Advanced Encryption Standard

Xinlin Chen, Nnenna Nwagbo, Zhonghan Li

8 June 2018

EECS 392

Prof. David Zaretsky

Table of Contents

List of Figures

Executive Summary

The design problem we targeted was secure message encryption. Our approach involved programming a Field Programmable Gate Array to encrypt a given plaintext message using the AES-128 algorithm, which is one of the three cipher blocks that makes up the Advanced Encryption Standard. The Advanced Encryption Standard (AES) was adopted by the National Institutes of Standards and Technology in 2001 and protects information up to the SECRET level. A subset of the Rijndael ciphers developed by Vincent Rijmen and Joan Daemen, these algorithms, robust against brute-force attacks, outcompeted 15 designs to become the AES.

Introduction

The goal of our project is to design a system that is capable of encrypting a plaintext message of arbitrary length using the Advanced Encryption Standard (hereafter AES) and displaying the resulting ciphertext. Advanced Encryption Standard, or AES, is the encryption algorithm used by the U.S. government to protect classified information up to the SECRET level. In 2001, three types of Rijndael ciphers (the 128-, 192-, and 256-bit key ciphers with a block size of 128 bits) were selected by the National Institutes of Standards and Technology, out of 15 competing designs, to become the AES.[1] Rijndael ciphers are algorithms developed by Vincent Rijmen and Joan Daemen,[2] and are robust against brute-force attacks. We focus on the variant that uses 128-bit keys, AES-128. The key length of 128 bits means that the key-space of the algorithm is 3.402×10^{38} keys, a space which would require about 1019 years to cover with a computer capable of testing 1 tera-keys/second.[3] AES-128 requires 10 cycles of repetitions of transformation rounds to obtain a final output, or ciphertext, from a given input, or plaintext message.[4] A biclique cryptanalysis on the full AES-128 was proposed in 2011, with

[1] FIPS, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," NIST, Gaithersburg, MD, USA. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

[2] J. Daemen and V. Rijmen, "The Rijndael Block Cipher," NIST, Gaithersburg, MS, USA. [Online]. Available: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf

[3] A.H. Khan, M.A. Al-Mouhammed, A. Almousa, A. Fatayar, A.R. Ibrahim and A.J. Siddiqui, "AES-128 ECB Encryption on GPUs and Effects of Input Plaintext Patterns on Performance," in *15th IEEE/ACIS SNPD*, Las Vegas, NV, 2014, pp. 207-212.

[4] N. Mathur and R. Bansode, "AES Based Text Encryption Using 12 Rounds with Dynamic Key Selection," *Procedia Comput. Sci.* 2016, pp. 1036-1043.

computational complexity $2^{126.1}$.[5] Message encryption is desirable to protect individuals' privacy, both in the context of informal chat messaging services for personal use and confidential emails for business use. This is especially true if there is a third-party interested in eavesdropping on these communications. It is undesirable to leave your communications vulnerable to stalkers, corporate spies, or future employers, to name a few such parties.

We sought to encrypt using the AES-128 variant, which uses 128-bit cipher keys. AES-128 requires a 16-byte cipherkey and 16-byte plaintext message blocks. The cipherkey is used to generate 10 roundkeys which are used to encrypt the plaintext message.

[5] A. Bogdanov, D. Khovratovich, C. Rechberger, "Biclique Cryptanalysis of the Full AES," *ASIACRYPT* 2011, pp. 344-371.

Broader considerations

Message encryption is useful both for personal use and, in any industry that deals with proprietary information, for business use. It has applications for many communications methods, including instant messaging, emails, and file transfers. Encryption by AES-128, in particular, is powerful against brute force attacks, and has been deemed a secure method to protect classified information.

Email encryption is not necessarily the default. Microsoft, with its Outlook programs for emails, offers it as an option for outgoing messages, while requiring you to share your digital ID, or public key certificate. Google Mail offers S/MIME (enhanced encryption) which only encrypts messages if you have it enabled on your account, and TLS (Transport Layer Security, or standard encryption). While for personal use, Facebook Messenger does not default encrypt you messages, with the option to have “secret” self-destruct messages in a new opt-in program. Furthermore, most cell phones transmitted communication is not encrypted, making there a need for personal encryption of messages are needed.

Design constraints and requirements

The main constraint of our project was adapting an AES-128 encryption method for an FPGA board. The key requirement was for the design be able to input the desired key and message needed to be encrypted, and for the encrypted message be displayed in a way for the user to communicate the message to their desired recipient. This was done by the input through a ps2 keyboard and the outputs displayed on the LED and LCD screens on the DE2-112. Due to the contained nature of the encryption algorithm, the design was completely contained within the DE2-115 board. Therefore, the design was not constrained by design challenges such as power, availability, or cost.

