Text

Description automatically generated with medium confidence

ENGG\*4550 Project – AES Coprocessor in CBC Mode

Pietro Baggetta (1092160), Owen Walsh (1035794)

April 6, 2023

Lab TA: Marc Baucas

Professor: Radu Muresan

Table of Contents

[Introduction 3](#_Toc131752175)

[Background 3](#_Toc131752176)

[Design Methodology 4](#_Toc131752177)

[Test Results 6](#_Toc131752178)

[Conclusions 7](#_Toc131752179)

[References 8](#_Toc131752180)

[Appendix 8](#_Toc131752181)

Table of Figures

[Figure 1 - CBC Mode Flow Diagram (Dworkin, 2001) 4](#_Toc131345515)

[Figure 2 - Platform Designer Signals & Interface Creation Window of APB IP Module 4](#_Toc131345516)

[Figure 3 - Platform Designer System Contents Window Showing the APB IP Module Connections 5](#_Toc131345517)

[Figure 4 - 128-bit CBC Mode Test Input (Dworkin, 2001) 6](#_Toc131345518)

[Figure 5 - Terminal Interface During Test 7](#_Toc131345519)

Table of Appendix Figures

[Appendix Figure 1 - AES\_IP (APB) Code Lines 1 to 60 8](#_Toc131345520)

[Appendix Figure 2 - AES\_IP (APB) Code Lines 60 to 109 9](#_Toc131345521)

[Appendix Figure 3 - AES\_IP (APB) Code Lines 109 to 155 9](#_Toc131345522)

[Appendix Figure 4 - C Code Implementation for Reading and Writing to Hardware 10](#_Toc131345523)

# Introduction

In this lab we are tasked to create an Advanced Encryption Standard coprocessor in CBC mode using the DE1-SoC board. To implement the AES coprocessor onto our FPGA development board, we had to implement a simple APB IP module to interface between our software and hardware. After completion of the APB IP module, our hardware from the DE1-SoC board will be able to communicate to the software on the board and read in values that are entered through the terminal. The understanding of how the hardware and software on an FPGA board communicate are important towards our understanding of how systems work with one another. Understanding this complex project implemented on the SoC board allows us to build and use design techniques learned in our courses and previous labs and working with a memory-mapping coprocessor module enables us to implement virtually any implementation we choose.

# Background

Some of the different methods that were used throughout this project in order to implement the AES coprocessor onto the FPGA board are state machines, memory mapping, address referencing and CBC mode support. Our program uses state machines within the encryption algorithm itself. The encryption algorithm used in our implementation was given to us by professor Radu Muresan, but we had the choice of creating our own or using one that can be found online. The encryption algorithm that he had implemented uses state machines to create the different rounds of encryption within one encryption block (Muresan, ENGG 4560/6600 Laboratory Manual, 2022). State machines are useful with one state can lead to another like in our case where the output of the previous round is the input to the next. One of the disadvantages to our state machine in particular is that our algorithm is complex and none of the states run in parallel with one another meaning the execution is slow for just one encryption block.

Our memory mapping uses the HPS-FPGA as our base for the main connections to the board which will interface with the software. The HPS is wired and implemented in the *Platform Designer* within the Quartus Prime software IDE. Memory mapping has many different advantages for our use case but the main one is that we will be able to communicate between the hardware and software of the board in order to store values and access memory addresses. We use the HPS memory mapping module to refer to the different addresses where we store the values and data for the inputs and output of our AES coprocessor.

The APB module which we setup and created in the *Platform Designer* connects to a specific address which we have defined as seen in Figure 2. We initialized all of the different signals that are going to be used within our project so that we can communicate with the hardware addresses and read data from the terminal which entered from the user’s keyboard. Some benefits to using an ABP IP module in our design is that we are able to communicate and check anything that is being sent to the board through a minimal number of signals. We just need one module which has a couple of read, enable, and write signals to which map to their respective 32-bit input and output signals, where we are able to read and write information to and from the DE1-SoC FPGA board.

Implementing the CBC mode into our encryption was very simple, as CBC mode xor’s the *Plaintext* input with the Initialization Vector (IV) for its first encryption block, but after that the *Ciphertext* output is becomes the new IV for the next block which takes in another *Plaintext* input. To have a better understanding of the CBC implementation look at Figure 1 and at the ***Design Methodology*** section of this report which outlines in detail how our encryption works in CBC mode. The first block of encryption takes in the ptext, IV and key that is entered into the terminal for our implementation. In CBC mode the ptext and IV are xor’d with each other and are then encrypted based on the key entered. The next blocks in the sequence take the previous’ block *Ciphertext* and use that to xor with the newly entered ptext. This is how the security within the CBC mode is very strong especially with 128-bit encryption.

