Efficient Implementation of IEEE 802.11i Wi-Fi Security (WPA2-PSK) Standard Using

FPGA

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

<name of your degree here>

<your academic unit>

<Month of graduation> <Year of graduation>

# **Acknowledgements**

# **Abstract**

The rationale behind the thesis was to design efficient implementations of cryptography algorithms used for Wi-Fi Security as per IEEE 802.11i Wi-Fi Security (WPA2-PSK) standard. The focus was on software implementation of Password-Based Key Derivation Function 2 (PBKDF2) using Keyed-Hash Message Authentication Code (HMAC)-SHA1, which is used for authentication, and , hardware implementation of AES-256 cipher, which is used for data confidentiality.

In this thesis, PBKDF2 based on HMAC-SHA1 was implemented on software using C programming language, and, AES-256 was implemented on hardware using Verilog HDL. The overall implementation was designed and tested on Nexys4 FPGA board. The performance of the implementation was compared with other existing designs. Latency (us) was used as the performance metric for PBKDF2, whereas, throughput (Gb/s), resource utilization (Number of Slices), efficiency (GB/s per slice) and latency (ns) were used as performance metrics for AES-256. MRF24WG0MA PMOD Wi-Fi module was the 2.4 GHz Wi-Fi module which was interfaced with Nexys4 FPGA board for wireless communication.

When the correct security credentials were entered in the implemented system interfaced to the Wi-Fi module, it was successfully authenticated by a 2.4 GHz wireless router (or mobile hotspot) configured to work in WPA2-PSK security mode. Once this system was authenticated, the implemented AES-256 cipher within the system was used to provide a layer of encryption over the data being communicated in the network.

# **Table of Contents**

[**Acknowledgements** 2](#_Toc529182884)

[**Abstract** 3](#_Toc529182885)

[**Table of Contents** 4](#_Toc529182886)

[**List of Figures** 6](#_Toc529182887)

[**List of Tables** 7](#_Toc529182888)

[**Abbreviations** 8](#_Toc529182889)

[**Chapter 1: Introduction** 9](#_Toc529182890)

[**1.1** **Background** 9](#_Toc529182891)

[**1.2** **Wireless Security Principle** 10](#_Toc529182892)

[**1.3** **WPA2-PSK Overview** 11](#_Toc529182893)

[**1.4** **Project Scope** 13](#_Toc529182894)

[**Chapter 2: WPA2-PSK Theory** 14](#_Toc529182895)

[**2.1**  **WPA2-PSK Device Authentication** 14](#_Toc529182896)

[**2.1.1**  **PBKDF2 (Password-Based Key Derivation Function 2)** 16](#_Toc529182897)

[**2.1.2**  **HMAC (Keyed-Hashing for Message Authentication)** 19](#_Toc529182898)

[**2.1.3**  **SHA1** 20](#_Toc529182899)

[**2.2** **WPA2-PSK Data Confidentiality** 25](#_Toc529182900)

[**2.2.1** **AES-256 Key Expansion** 29](#_Toc529182901)

[**2.2.2** **AES-256 Encryption** 31](#_Toc529182902)

[**2.2.3**  **AES-256 Decryption** 32](#_Toc529182903)

[**Chapter 3: Software Implementation** 35](#_Toc529182904)

[**3.1**  **Overview** 35](#_Toc529182905)

[**3.2**  **MicroBlaze™ Environment** 35](#_Toc529182906)

[**3.3**  **WPA2-PSK Device Authentication** 36](#_Toc529182907)

[**3.4**  **AES-256 Key Expansion** 40](#_Toc529182908)

[**Chapter 4: Hardware Implementation** 44](#_Toc529182909)

[**4.1**  **Overview** 44](#_Toc529182910)

[**4.2**  **Nexys4** 44](#_Toc529182911)

[**4.3**  **AXI Interconnect** 45](#_Toc529182912)

[**4.4**  **AES-256 Implementation** 46](#_Toc529182913)

[**4.5**  **AES-256 Interface with MicroBlaze** 55](#_Toc529182914)

[**Appendices** 56](#_Toc529182915)

[**1.1**  **AES Lookup Tables** 56](#_Toc529182916)

[**1.2**  **Code Snippets** 60](#_Toc529182917)

[**References** 63](#_Toc529182918)

# **List of Figures**

[**Figure 1: Example of Wi-Fi Network** 10](#_Toc529445549)

[**Figure 2: WPA2-PSK Security** 12](#_Toc529445550)

[**Figure 3: WPA2-PSK Authentication** 12](#_Toc529445551)

[**Figure 4: WPA2-PSK Data Confidentiality** 13](#_Toc529445552)

[**Figure 5: Block Diagram for PBKDF2** 17](#_Toc529445553)

[**Figure 6: PBKDF2 with HMAC-SHA1** 18](#_Toc529445554)

[**Figure 7: Block Diagram for HMAC** 20](#_Toc529445555)

[**Figure 8: Block Diagram for SHA1 Processing Function** 24](#_Toc529445556)

[**Figure 9: Input Bytes Arranged in State Array at Beginning of AES Operation** 25](#_Toc529445557)

[**Figure 10: Output Bytes Arranged from State Array at End of AES Operation** 26](#_Toc529445558)

[**Figure 11: Rcon( ) Values** 26](#_Toc529445559)

[**Figure 12: Example of ShiftRows()** 27](#_Toc529445560)

[**Figure 13: MixColumns() Calculation** 28](#_Toc529445561)

[**Figure 14: Example if InvShiftRows()** 28](#_Toc529445562)

[**Figure 15: InvMixColumns Calculation** 29](#_Toc529445563)

[**Figure 16: Pseudo Code for Key Expansion for Encryption** 30](#_Toc529445564)

[**Figure 17: Additional Pseudo Code to be Added for Key Expansion for Decryption** 31](#_Toc529445565)

[**Figure 18: Pseudo Code for AES Encryption** 32](#_Toc529445566)

[**Figure 19: Pseudo Code for the Equivalent Inverse Cipher** 33](#_Toc529445567)

[**Figure 20: AES-256 Block Diagram** 34](#_Toc529445568)

[**Figure 21: MicroBlaze Core Block Diagram** 35](#_Toc529445569)

[**Figure 22: Layered Software Implementation of WPA2-PSK Authentication** 37](file:///C:\Thesis\Report\New%20folder\Report_CHP5_1st_Draft.docx#_Toc529445570)

[**Figure 23: Nexys4 Board Features** 45](#_Toc529445571)

[**Figure 24: Key BRAM Organization** 48](#_Toc529445572)

[**Figure 25: Data BRAM Organization** 48](#_Toc529445573)

[**Figure 26: AES Core with Input and Output Signals** 49](#_Toc529445574)

[**Figure 27: Internal Design of AES-256** 52](#_Toc529445575)

[**Figure 28: Block Diagram of AES-256 IP Core with MicroBlaze** 55](#_Toc529445576)

[**Figure 29: Overall Block Diagram for Testing** 56](#_Toc529445577)

[**Figure 30: Mobile Hotspot Configuration** 57](#_Toc529445578)

[**Figure 31: Nexys4 Board with MRF24WG0MA PMOD Wi-Fi** 58](#_Toc529445579)

[**Figure 32: Failed Authentication with Incorrect SSID and Incorrect Password** 59](#_Toc529445580)

[**Figure 33: Failed Authentication with Correct SSID and Incorrect Password** 60](#_Toc529445581)

[**Figure 34: Failed Authentication with Incorrect SSID and Correct Password** 60](#_Toc529445582)

[**Figure 35: Successful Authentication with Correct SSID and Correct Password** 61](#_Toc529445583)

[**Figure 36: Encrypted Data Sent by TCP Client to TCP Server** 63](#_Toc529445584)

[**Figure 37: Decryption of Data Received by TCP Server from TCP Client** 64](#_Toc529445585)

[**Figure 38: Encrypted Data Sent by TCP Server to TCP Client** 65](#_Toc529445586)

[**Figure 39: Decryption of Data Received by TCP Client from TCP Server** 66](#_Toc529445587)

[**Figure 40: AES Webpage hosted by HTTP server** 67](#_Toc529445588)

[**Figure 41:Decryption and Encryption of Data received by HTTP server** 68](#_Toc529445589)

[**Figure 42: Resources Utilization Table for LUT Implementation** 69](#_Toc529445590)

[**Figure 43: Graph for Resource Utilization for LUT Implementation** 69](#_Toc529445591)

[**Figure 44: Resources Utilization Table for BRAM Implementation** 70](#_Toc529445592)

[**Figure 45: Graph for Resource Utilization for BRAM Implementation** 70](#_Toc529445593)

[**Figure 46: Implementation of S-Box in C** 76](#_Toc529445594)

[**Figure 47: Implementation of Mul9 Lookup Table in C** 76](#_Toc529445595)

[**Figure 48: Implementation of Mul11 Lookup Table in C** 77](#_Toc529445596)

[**Figure 49: Implementation of Mul13 Lookup Table in C** 77](#_Toc529445597)

[**Figure 50: Implementation of Mul14 Lookup Table in C** 78](#_Toc529445598)

# **List of Tables**

[**Table 1: Table of Test Cases and Results for Authentication** 59](#_Toc529208992)

[**Table 2: S-Box Lookup Table** 63](#_Toc529208993)

[**Table 3: Inverse S-Box Lookup Table** 63](#_Toc529208994)

[**Table 4: Mul2 Lookup Table** 64](#_Toc529208995)

[**Table 5: Mul3 Lookup Table** 64](#_Toc529208996)

[**Table 6: Mul9 Lookup Table** 65](#_Toc529208997)

[**Table 7: Mul11 Lookup Table** 65](#_Toc529208998)

[**Table 8: Mul13 Table** 66](#_Toc529208999)

[**Table 9: Mul14 Table** 66](#_Toc529209000)

# **Abbreviations**

# **Chapter 1: Introduction**

## **Background**

As most technologies have continued to transition from traditional wired systems to wireless ones, the number of wireless devices has grown by leaps and bounds over the last decade. Wireless devices have become a part of our day-to-day lives with its presence seen in household, educational and business institutions, to name a few. These devices are inter-connected with one another and share a variety of data, ranging from mundane to very personal and confidential information. Such interconnected devices that share data among themselves form a network. There can be various types of networks based on topology, size, area, organization, etc. One such type of network based on area is called Local Area Network (LAN). Such network is confined within a localized area such as a room, building or a group of buildings. However, it can be inter-connected to other LANs using wired or wireless media. If wireless medium is used to connect such LANs, then the overall network is called Wireless LAN (WLAN) [1].

The communication between the devices within a network is governed by a set of rules called communication protocols. The devices within a network must adhere to such protocols to successfully share and interpret data among other devices connected to the network. To maintain interoperability between the devices manufactured by various vendors, standardized communication protocols are defined for different type of networks. One such protocol for communication between wireless devices over LAN is the IEEE 802.11 protocol and is commonly known as ***Wi-Fi*** [2]. An example of ***Wi-Fi*** network is shown in Figure 1.



**Figure 1: Example of Wi-Fi Network**

## **Wireless Security Principle**

Security is paramount in any type of network, but it is more so in the case of wireless networks, as they are far more vulnerable to attack in comparison to wired networks. In a wired network, the communicating devices must be physically connected using a cable. Hence, it is easier to verify the identity of the device to which the data is being communicated, as opposed to in wireless networks, where this is not quite easy. Also, unlike in wired networks, where the data is communicated through copper wires or optical fibers, in wireless networks, the wireless devices use RF signals in open air as their communication medium. So, theoretically any transceiver which is within the range of this RF signal and tuned to its frequency can read and/or meddle with the data being communicated [3]. Hence, for a secure communication, it is necessary to identity whether a device trying to connect to the network has proper security credential or not. This process is called authentication [3]. After a wireless device is authenticated to a network, the data being communicated within that network must be made confidential using a secure cryptography algorithm [3].

## **WPA2-PSK Overview**

The current standardized security protocol for ***Wi-Fi*** is IEEE 802.11i standard. This is also commonly known as ***Wi-Fi*** Protected Access II (WPA2). WPA2 was launched in September 2004 and supports PSK technology and includes an advanced encryption mechanism using the Counter-Mode/CBC-MAC Protocol (CCMP) called the Advanced Encryption Standard (AES) [4]. The PSK technology (in personal networks) is used to verify the identity of the communicating wireless devices. In PSK, the authentication process is performed by the access point (wireless router, mobile hotspot, etc.). With PSK, we can configure the access point (wireless router or hotspot) with a passphrase of 8 to 63 printable ASCII characters [5]. Using a technology called ***PBKDF2***, that passphrase, along with the network SSID, is used to generate unique encryption keys for wireless clients. In ***WPA2-PSK*** security, the same set of SSID and PSK is shared between all ***Wi-Fi*** end devices and the access point as shown in Figure 2 [6]. The SSID is analogous to Username and PSK is analogous to Passphrase in Figure 2. The wireless devices are authenticated and granted access to the network, if the password to the particular SSID matches [5]. After authentication, AES cipher is used to maintain the confidentiality of the data being communicated within the network.



