# A 2 Gb/s Balanced AES Crypto-Chip Implementation

F. K. Gürkaynak, A. Burg, N. Felber, W. Fichtner
Integrated Systems Laboratory
ETH Zurich
{apburg,kgf,felber,fw}@iis.ee.ethz.ch

D. Gasser, F. Hug
Students of Computer Science
ETH Zurich
{dog,franco}@student.ethz.ch

H. Kaeslin
Microelectronics Design Center
ETH Zurich
kaeslin@ee.ethz.ch

## ABSTRACT

We present a balanced 2 Gb/s en-/decryption ASIC realization of the AES algorithm that supports all standard operation modes and key lengths. Rather than optimizing only for throughput, special care is taken to balance the more involved decryption path with that of the encryption path using a number of high-level architectural and register transfer level optimizations. The fabricated en-/decryption core requires an active area of only 3.56 mm$^2$ (less than 120,000 gate equivalents) in a modest 0.25 $\mu$m CMOS technology.

## Categories and Subject Descriptors

B.7.1 [**Hardware**]: Integrated Circuits—*Algorithms implemented in hardware*

## General Terms

Design, Security

## Keywords

AES, Rijndael, ASIC implementation

## 1. INTRODUCTION

The Advanced Encryption Standard (AES) has been selected as the replacement of the aging Data Encryption Standard (DES) algorithm by the National Institute of Standards (NIST) after a rigorous evaluation of 15 candidate algorithms in October 2000. Starting with the evaluation process, several studies have been performed to determine the hardware efficiency of candidate algorithms. Most of them have concentrated on FPGA implementations [2, 3], and on estimated ASIC performance [4, 9]. After Rijndael [1] was announced as the AES winner, several optimized implementations have been reported both for FPGAs [8] and ASICs [11, 6, 7, 5, 10].

In this paper, we review the basic architectural and RTL level transformations that can be applied for designing efficient AES ASICs. The general hardware architecture and RTL level transformations are discussed in sections 2 and 3 respectively. Fastcore, an optimized AES implementation is presented in section 4 and finally in section 5 conclusions are drawn.


*GLSVLSI'04,* April 26–28, 2004, Boston, Massachusetts, USA.


## 2. HARDWARE ARCHITECTURE

The AES algorithm is a block-cipher operating on 128-bit data blocks[1] supporting three different key lengths of 128, 192 and 256 bits. An AES encryption operation consists of a number of encryption rounds (*NR*) that depends on the length of the key. The standard calls for 10 rounds when using a 128-bit key, 12 rounds for a 192-bit key and 14 rounds for a 256-bit key. In encryption mode, each round is composed of a set of four different basic operations. In decryption mode the inverse of these operations are applied in reverse order. Both procedures are summarized below:

**Encryption**

```
AddRoundKey;
do (NR-1) times {
   SubBytes;
   ShiftRows;
   MixColumns;
   AddRoundKey;
}
SubBytes;
ShiftRows;
AddRoundKey;
```

**Decryption**

```
AddRoundKey;
do (NR-1) times {
   InverseShiftRows;
   InverseSubBytes;
   AddRoundKey;
   InverseMixColumns;
}
InverseShiftRows;
InverseSubBytes;
AddRoundKey;
```

Of the four basic operations used in the AES, *AddRoundKey* is an involutory operation and can be realized using simple 2-input XOR gates. The *ShiftRows* (and the *InvShiftRows*) perfoms a fixed permutation of its input. For architectures using a full 128-bit data path, *ShiftRows* can be realized without any hardware, simply by interconnection wires. As a result an efficient hardware realization of an AES algorithm depends mainly on the implementation of the remaing two operations: *SubBytes* and *MixColumns* (and their inverses).

In systems where both en- and decryption are supported, significant area can be saved by sharing common parts of the *SubBytes* [7] or *MixColumns* [6] operations. However, such systems can be used for either encryption or decryption at a given time, but not both.

[1]The original AES specification required block sizes of 128, 192 and 256 bits and Rijndael supports all three data block lengths. However the AES standard supports only 128-bit data blocks.

One problem with systems that implement both en- and decryption is that the decryption operation is more difficult than the encryption operation, mainly due to the fact that the *InvMixColumns* operation is more complicated than the *MixColumns* operation. The *MixColumns* operation uses modulo multiplications with the constants 0x03, 0x01 and 0x02 whereas the *InvMixColumns* operation requires modulo multiplications with the constants 0x0e, 0x0b, 0x0d and 0x09 which inherently leads to more complex hardware.

