Arbitrary-Precision Montgomery Arithmetic

Alexander Laing
University of Victoria
Dept. of Electrical and Computer Engineering
`V00799006`
`lainga@uvic.ca`

November 29, 2019

Abstract

Many modern cryptographic systems rely on the efficient calculation of large modular exponents, of the form $(a^k \mod n)$. Most famous of these is the RSA cryptosystem. As naïve calculations become intractably slow for large terms, *Montgomery multiplication* has been developed. It transforms the mantissa(e) into a form where, for a well-chosen modulus, arithmetic is faster in hardware, especially for repetitive calculations like modular exponentiation.

In this report, we present the implementation and optimisation of the Montgomery technique, targeting the ARMv5 microprocessor architecture. Starting with a functional implementation of the "naïve" technique, we proceed to implement Montgomery arithmetic routines; optimise the code, using well-established techniques, at identified bottlenecks; and finally, consider a hand-optimised assembly routine for the largest bottleneck.

At each step, we check correctness by using the routines to perform round-trip RSA encryption and decryption on plaintext samples. We also profile the code using `gprof`. To this end, the routines have primarily been tested with 2048-bit mantissae, and 2047-bit exponents and moduli – recommended keylengths for modern RSA. However, the code is written to support numbers of any arbitrary precision, so long as it is a multiple of 64 bits.

If this report is in electronic form, C sources for these routines should be attached.

Special thanks is made to Dr. Mihai Sima for his support and patience during the preparation of this project.

1. Background

1.1 Modular operations

We assume familiarity with the concepts of residues, the modulo operation, and congruence classes.

Many of the familiar arithmetic operations on the integers $\mathbb{Z}$ can be extended to operations on the integers *modulo* some number n, written $\mathbb{Z}/n\mathbb{Z}$. Addition, subtraction, and multiplication all hold under distribution of the modulo operation:

$$\begin{aligned}(a+b \mod c) &\equiv (a \mod c) + (b \mod c)\\(ab \mod c) &\equiv (a \mod c)(b \mod c)\\(a^b \mod c) &\equiv (a \mod c)(a \mod c)\dots(b \text{ times})\end{aligned}$$

Notably, however, division cannot be carried over this way, *unless* $\gcd(a, n) = 1$, i.e., the number and modulus are coprime. Then there exists some unique element $e \in \mathbb{Z}/n\mathbb{Z}$ such that $ae = (1 \mod n)$.

In this case, we say that a is a *unit* of $\mathbb{Z}/n\mathbb{Z}$, and e is the *modular multiplicative inverse*, or just *inverse*, of a. [1] This mirrors what we know about division in other fields (e.g. the rational numbers): the product of a number and its inverse is one.

However, so long as there are some numbers which are *not* coprime to the modulus, we cannot define division as an operation on $\mathbb{Z}/n\mathbb{Z}$: it is only defined for some of the elements, and not others. An obvious way to remedy this is to make the modulus a prime number. Then *every* nonzero integer in the space is a unit, and with the multiplication operation, they actually form a cyclic multiplicative group.

1.2 Practical limitations

In practice, it is tough to make computers efficiently calculate operations on $\mathbb{Z}/n\mathbb{Z}$. In the abstract, we could simply carry out the operation on normal, non-residue integers, then find the modular residue of the result. This is not so tough for addition. Consider an unsigned 64-bit integer: it can represent any whole number in $[0, 2^{64} - 1]$. The sum of two such integers is, at most, $(2^{64} - 1) + (2^{64} - 1) = (2^{65} - 2)$: it can be represented with 65 bits, perhaps another 64-bit unsigned integer with a single-bit carry.

The task gets a little harder for multiplication. Now our result may range from 0 to:

$$(2^{64} - 1)(2^{64} - 1) = (2^{128} - 2(2^{64}) - 1)$$

[1] Some also write the inverse as a^{-1}, but this lends itself to confusion of terms. We will use $inv(a)$.

However, we can handle the result as two 64-bit integers. On some architectures, this operation is supported in a single instruction; but on others, we must leave behind our original strategy, and carry out the multiplication using some custom algorithm.

Exponentiation is the real issue. The result now ranges from 0 to:

$$(2^{64}-1)^{(2^{64}-1)} \approx 10^{10^{19}}$$

Which would have in the environment of 5 quintillion decimal digits. The problem becomes more acute when working in cryptography, where numbers are often 1024 – 4096 bits long. Our only recourse is to split the calculation into small steps, and replace the intermediate results with their modular residues frequently; or else they might exceed the resources of the computer or universe.

This does not affect the result in any way – recall the distributive property – but does slow down the calculation significantly. Finding the modular residue requires an expensive integer division by the modulus. The only exception is if the modulus is a power of two: then, the modulo operation itself reduces to discarding the uppermost bits of the number! Montgomery arithmetic takes advantage of this fact.

1.3 Montgomery arithmetic

Montgomery arithmetic [1] works by transforming the integers in $\mathbb{Z}/n\mathbb{Z}$ to an auxiliary space $\mathbb{Z}\mathbf{R}/n\mathbb{Z}$, for some power of two $\mathbf{R} > n$, given that n is prime. That is, it converts each integer into a unique, corresponding *Montgomery form*:

$$\begin{aligned} &\forall\, a \in \mathbb{Z}/n\mathbb{Z} && n \in \mathbb{P} \\ &\bar{a} = (a\mathbf{R} \mod n),\ \mathbf{R} = 2^k, \mathbf{R} > n,\ k, n \in \mathbf{Z} \end{aligned}$$

This form is specifically chosen to make the modular residue operation - the expensive component identified in the last section - easy. The Montgomery form of this operation is called *Montgomery reduction*. Given a modulus N, auxiliary modulus $\mathbf{R}$, and integer $A \in [0, \mathbf{R}N]$, we compute:

$$N' \leftarrow (-inv(N) \mod \mathbf{R}) \qquad i.e.,\ NN' = (-1 \mod R) \tag{1.1}$$

$$m \leftarrow (AN' \mod \mathbf{R}) \tag{1.2}$$

$$t \leftarrow \frac{(A + mN)}{\mathbf{R}} \tag{1.3}$$

$$\bar{a} = \begin{cases} t - N & t \geq N \\ t & otherwise \end{cases} \qquad s.t.\ \bar{a} = (A \times inv(\mathbf{R}) \mod N). \tag{1.4}$$

[2] [3, p. 600] Note that the only hard operations involved are addition and multiplication. Since $\mathbf{R}$ is a power of two, division and modular residue by $\mathbf{R}$ amount to bit shifting and bit masking, respectively.

Yet why does Montgomery reduction compute ($A/\mathbf{R} \mod N$), and not simply ($A \mod N$)? Because the product of two Montgomery-space numbers is

$$\begin{aligned}\bar{a} \times \bar{b} &= (a\mathbf{R} \mod N)(b\mathbf{R} \mod N)\\ &= ab\mathbf{R}^2 \mod N\\ &= \bar{ab}\mathbf{R}\end{aligned}$$

Montgomery reduction is designed to cleverly strip out this extra auxiliary modulus, and take the modular residue at the same time.

