Low–Level Space Optimization of an AES Implementation for a Bit–Serial Fully Pipelined Architecture

Raphael Weber[1] and Achim Rettberg[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany
[2] Carl von Ossietzky University Oldenburg, OFFIS, Escherweg 2, 26121 Oldenburg, Germany

Abstract. A previously developed AES (Advanced Encryption Standard) implementation is optimized and described in this paper. The special architecture for which this implementation is targeted comprises synchronous and systematic bit–serial processing without a central controlling instance. In order to shrink the design in terms of logic utilization we deeply analyzed the architecture and the AES implementation to identify the most costly logic elements. We propose to merge certain parts of the logic to achieve better area efficiency. The approach was integrated into an existing synthesis tool which we used to produce synthesizable VHDL code. For testing purposes, we simulated the generated VHDL code and ran tests on an FPGA board.

1 Introduction

People's demand to keep secrets, only accessible to chosen people, is as old as mankind. In order to keep something secret one has to make sure that only trustworthy people can understand the secret's contents. The most popular cipher algorithm is the Advanced Encryption Standard (AES) announced by the U.S. American National Institute of Standards and Technology (NIST) in late 2000.

In this paper we analyze the AES implementation for a special bit–serial, reconfigurable, fully pipelined, self–controlled architecture, covered in [11]. Our goal is to optimize the AES implementation targeted for resource restricted environments in terms of hardware usage.

Bit–serial architectures have the advantage of a low number of input and output lines leading to a low number of required pins. In synchronous design, however, the performance of these architectures is affected by the long wires, which are used to control the operators or the potential gated clocks. Nowadays, the wire delay in chip design is near to a break with the gate delay. Solutions to overcome this drawback are required. Basically, long control wires can be avoided by a local distribution of the control circuitry at the operator level. A similar approach is used for the architecture described in this work.

While the design of a fully interlocked asynchronous architecture is well understood, realizing a fully synchronous pipeline architecture still remains a difficult task. Through a one-hot implementation of the central control engine, its folding into the data path, and the use of a shift register, we realized a synchronous fully self–timed bit–serial and fully interlocked pipeline architecture called MACT (MACT = Mauro, Achim, Christophe and Tom).

The paper is organized as follows. In Section 2 we will shortly explain the AES cipher algorithm and the basics of the MACT architecture. In Section 3 we analyze the MACT AES implementation and present our low–level space optimization including a description of our modifications. Finally, Section 4 states the optimization results, sums up with a conclusion and gives an outlook.

2 Basics

2.1 The Advanced Encryption Standard

AES is a block cipher algorithm which has a constant input/output block size of 128 bits. Data is encrypted in a differing number of loops in which four transformations are applied to the block, called *state*. The number of loops depends on the key size which can either be 128, 192, or 256 bits. In this work we will only consider the AES-128 with a 128-bit key and 10 loops (rounds).

Figure 1 displays how the cipher works, utilizing four transformations, described below. The roundKey is generated from the key and changes each round. This procedure is called key expansion.

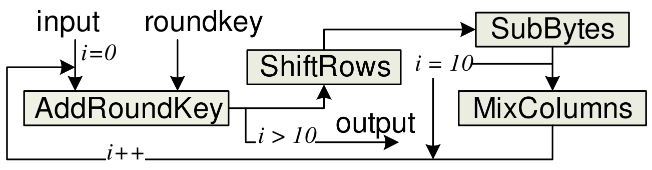


Fig. 1. AES-128 cipher.

`AddRoundKey` XORs the state byte–wise with the current round key. The RoundKey applied before the loop is equal to the key. The byte–wise `SubBytes` transformation is the most costly operation in terms of hardware utilization. First, each byte is considered as an element in the Gallois Field (GF(2^8)) and the multiplicative inverse is calculated. Second, an affine transformation is applied to the byte. This results in a highly non linear mapping, which can be stored in a so called S-Box. `SubBytes` can be implemented using combinational logic only using simple bit–wise XOR and AND operators [5, 9]. Other implementations use a look–up–table [2, 10, 1]. `ShiftRows` cyclically shifts the bytes of a row over a differing number of offsets. `MixColumns` considers the bytes in each column of the state as coefficients in GF(2^8) and performs GF(2^8) multiplication and XOR operations to the bytes of all four columns of the state. A multiplication in GF(2^8) can be performed by a series of shift lefts with a conditional XOR with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1 =$ `{01}{1b}`.

