[[1]](#footnote-1)

Low Area FPGA Implementation of the AES Standard

William Chandler, Elsie Baderinwa, Luke Locastro and Thomas Brown

*Abstract*—This report discusses the design, implementation and performance of a low area FPGA-based implementation of the AES standard, incorporating both encryption and decryption. Pipelining is used to improve performance. The resultant system gives reasonable performance with a non-dynamic key and performs equally well for encryption and decryption.

*Index Terms*—AES, Low-Area, FPGA

# INTRODUCTION

T

HIS report discusses the design decisions, design process and resultant performance of a low area FPGA implementation of the AES standard. It incorporates both encryption and description and attempts to share resources where possible. Pipelining is used to improve the maximum clock frequency achievable by breaking up long sections of combinatorial logic. Previous work LIT REVIEW

# Brief Description of AES

AES is a symmetric block cipher used for encrypting data. Symmetry in encryption means that the same key is used to encrypt and decrypt. It consists of a number of steps executed in a loop with keys being generated for each round.

DIAGRAM

As FIGURE SOMETHING shows, the main steps are SubBytes, ShiftRows, MixColumns and an XOR with the round key.

SubBytes is a non-linear substitution step. The replacement of the bytes implemented using a look up table called the s-box which consists of two steps- Multiplicative inversion and affine transformation.

Implementation of the Shift Rows function occurs in every round of an AES encryption process, after Sub Bytes and before Mix Columns. Its function is to systematically alter the order of data in a 16-byte packet, depending on the location of the data when in it is input. If the data packet is displayed in a 4x4 grid as shown in Figure 1, the output data undergoes a left barrel shift of magnitude relevant to the row it appears in the grid. The first column of input data is highlighted to outline this shift. For example, data in row 0 (the top row) is shifted left by a magnitude of 0; therefore the data output in the same order it was input. In row 1 the data undergoes a single left shift, with the data in the first square coming around and appearing in the furthest right square. Rows 2 and 3 are right shifted by a magnitude of 2 and 3 respectively.

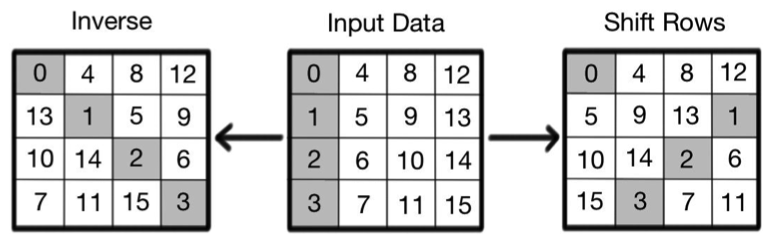


Figure 1 Forward and inverse Shift Rows

The inverse function of Shift Rows can also be seen in Figure 1. It works in exactly the same way as the forward function however the left shifts all become right shifts, so that if the inverse function proceeds after the forward, the output data would be the same as the input.

MixColumns…..

Keys are generated for each round using a key generator which consists of various steps including a rotation, SubBytes substitution and XOR with an RCon value.

# System Architecture

## Top level design decisions

Data in AES is processed in ‘states’. Each state consists of 16 bytes, usually arranged in 4 columns. Some larger implementations of AES process the whole state concurrently which is very fast but also requires a large area on the FPGA. In order to maintain a low-area it was decided that our design should operate on one byte at a time. This ensures the design will be as small as possible by eliminating the need for duplicate modules.

In a similar fashion, large area designs can ‘unroll’ the loop which means the 10 looped operations are formed by duplicating all the modules 10 times. This allows the designs to be very fast by allowing 10 times as many states to be processed in parallel. For the low area application this is not suitable and the loop is not unrolled. In order to achieve this, a multiplexer is added to the start of the loop which can switch between input data and the output of the loop. Data is only accepted into the loop after 9 full loop iterations whilst the final data is being output.

This low area 8-bit rolled datapath requires more control than the high throughput 128-bit unrolled datapath which means there is a penalty to the throughput per slice. This is a trade off between area and efficiency.

Each of the blocks in the datapath, the key generator and the controller were designed specifically to be low area whilst also striving for high efficiency.

