CompArch Lab 3

Maggie Jakus & Logan Sweet

12 December 2017

1 Introduction

Advanced Encryption Standard (AES) is an encryption specification used by the National Institute of Standards and Technology (NIST). For our project, we decided to implement AES in Verilog. Our ultimate goal was to enter a message and a key, encrypt it, then enter the encrypted message and key back into our Verilog and recover the original message.

We chose this project because we were both interested in cryptography, but didn't have any experience actually implementing it. We also were interested in following a specification to see if we could implement something complicated

We found that one of the challenges of the project was not being able to easily follow our data through the algorithm. Since it gets encrypted and rearranged at every step, we had to create and test each of our modules separately, then link them together after we knew they worked alone.

2 Advanced Encryption Standard History

In 1997, NIST put out a request for encoding specifications after the previous standard, Data Encryption Standard was deemed less than satisfactory. Better optimization methods allow computing speeds increase over time and malignant decryption becomes more effective over time, so Data Encryption Standard was less secure and not as fast as it could be. The new standard was supposed to remain valid "well into the 21st century," and became compulsory encryption method for sensitive government information. The NIST opened a forum for comments after the announcement, and AES was fairly well-received.

Algorithms submitted for consideration were evaluated on Security (resistance of the algorithm to cryptanalysis, soundness of its mathematical basis, randomness of the algorithm output, and relative security as compared to other candidates), Cost (licensing requirements, computational efficiency on various platforms, and memory requirements), and Algorithm and Implementation Characteristics (flexibility, hardware and software suitability, and algorithm simplicity).

The first round of submissions had 15 candidate algorithms, and after multiple rounds of review, analysis, and call for public comments, the Rijndael (pronounced Rhine-Dahl) algorithm was selected for the AES specification. This method's name is a conglomeration of the names of its creators, and was fairly well-received. Since 2001, the Rijndael algorithm has served as the AES standard.

3 Advanced Encryption Standard Theory

AES encryption takes in a secret key, which can be 128, 192, or 256 bits long, depending on the level of security, and a 128 bit message to be encrypted. We will focus on the 128-bit key, and we envision both the key and the message as a 4x4 matrix where each entry is 8 bits (one byte) long. We read these from the upper lefthand entry and go down the columns left to right, such that the most significant byte is in the upper left corner and the least significant is in the lower right.

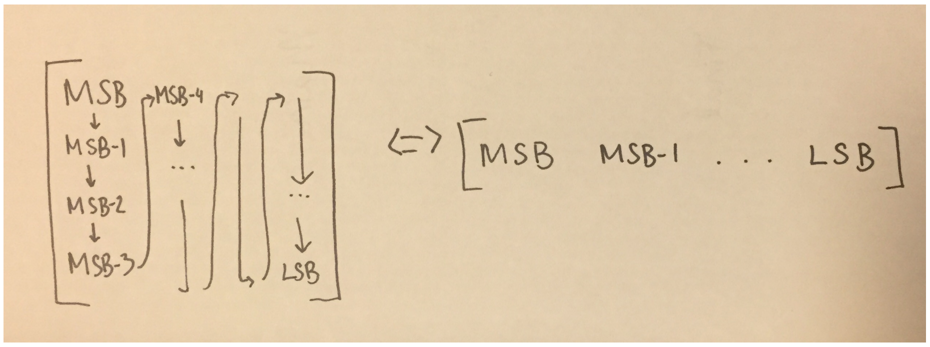


Figure 1: While we write 128-bit vectors in Verilog, we visualize the key and state as 4x4 matrices where each entry is 8 bits long. MSB = most significant byte, LSB = least significant byte.

Encryption in AES occurs in three "rounds." In each round, the message (known as the "State") and the key are manipulated. The manipulation focuses on two important principles of encryption: confusion and diffusion. These two principles come from Claude Shannon's 1949 "Communication Theory of Secrecy Systems" (Claude Shannon is the "father of information theory"). Confusion means that the relationship between the key and the encrypted message is not clear. Diffusion describes the relationship between the input message and the output encrypted bits; if we change a single bit of the plaintext, approximately half of the ciphertext bits should change. This is also known as the avalanche effect, where changing one input bit affects many output bits.

