

A 2.3Gb/s Fully Integrated and Synthesizable AES Rijndael Core

Nam Sung Kim, Trevor Mudge, Richard Brown
 Department of Electrical Engineering and Computer Science
 University of Michigan
 Ann Arbor, MI 48019, U.S.A.

Abstract

The growth of the Internet as a vehicle for secure communication and electronic commerce has brought cryptographic processing performance to the forefront of high throughput system design. This trend will gain further momentum with the widespread adoption of secure protocols such as secure IP (IPSEC) and virtual private networks (VPNs). In this paper, we present a fully integrated and synthesizable cipher core supporting the Advanced Encryption Standard — Rijndael. We designed and fabricated the fully integrated core — key scheduler, encipher, and decipher using TSMC 0.18 μ m technology. The core operating frequency is 465MHz and throughput is 2.3Gb/s.

I. Introduction

The widespread adoption of the Internet as a trusted medium for communication and commerce has made cryptography an essential component of modern information systems. The trend towards virtual private networks (VPNs) [1] and secure IP (IPSEC) [2] will further emphasize the significance of cryptographic processing among all types of communication. As demand for secure communication bandwidth continues to grow, efficient cryptographic processing will become increasingly critical to good system performance. In contrast to earlier efforts [4][5][6], the chip presented here is a complete fabricated encipher/decipher system with key scheduler.

II. Background

A sequence of four primitive functions, *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*, are executed $N_r - 1$ times. Each loop is called a *round* and the number of iterations of a loop, N_r , can be 10, 12, or 14 depending on the key and plain text lengths. Prior to this main loop, *AddRoundKey* is executed for initialization. After executing the main loop, a sequence of *SubBytes*, *ShiftRows*, and *AddRoundKey* is executed as the final round.

SubBytes can be a 16, 24, or 32-byte (128, 196, or 256-bit) input and output nonlinear transformation that uses one-byte substitution tables (S-Boxes). Each S-Box is a multiplicative inversion on a *Galois field* $GF(2^8)$ followed by an affine transformation. The irreducible polynomial used by the field is

$$m(x) = x^8 + x^4 + x^3 + x + 1 \quad (\text{EQ 1})$$

ShiftRows is a cyclic shift operation in each row of the 4 \times 4-byte plain text. The shift amount is 0 bytes for the first row, 1 for the second, and so on. *MixColumns* treats the 4-byte text in each column as coefficients of a 4-term polynomial, and multiplies the text modulo $x^4 + 1$ with the fixed polynomial given by

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{01\} \quad (\text{EQ 2})$$

AddRoundKey is a simple bit-wise XOR operation on the 128, 196, or 256-bit round keys, $K_0 \sim K_{N_r}$, and the text. In the decryption process, the inverse operations of each primitive function are executed. The inverse of *AddRoundKey* is *AddRoundKey* itself. *InvSubBytes*, which is the inverse of *SubBytes*, executes an affine transformation before the multiplicative inversion. *InvShiftRows* is a cyclic rotation in the reverse direction. *InvMixColumns* uses the following polynomial for the multiplications:

$$c(x)^{-1} = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \quad (\text{EQ 3})$$

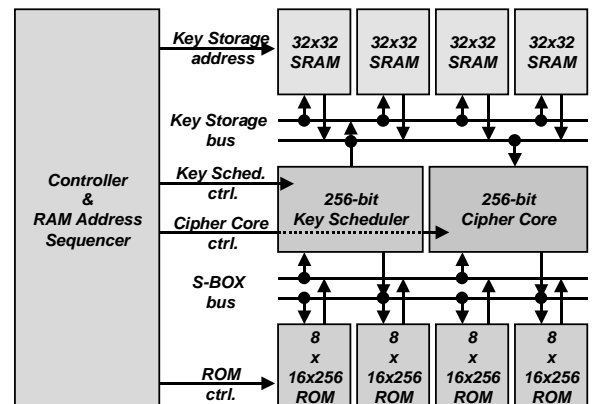
The detailed Rijndael algorithm is described in [3].

III. Cipher System Architecture

One of the main goals of this work is to build a high-performance fully integrated and synthesizable Rijndael core as a piece of soft *intellectual property* (IP). In this section, we present a Rijndael cipher system architecture and discuss the design trade-offs among the various design choices to determine the architectures of the entire cipher system and functional units.

