A Crypto-Engine

George I. Davida
Frank B. Dancs

University of Wisconsin-Milwaukee
Milwaukee, WI 53201

Abstract

In this paper we present a design for a crypto-engine. We shall discuss the design and show the instruction set of this coprocessor and then show how this could be used to implement most of the known encryption algorithms. We will discuss why a coprocessor approach may be a better solution than adoption of specific encryption algorithms which can be broken or decertified.

I. Introduction

With the ever increasing use of encryption techniques to safeguard data, a need has become apparent for special hardware to implement these schemes due to the inefficiency of software methods. Special hardware such as DES chips have the draw back of being useful for only one encryption technique. Many have discussed the weekness of the method and site potential problems with the sboxs. Furthermore, if DES is not certified as the standard for encryption, many existing chips will become obsolete.

Taking this into consideration, a hardware design that could increase the efficiency of computation while allowing versatility for many existing encryption algorithms and possibly future ones has been considered. Much like a floating point coprocessor, we have designed an encryption coprocessor. With basic instructions common to the variety of algorithms used today, the design presented in this paper facilitates implementing of most encryption algorithms.

The design of this coprocessor has been based on the Motorola 68000[1] coprocessor protocol. It has been implemented in software for testing purposes. Implementation details will also be discussed later in the paper.

The research reported in this paper was supported in part by NSF grant DCR-8504620.

C. Pomerance (Ed.): Advances in Cryptology - CRYPTO '87, LNCS 293, pp. 257-268, 1988.

II. Need For a Coprocessor

With chip prices coming down, replacing software tools with hardware tools has become more practical. The problem with hardware is that it may not always be as flexible as one would want. Take for example the DES chips that are on the market. Although they are excellent for preforming their assigned tasks, they can only preform one task and that is encrypting data with the DES standard. If this standard is broken or decertified as the standard, these chips would be obsolete. Replacement costs for this specialized hardware could be expensive.

Unlike other types of hardware, encryption hardware has the drawback of potentially becoming useless with the ever possible chance that the algorithm the hardware implements is broken. Other types of hardware could become obsolete with the advent of a new and better device, however; the old hardware is still useful.

The other disadvantage of a standard encryption chip is that standards can be changed. A specialized chip can only implement one encryption technique. If the standard would change then those using the standard would not be able to communicate with those using the old standard. Since reprogramming is not possible, this desired versatility cannot be achieved by these specialized chips. After all, what is the difference between a calculator and a computer? It is not the greater capability of the computer or its faster processing. It is the versatility of the computer brought about by the ability to be programmed for many different applications.

With this in mind a coprocessor with special instructions that would be useful for encryption appears to be a better solution. If a standard encryption technique is decertified a coprocessor could easily be reprogrammed. A new technique could easily be installed. Not only could a new standard be adopted without any hardware changes, different techniques could be used simultaneously for different applications. Furthermore, techniques could be altered to meet specific requirements of local environments. The advantage over just pure software implementations of the encryption techniques is, of course, special instructions implemented in hardware would bring about a speed advantage.

III. Coprocessor Design

To make a hardware device that can implement most of the known algorithms of today and still leave room for tomorrows, we need to consider what are the basic operations of encryption algorithms. There are two basic operations that are used for encryption. They are substitution and transposition [2]. Most crytographic techniques use a combination of both.

Substitution operations can be table lookup, many types of arithmetic operations such as multiplication, exponentiation, etc. Transposition operations on the other hand are of the form, permutation on the bits of a word, shift registers, and modular arithmetic, where the permutation is of the entire message space.

With this in mind, we propose a set of encryption primitives. The substitution instructions include all arithmetic in large register form, an actual table lookup and its supporting instructions, and three of the boolean operators, *and*, *or*, and *xor*. For the permutation operations, we included a bit permuter, three different shift operations, and the modular arithmetic operation.

