Tales from the Encrypt

AiS Challenge
Final Report
April 3, 2002

Team #050
Manzano High School

Team Members:
Robert Cordwell
Brian Rosen

Teacher:
Stephen Schum

Table of Contents

Team 050 -- Manzano H.S. -- "Tales from the Encrypt"

Executive Summary

In encryption, the lock is the encryption and the key is the method of decryption. Recently, concerns have arisen that the old encryption standard, the Digital Encryption Standard (DES), is insufficiently secure because increased computer speeds make a brute force attack feasible in which every possible key is checked. Also of concern is the fact that "weak" and "semi-weak" keys allow for a shortcut attack that takes even less processor time to break. Finally, some in the encryption community were concerned about the possibility of a "backdoor" in DES which would enable the U.S. Government to read whatever message was chosen. The Commerce Department's National Institute of Standards and Technology hosted an international competition, inviting programmers to create a new encryption algorithm. The winning algorithm, chosen for its security and speed was called Rijndael after its creators, John Daemen and Vincent Rijmen and will become effective as the Advanced Encryption Standard (AES) on May 26, 2002.

AES is a block-encryption algorithm. In other words, it encrypts blocks of varying bit lengths from the smallest of 128 bits to the largest of 256 bits. We have worked with the 128-bit key version, although our work can extend to the larger bit sizes.

In our approach to encryption, we used the same encryption algorithm as AES, but implemented it in a different way by using a technique called bitslicing. Bitslicing is a programming technique which can be used to encrypt multiple values at once. Instead of encrypting the first entire 128-bit block, then the second, bitslicing breaks off each of the individual bits, and groups them into "extended bits". Thus, one extended bit contains all of the first bits for a set of values that need encrypting. The main disadvantage of bitslicing is that very efficient "lookup tables", are not possible. A "lookup table" is a method of retrieving previously calculated solutions to a complex mathematical equation. Thus, in bitslicing, all mathematical operations must be calculated directly using basic register operations such as XOR, AND, OR, etc…. However, bitslicing does have a tremendous potential speed advantage because it encrypts many values simultaneously. Therefore, the speed of bitslicing may be similar to or even surpass the speed achieved by using lookup tables.

The purpose of our project was to test the effects of bitslicing on the Advanced Encryption Standard. Our first test program implementing AES used mathematical operations rather than lookup table to accomplish the basic, non-bitsliced encryption. The program was slow, but it worked, leading to a second test program which used bitslicing and similar mathematical methods to encrypt many values simultaneously. A third program, also using bitslicing but taking advantage of faster mathematical methods and processor properties is under development.

Our hypothesis was that the bitsliced program would encrypt values about 4 times slower than the lookup-table implementation, based on the much larger number of operations, but benefiting from the simultaneity of bitslicing. The computer that we used had 64-bit MMX registers as the largest available. To make use of these special registers necessitated using assembly level instructions. As it turned out, our hypothesis, based on counting the number of steps in the assembly level instructions, was quite accurate (within about 10 percent).

Most of our bitslicing program was built using assembly level code, which allows for only basic operations. This is not a major disadvantage, because any complicated mathematical Boolean operation can be implemented with simple operations. Moreover, some operations, such as permutations, are much more efficient under bitslicing. The main advantage of using the larger 64-bit MMX registers (and hence the assembly code) is the twofold speed improvement over the 32-bit regular registers. Finally, because assembly level code is taken by the compiler at face value, it is easier to optimize by directly changing some of the code around.

Our results showed that for 64-bit registers the speed at which the unimproved bitsliced program ran was slower, but still comparable to the lookup-table program. We think that we can almost double the speed of the bitsliced program by taking advantage of processor pipelining of instructions. Moreover, as register size doubles (128-bit registers are commercially available), the amount of time taken for the bitslicing is the same, but the number of encryptions also doubles, while the time taken for a lookup table program stays the same. In a computer with 256-bit registers, the bitsliced implementation could then potentially be twice as fast as the lookup table implementation. This has important implications for future implementations of the AES algorithm whenever large amounts of data need to be encrypted quickly, such as the streaming video from an Unmanned Aerial Vehicle.

The size of an extended bit can be as long as the largest register size. For an Intel Pentium II, the normal register size is 32 bits, but there are also 8 64-bit registers available, which we made use of. Taking an XOR operation of the first extended bit with the second is equivalent to taking an XOR operation of the first bit with the second in all 64 blocks simultaneously.

Introduction

Our project is an attempt to increase the speed of the soon-to-be Advanced Encryption Standard by using the bitslicing programming technique. Bitslicing is a programming technique which allows for encryption of multiple values simultaneously. Our main program uses the bitslicing technique to encrypt a set of values. The results are the same as those generated by AES, but the techinque used is radically different.

We are also including possible methods to make the encryption more secure, such as cipher feedback mode. It is important to notice that when cipher feedback mode is used, decryption of the cipher does not require the decryption of the AES algorithm. Thus, a decryption algorithm is entirely unnecessary when using cipher feedback mode (and it saves space not to have a decryption algorithm). We do not currently have a decryption algorithm but may make one by the time of the final presentation.

Possible applications range from a faster encryption of very large files to faster on-the-site encryption of streaming video from battlefield robots and unmanned vehicles. We may also present our project and corresponding results at a cryptology conference. Possible further developments include "pipelining" the code to make it more efficient, reducing the amount of code needed by creating the S-box inverse directly, and also developing a program which can encrypt and decrypt files and pictures.

Abstract

Using Bitslicing in the Advanced Encryption Standard

The purpose of this project is to test the effects of bitslicing, a technique for parallel encryption, on the Advanced Encryption Standard (AES). The initial idea was to write a program that would perform the same operations as the AES, but would use mathematical operations instead of lookup tables (the current standard) to encrypt values. This led to a second program, which utilized the bitslicing technique to improve the speed.

Both programs were built using Microsoft Visual C++. All speed tests used the built-in timer function to eliminate human error and were performed under identical conditions. Sub-functions of the AES and of the programs (except related to the encryption key-expansion) were tested separately, millions of times each. The final speed values were determined by combining the times taken to perform each of the individual functions.

The first program was approximately a thousand times slower than the lookup table program. The second, bitsliced program took approximately 11.4 seconds for 10 million "rounds" (excluding key-expansion operations) on an Intel 450 MHz Pentium II. The standard lookup table program took about 2.5 seconds on the same processor.

This project has potentially important implications for large-scale encryption, such as for streaming video or large program files. By using larger register sizes (special dedicated hardware), the speed could be greatly increased. Every time the register size doubles, the number of encryptions per run does also. The speed of future versions of the program can also be increased by making the code more efficient.

Background for Encryption

Any procedure used in cryptography to convert plain-text into cipher-text in order to prevent any but the intended recipient from reading that data.

For the longest of times, the problem of the wrong people intercepting messages not intended for them has plagued mankind. Encryption, though it has changed over the years, was and is usually the solution. Over time, people actually began to ‘cipher’ messages, rather then simply using words unknown to the enemy or hiding the message. For a cipher, the words and letters would be altered in some way, possibly by the transposition of letters or the substitution of other letters. Thus, the art of codebreaking also arose. All algorithms, except for the theoretically unbreakable "one-time pads" will at some point be broken, even if it takes a hundred years on current technology. One-time pads Still, though, encryption is used for many purposes, be it security, messages from government to army, or even just two friends talking.

It is mindless to put years of time into developing thousands of ‘unbreakable’ algorithm one-time pads. Instead, it is wiser to develop an encryption program which would take a year to break a message (only to find out that it was a confirmation of an online order). If there are an extremely large number of possible keys, a code breaker could be slowed down. It is also important to change the key occasionally. If an enemy army intercepts the message "%8Un" and the good army goes west, the next time the enemy army intercepts a message saying "%8Un" they will know that the good army will go west. This method of codebreaking is known as traffic analysis.

Another problem some current algorithms have is that they are too slow for certain applications. If it takes a long time to encrypt streaming video from an Unmanned Aerial Vehicle, the operator may not be able to see the enemy soldiers with their AA missile launchers. The solution to this comes in many different forms, such as bitslicing or dedicated hardware.

Modern algorithms have many loops, some of which seem extraneous and a waste of time; however, they are needed to prevent certain shortcut methods of codebreaking. The fact that computers almost never get "confused" or drop a byte allows many modern algorithms to be quite complicated.

History of the AES Competition

A year ago, researchers from 12 different countries submitted 15 candidates for the Advanced Encryption Standard (AES) —the new encoding method that eventually will be adopted by the federal government. These candidates have been subjected to many different simulated "attacks" by cryptographers, narrowing the choice to only 5 possible candidate algorithms.

The AES will be a public algorithm designed to protect sensitive government information well into the next century and will replace the aging Data Encryption Standard, currently used in both the private and the government sectors.

The following five candidates were selected: MARS, RC6™, Rijndael, Serpent, and Twofish. No significant security vulnerabilities were found for the five finalists. At a 128 bit key size, there are approximately 340,000,000,000,000,000,000,000,000,000,000,000,000,000 (340 followed by 36 zeroes) possible keys. An exhaustive search of all of the keys would take a tremendous amount of time, even for a massive supercomputer or many computers working in tandem.

The final decision was determined by several factors, including speed, security, and the simplicity of the algorithm, among others. The final winner was Rijndael, which had a simple code, was easy to implement (even on a Smart Card), and is, in theory, perfectly secure. Rijndael will become the new Advanced Encryption Standard as of May 26, 2002.

Reference: Various excerpts were taken from an online publication at http://www.nist.gov .

Mathematical Background of AES

AES uses several different mathematical techniques including base 2 polynomial multiplication, and Galois Fields for the s-box and mixcolumns transformations. To fully understand what happens when these transformations are performed, some mathematical background is in order.

In base 2, there are only two numbers, 0 and 1. Base 2 polynomials are like normal polynomials, except they use only 0 and 1 for coefficients. For example, $x^4 + x^3 + x + 1$ would be a base 2 polynomial. If this polynomial were added to the polynomial $x^6 + x^3 + x$ using normal polynomial addition, the result would be $x^6 + x^4 + 2x^3 + 2x + 1$. However, with base 2 polynomials, the answer would be $x^6 + x^4 + 1$, because all 2s are "cancelled out". Notice that addition is the same thing as subtraction. Polynomial multiplication works in much the same way.

Irreducible polynomials are those for which two other polynomials cannot be chosen which multiply out to that particular polynomial. For example, $x^2 + 1$ is not irreducible, because in base 2 it is equal to
$(x + 1)(x + 1)$. Irreducible polynomials are used to generate Galois fields, which are fields of all base prime number (always 2 in this case) polynomials modulus an irreducible polynomial. For example, there is a Galois field created with the polynomial $x^2 + x + 1$. Suppose that I multiply $(x +1)(x +1)$ in this field. The answer would be $(x^2 + 2x + 1) + (x^2 + x + 1) = 2x^2 + 3x + 2 = x$.

A property of any field is that every element has an inverse. This also holds for Galois fields (when the zero element is removed). In the field created by the polynomial $x^2 + x + 1$, the inverse of $(x+1)$ is x and visa versa, because $(x + 1)*x = x^2 + x = 1$. Legrange's theorem says that any value in a finite group to the power of the number of elements equals one. This means that $(x + 1)^3 = 1$. Also notice that because this is true for any element, an element to the power of the number of elements of the group – 1 is equal to the inverse. Thus, $(x + 1)^2 = x$, and this was previously demonstrated.
This property is used in the S-box transformation. The S-box inverse transformation takes the inverse of a polynomial in the Galois field determined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. There are 256 elements in this field, 255 in the resulting multiplicative group (taking away 0), and thus the inverse can be determined by taking the polynomial to the 254^{th} power. Obviously, it is inefficient to take the polynomial to such a high power using 253 multiplication steps. Instead, we notice that squaring a polynomial over base 2 is relatively easy, while multiplying two different polynomials is fairly difficult.

We can get the 254 power by performing the following steps. Notice that there are only four multiplications.

Start with a polynomial to be inverted. Call it x.
Square it and store that result in "a". Now we have x^2
Square it again and multiply that result by "a". Now we have x^6 stored in "a".
Square this twice and store it in "b".
Multiply "b" by "a". Now we have x^{30} stored in "a" and x^{24} stored in "b"
Square "b" twice and multiply it by "a". This gives us x^{126}.
Multiply this by the original x. This produces x^{127}.
Square this value. This gives x^{254} and the inverse.

The affine transformation is much simpler than the inverse. It treats the 8 bits as though they were a vector and multiplies them by a matrix. Finally, four of the eight bits are inverted (one goes to zero, zero goes to one).

The shiftrows transformation simply moves bytes around. No mathematical explanation is needed.

