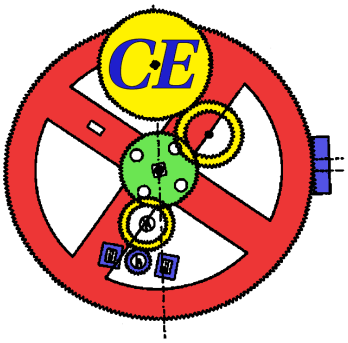


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2004

MSc THESIS

The AES targeted on the MOLEN processor

CE-MS-2004-03

Abstract

These days, a lot of network security services and systems are implemented, such as Public Key Infrastructure (PKI) systems, web appliances, high-speed routers and Firewalls, that are used for securing information. The communication in these systems is based on various protocols, in which often a combination of various cryptographic algorithms is utilized. The choice for a certain cryptographic algorithm within a communication process, depends on several factors such as company policies on encryption strength and government restrictions on encryption export. Considering these facts and the fact that cryptographic algorithms are relative frequently upgraded, cryptographic flexibility and high speed performance are requirements in network systems. The $\rho\mu$-coded processor, also known as the MOLEN processor[16], provide run-time re-programmability and high speed performance. Therefore, this platform is a suitable candidate for cryptographical network systems. In this thesis, the 128-bit version of the Advanced Encryption Standard [7] cipher is implemented for the MOLEN processor. The main goal of this thesis is to analyze the speed performance of the AES cipher that was targeted on the MOLEN processor.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

The AES targeted on the MOLEN processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Raoel Ashruf
born in Paramaribo, Suriname

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

The AES targeted on the MOLEN processor

by Raoel Ashruf

Abstract

Laboratory : Computer Engineering
Codenumber : CE-MS-2004-03

Committee Members :

Advisor:	Georgi N. Gaydadjiev
Chairperson:	Stephan Wong
Member:	Georgi N. Gaydadjiev
Member:	Stephan Wong
Member:	Arjan van Genderen

To my parents for their endless love and support

Contents

List of Figures **ix**

List of Tables **xi**

Acknowledgments **xiii**

1 Introduction **1**
1.1 Methodology, Objective and Contribution . . . 3
1.2 Framework . . . 4

2 Background Information **5**
2.1 Overview of Cryptography . . . 5
2.1.1 Basic terminology and concepts . . . 5
2.1.2 Ciphering Techniques . . . 6
2.1.3 Cryptanalysis . . . 7
2.2 Related work on the AES cipher . . . 8
2.2.1 AES performances on different platforms . . . 8
2.2.2 Attacks on the AES . . . 11
2.3 The MOLEN architecture . . . 11
2.3.1 The machine organization . . . 11
2.3.2 The MOLEN Instruction Set Architecture . . . 13
2.3.3 Reconfiguration and execution microcode . . . 14
2.4 Chapter summary . . . 14

3 The design of the MOLEN AES **17**
3.1 Introduction . . . 17
3.2 The AES algorithm scheme . . . 17
3.2.1 ByteSub and Inverse ByteSub . . . 20
3.2.2 ShiftRow and Inverse ShiftRow . . . 20
3.2.3 MixColumn and Inverse Mixcolumn . . . 21
3.2.4 AddRoundKey . . . 22
3.2.5 The Key Scheduler . . . 22
3.3 Time critical operations . . . 23
3.3.1 Analysis on arithmetical level . . . 23
3.3.2 Analysis on instruction level . . . 24
3.3.3 Conclusion . . . 27
3.4 The design of the AES Custom Computing Units . . . 27
3.4.1 Matrix multiplication . . . 27
3.4.2 Addition over Galois field . . . 28
3.4.3 Multiplication over Galois field . . . 28

3.4.4 The circuit . . . 30
3.4.5 The design verification . . . 31
3.4.6 The synthesis results . . . 32
3.5 The MOLEN AES . . . 35
3.5.1 Function descriptions of the Transformations . . . 35
3.5.2 The reconfiguration and execution microcode . . . 37
3.5.3 Scheduling the SET and the EXECUTE MOLEN instructions . . . 37
3.6 Chapter summary . . . 38