**Figure 2: WPA2-PSK Security**

In Figure 3, SSID and Passphrase goes through ***PBKDF2*** to derive the 256-bit PMK which is used as the main key for AES cipher. The validity of this key is confirmed using the 4-way handshake process (Figure 2) between the Wi-Fi device and the access point [6]. If the key matches, then, the ***Wi-Fi*** device is successfully authenticated by the access point.



**Figure 3: WPA2-PSK Authentication**

After successful authentication, the data between ***Wi-Fi*** end devices and the access point is encrypted using AES cipher with the 256-bit PMK as the main key (Figure 4).



**Figure 4: WPA2-PSK Data Confidentiality**

## **Project Scope**

The purpose of this thesis is to optimize the cryptography algorithms used in device authentication and data confidentiality in ***Wi-Fi*** networks configured with ***WPA2-PSK*** security. To achieve this, the main key derivation part of the authentication process, as well as, the AES cipher algorithm required for data confidentiality will be optimized. The scope of the implementation will encompass the following areas:

* Efficient software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** which is used for device authentication.
* Efficient hardware implementation of ***AES-256*** cipher which is used for data confidentiality.

The performance of these implementations will be compared with other existing designs. Latency (us) will be used as the performance metric for ***PBKDF2***, whereas, throughput (Gb/s), resource utilization (Number of Slices), efficiency (GB/s per slice) and latency (ns) will be used as performance metrics for ***AES-256***.

# **Chapter 2: WPA2-PSK Theory**

## **2.1 WPA2-PSK Device Authentication**

Authentication is the process by which you prove that you are eligible to join a network (and that the network is legitimate) [3]. Pre-Shared Key (PSK) is a device authentication method used in ***WPA2-PSK*** networks, and it uses a passphrase of 8 to 63 printable ASCII characters to generate unique encryption keys [5]. The general idea of PSK mode is to use the same secret key on an access point and on a ***Wi-Fi*** device to authenticate the device and establish an encrypted connection for networking [6]. Hence, both ***Wi-Fi*** device and access point must prove to each other that they know the pre-shared key to ensure a secure connection. In ***WPA2-PSK***, the access point (wireless router, hotspot, etc.) with a network SSID is configured with a passphrase. Using ***PBKDF2***, that passphrase along with network SSID is used to generate the 256-bit Pairwise-Master-Key (PMK). The ***Wi-Fi*** device must also derive the same PMK using the same passphrase and SSID for the access point to authenticate the device.

PMKs are never transmitted across the network as the channel of communication is not secure before the authentication process has completed. Because, without authentication, sharing of PMK would be done through an unencrypted channel and susceptible to be discovered by outside parties [7]. To overcome this, ***WPA2-PSK*** uses 4-way handshake to verify whether the ***Wi-Fi*** device and the access point have the same PMK or not (Figure 2). The 4-way handshake is designed so that the access point and ***Wi-Fi*** device can independently prove to each other that they know the PMK, without ever disclosing it. In Figure 2, the 4-way handshake is broken down into 4 messages [7]:

* **Message 1 (From Access Point to Wi-Fi Device):** The first step is for the access point to generate a nonce value. The nonce value is a pseudo random value generated by a publicly known and repeatable process. This pseudo random value is generated by the Pseudo Random Function 256 or PRF-256, as defined by WPA2 specifications. The nonce value generated by access point is called A-nonce. The access point sends a message containing this A-nonce value to the ***Wi-Fi*** device.
* **Message 2 (From Wi-Fi Device to Access Point):** The ***Wi-Fi*** device generates a nonce value using the same process as the access point and it is denoted as S-nonce. When the ***Wi-Fi*** device receives Message 1, it will generate Pairwise-Transient-Key (PTK). This key is required to be generated by both parties, and allows each party to verify that the other has the correct PMK. The creation of PTK is performed via another Pseudo Random Function (PRF), which uses a combination of the PMK, Access Point MAC Address, Wi-Fi Device MAC Address, A-nonce and S-nonce [8]. A part of the PTK is known as the message integrity check (MIC). This value, along with the S-Nonce is then transmitted back to the access point.
* **Message 3 (From Access Point to Wi-Fi Device):** When the access point receives Message 2, it has all the values required to generate the PTK. The access point then generates the PTK, and checks whether the MIC value in Message 2 matches the MIC value that it has just generated. If the two MIC values matches, this proves that the Wi-Fi device knows the value of the PMK. If the MIC value is correct, the access point, then sends Message 3 to the ***Wi-Fi*** device. Message 3 allows the ***Wi-Fi*** device to ensure that the access point is a trusted party. If the access point did not have a matching PMK, the MIC would be different. Message 3 also informs the ***Wi-Fi*** device that the communication channel is about to be encrypted.
* **Message 4 (From Wi-Fi Device to Access Point):** The final part of the handshake allows the ***Wi-Fi*** device to acknowledge that the access point is now going to use encryption for the communication. After the ***Wi-Fi*** device transmits Message 4, it will install the encryption keys on the channel. After the access point receives message 4, it will install the encryption keys as well. All further unicast communication is protected by this encryption, until the client disconnects from the access point [3].

### **2.1.1 PBKDF2 (Password-Based Key Derivation Function 2)**

Using ***PBKDF2***, the passphrase and SSID are hashed 4096 times to produce a 256-bit PMK [9]. Internally, the ***PBKDF2*** key derivation function employed in ***WPA2-PSK*** utilizes 4096 iterations of ***HMAC-SHA1*** to obtain 160-bit hash outputs. Since the PMK in ***WPA2-PSK*** is of 256-bits, two rounds of ***PBKDF2*** are necessary [10]. Their outputs are concatenated, but for the second iteration the output is truncated to 96 bits to achieve the 256-bit PMK. The ***PBKDF2*** key derivation function is defined as follow:

DK = PBKDF2(PRF, P, S, C, dkLen) ................... (1)

where,

DK: Derived key

PRF: Pseudorandom function of two parameters with output length hLen

P: Password

S: Salt (sequence of bits)

C: Iteration count, a positive integer

dkLen: Length of Derived key

To derive key from ***PBKDF2***, each hLen bit block Ti of derived key DK, is computed as follows:

DK = T1 || T2 || ... || Tdklen/hlen ..................... (2)

Ti = F (P, S, C, i) ...................... (3)

In equation (3), the function F is the Exclusive-OR operations of C iterations of PRFs (as shown in equation (4)). In the first iteration, the PRF uses Password as the key and Salt concatenated with i (encoded as a big-endian 32-bit integer) as the 2 parameters (as shown in equation (5)). For, subsequent iterations, PRF uses Password as the key and the output of the previous PRF computation as the salt (as shown in equations (6) and (7)). The block diagram for ***PBKDF2*** key derivation function is shown is Figure 5.

F (Password, Salt, C, i) = U1 ⊕ U2 ⊕ ... ⊕ Uc .... (4)

where,

U1 = PRF (Password, Salt || INT\_32\_BE(i)) ......... (5)

U2 = PRF (Password, U1) ........................... (6)

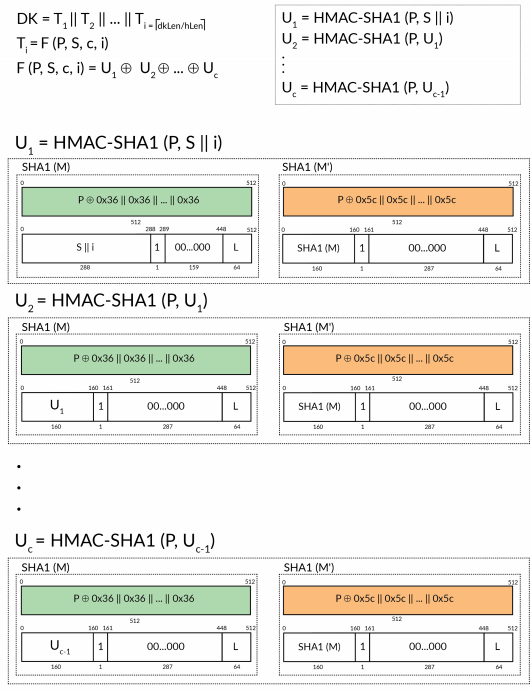
...

Uc = PRF (Password, Uc - 1) ......................... (7)



**Figure 5: Block Diagram for PBKDF2**

The overview of ***PBKDF2*** along with HMAC and SHA1 is shown in Figure 6 [11].



**Figure 6: PBKDF2 with HMAC-SHA1**

In case of ***WPA2-PSK***, the output and parameters in equation (1) are as follows:

PMK = PBKDF2(HMAC−SHA1, passphrase, ssid, 4096, 256) ...... (8)

### **2.1.2 HMAC (Keyed-Hashing for Message Authentication)**

***HMAC*** provides a mechanism to calculate a message authentication code (MAC) based around a cryptographic hashing function [7]. A message authentication code (MAC) is a short piece of information used to authenticate a message. MACs are used between two parties that share a secret key to validate information transferred between them [12]. The use of HMAC in PBKDF2 is shown in Figure 6. The definition of ***HMAC*** requires a cryptographic hash function denoted by H, the secret key denoted by M and the message to be authenticated denoted by m. The ***HMAC*** function is defined as follows:

HMAC (M, m) = H ((M' ⊕ opad) || H ((M' ⊕ ipad)|| m)) ..........(9)

where,

H: a cryptographic hash function

M: the secret key

m: the message to be authenticated

M': another secret key, derived from the original key K

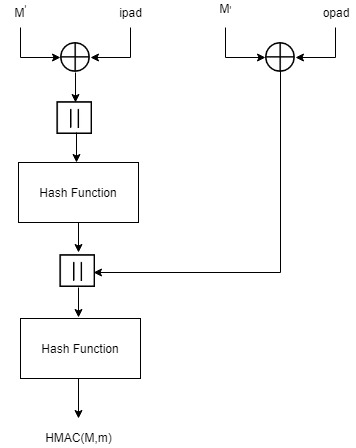
(by padding K to the right with extra zeroes to the

input block size of the hash function, or by hashing K if it is longer than that block size)

opad: the outer padding (0x5c5c5c…5c5c, one-block- long hexadecimal constant

ipad: the inner padding (0x363636…3636, one-block- long hexadecimal constant).

The block diagram for ***HMAC*** function is shown in Figure 7.



**Figure 7: Block Diagram for HMAC**

For ***WPA2-PSK***, the parameters in equation (9) are as follows:

HMAC-SHA1 (passphrase, ssid) = SHA1 ((passphrase ⊕ opad)

|| SHA1((passphrase ⊕ opad ⊕ ipad)

||ssid) ... (10)

### **2.1.3 SHA1**

Hashing algorithms are used to process a message and produce a condensed representation of the message which is called a message digest, and for a perfect hashing function, it should be only one-way and a unique digital signature of the message [7]. The use of ***SHA1*** in ***PBKDF2*** is shown in Figure 6. The ***SHA1*** hashing algorithm is valid for messages with a size less than 264 bits, it operates on blocks of size 512 bits, it uses a word size of 32 bits, and has a resultant message digest of 160 bits [7]. ***SHA1*** algorithm primarily consists of 6 steps [13]:

**Step1: Append Padding Bits**: The original message is padded based on the following rules:

* The original message is first padded with one bit ‘1’.
* Zeros ‘0’ are then padded to bring the length of message to 64 bits less than multiple of 512.

**Step2: Append Length:** A 64-bit value indicating the length of the original message is appended to end the message obtained from Step 1 based on the following rules:

* 64-bit value of the original message is appended at the end of the padded message. If overflow occurs, the lower order of the 64-bit value is appended.
* The lower 32-bit word of the 64-bit value is appended first followed by the upper 32-bit value.

