IRACST – International Journal of Computer Networks and Wireless Communications (IJCNWC), ISSN: 2250-3501
Vol. 2, No. 1, 2012

PERFORMANCE ANALYSIS OF CRYPTOGRAPHIC VLSI DATA

T. Muthumanickam,
Research Scholar,
Vinayaka Missions Research Foundation, Deemed University,
Salem, Tamilnadu, India
aswmuthu@yahoo.com

Dr. A. Nagappan,
Principal,
V.M.K.V. Engineering College,
Salem, Tamilnadu, India
nags_slm@yahoo.com

T.Sheela,
Research Scholar,
Vinayaka Missions Research Foundation, Deemed University,
Salem, Tamilnadu, India
sheelamuthu@gmail.com

***Abstract*— Cryptographic algorithms are more efficiently implemented in custom hardware than in software running on general-purpose processors. The hardware implementation approaches for the AES (Advanced Encryption Standard) Algorithm describes the design and performance testing Rijndael algorithm. Compared to software implementation, hardware implementations provide more physical security as well as higher speed. An optimized coding for the implementation of Rijndael algorithm for 256 bytes has been developed. The speed factor of the algorithm implementation has been targeted and a software code in verilog which boasts of a throughput of 2.18 Gb/sec has been developed. The architectural innovations that have been incorporated in the coding include lookup table for round key generation, which facilitates simultaneous execution of sub bytes, shift rows and mix columns and round key generation. This implementation will be useful in wireless security like military communication and mobile telephony where there is a greater emphasis on the speed of communication.**

Keywords- Decryption, Encryption, FPGA, Security (key words)

I. INTRODUCTION

Cryptography is an ancient art and science of writing in secret code. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium for e.g., network like Internet, high throughput applications, such as the encryption of the physical layer of internet traffic, require an ASIC that does not affect the data throughput. For example, software implementation of the Rijndael algorithm on a Pentium 200 Pro yields a throughput of around 100 Mbits/sec, which is too slow for high-end Internet routers.

A high performance, high throughput and area efficient VLSI architecture for Rijndael algorithm that is suitable for low cost silicon implementation is proposed. The architecture is optimized for high throughput in terms of the encryption and decryption data rates using pipelining. Polynomial multiplication is implemented using XOR operation instead of using multipliers to decrease the hardware complexity. Selective use of look-up tables and combinational logic further enhances the architecture's memory optimization, area, and performance.

AES RIJNDAEL ALGORITHM

The proposed standard for the AES which is developed as a computer security standard that became effective on May 26, 2002 by NIST to replace the DES. The cryptography scheme is a symmetric block cipher that encrypts and decrypts 128,192 or 256 bit blocks of data. Lengths of 128, 192, and 256 bits are standard key lengths used by AES. All encryptions are done in a certain number of rounds, which varies between 10, 12, and 14, and it depends on the size of the block length and the key length chosen.

The main flow of the algorithm, as shown in Figure 1.1, uses many lookup tables and XOR operations. The algorithm accepts blocks of size 128, 192, or 256 bits. Independently, the key length can be 128, 192, or 256 bits as well. All encryptions are done in a certain number of rounds, which varies between 10, 12, and 14, and it depends on the size of the block length and the key length chosen. An encryption module is used to generate all the intermediate encryption data, and a separate key scheduling module is used to generate all the sub-round keys from the initial key.

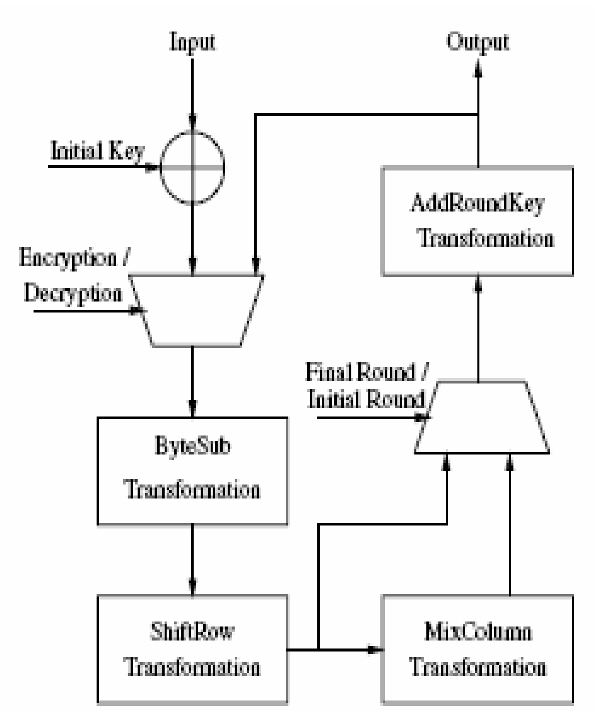


Figure 1.1 Rijndael algorithm flow diagram.

