Low area implementation of AES ECB on FPGA

***Abstract*—This project aimed to create a low area implementation of the Rajindael cipher (AES) on FPGA. The design needed to support encryption and decryption and should not use external block RAMs. In the end we demonstrated an implementation capable of encryption/decryption in 326 and 482 clocks respectively, using only 456 slices, which was implemented on a Spartan 3E series FPGA.**

***Index Terms*— AES, FPGA, Low Area, Spartan 3**

I. Introduction

The Advanced Encryption Standard (AES) has been the defacto block cipher for many secure applications since its selection in 2000 and its incorporation into FIPS 197 [1]. Numerous hardware implementations have been studied to cover the many use cases, from high throughput server chips to low area, low power mobile chips.

II. Implementation of Low Area design.

A. System Overview

Our system is a 10 round Electronic Code Book (ECB) implementation of the AES algorithm. For the project an 8-bit path with minimised round-loop architecture was used, as it supports both encryption and decryption. The design processes 2 blocks at a time to improve throughput, especially during decryption, where initial key processing produces significant delay.
The main system diagram is shown in Figure 1.

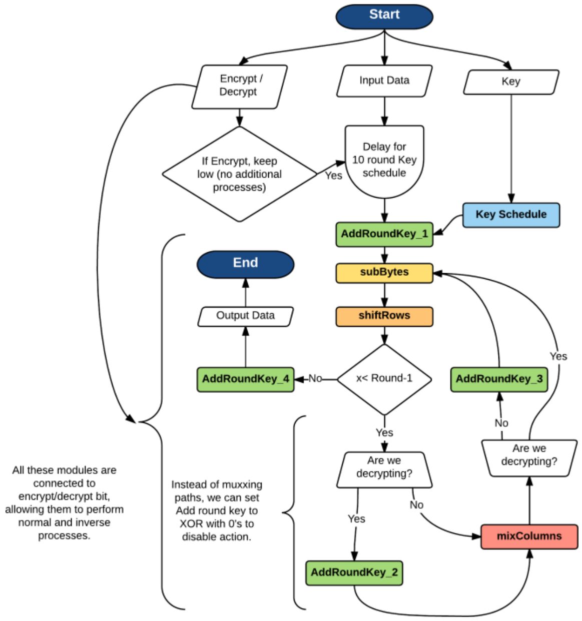
Start
Encrypt / Decrypt
Input Data
Key
If Encrypt, keep low (no additional processes)
Yes
Delay for 10 round Key schedule
Key Schedule
AddRoundKey_1
End
subBytes
Output Data
shiftRows
AddRoundKey_4
No
x< Round-1
AddRoundKey_3
Yes
No
Yes
Are we decrypting?
Are we decrypting?
All these modules are connected to encrypt/decrypt bit, allowing them to perform normal and inverse processes.
Instead of muxxing paths, we can set Add round key to XOR with 0's to disable action.
Yes
No
mixColumns
AddRoundKey_2

Figure 1. Main Architecture, supporting encryption and decryption.

The structure of the loop allows for encryption and decryption with simple path mux changes and the setting of a global “decryption” signal, telling sub components to perform their inverse operations.
In such a loop, it is advantageous to hold an amount of bytes equal to a multiple of 16 (the AES block size). The total delay through each component of the round loop was 20 clocks, so it was decided an additional 12 clock shift register be added, so that 2 blocks (32bytes) would be processed in each loop.

SubBytes	ShiftRows	AddRoundkey_2	MixCollumns	AddRoundkey_3	**Total**
1	13	1	4	1	**20**

Table1. Delays between byte entering component and data exiting component

The system operates as a simple state machine with a counter used to provide timing for various re-sychronisation and scheduling required in the system. The system begins in an IDLE_1 state, where it is waiting for the data key to be presented. When a key is loaded, the encryption or decryption mode of the chip is then fixed for the remainder of the processs. If decryption is chosen, the device begins to process the key to generate the 10th key and will not accept data until it has done so. If encryption mode is chosen, or after the 10th key has been generated for decryption, the system enters the IDLE_2 state, where it will wait for up to 2 blocks of data for processing.
When data is ready on the output, “Output_Data_Ready” is held high and this occurs 308 clocks into the “PROCESSING” state.

The top system also handles start/stop of the key scheduler, along with key repetition; a requirement given the dual block processing of the system.

It is worth noting that the order ShiftRows and SubBytes components are interchangeable and that in the final iteration of our system they were re-ordered to prevent adding excess registers.