**Step3: Prepare Processing Functions:** ***SHA1*** has 80 processing rounds. There are 4 mathematical operations assigned to each of the 4 sets of 20 rounds. These operations are as follows:

for 0 <= r <= 19,

F (r: B, C, D) = (B & C) | ((! B) & D) .............(11)

for 20 <= r <= 39,

F (r: B, C, D) = B ⊕ C ⊕ D................(12)

for 40 <= r <= 59,

F (r: B, C, D) = (B & C) | (B & D) | (C & D) .......(13)

for 60 <= r <= 79,

F (r: B, C, D) = B ⊕ C ⊕ D .....................(14)

**Step4: Prepare Processing Constants**: ***SHA1*** has 4 different constants assigned to 4 sets of 20 rounds. These constants are as follows:

for 0 <= r <= 19,

K(r) = 0x5A827999 ............................... (15)

for 20 <= r <= 39,

K(r) = 0x6ED9EBA1 ............................... (16)

for 40 <= r <= 59,

K(r) = 0x8F1BBCDC ............................... (17)

for, 60 <= r <= 79

K(r) = 0xCA62C1D6 ............................... (18)

**Step5: Initialize Buffer: *SHA1*** has five 32-bit buffers which are initialized as follows:

H0 = 0x67452301 ................................. (19)

H1 = 0xEFCDAB89 ................................. (20)

H2 = 0x98BADCFE ................................. (21)

H3 = 0x10325476 ................................. (22)

H4 = 0xC3D2E1F0 ................................. (23)

**Step6: Process 512-bit block messages:** The algorithm to process this 512-bit block of message is as follows:

For loop on k = 1 to N /\* 1st For loop \*/

(W (0), W (1) ..., W (15)) = M[k] /\*Divide M[k] into 16 words\*/

For t = 16 to 79 do: /\* 2nd For loop \*/

     W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W (t-16)) << 1

End of For loop /\* 2nd For loop \*/

 A = H0, B = H1, C = H2, D = H3, E = H4

  For t = 0 to 79 do: // 3rd for loop

        TEMP = A<<5 + f (t: B, C, D) + E + W(t) + K(t)

        E = D, D = C, C = B<<30, B = A, A = TEMP

   End of For loop // 3rd for loop

  H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D,

H4 = H4 + E

  End of for loop /\* End of 1st For loop \*/

Output = H0 << 128 | H1 << 96 | H2 << 64 | H3 << 32 | H4

The block diagram for ***SHA1*** processing function is given in Figure 8.

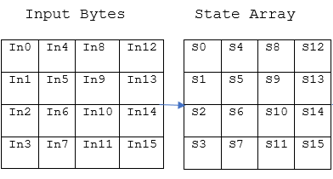


**Figure 8: Block Diagram for SHA1 Processing Function**

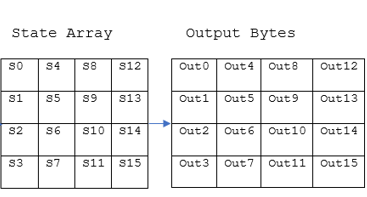
## **2.2 WPA2-PSK Data Confidentiality**

***WPA2-PSK*** uses Advanced Encryption Standard (AES) cipher for data confidentiality. The AES algorithm is a symmetric block cipher that can encrypt and decrypt information. Encryption converts data to an unintelligible form called ciphertext and decryption converts the ciphertext back into its original form, called plaintext [14]. AES has input block size of 128 bits and key size can be of 128, 192 or 256 bits. ***WPA2-PSK*** uses key size of 256 bits i.e. ***AES-256***. It requires 60 rounds for key expansion and the size of the expanded key is 240 bytes. When decryption is performed using Equivalent Inverse Cipher method, then there are separate set of expanded keys for encryption and decryption processes [14]. Hence, in total, there will be 480 bytes of expanded keys when decryption is performed using Equivalent Inverse Cipher method. AES-256 requires 14 rounds each for the completion of encryption and decryption processes and the size of output block is 128 bits.

AES algorithm’s operations are performed on a two-dimensional array of bytes called the State [14]. For AES-256, the State consists of four rows of bytes, each containing 4 bytes. At the start of the encryption and decryption, the input – the array of bytes In0, In1, … In15 – is copied into the State array as illustrated in Figure 9. The encryption and decryption operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes Out0, Out1, … Out15 as shown in Figure 10. [14].



**Figure 9: Input Bytes Arranged in State Array at Beginning of AES Operation**



**Figure 10: Output Bytes Arranged from State Array at End of AES Operation**

AES consists of key expansion, encryption and decryption processes. These processes are realized with the help of following transformations [14]:

* **RotWord():**This function takes 4 bytes as an argument. It performs circular left shift on the 4 input bytes.

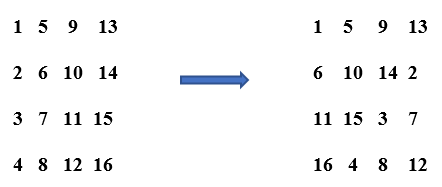
Example: 1,2,3,4 to 2,3,4,1

* **Rcon ():** This function returns a 4-byte value based on Figure 11.



**Figure 11: Rcon( ) Values**

* **AddRoundKey():** In the AddRoundKey() transformation, a Round Key is added to the state by a simple bitwise XOR operation.
* **SubBytes():**In this transformation, each byte of data is substituted with the corresponding value from the S-box lookup table (Table1 in Appendix 1.1).
* **ShiftRows():**This function arranges the bytes of the state in 4x4 matrix and performs byte-wise circular left shift. The order of the shift varies with rows. The shift operation is not performed for the first row. The second row is shifted by 1 byte, the third row is shifted by 2 bytes and the fourth row is shifted by 3 bytes. An example of shift row operation is shown in Figure 12.



**Figure 12: Example of ShiftRows()**

* **MixColumns():**The matrix obtained from the ShiftRows() operation goes through the multiplication over Galois Field (Figure 13) . The lookup tables required for multiplication over Galois Field in theMixColumns() operation are shown in Table 3 and Table 4 in Appendix 1.1.



**Figure 13: MixColumns() Calculation**

* **InvSubBytes():**In this transformation, each byte of data is substituted with the corresponding value from the inverse S-box table (Table2 in Appendix 1.1).
* **InvShiftRows():**This function arranges the bytes of the state in 4x4 matrix and performs byte-wise circular right shift. The order of the shift varies with rows. The shift operation is not performed for the first row. The second row is shifted by 1 byte, the third row is shifted by 2 bytes and the fourth row is shifted by 3 bytes. An example of shift row is shown in Figure 14 [15].



**Figure 14: Example if InvShiftRows()**

* **InvMixColumns():**The matrix obtained from the InvShiftRows() operation goes through the multiplication over Galois Field (Figure 15) . The lookup tables required for multiplication over Galois Field in theInvMixColumns() operation are shown in Table 5, Table 6, Table 7 and Table 8 in Appendix 1.1

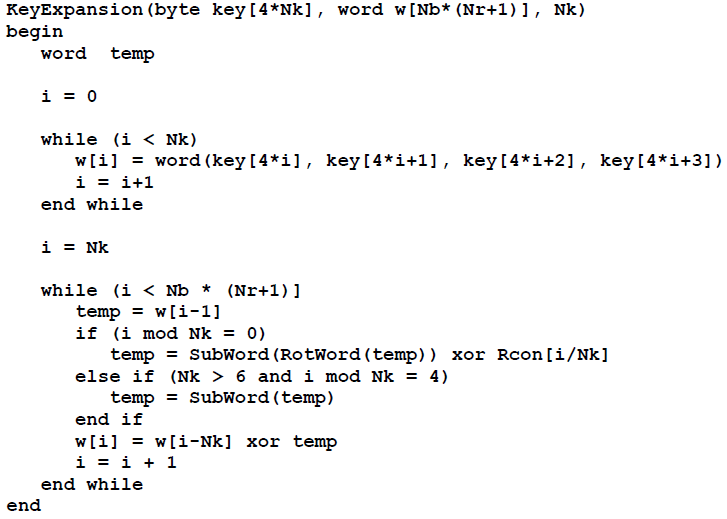


**Figure 15: InvMixColumns Calculation**

### **2.2.1 AES-256 Key Expansion**

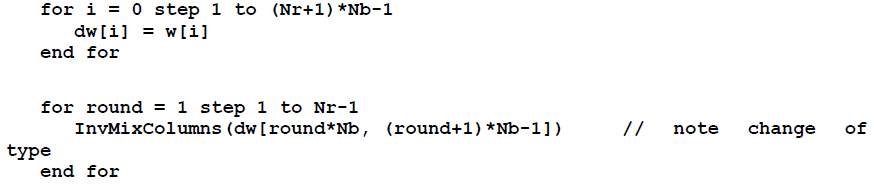
The AES algorithm takes the Cipher Key and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of Nb (Nr + 1) words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data [14]. The resulting key schedule consists of a linear array of 4-byte words, denoted [wi ], with i in the range 0< i< Nb(Nr + 1) [14].

In ***AES-256***, Nb = 4, Nk = 8 and Nr = 14, so the key expansion routine generates a total of 60 words (240 bytes). The key expansion routine runs for 14 rounds and generates 240 bytes of expanded key. Two different sets of 240 bytes of expanded keys are generated for encryption and decryption when the decryption is done using Equivalent Inverse Cipher method. The expansion of the input key into the key expansion routine proceeds according to the pseudo code in Figure 16 [14]. For ***AES-256***, Nb = 4, Nk = 8 and Nr = 14.



**Figure 16: Pseudo Code for Key Expansion for Encryption**

For the Equivalent Inverse Cipher, the following pseudo code shown in Figure 17 must be added to the end of the pseudo code shown in Figure 16 [14]. For ***AES-256***, Nb = 4, Nk = 8 and Nr = 14.

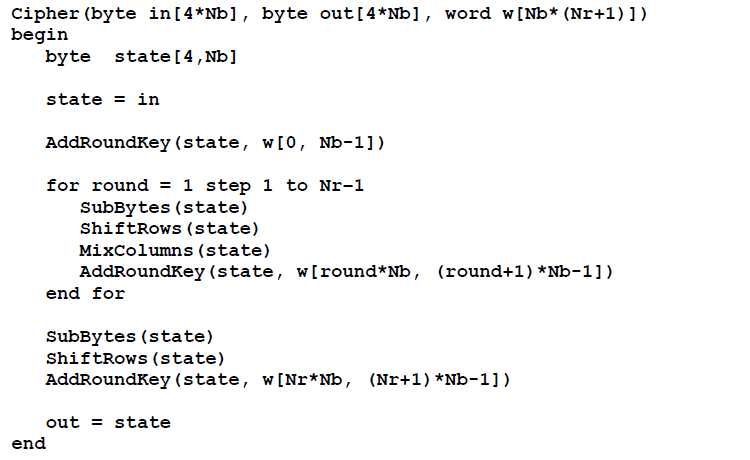


**Figure 17: Additional Pseudo Code to be Added for Key Expansion for Decryption**

### **2.2.2 AES-256 Encryption**

In each round of AES encryption, the cipher makes four different transformations to the block of data. The four transformations are: AddRoundKey (), SubBytes (), ShiftRows () and MixColumns (). The exception is the final round, which only has three transformations since it does not have the MixColumns()operation [16]. During these rounds, each block of data is depicted as 4 x 4 byte matrix and the key is divided into 4 x 4 byte matrix as well. Each round gets these matrices as an input and produces 4 x 4 byte state matrix as an output.

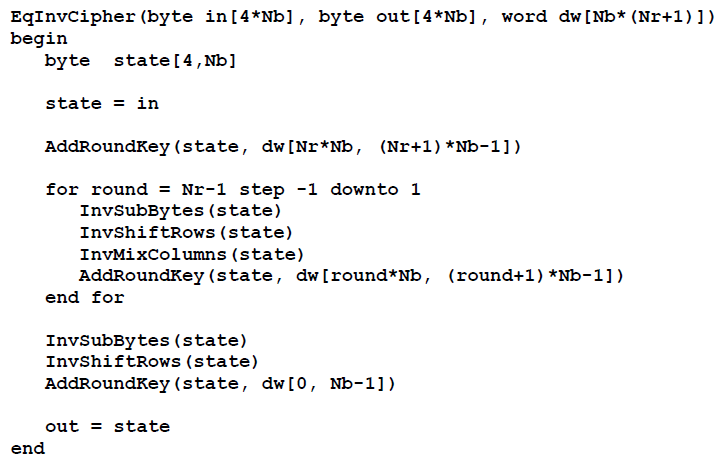
The pseudo code for AES encryption is shown in Figure 18. For ***AES-256***, Nb = 4 and Nr = 14.



**Figure 18: Pseudo Code for AES Encryption**

### **2.2.3 AES-256 Decryption**

Like encryption, in each round of decryption, the cipher makes four different transformations to the block of data. The four transformations being: InvAddRoundKey (), InvSubBytes (), InvShiftRows () and InvMixColumns (). The exception is the final round, which only has three transformations since it does not have the InvMixColumns()operation [15]. Similarly, during these rounds, each block of data is arranged as 4 x 4 byte matrix and the key is also arranged as 4 x 4 byte matrix. But, the expanded keys used here are different to the ones used for encryption. Each round gets these matrices as an input and produces 4 x 4 byte state matrix as an output. Pseudo code for the Equivalent Inverse Cipher is shown in Figure 19 [14]. For ***AES-256***, Nb = 4, Nk = 8 and Nr = 14.



**Figure 19: Pseudo Code for the Equivalent Inverse Cipher**

The overall block diagram of ***AES-256*** is shown in Figure 20.