The three rounds can be thought of as an initial round, a main round, and a final round. The initial round consists of just the AddRoundKey manipulation. The main round consists of SubBytes, ShiftRows, AddRoundKey, and MixColumns. The final round consists of SubBytes, ShiftRows, and AddRoundKey. An overview of this can be seen in figure 2.

All operations in AES are performed in a Galois, or finite field. A finite field is, as the name suggests, a field with only limited numbers. We normally work in an infinite field - there is not max value you can reach. In a finite field, you eventually reach the end of the allowable numbers and so cycle back to the initial value. The Galois field used is GF(2^8), which contains 256 numbers (0 - 255). In binary, this limits us to 8 bit numbers.

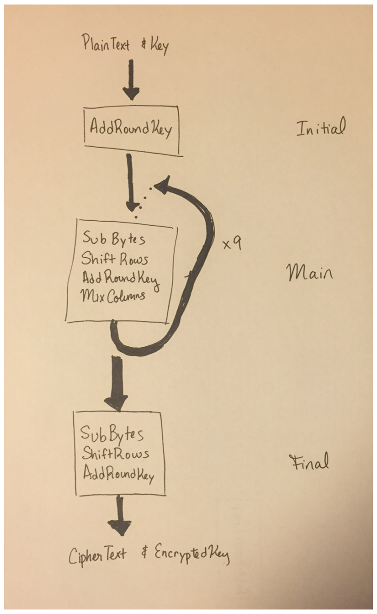


Figure 2: The three main rounds. The output of the main round is both an encrypted key and an encrypted state. These get fed back into the main round. After nine main rounds of encryption, the encrypted key and state are fed into the final round.

4 Encryption

In order to hide the message or data to be transmitted, encryption must occur before it is sent. This applies the characteristics of both confusion and diffusion and allows a message to be sent securely.

4.1 Key Expansion

In each round of AES, a new key is created based on the last round's key. Key expansion works on each column of the key individually. Here we describe the process that happens to one column of the four; the process is the same for all four columns. First, the bytes are shifted one to the left. The most significant byte does not have a free byte to roatate into, so as each byte moves up one level of significance, the the most significant byte becomes the least significant byte. A visual representation of this can be seen in figure 3.

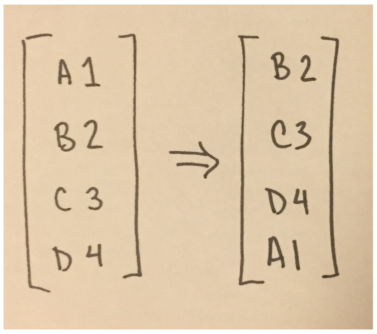


Figure 3: The 8 most significant bits become the 8 least significant, and all the others shift left 8.

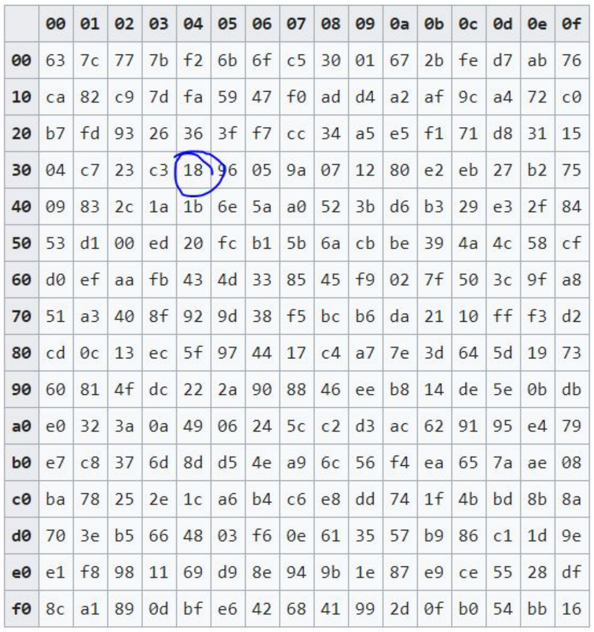
Next, each byte in the column is replaced with the corresponding value from Rijndael's S-Box. A substitution box ("S-Box") is a substitution cipher - for a given block of bits, you will always get the same substitution block of bits out. A good substitution box takes advantage of the avalanche effect, so that if just one input bit is changed, many of the output bits change as a result. A specific S-Box is used for AES (the Rijndael S-Box), and this is derived from some complicated math within the specified Galois field. Suffice it to say, we copied the S-Box from Wikipedia, after confirming with a couple online sources that Wikipedia did have the correct S-box. (figure 5).

