AES Implementations Optimized for Mid-Range FPGAs

Bryan M. Sobczyk

Abstract— The Rijndael Algorithm was chosen for the Advanced Encryption Standard (AES) in 2001 and formally published in FIPS Publication 197. Since Rijndael was released as a candidate a number of cores were created to test and benchmark the algorithm in both hardware and software. Rijndael was chosen partly based on its ability to be efficiently implemented in Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). In AISC design, heavy use of combinational logic is advantageous while in FPGA designs each logic cell has local memory available and all free logic cells are equally valuable for design use. A survey of published AES architectures found they did not fully take advantage of ROM blocks to simplify and shorten critical paths in the algorithm’s rounds. This paper will present a T-box design that will utilize FPGA memory to create a core compatible with a variety of standard 32-bit bus width that will sustain throughput of 20 Mbyte/sec.

Index Terms—Advanced Encryption Standard, AES, Tbox, Cryptography, AES-128, AES-192, AES-256.

I. AES OVERVIEW

The Advanced Encryption Standard (AES) specification is documented in the National Institute of Standards and Technology’s (NIST) FIPS 197 publication.[5] J. Daemen and V. Rijmen submitted Rijndael as part of NIST’s AES contest. Candidates for the contest were tested based on strength of the algorithm against attacks, maximum throughput, and resources required for both software and hardware implementations. Rijndael was originally designed with a variety of key lengths and variable block lengths in mind. When the variable block length requirement was dropped, Rijndael was amended to fixed 128-bit block length. Since the chosen core would be a US Federal Standard all teams participating had to openly publish their standard and must be free of Intellectual Property that must be licensed. The finalists were evaluated equally but each submission differed in implementation costs, throughput, and versatility in implementation. Flexible algorithms that could run efficiently on Application Specific Integrated Circuits (ASICs) for smart cards, 32-bit microprocessors, and even 8 bit microcontrollers proved a challenge during final selection. [11] On October 2nd, 2000 NIST announced that Rijndael was the winner and new AES standard, based on the evaluation criteria, peer review, and excellent performance across a number of platforms.

AES supports standard key sizes of 128, 192, and 256 bits and is used as a block cipher with a message size of 128 bits. The block cipher structure can be used in a variety of modes to create a secure stream cipher based on AES encryption and/or decryption. Using the basic Electronic Code Book (ECB) mode, a 128-bit message is encrypted with a key of 128, 192, or 256 bits to produce a 128-bit cipher text as shown in figure YY. A key expansion is first performed on the initial key values based on a key schedule to generate unique keys for all rounds of encryption. The key schedule was developed to use small amounts of memory, have no symmetries, have efficient diffusion of keys, and be non-linear. [11] Diffusion allows small changes in a previous key to cause significant changes in the next expanded key. Elimination of symmetries and linear functions allows generation of expanded keys that resist attacks and analysis on the cipher text. A perfect key expansion would generate seemingly random keys that are unique and easily computed from the initial key and subsequent expanded keys. With no pattern to attack, the attacker would have to pick from all possible keys for every round of the algorithm. AES uses rotations and XOR operations for permutations and table lookups from an Sbox table specified in FIPS 197 for direct substitutions on each byte of the key. To further mix each key in the schedule a RCON value is to add a unique constant determined by the current key being generated. On every new round of four, six, or eight keys an incrementing RCON value is XORed with the key to eliminate symmetries in the expanded keys.[5]
key undergoes a substitution to keep the same transform being applied over more than three consecutive expanded keys. The RCON values are listed in FIPS 197 along with example key expansions for 128, 192, and 256-bit key lengths.

Based on the round structure chosen, there are a number of options for implementing the key schedule. If each round is calculated iteratively, it is easy to calculate the expanded key on the fly by expanding only the current key and retaining the previous key values required to calculate the next key. To implement AES with a 128-bit key, only four previous values are required to calculate the keys on the fly. A full key expansion for all modes, some times referred to as a 3 in 1 design, requires up to eight previous values to be retained. If a full round is calculated for each cycle, four keys must be simultaneously calculated and up to eight values need to be retained for compatibility with all modes. A potential drawback to on the fly calculation is the worst-case delay of looking up a value in a ROM table, two bit-wise XORs, and the gate delay of two muxes. The full round in a cycle approach can provide very high throughputs (128 bits computed every cycle) but require fast key expansion or the entire key schedule to be pre-computed and stored for use during the rounds. Storing the entire key schedule requires up to sixty 32-bit words to be stored for the 256-bit mode but allows the keys to be referenced in a single clock cycle if fast RAM blocks are used. For most modes having the pre-computed key values can speed the calculation of the rounds. If a mode with a changing key is used the benefits of pre-computation are diminished compared to the memory resources required. For memory limited applications, on the fly calculation provides fast key expansion without RAM blocks.