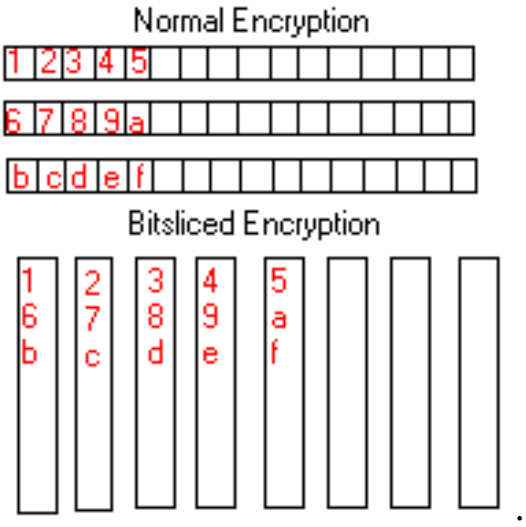
The mixcolumns operation uses many of the same basic principles as the S-box. However, mixcolumns looks at four bytes simultaneously in a four by one vector. The operation can be described as follows:

$$\begin{pmatrix} x & 1 & 1 & x+1 \\ x+1 & x & 1 & 1 \\ 1 & x+1 & x & 1 \\ 1 & 1 & x+1 & x \end{pmatrix} \begin{pmatrix} a_7x^7 + a_6x^6 + ...a_0x^0 \\ b_7x^7 + b_6x^6 + ...b_0x^0 \\ c_7x^7 + c_6x^6 + ...c_0x^0 \\ d_7x^7 + d_6x^6 + ...d_0x^0 \end{pmatrix} = \begin{pmatrix} Byte\ 1 \\ Byte\ 2 \\ Byte\ 3 \\ Byte\ 4 \end{pmatrix}$$

Each coefficient is actually an element of $GF(2^8)$. The resulting polynomial in x is reduced mod $(x^4 + 1)$. If this polynomial were irreducible, we would get a field of $GF(256^4)$. The polynomial is not irreducible, but each row element has an inverse mod $x^4 + 1$. Thus, the matrix has an inverse.

Bitslicing

Bitslicing is a method of improving the speed of some encryption algorithms by encrypting many different values simultaneously. In normal encryption of 64 values, I start with a 16 byte value, run the encryption program on it, then move on to the other 63 values. In bitslicing, I look at the first bits of all 64 values simultaneously and perform operations simultaneously. Instead of looking up values for the s-box transformation, I must calculate them, using logical gates such as XOR and AND.



The advantage of bitslicing occurs when the processor one is using is able to process long strings simultaneously. For example, many processors nowadays can process 128-bit strings. As processors become faster and faster and the demands increase, the string processing length will also increase. This will continue to make bitslicing more efficient.

Scientific Method:

Research Plan

Objective: Develop a simple encryption algorithm which does not relate to the AES. This will be used to gain a greater understanding of computer programming as well as possibly being used to perform tests on the AES algorithm later on.

I: Develop an algorithm which performs the same tasks as the actual Rijndael algorithm, but does not utilize lookup tables. This will be considered the "control" experiment. The control will mainly be used to gain a greater understanding of the programming and mathematical techniques involved.

II: Develop an algorithm which uses bitslicing techniques and assembly code. Optimize the bitslice program by using various programming techniques and by using mathematics to determine the fastest way to perform certain calculations. This optimization may include using a different processor with larger registers or by using programming techniques specific to a certain processor.

III: Compare the speed of the "control" program with the bitsliced algorithm. Test the different functions separately, however many times it takes to get a meaningful result (may be in the millions). Use these results to try to optimize the bitsliced algorithm further.

IV: Compare the speed of the bitsliced AES algorithm to the lookup table algorithm. One test will involve testing the different sub-functions of the AES algorithm and comparing their speed against the speed of the bitsliced algorithm. Final speed will be determined by a combination of the times for the different functions. Finally, determine the necessary conditions for the bitsliced algorithm to become more efficient.

V. Compare the speed, effectiveness and complexity of the bitslicing encryption method to Brian's alternative encryption program. (See Project Code - Method 2.)

VI: Write up results and conclusions. Some future research may include further optimization, the development of dedicated hardware, and/or the use/inclusion of other encryption techniques to achieve faster speeds.

Hypothesis

A bitsliced Advanced Encryption Standard or AES program will be less efficient than one which uses lookup tables for 64-bit registers. However, if 256-bit registers (currently not used in most Intel-based desktops) were used, then the bitsliced program would become more efficient than the lookup table program (the current official AES program).

Procedure

A computer program combining C++ and assembly language-based code was written to implement/simulate the mathematical equations (algorithm) explained in the Mathematical Background for AES and Appendices A,B, and C. The computer program flow chart for the AES Encryption Algorithm is outlined in the next section.

Flowchart for AES Encryption Algorithm

I: **Preliminaries**

The input is a 128-bit "text" to be encrypted and a 128-bit key. These are transformed to produce an encrypted 128-bit output.

A: **Starting Operation**

An initial bitwise XOR operation of the text with the Roundkey is performed before any other operations.

II: **Rounds**

There are 10 Rounds in the encryption algorithm. Most consist of four operations except for the last.

A: **S-Box**

The S-Box is the first operation in the Round. It takes single bytes at a time for all 16 bytes and consists of two parts.

1: Inverse

The inverse treats the byte as the coefficients of a polynomial in $GF(2^8)$ and takes its inverse modulus the polynomial $x^8 + x^4 + x^3 + x + 1$. See the Mathematical Background section.

2: Affine

The Affine treats the byte as a vector, multiplying it by a matrix of ones and zeroes (addition is performed by a bitwise XOR operation.) and then adding it to another vector.

B: **ShiftRows**

This transformation is performed once per round. It treats the bytes as though they were in a 4 x 4 matrix and does a number of left circular shifts one some of the rows. This is to allow for "diffusion" when the next operation is performed.

C: **MixColumns**

This transformation looks at each "column" in the 4 x 4 matrix of bytes individually. It treats each byte as a polynomial in $GF(2^8)$ and multiplies the vector determined by the 4-byte column by a 4 x 4 matrix consisting of polynomials in $GF(2^8)$.

D: **AddRoundKey**

This transformation finishes every round and performs a bitwise XOR of the RoundKey with the current encrypted state.

III: **Keys**

The key is once at the beginning, but a special Roundkey is generated for each round.

A: **Roundkey Generation**

Each Roundkey is generated from the previous Roundkey (or the original key). The first four bytes are found by using a tranformation on the last 4 bytes of the original Roundkey which uses the S-box on each one and then rotates them, finally adding a constant. The next 4 bytes are found by a bitwise XOR of the previous four bytes and the four bytes in the same corresponding position in the previous Roundkey. Same for the next 8 bytes.

Results -- Data

Human error was removed by causing the computer to compute the start time for the calculation and the end time for the calculation. All programs were run without any side programs on, such as MS Word. Thus, the results are accurate to one second, the resolution of the timing function used. Millions of repetitions also helped to dispel any random errors.

Ten trials were performed for four different implementation aspects: (times in seconds)

640,000,000 runs of S-box for bitsliced:

Time: 43 43 43 44 43 43 43 43 43 43 Average = 43.1 sec

640,000,000 runs of Mixcolumns for bitsliced:

Time: 10 10 10 10 10 10 10 10 10 10 Average = 10.0 sec

2,000,000,000 runs of table lookup operation:

Time: 20 20 20 20 21 20 20 20 21 20 Average = 20.2 sec

2,000,000,000 runs of XOR operation:

Time: 6 5 6 6 5 5 6 5 5 6 Average = 5.5 sec

Overall time for 10,000,000 bitsliced rounds (16 S-boxes and 4 Mixcolumns): 11.4 sec

Overall time for 10,000,000 lookup rounds (16 lookups and 16 XORs): 2.50 sec

A sample output from the program is shown in Table I on. the next page.

Table I. Sample Output from the "Bitslice10.cpp" Program.

```
************************************************************
0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0,
```

```
0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0,
1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1,
0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0,

Press any key to continue
************************************************************
```

The above output is the result of an input of "00112233445566778899aabbccddeeff" (in hexadecimal notation) for the plain text to be encrypted and "000102030405060708090a0b0c0d0e0f" for the key (and all of the expanded keys).

The output should be, according to known values in the FIPS publication, equal to: 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a. To check this, look at each 4 bit output group. The first should be equal to 6, or 0*8+1*4+1*2+0*1 = 6. The second is equal to 9, because 1*8+ 0*4+ 0*2+ 1*1 = 9. All of the rest of the results can be checked.

Analysis

This project involved applying a fairly well known technique in a unique and creative way. Our most original result is that we managed to demonstrate that the bitslicing encryption technique could be used to increase the speed of the AES algorithm for high volumes in the very near future. At first, we did not expect that the bitsliced program would be so fast. Certainly, our data shows that cryptographers should consider the bitslicing technique for implementation in the Advanced Encryption Standard.

It is critical to first note that when a larger register is used (128-bit or hypothetical 256-bit), is performance speed for moves and logical gate operations is identical to the speed of a smaller register. This property of the Intel chip architecture allows me to predict the performance of my bitslicing program on larger register types. The second critical thing to note is that computers in general are extremely consistent in their speed of performance for a given task. That said, we can use a combination of theoretical and actual timed calculations to give an idea for how long the bitsliced version would take compared to the table-lookup and the control versions.

Conclusions

By adding some optimization measures to the bitsliced program, the bitslicing program could possibly gain a speed advantage over the lookup table implementation. Such optimization measures could include a more efficient routine for taking a polynomial to the fourth power or even a subroutine to compute the inverse directly without using polynomial multiplication. Another one would be "pipelining," or switching around commands to move and XOR registers so that the two sub-processors could work simultaneously. Because the Intel chip architecture, as explained in Analysis, is designed so that operations with larger registers take exactly the same time as operations with smaller registers, time calculations can be made for currently hypothetical large registers. As register sizes increase and chips begin to use 256 and 512-bit registers, bitslicing could become the accepted method for AES encryption.

Acknowledgments

We would like to thank everyone who has generously contributed their time and effort to help us in our endeavor: my parents for driving, my sisters for being amenable to scheduling their home computer work around our computer time needs, and all of our teachers. We could never have developed our project had it not been for their aid and support.

A special thanks goes to our mentor, Dr. William R. Cordwell, who gave us the idea of using bitslicing as a technique in encryption. His patient guiding and humor helped us get through the times when we thought that we would never get the bugs out of our code. He was supportive, never dominating.

We are also extremely grateful to our teacher, Stephen Schum, who gave us computer time on the Manzano HS computers and also taught us programming techniques in the Adventures in Supercomputing course. He was willing to give up his own personal time over vacation and in the evenings to support us. His encouragement and wise guidance kept us on track.

References

Biham, Eli, "A Fast New DES Implementation in Software" Israel, 1997

Daemen, John and Rijmen, Vincent "AES Proposal: Rijndael," 1999

Federal Information Processing Standards, "Announcing the Advanced Encryption Standard", November 2001

"Intel Architecture Developers Manuel Volume 1," 2001, pp. 231-240

"Intel Architecture Developers Manuel Volume 2," 2002, pp. 472, 516-580

http://www.nist.gov/public_affairs/releases/g99-111.htm, "NIST Announces Encryption Standard Finalists, " August 9, 1999