Design description

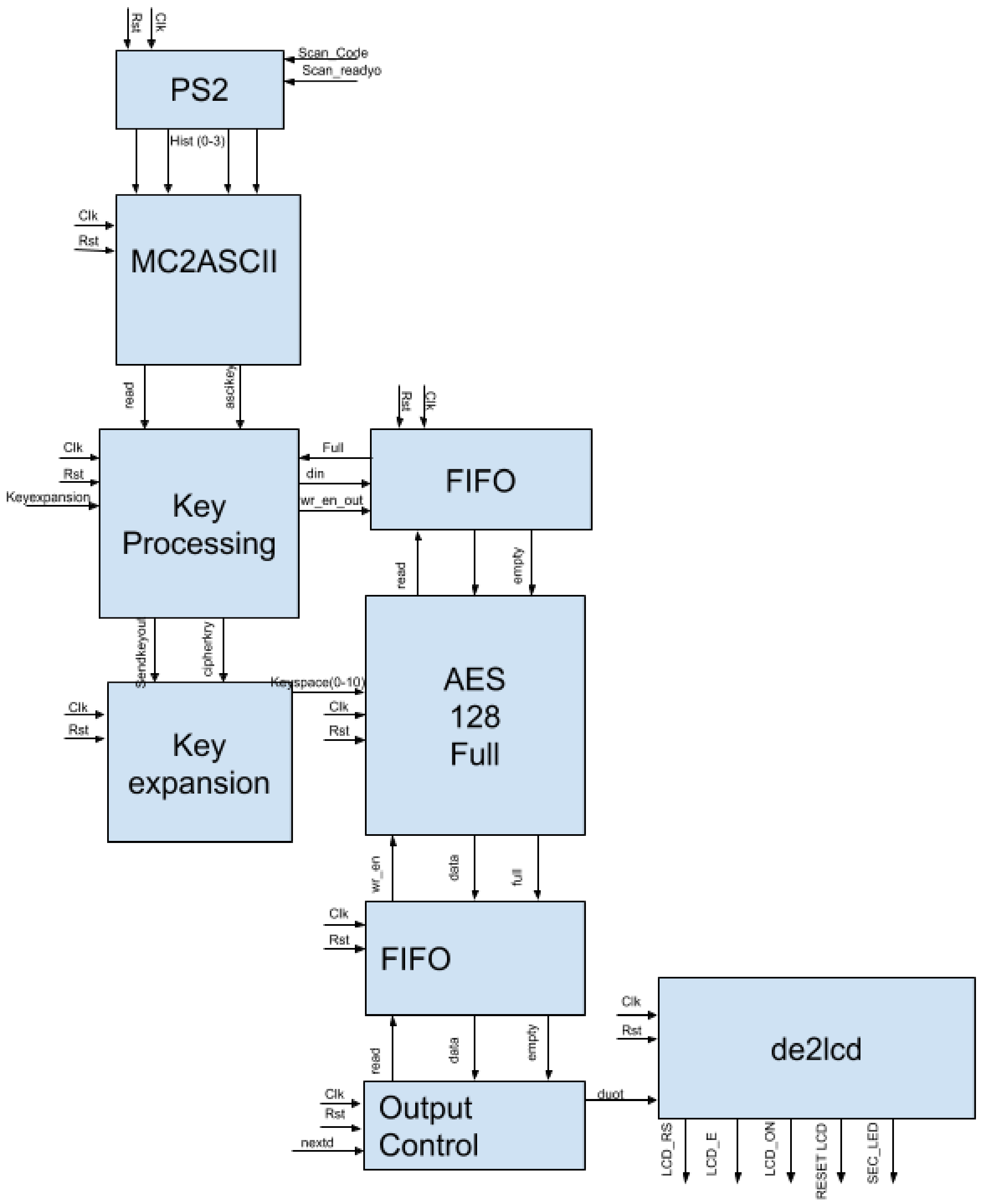


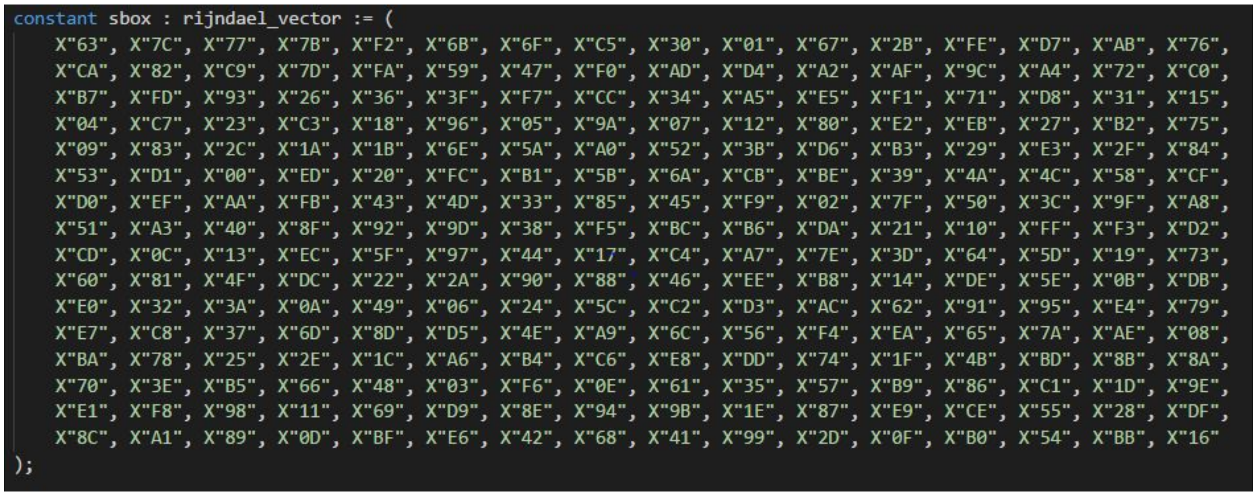
Figure 1: Design Block diagram

Our system consists of an Altera DE2-115 board and keyboard. When switch 0 on the board is high, the system treats the input from the keyboard as the cipherkey up until the user has pressed more than 16 valid keys - keys that have an ASCII character equivalent - or until the user presses ‘Enter,’ whichever occurs first. When switch 0 is low, the system treats the input from the keyboard as the plaintext message until the user presses ‘Enter’.

The output of the keyboard is a series of ‘make’ and ‘break’ codes. Make codes are generated when a key is pressed - e.g. if the space bar is held down for a number of seconds, the make code, ‘29’ in hex, is sent continuously during this time. Break codes are generated when a key is released - continuing the previous example, once the space bar is released, the code sequence ‘F0’ ‘29’, in hex, is sent once. Break codes consist of ‘F0’ followed by the make code of the relevant key.

Our system processes these inputted make and break codes, converting appropriate inputs into ASCII characters with the ‘make code to ascii’ module. The resulting ASCII characters are then processed by the KeyProcessing module, which fills up either the cipherkey or message character by character, depending on the state of the switch. If a cipherkey has been generated, the 16-byte vector is sent to the KeyExpansion module, where it is used to generate the 10 roundkeys used in the transformation rounds of the AES. If a message has been generated, it is sent to the message FIFO, which stores the information until the encryption module, AES128_full, is ready for the message.

