Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor

Patrick R. Schaumont
UCLA Dept of EE
Los Angeles, CA
schaum@ee.ucla.edu

Henry Kuo
UCLA Dept of EE
Los Angeles, CA
henrykuo@ee.ucla.edu

Ingrid M. Verbauwhede
UCLA Dept of EE
Los Angeles, CA
ingrid@ee.ucla.edu

ABSTRACT

This contribution describes the design and performance testing of an Advanced Encryption Standard (AES) compliant encryption chip that delivers 2.29 GB/s of encryption throughput at 56 mw of power consumption. We discuss how the high level reference specification in C is translated into a parallel architecture. Design decisions are motivated from a system level viewpoint. The prototyping setup is discussed.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-based Systems]**

General Terms

Measurement, Performance, Design, Security.

Keywords

Rijndael, Encryption, Domain-Specific, Low-Power.

1. INTRODUCTION

Recently we developed and tested an AES compliant encryption processor that implements Rijndael [1] at 2.29 Gb/s and 56 mW of power consumption. This device can be used to instrument a digital broadband platform like a router or a wireless base station with security services. The processor is programmable and supports Rijndael in any combination of key length (128,192,256 bits) and data size (128,192,256 bits). It is integrated into the host platform through a 16 bit data bus and operated through a small instruction set. The implementation uses 173 KGates of 1.8V 0µ18 CMOS standard cell technology, and has been verified operational at up to 154 MHz clock frequency in a prototype setup.

The paper is organized as follows. In section 2, we will review the system level architecture. This architecture was designed starting from the NIST reference implementation in C [2]. We will discuss the transformations that are needed on this description to create the system architecture. In addition we touch upon integration issues for our architecture.

In section 3 we provide motivation for some design decisions with a system level look at the chip. In addition, we also touch upon related work and other Rijndael implementations that have been done by academia and industry.

We used an HDL based design flow for this chip, which will be discussed in section 4. We will also discuss design statistics with respect to design efficiency. Finally the prototyping approach and measurements are shown in section 5, and conclusions and outlook are given in section 6.

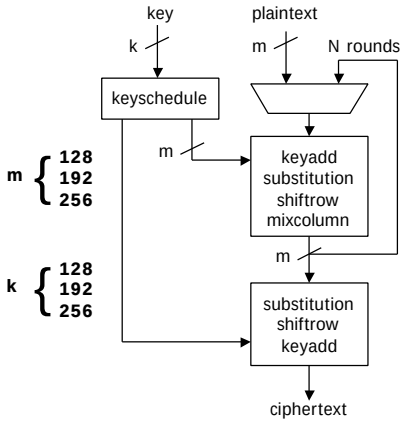

Figure 1: Rijndael Encryption

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.
Copyright 2002 ACM 1-58113-461-4/02/0006…$5.00.

2. SYSTEM SPECIFICATION AND ARCHITECTURE

The Rijndael encryption algorithm is a block cipher that converts cleartext data blocks of 128, 192 or 256 bit into ciphertext blocks of the same length. It uses a key of selectable length (128, 192, 256 bit). The algorithm is organized as a set of iterations called *rounds* as illustrated in Figure 1. The number of rounds is dependent on the data block length. For each round, a subkey is created out of the original key by means of a key schedule. The operations performed on the data blocks include byte substitution by means of a lookup table, transpositions and rotations, modulo-2 addition of a subkey as well as Galois field operations. They always affect a complete 128, 192 or 256-bit data block at a time.

2.1 System Architecture

The approach we took was to design a high-speed hardware accelerator of the Rijndael algorithm. As will be demonstrated, the energy efficiency of this approach is orders of magnitude better than using a general-purpose programmable platform.