As the overall system throughput will be dictated by the slowest of encryption and decryption operations, the latter will be the limiting factor in an AES implementation.

## 2.1 Number of hardware rounds

It has been pointed out by several authors [2, 3, 7] that the implementation of non-feedback AES modes allows for almost the full set of high-level architectural tradeoffs that are available to the ASIC/FPGA designer at the round-level. Significant (linear) speedup can be achieved by simply replicating hardware to process subsequent data blocks in parallel. Alternatively, loop-unrolling combined with pipelining between the subsequent rounds (or to a certain degree also within a round) processes subsequent blocks in an interleaved fashion.

However, as mentioned earlier, these improvements are only relevant for electronic code book (ECB) and counter (CTR) modes of operation. The remaining standard modes of operation; output feedback (OFB), cipher feedback (CFB), and cipher block chaining (CBC) can not tolerate the latency generated by such pipelined systems and do not benefit from parallel instantiations of multiple en-/decryption rounds. In order to increase the throughput for all modes of operation, the only solution is to reduce the critical path of the en/decryption rounds. Due to area constraints this is also often the only practical approach, even for non-feedback operation.

## 2.2 Key generation

For each round of the en/decryption operation a separate 'round key' is required. In encryption mode they are derived iteratively from the main key using an algorithmic key expansion. The aforementioned key expansion algorithm is similar in structure to the en-/decryption round transformation and uses the same *SubBytes* function that is used for encryption. As the decryption process is the inverse of encryption, the round keys need to be derived iteratively in reverse order, starting with the last round key.

There are basically two alternatives for implementing key management: The keys can be generated in parallel with the round transformations, or the round keys can be generated once and stored in memory to be used later on.

Figure 1 shows three possible variations to generate the round keys for an AES system with both en-/decryption units. A shared round key memory can be used to provide both the en- and decryption datapath with round keys (Figure 1a). In this case the same key will be used for en-/decryption. Since the round keys can be calculated prior to the actual round transformations, the key generator hardware does not have to be designed to calculate round keys at the rate of round transformations. Without this throughput constraint a more area efficient key generator can be implemented.

Generating keys on the fly, increases the so called 'key agility', the capability to change keys rapidly. Especially for high throughput systems, attention must be paid to the design of on-the-fly round key generators in order to keep the length of the critical path of the key generator on par with that of the round transformations. It is interesting to note that, for the specific technology and method used, the area required for storing and generating the roundkeys are similar. To generate the round keys on-the fly for a decryption datapath

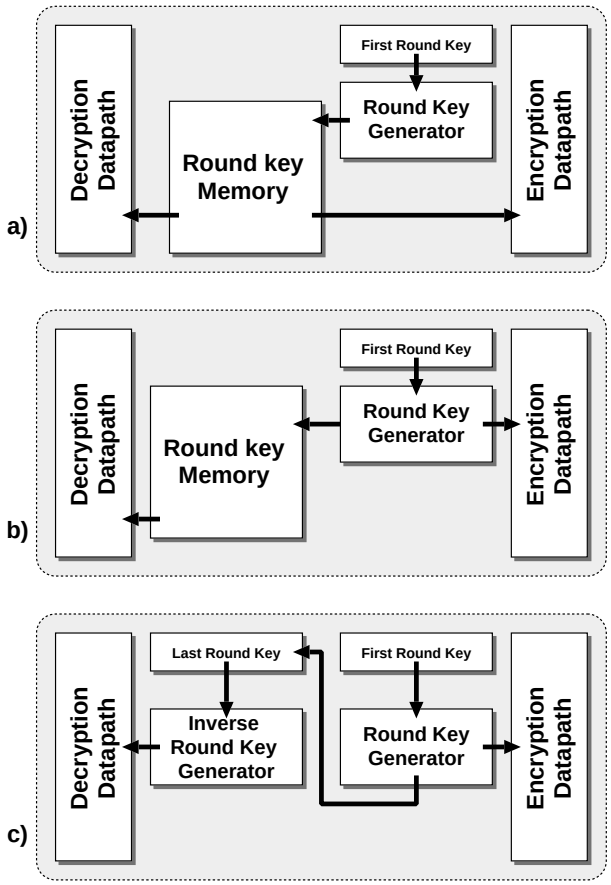


**Figure 1: Three variations of the round key generation organization for a system with parallel en/decryption datapaths: a) shared round key memory, b) on-the-fly key generation for encryption, memory access for decryption c) separate on-the fly key generators for en/decryption**

the last round key must be present. If a new main key for the decryption operation is to be used, the last key must be determined by using the 'forward' key generator first (Figure 1c). Such a system has the advantage of using two separate keys for en/decryption at the same time.