**Figure 20: AES-256 Block Diagram**

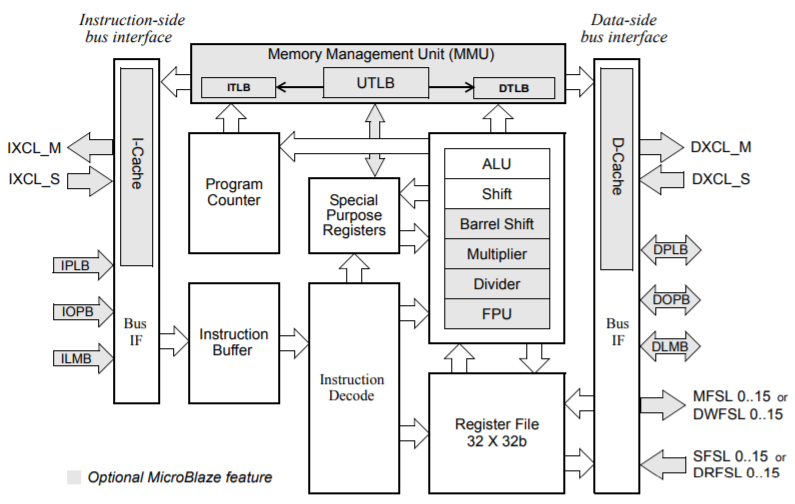
# **Chapter 3: Software Implementation**

## **3.1 Overview**

Derivation of PMK using ***PBKDF2*** for authentication, and key expansion for encryption and decryption for ***AES-256***, was implemented in software. All software development in this thesis was implemented using combination of C and C++ programming languages for the MicroBlaze™ embedded processor soft core in Xilinx Software Development Kit (SDK). Code related to ***PBKDF2*** and ***AES-256*** key expansion was written in C, while the application code was written in C++.

## **3.2 MicroBlaze™ Environment**

The MicroBlaze™ embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx® Field Programmable Gate Arrays (FPGAs). Figure 21 shows a functional block diagram of the MicroBlaze core [17].



**Figure 21: MicroBlaze Core Block Diagram**

The MicroBlaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design. The fixed feature set of the processor includes [17].

* Thirty-two 32-bit general purpose registers
* 32-bit instruction word with three operands and two addressing modes
* 32-bit address bus
* Single issue pipeline

MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses [18]. It does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses [18]: Local Memory Bus (LMB), IBM’s On-chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). MicroBlaze also supports reset, interrupt, user exception, break and hardware exceptions. For interrupts, MicroBlaze supports only one external interrupt source (connecting to the Interrupt input port) [18]. If multiple interrupts are needed, an interrupt controller must be used to handle multiple interrupt requests to MicroBlaze. The stack convention used in MicroBlaze starts from a higher memory location and grows downward to lower memory locations when items are pushed onto a stack with a function call [18]. Items are popped off the stack the reverse order they were put on. Writing software to control the MicroBlaze processor must be done in C/C++ language [18].

## **3.3 WPA2-PSK Device Authentication**

The code for ***WPA2-PSK*** authentication was written using C programming language. The overall code implementation contained 4 layers. At the bottom layer, there was code for ***SHA1*** Hash algorithm. The second layer of code was for ***HMAC-SHA1***, which called functions from the first layer. The third layer of code was for ***PBKDF2*** which would call functions from the second layer. Finally, the application layer called the functions in the ***PBKDF2*** layer for the complete ***WPA2-PSK*** functionality. This layered architecture is illustrated in Figure 22.

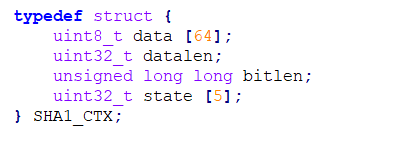


**Figure 22: Layered Software Implementation of WPA2-PSK Authentication**

#### **3.3.1 Layer1: SHA1-HASH Implementation**

This layer dealt with all the functions related to the implementation of ***SHA1*** Hash algorithm. The information regarding relevant data types and the function prototypes for all the functions in this layer are given below:

* **Data Type**: SHA1\_CTX



This data type was used to maintain information relevant to an iteration of an ***SHA1***-Hash process. It held information regarding 512-bits of input block, 160-bits of output hash, data length and bit length. This data type was used to pass information to and store information from all the functions defined in the ***SHA1***-Hash Layer.

* **Function Prototype 1**: SHA1Init()



This Function was used to initialize a new context for the ***SHA1***-Hash process. It initialized datalen and bitlen to 0. It also initialized the states of the context to the initial Hash values of 0x67452301 ,0xEFCDAB89, 0x98BADCFE,

0x10325476 and 0xC3D2E1F0.

* **Function Prototype 2**: SHA1Update()



This function was used to update data, data length and bit length for the context of ***SHA1***. If the data length was 64 bytes (512 bits) i.e. input block size of ***SHA1***, we started the ***SHA1***-Hash process by calling SHA1Transform().

* **Function Prototype 3**: SHA1Final()



This was the function where the initial processing before the actual ***SHA1***-Hash processing was done. The initial padding and the appending of the data length to the input block was done in this function. After the initial processing, it called SHA1Transform () to perform the ***SHA1***-Hash algorithm.

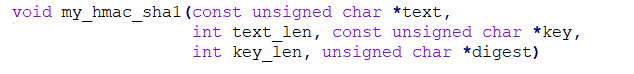
* **Function Prototype 4**: SHA1Transform()



This was the main function where the processing part of the actual ***SHA1***-Hash Algorithm was implemented. First, the 32-bit words of the 512-bit input block were stored into initial 16 arrays of size 32-bits. W (16) to W (79) values were then calculated from these values. 80 rounds of ***SHA1***- Hash transform was performed on the data to get the 160-bits of ***SHA1***-Hash Output value. Finally, the new Hash for the context was updated with the new output Hash value.

#### **3.3.2 Layer2: HMAC\_SHA1 Implementation**

This layer dealt with the function related to the implementation of ***HMAC-SHA1***. It contained a single function that called functions defined in ***SHA1***-Hash algorithm. The prototype for this function is given below:



This single function was used for the ***HMAC*** operation over ***SHA1***-Hash. The first step of the code performed the initial padding and appending operations required for ***HMAC*** operation. After this initial process, this function would successively call SHA1Init (). SHA1Update (), SHA1Final () and SHA1Transform ().

#### **3.3.3 Layer3: PBKDF2 Implementation**

This layer dealt with the function related to the implementation of ***PBKDF2*** operation. It contained a single function that called my\_hmac\_sha1() for 4096 iterations after some initial processing. The prototype for this function is given below:

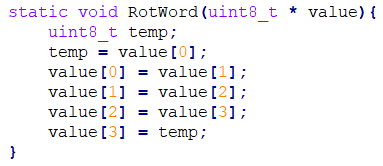


256-bit PMK was derived from this function which was used as the first key in AES encryption. The application code would use this function to obtain the PMK from SSID-Passphrase combination, which is used to authenticate a wireless device using ***WPA2-PSK*** before starting the wireless communication.

### **3.4 AES-256 Key Expansion**

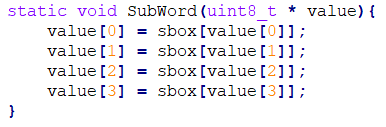
The code for ***AES-256*** key expansion was written using C programming language and for 32-bit soft processor MicroBlaze. The information regarding relevant function definitions for all the functions used for key expansion are given below:

* **Function Definition 1**: RotWord()



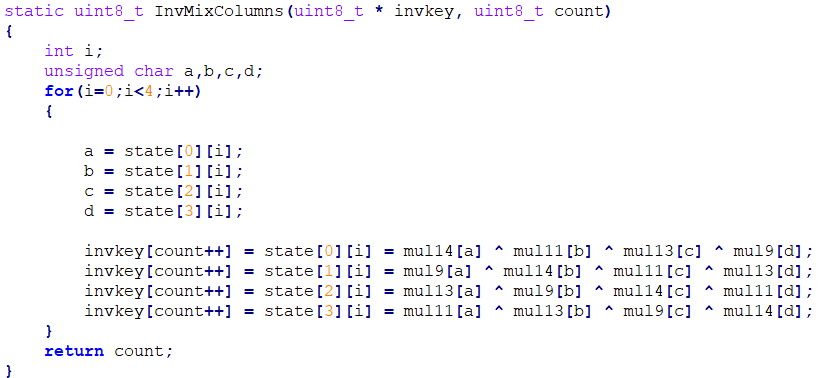
This function took an array which contained 4 one-byte values [a0,a1,a2,a3] as input, performed a cyclic permutation, and returned [a1,a2,a3,a0] as the output.

* **Function Definition 2**: SubWord()



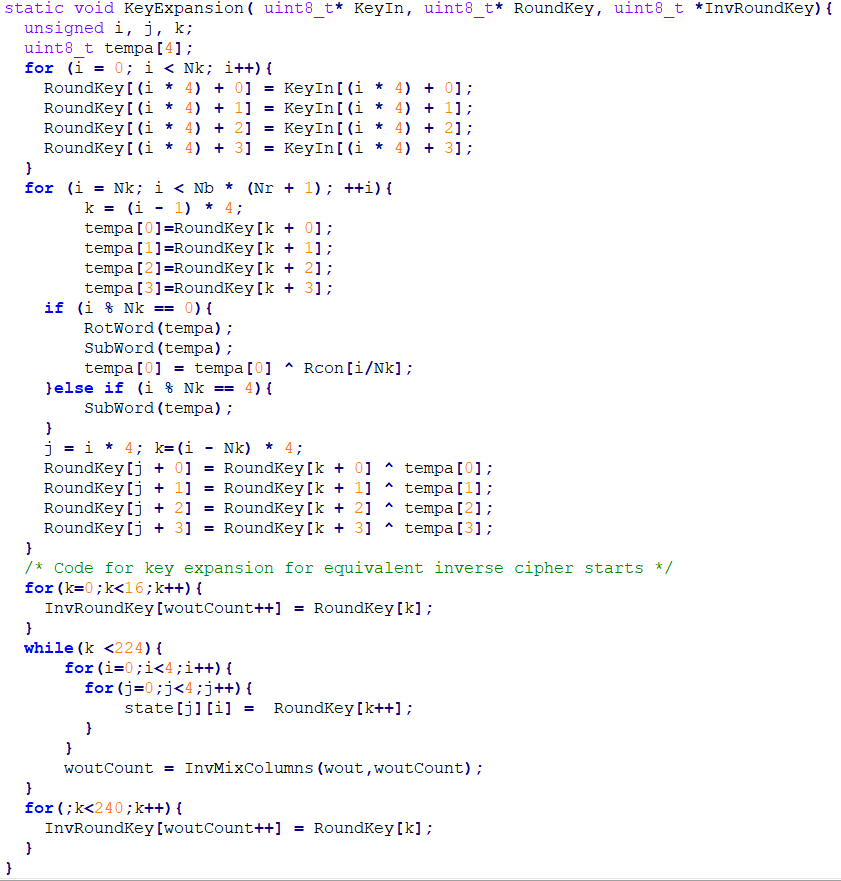
This function took an array which contained 4 one-byte values [a0,a1,a2,a3] as input and returned S-Box substituted values of the array as the output. The code snippet for S-Box lookup table implementation in C is shown in Figure 29 (Appendix1.2).

* **Function Definition 3**: InvMixColumns()



This function took an array containing 16 one-byte values [a0,a1,a2 … a15] and number of bytes of completed expanded key for equivalent inverse cipher as an input, performed Inverse Mix Column transformations, and returned the transformed values and new count of the completed expanded key as the output. In this function definition , state was a global 4x4 one-byte array. The code snippets for mul9, mul11, mul13 and mul14 lookup tables implementations in C are shown in Figure 30, Figure 31, Figure 32 and Figure 33 respectively in Appendix1.2.

* **Function Definition 4**: KeyExpansion()



This function took an array which contained 32 bytes of PMK as input, expanded it to 240 bytes each of expanded key for encryption and decryption, and returned these expanded keys as the output. These expanded keys were stored in BRAM of the FPGA. Chapter 4 describes how these keys were arranged in the BRAM after the expansion process, retrieved from the BRAM, and how they would be used in the hardware implementation of the ***AES-256*** module to encrypt and decrypt data. Chapter 4 also shows how the encrypted/decrypted data were stored in BRAM after the transformation.

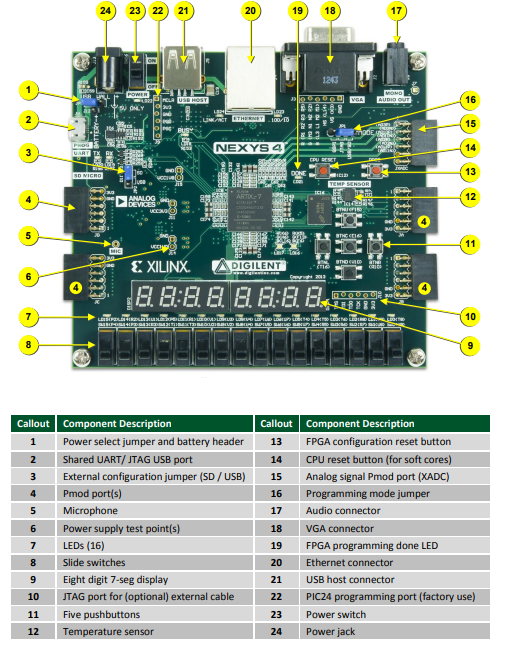
# **Chapter 4: Hardware Implementation**

## **4.1 Overview**

Encryption and decryption on data with ***AES-256*** module was implemented on hardware using Verilog HDL. This design was targeted for Nexys4 FPGA board. The Verilog Code was written on Xilinx Vivado Design Suite 2017.2 IDE. After ***AES-256*** module designed using Verilog HDL was synthesized and implemented, the design was packaged into an IP core using Vivado IP Packager tool. This ***AES-256*** IP core was then interfaced to MicroBlaze softcore processor with the help of AXI Interconnect.