B. SubBytes

The implementation of SubBytes is achieved by moving away from the use of a look-up table (LUT). LUT approaches require substantial memory allocation or a large area on the FPGA. As well as size, LUT’s have an unbreakable delay associated with them, limiting system throughput. To implement a design to avoid this, a composite field data path is used for both SubBytes and InvSubBytes. A sub pipelined architecture can now be implemented to increase throughput, although not of paramount importance in a low area application.

Composite fields can be used to decompose high order Galois Field multiplications to lower order fields with irreducible polynomials such that the field $GF(2^k)$ can be represented as $GF((2^n)^m)$ where $k = n.m$. This often reduces system complexity through simpler representation of field components. With lower order fields it is essential isomorphic and inverse isomorphic mapping functions are used to map an element from a field to a chosen composite field and back, respectively. The only part of the AES algorithm that can be efficiently implemented in composite field is SubBytes and InvSubBytes. Every other transformation should be kept in $GF(2^8)$.

SubBytes is a non-linear transformation used to compute the multiplicative inversion followed by an affine transformation of each byte in a block in $GF(2^8)$. Implementation of this multiplicative inversion in $GF(2^8)$ requires roughly 620 gates with large delays [2]. Reduction of this gate count is implemented through composite fields and reduction of in $GF(2^8)$ into in $GF((2^4)^2)$.

1) *Isomorphic/Affine*

The isomorphic mapping used for decomposition is presented in Fig.2 along with the inverse mapping that has combined with the affine transformation for reduction of gates. Where x represents a single byte:

$$\delta.x = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} . \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix}$$

$$\delta^{-1}A.x = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Fig.2: (a) Isomorphic (b) Inverse Isomorphic combined with Affine

2) *Multiplicative Inversion*

According to Eucledian's Extended algorithm the multiplicative inversion can be represented as shown in Fig.3. This is carrying out the inversion in $GF((2^4)^2)$. Each of the individual blocks can be decomposed into simple logic gates as discussed in [3], representing the operation. It can be noted that multiplication in $GF((2^2)^2)$ has to be further decomposed so that multiplication in $GF(2)$ is represented by a simple AND gate.

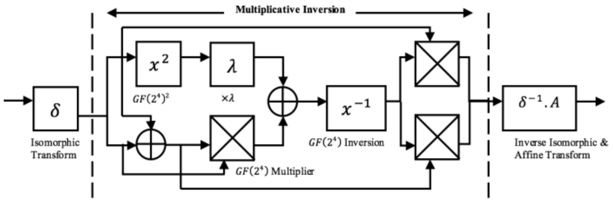

Fig.3 -SubBytes $GF((2^4)^2)$ Multiplicative Inversion

Inversion in $GF(2^4)$ can be done via multiple different methods as discussed in [3]. The most efficient method is using a direct logic derivation which can be represented by 14 XOR gates and 9 AND gates. Each inversion bit can be calculated individually as shown.

$$\begin{aligned} x_3^{-1} &= x_3 + x_3x_2x_1 + x_3x_0 + x_2 \\ x_2^{-1} &= x_3x_2x_1 + x_3x_2x_0 + x_3x_0 + x_2 + x_2x_1 \\ x_1^{-1} &= x_3 + x_3x_2x_1 + x_3x_1x_0 + x_2 + x_2x_0 + x_1 \\ x_0^{-1} &= x_3x_2x_1 + x_3x_2x_0 + x_3x_1 + x_3x_1x_0 + x_3x_0 \\ &\quad +x_2 + x_2x_1 + x_2x_1x_0 + x_1 + x_0. \end{aligned}$$

C. *InvSubBytes*

Inverse SubBytes is similar to the forward SubBytes in that it still uses the multiplicative inversion section. The only difference is the mapping functions applied before and after the inversion.

$$A^{-1}\delta.x = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} . \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \delta^{-1}.x$$

$$= \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix}$$

(a) (b)

Fig.4: (a) Isomorphic combined with Inverse Affine (b) Inverse Isomorphic

To select between SubBytes and its inverse in a system with both encryption and decryption it is possible to use 2 to 1 multiplexers to select between the different mapping and use the same inversion unit.

