AES File Encryption and Decryption

Group Report – ENSC 452 (Advanced Digital System Design)

Jason Tsang and Daniel Dixon

Table of Contents

Revision History

Version	Status	Published/ Revision Date	Authors
1.0	Created	2018/04/09	Daniel Dixon Jason Tsang

1 Introduction

In this modern digital world, an increasing amount of our personal data is being transferred online. As a result, the need for securing that data is as important as ever. Personal banking, tax returns, photo storage, and more have found their way onto cloud platforms. This consequently creates large targets for malicious individuals to intercept this data.

Many have turned to cloud based file hosting services such as Dropbox to save their digital content. However, at what level can we trust hosting businesses to secure our content and commit to its privacy? In some ways, that responsibility falls back to the us.

As a result, we (Jason and Daniel) have dipped into the world of cryptography and implemented a file encryption and decryption system to secure important files. By creating an onboard SD card, and external Ethernet interface on a ZedBoard, one can quickly encrypt files before uploading it to services such as Dropbox.

2 Background

Encryption is a process which protects information or data from unauthorized access through encoding. A specification for this encoding is Advanced Encryption Standard (AES), developed in 2001 by the U.S. National Institute of Standards and Technology (NIST). Originally named Rijndael, AES is now one of the most widely adopted methodologies, thanks to its use by the U.S. government to protect classified data. As such, AES is the same methodology used in this project [1].

AES Encryption uses both confusion and diffusion techniques to cipher its data. This technique results in a file's bytes being substituted, permutated and linearly transformed: thus, increasing the difficulty for it to be deciphered. At any given stage in the encryption process, AES works on a fixed block size of 16 bytes (128 bits), represented in a 4x4 column-major matrix. This matrix is termed the state. The algorithm for a 128-bit AES encryption key is listed below [2]. The steps outline the ECB (Electronic Codebook) block cipher mode of operation. One iteration of this list represents the encryption of a 16 bytes chunk of a file.

128-bit AES-ECB Encryption Algorithm Steps:

1. RoundKeyExpansions
2. AddRoundKey
3. Rounds (10 times)
    1. SubBytes
    2. ShiftRows
    3. MixColumns
    4. AddRoundKey
4. Final Round
    1. SubBytes
    2. ShiftRows
    3. AddRoundKey

In the RoundKeyExpansion procedure, round-keys (round-specific keys derived from the initial encryption key) are created using the Rijndael key schedule, a calculation algorithm. In the instance of 128-bit AES, 11 round-keys are created. Next, AddRoundKey is ran once, where the first round-key is bitwise XOR with the state. Now 10 rounds of SubByte, ShiftRows, MixColumns, and AddRoundKey are processed. In SubByte, a Rijndael S-box (lookup table) is used to substitute each byte with another. This provides the confusion and non-linear aspects of AES. Following that procedure, ShiftRows cyclically shifts the four rows of the matrix to the left by offsets of their row number. This step contributes to the diffusion and linear mixing of the bytes over 10 rounds. More diffusion is then carried out by MixColumns, where a fixed matrix is multiplied against the state. The round then ends with AddRoundKey. After 10 iterations, the algorithm ends with a final round processed without the MixColumns procedure. In the end an encrypted 16 bytes chunk is produced, and the algorithm moves onto the next chunk of the file [2].

The procedure is similar to decrypt the file, with the stages transformed with some inverted versions of the steps used for encryption. The ECB decryption steps are outlined below [2].

128-bit AES-ECB Decryption Algorithm Steps:

1. RoundKeyExpansions
2. AddRoundKey
3. Rounds (10 times)
    1. InverseShiftRows
    2. InverseSubBytes
    3. InverseMixColumns
    4. AddRoundKey
4. Final Round
    1. InverseShiftRows
    2. InverseSubBytes
    3. AddRoundKey

A disadvantage of ECB is the lack of diffusion. This results in identical blocks being transformed into identical encrypted blocks, and thus potentially revealing patterns within data. As a result, CBC (Cipher Block Chaining) is an alternative block cipher mode of operation that helps solve this diffusion issue. In CBC, each input state is XOR'd with the previous encrypted state before being encrypted [3]. Hence, each encrypted state depends upon all the previous input states processed up to that point. A drawback to this mode however, is the lack of ability for parallelized encryption. Furthermore, it requires an additional "key" in the form of an initialization vector for the first state. The decryption of CBC acts in the same manner as ECB, though, before the original state is retrieved, it is XOR'd with the previous encrypted state.