The KeyExpansion module generates 10 16-byte roundkeys from the 16-byte cipherkey, and outputs all 11 16-byte vectors. The roundkeys are generated using circular left shift, byte-wise substitution using Rijndael’s S-box(shown in figure 2)and adding a round constant based on the step number. KeyExpansion handles the upper level which is simply an array of 10 roundkey generations steps. The AES128 algorithm itself is implemented in a similar manner to KeyExpansion. A message to be encrypted goes through ten rounds of SubBytes(again using Rijndael’s S-box), ShiftRows(circular), MixColumns(finite field multiplication) and AddRoundKeys with the corresponding roundkey for that step being provided from the KeyExpansion module.



```
constant sbox : rijndael_vector := (
    X"63", X"7C", X"77", X"7B", X"F2", X"6B", X"6F", X"C5", X"30", X"01", X"67", X"2B", X"FE", X"D7", X"AB", X"76",
    X"CA", X"82", X"C9", X"7D", X"FA", X"59", X"47", X"F0", X"AD", X"D4", X"A2", X"AF", X"9C", X"A4", X"72", X"C0",
    X"B7", X"FD", X"93", X"26", X"36", X"3F", X"F7", X"CC", X"34", X"A5", X"E5", X"F1", X"71", X"D8", X"31", X"15",
    X"04", X"C7", X"23", X"C3", X"18", X"96", X"05", X"9A", X"07", X"12", X"80", X"E2", X"EB", X"27", X"B2", X"75",
    X"09", X"83", X"2C", X"1A", X"1B", X"6E", X"5A", X"A0", X"52", X"3B", X"D6", X"B3", X"29", X"E3", X"2F", X"84",
    X"53", X"D1", X"00", X"ED", X"20", X"FC", X"B1", X"5B", X"6A", X"CB", X"BE", X"39", X"4A", X"4C", X"58", X"CF",
    X"D0", X"EF", X"AA", X"FB", X"43", X"4D", X"33", X"85", X"45", X"F9", X"02", X"7F", X"50", X"3C", X"9F", X"A8",
    X"51", X"A3", X"40", X"8F", X"92", X"9D", X"38", X"F5", X"BC", X"B6", X"DA", X"21", X"10", X"FF", X"F3", X"D2",
    X"CD", X"0C", X"13", X"EC", X"5F", X"97", X"44", X"17", X"C4", X"A7", X"7E", X"3D", X"64", X"5D", X"19", X"73",
    X"60", X"81", X"4F", X"DC", X"22", X"2A", X"90", X"88", X"46", X"EE", X"B8", X"14", X"DE", X"5E", X"0B", X"DB",
    X"E0", X"32", X"3A", X"0A", X"49", X"06", X"24", X"5C", X"C2", X"D3", X"AC", X"62", X"91", X"95", X"E4", X"79",
    X"E7", X"C8", X"37", X"6D", X"8D", X"D5", X"4E", X"A9", X"6C", X"56", X"F4", X"EA", X"65", X"7A", X"AE", X"08",
    X"BA", X"78", X"25", X"2E", X"1C", X"A6", X"B4", X"C6", X"E8", X"DD", X"74", X"1F", X"4B", X"BD", X"8B", X"8A",
    X"70", X"3E", X"B5", X"66", X"48", X"03", X"F6", X"0E", X"61", X"35", X"57", X"B9", X"86", X"C1", X"1D", X"9E",
    X"E1", X"F8", X"98", X"11", X"69", X"D9", X"8E", X"94", X"9B", X"1E", X"87", X"E9", X"CE", X"55", X"28", X"DF",
    X"8C", X"A1", X"89", X"0D", X"BF", X"E6", X"42", X"68", X"41", X"99", X"2D", X"0F", X"B0", X"54", X"BB", X"16"
);
```

Figure 2: Rijndael’s S-box

The final ciphertext message is presented on the LCD display on the DE2-115 board.

Design optimizations:

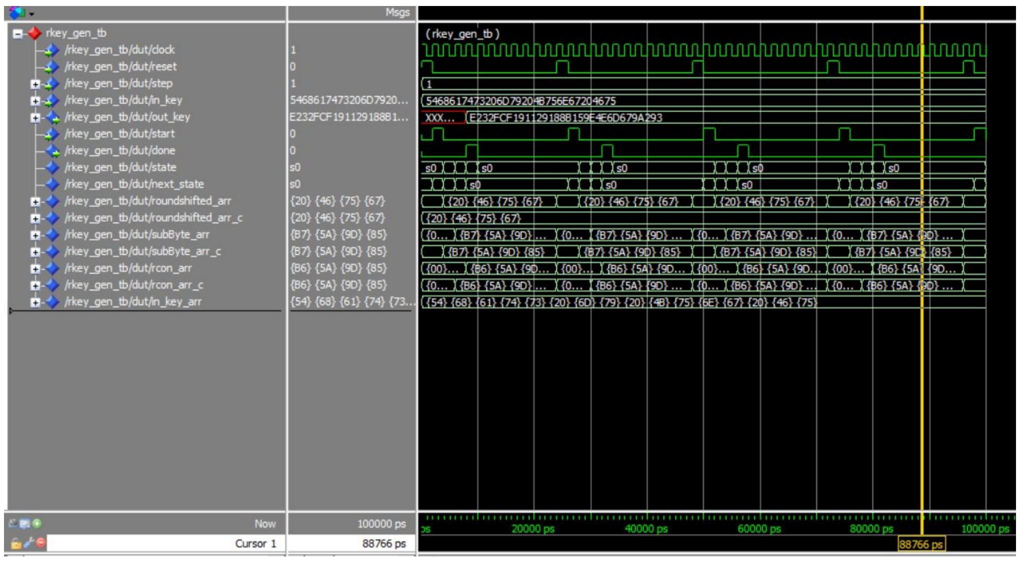
Our design was built with potential scaling and optimizations in mind. First, it makes use of FIFOs along the AES128 path. This allows messages to be typed in a continuous stream as messages will be buffered and encrypted as soon as possible. This also allows for the AES128 encryption component to potentially be repurposed to encrypt large blocks of text in a continuous stream. By specifically laying out all the steps to be synthesized as separate blocks, we could implement pipelining to further speed up the process when dealing with large chunks of data. This isn't immediately necessary within the scope of our objective: to create a real-time AES128 encryption mechanism that encrypts messages as one types on the keyboard. Because humans can't realistically type messages faster than the rate that this design can encrypt them, it wasn't a critical feature to have, but our design is written to allow for this optimization.

Testing/Simulation

Testing was done by first testing and simulating the individual modules of the AES-128 algotherm before compiling components into a larger mid-level file to test the interaction of the coaplents to each other. Finally, the whole algorithm was assembled and simulated in the top level file to test the whole system.

Single Round Key Generation Step

This step follows the steps in generating a single roundkey from the cipher key. The cipher key must be 16 bytes long, or 128 bits. It may be the result of a hash, but for our purposes, we are simply using a typed passphrase as the key. A hashing function may be implemented in the future. The first step is to convert the 128-bit cipher key into 16 easy to manage bytes. Our primary concern is the last word of the key. It undergoes a circular byte left shift first, then a byte-substitution step using Rijndael's S-box, then finally a round constant is added based on the step number in the sequence. These steps are implemented using a finite state machine. A simulation of this step is shown below:

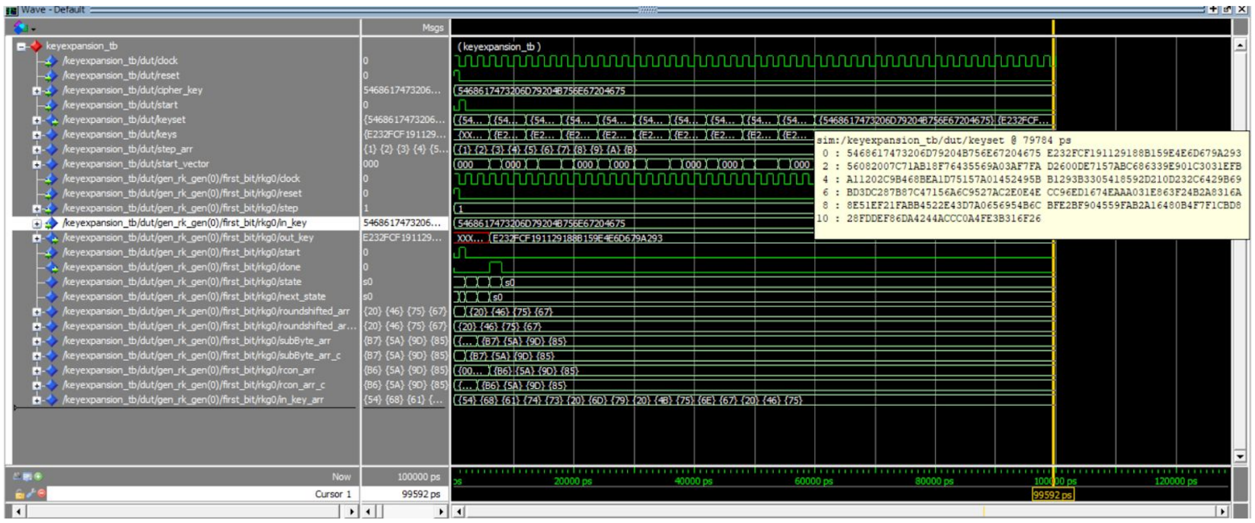


```
                                              Msgs
rkey_gen_tb
  /rkey_gen_tb/dut/clock                      1
  /rkey_gen_tb/dut/reset                      0
  /rkey_gen_tb/dut/step                       1
  /rkey_gen_tb/dut/in_key                     5468617473206D7920...
  /rkey_gen_tb/dut/out_key                    E232FCF191129188B1...
  /rkey_gen_tb/dut/start                      0
  /rkey_gen_tb/dut/done                       0
  /rkey_gen_tb/dut/state                      s0
  /rkey_gen_tb/dut/next_state                 s0
  /rkey_gen_tb/dut/roundshifted_arr           {20} {46} {75} {67}
  /rkey_gen_tb/dut/roundshifted_arr_c         {20} {46} {75} {67}
  /rkey_gen_tb/dut/subByte_arr                {B7} {5A} {9D} {85}
  /rkey_gen_tb/dut/subByte_arr_c              {B7} {5A} {9D} {85}
  /rkey_gen_tb/dut/rcon_arr                   {B6} {5A} {9D} {85}
  /rkey_gen_tb/dut/rcon_arr_c                 {B6} {5A} {9D} {85}
  /rkey_gen_tb/dut/in_key_arr                 {54} {68} {61} {74} {73...

in_key     5468617473206D79204B756E67204675
out_key    XXX...  E232FCF191129188B159E4E6D679A293
in_key_arr {54} {68} {61} {74} {73} {20} {6D} {79} {20} {4B} {75} {6E} {67} {20} {46} {75}

Now       100000 ps
Cursor 1  88766 ps
```

Figure 3: Simulation of Round Key Step

KeyExpansion

The KeyExpansion is an array of the roundkey generation steps. It serves to interconnect the output roundkey of one step to the input roundkey of the next. It also interfaces with the key processing module to know when a cipher key is typed as an input. The outputs are 11 keys total: the cipher key and the 10 other roundkeys generated through the sequence. In the simulation below, the output shows the correct array based on an example case we followed:

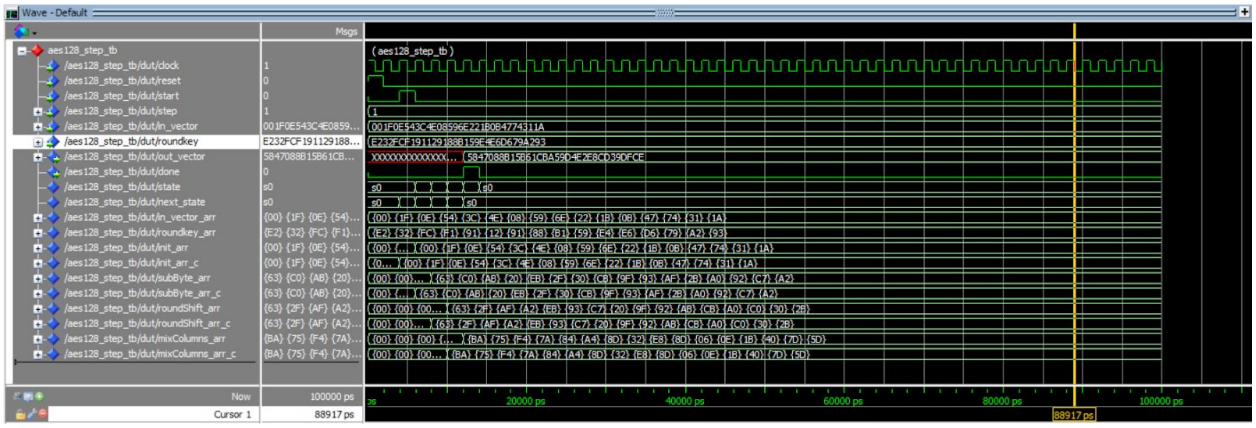


```
keyexpansion_tb
/keyexpansion_tb/dut/clock                                         0
/keyexpansion_tb/dut/reset                                         0
/keyexpansion_tb/dut/cipher_key                                    5468617473206...
/keyexpansion_tb/dut/start                                         0
/keyexpansion_tb/dut/keyset                                        {5468617473206...
/keyexpansion_tb/dut/keys                                          {E232FCF191129...
/keyexpansion_tb/dut/step_arr                                      {1} {2} {3} {4} {5...
/keyexpansion_tb/dut/start_vector                                  000
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/clock            0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/reset            0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/step             1
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/in_key           5468617473206...
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/out_key          E232FCF191129...
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/start            0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/done             0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/state            s0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/next_state       s0
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/roundshifted_arr {20} {46} {75} {67}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/roundshifted_ar... {20} {46} {75} {67}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/subByte_arr      {B7} {5A} {9D} {85}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/subByte_arr_c    {B7} {5A} {9D} {85}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/rcon_arr         {B6} {5A} {9D} {85}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/rcon_arr_c       {B6} {5A} {9D} {85}
/keyexpansion_tb/dut/gen_rk_gen(0)/first_bit/rkg0/in_key_arr       {54} {68} {61} {...

Now      100000 ps
Cursor 1  99592 ps

sim:/keyexpansion_tb/dut/keyset @ 79784 ps
 0 : 5468617473206D79204B756E67204675 E232FCF191129188B159E4E6D679A293
 2 : 56082007C71AB18F76435569A03AF7FA D2600DE7157ABC686339E901C3031EFB
 4 : A11202C9B468BEA1D75157A01452495B B1293B3305418592D210D232C6429B69
 6 : BD3DC287B87C47156A6C9527AC2E0E4E CC96ED1674EAAA031E863F24B2A8316A
 8 : 8E51EF21FABB4522E43D7A0656954B6C BFE2BF904559FAB2A16480B4F7F1CBD8
10 : 28FDDEF86DA4244ACCC0A4FE3B316F26
```

