The Tiny Encryption Algorithm (TEA)

One of the most secure cipher algorithms ever devised ...
... and certainly the simplest!

Simon Shepherd, Professor of Computational Mathematics
Director of the Cryptography and Computer Security Laboratory,
Bradford University, England.

What is the Tiny Encryption Algorithm?

The Tiny Encryption Algorithm is one of the fastest and most efficient cryptographic algorithms in existence. It was developed by David Wheeler and Roger Needham at the Computer Laboratory of Cambridge University. It is a Feistel cipher which uses operations from mixed (orthogonal) algebraic groups - XOR, ADD and SHIFT in this case. This is a very clever way of providing Shannon's twin properties of diffusion and confusion which are necessary for a secure block cipher, without the explicit need for P-boxes and S-boxes respectively. It encrypts 64 data bits at a time using a 128-bit key. It seems highly resistant to differential cryptanalysis, and achieves complete diffusion (where a one bit difference in the plaintext will cause approximately 32 bit differences in the ciphertext) after only six rounds. Performance on a modern desktop computer or workstation is very impressive.

You can obtain a copy of Roger Needham and David Wheeler's original paper describing TEA, from the Security Group ftp site at the world-famous Cambridge Computer Laboratory at Cambridge University. There's also a paper on extended variants of TEA which addresses a couple of minor weaknesses (irrelevant in almost all real-world applications), and introduces a block variant of the algorithm which can be even faster in some circumstances.

How secure is TEA?

Very. There have been no known successful cryptanalyses of TEA. It's believed to be as secure as the IDEA algorithm, designed by Massey and Xuejia Lai. It uses the same mixed algebraic groups technique as IDEA, but it's very much simpler, hence faster. Also it's public domain, whereas IDEA is patented by Ascom-Tech AG in Switzerland. IBM's Don Coppersmith and Massey independently showed that mixing operations from orthogonal algebraic groups performs the diffusion and confusion functions that a traditional block cipher would implement with P- and S-boxes. As a simple plug-in encryption routine, it's great. The code is lightweight and portable enough to be used just about anywhere. It even makes a great random number generator for Monte Carlo simulations and the like. The minor weaknesses identified by David Wagner at Berkeley are unlikely to have any impact in the real world, and you can always implement the new variant TEA which addresses them. If you want a low-overhead end-to-end cipher (for real-time data, for example), then TEA fits the bill.
TEA, a Tiny Encryption Algorithm

David J. Wheeler
Roger M. Needham
Computer Laboratory
Cambridge University
England

Abstract. We give a short routine which is based on a Feistel iteration and uses a large number of rounds to get security with simplicity.

Introduction
We design a short program which will run on most machines and encypher safely. It uses a large number of iterations rather than a complicated program. It is hoped that it can easily be translated into most languages in a compatible way. The first program is given below. It uses little set up time and does a weak non linear iteration enough rounds to make it secure. There are no preset tables or long set up times. It assumes 32 bit words.

Encode Routine
Routine, written in the C language, for encoding with key k[0] - k[3]. Data in v[0] and v[1].

```c
void code(long* v, long* k) {
unsigned long y=v[0],z=v[1], sum=0, /* set up */
delta=0x9e3779b9, /* a key schedule constant */
n=32;
while (n-->0) { /* basic cycle start */
    sum += delta;
    y += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
    z += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
} /* end cycle */
v[0]=y ; v[1]=z ; }
```

Basics of the routine
It is a Feistel type routine although addition and subtraction are used as the reversible operators rather than XOR. The routine relies on the alternate use of XOR and ADD to provide nonlinearity. A dual shift causes all bits of the key and data to be mixed repeatedly.

The number of rounds before a single bit change of the data or key has spread very close to 32 is at most six, so that sixteen cycles may suffice and we suggest 32.
The key is set at 128 bits as this is enough to prevent simple search techniques being effective.

The top 5 and bottom four bits are probably slightly weaker than the middle bits. These bits are generated from only two versions of z (or y) instead of three, plus the other y or z. Thus the convergence rate to even diffusion is slower. However the shifting evens this out with perhaps a delay of one or two extra cycles.

The key scheduling uses addition, and is applied to the unshifted z rather than the other uses of the key. In some tests k[0] etc. were changed by addition, but this version is simpler and seems as effective. The number delta, derived from the golden number is used where

\[ \text{delta} = (\sqrt{5} - 1)2^{31} \]

A different multiple of delta is used in each round so that no bit of the multiple will not change frequently. We suspect the algorithm is not very sensitive to the value of delta and we merely need to avoid a bad value. It will be noted that delta turns out to be odd with truncation or nearest rounding, so no extra precautions are needed to ensure that all the digits of sum change.