A survey of other papers has found both preprocessed and on the fly key expansion used in high throughput designs. McCloone and McCanney utilized preprocessing of the key with a LUT based Tbox implementation discussed later in this paper. [15] Qing et al, Wang and Ni, Wang et al, Lofty et al, Rizk et al, and Standaert et al had all generated their keys on the fly to save on processing time and hardware.[29, 25, 26, 14, 17, 23] Two on the fly key scheduling units are chained to provide Schaumont et al Rijndael processor capable of 2.29 Gbit/sec throughput. [20] For a fully pipelined high-speed core, the keys are preprocessed to allow the rounds to compute as soon as possible with no calculation delays achieving a through put of 30-70 Gbits/sec.[9] Lin and Huang pipelined the computation of the key to matched their pipelined round structure. [13] The timesavings in pre-expanding the key in the pipelined and single cycle architectures justified the use of more memory resources to store the keys.

The AES standard is round based and operates on a thirty-two bit by thirty-two bit State array. The array is divided into sixteen bytes as shown in figure XX. It should be noted that the indexes into the array are row then column. The indexes correspond to the byte sequence; every four bytes form a new row. The state is then taken through a number of rounds determined by the key length summarized in table XX. All operations are performed in the Galois Field GF(2^8). A full explanation of Galois Fields is presented in Stallings' textbook but the basic arithmetic operations can be explained briefly. [21] The equivalent of addition and subtraction in the field is XORing the two values. Multiplication and division involve a more complex algorithm but a simple table lookup can perform the multiplication of any number and two or three which is required for AES. As suggested by Daemen and Rijmen in [11], a table of all 256 values multiplied by 0x02 is listed as the xtime table. Any multiplied value can be decomposed into a sum of powers of 0x02. For example 0x03*X can be expressed as X*(0x01 XOR 0x02) which can be calculated as (X XOR xtime(X)). This technique will be utilized later to pre-calculate the tBox values for our architecture.

The plaintext message is loaded into the State array of figure YY and the AddRoundKey operation is performed. During this initial state the message is XORed with the initial keys. The new State array is taken through a series identical rounds stated in the FIPS 197 standard including SubBytes, ShiftRows, MixColumns, and finally AddRoundKey. The rounds are designed to be invertible for decryption while being simple to insure delays are low since rounds are repeated at least ten times. SubBytes is a non-linear substitution of bytes directly substituted from the sBox array. Details of the sBox's construction are detailed in [11], the authors provided the basis for choosing values as well as mathematically proving the table transformations have inverses that are used for decryption. ShiftRows provides a diffusion of values within the State array through simple shifts. MixColumns provides a linear permutation of the State array on a byte-by-byte basis, while being easily reversed by an inverse permutation. Daemen and Rijmen have stated in [11] that the performance of this step on 8-bit processors was a driving factor in the choice of permutation. The final step, AddRoundKey, XORs each 32-bit column of the State array with one 32-bit word of the expanded key. A complete round requires four 32-bit expanded keys for each round. The number of rounds chosen by Daemen and Rijmen based on known cryptanalysis attacks and the diffusion criteria that require two rounds to fully diffuse single bit changes. The foreknowledge of how attackers could exploit the algorithm have lead to logical choices in each step of Rijndael, and hence AES, to make known attacks computationally infeasible, while preventing easily exploited patterns in the algorithm from developing. [11 & 21]
After the specified number of rounds described above is done, the final round only includes SubBytes, ShiftRow, and AddRoundKey steps. The resulting State array is the cipher text and is output as a 128-bit word. Decryption is performed using the same round structure and sequences of operations, but inverse tables are used for each step. The expanded keys are generated in the same way but groups of four sequential 32-bit keys are used in reverse order.

II. ENCRYPTION MODES

Use of low area and high throughput encryption cores with an approved mode can match or exceed the throughput of ECB mode encryption and decryption. The use of approved modes can prevent identical plaintext blocks from having identical cipher text blocks. NIST SP 800-38A was released to describe the proper way to implement encryption mode. NIST has listed the following approved confidentiality modes as of September 3, 2008:

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feed Back (CFB)
- Output Feed Back (OFB)
- Counter (CTR)

If only authentication is required, SP 800-38B specifies the use of Cipher-based Message Authentication Code Mode (CMAC). For both authentication and confidentiality, SP 800-38C specifies Counter with Cipher Block Chaining – Message Authentication Code (CCM). For high throughput authentication with confidentiality Galois/Counter Mode (GCM) is specified by SP 800-38D but strict adherence to the recommended Initialization Vectors is required to satisfy the uniqueness requirement that provides the high degree of security.