Appendix A. Project Code

Team 050, Manzano High School, "Tales from the Encrypt"

```
//Program name:  Bitslice10.cpp     --   (Bitsliced Rijndael modification)
//Author:   Robert Cordwell
//Date:     April 1, 2002

#include <iostream.h>
#include <stdio.h>

unsigned long int inverse[2+1];
unsigned long int t[30+1];
unsigned long int c[16+1];
unsigned long int m[16+1];
unsigned long int tblock[256+1];
unsigned long int block[256+1];
unsigned long int GFinverse[16+1];

unsigned long int roundkey[2816+1];

unsigned long int* inverse_ptr;
unsigned long int* t_ptr;
unsigned long int* c_ptr;
unsigned long int* m_ptr;
unsigned long int* tblock_ptr;
unsigned long int* block_ptr;
unsigned long int* roundkey_ptr;

unsigned long int* block_byte_ptr;
unsigned long int* tblock_byte_ptr;

int p = 0;

void sbox(unsigned long int*, unsigned long int*);
void square(void);
void mult(void);
void affine(void);
void mixcolumns(unsigned long int*, unsigned long int*);
void build_round_key(unsigned char*, unsigned long int*, int);
void addroundkey(unsigned long int*, unsigned long int*, int);

const unsigned long int one = 0XFFFFFFFF;
const unsigned long int zero = 0X00000000;

int   i;
int j;
char pause;

int row;
int col;
int l;
//The roundkeys in this case are already pregenerated, as though they had just been
created
//from the function int keyexpansion(unsigned char* key, unsigned long int* expkey).
// This function is shown below, and is used in the non-bitsliced program.

//START FUNCTION
//Expands the key for the roundkeys.  This generates seveal different keys, each of
//which is used individually.
```

```
/*
  int keyexpansion(unsigned char* key, unsigned long int* expkey)
      {
      unsigned long int temp;
      int i;
      double p;

      for(i=0;i<4;i++)
            expkey[i] = (key[4*i]^key[4*i+1]<<8^key[4*i+2]<<16^key[4*i+3]<<24);

      for(i=4;i<44;i++)
      {
            temp = expkey[i-1];
            p = i;
            if(fmod(p,4.0)==0)
                  temp = subword(temp)^rcon(i/4);
            expkey[i] = expkey[i - 4] ^ temp;
      }

      return 0;
}
*/
//END FUNCTION
unsigned char oldroundkey[176] =
{
0X00, 0X01, 0X02, 0X03, 0X04, 0X05, 0X06, 0X07,
      0X08, 0X09, 0X0A, 0X0B, 0X0C,       0X0D, 0X0E, 0X0F,
0XD6, 0XAA, 0X74, 0XFD, 0XD2, 0XAF, 0X72, 0XFA,
      0XDA, 0XA6, 0X78, 0XF1, 0XD6,       0XAB, 0X76, 0XFE,
0XB6, 0X92, 0XCF, 0X0B, 0X64, 0X3D, 0XBD, 0XF1,
      0XBE, 0X9B, 0XC5, 0X00, 0X68,       0X30, 0XB3, 0XFE,
0XB6, 0XFF, 0X74, 0X4E, 0XD2, 0XC2, 0XC9, 0XBF,
      0X6C, 0X59, 0X0C, 0XBF, 0X04,       0X69, 0XBF, 0X41,
0X47, 0XF7, 0XF7, 0XBC, 0X95, 0X35, 0X3E, 0X03,
      0XF9, 0X6C, 0X32, 0XBC, 0XFD,       0X05, 0X8D, 0XFD,
0X3C, 0XAA, 0XA3, 0XE8, 0XA9, 0X9F, 0X9D, 0XEB,
      0X50, 0XF3, 0XAF, 0X57, 0XAD,       0XF6, 0X22, 0XAA,
0X5E, 0X39, 0X0F, 0X7D, 0XF7, 0XA6, 0X92, 0X96,
      0XA7, 0X55, 0X3D, 0XC1, 0X0A,       0XA3, 0X1F, 0X6B,
0X14, 0XF9, 0X70, 0X1A, 0XE3, 0X5F, 0XE2, 0X8C,
      0X44, 0X0A, 0XDF, 0X4D, 0X4E,       0XA9, 0XC0, 0X26,
0X47, 0X43, 0X87, 0X35, 0XA4, 0X1C, 0X65, 0XB9,
      0XE0, 0X16, 0XBA, 0XF4, 0XAE,       0XBF, 0X7A, 0XD2,
0X54, 0X99, 0X32, 0XD1, 0XF0, 0X85, 0X57, 0X68,
      0X10, 0X93, 0XED, 0X9C, 0XBE,       0X2C, 0X97, 0X4E,
0X13, 0X11, 0X1D, 0X7F, 0XE3, 0X94, 0X4A, 0X17,
0XF3, 0X07, 0XA7, 0X8B, 0X4D,       0X2B, 0X30, 0XC5
};  //This contians the original 16-byte roundkey in addition to all 10 of the
//expanded roundkeys.

//=============================================================================
======
void main(void)
{
ios::sync_with_stdio();

cout.setf(ios::hex,ios::basefield);
cout.setf(ios::uppercase);
cout.setf(ios::right);
cout.setf(ios::internal);
```

```
cout.setf(ios::right,ios::adjustfield);
cout.fill('0');

//Pseudocode for S-box
//Load values into mmx.  Values are stored in block
//Square and modulus.  Store as M
//Square and modulus again.  Store as C
//Mulitiply M and C
//Square and modulus again.  Store as C
//Multiply (A * B) * C.  Store as A
//Square and modulus three times. Store as B. Program has yet to be generated to do
this efficiently
//Mulitiply (A * B) * S  Store as A
//Square and modulus it.  This is the inverse.
//Run affine transformation.
//Notice that efficiency could be increased by changing the multiplication function.

//square and modulus of mms to mms

//__asm{
//          emms
//          pusha
//}

/*When I am calling up 8 byte "quadwords" the computer expects a memory address which
is divisible
by 8.  However, my arrays are made up of 4 byte unsigned integers.  I could have used
8 byte
doubles, but the computer expects these to be in a specific format.  Thus, if the
memory
location of the array is divisible by 4 but not by 8, I create a temporary new array
location
which is divisible by 8.*/

roundkey_ptr = roundkey;
if ((((unsigned long int)roundkey_ptr)%8)!=0)
      roundkey_ptr++;

inverse_ptr = inverse;
if ((((unsigned long int)inverse_ptr)%8)!=0)
      inverse_ptr++;

t_ptr = t;
if ((((unsigned long int)t_ptr)%8)!=0)
      t_ptr++;

c_ptr = c;
if ((((unsigned long int)c_ptr)%8)!=0)
c_ptr++;

m_ptr = m;
if ((((unsigned long int)m_ptr)%8)!=0)
m_ptr++;

tblock_ptr = tblock;
if ((((unsigned long int)tblock_ptr)%8)!=0)
tblock_ptr++;

block_ptr = block;
if ((((unsigned long int)block_ptr)%8)!=0)
```

```
block_ptr++;

// inverse of 10101101 should be 11100111, + affine should be 173 = 10010101
// inverse of 255 should be 00011100, + affine should be 22 = 00010110

//These should be 0XFFFFFFFF, but they are only 1 so that it is easier to read the
//output.
//We only test a single value for the bitsliced method because the same operations
//are performed on all of the values, so if it works for a single value, it works for
//every value.
inverse_ptr[0] = 1;
inverse_ptr[1] = 1;
inverse_ptr[2] = 1;

//Temporary block settings.  These are used to test the program - they correspond to
//a known value test shown in the FIPS publication.
//Notice that only single ones and zeroes are used.
//This is for ease of reading the value off.
block_ptr[0] = 0;
block_ptr[1] = 0;
block_ptr[2] = 0;
block_ptr[3] = 0;
block_ptr[4] = 0;
block_ptr[5] = 0;
block_ptr[6] = 0;
block_ptr[7] = 0;
block_ptr[8] = 0;
block_ptr[9] = 0;
block_ptr[10] = 0;
block_ptr[11] = 0;
block_ptr[12] = 0;
block_ptr[13] = 0;
block_ptr[14] = 0;
block_ptr[15] = 0;

block_ptr[16] = 1;
block_ptr[17] = 1;
block_ptr[18] = 0;
block_ptr[19] = 0;
block_ptr[20] = 0;
block_ptr[21] = 0;
block_ptr[22] = 0;
block_ptr[23] = 0;
block_ptr[24] = 1;
block_ptr[25] = 1;
block_ptr[26] = 0;
block_ptr[27] = 0;
block_ptr[28] = 0;
block_ptr[29] = 0;
block_ptr[30] = 0;
block_ptr[31] = 0;

block_ptr[32] = 0;
block_ptr[33] = 0;
block_ptr[34] = 1;
block_ptr[35] = 1;
block_ptr[36] = 0;
block_ptr[37] = 0;
block_ptr[38] = 0;
block_ptr[39] = 0;
```

```
block_ptr[40] = 0;
block_ptr[41] = 0;
block_ptr[42] = 1;
block_ptr[43] = 1;
block_ptr[44] = 0;
block_ptr[45] = 0;
block_ptr[46] = 0;
block_ptr[47] = 0;

block_ptr[48] = 1;
block_ptr[49] = 1;
block_ptr[50] = 1;
block_ptr[51] = 1;
block_ptr[52] = 0;
block_ptr[53] = 0;
block_ptr[54] = 0;
block_ptr[55] = 0;
block_ptr[56] = 1;
block_ptr[57] = 1;
block_ptr[58] = 1;
block_ptr[59] = 1;
block_ptr[60] = 0;
block_ptr[61] = 0;
block_ptr[62] = 0;
block_ptr[63] = 0;

block_ptr[64] = 0;
block_ptr[65] = 0;
block_ptr[66] = 0;
block_ptr[67] = 0;
block_ptr[68] = 1;
block_ptr[69] = 1;
block_ptr[70] = 0;
block_ptr[71] = 0;
block_ptr[72] = 0;
block_ptr[73] = 0;
block_ptr[74] = 0;
block_ptr[75] = 0;
block_ptr[76] = 1;
block_ptr[77] = 1;
block_ptr[78] = 0;
block_ptr[79] = 0;

block_ptr[80] = 1;
block_ptr[81] = 1;
block_ptr[82] = 0;
block_ptr[83] = 0;
block_ptr[84] = 1;
block_ptr[85] = 1;
block_ptr[86] = 0;
block_ptr[87] = 0;
block_ptr[88] = 1;
block_ptr[89] = 1;
block_ptr[90] = 0;
block_ptr[91] = 0;
block_ptr[92] = 1;
block_ptr[93] = 1;
block_ptr[94] = 0;
block_ptr[95] = 0;
```

```
block_ptr[80+16] = 0;
block_ptr[81+16] = 0;
block_ptr[82+16] = 1;
block_ptr[83+16] = 1;
block_ptr[84+16] = 1;
block_ptr[85+16] = 1;
block_ptr[86+16] = 0;
block_ptr[87+16] = 0;
block_ptr[88+16] = 0;
block_ptr[89+16] = 0;
block_ptr[90+16] = 1;
block_ptr[91+16] = 1;
block_ptr[92+16] = 1;
block_ptr[93+16] = 1;
block_ptr[94+16] = 0;
block_ptr[95+16] = 0;

block_ptr[80+32] = 1;
block_ptr[81+32] = 1;
block_ptr[82+32] = 1;
block_ptr[83+32] = 1;
block_ptr[84+32] = 1;
block_ptr[85+32] = 1;
block_ptr[86+32] = 0;
block_ptr[87+32] = 0;
block_ptr[88+32] = 1;
block_ptr[89+32] = 1;
block_ptr[90+32] = 1;
block_ptr[91+32] = 1;
block_ptr[92+32] = 1;
block_ptr[93+32] = 1;
block_ptr[94+32] = 0;
block_ptr[95+32] = 0;

block_ptr[128] = 0;
block_ptr[129] = 0;
block_ptr[130] = 0;
block_ptr[131] = 0;
block_ptr[132] = 0;
block_ptr[133] = 0;
block_ptr[134] = 1;
block_ptr[135] = 1;
block_ptr[136] = 0;
block_ptr[137] = 0;
block_ptr[138] = 0;
block_ptr[139] = 0;
block_ptr[140] = 0;
block_ptr[141] = 0;
block_ptr[142] = 1;
block_ptr[143] = 1;

block_ptr[128+16] = 1;
block_ptr[129+16] = 1;
block_ptr[130+16] = 0;
block_ptr[131+16] = 0;
block_ptr[132+16] = 0;
block_ptr[133+16] = 0;
block_ptr[134+16] = 1;
block_ptr[135+16] = 1;
block_ptr[136+16] = 1;
```

```
block_ptr[137+16] = 1;
block_ptr[138+16] = 0;
block_ptr[139+16] = 0;
block_ptr[140+16] = 0;
block_ptr[141+16] = 0;
block_ptr[142+16] = 1;
block_ptr[143+16] = 1;

block_ptr[128+32] = 0;
block_ptr[129+32] = 0;
block_ptr[130+32] = 1;
block_ptr[131+32] = 1;
block_ptr[132+32] = 0;
block_ptr[133+32] = 0;
block_ptr[134+32] = 1;
block_ptr[135+32] = 1;
block_ptr[136+32] = 0;
block_ptr[137+32] = 0;
block_ptr[138+32] = 1;
block_ptr[139+32] = 1;
block_ptr[140+32] = 0;
block_ptr[141+32] = 0;
block_ptr[142+32] = 1;
block_ptr[143+32] = 1;

block_ptr[128+48] = 1;
block_ptr[129+48] = 1;
block_ptr[130+48] = 1;
block_ptr[131+48] = 1;
block_ptr[132+48] = 0;
block_ptr[133+48] = 0;
block_ptr[134+48] = 1;
block_ptr[135+48] = 1;
block_ptr[136+48] = 1;
block_ptr[137+48] = 1;
block_ptr[138+48] = 1;
block_ptr[139+48] = 1;
block_ptr[140+48] = 0;
block_ptr[141+48] = 0;
block_ptr[142+48] = 1;
block_ptr[143+48] = 1;

block_ptr[128+64] = 0;
block_ptr[129+64] = 0;
block_ptr[130+64] = 0;
block_ptr[131+64] = 0;
block_ptr[132+64] = 1;
block_ptr[133+64] = 1;
block_ptr[134+64] = 1;
block_ptr[135+64] = 1;
block_ptr[136+64] = 0;
block_ptr[137+64] = 0;
block_ptr[138+64] = 0;
block_ptr[139+64] = 0;
block_ptr[140+64] = 1;
block_ptr[141+64] = 1;
block_ptr[142+64] = 1;
block_ptr[143+64] = 1;

block_ptr[128+80] = 1;
```

```
block_ptr[129+80] = 1;
block_ptr[130+80] = 0;
block_ptr[131+80] = 0;
block_ptr[132+80] = 1;
block_ptr[133+80] = 1;
block_ptr[134+80] = 1;
block_ptr[135+80] = 1;
block_ptr[136+80] = 1;
block_ptr[137+80] = 1;
block_ptr[138+80] = 0;
block_ptr[139+80] = 0;
block_ptr[140+80] = 1;
block_ptr[141+80] = 1;
block_ptr[142+80] = 1;
block_ptr[143+80] = 1;

block_ptr[128+96] = 0;
block_ptr[129+96] = 0;
block_ptr[130+96] = 1;
block_ptr[131+96] = 1;
block_ptr[132+96] = 1;
block_ptr[133+96] = 1;
block_ptr[134+96] = 1;
block_ptr[135+96] = 1;
block_ptr[136+96] = 0;
block_ptr[137+96] = 0;
block_ptr[138+96] = 1;
block_ptr[139+96] = 1;
block_ptr[140+96] = 1;
block_ptr[141+96] = 1;
block_ptr[142+96] = 1;
block_ptr[143+96] = 1;

block_ptr[128+112] = 1;
block_ptr[129+112] = 1;
block_ptr[130+112] = 1;
block_ptr[131+112] = 1;
block_ptr[132+112] = 1;
block_ptr[133+112] = 1;
block_ptr[134+112] = 1;
block_ptr[135+112] = 1;
block_ptr[136+112] = 1;
block_ptr[137+112] = 1;
block_ptr[138+112] = 1;
block_ptr[139+112] = 1;
block_ptr[140+112] = 1;
block_ptr[141+112] = 1;
block_ptr[142+112] = 1;
block_ptr[143+112] = 1;


for(i=0;i<2816;i++)
roundkey_ptr[i] = 0;

int p;
for(p=0;p<11;p++)