The use of multiplication is an effective mixer, but needs shifts anyway. It was about twice as slow per cycle on our implementation and more complicated.

The use of a table look up in the cycle was investigated. There is the possibility of a delay ere one entry of the table is used. For example if k[z&3] is used instead of k[0], there is a chance one element may not be used of \((3/4)^{32}\), and a much higher chance that the use is delayed appreciably. The table also needed preparation from the key. Large tables were thought to be undesirable due to the set up time and complication.

The algorithm will easily translate into assembly code as long as the exclusive or is an operation. The hardware implementation is not difficult, and is of the same order of complexity as DES [1], taking into account the double length key.

Usage

This type of algorithm can replace DES in software, and is short enough to write into almost any program on any computer. Although speed is not a strong objective with 32 cycles (64 rounds), on one implementation it is three times as fast as a good software implementation of DES which has 16 rounds.

The modes of use of DES are all applicable. The cycle count can readily be varied, or even made part of the key. It is expected that security can be enhanced by increasing the number of iterations.

Selection of Algorithm

A considerable number of small algorithms were tried and the selected one is neither the fastest, nor the shortest but is thought to be the best compromise for safety, ease of implementation, lack of specialised tables, and reasonable
performance. On languages which lack shifts and XOR it will be difficult to code. Standard C does make an arithmetic right shift and overflows implementation dependent so that the right shift is logical and y and z are unsigned.

Analysis

A few tests were run to detect when a single change had propagated to 32 changes within a small margin. Also some loop tests were run including a differential loop test to determine loop closures. These tests failed to show any unexpected behaviour.

The shifts and XOR cause changes to be propagated left and right, and a single change will have propagated the full word in about 4 iterations. Measurements showed the diffusion was complete at about six iterations.

There was also a cycle test using up to 34 of the bits to find the lengths of the cycles. A more powerful version found the cycle length of the differential function.

\[ d(x) = f(x \ XOR \ 2^p) \ XOR \ f(x) \]

which may test the resistance to some forms of differential crypto-analysis \[2\].

Conclusions

We present a simple algorithm which can be translated into a number of different languages and assembly languages very easily. It is short enough to be programmed from memory or a copy. It is hoped it is safe because of the number of cycles in the encoding and length of key. It uses a sequence of word operations rather than wasting the power of a computer by doing byte or 4 bit operations.

Acknowledgements

Thanks are due to Mike Roe and other colleagues who helped in discussion and tests and to the helpful improvements suggested by the editor.

References

2 E. Biham and A. Shamir, Differential Analysis of the Data Encryption Standard, Springer-Verlag, 1993
Appendix

Decode Routine

void decode(long* v, long* k) {
    unsigned long n=32, sum, y=v[0], z=v[1],
    delta=0x9e3779b9;
    sum=delta<<5;
        /* start cycle */
    while (n-->0) {
        z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
        y-= ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
        sum-=delta;
    }        /* end cycle */
    v[0]=y; v[1]=z;
}

Implementation Notes

It can be shortened, or made faster, but we hope this version is the simplest to implement or remember.

A simple improvement is to copy k[0-3] into a,b,c,d before the iteration so that the indexing is taken out of the loop. In one implementation it reduced the time by about 1/6th.

It can be implemented as a couple of macros, which would remove the calling overheads.
Tea extensions

Roger M. Needham and David J. Wheeler

Notes October 1996
Revised March 1997
Corrected October 1997

Introduction
The two minor weaknesses of TEA pointed out by David Wagner are eliminated (if desired) by some minor changes.

Extensions to TEA
The mixing portion of TEA seems unbroken but related key attacks are possible even though the construction of $2^{32}$ texts under two related keys seems impractical. The second weakness, that the effective length of the keys is 126 bits not 128 does affect certain potential applications but not the simple cypher decipher mode.

To cater for these weaknesses, we can readily adjust the algorithm, while trying to retain the original objectives of little set up time, simplicity and using a large number of rounds to prevent attacks, and avoid complicated analysis.

The first adjustment, is to adjust the key schedule, and the second is to introduce the key material more slowly. This results in the algorithm given below which repairs the two minor weaknesses and retains almost the same simplicity.

