CSEE 4840

128-bit AES decryption

Shrivathsa Bhargav

Larry Chen

Abhinandan Majumdar

Shiva Ramudit

CSEE 4840 – Embedded System Design

Spring 2008, Columbia University

Table of Contents

ABSTRACT 3
1. INTRODUCTION 3
2. HARDWARE DESIGN 3
  2.0 Hardware overview 3
  2.1 AES decrypto 4
    2.1.1 Algorithm 4
    2.1.2 Optimized Hardware Design 6
    2.1.3 Timing 6
  2.2 SD-card SPI interface 7
    2.2.1. MMC/SD Card Pin Assignments in SPI Mode 7
    2.2.2. SPI Commands 7
    2.2.3. SPI Clock Control 8
    2.2.4. Mode Selection 8
    2.2.5. Initialization Sequence 8
    2.2.6. Data Read 8
    2.2.7. Implementation 8
  2.3 VGA and SRAM controller 9
    2.3.1 VGA implementation 9
    2.3.2 SRAM implementation 10
  2.4 LCD Display and Keyboard 10
  2.5 Resource consumption 10
3. SOFTWARE DESIGN 10
4. RESULTS 11
5. TASK DIVISION 11
6. LESSONS LEARNED 11
7. ADVICE FOR FUTURE STUDENTS 11
8. ACKNOWLEDGMENTS 11
9. REFERENCES 11
ABOUT THE AUTHORS 11
APPENDICES

FPGA-based 128-bit AES decryption

Shrivathsa Bhargav, Larry Chen, Abhinandan Majumdar, Shiva Ramudit

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

ABSTRACT

The original objective of the AES project was to create an AES decryption system for images. The end result has exceeded the original objective and the AES group designed and implemented an FPGA-based high-speed 128-bit AES decryption system for 6 fps "video" comprised of sequential images. The images are pre-encrypted, and are read as .BMP files from an SD-card.

1. INTRODUCTION

The Advanced Encryption Standard (AES, also known as Rijndael) [1] is well-known block-cipher algorithm for portability and reasonable security. The nature of encryption lends itself very well to the hardware capabilities of FPGAs. The goal of this project is to create a reasonably fast AES decryption implementation. The data being fed into the decrypter is a sequence of pre-encrypted 8-bit grayscale Windows Bitmap images.

This report is structured as follows: First, the hardware design is presented which details all the modules (decryption, SD-card interface, and the VGA and SRAM controller) including timing descriptions where appropriate, followed by the software implementation.

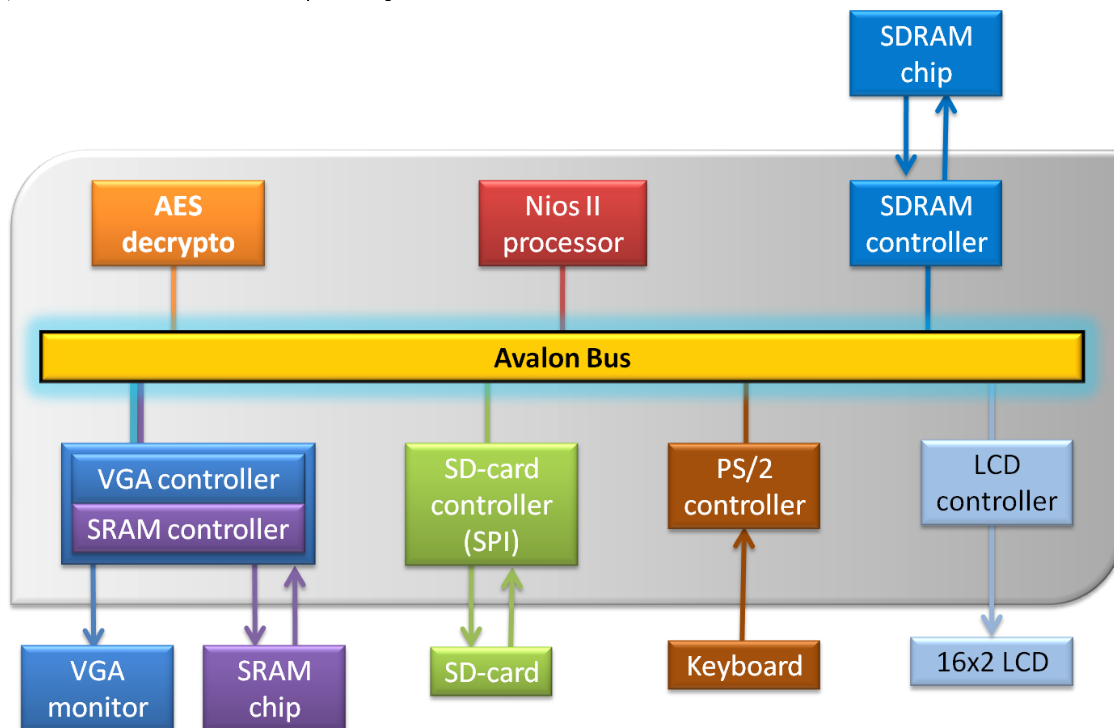


Figure 1 - The block diagram of the entire system. The gray boundary represents the blocks within the FPGA. Arrows show data flow; lines show hard connections between modules.

2. HARDWARE DESIGN

2.0 Hardware overview

There are two main peripherals: the VGA monitor and the SD-card reader. These are controlled by the VGA controller and the SPI controller, respectively. A VGA controller is needed to maintain the frame-buffer, and to provide display 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 other (minor) peripherals are the keyboard (to allow the user to enter a 32 hex-digit decryption key) and the 16x2 character LCD-display that displays the key as the user enters it, and allows the user to check the key before encryption begins. The Nios-II processor (hereafter Nios) uses the SD-RAM as its operating memory. 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 user-entered 128-bit key. The results of this process are stored in the SRAM.
2. **SD-card SPI interface**. This is needed to read and buffer raw and encrypted image data from the MMC/SD-card. It uses the SPI protocol, which is a serial communication protocol.
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), which is used to house the decrypted data, and will act as a frame buffer for the VGA controller.
4. **Keyboard and LCD module**. The user enters the 32 hex-digit passphrase with the keyboard, and can verify it on the 16x2 character display before beginning the decryption process.
5. **Nios**. Nios will read the key entered by the user, supervise the whole operation (which will be sequential, and act as the conduit for data traveling between various blocks).

The hardware blocks are described in detail below, while the high-level code for keyboard and LCD are described in the software section, along with the Nios implementation.

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 results in 128 bits of decrypted 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. The basic modules constituting AES Decryption are explained in excruciating detail below:

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).

The RTL for key expansion module is below. The generate roundkey module performs the algorithm that generates a single round key. Its input is multiplexed between the user inputted key and the last round's key. The output is stored in a register to be used as input during the next iteration of the algorithm. The expansion keys module is a RAM which stores the original key and the 10 rounds of generated keys for use during the decryption algorithm.

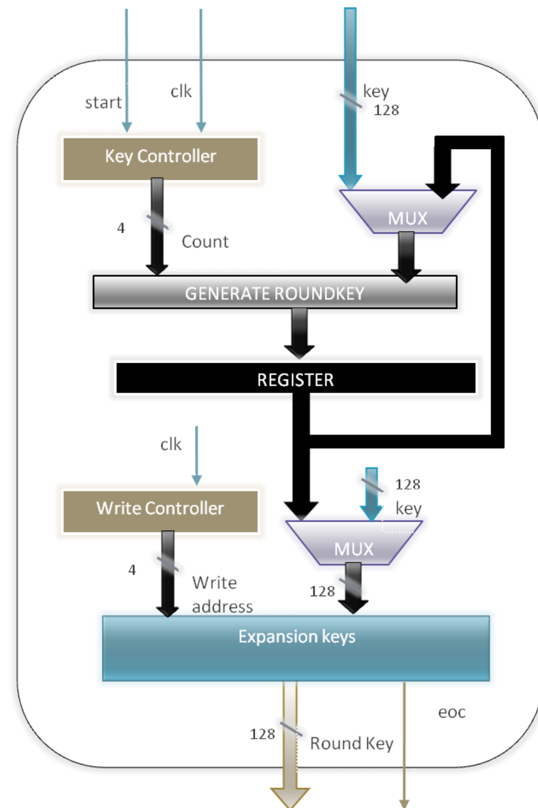


Figure 2 - AES key expansion

a) Key Expansion - The algorithm for generating the 10 rounds of the round key is as follows: The 4th column of the i-1 key is rotated such that each element is moved up one row.

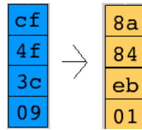
2b	28	ab	09
7e	ae	f7	cf
15	d2	15	4f
16	a6	88	3c

→

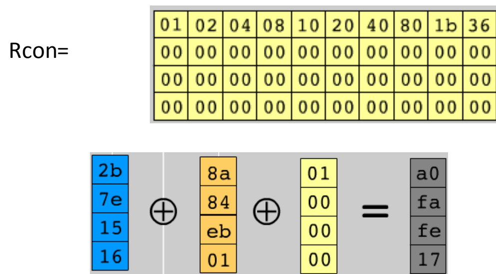
cf
4f
3c
09

It then puts this result through a forwards Sub Box algorithm which replaces each 8 bits of the matrix

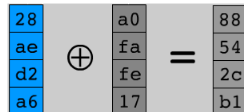
with a corresponding 8-bit value from S-Box. (See figure for Inverse Sub Byte below)



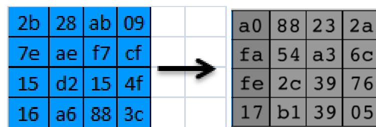
To generate the first column of the ith key, this result is XOR-ed with the first column of the $i\text{-}1^{th}$ key as well as a constant (Row constant or Rcon) which is dependent on i.



The second column is generated by XOR-ing the 1st column of the ith key with the second column of the $i\text{-}1^{th}$ key.

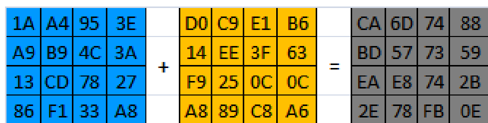


This continues iteratively for the other two columns in order to generate the entire i^{th} key.

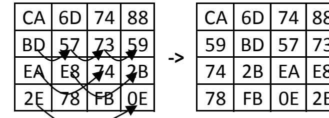


Additionally this entire process continues iteratively for generating all 10 keys. As a final note, all of these keys are stored statically once they have been computed initially as the ith key generated is required for the $(10\text{-}i)^{th}$ round of decryption.

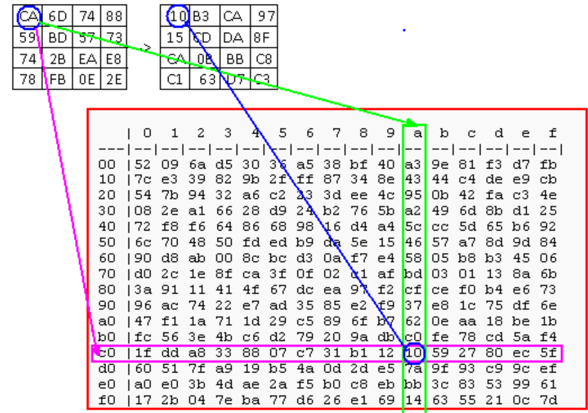
b) Inverse Add Round Key – Performs XOR operation between the cipher text and intermediate expanded key corresponding to that particular iteration. E.g., if the diagrams on the left represent the cipher and the key values, the final value after it has generated by this step is shown on the right.



c) Inverse Shift Row – This step rotates each ith row by i elements right wise, as shown in the figure.



d) Inverse Sub Bytes – This step replaces each entry in the matrix from the corresponding entry in the inverse S-Box[2] as shown in figure.



e) Inverse Mix Column - The Inverse MixColumns[3] operation performed by the Rijndael cipher, along with the shift-rows step, is the primary source of all the 10 rounds of diffusion in Rijndael. Each column is treated as a polynomial over Galois Field (2^8) and is then multiplied modulo $x^4 + 1$ with a fixed inverse polynomial is $c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$. The Multiplication is done as shown below.

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

As shown in the block level diagram below, the AES decrypto initially performs key-expansion on the 128-bit key block. Then the round key signals the start of the actual decryption process once the data process is ready. It starts by executing an inverse add round key between cipher text with the modified key (generated in the last iteration of the encryption process) from key expansion. After this step, the AES decrypto repeats the inverse shift row, inverse sub, inverse add round key, and inverse mix column steps nine times. At the last iteration, it does an inverse shift row, inverse sub bytes and inverse add round key to generate the original data.

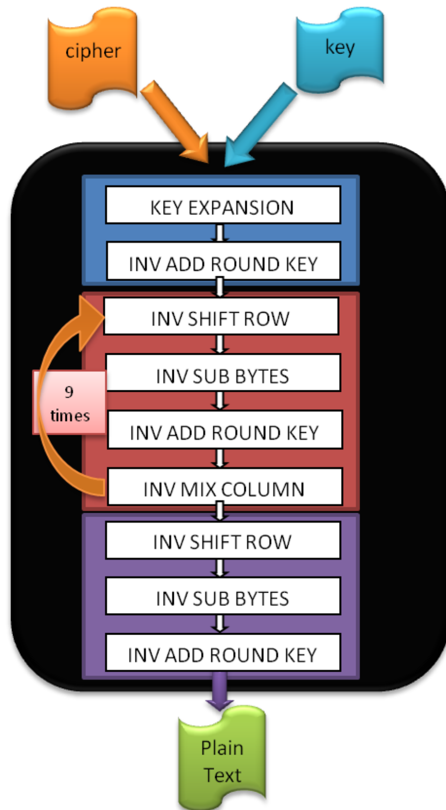


Figure 3 - AES 128-bit Decryption Algorithm

2.1.2 Optimized Hardware Design

Considering that the SD-card is the main source of latency in reading the block, the design was optimized at four levels.

a) Elimination of inverse shift row by swapping the respective lines before sending it to inverse sub bytes.
b) Optimization of inverse mix columns to remove multiplication operations by turning them into shift operation / comparison operations
c) Elimination of duplicate modules to save FPGA resources.
d) Sharing of 32-bit input line both for accepting key and cipher text.

Since Nios has a 32-bit MM Master Port (and therefore can transmit up to 32 bits of data at a time), we buffered the 32-bit data into the 128-bit bus one by one, before we actually proceed with decryption. The 32-bit data line is used as a common bus to accept both the key and the cipher text. Initially, the key used for encryption is being sent to the key expansion module to generate and store all intermediate key-values required for corresponding iteration into the key-table. Then, the same 32-bit bus as used to send the input cipher text, and uses the intermediate keys stored in key table to perform its decryption. The eoc (end of computation) signal both from key expansion and AES Decrypto is multiplexed into the final eoc indicating which corresponding unit (key expansion or AES Decrypto) is done with its computation.

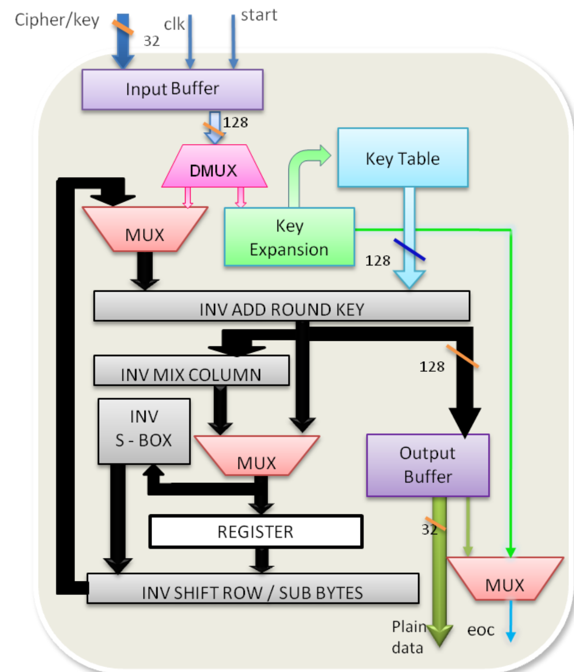


Figure 4 - AES 128-bit Decrypto Datapath

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 dependent 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, the plain text is generated after 10 rounds of decryption, where it is sent to Nios through the Avalon bus in 32-bit chunks.

2.1.3 Timing

Since we are using the Avalon bus to transfer 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 data-path) 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 shown in figures 5 and 6. The overall AES block can run at 88.31 MHz.

