A Low Cost Advanced Encryption Standard (AES) Co-Processor Implementation

Orlando J. Hernandez, Thomas Sodon, Michael Adel, and Nathan Kupp
E-mail: hernande@tcnj.edu
Department Electrical and Computer Engineering, The College of New Jersey
Ewing, New Jersey 08628-0718, USA

ABSTRACT

The need for privacy has become a major priority for both governments and civilians desiring protection from signal interception. Widespread use of personal communications devices has only increased demand for a level of security on previously insecure communications. This paper presents a novel low-cost architecture for the Advanced Encryption Standard (AES) algorithm utilizing a field programmable gate array (FPGA). In as much as possible, this architecture uses a bit-serial approach, and it is also suitable for VLSI implementations. In this implementation, the primary objective was not to increase throughput or decrease latency, but to balance these factors in order to lower the cost. A focus on low cost resulted in a design well-suited for SoC implementations. This allows for scaling of the architecture towards vulnerable portable and cost-sensitive communications devices in consumer and military applications.

Keywords: AES; Cryptographic Architectures; FPGA Design; Specialized Architectures; VLSI Design

1. INTRODUCTION

The Advanced Encryption Standard (AES) was officially named the successor to the Data Encryption Standard (DES) in 2001. In 1997, the National Institute of Standards and Technology promoted worldwide research into a replacement for DES, in response to the discovery of theoretical weaknesses in the encryption of DES as well as successful brute force attacks carried out against the algorithm. After four years of research and testing, the Rijndael algorithm was selected from a set of 15 candidates on the merits of its reliability and speed in encryption and decryption, key and algorithm expansion time and resistance to attacks. In December of 2001, the Federal Information Processing Standard document FIPS-197 [1] was released, specifying that all sensitive and unclassified government documents will use AES for data encryption.

Both DES and AES are defined as symmetric key block ciphers, with the primary distinction being the length of the key (56 bit for DES, in contrast to the 128, 192, and 256 bit modes of AES). These symmetric-key encryption schemes use the same key for both the sender and receiver, and as a result eliminate the need for the verification server needed in public keying. Symmetric keying lends itself well to working independently of an open network and in turn a higher level of system interoperability.

Since the decommissioning in 2001 of DES and the approval of AES as its successor, various AES implementations have been proposed both in software and hardware. This paper presents a low-cost and AES hardware architecture. By incorporating most of the AES algorithm complexity into a controller, components are reused and efficiency is increased. A Verilog® hardware implementation in an FPGA is presented, allowing for easy migration to an ASIC implementation in an SoC overall architecture.

2. THE AES ALGORITHM

The AES encryption and decryption processes for a 128-bit plain text block are shown in Fig. 1. The AES algorithm specifies three encryption modes: 128-bit, 192-bit, and 256-bit. Each cipher mode has a corresponding number of rounds Nr based on key length of Nk words. The state block size, termed Nb, is constant for all encryption modes. This 128-bit block is termed the state. Each state is comprised of 4 words. A word is subsequently defined as 4 bytes. Table 1 shows the possible key/state block/round combinations.

Both encryption and decryption begin with the round key expansion created by the key schedule function. Using the RCON values in combination with a series of XOR, SubBytes, and RotWord (rotate word) operations, an expanded round key is generated with a size of $(N_r+1)\cdot N_b$ bytes. For the 256-bit key expansion, the SubBytes operation is reapplied 4 words after each use of the RCON.

The AES algorithm consists of the sequential execution of the four operations SubBytes, ShiftRows, MixColumns, and AddRoundKey Nr times in a loop (these four operations in sequence constitute a round). For encryption, Nr is initialized to 10, 12 or 14 rounds, corresponding to the 128, 192, and 256 bit key lengths, respectively. The four operations of the AES algorithm are then executed Nr-1 times resulting in a 128-bit block of cipher text. For decryption, the same process occurs simply in reverse order – taking the 128-bit block of cipher text and converting it to plaintext by the application of the inverse of the four operations. AddRoundKey is the same for both encryption and decryption. However the three other functions have inverses used in the decryption process: Inverse SubBytes, Inverse ShiftRows, and Inverse MixColumns.

