CSEE 4840

128-bit AES decryption

Shrivathsa Bhargav

Larry Chen

Abhinandan Majumdar

Shiva Ramudit

Spring 2008 Project – Design Document

CSEE 4840

Table of contents

FPGA-based 128-bit AES decryption

Shrivathsa Bhargav, Larry Chen, Abhinandan Majumdar, Shiva Ramudit

{sb2784, lc2454, am2993, syr9}@columbia.edu

ABSTRACT

The goals for this project are clearly outlined below. The first is to successfully read a Bitmap image from an SD-card, and display it; the second is to decrypt an encrypted image (stored in SRAM). The final goal of this project will be to interface these two modular processes.

1. INTRODUCTION

There will be two peripherals: the VGA monitor and the SD-card reader. These will be controlled by the VGA controller and the SPI controller respectively. A VGA controller is needed to maintain the frame-buffer, as well as provide the important data as well as HSYNC, VSYNC, and blanking signals to the VGA peripheral. An SPI controller is the easiest way to interface to an SD-card since the SD-card will not have a file-system; rather, it will have the encrypted Bitmap image stored as raw data (starting from block 0) in an 8-bit grayscale format.
The main function of Nios-II processor (hereafter Nios) is to supervise the whole process, as well as act as a conduit for data flowing between the individual blocks. When the decryption process is done, the VGA controller (which will use the SRAM as a frame-buffer for the decrypted image to be displayed) will display the decrypted image.

2. HARDWARE DESIGN

The entire design can be broken up into several modules, listed below:

1. **AES decrypto**. This module takes in 128-bit blocks of data, performs AES (AKA Rijndael) decryption with a hardcoded 128-bit key. The results of this process are stored in the SRAM.
2. **SD-card SPI interface**. This is needed to read raw image data from the MMC/SD-card.
3. **VGA and SRAM controller**. The decrypted image, assumed to be stored in the SRAM at block 0, is used as a frame-buffer. The image is then shown on the VGA monitor. This block communicates with the off-chip SRAM (512k). The SRAM will be used to house the decrypted data, and will act as a frame buffer for the VGA controller.
4. **Nios**. Nios will supervise the whole operation (which will be sequential, as the main bottleneck of the operation is the speed with which data can be retrieved from the SD-card) and act as the conduit for data traveling between various blocks. As such, it will need to be developed simultaneously with the decryption and the read and display modules.

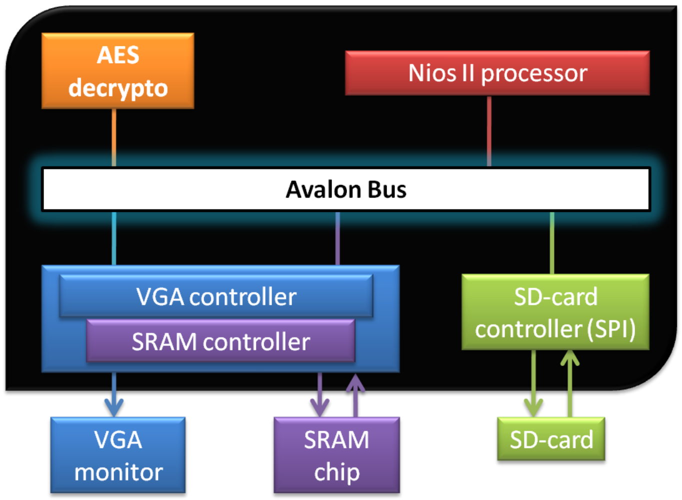


Figure 1 - The block diagram of the entire system. The black boundary represents the blocks within the FPGA. Arrows show data flow; lines show hard connections between modules.

2.1 AES decrypto

This fancily-named block performs the most important operation in the whole project; it accepts 128-bit data from Nios, decrypts it and then sends it back to Nios. 128-bit decryption needs a 128-bit key and 128-bit cipher text to decrypt, and generates the 128-bit decrypted original data.
It must be noted here that the source data is encrypted beforehand (even before it is placed on the SD card) through a custom-coded C program that can encrypt and decrypt arbitrary size files. This program's code is listed in Appendix A.

2.1.1 Algorithm

The AES decryption [1] basically traverses the encryption algorithm in the opposite direction. From the block level diagram, it can be seen that AES decrypto initially performs key-expansion on the 128-bit key block that creates all intermediate keys (which are generated from the original key during encryption for every round). Then it executes an inverse add round key which performs an xor operation of the cipher text with the modified key (generated in last iteration of the encryption process) from key expansion. After this step, the AES decrypto repeats the following steps 9 times: inverse shift row (which shifts each i^{th} row of the matrix by i elements right), inverse sub bytes which replaces each 8 bits of the matrix by a corresponding 8 bit value from the inverse S-Box, an inverse add round key, and an inverse mix column which performs modulo multiplication with MDS matrix in Rijndael's finite field. As a last iteration, it does an inverse shift row, inverse sub bytes and inverse add round key to generate the original data.

