Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm

Henry Kuo, Ingrid Verbauwhede

Electrical Engineering Department, University of California Los Angeles.
henrykuo@ee.ucla.edu ingrid@ee.ucla.edu

Abstract. This paper discusses the architectural optimizations for a special purpose ASIC processor that implements the AES Rijndael Algorithm. In October 2000 the NIST chose Rijndael as the new Advanced Encryption Standard (AES). The algorithm has variable key length and block length between 128, 192, or 256 bits. VLSI architectural optimizations such as parallelism and distributed memory are discussed, and several hardware design techniques are employed to increase performance and reduce area consumption. The hardware architecture is described using Verilog XL and synthesized by Synopsys with a 0.18µm standard cell library. Results show that with a design of 173,000 gates, data encryption can be done at a rate of 1.82 Gbits/sec.

1 Introduction

Although many encryption algorithms can be relatively efficiently implemented in software on general-purpose or embedded processors, there is still a need for special purpose cryptographic processors.

First of all, 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 [1], which is too slow for high-end Internet routers.

Moreover, in terms of mobile application like cellular phones, PDA's, etc., software implementation on general-purpose processors consumes much more power than special purpose ASIC's do. Last of all, it is often the case that applications require the encryption logic be physically isolated from the rest of the system, so that the encryption can be secured more easily. In this case a hardware accelerator is a more suitable solution as well.

The AES Rijndael algorithm was chosen in October 2000 and is expected to replace the DES and Triple DES because of its enhanced security levels [2]. In this paper, VLSI optimizations of the Rijndael algorithm are discussed and several hardware design modifications and techniques are used, such as memory sharing and parallelism. The circuits are synthesized using a 0.18µm CMOS standard cell library, and estimations are done on timing and gate counts.

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 51–64, 2001.

In the following sections, we will briefly discuss the algorithm flow, followed by detailed hardware implementations and techniques. After that we will present the simulation results, followed by future developments and conclusions.

2 Rijndael: Algorithm Flows

The main flow of the algorithm, as shown in Fig. 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.

For encryption, it can be divided into four blocks: Key Addition, Shift Row, Mix Column, and Substitution. The Key Addition module is byte XOR between the round key and the encryption data. The Shift Row and the Substitution modules involve mainly table lookups. Last of all, the Mix Column module composes of XOR operations. The algorithm flow is shown in Fig. 1. The Key Scheduling module is totally independent of the encryption module, and it also involves table lookups and XOR operations.

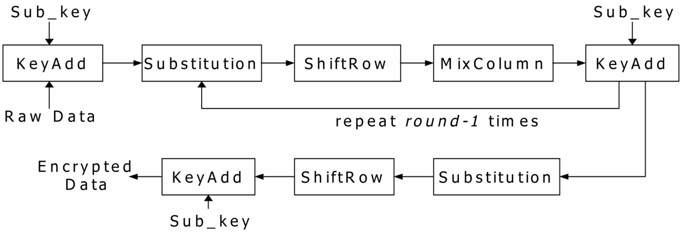

Fig. 1. Algorithm Flows.

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.

3 Architecture Optimizations

The initial specification of the Rijndael algorithm was implemented mainly in software. Although the algorithm is designed with hardware implementation in mind, 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 at the same time. Maximizing the throughput will minimize the critical paths and solve the memory access conflicts. As shown in Fig. 2

[3], there are a lot of regularities in the design of Rijndael algorithm. Therefore, with careful VLSI design, the critical path as well as the overall area can be minimized.