The system architecture of our implementation is shown in Figure 2. The central block of the architecture, `encrypt`, implements one round of a Rijndael encryption in a fully parallel, non-pipelined fashion. A Rijndael encryption can be completed at one clock cycle per round. Thus, following the standard we

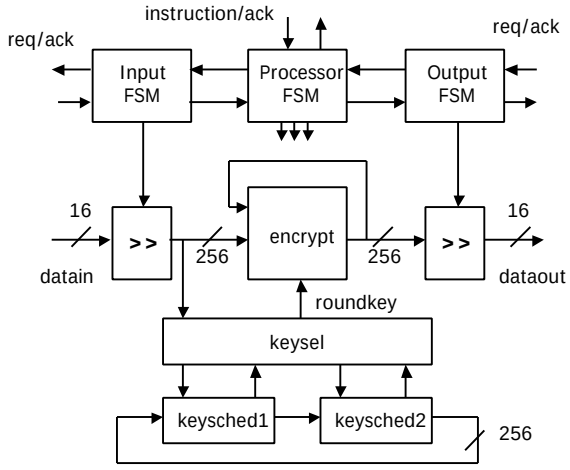

Figure 2: System Architecture

need from 10 clock cycles for 128 bit blocks up to 14 clock cyles for 256 bit blocks.

The architecture can support any combination of key-length and data-block length. As a result the key-scheduling process speed must be adjusted relative to the encryption process speed when data-blocks and sub-keys have different lengths. For example, when 256 bit data blocks and 128 bit subkeys are needed, then 2 key schedule iterations are needed for each data block. With a dual implementation of key schedulers, this double rate can be supported. Double-rate key scheduling is only a worst case situation, but also non-integral rates can occur. The combination of 192-bit data blocks with 128-bit keys for instance requires 1.5 key schedule iterations per data block.

The processor has three controllers, two for I/O interfacing and one for instruction sequencing. They communicate through request/acknowledge protocols with the host system. This asynchronous interfacing method allows the chip to be clocked much faster than the bus it is connected to. It also brings considerable simplification of the performance testing process. Separation of I/O controllers from the instruction sequencing controller makes this block easily portable to a different context, were a different data-bus or I/O interface protocol would be used.

Figure 3 shows the internals of the central encrypt block and illustrates the nature of the data processing that is done by this processor. Data words are 256 bit wide. The data is organized conceptually in a 4 by 8 matrix of bytes, totaling 32 bytes. If the processor is working in 128 or 192 bit mode, then the leftmost 2 resp. 4 columns of the matrix are unused. The processing starts at the bottom `keyadd` block which performs modulo-2 addition of a subkey to each data byte. Next, each data byte is fed into an S-box, implemented as a lookup array in the `substitution` block. This happens in parallel for all 32 bytes, so 32 lookup tables (of 256 8-bit entries) are used. In the `shiftrow` block, the matrix rows are circularly shifted with a parameter- and row-dependent amount of positions. Finally in `mixcolumn` each column is transformed linearly using galois-field constant multiplications. All of these operations are performed within one clock cycle. Yet by careful design the critical path of this block is

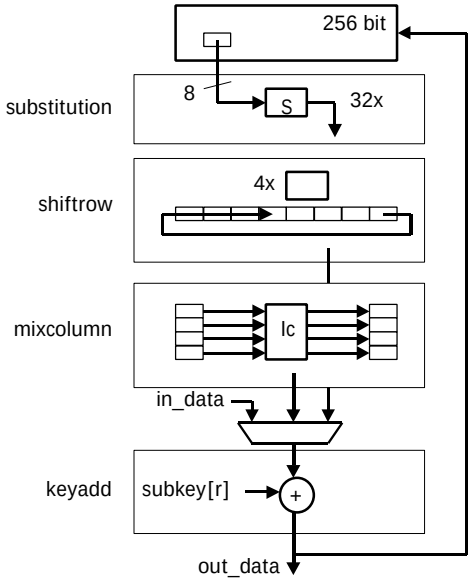

Figure 3: Encrypt Block

6ns in 0µ18 technology. The overall critical path of the design is 10 ns and resides in the key schedule block. A detailed discussion of the architecture optimization has been published [3].

2.2 System Specification

The Rijndael encryption algorithm is available as a C program on the NIST website [2]. For hardware implementation, the reference implementation in C presents a number of issues that need to be resolved before architecture design can start. For a hardware designer, these are best described as *ambiguities*. We enumerate those below to demonstrate what problems would be encountered by a C-to-hardware compiler, assuming it would start from the reference implementation.