2.2 The MACT Architecture

MACT is an architecture that breaks with classical design paradigms. Its development came in combination with a design paradigm shift to adapt to market requirements. The architecture is based on small and distributed local control units instead of a global control instance. MACT is a synchronous, de-centralized and self-controlling architecture. Data and control information is combined into one packet which is shifted through a network of operators using one single wire only (refer to Figure 2). To our knowledge, this is the second approach to implement a fully interlocked synchronous architecture after that of [4] and the first one which does not rely on gated clocks to realize the local control of operators.

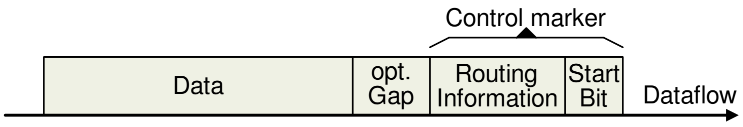


Fig. 2. Example data packet.

The controlling operates locally, only based on arriving data. Thus, there are no long control wires, which would limit the operating speed due to wire delays [6]. This enables a high frequency. Yet, the architecture operates synchronously, thus enabling accurate estimation of the latency, etc. a priori. To overcome the increased latency of the bit–serial operation, MACT uses pipelining, i. e., there are no buffers, operators are placed following each other immediately. MACT implementations are based on data flow graphs. Nodes of these graphs are directly connected, similar to a shift register.

We consider the flow of data through the operator network as processing in waves, i. e., valid data alternates with gaps. Additionally, we have to ensure that the control marker is not modified by an operator. This can be achieved by the two additional signals *open bypass* and *close bypass*. If *open bypass* is true the control marker and the gap of the data packet are routed around the operating unit inside the operator. If *close bypass* is true the data of the data packet is directed to the operating unit.

MACT is characterized by short and local control wires and no necessity to implement costly parallel/serial decoders or encoders. Thus, it may run with high speed, compensating the drawbacks of bit–serial processing. Furthermore, the local control structure avoids complex controllers. Additionally, the fully interlocked pipeline allows the architecture to support multiple applications within one implementation. The architecture is described in more detail in [8, 7].

In order to realize reconfiguration within our architecture a component called router was developed. The router offers path selection, which can be controlled by the extension of the control marker in the data packet. That means, the control marker contains the routing information, see Figure 2. The realization of loops can also be achieved with routers.

3 Low–Level Space Optimization of the Implementation

MACT is a data flow oriented architecture, logic circuits can be generated from a data flow graph specification by a high level synthesis tool. We used this tool to draw our data flow graphs for all AES components including the key expansion in order to get a working prototype [11]. This might have been a straight forward realization for the MACT architecture. However, while analyzing and testing the design, we discovered that it was not as space–saving as we had expected.

When dealing with bit–serial designs one might expect small operators and few input/output pins. While the latter applies for MACT, the first does not necessarily. Since each MACT operator contains not only the actual operator logic but also the control logic, it naturally results in a higher hardware utilization, when compared with a bit–parallel operator of the same size without control logic. Our combinational S-Box implementation uses a huge amount of simple bit–wise operators like ANDs and XORs, which contribute to the size.

3.1 Analyzing the Design

After we implemented and tested the combinational S-Box we synthesized it, utilizing a Spartan 3 FPGA evaluation board with the Software Xilinx ISE. The synthesis report stated a total of 467 occupied slices and 713 utilized 4–input–look–up–tables. To us, this seemed a rather high device utilization.

