/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Lab 9 Nios Software

Dong Kai Wang, Fall 2017

Christine Chen, Fall 2013

For use with ECE 385 Experiment 9

University of Illinois ECE Department

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

#include "aes.h"

// Pointer to base address of AES module, make sure it matches Qsys

volatile unsigned int \* AES\_PTR = (unsigned int \*) 0x00002000;

// Execution mode: 0 for testing, 1 for benchmarking

int run\_mode = 0;

/\*\* charToHex

\* Convert a single character to the 4-bit value it represents.

\*

\* Input: a character c (e.g. 'A')

\* Output: converted 4-bit value (e.g. 0xA)

\*/

char charToHex(char c)

{

char hex = c;

if (hex >= '0' && hex <= '9')

hex -= '0';

else if (hex >= 'A' && hex <= 'F')

{

hex -= 'A';

hex += 10;

}

else if (hex >= 'a' && hex <= 'f')

{

hex -= 'a';

hex += 10;

}

return hex;

}

/\*\* charsToHex

\* Convert two characters to byte value it represents.

\* Inputs must be 0-9, A-F, or a-f.

\*

\* Input: two characters c1 and c2 (e.g. 'A' and '7')

\* Output: converted byte value (e.g. 0xA7)

\*/

char charsToHex(char c1, char c2)

{

char hex1 = charToHex(c1);

char hex2 = charToHex(c2);

return (hex1 << 4) + hex2;

}

/\*\* encrypt

\* Perform AES encryption in software.

\*

\* Input: msg\_ascii - Pointer to 32x 8-bit char array that contains the input message in ASCII format

\* key\_ascii - Pointer to 32x 8-bit char array that contains the input key in ASCII format

\* Output: msg\_enc - Pointer to 4x 32-bit int array that contains the encrypted message

\* key - Pointer to 4x 32-bit int array that contains the input key

\*/

void AddRoundKey (unsigned char \*state, unsigned char \*keyschedule )

{

int i;

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

state [i] = state[i] ^ keyschedule [i]; // Since each char takes 8 bits (1 Byte), loop 16 times will cover all 128 bits in State and first row in keyschedule

}

}

void SubBytes (unsigned char \*state)

{

int i;

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

state [i] = aes\_sbox[(uint)state[i]];

}

}

void ShiftRows (unsigned char \*state)

{

unsigned char temp, temp2;

temp = state [1];

state[1] = state[5];

state[5] = state[9];

state[9] = state[13];

state[13] = temp; //Switch the second row by brutal force

temp = state[2];

temp2 = state[6];

state[2] = state[10];

state[6] = state[14];

state[10] = temp;

state[14] = temp2; //Switch the third row

temp = state[3];

state[3] = state[15];

state[15] = state[11];

state[11] = state[7];

state[7] = temp;

}

void MixColumns (unsigned char \*state)

{

uchar temp[16];

int i;

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

temp[i] = state[i];

}

for(i = 0; i < 4; i++){

state[4\*i] = gf\_mul[temp[i\*4]][0] ^ gf\_mul[temp[i\*4+1]][1] ^ temp[i\*4+2] ^ temp[i\*4+3];

state[4\*i+1] = temp[i\*4] ^ gf\_mul[temp[i\*4+1]][0] ^ gf\_mul[temp[i\*4+2]][1] ^ temp[i\*4+3];

state[4\*i+2] = temp[i\*4] ^ temp[i\*4+1] ^ gf\_mul[temp[i\*4+2]][0] ^ gf\_mul[temp[i\*4+3]][1];

state[4\*i+3] = gf\_mul[temp[i\*4]][1] ^ temp[i\*4+1] ^ temp[i\*4+2] ^ gf\_mul[temp[i\*4+3]][0];

}

}

uint Subword (uint word)

{

unsigned char \*bytes = (unsigned char\*)&word;

int i;

for (i = 0; i < 4; ++i) {

bytes[i] = aes\_sbox[bytes[i]];

}

return word;

}

void KeyExpansion (unsigned char \*key, uint \*keyschedule)

{

int i, j, k; //i stands for the column number and j stands for row number

uint temp;

for (i = 0; i < 4; i++) {

keyschedule[i] = ((uint\*)key)[i];

}

for (i = 4; i < 44; i++) { //starting from the second coloumn

temp = keyschedule[i-1];

if (i%4 ==0){

temp = (temp >> 8) | (temp << 24);

temp = Subword(temp);

temp = temp ^ (Rcon[i/4] >> 24); // Rcon: big endian -> little endian

}

keyschedule[i] = keyschedule[i-4] ^ temp;

}

}

void encrypt(unsigned char \* msg\_ascii, unsigned char \* key\_ascii, unsigned int \* msg\_enc, unsigned int \* key)

{

unsigned char state[16];

unsigned char key\_schedule[176];

unsigned char key\_state[16];

int i;

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

state[i] = (unsigned char)charsToHex((char)msg\_ascii[2\*i],(char)msg\_ascii[(2\*i)+1]);

key\_state[i] = (unsigned char)charsToHex((char)key\_ascii[2\*i],(char)key\_ascii[(2\*i)+1]);