IV. Design Details

The registers, illustrated below, that will be needed to support these instructions must be of a larger than normal size. For example, RSA uses numbers approximately of 200 decimal digits large. For this type of arithmetic, there are four large registers, 1024 bits each. This will allow for roughly 300 decimal digits. These registers will support all the arithmetic instructions. There will also be the need for smaller registers for substitution and permutation operations. There are five 128 bit registers. One of these registers, the general register, can be addressed as four 32 bit registers, two 64 bit registers, or the entire 128 bit register. Finally, there are 16 tables consisting of 256 bytes each supporting the table lookup operation.

Register Layout of the Encryption Coprocessor

The larger register layout

1024 bits L1 Large Purpose register
1024 bits L2 Large Purpose register
1024 bits L3 Large Purpose register
1024 bits L4 Large Purpose register (ML) Modules constant

General register, key register and modular constant layout (128 bits)

Ghl(high,left)	Ghr(high,right)	Gll(low,left)	Glr(low,right)
128 bits K0 key register			
128 bits K1 key register			
128 bits K2 key register			
128 bits MS Small Modular constant register			

Sbox Array

Array For Slice 1

byte 0	byte 1	byte 2	byte 3	. . .	byte 255

Array For Slice 2

byte 0	byte 1	byte 2	byte 3	. . .	byte 255

. . .
. . .
. . .

Array For Slice 16

byte 0	byte 1	byte 2	byte 3	. . .	byte 255

The instructions listed in Appendix A are the instructions that we have considered necessary to fulfill the requirements of our list of basic operations. The instructions listed in the Appendix follow these rules of use. Those that have <reg>, <greg>, or <lreg> can support the register direct addressing mode on any register, the general re-

gister only, and one of the large registers only, respectively. The instructions that have <reg1> or <reg2> as their operands have the following addressing modes: register direct, register indirect, memory immediate, and memory indirect. Furthermore, the indirect addressing mode using the main processors registers *a0*, *a6*, and *a7* has the following subdivisions: register indirect, register indirect with auto-increment, register indirect with auto-decrement, and register indirect with index.

V. Detailed Description Of Some Selected Instructions

The brief description of each instruction presented in the Appendix may not give enough information on some of the more complex instructions, and may not completely show the versatility of them.

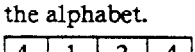
The sbox substitution is probably the most complex operation. The *esbox* instruction can do substitution on a number of bit combinations. The programmer can choose from 8, 6, or 4 bits into the substitution and 8, 6, or 4 bits out of the substitution. The number of input bits does not have to be the same as the number of output bits. To initialize this we will need the *einitsbox* instruction. This will set up the *esbox* instruction for operation with one of these combinations of input bits and output bits. We will refer to a string of bits that will be used for one atomic substitution, either 8, 6, or 4 bits in size, as a substitution word.

As seen by the register layout diagram, there are 16 sbox arrays, each having 256 bytes of storage. Each of these arrays are to be used for one substitution. That is, one substitution word will be an index to one of these sbox arrays. When the substitution word indexs one of the bytes in the array, 8 bits, 6 bits, or 4 bits will be used depending on the number of bits that has been set up by the *einitsbox* instruction for output.

All of the memory in the sbox array will not be utilized for every substitution combination. For example, if a programmer where to initialize this operation to 4 bits in and 8 bits out with a 64 bit register, all of the 16 sbox arrays would be used, however, only 16 bytes out of each array would be used.

To load the sbox arrays the *eldsbox* instruction will be used. The number of the array will be needed as well as the number of bytes in that array. In the example above, there will be 16 *eldsbox* instructions needed to load all of the arrays. Each of these instruction will specify *n-1* as the number of bytes to load and *m* for the slice number. The only address mode that will be allowed is memory indirect. This will allow the programmer to initialize their sbox arrays somewhere in memory with a label, and use that label with the instruction.

There are two permutation instructions, *eperms* and *epermd*, representing control from the source and control for the destination. These will be used for two different types of permutations. The *eperms* instruction will allow permutation of one bit to multiple bits, while, the *epermd* instruction, will allow multiple bits to be permuted to one bit. Below is a diagram that should clear up the two different permutation instructions. Note, only four bits of the general register is used, and the first four bytes of the *l1* registers are used. In the *l1* registers we will store the following first four control bytes, these will control the first four bits in the general register signified by a letter in the alphabet.