Diagram, schematic

Description automatically generated

Figure 1 - CBC Mode Flow Diagram (Dworkin, 2001, p.10)

# Design Methodology

Our implementation consisted of four parts: implementing a simple APB IP module, implementation of the AES-128-bit coprocessor, memory mapping the hardware & software interfaces, and designing the C code to test our APB and AES coprocessor interfacing. To implement the APB IP module, we first needed to enter into the *Platform Designer* window within the Quartus Prime software and create a new component utilizing the APB\_slave peripheral (Muresan, ENGG\*4560 Laboratory Project Tutorial: Design of Custom APB IP with the Platform Designer, 2022). As seen in Figure 2 below we have initialized the associated signals with the module so that we can later utilize them in memory mapping to interface with our software.

Graphical user interface

Description automatically generated

Figure 2 - Platform Designer Signals & Interface Creation Window of APB IP Module

After creation of our APB IP module in the *Component Editor* within the *Platform Designer* window, we connected the aps\_s0, clock and reset signals to the HPS which was created previously so we are able to use the memory within the board itself and use the internal clock and reset signals from the board, so that we do not need to create our own clock and reset buttons or signals. The aps\_s0, clock and reset signal connections can be seen in Figure 3 below. This figure also shows us the other connections between the HPS modules which hold the signals for the addresses, clock, reset and various enable signals that are integrated within the FPGA board.

Table

Description automatically generated

Figure 3 - Platform Designer System Contents Window Showing the APB IP Module Connections

Now that the APB module is created, we are able to configure its purpose in Verilog and implement our AES-128-bit coprocessor. For our purposes we want to be able to read and write 128 bits at any given time. As seen in Appendix Figures 1 to 3, we initialized many unique signals, but the main ones we will look at are the *data\_out* register, ptext, key, IV, and encrypted wires. The *data\_out* register is used as our main memory storage space where the data is held within the code and on board. This signal has 24 memory spaces each with 32 bits where data is stored from the associated signals. Each of ptext, key, IV, and ptext2 each take four memory spaces each, totaling to 128-bits in total and are stored within the *data\_out* index’s 0 – 11 and 16 – 19 respectively, which is seen from lines 39 – 57 in Appendix Figure 1. All we have done so far is assign the different signals which we will use later to the respective *data\_out* signal indexes, from lines 61 to 148 is where the main reading and writing between the hardware and software happen.

From lines 61 to 97 is where the different *data\_out* signals are mapped to their respective addresses which will be called in the software portion of the AES. By assigning the different addresses to their respective variables allows us to read 32-bits 4 times at once which will be faster than just reading 128-bits one time since this will be done in parallel. The data which is being written to the terminal is being read and assigned when we are referring to the *aps\_s0\_pwdata* signal from the ABP IP module. The code from lines 111 to 135 in Appendix Figure 3 is when the associated address is being called in software so the data can be written to it in hardware. The last part of our ABP IP module code is writing the encrypted data to the associated *data\_out* memory so that we are able to reference the addresses in software and output the *Ciphertext* created by our encryption. As explained, the *data\_out* register is used as a write and read register since we are able to reference and store data based on the address that we send from the software code. This allows us to read or write to any portion of the *data\_out* array that is defined and within our constraints. As seen on lines 151 and 153, we only iterate through the encryption process two times, as we only implemented two blocks/iterations of CBC mode. This was done due to the long compilation and was in the best interest of us to ensure that functionality was complete.

Our hardware is now configured to communicate with our software implementation which is in C. The C code consists of two main parts which can be seen in Appendix Figure 4 between lines 77 to 83 and 104 to 110. The first section of the code outlines how we take input from the terminal. We define that the user must enter hexadecimal values and 32 bits four times since that is what our ABP module is configured for. We are able to store the associated data from the user into the specified addresses by using the calculation on line 81 to get the exact address space based on our iteration in the for loop. We then assign the *mem\_data* that was entered to the associated address depending on what the program is asking the user for (ptext, key, or IV). We are then able to fetch the data from memory by using a similar process. Instead of prompting the user to enter values, we read from the address where the 128 encrypted bits are stored and print out the data within the address as seen on line 109.