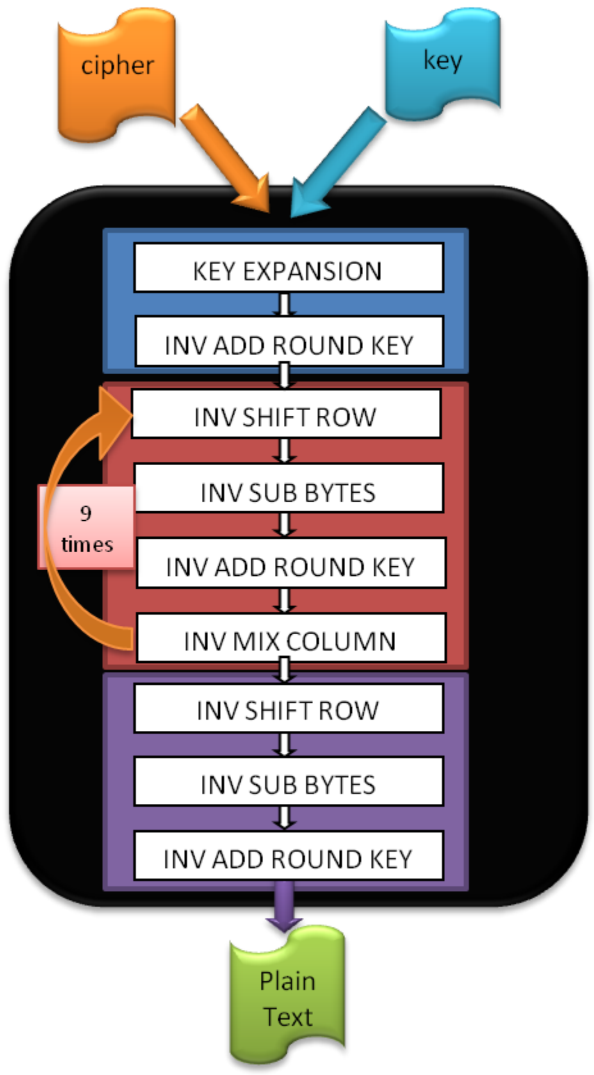


Figure 2 – AES 128-bit Decryption Algorithm

2.1.2 Optimized Hardware Design

Considering SD CARD as the main source of latency in reading the block, we plan to optimize our design at three levels.

a) Elimination of Key Expansion Unit by statically storing all processed keys being used in successive iterations.
b) Elimination of inverse shift row by swapping the respective lines before sending it to inverse sub bytes.
c) Elimination of duplicate modules to save FPGA resources.

Since Nios has a 32 bit MM Master Port and AVALON bus can at most transmit 32 bit data at a time, we will need additional buffer space to store all the 128-bit data before we actually proceed with decryption. There are various dependencies within this process: each iteration is dependent upon the previous iteration's results; within a single iteration, the input values for a particular module depends upon the previous module; the data being accessed is depended on the 32 bit chunk from SD Card. Because of these dependencies, pipelining either at the inter-loop or intra-loop level is not advantageous. After buffering all the data, plain text is generated after 10 rounds of decryption and the module finally stores the data in the Output Buffer, where it is sent to Nios through the Avalon bus in 32 bit chunks.

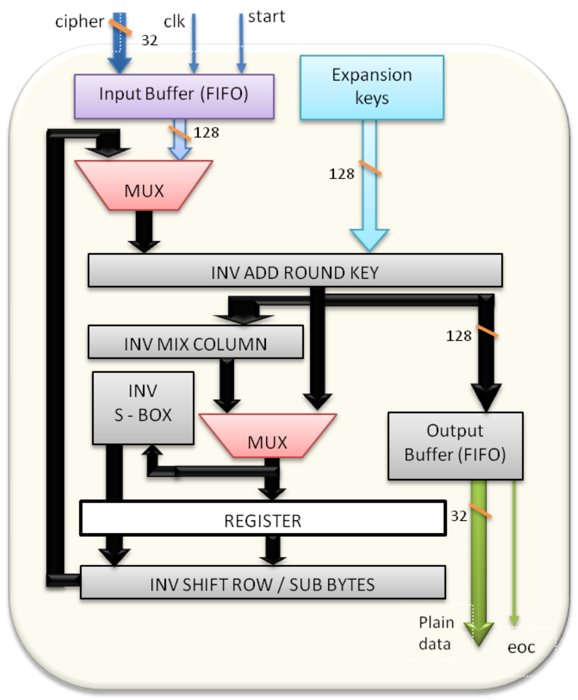


Figure 3 - AES 128-bit Decrypto Datapath

2.1.3 Timing

Since the Avalon bus transfers 32 bit data at a time, it'll take four clock cycles to buffer the input data. Once all 128-bits are buffered, the controller (not shown in the datapath) asserts the start signal instructing the decrypto unit to start the computation.
After start is asserted, it takes 1 clock cycle for initial processing (inv add round key) and 9 clock cycles for further iterations. After 9+1 = 10 clock cycles, it stores the

plain 128-bit text into the output buffer and sets the 'eoc' (end of computation) signal after 1 clock cycle instructing Nios to accept the data in 32 bit chunks.
These timings are illustrated in figures 4 and 5.