This is what is stored in the first four bits of the original general register, *g*.

A	B	C	D

Permuting the original with the instruction: *epermś l1,g*

D	A	B	D

Permuting the original with the instruction: *epermd l1,g*

B	C	0	A ⊕ D

The example above points out the situation for the *epermd* instruction where no bit was assigned to destination of the third bit location. In this case, the third location obtains the value zero. Furthermore, if two or more bits go to the same destination, the values of those two or more bits are *xor*ed together.

The *epermś* instruction would be used in something like the sbox expansion of DES, and the *epermd* instruction would be used in something like a linear shift register with a finite state machine.

VI. Details of Various Algorithm Implementation

The following implementations are written in the high level language of C and the program listings are found in Appendix B. The asm is a construct that allows a programmer to give an instruction directly to the assembler. After compiling this code the modified assembler can translate the coprocessor instructions into machine language. All of the coprocessors instructions are noted by italics. Note, some parts of main line, functions, and array initializations are omitted to conserve space.

DES[3] was an important consideration in the design of the coprocessor. As long as DES is the standard there will be a need to implement it.

A few points need to be noted for clarity. First, the sbox expansion is done like a standard DES sbox expansion. However, after the *xor* with the key, the bits must be permuted again to facilitate the b5b0b4b3b2b1 pattern. In other words, since the sbox substitution instruction of the coprocessor does a direct table lookup on the address that the six given bits generate, they must be permuted to follow the pattern that is necessary to follow the DES algorithm. This is because the algorithm calls for the first and last bit of the six bit substitution word be the index to the row and the middle bits be the index to the column. In the coprocessor the data is layed out continuously instead of having a row and column; thus, the last bit needs to be moved to the second bit. Since the large registers are capable of holding a 128 bit permutation, the sbox_expansion array will hold both permutations, which in turn is moved to the *l4* register.

The RSA implementation is pretty straight forward since the algorithm is nothing more than:

$C = M^e \bmod n$ (encrypt)
$M = C^d \bmod n$ (decrypt)

where *e* is the encryption key and *d* is the decryption key. This means that the only steps that need to be done are load one of the large registers with the data, do an exponential instruction with an automatic modular arithmetic with the *ml* register, and move the data from the large register back into memory.

The Pohlig-Hellman scheme is another exponential encryption technique; however, it could be implemented in two different ways. The basic algorithm is [4]

$C = M^e \bmod p$

$M = C^d \bmod p$

where *e* is the enciphering key and *d* is the deciphering key. The *p* could either be a prime or a $GF(2^m)$ irreducible polynomial. Either way, it could be implemented quickly and easily.

The first implementation, where p is a prime, could be done exactly like the RSA listed in Appendix B, except ml would receive the prime value. To do the next implementation, do the exact implementation as the RSA example with these two changes. Put a irreducible polynomial in *ml*, and replace the following line:

asm(" *eexpml* *l1,l2*");

with

asm(" *eexpgfml* *l1,l2*");

VII. Remarks

Many other algorithms could be implemented. For example, the Shamir [5] Lagrange Interpolating Polynomial Scheme could be done with the arithmetic under Galois Fields instructions. All the necessary instructions are available to implement this scheme: addition, division and multiplication all in $GF(2^m)$. Consider the irreducible polynomial $p(x) = x^3 + x + 1$; the corresponding binary representation would be 1011. This number would be put in the *ml* or *ms* register depending on the size of the integer arithmetic. This would then mean that all the arithmetic would be done under the Galois Field $GF(2^3)$ with 1011 as the irreducible polynomial.

Another method for encryption could be implemented just as easily. The Block Ciphers with Subkeys [6] could be implemented with the basic arithmetic mod instructions. In this scheme the basic operations are inverse (*edivml*), multiplication (*emultml*), and addition (*eaddml*).