4 The Simulation Framework **41**
4.1 The SimpleScalar tool-set . . . 41
4.2 Extending SimpleScalar ISA with MOLEN instructions . . . 43
4.3 Simoutorder's structure and extension of this structure . . . 43
4.4 The timing characteristics of MOLEN hardware units . . . 46
4.5 The MOLEN module . . . 47
4.5.1 The arbiter . . . 47
4.5.2 The Custom Computing Unit . . . 47
4.5.3 The $\rho\mu$-code unit . . . 49
4.5.4 Sim-outorder's pipeline extension . . . 51
4.6 The application interface . . . 60
4.6.1 the app_config.c file . . . 60
4.6.2 the hardware.c file . . . 62
4.6.3 the molen_config.c file . . . 63
4.7 Chapter summary . . . 64

5 The MOLEN AES performance results **65**
5.1 The simulation conditions . . . 65
5.1.1 The simulator platform . . . 65
5.1.2 The simulator configuration . . . 66
5.1.3 Compile options and testvector . . . 66
5.1.4 The performance gain metric . . . 67
5.1.5 Dealing with overhead . . . 68
5.2 The performance results . . . 69
5.2.1 The MOLEN AES performance . . . 70
5.3 Chapter summary . . . 71

6 Conclusion and Future Research Directions **73**

Bibliography **78**

A Sim-outorder extended for the MOLEN module **79**

B The MOLEN module **91**

C The MOLEN AES **119**
C.1 rijndael.c 119
C.2 app_config.c 132
C.3 molen_config.c 135
C.4 hardware.c 137

D AES hardware engines VHDL source code **147**

List of Figures

2.1 The MOLEN machine organization 12

3.1 The design methodology for the MOLEN AES 18
3.2 The AES encryption diagram flow 19
3.3 The AES decryption diagram flow 19
3.4 The procedure for obtaining a new round key 23
3.5 The number of executed instruction in terms of percentage for the encryption process 25
3.6 Instruction type that are executed during the encryption process 25
3.7 The number of executed instruction in terms of percentage for the decryption process 26
3.8 Instruction type that are executed during the decryption process 26
3.9 The hardware scheme of processing four bytes 31
3.10 The Mixcolumn block scheme 31
3.11 Design verification of the arithmetical implementation 32
3.12 The diagram flow of the MOLEN AES 39

4.1 The structure of the sim-outorder simulator extended for the MOLEN architecture 44
4.2 The utilization of the application interface 45
4.3 The pipeline stages of the simulation framework 52
4.4 The diagram flow of the MOLEN dispatch routine. 54
4.5 The flow diagram of loading microcode to the ρ-Control Store. 57

5.1 The testvector, represented by 29.276 bytes 67
5.2 The trace routine function 68
5.3 The throughput performances on the MOLEN platform 70

List of Tables

2.1 Performance of software based AES implementations 10
2.2 The performance of various AES hardware implementations running in CBC mode 10

3.1 The offsets that are used in the ShiftRow and Inverse ShiftRow transformations 20
3.2 The maximum operating frequency for the Mixcolumn and Inverse Mixcolumn hardware units 33
3.3 Mixcolumn hardware unit synthesis parameters 34
3.4 Inverse Mixcolumn hardware unit synthesis parameters 34

4.1 The MOLEN SET and EXECUTE instruction 44

5.1 The performance results for the encryption direction 69
5.2 The performance results for the decryption direction 69

Acknowledgments

During the time I was performing my master thesis, I came across many people who have supported and assisted me during this project. I want to thank Prof. dr. Stamatis Vassiliadis for giving me the opportunity to do my Masters at the Computer Engineering Laboratory. I want to thank Georgi Gaydadjiev, who has taught me many things and has supported me during this project. I want to thank Stephan Wong, who provided me his work on the SimpleScalar tool set and thought me many things related to this tool set. I want to thank Bert Meijs for maintaining the computers that I worked on. I want to thank Lidwina Tromp for her support in doing all the paper work. Finally, I want to thank all my friends and family who helped me, making this project a nice experience.