Encryption is divided into four blocks:

Key Addition, Byte substitution, Shift row, Mix column

The key addition is byte XOR between the round key and the encryption data. The Shift Row and the Substitution modules involve mainly table lookups. The Mix Column module composes of XOR operations. The algorithm flow is shown in Figure1.1. The Key Scheduling module is totally independent of the encryption module, and it also involves table lookups and XOR operations. There are a total of three sets of tables used by key scheduling and encryption. One of them is 256 bytes; one of them contains 30 bytes; the remaining one has 24 bytes of entries.

ARCHITECTURE OF RIJNDAEL ALGORITHM

The initial specification of the algorithm was implemented mainly in software, though it is designed for hardware implementation, the transition from software to hardware involves modifications. The main challenge in the hardware implementation is to maximize the encryption throughput while minimizing the area consumption. Maximizing the throughput will minimize the critical paths and solve the memory access conflicts.

The data unit of the architecture consists of the initial round of key addition, Nr − 1 standard rounds, and a final round. The architecture for a standard round composed of four basic blocks is shown in Figure 1.2 For each block, both the transformation and the inverse transformation needed for encryption and decryption respectively are performed using the same hardware resources. This implementation generates one set of subkey and reuses it for calculating all other subkeys in real-time.

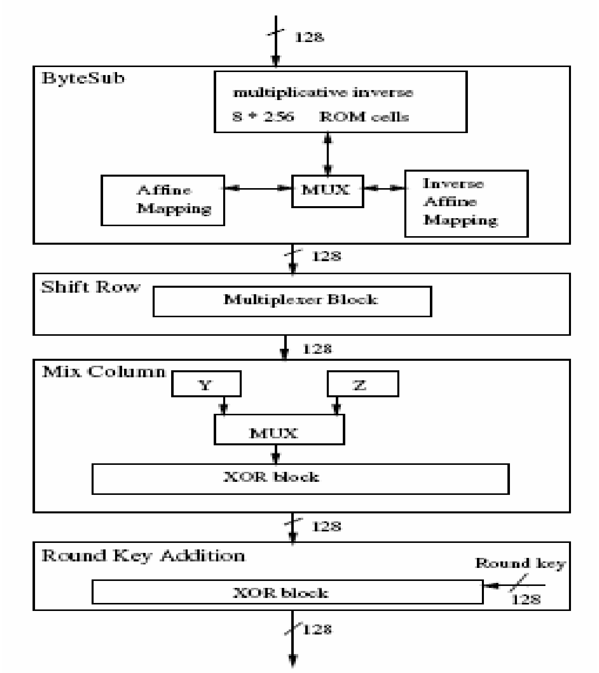


Figure 1.2 Architecture of Standard Round in Data Unit

BYTE SUBSTITUTION

In the architecture, each block is replaced by its substitution in an S-Box table consisting of the multiplicative inverse of each byte of the block state in the finite field *GF* (2^8). In order to overcome the performance bottleneck, the implementation of multiplicative inverses is carried out using look-up tables (stored in a table of 8 × 256). The implementation shown in the Figure 1.3 includes the multiplication inverse with the affine mapping of the input in encryption process.

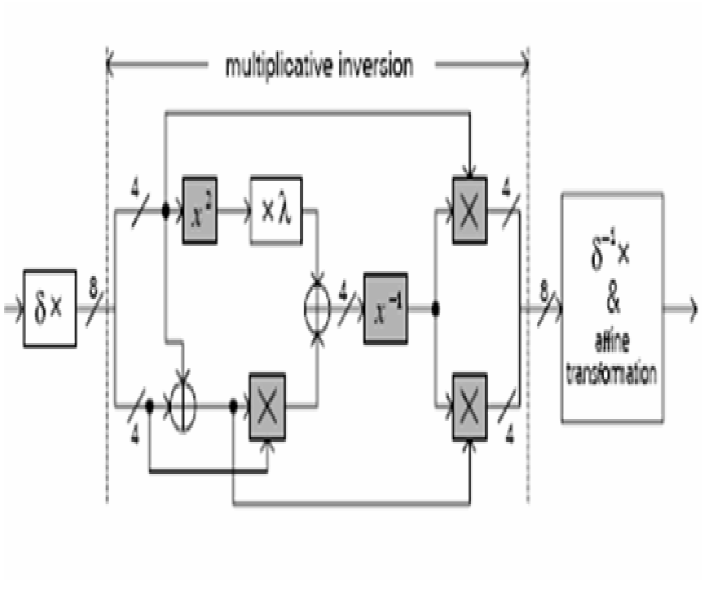


Figure 1.3 Implementation of sub-byte transformation

Each individual blocks in the sub-byte transformation is shown in the following figures.

