An FPGA Implementation of AES-128 Encryption

Ezra Santos

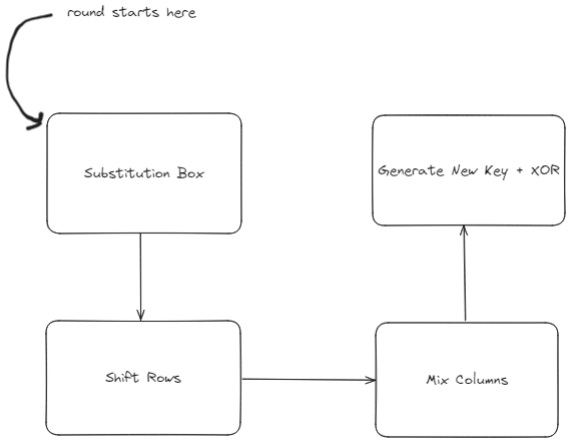
August 22, 2024

Chapter 1

Introduction to AES-128

The Advanced Encryption Standard, as the name suggests, is a reliable way to encrypt data. In fact, it's approved by the NSA and is used to classify information. The most surprising part about this is that AES-128 is relatively easy to learn: all you need is binary arithmetic, linear algebra, and basic Galois theory.

AES-128 takes a two 128-bit inputs: one for the message to be encrypted, and one for the key to be used. The process starts with a bitwise XOR between the cipher and the key. Then, a typical AES-128 round goes like this:



Note that the last stage, generating a new key, can be parallel to the entire process. I put all the steps as a sequence for a simpler design since time is not something this design is worried about.

For AES-128, there will be ten rounds. It's good to note that the final round will not include the "Mix Columns" part, as it has been proven that it does not add that much security.

Before I briefly explain these steps, it's important to represent 128-bit messages into what we call a "byte array." From the most significant byte b_0, we construct the following array:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

This will be incredibly useful in explaining some of the steps.

1.1 Substitution Box

The substitution box is a way of transforming data so that no two bytes will produce the same byte. In applications, people often use a look-up table. However, if you want to use Galois theory, you can map an input x to its multiplicative inverse in $GF(2^8) = GF(2)\frac{[x]}{x^8+x^4+x^3+x+1}$. Then, you take that multiplicative inverse and apply the following affine transformation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

1.2 Shift Rows

Shifting the rows is very simple. We take our current byte array and do the following:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix} \rightarrow \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{bmatrix}$$

1.3 Mix Columns

Mixing columns involves the multiplication of the byte array:

$$\begin{bmatrix} s_0 & s_4 & s_8 & s_{!2} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

A good thing to note is that any form of addition here is an XOR. Thus, when multiplying by 3, you shift the byte once to the left and XOR it with the original byte. If this is not clear, I suggest watching a quick YouTube video about $GF(2^8)$.

1.4 Round Key Generation

A fun thing about this round is that you can actually generate all of the keys for every round from the beginning, I just chose not to for simplicity sake. We take the original key and its byte array, and take each column as a four-byte "word." The goal is to generate the 40 remaining words. For a group of words $w_i, w_{i+1}, w_{i+2}, w_{i+3}$, we first generate w_{i+4} with the following steps done to w_{i+3}:

1. Rotate the word by one byte.
2. Perform a substitution using the same substitution box.
3. XOR the resulting byte with a "round constant," which is different for every round.

After that, we do the following to the next three words: $w_{i+5} = w_{i+4} \oplus w_{i+1}$, $w_{i+6} = w_{i+5} \oplus w_{i+2}, w_{i+7} = w_{i+6} \oplus w_{i+3}$. Repeat this until all the keys are generated.

If you want an in-depth explanation on what's going on in each step, I highly suggest visiting Purdue's explanation: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf

Chapter 2

FPGA Implementation

We will split this into the following parts: the encryption module, the RX module, and the TX module. The FPGA will receive the text and the key from the PC and it will also transmit the ciphered text to the PC as well. This data transmission will be done through UART.

2.1 Encryption

First, I programmed every part of a round: sbox, shiftrows, mixcolumns, and roundkey. For the sbox, I did not bother using Galois theory and just used a look-up table to make everything quicker (this **code** made everything easier instead of just typing everything out). The shiftrows module is very easy to code. For mixcolumns, I made the input four bytes which represents one column. Thus, in the final design, each round will have four mixcolumns modules. The roundkey module is an amalgamation of what I've already done with the other modules, plus some Galois theory and basic binary arithmetic (and by Galois theory, I meant just XORing with an irreducible polynomial, I just wanted to sound fancy).

Each "round" module will consist of all parts mentioned above, while the "roundfinal" module will exclude the mixcolumns module.

In the latter half of this journey, there was a problem of ciphered messages being submitted continuously instead of just once. This was solved by incorporating a handshake mechanism for each part. When the user presses the enter key, it will send a signal to the sbox module of the first round, and when that module is done, it will send a signal to the shiftrows module. This signal passing occurs to all parts of all rounds until the ciphered text is completely ready for transmission.