Raoel Ashruf
Delft, The Netherlands
September 24, 2004

1 Introduction

Since privacy issues and network security are emerging due to the wide proliferation of Internet, the research in protecting information is increasing. Cryptographic algorithms, also known as ciphers, forms the fundamental aspect within this research field. The most used and analyzed cryptographic algorithm of the last 20 years, is the Data Encryption Standard (DES) [1]. With the introduction of this cipher in the early 70s, there were several accusations concerning hidden back-doors, not transparent S-boxes and the length of the key. Despite all the criticism, DES became the encryption standard in 1977. In 1983 it was shown [1] that the cipher is vulnerable due to its short key length. Considering the fact that the computing capacity is always increasing, the vulnerability of DES was a thorn in the eye. Therefore, an enhanced version of the cipher was introduced. This enhanced version known as Triple-DES [1] performs DES three times sequentially and therefore it is more secure than DES. However, the speed performance of Triple-DES on software based platforms were not interesting for practical applications. Therefore in 1997, the National Institute of Standards and Technology (NIST) organized a contest in order to develop a new cryptographic algorithm standard which would replace both DES and Triple-DES. More precisely, the main objective was to develop an algorithm that would at least offer the same security level which was provided by Triple-DES, but that should have higher performance than the performance of Triple-DES. Fifteen new block cryptographic algorithms were submitted [14]. On November 26, 2001, the algorithm known as Rijndael (pronounced Rhine-dall) was chosen to be the replacement for DES and since then it is known as the Advanced Encryption Standard (AES). This algorithm satisfies the following National Institute of Standard and Technology (NIST) statement:*"Assuming that one could build a machine that could recover a DES key in a second, then it would take that machine approximately 149 thousand-billion (149 trillion) years to crack a 128-bit AES key. To put that into perspective, the universe is believed to be less than 20 billion years old."*[13]

The development of high speed networks, has directed the research framework of protecting information, to a more broader aspect then that of solely developing ciphers. Cipher performance, key management, policies and reliability aspects are important topics nowadays. These days, a lot of network security services and systems are implemented, such as Public Key Infrastructure (PKI) systems, web appliances, high-speed routers and Firewalls, that are used for securing information. The communication in these systems is based on various protocols, for example the Secure Sockets Layer (SSL) protocol, the IP Security Protocol (IPsec) and the Transport Layer Security (TLS) protocol. Such protocols are not limited to one or two cryptographic algorithms, but they often use a combination of various cryptographic algorithms. The choice for a certain cipher within a communication process, depends on several factors such as company policies on encryption strength and government restrictions on encryption export. Considering these

facts and the fact that cryptographic algorithms are relative frequently upgraded, cryptographic flexibility and high speed performance are requirements in network systems.

The choice for a certain platform (e.g software, ASICs or FPGAs) for implementing cryptographic applications is driven by several design aspects such as performance, costs, power and flexibility. The performance, costs and power aspects are expressed by well known metrics. However, these metrics do not completely characterize the designs that are implemented in reconfigurable hardware or software. For these designs, flexibility in redesign or hardware reconfiguration, is also an important design issue. Flexibility is defined by IEEE as: *"the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed"* [15].

Since DES, and therefore also Triple-DES, was primarily designed for hardware based platforms, both cryptographic algorithms were often implemented in Application Specific Integrated Circuits (ASIC). These systems showed adequate speed performance. Also the AES implemented in ASIC results in high speed performance [6]. Futhermore, since ASICS are often produced in large quantities, they have favorable costs. However, since ASICs are completely hard-wired they lack flexibility. Their redesign is a complex and expensive process. Every change of the IC design leads to new IC process masks, that is one of the major cost factor. Moreover, a prediction for future semiconductor technologies is that the cost of the mask will grow exponentially and will soon dominate the total cost of the production process. In short, although ASICs based solutions show adequate speed performance they are not suitable for demanding cryptographic network systems.

In contrast to ASIC technology, a Field-Programmable Gate Array (FPGA) is fully reprogrammable. FPGAs provide reconfigurable hardware, flexible interconnect, and field-programmable ability without introducing extra costs. Therefore, substantial amount of work has been reported on various cryptographic algorithms implemented on FPGAs. Although these solutions show adequate speed performance and provide flexibility, they are not interesting for mass productions, since the cost of FPGA devices are still a bottleneck. FPGA devices are very expensive compared to ASIC and software based solutions. This applies especially for cryptographic algorithms that require large FPGA devices. Furthermore, FPGAs based solutions are usually idle for the time a new hardware configuration is being loaded. This fact may be prohibitive for some network systems, especially in the finance and government environments, e.g. transaction verification systems.