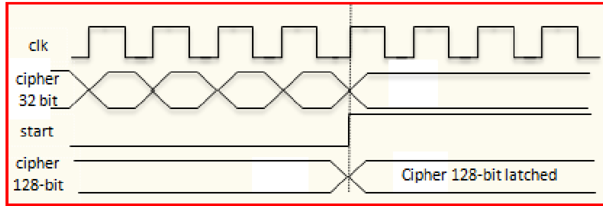


Figure 5 - Timing of Input Data Buffering

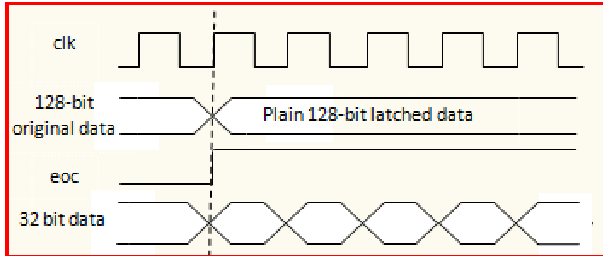


Figure 6 - Timing of final data traversal

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 raw image data will be read from the card, buffered into 512-byte blocks and stored in the SRAM via Nios.
Prof. Edwards has built a simple SPI-controller module for use in his Apple II demonstration. [5]

To facilitate communication with the SD card via the SPI interface, we refer to engineering application notes [6] 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. 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.

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

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

2.2.2. SPI Commands

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

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.

Table 2 - SPI Commands

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.

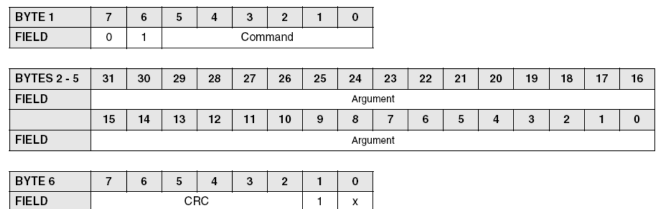


Figure 7 - Command Transmission

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 to 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 8 - Data Token Start Byte and Data Error Token Structure

2.2.3. SPI Clock Control

The SPI bus clock signal can be used by the SPI host to set the cards to energy saving mode or to control data flow (to avoid under-run or over-run conditions) on the bus. The host is allowed to change the clock frequency or stop it altogether. There are a few restrictions the SPI host must follow:

a) The bus frequency can be changed at any time, but only up to the maximum data transfer frequency, defined by the MultiMediaCards.

b) It is an obvious requirement that the clock must be running for the MultiMediaCard to output data or response tokens. After the last SPI bus transaction, the host is required to provide 8 clock cycles for the card to complete the operation before shutting down the clock. During this 8-clock period, the state of the CS signal is irrelevant. It can be asserted or de-asserted.

2.2.4. 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).

2.2.5. 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.

2.2.6. 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.

2.2.7. Implementation

Given that Professor Edwards already have a working implementation of a hardware-based SPI controller, we attempted to understand his implementation and modify it to fit our project requirements. In fact, upon reviewing his code, it was determined that SPI initialization sequence as detailed in section 2.2.5 is executed in his implementation accordingly with minor changes on the number of clock cycles in between steps of the operation. Furthermore, it follows the stated protocol closely to establish block length configuration and data communication with the SD card. Therefore, overall, Professor Edwards' code implemented the SD card protocol discussed above closely and eased our implementation efforts. However, it is important to note that several changes were still needed before it could function per our project's requirements.

Firstly, Professor Edwards' code is implemented completely in hardware, and it did not interface with any

software components. However, as mentioned in section 2.0, the AES project uses NIOS as a conduit of data transfers and the SPI interface must interact with it. Therefore, to enable this interaction, the Avalon Bus slave interface is appended to the SPI interface, and the NIOS uses a start bit to ask the SPI interface to request data from the SD card and monitors an eor (end or read) bit to sense when the previously issued data read request has completed. Also, the Avalon Bus interface is also used for NIOS to fetch data from the SPI interface buffer.

Secondly, upon trying different SD cards, it was discovered that the existing SD card implementation does not work with the types of SD cards the group has available. Fortunately, the Professor indicated a patch that would fix this issue. The patch involves sending extra pulses to the SD card before initialization. However, while this patch alone made it possible to read a single block of data, it was not sufficient to read consecutive blocks. In fact, it was also necessary to increase the number of wait clock cycles from 8 to 16 between block reads to successfully read consecutive blocks of data. This was one of the more difficult issues to debug since it deviates from the protocol as shown in section 2.2.3.

Thirdly, another modification needed was to use a 512-byte block length and correspondingly sized buffer in the SPI interface. During the original design phase, the SPI interface was envisioned to read data in 32-bit blocks and transfer that to the processor before reading another block. This is described in section 2.2 of the design document. However, after the design implementation was partially completed, it was soon realized that this imposed a significant bottleneck to the system. This is because after a read data request is sent to the SD card, it takes a finite amount of time before the SD processes that message and responds with data of length specified by the previously issued set block length command. Therefore, if that finite amount of time is termed T, and 512 bytes of data were requested, the original scheme will require 64 T (512 byte / 32 bits), whereas a 512-byte block length will only require 1 T in addition to the time needed to transfer the individual data bits per clock cycle. Therefore, it was concluded that using a larger block length can significantly reduce the amount of time needed to read data from the SD card. This hypothesis was confirmed through experimentation, where a compiled design with 32-bit block length and a compiled design with 512-byte block length were both run to measure the difference in time and the results confirmed the previously stated theory. Therefore, in order to avoid slowing down the entire system, a 512-byte block length and buffer was introduced to alleviate the bottleneck.

Fourthly, and finally, accompanied by the previous hardware modification is also a software change to our SPI interface for it to work properly. The change is necessitated because while our SPI interface reads data in 512-byte blocks, a single frame is 77888 bytes (320*240=76800-bytes image data + 1078-bytes header data + 10-bytes zero padding), which is not divisible by 512-byte blocks. In other words, after each frame is read, the SPI controller has already buffered partial data for the next frame, or spilled into the next frame. The diagram below illustrates this concept. Each block in the diagram is 512*152 bytes = 77824 bytes, which is the largest 512-byte multiple needed for a single frame.

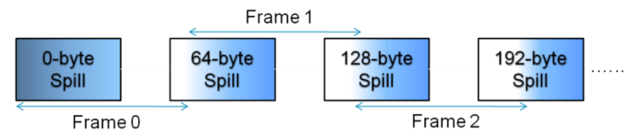


Figure 9 - SPI buffer spilling

To workaround this issue, we calculate the buffer spill in software to ensure that we do not read duplicate data. Given that a frame is 77888 bytes, and the largest 512-byte multiple in a frame is 77824, an individual spill is 64-bytes. In fact, as the figure above implies, the spill will be multiples of 64-bytes, and it will take 512-byte/64-byte = 8 spills to go back to a 0-byte spill block. Therefore, given that it is only the first 512-byte block of each frame that will be spilled, we implement a check in software to monitor the start block of each frame and offset it by 64*(frame % 8) to read the correct data contents. After several experimentations, it was proven that this spill calculation technique is functional and the SPI interfaces reads frames of data correctly.

2.3 VGA and SRAM controller

2.3.1 VGA implementation

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

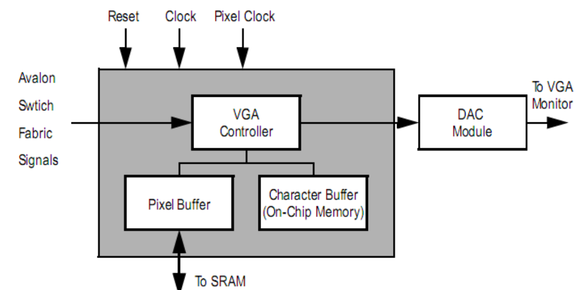


Figure 10 - Block diagram of the VGA controller

One of the major concerns prior to the implementation was 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 provided the basis for the VGA

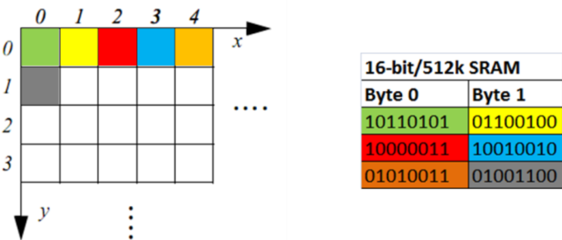
controller. Some of the details of the controller are discussed below.

- 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.
- The maximum resolution of the VGA controller is 640*480 in its “pixel mode”. The image size is chosen as 320*240, and only occupies a quarter of the screen (hence, this resolution is known as QVGA).
- While pixel mode supports 30-bit color, our design forces the RGB values to a single value (thereby enforcing grayscale).

The data for the current pixel is stored in the SRAM, and is fetched using the address calculation shown below.

$$address = \begin{cases} x * \dfrac{(edge_{bottom} - edge_{top} + 1)}{2} + y + header_{Bitmap} & \textbf{when inside rectangle} \\ 0 & \textbf{otherwise} \end{cases}$$

The address calculation is illustrated below.



16-bit/512k SRAM	
Byte 0	Byte 1
10110101	01100100
10000011	10010010
01010011	01001100

Figure 11 - VGA pixel addressing and SRAM retrieval

Since the SRAM is 16 bits wide, the VGA reads in two pixels at a time, and a small state-machine toggles between the pixels during display.

2.3.2 SRAM implementation

The fact that the SRAM is single-ported is a significant hurdle, since this means that data cannot be written into it when the VGA needs to read from it, which is once every two clock cycles (the VGA runs at 25 MHz, which is half the CPU clock frequency of 50 MHz). The synchronization issues that will result from attempting to toggle the SRAM data path between the VGA and Nios every clock cycle are rather difficult to handle correctly, and so a compromise was reached: the SRAM simply cuts off the VGA when Nios requests to write into it. In this state, the VGA senses ‘Z’s on the SRAM data path, which makes it hold the last pixel value that it *did* manage to read from the SRAM.

2.4 LCD Display and Keyboard

For the LCD display and keyboard interfacing, we used the provided HDL and SDK .C files from Altera (as part of their University IP core program [8]) with their associated commands to read in the entered key and display it on the LCD display.

2.5 Resource consumption

A table with our used resources is shown below.

Module	LC Combinationals	LC registers	Logic Elements	Memory bits
AES decrypto	5328	830	6158	2048
CPU	4684	3096	7780	174976
SPI	3751	4308	8059	0
Others	1654	633	514	1024
Total	15417	8867	22511	178048
Percentage	46%	26%	53%	37%

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
- Display message to LCD display and request User Input for decryption key
- User inputs the 128-bit key via the keyboard which is sent to the AES decrypto once entry is complete
- 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) **only when outside the image frame**
- When decryption is complete, notify the VGA/SRAM controller
- Wait for the reset button, then restart the whole process

This repeated polling of the VGA controller to figure out if the VGA is currently drawing inside the image frame causes a significant slowdown in the frame rate (from around 8.5 fps to about 6 fps).

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.

For the keyboard, state-machines were written as needed to handle the IBM keyboard scan codes [9]. Only the keys corresponding to the hex values A-F, 0-9 were utilized as well as the Backspace and Enter keys. All other keys are ignored by the software. The LCD functions were accessed during the keyboard functions to auto-update the LCD as the decryption key was entered by the user.

4. RESULTS

The AES block was written from scratch, and is pretty fast, taking just 10 cycles for decrypting 128-bits of data. The SPI controller, while being clocked at 50 MHz, ends up being slowed down by the SD-card, which is only fast during sustained reads.

We were able to get frames decrypted, decoded and displayed at around 6 frames per second (at 320*240, 8-bit grayscale). This translates to around 3.74 Mbps for the entire system.

5. TASK DIVISION

The project was modularized right at the planning stage, and the different tasks were spread thusly:

- Shrivathsa Bhargav - VGA & SRAM controller
- Larry Chen – SPI controller
- Abhinandan Majumdar – AES decrypto (design)
- Shiva Ramudit – AES decrypto (key expansion)

6. LESSONS LEARNED

The success of this project is materialized through our early start and modular division of relevant tasks. Our early start gave us time to consider alternative design choices and evaluate different options carefully before executing them. In fact, it also gave us various opportunities to improve upon our existing solution and implementation after asking for advice from the TA and the professor. Additionally, our modular division of relevant tasks made it possible for all team members to work independently without depending on one another. Furthermore, since we clearly defined the communication interface between modules, it was also relatively straightforward to integrate the various components. Finally, it also made debugging significantly more manageable since we already know which modules work and which had problems.

7. ADVICE FOR FUTURE STUDENTS

There are several pieces of advice that might be beneficial to future students. We shall now dole out said advice in a condescending tone. Firstly, it is very important to start early for the project. Given a project of reasonable workload and difficulty, it will take a significant amount of time to figure out the project requirements and the important design decisions. Therefore, it is imperative to allocate sufficient time such that the project members can work through these issues carefully and effectively. Secondly, it may be helpful if the group is broken down to subgroups instead of individuals. While this style of organization allows parallel and concurrent progress, it also ensures that a group member can ask another for assistance instead of having to going it alone. Thirdly, during the process of hardware implementation, it is useful to thoroughly simulate a module before attempting to deploy it on the FPGA. In doing so, the project members ensure that a behavior model of the element is established and verified. Finally, if possible, a project member should avoid reinventing the wheel when it comes to development. If there is an existing software or hardware module elsewhere that is well established and documented, it will save a lot of time to adopt (and modify if needed) this implementation to fit a given project requirement. Otherwise, it may be too time-consuming to redesign and re-implement all the modules from scratch.

8. ACKNOWLEDGMENTS

We would like to thank Prof. Edwards and TA David Lariviere for pushing us to do more than we would have been content to settle with. We would also like to thank the cleaning staff at Columbia for keeping the lab habitable; putting so many engineers into a single room for extended periods of time is never a good idea.

9. REFERENCES

[1] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[2] http://en.wikipedia.org/wiki/Rijndael_S-box
[3] http://en.wikipedia.org/wiki/Rijndael_mix_columns
[4] 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
[5] Apple II demo by Prof. Edwards http://www1.cs.columbia.edu/~sedwards/apple2fpga/
[6] Interfacing a MultiMediaCard to the LH79520 System-On-Chip http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/lh79520.mmc.interfacing.pdf
[7] Embedded Systems Lab CSEE 4840 : Imagic Design Document http://www1.cs.columbia.edu/~sedwards/classes/2007/4840/designs/Imagic.pdf
[8] Altera University IP cores http://university.altera.com/materials/unv-ip-cores.html
[9] IBM Keyboard scan codes http://www.computer-engineering.org/ps2keyboard/scancodes2.html

ABOUT THE AUTHORS

Shrivathsa Bhargav graduated from the SUNY, Stony Brook with a Bachelors' Degree in Electrical Engineering (Cum Laude). He is currently interested in digital VLSI circuit design and embedded systems.

Larry Chen graduated from the University of Waterloo, Canada with a Bachelors' Degree in Computer Engineering. His interests are embedded systems and circuits.

Abhinandan Majumdar graduated from National Institute of Technology, Surathkal, India with a Bachelors' Degree in Computer Engineering. His current interests are in embedded systems design, and digital VLSI circuit design.

Shiva Ramudit graduated from Columbia University with a Bachelors' Degree in Computer Engineering. He is currently a VP of Information Security Compliance at Citi Corp., New York. His interests include embedded systems design and information security.