Using two bits of sum to select the keywords saves using a separate count for n, while the shift of 11 causes the sequence to be irregular. However 11 is chosen so that all 4 keywords are used in the the first two cycles. The cycle uses roughly the same formula to ensure “mixing” at the same rate.

It is true that in order to gain these advantages, the time to complete change diffusion is delayed by one cycle, however it seems unnecessary to increase 32 to 33, as it is still believed that the number needed is about 16, and the safety factor against inadequate analysis is much the same.

*The declaration in teab has been changed from sum to sum=0 on page 3. Error found by Keith Lockstone.

1 Private Communication, possibly due to be published at Eurocrypt 1997, email David Wagner law@cs.berkeley.edu,.
Coding and Decoding Routine

v gives the plain text of 2 words, 
k gives the key of 4 words, 
N gives the number of cycles, 32 are recommended, 
if negative causes decoding, N must be the same as for coding, 
if zero causes no coding or decoding. 
asumes 32 bit “long” and same endian coding or decoding

tean( long * v, long * k, long N) {
unsigned long y=v[0], z=v[1], DELTA=0x9e3779b9 ;
if (N>0) {
    /* coding */
    unsigned long limit=DELTA*N, sum=0 ;
    while (sum!=limit)
        y+= (z<<4 ^ z>>5) + z ^ sum + k[sum&3],
            sum+=DELTA,
        z+= (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3] ;
}
else {
    /* decoding */
    unsigned long sum=DELTA*(-N) ;
    while (sum)
        z-= (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3],
            sum-=DELTA,
        y-= (z<<4 ^ z>>5) + z ^ sum + k[sum&3] ;
    v[0]=y, v[1]=z ;
    return ;
}

Block TEA

The algorithm can be modified to cater for larger block sizes. The intrinsic mixing is kept similar to TEA, but we run cyclically round the block. This means the mixing of various stages is done together, and we get a potential saving of time, and a natural binding together of a complete block of N words. The key scheduling is also slightly changed, but probably has the same characteristics.

Thus we do the mix operation along the words of a block and do the whole block operation a number of times. The operation is v[n]+= mix(v[n-1],key]
The operation rolls around the end. If we assume that the mixing is sufficient in TEA with 64 operations, and that we need at least six operations on each word, so the flow of data into the mth word is at least as much as the key, then we need to operate on each word 6+52/n times. This gives a minimum of 6 mixes on each word. A differential attack exists for 3 mixes, which
utilises the changes from the last word to the first word. Also the bandwidth of changes into
one word from the previous word is about 6 times 32, which may give sufficient margin against
solving a “cut”.

We now find the work needed for the longer blocks is about 6 mix operations per word and
that when \( n > 4 \) the block algorithm is faster than TEA in spite of the increased overheads.
For \( n=2 \) the routine is about twice as slow as TEA. In fact for \( n=2-10 \) the time is roughly
independent of block size in one implementation.

The question of whether it as secure remains open. The same device can be used on DES,
although as it has only 16 rounds and not 64, the gains are not as spectacular.

coding and decoding routine

teab is a block version of tean.
It will encode or decode \( n \) words as a single block where \( n > 1 \).
v is the \( n \) word data vector,
k is the 4 word key.
n is negative for decoding,
if \( n \) is zero result is 1 and no coding or decoding takes place,
otherwise the result is zero.
assumes 32 bit “long” and same endian coding and decoding.

```c
long teab( long * v, long n , long * k ) {
    unsigned long z=v[n-1], sum=0,e,
    DELTA=0x9e3779b9 ;
    long m, p, q ;
    if ( n>1 ) {
        /* Coding Part */
        q=6+52/n ;
        while (q-- > 0) {
            sum += DELTA ;
            e = sum>>2&3 ;
            for (p=0 ; p<n ; p++ )
                z=v[p] += (z<<4 ^ z>>5 ) + z ^ k[p&3^e] + sum ;
        }
        return 0 ; }
```
Comments

For ease of use and general security the large block version is to be preferred when applicable for the following reasons.