Fig. 1 illustrates the proposed Rijndael cipher system consisting of a *key scheduler*, *cipher core*, and *system controller*. Two datapaths, the key scheduler and the cipher core share key storage, which is implemented as a *static random access memory* (SRAM) array — the *round key cache*. The key scheduler datapath generates and stores expanded keys in the round key cache through the key storage bus. The cryptographic operations such as *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* can be performed in parallel with the key expansion process. Some previously proposed Rijndael processors do not have key scheduler [4]. Instead, they receive the expanded keys from external hardware or use pre-generated ones transmitted with the cipher or plain

Fig. 1: The proposed Rijndael Cipher System Architecture.



texts. For the completeness, however, we implemented the key scheduler. Also, we implemented the round key cache with an 128-bit wide memory I/O bus to provide one 128-bit round key for the cipher core datapath in one memory access cycle that equals to two system clock cycles.

In general cryptographic communications such as SSL, a series of texts are exchanged until the communication ends. A new public key is then exchanged at the start of the next communication. Therefore, we do not need to expand the round keys whenever we encrypt or decrypt the texts. Instead, we can use the round keys stored in the round key cache as long as the key is unchanged. From a system power perspective, fetching the expanded key from the proposed round cache would be more efficient than expanding the same round key every time we encrypt or decrypt texts although we can generate the necessary keys on-the-fly during the encryption and decryption rounds.

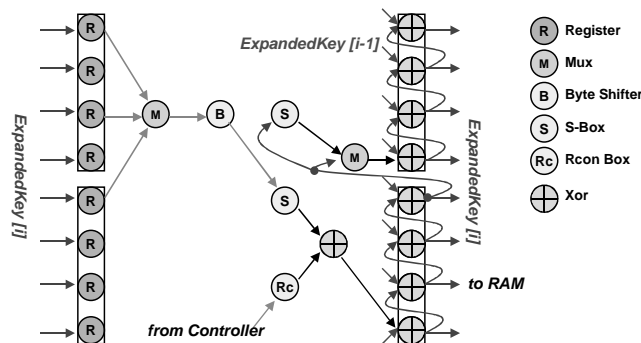
The cipher core datapath fetches pertinent round keys from the round key cache during the cryptographic round operations, and it substitute necessary bytes with ones in the S-BOX ROM array during the cryptographic operations. To support full 32-byte (256-bit) substitution in a cycle, we need 32 S-BOX's. The system controller contains two *finite state machines* (FSM's). One generates necessary control signals for two datapaths. The other generates pertinent memory addresses and control signals for both the round key cache SRAM and S-BOX ROM arrays.

IV. Key Scheduler and Round Key Cache

Fig. 2 shows the implementation of the key scheduler datapath. The cipher datapath supports all AES cipher key lengths including of 128, 196, and 256 bits. Each circle in Fig. 2 represents a 32-bit datapath component. Before starting key expansion, a current cipher key is stored in the registers of the key scheduler datapath shown in the left side of the Fig. 2. The cipher key is the first expanded key itself. The first expanded key is sent to the round key caches, and it is used to generate the next expanded key.

During the rest of key expansion process, ByteShift, SubByte, bit-wise XOR with *round constant* values [3] are applied to the highest 32-bit of the previous expanded key, called the *intermediate key*. The bit-wise XOR with the intermediate key is also applied to the lowest 32 bits of the previous expanded key to generate the lowest 32-bit current expand key as represented in the middle of Fig. 2.

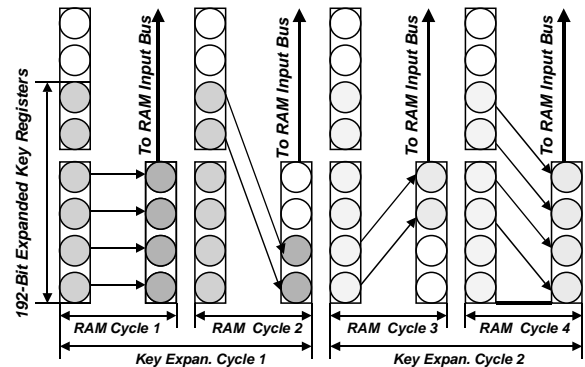
Fig. 2: The implementation of key scheduler datapath.