build_round_key(oldroundkey, roundkey_ptr,p);
//This "expands" the roundkey.  Every single bit of the nonexpanded roundkey becomes
//64 bits here because everything is repeated 64 times.
```

```
/*
cout<<"roundkey = "<<endl;
for(l=0;l<128;l++)
cout<<roundkey_ptr[l*2]<<", ";
cout<<endl<<endl;


cout<<"input = "<<endl;
for(l=0;l<128;l++)
cout<<block_ptr[l*2]<<", ";
cout<<endl;
*/

//Row and col do not correspond to row and column.  They are simply variables and
//should betreated as such.  The purpose of the  80*col+64*row term is to perform the
//shiftrows transformation without having to do any extra work.

int counter;

addroundkey(block_ptr, roundkey_ptr, 0);  //The initial addition of the roundkey.


for(counter=1;counter<10;counter++) //The 9 normal rounds in a full encryption
{
      for(row=0;row<4;row++)
      {     for(col=0;col<4;col++)
            {
block_byte_ptr = block_ptr + (80*col + 64*row)%256;
tblock_byte_ptr = tblock_ptr + (64*row + 16*col);
sbox(block_byte_ptr, tblock_byte_ptr); //16 elements per bit-sliced byte, 4 columns
per row
            }
      }

for(col=0;col<4;col++)
      {
block_byte_ptr = block_ptr + 64*col;
tblock_byte_ptr = tblock_ptr + 64*col;
mixcolumns(block_byte_ptr, tblock_byte_ptr);
      }

addroundkey(block_ptr, roundkey_ptr, counter);
}


//The final round.  Notice that it is missing the MixColumns operation.
      for(row=0;row<4;row++)
      {     for(col=0;col<4;col++)
            {
block_byte_ptr = block_ptr + (80*col + 64*row)%256;
tblock_byte_ptr = tblock_ptr + (64*row + 16*col);
sbox(block_byte_ptr, tblock_byte_ptr);
            }
      }


addroundkey(tblock_ptr, roundkey_ptr, 10);

/*
```

```
for(row=0;row<4;row++)
{     for(col=0;col<4;col++)
      {
block_byte_ptr = block_ptr + (80*col + 64*row)%256;
tblock_byte_ptr = tblock_ptr + (64*row + 16*col);
sbox(block_byte_ptr, tblock_byte_ptr); //16 elements per bit-sliced byte, 4 columns
per row
      }
}


addroundkey(block_ptr, roundkey_ptr, 10);
*/


//This small function outputs the "top" value in each of the 64-bit extended bits.
//Notice that the outputs are normally in a strange format.  For each byte, the
//values are low to high.  That is changed to a high to low normal format in this
//output function.

cout<<endl;
for(l=0;l<16;l++)
{
      for(i=0;i<8;i++)
cout<<tblock_ptr[14-2*i+l*16]<<", ";
}
cout<<endl<<endl;


}


//-------------------------------------------------------------------------------
-
void sbox(unsigned long int* block_byte_ptr, unsigned long int* tblock_byte_ptr)
{
//The purpose of the assembly level code is to access the mmx registers.
//Without writing in assembly level code, I do not know of any way to access these
//registers.
//Assembly level coding also allows me to get the most efficiency out of my program.
      __asm{
      emms
//might want to push eax here, and pop it later

      mov eax, [block_byte_ptr]  //note: movq mm0, [block_byte_ptr]  does not work;
                                                                        //  it puts
the value of the pointer in mm0,
                                                                        //  not the
value of block[0]

      movq mm0, [eax]         //Calling values from block
      movq mm1, [eax+8]
      movq mm2, [eax+16]
      movq mm3, [eax+24]
      movq mm4, [eax+32]
      movq mm5, [eax+40]
      movq mm6, [eax+48]
      movq mm7, [eax+56]
```

```
//Temporary debug step shown.  These will appear at different points in the program
//and were used for debugging (just to show what debugging was like)
//zzzzz (to easialy be able to find a certain point by searching for zzzzz)
//store values temporarily to print out inverse of GF(2^8) entry
movq [GFinverse], mm0
movq [GFinverse+8], mm1
movq [GFinverse+16], mm2
movq [GFinverse+24], mm3
movq [GFinverse+32], mm4
movq [GFinverse+40], mm5
movq [GFinverse+48], mm6
movq [GFinverse+56], mm7
}
cout<<"input values of block to sbox function = \n";
for (i=0;i<8;i++)
      cout<<"%08X %08X\n"<<GFinverse[2*i]<<GFinverse[2*i+1];
cout<<"Hit key to continue...\n";
scanf("%c",&pause);
end debug*/


//This entire first part of the s-box transformation is mainly concerned with finding
//the inverse of the GF(2^8) polynomial in the mmx state.
//In order to do this, the polynomial must be taken to the 254th power.
//Much of this is repetitive, so it takes very few comments to understand the code.


//This small subroutine is repeated fairly often.  It takes the polynomial
//corresponding to the current mmx state and determines its square.
            movq [t+8], mm1
            movq [t+16], mm2
            movq [t+24], mm3
            movq [t+40], mm5

            pxor mm0, mm4
            pxor mm0, mm6

            movq mm1, mm7
            pxor mm1, mm4
            pxor mm1, mm6

            movq mm2, mm5
            pxor mm2, [t+8]

            movq mm3, mm4
            pxor mm3, mm5
            pxor mm3, mm6
            pxor mm3, mm7

            pxor mm4, [t+16]
            pxor mm4, mm7

            pxor mm5, mm6

            pxor mm7, mm6

            movq mm6, [t+24]
            pxor mm6, [t+40]

            movq [m], mm0
```

```
            movq [m+8], mm1
            movq [m+16], mm2
            movq [m+24], mm3
            movq [m+32], mm4
            movq [m+40], mm5
            movq [m+48], mm6
            movq [m+56], mm7        //m now has power 2

            movq [t+8], mm1
            movq [t+16], mm2
            movq [t+24], mm3
            movq [t+40], mm5

            pxor mm0, mm4
            pxor mm0, mm6

            movq mm1, mm7
            pxor mm1, mm4
            pxor mm1, mm6

            movq mm2, mm5
            pxor mm2, [t+8]

            movq mm3, mm4
            pxor mm3, mm5
            pxor mm3, mm6
            pxor mm3, mm7

            pxor mm4, [t+16]
            pxor mm4, mm7

            pxor mm5, mm6

            pxor mm7, mm6

            movq mm6, [t+24]
            pxor mm6, [t+40]

            movq [c], mm0
            movq [c+8], mm1
            movq [c+16], mm2
            movq [c+24], mm3
            movq [c+32], mm4
            movq [c+40], mm5
            movq [c+48], mm6
            movq [c+56], mm7        //c now has power 4

//This subcode "multiplies" the value stored in m by the value currently in the
//registres, which is stored in c.  Multiplication is very time consuming - however,
//there is no way to get around the fact that at least 4 multiplications must be
//performed to find the inverse.

                  pand mm0, [m]
                  movq [t], mm0
                  movq mm0, [c]

                  pand mm1, [m]
                  pand mm0, [m+8]
                  pxor mm1, mm0
                  movq [t+8], mm1
```

```
movq mm0, [c]
movq mm1, [c+8]

pand mm2, [m]
pand mm1, [m+8]
pand mm0, [m+16]
pxor mm2, mm1
pxor mm2, mm0
movq [t+16], mm2
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]

pand mm3, [m]
pand mm2, [m+8]
pand mm1, [m+16]
pand mm0, [m+24]
pxor mm3, mm2
pxor mm3, mm1
pxor mm3, mm0
movq [t+24], mm3
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]

pand mm4, [m]
pand mm3, [m+8]
pand mm2, [m+16]
pand mm1, [m+24]
pand mm0, [m+32]
pxor mm4, mm3
pxor mm4, mm2
pxor mm4, mm1
pxor mm4, mm0
movq [t+32], mm4
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]


pand mm5, [m]
pand mm4, [m+8]
pand mm3, [m+16]
pand mm2, [m+24]
pand mm1, [m+32]
pand mm0, [m+40]
pxor mm5, mm4
pxor mm5, mm3
pxor mm5, mm2
pxor mm5, mm1
pxor mm5, mm0
movq [t+40], mm5
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
```

```
movq mm5, [c+40]

pand mm6, [m]
pand mm5, [m+8]
pand mm4, [m+16]
pand mm3, [m+24]
pand mm2, [m+32]
pand mm1, [m+40]
pand mm0, [m+48]
pxor mm6, mm5
pxor mm6, mm4
pxor mm6, mm3
pxor mm6, mm2
pxor mm6, mm1
pxor mm6, mm0
movq [t+48], mm6
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]

pand mm7, [m]
pand mm6, [m+8]
pand mm5, [m+16]
pand mm4, [m+24]
pand mm3, [m+32]
pand mm2, [m+40]
pand mm1, [m+48]
pand mm0, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
pxor mm7, mm0
movq [t+56], mm7
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+8]
pand mm6, [m+16]
pand mm5, [m+24]
pand mm4, [m+32]
pand mm3, [m+40]
pand mm2, [m+48]
pand mm1, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
```

```
pxor mm7, mm1
movq [t+64], mm7
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+16]
pand mm6, [m+24]
pand mm5, [m+32]
pand mm4, [m+40]
pand mm3, [m+48]
pand mm2, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
movq [t+72], mm7
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+24]
pand mm6, [m+32]
pand mm5, [m+40]
pand mm4, [m+48]
pand mm3, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
movq [t+80], mm7
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+32]
pand mm6, [m+40]
pand mm5, [m+48]
pand mm4, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
movq [t+88], mm7
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+40]
pand mm6, [m+48]
pand mm5, [m+56]
pxor mm7, mm6
pxor mm7, mm5
movq [t+96], mm7
```

```
                    movq mm6, [c+48]
                    movq mm7, [c+56]

                    pand mm7, [m+48]
                    pand mm6, [m+56]
                    pxor mm7, mm6
                    movq [t+104], mm7
                    movq mm7, [c+56]

                    pand mm7, [m+56]
                    movq [t+112], mm7


//Modulus for multiplication.  It is stored in m[16]
//-----------------------------------------------------------------------------

                    movq mm0, [t]
                    movq mm1, [t+8]
                    movq mm2, [t+16]
                    movq mm3, [t+24]
                    movq mm4, [t+32]
                    movq mm5, [t+40]
                    movq mm6, [t+48]
                    movq mm7, [t+56]

                    pxor mm0, [t+64]
                    pxor mm0, [t+96]
                    pxor mm0, [t+104]

                    pxor mm1, [t+64]
                    pxor mm1, [t+72]
                    pxor mm1, [t+96]
                    pxor mm1, [t+112]

                    pxor mm2, [t+72]
                    pxor mm2, [t+80]
                    pxor mm2, [t+104]

                    pxor mm3, [t+64]
                    pxor mm3, [t+80]
                    pxor mm3, [t+88]
                    pxor mm3, [t+96]
                    pxor mm3, [t+104]
                    pxor mm3, [t+112]

                    pxor mm4, [t+64]
                    pxor mm4, [t+72]
                    pxor mm4, [t+88]
                    pxor mm4, [t+112]

                    pxor mm5, [t+72]
                    pxor mm5, [t+80]
                    pxor mm5, [t+96]

                    pxor mm6, [t+80]
                    pxor mm6, [t+88]
                    pxor mm6, [t+104]

                    pxor mm7, [t+88]
                    pxor mm7, [t+96]
```

```
        pxor mm7, [t+112]

        movq [m], mm0
        movq [m+8], mm1
        movq [m+16], mm2
        movq [m+24], mm3
        movq [m+32], mm4
        movq [m+40], mm5
        movq [m+48], mm6
        movq [m+56], mm7  //Now m has power 6
        //Doublesquare - should be replaced.

movq [t+8], mm1
movq [t+16], mm2
movq [t+24], mm3
movq [t+40], mm5

pxor mm0, mm4
pxor mm0, mm6

movq mm1, mm7
pxor mm1, mm4
pxor mm1, mm6

movq mm2, mm5
pxor mm2, [t+8]

movq mm3, mm4
pxor mm3, mm5
pxor mm3, mm6
pxor mm3, mm7

pxor mm4, [t+16]
pxor mm4, mm7

pxor mm5, mm6

pxor mm7, mm6

movq mm6, [t+24]
pxor mm6, [t+40]

movq [t+8], mm1
movq [t+16], mm2
movq [t+24], mm3
movq [t+40], mm5

pxor mm0, mm4
pxor mm0, mm6

movq mm1, mm7
pxor mm1, mm4