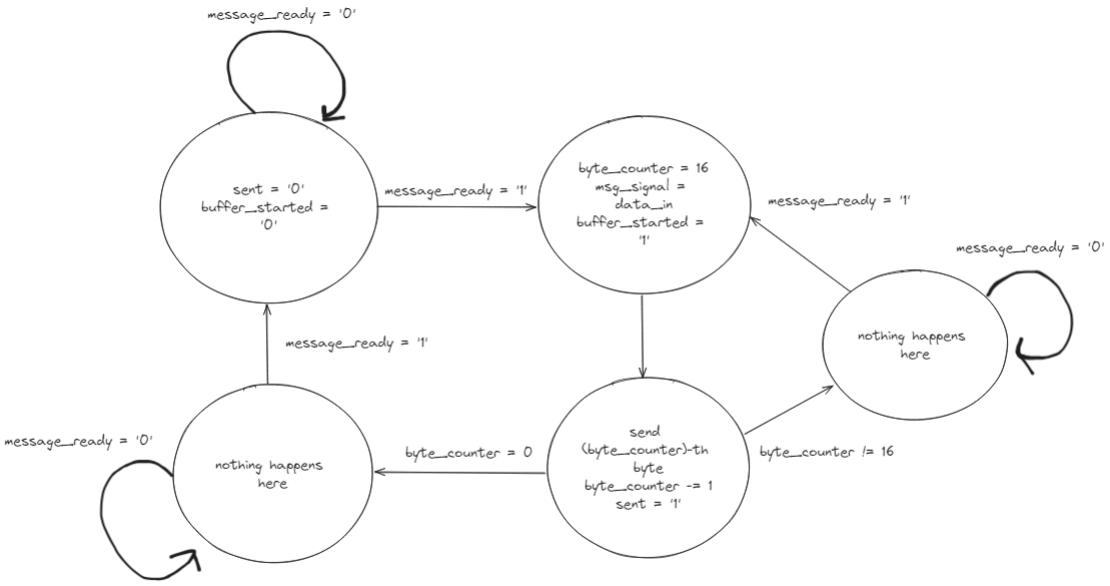
Five test vectors were used to testbench the encryption module.

2.2 TX

I was not familiar with UART transmission before, so I had to learn a lot of this from scratch. In fact, half of the project was just me trying to figure out UART; the AES side of things was actually not that hard.

There will be three parts to the transmission side of things: an intermediate, a buffer, and the transmitter. The transmitter is your usual UART transmitter that you can just copy from some random GitHub page, but for intellectual pursuit, I made one from scratch. The intermediate serves as a medium between the encryption module and the buffer: the ciphered text is stored in the module and is notified whenever the encryption module is done. Whenever the buffer is ready, it sends a signal to the module that it's ready for the ciphered text, and then the message is passed on. This solved half of my problems solely because it was hard trying to pass data only once when the transmission and the encryption modules are working at different clock rates. The buffer is a necessary component for byte sending. Basic UART transmitter

modules can only take a byte as an input, and that byte is sent into bits in a serial manner. The buffer handles the 128-bit ciphered text and sends each byte to the transmitter, and when each byte is done, the buffer gets notified and sends another byte. For reference, here is a transition state diagram for the TX buffer:



The process starts with the stage on the top left. The "sent" signal notifies the UART transmitter module that a byte has been sent. The "buffer started" signal notifies the intermediate that it has received the ciphered text, and this prevents data to be sent more than once. The "message ready" signal is received from the intermediate once the intermediate has received the ciphered text.

As you will see later on, the RX buffer kind of works the same way.