## **4.2 Nexys4**

Nexys4 board is a development platform based on the latest Artix-7™ (Xilinx part number XC7A100T-1CSG324C) Field Programmable Gate Array (FPGA) from Xilinx [19]. The Artix-7 FPGA is designed for high performance and it features 15850 logic slices (each with 6-input LUTs and 8 flip-flops), 240 DSP slices and 4860 KB of fast block RAM [20]. Nexys4 has generous external memories, and collection of USB, Ethernet, and other ports, and can host designs ranging from introductory combinational circuits to powerful embedded processors [19]. It also has several built-in peripherals such as an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and a lot of I/O devices [19]. Nexys4 board with its component description is shown in Figure 23.



**Figure 23: Nexys4 Board Features**

## **4.3 AXI Interconnect**

Advanced extensible Interface (AXI) is a part of the Arm Advanced Microcontroller Bus Architecture (AMBA) specification that provides the interface between the processing system and programmable logic sections of the chip [21]. The AXI specifications describe an interface between a single AXI master and a single AXI slave, representing IP cores that exchange information with each other [22]. Memory mapped AXI masters and slaves can be connected using a structure called an Interconnect block [22]. The Xilinx AXI Interconnect IP contains AXI-compliant master and slave interfaces and can be used to route transactions between one or more AXI masters and slaves [22] . AXI Interconnect connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices [23]. When connecting one master to one slave, the AXI Interconnect core can perform address range checking. Also, it can perform any of the normal data-width, clock rate, or protocol conversions and pipelining [23]. When not performing any conversions or address range checking, the AXI Interconnect core is implemented as wires, with no resources, no delay and no latency [23].

## **4.4 AES-256 Implementation**

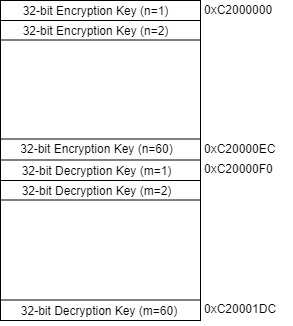
This hardware implementation of ***AES-256*** primarily contained 2 components: BRAMs (Key BRAM and Data BRAM) and AES core. BRAMs were used to store expanded keys, plaintext and ciphertext ,whereas, the AES core had modules for encryption and decryption blocks that used the expanded key, plaintext and ciphertext for encryption and decryption.

Since block size of AES is 128 bits (16 Bytes), both encryption and decryption processes worked on 128-bits (16 Bytes) of data at a time. If the data to be transformed was less than 128 bits, then, it was padded with trailing 0’s to make it 128-bit block before being transformed. If the data to be transformed was greater than 128 bits but not a multiple of 128 bits, then, it was also padded with trailing 0’s until we had a data block which is multiple of 128 bits. After padding, the encryption/decryption was done on one 128-bit block of data at a time. The data to be transformed was initially stored in a certain location of Data BRAM. At the beginning of the transformation, they were read from the Data BRAM and passed to the AES core. After the transformation was completed, the converted data was stored in a separate location of the Data BRAM that was allocated for the transformed data.

### **4.4.1 BRAMs**

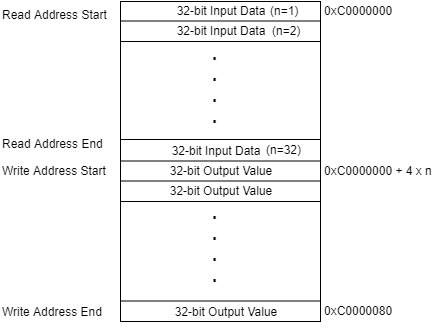
BRAMs are blocks of 32-bit memory locations used to store expanded key, original data to be encrypted or decrypted, and transformed data after encryption or decryption . It is a synchronous memory block with 32-bit data being clocked in or out at every clock. Two instances of BRAMs were created for the implementation of ***AES-256*** module: Key BRAM and Data BRAM.

* **Key BRAM**: The Data width of the Key BRAM was 32-bits. The expanded keys were stored in Key BRAM. 4 Bytes of expanded keys were stored per BRAM location. Hence, 60 memory locations were used to store 240 bytes of expanded key for encryption. Similarly, further 60 memory locations were used to store the other 240 bytes of expanded key for decryption. These keys were only read once from the BRAM at the beginning of the encryption/decryption process and saved into a temporary buffer. For multiple blocks of encryption/decryption, the keys were accessed directly from the temporary buffer instead of BRAM. The start address for Key BRAM was 0xC2000000. The overall memory organization for Key BRAM locations used in the design of ***AES-256*** module is shown in Figure 24.



**Figure 24: Key BRAM Organization**

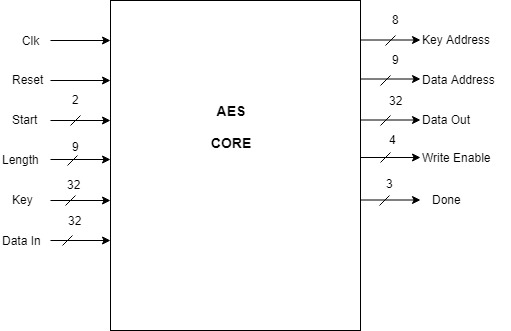
* **Data BRAM**: It was a Read/Write Memory block where original data and data to be transformed were stored. The Data width of the Data BRAM was 32-bits. 32 such memory locations were allocated each for original data and transformed data. Hence, 128 bytes of BRAM memory was used for storing original and transformed data. The start address for Data BRAM was 0xC0000000. The overall memory organization for Data BRAM is shown in Figure 25.



**Figure 25: Data BRAM Organization**

### **4.4.2 AES Core**

The block diagram of top level of ***AES-256*** core implementation is shown in Figure 26. It has the following port definitions:



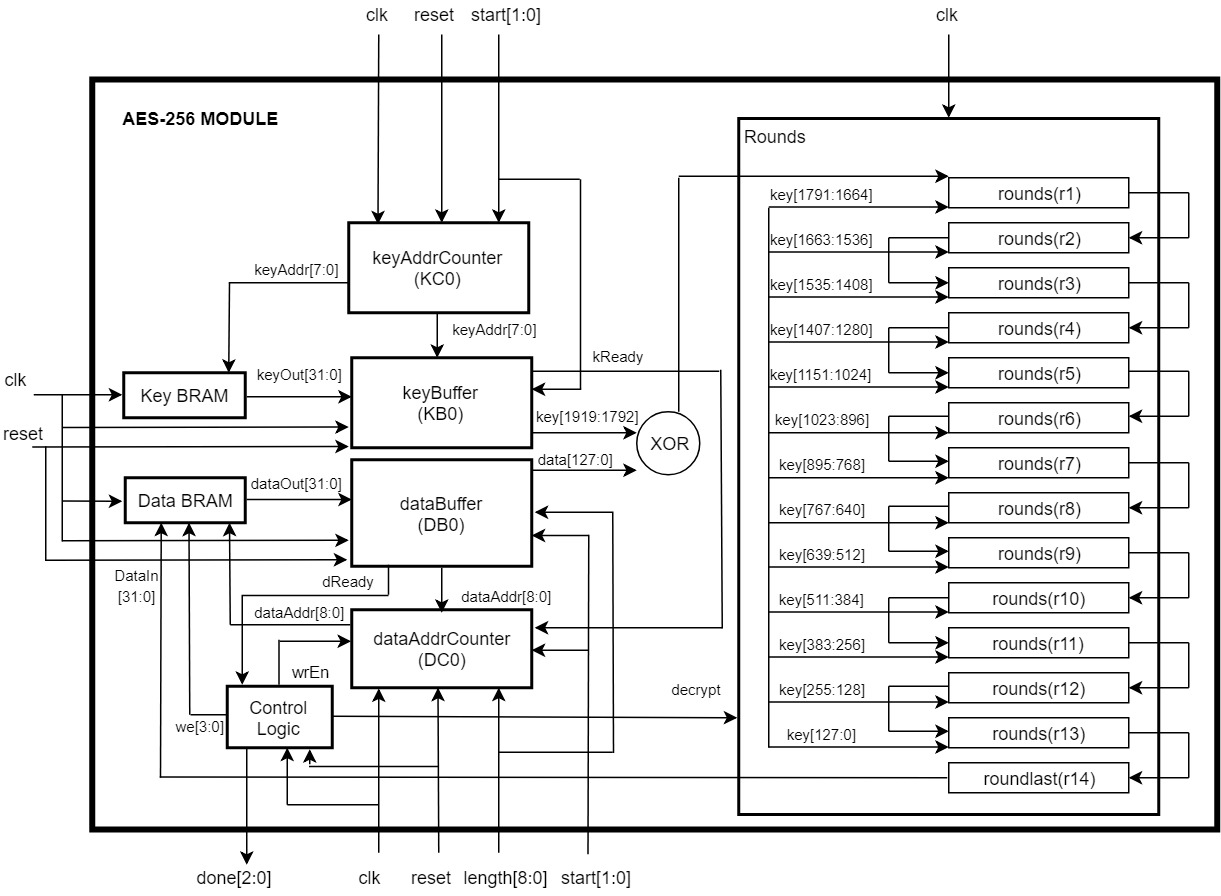
**Figure 26: AES Core with Input and Output Signals**

* **Input Ports**:
  + **Clk**: It was the clock source to the AES core. The frequency of clock used in the design was 100 Mhz.
  + **Reset**: It was the synchronous reset signal to the AES core. It worked on active low logic. The reset signal was controlled by bit 11 of Slave register0 from the AXI BUS of MicroBlaze.
  + **Start**: It was a 2-bit wide input signal which was used to start the encryption or decryption signal. If **Start** = 2’b01, the AES core would perform the encryption operation, if **Start** = 2’b10, the AES core would perform the decryption operation. All other possible values of **Start** signal were DON’T CARE cases. **Start** signal was controlled by bits[1:0] of Slave register0 from the AXI BUS of MicroBlaze.
  + **Length**: It was a 9-bit input value which signified the number of 32-bit input data( after zero padding to make it multiple of 128-bit ) to be either encrypted or decrypted. The **Length** signal was controlled by bits[10:2] of Slave register0 from the AXI BUS of MicroBlaze.
  + **Key**: It was a 32-bit expanded key input to the AES core. This input port was connected to 32-bit Data output port of Key BRAM.
  + **Data In**: It was a 32-bit plaintext or ciphertext to be encrypted or decrypted respectively. This input port was connected to 32-bit Data output port of Data BRAM.
* **Output Ports**
  + **Key Address**: It was a 32-bit output value which denoted the memory location of the Key BRAM from which the expanded key was to be retrieved during the encryption/decryption process. On reset the value of **Key Address** was 0. This output port was connected to 32-bit address input port of Key BRAM.
  + **Data Address**: It was a 32-bit output value which denoted the memory location of the Data BRAM from which the expanded key was to be retrieved during the encryption/decryption process. On reset the value of **Data Address** was 0. This output port was connected to 32-bit address input port of Data BRAM.
  + **Data Out**: It was a 32-bit output value which signified ciphertext in case of encryption operation and plaintext in case of decryption operation. On reset the value of **Data Out** was 0. This output port was connected to 32-bit Data input port of Data BRAM.
  + **Write Enable**: It was a 4-bit output signal that was used to enable the write operation of the ciphertext in case of encryption or plaintext in case of decryption, to the allocated memory locations in the Data BRAM. On reset the value of **Write Enable** was 0. This output port was connected to 4-bit Write Enable input port of Data BRAM.
  + **Done**: It was a 3-bit output signal that showed the completion of the AES operation. The value of **Done** signal would become 3’b111 after the AES transformation was completed and the new data was written into the allocated memory locations in the Data BRAM. On reset the value of **Done** was 0. This output signal was connected to bits[2:0] of Slave register1 from the AXI BUS of MicroBlaze.

This implementation took 41 clock cycles from the start signal to complete the encryption/decryption of the first 128-bit (16-Byte) block of data. After the first block of data was transformed, it only took 4 further clock cycles per block, for the other blocks of input to be encrypted or decrypted. This implementation supported transformation of 8 blocks of input data at time i.e. the depth of the input data buffer and output data buffer was 128 bytes.

### **4.4.3 AES-256 Internal Design**

Internally, ***AES-256*** module contained several sub-modules. The internal design of ***AES-256*** with the help of these sub-modules is shown in Figure 27.