Implementation of the inverse operation in the datapath seems rather inefficient. It consists of a number of multiplexers which change the arrangement of the processes to an almost reversed configuration. ShiftRows and SubBytes do not need to be reversed as ShiftRows is not dependent on byte values and SubBytes only operates on byte values. The delay after MixColumns is also left in that position as it has no effect on the output result. There are other methods for implementing the inverse which were not explored in this project due to time constraints.

## SubBytes

### Composite field arithmetic

In this design, composite field arithmetic is used to reduce hardware complexity. This technique is frequently used in cases where lookup tables are required for arithmetic operations by using smaller tables through subfield arithmetic. This is possible because while Galois field of the same order are isomorphic, the way in which the field elements are represented, have an effect on the complexity of the operations to be carried out on said fields i.e. GF(2x)y) is isomorphic to GF(2z) for z= xy. Therefore, instead of operations being performed on composite field GF (28), GF (2) is utilised iteratively to obtain GF (((22)2)2). To achieve this isomorphic mapping an 8x8 matrix (delta matrix, δ) is multiplied with the input to map the elements in GF (28) to its composite fields. It should also be noted that the application of the inverse delta matrix to a composite field element maps it back to the GF (28) domain.

To simply the system, it was necessary to account for this mapping in the design. This changed the suubbyte implementation from a two-step process. This new process is shown by the block diagram below.

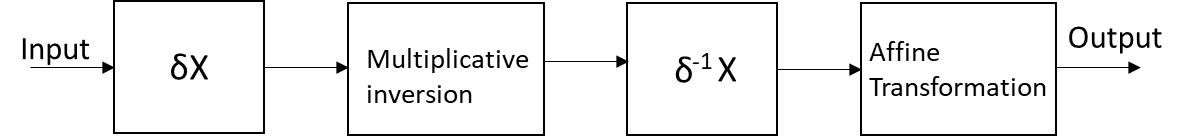


Figure 2 Block diagram showing forward SubBytes operation

### Multiplicative inversion

Although this stage was greatly improved by the use of composite field arithmetic (GF (28) would require 620 gates, but the use of composite field arithmetic reduced it down to 159 gates). The multiplicative inversion block was extremely modular and required bottom up implementation. The smallest blocks were built and tested first as they were required in making the larger ones. It was also necessary to write the test benches by hand and the results were compared with the simulated outputs. The higher up the hierarchy, the more complicated it was to write the test benches. There were no known values to test simulations against until the multiplicative inversion block was completed, so it was important to have the test benches as a reference. While there was data available on the multiplicative inversion block, the values were different to those oh our block due to the use of composite field arithmetic.

### Affine transformation

The implementation of this block was quite straightforward as, with the composite field transformation, it required a matrix multiplication, which was implemented using 40 XOR gates.

### Inverse subbytes

The implementation of the subBytes block for decryption is similar to that for the encryption process as they share a number of blocks. The main difference is that the inverse affine transformation is performed at the beginning of the process.

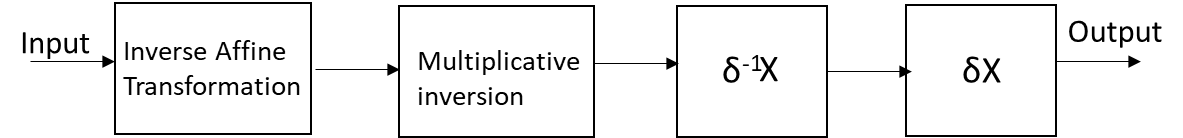


Figure 3 Block diagram showing inverse subBytes operation

As the forward ad inverse subBytes operations have some blocks in common, it is more efficient to implement them as a singular unit. Multiplexers were used to select whether the forward or inverse operation was required and flip flops were inserted into the design to add 3 cycles of delay. This was done to contribute to the 32 cycles of delay that was required by our design. The block diagram reflecting these design decisions is shown below.