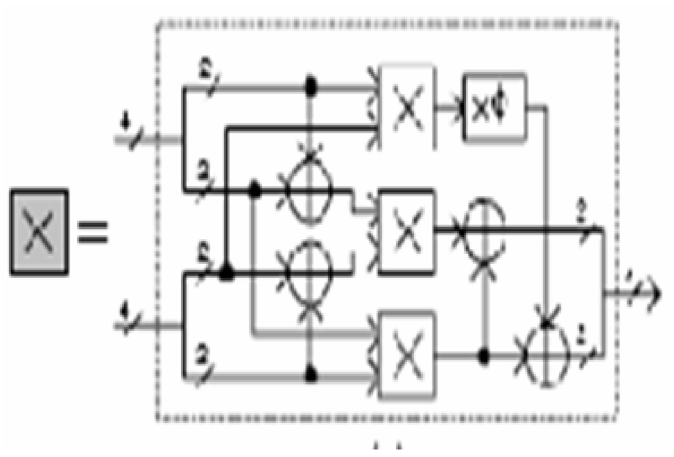


Figure 1.4 Mutliplier in galois field (2^8)

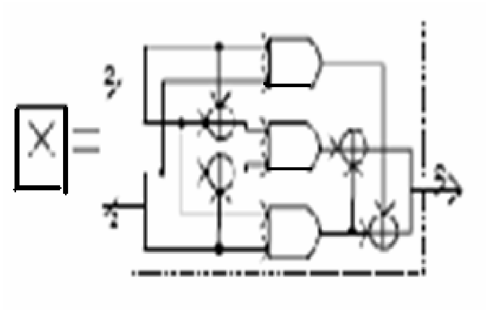


Figure 1.5 Multiplier In Galois Field (2^4)

AFFINE TRANSFORMATION

Ordinary vector algebra uses matrix multiplication to represent linear transformations, and vector addition to represent translations. Using a trick, it is possible to represent both using matrix multiplication. The trick requires that all vectors are augmented with a "1" at the end, and all matrices are augmented with an extra row of zeros at the bottom, an extra column(the translation vector) to the right, and a "1" in the lower right corner, If A is matrix,

$$\begin{bmatrix}\vec{y}\\1\end{bmatrix} = \begin{bmatrix}A & \vec{b}\\0,\ldots,0 & 1\end{bmatrix}\begin{bmatrix}\vec{x}\\1\end{bmatrix}$$

is equivalent to the following.

$$\vec{y} = A\vec{x} + \vec{b}.$$

Ordinary matrix-vector multiplication always maps the origin to the origin. Since the set of vectors with 1 in the last entry does not contain the origin, translations within this subset using linear transformation are possible. This is the homogeneous coordinates system. The advantage of using homogeneous coordinates is that one can combine any number of offline transformations into one by multiplying the matrices. This is used extensively by graphics software.

Affine Mapping:
Out[i]
=In[i]⊕In[(i+4)mod8]⊕In[(i+5)mod8]⊕In[(i+6)mod8]⊕In[(i+7)mod8]⊕CE[i],
Where CE=01100011 is a constant, leftmost bit the being MSB.

Inverse Affine Mapping:
Out[i]
= In[(i+2)mod8]⊕In[(i+5)mod8]⊕In[(i+7)mod8]⊕CD[i],
where CD = 00000101 is a constant, leftmost bit being the MSB

SHIFT ROW

In this transformation the rows of the block state are shifted over different offsets. The amount of shifts is determined by the block length. The proposed architecture implements the shift row operation using Combinational logic considering the offset by which a row should be shifted.

Inside Shift Row, the 256 bit data is broken down into four chunks. Each of the 64-bit chinks is called a roll and it contains eight bytes. Byte-wise cyclic shifts will be performed on each "row" (Figure 1.6), and the amount of shifts is determined by the block length through a simple table lookup (24 entries). Modulus 4, 6, and 8 operations determine the boundaries on which wrap around happens.

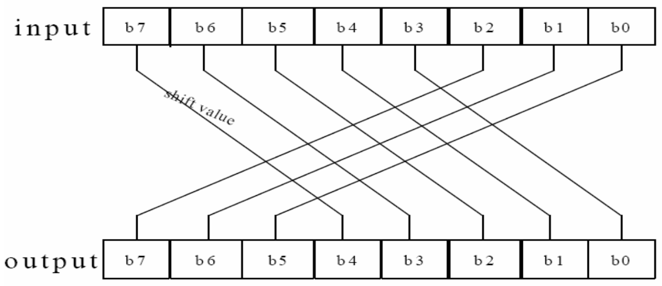


Figure 1.6 Block diagram of shift row (only one of four eight-byte row)

MIXCOLUMN

In this transformation each column of the block state is considered as a polynomial over GF (2^8). It is multiplied with a constant polynomial C(x) or D(x) over a finite field in encryption or decryption, respectively. In hardware, the multiplication by the corresponding polynomial is done by XOR operations and multiplication of a block by X. This is implemented using a multiplexer; the control being the MSB is 1 or 0. The equations implemented in hardware for MixColumn in encryption are as follows.
In encryption process,
Y = In0 ⊕ In1 ⊕ In2 ⊕ In3 and Z = Y.
In decryption process, T0 = In0 ⊕ In1 ⊕ In2 ⊕ In3, T1 =T 0 ⊕ [In2Trans (In2Trans (T0))],
Y = T 1 ⊕ [In2Trans (In2Trans (In0 ⊕ In2))], and Z =T 1 ⊕ [In2Trans (In2Trans (In1 ⊕ In3))].
Out0 = In0 ⊕ [Y ⊕ In2Trans (In0 ⊕ In1)], Out1 =In1 ⊕ [Z ⊕ In2Trans (In1⊕ In2)]
Out2 = In2 ⊕ [Y ⊕ In2Trans (In2 ⊕ In3)], and Out3 =In3 ⊕ [Z ⊕ In2Trans (In3⊕In0)].
Where, In2Trans (K) is the multiplication of the byte by X (hexadecimal value 02) over GF (28). In0 is the least significant 8 bits of a column of a matrix. Architecture of different units is shown, and the architecture of Mix Column transformation is shown in the Figure 1.7.