pxor mm1, mm6

movq mm2, mm5
pxor mm2, [t+8]

movq mm3, mm4
pxor mm3, mm5
pxor mm3, mm6
```

```
        pxor mm3, mm7

        pxor mm4, [t+16]
        pxor mm4, mm7

        pxor mm5, mm6

        pxor mm7, mm6

        movq mm6, [t+24]
        pxor mm6, [t+40]

movq [c], mm0
movq [c+8], mm1
movq [c+16], mm2
movq [c+24], mm3
movq [c+32], mm4
movq [c+40], mm5
movq [c+48], mm6
movq [c+56], mm7 //Now c has power 24


              pand mm0, [m]
              movq [t], mm0
              movq mm0, [c]

              pand mm1, [m]
              pand mm0, [m+8]
              pxor mm1, mm0
              movq [t+8], mm1
              movq mm0, [c]
              movq mm1, [c+8]

              pand mm2, [m]
              pand mm1, [m+8]
              pand mm0, [m+16]
              pxor mm2, mm1
              pxor mm2, mm0
              movq [t+16], mm2
              movq mm0, [c]
              movq mm1, [c+8]
              movq mm2, [c+16]

              pand mm3, [m]
              pand mm2, [m+8]
              pand mm1, [m+16]
              pand mm0, [m+24]
              pxor mm3, mm2
              pxor mm3, mm1
              pxor mm3, mm0
              movq [t+24], mm3
              movq mm0, [c]
              movq mm1, [c+8]
              movq mm2, [c+16]
              movq mm3, [c+24]

              pand mm4, [m]
              pand mm3, [m+8]
              pand mm2, [m+16]
              pand mm1, [m+24]
```

```
pand mm0, [m+32]
pxor mm4, mm3
pxor mm4, mm2
pxor mm4, mm1
pxor mm4, mm0
movq [t+32], mm4
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]


pand mm5, [m]
pand mm4, [m+8]
pand mm3, [m+16]
pand mm2, [m+24]
pand mm1, [m+32]
pand mm0, [m+40]
pxor mm5, mm4
pxor mm5, mm3
pxor mm5, mm2
pxor mm5, mm1
pxor mm5, mm0
movq [t+40], mm5
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]

pand mm6, [m]
pand mm5, [m+8]
pand mm4, [m+16]
pand mm3, [m+24]
pand mm2, [m+32]
pand mm1, [m+40]
pand mm0, [m+48]
pxor mm6, mm5
pxor mm6, mm4
pxor mm6, mm3
pxor mm6, mm2
pxor mm6, mm1
pxor mm6, mm0
movq [t+48], mm6
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]

pand mm7, [m]
pand mm6, [m+8]
pand mm5, [m+16]
pand mm4, [m+24]
pand mm3, [m+32]
pand mm2, [m+40]
```

```
pand mm1, [m+48]
pand mm0, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
pxor mm7, mm0
movq [t+56], mm7
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+8]
pand mm6, [m+16]
pand mm5, [m+24]
pand mm4, [m+32]
pand mm3, [m+40]
pand mm2, [m+48]
pand mm1, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
movq [t+64], mm7
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+16]
pand mm6, [m+24]
pand mm5, [m+32]
pand mm4, [m+40]
pand mm3, [m+48]
pand mm2, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
movq [t+72], mm7
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+24]
pand mm6, [m+32]
pand mm5, [m+40]
```

```
                pand mm4, [m+48]
                pand mm3, [m+56]
                pxor mm7, mm6
                pxor mm7, mm5
                pxor mm7, mm4
                pxor mm7, mm3
                movq [t+80], mm7
                movq mm4, [c+32]
                movq mm5, [c+40]
                movq mm6, [c+48]
                movq mm7, [c+56]

                pand mm7, [m+32]
                pand mm6, [m+40]
                pand mm5, [m+48]
                pand mm4, [m+56]
                pxor mm7, mm6
                pxor mm7, mm5
                pxor mm7, mm4
                movq [t+88], mm7
                movq mm5, [c+40]
                movq mm6, [c+48]
                movq mm7, [c+56]

                pand mm7, [m+40]
                pand mm6, [m+48]
                pand mm5, [m+56]
                pxor mm7, mm6
                pxor mm7, mm5
                movq [t+96], mm7
                movq mm6, [c+48]
                movq mm7, [c+56]

                pand mm7, [m+48]
                pand mm6, [m+56]
                pxor mm7, mm6
                movq [t+104], mm7
                movq mm7, [c+56]

                pand mm7, [m+56]
                movq [t+112], mm7


//Modulus for multiplication.  It is stored in m[16]
//-----------------------------------------------------------------------------
                movq mm0, [t]
                movq mm1, [t+8]
                movq mm2, [t+16]
                movq mm3, [t+24]
                movq mm4, [t+32]
                movq mm5, [t+40]
                movq mm6, [t+48]
                movq mm7, [t+56]

                pxor mm0, [t+64]
                pxor mm0, [t+96]
                pxor mm0, [t+104]

                pxor mm1, [t+64]
                pxor mm1, [t+72]
```

```
            pxor mm1, [t+96]
            pxor mm1, [t+112]

            pxor mm2, [t+72]
            pxor mm2, [t+80]
            pxor mm2, [t+104]

            pxor mm3, [t+64]
            pxor mm3, [t+80]
            pxor mm3, [t+88]
            pxor mm3, [t+96]
            pxor mm3, [t+104]
            pxor mm3, [t+112]

            pxor mm4, [t+64]
            pxor mm4, [t+72]
            pxor mm4, [t+88]
            pxor mm4, [t+112]

            pxor mm5, [t+72]
            pxor mm5, [t+80]
            pxor mm5, [t+96]

            pxor mm6, [t+80]
            pxor mm6, [t+88]
            pxor mm6, [t+104]

            pxor mm7, [t+88]
            pxor mm7, [t+96]
            pxor mm7, [t+112]

            movq [m], mm0
            movq [m+8], mm1
            movq [m+16], mm2
            movq [m+24], mm3
            movq [m+32], mm4
            movq [m+40], mm5
            movq [m+48], mm6
            movq [m+56], mm7 //Now m has power 30


movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

      movq [t+8], mm1
      movq [t+16], mm2
      movq [t+24], mm3
      movq [t+40], mm5

      pxor mm0, mm4
      pxor mm0, mm6

      movq mm1, mm7
      pxor mm1, mm4
```

```
        pxor mm1, mm6

        movq mm2, mm5
        pxor mm2, [t+8]

        movq mm3, mm4
        pxor mm3, mm5
        pxor mm3, mm6
        pxor mm3, mm7

        pxor mm4, [t+16]
        pxor mm4, mm7

        pxor mm5, mm6

        pxor mm7, mm6

        movq mm6, [t+24]
        pxor mm6, [t+40]

        movq [t+8], mm1
        movq [t+16], mm2
        movq [t+24], mm3
        movq [t+40], mm5

        pxor mm0, mm4
        pxor mm0, mm6

        movq mm1, mm7
        pxor mm1, mm4
        pxor mm1, mm6

        movq mm2, mm5
        pxor mm2, [t+8]

        movq mm3, mm4
        pxor mm3, mm5
        pxor mm3, mm6
        pxor mm3, mm7

        pxor mm4, [t+16]
        pxor mm4, mm7

        pxor mm5, mm6

        pxor mm7, mm6

        movq mm6, [t+24]
        pxor mm6, [t+40]

movq [c], mm0
movq [c+8], mm1
movq [c+16], mm2
movq [c+24], mm3
movq [c+32], mm4
movq [c+40], mm5
movq [c+48], mm6
movq [c+56], mm7  //Now c has power 96

             pand mm0, [m]
```

```
movq [t], mm0
movq mm0, [c]

pand mm1, [m]
pand mm0, [m+8]
pxor mm1, mm0
movq [t+8], mm1
movq mm0, [c]
movq mm1, [c+8]

pand mm2, [m]
pand mm1, [m+8]
pand mm0, [m+16]
pxor mm2, mm1
pxor mm2, mm0
movq [t+16], mm2
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]

pand mm3, [m]
pand mm2, [m+8]
pand mm1, [m+16]
pand mm0, [m+24]
pxor mm3, mm2
pxor mm3, mm1
pxor mm3, mm0
movq [t+24], mm3
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]

pand mm4, [m]
pand mm3, [m+8]
pand mm2, [m+16]
pand mm1, [m+24]
pand mm0, [m+32]
pxor mm4, mm3
pxor mm4, mm2
pxor mm4, mm1
pxor mm4, mm0
movq [t+32], mm4
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]


pand mm5, [m]
pand mm4, [m+8]
pand mm3, [m+16]
pand mm2, [m+24]
pand mm1, [m+32]
pand mm0, [m+40]
pxor mm5, mm4
pxor mm5, mm3
pxor mm5, mm2
pxor mm5, mm1
```

```
pxor mm5, mm0
movq [t+40], mm5
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]

pand mm6, [m]
pand mm5, [m+8]
pand mm4, [m+16]
pand mm3, [m+24]
pand mm2, [m+32]
pand mm1, [m+40]
pand mm0, [m+48]
pxor mm6, mm5
pxor mm6, mm4
pxor mm6, mm3
pxor mm6, mm2
pxor mm6, mm1
pxor mm6, mm0
movq [t+48], mm6
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]

pand mm7, [m]
pand mm6, [m+8]
pand mm5, [m+16]
pand mm4, [m+24]
pand mm3, [m+32]
pand mm2, [m+40]
pand mm1, [m+48]
pand mm0, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
pxor mm7, mm0
movq [t+56], mm7
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+8]
pand mm6, [m+16]
pand mm5, [m+24]
pand mm4, [m+32]
pand mm3, [m+40]
```

```
pand mm2, [m+48]
pand mm1, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
movq [t+64], mm7
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+16]
pand mm6, [m+24]
pand mm5, [m+32]
pand mm4, [m+40]
pand mm3, [m+48]
pand mm2, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
movq [t+72], mm7
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+24]
pand mm6, [m+32]
pand mm5, [m+40]
pand mm4, [m+48]
pand mm3, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
movq [t+80], mm7
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [m+32]
pand mm6, [m+40]
pand mm5, [m+48]
pand mm4, [m+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
movq [t+88], mm7
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]
```

```
                    pand mm7, [m+40]
                    pand mm6, [m+48]
                    pand mm5, [m+56]
                    pxor mm7, mm6
                    pxor mm7, mm5
                    movq [t+96], mm7
                    movq mm6, [c+48]
                    movq mm7, [c+56]

                    pand mm7, [m+48]
                    pand mm6, [m+56]
                    pxor mm7, mm6
                    movq [t+104], mm7
                    movq mm7, [c+56]

                    pand mm7, [m+56]
                    movq [t+112], mm7


//Modulus for multiplication.  It is stored in m[16]
//-------------------------------------------------------------------------------

                    movq mm0, [t]
                    movq mm1, [t+8]
                    movq mm2, [t+16]
                    movq mm3, [t+24]
                    movq mm4, [t+32]
                    movq mm5, [t+40]
                    movq mm6, [t+48]
                    movq mm7, [t+56]

                    pxor mm0, [t+64]
                    pxor mm0, [t+96]
                    pxor mm0, [t+104]

                    pxor mm1, [t+64]
                    pxor mm1, [t+72]
                    pxor mm1, [t+96]
                    pxor mm1, [t+112]

                    pxor mm2, [t+72]
                    pxor mm2, [t+80]
                    pxor mm2, [t+104]

                    pxor mm3, [t+64]
                    pxor mm3, [t+80]
                    pxor mm3, [t+88]
                    pxor mm3, [t+96]
                    pxor mm3, [t+104]
                    pxor mm3, [t+112]

                    pxor mm4, [t+64]
                    pxor mm4, [t+72]
                    pxor mm4, [t+88]
                    pxor mm4, [t+112]

                    pxor mm5, [t+72]
                    pxor mm5, [t+80]
                    pxor mm5, [t+96]
```

```
pxor mm6, [t+80]
pxor mm6, [t+88]
pxor mm6, [t+104]

pxor mm7, [t+88]
pxor mm7, [t+96]
pxor mm7, [t+112]

movq [c], mm0
movq [c+8], mm1
movq [c+16], mm2
movq [c+24], mm3
movq [c+32], mm4
movq [c+40], mm5
movq [c+48], mm6
movq [c+56], mm7  //Now c has power 126

pand mm0, [eax]
movq [t], mm0
movq mm0, [c]

pand mm1, [eax]
pand mm0, [eax+8]
pxor mm1, mm0
movq [t+8], mm1
movq mm0, [c]
movq mm1, [c+8]

pand mm2, [eax]
pand mm1, [eax+8]
pand mm0, [eax+16]
pxor mm2, mm1
pxor mm2, mm0
movq [t+16], mm2
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]

pand mm3, [eax]
pand mm2, [eax+8]
pand mm1, [eax+16]
pand mm0, [eax+24]
pxor mm3, mm2
pxor mm3, mm1
pxor mm3, mm0
