IMPLEMENTING THE RIVEST SHAMIR AND ADLEMAN PUBLIC KEY ENCRYPTION ALGORITHM ON A STANDARD DIGITAL SIGNAL PROCESSOR

Paul Barrett, MSc (Oxon)
COMPUTER SECURITY LTD
August 1986

ABSTRACT

A description of the techniques employed at Oxford University to obtain a high speed implementation of the RSA encryption algorithm on an "off-the-shelf" digital signal processing chip. Using these techniques a two and a half second (average) encrypt time (for 512 bit exponent and modulus) was achieved on a first generation DSP (The Texas Instruments TMS 32010) and times below one second are achievable on second generation parts. Furthermore the techniques of algorithm development employed lead to a provably correct implementation.

WHY DSP?

At the time we started work we considered several implementation options:

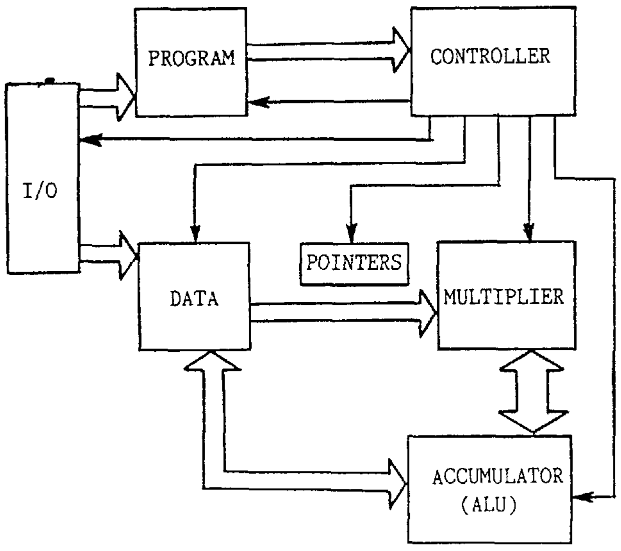
1. The first and most available option was an eight bit micro-processor - best estimates of 512 bits in 4 minutes (ie. 2 bits per second) did not seem very promising.
2. A 16 bit micro-processor - might make it in 50 seconds - but that's still too slow.
3. Discrete logic - was going to be extremely complex and messy.
4. A bit slice system would be very expensive to develop and implement.
5. And although a custom/semi-custom chip would be cheap to manufacture, it would be expensive to develop and would be too inflexible to allow commitment to the high volumes necessary to make this approach economically viable.

One thing we did know about implementing the RSA algorithm is that it involved lots of multiplication and so we decided to see if we could utilise a dedicated hardware multiplier/accumulator or MAC.

6. A MAC taking 100 ns for a 16 x 16 multiply was available and looked very promising. However, we quickly realised that we needed some fairly specialised hardware to drive it and feed it with data. Certainly no ordinary micro-processor would be able to keep up with the MAC's performance.

Just as we were beginning to despair the answer came to us courtesy of Texas Instruments who announced a new type of chip : the Digital Signal Processor or DSP.

DIAGRAM ONE - DSP ARCHITECTURE



7. The DSP - is a MAC and a fast microprocessor on a single chip which seemed to be the ideal combination....... The first one available was the TMS320 which has a 200ns cycle time for most instructions including multiply. Our early performance estimates suggested that with this chip five seconds for a 512 bit exponentiation should be fairly easily achievable.

THE IMPLEMENTATION

Having decided to use a DSP we have to develop a program for it. The first problem is that there are no suitable DSP compilers available and, although we might expect to eventually have to tune the assembler code to take full advantage of the DSP architecture and optimise performance, assembler is no good as a design language. Furthermore, our choice of implementation technique must take into consideration the nature of the application and in particular the requirement for integrity. With this in mind we chose to use the program development and validation techniques expounded by Prof. David Gries of Cornell University. The notation used is a combination of predicate logic and the "guarded command" form of computation guru Edsger Dijkstra.

THE ALGORITHM

In our notation the RSA algorithm can be specified in terms of pre- and post- conditions thus:

```
spec fastexp.0 (in: A,E,M; out: c);
       { pre: $0 \le A < M$ & $0 \le E$ }
       { post: $c = A^E \bmod M$ }
       endspec
```

Where the pre conditions require that: the input data A is in the range 0 to M, the modulus minus one and the exponent E is positive; and post: the output data c equals A to the power E modulo M.