- A single bit change will change about one half of the bits of the entire block, leaving no place where the changes start.
- There is no choice of mode involved.
- Even if the correct usage of always changing the data sent (possibly by a message number) is employed, only identical messages give the same result and the information leakage is minimal.
- The message number should always be checked as this redundancy is the check against a random message being accepted.
- Cut and join attacks do not appear to be possible.
- If it is not acceptable to have very long messages, they can be broken into chunks say of 60 words and chained analogously to the methods used for DES.
TEA Source Code

Here is source code for the Tiny Encryption Algorithm in a variety of forms:

ANSI C

Motorola PowerPC
Motorola 680x0
New Variant (in ANSI C)
New Variant (in 16-bit x86 assembly language)

Please feel free to use any of this code in your applications. The TEA algorithm (including new-variant TEA) has been placed in the public domain, as have my assembly language implementations.

ANSI C

```c
void encipher(unsigned long *const v,unsigned long *const w,
              const unsigned long *const k)
{
    register unsigned long       y=v[0],z=v[1],sum=0,delta=0x9E3779B9,
                                  a=k[0],b=k[1],c=k[2],d=k[3],n=32;

    while(n-->0)
    {
        sum += delta;
        y += (z << 4)+a ^ z+sum ^ (z >> 5)+b;
        z += (y << 4)+c ^ y+sum ^ (y >> 5)+d;
    }
    w[0]=y; w[1]=z;
}

void decipher(unsigned long *const v,unsigned long *const w,
               const unsigned long *const k)
{
    register unsigned long       y=v[0],z=v[1],sum=0xC6EF3720,
                                  delta=0x9E3779B9,a=k[0],b=k[1],
                                  c=k[2],d=k[3],n=32;

    /* sum = delta<<5, in general sum = delta * n */

    while(n-->0)
    {
        z -= (y << 4)+c ^ y+sum ^ (y >> 5)+d;
        y -= (z << 4)+a ^ z+sum ^ (z >> 5)+b;
        sum -= delta;
    }
    w[0]=y; w[1]=z;
}
```

Motorola PowerPC (Metrowerks CodeWarrior Style)
asm void encipher(register const unsigned long * const v,
    register unsigned long * const w,
    register const unsigned long * const k)
{
    // On entry, v = r3, w = r4, k = r5
    // use r3 and r5 as scratch
    // r0 = v[0]
    // r6 = v[1]
    // r7 - r10 = k[0] - k[3]
    // r11 = sum
    // r12 = delta
    li r11,0 // sum = 0
    li r12,0x79B9 // delta =0x9E3779B9
    addis r12,r12,0x9E37
    li r0,16 // loop counter into count register
    mtctr r0
    lwz r0,0(r3) // put the contents of v and k into the registers
    lwz r6,4(r3)
    lwz r7,0(r5)
    lwz r8,4(r5)
    lwz r9,8(r5)
    lwz r10,12(r5)

    loop: add r11,r11,r12 // sum += delta
    slwi r5,r6,4 // z << 4
    add r5,r5,r7 // (z << 4) + a
    add r3,r6,r11 // z + sum
    xor r3,r5,r3 // ((z << 4) + a) ^ (z + sum)
    srwi r5,r6,5 // z >> 5
    add r5,r5,r8 // (z >> 5) + b
    xor r3,r5,r3 // ((z << 4)+a)^((z+sum)<<(z > 5)+b);
    add r0,r0,r3 // y += result
    slwi r5,r0,4 // y << 4
    add r5,r5,r9 // (y << 4) +c
    add r3,r0,r11 // y + sum
    xor r3,r5,r3 // ((y << 4) +c) ^ (y + sum)
    srwi r5,r0,5 // y >> 5
    add r5,r5,r10 // (y >> 5) + d
    xor r3,r5,r3 // ((y << 4)+c)^(y+sum)^((y >> 5)+d);
    add r6,r6,r3 // z += result
    bdnz+ loop // decrement CTR and branch
    stw r0,0(r4) // store result back in w
    stw r6,4(r4)
    blr
}

asm void decipher(register const unsigned long * const v,
    register unsigned long * const w,
    register const unsigned long * const k)
{
    // On entry, v = r3, w = r4, k = r5
    // use r3 and r5 as scratch
    // r0 = v[0]
    // r6 = v[1]
    // r7 - r10 = k[0] - k[3]
    // r11 = sum
    // r12 = delta
li    r11,0x9B90;       // sum =0xE3779B90
addis r11,r11,0xE378;
li    r12,0x79B9  // delta =0x9E3779B9
addis r12,r12,0x9E37
li    r0,16 // loop counter into count register
mtctr r0
lwz   r0,0(r3) // put the contents of v and k into the registers
lwz   r6,4(r3)
lwz   r7,0(r5)
lwz   r8,4(r5)
lwz   r9,8(r5)
lwz   r10,12(r5)