APPENDIX

Table of Contents

Software - C code

- projectv3.c
- Keyboard
    - keyboard.c
    - keyboard.h

Hardware – VHDL

- toplevel.vhd
- AES
    - aes128_nios.vhd
    - AES_decrypto.vhd
    - controller.vhd
    - demux1_2.vhd
    - expansion_keys.vhd
    - generate_roundkey.vhd
    - inv_addroundkey.vhd
    - inv_mixcolumns.vhd
    - inv_mtimes.vhd
    - inv_multiply.vhd
    - inv_multiply_row.vhd
    - inv_sbox.vhd
    - inv_shiftrow_subbytes.vhd
    - key_controller.vhd
    - mux128_1.vhd
    - regis128.vhd
    - sbox.vhd
    - write_controller.vhd
- SPI
    - spi_controller.vhd
- VGA/SRAM
    - vga_sram_supercontroller.vhd
    - de2_vga_raster.vhd
    - de2_sram_controller.vhd

```
/*******************************************************************************/
/*
Main Software File for 128-bit Decryption Project
Written for 128-bit AES decryption project
Course: CSEE 4840 - Embedded System Design, Spring 2008
Authors:        Shrivathsa Bhargav (sb2784)
                Larry Chen (lc2454)
                Abhinandan Majumdar (am2993)
                Shiva Ramudit (syr9)

Last modified: 5-8-2008
*/
/*******************************************************************************/

#include <io.h>
#include <system.h>
#include <stdio.h>
#include <alt_types.h>
#include <math.h>
#include "alt_up_character_lcd.h"
#include "keyboard.h"

//Macros to interface to the AES block
#define AES_WRITE_KEY(offset,data)   IOWR_32DIRECT(AES128_BASE,(offset*4),data)
#define AES_READ_KEY(offset)         IORD_32DIRECT(AES128_BASE,(offset*4))
#define AES_WRITE_DATA(addr,data)    IOWR_32DIRECT(AES128_BASE,16+(addr*4),data)
#define AES_READ_DATA(addr)          IORD_32DIRECT(AES128_BASE,16+(addr*4))

//Macros to write into VGA
#define VGA_GO()                     IOWR_16DIRECT(VGASRAM_BASE,524288*2,1)       //Tell VGA that it is safe
to read data from SRAM
#define VGA_NO()                     IOWR_16DIRECT(VGASRAM_BASE,524288*2,0)       //Tell VGA to not read
data
#define VGA_BUSY()                   IORD_16DIRECT(VGASRAM_BASE,524288*4)         //Are we inside the
rectangle?

//Macros to get data from SPI
#define SPI_CHOOSE_CARD_ADDR(addr)   IOWR_32DIRECT(SPI_BASE, 2*4, addr)
#define SPI_START()                  IOWR_32DIRECT(SPI_BASE, 1*4, 1)
#define SPI_END()                    IOWR_32DIRECT(SPI_BASE, 1*4, 0)
#define SPI_READY()                  IORD_32DIRECT(SPI_BASE, 16*4)
#define SPI_CHOOSE_BUFF_ADDR(addr)   IOWR_32DIRECT(SPI_BASE, 64*4, addr)
#define SPI_GET_BUFF_DATA()          IORD_32DIRECT(SPI_BASE, 128*4)

//Macros to write into SRAM
#define SRAM_WRITE(addr, data)       IOWR_16DIRECT(VGASRAM_BASE,(addr*2),data)
#define SRAM_READ(addr)              IORD_16DIRECT(VGASRAM_BASE,(addr*2))

#define NUM_FRAME                    84  //short = 84 //long = 3270

int main()
{
    printf("\nWelcome to the AES 128-bit decryption project!");
```

```
    unsigned int data = 0;
    unsigned long int i = 0, addr=0;
    int j =0,count=0,k=0, m=0, limit=0;
    unsigned short *key;
    unsigned long int frame = 0, frameStartAddr=0, frameSize=0x13040;     //0x13038
    unsigned long int frameEndAddr=0, startBlock, endBlock;
    unsigned long int keydata[4]={0x00000000,0x00000000,0x00000000,0x00000000};
                                        //{0x2b28ab09,0x7eaef7cf, 0x15d2154f, 0x16a6883c}; //This is the passphrase!

    printf("\n\nPlease enter the decryption key (32 hex digits).\n");


    //Passphrase keyboard entry; If this section is
    key = getKey();
    for(i=0;i<32;i++)
        keydata[i>>3] = (keydata[i>>3]<<4)|(key[i]&0xF);


    // Write the passphrase to the AES module
    for(i=0;i<4;i++)
        AES_WRITE_KEY(i,keydata[i]);

    printf("\nStarting Decryption...");

    for(frame = 0; frame <= NUM_FRAME; frame=frame+1)
    {
        VGA_GO();
        addr = 0;       //Reset the SRAM address
        frameStartAddr = frame*frameSize;
        frameEndAddr = frame*frameSize + frameSize;

        // Calculate the address to the nearest 512-byte multiple
        startBlock = abs(frameStartAddr/512)*512;
        endBlock = abs(frameEndAddr/512 )*512;

        for(i = startBlock; i <= endBlock; i = i+512)
        {
            // Calculate the buffer spill
            m = (i==startBlock)?(frame%8):0;

            // If there is no spill, then the data is no presently in the buffer, and
            // a new SD card read is issued
            if (m == 0)
            {
                SPI_CHOOSE_CARD_ADDR(i);
                SPI_START();
                SPI_END();
            }

            // Poll the SPI ready bit until it is done. This signals the buffer has been
            // filled
            while(!SPI_READY());
```

```
            // Calculate the ending address of the buffer. We want to avoid writing
            // images belonging to the next frame into the SRAM
            limit = (i==endBlock)?(64*((frame+1)%8)):512;

            // Since the spill will be multiple of 64-bytes, multiple it from the previous
            // calculation
            k = m*64;

            // Loop through the contents of the SPI buffer and feed it to the AES descrypto,
            // and then write it to the SRAM
            for (;k<limit; k=k+4)
            {
                // Read data from SPI buffer
                SPI_CHOOSE_BUFF_ADDR(k);
                data = SPI_GET_BUFF_DATA();
                // Feed data to AES decrypto
                AES_WRITE_DATA(count++,data);

                if(k%8 == 0)
                  while(!VGA_BUSY());

                // When we have 4 32-bits, or 128-bits, start the decryption process and
                // feed the result into SRAM
                if(count==4)
                {
                    for (j=0;j<4;j++)
                    {
                        // Start decryption
                        data = AES_READ_DATA(j);
                        // Ask VGA to not retrieve data from the SRAM
                        VGA_NO();
                        // Write 2x16 bits of data into the SRAM
                        SRAM_WRITE(addr++,(data&0xFFFF0000)>>16);
                        SRAM_WRITE(addr++,(data&0x0000FFFF));
                        // Give the VGA thumbs up to read from the SRAM
                        VGA_GO();
                    }
                    count=0;
                }
            }
        }
        // Restart playback when the last frame is reached
        if(frame >= NUM_FRAME)
          frame = 0;
    }
    printf("\nEncryption Completed.");
    return 0;
}
```

```c
/*******************************************************************************/
/*
Keyboard Header File for 128-bit AES decryption project
Written for 128-bit AES decryption project
Course: CSEE 4840 - Embedded System Design, Spring 2008
Authors:        Shrivathsa Bhargav (sb2784)
                Larry Chen (lc2454)
                Abhinandan Majumdar (am2993)
                Shiva Ramudit (syr9)

Last modified: 5-8-2008
*/
/*******************************************************************************/
#include <io.h>
#include <system.h>
#include <stdio.h>
#include <alt_types.h>

//Macros to read from the PS/2 keyboard
#define KEYBOARD_READY()            IORD_8DIRECT(KEYBOARD_BASE, 0)     //Poll status of keyboard
#define KEYBOARD_READ()             IORD_8DIRECT(KEYBOARD_BASE, 4)     //Get one byte from the keyboard

// State Machine for Getting Relevant Make Code
extern unsigned char checkCode(unsigned char makecode);

// Function for writing a character to the key board
extern void writeToLCD(unsigned char letter, int position);

// Remove character from LCD
extern void backSpace(int position);

// Main Function for running storing and writing the keys to the LCD
extern unsigned short *getKey();
```

```
/******************************************************************************/
/*
Keyboard Software for 128-bit AES Decryption Project
Written for 128-bit AES decryption project
Course: CSEE 4840 - Embedded System Design, Spring 2008
Authors:        Shrivathsa Bhargav (sb2784)
                Larry Chen (lc2454)
                Abhinandan Majumdar (am2993)
                Shiva Ramudit (syr9)

Last modified: 5-8-2008
*/
/******************************************************************************/
#include "keyboard.h"
#include <io.h>
#include <system.h>
#include <stdio.h>
#include <alt_types.h>

// Function for key entry (keys are based on makecodes and only 0-9,A-F,Backspace,
// and Enter are valid
unsigned char checkKey(unsigned char makecode)
{
    unsigned char result = 0;
    int notDone = 0;

    // printf("Make Code %X : /n", makecode);

    while (notDone == 0)
    {

        makecode = KEYBOARD_READ();

        while (makecode == 0xF0)
        {
            notDone = 1;
            makecode = KEYBOARD_READ();

        }

    }

    switch(makecode)
    {

        // A
        case 0x1C :
            result = 65;
            break;
        // B
        case 0x32 :
            result = 66;
            break;
```

```
        //C
        case 0x21 :
            result = 67;
            break;
        //D
        case 0x23 :
            result = 68;
            break;
        //E
        case 0x24 :
            result = 69;
            break;
        //F
        case 0x2B :
            result = 70;
            break;
        //0
        case 0x45 :
            result = 48;
            break;
        //1
        case 0x16 :
            result = 49;
            break;
        //2
        case 0x1E :
            result = 50;
            break;
        //3
        case 0x26 :
            result = 51;
            break;
        //4
        case 0x25 :
            result = 52;
            break;
        //5
        case 0x2E :
            result = 53;
            break;
        //6
        case 0x36 :
            result = 54;
            break;
        //7
        case 0x3D :
            result = 55;
            break;
        //8
        case 0x3E :
            result = 56;
            break;
        //9
```

```
        case 0x46 :
            result = 57;
            break;
        // Enter
        case 0x5A :
            result = 1;
            break;
        // Backspace
        case 0x66 :
            result = 2;
            break;


        default :
            break;

    }

    return result;

}

// Function for writing a character to the key board
void writeToLCD(unsigned char letter, int position)
{

    int x = position;
    int y = 1;
    unsigned char temp[1];

    temp[0] = letter;

    if (position >= 16)
    {
        x = position - 16;
        y = 2;
    }


    alt_up_character_lcd_write(temp,1);

    if (position == 15)
    {
        alt_up_character_lcd_set_cursor_pos(0,2);
    }

}

void backSpace(int position)
{

    int x = position;
    int y = 1;
```

```
    unsigned char temp[1];

    temp[0] = 32;

    if (position >= 16)
    {
        x = position - 16;
        y = 2;
    }

    alt_up_character_lcd_set_cursor_pos(x,y);
    //alt_up_character_lcd_write(temp,1);


    if (position == 16)
    {
        alt_up_character_lcd_set_cursor_pos(15,1);
    }
    else
    {
        alt_up_character_lcd_set_cursor_pos(x-1,y);
    }
    alt_up_character_lcd_write(temp,1);
    if (position == 16)
    {
        alt_up_character_lcd_set_cursor_pos(15,1);
    }
    else
    {
        alt_up_character_lcd_set_cursor_pos(x-1,y);
    }
    alt_up_character_lcd_set_cursor_pos(x-1,y);


}

unsigned short *getKey()
{

  unsigned char makecode = 0;
  unsigned char letter = 0;
  int q = 0;
  unsigned char aes_key[33];
  unsigned short *hex_key = (unsigned short *) malloc(sizeof(unsigned short));


  alt_up_character_lcd_init();
  alt_up_character_lcd_set_cursor_pos(0,1);


  for( q=0; q<33; q++)
  {
```

```
        while(!KEYBOARD_READY());
        //letter = translate_make_code();  //Get one byte (character)
        makecode = KEYBOARD_READ();

        // printf(" %X \n",makecode);

        letter = checkKey(makecode);

        // Delay for keyboard

        //Check if the letter entered is within bounds (0-9,a-f) or if it is BCKSPACE or ENTER
        if( letter >= 48 && letter <= 57 && q < 32)
        {
            //printf("%c %d\n",letter, q);
            aes_key[q] = letter;
            writeToLCD(letter,q);

            hex_key[q] = letter - 48;
        }
        else if( letter >= 65 && letter <= 70 && q < 32)      // A-F letter
        {
            //printf("%c %d\n",letter, q);
            aes_key[q] = letter;
            writeToLCD(letter,q);

            hex_key[q] = letter - 55;
        }
        // If Backspace go back a character
        else if( letter == 2 && q > 0)
        {
            //printf("Backspace\n");
            backSpace(q);
            q=q-2;
        }
        // If Enter and last character return string
        else if( letter == 1 && q == 32)
        {
            //printf("Enter\n");
            aes_key[q] = 0;
        }
        else        //It's neither!
            q--;

}

alt_up_character_lcd_init();
alt_up_character_lcd_set_cursor_pos(0,1);
alt_up_character_lcd_write("Key Entered.",12);
alt_up_character_lcd_set_cursor_pos(0,2);
alt_up_character_lcd_write("Processing...",13);
alt_up_character_lcd_set_cursor_pos(13,2);
```

```
  printf("\nDECRYPTION KEY ENTERED - %s",aes_key);
  printf("\nThank you. Beginning decryption.");

  return hex_key;

}
```

```
/*****************************************************************************/
/*
Software(C) Implementation of AES Encryption
Written for 128-bit AES decryption project
Course: CSEE 4840 - Embedded System Design, Spring 2008
Authors:        Shrivathsa Bhargav (sb2784)
                Larry Chen (lc2454)
                Abhinandan Majumdar (am2993)
                Shiva Ramudit (syr9)

Last modified: 5-8-2008
*/
/*****************************************************************************/

#include <stdio.h>

//Used for Debug Prints
//0 -> No Debug Prints
//1 -> With Debug Prints
#define VERBOSE 0


//Variable Declaration for key and text
unsigned short int key[4][4];
unsigned short int text[4][4];

//Function to read key file *Should be key.txt*
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;
    }
}

//Function to read plain text file *Supplied as first argument*
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);
}

//Function to display the key
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");
    }
}

//Function to display the plain text
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");
    }
}

//S-Box Declaration
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
            };

//Function to do Sub_bytes
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;
            text[i][j] = sbox[ri][ci];
        }
}

//Function to do Shift_row
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;
        }
}

//Function to do Mix_Column
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;
        }


        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 */

    }
}

//Function to do Add_roundkey
void add_roundkey() {
    int i,j;
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
            text[i][j]^=key[i][j];
}

//Function to do Key_schedule or Key_expansion
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];

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

    a[i]=t;

    /* sub_bytes */
    for(i=0;i<4;i++){
        ri = a[i] >> 4;
```

```
        ci = a[i] & 0x0F;
        a[i] = sbox[ri][ci];
    }

    /*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];
    }

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

//Function to do Encryption of 128bit plain text
void encrypt_block() {
    int count;
    add_roundkey();

    for (count = 0 ; count < 9 ; count++) {
        key_schedule(count);
        sub_bytes();
        shift_row();
        mix_column();
        add_roundkey();

        #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
}

//Function to write the generated cipher text (encrypted data) into a file (with extension.enc)
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);
        }
}

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

    FILE *fk = fopen("key.txt","r");      //File pointer for Key file. Always key.txt
    FILE *ft = fopen(argv[1],"r");        //File pointer for source file. Supplied as first argument
    int sz=1, count=0;
    int filesize,fcount = 0;
    char destname[strlen(argv[1])+4];
    sprintf(destname,"%s.%s",argv[1],"enc");
    FILE *fw = fopen(destname,"w");       //File pointer for encrypted/cipher file. Stored as .enc file

    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);
}
```

```
/******************************************************************************/
/*
Software(C) Implementation of AES Decryption
Written for 128-bit AES decryption project
Course: CSEE 4840 - Embedded System Design, Spring 2008
Authors:        Shrivathsa Bhargav (sb2784)
                Larry Chen (lc2454)
                Abhinandan Majumdar (am2993)
                Shiva Ramudit (syr9)

Last modified: 5-8-2008
*/
/******************************************************************************/

#include <stdio.h>

//Used for Debug Prints
//0 -> No Debug Prints
//1 -> With Debug Prints
#define VERBOSE 0

//Variable Declaration for key, text and roundkey
unsigned short int key[4][4];
unsigned short int text[4][4];

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

//Function to read key file *Should be key.txt*
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;
    }
}

//Function to read cipher text file *Supplied as first argument* *Can be recognized by .enc extension*
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);
}

//Function to display the key
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");
    }
}

//Function to display the roundkey
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");
    }
}

//Function to display the cipher text
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");
    }
}

//Declaration for sbox. Used for Key Expansion
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
            };

//Declaration for inverse sbox. Used for Decryption
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
            };

//Function for inverse_sub_bytes
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;

            text[i][j] = inv_sbox[ri][ci];
        }
}



//Function for inverse_shift_row
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;
        }
}



//Function for inverse_mix_column

// 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));
    }
}



//Function for inverse_add_roundkey
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];
}



//Function for key_schedule (for one iteration)
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];

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

    a[i]=t;

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

    /*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];
    }

    /*last xor */
    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];
}

//Function for key_expansion
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];
            for(count=1;count<11;count++) {
                key_schedule(count);
            }
}

//Function for Decryption of 128bit block
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--) {
        inv_shift_row();
        inv_sub_bytes();
        inv_add_roundkey(count);
        inv_mix_column();

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

    inv_shift_row();
    inv_sub_bytes();
    inv_add_roundkey(count);

    #if VERBOSE
```

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

//Function to write the generated plain text (decrypted data) into a file (with extension .text)
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);
        }
}

//Main Function
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);
}
```

```vhdl
-------------------------------------------------------------------------------
--
-- DE2 top-level module for AES decryption project
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- DE2 top-level module for the Apple ][
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--
-------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity AES128_toplevel is

  port (
    -- Clocks

    CLOCK_27,                                         -- 27 MHz
    CLOCK_50,                                         -- 50 MHz
    EXT_CLOCK : in std_logic;                         -- External Clock

    -- Buttons and switches

    KEY : in std_logic_vector(3 downto 0);            -- Push buttons
    SW : in std_logic_vector(17 downto 0);            -- DPDT switches

    -- LED displays

    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
    : out unsigned(6 downto 0);
    signal LEDG : out std_logic_vector(8 downto 0);          -- Green LEDs
    LEDR : out std_logic_vector(17 downto 0);         -- Red LEDs

    -- RS-232 interface

    UART_TXD : out std_logic;                         -- UART transmitter
    UART_RXD : in std_logic;                          -- UART receiver
```