The basic algorithm we will work with to satisfy these conditions is Knuth's 'square and multiply' exponentiation method with modulo reduction incorporated. Thus:

```
proc fastexp.1 (in: A,E,M; out: c);
       { pre: $0 \le A < M$ & $0 \le E$ }
       a, e, c  := A,E,1 ;
```

```
        { inv: $c * a^e \bmod M = A^E \bmod M$ }
        { bound: $t = 2 * \log_2 e + 1$ }
        do    $e \neq 0$ & $e \bmod 2 = 0 \rightarrow$
                  e,a := e div 2, a * a mod M
        ▒  $e \bmod 2 \neq 0 \rightarrow$
                  e,c := e-1, c * a mod M      od
        { post: $c = A^E \bmod M$ }
endproc
```

Notice that after initialisation of the variables the executable portion of this fastexp has been reduced to a single loop command albeit with two branches. Writing the algorithm in this very concise form which may not at first seem natural, allows us to prove its correctness more easily at a later stage.

Obviously this basic algorithm will need to be written in a substantially different form before our target DSP can execute it and in order to arrive at an assembler code version we go through a process of step-wise refinement. At each step of refinement the algorithm is re-written in a form which can be proven to be equivalent to its predecessor. In the case of our RSA algorithm most of the refinement is necessary in order to be able to represent and operate on the several hundred bit long integers within the constraints of a 16 bit architecture; the implementation of conditions, loops and other program constraints being fairly straightforward on the micro-processor-like DSP.

In order to keep our top level program simple and well structured we introduce two procedures (subroutines) which we call 'longmult' and 'longmod' to handle respectively the long integer multiplication and modulo reduction.

Here is the specification of these procedures, once again using the pre/post condition form:

```
    spec  longmult.0 (in: u,v; out: w);
          { pre:  $0 \le u,v < b^n$ }
```

```
        { post:  w = u * v }
    endspec

    spec  longmod.0 (in: w,m; out: v);
          { pre: 0 ≤ w < m^2 }
          { post: v = w mod m }
    endspec
```

THE HEART OF THE ALGORITHM

These two procedures really are the heart of the algorithm; and the key to performance is going to be their design. First let us consider what algorithm to use for long multiplication. The problem we have is similar to one we learned to solve at school. There we knew, from a memorised table, how to multiply up to 12 times 12 but faced with a larger multiplication (and assuming that we all went to school before the advent of the pocket calculator) we used a paper and pencil algorithm which went something like this (referring to diagram two): 6 times 2 is 12, 2 down carry 1, 2 times 2 is 4 plus one is 5 and so on repeating for each row, shifting one column left each time and finishing with a final addition sum. This is a fairly convenient method of hand calculation but how efficient is it?

Taking the general case of an n by n digit multiply - for each row we have to do n multiplications, 2n fetches, n + 1 stores and, n carry and add operations. Plus the final additions which require n^2 fetches and adds plus carries etc. Assuming all perations are equivalent to execute that makes in the order of $6n^2$ instructions.

Let's try it another way using the same principle but working in columns not rows and saving all the carries till we sum each column.

DIAGRAM TWO - LONG MULTIPLICATION AT SCHOOL

```
            3 0 7 8 2 6
            4 1 5 1 3 2  x
            -----------
            6 1 5 6 5 2
          9 2 3 4 7 8 -
        3 0 7 8 2 6 - -
    1 5 3 9 1 3 0 - - -
    3 0 7 8 2 6 - - - -
1 2 3 1 3 0 4 - - - - -
-----------------------
1 2 7 7 8 8 4 2 3 0 3 2
```

DIAGRAM THREE - ALTERNATIVE LONG MULTIPLICATION

```
                            3   0   7   8   2   6
                            4   1   5   1   3   2   x
                         ________________________
                            6   0  14  16   4  12
                        9   0  21  24   6  18   -
                    3   0   7   8   2   6   -   -
               15   0  35  40  10  30   -   -   -
            3   0   7   8   2   6   -   -   -   -
       12   0  28  32   8  24   -   -   -   -   -
 ________________________________________________
    1   2   7   7   8   8   4   2   3   0   3   2
```

Referring to diagram 3: here 6 times 2 is 12, 2 down 1 to carry, 2 times 2 is 4, 6 times 3 is 18, 18 plus 4 is 22 plus 1 is 23, 3 down 2 to carry and so on for the other columns. This time we have the same number of multiplies and adds but have saved a set of fetches and carries leaving an order of $4n^2$ instructions, ie a saving of 33% over the previous method. A further 50% saving can be obtained at implementation by taking advantage of a feature of the TMS320 DSP which allows auto increment and decrement of data pointers during multiply and accumulate operations - this effectively gives us the data fetching for free. Using this feature the core of our multiply program is as shown in diagram four.

In the DSP we have two auxiliary registers AR0 and AR1 which we use as data pointers and a T register which contains the multiplicand for any multiplication instructions.

The MPY * star instruction multiplies the contents of the T-register by the data pointed to by the current auxiliary register.
The LTA * star instruction loads the T register (with new data pointed to by the current auxiliary register) and adds the result of the previous multiply into the accumulator.