Conversion to and from Montgomery form can be done using a conventional algorithm to take the modular residue N and divide by $\mathbf{R}$. However, for simplicity, it can also be achieved using the Montgomery reduction algorithm. Given an integer a and its corresponding Montgomery form $\bar{a}$ over $\mathbb{Z}\mathbf{R}/n\mathbb{Z}$,

$$\bar{a} = \frac{a \times \mathbf{R}^2}{\mathbf{R}} \mod n \qquad = \mathit{Reduce}(a \times (\mathbf{R}^2 \mod n)) \tag{1.5}$$

$$a = \frac{\bar{a}}{\mathbf{R}} \mod n \qquad = \mathit{Reduce}(\bar{a}) \tag{1.6}$$

Although this requires precalculation of the constant ($\mathbf{R}^2 \mod n$).

1.4 Arbitrary precision

Modern cryptography resembles a race, in some ways, between the capacity of those who wish to encrypt secrets, and those who wish to crack those secrets. One consequence is a constant drive to larger and larger cryptographic key sizes. When RSA was first introduced, 192-bit keys were common; now, the standard is 2048- or 4096-bit keys.

Each 2048-bit RSA key is simply a triple of 2048-bit integers (so the name is a misnomer: it is actually 6144 bits long) (n, e, d). n is the modulus for $\mathbb{Z}/n\mathbb{Z}$; e is the public exponent, which can be shared freely; and d is the private exponent, which should never be shared.

Few computer architectures offer native support for integers of such size. However, they are integers just as `int` or `long long` are, and can be operated upon in the same way. To allow this, we must implement *arbitrary-precision arithmetic*. We represent each integer as an array of words in the machine architecture (e.g., 32-bit integers in ARM32), called *limbs*:

$$\begin{aligned}W &\in [0, 2^N - 1)\\ &= (w_{k-1}(2^{n(k-1)}) + w_{k-2}(2^{n(k-2)}) + \cdots + w_1(2^n) + w_0)\\ &\equiv (w_{k-1}||w_{k-2}||\ldots||w_1||w_0)\\ w_i &\in [0, 2^n - 1),\ k = \frac{N}{n}\end{aligned}$$

Then, we write software routines which mirror the hardware arithmetic instructions for smaller integers.

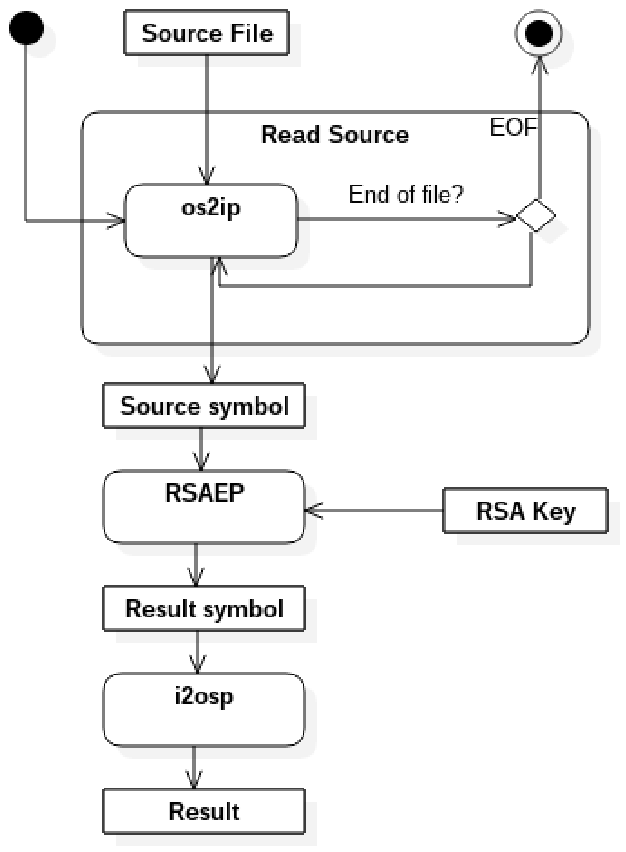


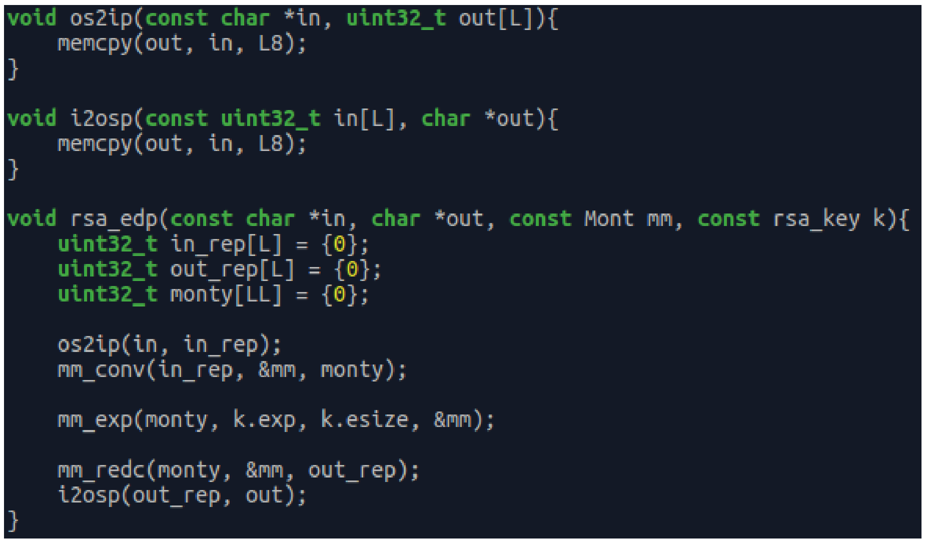
Figure 2.1: Testbench RSA activity diagram.

2. Design

2.1 Testbench

To test the various code versions, we encrypt and decrypt a plaintext corpus using RSA, and compare the output to the original file.

For performance profiling, we use `openssl` to generate a set of random, valid 2048-bit keypairs. Then, we encrypt and decrypt a fixed, 9-kilobyte corpus (the text of the 1948 Universal Declaration of Human Rights), once with each keypair. This evens out variations caused by the size of the private exponent, which generally varies between 2047 and 2044 bits. The basic operation of the



```
void os2ip(const char *in, uint32_t out[L]){
    memcpy(out, in, L8);
}

void i2osp(const uint32_t in[L], char *out){
    memcpy(out, in, L8);
}

void rsa_edp(const char *in, char *out, const Mont mm, const rsa_key k){
    uint32_t in_rep[L] = {0};
    uint32_t out_rep[L] = {0};
    uint32_t monty[LL] = {0};

    os2ip(in, in_rep);
    mm_conv(in_rep, &mm, monty);

    mm_exp(monty, k.exp, k.esize, &mm);

    mm_redc(monty, &mm, out_rep);
    i2osp(out_rep, out);
}
```

Figure 2.2: I2OSP, OS2IP, and RSAEP/RSADP primitives.

testbench is given in (Fig. 2.1). Names are from the RSAv2.2 specification [4]: RSAEP is the RSA encryption primitive; OS2IP is the octet-string-to-integer primitive, which converts strings of bytes (octets) into symbols [1]; and I2OSP is its reverse. [2] In practice, these three functions are very simple (Fig. 2.2).