# Test Results

To test our AES coprocessor implemented in CBC mode, we used test cases from the NIST AES Modes Operation manual. These test cases were used and manually typed into our terminal when testing the encryption process to ensure that any 128-bits can be entered with an associated key and IV, and our AES coprocessor would take in the ptext and encrypt it. These test cases ensured that our process was correct and that our hardware and software interface was coded and being references from correctly. Figure 4 below shows the inputs that were tested to our encryption block. The *Plaintext* is what is entered into the terminal by the user, *Input Block* represents the xor output between the *Plaintext* and *IV* and the *Ciphertext* represents the output from the AES encryption block (Dworkin, 2001).

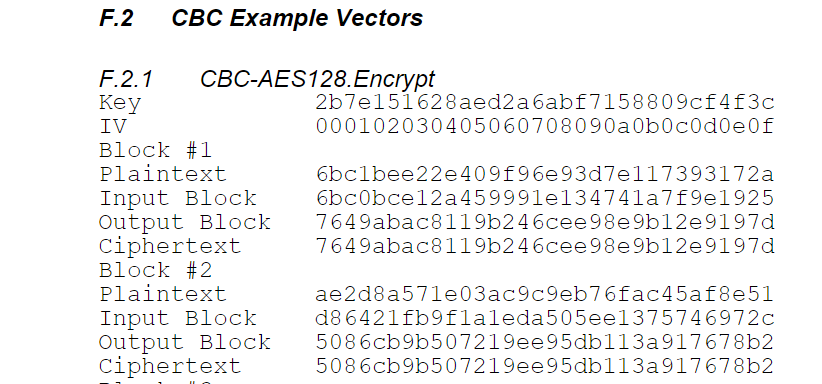


Figure 4 - 128-bit CBC Mode Test Input (Dworkin, 2001, p.27)

Figure 5 below outlines the results from the testing of our AES coprocessor in CBC mode. The data is entered through the terminal of the host PC. Looking at the figure, numbers 1, 2, 3 and 5 outlines on the image are inputs to the terminal typed by the user from the keyboard and 4, and 6 are the encryption block outputs after the encryption process is done for each associated block. The first input to the system is the *Plaintext* for the first block, followed by the *Key* that will be used for encryption and the *IV* which will be xor’d with the *Plaintext.* The system will then compute the first block in CBC mode and output the *Ciphertext* after the encryption is completed for that block. As soon as the block is finished encrypting, it will ask the user for the *Plaintext* that will be used in the second iteration of the CBC encryption. After the input is appropriately inputted into the system, the computation will begin on for the second block of encryption which takes the same *Key* but uses the previous block *Ciphertext* to xor with the newly entered *Plaintext.* Once that computation is complete the system will print out the second *Ciphertext* and finish executing.

Text

Description automatically generated

1

2

3

4

5

6

Figure 5 - Terminal Interface During Test

# Conclusions

In conclusion, this lab provided an opportunity to design and implement an Advanced Encryption Standard coprocessor in CBC mode using the DE1-SoC board. The successful implementation of a simple APB IP module allowed for communication between the software and hardware on the board, enabling the coprocessor to read values entered through the terminal. Through this project, we gained a better understanding of how hardware and software on an FPGA board communicate, which is crucial to understanding how systems work together. By working with a memory-mapping coprocessor module, we have acquired the skills to implement various designs in the future. Overall, this lab provided practical knowledge and hands-on experience that can be applied to real-world scenarios in the field of digital systems design and embedded systems.

# References

Dworkin, M. (2001, December). *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. Retrieved from NIST: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

Muresan, R. (2022, May). ENGG 4560/6600 Laboratory Manual. Guelph.

Muresan, R. (2022, March). ENGG\*4560 Laboratory Project Tutorial: Design of Custom APB IP with the Platform Designer. Guelph.

# Appendix

Text

Description automatically generated with medium confidence

Appendix Figure 1 - AES\_IP (APB) Code Lines 1 to 60

Table

Description automatically generated with medium confidence

Appendix Figure 2 - AES\_IP (APB) Code Lines 60 to 109

Text

Description automatically generated with medium confidence

Appendix Figure 3 - AES\_IP (APB) Code Lines 109 to 155

Table

Description automatically generated with medium confidence

Appendix Figure 4 - C Code Implementation for Reading and Writing to Hardware