A hybrid solution that uses an on-the-fly key generation unit to generate the round keys for the encryption datapath and stores the round keys in a memory to be used by the decryption unit is also feasible (Figure 1b). Such a system can still be engineered to use two separate keys: one constantly generated by the key generation unit and used by the encryption datapath, and the other precalculated and stored in the memory to be used by the decryption datapath. As opposed to the solution presented in Figure 1a, only a single-port memory is required.

# 3. RTL OPTIMIZATION

With an appropriate high-level architectural choice, register transfer level optimization aims at reducing the longest path to obtain the highest possible clock rate and possibly at reducing power consumption under a given area constraint.

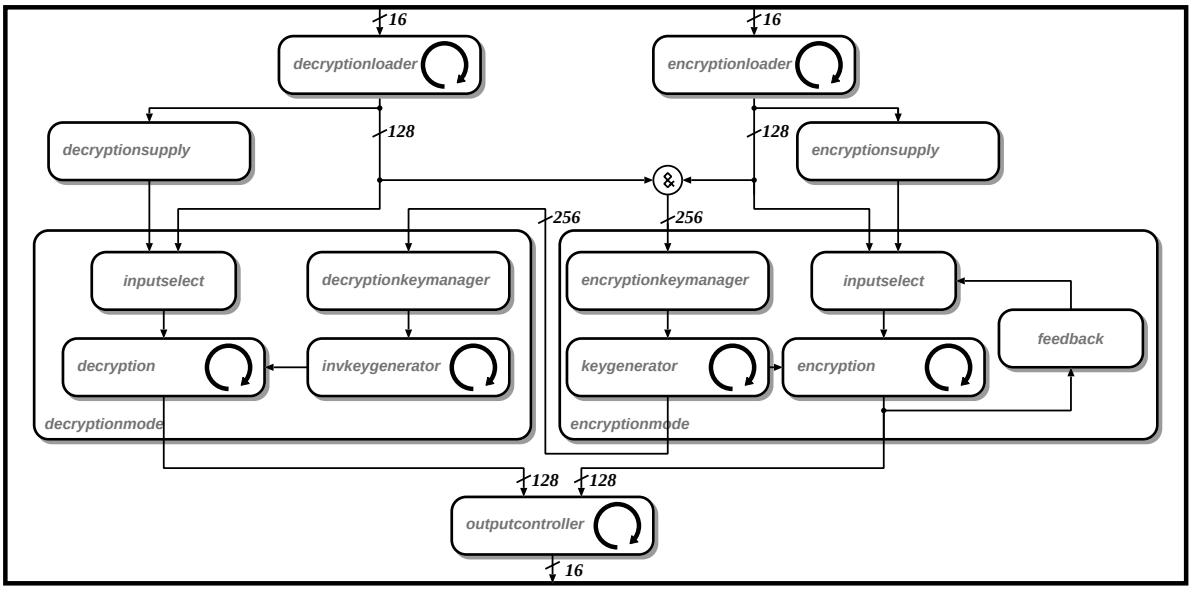


**Figure 2: Block Diagram of Fastcore**

## 3.1 Round reorganization

The AES algorithm consists of a series of almost identical sets of operations that are repeated. Although the algorithm specifies a grouping as to what constitutes an en-/decryption round, for hardware realizations different groupings can be obtained by changing the order of specific operations and/or by moving the starting point of the round. This technique has been used in [7] to balance the delay of intra-round pipeline stages of a partially shared en/decryption system.

Reorganizing the round structure can not directly reduce the delay of the entire calculation. Such changes may, however, result in a structure where subsequent operations can be partially or fully merged for more efficient implementations.

The final round of the AES algorithm is a special case. Depending on the grouping used, it contains one or more fewer operations than a regular round. As a result either the hardware round is constructed in a way that allows to 'skip' certain operations when the last round is being performed, or the last round has to be implemented separately. Multiplexers required for such 'skip' operations are usually within the critical path and reduce the overall throughput.