loop: slwi  r5,r0,4     // y << 4
        add r5,r5,r9    // (y << 4) + c
        add r3,r0,r11   // y + sum
        xor r3,r5,r3    // ((y << 4) + c) ^ (y + sum)
srwi  r5,r0,5     // y >> 5
        add r5,r5,r10   // (y >> 5) + d
        xor r3,r5,r3    // ((y << 4)+c)^ (y+sum)^(y >> 5)+d
    sub r6,r6,r3 // z -= result
    slwi  r5,r6,4 // z << 4
    add r5,r5,r7    // (z << 4) + a
    add r3,r6,r11   // z + sum
    xor r3,r5,r3    // ((z << 4) + a) ^ (z + sum)
srwi  r5,r6,5     // z >> 5
    add r5,r5,r8    // (z >> 5) + b
    xor r3,r5,r3    // ((z << 4)+a)^(z+sum)^(z >> 5)+b);
    sub r0,r0,r3 // y -= result
    sub r11,r11,r12 // sum -= delta
bdnz+ loop  // decrement CTR and branch
    stw   r0,0(r4) // store result back in w
    stw   r6,4(r4)
blr

Motorola 680x0 (Metrowerks CodeWarrior style):

asm void encipher(const unsigned long* const v,unsigned long* const w, const unsigned long* const k)
{
    fralloc
    movem.l  d3-d7/a2,-(a7)
    /* load initial registers
    d0:  y = v[0]
d1:  z = v[1]
d2:  a = k[0]
d3:  b = k[1]
d4:  c = k[2]
d5:  loop counter (k[3] in a2)
d6:  scratch register 1
    */
}
d7: scratch register 2
a0: sum
a1: delta = 0x9E3779B9;
a2: d = k[3] */

move.l v,a0
move.l (a0),d0
move.l 4(a0),d1
move.l k,a0
move.l (a0),d2
move.l 4(a0),d3
move.l 8(a0),d4
move.l 12(a0),a2
move.l #0x9E3779B9,a0
move.l #0x9E3779B9,a1
moveq.l #15,d5      // sixteen rounds

// d6 = (z<<4)+a
loop: move.l d1,d6
lsll.l #4,d6
add.l d2,d6

// d7 = z+sum
move.l d1,d7
add.l a0,d7

// d7 = ((z<<4)+a)^/(z+sum)
eor.l d6,d7

// d6 = (z>>5)+b
move.l d1,d6
lsrl.l #5,d6
add.l d3,d6

// d7 = ((z<<4)+a)^/(z+sum)^/(z>>5)+b)
eor.l d6,d7

// add back into y
add.l d7,d0

// d6 = (y<<4)+c
move.l d0,d6
lsll.l #4,d6
add.l d4,d6

// d7 = y+sum
move.l d0,d7
add.l a0,d7

// d7 = ((y<<4)+c)^/(y+sum)
eor.l d6,d7

// d6 = (y>>5)+d
move.l d0,d6
lsrl.l #5,d6
add.l a2,d6

// d7 = ((y<<4)+c)^/(y+sum)^/(y>>5)+d)
eor.l d6,d7

// add back into z
add.l d7,d1

// sum+=delta
adda.l a1,a0

// branch back and do it again
dbra    d5,loop

// place the result back into w
move.l w,a0
move.l d0,(a0)
move.l d1,4(a0)
movem.l (a7)+,d3-d7/a2

frfree
rts
}

asm void decipher(const unsigned long *const v,unsigned long *const w,
    const unsigned long *const k)
{
    fralloc
    movem.l d3-d7/a2,-(a7)

    /* load initial registers
     * d0:   y = v[0]
     * d1:   z = v[1]
     * d2:   a = k[0]
     * d3:   b = k[1]
     * d4:   c = k[2]
     * d5:   loop counter (k[3] in a2)
     * d6:   scratch register 1
     * d7:   scratch register 2
     * a0:   sum = 0xE3779B90 (delta * 16)
     * a1:   delta = 0x9E3779B9;
     * a2:   d = k[3] */

    move.l   v,a0
    move.l   (a0),d0
    move.l   4(a0),d1

    move.l   k,a0
    move.l   (a0),d2
    move.l   4(a0),d3
    move.l   8(a0),d4
    move.l   12(a0),a2

    move.l   #0xE3779B90,a0
    move.l   #0x9E3779B9,a1
    moveq.l  #15,d5      // sixteen rounds