**Figure 27: Internal Design of AES-256**

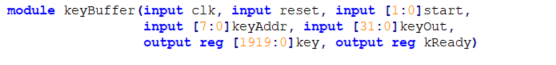
The information regarding prototypes for various sub-modules of ***AES-256*** are given below :

* **Module Prototype 1**: keyAddrCounter()



This module was used to keep track of read address for Key BRAM. If the value of the input signal start was 1, then the read address would start from the memory location containing the first encryption keys. If the value of the input signal start was 2, then the read address would start from the memory location containing the first decryption keys.

* **Module Prototype 2**: keyBuffer()



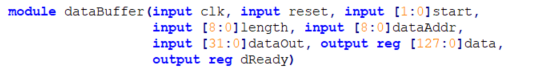
This module was used to buffer the expanded key values from the Key BRAM to internal buffer. If the value of the input signal start was 1, then the module would buffer expanded keys for encryption. If the value of the input signal start was 2, then the module would buffer expanded keys for decryption. After the buffering is completed, it would always send a kReady = 1 signal.

* **Module Prototype 3**: dataAddrCounter()



This module was used to keep track of read and write address for Data BRAM. If the value of the input signal wrEn was 0, then the read address would start from the memory location containing the first input data. If the value of the input signal wrEn was 1, then the write address would start from the memory location for the first output data.

* **Module Prototype 4**: dataBuffer()



This module was used to buffer the input data from the Data BRAM to internal buffer. It could buffer up to 128 bytes of data at time. After the buffering was completed, it would send a dReady = 1 signal.

* **Module Prototype 5**: rounds()



Within this module, AddRoundKey (), SubBytes (), ShiftRows (), MixColumns ()InvAddRoundKey (), InvSubBytes (), InvShiftRows () and InvMixColumns () transformations were implemented. In this implementation, SubBytes () and ShiftRows(), and, InvSubBytes () and InvShiftRows ()were implemented inside a single block and within in a single clock. If the input signal decrypt = 0, then this module would perform encryption, and if decrypt = 1, this module would perform decryption. 13 instances of rounds()were instantiated each for encryption and decryption. These 13 instances were cascaded to each other and the last instance was cascaded to the roundlast() module.

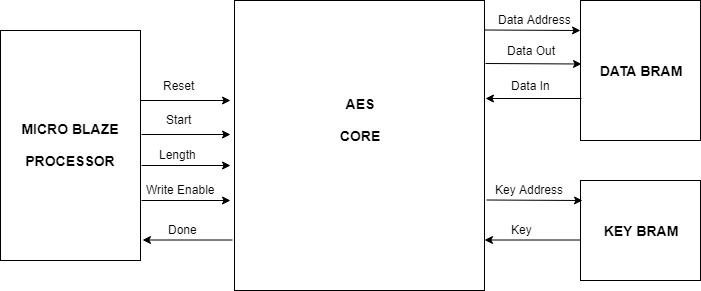
* **Module Prototype 6**: roundlast()



Within this module, AddRoundKey (), SubBytes (), ShiftRows (), InvAddRoundKey (), InvSubBytes () and InvShiftRows transformations were implemented. In this implementation, SubBytes () and ShiftRows(), and, InvSubBytes () and InvShiftRows ()were implemented inside a single block and within in a single clock. If the input signal decrypt = 0, then this module would perform encryption, and if decrypt = 1, this module would perform decryption. Hence, the output of this module would be the encrypted data for input signal decrypt = 0,and, the output of this module would be the decrypted data for input signal decrypt = 1.

## **4.5 AES-256 Interface with MicroBlaze**

The implemented ***AES-256*** module was packaged into a custom IP core using Vivado IP Packager tool. This ***AES-256*** IP core was connected to MicroBlaze using AXI Interconnect. In this interface, MicroBlaze was the master device and ***AES-256*** IP core was the slave device. The overall block diagram for interfacing ***AES-256*** IP core and MicroBlaze processor is shown in Figure 28.



**Figure 28: Block Diagram of AES-256 IP Core with MicroBlaze**

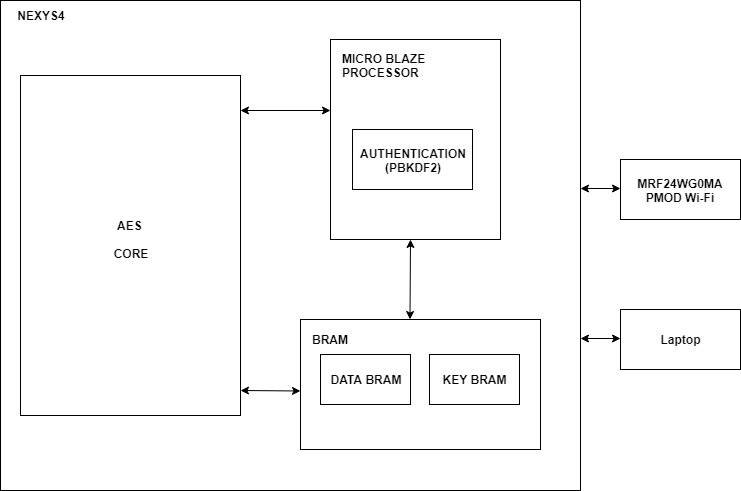
# **Chapter 5: Testing and Result**

## **5.1 Overview**

The goal of this thesis was efficient software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** using C programming language, and, efficient hardware implementation of ***AES-256*** cipher using Verilog HDL. ***PBKDF2*** was used for device authentication, while ***AES-256*** was used for data confidentiality, in the ***Wi-Fi*** communication. To test the validity of the goals achieved by the implementations described in Chapter 3 and Chapter 4, test setup illustrated by the block diagram shown in Figure 29 was arranged. The setup has 3 main components:

* Nexys4 board with code for the implemented hardware and software design.
* MRF24WG0MA PMOD ***Wi-Fi***
* Laptop

For testing software implementation of ***PBKDF2*** based on ***HMAC-SHA1***, a single setup shown in Figure 29 was used. For testing hardware implementation of ***AES-256*** cipher, two setups shown in Figure 29 were used. Additionally, a wireless access point was created using a mobile hotspot to complete the test setup for both the tests. The mobile hotspot and wireless access point will be used interchangeably in this chapter.

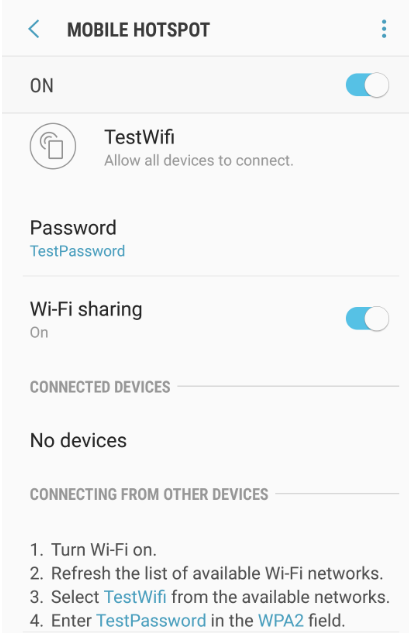


**Figure 29: Overall Block Diagram for Testing**

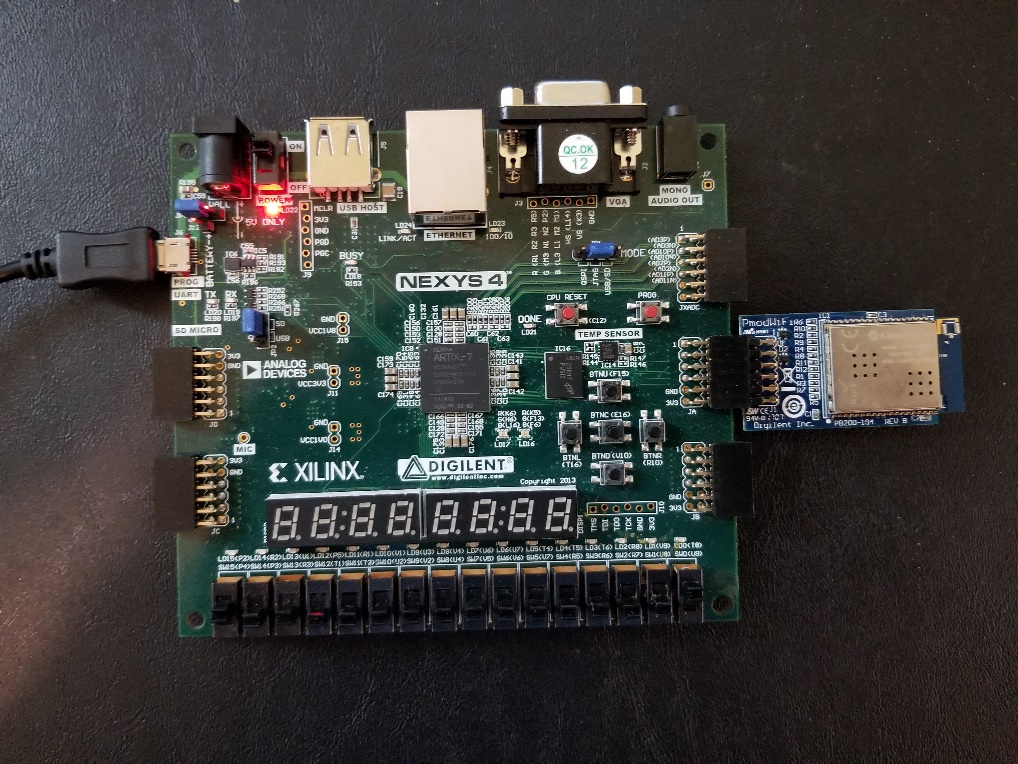
## **5.2 WPA2-PSK Testing and Result**

The test for authentication of ***Wi-Fi*** using ***WPA2-PSK*** was performed with mobile hotspot as the access point. The access point was configured with “**TestWifi**” as the SSID and “**TestPassword**” as the Passphrase (Figure 30). The wireless module (MRF24WG0MA PMOD ***Wi-Fi***) interfaced with Nexys4 board (Figure 31) running the implemented design was used as the end node to connect to the access point. The Nexys4 board had USB-to-UART module running on it. The board was connected to a serial terminal software called **TERA TERM,** which opened a serial COM port with settings:

* Baud rate : 115200 bps
* Data size : 8- bit
* Parity : None
* Stop bits: 1-bit
* Flow Control: None



**Figure 30: Mobile Hotspot Configuration**



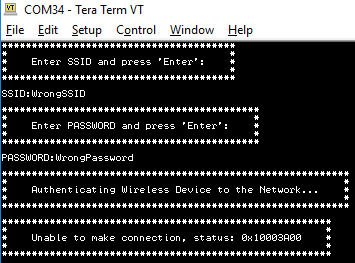
**Figure 31: Nexys4 Board with MRF24WG0MA PMOD Wi-Fi**

When this test setup connected to the serial term software was running, the values for SSID and Passphrase were prompted to the user on the terminal screen. The user would then type the values of SSID and Passphrase on the terminal. 4 test cases of SSID and Passphrase were used to check the validity of the authentication process. A table containing these 4 test cases along with the results of the tests is shown in Table 1.

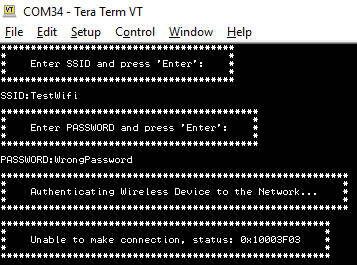
**Table 1: Table of Test Cases and Results for Authentication**

|  |  |
| --- | --- |
| **CASE** | **RESULT** |
| Incorrect SSID and Incorrect Passphrase | Authentication Failed |
| Correct SSID and Incorrect Passphrase | Authentication Failed |
| Incorrect SSID and Correct Passphrase | Authentication Failed |
| Correct SSID and Correct Passphrase | Authentication Successful |

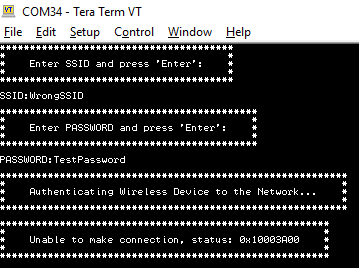
When incorrect values of SSID and/or Passphrase were entered in the end nodes using the terminal software, their authentication failed, and a communication channel was not created (Figure 32, Figure 33 and Figure 34). When SSID and Passphrase information of the access point were correctly entered in the end node, the access point was able to successfully authenticate it (Figure 35). After successful authentication , the access point was able to create a communication channel between itself and the end node.



