A Hardware Accelerator for AES

by Saurav Sachin Kale (EE19B141), Surya Prasad S (EE19B121), Ruban Vishnu Pandian V (EE19B138)
EE2003 Git Link

December 11, 2021

1 AES, the algorithm

The Advanced Encryption Standard, or AES, is a specification for data encryption. AES takes in 128 bits of input, and generates 128 bits of corresponding output (encryption or decryption) given a key. AES has three modes depending on the length of this key. AES-128 accepts a 128 bit key, AES-192 and AES-256 defined similarly. AES encrypts or decrypts by repeatedly applying a set of operations (called a round) to the input. The number of rounds is fixed for each variant of AES as follows:

- 10 rounds for AES-128
- 12 rounds for AES-192
- 14 rounds for AES-256.

Each round requires a unique Round Key. Therefore enough round keys are extracted from the provided key (called the Short Key) using a set of operations called Key Expansion the details of which are elaborated in a later section.
Each round consists of the following operations as specified in Figure 1 for encryption and decryption as shown:

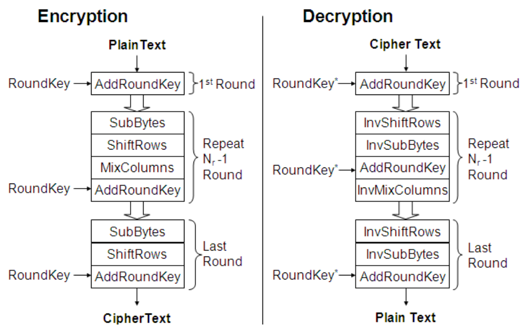


Figure 1: Flowcharts for AES encryption and decryption [3]

Each operation is done for all the bytes of the 128 bit input arranged in a 4x4 matrix in column major fashion, where each cell is one byte. We shall from now on refer to the input purely as a 4x4 matrix and call it the “state”. All operations must produce one output byte for one input byte, hence operate in a finite field where each number in the field is one byte long. This is called a galois field over 2^8 written as $GF(2^8)$. We now explain how we handled some of the important boxed operations in the flowchart.

1.1 SubBytes

This is a nonlinear function also known as S-Box which maps one byte to another byte in $GF(2^8)$. Each of the 16 cells of the state matrix are replaced by its mapped byte. This mapping is cleverly selected to be hard to model or back calculate. This is one of the reasons why AES is secure. Finding better mappings which translate to more compact and secure implementations are a topic of research, and one such implementation is what we will adapt in our case. Usually the S-Box is defined as follows

$$SBox(a) = \begin{cases} Ma^{-1} + N & a \neq 0 \\ 0 & a = 0 \end{cases}$$

Where M is an 8x8 binary matrix, $N \in GF(2^8)$ and a^{-1} is the multiplicative inverse in $GF(2^8)$. The irreducible polynomial for this field over which all the finite field operations are defined is $q(x) = x^8 + x^4 + x^3 + x + 1$. Multiplicative Inverse calculation is an expensive operation, therefore we try to optimise it. We have taken advantage of the fact that there exist isomorphisms between $GF(2^8)$ and a composite field of the same cardinality. Certain composite fields which are isomorphic to $GF(2^8)$ lead to simplified and faster computations of the multiplicative inverse. We first define a mapping between $GF(2^8)$ and the composite field (which is just a simple 8x8 matrix), then find the inverse in the composite field, and use the inverse mapping to get the inverse in $GF(2^8)$.

1.2 Inverse SubBytes

Inverse S-Box is formally defined as:

$$invSBox(b) = \begin{cases} M^{-1}(b+N)^{-1} & b \neq 0 \\ 0 & b = 0 \end{cases}$$

Again in this case the computation of the multiplicative inverse is optimized using the composite fields approach mentioned above.

2 Peripheral

The need for a dedicated peripheral for AES was felt since the implementation with C was found to be slow (detailed performance measurements are discussed in section 4). Our goal is to accelerate the execution of this algorithm with dedicated hardware. We now present the design we came up with for such a peripheral. A notable feature is that encryption and decryption can run together as long as the short key is the same. In a realistic workflow, the key is unlikely to change often. Hence we can utilize parallelism to our advantage and get better performance.