    // d6 = (y<<4)+c
    loop: move.l   d0,d6
          lsl.l    #4,d6
          add.l    d4,d6

    // d7 = y+sum
    move.l   d0,d7
    add.l    a0,d7

    // d7 = ((y<<4)+c)^(y+sum)
    eor.l    d6,d7

    // d6 = (y>>5)+d
    move.l   d0,d6
    lsr.l    #5,d6
    add.l    a2,d6
void encipher(const unsigned long *const v,unsigned long *const w,
const unsigned long * const k)
{
    register unsigned long       y=v[0],z=v[1],sum=0,delta=0x9E3779B9,n=32;
    while(n-->=0)
    {
        y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        sum += delta;
        z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>(11 & 3)];
    }
    w[0]=y; w[1]=z;
}

ANSI C (New Variant)
void decipher(const unsigned long *const v,unsigned long *const w,  
const unsigned long * const k)  
{  
    register unsigned long       y=v[0],z=v[1],sum=0xC6EF3720,  
        delta=0x9E3779B9,n=32;  
/* sum = delta<<5, in general sum = delta * n */  
while(n-->0)  
{  
    z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];  
    sum -= delta;  
    y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];  
}  
    w[0]=y; w[1]=z;  
}  

16-bit x86 (New Variant)

Many thanks to Rafael R. Sevilla for contributing this version.

;;  
;; An implementation of the XTEA algorithm in 16-bit 80x86 assembly  
;; language. This should work on any processor in the 80x86 family  
;; but works best with the 16-bit members of the family (80x86 for  
;; x <= 2). This assembly language is suitable for use with linux-86  
;; and the as86 assembler, but should be fairly trivial to convert  
;; so it will assemble with some other assembler easily. It has been  
;; tested with 16-bit objects for Linux-8086 (ELKS), and should work  
;; under DOS with the tiny and small memory models. To make it  
;; work with the large and huge memory models, it will probably be  
;; necessary to reset the parameters (bp+4 becomes bp+6 and so on),  
;; and segment loads may need to be done as well (les and lds).  
;;  
;; Wasn't so easy to write because the number of registers available  
;; on the 80x86 is kinda small...and they're 16-bit registers too!  
;;  
;; Placed in the Public Domain by Rafael R. Sevilla  
;;

.text
export _xtea_encipher_asm
_xtea_encipher_asm:
push   bp
mov    bp,sp
sub    sp,#14      ; space for y, z, sum, and n
push   si
push   di
;; bp+8 = pointer to key information  
;; bp+6 = pointer to ciphertext to return to caller  
;; bp+4 = pointer to plaintext  
;; bp+2 = return address from caller  
;; bp = pushed bp  
;; bp-2 = y high word  
;; bp-4 = y low word
mov bx,[bp+4] ; get address of plaintext
mov ax,[bx]  ; low word of first dword of plaintext
mov ax,[bp-4],ax
mov ax,[bx+2] ; high word
mov [bp-2],ax
mov ax,[bx+4] ; second dword of plaintext (low)
mov ax,[bp-8],ax
mov ax,[bx+6] ; (high)
mov [bp-6],ax
xor ax,ax  ; zero the sum initially
mov [bp-10],ax
mov [bp-12],ax
mov byte ptr [bp-14],#32 ; set n (just 8 bits), # rounds

encipher_rounds:
    ;; compute new y
mov ax,[bp-8] ; low word z
mov bx,[bp-6] ; high word z
mov cx,ax  ; copy to the rest of the registers
mov dx,bx
mov si,ax
mov di,bx
;; (z<<4) ^ (z>>5)
shl ax,#1  ; shift left once
rcl bx,#1
shl ax,#1  ; shift twice
rcl bx,#1
shl ax,#1  ; shift three times
rcl bx,#1
shl ax,#1  ; shift four times
rcl bx,#1
shr dx,#1  ; shift right once
rcr cx,#1
shr dx,#1  ; shift right twice
rcr cx,#1
shr dx,#1  ; shift right three times
rcr cx,#1
shr dx,#1  ; shift right four times
rcr cx,#1
shr dx,#1  ; shift right five times
rcr cx,#1
xor ax,cx  ; combine
xor dx,bx  ; dx:ax has result
xor si,[bp-12] ; combine to sum
xor di,[bp-10]
add ax,si  ; add them together
adc dx,di
mov bx,[bp-12] ; get low word of sum (all we need for this)
and bx,#3  ; get low two bits (modulo 4)
shl bx,#1  ; convert to dword offset
shl bx,#1
add bx,[bp+8] ; add to base address of key info
add ax,[bx]  ; low word of key
adc dx,[bx+2] ; high word of key
add [bp-4],ax ; add back to y
adc [bp-2],dx