First, a parallel version of the sequential specification is needed. This requires dataflow analysis that crosses the boundaries of function calls. The hardware in Figure 3 for example is the result of merging several C function calls, and unrolling and merging the for-loops in each function. It is nontrivial because the loop bounds in C are variable and parameter-dependent. In addition, not all C functions are treated the same way. For example, the evaluation of sub-keys can be done online (in parallel with the encryption process) or offline. In C, the notion of parallelism is absent, and the distinction between on-line and off-line is merely an assumption reflected in the ordering of function calls. In hardware however, it has major impact on the memory architecture as all subkeys need to be stored first in offline mode.

A second aspect involves parameterization. The introduction of runtime parameters like selectable key-length has a direct impact on the implementation overhead of the hardware. The C reference code implements parameterization with lookup arrays and control constructs such as `if-then`. In hardware, an efficient implementation requires us to get rid of runtime variability as much as possible and for example apply constant propagation whenever a parameter is known constant. Also if the parameter range is restricted, we can simplify the

Original

```
word8 Alog[255] = { ..};
word8 Log[255]  = { ..};
word8 mul(word8 a, word8 b) {
if (a&&b)
  return Alog[(Log[a]+Log[b])% 255];
}
```

Optimized

```
/* we know that only mul(a,2) and mul(a,3)
   are used. Do both of them in this
   function without lookup tables */
void mul23(word8 a, word *m2,
                    word *m3) {
  *m2 = (a & 0x80) ? (a<<1) ^ 0x1C
                   : (a<<1);
  *m3 = *m2 ^ a;
}
```

Figure 4: An example of code specialization

implementation. One particular example is shown in Figure 4 and demonstrates the implementation of the Galois Field multiplications. In the reference code, the Galois Field multiplications are implemented using a Log-Alog lookup table (with two such multiplications per matrix row en per round). The Log-Alog lookup table is elegant to describe and easy to understand but expensive as it uses large lookup tables. On closer inspection it turns out that for encryption only two different constant multiplications are used, with 2 and with 3. We thus can replace the lookup-table method with a dedicated structure.

A third aspect is memory architecture, which is richer and more elaborate in a hardware implementation then in software. The C reference implementation uses arrays for both the storing of intermediate results, the storing of keys and data blocks, as well as lookup tables. In the hardware implementation, these translate to wiring, registers, random logic tables and dedicated computation blocks. The problem that the designer is facing is that the initial description in C gives no clue about how to treat each array. This problem is also worsened by the lack of parallelism in the initial description.

A fourth aspect involves using transformations that cross memory-computation boundaries. An example is the implementation of the key scheduling. Subkeys are created in our architecture online. Because of variable key and datablock length, a setting can occur where two subkeys per round are needed, each 128 bit, for one datablock of 256 bit. As a result the key scheduling hardware is implemented twice, as illustrated in Figure 2. This way the hardware can evaluate up to two subkeys per clock cycle. In the C reference implementation, the key scheduling function initializes one array that contains all subkey material for one data block. It uses a parameter-dependent `while` loop to ensure enough subkeys have been evaluated. Clearly such a while loop cannot be transformed into a fixed rate hardware implementation. The kind of transformation that is needed here crosses the boundaries of memory and computation space.

Summarizing, we observed the need for the following analysis techniques of the C code.

- Dataflow analysis that can cross the borders of function calls, and that gives feedback to the designer as to what strategies can be used for parallelization of the code, both at function-level as well as block-level. Such feedback includes memory-access and operation profiling.
- Source-level loop transformations and analysis of the storage architecture alternatives [5].
- Source-level constant propagation and folding, also for variables for which a constant input can be guaranteed. Reduction of source-level control-flow complexity as a result of this.

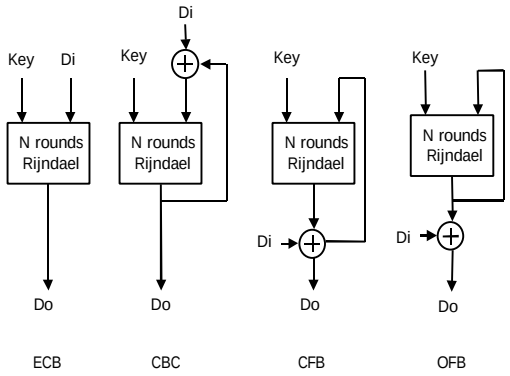

Figure 5: Feedback Modes prevent Pipelining

These techniques are known in the high-performance software compiler community. The existing C-to-hardware technology however focuses on datapath- and sequential controller synthesis. High level synthesis support is available for fixed architecture templates such as VLIW or ASIP with customizable instruction sets. Since our architecture could not be classified as one of those, we choose to base our implementation flow on HDL and do manual architecture design.