SubBytes is a nonlinear transformation in which one byte is substituted for another by means of the affine transformation over the Galois Field GF(2)

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (1)$$

ShiftRows is a shift operation performed on the last three rows of the state. The last three rows are rotated to the left by 1, 2, or 3 bytes, as seen in Fig. 1. MixColumns is finite field matrix multiplication applied every round except the

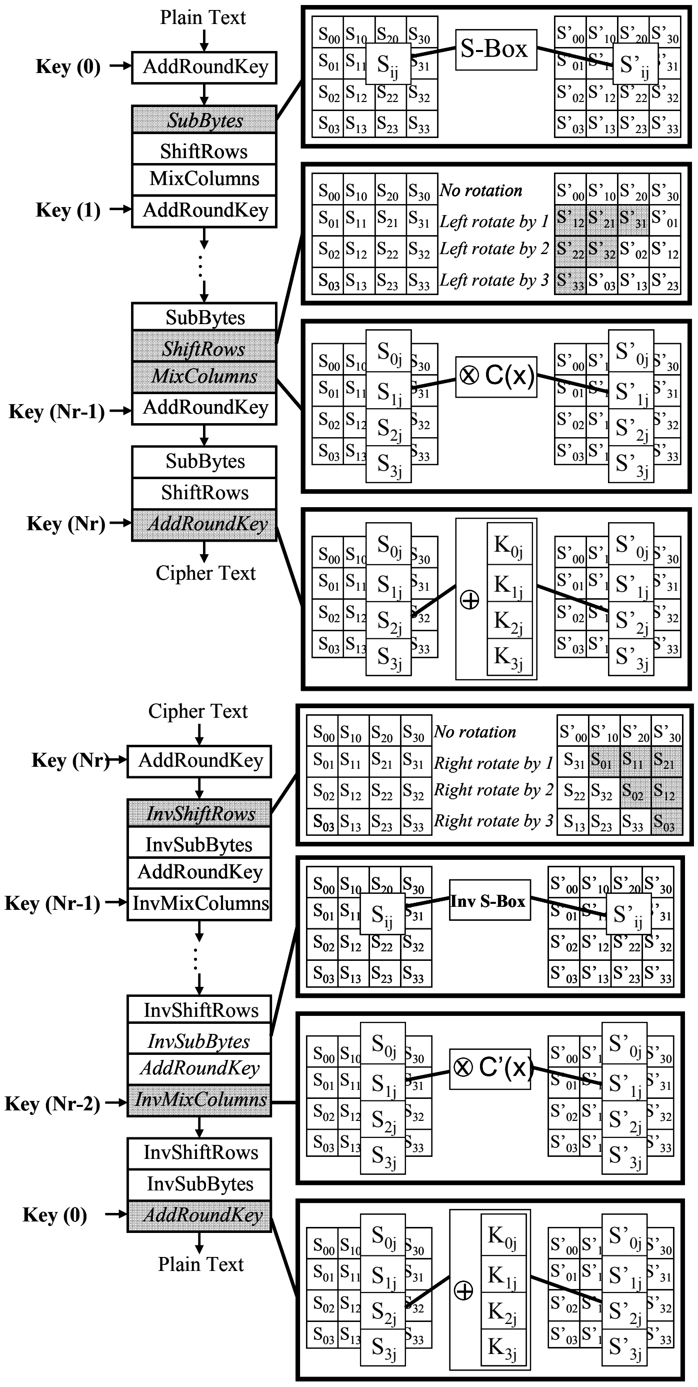


Fig. 1. AES encryption and decryption.

Table 1. AES Bit-Mode specifications.

Bit Mode	**Key Length**	**Block Size**	**Number of Rounds**
	(N_k words)	(N_b words)	(N_r)
128	4	4	10
192	6	4	12
256	8	4	14

last. Each column is multiplied as a four-term polynomial in GF (2^8) mod ($x^4 + 1$) using the array

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < N_b . \tag{2}$$

AddRoundKey performs a bitwise XOR operation with the current state and the expanded round key every round including an initial round and the last round. The round key is read from round 0 to (Nr – 1) for encryption and vice versa for decryption.