2.1 Key Expansion

Key expansion was implemented using a state machine which generates one 128 bit round key per clock cycle. This is significant, since the psuedocode provided in [2] takes one 32 bit word per iteration, hence producing one round key takes 4 iterations. Every clock cycle, four 32 bit words are operated on, and the results concatenated to produce a 128 bit round key, which is stored in a dedicated set of registers (key_mem in figure below) when the programmer issues a key expansion command. The encrypt and decrypt modules then read from the key_mem the round keys they need to actually perform the encryption/decryption. Key_mem is capable of parallely supplying keys to both Encrypt and Decrypt modules.

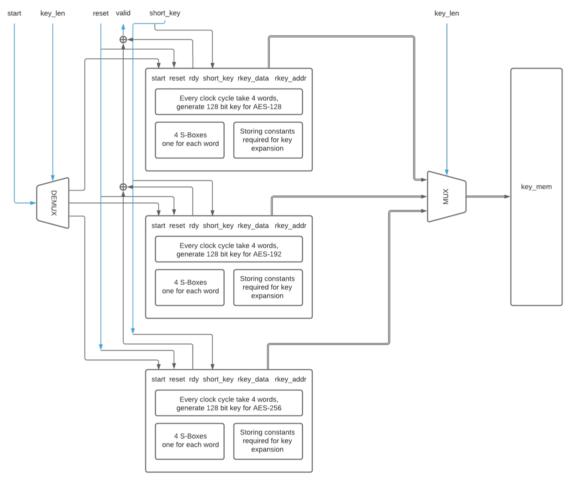

start
key_len
reset
valid
short_key
key_len
DEMUX
MUX
key_mem
start reset rdy short_key rkey_data rkey_addr
Every clock cycle take 4 words, generate 128 bit key for AES-128
4 S-Boxes one for each word
Storing constants required for key expansion
start reset rdy short_key rkey_data rkey_addr
Every clock cycle take 4 words, generate 128 bit key for AES-192
4 S-Boxes one for each word
Storing constants required for key expansion
start reset rdy short_key rkey_data rkey_addr
Every clock cycle take 4 words, generate 128 bit key for AES-256
4 S-Boxes one for each word
Storing constants required for key expansion

Figure 2: Simplified Flow for Key Expansion

2.2 S-Box

The S-Box is generally implemented using two methods. In the first method, an S-Box mapping is directly stored as a look up table, and then the substitute byte is directly obtained from it. The second way is to use composite finite field arithmetic and inverse to compute the substitute byte. We chose to go with the second approach since composite finite field methods give more compact and secure implementation [1]. As mentioned in [1], we split the $GF(2^8)$ field into a composite field $GF((2^4)^2)$, since it provides better utilization of FPGA hardware (taking advantage of the 6 input LUTs). We are working with the below mentioned polynomials for the composite finite fields:

Q(z)	P(y)
$z^4 + z + 1$	$y^2 + y + 13$

Functions were created for each of the steps shown in the block diagram below (for ease of use), and the inverse map and affine transform step is combined into one for better resource utilization.

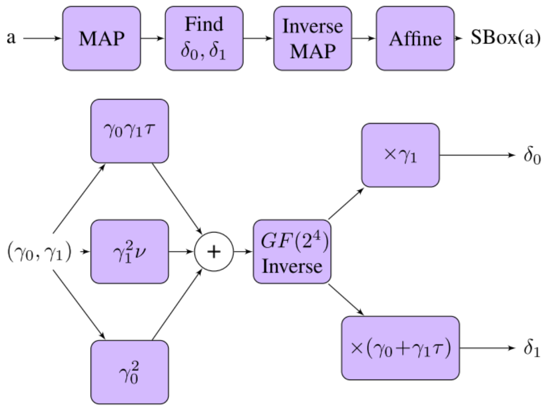
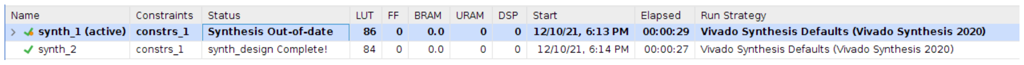


Figure 3: High level view of the AES SBox implemented with composite fields[1]

On synthesis using Xilinx Vivado v2020.1 the following resource utilizations were observed for our S-Box:



Name	Constraints	Status	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
> ✓ **synth_1 (active)**	**constrs_1**	**Synthesis Out-of-date**	**86**	**0**	**0.0**	**0**	**0**	**12/10/21, 6:13 PM**	**00:00:29**	**Vivado Synthesis Defaults (Vivado Synthesis 2020)**
✓ synth_2	constrs_1	synth_design Complete!	84	0	0.0	0	0	12/10/21, 6:14 PM	00:00:27	Vivado Synthesis Defaults (Vivado Synthesis 2020)

Figure 4: Vivado resource utilization reports with Vivado synthesis defaults and default constraints. The one highlighted in blue is our implementation, while the other is the implementation in [1]

Hence the S-Box was successfully implemented taking cues from the high level design in [1], keeping the resource utilization approximately the same, yet fulfilling our other design requirements.

2.3 Encryption

Encryption is implemented as a state machine which goes through all the rounds depending upon what variant of AES is being used, and gets all the round keys using the key_mem described in Section 2.1 Reference being [7]

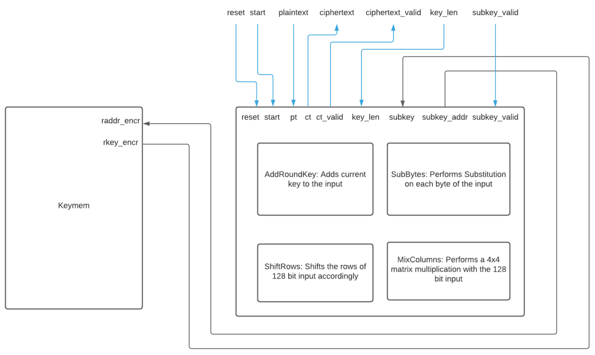

reset
start
plaintext
ciphertext
ciphertext_valid
key_len
subkey_valid
Keymem
raddr_encr
rkey_encr
reset
start
pt
ct
ct_valid
key_len
subkey
subkey_addr
subkey_valid
AddRoundKey: Adds current key to the input
SubBytes: Performs Substitution on each byte of the input
ShiftRows: Shifts the rows of 128 bit input accordingly
MixColumns: Performs a 4x4 matrix multiplication with the 128 bit input

Figure 5: Simplified Flow for Encryption

2.4 Decryption

Decryption is implemented as a state machine which goes through all the rounds depending upon what variant of AES is being used, and gets all the round keys using the key_mem described in Section 2.1. Note that this accesses the keys in reverse order as Encryption. Reference being [7]

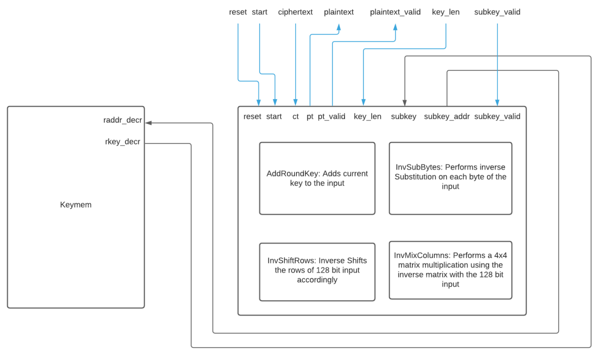

reset
start
ciphertext
plaintext
plaintext_valid
key_len
subkey_valid
raddr_decr
rkey_decr
Keymem
reset
start
ct
pt
pt_valid
key_len
subkey
subkey_addr
subkey_valid
AddRoundKey: Adds current key to the input
InvSubBytes: Performs inverse Substitution on each byte of the input
InvShiftRows: Inverse Shifts the rows of 128 bit input accordingly
InvMixColumns: Performs a 4x4 matrix multiplication using the inverse matrix with the 128 bit input

Figure 6: Simplified Flow for Decryption

3 Interfacing with PicoRV32

We integrated the peripheral to the PicoRV32 CPU using the AXI4-Lite Master standard. The programmer is given control for feeding the required inputs and control signals to specific memory mapped addresses. The programmer can separately and/or simultaneously fire the encrypt, and decrypt modules. Key Expansion cannot be fired in parallel, because the key_mem cannot change when encrypt or decrypt is running. So in a typical programming scenario, one would first initiate key expansion, then start encrypt or decrypt processes. The next section demonstrates an alternative interfacing method we tried to implement. Note that we were not able to get it to work and hence the performance metrics are of the AXI4-Lite Master standard interfacing mentioned above.

