requires large amounts of processing power. This is an issue even for gigahertz desktop computers. For embedded systems, which typically employ much slower processors, it is an overwhelming problem. Consumers simply will not tolerate digital set-top boxes that pause for long periods of time while exchanging secret information with Web sites. Users will also look dimly upon smart-access devices (smart cards and the associated readers) that force them to wait in the rain for just a little too long while checking their access privileges.

Consequently, RSA encryption and decryption speed is becoming important to a variety of embedded devices. This paper describes how to quadruple that speed by using the 64-bit capabilities of MIPS-based™ processors.

RSA: A Brief Introduction

There are two kinds of cryptographic systems: symmetric and public-key.

The concept of symmetric (or secret key) cryptography is simple: use a key to obscure the contents of a message, then send the obscured message to someone else. The algorithm used to obscure the message may be a well-known standard, such as DES or AES, but only recipients that hold the original key can recover the original message. Recipients without the key see only random noise.

The challenge with symmetric cryptography algorithms is in securely transferring the key between two parties that never meet, as is the case in e-commerce transactions. If the key is sent with the message, then it, too, must be encrypted; and that would require a second key to be sent securely, which brings us back to the same problem.

Public-key cryptography solves this problem by taking a different approach. The basis of any public-key system is some mathematical algorithm that uses different keys to encrypt and decrypt messages. Anyone who wants to receive encrypted messages is responsible for generating a pair of encryption and decryption keys, and then making the encryption key public. Senders use this "public key" to encrypt messages for that person, who can then decipher them because he (and only he) knows the decryption key. Obviously, the two keys must be related in some way, but they are generated in a way that makes it impossible for anyone to derive one key from the other.

RSA encryption is based on the calculation $C = M^e \bmod n$ and decryption is $M = C^d \bmod n$, where M is the plain-text message and C is the encrypted version. Note that the same basic calculation, known as modular exponentiation, is used for both encryption and decryption.

The security of the system comes from the method by which the modular exponentiation parameters are derived. The public exponent (e) must be a prime number but can be chosen somewhat arbitrarily; 17 is frequently used. The modulus (n) must be the product of two prime numbers (usually called p and q). The private exponent (d) is derived from p, q and e.

Deducing the prime factors of a large number is difficult at best and becomes intractable as the number gets larger. Thus, if p and q are sufficiently large, it is impossible to deconstruct n and obtain the private keys within a reasonable period of time, even with the fastest computers. Today, n is typically chosen to be a 512-, 768- or 1,024-bit number. For numbers this large it would take hundreds of thousands of high-speed computers many years to derive p and q from the public keys.

Boosting the Speed of RSA: The Montgomery Algorithm

The traditional method of performing modular exponentiation relies on repeated division. Since division is an inherently slow process, especially when the divisor and dividend are large, RSA encryption and decryption have historically been very slow processes.

In 1985, researcher Peter Montgomery invented a clever algorithm that relies on multiplication and shifting rather than on division. This innovation resulted in a significant boost in speed and has become the accepted method of performing RSA decryption today.

It is beyond the scope of this paper to explain the math behind the Montgomery algorithm, but the code to implement it is summarized in the pseudo-code below. Montgomery's algorithm is explained in reference 1.

```
M, e and n are all k bit numbers.
r is the power of 2 that is larger than n
r^-1 is a number such that (r^-1. r) mod n = 1
n' is the number such that (r^-1. r) – (n . n') = 1

function ModExp (M, e, n)

    calculate n'

    M' := M . r mod n
    x  := 1 . r mod n

    for i=k-1 down to 0
        x := MonPro (x, x)
        if (e[i] = 1) then
            x := MonPro (M', x)
    endloop

    x := MonPro (x, 1)
    return (x)

end

function MonPro (a, b)
    t := a . b
    m := t . n' mod r
    t := (t + m . n) / r

    if (t >= n) then
        return (t-n)
    else
        return (t)
end
```

Typically, in an RSA application, the values being manipulated are 512- or 1,024-bit numbers (i.e. k=512 or k=1,024). The modular exponentiation calculation (ModExp) consists of some initial pre-calculation and then 512 or 1,024 iterations of the loop calling the Montgomery product function (MonPro).

The most significant tasks performed by MonPro are the three multiplications of large numbers. Since r is defined as a power of 2, the mod r operation is trivial and the divide by r operation is a simple right shift. The compare and subtract operations at the end require arithmetic on large numbers, but these are simpler than the multiplications.