The decryption process is similar to the encryption process, simply executing the inverse of each function. Inverse SubBytes involves taking an inverse affine transformation. Inverse ShiftRows rotates the bytes to the right by: 3, 2, or 1 byte(s). Inverse MixColumns uses the same operations as MixColumns but uses the inverse matrix

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} . \tag{3}$$

Because of the inherent symmetry of addition modulo-2 (XOR), AddRoundKey is the same for both encryption and decryption processes.

3. RECENT RELATED WORK

Recent AES implementations have focused on speed gains obtained by manipulations in the SubBytes and MixColumns, two of the more time-consuming functions in the algorithm. The work of References [2] and [3] take an approach involving a high level of parallelism, enabling very high throughput to be achieved (29 Gbps in Reference [2] and 21.56 Gbps in [3]), but at a high cost. The use of pipelining and parallelization techniques increases throughput to above 20 Gbps. However, these designs are primarily focused on speed, and cost and power are not a concern. In Reference [2], the power usage exceeds 2 watts, far too high for portable applications where power is a major concern. In as much as possible, this implementation takes a bit-serial approach, resulting in lower throughput but utilizing far fewer gates and limiting cost and power usage.

There has been little to no research done regarding lowering cost and power requirements by de-emphasizing processing speed. Relatively high throughput (2.381 Gbps) for 128-bit key mode was achieved in one FPGA implementation with a cost of only 58.5K gates on a single chip [4]. This was done by introducing a 4-stage pipeline for the main functions and performing a basis transformation on SubBytes. The alternative S-Box design would be to use a ROM/RAM address lookup table (LUT), which is cost-effective. The approaches to S-Box implementation in the literature generally fall into these two categories: this basis transformation – an on-the-fly generation of the S-Box values and look up tables (LUTs).

It is clear that out of the four functions, manipulating SubBytes is the key to increasing performance. However, modifications to the SubBytes and MixColumns functions will often result in increased sensitivity to noise and operating temperature, as well as extremely large fan-outs, adding higher propagation delays. Based on previous implementations, it was decided this design would use a straightforward, simplified approach using LUTs, keeping functions independent of one another.

4. ARCHITECTURE AND RESULTS

The AES algorithm specification FIPS-197 [1] was followed, while minimizing redundancy. Processes shared between the encryption and decryption processes were reused as often as possible. It was observed that the MixColumns and SubBytes functions took up significant processing time. MixColumns specifically required sequential left shifts, each followed by a conditional XOR operation. The condition of the XOR operation depends on the existence of a 1 in the most significant bit of the current byte before it is shifted. If the condition is true, the shifted byte is XOR-ed with byte {1b}, the irreducible GF polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1 . \tag{4}$$

The affine transformation used by SubBytes and Inverse SubBytes can be implemented as a 16x16 lookup table. An attempt at splitting the search process into sixteen smaller comparisons did not significantly increase efficiency. The LUT used a great deal of memory in software, and similarly a large number of gates in the hardware design.

Various methods for reducing GF circuit size exist, such as composite (or tower field inversion), Fermat's little theorem or extended Euclidean algorithms [5]. However, these methods introduce large propagation delays and increased power consumption. The low-cost and low-power approach minimizes the complexity of GF operations by sacrificing speed.

FPGA Implementation

The main components of the data path are the Register File (RF), 8-bit XOR gate, S-Box, Inverse S-Box, Working Register, Round Constant lookup table (RCON LUT), Instruction Module (IM), the Multi-Staged Controller, ModFlag block and the MixColumns accumulator (MCACC). Fig. 2 shows a block diagram of the overall design including all these components.

The IM consists of a sequencer, an output mux, as well as six separate ROMs comprised of each bit-mode and its corresponding encrypt and decrypt functions. The IM is used simply to correctly sequence the encryption and decryption processes for each bit mode. The bit-mode and the crypt-mode are selected by the main processor, or more specifically the user. The user can also choose to clear all the data in the co-processor using the Reset line. It is configured for synchronous reset.

To maximize throughput, the data is handled in parallel upon completion of the state operations. The 8-bit register is the core component of the architecture. It has parallel read/write functionality and the capability to right shift. The Working Register is a derivative of the 8-bit register, and is covered in greater detail later in this section.