;; update sum
mov ax,#0x79b9 ; low word of delta
add [bp-12],ax
mov ax,#0x9e37 ; high word of delta
adc [bp-10],ax

;; compute new z
mov ax, [bp-4] ; low word of y
mov bx, [bp-2] ; high word of y
mov cx, ax ; copy to the rest of the registers
mov dx, bx
mov si, ax
mov di, bx

;; (y<<4) ^ (y>>5)
shl ax, #1 ; shift left once
rcl bx, #1
shl ax, #1 ; shift twice
rcl bx, #1
shl ax, #1 ; shift three times
rcl bx, #1
shl ax, #1 ; shift four times
rcl bx, #1
shr dx, #1 ; shift right once
rcr cx, #1
shr dx, #1 ; shift right twice
rcr cx, #1
shr dx, #1 ; shift right three times
rcr cx, #1
shr dx, #1 ; shift right four times
rcr cx, #1
shr dx, #1 ; shift right five times
rcr cx, #1
xor ax, cx ; combine
xor dx, bx ; dx:ax has result
xor si, [bp-12] ; combine to sum
xor di, [bp-10]
add ax, si ; add them together
adc dx, di
mov bx, [bp-12] ; get sum low word
mov cx, [bp-10] ; get sum high word
shr cx, #1 ; shift right once
rcr bx, #1
shr cx, #1 ; shift right twice
rcr bx, #1
shr cx, #1 ; shift right three times
rcr bx, #1
shr cx, #1 ; shift right four times
rcr bx, #1
shr cx, #1 ; shift right five times
rcr bx, #1
shr cx, #1 ; shift right six times
rcr bx, #1
shr cx, #1 ; shift right seven times
rcr bx, #1
shr cx, #1 ; shift right eight times
rcr bx, #1
shr cx, #1 ; shift right nine times
rcr bx, #1
shr cx, #1 ; shift right ten times
rcr bx, #1
shr cx, #1 ; shift right eleven times
rcr bx, #1
and bx, #3 ; convert to dword offset
shl bx, #1
add bx, [bp+8] ; add to base address of key
add ax, [bx] ; low word of key
adc dx, [bx+2] ; high word of key
add [bp-8], ax ; add back to z
adc [bp-6], dx
dec byte ptr [bp-14] ; decrement rounds counter
jz finish_encipher
jmp     near encipher_rounds

finish_encipher:
    mov bx,[bp+6] ; get address of ciphertext storage
    mov ax,[bp-4] ; y, low word
    mov [bx],ax
    mov ax,[bp-2] ; y, high word
    mov [bx+2],ax
    mov ax,[bp-8] ; z, low word
    mov [bx+4],ax
    mov ax,[bp-6] ; z, high word
    mov [bx+6],ax
    pop di
    pop si
    add sp,#14  ; discard local vars
    pop bp
    ret