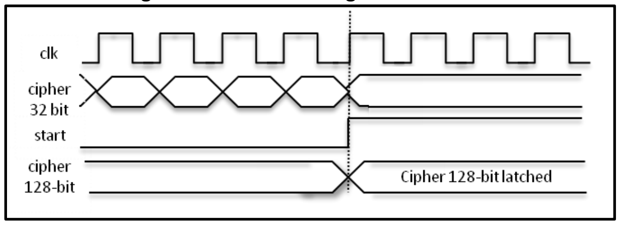


Figure 4 – Timing of Input Data Buffering

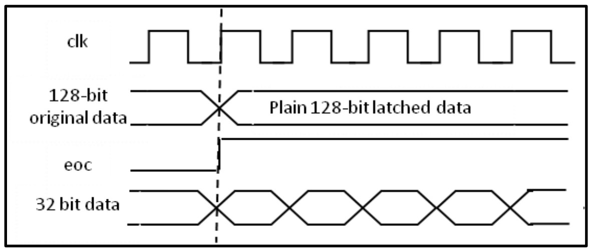


Figure 5 – Timing of final data traversal

2.2 SD-card SPI interface

It was decided that there will be no file-system implemented on the SD-card since it'll only be a hassle and a hurdle to getting the data to the AES decrypto block. Instead, an SPI interface will be used to communicate directly with the SD-card module, and the data will be read from the card, buffered into 32-bit blocks and stored in the SRAM via Nios.

Prof. Edwards has built a simple SPI-controller module for use in his Apple II demonstration. [3]

To facilitate communication with the SD card via the SPI interface, we refer to engineering application notes [4] that implement a similar functionality. While the application note discusses the interface for a MMC card, MMC's backward compatibility with SD makes the following discussion valid for our purposes [5]. However, to make clear that the interface discusses MMC and is only backward compatible with SD, we will continue our SPI interface discussion using MMC/SD instead of just SD.

2.2.1 MMC/SD Card Pin Assignments in SPI Mode

As shown in table 1, there are 7 pins defined for the MMC/SD card when it is operating in SPI mode. In particular, when pin 1 is pulled low, the corresponding MMC/SD card is selected. There is also a pull-up resistor on the DataIn and DataOut pins because MMC/SD cards drive pins in 'Open Drain' mode.

Table 1 - MMC/SD Card Pin Assignments in SPI Mode

PIN	NAME	TYPE	SPI DESCRIPTION
1	nCS	Input	Chip Select (Active LOW)
2	DataIn	Input	Host-to-Card Commands and Data
3	VSS1	Power	Supply Voltage Ground
4	VDD	VCC	Supply Voltage
5	CLK	Input	Clock
6	VSS2	Power	Supply Voltage Ground
7	DataOut	Output	Card-to-Host Data and Status

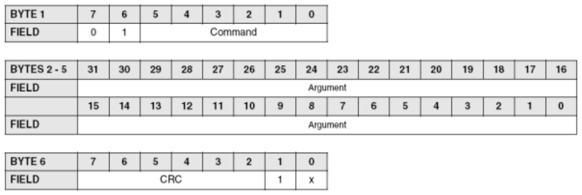
2.2.2 SPI Commands

Table 2 shows a subset of all available SPI commands used to communicate to the MMC/SD card.

Table 2 - SPI Commands

CMD INDEX	ARGUMENT	RESPONSE	COMMAND DESCRIPTION
CMD0	None	R1	Resets the MultiMediaCard
CMD1	None	R1	Activates the card Initialization process
CMD13	None	R2	Asks the selected card to send its status register
CMD16	[31:0]block length	R1	Selects a block length (in bytes) for all following block commands (read and write).
CMD17	[31:0]data address	R1	Reads a block of size selected by the SET_BLOCKLEN command
CMD24	[31:0]data address	R1	Writes a block of the size selected by the SET_BLOCKLEN command
CMD32	[31:0]data address	R1	Sets the address of the first sector of the erase group
CMD33	[31:0]data address	R1	Sets the address of the last sector in a continuous range within the selected erase group, or the address of a single sector to be selected for erase.
CMD34	[31:0]data address	R1	Removes one previously selected sector from the erase selection
CMD38	[31:0]don't care	R1b	Erases all previously selected sectors
CMD59	[31:1]don't care [0:0]CRC option	R1	Turns the CRC option on or off. A '1' in the CRC option bit will turn the option on. A '0' will turn it off.