The choice of the listed modes provides stronger security than the basic ECB mode but has consequences for the operation of the cipher. The simple Counter (CTR) mode provides a fast streaming cipher that only requires block encryption. Each message is XORed with the counter value to produce a block of cipher text. CTR mode encryption and decryption can be done in parallel, is easily pipelined, and allows changes to individual blocks without affecting other blocks. The Initial Vector (IV) can be provided by a simple counter or another unique but changing value like a sector address. It should be noted the counter width should produce enough unique values to prevent repeats or spread out repeats so $2^{256}$ unique values get used before repeats occur between the 128-bit blocks used with AES. If a single bit error is encountered with the message CTR will cause a single bit error in the cipher text. CTR provides a simple way to use an optimized encryption core for both stream encryption and decryption. A subset of CTR can utilize only a portion of the counter’s bits, but it should be noted that only the same portion of the message could be encrypted with this method. Utilizing a larger AES key will not overcome this limitation since only a portion of the 128-bit State array can be used, regardless of the key size used.

OFB mode chains the resulting encryption from the previous block as input to the next block’s encryption block. The resulting cipher text is XORed with the message to provide the cipher text. Since each block encryption is dependent on the previous result it is not possible to pipeline or use parallelism to speed up the calculation. OFB does not depend on a sequence of Initial Vectors as CTR mode did. The first IV value provides the seed value used for all subsequent blocks. Again the same encryption module can perform encryption and decryption. OFB can provide security for noisy channels since small bit errors do not propagate through other blocks.

CFB mode differs from OFB in that the feedback is taken from the previous cipher text and not the encryption step. The drawback to this approach is that errors can propagate between each subsequent block. CBC mode is unique from previous modes since an IV is XORed with the message before encryption/decryption is performed. Unlike the other modes encryption and decryption modules are required but errors affect the current and next blocks of data.

A summary of modes is depicted in figure QQ. Of the available modes, this paper’s architecture makes use of Counter mode for increased strength over ECB mode. A simple modification can allow OFB to be implemented due to its similar structure to CTR mode. These modes allow the encryption module to be utilized for both block encryption and decryption without requiring extra tables to be stored.

III. TBOX IMPLEMENTATION

Since the publication of Rijndael and the final AES
specification a number of architectures to improve performance have been proposed and tested. One method to increase throughput without resorting to pipelining was to shorten the rounds to a single cycle. The tBox table approach was suggested and derived by Daemen and Rijmen and utilized by a number of researchers for dedicated 32-bit hardware to improve performance. [11] An excellent explanation and datapath example is derived and explained by Gaj and Chodowiec in a chapter of Cryptographic Engineering. [7] The tBox structure combines the SubBytes, ShiftRows, and MixColumns steps utilizing linear algebra rules in the GF($2^8$) creating a 256 value table of thirty two bit words called a TBox. Every standard eight-bit sBox lookup in SubBytes can now be looked up in the tBox, which returns a thirty-two-bit result.

As stated in [11] and [7] the transformations for one full round of AES are listed below:

\[
\begin{align*}
&\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = \begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix} \begin{bmatrix}
S(a_{0,j}) \\
S(a_{1,j+1}) \\
S(a_{2,j+2}) \\
S(a_{3,j+3})
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
&\begin{bmatrix}
&k_{0,j} \\
&k_{1,j} \\
&k_{2,j} \\
&k_{3,j}
\end{bmatrix} = \begin{bmatrix}
t_{0,j} \\
t_{1,j} \\
t_{2,j} \\
t_{3,j}
\end{bmatrix} \oplus \begin{bmatrix}
00 & 01 & 01 & 01 \\
01 & 00 & 01 & 01 \\
01 & 01 & 00 & 01 \\
01 & 01 & 01 & 00
\end{bmatrix} \begin{bmatrix}
S(a_{0,j+1}) \\
S(a_{1,j+2}) \\
S(a_{2,j+3}) \\
S(a_{3,j+4})
\end{bmatrix}
\end{align*}
\]