export _xtea_decipher_asm

_xtea_decipher_asm:
    push bp
    mov bp,sp
    sub sp,#14  ; space for y, z, sum, and n
    push si
    push di
    ;; bp+8 = pointer to key information
    ;; bp+6 = pointer to plaintext to return to caller
    ;; bp+4 = pointer to ciphertext
    ;; bp+2 = return address from caller
    ;; bp = pushed bp
    ;; bp-2 = y high word
    ;; bp-4 = y low word
    ;; bp-6 = z high word
    ;; bp-8 = z low word
    ;; bp-10 = sum high word
    ;; bp-12 = sum low word
    ;; bp-14 = n
    ;; bp-16 = pushed si
    ;; bp-18 = pushed di
    mov bx,[bp+4] ; get address of ciphertext
    mov ax,[bx] ; low word of first dword of ciphertext
    mov [bp-4],ax
    mov ax,[bx+2] ; high word
    mov [bp-2],ax
    mov ax,[bx+4] ; second dword of ciphertext (low)
    mov [bp-8],ax
    mov ax,[bx+6] ; (high)
    mov [bp-6],ax
    mov ax,#0x3720 ; low word of initial sum
    mov [bp-12],ax
    mov ax,#0xc6ef
    mov [bp-10],ax
    mov byte ptr [bp-14],#32 ; set n (just 8 bits), # rounds
    ;; begin decryption
decipher_rounds:
    mov ax,[bp-4] ; low word of y
    mov bx,[bp-2] ; high word of y
    mov cx,ax ; copy to the rest of the registers
    mov dx,bx
    mov si,ax
    mov di,bx
    ;; (y<<4) ^ (y>>5)
    shl ax,#1 ; shift left once
    rcl bx,#1
    shl ax,#1 ; shift twice
    rcl bx,#1
    shl ax,#1 ; shift three times
    rcl bx,#1
shl ax,#1 ; shift four times
rcl bx,#1
shr dx,#1 ; shift right once
rcr cx,#1
shr dx,#1 ; shift right twice
rcr cx,#1
shr dx,#1 ; shift right three times
rcr cx,#1
shr dx,#1 ; shift right four times
rcr cx,#1
shr dx,#1 ; shift right five times
rcr cx,#1
xor ax,cx ; combine
xor dx,bx ; dx:ax has result
xor si,[bp-12] ; combine to sum
xor di,[bp-10]
add ax,si ; add them together
adc dx,di
mov bx,[bp-12] ; get sum low word
mov cx,[bp-10] ; get sum high word
shr cx,#1 ; shift right one
rcr bx,#1
shr cx,#1 ; shift right twice
rcr bx,#1
shr cx,#1 ; shift right three times
rcr bx,#1
shr cx,#1 ; shift right four times
rcr bx,#1
shr cx,#1 ; shift right five times
rcr bx,#1
shr cx,#1 ; shift right six times
rcr bx,#1
shr cx,#1 ; shift right seven times
rcr bx,#1
shr cx,#1 ; shift right eight times
rcr bx,#1
shr cx,#1 ; shift right nine times
rcr bx,#1
shr cx,#1 ; shift right ten times
rcr bx,#1
shr cx,#1 ; shift right eleven times
rcr bx,#1
and bx,#3 ; shift right six times
shl bx,#1 ; convert to dword offset
shl bx,#1
add bx,[bp+8] ; add to base address of key
add ax,[bx] ; low word of key
adc dx,[bx+2] ; high word of key
sub [bp-8],ax ; subtract from z
sbb [bp-10],ax
;; compute new y
mov ax,#0x79b9 ; low word of delta
sub [bp-12],ax
mov ax,#0x9e37 ; high word of delta
sbb [bp-10],ax
;; (z<<4) ^ (z>>5)
mov ax,[bp-8] ; low word z
mov bx,[bp-6] ; high word z
mov cx,ax ; copy to the rest of the registers
mov dx,bx
mov si,ax
mov di,bx
;; (z<<4) ^ (z>>5)
shl ax,#1 ; shift left once
rcl bx,#1
shl ax,#1 ; shift twice
rcl bx,#1
shl ax,#1  ; shift three times
rcl bx,#1
shl ax,#1  ; shift four times
rcl bx,#1
shr dx,#1  ; shift right once
rcr cx,#1
shr dx,#1  ; shift right twice
rcr cx,#1
shr dx,#1  ; shift right three times
rcr cx,#1
shr dx,#1  ; shift right four times
rcr cx,#1
shr dx,#1  ; shift right five times
rcr cx,#1
xor ax,cx  ; combine
xor dx,bx  ; dx:ax has result
xor si,[bp-12] ; combine to sum
xor di,[bp-10]
add ax,si  ; add them together
adc dx,di
mov bx,[bp-12] ; get low word of sum (all we need for this)
and bx,#3  ; get low two bits (modulo 4)
shl bx,#1  ; convert to dword offset
shl bx,#1
add bx,[bp+8] ; add to base address of key info
add ax,[bx]  ; low word of key
adc dx,[bx+2] ; high word of key
sub [bp-4],ax ; subtract from y
sbb [bp-2],dx
dec byte ptr [bp-14] ; decrement rounds counter
jz finish_decipher
jmp near decipher_rounds

finish_decipher:
  mov bx,[bp+6] ; get address of ciphertext storage
  mov ax,[bp-4] ; y, low word
  mov [bx],ax
  mov ax,[bp-2] ; y, high word
  mov [bx+2],ax
  mov ax,[bp-8] ; z, low word
  mov [bx+4],ax
  mov ax,[bp-6] ; z, high word
  mov [bx+6],ax
  pop di
  pop si
  add sp,#14  ; discard local vars
  pop bp
  ret