This takes up space in a potential processor, so if hardware space is limited deriving the S-Box is more efficient than hard-coding it as we did. We spent some time on creating the relevant math to do calculations in the Galois field, but found that it did not align with our goals of focusing on encryption and might have been enough work to justify a project in its own right.

Finally, the most significant byte is XORed with a round constant. We represent this round constant with the function $RCON[i]$, where $RCON[i] = [x^{i-1}, 00, 00, 00]$. For example, $RCON[2] = [02, 00, 00, 00]$, where each value is in hexadecimal. This makes sense; $x^{(2-1)} = x$, which is represented by 0000 0010. This goes back to our Galois field - each 8 bit binary number represents a polynomial with the largest exponent allowed being x^7. You then have a new round key! This gets passed back into the key expansion for the next round of the algorithm. For each round, i increases by one. We chose to hard-code the ten $RCON$ values we knew we would use in encryption, although calculating them wouldn't be that difficult and would be more efficient.

4.2 SubBytes

In SubBytes, the 16 bytes in the state are replaced with the corresponding values found in the Rijndael S-Box (figure 5). This is a major source of confusion in AES since it makes it more difficult to associate a piece of the orinal message with a piece of the encrypted message.



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 4: The row specifies the most significant four bits of the entry. The column specifies the least significant four bits of the entry. For example, if you have the number 34 (34 = 0011 0100), you would substitute 34 with 18.

4.3 ShiftRows

In shift rows, each row in the state is shifted a different amount. The first row is left unchanged, the second row is shifted left one byte, the second shifts left two bytes, and the final row shifts left three bytes.

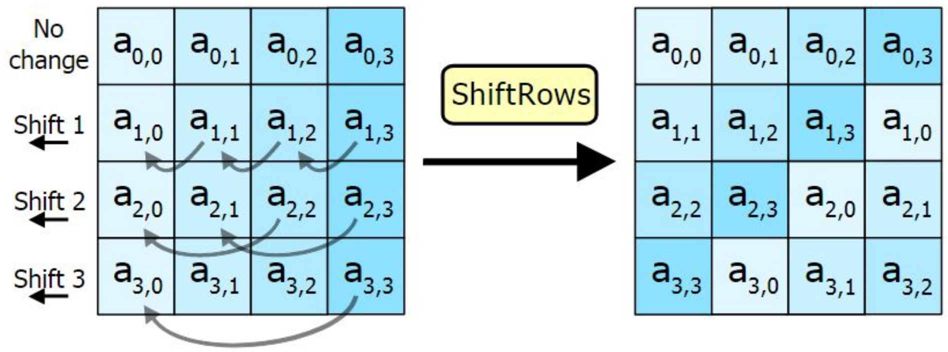

No change
Shift 1
Shift 2
Shift 3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Figure 5: A visualization of the Shift Rows operation for AES (blog.nindalf.com/posts/tech/implementing-aes/)

4.4 AddRoundKey

Add round key adds the value of the round key to the current state. In the Galois field, addition and subtraction are interpreted as a bitwise XOR, so this step is equivalent to a bitwise XOR between the round key and state matrices. This is simple and fast to implement in computer hardware, so it is convenient for widespread use in AES.

This step is easy to undo since XORing something with the same key twice results in the original input. In the equation below, the first input matrix 0110 is XORed with 1100. This result is XORed with 1100 a second time to recover the original 0110 matrix.

$$\begin{bmatrix}0\\1\\1\\0\end{bmatrix} xor \begin{bmatrix}1\\1\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\\0\end{bmatrix} xor \begin{bmatrix}1\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\\1\\0\end{bmatrix} \tag{1}$$

4.5 MixColumns

MixColumns works on each column of the state independently. True to its name, MixColumns mixes up the bits within a column. MixColumns treats each column as a polynomial with x^3 the maximum allowable exponent. We can think of a column of values $[b_0 b_1 b_2 b_3]$ as a polynomial $b_3x^3 + b_2x^2 + b_1x + b_0$. We then multiply this polynomial modulo $x^4 + 1$ by a constant polynomial $3x^3 + x^2 + x + 2$. Because this sounds difficult to implement in Verilog, we chose to follow an equivalent but simpler method of diffusion. This method is shown in equation 2.

$$\begin{bmatrix}2 & 3 & 1 & 1\\1 & 2 & 3 & 1\\1 & 1 & 2 & 3\\3 & 1 & 1 & 2\end{bmatrix} \begin{bmatrix}b_0\\b_1\\b_2\\b_3\end{bmatrix} = \begin{bmatrix}d_0\\d_1\\d_2\\d_3\end{bmatrix} \tag{2}$$

As you can see, each element d_x depends on all four values b_0, b_1, b_2, b_3.