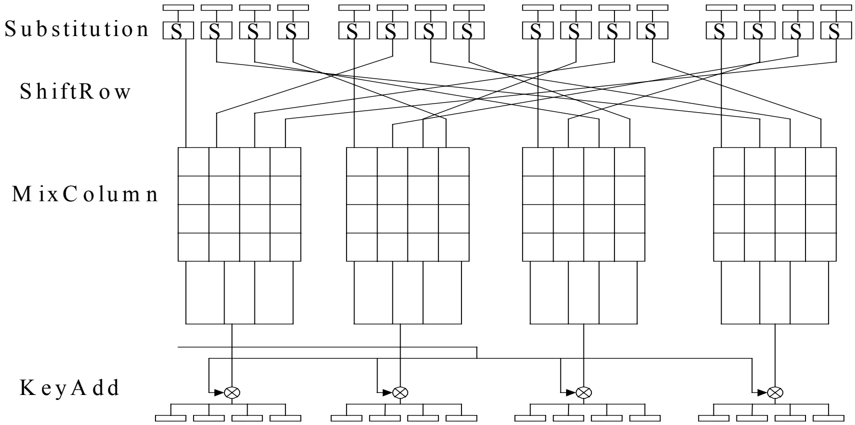

Fig. 2. 2D diagram illustrating data flow, adapted from [3].

3.1 Basic Architecture Decisions

In our implementation of the algorithm, there is only hardware for one encryption round and we re-use the same piece of hardware to complete the whole encryption process. While this implementation can help conserve most area, the main reason for this design is to incorporate different kinds of feedback modes that are currently available in the industry. Although NIST is currently initiating another new counter mode of operation, the common mode of operations used today do not allow pipelining of encryption modules. Therefore having two or more encryption modules in the processor is not the most flexible design.

Besides having hardware for one encryption round, we also designed the processor to complete one encryption round in one clock cycle. This design is very important, for example, in high throughput systems, because it ensures that the design is run at the lowest clock frequency possible with the same throughput. The drawback of this design is that we have to duplicate some of the modules, especially lookup tables, in order to finish all the required operations in one clock cycle for one encryption round.

The third basic architecture decision we made was the key scheduling. There are two ways for generating the round keys for encryption, either by generating all the sub-keys beforehand and storing them in a buffer, or generating all the sub-keys on the fly in parallel with the encryption module.

Since buffer storage could take up substantial amount of space, we decided to generate the sub-keys on the fly during encryption. Therefore we implemented the hardware required to generate one set of sub-key and re-use it for calculating all the sub-keys, and at the same time also use one clock cycle for one sub-key generation.

3.2 System Setup

The general block diagram is shown in Fig. 3. Besides the Encryption and Key Scheduling modules, there are one controller for the input channel, one controller for the output channel, and a top-level controller interfacing with the user modules. There is only one system clock, and it is fed to all the modules.

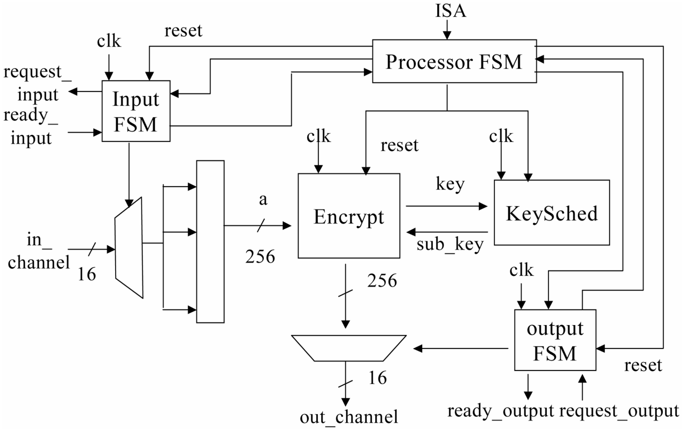

Fig. 3. Overall block diagram.

Both the input and output channels are 16 bits wide. Therefore, in order to read in the whole cipher or key, a handshaking protocol is used. The top-level controller takes in a 4-bit instruction and returns a ready signal when it is idle. In order to allow both 128, 192, and 256 bits for Encryption and Key Scheduling, the internal data path are all 256 bits. The user has the ability to set the block length and the key length using specific instructions, and the input and output controller will automatically adjust the input and output sequences.

Specifically, pipelining and unrolling are not implemented in the system. As a result, there is only one module for Encryption and one module for Key Scheduling, and these modules are reused to generate all the intermediate data and key. This design should be the most area efficient with the best module utilization.