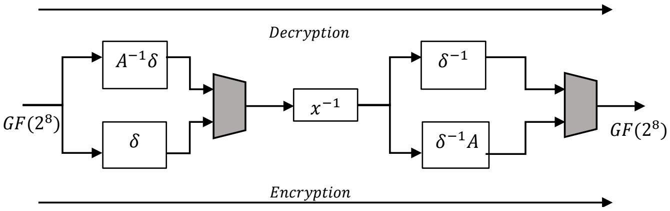

Fig.5: SubBytes Encryption/Decryption Block

D. *ShiftRows*

ShiftRows initially looks difficult in a byte path architecture, as block bytes 0 and 15 are needed within 4 clocks of each other. Chu and Benaissa [4] show a method of implementing ShiftRows using 2 byte-wide SLRs with a 12 clock delay and parallel addressing to produce the desired output.

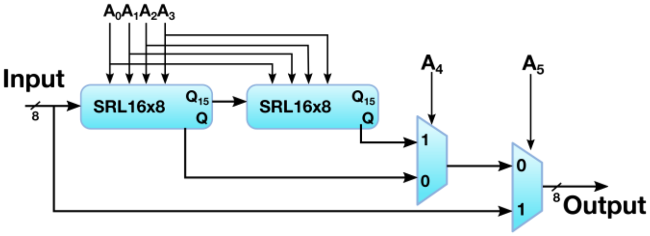

Fig 6: Diagram of ShiftRows implementation

The address pattern required was verified using an Excel sheet. Due to the size of the table and its interactive nature, this document has not been included in this paper, but has been made available online[1].

It was noticed that introducing an extra clock of delay reduced the complexity of the addressing signals significantly, reducing the addressing to only 3 varying bits, with A_0, A_1 and A_5 = '0' at all times. As A_5 was held low, we were also able to remove the second mux, reducing area.

Using the Excel document it was also possible to determine the pattern needed to implement an Inverse shift rows function. It turns out that this also only requires 3bits of variation and, in fact, shares a common definition (A_2) and a time shifted one (A_3).

Encryption	**A_4**	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	1
	A_3	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	A_2	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Decryption	**A_4**	0	0	0	0	0	1	0	0	0	1	1	0	0	1	1	1
	A_3	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
	A_2	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Table 2. Address pattern for encryption and decryption.

The component requires a re-synchronisation signal at the beginning of data input, so it can set its addressing to initial state from which to proceed.

E. *MixColumns*

The MixColumns transformation for encryption works in a byte systolic manner. It treats each column separately every 4 clock cycles as each byte is fed into the system. The MixColumn multiplier is shown in table 3. On every clock cycle a new byte is fed into the system and stored in the 4 registers. These registers store the intermediate results for the MixColumn transformation as shown in figure 7. The multipliers on the input to the registers represent the Galois multiplication by $\{02\}_H$ and $\{03\}_H$ that occur in the generalised matrix multiplication for MixColumns. The same multipliers are used for each row of a column just in a cyclically shifted order. The intermediate results are an addition of the new byte and a cyclic shift of previous bytes. The enable *(en)* signal is used to enable cyclic shift (*en = logic 1)* or mask the registers to 0 before the first byte of a column (*en = logic 0*). After 4 clock cycles the output is transferred to a byte wide parallel-to-serial converter using a *load* signal. All control of the transformation is implemented as local logic. A full block MixColumn transformation is completed every 16 clock cycles with natural 4 stage pipelining.

The inverse MixColumn transformation for decryption has the same hardware layout, the only difference being the different cyclically shifted Galois field multiplication of the inverse MixColumn matrix. The registers inputs have the multipliers $\{09\}_H$, $\{13\}_H$, $\{11\}_H$ and $\{14\}_H$.

REGISTER	**CLOCK PULSE**			
	t=0	**t=1**	**t=2**	**t=3**
$\boldsymbol{R_0}$	x_0	$x_0 \oplus x_1$	$\{03\}x_0 \oplus x_1 \oplus x_2$	$\{02\}x_0 \oplus \{03\}x_1 \oplus x_2 \oplus x_3$
$\boldsymbol{R_1}$	x_0	$\{03\}x_0 \oplus x_1$	$\{02\}x_0 \oplus \{03\}x_1 \oplus x_2$	$x_0 \oplus \{02\}x_1 \oplus \{03\}x_2 \oplus x_3$
$\boldsymbol{R_2}$	$\{03\}x_0$	$\{02\}x_0 \oplus \{03\}x_1$	$x_0 \oplus \{02\}x_1 \oplus \{03\}x_2$	$x_0 \oplus x_1 \oplus \{02\}x_2 \oplus \{03\}x_3$
$\boldsymbol{R_3}$	$\{02\}x_0$	$x_0 \oplus \{02\}x_1$	$x_0 \oplus x_1 \oplus \{02\}x_2$	$\{03\}x_0 \oplus x_1 \oplus x_2 \oplus \{02\}x_3$