In all cases, the code has been tested on `qemu-arm`, on a host AMD64 machine. Performance may be better on actual ARM hardware, although call counts and behaviour will not.

2.2 Montgomery arithmetic code

The operation of the modular exponentiation routine is as described in the background section. During an encryption session, we set up a series of Montgomery constants once; then, for each symbol in the source, we convert it to Montgomery form, take its exponent using the square-and-multiply algorithm, and convert it back to regular form (Fig. 2.3). Decryption proceeds identically, but with the other exponent of the RSA key.

To begin, arbitrary-precision addition, subtraction and multiplication operations are defined (e.g. Fig. 2.4). There is not much room for innovation here, but the carry calculation is taken from BearSSL [5]. These are used to implement a function `mm_init`, which, given a certain RSA modulus N, uses Stein's binary GCD algorithm [3, p. 606] to calculate $N' = (-inv(N) \mod R)$. It also calculates the conversion factor $CF = (R^2 \mod N)$. Then, the Montgomery

[1]In the cryptographic sense: an integer upon which the cryptosystem operates.

[2]Note that the testbench does *not* implement any proper version of RSA: it does not use a proper padding scheme for the symbols.

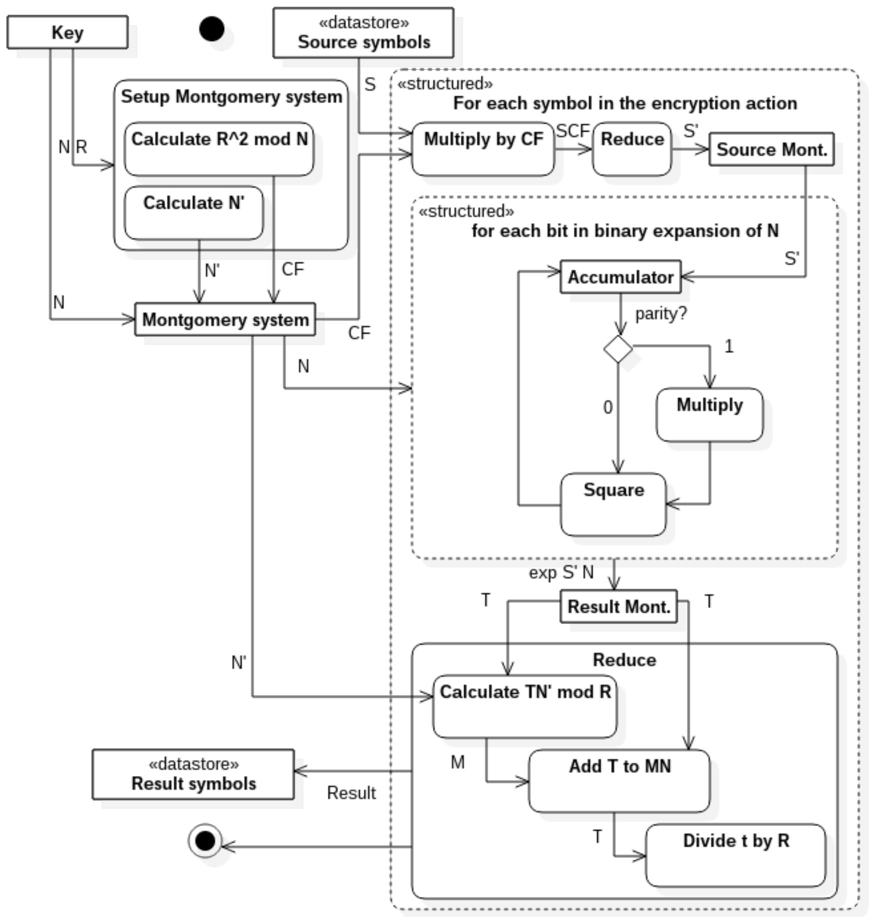
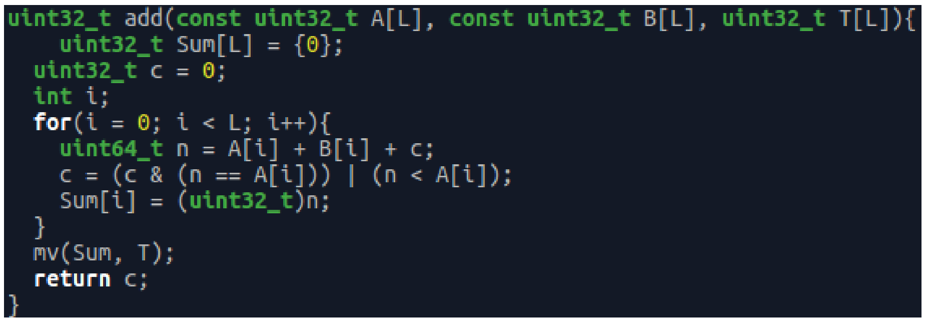


Figure 2.3: Montgomery exponentiation activity diagram.



```
uint32_t add(const uint32_t A[L], const uint32_t B[L], uint32_t T[L]){
    uint32_t Sum[L] = {0};
  uint32_t c = 0;
  int i;
  for(i = 0; i < L; i++){
    uint64_t n = A[i] + B[i] + c;
    c = (c & (n == A[i])) | (n < A[i]);
    Sum[i] = (uint32_t)n;
  }
  mv(Sum, T);
  return c;
}
```

Figure 2.4: Arbitrary-precision addition.