The 3-bit BitMode, Enc/Dec and Start lines select the desired ROM to process commands sequentially. The microinstructions are then output to the Controller, RF, RCON LUT and the MixColumns. Much of the work in decreasing the data path can be done within the IM, as a majority of the functions are shared between the bit-modes. While the 128 and 192-bit modes are quite similar, the 256

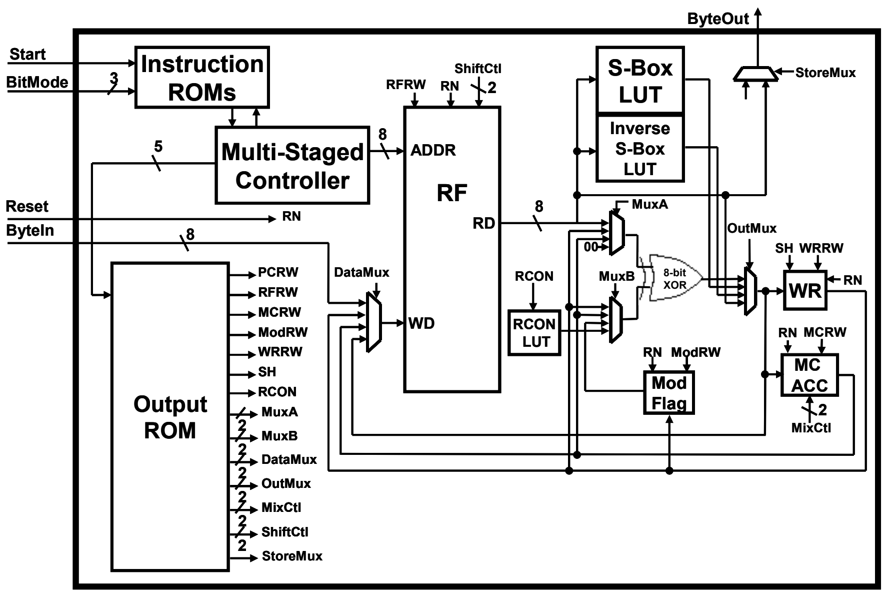
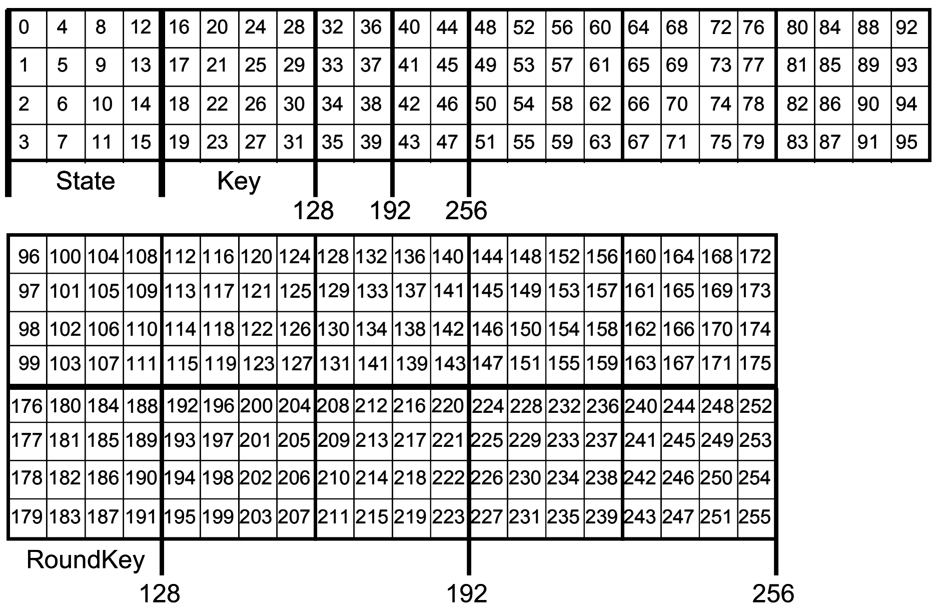


Fig. 2. Overall hardware design.