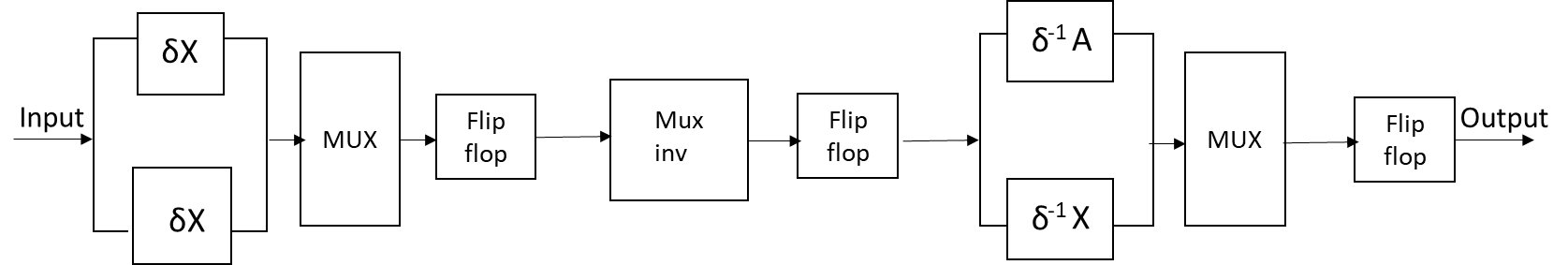


Figure 4 Block diagram showing the full subBytes block

With this design, the final subBytes block used a total of \_\_\_\_\_\_ slices blah balh

## Shift Rows

### Hardware Design

Due to the shifting nature of serial input data in this function, hardware implementation would always cause some initial latency in the output. Looking at the output data in Figure 1, the forward function requires bit 15 to be output 4th and on the inverse it requires bit 13 to be output 2nd. For this data to be output, the system will first need to wait until it input. Therefore the minimum amount of latency possible would be 12 clock cycles, as this would line up both bit 15 on the input and output of the forward and bit 13 on the input and output of the inverse. In hardware the most convenient way of doing this would be to shift input data though a register so that it can be stored until it is required at the output. This way, once the register has been filled up after the 12 clock cycles of latency the system can deal with continuous input data.

### SRLC16E

SRLC16E blocks were used as the registers in the system as they take up very little space, which is desirable since the design is supposed to be for low area. They also allow initial conditions to be set for a barrel shift as well as having an addressable output, meaning they can be used for the registers that hold the data flow, but also they can be used to hold the address data for the data registers.

### Complete System

The system design for Shift Rows can be seen in Figure 5, complete with the data path and the addressing signals. Here it is possible to see how the main data path can be controlled using shift registers as temporary addressable storage.