2.3 Integration Issues

The interface that attaches the accelerator processor to the host system is used to transfer blocks of plaintext and keys onto the processor, and extract ciphertext out of it. It also presents an instruction set to the host system to control the execution of the Rijndael algorithm. Two factors were taken into account here.

First, the interface operation was designed to be independent of the processor clock. Typically the interface clock speed is decided by the system communication bus present in the host. The Rijndael processor clock speed on the other hand is as fast as critical path and power consumption constraints allow. Therefore, two-way handshakes were used on both the data bus as well as the instruction bus (Figure 2).

The second issue is that the interface should allow easy porting over a range of host processors, with potentially different buswidths. With separate I/O controller FSMs, the amount of code that needs to be revised in a new context is kept minimal.

3. SYSTEM ISSUES AND RELATED WORK

We now motivate several design decisions in our architecture and indicate some related work along the way.

3.1 Use of Pipelining

In hardware implementation, pipelining is used as a throughput enhancing technique. Considering the system architecture in Figure 2, one possible transformation is to unroll the iterative

encryption algorithm (`encrypt` block) and next insert pipeline stages. This leads to very high throughputs (25 Gbps is claimed by some vendors of 0µ18 CMOS softcores). Such pipelined implementations have only limited application for a block cipher. These ciphers are always used in well defined *mode of operation* (Figure 5). The modes include Electronic Code Book, Cipher-Block Chaining, Cipher Feedback and Output Feedback. Three of these four modes rely on feedback of the ciphertext to the input. For these modes, pipelining of the Rijndael block is not useful to increase throughput. In addition, the mode that can be pipelined (ECB), is also cryptographically weakest since it translates identical plaintext blocks to identical ciphertext blocks. An ECB cipher will for example not hide recurring data patterns [7]. For this reason, our implementation does not use pipelining. The 2 Gb/s performance figure of our chip holds for any encryption mode.

3.2 Online Subkey Computation

The subkeys that are created by the key scheduling process need to be evaluated only once per key. After that the subkeys can be reused for all data blocks as long as the key is not changed. Therefore some implementations choose to evaluate all subkeys offline before the encryption process starts, and to store the subkeys in a lookup table. This approach is good for batchmode encryption applications, like for instance secure document storage, but not for applications where keys are changed frequently.

In our implementation, subkeys are calculated online as needed and can be changed at each data block. This situation occurs for example in Internet routers with IPSEC support [4]. In an IPSEC router, the key and subkey material (1280 bits for 128-bit data blocks) is potentially different for each secured packet route. Due to the large number of active routes, subkeys cannot be stored in on-chip memory. They need to be either calculated on-line or else moved together with a packet payload onto chip. However, half of the Internet packets are only 64 bytes in length (512 bits) [6]. With offline calculation and off-chip storage of subkeys the router will use more bus bandwidth for the communication of subkeys than for the communication of useful data payloads. The effective solution for this is online computation of subkeys such as is done in our architecture.

3.3 Decryption and Authentication

While our chip does not contain a Rijndael decryption algorithm, it can still be used to build a decryption unit. The OFB and CFB modes shown in Figure 5 create ciphertext from plaintext by means of a simple XOR operation. Since this operation is its' own inverse we can use the same Rijndael encryption block for both encryption and decryption. In communication systems, the use of the OFB mode is popular for securing data streams because it prevents the propagation of communication errors into multiple blocks of data.

Another application where only an encryption algorithm is used is message authentication. Here a sender creates a message signature by encrypting the message with a CBC-mode cipher, and sending the last encryption output as the signature. The signature is hard to compute without knowing the encryption key. A receiver who shares this secret key, can then verify the authenticity of the message by recomputing the signature using the same CBC encryption on the received data.