From the table, we can see that in fact, followed by optional arguments and CRC, all commands are 6 bytes long and are transmitted MSB first. The command transmission is shown below.



BYTE 1	7	6	5	4	3	2	1	0
FIELD	0	1	Command					

BYTES 2 - 5	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	Argument															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	Argument															

BYTE 6	7	6	5	4	3	2	1	0
FIELD	CRC						1	x

Figure 6 - Command Transmission

Upon receiving the commands, the MMC/SD will first respond with a R1, R1b or R2 response that signals to the host processor the state of the received commands. If there is a CRC error or an illegal command code, the MMC/SD card will communicate that through the response. Similarly, when data is written to the MMC/SD card, the MMC/SD card will generate a data response in return. However, since we do not expect to write to the MMC/SD card in our project, we will not elaborate on that in this document. On the other hand, when we execute read commands, there are data transfers associated with them, and they are transmitted via four 515 bytes long data tokens. In the event that a read command failed, instead of transmitting the required data, it will transmit a data error token. The data token start byte and data error token structure are illustrated in the figure below.

Data Token Start Byte (Byte 1) Structure

BIT	7	6	5	4	3	2	1	0
FIELD	1	1	1	1	1	1	1	0

Data Error Token Structure

BIT	7	6	5	4	3	2	1	0
FIELD	0	0	0	0	Out_of_Range	Card_ECC_Failed	CC_Error	Error

Figure 7 - Data Token Start Byte and Data Error Token Structure

2.2.3 Mode Selection

Upon activation, the MMC/SD card will wake up in MMC mode. It will enter the SPI mode if the CS signal is asserted low during the reception of the Reset command (CMD0). In SPI mode, CRC checking is disabled by default. However, since the MMC/SD card wakes up in MMC mode, it is necessary to transfer a CRC along with CMD0. This can be confusing as the CMD0 is transferred in SPI structure, but this is defined in the specification. It is only after the MMC/SD card enters the SPI mode that the CRC becomes disabled by default.

CMD0 is a static command and always generates the same 7-bit CRC of 4Ah. Adding the '1' end bit (bit 0) to the CRC creates a CRC byte of 95h. The following hexadecimal sequence can be used to send CMD0 in all situations for SPI mode, since the CRC byte (although required) is ignored once in SPI mode. The entire CMD0 appears as: 40 00 00 00 00 95 (hexadecimal).

2.2.4. Initialization Sequence

To wake up the SD card properly, the following sequence of commands is necessary.

1. Send 80 clocks to start bus communication
2. Assert nCS LOW
3. Send CMD0
4. Send 8 clocks for delay
5. Wait for a valid response
6. If there is no response, back to step 4
7. Send 8 clocks of delay
8. Send CMD1
9. Send 8 clocks of delay
10. Wait for valid response
11. Send 8 clocks of delay
12. Repeat from step 9 until the response shows READY.

It will take a large number of clock cycles for CMD1 to finish its execution. However, once the CMD1 process is finished, the idle bit in the response will become low. It is often after this the MMC/SD card can read and write.

2.2.5 Data Read

The SPI mode supports single block read operations only. Upon reception of a valid Read command, the card will respond with a Response token followed by a Data token in the length defined by a previous SET_BLOCK_LENGTH command. The start address can be any byte address in the valid address range of the card. Every block however, must be contained in a single physical card sector. After the Data Read command is sent from microcontroller to the card, the microcontroller will need to monitor the data stream input and wait for Data Token 0xFE. Since the response start bit 0 can happen any time in the clock stream, it's necessary to use software to align the bytes being read.

2.3 VGA and SRAM controller

The VGA controller's duty is to read the raw image data from the SRAM buffer, which will be used to house the decrypted data received from the AES decrypto block (and piped through Nios).

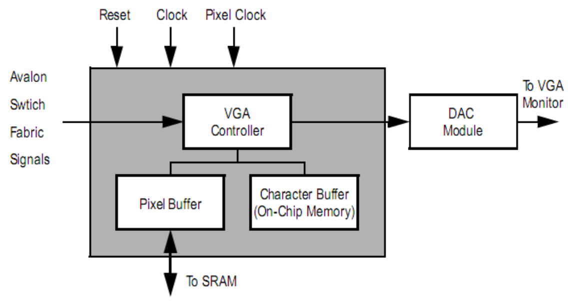


Figure 8 – Block diagram of the VGA controller

One of the major concerns at this time is the single-ported nature of the SRAM. To overcome this limitation, complete control of the SRAM is given over to the VGA controller, which then uses the stored image as a frame buffer. Lab 3's code made use of the SRAM as a frame buffer, and this code will provide the basis for the VGA controller. Some of the details of the controller are discussed below.