The Montgomery Product Calculation

In practice, the large-number MonPro multiplication and shift can be performed by a set of software loops. The pseudo-code below shows the main part of an optimized Montgomery product algorithm (taken from reference 2).

```
for i = 0 to s-1
    /* first inner loop */
    C := 0
    for j = 0 to s-1
        (C, S) := t[j] + a[j].b[i] + C
        t[j] := S
    endloop

    … deal with left over carry bits …

    /* calculate the value of m */
    m := t[0] * n'[0] mod W

    /* second inner loop */
    (C, S) := t[0] + m.n[0]
    for j = 1 to s-1
        (C, S) := t[j] + m.n[j] + C
        t[j-1] := S
    endloop

    … deal with left over carry bits …

endloop

… compare t to n & subtract if necessary …


note: s is the length of a and b, in processor words
```

This pseudo-code makes the calculation of m look fairly complicated. In this case, however, looks are deceiving. The n'[0] value is a single word that is derived simply from n (the modulus) at the start of the modular exponentiation. W is the maximum value that fits into a processor word, so calculating a value mod W takes no effort at all. The m calculation is just a simple multiply operation.

The two inner loops are where the processor spends most of its time. Both loops are similar, so, for purposes of this discussion, we shall concentrate on the implementation of only the first loop.

A MIPS32™-Based Implementation of the Montgomery Product

If we use the pseudo-code above to implement the first inner loop of the Montgomery calculation in C, we get the following code.

```c
for (j=0; j<s; j++) {
    Mult32x32(y, x, a[j], b[i]);

    sum    = x + carry;
    carry  = (sum < carry) ? 1 : 0;

    sum   += t[j];
    carry += (sum < t[j]) ? 1 : 0;

    t[j]  = sum;
    carry += y;
}
```

In this code, Mult32x32 is a function (or macro) that puts the full 64-bit result of multiplying a[j] and b[i] into x and y. This function is expanded in portable C code later in this paper. For now, we will consider how the first inner loop can be implemented in MIPS32 code:

```
    ….. loop initialisation code ….

3:  lw      v0, 0(t1)          # read au[j]
    lw      t3, 0(t5)          # read t[j]
    multu   v0, v1             # calculate au[j] * b[i]
    addu    t1, 4              # increment au pointer
    addu    t3, t3, t2         # add t[j] to previous carry value
    mflo    t0                 # read lo
    sltu    t6, t3, t2         # check for overflow
    mfhi    t2                 # read hi
    addu    t0, t0, t3         # add lo to t[j] + previous carry
    addu    t2, t6             # add overflow flag to hi
    sltu    t6, t0, t3         # check for overflow again
    addu    t5, 4              # increment t pointer
    addu    t2, t6             # add overflow flag to carry
    sw      t0, -4(t5)         # update t[j]
    bne     t5, t9, 3b         # repeat until the end of t
```

Many of the instructions in this loop check for overflow and manipulate carry bits. Some of these operations can be removed if we use multiply-accumulate instructions to add 32-bit values to the 64-bit multiply result, as shown below.

```
    li      t6, 1              # t6 holds the value 1

    ….. loop initialisation code ….

3:  lw      v0, 0(t1)          # read au[j]
    addu    t1, 4              # increment au pointer
    multu   v0, v1             # calculate au[j] * b[i]
    lw      t3, 0(t5)          # read t[j]
    maddu   t2, t6             # add previous carry value
    maddu   t3, t6             # add t[j]
    addu    t5, 4              # increment t pointer
    mflo    t0                 # read sum
    mfhi    t2                 # read carry
    sw      t0, -4(t5)         # update t[j]
    bne     t5, t9, 3b         # repeat until the end of t
```

On most modern MIPS processors (such as the MIPS32™ 4KEm™ and MIPS64™ 5Kc™ cores) the multiplier hardware is fast enough to do the multiply accumulate in a single clock cycle, so this implementation will improve the performance of the loop. If the multiplier hardware becomes a bottleneck, this optimization may actually reduce performance. Obviously, this approach would not be advisable on processors with slower multipliers, such as the MIPS32 4Kp™ core.

Using 64 Bits for Big-Number Arithmetic

Calculations on 512- and 1,024-bit numbers, such as those used in RSA, must be split into chunks that match the width of the processor's registers and ALUs. For many operations, doubling the word size halves the number of steps it takes to perform the calculation. So it is reasonable to assume that a 64-bit processor might double the speed of RSA calculations.