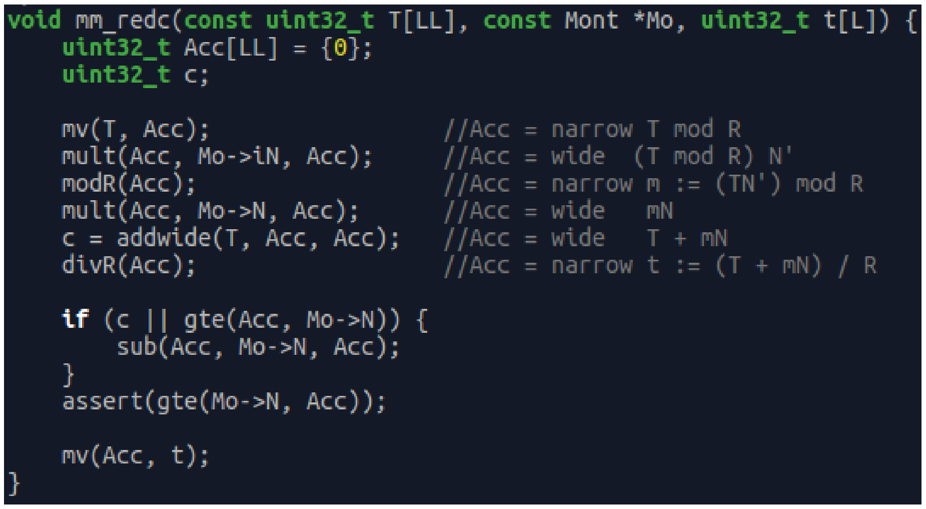

```
void mm_redc(const uint32_t T[LL], const Mont *Mo, uint32_t t[L]) {
    uint32_t Acc[LL] = {0};
    uint32_t c;

    mv(T, Acc);                 //Acc = narrow T mod R
    mult(Acc, Mo->iN, Acc);     //Acc = wide  (T mod R) N'
    modR(Acc);                  //Acc = narrow m := (TN') mod R
    mult(Acc, Mo->N, Acc);      //Acc = wide   mN
    c = addwide(T, Acc, Acc);   //Acc = wide   T + mN
    divR(Acc);                  //Acc = narrow t := (T + mN) / R

    if (c || gte(Acc, Mo->N)) {
        sub(Acc, Mo->N, Acc);
    }
    assert(gte(Mo->N, Acc));

    mv(Acc, t);
}
```

Figure 2.5: Functional reduction routine. From [3].

reduction routine `mm_redc` (Fig. 2.5) and exponentiation routine `mm_exp` are defined; and finally, RSAEP calls `mm_exp`.

3. Performance

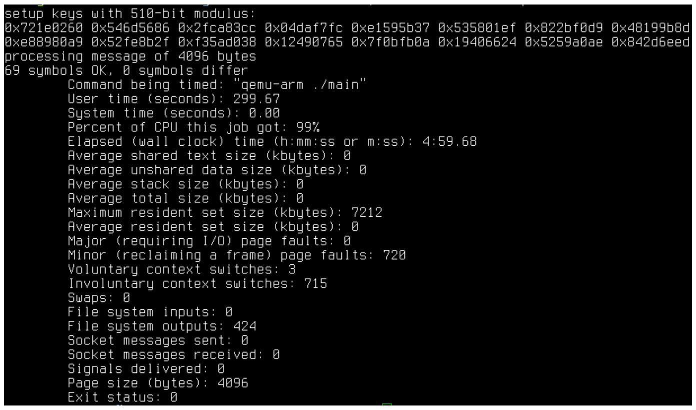
There should be `gprof` statistics files for each of the sections in this chapter, to demonstrate the speedups mentioned in the text.

3.1 Naïve approach

As mentioned, we may carry out a perfectly valid exponentiation by repeatedly multiplying an intermediate accumulator the mantissa, then taking the modular residue of the accumulator. To illustrate, we implemented exponentiation this way, then tried to encrypt and decrypt the first half of the corpus, a single time, with a 512-bit modulus. Results were predictably dismal: almost 5 minutes, real-time (Fig. 3.1).

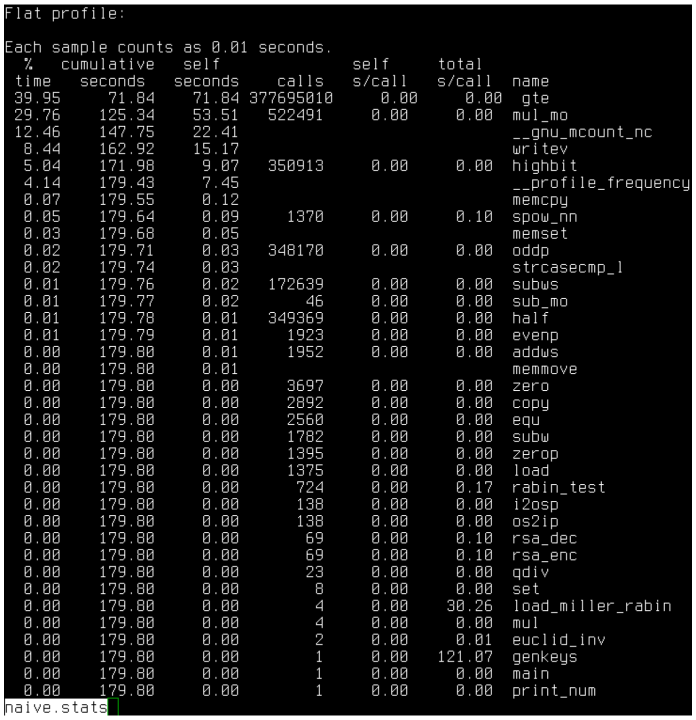
Profiling shows a large share of time is spent within `mul_mo`, the modular multiplication routine (Fig. 3.2). This reinforces our original desire to try Montgomery multiplication: it a the bottleneck in the naïve version.

However, this implementation did sharpen focus for the project code proper.