## 3.2 Look-up tables

For a number of AES functions, look-up tables form an alternative solution to random logic. Whether it makes sense or not to use a LUT depends on the choice between FPGAs and ASICs, and on other circumstances. When compared to ASICs, the reconfigurability of programmable logic devices inflates their transistor counts, die areas, gate delays and interconnect delays. While this used to be true for both random logic and memories, the incorporation of hardwired RAMs in more recent FPGAs has put FPGAs roughly on par with ASICs as far as on-chip memories are concerned. Within the area required by a RAM to store some given number of bits, however, mask-programmed ASIC technology continues to accommodate substantially more gate-equivalents than FPGAs can. In summary, today's FPGA architectures tend to be more in favor of LUTs than ASICs are.

**Table 1: Three alternative realizations for the *SubBytes* function, numbers for UMC 0.25 $\mu$m CMOS.**

| | Area ($\mu$m$^2$) | Delay (ns) |
|---|---|---|
| 1) LUT stored in a 256x8 SRAM | 150.000 | 3.5 |
| 2) LUT synthesized to random-logic | 33.000 | 1.5 |
| 3) Arithmetic-based circuitry [12] | 12,000 | 4.0 |

Let us study three different options for implementing the *SubBytes* (and its inverse *InverseSubBytes*) operation of the AES standard. Comparative results are given in Table 1.

1. The most innocent idea is to store the truth table as LUT in a dedicated SRAM structure rightaway. While this approach is efficient and popular with FPGA-based AES implementations [8], there exist much better alternatives in the occurrence of mask-programmed ASICs.

2. The second option makes use of logic synthesis to turn the same truth table into a random logic network (in essence the 8-bit *SubBytes* function can be expressed as a 8bit-input 8bit-output boolean function). As can be seen from Table 1, this approach results in the fastest of the three circuits. More on the negative side, the area and delay of such gate-level LUTs is often hard to predict and strongly content-dependent. In the case of AES, experiments also showed that random-logic netlists so generated tend to result in very high placement and routing overhead.

3. The *SubBytes* operation and its inverse *InverseSubBytes* involve calculating a multiplicative inverse in the GF($2^8$) followed by an affine transformation. As opposed to options 1) and 2) which made no assumption on the nature and structure of the function to be implemented, the third option takes advantage of mathematical insight. A particularly efficient

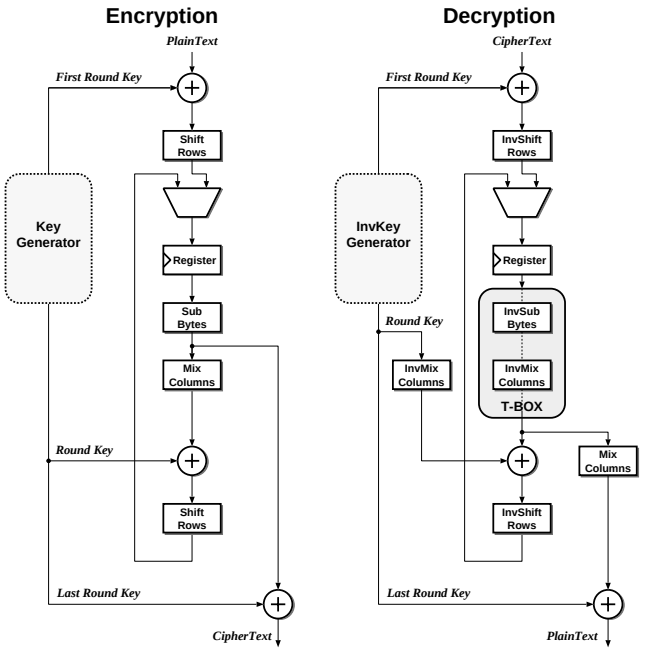


**Figure 3: Encryption (left) and Decryption (right) round structure of the Fastcore**

implementation of *SubBytes* has been presented by Wolkerstorfer [12]. It transforms the multiplicative inverse operation into several GF($2^4$) operations and forms the starting point for circuit 3) in Table 1.

# 4. FASTCORE: A FAST AND BALANCED AES IMPLEMENTATION

"Fastcore" was designed to be an efficient and complete implementation of the AES encryption and decryption algorithms using 128-bit data blocks and programmable key sizes of 128, 192 and 256 bits. All standard AES operation modes are supported (ECB, OFB, CFB and CBC).