- The VGA controller requires both a 25MHz and a 50MHz clock to function correctly. The 25MHz clock will be generated using a clock divider, similar to what was done in lab 3.
- While the maximum resolution of the controller is 640*480 in its "pixel mode", smaller resolutions are supported by duplicating the

pixels as necessary. For our chosen resolution of 320*240 (shockingly known as Quarter VGA, or QVGA), for instance, each pixel will be drawn as a 2x2 *mega-pixel*.
- While in pixel mode, the controller supports either RGB color in 565 mode, or 8-bit grayscale. We will be using the latter.

The pixel co-ordinate system is shown in figure 9.

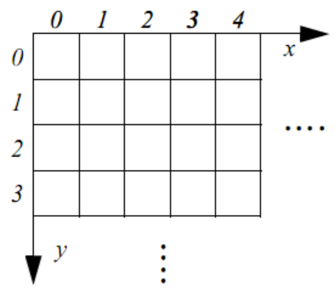


Figure 9 – Video co-ordinate system

All data for the VGA core is word-addressable (32-bit) [6]. For an 8-bit color pixel mode with the VGA controller as a slave, the addressing scheme is shown below.

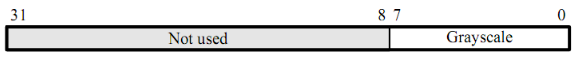


Figure 10 – Pixel slave's data format for 8-bit grayscale

The IMagic team has also based their project around the SRAM and SDRAM. [2]

3. SOFTWARE DESIGN

The software portion, while not terribly complicated, is a critical portion of the project. Nios' tasks can be broken down as follows:
- Initialize all the modules and peripherals when the system starts
- Check for presence of SD-card
- Read raw data from the SD-card in 32-bit chunks and pipe this data into the decrypto block
- Take results from the decrypto block and store them in SRAM (also in 32-bit chunks)
- When decryption is complete, notify the VGA/SRAM controller
- Wait for the reset button, then restart the whole process

There are limitations inherent in the individual modules when sending data across the Avalon bus (the main one is the 32-bit bus limit when sending data to and from Nios). Since Nios contains 64k of internal memory, there is ample room for the 32-bit blocks while they are in transit through Nios.

4. MILESTONES

Since the project is rather modular, there are two overlapping timeline projections.

25%: Communicate successfully with the SD-card (wake up) and implement minimal buffering to transport small (32-bit) blocks of data.
Implement the custom C code on Nios to perform decryption of hardcoded cipher text using hardcoded (and precomputed) keys; output on Nios terminal/console.

50%: Read unencrypted (320*240) 8-bit grayscale Bitmap images from SD-card, and display them on the VGA monitor using the SRAM as a frame buffer.
Start porting the decryption into hardware. Look into the feasibility of using the 128-bit AES open-core.

75%: Begin smoothly integrating the two modules. Discover that it is akin to breeding dogs with cats.
Give up and go home.