The software implementation was done on a Motorola 68010 based system. The 68000 protocol for coprocessor instruction identification is what is referred to as an F-line instruction. This F-line instruction will precede any coprocessor instruction. It will have the coprocessor identification number as well as other information. If the coprocessor doesn't exist in hardware, the processor will invoke a F-line emulation trap (vector 11). With this, we can trap the instruction in the kernel and start the emulation process. On a UNIX bsd 4.2 system this is just a quick addition in the trap.c code to capture the interrupt and a function call to the emulation routines. In the emulation routines, the first step is to check what the instruction is and call the appropriate emulation function.

The implementation of the Galois fields followed the Scott, Stafford and Peppard[7] algorithm for multiplication and the Davida and Litow[8] algorithm for inverse. Other implementations of arithmetic were done by methods that made the software easiest to write, not considering hardware problems.

Appendix A

Instruction Set

eadd <reg1>,<reg2> — The eadd instruction adds the source <reg1> to the destination <reg2> and stores the result in the

destination <reg2>.

eaddms <reg1>,<reg2>
eaddml <reg1>,<reg2>

The eaddms and eaddml instructions do the same addition mod the (ms) and (ml) registers respectively.

eaddgfms <reg1>,<reg2>
eaddgfml <reg1>,<reg2>

The eaddgfms and eaddgfml instructions do the addition in the Galios Field GF(2^m) using the (ms) and (ml) registers respectively to store the irreducible polynomial P(x).

esub <reg1>,<reg2>

The esub instruction subtracts the source <reg1> from the destination <reg2> and stores the result in the destination <reg2>.

esubms <reg1>,<reg2>
esubml <reg1>,<reg2>

The esubms and esubml instructions do the same subtraction mod the (ms) and (ml) registers respectively.

esubgfms <reg1>,<reg2>
esubgfml <reg1>,<reg2>

The esubgfms and esubgfml instructions do the subtraction in the Galios Field GF(2^m) using the (ms) and (ml) registers respectively to store the irreducible polynomial P(x).

emult <reg1>,<reg2>

The emult instruction multiplies the source <reg1> to the destination(reg2) and stores the result in the destination <reg2>.

emultms <reg1>,<reg2>
emultml <reg1>,<reg2>

The emultms and emultml instructions do the same multiplication mod the (ms) and (ml) registers respectively.

emultgfms <reg1>,<reg2>
emultgfml <reg1>,<reg2>

The emultgfms and emultgfml instructions do the multiplication in the Galios Field GF(2^m) using the (ms) and (ml) registers respectively to store the irreducible polynomial P(x).

ediv <reg1>,<reg2>

The ediv instruction divides the source <reg1> into the destination <reg2> and stores the result in the destination <reg2>.

edivms <reg1>,<reg2>
edivml <reg1>,<reg2>

The edivms and edivml instructions do the same division mod the (ms) and (ml) registers respectively.

edivgfms <reg1>,<reg2>
edivgfml <reg1>,<reg2>

The edivgfms and edivgfml instructions do the division in the Galios Field GF(2^m) using the (ms) and (ml) registers respectively to store the irreducible polynomial P(x).

eexp <reg1>,<reg2>

The eexp instruction takes destination <reg2> to the power of the source <reg1> and stores the result in the destination <reg2>.

eexpms <reg1>,<reg2>

eexpml <reg1>,<reg2>

The eexpms and eexpml instructions do the same exponentiation mod the (ms) and (ml) registers respectively.

eexpgfms <reg1>,<reg2>
eexpgfml <reg1>,<reg2>

The eexpgfms and eexpgfml instructions do the exponentiation in the Galios Field GF(2^m) using the (ms) and (ml) registers respectively to store the irreducible polynomial P(x).

eunsetsign
esetsigned

These two instructions will set all arithmetic in an unsigned mode and set all arithmetic in a signed mode, respectively.

emod <reg1>,<reg2>

The emod instruction will do a modular arithmetic operation. The destination <reg2> mod source <reg1> and place the results in the destination <reg2>.

emov <reg1>,<reg2>

The emov instruction will move <reg1> to <reg2>. This can also be used to load registers with the different address modes available.