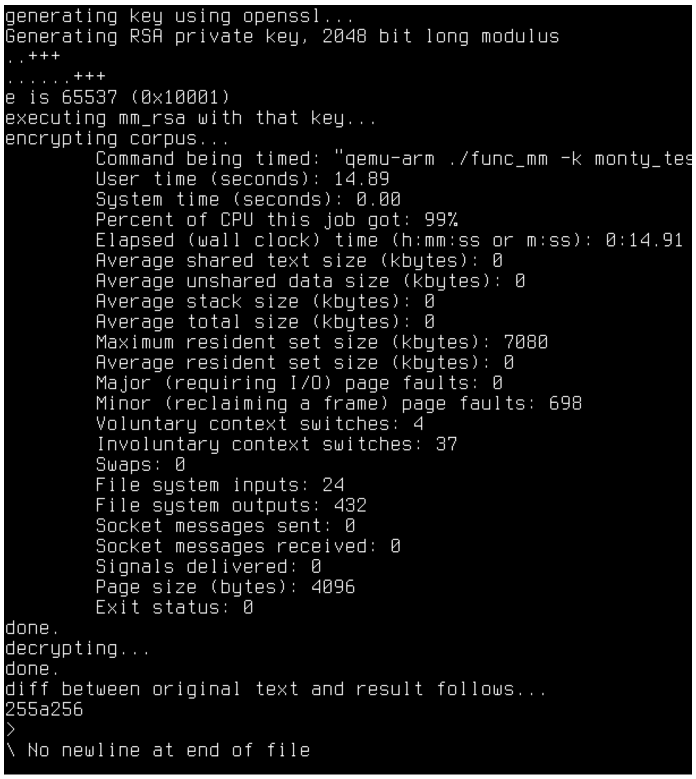
```
setup keys with 510-bit modulus:
0x721e0260 0x546d5686 0x2fca83cc 0x04daf7fc 0xe1595b37 0x535801ef 0x822bf0d9 0x48199b8d
0xe88980a9 0x52fe8b2f 0xf35ad038 0x12490765 0x7f0bfb0a 0x19406624 0x5259a0ae 0x842d6eed
processing message of 4096 bytes
69 symbols OK, 0 symbols differ
        Command being timed: "qemu-arm ./main"
        User time (seconds): 299.67
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 4:59.68
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 7212
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 720
        Voluntary context switches: 3
        Involuntary context switches: 715
        Swaps: 0
        File system inputs: 0
        File system outputs: 424
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Figure 3.1: `time -v` results for the naïve implementation. 1 round-trip encryption-decryption with a 512-bit key.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls   s/call   s/call  name    
 39.95     71.84    71.84 377695010     0.00     0.00  gte
 29.76    125.34    53.51   522491     0.00     0.00  mul_mo
 12.46    147.75    22.41                             __gnu_mcount_nc
  8.44    162.92    15.17                             writev
  5.04    171.98     9.07   350913     0.00     0.00  highbit
  4.14    179.43     7.45                             __profile_frequency
  0.07    179.55     0.12                             memcpy
  0.05    179.64     0.09     1370     0.00     0.10  spow_nn
  0.03    179.68     0.05                             memset
  0.02    179.71     0.03   348170     0.00     0.00  oddp
  0.02    179.74     0.03                             strcasecmp_l
  0.01    179.76     0.02   172639     0.00     0.00  subws
  0.01    179.77     0.02       46     0.00     0.00  sub_mo
  0.01    179.78     0.01   349369     0.00     0.00  half
  0.01    179.79     0.01     1923     0.00     0.00  evenp
  0.00    179.80     0.01     1952     0.00     0.00  addws
  0.00    179.80     0.01                             memmove
  0.00    179.80     0.00     3697     0.00     0.00  zero
  0.00    179.80     0.00     2892     0.00     0.00  copy
  0.00    179.80     0.00     2560     0.00     0.00  equ
  0.00    179.80     0.00     1782     0.00     0.00  subw
  0.00    179.80     0.00     1395     0.00     0.00  zerop
  0.00    179.80     0.00     1375     0.00     0.00  load
  0.00    179.80     0.00      724     0.00     0.17  rabin_test
  0.00    179.80     0.00      138     0.00     0.00  i2osp
  0.00    179.80     0.00      138     0.00     0.00  os2ip
  0.00    179.80     0.00       69     0.00     0.10  rsa_dec
  0.00    179.80     0.00       69     0.00     0.10  rsa_enc
  0.00    179.80     0.00       23     0.00     0.00  qdiv
  0.00    179.80     0.00        8     0.00     0.00  set
  0.00    179.80     0.00        4     0.00    30.26  load_miller_rabin
  0.00    179.80     0.00        4     0.00     0.00  mul
  0.00    179.80     0.00        2     0.00     0.01  euclid_inv
  0.00    179.80     0.00        1     0.00   121.07  genkeys
  0.00    179.80     0.00        1     0.00     0.00  main
  0.00    179.80     0.00        1     0.00     0.00  print_num
naive.stats
```

Figure 3.2: `gprof` results for the naïve implementation.



```
generating key using openssl...
Generating RSA private key, 2048 bit long modulus
..+++
......+++
e is 65537 (0x10001)
executing mm_rsa with that key...
encrypting corpus...
        Command being timed: "qemu-arm ./func_mm -k monty_tes
        User time (seconds): 14.89
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:14.91
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 7080
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 698
        Voluntary context switches: 4
        Involuntary context switches: 37
        Swaps: 0
        File system inputs: 24
        File system outputs: 432
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
done.
decrypting...
done.
diff between original text and result follows...
255a256
>
\ No newline at end of file
```

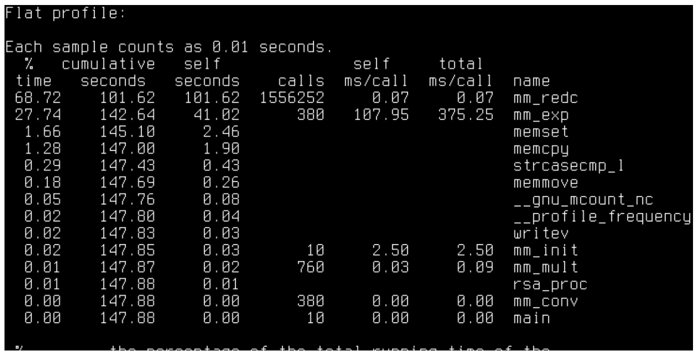
Figure 3.3: `test.sh` results for the functional implementation. 1 round-trip encryption-decryption with a 2048-bit key. Optimised versions had broadly similar call graphs.

Key generation with the Miller-Rabin test is hard to get working, and for the next versions, this was deferred to the established routines in `openssl`.

3.2 Functional code

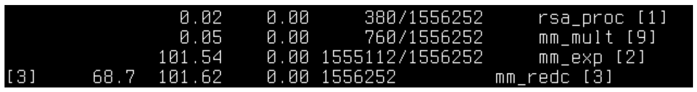
The functional Montgomery-arithmetic code provides palpable improvement over the naïve approach. For a start, this implementation can handle 2048-bit words and key sizes; the 9.4kb corpus takes about 15 seconds to process (Fig. 3.3) (ignoring the lack of newline in the result).

The majority of time is now spent in `mm_redc`, the Montgomery reduction function (Fig. 3.4). Two approaches present themselves: we may try to speed up `mm_redc`, but we may also try to reduce the vast number of calls to it made by `mm_exp`.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.72    101.62   101.62  1556252     0.07     0.07  mm_redc
 27.74    142.64    41.02      380   107.95   375.25  mm_exp
  1.66    145.10     2.46                             memset
  1.28    147.00     1.90                             memcpy
  0.29    147.43     0.43                             strcasecmp_l
  0.18    147.69     0.26                             memmove
  0.05    147.76     0.08                             __gnu_mcount_nc
  0.02    147.80     0.04                             __profile_frequency
  0.02    147.83     0.03                             writev
  0.02    147.85     0.03       10     2.50     2.50  mm_init
  0.01    147.87     0.02      760     0.03     0.09  mm_mult
  0.01    147.88     0.01                             rsa_proc
  0.00    147.88     0.00      380     0.00     0.00  mm_conv
  0.00    147.88     0.00       10     0.00     0.00  main
```

Figure 3.4: `test_prof.sh` results for the functional implementation. 10 rounds of encryption with 2048-bit keys.



