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.

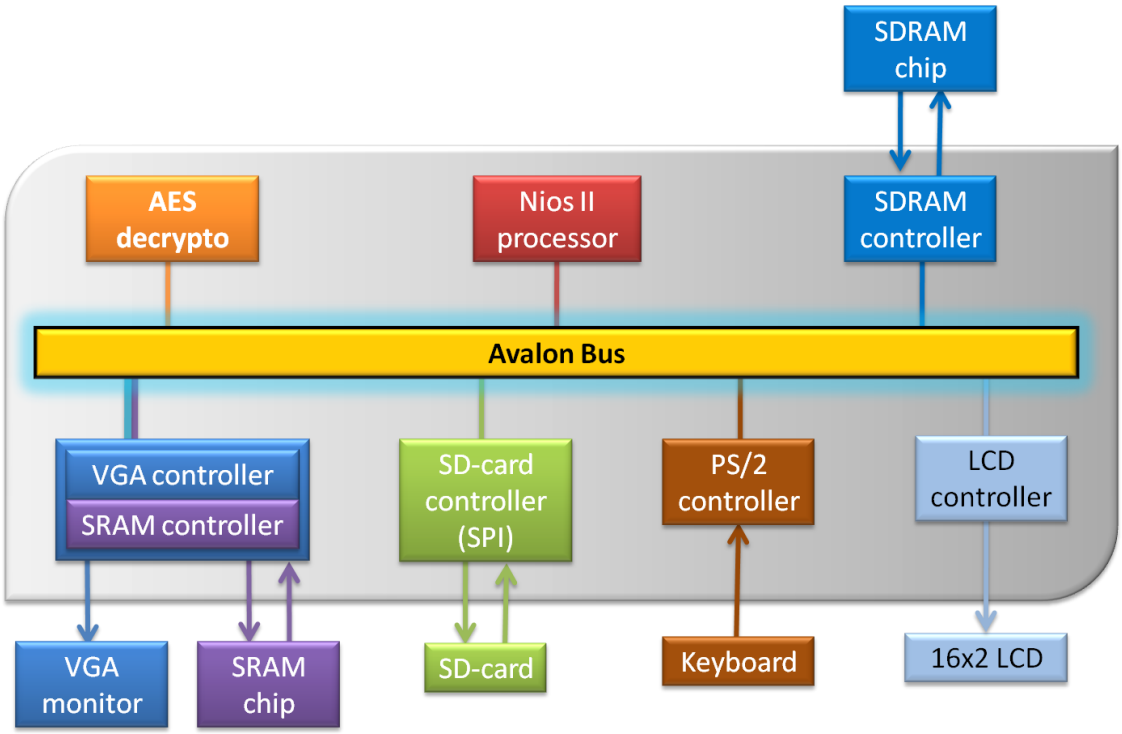

Figure 1 - The block diagram of the entire system. The gray boundary represents the blocks within the FPGA. Arrows show data flow; lines show hard connections between modules.

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.

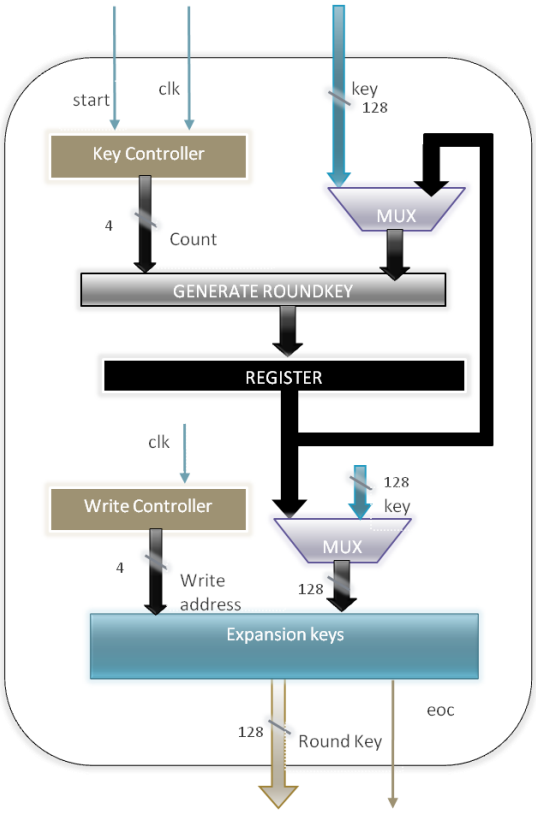

Figure 2 - AES key expansion

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)

cf	→	8a
4f		84
3c		eb
09		01

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.

Rcon=

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

2b	⊕	8a	⊕	01	=	a0
7e		84		00		fa
15		eb		00		fe
16		01		00		17

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.

28	⊕	a0	=	88
ae		fa		54
d2		fe		2c
a6		17		b1

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

2b	28	ab	09	→	a0	88	23	2a
7e	ae	f7	cf		fa	54	a3	6c
15	d2	15	4f		fe	2c	39	76
16	a6	88	3c		17	b1	39	05

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.

1A	A4	95	3E		D0	C9	E1	B6		CA	6D	74	88
A9	B9	4C	3A	+	14	EE	3F	63	=	BD	57	73	59
13	CD	78	27		F9	25	0C	0C		EA	E8	74	2B
86	F1	33	A8		A8	89	C8	A6		2E	78	FB	0E

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

CA	6D	74	88		CA	6D	74	88
BD	57	73	59	->	59	BD	57	73
EA	E8	74	2B		74	2B	EA	E8
2E	78	FB	0E		78	FB	0E	2E

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.