```
-- IRDA interface

-- IRDA_TXD : out std_logic;          -- IRDA Transmitter
IRDA_RXD : in std_logic;                          -- IRDA Receiver

-- SDRAM

DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
DRAM_LDQM,                                     -- Low-byte Data Mask
DRAM_UDQM,                                     -- High-byte Data Mask
DRAM_WE_N,                                     -- Write Enable
DRAM_CAS_N,                                    -- Column Address Strobe
DRAM_RAS_N,                                    -- Row Address Strobe
DRAM_CS_N,                                     -- Chip Select
DRAM_BA_0,                                     -- Bank Address 0
DRAM_BA_1,                                     -- Bank Address 0
DRAM_CLK,                                      -- Clock
DRAM_CKE : out std_logic;                      -- Clock Enable

-- FLASH

FL_DQ : inout std_logic_vector(7 downto 0);        -- Data bus
FL_ADDR : out std_logic_vector(21 downto 0);    -- Address bus
FL_WE_N,                                           -- Write Enable
FL_RST_N,                                          -- Reset
FL_OE_N,                                           -- Output Enable
FL_CE_N : out std_logic;                           -- Chip Enable

-- SRAM

SRAM_DQ : inout std_logic_vector(15 downto 0);         -- Data bus 16 Bits
SRAM_ADDR : out std_logic_vector(17 downto 0);         -- Address bus 18 Bits
SRAM_UB_N,                                     -- High-byte Data Mask
SRAM_LB_N,                                     -- Low-byte Data Mask
SRAM_WE_N,                                     -- Write Enable
SRAM_CE_N,                                     -- Chip Enable
SRAM_OE_N : out std_logic;                     -- Output Enable

-- USB controller

OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
OTG_CS_N,                                       -- Chip Select
OTG_RD_N,                                       -- Write
OTG_WR_N,                                       -- Read
OTG_RST_N,                                      -- Reset
OTG_FSPEED,                    -- USB Full Speed, 0 = Enable, Z = Disable
OTG_LSPEED : out std_logic;    -- USB Low Speed, 0 = Enable, Z = Disable
OTG_INT0,                                       -- Interrupt 0
OTG_INT1,                                       -- Interrupt 1
OTG_DREQ0,                                      -- DMA Request 0
```

```
    OTG_DREQ1 : in std_logic;                          -- DMA Request 1
    OTG_DACK0_N,                                       -- DMA Acknowledge 0
    OTG_DACK1_N : out std_logic;                       -- DMA Acknowledge 1

    -- 16 X 2 LCD Module

    LCD_ON,                         -- Power ON/OFF
    LCD_BLON,                       -- Back Light ON/OFF
    LCD_RW,                         -- Read/Write Select, 0 = Write, 1 = Read
    LCD_EN,                         -- Enable
    LCD_RS : out std_logic;         -- Command/Data Select, 0 = Command, 1 = Data
    LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

    -- SD card interface

    SD_DAT : in std_logic;          -- SD Card Data     SD pin 7 "DAT 0/DataOut"
    SD_DAT3 : out std_logic;        -- SD Card Data 3   SD pin 1 "DAT 3/nCS"
    SD_CMD : out std_logic;         -- SD Card Command  SD pin 2 "CMD/DataIn"
    SD_CLK : out std_logic;         -- SD Card Clock    SD pin 5 "CLK"

    -- USB JTAG link

    TDI,                            -- CPLD -> FPGA (data in)
    TCK,                            -- CPLD -> FPGA (clk)
    TCS : in std_logic;             -- CPLD -> FPGA (CS)
    TDO : out std_logic;            -- FPGA -> CPLD (data out)

    -- I2C bus

    I2C_SDAT : inout std_logic; -- I2C Data
    I2C_SCLK : out std_logic;       -- I2C Clock

    -- PS/2 port

    PS2_DAT,                        -- Data
    PS2_CLK : in std_logic;         -- Clock

    -- VGA output

    VGA_CLK,                                                -- Clock
    VGA_HS,                                                 -- H_SYNC
    VGA_VS,                                                 -- V_SYNC
    VGA_BLANK,                                              -- BLANK
    VGA_SYNC : out std_logic;                               -- SYNC
    VGA_R,                                                  -- Red[9:0]
    VGA_G,                                                  -- Green[9:0]
    VGA_B : out std_logic_vector(9 downto 0);               -- Blue[9:0]

    --  Ethernet Interface

    ENET_DATA : inout std_logic_vector(15 downto 0);        -- DATA bus 16Bits
    ENET_CMD,                 -- Command/Data Select, 0 = Command, 1 = Data
    ENET_CS_N,                                              -- Chip Select
```

```
    ENET_WR_N,                                               -- Write
    ENET_RD_N,                                               -- Read
    ENET_RST_N,                                              -- Reset
    ENET_CLK : out std_logic;                                -- Clock 25 MHz
    ENET_INT : in std_logic;                                 -- Interrupt

    -- Audio CODEC

    AUD_ADCLRCK : inout std_logic;                           -- ADC LR Clock
    AUD_ADCDAT : in std_logic;                               -- ADC Data
    AUD_DACLRCK : inout std_logic;                           -- DAC LR Clock
    AUD_DACDAT : out std_logic;                              -- DAC Data
    AUD_BCLK : inout std_logic;                              -- Bit-Stream Clock
    AUD_XCK : out std_logic;                                 -- Chip Clock

    -- Video Decoder

    TD_DATA : in std_logic_vector(7 downto 0);      -- Data bus 8 bits
    TD_HS,                                          -- H_SYNC
    TD_VS : in std_logic;                           -- V_SYNC
    TD_RESET : out std_logic;                       -- Reset

    -- General-purpose I/O

    GPIO_0,                                           -- GPIO Connection 0
    GPIO_1 : inout std_logic_vector(35 downto 0)      -- GPIO Connection 1
    );

end AES128_toplevel;

architecture AES128_toplevel_arch of AES128_toplevel is

  component CLK14MPLL is
    port (
      inclk0    : in std_logic;
      c0        : out std_logic);
  end component;

  component sdram_pll is
    port (
      inclk0    : in std_logic;
      c0        : out std_logic);
  end component;

  signal CLK_14M, CLK_2M, PRE_PHASE_ZERO : std_logic;
  signal IO_SELECT, DEVICE_SELECT : std_logic_vector(7 downto 0);
  signal ADDR : unsigned(15 downto 0);
  signal D, PD : unsigned(7 downto 0);

  signal ram_we : std_logic;
  signal VIDEO, HBL, VBL, LD194 : std_logic;
  signal COLOR_LINE : std_logic;
  signal COLOR_LINE_CONTROL : std_logic;
```

```
  signal GAMEPORT : std_logic_vector(7 downto 0);
  signal cpu_pc : unsigned(15 downto 0);

  signal K : unsigned(7 downto 0);
  signal read_key : std_logic;

  signal flash_clk : unsigned(22 downto 0);
  signal reset : std_logic;

  signal track : unsigned(5 downto 0);
  signal trackmsb : unsigned(3 downto 0);
  signal D1_ACTIVE, D2_ACTIVE : std_logic;
  signal track_addr : unsigned(13 downto 0);
  signal TRACK_RAM_ADDR : std_logic_vector(13 downto 0);
  signal tra : unsigned(15 downto 0);
  signal TRACK_RAM_DI : std_logic_vector(7 downto 0);
  signal TRACK_RAM_WE : std_logic;

  signal CS_N, MOSI, MISO, SCLK : std_logic;

  signal SLOW_CLK : std_logic;       -- Larry
  signal clk25 : std_logic := '0';   -- B

  signal temp : unsigned(3 downto 0) := "0000";
  signal temp2 : unsigned(3 downto 0) := "0000";
  signal temp3 : unsigned(3 downto 0) := "0000";
  signal debug : unsigned(7 downto 0) := "00000000";
  signal counter3 : unsigned (2 downto 0) := "000";
  signal counter2 : unsigned (2 downto 0) := "000";

  signal addr_input : std_logic_vector(31 downto 0);
  signal size_input : std_logic_vector(31 downto 0);
  signal start_input : std_logic := '0';

  type ram_type is array (7 downto 0) of unsigned(7 downto 0);
  signal RAM : ram_type := ( x"00",
 x"00",
 x"00",
 x"00",
 x"00",
 x"00",
 x"00",
 x"00");

begin

  reset <= SW(0);

  -- Clock divider for VGA
  process (CLOCK_50)
  begin
    if rising_edge(CLOCK_50) then
      clk25 <= not clk25;
```

```
    end if;
  end process;


  COLOR_LINE_CONTROL <= COLOR_LINE and SW(17);   -- Color or B&W mode

--Instantiate the sdram pll entity
neg_3ns : sdram_pll port map (CLOCK_50, DRAM_CLK);

nios : entity work.nios_system port map (

      -- the_sdram
    zs_addr_from_the_sdram => DRAM_ADDR,
    zs_ba_from_the_sdram(1) => DRAM_BA_1,
    zs_ba_from_the_sdram(0) => DRAM_BA_0,
    zs_cas_n_from_the_sdram => DRAM_CAS_N,
    zs_cke_from_the_sdram => DRAM_CKE,
    zs_cs_n_from_the_sdram => DRAM_CS_N,
    zs_dq_to_and_from_the_sdram => DRAM_DQ,
    zs_dqm_from_the_sdram(1) => DRAM_UDQM,
    zs_dqm_from_the_sdram(0) => DRAM_LDQM,
    zs_ras_n_from_the_sdram => DRAM_RAS_N,
    zs_we_n_from_the_sdram => DRAM_WE_N,

      -- VGA signals
    VGA_BLANK_from_the_vgasram  => VGA_BLANK,
    VGA_B_from_the_vgasram      => VGA_B,
    VGA_CLK_from_the_vgasram    => VGA_CLK,
    VGA_G_from_the_vgasram      => VGA_G,
    VGA_HS_from_the_vgasram     => VGA_HS,
    VGA_R_from_the_vgasram      => VGA_R,
    VGA_SYNC_from_the_vgasram   => VGA_SYNC,
    VGA_VS_from_the_vgasram     => VGA_VS,
    --go_to_the_vgasram => SW(0),

    -- the_lcd
     LCD_BLON_from_the_character_lcd => LCD_BLON,
     LCD_DATA_to_and_from_the_character_lcd => LCD_DATA,
     LCD_EN_from_the_character_lcd => LCD_EN,
     LCD_ON_from_the_character_lcd => LCD_ON,
     LCD_RS_from_the_character_lcd => LCD_RS,
     LCD_RW_from_the_character_lcd => LCD_RW,

    -- the_keyboard
    PS2_Clk_to_the_keyboard => PS2_CLK,
    PS2_Data_to_the_keyboard => PS2_DAT,


    SRAM_ADDR_from_the_vgasram => SRAM_ADDR,
    SRAM_CE_N_from_the_vgasram => SRAM_CE_N,
    SRAM_DQ_to_and_from_the_vgasram => SRAM_DQ,
    SRAM_LB_N_from_the_vgasram => SRAM_LB_N,
    SRAM_OE_N_from_the_vgasram => SRAM_OE_N,
```

```
    SRAM_UB_N_from_the_vgasram => SRAM_UB_N,
    SRAM_WE_N_from_the_vgasram => SRAM_WE_N,
    CS_N_from_the_spi => CS_N,
    MOSI_from_the_spi => MOSI,
    SCLK_from_the_spi => SCLK,
    MISO_to_the_spi => MISO,
    clk => CLOCK_50,
    reset_n => '1'
    );

  SD_DAT3 <= CS_N;
  SD_CMD  <= MOSI;
  MISO    <= SD_DAT;
  SD_CLK  <= SCLK;

  LEDG(0) <= SW(0);

end AES128_toplevel_arch;
```

```
-----------------------------------------------------------------------------
--
-- NIOS Wrapper for AES Decrypto Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--              Larry Chen (lc2454)
--              Abhinandan Majumdar (am2993)
--              Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--Wrapper Module for AES Decrypto Module with NIOS via Avalon Bus
entity aes128_nios is
port (
clk         : in std_logic;
reset   : in std_logic;
read        : in std_logic;
write       : in std_logic;
chipselect  : in std_logic;
address     : in std_logic_vector  (2 downto 0);  --Equivalent to enable lines. address(0) for demuxing Cipher-Text and
Key-Data. address(1) for enabling Key Expansion. address(2) for enabling AES Decrypto
readdata    : out std_logic_vector  (31 downto 0);
writedata   : in std_logic_vector  (31 downto 0)
);
end aes128_nios;

architecture aes128_nios_arch of aes128_nios is

component AES_decrypto is
port (
ci                      : in std_logic_vector(127 downto 0);
clk                         : in std_logic;
EN                      : in std_logic_vector(2 downto 0);
pl                      : out std_logic_vector(127 downto 0);
eoc                     : out std_logic
);
end component AES_decrypto;

signal ci : std_logic_vector(127 downto 0);
signal en : std_logic_vector(2 downto 0);
signal pl : std_logic_vector(127 downto 0);
signal eoc : std_logic;

begin


process (clk)
```

```
begin
if rising_edge(clk) then
    if reset = '1' then
       readdata <= (others => '0');
       en <= "000";
       ci <= (others => '0');
    else
        if chipselect = '1' then
            --Sends 128 bit data in 32 bit chunk.
            --After sending 4th 32 bit chunk, it starts AES Decrypton or Key Expansion based upon Address Line
            if write = '1' then
                if address(1 downto 0) = "00" then
                    ci(127 downto 96) <= writedata;
                    if address(2) = '0' then
                        en <= "000";
                    elsif address(2) = '1' then
                        en <= "100";
                    end if;
                elsif address(1 downto 0) = "01" then
                    ci(95 downto 64) <= writedata;
                    if address(2) = '0' then
                    en <= "000";
                    elsif address(2) = '1' then
                    en <= "100";
                    end if;
                elsif address(1 downto 0) = "10" then
                    ci(63 downto 32) <= writedata;
                    if address(2) = '0' then
                    en <= "000";
                    elsif address(2) = '1' then
                    en <= "100";
                    end if;
                elsif address(1 downto 0) = "11" then
                    ci(31 downto 0) <= writedata;
                    if address(2) = '0' then
                    en <= "001";
                    elsif address(2) = '1' then
                    en <= "110";
                    end if;
                end if;
            elsif read = '1' then
                --When eoc = '0' sends the original cipher text/key data being sent.
                if eoc = '0' then
                    if address(1 downto 0) = "00" then
                        readdata <= ci(127 downto 96);
                    elsif address(1 downto 0) = "01" then
                        readdata <= ci(95 downto 64);
                    elsif address(1 downto 0) = "10" then
                        readdata <= ci(63 downto 32);
                    elsif address(1 downto 0) = "11" then
                        readdata <= ci(31 downto 0);
                    else
                        readdata <= x"ABCD1234";
```

```
                    end if;
                --When eoc = '1', sends the plain text (after decryption) in 32 bit chunks based upon the address line.
                elsif eoc = '1' then
                    if address(1 downto 0) = "00" then
                        readdata <= pl(127 downto 96);
                    elsif address(1 downto 0) = "01" then
                        readdata <= pl(95 downto 64);
                    elsif address(1 downto 0) = "10" then
                        readdata <= pl(63 downto 32);
                    elsif address(1 downto 0) = "11" then
                        readdata <= pl(31 downto 0);
                    end if;
                end if;
            end if;
        else
            readdata <= (others => '1');
        end if;
    end if;
end if;
end process;

--AES Decrypto Central Module
aes128: AES_decrypto port map(
ci => ci,
clk => clk, en => en,
pl => pl,
eoc => eoc
);



end aes128_nios_arch;
```

```
-------------------------------------------------------------------------------
--
-- AES Decrypto Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--                  Larry Chen (lc2454)
--                  Abhinandan Majumdar (am2993)
--                  Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;


--AES Decrypto Central Module
entity AES_decrypto is
port (
    ci                      : in std_logic_vector(127 downto 0); --Input Cipher-Text/Key-Data
    clk                     : in std_logic; --Clock
    en                      : in std_logic_vector(2 downto 0); --Enable lines. en(0) for demuxing Cipher-Text and Key-Data. en(1) for enabling Key Expansion. en(2) for enabling AES Decrypto
    pl                      : out std_logic_vector(127 downto 0); --Plain Text
    eoc                     : out std_logic --End of Computation Signal

    );
end;


architecture AES_decrypto_arch of AES_decrypto is

component regis128 is
    port (
        i           : in std_logic_vector (127 downto 0);
        clk, en     : in std_logic;
        o           : out std_logic_vector (127 downto 0)
        );
end component regis128;

component mux128_1 is
    port (
        i0,i1 : in std_logic_vector (127 downto 0);
        s     : in std_logic;
        o : out std_logic_vector (127 downto 0)
        );
end component mux128_1;

component inv_mixcolumns is
    port (
        it : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
```