DIAGRAM FOUR - MULTIPLICATION PROGRAM CORE

```
MPY * +, 1
LTA * -, 0
```

```
                MPY * +, 1
                LTA * -, 0
                MPY * +, 1
                LTA * -, 0
                MPY * +, 1
```

The + and - respectively increment and decrement the current auxiliary register and the 0 or 1 at the end selects a new auxiliary register as current for the next instruction. Both arguments for each successive multiply can thus be changed for no overhead while we multiply and add; which is what we need for the column based multiplication procedure just described.

With this method we do have to ensure that we don't overflow the accumulator before the end of a column. However, it is a fairly simple calculation to work out the optimum word length to satisfy this condition.

In practice we are prevented from using 16 bit words (on the early DSP's anyway) because they take all data as being in two's compliment form. Some of the more recent DSP's do help out by providing 40 bit accumulators and unsigned arithmetic.

MODULO REDUCTION

Next let's consider the modulo reduction operation. We have an intermediate value (say W) which is the result of a long multiply calculation and we want to find the remainder when W is divided by the modulus M. That is we want:

$$X = W \bmod M = W - M * (W \operatorname{div} M)$$

where 'div' is normal integer division.

Division on a DSP is hard (that is to say expensive in time) but given that throughout any single exponentiation we will always be using the same modulus and that we have available easy or 'cheap' multiplication, we can calculate (once only for each M) R equals the reciprocal of M and subsequently obtain our result, X, by two multiplications and a subtraction:

$$X = W - M * (W * R)$$

The problem is that R in this case is a real number considerably smaller than one.

Thus, if we are to use this method we need to approximate and scale R. That is multiply R by some power of 2 and round off in order to represent R as an integer.

The trade off in this is fairly clear - the more accurately we represent R (and other intermediate values) the longer it will take to do the multiplications, the less accurately the greater the error we will have to correct at the end.

The mathematics of this trade-off are more complex than it would at first appear so I will just assume the results that we proved in our paper at Oxford.

LONGMOD PROCEDURE (refer to Diagram Five)

If M is represented as n base b digits (and therefore W is 2n base b digits) then R should be represented as the integer

$$R := b^{2n} \text{ div } M$$

Note that R here will have n + 1 digits as a result of the second precondition defining the range of M.

Next we multiply the most significant n + 1 digits of W by R and then multiply the n most significant digits of this result by M and subtracting the n + 1 least significant digits of this from the corresponding part of W. Our calculations show that the result x so obtained will always be in the range 0 to 3M - 1. In other words at most two further subtractions of M are required to give us the result we are looking for.

It is possible to show that for about 90% of the values of W and M, the initial value of X obtained will be less than M and that only in 1% of cases will X exceed 2M and thus require two correcting subtractions.

DIAGRAM FIVE - LONGMOD

```
proc longmod.1 (in: w, m; out: x);
    { pre: $0 \le w < m^2$ & $b^{n-1} < m < b^n$ & $3 < b$ }
    r := $b^{2n}$ div m ;
    y := w div $2^{n-1}$ * r ;
    x := w mod $b^{n+1}$ - m * y div $b^{n+1}$ ;
    { $0 \le x < 3m$ }
    do $x \ge m \rightarrow$ x := x - m od
    { x = w mod m }
endproc
```

It can be seen from all this that for large n this modulo reduction method takes about the same time to execute as two long multiplications. Actually we can do almost twice as well as this by only calculating half the product in each long multiplication since the other half of each product is not required.

Thus, apart from the small overhead of calculating the reciprocal R (which could of course be done in advance and stored with its corresponding M as part of the RSA key) the modulo calculation is not much slower than the long multiplication.

FASTEXP CONTINUED

Returning now to the top level Fastexp algorithm. If we represent the exponent E as a sequence of n base b digits where $b = 2^f$ then our next requirement of the algorithm will require two nested loops to take care of respectively the digits and bits of E. Skipping a couple of refinement steps, our fastexp procedure is as shown in Diagram six,

DIAGRAM SIX - PROC FASTEXP.4

```
proc fastexp.4 (in: A E M; out: c);
    { pre: $0 \le A < M$ & $0 \le E$ }
    $(e_{n-1} \ldots e_0)_b$ := E;
    a,c,i := A,I,O;
```

```
        do $(e_{n-1} \ldots e_i)_b = 0 \rightarrow$
           $(ei_{f-1} \ldots ei_0)_2 := ei$ ;
           $j := 0$;
           do $j < f \rightarrow$
              if $ei_j = 0 \rightarrow$ skip
              ▯ $ei_j = 0 \rightarrow c := c * a$ mod $M$
              fi;
              $a := a * a$ mod $M$;
              $j := j + 1$                 od;
           $i := i + 1$                 od
        { post: $c = A^E$ mod $M$ }
    endproc
```

----

which with a few further refinements, including insertion of our subroutines longmult and longmod and globalisation of the data (to save on parameter passing), can be translated almost directly into the TMS320 assembler code listed in Diagram seven. Notice how simple the program appears.