mode is more drawn out and has a significantly different key expansion. Despite this fact, about 60% of the functions are shared between the bit-modes. Experimentation in consolidating these six ROMs into one module was done. This would require internal addressing within the ROM and a method to compare values. An ALU would be needed for such functionality. An optimum balance could be reached between the gates decreased in the ROM and gates increased with the implementation of the ALU. Another consideration is that this can be done using a Finite State Machine. Further research can be done in regards to this, and is covered in greater detail in the Conclusions section.
The Program Counter (PC) is a register that contains the address of the current instruction used in the cryptographic process. The PC is automatically incremented after each instruction is fetched from the IM to point to the next instruction. It is not manipulated like an ordinary register, as special instructions are used to occasionally alter the program flow. However, for the purposes of minimizing the area of the data path, many of these commands have been excised. Such conditional jump and compare commands would require the use of an ALU, as previously stated, which would increase the gate count.
Assuming the data and key are first loaded by the main processor into the register files, the key expansion process can begin. The RF will receive addresses from the controller allowing individual bytes to be chosen for manipulation. Once the round key is generated, the values are held constant until the main processor assigns a new key. The current state values are contained in the RF and changes after each function call. The RF can be broken down intrinsically into three separate components: the Key, State and RoundKey RFs. Fig. 3 shows a layout of the RF, including these three RFs.

The Controller generates control signals for data transport, key expansion, encryption and decryption. The Controller utilizes combinational logic to generate 19-bit control signals that initiate the different modules of the architecture. An instruction set of 32 commands is used to delegate the signals for the various operations. Fig. 4 shows a block diagram of the Controller. Figs. 5 and 6 show the Controller microinstruction set format and a list of the Controller Instructions, respectively.
The Key RF is designed to hold the first 16, 24 or 32 bytes of the round key as shown in Fig. 2. Similarly, the RoundKey RF is simply a 208 byte extension of the Key RF design. The State RF consists of 4 sets of 4 8-bit shift registers. Each set constitutes a row and is configured so that the ShiftRows and Inverse ShiftRows operation can be performed internally. The data in the State RF is moved from one shift register to the next until the each byte reaches its respective register during either ShiftRows operation.
Three muxes control the output from the RF. They are muxes A, B and OutMux. Mux A selects between RD and the output from the MixColumns Accumulator. Mux B chooses between the MixColumns Accumulator, WR, ModFlag block and the 10-term RCON LUT. The RCON LUT, or Round Constant table, contains AES specified 8-bit values. The output of Muxes A and B are connected to the 8-bit XOR gate.
The OutMux chooses between four different inputs to output based on the 2-bit OutMux signal from the Controller. The OutMux selects between the S-Box, Inverse S-Box, RD and output from the XOR gate. The S-Box and Inverse S-Box LUTs are tied to the RDoutput of the RF and both operate similarly. The LUTs process bytes by combinational logic.



0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91	95

96	100	104	108	112	116	120	124	128	132	136	140	144	148	152	156	160	164	168	172
97	101	105	109	113	117	121	125	129	133	137	141	145	149	153	157	161	165	169	173
98	102	106	110	114	118	122	126	130	134	138	142	146	150	154	158	162	166	170	174
99	103	107	111	115	119	123	127	131	141	139	143	147	151	155	159	163	167	171	175
176	180	184	188	192	196	200	204	208	212	216	220	224	228	232	236	240	244	248	252
177	181	185	189	193	197	201	205	209	213	217	221	225	229	233	237	241	245	249	253
178	182	186	190	194	198	202	206	210	214	218	222	226	230	234	238	242	246	250	254
179	183	187	191	195	199	203	207	211	215	219	223	227	231	235	239	243	247	251	255

Fig. 3. Register file layout.