5. REFERENCES

[1]http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[2] IMagic. A project that read JPG files from SD-cards and displayed them on VGA. http://www1.cs.columbia.edu/~sedwards/classes/2007/4840/reports/Imagic.pdf
[3] Apple II demo by Prof. Edwards http://www1.cs.columbia.edu/~sedwards/apple2fpga/
[4] Interfacing a MultiMediaCard to the LH79520 System-On-Chip http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/lh79520.mmc.interfacing.pdf
[5] Embedded Systems Lab CSEE 4840 : Imagic Design Document http://www1.cs.columbia.edu/~sedwards/classes/2007/4840/designs/Imagic.pdf
[6] VGA core for ALtera DE2/DE1 boards, via the Altera University Program http://university.altera.com/materials/unv-ip-cores.html

```c
//AES encrypter
#include <stdio.h>

#define VERBOSE 0

unsigned short int key[4][4];
unsigned short int text[4][4];


void read_key (FILE *f) {

unsigned short int c=0x00000000;
void *t = &c;
int sz;
int i,j;
int cn;
i=j=0;


for (cn=0;cn<16;cn++) {

sz = fread(t,1,1,f);
key[i][j++] = c;
c=0x00000000;

if (j>=4) {i++;j=0;}
if(sz==0 || i >= 4) break;
}
}

int read_text (FILE *f) {

unsigned short int c=0x00000000;
void *t = &c;
int sz;
int i,j;
int cn;

i=j=0;

for (cn=0;cn<16;cn++) {

sz = fread(t,1,1,f);
text[i][j++] = sz?c:0x00;
c=0x00000000;
}

return feof(f);
}


void print_key() {
int i,j;
printf("key => \n");
for(i=0;i<4;i++) {
for(j=0;j<4;j++)
printf("%hx ",key[i][j]);
printf("\n");
}
}

void print_text() {
int i,j;
printf("text => \n");
for(i=0;i<4;i++) {
for(j=0;j<4;j++)
printf("%hx ",text[i][j]);
printf("\n");
}
}

unsigned short int sbox[16][16] = {
           0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,⏎
 0xab, 0x76,
           0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4,⏎
 0x72, 0xc0,
           0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8,⏎
 0x31, 0x15,
           0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27,⏎
 0xb2, 0x75,
           0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,⏎
 0x2f, 0x84,
           0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,⏎
 0x58, 0xcf,
           0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,⏎
 0x9f, 0xa8,
           0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,⏎
 0xf3, 0xd2,
           0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d,⏎
 0x19, 0x73,
           0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e,⏎
 0x0b, 0xdb,
           0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95,⏎
 0xe4, 0x79,
           0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a,⏎
 0xae, 0x08,
           0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd,⏎
 0x8b, 0x8a,
           0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1,⏎
 0x1d, 0x9e,
           0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,⏎
 0x28, 0xdf,
           0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,⏎
 0xbb, 0x16
           };
```

```c
void sub_bytes() {


int i,j,ri,ci;

i=j=ri=ci=0;

for(i=0;i<4;i++)
for(j=0;j<4;j++) {
ri = text[i][j] >> 4;
ci = text[i][j] & 0x0F;

//printf("ri =%x, ci = %x\n",ri,ci);
text[i][j] = sbox[ri][ci];
}
}

void shift_row() {
int i,j;
int t,count;

for (i=1;i<4;i++)
for(count=0;count<i;count++) {
t=text[i][0];
for(j=0;j<3;j++)
text[i][j]=text[i][j+1];
text[i][j]=t;
}
}


void mix_column() {

int MixCol[4][4] = {
            0x02, 0x03, 0x01, 0x01,
            0x01, 0x02, 0x03, 0x01,
            0x01, 0x01, 0x02, 0x03,
            0x03, 0x01, 0x01, 0x02
        };


int i,j,k;
unsigned short int a[4],b[4],h;

for (j = 0; j < 4; j++) {
for (i = 0; i < 4; i++) {
a[i]=text[i][j];
h=text[i][j] & 0x0080;
b[i]=(text[i][j] << 1) & 0x000000ff;
if(h == 0x80)
    b[i]^=0x1b;
}
//printf("\na=%hx %hx %hx %hx\tb=%hx %hx %hx %hx\n",a[0],a[1],a[2],a[3],b[0],b[1],b[2],b[3]);


text[0][j] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1]; /* 2 * a0 + a3 + a2 + 3 * a1 */
text[1][j] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2]; /* 2 * a1 + a0 + a3 + 3 * a2 */
text[2][j] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3]; /* 2 * a2 + a1 + a0 + 3 * a3 */
text[3][j] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0]; /* 2 * a3 + a2 + a1 + 3 * a0 */

}
}

void add_roundkey() {

int i,j;

for(i=0;i<4;i++)
for(j=0;j<4;j++)
text[i][j]^=key[i][j];
}


void key_schedule(int count) {

unsigned short int Rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36};
int i,j,ri,ci;
unsigned short int t,a[4];

for (j=3,i=0;i<4;i++)
a[i] = key[i][j];

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);

/* rotate column */
t=a[0];
for (i=0;i<3;i++)
a[i]=a[i+1];

a[i]=t;

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);

/* sub_bytes */
for(i=0;i<4;i++){
ri = a[i] >> 4;
ci = a[i] & 0x0F;
a[i] = sbox[ri][ci];
}

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);