```
                 0.02    0.00     380/1556252     rsa_proc [1]
                 0.05    0.00     760/1556252     mm_mult [9]
               101.54    0.00 1555112/1556252     mm_exp [2]
[3]     68.7  101.62    0.00 1556252          mm_redc [3]
```

Figure 3.5: `gprof` call tree for `mm_redc`.

3.3 Improving `mm_exp`

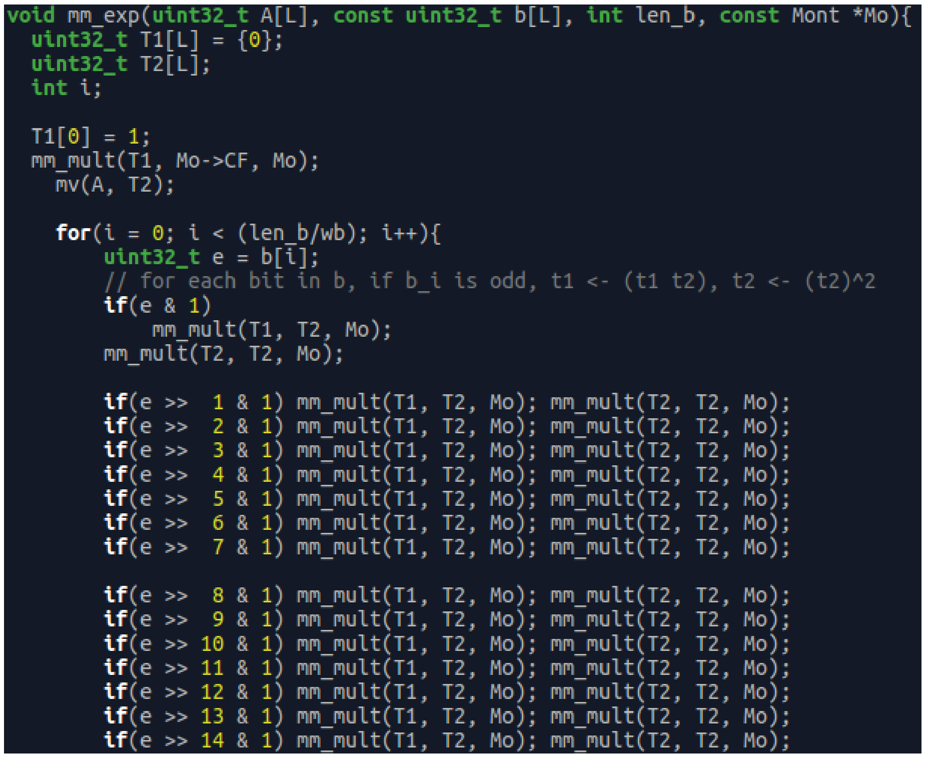
The functional implementation uses the Montgomery Ladder, a constant-time variant on the square-and-multiply technique (unrelated to Montgomery arithmetic except by inventor). This involves a conditional examination each bit in the (2048-bit) exponent. An obvious improvement is to unroll the loop 32 times, so that each iteration examines each of the 32 bits in a single word of the exponent at a time (Fig. 3.6). Another improvement is to declare the big-number arguments as fixed-size arrays, rather than pointers (same figure) - which has the dual effect of declaring those operands as `restrict` to the compiler, and tracking intended operand sizes for readability.

The unrolled loop still needs a small, unoptimised epilogue to handle the remainder of the exponent which does not fit into a whole 32-bit word. However, this was not optimised, as profiling found the bottleneck had moved to `mm_redc`.

The `mv` and `mvwide` functions, which move arbitrary-precision numbers between buffers, were actually just syntactic sugar for a fixed-size `memcpy`, and so could be inlined. Finally, during refactoring, one of the multiplication calls became entirely unnecessary, and was removed. This made the routine no longer constant-time, however.

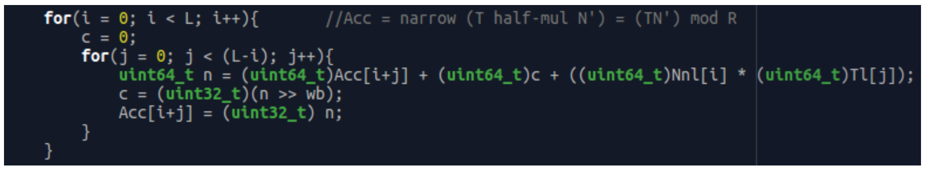
In total, execution time for the benchmark was reduced from 14.79 seconds to 10.39 seconds (average of 10 runs), a 29.7% speedup.

As an aside, an attempt was made to inline `mm_mult`, which consists only of a call to `mult` followed by `mm_redc`. However, this yielded no improvements, and actually made the code run somewhat slower (17.15 seconds); so the attempt probably obscured some pipelining opportunity from the compiler.



```c
void mm_exp(uint32_t A[L], const uint32_t b[L], int len_b, const Mont *Mo){
  uint32_t T1[L] = {0};
  uint32_t T2[L];
  int i;

  T1[0] = 1;
  mm_mult(T1, Mo->CF, Mo);
    mv(A, T2);

    for(i = 0; i < (len_b/wb); i++){
        uint32_t e = b[i];
        // for each bit in b, if b_i is odd, t1 <- (t1 t2), t2 <- (t2)^2
        if(e & 1)
            mm_mult(T1, T2, Mo);
        mm_mult(T2, T2, Mo);

        if(e >>  1 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  2 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  3 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  4 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  5 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  6 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  7 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);

        if(e >>  8 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >>  9 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >> 10 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >> 11 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >> 12 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >> 13 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
        if(e >> 14 & 1) mm_mult(T1, T2, Mo); mm_mult(T2, T2, Mo);
```

Figure 3.6: Start of the unrolled form of `mm_exp`.