Table 1: Comparison based on Energy Efficiency

AES 128bit key 128bit data	Throughput	Power	Figure of Merit (Gb/s/W)
This work (0u18 CMOS)	1.28 Gbits/sec	56 mW	22.8 (100%)
This work (on FPGA [i])	640 Mbit/s	1.63 W	0.39 (1.7%)
Pentium III [ii],[iii]	645 Mbits/sec	41.4 W	0.015 (0.06%)

[i] Xilinx Webpack + Xilinx Virtex Power Estimator on Domain Spec Processor Design
[ii] Helger Lipmaa, http://www.tcs.hut.fi/~helger/aes/rijndael.html: PIII assembly handcoded
[iii] Intel Pentium III Datasheet 1.13 GHz (Vcc=1.8V, Icc=23A)

3.4 Software Implementations

Several software implementations have been created with handcrafted assembly for highest performance. The best implementation we found requires 227 cycles for 10 rounds of 128 bit data blocks and 128 bit keys on a Pentium III, and 124 cycles on IA64 [8]. Since these run on processors clocked in the GHz range, this performance eventually approaches the efficiency of a dedicated hardware implementation. With respect to power consumption however, the software solution on general purpose processors is three orders of magnitude higher then a dedicated hardware implementation (Table 1).

3.5 Hardware Implementations

Initial designs from academia [9,10,11] and industry seem to have focused on the design of Rijndael cores in reconfigurable hardware. These implementations show that contemporary reconfigurable platforms with their rich distributed memory architecture are well suited for Rijndael prototype implementations. Most of them use pre-computed subkeys. For pipelined ciphers, very high performances have been obtained.

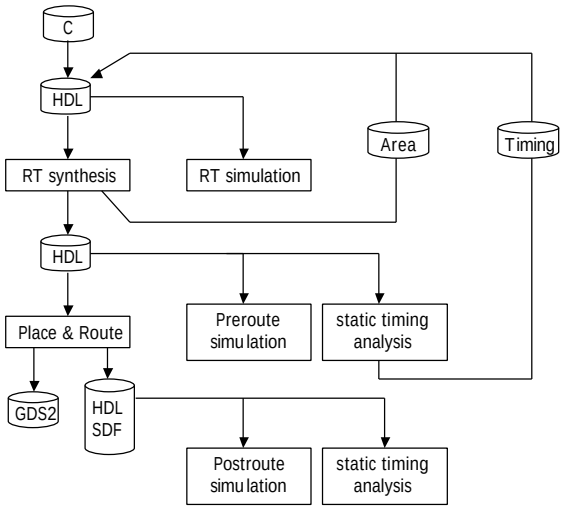

Figure 7: HDL Design Flow

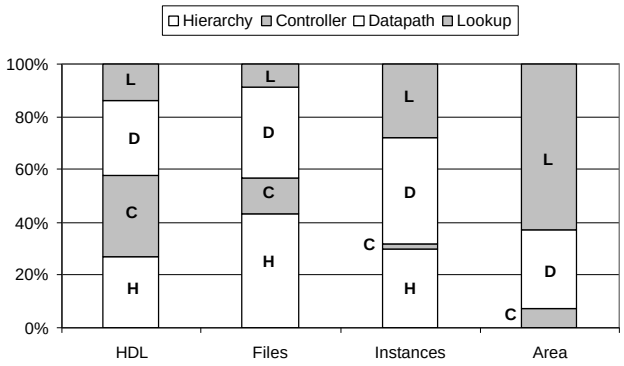

Figure 6 How HDL reflects in gates

For non-pipelined operation in FPGA, the 1.5 Gbps limit is to our knowledge still unbroken.

While almost no published power consumption figures exist, we estimate that our processor is at least two orders of magnitude better then an FPGA solution (Table 1).

Several softcore implementations have been designed as well, which advertise clock speeds ranging from 200 to 350 MHz in 0µ18 CMOS technology. One published ASIC design [12] implements a fully enrolled encryption unit in in 613 Kgates of 0u35 CMOS technology. An overview based on pre-tapeout performance figures can be found in [13].

3.6 AES standard implementation

Since we started design of the chip, NIST has finalized the standard for AES encryption. While the architecture of Figure 2 supports a superset of the AES parameters, it can also be simplified when only AES compliant settings are needed. AES restricts the data block size to 128 bits only. This means that the encrypt block of Figure 2 can be halved. In addition, one key scheduling block can be eliminated, since the keys are always larger then the data blocks. This would roughly reduce the chip gate count by half (85 Kgate). The critical path that currently resides in the key scheduler would be reduced as well. We estimate that the resulting design would reach at least 200 MHz with the same HDL synthesis methodology.