Encryption and decryption are realized using two independent units with dedicated on-the-fly key generators (Figure 2). While this architecture appears to be slightly larger than the hybrid solution discussed in section-2.2, it results in a more symmetric architecture and does not require significant amounts of storage. The initial key for the decryption key generator is the last 256-bit state used to generate the last encryption round key.

Since two independent key generators are used, Fastcore can be programmed to operate with different keys for en-/decryption while using the ECB and CBC modes. Note that, CFB and OFB modes of operation does not require AES decryption. While operating in these modes, Fastcore uses only the encryption core for realizing both encryption and decryption operations.

The core of the system has been designed to operate fully parallel with 128-bits, and the two independent datapaths can encrypt and decrypt simultaneously. However, the fabricated version uses two 16-bit inputs and and a shared 16-bit output, to keep the I/O pin count within allocated resources[2].

[2]Fastcore has been designed and sent to fabrication as part of a semester project at the Integrated Systems Laboratory of the Swiss Federal Institute of Technology (ETH). Due to the arrangement with the IC manufacturing foundry for multi-project wafers, such designs have stringent area limitations that also limit the available IO pins.

**Table 2: Throughput of Fastcore with different key lengths (All operation modes). The internal throughput is the sum of en-/decryption datapaths. This internal bandwidth is restricted by I/O limitations.**

| | Key length | | |
|---|---|---|---|
| | 128 | 192 | 256 |
| Encryption | 2.12 Gb/s | 1.77 Gb/s | 1.52 Gb/s |
| Decryption | 2.12 Gb/s | 1.77 Gb/s | 1.52 Gb/s |
| En-/Decryption | 2.65 Gb/s | 2.65 Gb/s | 2.65 Gb/s |
| Internal | 4.24 Gb/s | 3.54 Gb/s | 3.04 Gb/s |

As a result of this arrangement, Fastcore can not reach the full-bandwidth for simultaneous en-/decryptions. Table 2 summarizes the achievable throughputs in various modes of operation and compares them to the internal (I/O independent) throughput of Fastcore.

## 4.1 Encryption datapath

In the encryption datapath, the *ShiftRows* and *SubBytes* operations, which are independent of each other, are exchanged. Furthermore, the round register has been relocated to the start of the round to relax the timing constraint on the otherwise critical path through the first round. At first sight, this arrangement appears to be costly as the *ShiftRows* operation needs to be replicated. Fortunately, this operation is realized by wiring only, and its replication comes at no extra cost. For the last round, rather than using a multiplexer to bypass the *MixColumns* operation, the final *AddRoundKey* operation is replicated after the *SubBytes* operation (see Figure 3).

## 4.2 Balancing en-/decryption

If no additional algorithmic optimizations are made, the corresponding decryption round will exhibit a longer critical path, in a straightforward implementation by as much as 30%, simply because the *InvMixColumns* operation is more complex to implement. Therefore, it was decided to optimize the critical path through the decryption unit to match the path through the encryption unit.

One possible way to reduce the critical path is to combine this operation with another operation, with *InvSubBytes* being the only feasible candidate. Unfortunately, in the standard decryption path these two operations are separated by the *AddRoundKey* (Figure 4a).

Figure 4 illustrates the optimization steps that have been undertaken to reduce the critical path of the decryption datapath:

- The *AddRoundKey* operation is moved to after the *InvMixColumns* operation (Figure 4b). Functional equivalence to the original algorithm is maintained by applying the *InvMixColumns* transformation to the output of the round key generator, at additional hardware cost.

- In the original configuration, the last round output was conveniently taken after the final *AddRoundKey* operation. With the relocation of this operation, the need to skip the *InvMixColumns* operation arises for the last round. Using a configuration, much similar to that of encryption, the *AddRoundKey* operation (which has comparatively little hardware overhead) is replicated on the last round path.