```
        );
end component inv_mixcolumns;

component controller is
port (
    clk, en : in std_logic;
    count   : out std_logic_vector(3 downto 0);
    e0,e1,e2,e3,eoc   : out std_logic
    );
end component controller;

component inv_addroundkey is
    port (
        it : in std_logic_vector (127 downto 0);
        key : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
        );
end component inv_addroundkey;

component inv_shiftrow_subbytes is
    port (
        it : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
        );
end component inv_shiftrow_subbytes;

component demux1_2 is
    port (
        i : in std_logic_vector (127 downto 0);
        s     : in std_logic;
        o1, o2 : out std_logic_vector (127 downto 0)
        );
end component demux1_2;

component key_controller is
port (
    clk, en : in std_logic;
    count   : out std_logic_vector(3 downto 0);
    e0,e1,e2,eoc   : out std_logic
    );
end component key_controller;

component generate_roundkey is
    port (
        count : in std_logic_vector(3 downto 0);
        it : in std_logic_vector(127 downto 0);
        roundkey : out std_logic_vector(127 downto 0)
        );
end component generate_roundkey;

component write_controller is
port (
    clk, en : in std_logic;
```

```
    addr : out std_logic_vector(3 downto 0)
    );
end component write_controller;

component expansion_keys is
    port (
        clk, en : in std_logic;
        writeaddr : in std_logic_vector (3 downto 0);
        addr : in std_logic_vector (3 downto 0);
        it : in std_logic_vector (127 downto 0);
        key : out std_logic_vector (127 downto 0);
        writeComplete : out std_logic
        );
end component expansion_keys;

-- Signals for AES Module
signal temp1, temp2, temp3, temp4, temp5, temp6, key_itrn : std_logic_vector (127 downto 0);

signal count        : std_logic_vector (3 downto 0);
signal e0, e1, e2, e3       : std_logic;

-- Signals for Key Module
signal cnt         : std_logic_vector (3 downto 0);
signal writeaddr   : std_logic_vector (3 downto 0);
signal tempk1, tempk2, tempk3, tempk4, tempk5, tempk6 : std_logic_vector (127 downto 0);
signal key, cyphertext : std_logic_vector (127 downto 0);
signal ek0, ek1, ek2, ek3   : std_logic;

signal eodec, eokey : std_logic;
signal endec, enkey : std_logic;

begin

-- Muxs for selecting enable and end of computation
enkey <= en(0);

endec <= en(1);

-- Eoc being multiplexed whether key-expansion or AES Decrypto is active
eoc <= eodec when en(1) = '1' else
       eokey when en(0) = '1' else
    '0';

--Controller for AES decrypto
CONTROL :  controller port map (
          clk => clk, en => endec,
          count => count, e0 => e0, e1 => e1, e2 => e2, e3 => e3, eoc => eodec);

--Mux1 for AES Decrypto
MUX1 :  mux128_1 port map(
        i0 => cyphertext, i1 => temp1,
        s => e0,
        o => temp2
```

```
        );

--Module for Inverse Add Round Key
INVADDRKEY :  inv_addroundkey port map(
        it => temp2,
        key => key_itrn,
        ot => temp3
        );

--Module for Inverse Mix Column
INVMIXCOL :  inv_mixcolumns port map(
        it => temp3,
        ot => temp4
);

-- Mux2 for AES Decrypto
MUX2 :  mux128_1 port map(
        i0 => temp3, i1 => temp4,
        s => e1,
        o => temp5
        );

-- Register1 for AES Decrypto. Stores the temporary iteration results
REG1 : regis128 port map(
        i => temp5,
        clk => clk, en => e2,
        o => temp6
        );

-- Module for Inverse Shiftrow Subbytes
INVSHIFTROW_SUBBYTES:  inv_shiftrow_subbytes port map(
        it => temp6,
        ot => temp1
        );

--Register3 for AES Decrypto. Latches the final result
REG3 : regis128 port map(
      i => temp3,
      clk => clk, en => e3,
     o => pl
     );

-- For selecting between input and key
DEMUX : demux1_2 port map(
        i => ci,
        s => en(2),
        o1 => key, o2 => cyphertext
);


--- Key Expansion Related Modules:
KEYCONTROL : key_controller port map (
          clk => clk, en => enkey,
```

```
          count => cnt, e0 => ek0, e1 => ek1, e2 => ek2, eoc => ek3
);

-- Mux to select between key and current for input into key generator
MUX3 : mux128_1 port map(
        i0 => key, i1 => tempk3,
        s => ek0,
        o => tempk1
);

-- Round Key Generator
KEYGENERATE : generate_roundkey port map(
        count => cnt,
        it => tempk1,
        roundkey => tempk2
);

-- Mux to select between key and round key for writing to the expansion keys
MUX4 : mux128_1 port map(
        i0 => key, i1 => tempk3,
        s => ek2,
        o => tempk4
);

--Register4 for Key Expansion
REG4 : regis128 port map(
        i => tempk2,
        clk => clk, en => ek1,
        o => tempk3
);

--Register4 for Key Expansion
REG5 : regis128 port map(
        i => tempk4,
        clk => clk, en => enkey,
        o => tempk5
);

--WriteController for Key Expansion for writing the key for a particular iteration
WRITE_CONTROLLER1 : write_controller port map(
        clk => clk, en => ek3,
        addr => writeaddr
);

--Key Exansion that generates keys for every iteration
EXP_KEYS :  expansion_keys port map(
        clk => clk,
        en => ek3,
        writeaddr => writeaddr,
        addr => count,
        it => tempk5,
        key => key_itrn,
        writeComplete => eokey
```

```
);

end AES_decrypto_arch;
```

```
--------------------------------------------------------------------------------
--
-- AES Decryption Controller
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--              Larry Chen (lc2454)
--              Abhinandan Majumdar (am2993)
--              Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
--------------------------------------------------------------------------------

library IEEE;

use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

--Controller for AES Decrypto
entity controller is
port (
    clk, en : in std_logic;
    count   : out std_logic_vector(3 downto 0);
    e0,e1,e2,e3,eoc   : out std_logic
    );
end;

architecture controller_arch of controller is
type state_type is (S0, S1, S2);
signal state : state_type := S0;
signal t_count : std_logic_vector(3 downto 0) := "0000";

begin

process (clk)
begin

if rising_edge(clk) then

    case state is

        --Start State
        when S0 =>
                    if en = '1' then
                        t_count <= t_count + '1';
                        state <= S1;
                    end if;
        --Checking for the loop Condition
        when S1 =>
                    if en = '1' then
                        t_count <= t_count + '1';
                        if t_count = "1010" then
```

```
                        t_count <= "0000";
                        state <= S2;
                        end if;
                    end if;
        --Intialize to start state if en = '0' after final execution
        when S2 =>
                    t_count <= "0000";
                    if en = '0' then
                        state <= S0;

                       end if;

    end case;


end if;

end process;

-- Iteration Count
count <= t_count;

--Setting the control bits required for AES Datapath
e0 <= '0' when state = S0 and en = '1' else
      '1' when state = S1 and en = '1' else
      '1' when state = S2 and en = '1' else
      '0';

e1 <= '0' when state = S0 and en = '1' else
      '1' when state = S1 and en = '1' else
      '1' when state = S2 and en = '1' else
      '1';

e2 <= '1' when state = S0 and en = '1' else
      '1' when state = S1 and en = '1' else
      '0';

e3 <= '1' when state = S1 and en = '1' and t_count = "1010" else
      '0';

--Setting the EOC(End of Computation) signal once AES Decryption is done.
eoc <= '1' when state = S2 and en = '1' else
       '0';

end controller_arch;
```

```
--------------------------------------------------------------------------------
--
-- 1-to-2 De-Multiplexer
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
--------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;

--Demultiplexer Module
entity demux1_2 is
    port (
        i : in std_logic_vector (127 downto 0);
        s      : in std_logic;
        o1, o2 : out std_logic_vector (127 downto 0)
        );
end;

architecture demux1_2_arch of demux1_2 is
begin
o1 <= i when s = '0' else
     (others => 'X');

o2 <= i when s = '1' else
     (others => 'X');

end demux1_2_arch;
```

```
-------------------------------------------------------------------------------
--
-- Expansion Keys Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--Central Module for Key Expansion --
--All round keys are written here during key generation and read during decryption
entity expansion_keys is
    port (
        clk, en : in std_logic;
        writeaddr : in std_logic_vector (3 downto 0);
        addr : in std_logic_vector (3 downto 0);
        it : in std_logic_vector (127 downto 0);
        key : out std_logic_vector (127 downto 0);
        writeComplete : out std_logic
        );
end;

architecture expansion_keys_arch of expansion_keys is
type key_table_row is array(0 to 3) of std_logic_vector(7 downto 0);
type key_table_type is array(0 to 3) of key_table_row;
type key_table_array is array(0 to 11) of key_table_type;

signal key_table : key_table_array;

signal  in00, in01, in02, in03 : std_logic_vector (7 downto 0);
signal  in10, in11, in12, in13 : std_logic_vector (7 downto 0);
signal  in20, in21, in22, in23 : std_logic_vector (7 downto 0);
signal  in30, in31, in32, in33 : std_logic_vector (7 downto 0);
signal  key00, key01, key02, key03 : std_logic_vector (7 downto 0);
signal  key10, key11, key12, key13 : std_logic_vector (7 downto 0);
signal  key20, key21, key22, key23 : std_logic_vector (7 downto 0);
signal  key30, key31, key32, key33 : std_logic_vector (7 downto 0);

begin

process (clk)

begin

if rising_edge(clk) then
```

```
    if en = '1' then

        -- Writing the intermediate key for a specific iteration into
        -- the corresponding address of key-table
        key_table(conv_integer(writeaddr))(0)(0) <= in00;
        key_table(conv_integer(writeaddr))(0)(1) <= in01;
        key_table(conv_integer(writeaddr))(0)(2) <= in02;
        key_table(conv_integer(writeaddr))(0)(3) <= in03;
        key_table(conv_integer(writeaddr))(1)(0) <= in10;
        key_table(conv_integer(writeaddr))(1)(1) <= in11;
        key_table(conv_integer(writeaddr))(1)(2) <= in12;
        key_table(conv_integer(writeaddr))(1)(3) <= in13;
        key_table(conv_integer(writeaddr))(2)(0) <= in20;
        key_table(conv_integer(writeaddr))(2)(1) <= in21;
        key_table(conv_integer(writeaddr))(2)(2) <= in22;
        key_table(conv_integer(writeaddr))(2)(3) <= in23;
        key_table(conv_integer(writeaddr))(3)(0) <= in30;
        key_table(conv_integer(writeaddr))(3)(1) <= in31;
        key_table(conv_integer(writeaddr))(3)(2) <= in32;
        key_table(conv_integer(writeaddr))(3)(3) <= in33;

        if writeaddr = "0000" then
            writecomplete <= '1';
        else
            writecomplete <= '0';
        end if;

    end if;

end if;

end process;

--Splitting 128 bit bus into individual 8 bit lines
in00 <= it(127 downto 120);
in01 <= it(119 downto 112);
in02 <= it(111 downto 104);
in03 <= it(103 downto 96);
in10 <= it(95 downto 88);
in11 <= it(87 downto 80);
in12 <= it(79 downto 72);
in13 <= it(71 downto 64);
in20 <= it(63 downto 56);
in21 <= it(55 downto 48);
in22 <= it(47 downto 40);
in23 <= it(39 downto 32);
in30 <= it(31 downto 24);
in31 <= it(23 downto 16);
in32 <= it(15 downto 8);
in33 <= it(7 downto 0);

--Fetching the intermediate key-data from the key-table
```

```
key00 <= key_table(conv_integer(addr))(0)(0);
key01 <= key_table(conv_integer(addr))(0)(1);
key02 <= key_table(conv_integer(addr))(0)(2);
key03 <= key_table(conv_integer(addr))(0)(3);
key10 <= key_table(conv_integer(addr))(1)(0);
key11 <= key_table(conv_integer(addr))(1)(1);
key12 <= key_table(conv_integer(addr))(1)(2);
key13 <= key_table(conv_integer(addr))(1)(3);
key20 <= key_table(conv_integer(addr))(2)(0);
key21 <= key_table(conv_integer(addr))(2)(1);
key22 <= key_table(conv_integer(addr))(2)(2);
key23 <= key_table(conv_integer(addr))(2)(3);
key30 <= key_table(conv_integer(addr))(3)(0);
key31 <= key_table(conv_integer(addr))(3)(1);
key32 <= key_table(conv_integer(addr))(3)(2);
key33 <= key_table(conv_integer(addr))(3)(3);

--Putting individual 8 bits into 128 bit bus
key <= key00 & key01 & key02 & key03 &
         key10 & key11 & key12 & key13 &
         key20 & key21 & key22 & key23 &
         key30 & key31 & key32 & key33 ;

end expansion_keys_arch;
```

```
-----------------------------------------------------------------------------
--
-- Roundkey Generator Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Generates each rounds key
entity generate_roundkey is
    port (
        count : in std_logic_vector(3 downto 0);
        it : in std_logic_vector(127 downto 0);
        roundkey : out std_logic_vector(127 downto 0)
        );
end;

architecture roundkey_arch of generate_roundkey is

-- Used to store the rcon data
type rcon_row_type is array (0 to 3) of std_logic_vector (7 downto 0);
type rcon_type is array (0 to 9) of rcon_row_type;

-- Rcon is hard coded into a Rom
constant rcon_box : rcon_type :=
        (
            (x"01", x"00", x"00", x"00"),
            (x"02", x"00", x"00", x"00"),
            (x"04", x"00", x"00", x"00"),
            (x"08", x"00", x"00", x"00"),
            (x"10", x"00", x"00", x"00"),
            (x"20", x"00", x"00", x"00"),
            (x"40", x"00", x"00", x"00"),
            (x"80", x"00", x"00", x"00"),
            (x"1b", x"00", x"00", x"00"),
            (x"36", x"00", x"00", x"00")
        );

-- Stores the rotated last column
signal rotword0, rotword1, rotword2, rotword3 : std_logic_vector(7 downto 0);

-- Stores the subbytes result of the previous
signal subres0, subres1, subres2, subres3 : std_logic_vector(7 downto 0);
```

```
signal  temp00, temp01, temp02, temp03,
        temp10, temp11, temp12, temp13,
        temp20, temp21, temp22, temp23,
        temp30, temp31, temp32, temp33: std_logic_vector (7 downto 0);

-- Breaks up input key
signal it00, it01, it02, it03 : std_logic_vector (7 downto 0);
signal it10, it11, it12, it13 : std_logic_vector (7 downto 0);
signal it20, it21, it22, it23 : std_logic_vector (7 downto 0);
signal it30, it31, it32, it33 : std_logic_vector (7 downto 0);


begin

it00 <= it(127 downto 120);
it01 <= it(119 downto 112);
it02 <= it(111 downto 104);
it03 <= it(103 downto 96);
it10 <= it(95 downto 88);
it11 <= it(87 downto 80);
it12 <= it(79 downto 72);
it13 <= it(71 downto 64);
it20 <= it(63 downto 56);
it21 <= it(55 downto 48);
it22 <= it(47 downto 40);
it23 <= it(39 downto 32);
it30 <= it(31 downto 24);
it31 <= it(23 downto 16);
it32 <= it(15 downto 8);
it33 <= it(7 downto 0);

rotword0 <= it13;
rotword1 <= it23;
rotword2 <= it33;
rotword3 <= it03;

-- Performs Sbox transform
SUBBOX : entity work.sbox port map (
          it0 => rotword0, it1 => rotword1, it2 => rotword2, it3 => rotword3,
          ot0 => subres0, ot1 => subres1, ot2 => subres2, ot3 => subres3
);


temp00 <= it00 xor subres0 xor rcon_box(conv_integer(count))(0);
temp10 <= it10 xor subres1 xor rcon_box(conv_integer(count))(1);
temp20 <= it20 xor subres2 xor rcon_box(conv_integer(count))(2);
temp30 <= it30 xor subres3 xor rcon_box(conv_integer(count))(3);

temp01 <= it01 xor temp00;
temp11 <= it11 xor temp10;
temp21 <= it21 xor temp20;
temp31 <= it31 xor temp30;
```