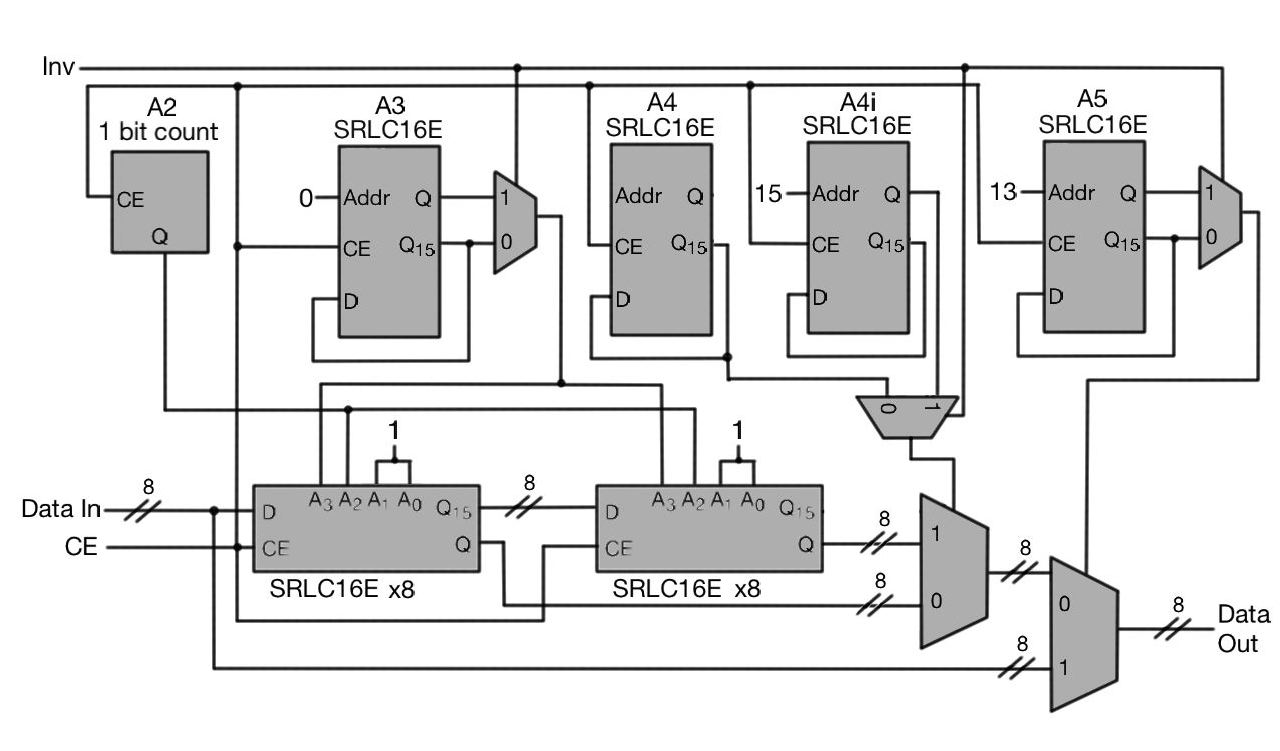


Figure 5 Full hardware design of Shift Rows

In the design SRLC16E are also used as barrel shifters to store the address sequence that is required for the desired data output.

## Mix Columns

The mix columns operation is implemented by multiplying the state as a matrix with the matrix shown in Figure 6within GF(28), this is described in [1].

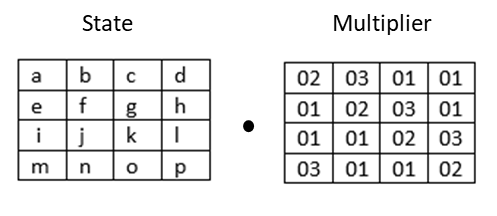


Figure 6 matrix multiplication

As 8 bit operation had been decided on for the system the mix column implementation had to operate at 8 bit. Because of this and the focus on low area mix column was implemented as shown in Figure 7. The implementation is based on the design first introduced in [2], data is input one byte at a time and the output has a 4 cycle delay.