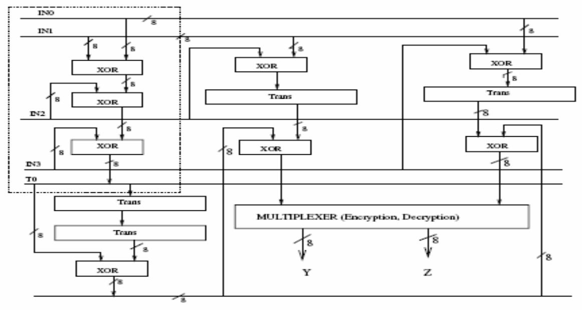


Figure 1.7 Computation of x,y of mix column

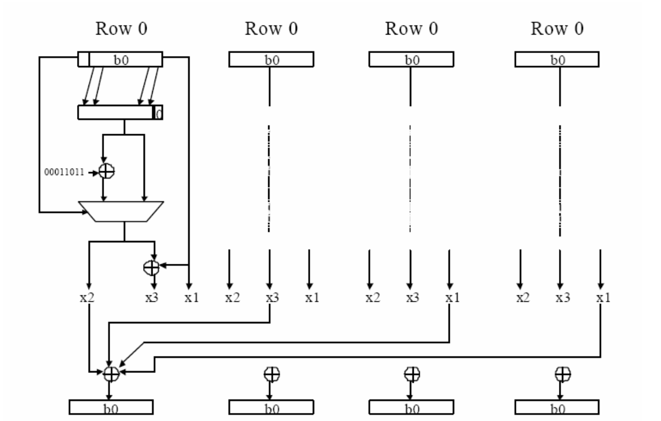


Figure 1.8 Block diagram of mix column (only 0 byte calculation is shown)

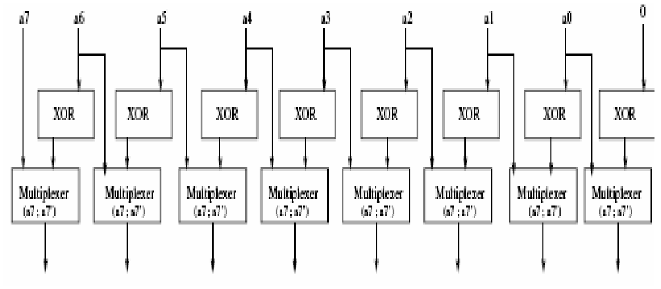


Figure 1.9 Multiplication by x (hex 02)

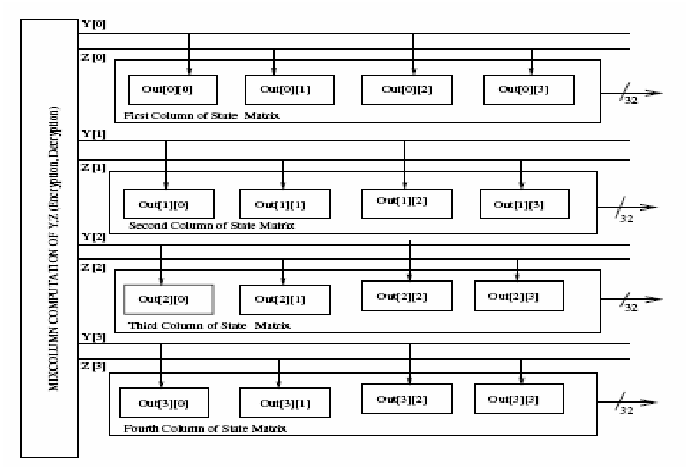

MIXCOLUMN COMPUTATION OF Y.Z (Encryption,Decryption)
Y[0]
Z[0]
Out[0][0]
Out[0][1]
Out[0][2]
Out[0][3]
First Column of State Matrix
Second Column of State Matrix
Third Column of State Matrix
Fourth Column of State Matrix
32

Figure 1.10 Architecture of mix column for 128 bits

ADDROUND KEY

In this transformation (architecture represented in Fig. 1.11), the round key obtained from the key scheduler is XORed with the block state obtained from the MixColumn transformation or ShiftRow transformation based on the type of round being implemented. In the standard round, the round key is XORed with the output obtained from the MixColumn transformation. In the final round the round key is XORed with the output obtained from the ShiftRow transformation. In the initial round, bitwise XOR operation is performed between the initial round key and the initial state block.