Using the rules and properties of matrix math in GF($2^8$):

\[
\begin{align*}
&\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = \begin{bmatrix}
02 & 01 & 03 & 01 \\
01 & 02 & 01 & 03 \\
03 & 01 & 01 & 01 \\
01 & 03 & 01 & 01
\end{bmatrix} \begin{bmatrix}
S(a_{0,j}) \\
S(a_{1,j+1}) \\
S(a_{2,j+2}) \\
S(a_{3,j+3})
\end{bmatrix}
\end{align*}
\]

The equation above can be implemented by tables that include the values of the Sbox multiplied by the 4x1 matrixes, which are now represented as Tboxes below:

\[
\begin{align*}
&\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \
&\begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}]
\end{align*}
\]

Each TBox table uses an eight-bit index into a 256-element array of thirty-two bit values. Storing these tables takes 8Kbits of ROM per table, for a total of 32Kbits of tables. A reduction in the number and size of the tables can take advantage of two FPGA strengths. First, the FPGA can use wires to copy or shift values at no cost in resources. This advantage is not possible with general-purpose processors that require separate processing cycles to shift values. Second, the four tables are built on the same eight bit values that are shifted to form the other tables. It is possible to use a 256-item table of twenty-four bit words and shift it a total of three times to get all four TBox table values. To save memory only one TBox could be used to give a four-time reduction in memory requirements. The down side is the table would cause a four-fold increase in processing time. For this paper a trade off was made to use four tables from the same Tbox initial values and shift them as necessary. This allows 32-bits of the State array to be computed every cycle and stored in the next State’s registers.

\[
\begin{align*}
&\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = \begin{bmatrix}
T_0[a_{0,j}] \oplus Shift(T_1[a_{1,j+1}]) \oplus Shift(T_2[a_{2,j+2}]) \oplus Shift(T_3[a_{3,j+3}])
\end{bmatrix} \oplus \begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix} \\
&\begin{bmatrix}
sameTBox0 \\
sameTBox0 \\
sameTBox0 \\
sameTBox0
\end{bmatrix} = \\
\begin{bmatrix}
T_0[a_{0,j}] \\
T_1[a_{1,j+1}] \\
T_2[a_{2,j+2}] \\
T_3[a_{3,j+3}]
\end{bmatrix}
\end{align*}
\]

At the cost of memory and logic element resources it is possible to build a high throughput core that computes the four instances of the above TBox equations in a single cycle. In this architecture a maximum of fourteen cycles would be needed to complete the 128-bit cipher text from a given plaintext already loaded at high speeds. Gaj and Chodowiec present an efficient version of this architecture that can be easily pipelined. [8] A similar decryption architecture using inverse tables can be used in the same way but must be supplied keys in the correct order to perform decryption correctly. Wang et al, McLoone and McCanny, and Rouvoy et al have published architectures similar or identical to the Tbox architecture as presented by Gaj and Chodowiec.[26, 15, 22] Of specific interest to mid-range FPGA implementations is the approached the UCL Crypto Group took to use RAM blocks, shift registers, and short datapaths to create a core that can encrypt and decrypt at 208 Mbit/sec on Spartan II devices. This was achieved with only 163 slices and 3 RAM blocks used. [22] The numbers compare favorably with a number of designs optimized for use on the larger and more capable Virtex FPGAs, while not using a majority of the device’s resources. This paper’s architecture will be compared to the UCL group’s efficient core in the comparison section.

It should be further noted that tbox implementations do not always meet or exceed the throughput of more conventional AES implementations optimized for speed and/or area. The tBox method is not well suited to ASIC design due to the large amount of memory required. Memory can be laid out very efficiently in Very Large Scale Integration (VLSI) processes but consumes a large area of the die compared to a sea-of-gates implementation of combinational logic functions. The UCL Crypto Group Core on a Spartan 3 FPGA achieves 208 Mbit/sec with 163 slices used. McLoone et al achieved 6956 Mbit/sec using 2222 slices but required a Virtex FPGA. Huang et al. bested the UCL Group by not using a tBox architecture and achieved 647 Mbit/sec at a cost of 148 slices and 11 Block RAMs of a Spartan 3 device.