As shown in table 1, the instructions are four bits long. Feedback modes (1110 and 0110) take in the raw data and encrypt the data for one thousand times using OFB feedback mode, and this is used for calculating the maximum operating frequency of the core during tests. Decryption is not implemented in the current design since it requires a separate datapath. Nonetheless, in order to implement decryption, either the generation of the entire sub round-keys has to be done beforehand, or there needs to be another datapath generating the inverse process of Key Scheduling. The first case requires an additional 3584 register storage while the second method requires more routing, both result in much larger area.

Table 1. Instruction sets used for this processor.

Reset		0000
Set Block Length	128 bits	1010
	192 bits	1011
	256 bits	1100
Set Key Length	128 bits	0010
	192 bits	0011
	256 bits	0100
Input Data		1001
Input Key		0001
Encrypt		1101
Encryption – Feedback Mode (for testing)		1110
Decryption		0101
Decryption – Feedback Mode (for testing)		0110
Output Data		0111

3.3 Memory Architecture Optimization

Since the design is based on one clock cycle for each encryption round, we have to duplicate memory modules several times. Consequently, the choice of memory architecture is very critical. Since all the table entries are fixed and defined in the standard, Random Access Memory (RAM) is not needed, but in fact Read Only Memory (ROM) is enough. Specifically, the algorithm will require a lot of small ROM modules instead of one large memory modules, since each lookup will only be based on a maximum of 8-bit address, which translates to 256 entries. However, the ROM has to be asynchronous; otherwise several clock cycles would be required for all the memory reads. In our design, combinational logic is used to implement the table lookups.

There are three types of tables we used in our design. The first one, which is the most used, is the S-box. It is a 256-entry table with each entry 8-bit. Using combinational network we were able to use around 2200 gates to translate the table, which converts to around 51000μm^2. The access time for the table is around 1.89ns. We have a total of 48 copies of the table in our design; 32 of them in the Encryption module and 16 in the Key Scheduling module.

The second table lookup is for deciding the shift amount in the shift row module, which has 24 entries. We implemented four copies of the tables in our design, and we were able to achieve that using 55 gates with an area of 1000μm^2. The last type of table lookup has 30 entries, and it is used to generate the round constant in the key-scheduling module. It is only accessed once in each round, so we have only one copy of the table, with 70 gates occupying 1300μm^2.

3.4 Simplification of Modulus Operation

There are several modulus operations in the algorithm: modulo 4, 6, and 8. Since the modulus values are known already, generic modulo operations are unnecessary since they require a lot of area. Therefore, it is beneficial to look into the data set and break down the modulus operations into more efficient combinational logic, which consumes less area.

For the modulus 4 and 8 operations, they are relatively easy to implement using simple shifting. Result of modulus 4 is the last 2 bits in the operand, and result of modulus 8 is the last 3 bits of the operand. In simplifying modulus 6, it is necessary to look at the set of values the operands take since there is no simple method for reducing them. In the algorithm, modulus 6 takes on values from 0 to 13, therefore a Karnaugh Map was used to implement the operation efficiently using gates.

3.5 Encryption Datapath

As discussed before, the encryption module can be broken down into four different sub-modules, and the same case applies on the hardware implementation of the algorithm. We implemented the four modules (Substitution, Shift Row, Mix Column, and Key Addition) using mainly lookup tables, XOR's, and pure combinational logic. Moreover, the datapath is 256 bits wide despite of the actual block length.

3.5.1 Substitution

The 256 bit data is broken down into 32 chunks, 8 bit each, and each of them is used as the address for S-box table lookup. The S-box contains 256 entries, and each entry is 8 bits wide. The S-box is implemented using combinational logic with an access time of around 1.89ns. In order to achieve parallelism and finish one round of encryption in one clock cycle, the same S-box is duplicated 32 times. Fig. 4 shows the block diagram for this module.