Figure 4: Simulation of KeyExpansion Step

AES 128 Single Step

AES128 operates on a 10-step process to create an encrypted message from a cipher and the original message. This entity is a single step in the process implemented with a finite state machine. The four rounds include SubBytes, ShiftRows, MixColumns and AddRoundkey. In SubBytes, every byte of the message is substituted using Rijndael's S-box. ShiftRows performs a circular left shift of 0,1,2, and 3 bytes depending on the row of the matrix. MixColumns performs a linear transformation on each of the four columns of the matrix. Each bit is processed using multiplication in GF(2^8). If the most significant bit 1 is, after performing the multiplication the product must be XORd by x"1B". AddRoundkey simply XORs the bits of the current matrix with the bits of the current roundkey. There is also a preprocessing step functionality in which the original message is XORd with the cipher key. The final step skips the MixColumns procedure, which this entity also accounts for.



```
Wave - Default
                                        Msgs
aes128_step_tb                                                  ( aes128_step_tb )
  /aes128_step_tb/dut/clock             1
  /aes128_step_tb/dut/reset             0
  /aes128_step_tb/dut/start             0
  /aes128_step_tb/dut/step              1                       1
  /aes128_step_tb/dut/in_vector         001F0E543C4E0859...     001F0E543C4E08596E221B0B4774311A
  /aes128_step_tb/dut/roundkey          E232FCF191129188...     E232FCF191129188B159E4E6D679A293
  /aes128_step_tb/dut/out_vector        5847088B15B61CB...      XXXXXXXXXXXXXX... 5847088B15B61CBA59D4E2E8CD39DFCE
  /aes128_step_tb/dut/done              0
  /aes128_step_tb/dut/state             s0                      s0 s0
  /aes128_step_tb/dut/next_state        s0                      s0 s0
  /aes128_step_tb/dut/in_vector_arr     {00} {1F} {0E} {54}...  {00} {1F} {0E} {54} {3C} {4E} {08} {59} {6E} {22} {1B} {0B} {47} {74} {31} {1A}
  /aes128_step_tb/dut/roundkey_arr      {E2} {32} {FC} {F1}...  {E2} {32} {FC} {F1} {91} {12} {91} {88} {B1} {59} {E4} {E6} {D6} {79} {A2} {93}
  /aes128_step_tb/dut/init_arr          {00} {1F} {0E} {54}...  {00} {... {00} {1F} {0E} {54} {3C} {4E} {08} {59} {6E} {22} {1B} {0B} {47} {74} {31} {1A}
  /aes128_step_tb/dut/init_arr_c        {00} {1F} {0E} {54}...  {0... {00} {1F} {0E} {54} {3C} {4E} {08} {59} {6E} {22} {1B} {0B} {47} {74} {31} {1A}
  /aes128_step_tb/dut/subByte_arr       {63} {C0} {AB} {20}...  {00} {00}... {63} {C0} {AB} {20} {EB} {2F} {30} {CB} {9F} {93} {AF} {2B} {A0} {92} {C7} {A2}
  /aes128_step_tb/dut/subByte_arr_c     {63} {C0} {AB} {20}...  {00} {... {63} {C0} {AB} {20} {EB} {2F} {30} {CB} {9F} {93} {AF} {2B} {A0} {92} {C7} {A2}
  /aes128_step_tb/dut/roundShift_arr    {63} {2F} {AF} {A2}...  {00} {00} {00... {63} {2F} {AF} {A2} {EB} {93} {C7} {20} {9F} {92} {AB} {CB} {A0} {C0} {30} {2B}
  /aes128_step_tb/dut/roundShift_arr_c  {63} {2F} {AF} {A2}...  {00} {00}... {63} {2F} {AF} {A2} {EB} {93} {C7} {20} {9F} {92} {AB} {CB} {A0} {C0} {30} {2B}
  /aes128_step_tb/dut/mixColumns_arr    {BA} {75} {F4} {7A}...  {00} {00} {00} {... {BA} {75} {F4} {7A} {84} {A4} {8D} {32} {E8} {8D} {06} {0E} {1B} {40} {7D} {5D}
  /aes128_step_tb/dut/mixColumns_arr_c  {BA} {75} {F4} {7A}...  {00} {00} {00... {BA} {75} {F4} {7A} {84} {A4} {8D} {32} {E8} {8D} {06} {0E} {1B} {40} {7D} {5D}

                              Now       100000 ps    ps   20000 ps   40000 ps   60000 ps   80000 ps   100000 ps
                         Cursor 1        88917 ps                                                   88917 ps
```

Figure 5: Simulation of a single step of the AES128 algorithm

Key processing

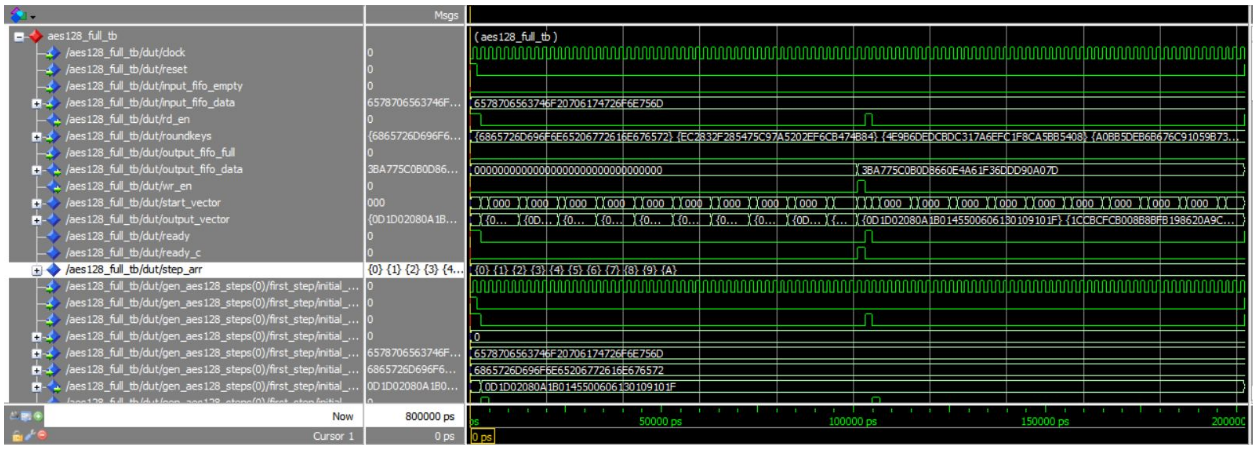
When keystrokes are inputted, they need to be converted to keystrokes and subsequently to meaningful ‘strings’ (hex values represented as ASCII characters) for a cipherkey or a message. The keyprocessing module passes the ASCII value in tempvector_c to either cipherkey (sent to the keyexpansion module, where roundkeys are generated) or din (sent to message_fifo and, eventually, aes128_full for encryption), based on the value of keyormsg (0-> cipherkey, 1-> din). The steps are walked through in the simulation screenshots below.