Software implantations would not benefit from the same techniques described above as software cannot make use of efficient shifts, cross wiring, and fast logic operations. The
sequential nature of all modern general-purpose processors makes all operations equally costly in terms of delay. The basic algorithm can be optimized for specific processors in order to utilize matrix instructions, parallel processing via multiple cores, and/or the use of efficient compilers to minimize the assembly instructions required. Bernstein and Schwabe have outlined fast AES techniques that use each processor unique structure such as 64 bit operations, verses 32 bit calculations, masked tables, and efficient use of caches to shorten the time required to compute all rounds of AES. [30,31] In a method similar to the matrix math performed for Tboxes, another manipulation of the matrix is possible to effectively compute across the rows of the state to save extra corrective shifts. Bertoni et al. have saved rotations and extra memory lookups to speed up the standard software implementations. Further Discussion of transformations that can use methods like the tBox to obtain faster software implementations. Bertoni et al. have saved rotations and extra memory lookups to speed up the standard software implementations. Further Discussion of transformations that can use methods like the tBox to obtain faster software implementations.

IV. FPGA CELL ARCHITECTURE

This paper aims for a Field Programmable Gate Array (FPGA) implementation of AES and warrants an investigation into the architecture of both the Xilinx and Altera part families. The Cyclone II and Spartan 3 part families are equivalent low cost, mid-range small FPGAs. These devices do not have the resources or speed of a Virtex or Stratix devices but do provide greater performance than the older APEX/MAXII and XC/CoolRunner parts.

Altera’s Cyclone II/III families are composed of Logic Array Blocks (LABs). Each LAB contains sixteen Logic Elements (LE). Dual ported RAM is also available via M4K RAM blocks with high clock rates (up to 260 Mhz). Each LE provides a four input Look Up Table (LUT), a register, carry chain logic, and connection Muxes. Unlike the Xilinx equivalents of LE blocks, called slices, Altera elected not to add extra discrete logic elements within the cells, all functions are implemented via LUTs. This gap in implementation paths is offset by Altera’s LE modes, a normal mode allows arbitrary functions while arithmetic mode configures the LE for fast and compact arithmetic functions. Using both the primary inputs and carry-in logic a six input function can be realized in each LE. For comparison, Xilinx’s slices can implement only four input functions in each slice.

Xilinx’s Spartan 3 family is built on Configurable Logic Blocks (CLBs) that contain four slices each. Every slice contains two logic function tables, two registers, carry logic, and individual logic gates. A number of slices provide shift registers and Distributed RAM as special functions. Better use of the cells provides high resource utilization, and allows large and complex designs to fit in smaller devices. The function tables can be used as Look Up Tables (LUTs) that can implement any four input function as stored values. Two four-input tables can be stored in each slice, and the output can be registered within the slice. Many adders and multipliers can be implemented to fit within the cell, and make use of the carry logic and spare gates for very high slice utilization. Spartan’s block RAMs offer 18Kbits of fast, dual-ported, synchronous storage. For 32-bit numbers, 512 elements can be stored in one 18K block. As outlined in the Xilinx Spartan 3 User Guide, each dual ported RAM can also be used for 128 states with up to 36 outputs in a single block. Use of the SRL16 shift register blocks can greatly lower device utilization if a reset is not required. SRL16 blocks configure the LUTs as a shift register, but do not use the slice’s flip-flops. The built-in Mux can select either of the slice’s LUTs to configure longer shift registers at low hardware cost.

Each manufacturer has gone to great lengths to point out the similarities between underlying hardware and promote any difference in performance. Altera claims their faster speed grades produce faster customer implementations, and Xilinx promotes the special configuration modes like SRL16 registers and the fast carry logic as leading to better, high performance designs. Xilinx has acknowledged that newer versions of the Quartus design suite from Altera have higher performance than earlier implementations. Due to Altera’s table centric cells, the software’s optimization and fitting of logic is very important to high throughput designs. Xilinx has a similar push for their ISE tools to recognize logical parts of designs that can fit into each cell and still recognize special cases like the SRL16 register. Ultimately only real life testing of designs across devices, vendors, and software tools can prove the advantages of each manufacturer’s product line.