4. DESIGN AND IMPLEMENTATION

4.1 HDL Design

From the design point of view, translating the architecture of Figure 2 into HDL is a straightforward process, once the RT level architecture is designed.

One aspect that was found to be difficult was the design of the control architectures. This is because there is no elegant HDL mechanism by which control structures (finite state machines) can be captured. A coding style based on `case` statements quickly becomes confusing and complex, and tends to increase the amount of coding as shown in Figure 6.

This figure shows the relative weights of design elements used to implement hierarchy, controller, datapath and lookup tables. The four categories are non-commented lines of HDL code, number of HDL files, number of instances in the design hierarchy and finally the silicon area. It is easily seen that less then 50% of the lines of HDL occupy more then 90% of the chip area (lookup tables and datapath). This observation is important when considering design productivity issues. Closer inspection reveals that the use of HDL hierarchy is very effective in datapath design, but not in the design of controllers. Each of the three controllers in our design has their own HDL design, and each translate to a single instance in the design hierarchy. Yet, they require one third of the coding. They also represent a semantically hard part because of the way FSM are encoded in HDL. While there are tools that capture FSM design in a graphic formalism, these require partioning of the design in a graphic and a non-graphic part.

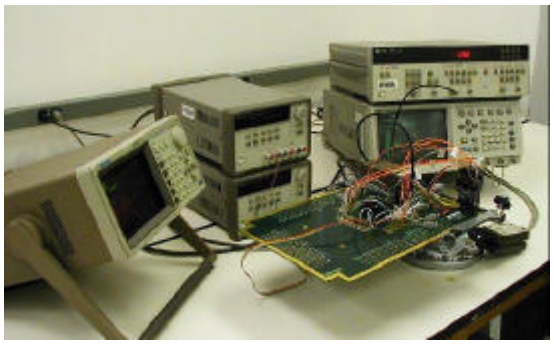

Figure 8: Test Setup

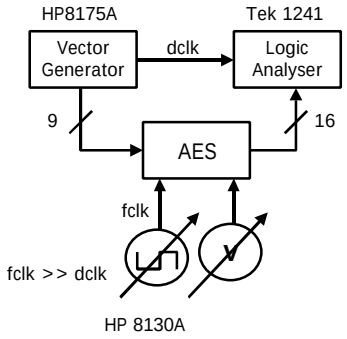

Figure 9: Measurement Setup

4.2 Design Flow

The chip was implemented with an HDL based design flow as shown in Figure 7. The inital RT level description, created out of the C reference implementation by careful analysis, was synthesized to gate level with Synopsys DC. The resulting netlist was then processed with Avanti's SIlicon Ensemble P&R. The verification path uses Cadence's Verilog-XL as HDL simulator and Synopsys' Primetime as static timing analysis tool. The chip was processed by National Semiconductor. Of the 16 test samples that were received from the foundry, 14 were operational.

5. TEST AND MEASUREMENTS

The test setup, shown in Figure 8, allows creating complete schmoo plots and at-speed testing of the chip. The test setup was made using readily off the shelf equipment, connected as shown in Figure 9. A vector generator provides testbench stimuli at a rate of 20 Mword/s. Only 9 bits of testvector input data are needed; by using simple keys and data patterns some of the input

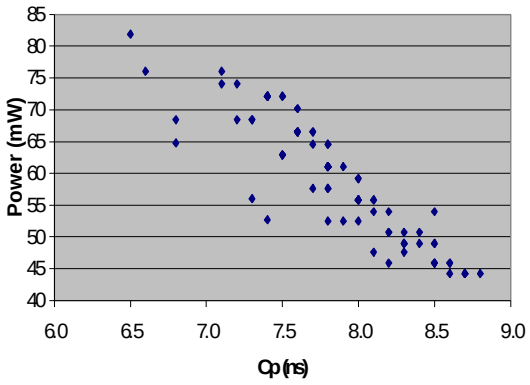

Figure 10: Power Consumption and Critical path

Table 2: Chip Characteristics

Feature		Value	Unit
CMOS Stcell Technology		0.18	um
Core Voltage (VDD)		1.8	V
IO Ring (VDDIO)		3.3	V
Package		68	PLCC
Min. Clock Period	(1.95 V)	6.5	ns
	(1.9 V)	7.5	ns
	(1.8V)	8	ns
Power Consumption	(1.95 V)	82	mW
	(1.9 V)	67	mW
	(1.8V)	54	mW
Gate Count		173	Kgate