The lowest 32-bit current key is fed to the XOR operator one of whose operands is provided from the next 32-bit previous expanded to generate next 32-bit key. These operations propagate up through XOR operation chains until they reach the highest 32-bit expanded key word shown at the right of Fig. 2, which illustrates an iteration of the expanded key generation. We use a 32-bit 3-to-1 multiplexer to select an appropriate upper 32 bits of the previous expanded key, because the position of the upper 32 bits of the expanded key can be varied for each cipher key length.

Fig. 3 illustrates the key scheduler timing when storing two 196-bit length expanded keys. It is simple to store the generated expanded keys for the 128 and 256-bit length cipher key, because the round key cache bandwidth is 128-bit wide: it takes 1 and 2 memory cycles respectively. However, we need a different key storage procedure for the 196-bit expanded key. We store the lower 128-bit and upper 64-bit expanded keys to the round key cache in the first and the second SRAM access cycles respectively. In the third and fourth SRAM access cycles, we store the lower 64-bit and upper 128-bit expanded keys respectively. However, we can hide key storage cycles, because we can store expanded keys during the encryption or decryption process.

Fig. 3: Two 196-bit expanded key storage timing.



V. Cipher Core Datapath

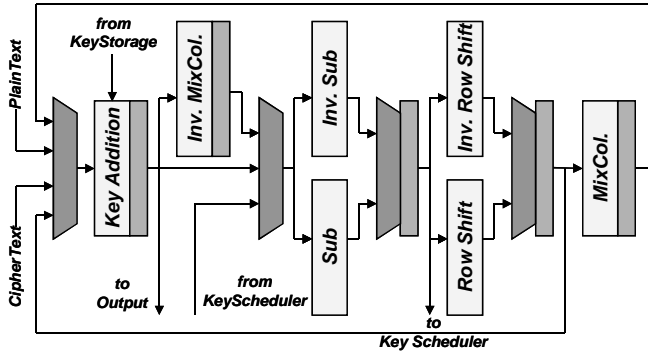
The cipher core datapath consists of 256-bit wide SubBytes, ShiftRows, MixColumns, and AddRoundKey blocks and each block includes its own inverse cipher operation except for KeyAddition block, because the inverse operation of XOR is XOR itself as mentioned in Section II.

The register is implemented with edge-triggered D flip/flops. To control the State flow, we need multiplexers between blocks, because the State should undergo different sub-operations depending on the current number of rounds or the mode of the cryptographic operation — encryption or decryption.

Fig. 4 presents the block diagram of the cipher core datapath. Between the cryptographic sub-operation such as Key-Addition and SubByte, we insert a 256-bit register to latch the results of each sub-operation. This pipeline architecture increases the operating frequency as well as hardware utilization efficiency of the cipher core when we implement both encipher and decipher together. Also, we may increase the throughput of the cipher with the pipeline structure in *electronic cook book* (ECB) mode, but we assume that the basic

cipher mode is *cipher block chaining* (CBC) where pipelined operations cannot be applied.

Fig. 4: The block diagram of the cipher core datapath.

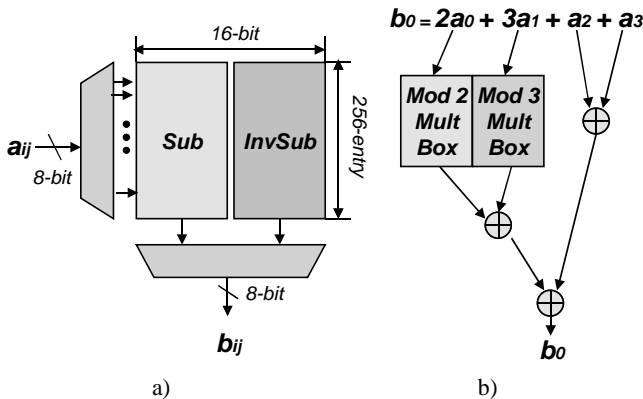


A. SubByte and MixColumn

To maximize the performance of SubByte, we should be able to substitute a 32-byte (256-bit) state in one memory cycle. This requires using 32 8-bit ROM's for each SubByte and InvSubByte. To minimize the number of the control signals and interconnection complexity, we introduce 32 16-bit ROM's. Each SROM can provides both 1-byte SubByte and InvSubByte values, and either or upper or lower 16-bit values are selected accordingly using a 8-bit 2-to-1 mux as illustrated in Fig. 5-a).