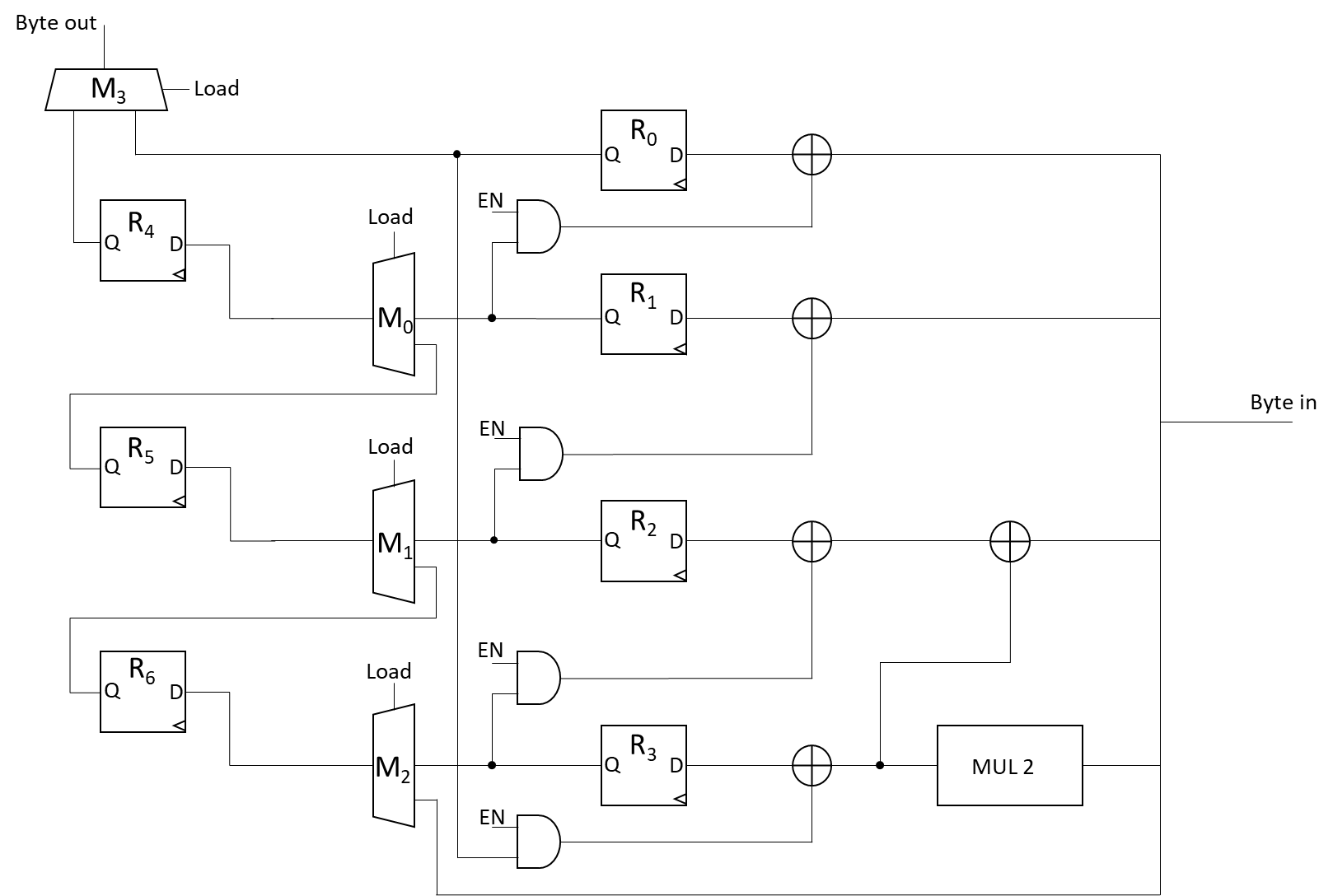


Figure 7 Mix columns implementation

The matrix multiplication requires multiplication by 2 and 3, multiplying by 2 over GF(28) takes 3 XOR gates as shown in [3], multiply by 3 reuses those gates with one extra XOR to add the input byte. Registers 4, 5, 6 and the multiplexers act as a parallel to serial converter, this is needed as mix columns is performed on a column of the state so is inherently 32bit. The converter allows the output to be 1 byte.

### Inverse Operation

The inverse operation (decryption) requires another matrix multiplication, in this design it is achieved using the pre-processing step shown in [1], this means simpler multipliers (see Figure 8) are needed reducing the hardware and processing required.

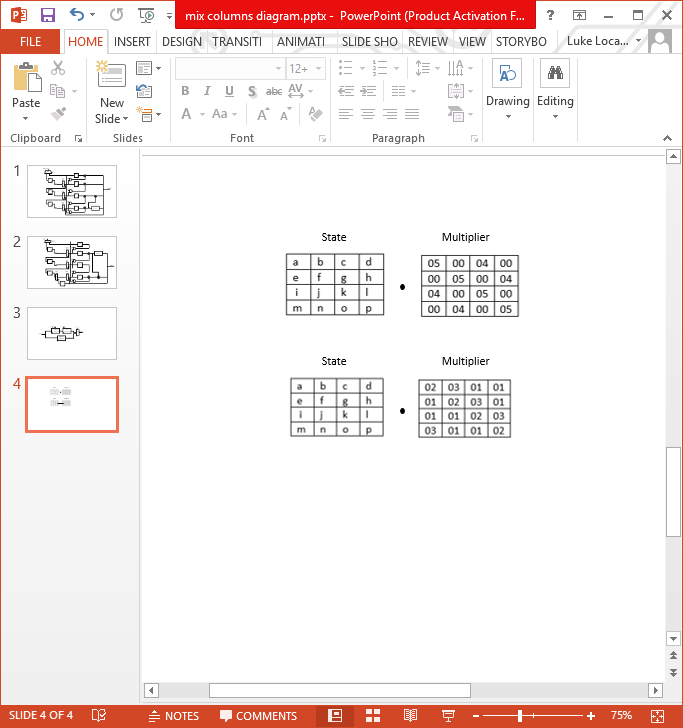


Figure 8 inverse pre-multiplier

Similarly to the multiply by 2 and 3 above, the multiply by 4 is implemented as in [3] using 6 XOR gates and multiply by 5 uses one more XOR. The design implementation is the same as the encryption block except with an added multiplexer and register (see Figure 9).