3 System Overview

This section provides an overview of the final implemented system.

3.1 System Block Diagram

Figure 3.1-1 describes the overall architecture of our ZedBoard AES system.

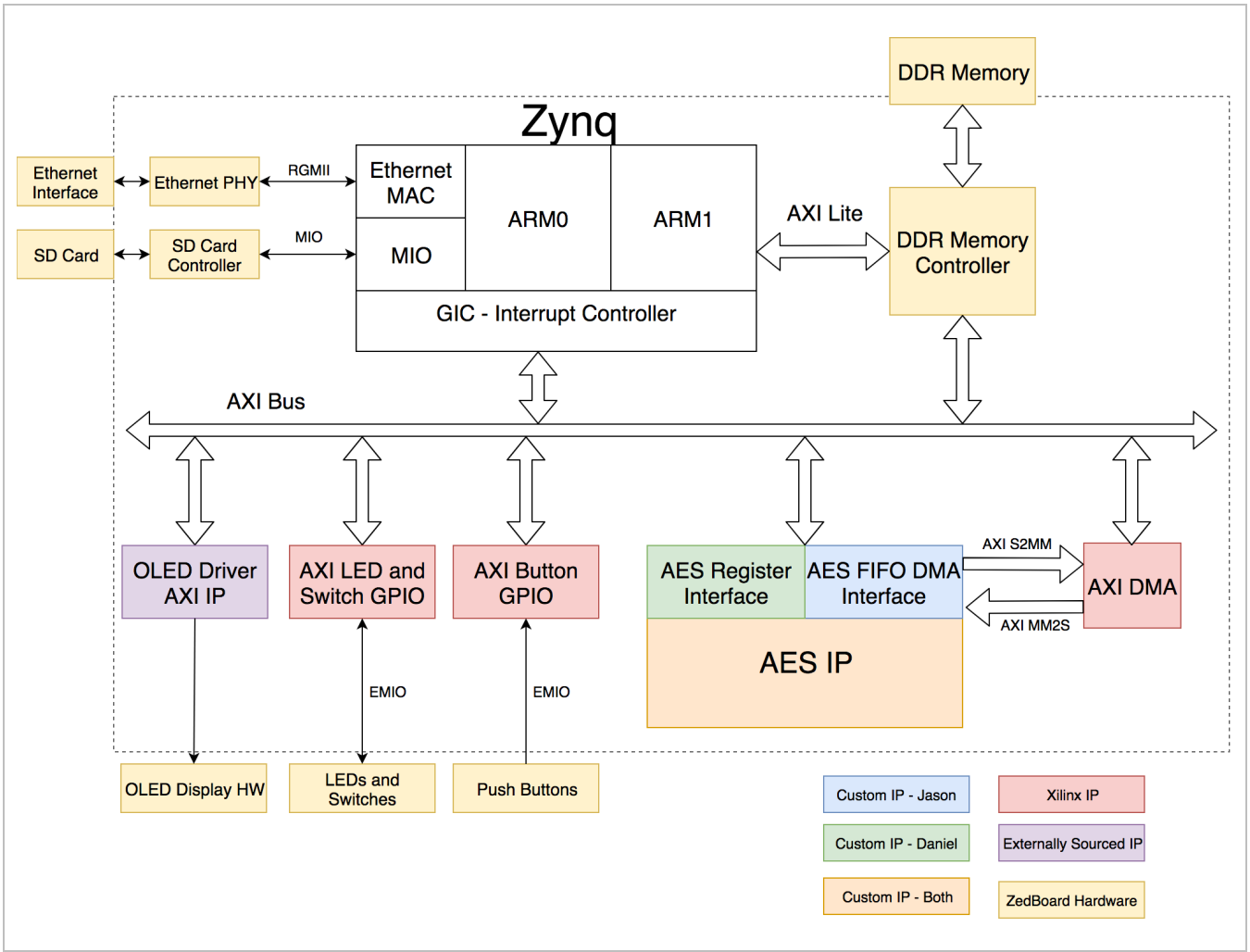

Figure 3.1-1: System Block Design

Both the Ethernet and SD card interfaces are routed directly to the Zynq SoC (System on Chip) through the MIO (Multipurpose IO), and does not go through the PL (Programmable Logic). This enabled us to directly use Xilinx libraries at the PS (Processor System) level, allowing for easier integration of those complex interfaces.