At first, this method seemed difficult to implement as well. However, multiplying by 2 in GF(2^8) is the same as shifting left by one and then XORing by 1B (which is 0001 1011, or $x^4 + x^3 + x + 1$, and thus will let you know if you have exceeded your maximum allowable exponent). Similarly, multiplying by 3 is the same as multiplying by 2 and adding 1. Adding 1 is the same as XORing, so you multiply by 2 and then XOR with the original value.

5 Decryption

Once the message has been encrypted, it must then be decrypted for it to be of any use. The functions used in decryption are inverse functions of those that are used in encryption, and they happen in reverse order. Figure 6 shows how encryption and decryption proceed.

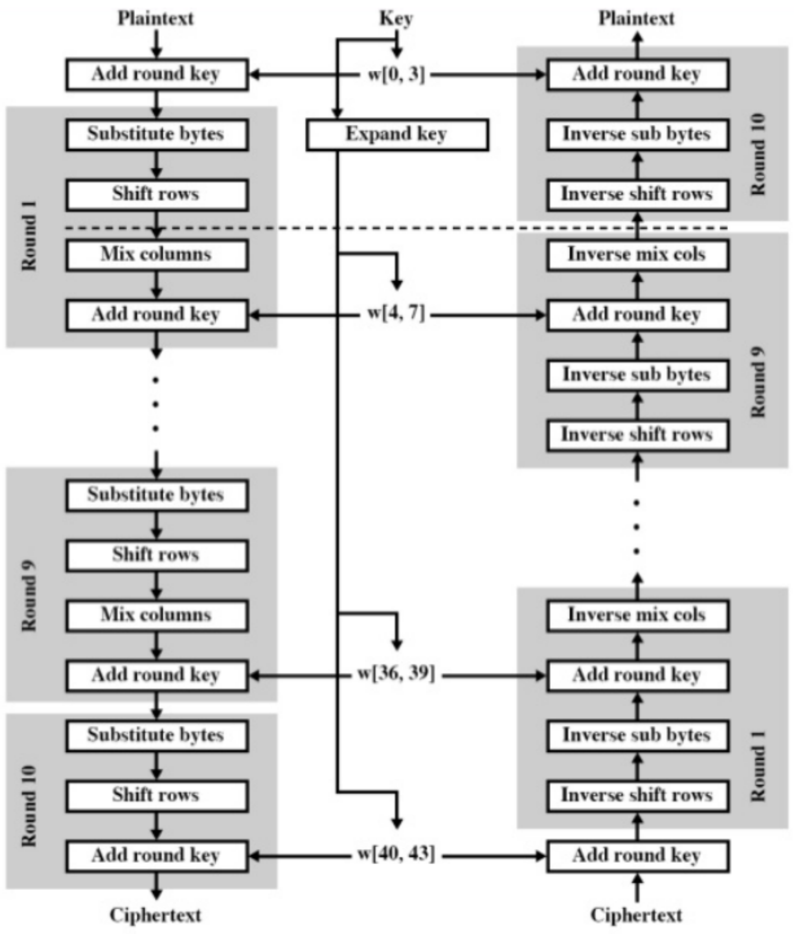


Figure 6: Decryption is encryption but in reverse order with inverted functions. `https://www.slideshare.net/ayyakathir/cryptography-and-network-security-52030373`.