We discovered a high level of concurrently operating MACT operators, each of which receiving its own control signals, even if they arrive at the same time. Two operators run concurrently, if the packets they process have been synchronized at some point in the data flow graph and stay synchronized.

When taking a closer look at the operators, one can distinguish between the logic and the control part of the operator. The latter is based on basic principles of the MACT architecture. Special signals are a result of the design of the architecture, such as *close_bypass*, *open_bypass*, stall, reset and clock signals. Figure 3 displays an XOR operator on the register transfer level (RTL).

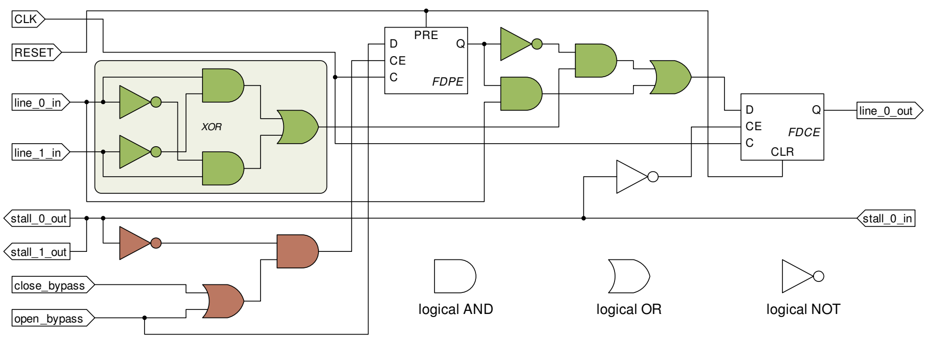
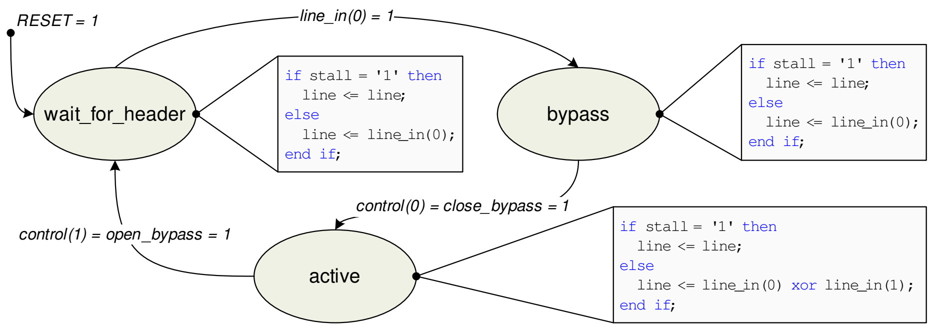


Fig. 3. Register transfer level design of the logical XOR operator.

The framed brightened part in the figure denoted by *XOR* represents the logic operators for the logical XOR. The rest of the logic in Figure 3 is dedicated to control handling. The operator's logic is necessary, but when two or more operators receive similar control signals, which eventually result in the same control behaviour, it might be possible to reduce the number of control signals needed to get the same behaviour. We propose a new way to exploit similar control signals of concurrently processing operators.

3.2 Merging Operators

In order to use the same control logic for different operators we part the control signal processing from the operator logic. This was done via a new modified Finite State Machine (FSM) design (see Figure 4). The separation improves the code analysis and processing of the high level synthesis, which was modified to comply with the new FSM VHDL code of the MACT operators. Thus, we can replace the operator logic by any other logic without touching the control logic.



```
if stall = '1' then
  line <= line;
else
  line <= line_in(0);
end if;
```

```
if stall = '1' then
  line <= line;
else
  line <= line_in(0);
end if;
```

```
if stall = '1' then
  line <= line;
else
  line <= line_in(0) xor line_in(1);
end if;
```

Fig. 4. Finite State Machine representation of the MACT XOR.