The *OLED Driver AXI IP* was externally sourced from Ali Aljaani from the Texas A&M University at Qatar [4]. It provides user friendly software drivers to print ASCII characters to the OLED display.

A majority of our custom IP is contained within the AES IP block, which is comprised of many hardware blocks connected together by a state machine. This IP block is unique as it contains two interfaces: one interface to initialize the IP block by setting the key and mode (encrypt or decrypt), an another to stream the state via the AXI DMA protocol. This design was chosen as the key and mode stays constant for every 16 bytes chuck of data it processes. By setting registers once, it reduces the overhead required in sorting and streaming content to the custom IP.

3.2 Custom AES IP Block Diagram

The implementation of our custom AES IP block is detailed in Figure 3.2-1, where “Key In” and “Mode In” are set by registers, and “State In” is streamed by the AXI DMA. This block implements the ECB mode of operation.

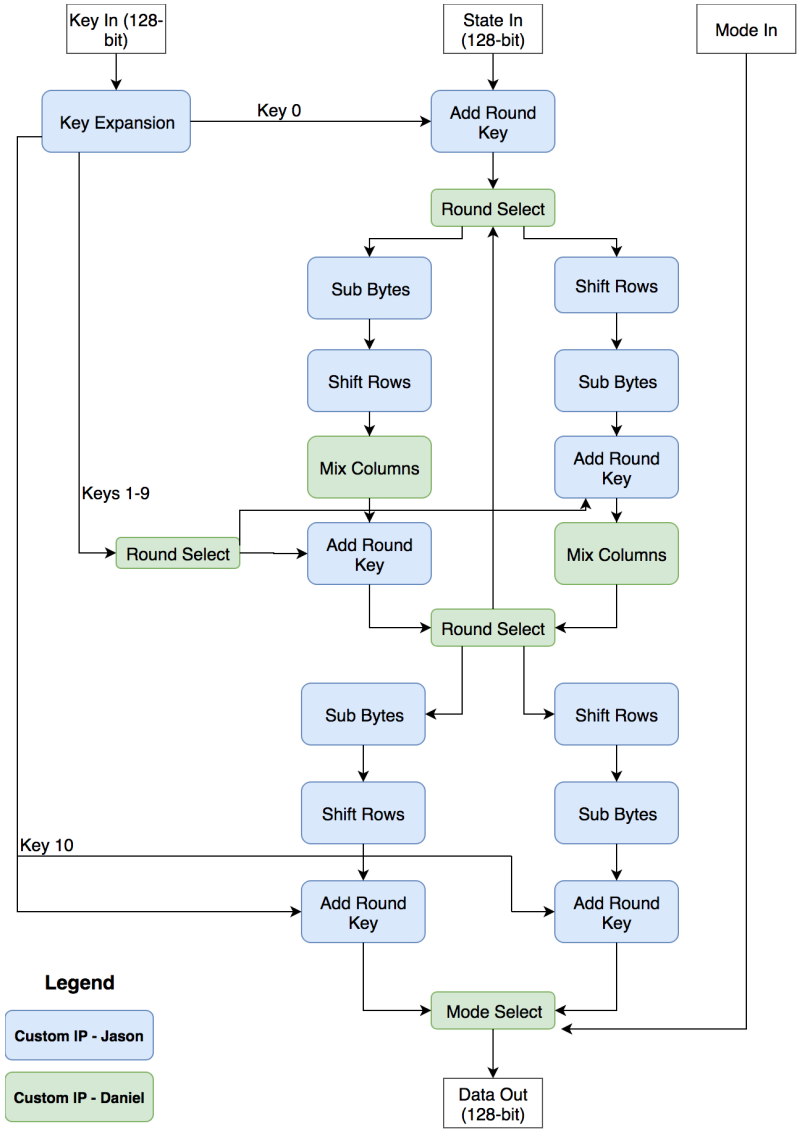

Figure 3.2-1: AES IP Block Diagram/ Algorithm Flow Chart

During the development process, each of the separate steps of the AES algorithm were designed as separate AXI4 IP blocks: using registers for its interface. This method was used as a way of validating each component within the AES algorithm. By using a known-good software implementation of the AES algorithm, *Tiny AES in C* [5], each hardware step was checked for byte-exactness on the ZedBoard at runtime.