Mid-Level testing

After the individual simulations of the step in the AES-128 algorithm, which group and tested the flow between key expansidio and key processing before entering the AES128 algorithm module.

AES128 Full

This is an array of AES128 Steps corresponding to each of the 10 steps in the encryption process. The output of each step goes into the input of the next, along with the corresponding generated roundkey from Keyexpansion. The final output at the last step is the encrypted message:



```
                                                                    Msgs
aes128_full_tb
  /aes128_full_tb/dut/clock                                         0
  /aes128_full_tb/dut/reset                                         0
  /aes128_full_tb/dut/input_fifo_empty                              0
  /aes128_full_tb/dut/input_fifo_data                               6578706563746F...
  /aes128_full_tb/dut/rd_en                                         0
  /aes128_full_tb/dut/roundkeys                                     {6865726D696F6...
  /aes128_full_tb/dut/output_fifo_full                              0
  /aes128_full_tb/dut/output_fifo_data                              3BA775C0B0D86...
  /aes128_full_tb/dut/wr_en                                         0
  /aes128_full_tb/dut/start_vector                                  000
  /aes128_full_tb/dut/output_vector                                 {0D1D02080A1B...
  /aes128_full_tb/dut/ready                                         0
  /aes128_full_tb/dut/ready_c                                       0
  /aes128_full_tb/dut/step_arr                                      {0} {1} {2} {3} {4...
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    0
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    0
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    0
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    0
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    6578706563746F...
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    6865726D696F6...
  /aes128_full_tb/dut/gen_aes128_steps(0)/first_step/initial_...    0D1D02080A1B0...
                                                      Now           800000 ps
                                                      Cursor 1      0 ps
```

(aes128_full_tb)
6578706563746F20706174726F6E756D
{6865726D696F6E652067726162E676572} {EC2832F285475C97A5202EF6CB474B84} {4E9B6DEDCBDC317A6EFC1F8CA5BB5408} {A0BB5DEB6B676C91059B73...
00000000000000000000000000000000
3BA775C0B0D8660E4A61F36DDD90A07D
000
{0D1D02080A1B0145500606130109101F} {1CCBCFCB008B8BFB198620A9C...
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {A}
0
6578706563746F20706174726F6E756D
6865726D696F6E652067726162E676572
0D1D02080A1B0145500606130109101F
50000 ps
100000 ps
150000 ps
200000
0 ps

Figure 6: Simulation of the full AES128 algorithm

Output Control

This entity controls the flow of encrypted messages to be more easily read on the LCD screen. Along with holding the value of whatever encrypted message is being read, it also allows scrolling down through encrypted messages if multiple 16-character encrypted messages were buffered in the preceding fifo using a single bit input “nextd”. As shown in the below figure, the next output is only shown when nextd goes up then down which can be done with a slide switch or push button.

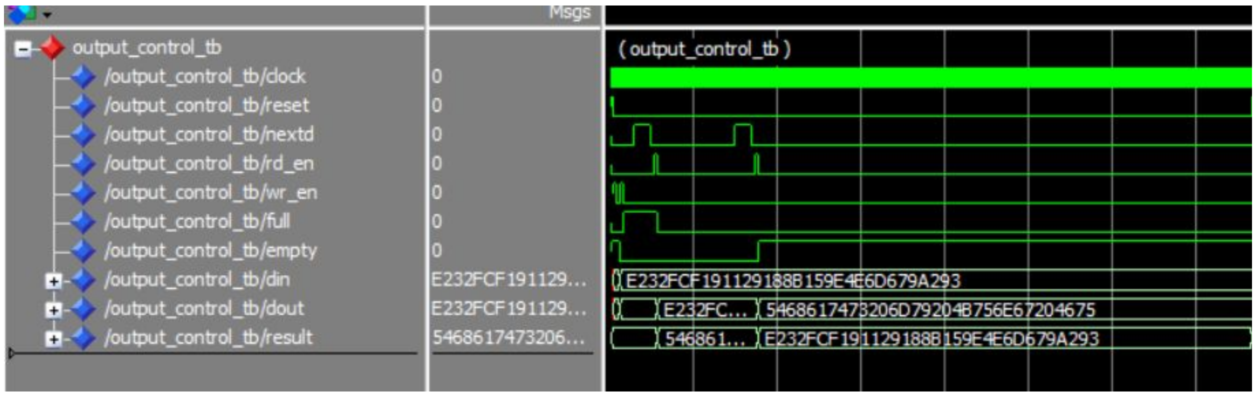


Figure 7: Output Control simulation

Implementation/Synthesis

The full design incorporates use of the LCD screen for showing the encrypted messages, 7-segment LED display to provide some keyboard touch feedback, and three switches corresponding to keyormsg, reset and nextd. The main input would be the keyboard presses along with the switches, while the main output is the encrypted key on the LCD screen as shown in the picture below.