3.1 Using a proposed Arbiter design

When large amounts of data needs to be encrypted, we wondered if it will be better if the peripheral could directly interface with the memory. We need an arbiter for this process as then there will be two masters on the same memory bus, the CPU and the peripheral.

In our implementation, we added four tasks: `r_arb`, `w_arb`, `arb_controller` and `aes_wrapper`. The former two sets the peripheral as another master on the memory bus and the arb_controller determines which master is prioritised. The aes_wrapper task updates registers related to fetching data from the memory, etc., and also gives the inputs to the peripheral. Our implementation is based on the idea that once the programmer has given the inputs for the peripheral, the person need not worry after that whether the cipher process is done or not. The conflicts arising due to this is handled by the new tasks.

All the tasks have two states which is controlled by arb_en.

When arbitration is off (`arb_en` = 0), the bus interface follows normal procedure of giving access to the CPU and all the other peripherals. The inputs to the accelerator are given in this state via memory mapped address locations.

When arbitration is enabled (`arb_en` = 1), the bus interface deals with all CPU requests through 3 task handlers for read requests, 4 task handlers for write requests and one control task handler to decide which master gets to access the appropriate slave. There are three kinds of conflicts which could occur:

1. Read confilcts - When the CPU tries to read data which is yet to be generated by the accelerator.
2. Write conflicts - When the CPU tries to write into data which is yet to be read by the accelerator. There is also a conflict if the CPU writes only into few bytes of the word expecting the other bytes to hold the processed data.

The following ideal arbiter is based on our approach of handling. Real arbiter is the expected behaviour of our implementation. Handling of Read conflicts:

1. Ideal arbiter: There are two cases here. One is when the CPU needs to be stalled for only a few cycles to let the peripheral generate the values. The other is when the stalling required is too many cycles. In the former case, we stall the CPU and in the latter, we force the next input to the peripheral such that the requested data can be generated quicker. The CPU is stalled during this process. This address is also put in a read table and the peripheral skips it if encountered.
2. Real arbiter: In our implementation, the CPU is stalled and we enable registers to change the input to the peripheral. We also store this address in a read table so that it need not be read again by the peripheral. The CPU will continue to be in stall till the requested data has been pushed to memory by the peripheral. We also need to have a large table for the whole range and this limits our CPU bursts whose reason is explained later (number of times a peripheral can access the memory on its own).

Handling of Write conflicts:

1. Ideal arbiter: Suppose the CPU writes only in terms of words then if the write address conflicts with the peripheral's read address then the arbiter should force the next input to be corresponding to this write address and the CPU is stalled till the data has been read. The arbiter also needs to keep track of these out-of-order reads because in this case the data has changed.
2. Real arbiter: In our implementation, these out-of-order reads are stored in the read table again. Here suppose we use a similar approach to directly mapped cache, we may end up loosing the tag part of the address due to another out-of-order read. To prevent that we need the table to have indices for all addresses. We can reduce this if store only those corresponding to the starting address of the inputs to the accelerator (since the inputs are 16 bytes). We also have to consider that the CPU may not write a word every time. Hence we need another write table which stores the address and strb value.

3.2 Need for an arbiter

Without an arbiter the programmer needs to fetch large amounts of information from memory, store them in appropriate variables and pass them to the accelerator. Since the accelerator is not pipelined (which would cost huge amounts of hardware), the programmer needs to explicitly check if the output is ready before collecting and storing it and then giving the next input. With an arbiter, the programmer only needs to share the starting and ending address (or number of inputs) and can continue programming as if the given data has been encrypted already. The programmer can expect some slowdown if he/she tries to access the data controlled by the arbiter during arbitration.