Throughput	**Data/Key**	**128**	**192**	**256**
	128	1.6	1.33	1.14
	192	2.0	2.0	1.71
	256	2.29	2.29	2.29

Interfaces	16 bit data-in with req-ack handshake
	16 bit data-out with req-ack handshake
	4 bit instruction with req-ack handshake
	reset, clock and processor status signals

bits could be tied to ground. The testvectors are processed by our AES chip which is clocked by a clock generator at a high rate (100 MHz nominally). The outputs of the chip are observed by a logic analyser. In order to save logic analyser memory the analyser is operating synchronously and clocked out of the vector generator. This setup works because the interfaces at the chip boundaries are implemented using two way handshakes. Once the test vectors are entered in the chip, the encryption process runs at full speed. The relevance of this testing approach is that it operates our core at full speed in order to demonstrate 2.29 Gb/s encryption throughput, without requiring expensive equipment that can feed test vectors at such rates into the chip. The AES chip also uses a variable voltage supply. In combination with the variable clock generator this allows to determine the power consumption at each critical clock frequency. This is illustrated in Figure 10. The chip characteristics are summarized in Table 2.

6. CONCLUSIONS AND OUTLOOK

In this contribution, we presented a 173 Kgate Rijndael encryption core that was verified at 2.29 GB/s encryption speed. The architecture was designed for best performance over several different cryptographic modes, and several different host systems. Our current research efforts will further improve this design in two areas. First, power consumption will remain a critical factor, especially when cryptographic applications will move into embedded context. This leads to the concept of power-optimized domain specific processors, which we expect to see much more in the future. Second, while state of the art compilers cannot yet take arbitrary high level code into implementation, there is a clear need for design at higher abstraction level. We are also developing a design environment that can tackle the design of domain specific processors at higher abstraction level.

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of National Semiconductor for chip processing, as well as the funding support of NSF and UC Micro.

8. REFERENCES

[1] J. Daemen, V. Rijmen, The block cipher Rijndael, Smart Card Research and Applications, LNCS 1820, J.-J. Quisquater, B. Schneier, Eds, Springer Verlag, 2000, pg. 288-296.

[2] NIST Advanced Encryption Standard, http://www.nist.gov/aes/

[3] H. Kuo, I. Verbauwhede, Architectural Optimization for a 1.82 Gb/s VLSI Implementation of the AES Rijndael algorithm, Cryptographic Hardware and Embedded Systems (CHES 2001), LNCS 2162, pp. 51-64, Springer-Verlag.

[4] R Oppliger, Security at the Internet Layer, IEEE Computer, September 1998.

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, "Custom Memory Management Methodology", ISBN 0-7923-8288-9, Kluwer Academic Publishers, 1998.

[6] D. Whiting, B. Schneier, S. Bellovin, AES Key agility issues in high speed IPsec implementations, Public Comments on AES Candidate Algorithms Round 2, http://www.nist.gov/aes

[7] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

[8] H. Lipmaa, AES Candidates: A survey of implementations, http://www.tcs.hut.fi/~helger/aes/rijndael.html

[9] M. McLoone, J. McCanny, High Performance Single Chip FPGA Rijndael Algorithm Implementations, Workshop on Cryptographic Hardware and Embedded Systems, Paris, 2001.

[10] V. Fischer, M. Drutarovsky, Two Methods of Rijndael Implementations in Reconfigurable Hardware, Workshop on Cryptographic Hardware and Embedded Systems, Paris, 2001.

[11] P. Chodowiec, K. Gaj, P. Bellows, B. Schott, Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board, Proc. Information Security Conference, Malaga, Spain, Oct 1-3, 2001.

[12] T. Ichikawa et al, Hardware Evaluation of the AES Finalists, in Proc. 3th AES Candidate Conference, New York, April 13-14, 2000.

[13] A. Satoh, S. Morioka, K. Takano, S. Munetoh, A Compact Rijndael Hardware Architecture with S-Box Optimization, Proc. ASIACRYPT 2001, LNCS 2248, pp. 239-254, 2000