The Rijndael algorithm requires that modulo polynomial multiplications be performed for the MixColumn operation on each column consisting of 4 bytes of the state. This is a more complicated type of multiplication compared to a conventional one, and as a result it can be one of the critical paths if a general-purpose modular polynomial multiplier is employed. However, we make use of the fact that one of two operands of the modulo matrix multiplication in the Rijndael algorithm are constant, not variable as shown in Equation 2; the algorithm requires only the three coefficients — 0x01, 0x02 and 0x03 — for the polynomial multiplicand of MixColumn. Furthermore, the modular multiplication result for the coefficient 0x01 is the operand itself. Thus we just imple-

Fig. 5: SubByte and MixColumn logic



mented special-purpose modular polynomial multipliers for the coefficients 0x02 and 0x03. For instance, to implement a special multiplier for the coefficient 0x02, we constructed a truth table by enumerating all possible 256 modulo multiplication results for one byte input — a ROM or PLA is another solution.

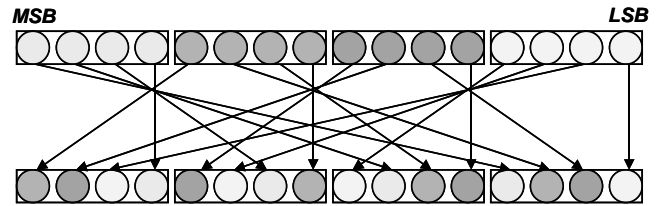
After extracting the truth table, we applied logic minimization techniques to the truth table described with Verilog HDL using the Synopsys Design Compiler™. In InvMixColumn, there are four coefficients for the multiplicands — 0x09, 0x0b, 0x0d, and 0x0e. As a result, we could generate 6 modulo multiplication blocks for each coefficient. To implement the operation shown in Equation 2, we instantiate two modular multiplication blocks for the coefficient 0x02 and 0x03 as shown in Fig. 5-b). Equation 3 can also be implemented in the same way shown in Fig. 5-b). With the special modular multipliers, we were able to implement fast and compact MixColumn and InvMixColumn logic compared to a general-purpose modular multiplier.

B. KeyAddition and RowShift

KeyAddition block can be implemented by 256 bit-wise XOR gates. There are two operands for KeyAddition: one is a round key from the round key cache or key scheduler and the other is the state from plain or cipher text, RowShift, or MixColumn blocks, which is selected according to the current number of the cryptographic round and the mode of the operation as shown in Fig. 4.

It is straightforward to implement a RowShift block in 2-dimensional array representation of the state in software as specified in [3], but it is complicated in a 1-dimensional array representation in hardware. In order to implement RowShift, we translated the 2-dimensional matrix indices to 1-dimensional ones, and traced the coordinate changes as they would be in a 2-dimensional matrix. Fig. 6 shows a RowShift operation for the 1-dimensional array. Each circle represents a byte and one rectangular represents a column composed of 4 bytes.

Fig. 6: RowShift operation.



VI. System Controller and External I/O

The main FSM for the system controller represents the global state of the cryptographic processor. There are five global states, *RESET*, *KEYEXPAND*, *KEYAVAILABLE*, *CRYPTOROUND*, and *fSTATEAVAILABLE*. When the system is powered up the reset signal is asserted, the main FSM enter *RESET* state. As soon as the controller receives the signal that notifies that the cipher key load has completed, the main FSM enters the *KEYEXPAND* state and transits to *KEYAVAILABLE* state. After the FSM receives a signal that the system has completed the load of the plain or cipher text, it transitions from *KEYAVAILABLE* to *CRYPTOROUND*

and starts the cryptographic process. This CRYPTOROUND state lasts until the full iteration of the cryptographic process has finished. As soon as the CRYPTOROUND is finished the main FSM transitions to the state ISTATEAVAILABLE, which means the cryptographic result is available. The number of the cycles staying KEYEXPAND and CRYPTOROUND depend on the cipher key length.

For the external I/O we multiplexed the I/O buses to reduce the number of pins. The external I/O consists of 3 groups: a 32-bit input bus for the plain or the cipher text and the cipher key, a 32-bit output bus for the final state — either encrypted or decrypted texts, and a control signal bus for the mode of operation and I/O multiplexer.