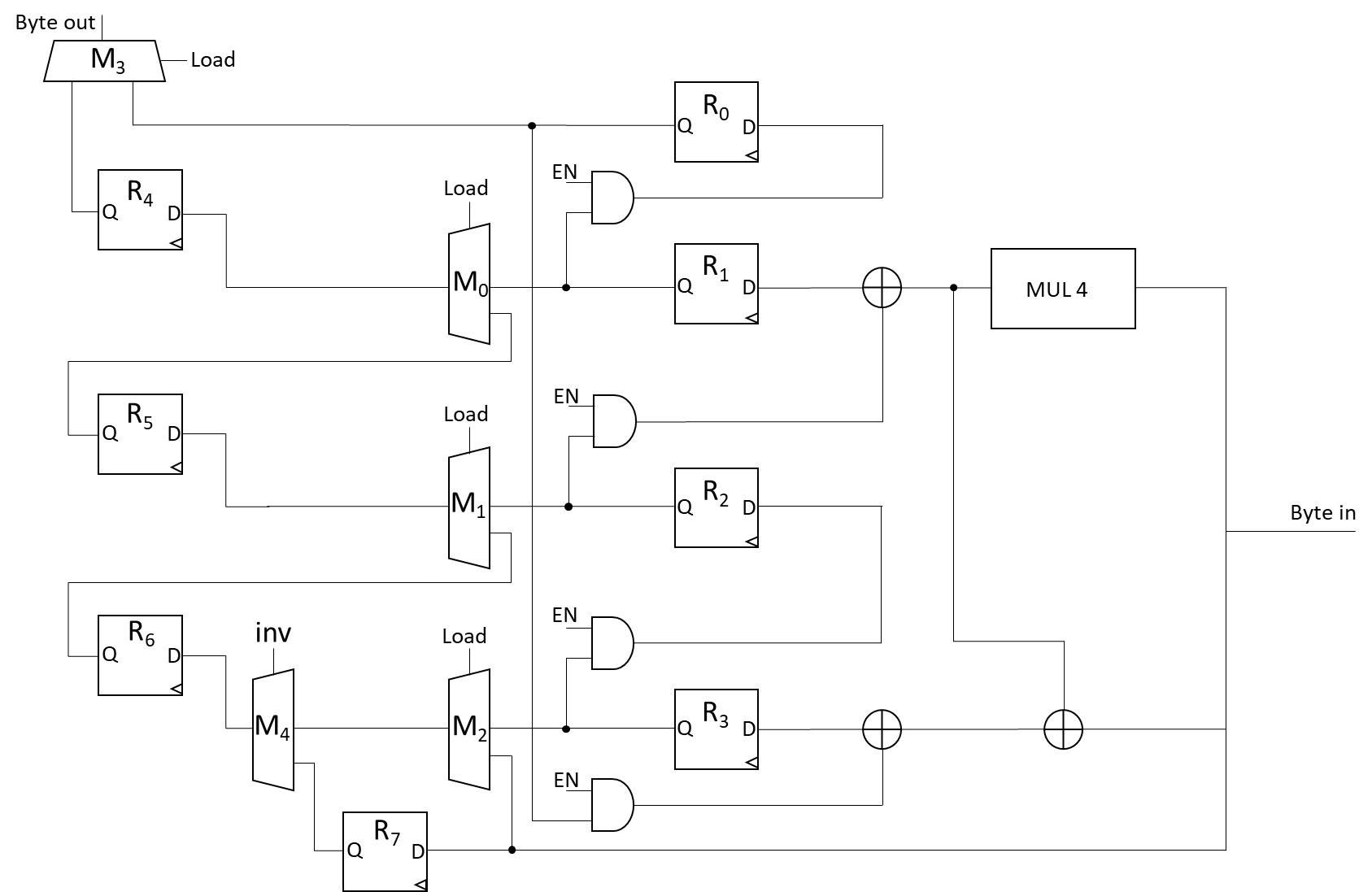


Figure 9 Inverse pre-processing block

The top level diagram (Figure 10) shows that the data path when performing mix columns requires data to pass through the inverse multiplier, when performing encryption the data must be passed through this block without multiplying. This has been implemented with the added multiplexer and register which passes the data through maintaining the 4 cycle delay. This gives the inverse operation a total delay of 8 cycles, on the 10th round the mix columns operation is avoided this is implemented using SRL16’s acting as a bypass register maintaining the 8 cycle delay.

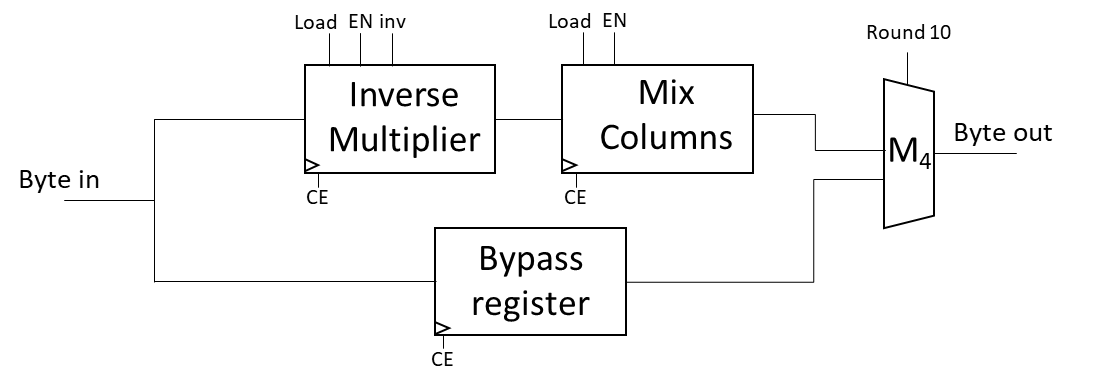


Figure 10 Mix columns top level

### Control block

Control signals load and EN are generated with a barrel shift register, this s implemented using a SRL16 to minimize slice usage. As the EN control signal is the inverse of load only one SRL16 is needed, these signals are shared by both the forward and inverse operation blocks. This means the control block takes up one slice.

Using this implementation mix columns is implemented in 91 slices, having functionality for both encryption and decryption.