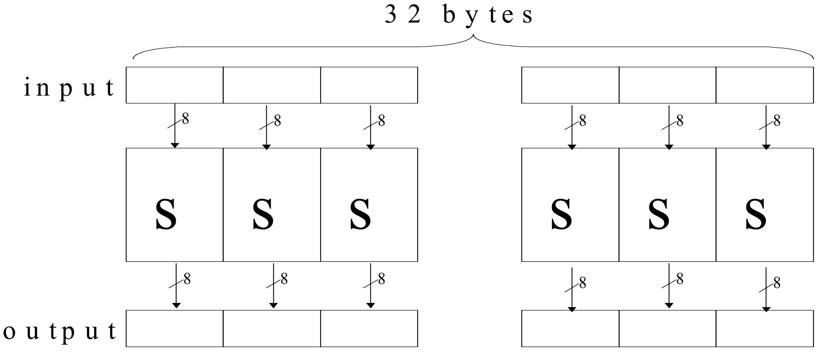

Fig. 4. Block diagrams for Substitution.

3.5.2 Shift Row

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" (Fig. 5), 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.

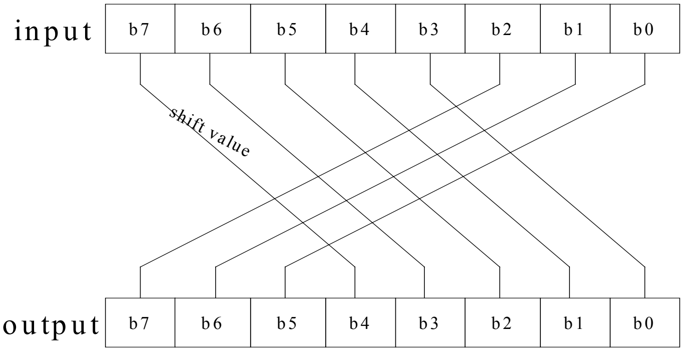

Fig. 5. Block diagram for Shift Row (only one of the four 8-byte "rows" is shown).

3.5.3 Mix Column

In Mix Column, four bytes in the corresponding position in the four "rows" are used for matrix multiplication in GF(2^8), which involves byte-wise multiplication and addition. Byte-wise additions are easily done by XOR, and several tricks are used for multiplications.

Byte-wise multiplications include multiplying the data by 1, 2, and 3. Multiplying by 1 the data remains the same. For multiplication by 2, the 8 bit data is left shifted by 1 bit, and the LSB is replaced by 0. Then the MSB of the original data is used for comparison. If it is 0, then the left shifted data is the result; if it is 1, then the left shifted value is XORed with the reduction polynomial, in this case 00011011, to generate the result. For multiplication by 3 we simply XOR the original byte with the result of multiplication by 2.

Using the above method, the multiplications by 1, 2, and 3 of each of the bytes in the data are determined. Then the correct combinations of values are XORed with each other to produce a new byte. The same process goes on until all the 32 bytes in the data are replaced.

Fig. 6 shows the block diagram for generating the first byte of each row.

3.5.4 Key Addition

In Key Addition, the 256 bit data is XORed with the 256 bit keys to generate the result, as shown in Fig. 7.

3.6 Key Scheduling Datapath

The datapath for Key Scheduling is also 256 bits wide to accommodate different key lengths. Moreover, the sub-keys are all generated on the fly, meaning that there is no buffer storage for keys generation.

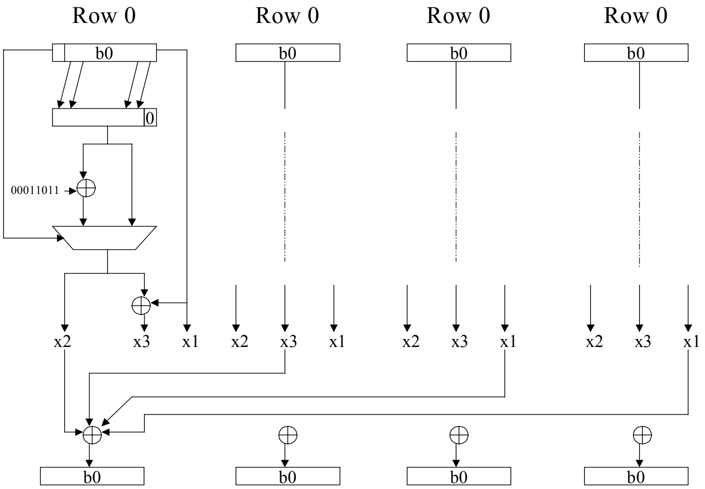

Fig. 6. Block diagram for Mix Column (only byte 0 calculation is shown).