movq [t+24], mm3
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]

pand mm4, [eax]
pand mm3, [eax+8]
pand mm2, [eax+16]
pand mm1, [eax+24]
pand mm0, [eax+32]
pxor mm4, mm3
pxor mm4, mm2
pxor mm4, mm1
```

```
pxor mm4, mm0
movq [t+32], mm4
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]


pand mm5, [eax]
pand mm4, [eax+8]
pand mm3, [eax+16]
pand mm2, [eax+24]
pand mm1, [eax+32]
pand mm0, [eax+40]
pxor mm5, mm4
pxor mm5, mm3
pxor mm5, mm2
pxor mm5, mm1
pxor mm5, mm0
movq [t+40], mm5
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]

pand mm6, [eax]
pand mm5, [eax+8]
pand mm4, [eax+16]
pand mm3, [eax+24]
pand mm2, [eax+32]
pand mm1, [eax+40]
pand mm0, [eax+48]
pxor mm6, mm5
pxor mm6, mm4
pxor mm6, mm3
pxor mm6, mm2
pxor mm6, mm1
pxor mm6, mm0
movq [t+48], mm6
movq mm0, [c]
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]

pand mm7, [eax]
pand mm6, [eax+8]
pand mm5, [eax+16]
pand mm4, [eax+24]
pand mm3, [eax+32]
pand mm2, [eax+40]
pand mm1, [eax+48]
pand mm0, [eax+56]
pxor mm7, mm6
pxor mm7, mm5
```

```
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
pxor mm7, mm0
movq [t+56], mm7
movq mm1, [c+8]
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [eax+8]
pand mm6, [eax+16]
pand mm5, [eax+24]
pand mm4, [eax+32]
pand mm3, [eax+40]
pand mm2, [eax+48]
pand mm1, [eax+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
pxor mm7, mm1
movq [t+64], mm7
movq mm2, [c+16]
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [eax+16]
pand mm6, [eax+24]
pand mm5, [eax+32]
pand mm4, [eax+40]
pand mm3, [eax+48]
pand mm2, [eax+56]
pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm4
pxor mm7, mm3
pxor mm7, mm2
movq [t+72], mm7
movq mm3, [c+24]
movq mm4, [c+32]
movq mm5, [c+40]
movq mm6, [c+48]
movq mm7, [c+56]

pand mm7, [eax+24]
pand mm6, [eax+32]
pand mm5, [eax+40]
pand mm4, [eax+48]
pand mm3, [eax+56]
pxor mm7, mm6
pxor mm7, mm5
```

```
                    pxor mm7, mm4
                    pxor mm7, mm3
                    movq [t+80], mm7
                    movq mm4, [c+32]
                    movq mm5, [c+40]
                    movq mm6, [c+48]
                    movq mm7, [c+56]

                    pand mm7, [eax+32]
                    pand mm6, [eax+40]
                    pand mm5, [eax+48]
                    pand mm4, [eax+56]
                    pxor mm7, mm6
                    pxor mm7, mm5
                    pxor mm7, mm4
                    movq [t+88], mm7
                    movq mm5, [c+40]
                    movq mm6, [c+48]
                    movq mm7, [c+56]

                    pand mm7, [eax+40]
                    pand mm6, [eax+48]
                    pand mm5, [eax+56]
                    pxor mm7, mm6
                    pxor mm7, mm5
                    movq [t+96], mm7
                    movq mm6, [c+48]
                    movq mm7, [c+56]

                    pand mm7, [eax+48]
                    pand mm6, [eax+56]
                    pxor mm7, mm6
                    movq [t+104], mm7
                    movq mm7, [c+56]

                    pand mm7, [eax+56]
                    movq [t+112], mm7


//Modulus for multiplication.  It is stored in mmx
//-------------------------------------------------------------------------------

                    movq mm0, [t]
                    movq mm1, [t+8]
                    movq mm2, [t+16]
                    movq mm3, [t+24]
                    movq mm4, [t+32]
                    movq mm5, [t+40]
                    movq mm6, [t+48]
                    movq mm7, [t+56]

                    pxor mm0, [t+64]
                    pxor mm0, [t+96]
                    pxor mm0, [t+104]

                    pxor mm1, [t+64]
                    pxor mm1, [t+72]
                    pxor mm1, [t+96]
                    pxor mm1, [t+112]
```

```
                    pxor mm2, [t+72]
                    pxor mm2, [t+80]
                    pxor mm2, [t+104]

                    pxor mm3, [t+64]
                    pxor mm3, [t+80]
                    pxor mm3, [t+88]
                    pxor mm3, [t+96]
                    pxor mm3, [t+104]
                    pxor mm3, [t+112]

                    pxor mm4, [t+64]
                    pxor mm4, [t+72]
                    pxor mm4, [t+88]
                    pxor mm4, [t+112]

                    pxor mm5, [t+72]
                    pxor mm5, [t+80]
                    pxor mm5, [t+96]

                    pxor mm6, [t+80]
                    pxor mm6, [t+88]
                    pxor mm6, [t+104]

                    pxor mm7, [t+88]
                    pxor mm7, [t+96]
                    pxor mm7, [t+112]

 //Now mmx has power 127

              movq [t+8], mm1
              movq [t+16], mm2
              movq [t+24], mm3
              movq [t+40], mm5

              pxor mm0, mm4
              pxor mm0, mm6

              movq mm1, mm7
              pxor mm1, mm4
              pxor mm1, mm6

              movq mm2, mm5
              pxor mm2, [t+8]

              movq mm3, mm4
              pxor mm3, mm5
              pxor mm3, mm6
              pxor mm3, mm7

              pxor mm4, [t+16]
              pxor mm4, mm7

              pxor mm5, mm6

              pxor mm7, mm6

              movq mm6, [t+24]
              pxor mm6, [t+40]
/*//debug
```

```
//zzzzz
//store values temporarily to print out inverse of GF(2^8) entry
movq [GFinverse], mm0
movq [GFinverse+8], mm1
movq [GFinverse+16], mm2
movq [GFinverse+24], mm3
movq [GFinverse+32], mm4
movq [GFinverse+40], mm5
movq [GFinverse+48], mm6
movq [GFinverse+56], mm7
}
printf("inverse of GF(2^8) entry = \n");
for (i=0;i<8;i++)
      printf("%08X %08X\n",GFinverse[2*i],GFinverse[2*i+1]);

printf("Hit key to continue...\n");
scanf("%c",&pause);
__asm
{
      mov eax, [block_byte_ptr]
}
*/
//end debug

//Affine transformation.  This multiplies the current mmx "state" by a matrix and
//then invertes several of the registers.
            movq [t], mm5
            pxor mm0, mm4
            pxor mm0, mm5
            pxor mm0, mm6
            pxor mm0, mm7

            pxor mm1, mm0
            pxor mm1, mm4

            pxor mm2, mm1
            pxor mm2, mm5

            movq mm5, mm0
            pxor mm5, mm3
            pxor mm5, mm2

            pxor mm3, mm2
            pxor mm3, mm6

            movq mm6, mm1
            pxor mm6, mm3
            pxor mm6, mm4

            pxor mm4, mm3
            pxor mm4, mm7

            movq mm7, [t]
            pxor mm7, mm2
            pxor mm7, mm4

            mov eax, [inverse_ptr]
            pxor mm0, [eax]
            pxor mm1, [eax]
            pxor mm5, [eax]
```

```
            pxor mm6, [eax]

            mov eax, [tblock_byte_ptr]
            movq [eax], mm0
            movq [eax+8], mm1
            movq [eax+16], mm2
            movq [eax+24], mm3
            movq [eax+32], mm4
            movq [eax+40], mm5
            movq [eax+48], mm6
            movq [eax+56], mm7

            movq [GFinverse], mm0
            movq [GFinverse+8], mm1
            movq [GFinverse+16], mm2
            movq [GFinverse+24], mm3
            movq [GFinverse+32], mm4
            movq [GFinverse+40], mm5
            movq [GFinverse+48], mm6
            movq [GFinverse+56], mm7
}
/*
            cout<<"inverse + affine transformation = "<<endl;
for (i=0;i<8;i++)
      cout<<"%08X %08X"<<endl<<GFinverse[2*i]<<GFinverse[2*i+1];

cout<<"Hit key to continue..."<<endl;
scanf("%c",&pause);

__asm
{
      mov eax, [block_byte_ptr]
}

//end debug
//debug
/*
      cout<<"It gets to here"<<endl;

for(p=0;p<8;p++)
      cout<<"%08X"<<endl<<block_ptr[2*p];
*/
//end debug

//    __asm{
//          popa
//          emms
//    }
}

//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------

//MixColumns Transformation
//The sets of values are stored in unsigned long int tblock[65]

/*This transformation involves taking four polynomials simultaneously and multiplying
  them by a matrix.  This matrix is multiplied by the vector.
( x         1           1           x+1   ) (byte 1)       (nbyte 1)
( x+1 x           1           1           ) (byte 2)       (nbyte 2)
```

```
( 1         x+1   x           1           ) (byte 3) =      (nbyte 3)
( 1         1           x+1   x           ) (byte 4)        (nbyte 4)
The way that my function performs this multiplication is that it lets nbytes 2, 3,
and 4 be
equal to byte 1.  It then multiplies byte 1 by x and lets nbyte 1 equal this value.
Then, it
xors nbyte 4 with the original value.

The function then loads bytes 2, 3, and 4 into memory and performs the necessary xor
operations with them.
Notice that the transformation x+1 is equal to taking an xor, then multiplying by x
and taking
another xor.
*/
void mixcolumns(unsigned long int* block_byte_ptr, unsigned long int*
tblock_byte_ptr)

{
            __asm{
            //This is necessary just like in the first function.
            mov eax, [block_byte_ptr]
            mov ebx, [tblock_byte_ptr]

            //Loading the first byte
            movq mm0, [ebx]
            movq mm1, [ebx+8]
            movq mm2, [ebx+16]
            movq mm3, [ebx+24]
            movq mm4, [ebx+32]
            movq mm5, [ebx+40]
            movq mm6, [ebx+48]
            movq mm7, [ebx+56]

            movq [eax+64], mm0  //Movement is allowed for byte 1 only
            movq [eax+72], mm1
            movq [eax+80], mm2
            movq [eax+88], mm3
            movq [eax+96], mm4
            movq [eax+104], mm5
            movq [eax+112], mm6
            movq [eax+120], mm7

            movq [eax+128], mm0
            movq [eax+136], mm1
            movq [eax+144], mm2
            movq [eax+152], mm3
            movq [eax+160], mm4
            movq [eax+168], mm5
            movq [eax+176], mm6
            movq [eax+184], mm7

            movq [eax+192], mm0
            movq [eax+200], mm1
            movq [eax+208], mm2
            movq [eax+216], mm3
            movq [eax+224], mm4
            movq [eax+232], mm5
            movq [eax+240], mm6
            movq [eax+248], mm7
```

```
            pxor mm0, mm7                       //Multiplying by x for byte 1
            pxor mm2, mm7
            pxor mm3, mm7

            movq [eax], mm7               //Because nbyte 1 has not yet been filled,
we can move directly into it
            movq [eax+8], mm0
            movq [eax+16], mm1
            movq [eax+24], mm2
            movq [eax+32], mm3
            movq [eax+40], mm4
            movq [eax+48], mm5
            movq [eax+56], mm6

            pxor mm7, [eax+192]
            movq [eax+192], mm7
            pxor mm0, [eax+200]
            movq [eax+200], mm0
            pxor mm1, [eax+208]
            movq [eax+208], mm1
            pxor mm2, [eax+216]
            movq [eax+216], mm2
            pxor mm3, [eax+224]
            movq [eax+224], mm3
            pxor mm4, [eax+232]
            movq [eax+232], mm4
            pxor mm5, [eax+240]
            movq [eax+240], mm5
            pxor mm6, [eax+248]
            movq [eax+248], mm6
            //Notice that, because byte 1 was not changed after being multiplied
            //by x, it does not need to be "reloaded" as some later bytes do.

            //Operations for byte 2

            movq mm0, [ebx+64]
            movq mm1, [ebx+72]
            movq mm2, [ebx+80]
            movq mm3, [ebx+88]
            movq mm4, [ebx+96]
            movq mm5, [ebx+104]
            movq mm6, [ebx+112]
            movq mm7, [ebx+120]

            pxor mm0, [eax]  //xor operations with nbyte 1
            movq [eax], mm0