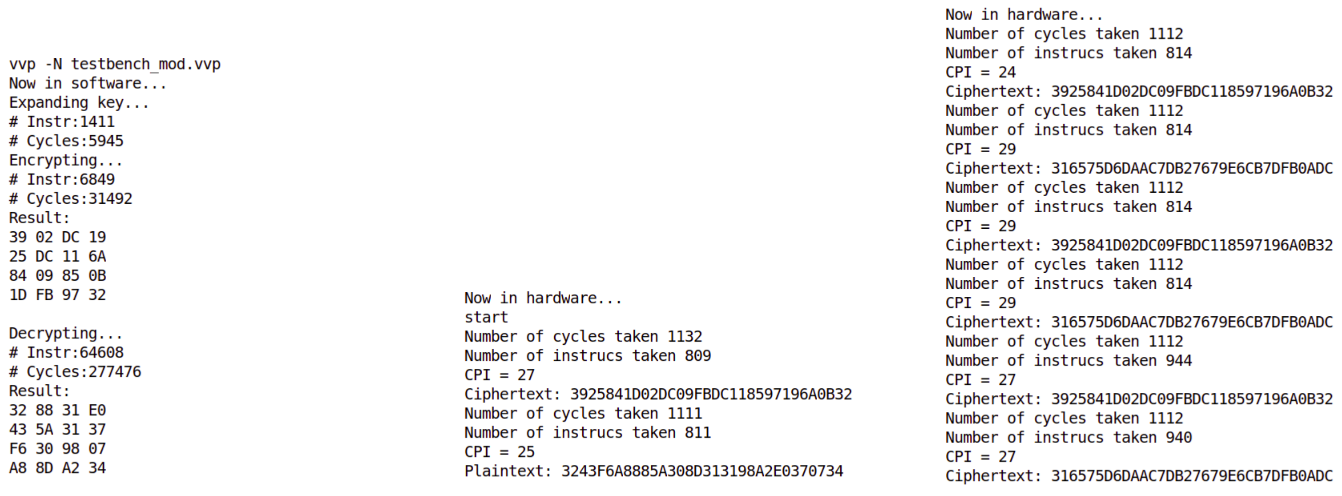
Problems:

1. Large amounts of circuitry needs to be added, this might reduce the operating frequency.
2. Scalability - It is feasible to add new peripherals by passing the input data via memory mapped addresses. Complications arise when we try to allow arbitration for other peripherals because then we need to include the case the peripherals conflict each other. The AXI standard includes AXI Interconnect which prevents conflicts by completely stalling the other masters. Ours being a single cpu with one peripheral we attempted at making ours faster.
3. We were able to ideate the design only for encryption and not decryption. We felt that handling parallelism became complicated when arbitration is designed for both.

We refered to a few sources for this, notably [5] and [6].

4 Results and Performance Statistics

A pure C (unaccelerated) implementation of AES was adapted from [4]. A problem we faced here was that `uint8_t` was not working, and unsigned char was not a suitable replacement, since char arrays utilize a function called `memcpy()` and the RV32 toolchain did not support it directly. Hence our workaround was to use a struct of unsigned char called "byte". Surprisingly, the compiler accepted arrays of this struct. Modifying [4] appropriately, we were able to get it to work. A typical log dump of one encrypt and one decrypt operation on 128 bits of input for code which is not accelerated vs. one which is accelerated is shown below:



```
vvp -N testbench_mod.vvp
Now in software...
Expanding key...
# Instr:1411
# Cycles:5945
Encrypting...
# Instr:6849
# Cycles:31492
Result:
39 02 DC 19
25 DC 11 6A
84 09 85 0B
1D FB 97 32

Decrypting...
# Instr:64608
# Cycles:277476
Result:
32 88 31 E0
43 5A 31 37
F6 30 98 07
A8 8D A2 34
```

```
Now in hardware...
start
Number of cycles taken 1132
Number of instrucs taken 809
CPI = 27
Ciphertext: 3925841D02DC09FBDC118597196A0B32
Number of cycles taken 1111
Number of instrucs taken 811
CPI = 25
Plaintext: 3243F6A8885A308D313198A2E0370734
```

```
Now in hardware...
Number of cycles taken 1112
Number of instrucs taken 814
CPI = 24
Ciphertext: 3925841D02DC09FBDC118597196A0B32
Number of cycles taken 1112
Number of instrucs taken 814
CPI = 29
Ciphertext: 316575D6DAAC7DB27679E6CB7DFB0ADC
Number of cycles taken 1112
Number of instrucs taken 814
CPI = 29
Ciphertext: 3925841D02DC09FBDC118597196A0B32
Number of cycles taken 1112
Number of instrucs taken 814
CPI = 29
Ciphertext: 316575D6DAAC7DB27679E6CB7DFB0ADC
Number of cycles taken 1112
Number of instrucs taken 944
CPI = 27
Ciphertext: 3925841D02DC09FBDC118597196A0B32
Number of cycles taken 1112
Number of instrucs taken 940
CPI = 27
Ciphertext: 316575D6DAAC7DB27679E6CB7DFB0ADC
```