After all of the components were completed, a state machine was designed to handle the flow of data that the software component had been performing in the early development stages. Finally, the entire custom AES IP block was compared against the full software implementation of the ECB mode of operation. Utilizing this methodology helped separate implementation and integration issues, allowing for full confidence in the final block design.

The CBC mode of operation was fulfilled in the software side. Although it is more efficient and quicker to process the additional XOR step in hardware, the additional complexity required in the given time frame of this project meant that there was no time remainin for that optimization.

4 User Manual

The following sections details instructions on how to operate the ZedBoard AES system, both by using the onboard SD card interface, and external Ethernet connected mode. Figure 4-1 below outlines the physical inputs and outputs a user will interact with and will be used as reference in this user manual.

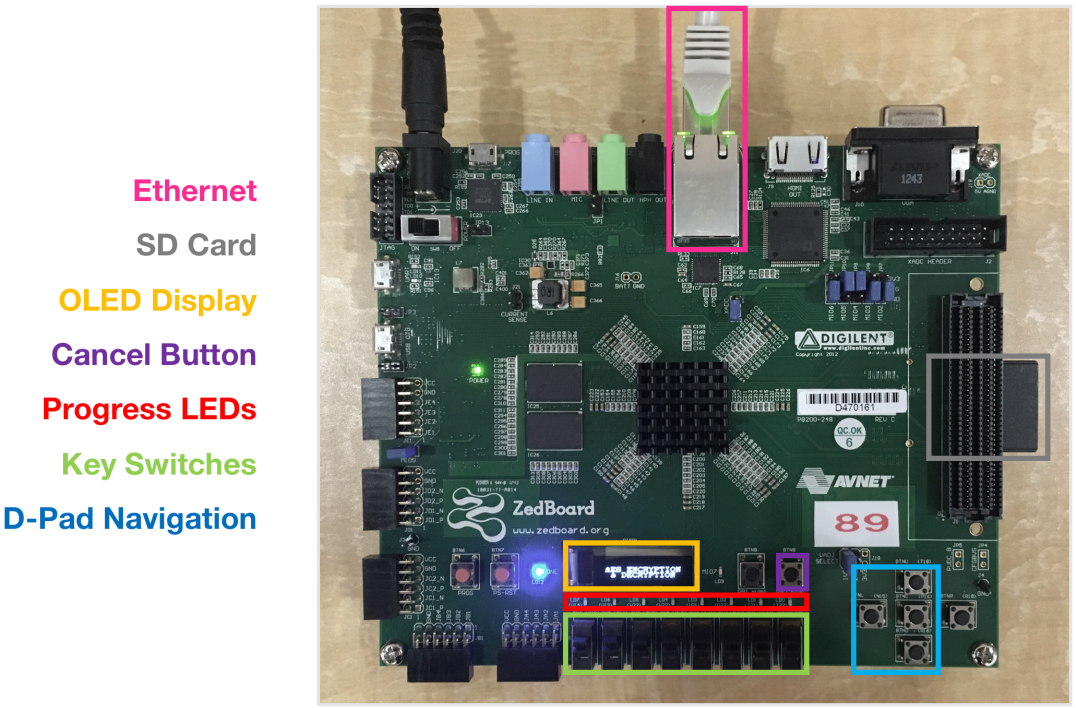

Figure 4-1: ZedBoard User I/O

4.1 Onboard SD Card Interface

The onboard SD Card Interface provides a quick method of encrypting files off an SD card. Referring to Figure 4-1: ZedBoard User I/O, the OLED display shows the navigation menu as shown in Figure 4.1-1. Using the D-Pad buttons, you can navigate up and down the menu with the up and down buttons. Pressing the center button will select the item in the list with a '*' next to it. Pressing the left D-pad button sends you back to the previous screen. The Menu choices available are listed below, with their corresponding child menus, prompts and confirmations.