However, while a 64-bit processor doubles the speed of long add, subtract and compare operations, the effect on the all-important multiplication operation is more dramatic. Constructing a 64-bit multiply actually takes four 32-bit multiplies, plus several additions to sum the partial products. On algorithms dominated by multiplies, such as the Montgomery product, the theoretical speedup expected from a 64-bit processor is, therefore, closer to 4x.

To illustrate this effect, the example below shows the C code required to compute the 64-bit result of a 32x32-bit multiply. There is no portable-C syntax to handle more than 32 bits, so the calculation must be split into four 16-bit multiplications.

```
typedef unsigned long u32;

#define LOHALF(n) (((n) << 16) >> 16)
#define HIHALF(n)  ((n) >> 16)

void Mult32x32 (u32 *hi, u32 *lo, u32 x, u32 y)
{
    u32 pp1, pp2, pp3, pp4;

    /* work out partial products */
    pp1 = LOHALF(x) * LOHALF(y);
    pp2 = LOHALF(x) * HIHALF(y);
    pp3 = HIHALF(x) * LOHALF(y);
    pp4 = HIHALF(x) * HIHALF(y);

    /* combine the two middle products & check for overflow */
    pp2 += pp3;
    if (pp2 < pp3) {
        pp4 += (1 << 16);
    }

    /* build the bottom half of the result & check for overflow */
    *lo = (pp2 << 16) + pp1;
    pp4 += (*lo < pp1) ? 1 : 0;

    /* build the top half of the result */
    *hi = (pp2 >> 16) + pp4;
}
```

A MIPS64-Based Implementation of the Montgomery Product

While achieving a 4x speedup should be theoretically possible, simply translating the MIPS32 code above to the MIPS64 equivalent would not deliver the full 4x speedup on current MIPS processors.

Obtaining the full 4x speedup in this way would require a multiplier capable of executing a 64x64 multiply in the same time it would take to perform a 32x32 multiply. That would take 4x as many transistors, which might be prohibitive for many embedded applications. Common implementations today, such as the MIPS64 5Kc core, perform a 32-bit multiply in 2 cycles but take 8-10 cycles for a 64-bit multiply.

All is not lost, however. Even the simplest MIPS processors use pipelining techniques that allow other instructions to execute during the multiply, stalling only if the product is needed before the multiply completes. This technique allows the programmer to schedule other useful operations in parallel with the multiplication.

In the case of the Montgomery product calculation, it is possible to arrange the inner loops so that data movement, additions, and loop-maintenance operations keep the machine occupied while the multiplier is at work. This allows almost any modern 64-bit MIPS processor to deliver close to peak theoretical 64-bit performance, without the silicon expense of a full 64x64-bit multiplier.

The example below shows an efficiently scheduled 64-bit implementation of the first inner loop of the Montgomery product calculation .

```
    /*
       for j=0 to s-1
            (C,S) := t[j] + a[j]*b[i] + C
            t[j] = S
    */

    ….. loop initialisation code ….

    dmultu v0, v1            # start working out a[0] * b[i]

1:  ld     v0, 0(t4)         # read next a[j]
    addu   t4, 8             # increment the a pointer
    ld     t3, 0(t5)         # read t[j]
    addu    t5, 8            # increment the t pointer
    mflo   t0                # read the bottom half of a[j] * b[i] (sum)
    mfhi   t1                # read the top half of a[j] * b[i] (carry)
    dmultu v0, v1            # start working out next a[j] * b[i]
    daddu  t3, t3, t2        # work out t[j] + previous carry
    sltu   t6, t3, t2        # check for overflow
    daddu  t0, t0, t3        # add sum
    sd     t0, -8(t5)        # update t[j]
    sltu   v0, t0, t3        # check for overflow, again
    daddu  t6, v0
    daddu  t2, t1, t6        # add overflow count to carry
    bne    t5, t9, 1b        # repeat until we reach the end of t
```