5.1 InvKeyExpansion

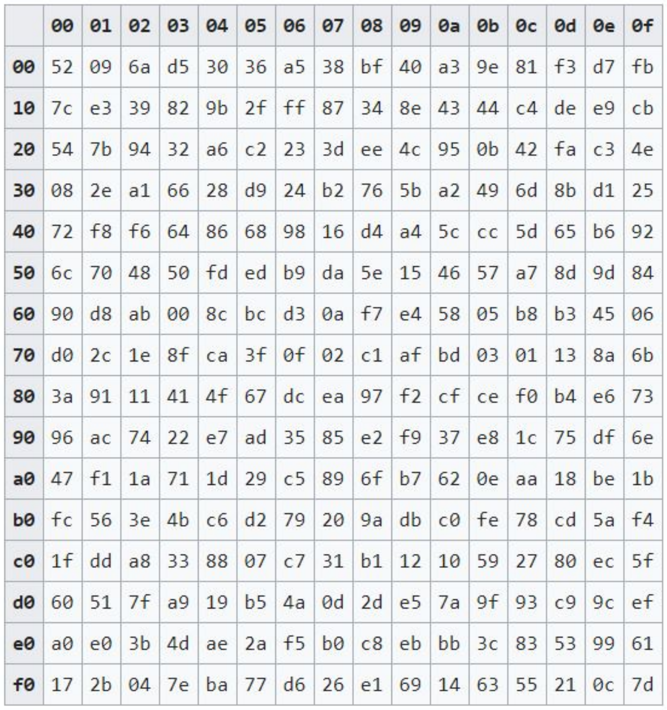
We begin decryption with the 10th encrypted round key, and we need to decrypt it to get back to the original key. In InvKeyExpansion, we do the inverse of the three steps we did in KeyExpansion. In KeyExpansion, the last step was to XOR the first byte with the round constant. In order to undo this, we can XOR the input (the encrypted key) with the same $RCON$ value we used in KeyExpansion. We then use Rijndael's Inverse S-Box to replace each value in the key. Finally, we need to rotate the column so that the least significant byte becomes the most significant byte.

5.2 InvShiftRows

The inverse shift rows operation performs the inverse of shift rows; that is, it performs the same operation but shifts right instead of left. The first row is left unchanged, the second row is shifted right one byte, the second shifts right two bytes, and the final row shifts right three bytes.

5.3 InvSubBytes

In InvSubBytes, you substitute each byte of the state with the corresponding value from Rijndael's Inverse S-Box. This is calculated through complicated math similar to that which was done to find the original S-Box. Once again, we hard coded rather this rather than calculating values each time.



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 7: We hard-coded this Inverse S-Box we found on Wikipedia.

5.4 InvMixColumns

In MixColumns, we treated each column as a polynomial and multiplied it by a constant polynomial $a(x) = 3x^3 + x^2 + x + 2$. The inverse of this is $a'(x) = 11x^3 + 13x^2 + 9x + 14$. We multiply each column of the state with the corresponding matrix. This is very similar to what we did in MixColumns, as can be seen in equation 3.

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} \tag{3}$$

6 Implementation

We built this from the simplest module to the most complex module. We began by building the base modules for encryption. These are AddRoundKey, ShiftRows, SubBytes, MixColumns, and KeyExpansion. We tested these modules to make sure they worked as we expected on their own. This was important since it would be more difficult to follow signals or values that were interpreted incorrectly in GTKwave since they would be encrypted. The early testing of the individual modules gave us higher confidence that our ultimate AES encryption would work correctly.

Next, we created the three modules that represent the three different "modes" we go through during encryption. These modules represent what occurs during round 0 (this is the "RoundE" module), what occurs during rounds 1-9 (this is the "RoundA" module), and what occurs during round 10 (this is the "RoundB" module). For each round, the inputs are a state and a key - for Rounds A and B, there is also an input iterate value (which is used in $RCON$ in KeyExpansion).

The zeroth round uses RoundE. In Round E, the original state is XORed with the original key. These values are passed into Round A, which occurs 9 times. In Round A, the original state is sent through SubBytes, ShiftRows, MixColumns, and AddRoundKey. The Key is also expanded. Iterate goes up by 1 each time.

The tenth and final round uses RoundB. RoundB is the same as RoundA except without the MixColumns. This outputs an encrypted key and an encrypted message. T

After creating these modules (Rounds A, B, and E), we connected them using a mux that will choose which output state and key to pass on to the next round. This is controlled by the FSM, which increments iterate and sets the controls. Because the output of the mux feeds straight into the inputs of rounds A, B, and E, we have a D Flip-Flop that sends the updated value at the beginning of every clock cycle. This way we aren't constantly spamming the modules which are all based on combinational logic. Finally, we pull the output of the system from the output of the mux. After Round B, the FSM sets the smaller mux to pass the new value to the output.