```
for(i = 0; i < L; i++){        //Acc = narrow (T half-mul N') = (TN') mod R
    c = 0;
    for(j = 0; j < (L-i); j++){
        uint64_t n = (uint64_t)Acc[i+j] + (uint64_t)c + ((uint64_t)Nnl[i] * (uint64_t)Tl[j]);
        c = (uint32_t)(n >> wb);
        Acc[i+j] = (uint32_t) n;
    }
}
```

Figure 3.7: Half multiplication in `mm_exp`.

3.4 Improving `mm_redc`

The first part of the reduction routine `mm_redc` involves calculation of $m = (TN' \bmod R)$, where R is the number base, 2^{2048}. As discussed, this is equivalent to a multiplication followed by a discarding of the resulting upper half. An obvious optimisation here is to inline the multiplication, and then modify its indices so that it only calculates the lower half of the result (Fig. 3.7).

This yielded good results, and other routines called in `mm_redc` were inlined as well, to remove the overhead associated with `memcpy`ing results between various temporary buffers. To improve memory locality further, the routine operands were also copied into local variables. The result was a further benchmark reduction from 10.39 seconds to 8.5 seconds, an 18.2% speedup.

3.5 Improving `mm_mult`

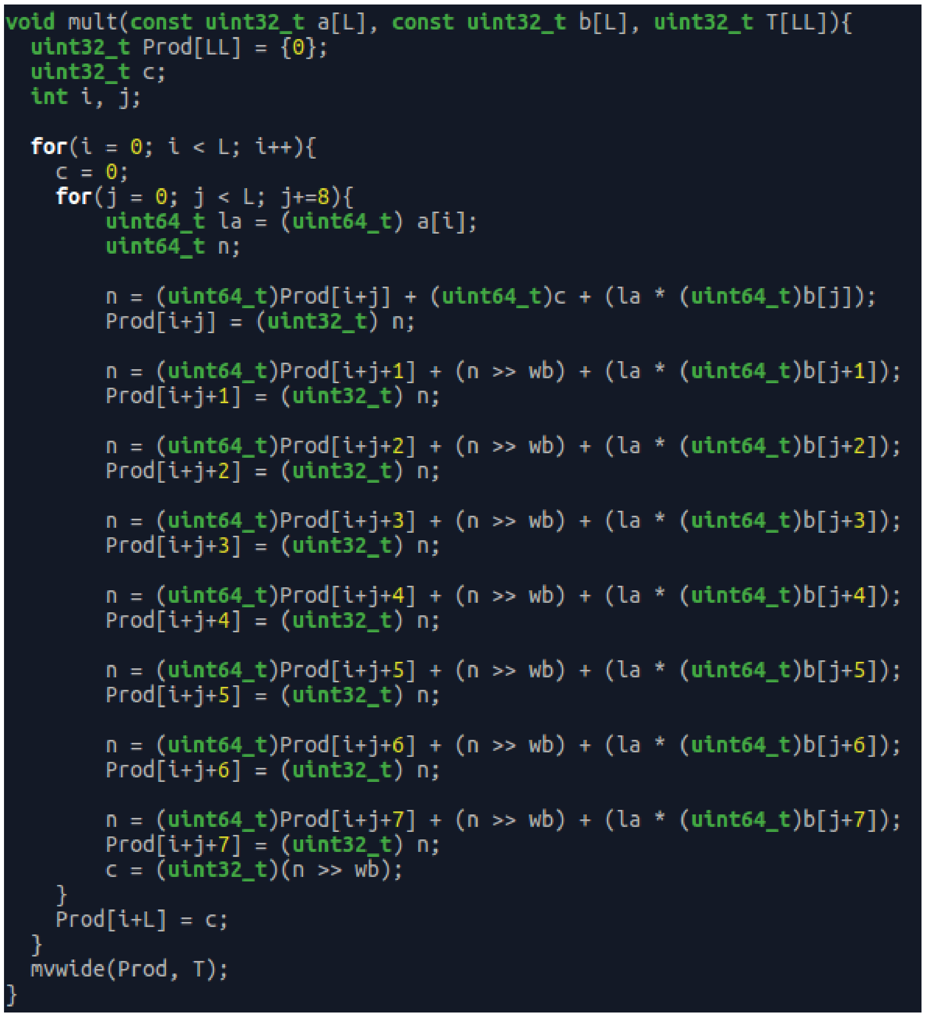
As they had worked well for `mm_redc`, loop unrolling and introduction of local variables were applied to `mm_mult` as well (Fig. 3.8). However, by this point, diminishing returns were arriving: from 8.5 seconds to 7.75 seconds, an 8.8% speedup.

3.6 Assembly for `mm_mult`

The next recourse was to try rewriting the sequence of assembly instructions for `mm_mult`. In its unrolled form as generated by GCC, it is not easy to tell what is going on (Fig. 3.9); the un-unrolled form was studied instead. The major effort was made on the multiplication loop, within `.L106`.

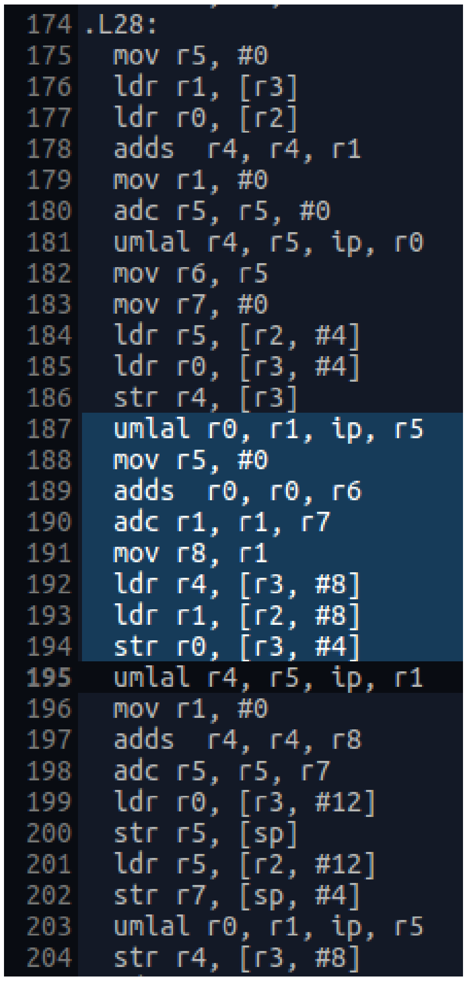
We may start by eliminating some of the unnecessary register moves. For instance, the carry between inner loop jumps (within `.L107`) can remain in `r0`, instead of moving between `r0` and `r7` across jumps. Operations were also reordered (as far apart as possible, per register) to avoid data-dependency-induced bubbling in the pipeline. Alas, there seems to be a lot of inherent dependency in the loop, and trying to keep register references apart was ”like a knife-fight in a phone booth” (Fig. 3.10).

It was hard to improve upon the instruction `umlal` (unsigned long multiply and accumulate), and it was kept – as it allows for calculation of the product $A_n \times B_n + c$ in a single instruction, for c the carry from the previous inner loop.



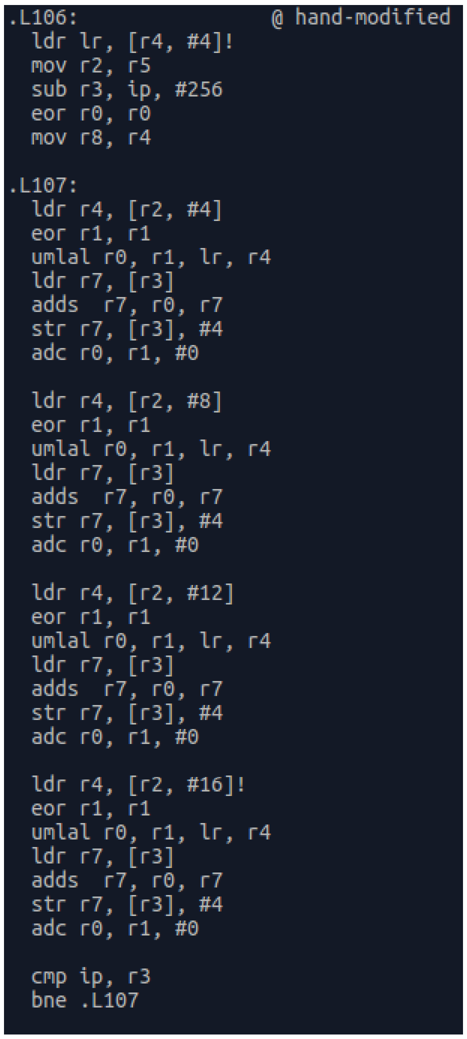
```
void mult(const uint32_t a[L], const uint32_t b[L], uint32_t T[LL]){
  uint32_t Prod[LL] = {0};
  uint32_t c;
  int i, j;