Table 3: MixColumn encryption register 4 clock contents

[1] http://tinyurl.com/shiftRows-addr

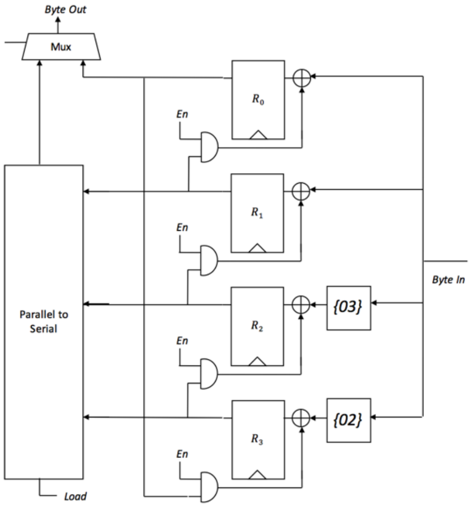

Figure 7: Mix Columns Transformation

F. Key Scheduler

The key schedule is used to manipulate the cipher key in order for it to be used for the *AddRoundKey* function. The current key, k_i is used to calculate the following key, k_{i+1}, by XOR'ing with the following column, with the exception of column 0, k_0. Column 0 of the next key is calculated by taking column 3, cyclically shifting it to obtain the 'rotword' (RW), performing the Rijndael s-box substitution (SB) on the rotword, and then XOR'ing with the current key column 0 and the round constant (RCON). Represented mathematically:

$$k_{0,i+1} = k_{0,i} \oplus SB\left\{RW\{k_{3,i}\}\right\} \oplus RCON$$

$$k_{1,i+1} = k_{0,i} \oplus k_{1,i}$$

$$k_{2,i+1} = k_{1,i} \oplus k_{2,i}$$

$$k_{3,i+1} = k_{2,i} \oplus k_{3,i}$$

The round constant is calculated by beginning with 1 and multiplying by 2 within the finite Galois Field of GF(2^8) each round, using the polynomial:

$$p(x) = x^8 + x^4 + x^3 + x + 1$$

Only the first byte of the shifted column 3 is XOR'ed with the round constant, remaining bytes are XOR'ed with 0, i.e. unchanged.

The key expansion can either be performed by computing all keys in advance and storing them, or by computing them when required. As this implementation is focussed on low area, the computing when required method is used, as storing keys takes up a significant amount of area [5]. An SRL16 was used due to its efficiency with regards to size [6] [7]. During encryption, a byte of the key schedule is output every clock cycle, with the ability to pause the key scheduler for synchronisation with the rest of the AES system. The original design for encryption only is shown below in figure 8.

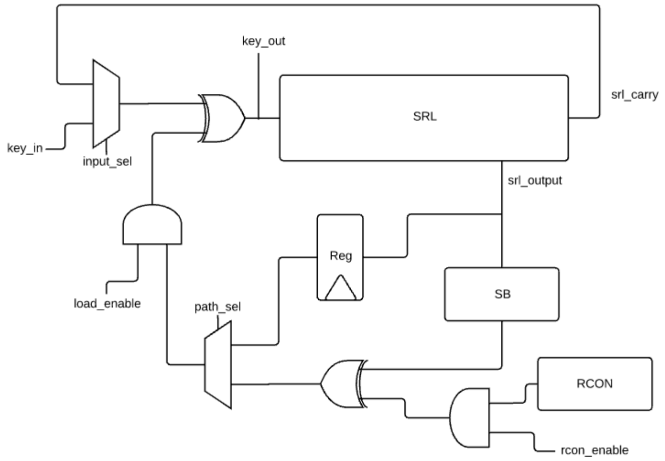

Figure 8 – KeySchedule Encrypt

Initially, the 'load enable' is held low to avoid the initial key being XOR'ed with any data that remains in the SRL16 output path from previous encryptions or decryptions and the 'input mux' selects the external key in. After the first key has been fully loaded into the SRL16, the SRL output is manipulated in order to perform the cyclic shift. The first column of the next key is created by feeding the SRL output through the s-box and XOR'ing with the round constant whenever it is enabled. This is then XOR'ed with the SRL carry to form column 0 of the next key. The remaining columns are formed by having the path mux select from the register output, which are then XOR'ed with the SRL carry.