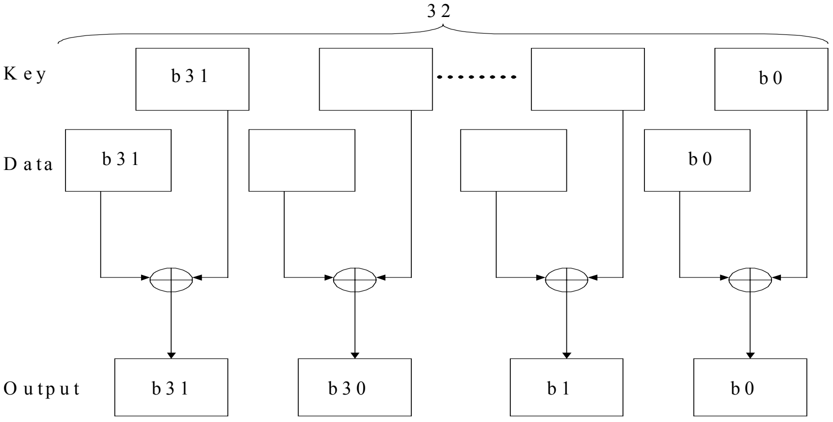

Fig. 7. Block diagram for Key Addition.

3.6.1 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.

In the third part, the 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. The block diagram is shown in Fig. 8. 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.

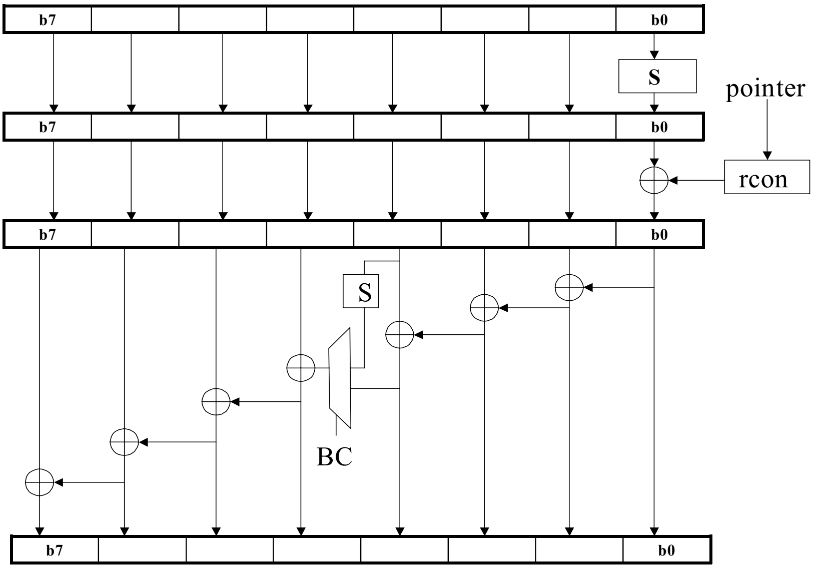

Fig. 8. Block diagram for Key Scheduling (only one of the four 8-byte "rows" is shown).

3.6.2 Key Alignment

Since the Rijndael algorithm allows different key lengths and block lengths, each sub-key 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-keys could be fed into the encryption module directly with any reorganization (Fig. 9a). 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 (Fig. 9b).

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 (Fig. 9c). In this case we will sometimes need to access the next sub-key and sometimes the previous sub-keys.