            pxor mm1, [eax+8]
            movq [eax+8], mm1
            pxor mm2, [eax+16]
            movq [eax+16], mm2
            pxor mm3, [eax+24]
            movq [eax+24], mm3
            pxor mm4, [eax+32]
            movq [eax+32], mm4
            pxor mm5, [eax+40]
            movq [eax+40], mm5
            pxor mm6, [eax+48]
            movq [eax+48], mm6
            pxor mm7, [eax+56]
            movq [eax+56], mm7
```

```
movq mm0, [ebx+64]  //Reloading byte 1
movq mm1, [ebx+72]
movq mm2, [ebx+80]
movq mm3, [ebx+88]
movq mm4, [ebx+96]
movq mm5, [ebx+104]
movq mm6, [ebx+112]
movq mm7, [ebx+120]

pxor mm0, [eax+128]  //xor operations with nbyte 3
movq [eax+128], mm0
pxor mm1, [eax+136]
movq [eax+136], mm1
pxor mm2, [eax+144]
movq [eax+144], mm2
pxor mm3, [eax+152]
movq [eax+152], mm3
pxor mm4, [eax+160]
movq [eax+160], mm4
pxor mm5, [eax+168]
movq [eax+168], mm5
pxor mm6, [eax+176]
movq [eax+176], mm6
pxor mm7, [eax+184]
movq [eax+184], mm7

movq mm0, [ebx+64]            //Reloading byte 1
movq mm1, [ebx+72]
movq mm2, [ebx+80]
movq mm3, [ebx+88]
movq mm4, [ebx+96]
movq mm5, [ebx+104]
movq mm6, [ebx+112]
movq mm7, [ebx+120]

pxor mm0, [eax+192]           //xor operations with nbyte 4
movq [eax+192], mm0
pxor mm1, [eax+200]
movq [eax+200], mm1
pxor mm2, [eax+208]
movq [eax+208], mm2
pxor mm3, [eax+216]
movq [eax+216], mm3
pxor mm4, [eax+224]
movq [eax+224], mm4
pxor mm5, [eax+232]
movq [eax+232], mm5
pxor mm6, [eax+240]
movq [eax+240], mm6
pxor mm7, [eax+248]
movq [eax+248], mm7

movq mm0, [ebx+64]            //Reloading byte 1
movq mm1, [ebx+72]
movq mm2, [ebx+80]
movq mm3, [ebx+88]
movq mm4, [ebx+96]
movq mm5, [ebx+104]
movq mm6, [ebx+112]
```

```
        movq mm7, [ebx+120]

        pxor mm0, mm7                           //Multiplying by x for byte 2
        pxor mm2, mm7
        pxor mm3, mm7

        pxor mm7, [eax]                     //xor operation with nbyte 1
        movq [eax], mm7
        pxor mm0, [eax+8]
        movq [eax+8], mm0
        pxor mm1, [eax+16]
        movq [eax+16], mm1
        pxor mm2, [eax+24]
        movq [eax+24], mm2
        pxor mm3, [eax+32]
        movq [eax+32], mm3
        pxor mm4, [eax+40]
        movq [eax+40], mm4
        pxor mm5, [eax+48]
        movq [eax+48], mm5
        pxor mm6, [eax+56]
        movq [eax+56], mm6

//Reloading byte 1.Notice that this reloades the original value of byte 1,
//so this byte must be multiplied by x again.
        movq mm0, [ebx+64]
        movq mm1, [ebx+72]
        movq mm2, [ebx+80]
        movq mm3, [ebx+88]
        movq mm4, [ebx+96]
        movq mm5, [ebx+104]
        movq mm6, [ebx+112]
        movq mm7, [ebx+120]

        pxor mm0, mm7
        pxor mm2, mm7
        pxor mm3, mm7

        pxor mm7, [eax+64]      //xoring with nbyte 2.
        movq [eax+64], mm7
        pxor mm0, [eax+72]
        movq [eax+72], mm0
        pxor mm1, [eax+80]
        movq [eax+80], mm1
        pxor mm2, [eax+88]
        movq [eax+88], mm2
        pxor mm3, [eax+96]
        movq [eax+96], mm3
        pxor mm4, [eax+104]
        movq [eax+104], mm4
        pxor mm5, [eax+112]
        movq [eax+112], mm5
        pxor mm6, [eax+120]
        movq [eax+120], mm6

        //Operations for byte 3

        movq mm0, [ebx+128]
        movq mm1, [ebx+136]
        movq mm2, [ebx+144]
```

```
movq mm3, [ebx+152]
movq mm4, [ebx+160]
movq mm5, [ebx+168]
movq mm6, [ebx+176]
movq mm7, [ebx+184]

pxor mm0, [eax]                    //xor operations
movq [eax], mm0
pxor mm1, [eax+8]
movq [eax+8], mm1
pxor mm2, [eax+16]
movq [eax+16], mm2
pxor mm3, [eax+24]
movq [eax+24], mm3
pxor mm4, [eax+32]
movq [eax+32], mm4
pxor mm5, [eax+40]
movq [eax+40], mm5
pxor mm6, [eax+48]
movq [eax+48], mm6
pxor mm7, [eax+56]
movq [eax+56], mm7

movq mm0, [ebx+128]
movq mm1, [ebx+136]
movq mm2, [ebx+144]
movq mm3, [ebx+152]
movq mm4, [ebx+160]
movq mm5, [ebx+168]
movq mm6, [ebx+176]
movq mm7, [ebx+184]

pxor mm0, [eax+64]
movq [eax+64], mm0
pxor mm1, [eax+72]
movq [eax+72], mm1
pxor mm2, [eax+80]
movq [eax+80], mm2
pxor mm3, [eax+88]
movq [eax+88], mm3
pxor mm4, [eax+96]
movq [eax+96], mm4
pxor mm5, [eax+104]
movq [eax+104], mm5
pxor mm6, [eax+112]
movq [eax+112], mm6
pxor mm7, [eax+120]
movq [eax+120], mm7

movq mm0, [ebx+128]
movq mm1, [ebx+136]
movq mm2, [ebx+144]
movq mm3, [ebx+152]
movq mm4, [ebx+160]
movq mm5, [ebx+168]
movq mm6, [ebx+176]
movq mm7, [ebx+184]

pxor mm0, [eax+192]
movq [eax+192], mm0
```

```
pxor mm1, [eax+200]
movq [eax+200], mm1
pxor mm2, [eax+208]
movq [eax+208], mm2
pxor mm3, [eax+216]
movq [eax+216], mm3
pxor mm4, [eax+224]
movq [eax+224], mm4
pxor mm5, [eax+232]
movq [eax+232], mm5
pxor mm6, [eax+240]
movq [eax+240], mm6
pxor mm7, [eax+248]
movq [eax+248], mm7

movq mm0, [ebx+128]
movq mm1, [ebx+136]
movq mm2, [ebx+144]
movq mm3, [ebx+152]
movq mm4, [ebx+160]
movq mm5, [ebx+168]
movq mm6, [ebx+176]
movq mm7, [ebx+184]

pxor mm0, mm7                          //Multiplying by x for byte 3
pxor mm2, mm7
pxor mm3, mm7

pxor mm7, [eax+64]
movq [eax+64], mm7
pxor mm0, [eax+72]
movq [eax+72], mm0
pxor mm1, [eax+80]
movq [eax+80], mm1
pxor mm2, [eax+88]
movq [eax+88], mm2
pxor mm3, [eax+96]
movq [eax+96], mm3
pxor mm4, [eax+104]
movq [eax+104], mm4
pxor mm5, [eax+112]
movq [eax+112], mm5
pxor mm6, [eax+120]
movq [eax+120], mm6

movq mm0, [ebx+128]
movq mm1, [ebx+136]
movq mm2, [ebx+144]
movq mm3, [ebx+152]
movq mm4, [ebx+160]
movq mm5, [ebx+168]
movq mm6, [ebx+176]
movq mm7, [ebx+184]

pxor mm0, mm7
pxor mm2, mm7
pxor mm3, mm7

pxor mm7, [eax+128]
movq [eax+128], mm7
```

```
pxor mm0, [eax+136]
movq [eax+136], mm0
pxor mm1, [eax+144]
movq [eax+144], mm1
pxor mm2, [eax+152]
movq [eax+152], mm2
pxor mm3, [eax+160]
movq [eax+160], mm3
pxor mm4, [eax+168]
movq [eax+168], mm4
pxor mm5, [eax+176]
movq [eax+176], mm5
pxor mm6, [eax+184]
movq [eax+184], mm6

//Operations for byte 4

movq mm0, [ebx+192]
movq mm1, [ebx+200]
movq mm2, [ebx+208]
movq mm3, [ebx+216]
movq mm4, [ebx+224]
movq mm5, [ebx+232]
movq mm6, [ebx+240]
movq mm7, [ebx+248]

pxor mm0, [eax]
movq [eax], mm0
pxor mm1, [eax+8]
movq [eax+8], mm1
pxor mm2, [eax+16]
movq [eax+16], mm2
pxor mm3, [eax+24]
movq [eax+24], mm3
pxor mm4, [eax+32]
movq [eax+32], mm4
pxor mm5, [eax+40]
movq [eax+40], mm5
pxor mm6, [eax+48]
movq [eax+48], mm6
pxor mm7, [eax+56]
movq [eax+56], mm7

movq mm0, [ebx+192]
movq mm1, [ebx+200]
movq mm2, [ebx+208]
movq mm3, [ebx+216]
movq mm4, [ebx+224]
movq mm5, [ebx+232]
movq mm6, [ebx+240]
movq mm7, [ebx+248]

pxor mm0, [eax+64]
movq [eax+64], mm0
pxor mm1, [eax+72]
movq [eax+72], mm1
pxor mm2, [eax+80]
movq [eax+80], mm2
pxor mm3, [eax+88]
movq [eax+88], mm3
```

```
pxor mm4, [eax+96]
movq [eax+96], mm4
pxor mm5, [eax+104]
movq [eax+104], mm5
pxor mm6, [eax+112]
movq [eax+112], mm6
pxor mm7, [eax+120]
movq [eax+120], mm7

movq mm0, [ebx+192]
movq mm1, [ebx+200]
movq mm2, [ebx+208]
movq mm3, [ebx+216]
movq mm4, [ebx+224]
movq mm5, [ebx+232]
movq mm6, [ebx+240]
movq mm7, [ebx+248]

pxor mm0, [eax+128]
movq [eax+128], mm0
pxor mm1, [eax+136]
movq [eax+136], mm1
pxor mm2, [eax+144]
movq [eax+144], mm2
pxor mm3, [eax+152]
movq [eax+152], mm3
pxor mm4, [eax+160]
movq [eax+160], mm4
pxor mm5, [eax+168]
movq [eax+168], mm5
pxor mm6, [eax+176]
movq [eax+176], mm6
pxor mm7, [eax+184]
movq [eax+184], mm7

movq mm0, [ebx+192]
movq mm1, [ebx+200]
movq mm2, [ebx+208]
movq mm3, [ebx+216]
movq mm4, [ebx+224]
movq mm5, [ebx+232]
movq mm6, [ebx+240]
movq mm7, [ebx+248]

pxor mm0, mm7                       //Multiplying by x for byte 4
pxor mm2, mm7
pxor mm3, mm7

pxor mm7, [eax+128]
movq [eax+128], mm7
pxor mm0, [eax+136]
movq [eax+136], mm0
pxor mm1, [eax+144]
movq [eax+144], mm1
pxor mm2, [eax+152]
movq [eax+152], mm2
pxor mm3, [eax+160]
movq [eax+160], mm3
pxor mm4, [eax+168]
movq [eax+168], mm4
```

```
            pxor mm5, [eax+176]
            movq [eax+176], mm5
            pxor mm6, [eax+184]
            movq [eax+184], mm6

            movq mm0, [ebx+192]
            movq mm1, [ebx+200]
            movq mm2, [ebx+208]
            movq mm3, [ebx+216]
            movq mm4, [ebx+224]
            movq mm5, [ebx+232]
            movq mm6, [ebx+240]
            movq mm7, [ebx+248]

            pxor mm0, mm7
            pxor mm2, mm7
            pxor mm3, mm7

            pxor mm7, [eax+192]
            movq [eax+192], mm7
            pxor mm0, [eax+200]
            movq [eax+200], mm0
            pxor mm1, [eax+208]
            movq [eax+208], mm1
            pxor mm2, [eax+216]
            movq [eax+216], mm2
            pxor mm3, [eax+224]
            movq [eax+224], mm3
            pxor mm4, [eax+232]
            movq [eax+232], mm4
            pxor mm5, [eax+240]

            movq [eax+240], mm5
            pxor mm6, [eax+248]
            movq [eax+248], mm6

            emms        //Empties the mmx registers.  Simply a formality.
}
      }
/*
for(p=0;p<4;p++)
{ for(q=0;q<4;q++)
cout<<block[2*p+8*p]<<endl;

cout << "Hit return to continue";
cin.get(pause);
}
*/

/*Tentative coding for fourth power.  This will replace the squaring twice in a row.
The fourth power coding is more efficient.
movq [t+32], mm4
movq [t+56], mm7

pxor mm4, mm1
pxor mm4, mm2
pxor mm4, mm5
pxor mm4, mm6

movq mm1, mm6
```

```
pxor mm1, mm5
pxor mm1, mm2
pxor mm1, mm3

pxor mm0, mm1
pxor mm0, mm7

pxor mm1, [t+32]

movq [t+32], mm2
pxor mm2, mm5
pxor mm2, mm7

pxor mm7, mm6
pxor mm7, mm5
pxor mm7, mm3

movq mm5, mm6
pxor mm5, mm3

movq mm6, [t+32]
pxor mm6, [t+56]

pxor mm3, mm2
pxor mm3, [t+32]
*/

void addroundkey(unsigned long int* block_ptr, unsigned long int* roundkey_ptr, int
key)
{
      unsigned long int* r;
      unsigned long int* key2;

//debug
//cout << "block_ptr[0+16], [0+17] = " <<block_ptr[0+16] << "," <<block_ptr[0+17] <<
endl;
//cout << "rounddkey_ptr[0+16], [0+17] = " <<roundkey_ptr[0+16] << ","
//          <<roundkey_ptr[0+17] << endl;
//debug

for(i=0;i<16;i++)
//for (i=0;i<2;i++)
{
//r = block_ptr + 64*i;
//key2 = 256*key + roundkey_ptr + 64*i;
r = block_ptr + 16*i;
key2 = roundkey_ptr + 16*i + 256*key;

//debug
//cout << "beginning of XOR round" << endl;
//cout << "r,r+1 = " <<r[0] << "," <<r[1] << endl;
//cout << "key2,key2+1 = " <<key2[0] << ","
//          <<key2[1] << endl;
//debug

      __asm
            {
mov eax, [r]
mov ebx, [key2]
movq mm0, [eax]         //Moving values from block.
```

```
movq mm1, [eax+8]
movq mm2, [eax+16]
movq mm3, [eax+24]
movq mm4, [eax+32]
movq mm5, [eax+40]
movq mm6, [eax+48]
movq mm7, [eax+56]

pxor mm0, [ebx]  //XORing values with the roundkey
pxor mm1, [ebx+8]
pxor mm2, [ebx+16]
pxor mm3, [ebx+24]
pxor mm4, [ebx+32]
pxor mm5, [ebx+40]
pxor mm6, [ebx+48]
pxor mm7, [ebx+56]

movq [eax], mm0  //Moving the values back into their positions.
movq [eax+8], mm1
movq [eax+16], mm2
movq [eax+24], mm3
movq [eax+32], mm4
movq [eax+40], mm5
movq [eax+48], mm6
movq [eax+56], mm7
}
//debug
//cout << "end of XOR round" << endl;
//cout << "r,r+1 = " <<r[0] << "," <<r[1] << endl;
//cout << "key2,key2+1 = " <<key2[0] << ","
//          <<key2[1] << endl;
//debug
}  //end for
}

void build_round_key(unsigned char* oldroundkey, unsigned long int* roundkey_ptr, int
key)
{
int a;
int b;
int k;
      for(j=0;j<16;j++)
{
      for(k=0;k<8;k++)
      {
//a = 7 - k;
                  b = 2*k+16*j+256*key;
//    if((oldroundkey[(j+16*key)]>>(a))&0X01==1)
      if((oldroundkey[(j+16*key)]>>k)&0X01==1)
      {
            //These should be 0XFFFFFFFF, but they are only a single digit 1 so that
the output is
            //easier to read.
            roundkey_ptr[b] = 1;
            roundkey_ptr[b+1] = 1;
      }
      }
}}
```