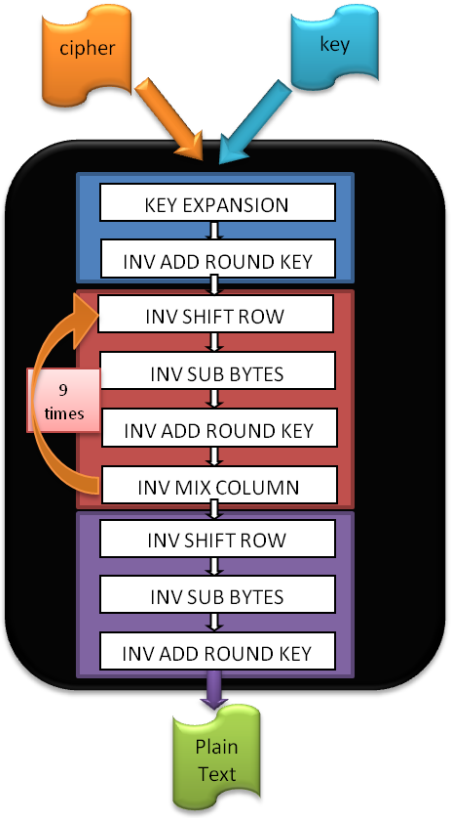

Figure 3 - AES 128-bit Decryption Algorithm

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.

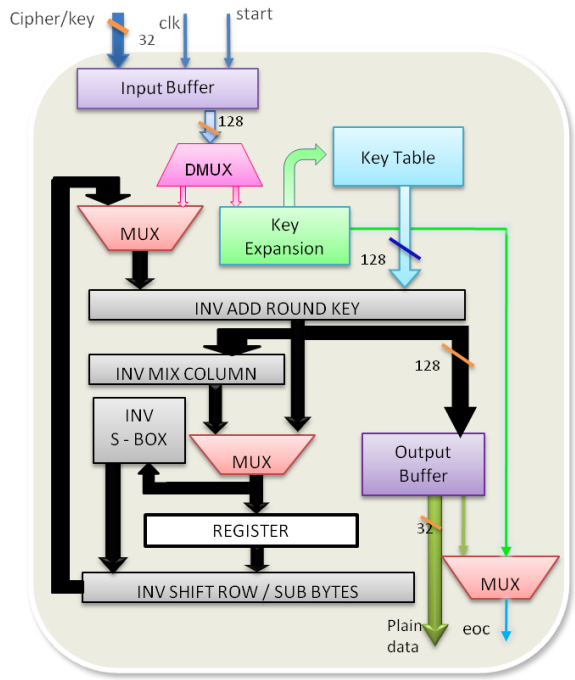

Figure 4 - AES 128-bit Decrypto Datapath

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 illustrated in figures 5 and 6.

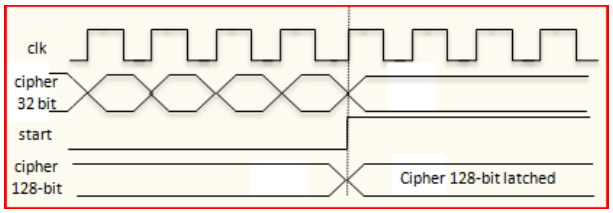

Figure 5 - Timing of Input Data Buffering

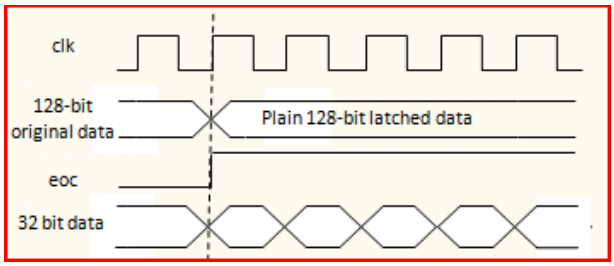

Figure 6 - Timing of final data traversal

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.

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

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

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

Figure 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.

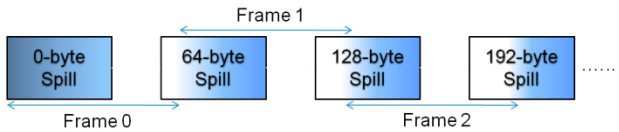

Figure 9 - SPI buffer spilling

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).

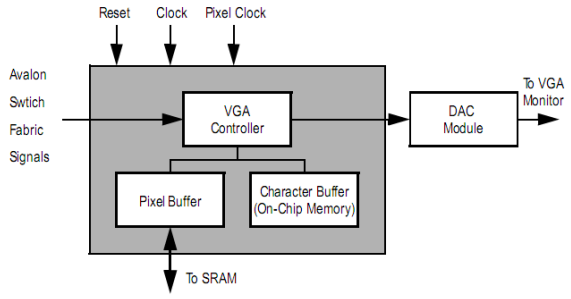

Figure 10 - Block diagram of the VGA controller

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.

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

The address calculation is illustrated below.

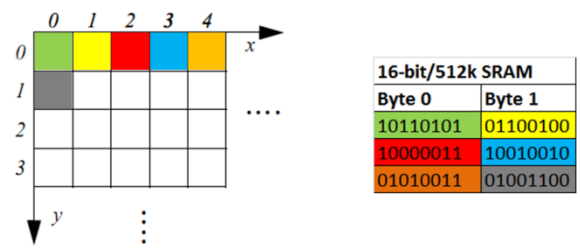

Figure 11 - VGA pixel addressing and SRAM retrieval

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.