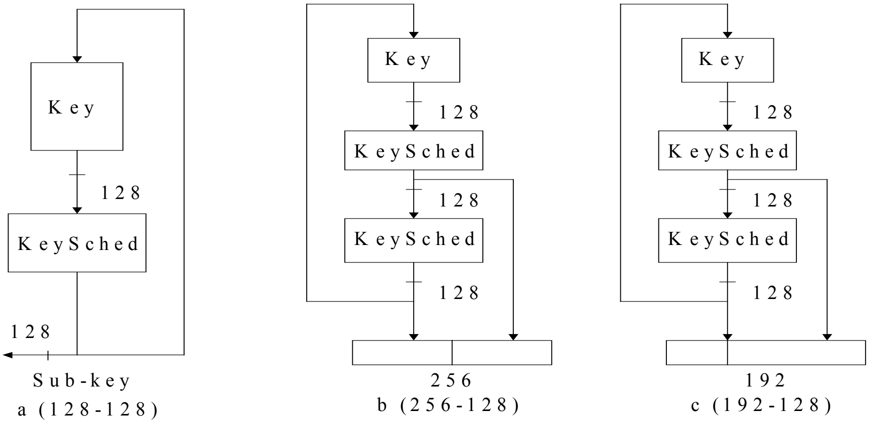

Fig. 9. Illustration of alignments of sub-keys.

3.6.3 Key Scheduling Architecture

By careful analysis of all the nine combinations between the Block Length and Key Length, we noticed that in the worst case the Key Scheduling module will need to maintain the previous, current, and also the next sub-keys in order to generate the appropriate set of keys that are fed into the encryption module.

Therefore, we decided to implement two sets of the encryption modules to achieve this. Fig. 10 shows the block diagram of our design. An extra selection module is used to select from the three sub-keys, based on the key length, block length, and the round count, the correct combination of keys that should be fed to the encryption module.

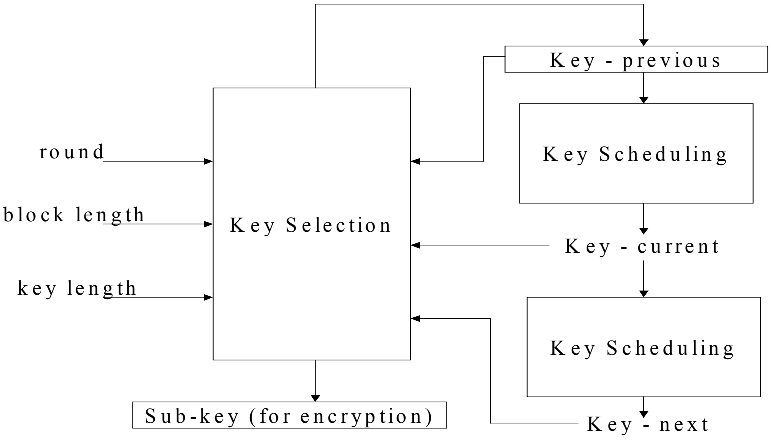

Fig. 10. Architecture of Key Scheduling used.

4 Results

The hardware design is done using the Cadence Verilog-XL, and synthesis was done using Synopsys DesignCompiler and National Semiconductor's 0.18µm standard cell library. The synthesis was done using two libraries: the worst-case library, which uses 1.2V at 120F and worst case processing, and the typical-case library, which uses 1.8V at 60F with best processing parameters. Results are in table 2.

Table 2. Synthesis Results.

	Worst-case library	Typical-case Library
Critical Path	21ns	10ns
Frequency	48MHz	100MHz
Chip Area	4.23mm^2	3.96mm^2
Gate Count	184,000	173,000
Max. Throughput (256 bits data / 128 bits key)	870 Mbits/sec	1.82 Gbits/sec
Min. Throughput (128 bits data / 256 bits key)	435 Mbits/sec	910 Mbits/sec

The critical path lies in the Key-Scheduling module, and it is shown in Fig. 11. It involves going through a S-box lookup XOR, and then the round constant lookup and XOR, followed by a sequence of XOR and one more S-box lookup. This path is duplicated one more time since we have two key-scheduling modules, and since one path is around 4.5ns, going through the two modules would take a total of 9ns. Together with the sub-key selection module, which is around 3ns, the whole critical becomes 10ns.