## Key Generator

Keys are given in the same format as the data, in ‘states’ of 16 bytes. The key generation process consists of an XOR of the N-4 byte with the N-16 byte. Additionally, the first column is rotated, the bytes are substituted and it is XORed with an RCon value. This process is shown in FIGURE. The inverse process requires the final key to be generated by running the forward process 10 times then to generate successive previous keys, the N+4 and N+16 bytes are used instead to generate the previous key.

Due to the low area constraint, and the rolled datapath, storing the keys after generation was not considered an effective solution. This meant that the keys would be generated on-the-fly when they are required by the datapath.

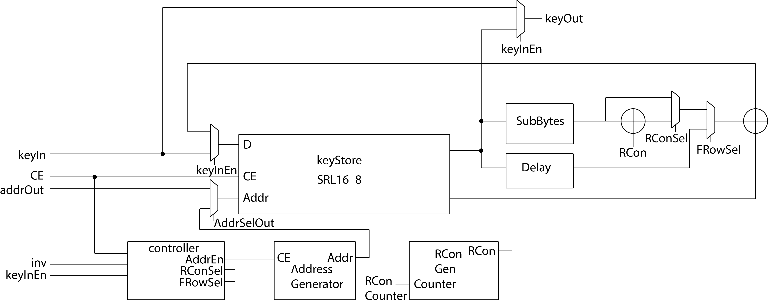
Initial work focused on the area efficiency of the SRL16 shift registers which allow the storage of 16 bits of information in only one LUT. They provide with two outputs, one addressable and one which always points to the final value. These were used to form an SRL16\_8 register which stores 16 bytes of data. 