```

```
/*1st xor */
for(j=0,i=0;i<4;i++) {
if(i==0)
key[i][j] = Rcon[count] ^ key[i][j] ^ a[i];
else
key[i][j] = key[i][j] ^ a[i];
}

//printf("key = %hx %hx %hx %hx\n",key[0][0],key[1][0],key[2][0],key[3][0]);

/*last xor */
for(j=1;j<4;j++)
for(i=0;i<4;i++)
key[i][j]^=key[i][j-1];


}

void encrypt_block() {
int count;
add_roundkey();
for (count = 0 ; count < 9 ; count++) {
key_schedule(count);
//print_key();

sub_bytes();
//print_text();
shift_row();
//print_text();
mix_column();
//print_text();
add_roundkey();
//print_text();

#if VERBOSE
printf("\n******************Round %d*************************\n",count+1);
print_key();
print_text();
#endif

}

key_schedule(count);
sub_bytes();
shift_row();
add_roundkey();

#if VERBOSE

printf("\n******************Round %d*************************\n",count+1);
print_key();
print_text();
#endif
```

```
}

void write_cipher(FILE *f) {

int i,j;
void *t;

for(i=0;i<4;i++)
for(j=0;j<4;j++) {
t = &text[i][j];
fwrite(t,1,1,f);
}
}

int main(int argc, char *argv[1]) {


FILE *fk = fopen("key.txt","r");
FILE *ft = fopen(argv[1],"r");
char destname[strlen(argv[1])+4];
sprintf(destname,"%s.%s",argv[1],"enc");
FILE *fw = fopen(destname,"w");

int sz=1, count=0;

int filesize,fcount = 0;

fseek(ft,0L,SEEK_END);
filesize = ftell(ft);
rewind(ft);


while(fcount < filesize) {
read_key(fk);
sz = read_text(ft);

#if VERBOSE
print_key();
print_text();
#endif

encrypt_block();

write_cipher(fw);

rewind(fk);
count++;
fcount+=16;
}

fclose(ft);
fclose(fk);
fclose(fw);
```

```
}
```

```
//AES decrypter
#include <stdio.h>

#define VERBOSE 0

unsigned short int key[4][4];
unsigned short int text[4][4];
unsigned short int roundkey[11][4][4];


void read_key (FILE *f) {

unsigned short int c=0x00000000;
void *t = &c;
int sz;
int i,j;
int cn;
i=j=0;


for (cn=0;cn<16;cn++) {

sz = fread(t,1,1,f);
key[i][j++] = c;
c=0x00000000;

if (j>=4) {i++;j=0;}
if(sz==0 || i >= 4) break;
}
}

int read_text (FILE *f) {

unsigned short int c=0x00000000;
void *t = &c;
int sz;
int i,j;
int cn;

i=j=0;

for (cn=0;cn<16;cn++) {

sz = fread(t,1,1,f);
text[i][j++] = sz?c:0x00;
c=0x00000000;
}

return feof(f);
}


void print_key() {
```

```
int i,j;
printf("key => \n");
for(i=0;i<4;i++) {
for(j=0;j<4;j++)
printf("%hx ",key[i][j]);
printf("\n");
}
}

void print_rkey(int count) {
int i,j;
printf("key => %d\n",count);
for(i=0;i<4;i++) {
for(j=0;j<4;j++)
printf("%hx ",roundkey[count][i][j]);
printf("\n");
}
}

void print_text() {
int i,j;
printf("text => \n");
for(i=0;i<4;i++) {
for(j=0;j<4;j++)
printf("%hx ",text[i][j]);
printf("\n");
}
}

unsigned short int sbox[16][16] = {
            0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,↵
 0xab, 0x76,
            0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4,↵
 0x72, 0xc0,
            0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8,↵
 0x31, 0x15,
            0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27,↵
 0xb2, 0x75,
            0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,↵
 0x2f, 0x84,
            0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,↵
 0x58, 0xcf,
            0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,↵
 0x9f, 0xa8,
            0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,↵
 0xf3, 0xd2,
            0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d,↵
 0x19, 0x73,
            0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e,↵
 0x0b, 0xdb,
            0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95,↵
 0xe4, 0x79,
            0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a,↵
 0xae, 0x08,
            0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd,⏎
 0x8b, 0x8a,
            0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1,⏎
 0x1d, 0x9e,
            0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,⏎
 0x28, 0xdf,
            0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,⏎
 0xbb, 0x16
            };

unsigned short int inv_sbox[16][16] = {
            0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3,⏎
 0xd7, 0xfb,
            0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde,⏎
 0xe9, 0xcb,
            0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa,⏎
 0xc3, 0x4e,
            0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b,⏎
 0xd1, 0x25,
            0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65,⏎
 0xb6, 0x92,
            0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d,⏎
 0x9d, 0x84,
            0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3,⏎
 0x45, 0x06,
            0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13,⏎
 0x8a, 0x6b,
            0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4,⏎
 0xe6, 0x73,