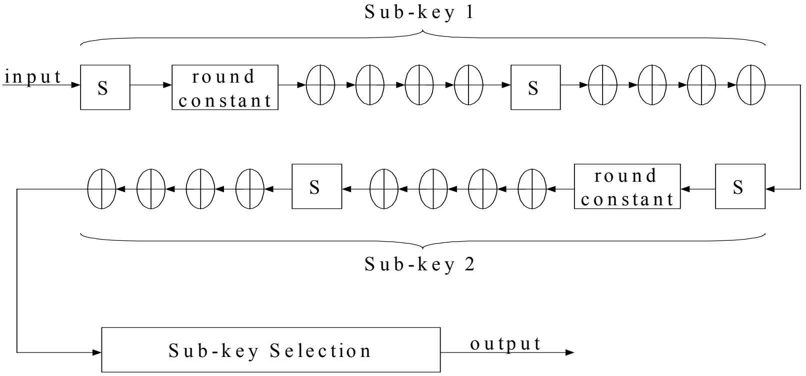

Fig. 11. Critical path for Key Scheduling.

The critical path in the Encryption module is illustrated in Fig. 12. It involves a S-box lookup, then the shift row module, which includes table lookup and XOR, four sets of XOR in Mix Column, and a final XOR operation in key addition. The overall path is around 6ns.

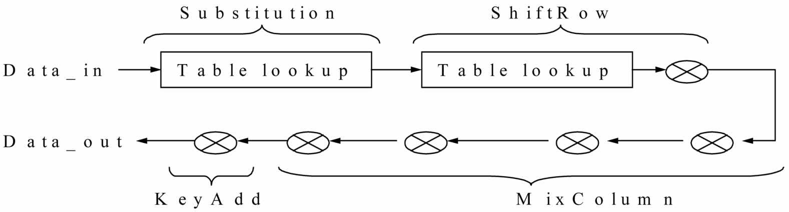

Fig.12. Critical path breakup on Encryption module.

Since the critical path is as long as 10ns, the system could operate under a clock of 100MHz in typical environment. When calculating the throughput, we measure the critical path of the processor core (Encryption and Key Scheduling modules), calculate the time to finish one encryption, and determine the throughput.

In the worst case, where the cipher is 128 bits, the key is 256 bits, and the encryption requires 14 rounds, the throughput is 910 Mbits/sec. In the best case, where the cipher is 256 bits and the encryption takes 14 rounds, the throughput is 1.82 Gbits/sec. For comparison, in software implementation, on a Pentium Pro 200MHz Pro system running Linux, the best-case throughput is about 100 Mbits/sec. Compared to the hardware implementation, the hardware implementation is about 18 times faster.

The whole chip has a size of around 3.96mm^2, with a gate count of around 173,000 gates. The input and output controller each takes 1.6% of the overall area, and the top-level controller takes around 3.9% of the overall area. The Key Scheduling module consumes about 35% of the area, and the remaining Encryption module occupies 57.5% of the overall area. All these data are summarized in table 3.

On the other hand, each 256 bytes table consumes about 5100µm^2. In the whole system, together with the four tables for Shift Row and the one for round constant are very small, all the lookup tables combine to 2.5mm^2, around 63% of the overall area. In terms of register storage, the current design requires a total of 13200µm^2 for registers, which is about 8% of the overall area. Therefore, all memory components, including registers and table lookups, occupy around 71% of the area of the chip.

Table 3. Comparison between Encryption and Key Scheduling modules.

	Encryption	Key Scheduling
Area	2.28mm^2	1.39mm^2
Gate Count	99,300	60,100
Percentage of Chip Size	57.5%	35%
Critical Path	6ns	10ns

Table 4 compares the design described in this paper with the design by National Security Agency (NSA) [6]. The research conducted by NSA was primary used as a reference for NIST, therefore it did not include special architecture techniques in order to create fair results between all AES candidates. Also, notice the library used was a 0.5µm library.