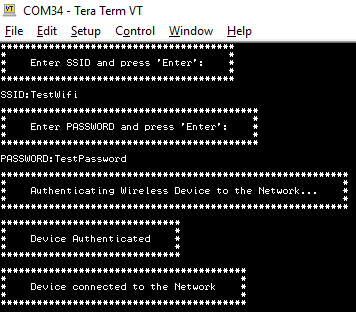
**Figure 32: Failed Authentication with Incorrect SSID and Incorrect Password**



**Figure 33: Failed Authentication with Correct SSID and Incorrect Password**



**Figure 34: Failed Authentication with Incorrect SSID and Correct Password**



**Figure 35: Successful Authentication with Correct SSID and Correct Password**

## **5.3 WPA2-PSK Performance Evaluation**

The performance of ***PBKDF2***, ***HMAC*** and ***SHA1*** operations done using the implemented software described in Chapter 3 was compared with the existing design used in the MRF24WG0MA PMOD ***Wi-Fi*** software library. Latency (us) was used as the performance metric for all 3 operations**.** Latency was defined as the time (in us) required to complete the given operation.

To determine the latency of an operation, initially, an AXI timer with a resolution of 10 ns was configured. Then, the following steps were performed:

* **Step1**: Timer was started.
* **Step2**: Timer value was read.
* **Step3**: The operation to be profiled was started.
* **Step4**: Timer was stopped after the completion of the function
* **Step5**: The Timer value was read.

These 5 steps were performed for ***SHA1***, ***HMAC*** and ***PBKDF2*** operations. The comparison of the latency results for these 3 operations is shown in Table 2.

**Table 2: Latency Comparison of Functions used in WPA2-PSK Authentication**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Latency(us) in this implementation** | **Latency(us) in existing code from library** |
| SHA1Transform() | 184.63 | 1404.27 |
| my\_hmac\_sha1() | 767. 36 | 2546.47 |
| pkcs5\_pbkdf2() | 4.96 | 10.04 |

In Table 2, the latency of 3 main operations: ***SHA1***, ***HMAC*** and ***PBKDF2*** were tabulated. In all the 3 operations, the values of the implemented design described in Chapter 3 was lesser compared to the values of the existing design from the MRF24WG0MA PMOD ***Wi-Fi*** software library. From these results, we can see that the implemented design was more efficient in terms of latency as compared to the design in MRF24WG0MA PMOD ***Wi-Fi*** software library.

## **5.4 AES-256 Testing and Result**

The test for data confidentiality in ***Wi-Fi*** using ***AES-256*** was performed with mobile hotspot as the access point. This access point was configured with “**TestWifi**” as the SSID and “**TestPassword**” as the Passphrase (Figure 30). Two setups shown in Figure 31 were used as end nodes to connect to the access point. When SSID and Passphrase information of the access point were correctly entered in the two end nodes, the access point was able to successfully authenticate them (Figure 35). Data was communicated between these nodes using TCP protocol through the access point. The following two tests were conducted to verify data confidentiality using ***AES-256***:

### **5.4.1 TCP Server and TCP Client**

In this test setup, one of the wireless end nodes was coded to run as a TCP server while the other node was coded to run as a TCP client. Both end nodes were authenticated when correct values of SSID and Passphrase were entered. After authentication, the TCP server opened a socket listening at address 192.168.43.7:80. The TCP Client would then try to connect to the server. After a successful TCP connection, encrypted data was transferred from the client to the server (Figure 36).



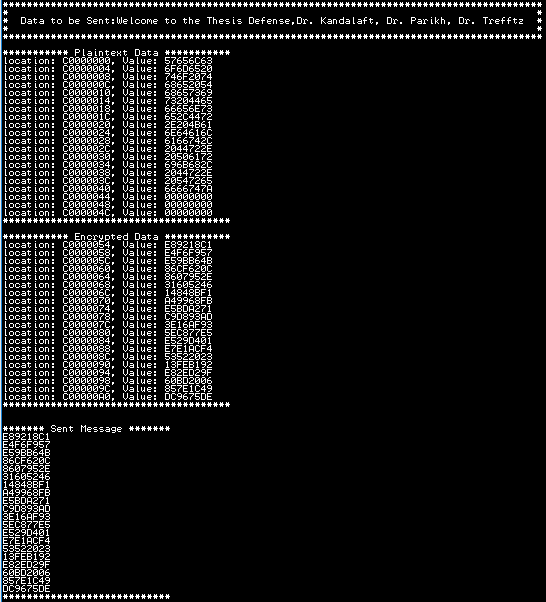
**Figure 36: Encrypted Data Sent by TCP Client to TCP Server**

When the server received the encrypted data, it would decrypt it. In Figure 37, the server has received the encrypted data and successfully decrypted it to obtain the original data sent by the client.



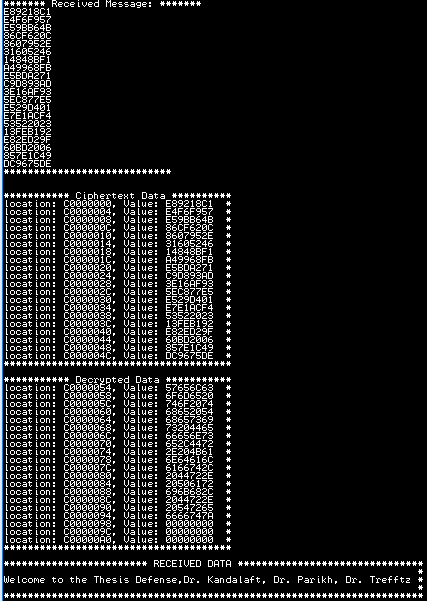
**Figure 37: Decryption of Data Received by TCP Server from TCP Client**

Now, the server would append additional data to decrypted data, encrypt this new data, and send this encrypted data back to the client (Figure 38).



**Figure 38: Encrypted Data Sent by TCP Server to TCP Client**

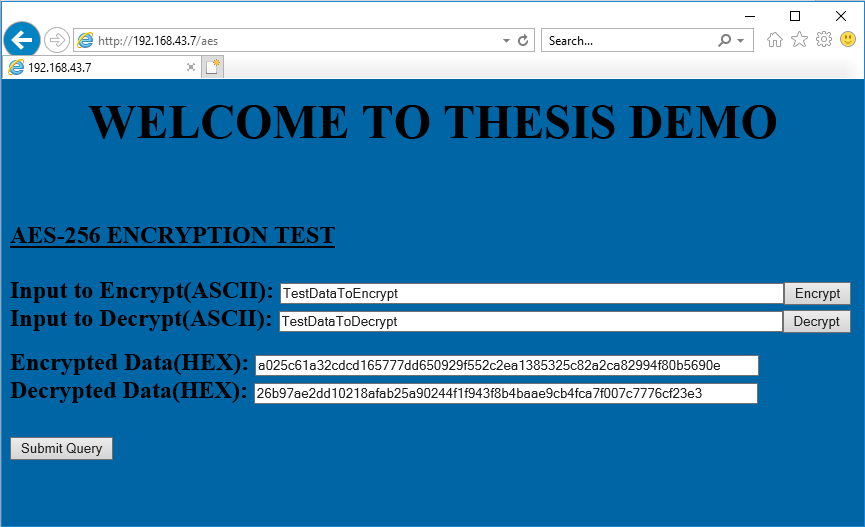
When the client received the encrypted data, it would decrypt it. In Figure 39, the client has correctly received the encrypted data and successfully decrypted it to obtain the original data sent by the server.



**Figure 39: Decryption of Data Received by TCP Client from TCP Server**

### **5.4.2 HTTP Server and Web Browser**

In this test setup, one of the wireless end nodes was coded to run as a HTTP server which could serve HTTP GET requests. This server was able to serve two webpages with URL **192.168.43.7/aes** and **192.168.43.7/config**. At the beginning, this device was first authenticated by entering correct values of SSID and Passphrase. After authentication, the HTTP server opened a socket listening at address 192.168.43.7:80. Now, a test laptop was connected to the access point and a web browser application was run on it. When the address [**http://192.168.43.7/aes**](http://192.168.43.7/aes)was entered as the URL, the web browser would show the webpage seen in Figure 40.



**Figure 40: AES Webpage hosted by HTTP server**

In the webpage shown in Figure 40, **Input to Encrypt(ASCII)** field and **Input to Decrypt(ASCII)** field were filled with test data. When the **Encrypt** and **Decrypt** buttons were pressed in the webpage, the **Encrypted Data(HEX)** and **Decrypted Data(HEX)** fields were filled respectively. Finally, when the **Submit Query** button was pressed, the data in the **Encrypted Data(HEX)** field and **Decrypted Data(HEX)** field were transferred to the HTTP server using HTTP protocol.

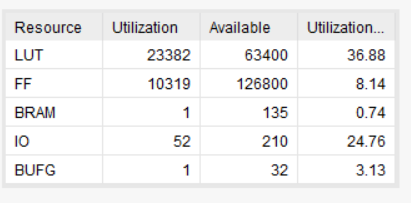
In Figure 41, the encrypted and decrypted set of data received by the HTTP server that was sent from the web browser is shown. The results of the decryption of the encrypted data ,and encryption of decrypted data is also shown in Figure 41. It can be observed that the original data entered in **Input to Encrypt(ASCII)** field and **Input to Decrypt(ASCII)** field from webpage in Figure 40 matches with the decrypted and encrypted values from Figure 41.



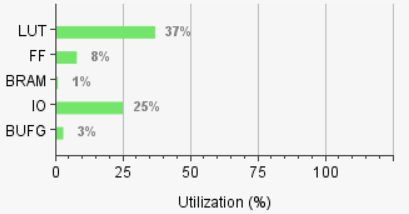
**Figure 41:Decryption and Encryption of Data received by HTTP server**

## **5.5 AES-256 Performance Evaluation**

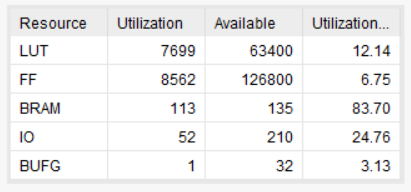
With reference to Figure 27 in Chapter 4, two different logics were written for AddRoundKey (), SubBytes (), ShiftRows (), MixColumns ()InvAddRoundKey (), InvSubBytes (), InvShiftRows () and InvMixColumns () transformations in the implementation of ***AES-256*** core. The first logic used the Look Up Table (LUT) implementation, whereas the second logic used the BRAM implementation. In the LUT implementation, the transformations were performed asynchronously, while in BRAM Implementation, the transformations were performed synchronously. The resource utilization for LUT implementation is shown in Figure 42 and Figure 43 , and resource utilization for BRAM implementation is shown in Figure 44 and Figure 45.



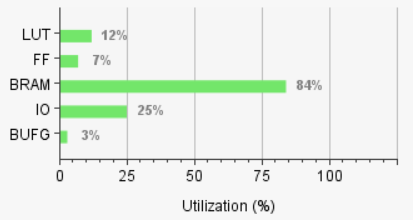
**Figure 42: Resources Utilization Table for LUT Implementation**



**Figure 43: Graph for Resource Utilization for LUT Implementation**



**Figure 44: Resources Utilization Table for BRAM Implementation**



**Figure 45: Graph for Resource Utilization for BRAM Implementation**

From Figures 42,43,44 and 45, we see that **LUT** and **FF** utilization is greater in the **LUT** implementation as compared to the **BRAM** implementation. Whereas, the **BRAM** utilization is lesser in **LUT** implementation in comparison to the **BRAM** implementation. The **IO** and **BUFG** utilization are same in both cases. The **LUT** utilization is greater in **LUT** implementation because, when transformations are implemented asynchronously, the memory in the design will be inferred as a lookup table. Whereas, the **BRAM** utilization is greater in **BRAM** implementation because, when transformations are implemented synchronously, the memory in the design will be inferred as a Block RAM. Hence, there is a trade-off between utilization of look up tables and BRAMs in the two designs.

In this thesis, the **LUT** implementation of ***AES-256*** was arbitrarily selected to be converted to the ***AES-256*** IP core and be interfaced with MicroBlaze Processor. The performance of this implementation was compared with the other existing implementations described in [24], [25], [26], [27]. For the purpose of comparison in the following sections in Chapter 5, the implemented design of AES-256 using **LUT** is referred as Design 1, and implementations referenced from [24], [25], [26], [27] are referred as Design 2, Design 3, Design 4 and Design 5.

### **5.5.1 Latency Comparison**

Latency was defined as the time taken by the implemented design to correctly produce one block of 128-bit output data for one block of 128-bit input data. This value depends on the number of clock cycles per encrypted/decrypted block and the frequency of the clock used . Latency was manually calculated using equation (24) and the values are tabulated in Table 3.

Latency = Clock Cycles per Output Block x Clock Frequency ...(24)

**Table 3: Comparison of Implemented Designs Based on Latency**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Design** | **Clock**  **Frequency**  **(MHz)** | **Clock Cycles per Encryption**  **Block** | **Clock Cycles per Decryption**  **Block** | **Latency (ns)**  **For**  **Encryption** | **Latency (ns)**  **For**  **Decryption** |
| Design 1 | 100 | 41 | 41 | 410 | 410 |
| Design 2 | 114.877 | 139 | 139 | 1209.981 | 1209.981 |
| Design 3 | 107.3 | 115 | 115 | 1071.761 | 1071.761 |
| Design 4 | - | 28 | 41 | 132.5 | 197.5 |
| Design 5 | 95.721 | 15 | 23 | 185.52 | 284.264 |

In Table 3, it can be seen that for Designs 1,2 and 3, the numbers of clock cycles required to encrypt a block of data is same as the number of clocks required to decrypt a block data. For Designs 4 and 5, the clock cycles are different for encryption and decryption process. It is seen that Design 1 had less latency in comparison to Design 2 and Design 3, while it wasn’t able to match or better the latency results from Design 4 and Design 5.