With our new FSM design, it is possible to retrieve the relevant information of a new operator from the VHDL source file. This approach can not completely replace the old data flow graph nodes, it merely combines a category of operators to a more abstract representation. This categorization is done in order to collect similar nodes of the same category and unite them to a new *merged* MACT node. A merged node has multiple inputs and outputs from several operators, but receives and processes its control signals only once, since it contains only one logic unit to handle its control signals.

For the merging of nodes to work correctly, the data flow graph has to be analyzed, since this approach is only applicable under certain circumstances. Operators only receive similar control signals when they are synchronized and have the same packet layout and duration. This applies for large parts of the S-Box. First off, we assume that the targeted data flow graph has been analyzed so that the synchronization information is available, this includes synchronized classes as described in our previous work in [3]. Our approach can be decomposed into three steps:

– look for all synchronized operators in the same category and the same synchronized class, for example XORs, ANDs, ORs, etc.
– replace these sets of operators by a single new merged node with as many in- and outputs as the operators in the associated set, store the information in the merged node for code generation
– generate the merged nodes with the FSM VHDL interface as described above

"Information" in the second step can for example refer to the mathematical representation of the operator, or the way the routing information is handled. This information can be stored in comments. Later on, the generated interfaces can be parsed using the very same function as for parsing MACT operators to retrieve operator information. Thus, our approach is scalable and extensible for future reuse.

3.3 Optimizing the Implementation via Merged Nodes

We implemented and integrated the conceptual approach explained in the last subsection into the high level synthesis tool and generated the AES cipher algorithm utilizing the new merged nodes.

As an example, where it can be observed what exactly changed, we applied our merging nodes optimization to the isomorphic mapping δ, which is a part of the combinational S-Box. Figure 5 displays the data flow structure of δ including the reduced amount of control signals for the operators. The b_8 till b_{20} represent XOR operators, the D's denote delay elements.

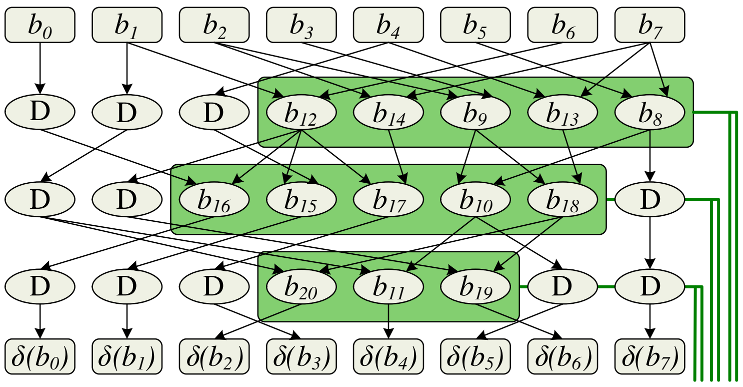


Fig. 5. Data flow graph of the optimized isomorphic mapping with merged nodes.

There are three merged nodes (the dark backgrounded shapes) containing 5, 5, and 3 XORs. For example the lower merged node only needs 2 control signals instead of 6 which results in a smaller control logic. Applying the merging to the other AES transformations resulted in less optimization possibilities. Nonetheless, the `AddRoundKey` and `MixColumns` transformations use some XORs, which have been merged. The next section will state the results of the prototype and the optimized implementation, compare them and draw a conclusion.

4 Results and Conclusion

We synthesized our implementation for an inexpensive Xilinx Spartan 3 board, running at 50 MHz. One AES round takes 62 cycles, capable of processing two blocks at once. The packets are 13 bits long, so the minimum loop duration is 26 cycles (the minimum gap between packets is also 13). The logic (including an RS232 interface) utilizes 4,745 of the 4-input LUTs.

With a 50 MHz clock frequency and encrypting 128 bit in a total of 626 clock cycles we calculate a throughput of 9.75 MBit per second. As we stated earlier, the S-Box has quite some parallelism in it. We minimized the number of control signals by merging logic nodes. Table 1 shows a direct comparison between the prototype and the optimized S-Box. The maximum clock frequency has improved by 38.3% from 141.824 MHz to 196.155 MHz.