Main Menu Structure:

1. ECB Mode
    - Encrypt File
        - File Selection
            - Setting Key Prompt
                - Action confirmation
    - Decrypt File
        - File Selection
            - Setting Key Prompt
                - Action confirmation
2. CBC Mode
    - Encrypt File
        - File Selection
            - Setting Key Prompt
                - Action confirmation
    - Decrypt File
        - File Selection
            - Setting Key Prompt
                - Action confirmation
3. Ethernet Mode
4. Reformat SD
5. Quit

Figure 4.1-1 shows the Main Menu of the interface. During the File Selection stage, a list of files from the base directory of the SD card is shown, as seen in Figure 4.1-2. The searching algorithm was ported from FatFS Module from ELM by ChaN [6], the same open-source library used by Xilffs [7], but unknowingly missing in their implementation.

Figure 4.1-1: OLED Main Menu

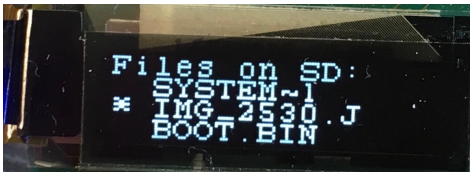

Figure 4.1-2: SD File Selection

Figure 4.1-3 shows the Action Confirmation shown before starting the encryption process. A similar confirmation is shown for decryption. Likewise, a Setting Key Prompt will occur before this screen to allow time for you to configure the 8 Key Switches in setting your cipher key. Figure 4.1-4 shows the "Done!" Screen when the file processing is complete.

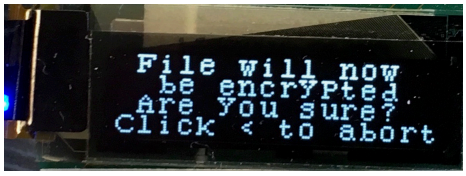

Figure 4.1-3: SD Action Confirmation

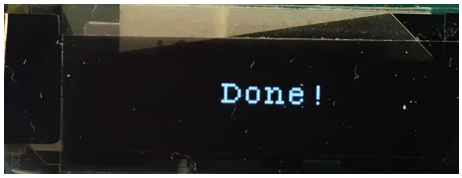

Figure 4.1-4: SD Done Response

During the cipher process, the OLED will display the approximate time remaining in seconds, as well as the approximate percentage completed. At the same time, the LED strip will loop taking 60 second to complete a loop. This SD Progress Monitor is shown in Figure 4.1-5, and is kicked started on CPU1 from CPU0 when the cipher process is handed over to the custom AES IP block.

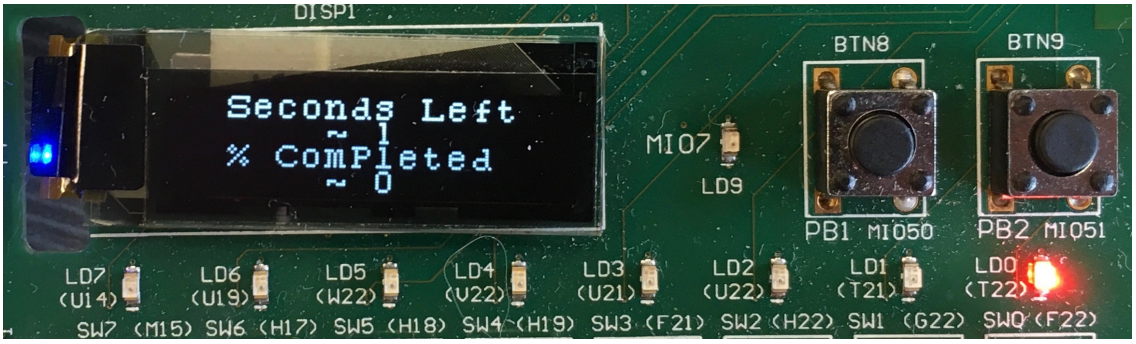

Figure 4.1-5: SD Progress Monitor

During the processing of a file, you may press the cancel button to abort the operation. As processes output isn't written till the end, the original file will remain unchanged.