// **********************End Bitslice10.cpp Program** -- Team 050 Manzano High School ********

Appendix B: Project Code Explanation -- Technical Aspects

The bitslicing program is written almost entirely in assembly level code. This makes it more efficient, but also makes reading the code difficult. The mm# commands refer to special registers called the MMX registers, which contain 64 bits each, as opposed to 32 bits for a normal register. The code consists mostly of movq commands and pxor commands, with a few pand commands thrown in. A movq command moves 8 bytes, or a "quadword," either from a register to a memory location, or to a memory location to a register. A pxor performs a bitwise XOR command between two registers or a register and a memory location and stores the result in the first register. Notice that the result for a pxor or a pand command cannot be directly stored in a memory location. This means that to take an XOR command of register mm0 and memory location [target], the following code must be used:

pxor mm0, [target]

movq [target], mm0.

Note that if mm0 is to be used for any other purpose after this operation, it must be "reloaded" to its original value.

A general pattern for the bitsliced program goes as follows:

1: Declare functions, variables, and arrays. Also declare several pointers to be used for optimization of the array storage (explained in detail below).

2: Input of test values for the program. No AddRoundKey transformation is used in the bitsliced program. However, there is a function for the creation of round keys in the control program.

3: For loops which control the s-box and the mixcolumns transformation. The for loops, as explained below, also allow the shiftrows transformation to be performed for free.

4: The output at the end of mixcolumns. The output is in the form smallest to largest, first byte to last byte. This requires a careful interpretation because the format that the AES algorithm information sheet uses for its test values is closer to largest to smallest, first byte to last byte.

5: The S-box transformation. This consists of three sub functions: multiplication, squares, and an affine transformation at the end. The square and multiplication functions are mixed as described in the Mathematical Background sheet.

5.1: The square transformation is the most simple of the three. It finds the square of the polynomial given by the current byte. The advantage it has over multiplication of two different

polynomials is that there are a lot of cancellations, allowing the code to be shorter. The simplification is fairly easy also – notice that the x^n term goes to the x^2n term, which can then be simplified.

5.2: The multiplication transformation is more complicated, because it multiplies two different polynomials, and there are no cancellations. Essentially, to find the n^{th} term, one must multiply the constant term of the first polynomial (by using a pand operation) with the n^{th} term of the second XOR the x term of the first polynomial AND the $(n-1)^{th}$ XOR…

5.3: The affine transformation consists of multiplication by a matrix and then inverting four of the eight bits (accomplished by a PXOR with a value set at 1).

6: The last part is the mixcolumns transformation function. This transformation is described in the Mathematical Background sheet, and the coding is fairly straightforward, involving a lot of reloading of values (because the PXOR operation does not allow the destination to be a memory register.)

Clarification of Code Pieces

In the bitsliced program, there are a number of lines which seem extraneous, but are in fact essential for the proper function of the program. Other lines may simply seem confusing. Some of these lines, listed below, are explained in detail.

```
inverse_ptr = inverse;
if ((((unsigned long int)inverse_ptr)%8)!=0)
        inverse_ptr++;
```

An unsigned long integer only stores 4 bytes of memory. This means that when an array of unsigned long integers is created, then the pointer to the first value in the array will be divisible by four bytes. However, the bitslicing operations run on 8 bytes of memory. There is a performance penalty if the pointer value for the array is not divisible by 8. By creating a new pointer to a place four bytes past the original if the original is not divisible by 8, the performance penalty is avoided and the code is optimized.

```
//-------------------------------------------------------------------------------------------

for(row=0;row<4;row++)
{       for(col=0;col<4;col++)
{
block_byte_ptr = block_ptr + (80*col + 64*row)%256;
```

```
tblock_byte_ptr = tblock_ptr + (64*row + 16*col);
sbox(block_byte_ptr, tblock_byte_ptr); //16 elements per bit-sliced byte, 4 columns per row
}
}
```

This for loop allows for no time to be spent on the shiftrows transformation. This loop runs the S-box 16 times, taking the values from the storage "block" and placing them into temporary storage in "tblock," while at the same time performing the shiftrows operation. Notice that by using the temporary storage block, the shiftrows takes effectively zero time.

```
//-----------------------------------------------------------------------------------------

for(col=0;col<4;col++)
{
block_byte_ptr = block_ptr + 64*col;
tblock_byte_ptr = tblock_ptr + 64*col;
mixcolumns(block_byte_ptr, tblock_byte_ptr);
}
```

This part of the program calls the mixcolumns function four times. It takes the temporary values from tblock and places them in block, where (if a full round were performed), they would be XORed with the Roundkey and then put back into the S-box.

```
//-----------------------------------------------------------------------------------------
mov eax, [block_byte_ptr]

movq mm0, [eax]
movq mm1, [eax+8]
movq mm2, [eax+16]
movq mm3, [eax+24]
movq mm4, [eax+32]
movq mm5, [eax+40]
movq mm6, [eax+48]
```

```
movq mm7, [eax+56]
```

The block_byte_ptr, as shown in the second example, is the position in the "block" where the values are coming from. Because it varies, it is critical to temporarily store it in the eax general-purpose register. Otherwise, the movq command moves the numerical value of the pointer into the register, as opposed to what the pointer is pointing to.

```
//-----------------------------------------------------------------------------------------------------
```

```
movq mm0, [ebx+192]
movq mm1, [ebx+200]
movq mm2, [ebx+208]
movq mm3, [ebx+216]
movq mm4, [ebx+224]
movq mm5, [ebx+232]
movq mm6, [ebx+240]
movq mm7, [ebx+248]
```

This part is repeated fairly often, and the purpose was explained previously. Because the PXOR command must be performed inside the mm# register, this changes the value of the register, which must be reloaded frequently.

Appendix C: Mathematical Equations

For the S-box, the polynomial used for modulus is $x^8 + x^4 + x^3 + x + 1$. The first transformation involves taking the inverse. This part is explained in the code itself. The matrix for the affine transformation is:

$$\begin{pmatrix} Endbyte\,0 \\ Endbyte\,1 \\ Endbyte\,2 \\ Endbyte\,3 \\ Endbyte\,4 \\ Endbyte\,5 \\ Endbyte\,6 \\ Endbyte\,7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} Current\,0 \\ Current\,1 \\ Current\,2 \\ Current\,3 \\ Current\,4 \\ Current\,5 \\ Current\,6 \\ Current\,7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

For the ShiftRows transformation, the "state" is divided into a 4 x 4 matrix of bytes. The transformation is as follows:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ f & g & h & e \\ k & l & i & j \\ p & m & n & o \end{pmatrix}$$

For the MixColumns transformation, the "state" is divided into a 4 x 4 matrix, broken down into vectors and multiplied, modulus $x^8 + x^4 + x^3 + x + 1$, with the matrix $\begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix}$.

For the ExpandRoundKey, the RoundKey is generated by first taking the last 4 bytes of the last RoundKey, Left Circular Shifting them and then applying the S-box. Next, a "round constant" determined by $x^{round\# - 1}$ modulus
$x^8 + x^4 + x^3 + x + 1$ is XORed. Finally, this value is XORed with the first 4 bytes of the last RoundKey. Successive four bytes are generated by taking the last 4 bytes in the current RoundKey and XORing them with the corresponding 4 bytes in the last RoundKey.