Logic utilization	prototype	optimized	reduced perc.
No. of occupied Slices	467	327	30.0%
No. of Slice FF	646	543	15.9%
Total no. 4–input LUTs	713	491	31.1%

Table 1. Comparison between the prototype and the optimized S-Box.

The table indicates an improvement of $\approx$ 30%. Occupied slices have been reduced by 30.0%. The number of occupied slice flip–flops has been reduced by only 15.9%. Table 2 shows the comparison between prototype and optimized AES-128. The maximum clock frequency shrunk by 24%.

Logic utilization	prototype	optimized	reduced perc.
No. of occupied Slices	3616	2960	18.1%
No. of Slice FF	5902	4663	21.0%
Total no. 4–input LUTs	4745	2905	38.8%

Table 2. Comparison between the prototype and the optimized AES-128.

The space reduction is $\approx$ 25% on average. As can be seen, the reduction in percent is about the same for the S-Box and the complete AES-128. This is due to the fact that the S-Boxes in the AES-128 make up the most costly part.

Special data computation algorithms proved to offer high optimization potential for our space reduction algorithm. They produced excellent results for the AES implementation, especially the combinational S-Box. Within the scope of this paper, we did not describe all optimization possibilities, we merely restricted the space reduction to the most important MACT parts. However, the prototype and optimized AES-128 algorithm proved the MACT architecture and the MHLS tool being capable of implementing and handling very complex designs.

In our opinion, MACT has a high potential, but scheduling and optimizing remains a difficult task. Future work could focus on a deeper analysis of MACT's unique properties and possible applcations to give new research directions.

References

1. K. Atasu, L. Breveglieri, and M. Macchetti. Efficient AES implementations for ARM based platforms. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 841–845, New York, NY, USA, 2004. ACM.
2. P. Chodowiec and K. Gaj. Very Compact FPGA Implementation of the AES Algorithm. In *Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), number 2779 in Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2003.
3. F. Dittmann, A. Rettberg, and R. Weber. Optimization techniques for a reconfigurable self-timed and bit-serial architecture. In *Proceedings of the SBCCI 2007*, Rio de Janeiro, Brazil, 3 - 6 Sept. 2007.
4. H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *8th Intern. Symposium on Asynchronous Circuits and Systems*, Apr. 2002.
5. E. N. Mui. Practical Implementation of Rijndael S-Box Using Combinational Logic. Available from: `http://www.xess.com/projects/Rijndael_SBox.pdf`, 2007.
6. D. Renshaw and P. Denyer. *VLSI Signal Processing: A Bit Serial Approach*. Addison-Wesley, 1985.
7. A. Rettberg, F. Dittmann, M. C. Zanella, and T. Lehmann. Towards a high-level synthesis of reconfigurable bit-serial architectures. In *Proceedings of the 16th Symposium on Integrated Circuits and System Design (SBCCI)*, Sao Paulo, Brazil, 8 - 11 Sept. 2003.
8. A. Rettberg, M. C. Zanella, C. Bobda, and T. Lehmann. A fully self-timed bit-serial pipeline architecture for embedded systems. In *Proceedings of the Design Automation and Test Conference (DATE)*, Messe Munich, Munich, Germany, 3 - 7 Mar. 2003.
9. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 239–254, London, UK, 2001. Springer-Verlag.
10. E. Trichina and L. Korkishko. Secure and Efficient AES Software Implementation for Smart Cards. In *WISA '04: 5th Workshop on Information Security Applications 2004*, pages 425–439. Springer-Verlag, 2004.
11. R. Weber and A. Rettberg. Implementation of the AES Algorithm for a Reconfigurable, Bit Serial, Fully Pipelined Architecture. In *Reconfigurable Computing: Architectures, Tools and Applications, 5th International Workshop, ARC 2009. Proceedings*, pages 330–335, Karlsruhe, Germany, 16 - 18 Mar. 2009.