eldsbox #array,#bytes,_memory_location

The eldsbox instruction will load one of the 16 sbox arrays. It will load the amount of bytes in that table that is specified by the #bytes field. The only addressing mode allowed is memory indirect, thus the program will have to give the location of where the sbox array is stored in memory.

einitsbox #c1,#c2

The einitsbox instruction will initialize the control for the sbox instruction. The #c1 will be the number of bits inputed for each substitution in the sbox operation and #c2 will be the number of bits output for each substitution. These bits can have the values four, six and eight.

esbox <greg>

The esbox instruction does a sbox array lookup with the <greg> and stores the result in the same register.

eperms <lreg>,<reg>

The eperm instruction will do a permutation on a the destination <reg> with the large control register <lreg>. Each byte in the control register will control where each bit in the destination <reg> will come from.

epermd <lreg>,<reg>

The eperm instruction will do a permutation on a the destination <reg> with the large control register <lreg>. Each byte in the control register will control the destination of corresponding bit in the destination <reg>.

eand <reg1>,<reg2>

The eand instruction will do a bitwise *and* on the two registers given and stores the result in the destination <reg2>.

eor <reg1>,<reg2>

The eor instruction will do a bitwise *or* on the two registers given and stores the result in the destination <reg2>.

exor <reg1>,<reg2>

The exor instruction will do a bitwise *xor* on the two registers given and stores the result in the destination <reg2>.

elshiftl #num of bits,<reg>
ershiftl #num of bits,<reg>

The elshiftl and ershiftl will do a left or right respectively logical shift on the register specified by the number of bits given.

elshiftc #num of bits,<reg>
ershiftc #num of bits,<reg>

The elshiftc and ershiftc will do a left or right respectively circular shift on the register specified by the number of bits given.

elshifta #num of bits,<reg>
ershifta #num of bits,<reg>

The elshifta and ershifta will do a left or right respectively arithmetic shift on the register specified by the number of bits given.

Appendix B

DES Source Listing

```
#include <stdio.h>

char    g[] = { 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
                0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
unsigned int    k0[32];
unsigned int    k1[4];
char            initial[] = { /* initial permutation control values here */ };

char            final[] = { /* final permutation control values here */ };

char            sbox_expansion[] = {127, 126, 125, 124, 123, 122, 121, 120,
                        119, 118, 117, 116, 115, 114, 113, 112, 47, 42, 46,
                        45, 44, 43, 41, 36, 40, 39, 38, 37, 35, 30, 34, 33,
                        32, 31, 29, 24, 28, 27, 26, 25, 23, 18, 22, 21, 20,
                        19, 17, 12, 16, 15, 14, 13, 11, 6, 10, 9, 8, 7, 5,
                        0, 4, 3, 2, 1, 63, 62, 61, 60, 59, 58, 57, 56, 55,
                        54, 53, 52, 51, 50, 49, 48, 0, 31, 30, 29, 28, 27,
                        28, 27, 26, 25, 24, 23, 24, 23, 22, 21, 20, 19, 20,
                        19, 18, 17, 16, 15, 16, 15, 14, 13, 12, 11, 12, 11,
                        10, 9, 8, 7, 8, 7, 6, 5, 4, 3, 4,  3, 2, 1, 0, 31};

char            internal[] = { /* internal permutation control values here */ };

char pc_1[]     =       { /* key pc-1 permutation value here */ };

char pc_2[]     =       { /* key pc-2 permutation control values here */ };

char sbox[8][64] =      {{13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,
                          1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
                          7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
                          2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11},

                         { /* sbox 7 */ }
                         { /* sbox 6 */ }
                         { /* sbox 5 */ }
```

```
                        { /* sbox 4 */ }
                        { /* sbox 3 */ }
                        { /* sbox 2 */ }
                        { /* sbox 1 */ }};

main(argc,argv)
int argc;
char *argv[];
{
FILE *fpin, *fpout, *fkey, *fopen();
int iterations;
int i,n,flag;

        /* open input and output files here */

        do_key(flag);

        asm(" eunsetsign    ");
        asm(" eldsbox _sbox,     #0,#63");
        asm(" eldsbox _sbox+64,  #1,#63");
        asm(" eldsbox _sbox+128, #2,#63");
        asm(" eldsbox _sbox+192, #3,#63");
        asm(" eldsbox _sbox+256, #4,#63");
        asm(" eldsbox _sbox+320, #5,#63");
        asm(" eldsbox _sbox+384, #6,#63");
        asm(" eldsbox _sbox+448, #7,#63");
        asm(" einitsbox #6,#4");
        asm(" emov   _initial,l1");
        asm(" emov   _final,l2");
        asm(" emov   _internal,l3");
        asm(" emov   _sbox_expansion,l4");

        while((n = getdata(fpin,flag,g)) > 0 ) {
                asm(" emov   _g,g");
                asm(" eperms  l1,g");
                for(iterations = 0; iterations < 16; iterations++) {
                        k1[0] = k0[t*2];
                        k1[1] = k0[t*2+1];

                        asm(" emov   g,k0");
                        asm(" emov   #0,gh");
                        asm(" emov   #0,gl");
                        asm(" eperms l4,gl");
                        asm(" exor   _k1,gl");
                        asm(" eperms l4,g");
                        asm(" emov   gh,gl");
                        asm(" esbox  gl");

                        asm(" eperms l3,gl");
                        asm(" emov   k0,gh");
                        asm(" exor   ghl,glr");

                        if(iterations == 15) {
                                asm(" emov   glr,gll");
                                asm(" emov   ghr,glr");
                        } else
                                asm(" emov   ghr,gll");
                }
                asm(" eperms l2,g");
                asm(" emov   g,_g");
                putdata(fpout,flag,8,g);
        }

}
/* getdata function */
```

```
/* putdata function */

do_key(flag)
        /* read in key and make 16 subkeys that are placed in the k0 array.
         * Note, this function will be done much like the main line */
```