```
temp02 <= it02 xor temp01;
temp12 <= it12 xor temp11;
temp22 <= it22 xor temp21;
temp32 <= it32 xor temp31;

temp03 <= it03 xor temp02;
temp13 <= it13 xor temp12;
temp23 <= it23 xor temp22;
temp33 <= it33 xor temp32;

roundkey <= temp00 & temp01 & temp02 & temp03 &
         temp10 & temp11 & temp12 & temp13 &
         temp20 & temp21 & temp22 & temp23 &
         temp30 & temp31 & temp32 & temp33 ;


end roundkey_arch;
```

```
-----------------------------------------------------------------------------
--
-- Inverse Add Round Key Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--Module for inverse add round key
entity inv_addroundkey is
    port (
        it : in std_logic_vector (127 downto 0);
        key : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
        );
end;

architecture inv_addroundkey_arch of inv_addroundkey is
begin

ot <= it xor key;

end inv_addroundkey_arch;
```

```
-----------------------------------------------------------------------------
--
-- Inverse Mix Columns Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Performs the mixcolumns module of the code
-- We specify all of the values of the for loop as inputs and outputs that would be in the C code
entity inv_mixcolumns is
    port (
        it : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
        );
end;

architecture inv_mixcolumns_arch of inv_mixcolumns is

-- Used in the Multiply calculation
signal multconst0 : std_logic_vector (7 downto 0) := x"0E"; -- 0x0E
signal multconst1 : std_logic_vector (7 downto 0) := x"0B"; -- 0x0B
signal multconst2 : std_logic_vector (7 downto 0) := x"0D"; -- 0x0D
signal multconst3 : std_logic_vector (7 downto 0) := x"09"; -- 0x09

signal mainconst :std_logic_vector (7 downto 0) := x"FF"; -- 0xFF

signal it00, it01, it02, it03 : std_logic_vector (7 downto 0);
signal it10, it11, it12, it13 : std_logic_vector (7 downto 0);
signal it20, it21, it22, it23 : std_logic_vector (7 downto 0);
signal it30, it31, it32, it33 : std_logic_vector (7 downto 0);
signal ot00, ot01, ot02, ot03 : std_logic_vector (7 downto 0);
signal ot10, ot11, ot12, ot13 : std_logic_vector (7 downto 0);
signal ot20, ot21, ot22, ot23 : std_logic_vector (7 downto 0);
signal ot30, ot31, ot32, ot33 : std_logic_vector (7 downto 0);

begin

it00 <= it(127 downto 120);
it01 <= it(119 downto 112);
it02 <= it(111 downto 104);
it03 <= it(103 downto 96);
it10 <= it(95 downto 88);
it11 <= it(87 downto 80);
```

```
it12 <= it(79 downto 72);
it13 <= it(71 downto 64);
it20 <= it(63 downto 56);
it21 <= it(55 downto 48);
it22 <= it(47 downto 40);
it23 <= it(39 downto 32);
it30 <= it(31 downto 24);
it31 <= it(23 downto 16);
it32 <= it(15 downto 8);
it33 <= it(7 downto 0);

multconst0 <= multconst0;
multconst1 <= multconst1;
multconst2 <= multconst2;
multconst3 <= multconst3;
mainconst <= mainconst;

-- Perform all 16 Row Calculations
Out00: entity work.inv_multiply_row port map(
        input0 => it00, const0 => multconst0,
        input1 => it10, const1 => multconst1,
        input2 => it20, const2 => multconst2,
        input3 => it30, const3 => multconst3,
        output => ot00);

Out10: entity work.inv_multiply_row port map(
        input0 => it00, const0 => multconst3,
        input1 => it10, const1 => multconst0,
        input2 => it20, const2 => multconst1,
        input3 => it30, const3 => multconst2,
        output => ot10);

Out20: entity work.inv_multiply_row port map(
        input0 => it00, const0 => multconst2,
        input1 => it10, const1 => multconst3,
        input2 => it20, const2 => multconst0,
        input3 => it30, const3 => multconst1,
        output => ot20);

Out30: entity work.inv_multiply_row port map(
        input0 => it00, const0 => multconst1,
        input1 => it10, const1 => multconst2,
        input2 => it20, const2 => multconst3,
        input3 => it30, const3 => multconst0,
        output => ot30);

Out01: entity work.inv_multiply_row port map(
        input0 => it01, const0 => multconst0,
        input1 => it11, const1 => multconst1,
        input2 => it21, const2 => multconst2,
        input3 => it31, const3 => multconst3,
        output => ot01);
```

```
Out11: entity work.inv_multiply_row port map(
        input0 => it01, const0 => multconst3,
        input1 => it11, const1 => multconst0,
        input2 => it21, const2 => multconst1,
        input3 => it31, const3 => multconst2,
        output => ot11);

Out21: entity work.inv_multiply_row port map(
        input0 => it01, const0 => multconst2,
        input1 => it11, const1 => multconst3,
        input2 => it21, const2 => multconst0,
        input3 => it31, const3 => multconst1,
        output => ot21);

Out31: entity work.inv_multiply_row port map(
        input0 => it01, const0 => multconst1,
        input1 => it11, const1 => multconst2,
        input2 => it21, const2 => multconst3,
        input3 => it31, const3 => multconst0,
        output => ot31);

Out02: entity work.inv_multiply_row port map(
        input0 => it02, const0 => multconst0,
        input1 => it12, const1 => multconst1,
        input2 => it22, const2 => multconst2,
        input3 => it32, const3 => multconst3,
        output => ot02);

Out12: entity work.inv_multiply_row port map(
        input0 => it02, const0 => multconst3,
        input1 => it12, const1 => multconst0,
        input2 => it22, const2 => multconst1,
        input3 => it32, const3 => multconst2,
        output => ot12);

Out22: entity work.inv_multiply_row port map(
        input0 => it02, const0 => multconst2,
        input1 => it12, const1 => multconst3,
        input2 => it22, const2 => multconst0,
        input3 => it32, const3 => multconst1,
        output => ot22);

Out32: entity work.inv_multiply_row port map(
        input0 => it02, const0 => multconst1,
        input1 => it12, const1 => multconst2,
        input2 => it22, const2 => multconst3,
        input3 => it32, const3 => multconst0,
        output => ot32);

Out03: entity work.inv_multiply_row port map(
        input0 => it03, const0 => multconst0,
        input1 => it13, const1 => multconst1,
        input2 => it23, const2 => multconst2,
```

```
        input3 => it33, const3 => multconst3,
        output => ot03);

Out13: entity work.inv_multiply_row port map(
        input0 => it03, const0 => multconst3,
        input1 => it13, const1 => multconst0,
        input2 => it23, const2 => multconst1,
        input3 => it33, const3 => multconst2,
        output => ot13);

Out23: entity work.inv_multiply_row port map(
        input0 => it03, const0 => multconst2,
        input1 => it13, const1 => multconst3,
        input2 => it23, const2 => multconst0,
        input3 => it33, const3 => multconst1,
        output => ot23);

Out33: entity work.inv_multiply_row port map(
        input0 => it03, const0 => multconst1,
        input1 => it13, const1 => multconst2,
        input2 => it23, const2 => multconst3,
        input3 => it33, const3 => multconst0,
        output => ot33);

ot <= ot00 & ot01 & ot02 & ot03 &
         ot10 & ot11 & ot12 & ot13 &
         ot20 & ot21 & ot22 & ot23 &
         ot30 & ot31 & ot32 & ot33 ;

end inv_mixcolumns_arch;
```

```
-------------------------------------------------------------------------------
--
-- Inverse Mtimes Module -- Performs the mtimes calculation as part of the inverse mix columns
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;

use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Module to perform mtimes calculation - modularized to reuse parts within calculation
entity inv_mtimes is
    port (
        input : in std_logic_vector (7 downto 0);
        output : out std_logic_vector (7 downto 0)
        );
end;

architecture inv_mtimes_arch of inv_mtimes is

-- Stores it << 1
signal temp : std_logic_vector (7 downto 0);

begin

-- Outputs out = (in << 1) ||
temp <= x"1b" when input(7) = '1' else
     (others => '0');

output <= (input(6 downto 0) & "0") xor temp;

end inv_mtimes_arch;
```

```
-------------------------------------------------------------------------------
--
-- Inverse Multiply Module (Part of Inverse Mix Columns Module)
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Module to perform mtimes calculation - modularized to reuse parts within calculation
entity inv_multiply is
    port (
        input : in std_logic_vector (7 downto 0);
        const : in std_logic_vector (7 downto 0);
        output : out std_logic_vector (7 downto 0)
        );
end;

architecture inv_multiply_arch of inv_multiply is

-- Store Mtimes results
signal mtimes1 : std_logic_vector (7 downto 0);
signal mtimes2 : std_logic_vector (7 downto 0);
signal mtimes3 : std_logic_vector (7 downto 0);

-- Store const >> # * mtimes^#(input)
signal term0 : std_logic_vector (7 downto 0);
signal term1 : std_logic_vector (7 downto 0);
signal term2 : std_logic_vector (7 downto 0);
signal term3 : std_logic_vector (7 downto 0);

begin

-- mtimes(input)
Calc1: entity work.inv_mtimes port map(
        input => input, output => mtimes1
);
-- mtimes(mtimes(input))
Calc2: entity work.inv_mtimes port map(
        input => mtimes1, output => mtimes2
);
-- mtimes(mtimes(mtimes(input)))
Calc3: entity work.inv_mtimes port map(
        input => mtimes2 , output => mtimes3
);
```

```
-- (const & 1) * input
term0 <= input when const(0) = '1' else
      (others => '0');
-- (const >> 1 & 1) * mtimes(input)
term1 <= mtimes1 when const(1) = '1' else
      (others => '0');
-- (const >> 2 & 1) * mtimes(mtimes(input))
term2 <= mtimes2 when const(2) = '1' else
      (others => '0');
-- (const >> 3 & 1) * mtimes(mtimes(mtimes(input)))
term3 <= mtimes3 when const(3) = '1' else
      (others => '0');

output <= term0 xor term1 xor term2 xor term3;

end inv_multiply_arch;
```

```
---------------------------------------------------------------------------
--
-- Inverse Multiply Row Module -- Part of the Mix Columns Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
---------------------------------------------------------------------------


library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Module to perform mtimes calculation - modularized to reuse parts within calculation
entity inv_multiply_row is
    port (
        input0, input1, input2, input3 : in std_logic_vector (7 downto 0);
        const0, const1, const2, const3 : in std_logic_vector (7 downto 0);
        output: out std_logic_vector (7 downto 0)
        );
end;

architecture inv_multiply_row_arch of inv_multiply_row is

-- Store each multiplication for the output
signal term0 : std_logic_vector (7 downto 0);
signal term1 : std_logic_vector (7 downto 0);
signal term2 : std_logic_vector (7 downto 0);
signal term3 : std_logic_vector (7 downto 0);

signal mainconst : std_logic_vector (7 downto 0) := x"FF";

begin

mainconst <= mainconst;

-- (Multiply(input1, const1) ^ Multiply(input2, const2) ^ Multiply(input3, const3) ^ Multiply(input4, const4));
Calc00: entity work.inv_multiply port map(
        input => input0, const => const0, output => term0);
Calc01: entity work.inv_multiply port map(
        input => input1, const => const1, output => term1);
Calc02: entity work.inv_multiply port map(
        input => input2, const => const2, output => term2);
Calc03: entity work.inv_multiply port map(
        input => input3, const => const3, output => term3);

output <= term0 xor term1 xor term2 xor term3;
```

```
end inv_multiply_row_arch;
```

```
----------------------------------------------------------------------------
--
-- Inverse SBox Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--                  Larry Chen (lc2454)
--                  Abhinandan Majumdar (am2993)
--                  Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--Module for inverse S-Box
entity inv_sbox is
    port (
        it : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector  (127 downto 0)
        );
end;

architecture inv_sbox_arch of inv_sbox is
type sbox_row_type is array (0 to 15) of std_logic_vector (7 downto 0);
type sbox_type is array (0 to 15) of sbox_row_type;

signal it00, it01, it02, it03 : std_logic_vector (7 downto 0);
signal it10, it11, it12, it13 : std_logic_vector (7 downto 0);
signal it20, it21, it22, it23 : std_logic_vector (7 downto 0);
signal it30, it31, it32, it33 : std_logic_vector (7 downto 0);
signal ot00, ot01, ot02, ot03 : std_logic_vector (7 downto 0);
signal ot10, ot11, ot12, ot13 : std_logic_vector (7 downto 0);
signal ot20, ot21, ot22, ot23 : std_logic_vector (7 downto 0);
signal ot30, ot31, ot32, ot33 : std_logic_vector (7 downto 0);


--Inverse S-table
constant i_s_box : sbox_type :=
        (
            (x"52", x"09", x"6a", x"d5", x"30", x"36", x"a5", x"38", x"bf", x"40", x"a3", x"9e",
 x"81", x"f3", x"d7", x"fb"),
            (x"7c", x"e3", x"39", x"82", x"9b", x"2f", x"ff", x"87", x"34", x"8e", x"43", x"44",
 x"c4", x"de", x"e9", x"cb"),
            (x"54", x"7b", x"94", x"32", x"a6", x"c2", x"23", x"3d", x"ee", x"4c", x"95", x"0b",
 x"42", x"fa", x"c3", x"4e"),
            (x"08", x"2e", x"a1", x"66", x"28", x"d9", x"24", x"b2", x"76", x"5b", x"a2", x"49",
 x"6d", x"8b", x"d1", x"25"),
            (x"72", x"f8", x"f6", x"64", x"86", x"68", x"98", x"16", x"d4", x"a4", x"5c", x"cc",
 x"5d", x"65", x"b6", x"92"),
            (x"6c", x"70", x"48", x"50", x"fd", x"ed", x"b9", x"da", x"5e", x"15", x"46", x"57",
```

```
x"a7", x"8d", x"9d", x"84"),
            (x"90", x"d8", x"ab", x"00", x"8c", x"bc", x"d3", x"0a", x"f7", x"e4", x"58", x"05",
x"b8", x"b3", x"45", x"06"),
            (x"d0", x"2c", x"1e", x"8f", x"ca", x"3f", x"0f", x"02", x"c1", x"af", x"bd", x"03",
x"01", x"13", x"8a", x"6b"),
            (x"3a", x"91", x"11", x"41", x"4f", x"67", x"dc", x"ea", x"97", x"f2", x"cf", x"ce",
x"f0", x"b4", x"e6", x"73"),
            (x"96", x"ac", x"74", x"22", x"e7", x"ad", x"35", x"85", x"e2", x"f9", x"37", x"e8",
x"1c", x"75", x"df", x"6e"),
            (x"47", x"f1", x"1a", x"71", x"1d", x"29", x"c5", x"89", x"6f", x"b7", x"62", x"0e",
x"aa", x"18", x"be", x"1b"),
            (x"fc", x"56", x"3e", x"4b", x"c6", x"d2", x"79", x"20", x"9a", x"db", x"c0", x"fe",
x"78", x"cd", x"5a", x"f4"),
            (x"1f", x"dd", x"a8", x"33", x"88", x"07", x"c7", x"31", x"b1", x"12", x"10", x"59",
x"27", x"80", x"ec", x"5f"),
            (x"60", x"51", x"7f", x"a9", x"19", x"b5", x"4a", x"0d", x"2d", x"e5", x"7a", x"9f",
x"93", x"c9", x"9c", x"ef"),
            (x"a0", x"e0", x"3b", x"4d", x"ae", x"2a", x"f5", x"b0", x"c8", x"eb", x"bb", x"3c",
x"83", x"53", x"99", x"61"),
            (x"17", x"2b", x"04", x"7e", x"ba", x"77", x"d6", x"26", x"e1", x"69", x"14", x"63",
x"55", x"21", x"0c", x"7d")
        );
begin

--Splitting 128 bit bus into individual 8 bit lines
it00 <= it(127 downto 120);
it01 <= it(119 downto 112);
it02 <= it(111 downto 104);
it03 <= it(103 downto 96);
it11 <= it(95 downto 88);
it12 <= it(87 downto 80);
it13 <= it(79 downto 72);
it10 <= it(71 downto 64);
it22 <= it(63 downto 56);
it23 <= it(55 downto 48);
it20 <= it(47 downto 40);
it21 <= it(39 downto 32);
it33 <= it(31 downto 24);
it30 <= it(23 downto 16);
it31 <= it(15 downto 8);
it32 <= it(7 downto 0);



--Replacing the data from Inverse S-box
ot00 <= i_s_box(conv_integer(it00(7 downto 4)))(conv_integer(it00(3 downto 0)));
ot01 <= i_s_box(conv_integer(it01(7 downto 4)))(conv_integer(it01(3 downto 0)));
ot02 <= i_s_box(conv_integer(it02(7 downto 4)))(conv_integer(it02(3 downto 0)));
ot03 <= i_s_box(conv_integer(it03(7 downto 4)))(conv_integer(it03(3 downto 0)));
ot10 <= i_s_box(conv_integer(it10(7 downto 4)))(conv_integer(it10(3 downto 0)));
ot11 <= i_s_box(conv_integer(it11(7 downto 4)))(conv_integer(it11(3 downto 0)));
ot12 <= i_s_box(conv_integer(it12(7 downto 4)))(conv_integer(it12(3 downto 0)));
ot13 <= i_s_box(conv_integer(it13(7 downto 4)))(conv_integer(it13(3 downto 0)));
```