2.3 RX

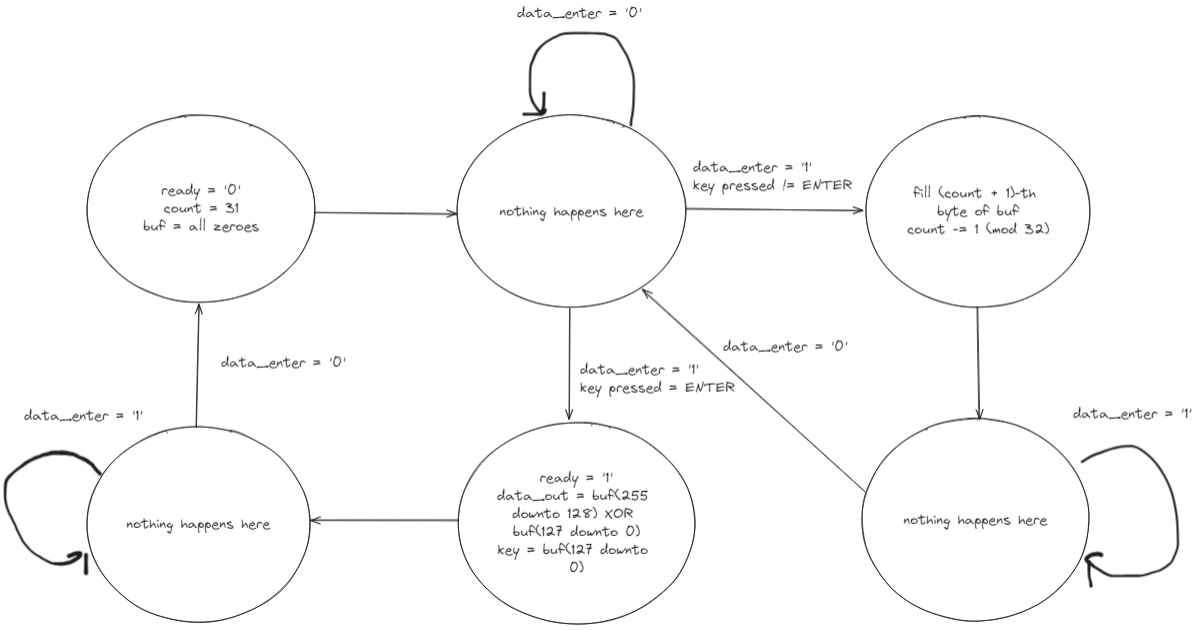
Unlike the TX side of things, the RX side does not need an intermediate simply because the UART clock is slower than the system clock, and thus any signal sent out by the RX is guaranteed to be picked up by the encryption module.

There will be two parts: the UART receiver module, and the RX buffer. Again, code and details about UART reception is prevalent online, I suggest relying on that instead of me. However, just like the TX buffer, I'd like to explain the RX buffer as well since this is an original part of the design.

Whenever the UART receiver receives a byte from the PC, it will be sent to the buffer. The 256-bit signal inside the buffer will have its most significant byte changed to whatever the UART receiver spits out. When the user puts another byte in, the buffer moves to the next significant byte. If the entire 256-bit signal is filled, the buffer wraps around and starts from the beginning. The 256-bit signal is sent whenever the user presses enter. Why 256? Well, the user also gets to input the 128-bit key after the 128-bit text.

So for example, if I immediately press enter, both the text and the key will be all zeroes. If I put "abcdefghijklmnop" for both the text and the key and I press the letter "q", the text would be"qbcdefghijklmnop."

Here is a transition state diagram:



The "buf" signal is a 256-bit message that stores what the user inputs. The "data enter" signal is received from the receiver whenever a key has been pressed. The "ready" signal acts as the "round start" signal for the sbox module of the first round (if you remember how that mechanism worked).

Chapter 3

Results and Limitations

After a few weeks of trying to put all the components together properly and slamming my head on the wall thinking of proper handshake mechanisms, I got something that works almost perfectly.

A demonstration is shown in a YouTube video here, but I will explain it in this document as well. One switch works as a reset whenever the user wants to do another encryption. With Tera Term open, the user can input whatever they want on their keyboard, as long as they're aware of what's happening in the buffer (e.g. wrapping around when 256 bits have been filled). Once the user hits enter, Tera Term displays the encrypted message.

If you know how UART transmission works, you might've already predicted one of the limitations. As a huge reminder for myself to potentially revisit this project when I'm not burned out anymore, here are a list of limitations that can be fixed:

1. Not all hex values have a visible ASCII representation. A possible improvement to this is to have another component that uses a look up table to turn the ciphertext into the bit representation of its hex value. This sounds really cursed, so if there's a better way to do this, don't hesitate to reach out. If I just want to confirm the 128-bit ciphertext, I can study up on ILAs and implement it on Vivado.

2. This problem is somewhat the same for the receiving side of things. You can only input bytes that are visible on the keyboard. A similar fix could be implemented: we can let the user input the hex values and use a look up table. Again, this is also somewhat cursed.

3. For some inputs, the output in Tera Term seems to be long, although most visible keys are correct. An example could be just pressing "a" and then pressing enter. This results to a line skip. This is a weird one, so this is a priority.

Another fix for the first two problems is an external software that will manually put hex values to the terminal and display the hex values of the transmitted bytes. I might look into this as well.

Chapter 4

Conclusion

I'm aware that this is not a perfect project, and there are still some minor issues to be fixed. However, I wanted to show this at 99% completion since I was personally excited to share what I've accomplished, since this is the first time I've embarked on a somewhat-large personal project.

I haven't looked at other FPGA implementations of AES encryption (except that one substitution box code), so I personally don't know if what I'm doing is unconventional. Since this design is well-pipelined (I think), this design passed basic timing requirements as expected. I tried my best to not break the laws of hardware programming and thought of how logic is generated into hardware. Hopefully, it's good enough practice for whatever project I'll do in the future.

If you have any questions, suggestions, or comments, feel free to email me at ezramsantos@gmail.com.