Table 4. Comparison with results from NSA.

	Design from NSA (0.5µm)	Design in this paper (0.18µm)
Chip Area	46mm^2	3.96mm^2
Gate Count	1.000,000	173,000
Max. Throughput	447 Mbits/sec	1.82 Gbits/sec
Min. Throughput	320 Mbits/sec	910 Mbits/sec

From our results, we noticed that the generation of sub-keys on the fly creates a serious bottleneck for the system. Since the encryption module has a critical path of around 6ns and the key scheduling module has a critical path of 10ns, the encryption module is idle for almost 4ns. If we could reduce path inside the key scheduling module to around 10ns the throughput would be maximized.

This implementation is entirely possible. As we have discussed, one key-scheduling module has a critical path of around 4.5ns, therefore if we implement some buffer storage for sub-key generation, where we only need to maintain one key-scheduling module, the critical path inside key-scheduling drop substantially from 10ns to at most 5ns, which matches precisely with the encryption module.

The tradeoffs with this implementation would be the excessive area for buffer storage and also the time required to generate all the sub-keys before encryption can start. By analyzing the current Key Scheduling module, each of the two sub-key generation parts consumes 0.53mm^2 and the sub-key selection module consumes 0.33mm^2, whereas 3584 bits of register storage takes up around 0.5mm^2. Therefore if we generate all sub-keys ahead of time, we can save the sub-key selection module and one sub-key generation module, replace that by 3584 bits of register storage and actually save around 0.35mm^2 of chip size.

On the other hand, although the critical path could be reduced from 10ns to 6ns, the new design would require time to initialize all the keys. In the worst case, where block size is 256 bits and key size is 128 bits, it would require 28 cycles to generate all the required sub-keys for encryption. Compared to the 14 cycles required for actual encryption, the overhead could be as much as 200%.

5 Conclusion

In this paper, a hardware implementation of the AES Rijndael algorithm is described. In order to better fit the algorithm for hardware implementation, several modifications are introduced, including memory access, modulo reduction, and key scheduling on the fly. Synthesized using a 0.18µm library, the gate count is estimated to be around 173,000. It can sustain a maximum throughput of around 1.82 Gbits/sec at a clock frequency of 100MHz, which is substantially faster than the software implementation.

Moreover, area tradeoff for memory sharing and addition of decryption is also discussed.

For future development, estimation on the real time required for key initialization and time for a whole encryption should be done on the real chip. Moreover, more detailed estimation should be done on the actual area increment for the addition of decryption. Power consumption analysis is essential as well for mobile application, and research on the actual resistance towards timing and power attack will be investigated [7]. Last of all, analysis on using buffer and sub-key pre-calculation should be implemented should be done as well.

Acknowledgements: UC Micro #00-097, Atmel Corporation, Panasonic, and National Semiconductor Corporation sponsored this work.

References

1. J. Daemen and V. Rijimen, "AES Proposal: Rijndael." Available at http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf
2. E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, J. Nechvatal, and E. Roback, "Report on the Development of the Advanced Encryption Standard (AES)." Available at http://csrc.nist.gov/encryption/aes/round2/r2report.pdf
3. J. Savard, "The Advanced Encryption Standard (Rijndael)." Available at http://home.ecn.ab.ca/~jsavard/crypto/co040801.htm
4. W. Diffic and M. Hellman, "Privacy and Authentication: An Introduction to Cryptography." Proceedings of IEEE, 67 (1979), pp. 397-427.
5. I. Verbauwhede, F. Hoornaert, H. De Man, and J. Vandewalle, "ASIC Cryptographical Processor Based on DES." Proceedings of EURO-ASIC-91, Paris, May 1991.
6. M. Bean, C. Ficke, T. Rozylowicz, and B. Weeks, "Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms." Available at http://csrc.nist.gov/encryption/aes/round2/NSA-AESfinalreport.pdf
7. J. Jaffe, B. Jun, and P. Kocher. "Introduction to Differential Power Analysis and Related Attacks." Available at http://www.cryptography.com/dpa/technical/index.html