```
ot20 <= i_s_box(conv_integer(it20(7 downto 4)))(conv_integer(it20(3 downto 0)));
ot21 <= i_s_box(conv_integer(it21(7 downto 4)))(conv_integer(it21(3 downto 0)));
ot22 <= i_s_box(conv_integer(it22(7 downto 4)))(conv_integer(it22(3 downto 0)));
ot23 <= i_s_box(conv_integer(it23(7 downto 4)))(conv_integer(it23(3 downto 0)));
ot30 <= i_s_box(conv_integer(it30(7 downto 4)))(conv_integer(it30(3 downto 0)));
ot31 <= i_s_box(conv_integer(it31(7 downto 4)))(conv_integer(it31(3 downto 0)));
ot32 <= i_s_box(conv_integer(it32(7 downto 4)))(conv_integer(it32(3 downto 0)));
ot33 <= i_s_box(conv_integer(it33(7 downto 4)))(conv_integer(it33(3 downto 0)));

-- Putting individual 8 bit lines to a 128 bit wide bus
ot <= ot00 & ot01 & ot02 & ot03 &
         ot10 & ot11 & ot12 & ot13 &
         ot20 & ot21 & ot22 & ot23 &
         ot30 & ot31 & ot32 & ot33 ;

end inv_sbox_arch;
```

```
-----------------------------------------------------------------------------
--
-- Inverse Shift Row Sub Bytes Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--Module for Inverse_shiftrow_subbytes
entity inv_shiftrow_subbytes is
    port (
        it : in std_logic_vector (127 downto 0);
        ot : out std_logic_vector (127 downto 0)
        );
end;

architecture inv_shiftrow_subbytes_arch of inv_shiftrow_subbytes is
begin
I_SB1 : entity work.inv_sbox port map(
        it => it,
        ot => ot
);
end inv_shiftrow_subbytes_arch;
```

```
----------------------------------------------------------------------------
--
-- Key Generation Controller
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
----------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

--Module for Key Controller
entity key_controller is
port (
    clk, en : in std_logic;
    count   : out std_logic_vector(3 downto 0);
    e0,e1,e2,eoc    : out std_logic
    );
end;

architecture key_controller_arch of key_controller is

type state_type is (S0, S1, S2, S3, S4);
signal state : state_type := S0;

signal t_count : std_logic_vector(3 downto 0) := "0000";
signal twocycle : std_logic_vector(1 downto 0) := "00";

begin

process (clk)
begin

if rising_edge(clk) then

    case state is

        -- First State - Store key in the SRAM
        when S0 =>
                    if en = '1' then

                        t_count <= t_count + '1';
                        state <= S1;

                    end if;
```

```
        -- Second State - Select Key for calculation of round key
        when S1 =>
                    if en = '1' then

                        t_count <= t_count + '1';
                        state <= S2;

                    end if;

        -- Third State - Select Last Round Key for calculation of Current Round Key
        when S2 =>

                    if en = '1' then
                        if t_count = "1001" then
                            state <= S3;
                        else
                            t_count <= t_count + '1';
                        end if;
                    end if;

        -- Added to wait two cycles for writing
        when S3 =>
                    if en = '1' then
                        if twocycle = "01" then
                            twocycle <= twocycle;
                            state <= S4;
                        else
                            twocycle <= twocycle + '1';
                        end if;
                    end if;

        when S4 =>
                    if en = '1' then
                        state <= S4;
                    elsif en = '0' then
                        state <= S0;
                        t_count <= "0000";
                        twocycle <= "00";
                    end if;

    end case;


end if;

end process;

-- Iteration Count
count <= t_count;

--Setting the control bits required for Key-Controller Datapath
e0 <= '0' when state = S0 and en = '1' else
```

```
      '1' when state = S1 and en = '1' else
      '1' when state = S2 and en = '1' else
      '1' when state = S3 and en = '1' else
      '0';

e1 <= '1' when state = S0 and en = '1' else
      '1' when state = S1 and en = '1' else
      '1' when state = S2 and en = '1' else
      '0' when state = S3 and en = '1' else
      '0';

e2 <= '0' when state = S0 and en = '1' else
      '1' when state = S1 and en = '1' else
      '1' when state = S2 and en = '1' else
      '1' when state = S3 and en = '1' else
      '1';

eoc <= '1' when state = S0 and en = '1' else
       '1' when state = S1 and en = '1' else
       '1' when state = S2 and en = '1' else
       '1' when state = S3 and en = '1' else
       '0';
end key_controller_arch;
```

```
-------------------------------------------------------------------------------
--
-- 2-to-1 Multiplexer
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;

use ieee.std_logic_1164.all;

--128x1 Mux Module
entity mux128_1 is
    port (
        i0,i1 : in std_logic_vector (127 downto 0);
        s     : in std_logic;
        o : out std_logic_vector (127 downto 0)
        );
end;

architecture mux128_1_arch of mux128_1 is
begin
o <= i0 when s = '0' else
     i1 when s = '1' else
     (others => 'X');

end mux128_1_arch;
```

```
------------------------------------------------------------------------
--
-- 128 Bit Register
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--                  Larry Chen (lc2454)
--                  Abhinandan Majumdar (am2993)
--                  Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;

--128bit register module
entity regis128 is
    port (
        i           : in std_logic_vector (127 downto 0);
        clk, en     : in std_logic;
        o           : out std_logic_vector (127 downto 0)
        );
end;

architecture regis128_arch of regis128 is
begin

process(clk)
    begin
    if rising_edge(clk) then
     if en = '1' then
     o <= i;
     end if;
    end if;
end process;

end regis128_arch;
```

```
-----------------------------------------------------------------------------
--
-- Forward Sub Box Module
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library IEEE;

use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Forwards SBox used for generating the round keys
entity sbox is
    port (
        it0, it1, it2, it3 : in std_logic_vector (7 downto 0);
        ot0, ot1, ot2, ot3 : out std_logic_vector (7 downto 0)
        );
end;

architecture sbox_arch of sbox is
type sbox_row_type is array (0 to 15) of std_logic_vector (7 downto 0);
type sbox_type is array (0 to 15) of sbox_row_type;

-- Details are hard coded into a rom
constant i_s_box : sbox_type :=
        (
            (x"63", x"7c", x"77", x"7b", x"f2", x"6b", x"6f", x"c5", x"30", x"01", x"67", x"2b",
 x"fe", x"d7", x"ab", x"76"),
            (x"ca", x"82", x"c9", x"7d", x"fa", x"59", x"47", x"f0", x"ad", x"d4", x"a2", x"af",
 x"9c", x"a4", x"72", x"c0"),
            (x"b7", x"fd", x"93", x"26", x"36", x"3f", x"f7", x"cc", x"34", x"a5", x"e5", x"f1",
 x"71", x"d8", x"31", x"15"),
            (x"04", x"c7", x"23", x"c3", x"18", x"96", x"05", x"9a", x"07", x"12", x"80", x"e2",
 x"eb", x"27", x"b2", x"75"),
            (x"09", x"83", x"2c", x"1a", x"1b", x"6e", x"5a", x"a0", x"52", x"3b", x"d6", x"b3",
 x"29", x"e3", x"2f", x"84"),
            (x"53", x"d1", x"00", x"ed", x"20", x"fc", x"b1", x"5b", x"6a", x"cb", x"be", x"39",
 x"4a", x"4c", x"58", x"cf"),
            (x"d0", x"ef", x"aa", x"fb", x"43", x"4d", x"33", x"85", x"45", x"f9", x"02", x"7f",
 x"50", x"3c", x"9f", x"a8"),
            (x"51", x"a3", x"40", x"8f", x"92", x"9d", x"38", x"f5", x"bc", x"b6", x"da", x"21",
 x"10", x"ff", x"f3", x"d2"),
            (x"cd", x"0c", x"13", x"ec", x"5f", x"97", x"44", x"17", x"c4", x"a7", x"7e", x"3d",
 x"64", x"5d", x"19", x"73"),
            (x"60", x"81", x"4f", x"dc", x"22", x"2a", x"90", x"88", x"46", x"ee", x"b8", x"14",
 x"de", x"5e", x"0b", x"db"),
```

```
            (x"e0", x"32", x"3a", x"0a", x"49", x"06", x"24", x"5c", x"c2", x"d3", x"ac", x"62",
 x"91", x"95", x"e4", x"79"),
            (x"e7", x"c8", x"37", x"6d", x"8d", x"d5", x"4e", x"a9", x"6c", x"56", x"f4", x"ea",
 x"65", x"7a", x"ae", x"08"),
            (x"ba", x"78", x"25", x"2e", x"1c", x"a6", x"b4", x"c6", x"e8", x"dd", x"74", x"1f",
 x"4b", x"bd", x"8b", x"8a"),
            (x"70", x"3e", x"b5", x"66", x"48", x"03", x"f6", x"0e", x"61", x"35", x"57", x"b9",
 x"86", x"c1", x"1d", x"9e"),
            (x"e1", x"f8", x"98", x"11", x"69", x"d9", x"8e", x"94", x"9b", x"1e", x"87", x"e9",
 x"ce", x"55", x"28", x"df"),
            (x"8c", x"a1", x"89", x"0d", x"bf", x"e6", x"42", x"68", x"41", x"99", x"2d", x"0f",
 x"b0", x"54", x"bb", x"16")
        );
begin

ot0 <= i_s_box(conv_integer(it0(7 downto 4)))(conv_integer(it0(3 downto 0)));
ot1 <= i_s_box(conv_integer(it1(7 downto 4)))(conv_integer(it1(3 downto 0)));
ot2 <= i_s_box(conv_integer(it2(7 downto 4)))(conv_integer(it2(3 downto 0)));
ot3 <= i_s_box(conv_integer(it3(7 downto 4)))(conv_integer(it3(3 downto 0)));

end sbox_arch;
```

```
-------------------------------------------------------------------------------
--
-- Expansion Keys Write Controller
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

--Module for write-controller
entity write_controller is
port (
    clk, en : in std_logic;
    addr : out std_logic_vector(3 downto 0)
    );
end;

architecture write_controller_arch of write_controller is

type state_type is (S0, S1);
signal state : state_type := S0;

signal t_addr : std_logic_vector(3 downto 0) := "1011";

begin

process (clk)
begin

if rising_edge(clk) then

    case state is

        -- First State - Store key in the SRAM
        when S0 =>
                    if en = '1' then

                        if t_addr = "0000" then
                            state <= S1;
                            t_addr <= t_addr;
                        else
                            t_addr <= t_addr - '1';
                        end if;
```

```
                    end if;


            -- Second State - Select Key for calculation of round key
            when S1 =>
                    if en = '1' then
                        state <= S1;
                    elsif en = '0' then
                        state <= S0;
                        t_addr <= "1011";
                    end if;

        end case;


end if;

end process;


addr <= t_addr;

end write_controller_arch;
```

```vhdl
-------------------------------------------------------------------------------
--
-- SD/MMC interface (SPI-style) for the Apple ][ Emulator
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Originally authored by Stephen A. Edwards (sedwards@cs.columbia.edu)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity spi_controller is

  port (

    -- Bus interface signals
    read        : in std_logic;
    write       : in std_logic;
    chipselect  : in std_logic;
    address     : in unsigned (23 downto 0);
    readdata    : out unsigned (31 downto 0);
    writedata   : in unsigned (31 downto 0);
    -- End of bus interface

    CS_N          : out std_logic := '1';       -- MMC chip select
    MOSI          : out std_logic;              -- Data to card (master out slave in)
    MISO          : in  std_logic;              -- Data from card (master in slave out)
    SCLK          : out std_logic;              -- Card clock

    clk           : in  std_logic;              -- System clock
    reset         : in  std_logic);

end spi_controller;

architecture rtl of spi_controller is

  type states is (RESET_STATE,
                  RESET_CLOCKS1,
                  RESET_CLOCKS2,
                  RESET_SEND_CMD0,
                  RESET_SEND_CMD1,
                  RESET_CHECK_CMD1,
                  RESET_SEND_SET_BLOCKLEN,
                  IDLE,
                  READ_BLOCK,
```

```
                    READ_BLOCK_WAIT,
                    READ_BLOCK_DATA,
                    READ_BLOCK_CRC,
                    SEND_CMD,
                    RECEIVE_BYTE_WAIT,
                    RECEIVE_BYTE
);

  signal state : states := RESET_STATE;
  signal return_state : states;
  signal sclk_sig : std_logic := '0';
  signal counter : unsigned(7 downto 0) := (others => '0');
  signal byte_counter : unsigned(31 downto 0) := (others => '0');
  signal command : unsigned(55 downto 0) := (others => '1');
  signal recv_byte : unsigned(7 downto 0);
  signal CRC_byte   : std_logic := '0';
  signal start      : std_logic := '0';
  signal addr       : unsigned(31 downto 0) := (others => '0');
  signal eor        : std_logic;
  signal rec32bit   : unsigned(31 downto 0);
  signal bufAddr    : integer;
  signal bufResult  : unsigned (31 downto 0);
  signal bufCounter : unsigned(8 downto 0) := "000000000";

  type ram_type is array (0 to 511) of unsigned(7 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

begin

  SCLK <= sclk_sig;

    process (clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                readdata <= (others => '0');
            else
                if chipselect = '1' then
                    if address(0) = '1' then
                        if write = '1' then
                            start <=  writedata(0);
                        end if;
                    elsif address(1) = '1' then
                        if write = '1' then
                            addr <= writedata;
                        end if;
                    elsif address(4) = '1' then
                        if read = '1' then
                            readdata <= "0000000000000000000000000000000" & eor;
                        end if;
                    elsif address(6) = '1' then
                        if write = '1' then
                            bufAddr <= to_integer(writedata);
```

```
                    end if;
                elsif address(7) = '1' then
                    if read = '1' then
                        readdata <= bufResult;
                    end if;
                end if;
            end if;
        end if;
    end if;
end process;

-- Give the data to the processor in 32 bit blocks
bufResult <= RAM(bufAddr) & RAM(bufAddr + 1) & RAM(bufAddr + 2) & RAM(bufAddr + 3);

  fsm_ff : process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        state <= RESET_STATE;
        sclk_sig <= '0';
        CS_N <= '1';
        command <= (others => '1');
        counter <= (others => '0');
        byte_counter <= (others => '0');
        eor <= '0';
      else
        case state is

          when RESET_STATE =>
            counter <= TO_UNSIGNED(160,8);
            state <= RESET_CLOCKS1;

            -- Output a series of clock signals to wake up the chip
          when RESET_CLOCKS1 =>
            if counter = 0 then
              counter <= TO_UNSIGNED(32, 8);
              CS_N <= '0';
              state <= RESET_CLOCKS2;
            else
              counter <= counter - 1;
              sclk_sig <= not sclk_sig;
            end if;

          when RESET_CLOCKS2 =>
            if counter = 0 then
              state <= RESET_SEND_CMD0;
            else
              counter <= counter - 1;
              sclk_sig <= not sclk_sig;
            end if;

            -- Send CMD0 : GO_IDLE_STATE
          when RESET_SEND_CMD0 =>
```

```
          command <= x"FF400000000095";
          counter <= TO_UNSIGNED(55, 8);
          return_state <= RESET_SEND_CMD1;
          state <= SEND_CMD;

          -- Send CMD1 : SEND_OP_CMD
        when RESET_SEND_CMD1 =>
          command <= x"FF410000000001";
          counter <= TO_UNSIGNED(55, 8);
          return_state <= RESET_CHECK_CMD1;
          state <= SEND_CMD;

          -- Wait for a response from the card
        when RESET_CHECK_CMD1 =>
          if recv_byte = x"00" then
            state <= RESET_SEND_SET_BLOCKLEN;
          else
           state <= RESET_SEND_CMD1;
          end if;

          -- Set the block length of the card to 512 bytes
        when RESET_SEND_SET_BLOCKLEN =>
          command <= x"FF500000020001";   -- CMD16: SET_BLOCKLEN (512 bytes)
          counter <= TO_UNSIGNED(55, 8);
          return_state <= IDLE;
          state <= SEND_CMD;

          -- Idle until a start signal is sent
        when IDLE =>
          eor <= '1';
          if start = '1' then
            state <= READ_BLOCK;
            eor <= '0';
          end if;

        -- Set the block read starting address
        when READ_BLOCK =>
          command <= x"FF51" & addr & x"01";   -- READ_SINGLE_BLOCK
          counter <= TO_UNSIGNED(55, 8);
          return_state <= READ_BLOCK_WAIT;
          state <= SEND_CMD;

          -- Wait for a 0 to signal the start of the block,
          -- then read the first byte
        when READ_BLOCK_WAIT =>
          if sclk_sig = '1' and MISO = '0' then
              state <= READ_BLOCK_DATA;
              byte_counter <= x"000001FF";      -- Set the byte counter to 511 [0-> 511]
              bufCounter <= (others => '0');
              counter <= TO_UNSIGNED(7, 8);
          end if;
          sclk_sig <= not sclk_sig;
```