Decrypting raises issues due to the complexity of calculating inverse keys when required, this is considered a weakness of the AES system [4]. This is because the keys are required to be output in the correct order, i.e. column 0 to column 3, however to generate column 0 of the previous key, column 3 of the previous key is desired. Column 0 is required to calculate column 1 and column 3 and thus needs to be stored for later use. Due to this, the key schedule can no longer output one byte every clock cycle, it now outputs the 16 bytes of the key every other 16 clock cycles. An additional mux is required as the output may come from two parts of the circuit. The modified design for decryption is shown below in figure 9.

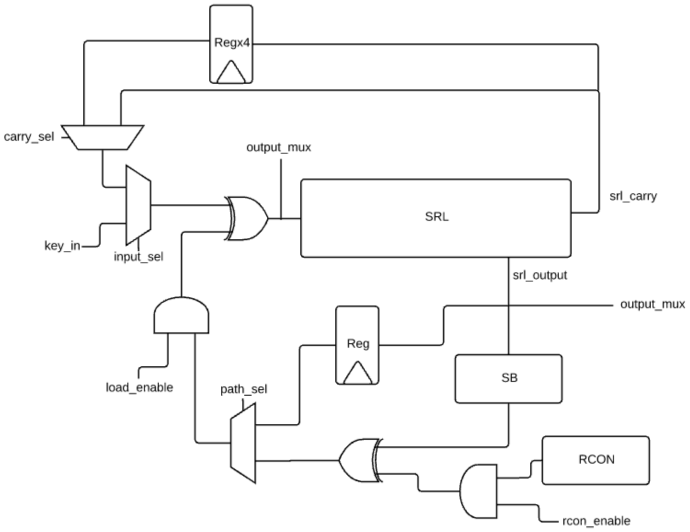

Figure 10 – Key Schedule Encrypt

At the end of the encryption, the SRL holds:

$k_{10,3}$	$k_{10,2}$	$k_{10,1}$	$k_{10,0}$

Where $k_{10,0}$ is key 10, column 0, etc. The SRL output points to $k_{10,1}$ and for 12 clock cycles the carry is XOR'ed with the SRL output to produce keys $k_{9,1}$, $k_{9,2}$ and $k_{9,3}$. The 4 registers on the SRL carry line are disabled once $k_{10,0}$ is clocked in to store them for future use. The SRL then holds:

$K_{9,3}$	$K_{9,2}$	$K_{9,1}$	$k_{10,3}$

The carry mux then selects $k_{10,1}$ from the bank of registers which is XOR'ed with $k_{9,4}$ which is fed through the SRL output through the s-box to generate $k_{9,1}$. Therefore, at the end of 16 cycles the SRL contains:

$k_{9,0}$	$k_{9,3}$	$k_{9,2}$	$k_{9,1}$

The next 4 clock cycles use the SRL output to send $k_{9,0}$ to key_out, selected using the output mux whilst the SRL is disabled. Afterwards, for 12 clock cycles the load enable is held low, the SRL is enabled, the output mux selects the output from the SRL input and the SRL carry shifts out $k_{9,1}$ to $k_{9,3}$. This leaves the key in the SRL in the same order it was when decrypt began, meaning the logic can be re-used for future decrypt rounds.

III. Design Results

A. Implementation results

After synthesis in Xlinx 14.7 the design is listed at 395 slices, but after place and routing for a Spartan 3E, our design becomes 456 slices. The critical path in the system exists in SubBytes with a timing of 13.714ns, thus a clock period of 14ns was chosen. Pipelining of SubBytes to reduce this critical path was considered, but this would have further complicated our KeySchedule design. Furthermore, the next critical path was 12.893ns, existing in the KeySchedule path, so pipelining of SubBytes would not have resulted in considerable gains.

B. Comparison to existing Implementations

Most low area implementations in the literature have focussed on encryption only implementations, which is a problem when trying to compare our design. Encryption only designs are viable through the use of Cipher FeedBack (CFB) or Cipher Block Chaining (CBC) block cipher modes, which can implement encryption and decryption with only an encryption primitive. Obviously an encryption only design is significantly less complex and can be implemented in a smaller area, but cannot utilise all possible block cipher modes. When we examined some of our design components and implemented an encryption only variant, we ended up with a reductions ranging from 20-50% in the subcomponents. If we assume a general system slice reduction of 35%, our slice count of 456 becomes 297.