A reformat SD option is given in the Main Menu if you would like to erase all the contents and format the SD card to FatFS. Note that if you are using the boot.bin to start the ZedBoard, that image will no longer exist. Similarly, if you encrypt that image without decrypting before restarting the board, that start image will also be lost.

4.2 External Ethernet Mode

Using the OLED and button interfaced described in Section 4.1, the user may select "Ethernet Mode" from the main menu as shown in Figure 4.2-1. Once the Ethernet hardware is initialized on the ZedBoard, the Ethernet routine will start. The ZedBoard must be connected to a network that has an active DHCP server in order for it to work. Ethernet mode uses DHCP to dynamically initialize an IP

address. That IP address is then printed to the OLED screen, as well as the static port set that the AES server listens for connections on. Figure 4.2-2 shows the OLED screen displaying the dynamic IP address and port for the Ethernet Mode.

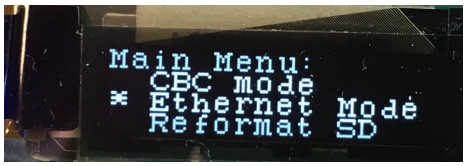

Figure 4.2-1: Ethernet Mode

Figure 4.2-2: Ethernet IP and Port Display

Any client that supports the TCP/IP protocol can connect to the ZedBoard AES system. Once a connection has been established, the host device may send a file at any time. If the ZedBoard AES system is not ready, the connection will be refused, and no file transfer will occur. Currently, the system is limited to one connection at a time, though multiple connection capability could be added in the future.

The file being transferred requires a header to be added by the host before the data is sent. The transmission format is outlined in Table 4.2-1.

Table 4.2-1: Ethernet Mode File Transmission Format

Byte	Data Type	Description
0x000000 - 0x000003	uint32_t	Length of data, including mode switch, excluding the length header itself
0x000004 - 0x000007	uint32_t	Mode switch, set to zero for encryption, non-zero for decryption
0x000007 - 0x2FAF10 (Max)	Raw byte array	100MB maximum byte array of data to be encrypted or decrypted

The data will then be processed on the ZedBoard, and immediately return to the host without any further host intervention once completed. The received format from the ZedBoard to the host is outlined in Table 4.2-2.

Table 4.2-2: Ethernet Mode File Receive Format

Byte	Data Type	Description
0x000000 - 0x000003	uint32_t	Length of data to be received by host
0x000004 - 0x2FAF0C (Max)	Raw byte array	100MB maximum byte array of encrypted or decrypted data

5 Outcome

The ZedBoard AES system functions as the original design intended. However, compared to the original proposal, we have created two interfaces for the system. Using the SD card for local interactions, and the Ethernet for external. Furthermore, the ZedBoard AES is a completely standalone system. By packaging a binary boot image on the SD card, the ZedBoard has the ability to boot with only power connected. A user can then navigate the device using the D-pad and OLED display. The system's ability of being network enabled over Ethernet allows flexibility in more applications, while leveraging the same AES encryption/decryption infrastructure as the SD card mode.

In terms of accurate encryption and decryption, hundreds of megabytes of data have been tested over both the SD and Ethernet interfaces. The original data has been compared to its encrypted and then decrypted counterparts and checked for byte-exactness. In SD card mode, we have a 100% accuracy rate with the AES process. However, in Ethernet mode, we have only achieved 99.9996% accuracy, meaning a few bytes out of many millions are incorrectly encrypted or decrypted. This is a known issue that requires further investigation, but in the tight timeframe of this project, could not source the root-cause.

The performance gain of using hardware acceleration over a pure software implementation has been measured to be around 9.8 times faster when the ARM CPU us clocked at 200MHz. While this gain in performance is substantial, it could be improved even further. Currently, the implementation uses the DMA to transfer 4 words at a time, wait for 4 words to be received back by the PS, and then continue to the next 4 words of data. This requires many calls to the DMA transfer function and is the major bottleneck of our system. Given more time, we would ideally implement a true streaming interface, where the DMA would be commanded to continually stream all of the file instead of in 16 bytes segments in the current implementation. However, at its current state, it takes around 4 seconds to encrypt or decrypt 10MB, which is already a respectable throughput. The maximum file size able to be processed on the system is 100MB, with this size being arbitrarily selected. Regardless, the entire application can be at most 512MB: the maximum DRR memory available on the ZedBoard.