```
        -- Read bytes from the data until all the bytes have been read
        -- If all bytes are read then we enter the READ_BLOCK_CRC state
      when READ_BLOCK_DATA =>
        if byte_counter = x"00" then
          counter <= TO_UNSIGNED(7, 8);
          return_state <= READ_BLOCK_CRC;
          state <= RECEIVE_BYTE;
        else
          byte_counter <= byte_counter - 1;
          return_state <= READ_BLOCK_DATA;
          counter <= TO_UNSIGNED(7, 8);
          state <= RECEIVE_BYTE;
        end if;

      when READ_BLOCK_CRC =>
        counter <= TO_UNSIGNED(15, 8);
        CRC_byte <= '1';
        return_state <= IDLE;
        state <= RECEIVE_BYTE;

      -- Send the command.  Set counter=54 and return_state first
      when SEND_CMD =>
        if sclk_sig = '1' then
          if counter = 0 then
            state <= RECEIVE_BYTE_WAIT;
          else
            counter <= counter - 1;
            command <= command(54 downto 0) & "1";
          end if;
        end if;
        sclk_sig <= not sclk_sig;

      -- Wait for a "0", indicating the first bit of a response.
      -- Set return_state first
      when RECEIVE_BYTE_WAIT =>
        if sclk_sig = '1' then
          if MISO = '0' then
            recv_byte <= (others => '0');
            counter <= TO_UNSIGNED(6, 8);   -- Already read bit 7
            state <= RECEIVE_BYTE;
          end if;
        end if;
        sclk_sig <= not sclk_sig;

      -- Receive a byte.  Set counter to 7 and return_state before entry
      when RECEIVE_BYTE =>
        if sclk_sig = '1' then
          recv_byte <= recv_byte(6 downto 0) & MISO;
          if counter = 0 then
            state <= return_state;
            -- Buffer all the data in RAM
            if CRC_byte = '0' then
                RAM(to_integer(bufCounter)) <= recv_byte(6 downto 0) & MISO;
```

```
                    bufCounter <= bufCounter + 1;
                -- Ignore bytes read after data block
                else
                    eor <= '1';
                    CRC_byte <= '0';
                end if;
              else
                counter <= counter - 1;
              end if;
            end if;
            sclk_sig <= not sclk_sig;

          when others => null;
        end case;
      end if;
    end if;
  end process fsm_ff;

-- Data sent to the SD Card, MSB first
MOSI <= command(55);

end rtl;
```

```
-------------------------------------------------------------------------------
--
-- VGA/SRAM supercontroller
--  This module integrates the VGA and SRAM modules
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--                  Larry Chen (lc2454)
--                  Abhinandan Majumdar (am2993)
--                  Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity vga_sram_supercontroller is

  port (
    reset : in std_logic;
    CLOCK_50 : in std_logic;

    signal chipselect : in std_logic;
    signal write, read : in std_logic;
    signal address  :  in std_logic_vector(20 downto 0);
    signal readdata : out std_logic_vector(15 downto 0);
    signal writedata : in std_logic_vector(15 downto 0);
    signal byteenable : in std_logic_vector(1 downto 0);

    VGA_CLK,                                          -- Clock
    VGA_HS,                                           -- H_SYNC
    VGA_VS,                                           -- V_SYNC
    VGA_BLANK,                                        -- BLANK
    VGA_SYNC : out std_logic;                         -- SYNC
    VGA_R,
    VGA_G,
    VGA_B : out std_logic_vector(9 downto 0); -- R=G=B[9:0]

    SRAM_DQ   : inout std_logic_vector(15 downto 0);
    SRAM_ADDR : out std_logic_vector(17 downto 0);
    SRAM_UB_N, SRAM_LB_N : out std_logic;
    SRAM_WE_N, SRAM_CE_N : out std_logic;
    SRAM_OE_N : out std_logic
    );

end vga_sram_supercontroller;

architecture vga_sram_supercontroller_arch of vga_sram_supercontroller is

  signal address2 : std_logic_vector(17 downto 0) := "000000000000110010";
```

```
  signal readdata2 : std_logic_vector(15 downto 0);
  signal go : std_logic := '1';
  signal clk25 : std_logic := '0';   --25 MHz clock for the VGA controller
  signal clock_div : unsigned(25 downto 0) := "00000000000000000000000000";
  signal rectangle_from_vga : std_logic := '0';
  signal readdata_from_sram : std_logic_vector(15 downto 0);

component de2_sram_controller is

  port (
    signal chipselect : in std_logic;
    signal write, read : in std_logic;
    signal address  :  in std_logic_vector(17 downto 0);
    signal readdata : out std_logic_vector(15 downto 0);
    signal writedata : in std_logic_vector(15 downto 0);
    signal byteenable : in std_logic_vector(1 downto 0);

    --From VGA
    signal address2 :  in std_logic_vector(17 downto 0);
    signal readdata2 : out std_logic_vector(15 downto 0);

    signal go : in std_logic;        --Go from Nios

    signal SRAM_DQ : inout std_logic_vector(15 downto 0);
    signal SRAM_ADDR : out std_logic_vector(17 downto 0);
    signal SRAM_UB_N, SRAM_LB_N : out std_logic;
    signal SRAM_WE_N, SRAM_CE_N : out std_logic;
    signal SRAM_OE_N : out std_logic
    );

end component de2_sram_controller;


component de2_vga_raster is

  port (
--   reset : in std_logic;
    clk    : in std_logic;       -- Should be 25.125 MHz

    -- SRAM signals
    address_out : out std_logic_vector(17 downto 0);
    data_in     : in std_logic_vector(15 downto 0);

    -- Rectangle
    rectangle_out : out std_logic;

    VGA_CLK,                          -- Clock
    VGA_HS,                           -- H_SYNC
    VGA_VS,                           -- V_SYNC
    VGA_BLANK,                        -- BLANK
    VGA_SYNC : out std_logic;         -- SYNC
    VGA_R,
    VGA_G,
```

```
    VGA_B : out std_logic_vector(9 downto 0) -- R=G=B[9:0]
    );

end component de2_vga_raster;
begin

  --Generate a 25 MHz clock for the VGA (clock divider)
  process (CLOCK_50)
  begin
    if rising_edge(CLOCK_50) then
        clk25 <= not clk25;
    end if;
  end process;


vga: de2_vga_raster port map (
    clk => clk25,
    address_out => address2,
    data_in => readdata2,
    rectangle_out => rectangle_from_vga,
    VGA_CLK => VGA_CLK,
    VGA_HS => VGA_HS,
    VGA_VS => VGA_VS,
    VGA_BLANK => VGA_BLANK,
    VGA_SYNC => VGA_SYNC,
    VGA_R => VGA_R,
    VGA_G => VGA_G,
    VGA_B => VGA_B
);

sram: de2_sram_controller port map (
    read => read,
    chipselect => chipselect,
    write => write,
    address  => address(17 downto 0),
    readdata => readdata_from_sram,
    writedata => writedata,
    byteenable => byteenable,
    address2 => address2,
    readdata2 => readdata2,

    go => go,

    SRAM_DQ => SRAM_DQ,
    SRAM_ADDR => SRAM_ADDR,
    SRAM_UB_N => SRAM_UB_N,
    SRAM_LB_N => SRAM_LB_N,
    SRAM_WE_N => SRAM_WE_N,
    SRAM_CE_N => SRAM_CE_N,
    SRAM_OE_N => SRAM_OE_N
  );

  --Check for go signal
```

```vhdl
  process(CLOCK_50)
  begin
  if rising_edge(CLOCK_50) then
    if address(19) = '1' then
      if write = '1' then
        go <= writedata(0);
      end if;
    end if;
    if address(20) = '1' then    --Are we inside the rectangle?
      if read = '1' then
        readdata <= "0000000000000000" & rectangle_from_vga;
      end if;
    end if;
  end if;
  end process;

end vga_sram_supercontroller_arch;
```

```
-------------------------------------------------------------------------------
--
-- Simple VGA raster display
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--          Larry Chen (lc2454)
--          Abhinandan Majumdar (am2993)
--          Shiva Ramudit (syr9)
--
-- Originally authored by Stephen A. Edwards (sedwards@cs.columbia.edu)
--
-- Last modified: 5-8-2008
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity de2_vga_raster is

  port (
    clk    : in std_logic;                -- Should be 25.125 MHz

    -- SRAM signals
    address_out : out std_logic_vector(17 downto 0);
    data_in     : in std_logic_vector(15 downto 0);

    -- Inside rectangle
    rectangle_out : out std_logic;

    VGA_CLK,                              -- Clock
    VGA_HS,                               -- H_SYNC
    VGA_VS,                               -- V_SYNC
    VGA_BLANK,                            -- BLANK
    VGA_SYNC : out std_logic;             -- SYNC
    VGA_R,
    VGA_G,
    VGA_B : out std_logic_vector(9 downto 0)  -- R=G=B[9:0]
    );

end de2_vga_raster;

architecture de2_vga_raster_arch of de2_vga_raster is

  -- Video parameters

  constant HTOTAL       : integer := 800;
  constant HSYNC        : integer := 96;
  constant HBACK_PORCH  : integer := 48;
  constant HACTIVE      : integer := 640;
  constant HFRONT_PORCH : integer := 16;
```

```
  constant VTOTAL       : integer := 525;
  constant VSYNC        : integer := 2;
  constant VBACK_PORCH  : integer := 33;
  constant VACTIVE      : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  --Positioning the frame at the top-left of the screen
  constant RECTANGLE_HSTART : integer := 0;
  constant RECTANGLE_HEND   : integer := 319;
  constant RECTANGLE_VSTART : integer := 0;
  constant RECTANGLE_VEND   : integer := 239;

constant header : integer := 539;


  -- Signals for the video controller
  signal Hcount : std_logic_vector(9 downto 0);   -- Horizontal position (0-800)
  signal Vcount : std_logic_vector(9 downto 0);   -- Vertical position (0-524)
  signal EndOfLine, EndOfField : std_logic;

  signal vga_hblank, vga_hsync,vga_vblank, vga_vsync : std_logic;   -- Sync. signals
  signal VGA_I : std_logic_vector(9 downto 0);   -- Grayscale intensity (=R=G=B)

  signal rectangle_h, rectangle_v, rectangle : std_logic;   -- rectangle area

  --RAM signals (to interface with the internally instantiated SRAM)
  signal read                        : std_logic;

  signal reset : std_logic := '0';

  signal row : integer := 0;
  signal col : integer := 0;
  signal addr : integer := 0;

  signal row_i, col_i : std_logic_vector(9 downto 0);
  signal data : std_logic_vector(7 downto 0);

begin
   -- Horizontal and vertical counters
  HCounter : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        Hcount <= (others => '0');
      elsif EndOfLine = '1' then
        Hcount <= (others => '0');
      else
        Hcount <= Hcount + 1;
      end if;
    end if;
  end process HCounter;
```

```
  EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

  VCounter: process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        Vcount <= (others => '0');
      elsif EndOfLine = '1' then
        if EndOfField = '1' then
          Vcount <= (others => '0');
        else
          Vcount <= Vcount + 1;
        end if;
      end if;
    end if;
  end process VCounter;

  EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

  -- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK
  HSyncGen : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' or EndOfLine = '1' then
        vga_hsync <= '1';
      elsif Hcount = HSYNC - 1 then
        vga_hsync <= '0';
      end if;
    end if;
  end process HSyncGen;

  HBlankGen : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        vga_hblank <= '1';
      elsif Hcount = HSYNC + HBACK_PORCH then
        vga_hblank <= '0';
      elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
        vga_hblank <= '1';
      end if;
    end if;
  end process HBlankGen;

  VSyncGen : process (clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        vga_vsync <= '1';
      elsif EndOfLine ='1' then
        if EndOfField = '1' then
          vga_vsync <= '1';
        elsif Vcount = VSYNC - 1 then
```

```
        vga_vsync <= '0';
      end if;
    end if;
  end if;
end process VSyncGen;

VBlankGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      vga_vblank <= '1';
    elsif EndOfLine = '1' then
      if Vcount = VSYNC + VBACK_PORCH - 1 then
        vga_vblank <= '0';
      elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
        vga_vblank <= '1';
      end if;
    end if;
  end if;
end process VBlankGen;

-- Rectangle generator
RectangleHGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' or Hcount = HSYNC + HBACK_PORCH + RECTANGLE_HSTART - 1 then
      rectangle_h <= '1';
    elsif Hcount = HSYNC + HBACK_PORCH + RECTANGLE_HEND - 1 then
      rectangle_h <= '0';
    end if;
  end if;
end process RectangleHGen;

RectangleVGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      rectangle_v <= '0';
    elsif EndOfLine = '1' then
      if Vcount = VSYNC + VBACK_PORCH - 1 + RECTANGLE_VSTART then
        rectangle_v <= '1';
      elsif Vcount = VSYNC + VBACK_PORCH - 1 + RECTANGLE_VEND then
        rectangle_v <= '0';
      end if;
    end if;
  end if;
end process RectangleVGen;

rectangle <= rectangle_h and rectangle_v;

-- Registered video signals going to the video DAC
row <= conv_integer(Hcount) - (HSYNC + HBACK_PORCH + RECTANGLE_HSTART);
row_i <= conv_std_logic_vector(row,10);
```

```
  col <= conv_integer(Vcount) - (VSYNC + VBACK_PORCH + RECTANGLE_VSTART);
  col_i <= conv_std_logic_vector(col,10);
  addr <= (col * ((RECTANGLE_HEND - RECTANGLE_HSTART + 1) / 2)) + conv_integer(row_i(9 downto 1
)) + header when rectangle = '1' else
          0;

  address_out <= conv_std_logic_vector(addr,18);

  data <= data_in(15 downto 8) when row_i(0) = '0' else
          data_in(7 downto 0) when row_i(0) = '1' else
          (others => '0');

  VideoOut: process (clk, reset)

  begin
    if reset = '1' then
      VGA_R <= "0000000000";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";
    elsif clk'event and clk = '1' then
      if rectangle = '1' then
        VGA_R <= data & "00";
        VGA_G <= data & "00";
        VGA_B <= data & "00";
--        end if;
      elsif vga_hblank = '0' and vga_vblank ='0' then
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      else
        VGA_R <= "0000000000";
        VGA_G <= "0000000000";
        VGA_B <= "0000000000";
      end if;
    end if;
  end process VideoOut;


  VGA_CLK <= clk;
  VGA_HS <= not vga_hsync;
  VGA_VS <= not vga_vsync;
  VGA_SYNC <= '0';
  VGA_BLANK <= not (vga_hsync or vga_vsync);

  --There's some slack on the right-edge and bottom edge of the screen
  --This is to prevent Nios from attempting a 128-bit block into SRAM at the very right or bottom of the screen
  -- and end up using the SRAM when the VGA wanders back into the image frame
  rectangle_out <= '1' when ((col_i > RECTANGLE_VEND) or ((row_i > RECTANGLE_HEND) and (row_i <
(RECTANGLE_HEND + 256)))) else
                   '0';

end de2_vga_raster_arch;
```

```
-----------------------------------------------------------------------------
--
-- SRAM Controller
--   This is, for the most part, taken from one of the labs
-- Written for 128-bit AES decryption project
-- Course: CSEE 4840 - Embedded System Design, Spring 2008
-- Authors: Shrivathsa Bhargav (sb2784)
--                  Larry Chen (lc2454)
--                  Abhinandan Majumdar (am2993)
--                  Shiva Ramudit (syr9)
--
-- Last modified: 5-8-2008
-----------------------------------------------------------------------------

library ieee;

use ieee.std_logic_1164.all;

entity de2_sram_controller is

  port (
    signal chipselect : in std_logic;
    signal write, read : in std_logic;
    signal address  :  in std_logic_vector(17 downto 0);
    signal readdata : out std_logic_vector(15 downto 0);
    signal writedata : in std_logic_vector(15 downto 0);
    signal byteenable : in std_logic_vector(1 downto 0);

    --From VGA
    signal address2    :  in std_logic_vector(17 downto 0);
    signal readdata2   : out std_logic_vector(15 downto 0);

    signal go           : in std_logic;

    signal SRAM_DQ   : inout std_logic_vector(15 downto 0);
    signal SRAM_ADDR : out std_logic_vector(17 downto 0);
    signal SRAM_UB_N, SRAM_LB_N : out std_logic;
    signal SRAM_WE_N, SRAM_CE_N : out std_logic;
    signal SRAM_OE_N            : out std_logic
    );

end de2_sram_controller;

architecture dp of de2_sram_controller is
begin

  --All these when-else pairs are how the VGA_GO/NO Mux is implemented

  SRAM_DQ <= writedata when write = '1' and go = '0' else
                      (others => 'Z');
  readdata <= SRAM_DQ;
  readdata2 <= SRAM_DQ when go = '1'; --else
              --(others => '0');
```

```
  SRAM_ADDR <= address2 when go = '1' else
              address;

  SRAM_UB_N <= '0' when go = '1' else not byteenable(1);

  SRAM_LB_N <= '0' when go = '1' else not byteenable(0);

  SRAM_WE_N <= '1' when go = '1' else not write;

  SRAM_CE_N <= '0' when go = '1' else not chipselect;

  SRAM_OE_N <= '0' when go = '1' else not read;

end dp;
```