### **5.5.2 Throughput Comparison**

Throughput was manually calculated using equation (25) and the values are tabulated in Table 4.

In equation (25), the block size is 128-bit.

Throughput = Block Size / Latency ................................ (25)

**Table 4: Comparison of Implemented Designs Based on Throughput**

|  |  |  |
| --- | --- | --- |
| **Design** | **Throughput for Encryption (Gb/s)** | **Throughput for Decryption (Gb/s)** |
| Design 1 | 0.312 | 0.312 |
| Design 2 | 0.106 | 0.106 |
| Design 3 | 0.119 | 0.119 |
| Design 4 | 0.91 | 0.62 |
| Design 5 | 0.689 | 0.45 |

In Table 4, it can be seen that for Designs 1,2 and 3, the throughput for encryption is same as the throughput for decryption . For Designs 4 and 5, the throughputs are different for encryption and decryption. It is also observed that Design 1 had higher throughput as compared to Design 2 and Design 3,but, had a lower throughput in comparison to Design 4 and Design 5.

### **5.5.3 Resource Utilization Comparison**

The LUT slice utilization for the 5 implemented designs are shown in Table 5.

**Table 5: Comparison of Implemented Designs Based on Resource Utilization**

|  |  |  |
| --- | --- | --- |
| **Design** | **Total Slices for Encryption** | **Total Slices for Decryption** |
| Design 1 | 23382 | 23382 |
| Design 2 | 1737 | 1737 |
| Design 3 | 1428 | 1428 |
| Design 4 | 15376 | 20324 |
| Design 5 | 76365 | 76365 |

In Table 5, it can be viewed that for Designs 1,2 3 and 5, the LUT slice utilization for encryption is same as the LUT slice utilization for decryption. For Design 4 , the LUT slice utilization is different for encryption and decryption. It is also observed that Design 1 had lesser LUT slice utilization as compared to Design 5, but, had a higher LUT slice utilization in comparison to Designs 2,3 and 4.

### **5.5.4 Efficiency Comparison**

Efficiency was manually calculated using equation (26) and the values are shown in

Table 6.

Efficiency = Throughput / Number of LUT slices ................... (26)

**Table 6: Comparison of Implemented Designs Based on Efficiency**

|  |  |  |
| --- | --- | --- |
| **Design** | **Efficiency for Encryption (Kb/s per slice)** | **Efficiency for Decryption (Kb/s per slice)** |
| Design 1 | 13.344 | 13.344 |
| Design 2 | 61.025 | 61.025 |
| Design 3 | 83.333 | 83.333 |
| Design 4 | 59.183 | 30.505 |
| Design 5 | 9.022 | 5.892 |

In Table 6, it can be seen that the efficiencies for encryption in Designs 1,2 and 3 are same as the efficiencies for decryption. For Designs 4 and 5, the efficiencies are different for encryption and decryption. It is also seen that Design 1 had better efficiency as compared to Design 5, but, had a lower efficiency when compared to Designs 2,3 and 4.

### **5.5.5 Summary**

From the results of different performance metrics seen in Tables 3,4,5 and 6, it can be inferred that there is a trade off between latency and LUT resource utilization. These two metrics seem to be inversely proportional to each other. Hence, an efficient design would contain a happy medium between latency and LUT resource utilization values. If we observe the formula to calculate latency in equation (24), we see that by using a larger clock frequency we can achieve lower latency values for the same number of clock cycles. Hence, using a FPGA board with a higher clock source could be a method to achieve better latency without increasing the LUT resource utilization. This in turn will improve the throughput and efficiency results as well. Another way to improve the efficiency would be to reduce the LUT resource utilization by using most of the Block RAM(BRAM) available in FPGA chip. FPGA chips have limited BRAMs in comparison to look up table. By selecting a FPGA chip supporting large amount of BRAMs and writing Verilog code in a manner that the memory locations infer BRAM instead of LUT, we can reduce the LUT resource utilization without increasing the latency. Hence, this will improve the overall efficiency of the design.

# **Appendices**

## **1.1 AES Lookup Tables**

**Table 7: S-Box Lookup Table**



**Table 8: Inverse S-Box Lookup Table**



**Table 9: Mul2 Lookup Table**



**Table 10: Mul3 Lookup Table**



**Table 11: Mul9 Lookup Table**



**Table 12: Mul11 Lookup Table**



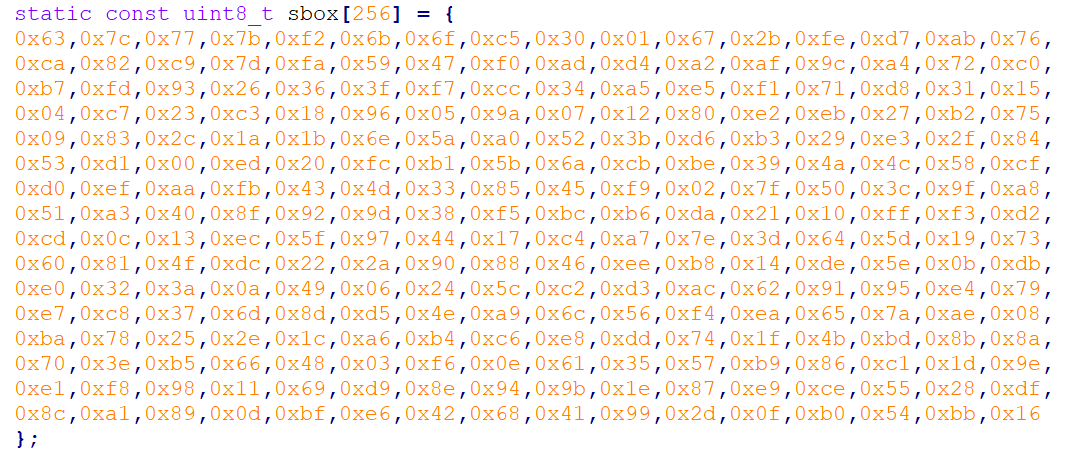
**Table 13: Mul13 Table**



**Table 14: Mul14 Table**



## **1.2 Code Snippets**



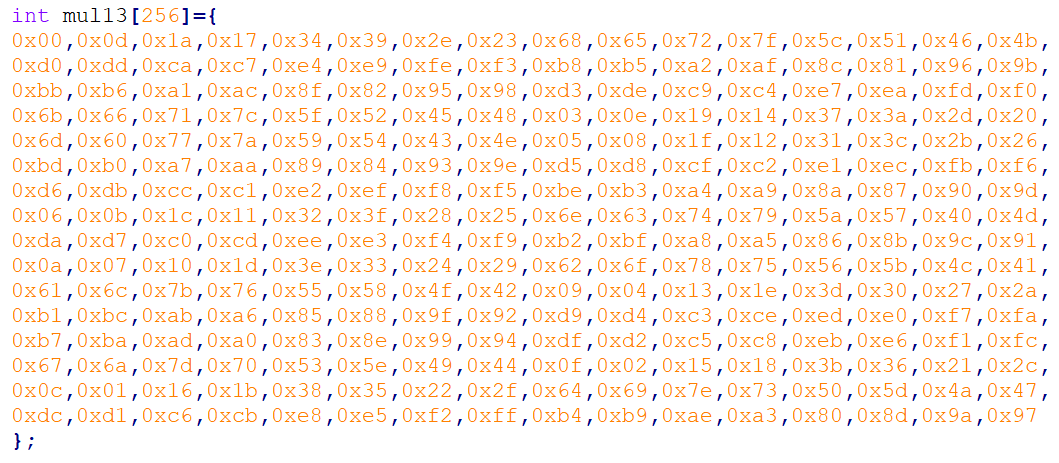
**Figure 46: Implementation of S-Box in C**



**Figure 47: Implementation of Mul9 Lookup Table in C**



**Figure 48: Implementation of Mul11 Lookup Table in C**



**Figure 49: Implementation of Mul13 Lookup Table in C**



**Figure 50: Implementation of Mul14 Lookup Table in C**

# **References**

|  |  |
| --- | --- |
| [1] | M. Souppaya and K. Scarfone, *Guidelines for Securing Wireless Local Area Networks (WLANs),* NIST Special Publication 800-153, 2012. |
| [2] | S. Banerji and R. S. Chowdhury, *On IEEE 802.11: Wireless LAN Technology,* International Journal of Mobile Network Communications & Telematics (IJMNCT) , 2013. |
| [3] | J. Edney and W. A. Arbaugh, Real 802.11 Security: Wi-Fi Protected Access and 802.11i, Addison-Wesley, 2004. |
| [4] | Wi-Fi Alliance, *Deploying Wi-Fi Protected Access,* 2005. |
| [5] | Juniper Networks, "Understanding PSK Authentication," Juniper Networks, 16 October 2018. [Online]. Available: https://www.juniper.net/documentation/en\_US/junos-space-apps/network-director3.0/topics/concept/wireless-wpa-psk-authentication.html. [Accessed 24 October 2018]. |
| [6] | V. Fadyushin, "Common WLAN Protection Mechanisms and their Flaws," 16 March 2016. [Online]. Available: https://hub.packtpub.com/common-wlan-protection-mechanisms-and-their-flaws/. [Accessed 24 October 2018]. |
| [7] | T. B. Johnson, "An FPGA architecture for the recovery of WPA/WPA2 keys," Graduate Theses and Dissertations, 2014. |
| [8] | "Understanding WPA/WPA2 Pre-Shared-Key Cracking," 27 December 2015. [Online]. Available: https://www.ins1gn1a.com/understanding-wpa-psk-cracking/. [Accessed 26 October 2018]. |
| [9] | O. Nakhila, A. Attiah, Y. Jin and C. Zou, "Parallel Active Dictionary Attack on WPA2-PSK Wi-Fi Networks". |
| [10] | M. Kammerstetter, M. Muellner, D. Burian, C. Kudera and W. Kastner, "Efficient High-Speed WPA2 Brute Force Attacks using Scalable Low-Cost FPGA Clustering," 2016. |
| [11] | A. Visconti, S. Bossi, H. Ragab and A. Calò, *On the weaknesses of PBKDF2,* 2015. |
| [12] | H. Krawczyk, B. M. and R. Canetti, *RFC2104: HMAC: Keyed-Hashing for Message Authentication,* 1997. |
| [13] | H. Yang, "SHA1 Message Digest Algorithm Overview," 2017. [Online]. Available: http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-Overview.html. [Accessed 27 October 2018]. |
| [14] | National Institute of Standards and Technology(NIST), *Announcing the ADVANCED ENCRYPTION STANDARD (AES),* Federal Information Processing Standards Publication 197, 2001. |
| [15] | A. Berent, *Advanced Encryption Standard by Example.* |
| [16] | J. Monttinen, *The Security of Advanced Encryption Standard,* 2015. |
| [17] | Xilinx, *MicroBlaze Processor Reference Guide,* 2016. |
| [18] | R. Jesman, F. M. Vallina and J. Saniie, *MicroBlaze Tutorial Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools.* |
| [19] | Digilent, *Nexys 4™ FPGA Board Reference Manual,* 2016. |
| [20] | Xilinx, *7 Series FPGAs Data Sheet: Overview,* 2018. |
| [21] | C. Toole, "Introduction to AXI Protocol: Understanding the AXI interface," 2016. [Online]. Available: https://community.arm.com/soc/b/blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface. [Accessed 4 November 2018]. |
| [22] | Xilinx, *AXI Reference,* 2011. |
| [23] | Xilimx, *AXI Interconnect v2.1 LogiCORE IP Product Guide,* 2017. |
| [24] | P. V. Kinge, S. Honale and C. Bobade, "Design of AES Algorithm for 128/192/256 Key Length in FPGA," *International Journal of Reconfigurable and Embedded Systems (IJRES),* pp. 49-53, 2014. |
| [25] | V. D. Orlic, M. Peric, Z. Banjac and S. Milicevic, *Some aspects of practical implementation of AES 256 crypto algorithm,* Belgrade: 20th Telecommunications forum TELFOR, 2012. |
| [26] | N. Sai Srinivas and M. Akramuddin, *FPGA Based Hardware Implementation of AES Rijndael Algorithm for Encryption and Decryption,* International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016. |
| [27] | M. Gupta and S. P. Mahto, *Implementation of 128, 192 & 256 bits Advanced Encryption Standard on Reconfigurable Logic,* International Journal of Engineering Trends and Technology (IJETT), 2017. |