From the implementation, the total on-chip power is 1.643W at junction temperatures of 43.9°C. Dynamic on-chip power accounts of 91% of the total on-chip power. From that dynamic on-chip power, 90% of it is being utilized by PS7 (1.378W). Taking a look at timing, the worse negative slack is located within the custom AES IP block, at -29.877ns. This slack is between the input key registers and the output of the key expansion. There is no concern for timing violations because this is a purely static combinational logic portion of the IP block that, it does not change during the AES decryption/encryption process. Once the key is set, the outputs of the key expansion will be constant, and timing violations related to this critical path will be irrelevant.

An additional improvement would be in Ethernet communication. Currently there is no feedback given by the board to the host during AES processing. As a result, the host program has no knowledge on the progress of the AES encryption or decryption, and thus, must poll on the receiver port until data starts being streamed back. Adding progress feedback over Ethernet would help extend the interface to be friendlier to users and developers. However, with the current implementation, there is the possibility to expand the Ethernet interface outside of just Python, so long as TCP/IP protocols are followed.

6 Description of IP Blocks

The following section details more information about each of the IP blocks used in this project.

6.1 AXI LED and Switch GPIO

The AXI LED and Switch GPIO were generated automatically using the AXI GPIO 2.0 IP Customizer. Under the Board Interface Configuration option, there is the ability to select predefined board interfaces. In this instance “leds 8bits” and “sws 8bits” were selected, and map to the Progress LEDs and Key Switches shown in Figure 4-1: ZedBoard User I/O. A maximum of two GPIOs are possible per AXI GPIO block. The first GPIO is addressed by the upper 2 bytes, and the second is addressed by the lower 2 bytes within a GPIO API call in the SDK.

6.2 AXI Button GPIO

Similar to 6.1, the AXI Button GPIO was created using the AXI GPIO 2.0 IP Customizer, and is a predefined board interface called “btns 5bits”. They are mapped to the D-pad Navigation shown in Figure 4-1: ZedBoard User I/O.

6.3 AXI OLED Driver

The AXI OLED Driver is sourced from Ali Aljaani from the Texas A&M University at Qatar [4]. Written in Verilog, it come pre-packaged and instantly usable when added to the IP Repository. Constraints must be defined for its external pins and are found in the Quick Start guide provided with that driver. Wrappers were created in the SDK to more easily interact with their OLED API. For instance, a function to handle scrollable menus.

6.4 AES IP Block

The AES IP Block is the core custom hardware of our system. It contains all of the logic necessary to either encrypt or decrypt a 16 bytes state with a given 128-bit key and mode. There is both a streaming and register interface for data processing and configuration respectively.

6.4.1 Structure

The AES IP Block was created from a large subset of self-contained components, each representing a step in the AES encryption/decryption algorithm. In the file structure, and throughout the VHDL code, these functions remain self-contained, and are held together by a higher-level state machine. In our design, both encryption and decryption are performed in parallel, and the final data output depends on the mode switch selected. The hardware performs one round of encryption per cycle, takes in 32-bits of streaming data per cycle, and outputs 32-bits of streaming data per cycle. Therefore, it takes 4 clock cycles to read in and send out each 16 bytes state processed. This gives a total of 18 clock cycles to encrypt or decrypt 16 bytes of data. There are many implementations of AES on FPGA online. In our implementation, the AES algorithm, *Tiny AES in C* [5] was used as our primary reference. Abdeladim Sadiki’s and Jevin Sweval’s implementations were also used as secondary references in our component design [8] [9].

6.4.2 Interface

As mentioned earlier, there are two interfaces to the AES IP Block, an AXI DMA streaming interface for the data, and register interface to the initialization.

The streaming interface follows the AXI DMA protocol, with 32-bit wide data, and signal registers to start and end transfers. The AES state machine controls the “ready” signals that the DMA protocol uses to send and receive data. An entire 16 bytes state must be encrypted or decrypted before the next state of data can be processed. In the current implementation, the IP block must be reset after completion of each state in order to return it to the initial step in the state machine.