V. OUR ARCHITECTURE

The architecture described in this paper is based on the tBox implementation but targets smaller CPLD families like Altera’s Cyclone II/III and Xilinx’s Spartan III. The specification for the core included a number of items that were included for interfacing and re-utilizing costly resources and not for performance. The specification included:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements LE</td>
<td>4.608</td>
<td>8.256</td>
<td>14.446</td>
<td>18.752</td>
<td>33.216</td>
<td>50.528</td>
<td>68.416</td>
</tr>
<tr>
<td>M4K RAM (4 Mbit RAM)</td>
<td>26</td>
<td>26</td>
<td>52</td>
<td>52</td>
<td>105</td>
<td>129</td>
<td>250</td>
</tr>
<tr>
<td>Minimum User I/O Pins</td>
<td>89</td>
<td>85</td>
<td>152</td>
<td>142</td>
<td>322</td>
<td>294</td>
<td>422</td>
</tr>
<tr>
<td>SRL16/Blackboard</td>
<td>144-128</td>
<td>144-128</td>
<td>NA</td>
<td>144-128</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Schematic/Logic Elements</td>
<td>144-128</td>
<td>144-128</td>
<td>264-180</td>
<td>256-180</td>
<td>484-85A</td>
<td>484-85A</td>
<td>672-85A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xilinx Spartan 3 Family</th>
<th>XC3S50</th>
<th>XC3S150</th>
<th>XC3S400</th>
<th>XC3S450</th>
<th>XC3S500</th>
<th>XC3S550</th>
<th>XC3S750</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk (12 MHz Clock)</td>
<td>190</td>
<td>280</td>
<td>496</td>
<td>1.195</td>
<td>3.378</td>
<td>9.126</td>
<td>6.912</td>
</tr>
<tr>
<td>Distributed RAM Bits</td>
<td>12K</td>
<td>20K</td>
<td>50K</td>
<td>120K</td>
<td>289K</td>
<td>90K</td>
<td>433K</td>
</tr>
<tr>
<td>Block RAM</td>
<td>12K</td>
<td>21K</td>
<td>28K</td>
<td>43K</td>
<td>57K</td>
<td>72K</td>
<td>178K</td>
</tr>
<tr>
<td>Minimum User I/O Pins</td>
<td>63</td>
<td>63</td>
<td>97</td>
<td>173</td>
<td>221</td>
<td>333</td>
<td>489</td>
</tr>
<tr>
<td>Schematic/Logic Elements</td>
<td>VQ100</td>
<td>VQ100</td>
<td>TQ144</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
• Throughput of at least 20Mbyte/second (160 Mbit/second) through the core
• 32 bit or 8 bit Input/Output registers utilizing interrupt-like triggering
• tBox-based round architecture to achieve single cycle calculations inside the rounds
• Reuse of most registers, minimal use of RAM
• Use of only standard IEEE compatible VHDL code to insure code can be ported across vendor tools
• Register-Transfer-Level (RTL) style coding to avoid excess hardware creation
• No vendor specific libraries, macros, or Intellectual Property (IP) cores
• Modular bottom-up construction to aid in testing and reuse
• Separation of data path and controllers to keep control hardware apart from the data path.
• Strict use of only STD_LOGIC and STD_LOGIC_VECTORS to insure no type conversions are necessary
• Simple external control of the core for I/O, mode selection (AES128/192/256), and status

The tBoxes used in this paper were computed manually utilizing the xtime tables provided by Daemen and Rijmen and the results were copied to a VHDL description of a ROM. [11] It should be noted that McLoone and McCanny published three tables used to create the tBoxes. An Sbox, Sbox * 02, and Sbox * 03 table were presented to form all four tBoxes as necessary. A comparison of this paper’s tables with the tables published by McLoone and McCanny turned up five discrepancies:

• Sbox * 02 table for B3 & B4 are reversed (AF 25 instead of 25 AF)
• Sbox * 03 table for B3 & B4 are also reversed (AF 25 instead of 25 AF)
• Sbox * 03 value for 56 contains a typo (E0 instead of E8)

Examination of the specific terms suggest the swapped terms in the Sbox * 02 table were continued to the Sbox * 03 table, since the correct values were listed. The E0/E8 type looks like a simple typing mistake. The remainder of the tables successfully confirmed the table data in this paper’s implementation when NIST test vectors were run and verified.

Original planning for the tables included only one tBox selected by multiple XORs, but it was realized that having only one tBox allowed only look up per clock cycle. This would require multiple clock cycles per round. During key expansion and final rounds the need for the sbox values would also take a performance hit since only one eight bit value can be looked up at a time. It was decided to use the same 24-bit wide table four times to allow single cycle rounds and simultaneous 32-bit sBox lookups.

In order to better utilize each Slice or Logic Element an experiment was conducted. It was suggested that the synthesis tools would use memory resources better if the tables were not 256 items x 24-bit words but left as 256 items x 8 bit tables. Both the Xilinx and Altera synthesis tools were given a two-register bank and one table design. One used a 24 bit x 256 word ROM and another using three separate 8 bit x 256 word ROM tables. No code was added to specify the ROM should