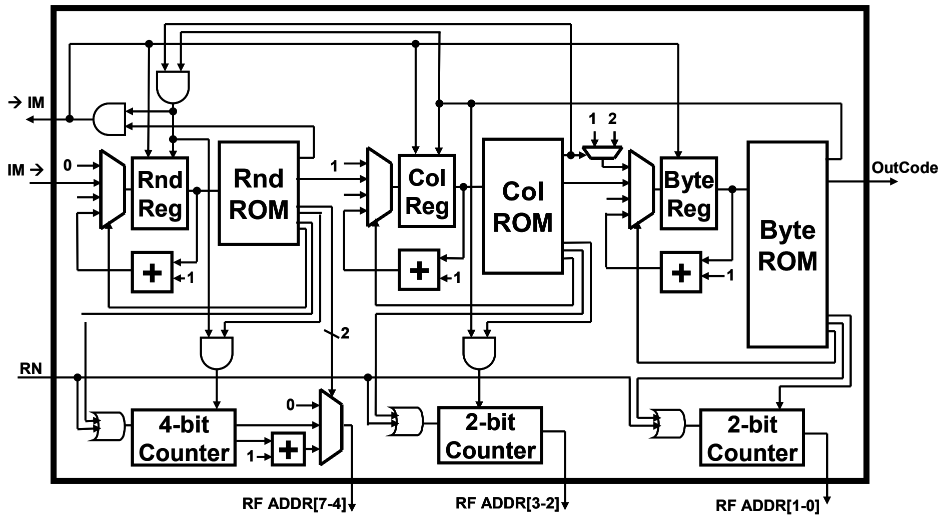


Fig. 4. Multi-Staged controller.

An 8-bit temporary register, called the Working Register or WR, is placed at the output of the OutMux to assist in byte operations during the key expansion and MixColumns. The Working Register operates in tandem with the 4-byte MixCols Accumulator to compute both MixColumns functions. The MixCols Accumulator is simply a temporary register comprised of four 8-bit registers with specific register select capability and without shifting.

The controller reads flags from the ModFlag block and Instruction Memory (IM). The ModFlag block signals the controller as to the status of the most significant bit in the current output byte for use in the conditional XOR during MixColumns. The byte values {1b} and {00} are output from the block depending on the value in the register. The IM contains a listing of all the possible commands the co-processor will respond to when referenced by an address line.

To test the Verilog design in actual hardware, the Spartan-3 XC3S200 FPGA was chosen for prototyping. This FPGA board utilizes 200K+ gates, 1 MB SRAM and 80 dedicated I/O ports. The amount of I/O ports and number of gates suited the requirements for the design.

5. CONCLUSIONS

IM Functions (Sequencing)

RndCode
6-bit

Rnd Instruction (Round Control)

ColCode	RndCtl	RndCntCtl	RndCntIncr	RndCntRN	RndDone
6-bit	2-bit	1-bit	1-bit	1-bit	1-bit

Col Instruction (Column Control)

ByteAddress	ColCtl	ColCntIncr	ColCntRN	ColDone
8-bit	2-bit	1-bit	1-bit	1-bit

Byte Instruction (Byte Control)

OutCode	ByteCtl	ByteCntIncr	ByteCntRN	ByteDone
5-bit	2-bit	1-bit	1-bit	1-bit

Fig. 5. Microinstruction command format.

Output ROM	RN	RFRW	MCRW	ModRW	WRRW	SH	RCON	MuxA[1]	MuxA[0]	MuxB[1]	MuxB[0]	DataMux[1]	DataMux[0]	OutMux[1]	OutMux[0]	MixCtl[1]	MixCtl[0]	ShiftCtl[1]	ShiftCtl[0]	StoreMux
NoOp	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LB	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
LWR	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
SHROW1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
SHROW2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
SHROW3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
XORRCONMC0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0
SBX	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0
ISBX	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
MODSH	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
XORMOD	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
LMC0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
LMC1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
LMC2	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
LMC3	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
SMC0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SMC1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
SMC2	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
SMC3	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
SBXMC0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
SBXMC1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
SBXMC2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
SBXMC3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
XORWRMC0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
XORWRMC1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0
XORWRMC2	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0
XORWRMC3	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	1	0	0	0
Loop	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6. Controller instruction set.

In this paper we presented a low-cost AES co-processor hardware architecture. Tables 2 and 3 show the results for the FPGA synthesis of the architecture. This is shown for different Xilinx FPGA families and different sets of constraints. Table 4 displays a comparison between this design and others in recent literature from lowest to highest Throughput/Slice in Megabits/Seconds/Slice. Our design accomplishes a throughput/slice ratio near the best in the literature [2]. This is despite maintaining significantly lower utilization of CLB slices than the work in [2]. and others. In our analysis, the critical path in the hardware implementation was determined to be the Inverse MixColumns function. Several design modifications have been evaluated: speed of XOR operation in serial versus 8-bit parallel, modified SubBytes design, and a reduced MixColumns algorithm. By focusing on minimization of redundancies within these functions, cost can be reduced while maintaining a suitable operating speed.