The register interface is designed for initialization. Table 6.4.2-1 describes the 9 registers created for this custom AES block. Each register contains a width of 32-bits.

Table 6.4.2-1: AES IP Block Initialization Registers

Register Address Offset	**Description**
0x0	AES Mode, set to zero for encryption, non-zero for decryption
0x1-0x4	Unused
0x5-0x8	128-bit AES Key, starting from lowest offset to highest offset

Once these registers have been set, the AES IP Block is ready to receive and process data over the streaming interface.

6.5 Test Methodology

The testing of the AES IP block is outlined within 3.2 Custom AES IP Block Diagram. For the overall test methodology of every IP block, temporary user-facing test benches were created in the SDK. This gave us an interactive demo for each milestone and the confidence that each interface operates as expected. Thus, during design integration phases, if problems were to arise, we would be sure that the issue did not reside within the IP block, but with the way those IP blocks interacted with one another. This resulted in less time spent debugging integration issues. Xilinx sample projects were used that the primary base for temporary test benches we created. For the AXI LED, Switches, and Button GPIO, the Zynq-7000 All Programmable SoC: Embedded Design Tutorial [10] was used. For the SD card and Ethernet interface, the example xilffs_polled_example and lwip_echo_server SDK projects were used respectively [11] [12].

7 Description of Design Tree

Git has been used as our source control method, with our repository being hosted on Github. The repository can be found at the following link: https://github.com/jasonrtsang/zedboard_aes, and contains the directory structure outlined in Table 7-1.

Table 7-1: zedboard_aes Repository Structure

File/ Directory	Description
./ip_repo	Directory holding custom IP blocks
./sdk	Directory containing application and board support packages
./src	Directory containing files built by build.tcl (including constraints, block diagram, and system wrapper)
.gitignore	Files and directories to be ignored by git
README.md	Readme file containing information on the repository
build.bat	Launcher for build.tcl
build.tcl	Generated from "Write Project to tcl," builds the Vivado project

This repository requires a Windows machine running Vivado 2017.3 and Xilinx SDK 2017.3.

7.1 Steps to Build and Run Repository

1. Run build.bat
2. Open zedboard_aes.xpr
3. Generate Block Design
4. Generate Bitstream and Export Hardware including Bitstream
5. Launch SDK
6. Import all projects from ./sdk into SDK as "Existing Projects"

References

[1] S. Seleborg, "About AES – Advanced Encryption Standard," [Online]. Available: https://www.axantum.com/AxCrypt/etc/About-AES.pdf.

[2] F. I. P. S. P. 197, "ADVANCED ENCRYPTION STANDARD (AES)," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf.

[3] C. H. W. M. J. L. S. W. L. T. William F. Ehrsam, "Message verification and transmission error detection by block chaining". US Patent 4074066, 1976.

[4] A. Aljaani, "ZedboardOLED-v1.0-IP," [Online]. Available: https://github.com/ama142/ZedboardOLED-v1.0-IP.

[5] kokke, "Tiny AES in C," [Online]. Available: https://github.com/kokke/tiny-AES-c.

[6] E. b. ChaN, "f_findfirst," [Online]. Available: http://elm-chan.org/fsw/ff/doc/findfirst.html.

[7] Xilinx, "xilffs," [Online]. Available: http://www.wiki.xilinx.com/xilffs.

[8] A. Sadiki, "A VHDL implementation of the AES algorithm," [Online]. Available: https://github.com/AbdeladimSadiki/AES-VHDL.

[9] J. Sweval, "A VHDL implementation of 128 bit AES encryption with a PCIe interface," [Online]. Available: https://github.com/jevinskie/aes-over-pcie.

[10] Xilinx, "Zynq-7000 All Programmable SoC: Embedded Design Tutorial UG1165," [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1165-zynq-embedded-design-tutorial.pdf.

[11] Xilinx, "xilffs_polled_example.c," [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilffs/examples/xilffs_polled_example.c.

[12] Xilinx, "lwip_echo_server," [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/master/lib/sw_apps/lwip_echo_server.