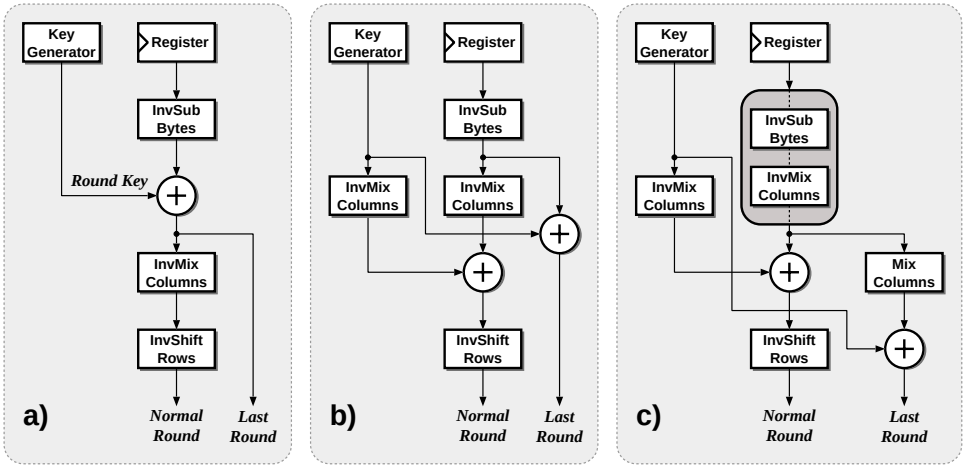


**Figure 4: Transformation of the decryption round structure: a) initial state, b) after relocating *AddRoundKey*, c) final stage, combining *InvMixColumns* and *InvSubBytes*.**

- With the *AddRoundKey* operation out of the way, the *InvSubBytes* and the *InvMixColumns* operation can be merged into a common lookup table, called a T-BOX (Figure 4c). Essentially four 8-bit *InvSubBytes* operators and a 32-bit *InvMixColumns* operation are optimized together. This organization can reduce the critical path at the expense of larger area. Note that a similar transformation could also be used to reduce the critical path for the encryption datapath results.

- This last optimization has the undesired consequence of making the result of the *InvSubBytes* operation inaccessible for the last round. The output of the T-BOX contains an additional *InvMixColumns* operation that must be reversed for the last round. There are two approaches to this problem. The output for the last round can be taken before the T-BOX and the *InvSubBytes* operation can be replicated or the output can be taken from the output of the T-BOX and an additional *MixColumns* operation is used to reverse the undesired *InvMixColumns* operation. In Fastcore the latter alternative was implemented, as the overhead of an additional *MixColumns* is much less than that of an additional *InvSubBytes* operation.

The final round structure for encryption and decryption can be seen in Figure 3. The resulting decryption round has a critical path that is roughly 27% shorter than the original one, matching the critical path to that of the encryption round. The final design has the desired balanced throughput for both its encryption and decryption datapaths. However, this comes at the expense of 35% more active area in the decryption path.

## 4.3 Implementation

The chip micrograph shown in Figure 5 highlights the en- and decryption engines. It can be clearly seen that the decryption round occupies more area than the encryption round due to the added optimizations to increase its throughput.

The design has been implemented using a 0.25 $\mu$m 5 Metal CMOS process and occupies a core area of 3.56 mm$^2$. The fabricated chips have been tested extensively. The throughput figures given in Table 2 are values measured for the slowest of the 10 fabricated samples.

While running a benchmark using all available modes of operation at 166 MHz using a 2.5 V supply, Fastcore consumes 600 mW power. Table 3 shows the simulated dynamic power consumption breakdown of Fastcore. Roughly one third of the power is consumed in either datapath. It is interesting to note that the on-the-fly key generators consume around one third of the power of an en-/decryption datapath.

**Table 3: Power consumption breakdown**

| Instance | % |
|---|---|
| chip | 100.0 |
| +-> clock tree | 7.8 |
| +-> output buffers | 2.4 |
| +-> core | 88.8 |
| +---> decryption | 34.4 |
| +-----> inv.key generation | 12.5 |
| +---> encryption | 34.5 |
| +-----> key generation | 11.8 |
| +---> I/O buffers | 12.2 |
| +---> mode support | 7.7 |

Table 4 compares the main design parameters of Fastcore to some recently presented AES implementations. Note that the design presented in [11] and [5] are burdened by supporting block lengths of 192 and 256 bits that were part of the original Rijndael specification but are not in the AES standard. For these designs the throughput figure corresponding to 128 bit block lengths is given. Fastcore contains additional hardware and interconnection to support all standard operational modes without external components. The throughput figure of Fastcore is only bettered by [10] which uses a 4 stage pipeline and thereby can not achieve the same throughput for the feedback modes, and by [9] which reports a simulated throughput of 2.6 Gb/s using a much more advanced technology.