Figure 11 Key generator core

The SRL16\_8 register was used to form the main key store with the input data being switched by a multiplexer between new key inputs and the last generated key. The rotation of the first column was achieved by changing the address given to the register.

The two outputs of the register allow both N-4 and N-16 bytes to be selected concurrently.

A separate SubBytes module is instantiated here as the one in the datapath is fully utilized. The module used here does not have the inverse logic as it is not required. It has 3 cycles delay caused by internal pipelining which means that the addressing starts three cycles before the output is expected. It also necessitates the addition of a delay block (formed by an SRL16\_8), so that when the first column is complete, further output appears at the final XOR at the appropriate cycle.

The RCon value is zero for all but the first byte, so due to it only being used in an XOR operation where zeros produce no result, only the first value is important. This value is generated using a simple lookup based on a counter passed from the main controller.

Selecting between the first byte output (RCon XOR output), first row output (SubBytes output) and all other bytes is achieved using multiplexers.

The inverse operation can is achieved by feeding in the final key backwards, the control for which is in the main controller.

In order to manage external inputs as well as managing the inverse key reversal, a top controller module was created. This consisted of an external keystore (SRL16\_8) which stores the first or final key for forward and inverse operations respectively. Some control signals are also changed for the inverse.

Due to some lack of foresight and some changes to the top-level design, significant changes were required. First the datapath was changed from a 16-register cycle to a 32-register cycle which holds two full states before reaching the next cycle. This means that each key must be outputted twice, once for each state. Second, for the inverse operation, the key must still be passed to the datapath forwards. Which means the key needs to be inverted before being output. Third, as the state in the datapath is being output, a new state is being fed in to avoid any bubbles in the pipeline. This means the first key (or final in the inverse case) must be passed to the first XOR in the datapath whist the final key is being passed to the second XOR in the datapath. These changes added significant complexity to the design.

This complexity is mostly handled in the top controller module. An additional SRL16\_8 buffer is used to store the key that is outputted from the key generator. The output to the second XOR in the datapath is then multiplexed between the output of the key generator and the output of the buffer which has the effect of repeating the key. In order to output the inverse key bytes in the forward order, the key is generated whilst the key is being sent from the buffer, a counter is then used to address the key store inside the key generator to select the bytes. The external key store now stores the second key, which is fed back in to the key generator every 10 rounds. A second key store is used to store the first key, which is constantly output to the first XOR in the datapath. Assuming the key is not dynamic, these stores allow the pipeline to avoid bubbles, as well as allowing the inverse operation to be as fast (after setup) as the forward operation. All this extra complexity required a lot of control signals to be generated, so counters were added to keep track of the round as well as the byte within the round. These counters were then used to create control signals.

Much of this complexity is avoidable by using a three or four port RAM.

## Controller

Fortunately, the controller for the datapath did not require much more logic area because of the counters already used by the key generator controller. *keyInReady* and *dataInReady* signals are used to indicate to the serial wrapper that a key or data block can be input to the system. When either is being input, the corresponding *keyInEn* or *dataInEn* signals are set high by the wrapper. When the inverse input signal is set high, the *dataInReady* signal will go low until a key has been input and the final key has been generated. Once dataInEn is set high, signals are generated to start the components at the correct time. The system will stop and wait if no data is given when dataInEn is high. This simplifies the control but does mean that the final state will not be output until new data is given.

# Performance

# Conclusion

The performance of this AES implementation is not as good as some other more developed solutions. It is still functional and gives good results considering the late changes that had to be made to support the inverse operation. Due to the experience gained during this project, future designs will be more thoroughly planned out before starting the design phase. This should prevent unexpected complexities and increase in area.

# References

|  |  |
| --- | --- |
| [1] | J. Daernen and V. Rijrnen, "8-Bit Platfonns," in *The Design of Rijndael: AES - The Advanced Encryption Standard*, Berlin, Springer, 2002, pp. 55 - 56. |
| [2] | P. Hamalainen, T. Alho, M. Hannikainen and T. D. Hamalainen, "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core," in *9th EUROMICRO Conference on Digital System Design*, Dubrovnik, 2006. |
| [3] | X. Zhang and K. K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 12, no. 9, pp. 957-967, 2004. |

1. [↑](#footnote-ref-1)