Preliminary Tbox Resources

<table>
<thead>
<tr>
<th></th>
<th>Xilinx ISE 9.1 Webpack</th>
<th>Altera Quartus II Web Ed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Size</td>
<td>24x256 3x8x256 4x24x256</td>
<td>24x256 3x8x256 4x24x256</td>
</tr>
<tr>
<td>Slice</td>
<td>18 18 74 74</td>
<td>547 539 2156 2156</td>
</tr>
<tr>
<td>Logic Elements</td>
<td>1 3 2 6</td>
<td></td>
</tr>
<tr>
<td>Max CLK HZ</td>
<td>134 MHz 139 MHz 90.6 MHz 97.2 MHz</td>
<td>205 MHz 205 MHz 166 MHz 166 MHz</td>
</tr>
</tbody>
</table>

---

The image contains diagrams and tables, but the text focuses on the key points mentioned above. The diagrams illustrate the tBox architecture and control logic.
be implemented as chained Lookup Tables (LUTs) or static RAM blocks. Xilinx ISE Webpack utilized BRAM blocks; achieving a final clock rate of 134 MHz. Splitting up the tables utilized 3 BRAM blocks with a slightly higher clock rate of 139 MHz. Altera’s Quartus II Web Edition was set to use RAM and ROM for any size table but when optimizing for speed implemented Logic Element LUTs instead of RAM blocks. Quartus reported a final clock rate of 205 MHz. A second test with four instances of the first test was run and summarized in table YY. Based on the findings the individual tables were used to provide a small increase in speed for the Xilinx devices.

Key Expansion was originally going to be pre-computed and stored in a RAM block until required. This design requires both a counter and a large 32-bit x 60 item RAM block. The requirement for a design that can work for all three modes of AES (128/192/256 bit) without reloading the device required a simple but flexible design. A straightforward design with 32-bit x 4 word register for the key and a maximum of 32-bit x 8 word temporary register bank allows enough storage for the largest 256 bit key size. A small table and a counter that advances on every count that equals zero modulo 4, 6, or 8 respectively supply the RCON constant. As stated before, four separate sBox lookups are performed simultaneously to insure the next key is calculated in one cycle.

It should be noted that if all four lookups of this design are done in one cycle, using a network of 16 Tbox tables described previously, they could be stored in Look Up Tables (LUTs) in each individual Logic Element (Altera’s LE) or Configurable Logic Block (Xilinx’s CLB). This approach allows 128 bits to be calculated every cycle but comes at a high logic element cost. This made it possible to implement the core on smaller FPGA devices, but remains better suited for the larger logic cells of the Stratix and Virtex families.

Aside from memory issues the ports on the core had to be planned to not exceed the I/O pins available on smaller devices. The 32-bit I/O requirement was based on standard bus widths for interfacing to other logic on the FPGA or to an external processor. A survey of open AES cores found most were based on 128-bit I/O to the core, which would not fit in many smaller FPGA families. The I/O bottleneck was also important to consider if the core was partnered with a transceiver chip for use with Universal Serial Bus (USB), Ethernet, or serial port. A fast core throughput would be wasted if it could not exchange data with these interfaces effectively. A throughput of 20Mbyte/second was decided based on the actual throughput seen on a number of USB transceivers and microcontrollers after the USB stack and processor overhead is taken into account. This speed is not the raw 480 Mbyte/second data rates advertised in the USB Version 2.0 specification but the average observed data rates