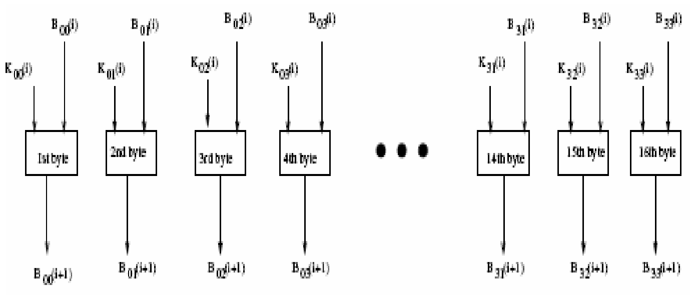


Figure 1.11 Add round key transformation

ARCHITECTURE FOR KEY SCHEDULING

In the key scheduling module, the initial key is expanded and the generated round keys are stored in four 32-bit registers. Both the forward and reverse key scheduling are done in the same device. The ByteSub required in the key expansion unit is implemented using the S-Boxes. Four S-Boxes are needed for a 128-bit key and 128-bit data block implemented using 8×256 ROM cells. Multiplexers are used as a control signal to distinguish between the initial key and the round key (obtained from the initial key using a “key expansion unit”). The least significant 32 bits of the 128-bit key is cyclically shifted to the left by a byte, implemented using combinational logic.

The resulting word after the left shift operation is sent through the S-boxes and the affine mapping operation, in order to perform ByteSub. The key resulting from the ByteSub is XORed with the Round Constant (RCON). In this architecture, the round constant is generated using the combinational logic. The round constant should be symmetric with the round key being generated.

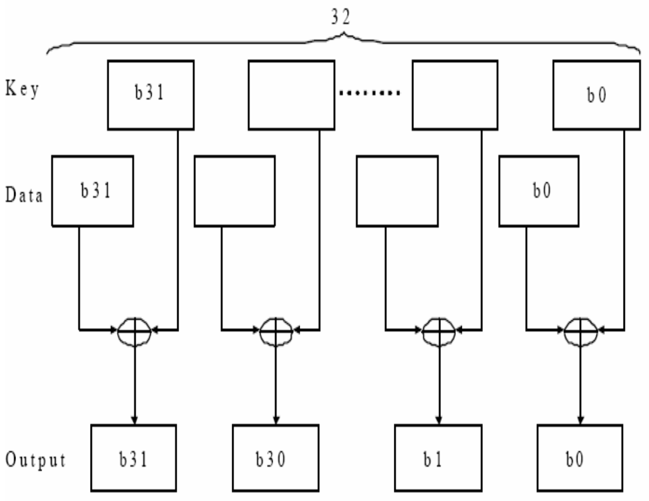


Figure 1.12 Block diagram of key addition unit

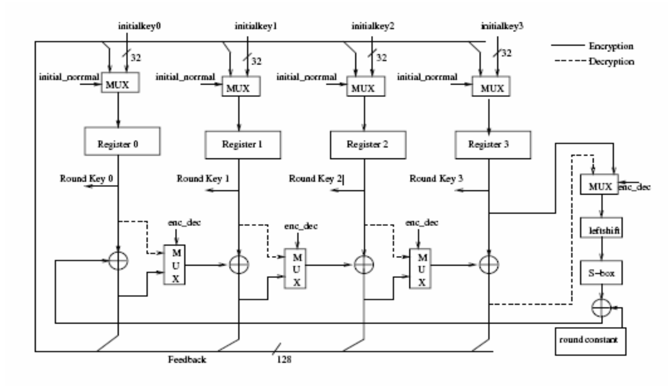


Figure 1.13 Architecture for key scheduling unit

The total number of round constants that need to be generated is equal to the number of rounds. The round constant is obtained in real-time by multiplying the previous round constant by X. This is amenable for implementation in the hardware using XOR operations. For the reverse key scheduling, the last round key should be generated with forward key scheduling for the first time. The last round key is expanded to generate the reverse round keys. Decryption requires more cycles than encryption because it needs pre-scheduling to generate the last key value. Since the Rijndael algorithm allows different key lengths and block lengths, each round key is carefully set to have the same length as the data block. In the case where key length and the block length are not equal, previous, current and also the next round keys are needed in order to generate the appropriate set of round keys that are fed into the encryption module, which is performed by a "key alignment unit".

II. DATAPATH BREAKDOWN

The datapath can be broken down into three parts. In the first part, the 256-bit key is separated into four 64-bit "rows," and the lowest byte of each "row" is used as the address to access the S-box. The returned 8-bit result is XOR with the original byte to produce the new byte. For parallel access the S-box is duplicated four times.

The second part involves XOR between the zeroth byte with the round constant. A pointer, which increments every clock cycle, is used as an address to access the 30-entry round constant table for the round constant.

The third part involves 256-bit data is again broken down into four "rows" of 64 bits each. Each "row" contains eight bytes, and each byte is XORed with the previous byte in a sequential manner. Since the datapath is slightly different for Key Length of 256 bits, a MUX is used for the selection of the fourth byte and is controlled by the key length.