Figure 7: Typical Performance Metrics of single input AES-128 in unaccelerated (left) vs. accelerated (middle) C code, and multiple input accelerated code (right)

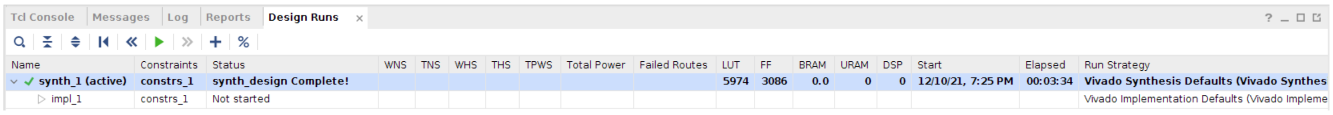
The approximate speedup for encryption is

$$\frac{31492}{1132} \approx 27.82$$

Approximate speedup for decryption is

$$\frac{277476}{1111} \approx 249.75$$

Both of these are quite significant speedups for single input workloads. We tried checking the hardware accelerated performance for 10 consecutive inputs, and changed the keys for each iteration to make the AES recompute the key expansion for worst case. We can see that the results are scaling linearly, and even the cumulative time is less than that of even single input non accelerated code. We concluded these tests were sufficient to demonstrate the speedup we got. FPGA resource utilization was studied by synthesizing using Xilinx Vivado v2020.1 and produced the following results:



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
✓ synth_1 (active)	constrs_1	synth_design Complete!								5974	3086	0.0	0	0	12/10/21, 7:25 PM	00:03:34	Vivado Synthesis Defaults (Vivado Synthes
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Impleme

Figure 8: Utilization for the entire AES implementation

5 Conclusion and Division of Work

We have obtained quite a significant speedup for several AES workloads as demonstrated in the previous section. There are several resource utilization optimisations we could have made but due to lack of time we did not explore that area, and focussed on improving the time of execution. We also tried making an arbiter for even more speedup, so AES could directly write to memory as an independent controller alongside the CPU, however due to lack of time, and the huge complexity of arbiter, we were unable to demonstrate it working. Ultimately, our goal of achieving a considerable speedup in AES has been achieved. The division of work was as follows:

1. S-Box: Saurav
2. Key Expansion: Saurav, Ruban
3. Encryption: Surya
4. Decryption: Ruban
5. Interfacing with PicoRV32: Surya, Ruban, Saurav
6. Arbiter: Surya, Saurav
7. Performance metrics measurement and comparison: Ruban, Saurav, Surya

The entire code for this project can be found here: https://git.ee2003.dev.iitm.ac.in/ee19b141/AES-Accelerator.git

References

[1] A. Pradeep, V. Mohanty, A. M. Subramaniam, and C. Rebeiro, "Revisiting AES SBox Composite Field Implementations for FPGAs", in IEEE EMBEDDED SYSTEMS LETTERS, Vol. 11, No. 3, Sept 2019

[2] Federal Information Processing Standards Publication 197, "ADVANCED ENCRYPTION STANDARD (AES)", Nov 2001

[3] P. Telagarapu, B. Biswal, V. S. Guntuku, "Design and Analysis of Multimedia Communication System", IEEE- Third International Conference on Advanced Computing, Dec 2011

[4] https://github.com/kokke/tiny-AES-c/blob/master/aes.c

[5] https://www.allaboutcircuits.com/technical-articles/simple-priority-arbiter-allocating-resources-embedded-systems-vhdl-logism/

[6] https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[7] Ako Muhamad Abdullah, "Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data", Eastern Mediterranean University - Cyprus