# 5. CONCLUSIONS

In this paper, basic architectural issues concerning the design of efficient ASIC implementations have been reviewed. In light of these guidelines, a throughput-optimized AES implementation named Fastcore has been designed and fabricated. The AES imple-

**Table 4: Main parameters of Fastcore, compared to other reported ASIC realizations.**

| | This work | Verbauwhede [11] | Kim [5] | Satoh [9] | Su [10] | Lu [6] |
|---|---|---|---|---|---|---|
| Technology | 0.25$\mu$m | 0.18 $\mu$m | 0.18 $\mu$m | 0.11 $\mu$m | 0.25 $\mu$m | 0.25$\mu$m |
| Throughput | 2.12 Gb/s | 1.6 Gb/s | 1.64 Gb/s | 2.6 Gb/s | 2.97 Gb/s | 0.61 Gb/s |
| Core Area | 3.56 mm$^2$ | 3.96 mm$^2$ | N/A | 0.205 mm$^2$ | 1.62 mm$^2$ | N/A |
| Gate Equivalents | 119.000 | 173.000 | 28.626 + 4Kb RAM + 128Kb ROM | 21.337 | 63.400 + 4Kb RAM | 31.957 |
| Power Consumption | 600 mW | 56 mW | 314 mW | N/A | N/A | N/A |
| Clock Frequency | 166 MHz | 154 MHz | 465 MHz | 224.22 MHz | 250 MHz | 100 MHz |
| Directly Supported Modes | ECB, OFB, CFB, CBC | ECB | ECB | ECB | ECB | ECB |
| En-/Decryption | Both | Encryption | Both | Both | Both | Both |
| Notes | Parallel En-/Decryption | Supports 256 bit blocks | Supports 256 bit blocks | Synthesis results only | Pipelined architecture | Synthesis results only? |


**Figure 5: Chip micrograph of the Fastcore showing the encryption and decryption regions.**

mentation has been measured to achieve 2.12 Gb/s throughput using 128-bit keys for all modes of operation for both encryption and decryption in a relatively mature 0.25 $\mu$m CMOS technology using only 3.56 mm$^2$ core area. In terms of raw measured throughput performance for all modes of operation Fastcore compares fairly well to other published AES implementations

## 6. REFERENCES


[1] J. Daemen and V. Rijmen. *The design of Rijndael: AES–The Advanced Encryption Standard*. Springer-Verlag, 2002.

[2] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based Performance Evaluation of the AES Block Cipher Candidate Algorithm FInalists. *IEEE Transactions on VLSI Systems*, 9(4):545–557, Aug. 2001.

[3] K. Gaj and P. Chodowiec. Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays. In *Proc. RSA Security Conf. San Francisco, CA*, pages 84–99, Apr. 2001.

[4] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware Evaluation of the AES Finalists. In *Proc. 3rd AES Candidate Conf., New York*, pages 279–285, Apr. 2000.

[5] N. S. Kim, T. Mudge, and R. Brown. A 2.3 Gb/s Fully Integrated and Synthesizable AES Rijndael Core. In *Proc. IEEE Custom Integrated Circuits Conference*, pages 193–196, Sept. 2003.

[6] C.-C. Lu and S.-Y. Tseng. Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter. In *Proc. Application-Specific Systems, Architectures and Processors*, pages 277–285, July 2002.

[7] A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner. 2 Gb/s Hardware Realizations of RIJNDAEL and SERPENT: A comparative analysis. In *Proc. Cryptographic Hardware and Embedded Systems - CHES 2002, LNCS 2523*, pages 144–158. Springer-Verlag, Aug. 2002.

[8] M. McLoone and J. V. McCanny. Rijndael FPGA Implementations Utilising Look-Up Tables. *Journal of VLSI Signal Processing*, 34(3):261–275, July 2003.

[9] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *Proc. ASIACRYPT 2001, LNCS 2248*, pages 239–254. Springer-Verlag, 2001.

[10] C.-P. Su, T.-F. Lin, C.-T. Huang, and C.-W. Wu. A Highly Efficient AES Cipher Chip. In *Proc. of Asia and South Pasific Design Automation Conference ASP-DAC 2003*, pages 561–562, Jan. 2003.

[11] I. Verbauwhede, P. Schaumont, and H. Kuo. Design and Perfomance Tesing of a 2.29-GB/s Rijndael Processor. *IEEE Journal of Solid-State Circuits*, 38(3):569–572, Mar. 2003.

[12] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES S-boxes. In *Proc. RSA Security Conf. San Jose, CA*, pages 67–78, Feb. 2002.