Key Alignment

Since the Rijndael algorithm allows different key lengths and block lengths, each subkey is carefully set to have the same length as the data do. From the specification of the algorithm, the original key is used to generate a sequence of the entire sub-key stream, and chunks of sub-keys are selected for the encryption module according to the block length. This algorithm works if we have a buffer storage in our design to store the whole sub-key sequence, but is not applicable to our implementation.In the case of 128-128 (block-key), 192-192, and 256-256 the generated sub-keyscould be fed into the encryption module directly with any reorganization .

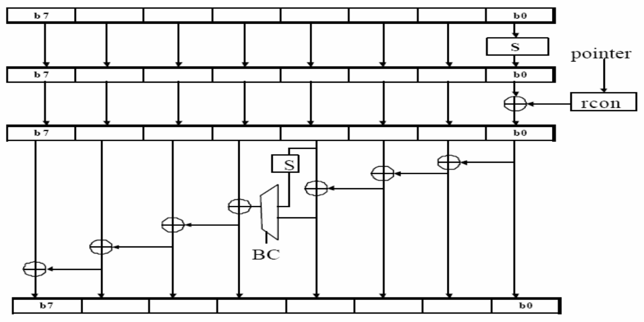


Figure 2.1 Block diagram for Key Scheduling (only one of the four 8-byte "rows" is shown)

However, in the case of 256-128, since both the encryption and key-scheduling modules are sharing the same clock, it means that the key-scheduling module has to create two set of 128-bit sub-keys to combined for the 256-bit sub-key for the encryption module. On the other hand, in the case of 192-128, the original 128-bit keys are used for the lower 128 bits of the sub-key fed to the encryption module. Then the 128-bit key goes through the key-scheduling module to generate the next set of 128-bit sub-key. The lower half of this key is used as the upper 64 bits of the first sub-key fed into the encryption module, and the upper half is used for the next sub-key . In this case we will sometimes need to access the next sub-key and sometimes the previous sub-keys.

III. CONCLUSION

It is evident that the Rijndael's S-Boxes are the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit×8-bit look-up-table, requiring more hardware resources. However, the remaining components of the Rijndael round function– byte swapping, constant multiplication by an element of Galois Field, and key addition were found to be simpler in structure, resulting in these elements of the round function requiring fewer hardware resources.

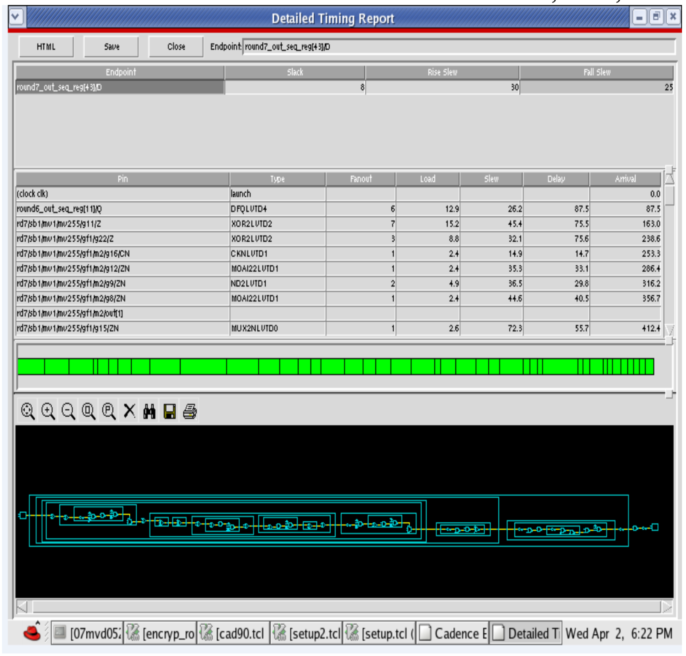
Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA. Partially pipelined implementation with one sub-pipeline stage provided one area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. The 2-stage loop unrolling was found to yield the highest throughput when operating in FeedBack (FB) mode.

Since the design is based on one clock cycle for each encryption round, the memory modules had to be duplicated. For example, in the ByteSub, the S-boxes need to be duplicated 16 times. Consequently, the choice of memory architecture is very critical. Since all the table entries are fixed and defined in the standard, the usage of ROM is preferred. Specifically, the architecture requires several small ROM modules instead of one large module, since each lookup will only be based on a maximum of 8-bit address, which translates to 256 entries.

Here the implemented multiplicative inverse function using the look-up table of size 8×256. We have a total of 20 copies of the S-boxes in our design; 16 of them in encryption module and 4 in the key scheduling module.

IV. DESIGN FLOW

The proposed architecture was implemented using the CADENCE virtuoso layout design tool. The method adopted was a custom designed at the transistor level based on a custom cell library of 0.35µ CMOS primitive standard cells. A hierarchical approach was followed in the implementation. The layout for each module was generated and later integrated to obtain the final chip. The generated netlist was then simulated with HSPICE using the MOSIS CMOS model. Once the creation of layout design is finished, the I/O pins have to be added to the circuit. The design layouts of different architectural units are shown in the Figure 4.1.