The process for decryption is very similar. We began by building the AddRoundKey, InvSubBytes, InvShiftRows, InvMixColumns, and InvKeyExpansion modules. We combine these into the modules RoundC, RoundD, and RoundF. RoundC is equivalent to RoundA, RoundD is equivalent to RoundB, and RoundF is equivalent to RoundE. Once again, RoundF is the zeroth round, RoundC is rounds 1-9, and RoundD is the tenth and final round. These are combined using a mux and D Flip-Flop, just as they were in encryption. For Rounds C and D, iterate goes decreases by 1 starting at 10.

These are combined in the MainAlgorithm module. This is summarized in figure 8.

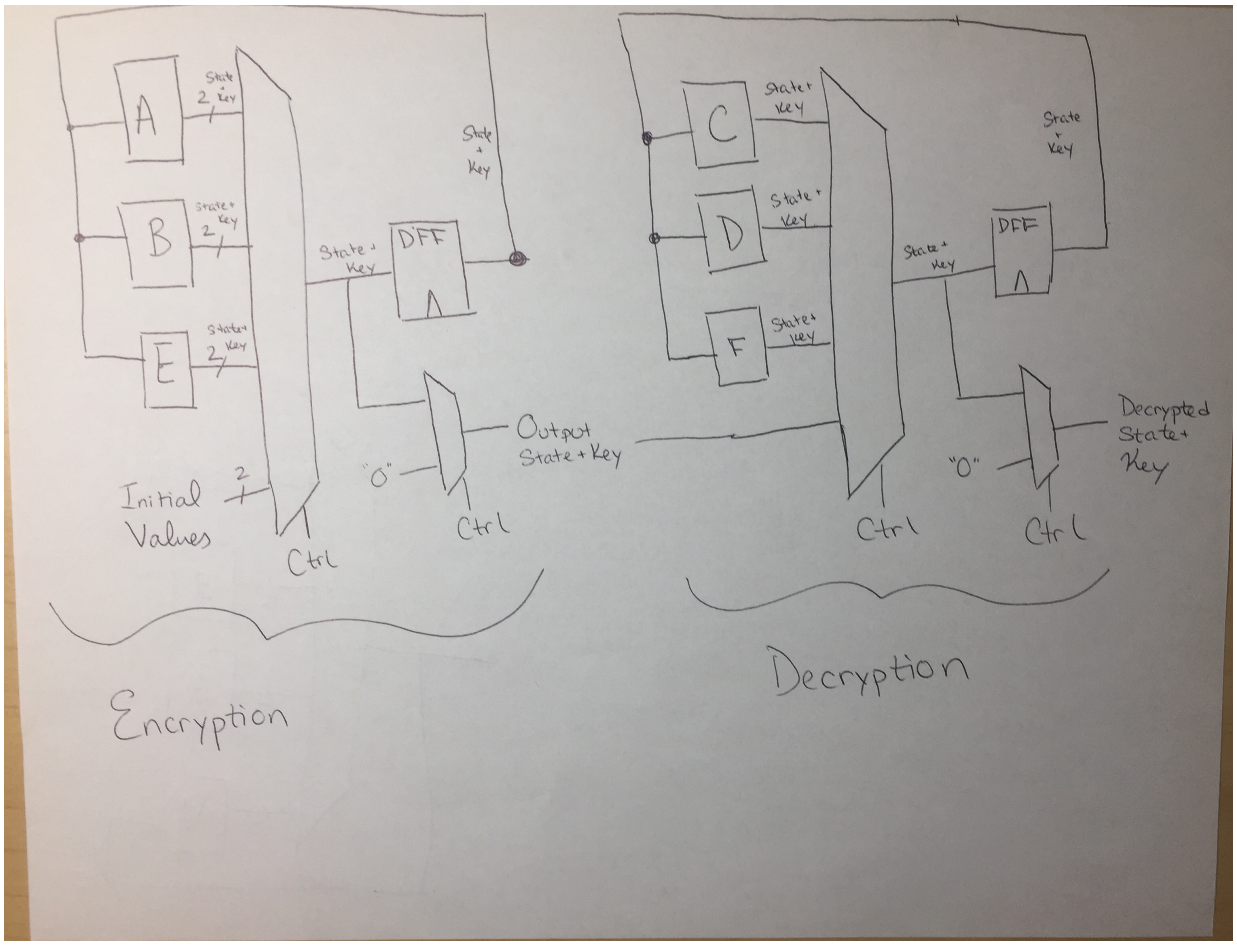


Figure 8: An overview of the system.