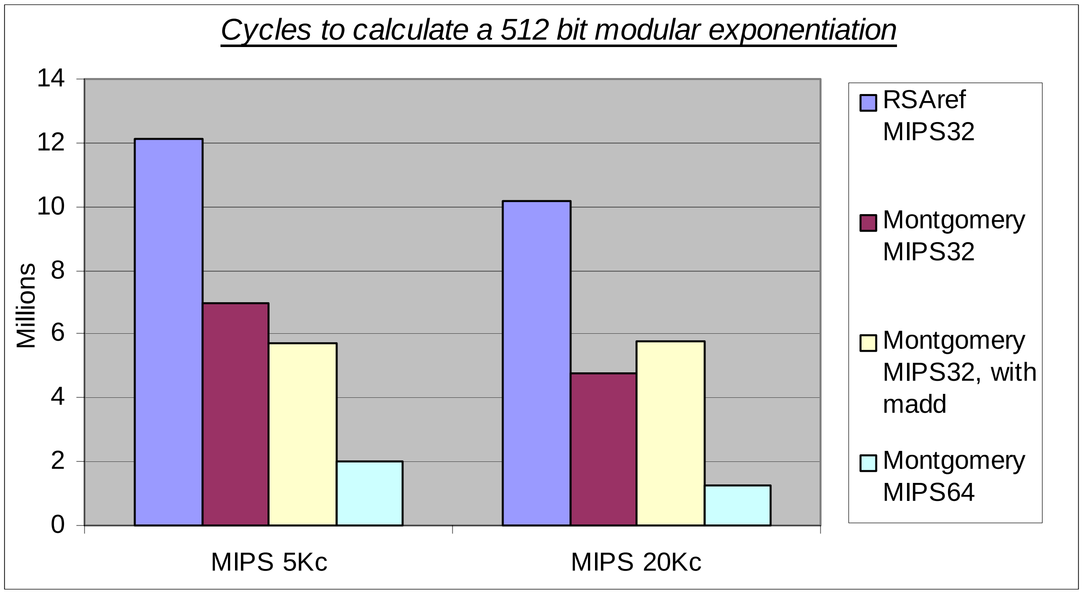
This example is similar to the one previously given for MIPS32, but the loop now uses software pipelining so that the result of the multiply is not used until the next iteration of the loop. The 12 instructions executed between the start of the multiply and the use of its result completely hide the latency of the multiplier hardware.

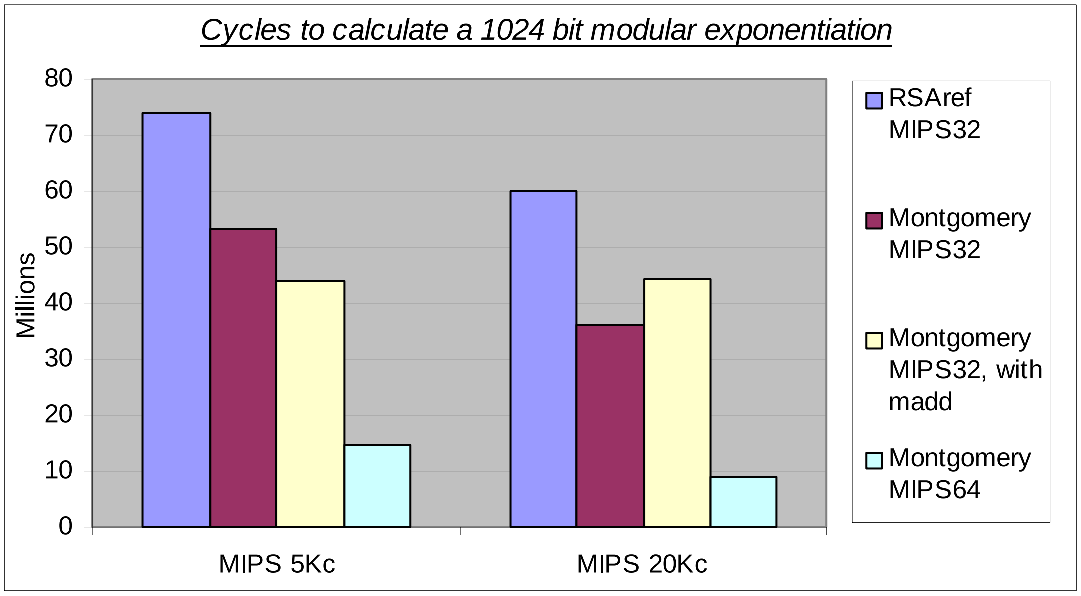
Covering the multiplier's latency in this way removes the opportunity to use a multiply-accumulate to perform the double-length addition, so the additions must be done the long way. Since addition is normally faster and cheaper than multiplication, the trade-off is a good one.

Measuring the Performance Gains

We have seen how the width of a 64-bit datapath and multiplier can be effectively brought to bear on the inner loops of the Montgomery product. To the extent that these dominate the modular exponentiation calculation, the speedup should approach 4x over a 32-bit processor.