}

KeyExpansion(key\_state, (uint\*)key\_schedule);

AddRoundKey(state, key\_schedule);

for (i = 1; i < 10; i++) {

SubBytes((unsigned char \*)state);

ShiftRows(state);

MixColumns(state);

AddRoundKey(state, key\_schedule+16\*i);

}

SubBytes(state);

ShiftRows(state);

AddRoundKey(state, key\_schedule+160);

for (i = 0; i < 4; i++) {

msg\_enc[i] = (state[4\*i] << 24) + (state[4\*i + 1] << 16) + (state[4\*i + 2] << 8) + state[4\*i+3];

key[i] = (int)(key\_state[i\*4]<<24)+(int)(key\_state[i\*4+1]<<16)+ (int)(key\_state[i\*4+2]<<8)+(int)(key\_state[i\*4+3]);

}

}

/\*\* decrypt

\* Perform AES decryption in hardware.

\*

\* Input: msg\_enc - Pointer to 4x 32-bit int array that contains the encrypted message

\* key - Pointer to 4x 32-bit int array that contains the input key

\* Output: msg\_dec - Pointer to 4x 32-bit int array that contains the decrypted message

\*/

void decrypt(unsigned int \* msg\_enc, unsigned int \* msg\_dec, unsigned int \* key)

{

// Implement this function

}

/\*\* main

\* Allows the user to enter the message, key, and select execution mode

\*

\*/

int main()

{

// Input Message and Key as 32x 8-bit ASCII Characters ([33] is for NULL terminator)

unsigned char msg\_ascii[33];

unsigned char key\_ascii[33];

// Key, Encrypted Message, and Decrypted Message in 4x 32-bit Format to facilitate Read/Write to Hardware

unsigned int key[4];

unsigned int msg\_enc[4];

unsigned int msg\_dec[4];

printf("Select execution mode: 0 for testing, 1 for benchmarking: ");

scanf("%d", &run\_mode);

if (run\_mode == 0) {

// Continuously Perform Encryption and Decryption

while (1) {

int i = 0;

//sprintf(msg\_ascii, "%s", "ece298dcece298dcece298dcece298dc");

//sprintf(key\_ascii, "%s", "000102030405060708090a0b0c0d0e0f");

printf("\nEnter Message:\n");

scanf("%s", msg\_ascii);

printf("\n");

printf("\nEnter Key:\n");

scanf("%s", key\_ascii);

printf("\n");

encrypt(msg\_ascii, key\_ascii, msg\_enc, key);

AES\_PTR[0] = key[0];

AES\_PTR[1] = key[1];

AES\_PTR[2] = key[2];

AES\_PTR[3] = key[3];

AES\_PTR[4] = msg\_enc[0];

AES\_PTR[5] = msg\_enc[1];

AES\_PTR[6] = msg\_enc[2];

AES\_PTR[7] = msg\_enc[3];

printf("\nEncrpted message is: \n");

for(i = 0; i < 4; i++){

printf("%08x", msg\_enc[i]);

}

printf("\n");

decrypt(msg\_enc, msg\_dec, key);

printf("\nDecrypted message is: \n");

for(i = 0; i < 4; i++){

printf("%08x", msg\_dec[i]);

}

printf("\n");

}

}

else {

// Run the Benchmark

int i = 0;

int size\_KB = 2;

// Choose a random Plaintext and Key

for (i = 0; i < 32; i++) {

msg\_ascii[i] = 'a';

key\_ascii[i] = 'b';

}

// Run Encryption

clock\_t begin = clock();

for (i = 0; i < size\_KB \* 64; i++)

encrypt(msg\_ascii, key\_ascii, msg\_enc, key);

clock\_t end = clock();

double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

double speed = size\_KB / time\_spent;

printf("Software Encryption Speed: %f KB/s \n", speed);

// Run Decryption

begin = clock();

for (i = 0; i < size\_KB \* 64; i++)

decrypt(msg\_enc, msg\_dec, key);

end = clock();

time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

speed = size\_KB / time\_spent;

printf("Hardware Encryption Speed: %f KB/s \n", speed);

}

return 0;

}

// unsigned char a[4];

// unsigned char b[16];

// int i, j;

// for ( i = 0; i < 4; i++) {

// a[0] = state[i];

// a[1] = state[i+4];

// a[2] = state[i+8];

// a[3] = state[i+12];

// b[i] = (gf\_mul[a[0]][0]) ^ (gf\_mul[a[1]][1]) ^ a[2] ^ a[3];

// b[i+4] = a[0] ^ (gf\_mul[a[1]][0]) ^ (gf\_mul[a[2]][1]) ^ a[3];

// b[i+8] = a[0] ^ a[1] ^ (gf\_mul[a[2]][0]) ^ (gf\_mul[a[3]][1]);

// b[i+12] = (gf\_mul[a[0]][1]) ^ a[1] ^ a[2] ^ (gf\_mul[a[3]][0]);

// }

// for (j = 0; j < 16; j++) {

// state [j] = b[j];

// }

gcc -Wall filename.c -o objectname