DIAGRAM SEVEN - PROC FASTEXP.7

```
*       proc fastexp.7(var A,E,M,R,C)
*
EXP          LARP 1             use AR1 as a counter
        LAR  AR1,N      to initialize C
        MAR  *-         AR1 := N-1
        LACK C0         C0 is XRAM relative address of C0
        ADDS DATA0      DATA0 is XRAM data page address
        ADDS N          ACC is pointer to CAR1
*
LOOP1   SUBS ONE        decrement ACC
        TBLW ZERO       "CN-1 ... C0 := 0"
        BANZ LOOP1      repeat LOOP1 while AR1>0 and dec AR1
ENDL1
        TBLW ONE        "C0 := 1"
        ZAC
        SACL I          "i := 0"
```

```
*
LOOP2    LACK E0         E0 is XRAM relative address of E0
         ADDS DATA0
         ADDS I
         TBLR EI         "EI := Ei"
         LACK F
         SUBS ONE
         SACL J          "j := f-1"
*
LOOP3    ZALS EI
         AND  ONE        "ACC := EI0"
         BZ   LSB0       "if ACC = 0 → skip" (to LSB0)
                         "if ACC = 1 →"
LSB1          LACK C0
         ADDS DATA0
         SACL X          X := address of C0
         CALL LONMUL     "call longmult(C)"
         CALL LONMOD     "call longmod(C)"

*
LSB0          LACK A0
         ADDS DATA0
         SACL X          X := address of A0
         CALL LONMUL     "call longmult(A)"
         CALL LONMOD     "call longmod(A)"

*
         LAC  EI,15
         SACH EI         "EI := EI div 2"
         ZALS J
         SUBS ONE
         SACL J          "j := j-1"
         BGEZ LOOP3      "repeat LOOP3 while j>0"


ENDL3
         ZALS I
         ADDS ONE
         SACL I          "i := i+1"
         SUBS NE
         BLZ  LOOP2      "repeat LOOP2 while i<ne"
ENDL2
*        endproc
```

There are only 43 machine code instructions required apart from the multiplcation and modulo procedures.

This simplicity is, another direct benefit of the rigourous development methodology employed.

PERFORMANCE AND SECOND GENERATION DSP'S

This implementation of 'fastexp' takes on average (that is with an exponent composed of half 0's and half 1's) 2.6 seconds to execute with 512 bit modulus and exponent on a Texas Instruments TMS32010 running at its maximum clock rate of 20 MHz. The 32010 (originally just called the TMS320) was the first general purpose DSP on the market but second generation DSP's are appearing now from most manufacturers and speed calculations using our algorithm suggest that times below 1 second will be possible on the TMS320C25 and below one quarter of a second on the Motorola DSP56200 which has a 24 x 24 multiplier and 56 bit accumulator.

The third (or is it fifth?) generation DSP from Inmos (the IMSA 100) which is part of the Transputer family, has on board no less than 32 16 x 16 multiplier/accumulators and should prove to be the fastest yet once we have refined our algorithm into the OCCAM parallel processing language which is executed directly by the transputer hardware.

CUSTOM CHIPS

Finally, I know that I started this presentation by stating that we decided against a custom silicon RSA implementation on the grounds of development cost and inflexibility, but a number of developments have taken place since we originally came to that conclusion. Most importantly the advent of silicon compilers and low volume custom silicon processes has reduced the turnaround time and development cost to a point where manufacture of a few hundred chips is a viable proposition. Furthermore, the increase in demand for fast RSA solutions plus the ultimate unit cost and performance advantages has led Computer Security Limited's sister company, RAANND Systems Ltd, to develop a custom RSA chip. Dr Gordon Rankine, the Managing Director of RAANND and the architect of this RSA chip, code named Thomas, has documented his presentation of this design elsewhere in the proceedings.

REFERENCES

R L Rivest, A Shamir and L Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications ACM Vol 21 (2) (Feb 1978)

Dorothy E R Denning, "Cryptography and Data Security", Addisson-Wesley (1983)

Texas Instruments, "TMS 32010 User's Guide"' (1983)

Donald E Knuth, "The Art of Computer Programming Volume 2-Seminumerical Algorithms", Addisson-Wesley (second edition - 1981)

P D Barrett, "Communications Authentication and Security using Public Key Encryption - A Design for Implementation." (Oxford University Programming Research Group MSc Thesis (1984)

C A R Hoare, "Notes of Communicating Sequential Processes", Oxford University Computing Laboratory (1983)

David Gries, "The Science of Computer Programming", Springer-Verlay (1981)

Edsger Dijkstra, "A Discipline of Programming", Prentice Hall (1976)