RSA Source Listing

```
#include <stdio.h>

#define SIZE 128

unsigned int m1[32];
unsigned int l1[32];
char l2[SIZE];

main(argc,argv)
int argc;
char *argv[];
{
FILE *fpin, *fpout, *fkey, *fopen();
char c;
int n,size;

        /* open files here for data in, data out */

        read_key(stdin,m1);
        size = find_m1_size(m1);
        read_key(stdin,l1);

        asm(" eunsetsign   ");
        asm(" emov   _m1,m1");
        asm(" emov   _l1,l1");

        while((n = getdata(fpin,flag,size,l2)) > 0 ) {

                asm(" emov   _l2,l2");
                asm(" eexpml l2,l1");
                asm(" emov   l2,_l2");
                putdata(fpout,flag,n,l2);
        }

}

/* find_m1_size, determines the size of the modulus so we don't read more
 * into the large register than we have room for in the modulus */
find_m1_size(m1)

        /* body of routine */
```

References

1. Motorola Inc, *MC68020 32-bit Microprocessor User's Manual*, Prentice-Hall Inc., Englewood Cliffs (1985).
2. D. Denning, *Cryptography and Computer Security*, Addison Wesley, Reading, MA (1982).
3. NBS, "Data Encryption Standard," *National Bureau of Standards*, FIPS PUB 46, (Jan 1979).
4. S. C. Pohlig and M. E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance," *IEEE Transactions on Information Theory* IT-24(January 1978).
5. A. Shamir, "How to Share a Secret," *Communications of the ACM* 22 pp. 612-613 (November 1979).
6. G. I. Davida, L. D. Wells, and J. B. Kam, "A Database Encryption System with Subkeys," *ACM Trans. on Database Syst.* 6(2) pp. 312-328 (June 1981).
7. P. Scott, "A Fast VLSI Multiplier for GF(2^m)," *IEEE Journal on Selected Areas in Communications* SAC-4 pp. 62-65 (January 1986).
8. G. I. Davida and B. Litow, "Fast Parallel Inversion in Finite Fields," *CISS, The Johns Hopkins University*, (Davi85).