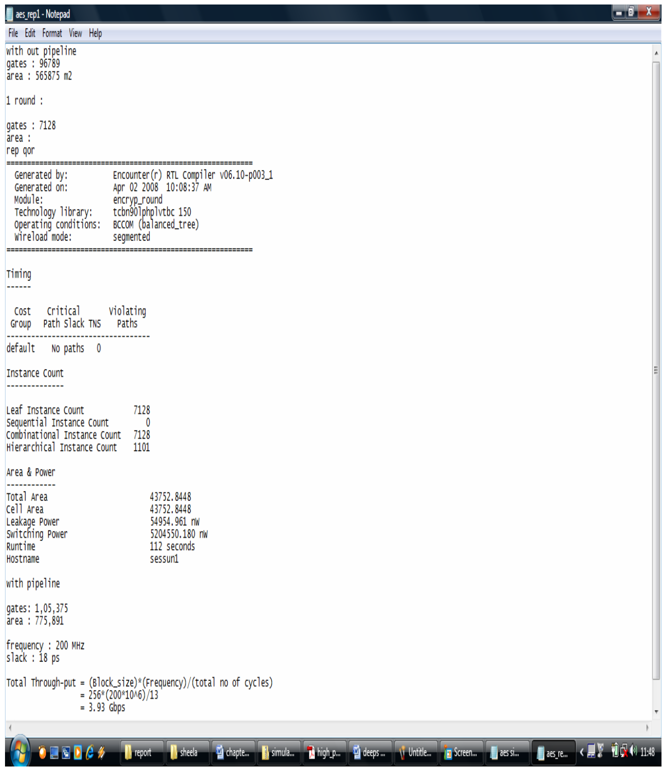
Endpoint	Slack	Rise Slew	Fall Slew
round7_out_seq_reg[43]/D	8	30	25

Pin	Type	Fanout	Load	Slew	Delay	Arrival
(clock clk)	launch					0.0
round6_out_seq_reg[11]/Q	DFQLVTD4	6	12.9	26.2	87.5	87.5
rd7/sb1/mv1/mv255/g11/Z	XOR2LVTD2	7	15.2	45.4	75.5	163.0
rd7/sb1/mv1/mv255/gf1/g22/Z	XOR2LVTD2	3	8.8	32.1	75.6	238.6
rd7/sb1/mv1/mv255/gf1/m2/g16/CN	CKNLVTD1	1	2.4	14.9	14.7	253.3
rd7/sb1/mv1/mv255/gf1/m2/g12/ZN	MOAI22LVTD1	1	2.4	35.3	33.1	286.4
rd7/sb1/mv1/mv255/gf1/m2/g9/ZN	ND2LVTD1	2	4.9	36.5	29.8	316.2
rd7/sb1/mv1/mv255/gf1/m2/g8/ZN	MOAI22LVTD1	1	2.4	44.6	40.5	356.7
rd7/sb1/mv1/mv255/gf1/m2/out[1]						
rd7/sb1/mv1/mv255/gf1/g15/ZN	MUX2NLVTD0	1	2.6	72.3	55.7	412.4

Figure 4.1 Screenshot showing detailed timing report.

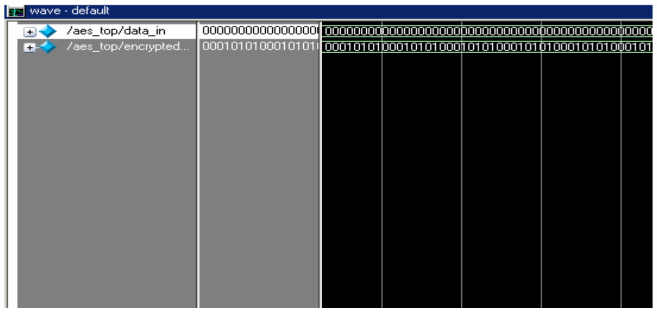
Synthesis results of AES Rijndael Algorithm

The results of a configuration that was used during simulation/synthesis iterations.



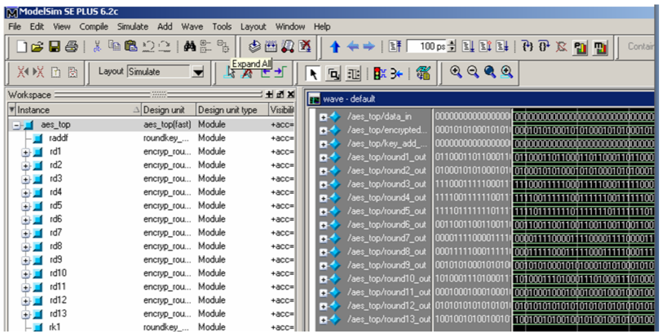
```
with out pipeline
gates : 96789
area : 565875 m2

1 round :

gates : 7128
area :
rep qor
==============================================================
  Generated by:          Encounter(r) RTL Compiler v06.10-p003_1
  Generated on:          Apr 02 2008  10:08:37 AM
  Module:                encryp_round
  Technology library:    tcbn90lphplvtbc 150
  Operating conditions:  BCCOM (balanced_tree)
  Wireload mode:         segmented
==============================================================


Timing
------


  Cost    Critical       Violating
 Group   Path Slack TNS    Paths
------------------------------------
default    No paths   0

Instance Count
--------------


Leaf Instance Count          7128
Sequential Instance Count       0
Combinational Instance Count 7128
Hierarchical Instance Count  1101

Area & Power
------------
Total Area                      43752.8448
Cell Area                       43752.8448
Leakage Power                   54954.961 nW
Switching Power                 5204550.180 nW
Runtime                         112 seconds
Hostname                        sessun1

with pipeline

gates: 1,05,375
area : 775,891

frequency : 200 MHz
slack : 18 ps

Total Through-put = (Block_size)*(Frequency)/(total no of cycles)
                  = 256*(200*10^6)/13
                  = 3.93 Gbps
```