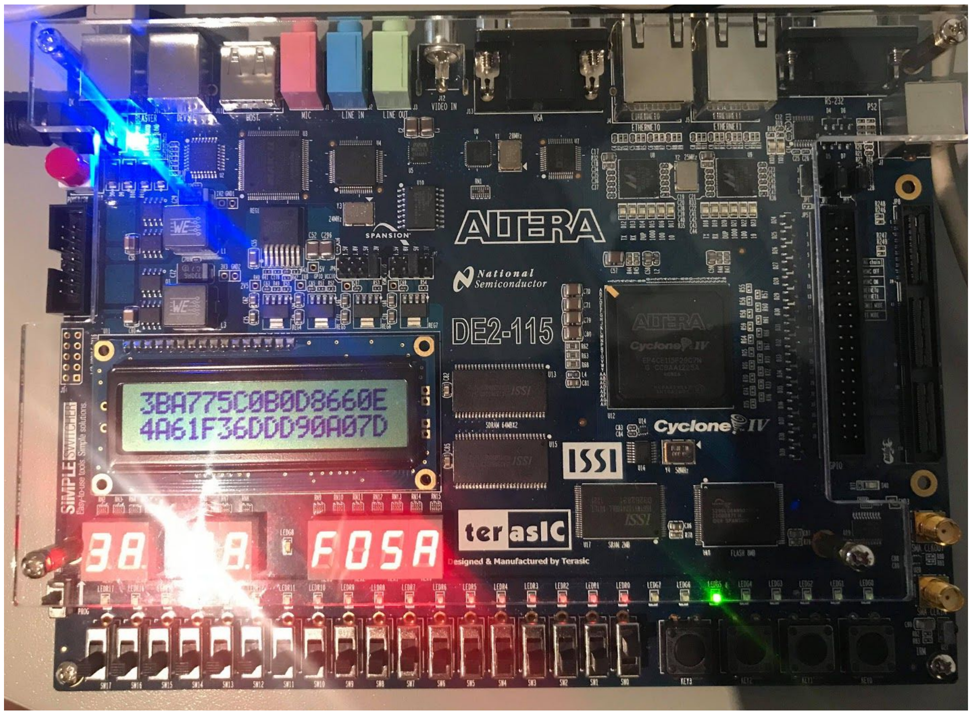
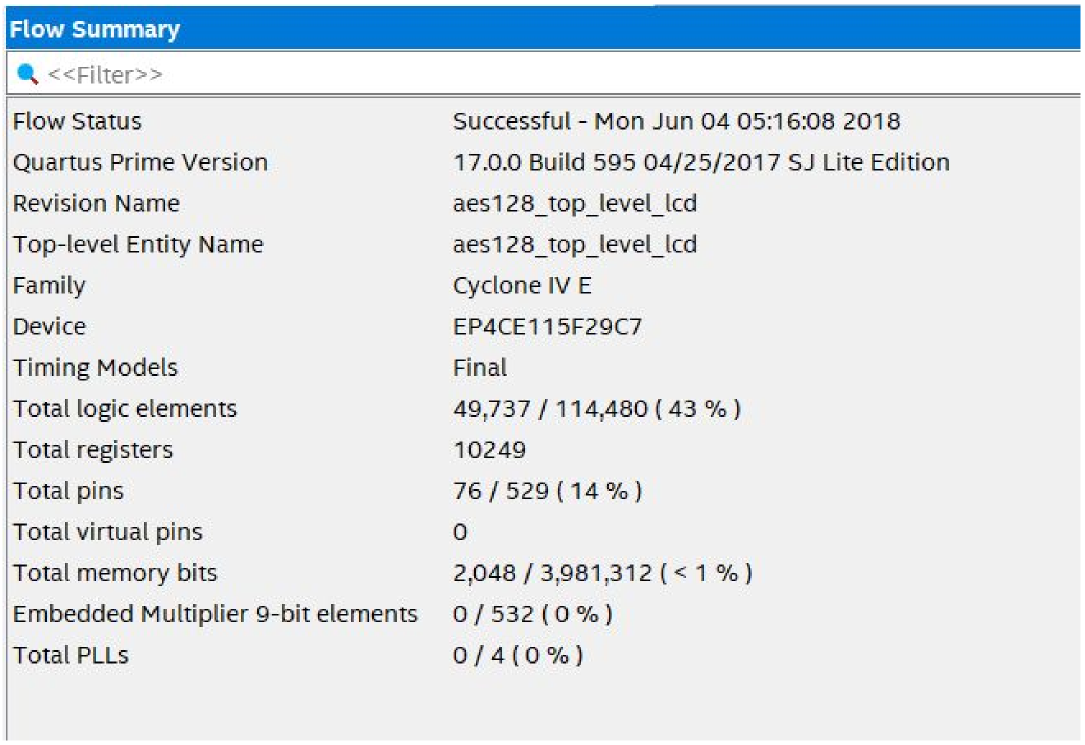


Figure 8: Encrypted message displayed on DE2-115 board

After synthesis, the resource usage is outlined below:



Flow Summary

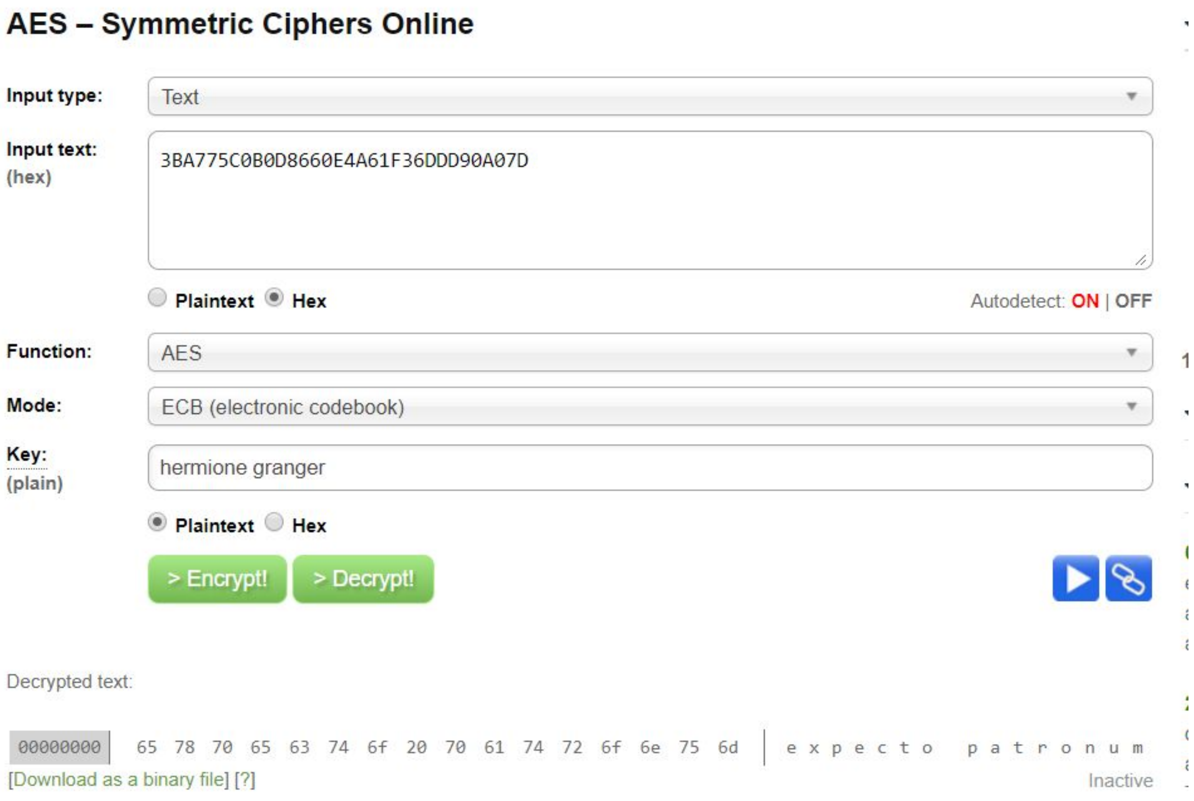
<<Filter>>

Flow Status	Successful - Mon Jun 04 05:16:08 2018
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	aes128_top_level_lcd
Top-level Entity Name	aes128_top_level_lcd
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	49,737 / 114,480 (43 %)
Total registers	10249
Total pins	76 / 529 (14 %)
Total virtual pins	0
Total memory bits	2,048 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 9: Flow summary report

Our design requires a lot of registers as we are working with fairly large 128 bit arrays throughout the entire process, scaled up by the numerous steps needed to generate the roundkeys and complete the encryption steps. Even so, there are still many more resources on the board to spare.

Because our board only dealt with encryption, we had to use an online AES128 decryptor to verify that our encrypted message was correct. A sample screenshot of the verification is depicted below using aes.online-domain-tools.com. The input text is the hex representation of our encrypted message and the key is the same key we used for encryption. Note the final output of the decryption, which was our input for that trial of encryption.



AES – Symmetric Ciphers Online

Input type: Text

Input text: (hex)

```
3BA775C0B0D8660E4A61F36DDD90A07D
```

Plaintext [✓ Hex] Autodetect: ON | OFF

Function: AES

Mode: ECB (electronic codebook)

Key: (plain) hermione granger

[✓ Plaintext] Hex

Decrypted text:

```
00000000  65 78 70 65 63 74 6f 20 70 61 74 72 6f 6e 75 6d | e x p e c t o   p a t r o n u m
```

Figure 10: Message decryoted using aes.online-domain-tools.com

Conclusions

In conclusion, over the course of our project, we designed, programmed and implemented AES-128 encryption on an Altera DE2-115 board. Our design used cipherkeys up to 128 bits and was capable of accepting messages longer or shorter than 128 bits. It encrypted and displayed these messages, in the chronological order of input, on the Altera board's LCD screen. We tested many combinations of cipherkeys and messages, and verified that the encryption worked as designed using an online AES-128 decryption tool.

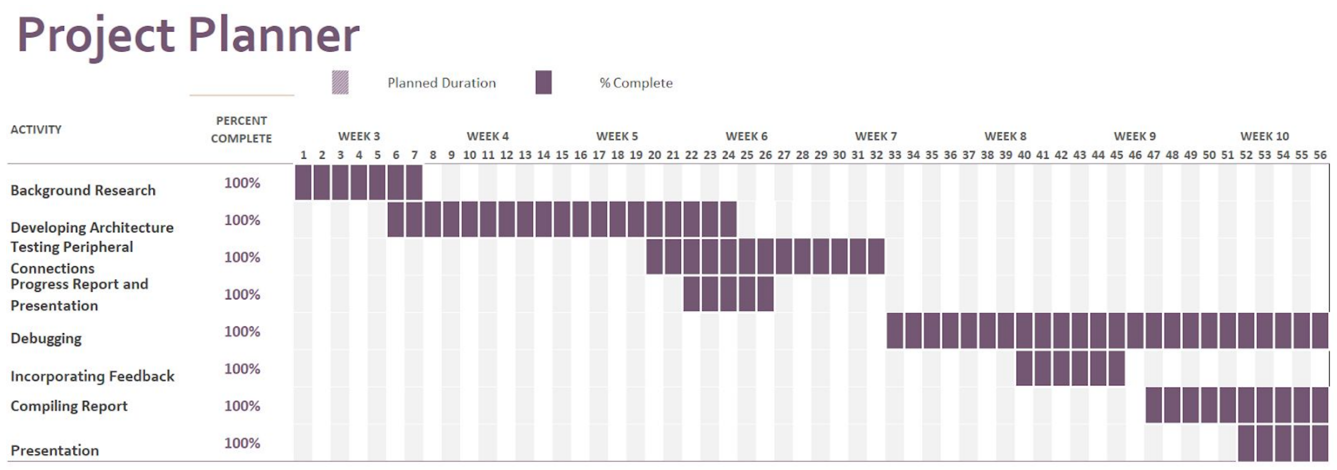
One limitation of our final design is that it works only with ASCII characters, while our initial design specifications aimed for the flexibility to work with non-ASCII characters such as Greek letters or symbols like the British pound and the Euro. Potential future steps for our program would be to incorporate non-Latin alphabets and ideograms. Finally, as we originally envisioned our design for use in chat messaging applications, incorporating data transmission into the device through a Raspberry Pi is another potential future step.

References

1. FIPS, “Announcing the ADVANCED ENCRYPTION STANDARD (AES),” NIST, Gaithersburg, MD, USA. [Online].
2. J. Daemen and V. Rijmen, “The Rijndael Block Cipher,” NIST, Gaithersburg, MS, USA. [Online].
3. .H. Khan, M.A. Al-Mouhammed, A. Almousa, A. Fatayar, A.R. Ibrahim and A.J. Siddiqui, “AES-128 ECB Encryption on GPUs and Effects of Input Plaintext Patterns on Performance,” in *15th IEEE/ACIS SNPD*, Las Vegas, NV, 2014, pp. 207-212.
4. N. Mathur and R. Bansode, “AES Based Text Encryption Using 12 Rounds with Dynamic Key Selection,” *Procedia Comput. Sci.* 2016, pp. 1036-1043.
5. A. Bogdanov, D. Khovratovich, C. Rechberger, “Biclique Cryptanalysis of the Full AES,” *ASIACRYPT* 2011, pp. 344-371.
6. aes.online-domain-tools.com

Appendices

Gantt Chart:


Project Planner
Planned Duration
% Complete
ACTIVITY
PERCENT COMPLETE
WEEK 3
WEEK 4
WEEK 5
WEEK 6
WEEK 7
WEEK 8
WEEK 9
WEEK 10
Background Research 100%
Developing Architecture 100%
Testing Peripheral Connections 100%
Progress Report and Presentation 100%
Debugging 100%
Incorporating Feedback 100%
Compiling Report 100%
Presentation 100%