  for(i = 0; i < L; i++){
    c = 0;
    for(j = 0; j < L; j+=8){
        uint64_t la = (uint64_t) a[i];
        uint64_t n;

        n = (uint64_t)Prod[i+j] + (uint64_t)c + (la * (uint64_t)b[j]);
        Prod[i+j] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+1] + (n >> wb) + (la * (uint64_t)b[j+1]);
        Prod[i+j+1] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+2] + (n >> wb) + (la * (uint64_t)b[j+2]);
        Prod[i+j+2] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+3] + (n >> wb) + (la * (uint64_t)b[j+3]);
        Prod[i+j+3] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+4] + (n >> wb) + (la * (uint64_t)b[j+4]);
        Prod[i+j+4] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+5] + (n >> wb) + (la * (uint64_t)b[j+5]);
        Prod[i+j+5] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+6] + (n >> wb) + (la * (uint64_t)b[j+6]);
        Prod[i+j+6] = (uint32_t) n;

        n = (uint64_t)Prod[i+j+7] + (n >> wb) + (la * (uint64_t)b[j+7]);
        Prod[i+j+7] = (uint32_t) n;
        c = (uint32_t)(n >> wb);
    }
    Prod[i+L] = c;
  }
  mvwide(Prod, T);
}
```

Figure 3.8: Unrolled form of `mm_mult`.



```
.L28:
  mov r5, #0
  ldr r1, [r3]
  ldr r0, [r2]
  adds  r4, r4, r1
  mov r1, #0
  adc r5, r5, #0
  umlal r4, r5, ip, r0
  mov r6, r5
  mov r7, #0
  ldr r5, [r2, #4]
  ldr r0, [r3, #4]
  str r4, [r3]
  umlal r0, r1, ip, r5
  mov r5, #0
  adds  r0, r0, r6
  adc r1, r1, r7
  mov r8, r1
  ldr r4, [r3, #8]
  ldr r1, [r2, #8]
  str r0, [r3, #4]
  umlal r4, r5, ip, r1
  mov r1, #0
  adds  r4, r4, r8
  adc r5, r5, r7
  ldr r0, [r3, #12]
  str r5, [sp]
  ldr r5, [r2, #12]
  str r7, [sp, #4]
  umlal r0, r1, ip, r5
  str r4, [r3, #8]
```

Figure 3.9: GCC-generated assembly for `mm_mult`. One of the unrolled loop iterations is highlighted. .L28 is the start of the inner loop of the routine. `umlal` is an unsigned long (i.e., `uint64_t`) multiply.



```
.L106:                @ hand-modified
  ldr lr, [r4, #4]!
  mov r2, r5
  sub r3, ip, #256
  eor r0, r0
  mov r8, r4

.L107:
  ldr r4, [r2, #4]
  eor r1, r1
  umlal r0, r1, lr, r4
  ldr r7, [r3]
  adds  r7, r0, r7
  str r7, [r3], #4
  adc r0, r1, #0

  ldr r4, [r2, #8]
  eor r1, r1
  umlal r0, r1, lr, r4
  ldr r7, [r3]
  adds  r7, r0, r7
  str r7, [r3], #4
  adc r0, r1, #0

  ldr r4, [r2, #12]
  eor r1, r1
  umlal r0, r1, lr, r4
  ldr r7, [r3]
  adds  r7, r0, r7
  str r7, [r3], #4
  adc r0, r1, #0

  ldr r4, [r2, #16]!
  eor r1, r1
  umlal r0, r1, lr, r4
  ldr r7, [r3]
  adds  r7, r0, r7
  str r7, [r3], #4
  adc r0, r1, #0

  cmp ip, r3
  bne .L107
```

Figure 3.10: Handwritten nested loop for `mm_mult`. `r5` holds the address of the first operand; `r3`, the result.

3.7 Analysis

It would appear that certain techniques, like operator-strength reduction and parallelisation, were not considered. However, the nature of the problem and its domain meant that these techniques either were necessarily used (in the functional implementation), or impossible to use.

Specifically, operator strength reduction was arguably the point of the problem – to avoid division (by Montgomery arithmetic), and then multiplication (by using the square-and-multiply algorithm).

Parallel instructions (with ARM NEON) would not be applicable to this problem. Most of the arbitrary-precision calculations involved, including exponentiation, involve chained carrying and are not parallelisable. Mutliplication might be parallelised by precomputing the product of one operand by each of the limbs of the other operand, in parallel. Or, it might be achievable using a divide-and-conquer algorithm, like Karatsuba multiplication. However, neither of these were attempted.

At a wider scale, the encryption function *is* parallelisable: it is $c = (p^e \bmod n)$, for each symbol p, and unchanging exponent and modulus (the RSA key). However, this would require writing NEON instructions encompassing the entire RSAEP and RSADP operations, and still operating on 2048-bit words or larger, which is perhaps not feasible for space reasons.

4. Conclusions

By iterative profiling and testing, an implementation of Montgomery arithmetic for ARMv5 was optimised from a benchmark emulated performance of 14.71 seconds to 7.75 seconds – a 48% speedup – and an order of magnitude faster than a naïve, non-Montgomery implementation, at over 300 seconds. Using modern, industry-standard 2048-bit RSA keys, this represents approximately 204ms for each 256-byte symbol, or $796\mu s$ per byte; performance may be better on real, non-emulated hardware.

Intensive optimisation at the assembly level, on identified bottlenecks, seems to have reached some sort of lower bound on instruction count; further improvements will have to be made by improved algorithms and math fundamentals.

Bibliography

[1] P. Montgomery, "Modular Multiplication without Trial Division.", *Mathematics of Computation*, vol. 44-170, pp. 519-521. (Online) Available: https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf

[2] H. Warren, "Note: Montgomery Multiplication", Jul. 2012. (Online) Available: http://www.hackersdelight.org/MontgomeryMultiplication.pdf

[3] A. Menezes, et al., *Handbook of Applied Cryptography.* Boca Raton, FL: CRC Press, 1996.

[4] *PKCS #1: RSA Cryptography Specifications Version 2.2*, IETF RFC 8017, Nov. 2016.

[5] T. Pornin, "Constant-Time Crypto", BearSSL, 2018. (Online) Available: https://www.bearssl.org/constanttime.html