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>WC/Mode/Comment</th>
<th>Part Family</th>
<th>Device Listed</th>
<th>CLB Slices/LE</th>
<th>BRAMs</th>
<th>Device Utilization</th>
<th>Throughput Reported</th>
<th>Throughput per Block</th>
<th>Throughput per Slice</th>
<th>Optimizations Listed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang/Chang/Liu</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>XCV200E</td>
<td>2022 slices</td>
<td>150 RAMs</td>
<td>Not listed</td>
<td>6956 Mbit/sec</td>
<td>3.1 Mbit/sec</td>
<td>1004 KB/s</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>McLoone/McCanny</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>XCV200E</td>
<td>2020</td>
<td>224 RAMs</td>
<td>Not listed</td>
<td>12020 Mbit/sec</td>
<td>2.47 Mbit/sec</td>
<td>256 KB/s</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Panato/Barcelo/Reis</td>
<td>Not listed</td>
<td>Altera Cyclone</td>
<td>EP1C20F400C6</td>
<td>4057</td>
<td>10 RAMs</td>
<td>Not listed</td>
<td>256 Mbit/sec</td>
<td>4.7 Mbit/sec</td>
<td>1.25 Mbit/sec</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Huang/Chang/Liu/Tai</td>
<td>Not listed</td>
<td>Altera Cyclone</td>
<td>CYCZ18</td>
<td>3046</td>
<td>40960 bits</td>
<td>Not listed</td>
<td>9685 Mbit/sec</td>
<td>4.25 Mbit/sec</td>
<td>3.59 Mbit/sec</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Wang/Chang/Liu</td>
<td>Not listed</td>
<td>Altera Virtex</td>
<td>CYCXV10K</td>
<td>32064</td>
<td>39.5% LE, 100% RAM</td>
<td>Not listed</td>
<td>3.93 Mbit/sec</td>
<td>2.45 Mbit/sec</td>
<td>Not listed</td>
<td>Tbox, pipelining</td>
</tr>
<tr>
<td>Panato/Barcelo/Reis</td>
<td>Not listed</td>
<td>Xilinx Virtex II</td>
<td>CYCXV150</td>
<td>39000</td>
<td>100% LE, 100% RAM</td>
<td>Not listed</td>
<td>80 Mbit/sec</td>
<td>358 Mbit/sec</td>
<td>Not listed</td>
<td>Handel-C</td>
</tr>
<tr>
<td>Huang/Chang/Liu/Tai</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>CYCXV150</td>
<td>39000</td>
<td>0.5% LE, 100% RAM</td>
<td>Not listed</td>
<td>80 Mbit/sec</td>
<td>2.45 Mbit/sec</td>
<td>Not listed</td>
<td>Handel-C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>WC/Mode/Comment</th>
<th>Part Family</th>
<th>Device Listed</th>
<th>CLB Slices/LE</th>
<th>BRAMs</th>
<th>Device Utilization</th>
<th>Throughput Reported</th>
<th>Throughput per Block</th>
<th>Throughput per Slice</th>
<th>Optimizations Listed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang/Chang/Liu</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>XCV200E</td>
<td>2022 slices</td>
<td>150 RAMs</td>
<td>Not listed</td>
<td>6956 Mbit/sec</td>
<td>3.1 Mbit/sec</td>
<td>1004 KB/s</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>McLoone/McCanny</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>XCV200E</td>
<td>2020</td>
<td>224 RAMs</td>
<td>Not listed</td>
<td>12020 Mbit/sec</td>
<td>2.47 Mbit/sec</td>
<td>256 KB/s</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Panato/Barcelo/Reis</td>
<td>Not listed</td>
<td>Altera Cyclone</td>
<td>EP1C20F400C6</td>
<td>4057</td>
<td>10 RAMs</td>
<td>Not listed</td>
<td>256 Mbit/sec</td>
<td>4.7 Mbit/sec</td>
<td>1.25 Mbit/sec</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Huang/Chang/Liu/Tai</td>
<td>Not listed</td>
<td>Altera Cyclone</td>
<td>CYCZ18</td>
<td>3046</td>
<td>40960 bits</td>
<td>Not listed</td>
<td>9685 Mbit/sec</td>
<td>4.25 Mbit/sec</td>
<td>3.59 Mbit/sec</td>
<td>Tbox LUTs</td>
</tr>
<tr>
<td>Wang/Chang/Liu</td>
<td>Not listed</td>
<td>Altera Virtex</td>
<td>CYCXV10K</td>
<td>32064</td>
<td>39.5% LE, 100% RAM</td>
<td>Not listed</td>
<td>3.93 Mbit/sec</td>
<td>2.45 Mbit/sec</td>
<td>Not listed</td>
<td>Tbox, pipelining</td>
</tr>
<tr>
<td>Panato/Barcelo/Reis</td>
<td>Not listed</td>
<td>Xilinx Virtex II</td>
<td>CYCXV150</td>
<td>39000</td>
<td>100% LE, 100% RAM</td>
<td>Not listed</td>
<td>80 Mbit/sec</td>
<td>358 Mbit/sec</td>
<td>Not listed</td>
<td>Handel-C</td>
</tr>
<tr>
<td>Huang/Chang/Liu/Tai</td>
<td>Not listed</td>
<td>Xilinx Virtex</td>
<td>CYCXV150</td>
<td>39000</td>
<td>0.5% LE, 100% RAM</td>
<td>Not listed</td>
<td>80 Mbit/sec</td>
<td>2.45 Mbit/sec</td>
<td>Not listed</td>
<td>Handel-C</td>
</tr>
</tbody>
</table>
once the delays of the USB stack and the 8/16/32 bit microcontrollers that contained a USB physical layer.

VI. RESULTS

To be added later….

VII. PERFORMANCE AND RESOURCE COMPARISONS

To be added later….

VIII. CONCLUSION

To be added later….

REFERENCES


SOFTWARE REFERENCES