            0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75,⏎
 0xdf, 0x6e,
            0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18,⏎
 0xbe, 0x1b,
            0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd,⏎
 0x5a, 0xf4,
            0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80,⏎
 0xec, 0x5f,
            0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9,⏎
 0x9c, 0xef,
            0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53,⏎
 0x99, 0x61,
            0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21,⏎
 0x0c, 0x7d
            };



void inv_sub_bytes() {


int i,j,ri,ci;
```

```
i=j=ri=ci=0;

for(i=0;i<4;i++)
for(j=0;j<4;j++) {
ri = text[i][j] >> 4;
ci = text[i][j] & 0x0F;

//printf("ri =%x, ci = %x\n",ri,ci);
text[i][j] = inv_sbox[ri][ci];
}
}

void inv_shift_row() {
int i,j;
int t,count;

for (i=1;i<4;i++)
for(count=0;count<i;count++) {
t=text[i][3];
for(j=3;j>0;j--)
text[i][j]=text[i][j-1];
text[i][j]=t;
}
}

// xtime is a macro that finds the product of {02} and the argument to xtime modulo {1b}
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

// Multiplty is a macro used to multiply numbers in the field GF(2^8)
#define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 & 1) * ⏎
xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) * ⏎
xtime(xtime(xtime(xtime(x))))))

void inv_mix_column() {
int i;
    unsigned short int a,b,c,d;
    for(i=0;i<4;i++)
    {

        a = text[0][i];
        b = text[1][i];
        c = text[2][i];
        d = text[3][i];


        text[0][i] = 0xFF & (Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ ⏎
Multiply(d, 0x09));
        text[1][i] = 0xFF & (Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ ⏎
Multiply(d, 0x0d));
        text[2][i] = 0xFF & (Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ ⏎
Multiply(d, 0x0b));
        text[3][i] = 0xFF & (Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ ⏎
Multiply(d, 0x0e));
```

```c
    }
}

void inv_add_roundkey(int count) {

int i,j;

for(i=0;i<4;i++)
for(j=0;j<4;j++)
text[i][j]^=roundkey[count][i][j];
}


void key_schedule(int count) {

unsigned short int Rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36};
int i,j,ri,ci;
unsigned short int t,a[4];

for (j=3,i=0;i<4;i++)
a[i] = roundkey[count-1][i][j];

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);

/* rotate column */
t=a[0];
for (i=0;i<3;i++)
a[i]=a[i+1];

a[i]=t;

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);

/* sub_bytes */
for(i=0;i<4;i++){
ri = a[i] >> 4;
ci = a[i] & 0x0F;
a[i] = sbox[ri][ci];
}

//printf("a = %hx %hx %hx %hx\n",a[0],a[1],a[2],a[3]);

/*1st xor */
for(j=0,i=0;i<4;i++) {
if(i==0)
roundkey[count][i][j] = Rcon[count-1] ^ roundkey[count-1][i][j] ^ a[i];
else
roundkey[count][i][j] = roundkey[count-1][i][j] ^ a[i];
}

//printf("key = %hx %hx %hx %hx\n",roundkey[count][0][0],roundkey[count][1][0],roundkey[count][2][0],roundkey[count][3][0]);

/*last xor */
```

```c
for(j=1;j<4;j++)
for(i=0;i<4;i++)
roundkey[count][i][j]=roundkey[count][i][j-1] ^ roundkey[count-1][i][j];
}

void keyexpand() {
int i,j;
int count = 0;

for(i=0;i<4;i++)
for(j=0;j<4;j++)
roundkey[count][i][j]=key[i][j];
//print_rkey(count);
//printf("\n");
for(count=1;count<11;count++) {
key_schedule(count);
//print_rkey(count);
//printf("\n");
}
}

void decrypt_block() {
int count = 10;
keyexpand();
inv_add_roundkey(count);
#if VERBOSE
printf("\n*****************Round %d************************\n",count);
print_rkey(count);
print_text();
#endif

for (count = 9; count > 0 ; count--) {
//key_schedule(count);
//print_key();


inv_shift_row();
//print_text();
inv_sub_bytes();
//print_text();


//print_text();
inv_add_roundkey(count);
//print_text();
inv_mix_column();

#if VERBOSE
printf("\n*****************Round %d************************\n",count);
print_rkey(count);
print_text();
#endif
}
```

```c
inv_shift_row();
inv_sub_bytes();
inv_add_roundkey(count);

#if VERBOSE

printf("\n******************Round %d*************************\n",count);
print_rkey(count);
print_text();
#endif
}

void write_cipher(FILE *f) {

int i,j;
void *t;

for(i=0;i<4;i++)
for(j=0;j<4;j++) {
t = &text[i][j];
fwrite(t,1,1,f);
}
}

int main(int argc, char *argv[1]) {


FILE *fk = fopen("key.txt","r");
FILE *ft = fopen(argv[1],"r");
char destname[strlen(argv[1])+5];
sprintf(destname,"%s.%s",argv[1],"text");
FILE *fw = fopen(destname,"w");

int sz=1, count=0;

int filesize,fcount = 0;

fseek(ft,0L,SEEK_END);
filesize = ftell(ft);
rewind(ft);


#if VERBOSE
printf("filesize = %d\n",filesize);
printf("Step1 ...%d\n",feof(ft));
#endif



while(fcount < filesize) {
read_key(fk);
sz = read_text(ft);
#if VERBOSE
print_key();
print_text();
#endif

decrypt_block();

write_cipher(fw);

rewind(fk);
count++;
fcount+=16;
}


fclose(ft);
fclose(fk);
fclose(fw);
}
```