The graphs below compare the performance that was actually achieved when this software runs on the MIPS 5Kc and 20Kc™ processor cores. For comparison, the graphs also show the execution time for a non-Montgomery modular exponentiation function, using code from the RSAref package from RSA Data Security Inc. compiled for MIPS32. The runtimes were obtained using MIPS Technologies' MIPSsim™ cycle-accurate simulator.

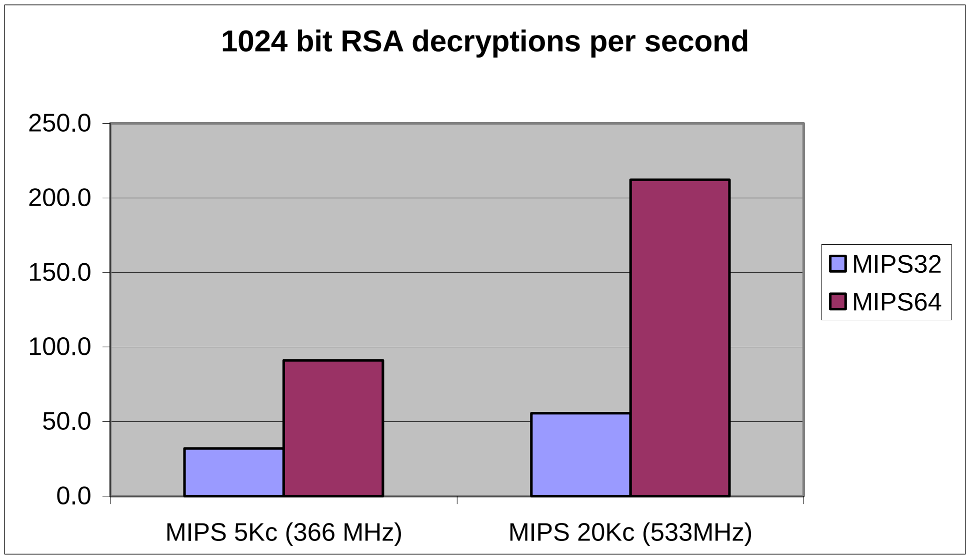




These results show that the Montgomery algorithm delivers a significant improvement over the traditional algorithm for modular exponentiation, approximately doubling the speed of the calculation.

The MIPS32 results demonstrate the importance of taking into consideration the characteristics of the processor when fine-tuning a software algorithm for maximum performance. On the 5Kc processor, use of the multiply-accumulate instruction for double-length addition improves performance by about 20 percent. On the superscalar 20Kc processor, however, the multiply-accumulate becomes a bottleneck and this optimization actually degrades performance. In this case, a different technique is called for.

The MIPS64 results confirm the theoretical benefits of using the Montgomery algorithm on a 64-bit processor. On the MIPS 5Kc core, there is a 3.5x performance difference between the MIPS32 implementation and the MIPS64 implementation. On the MIPS 20Kc core the improvement is 4x (5x using the MIPS32 implementation that is less optimal for the 20Kc core). Using these results we can estimate the throughput of a full RSA decryption engine running on these processors. The estimates are shown in the graph below.



These estimates assume that the 5Kc and 20Kc processors are operating at 366 and 533 MHz, respectively, and that the RSA decryption package takes advantage of the Chinese Remainder Theorem. This theorem allows the 1,024-bit RSA decryption operation to be split into two 512-bit modular exponentiations, with some simple arithmetic to combine the results.

Conclusion: Montgomery and 64 Bits Yield Big Gains
When Montgomery's algorithm was introduced 20 years ago, delivering approximately a 2x speedup over older approaches, it totally changed how RSA encryption and decryption are carried out. There have been no significant improvements to the RSA problem since.

Today, simply applying a 64-bit processor to the problem can deliver another 4x speedup, with just a small amount of software effort and no expertise in cryptography. Considering that the rapidly declining cost of transistors has nearly eliminated the silicon cost differential between 32- and 64-bit processors, selecting a 64-bit processor is an economically prudent alternative to hiring an army of mathematicians to create a new algorithm that will run efficiently on 32-bit processors. Of course, any algorithm these mathematicians devise is also likely to run faster on a 64-bit machine than a 32-bit one.

References

1. C. K. Koc, "High-Speed RSA Implementation," RSA Laboratories, 1994
   ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf

Analyzing and Comparing Montgomery Multiplication Algorithms

2. C. K. Koc, T. Acar & B.S. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," IEEE Micro, June 1996
   http://security.ece.orst.edu/papers/j37acmon.pdf