7 Other Considerations

AES should not be used on its own for encryption, since that is a less secure mode of operation. Using AES on its own is A form of Electronic Code Book (ECB). In ECB, each block of regular text has a corresponding block of encrypted text. This allows patterns to emerge due to repetition of words or phrases, which can make pieces of the original text easier to decipher. AES has multiple steps to mix and remix data so decryption without the key is not easy, but security can be increased by using a different mode of operation.

To avoid this issue, AES should be used with Cipher Block Chaining (CBC). CBC is a more secure mode of operation for a block cipher. As you can see in the figures below, the block cipher encryption (AES, in our case) is the only thing providing diffusion and confusion in ECB. With only one ‘level’ of action being taken on the message, this is less secure than CBC. CBC uses the outcomes of one block cipher’s encryption in the calculation of the next one using XOR. (XOR, as mentioned above, is easily reversible and efficient) For example, in order to decrypt the ciphertext on the far bottom right in figure 10, you would need to know both the key for that block cipher encryption as well as the ciphertext from the previous iteration. To know the ciphertext from the previous iteration you would need to know both the key for that block as well as the ciphertext from the previous iteration, and so on until you can crack the encryption of the first level using the key, block cipher, and initialization vector.

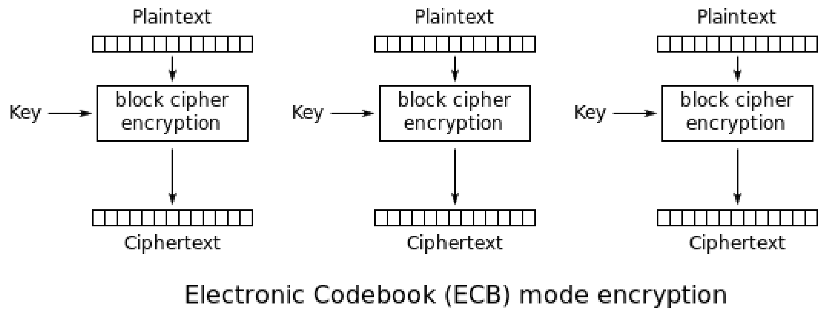


Figure 9: Electronic Code Book diagram

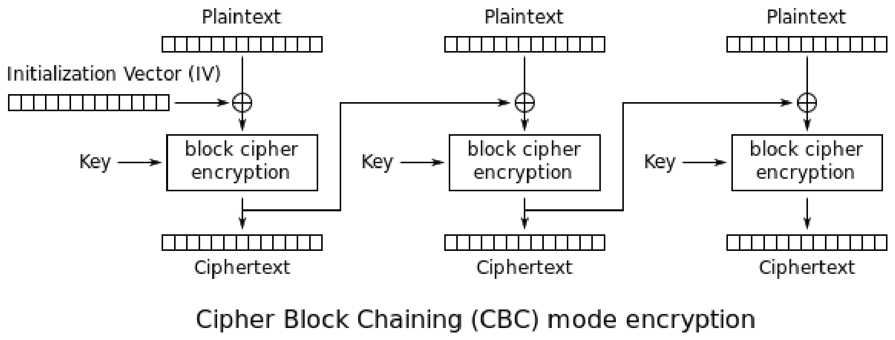


Figure 10: Cipher Block Chaining Diagram

8 Future Directions

If we were to take this project farther, we would implement additional key lengths. AES can be used with 128, 192, or 256 bit keys. To do this, we would also have to adjust the number of cycles of transformations. 128-bit encryption takes 10 rounds, while 192 uses 12 and 256 uses 14.

There is significant room for improving the efficiency of our code. For example, as was previously mentioned, we could calculate the S-Box and Inverse S-Box, rather than hard-coding them, which would save space but would require much more thought on our part. Similarly, we could have combined Rounds A and B and Rounds C and D to save space, but we would have had to include a mux and that just seemed too difficult in the moment. Rounds E and F are the same - there is no reason for us to have two, but we included two anyway.