VII. Implementation and Tests

The design was described with Verilog HDL and synthesized with Synopsys' Design Compiler™ and an Artisan™ CMOS standard cell library. The place and route were performed with Cadence Silicon Ensemble™. In addition, we described the design with HDL in a technology and synthesizer independent way. For the memory I/O interface, we picked a generic synchronous memory timing to increase the portability of the Verilog HDL description of the round key cache and S-BOX blocks. In this implementation, the memory was generated with an Artisan™ memory compiler and the entire design was fabricated with the TSMC 0.18μm process.

For the testability of the design, we included scan chains in all the registers in the system using the Synopsys Test Compiler™. To scan the registers efficiently, we inserted separate scan chains according to the category of the registers — KeyAddition, MixColumn, SubByte, and RowShift as shown in Fig. 4, because our design has almost 1400 registers. By inserting independent scan chains, we could control and observe each cryptographic sub-operation in the cryptographic core datapath as well as in the key scheduler datapath. Furthermore, we could control internal control signals by controlling FSM registers.

TABLE 1. The chip specifications.

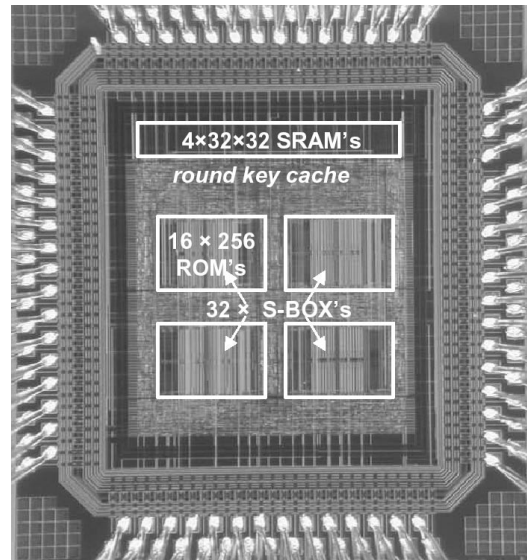
Feature	Values			
Technology	0.18μm TSMC CMOS			
Core voltage	1.8			
I/O voltage	3.3			
Package	144 PGA			
System clock	465 MHz			
Throughput (Gb/s)	128	196	256 Text Length	
	128	1.64	2.09	2.33
	196	1.36	2.09	2.33
	256 Key Length	1.16	1.78	2.33
Power	0.314W			
Gate counts (logic)	Key scheduler	Cipher core	Others	Total
	640	25476	2510	28626
Memory	SRAM		ROM	
	4×32×32		32×16× 256	

The fabricated chip operates at the nominal voltage of 1.8V, the system clock frequency of 465MHz. Table 1 shows the chip specifications including technology, packaging, power consumption and throughputs for the various key and text lengths.

VIII. Conclusion and Future Work

In this paper, we describe the fully-integrated and synthesizable ASIC implementation of the Rijndael algorithm. The chip can process all the standard key sizes and plain text lengths — 128, 196, and 256 bits. The design was fabricated with TSMC 0.18μm process and the tested result shows that the maximum throughput is 2.33Gb/s with a power consumption of 0.314W. We also proposed the round key cache to improve key scheduling efficiency. It takes advantage of the temporal locality of the cipher key in cryptographic communication.

Fig. 7: The chip die photo of the fabricated cipher core.



IX. Acknowledgement

This work was supported by an Intel Graduate Fellowship, by DARPA contract number F33615-00-C-1678, by SRC 2001-HJ-904.

X. Reference

- [1] P. Ferguson and G. Huston, What is a VPN, <http://www.employees.org/ferguson/vpn.pdf>, 1998.
- [2] R. Atkinson, Security architecture for the internet protocol, IETF Draft Architecture ipsec-arch-sec00, 1996.
- [3] J. Daemen and V. Rijmen, "The Rijndael Block Cipher", AES Proposal, <http://csrc.nist.gov/encryption/aes>, 2001.
- [4] H. Kuo et al., "A 2.29 Gbits/sec, 56mW Non-Pipelined Rijndael AES Encryption IC in a 1.8V, 0.18um CMOS Technology", Custom Integrated Circuits Conf., 2002.
- [5] C. Lu et al., "Integrated design of AES encrypter and decrypter", Proc. Intl. Conf. on Application-Specific Systems, Architectures and Processors, 2002.
- [6] T. Lin et al., "A high-throughput low-cost AES cipher chip", Proc. Asia-Pacific Conf. on ASIC, 2002.
- [7] S. Morioka et al., "A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture", Intl. Conf on Computer Design: VLSI in Computers and Processors, 2002.