Another issue in comparison is the used of BRAM in designs, which are often not counted in the device slice count. For this, we have used the assumption from Yong-Sung et al [8] that 32bits of BRAM is equal to 1 additional slice. By using this, we can more fairly compare against implementations with and without BRAM.

Table 4 shows a comparison of existing AES on FPGA implementations with our own design, including some of the comparison methods listed above.

As seen in the table, our design provides a good throughput and slice efficiencies, particularly in encryption mode. Decryption mode obviously incurs the large 160 clock delay of computing the 10th key, which significantly reduces throughput, but still provides adequate throughput for low area applications.

IV. Conclusions

We have presented a 456 slice implementation of 128bit AES in ECB mode, implemented on a Spartan 3E. Whilst not the smallest nor highest throughput, the design achieves a decent throughput and provides a good base for further development. Below we outline some future development points, which

Design	**Device**	**Architecture**	**Freq (MHz)**	**N_{Slices}**	**N_{bram}**	**$Equiv_{Slice}$**	**$N_{ClockCycles}$**	**Tput (Mbps)**	**Eff$\left(\frac{kbps}{Slices}\right)$**
Chodowiec & Gai [9]	Spartan 2	32-bit	60	222	3	522	44	266	318
Rouvroy & al [10]	Spartan 3	32-bit	71	163	3	1231	46	208	169
T.Good & M.Benaissa [8]	Spartan 2	8-bit	67	124	2	264	3691	2.2	8.3
J.Chu & M.Benaissa [4]	Spartan 3	8-bit	45.642	184	0	184	160	36.5	198
Yong Sung Jeon & al [11]	Spartan 2	8-bit	66	258	0	258	352	24	93
Our Design	Spartan 3E	8-bit	71	456	0	456	326 (E) 482 (D)	55.8 (E) 37.7 (D)	122.3 (E) 82.7 (D)

Table 4. Comparison of AES implementations

became apparent to us during the project but were not implemented due to time constraints.

A. Future progression

It is obvious that future development of this design should focus on an alternate mode of AES, such as CFB, where only the encryption primitive is needed. This should significantly reduce the complexity of the design and would likely allow better pipelining options, increase performance and decreasing slice count.

It should also be noted that ECB is considered 'insecure' as identical blocks in a file produce identical output, thus failing to obscure patterns in the encrypted data.

Another development area would be to move away from traditional ShiftRows output, instead opting for an output order that would simplify the MixCollumn process but that results in the correct output afterward. This would be more difficult to test, as ShiftRows and MixCollumns would have to be tested as a combined process, but it may provide a reduction to the total delay delay through ShiftRow and MixCollumns, along with reduction in control and arithmetic circuitry.

V. Bibliography

[1] National Institute of Standards and Technology, "ADVANCED ENCRYPTION STANDARD (AES)," 26 November 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. [Accessed 19 April 2016].

[2] M. H. Jing, Y. H. Chen, Y. T. Chang and C. H. Hsu, "The design of a fast inverse module in AES," in *Info-tech and Info-net*, Beijing, 2001.

[3] Zhang, Xinmiao, Parhi and K. K, "High-speed VLSI architectures for the AES algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 12, pp. 957-967, 2004.

[4] J. Chu and M. Benaissa, "Low area memory-free FPGA implementation of the AES algorithm," in *22nd International Conference on Field Programmable Logic and Applications (FPL),* 2012.

[5] P. Hämäläinen, T. Alho, M. Hännikäinen and T. D. Hämäläinen, "Design and implementation of low-area and low-power AES encryption hardware core," in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006.

[6] Xilinx, "Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs," [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp465.pdf . [Accessed 18 04 2016].

[7] K. Chapman, "Saving Costs with the SRL16E," Xilinx, [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp271.pdf. [Accessed 18 April 2016].

[8] T. Good and M. Benaissa, "AES on FPGA from the Fastest to the Smallest," in *Int. Workshop on Cryptographic Hardware and Embedded Systems*, Edinburgh, 2005.

[9] P. Chodowiec and K. Gaj, "Very Compact FPGA Implementation of the AES Algorithm," in *Cryptographic Hardware and Embedded Systems-CHES 2003*, 2003.

[10] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater and J.-D. Legat, "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications," in *Information Technology: Coding and Computing, 2004. (Proceedings of)*, 2004.

[11] Y.-S. Jeon, Y.-J. Kim and D.-H. Lee, "A compact memory-free architecture for the AES algorithm using resource sharing methods," *Journal of Circuits, Systems, and Computers,* vol. 19, no. 05, pp. 1109-1130, 2010.