As of the writing of this (11 PM on Dec. 11), the key expansion and inverse key expansion worked perfectly - the key that went into encryption was the same that came out of decryption. Because we had timing issues, we were not using the MainAlgorithm module but were instead hard-coding the output of the Encrypt module as the input of the Decrypt module. We individually tested each base level module to make sure that running the encryption and decryption of each would return the initial value (for example, if you encrypt "123" with MixColumns and then decrypt the result with InvMixColumns, you get "123" as your result). We also ran an encryption and decryption where we only used rounds 0 and 10. This was successful - we were able to get out what we put in. Because of these successes, it is too bad that we don't get the decrypted value when we run the full Encryption/Decryption. We believe that there must be one typo in the S-Box - we incorrectly listed one value. Because we value our sanity, we will not go looking for it tonight. This was not an issue with the tiny encryption/decryption (where we skipped rounds 1-9) because we fortunately managed to not need whichever value has the typo. Because there is so much more confusion and diffusion when we incorporate 9 additional rounds of encryption and 9 additional rounds of decryption, the one typo is highly effective at completely scrambling what we have to say!

9 Reflection

9.1 Maggie

Crypto Stack Exchange is awesome and the few people who are on it give great answers! I really liked this assignment - I found it simple enough to wrap my head around (no confusing timing/controls) and interesting. It was also exciting to see whether the correct answer would come out. I'm very satisfied with where this ended up (almost functional, but with the pesky S-Box typo). With another day or two to work on this, I'd have time to find the typo and hopefully get everything working properly.

I liked this assignment because I finally felt like I knew what was going on in Verilog, even if that was often not the case. I'd like to thank Ben for explaining regs vs wires and combinational vs stateholding logic to me about 5 times over the course of this project. I liked that I got to learn about macros and improving the structure and hygiene of my code.

There were a lot of good puzzles in thinking about this project. It was tricky to wrap my head around how exactly a specific module worked, how timing worked, or what exactly a Galois field is and how can I get around directly dealing with that in Verilog. I liked that I had the opportunity to learn about a lot of different but related things. This project was interesting and challenging, but just the right amount. I have no idea how much time I spent on this, but it was probably quite a lot, since I remember working on it in 3-4 hour intervals, and I did that a significant number of times. Overall, stellar project, and I'll probably try to find the typo in the S-Boxes sometime soon.

9.2 Logan

I definitely learned quite a bit as part of this project, but not as much as I had hoped. Most of my learning was focused around cryptography and the different ways that encryption functioned and how effective those methods were, along with some study of the history of NIST and AES.

I have a crazy overcommitted semester, and as a result of that I was a less that stellar partner to Maggie. (I was actually pretty bad) Most of the code that I wrote was in implementing the Galois Field, which we decided was not within the scope of the project after 4 hours and I did not complete. This code is in the GField Folder.

I learned a lot from reading and understanding the code that Maggie wrote, but I did not contribute to writing as much as I would have liked since I was usually already in meetings when Maggie could meet with Ben, and then would get behind on what the current issues and paths forward in the modules were. I think that maybe if we had scoped the project to be a little smaller or if I had been better about scheduling later on in the project I would have been able to contribute more. At the beginning we were good about scheduling and meeting often, but I did not do a good job of that later on.

10 References/ Resources/ Images

federalregister.gov/documents/2001/12/06/01-30232/announcing-approval-of-federal-information-processing-standard-fips-197-advanced-encryption-standard

nvlpubs.nist.gov/nistpubs/jres/104/5/j45nec.pdf

csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-developmentoverview

moserware.com/2009/09/stick-figure-guide-to-advanced.html

nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

csrc.nist.gov/csrc/media/publications/fips/140/2/final/documents/fips1402.pdf

samiam.org/rijndael.html

en.wikipedia.org/wiki/File:ECB$_{e}ncryption.svg$

en.wikipedia.org/wiki/File:CBC$_{e}ncryption.svg$

lideshare.net/ayyakathir/cryptography-and-network-security-52030373

blog.nindalf.com/posts/tech/implementing-aes/

https://crypto.stackexchange.com/questions/2402/how-to-solve-mixcolumns

https://crypto.stackexchange.com/questions/2418/how-to-use-rcon-in-key-expansion-of-128-bit-advanced-encryption-standard

https://crypto.stackexchange.com/questions/2569/how-does-one-implement-the-inverse-of-aes-mixcolumns