A possible improvement would be to further minimize the IM. A great deal of the encryption/decryption process is repetitious, especially between the 128 and 192-bit modes. An estimated 60% of the cryptographic processes are shared between the modes. The remaining 40% could be more efficiently distributed using if and unrolled for loop structures. Prototype IMs utilizing such functionality have been written, however to fully remove any repetition and maximize reusability would require extrapolating each

Table 2. Synthesis results – optimized for maximum speed.

	Maximum Speed			
	Clock	**Slices**	**Slice Flip Flops**	**4 Input LUTs**
Spartan 3:	**65.984 MHz**	**1985**	**1654**	**3670**
Spartan 2E:	32.351 MHz	2121	1665	4011
Spartan 2:	30.323 MHz	2172	1676	4115
Virtex E:	31.382 MHz	2126	1664	4011

Table 3. Synthesis results – optimized for minimum area.

	Min Area			
	Clock	**Slices**	**Slice Flip Flops**	**4 Input LUTs**
Spartan 3:	**56.855 MHz**	**1880**	**1571**	**3454**
Spartan 2E:	23.715 MHz	1933	1588	3614
Spartan 2:	18.604 MHz	1933	1588	3614
Virtex E:	23.120 MHz	1933	1588	3614

Table 4. Comparison of recent designs.

Recent Work Comparison			
Source	**CLB Slices**	**Throughput (Megabits/Sec)**	**Throughput/Slice**
[6]	5302	300	0.057
[3]	10992	1938	0.176
[7]	1125	215	0.191
[8]	2419/5068	601/1050	0.248/0.207
[9]	2358/17314	259/3650	0.110/0.211
[10]	11022	21560	1.956
[11]	163/146	208/358	1.276/2.452
This work	**1880/1985**	**7277/8446**	**3.871/4.255**
[12]	2457	12000	4.884
[2]	5408	29770	5.505

cryptographic process for every bit-mode and comparing them. This is the subject of our continued research.

6. REFERENCES

[1] National Institute of Standards and Technology (US), Advanced Encryption Standard, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
[2] S.-M. Yoo, D. Kotturi, D.W. Pan and J. Blizzard, An AES crypto chip using a high-speed parallel pipelined architecture, Microprocessors and Microsystems 29 (2005) 317-326.
[3] X. Zhang, and K. Parhi, High-speed VLSI architectures for the AES algorithm, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12 (2004) 957-967.
[4] C-P. Su, T.-F. Lin, C.-T. Huang and C.-W. Wu, A high throughput low-cost AES processor, IEEE Communications 41 (2003) 86-91.
[5] V. Fischer and M. Drutarovsky, Two methods of Rijndael implementation in reconfigurable hardware, Proc. CHES, Vol. 2162, France, 2001, pp.81-96.
[6] E. Mang, I. Mang and C. Popescu, AES Candidate Algorithm Finalists: FPGA implementation and performance evaluation, Proc. of the IASTED International Conference, 2003, pp. 147-152.
[7] N. Pramstaller, S. Mangard, S. Dominikus and J. Wolkerstorfer, Efficient AES implementations on ASICs and FPGAs, Lecture Notes in Computer Science 3373 (2005) 98-112.
[8] L. Chang-Shu, P. Gen-Peng and W. Xio-Zhuo, Two methods of AES implementation based on CPLD/FPGA, Transactions of Tianjin University 10 (2004) 285-290.
[9] A. J. Elbirt, W. Yip, B. Chetwynd and C. Paar, An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9 (2001) 545-557.
[10] N. Sklavos and O. Koufopavlou, Architectures and VLSI implementations of the AES - Proposal Rijndael, IEEE Transactions on Computers 51 (2002) 1454-1459.
[11] 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, Proc. of the International Conference on Information Technology: Coding Computing - ITCC, Vol. 2, USA, 2004, pp. 583-587.
[12] M. McLoone and J. McCanny, Rijndael FPGA implementations utilizing look-up tables, Journal of VLSI Signal Processing, 34 (2003) 261-275.

Received: Sep. 2007. Accepted: Oct. 2007.