Figure 4.2 Synthesized results



```
/aes_top/data_in       0000000000000000  00000000000000000000000000000000000000000000000
/aes_top/encrypted...  0001010100010101  0001010100010101000101010001010100010101000101
```

Figure 4.3 Encrypted output



```
Instance     Design unit     Design unit type  Visibili
aes_top      aes_top(fast)   Module            +acc=
  raddf      roundkey_...    Module            +acc=
  rd1        encryp_rou...   Module            +acc=
  rd2        encryp_rou...   Module            +acc=
  rd3        encryp_rou...   Module            +acc=
  rd4        encryp_rou...   Module            +acc=
  rd5        encryp_rou...   Module            +acc=
  rd6        encryp_rou...   Module            +acc=
  rd7        encryp_rou...   Module            +acc=
  rd8        encryp_rou...   Module            +acc=
  rd9        encryp_rou...   Module            +acc=
  rd10       encryp_rou...   Module            +acc=
  rd11       encryp_rou...   Module            +acc=
  rd12       encryp_rou...   Module            +acc=
  rd13       encryp_rou...   Module            +acc=
  rk1        roundkey_...    Module            +acc=
```

```
/aes_top/data_in       0000000000000000  00000000000000000000000000000000000
/aes_top/encrypted...  0001010100010101  00010101000101010001010100010101
/aes_top/key_add_...   0000000000000000  00000000000000000000000000000000000
/aes_top/round1_out    0110001101100011  011000110110001101100011011000
/aes_top/round2_out    0100010101000101  0100010101000101010100010101000
/aes_top/round3_out    1110001111100011  1110001111100011111000111111000
/aes_top/round4_out    1111001111110011  1111001111110011111100111111100
/aes_top/round5_out    1111011111110111  1111011111110111111101111111101
/aes_top/round6_out    0011001100110011  0011001100110011001100110011001100
/aes_top/round7_out    0000111100001111  000011110000111100001111000011
/aes_top/round8_out    0001111100011111  000111110001111100011111000111
/aes_top/round9_out    0010101000101010  00101010001010100010101000101010
/aes_top/round10_out   1010001110100011  101000111010001110100011101000
/aes_top/round11_out   0001000100010001  00010001000100010001000100010001000100
/aes_top/round12_out   0101010101010101  0101010101010101010101010101010101
/aes_top/round13_out   1001001010010010  1001001010010010100100101001001001
```

Figure 4.4 Round key with encrypted output waveform

REFERENCES

[1] M.B. Abdelhalim*, H. K. Aslan**, A. Mahmoud** and H. Farouk**, “A Design For An Fpga Implementation Of Rijndael Cipher”, *ICGSTPDCS Journal*, Volume 9, Issue 1, October 2009.

[2] Daemen J., and Rijmen V., “AES proposal: Rijndael-The Rijndael Block Cipher”, A technical Report Version Presented to the National Institute of Standards and Technology (NIST), 1999.

[3] Edwin NC Mui Custom R & D Engineer, “Practical Implementation of Rijndael SBox Using Combinational Logic”.

[4] Daemen J., and Rijmen V. "The Design of Rijndael: *AES-the Advanced Encryption Standard*". Springer-Verilog., 2002.

[5] Marko Mali, Franc Novak, Anton Biasizzo, “Hardware Implementation Of Aes Algorithm”, *Journal of Electrical Engineering*, VOL. 56, NO. 9-10, 2005, 265–269.

[6] G. Rouvroy, F. Standaert, J. Quisq uater and J. Legat, "Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael VeryWell Suited for Small Embedded Applications", *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’04), IEEE 2004*.

[7] M. McLoone , J.V McCanny:” High Performance Single-Chip FPGA Rijndael Algorithm Implementations”, *CHES 2001*, pp. 65-76.

[8] Pravin B. Ghewari, Mrs. Jaymala K. Patil, Amit B. Chougule *International Journal of Engineering Science and Technology* Vol. 2(3), 2010, 213-219.

[9] W. Diffic and M. Hellman, “Privacy and Authentication: An Introduction to Cryptography”, *Proceedings of IEEE*, 67 (1979), pp. 397 -427.

[10] I. Verbauwhede, F. Hoornaert, H. De Man, and J. Vandewalle, “ASIC Cryptographical Processor Based on DES”, *Proceedings of EURO -ASIC-91*, Paris,