Software based solutions, e.g. targeting applications on general-purpose microprocessors, digital signal processors or micro-controllers, are fully reprogrammable. Beside the flexibility aspect, the cost aspect of such solutions are most favorable. However, the disadvantage of these solutions is that the speed performance is significantly lower than that based on ASICs and FPGAs. Therefore, even software based solutions are not suitable for some demanding cryptographic network systems.

Solutions consisting of reconfigurable hardware and one or more general purpose processors offer flexibility in combination with high performance and reduced costs. Furthermore, these solutions usually provide enhanced performance combined with savings in both silicon and software code size. In addition, run-time reconfiguration of system

functionality, such as changing to another cryptographic algorithm without interruption of the computation, can be easy realized in these systems. Therefore, such systems offer a good perspective for demanding cryptographic network systems. A disadvantage of these system is that, they have complex internal structures and therefore the presence of implementation defects is hard to avoid. However, simulation methodologies (which are usually special software tools that mimic the hardware behavior) provide an cost effective approach in the analysis of functional and performance behavior of cryptographic algorithms, such as the AES, during the development of new security processors. The advantage of these methodologies is that hardware/software interactions can be studied in a deterministic manner long before the physical platform becomes available.

1.1 Methodology, Objective and Contribution

The $\rho\mu$-coded processor, also known as the MOLEN processor[16], is designed at the Delft University of Technology as part of the MOLEN project. This project focuses on the embedded processor design that utilizes a general purpose processor and a reconfigurable hardware, e.g. FPGA. In this thesis the execution of the 128-bit AES algorithm will be accelerated by implementing it on the MOLEN platform. The purpose of the chosen platform is to gain increased flexibility and to achieve higher performance. The flexibility of this solution relies in the fact that reconfiguration from the 128-bit AES algorithm to a larger key length or even to a complete different cipher is done run-time. The performance gain will be established by executing the non time-critical operations of the AES algorithm on the general-purpose processor while the time-critical operations will be executed by special hardware engines. The most important criteria of selecting the AES algorithm above other cryptographic algorithms, is the representativeness of this cipher. Since DES is replaced by AES it is expected that it will be frequently used in applications for the next decades.

Since the $\rho\mu$-coded processor is complex, the presence of implementation defects is hard to avoid. However, simulation methodologies provide an cost effective approach in the analysis of functional and performance behavior of the AES algorithm. The SimpleScalar tool-set and various cad tools, provide several powerful simulators to verify the functionality as well as to track performance bottlenecks. In this thesis, the SimpleScalar tool-set will be modified and extended for the MOLEN architecture in order to analyze the performance of the AES algorithm that is mapped on the MOLEN processor. Besides the SimpleScalar tool-set, the Modelsim cad tool and Leonardo spectrum cad tool will be frequently used in order to get an efficient implementation of the AES algorithm that is targeting the MOLEN processor. In this thesis, the AES algorithm that will be modified for the MOLEN platform will be referred as the MOLEN AES.

The main goal of this thesis is to analyze the speed performance of the AES cipher that is targeted on the MOLEN processor. The following questions will be used for this analysis. How much is the gain in speed performance, with respect to pure software based implementations? What is the minimum number of logical cells of the reconfigurable hardware unit that is required by this solution? What is the reconfiguration time of the platform? For what applications can this solution be of interest?

The main contributions of this thesis can be summarized as follows:

- In order to investigate time-critical operations of the AES algorithm, the AES algorithm will be first implemented in software. Afterwards, the implementation will be analyzed on time-critical operations and they will be targeted to reconfigurable hardware. Hereby, a carefully investigation will be done using two methods. The first method is based on analyzing AES operations on arithmetical level, while the second method is based on analyzing the operations on the amount of executed instructions.

- The implemented AES cipher, will be modified for the MOLEN architecture. More precisely, the time-critical AES operations will be replaced by scheduling the SET and EXECUTE MOLEN instructions.

- For the time-critical AES operations, several new AES hardware units will be designed. These units will be described in Very High Speed Integrated Circuit Hardware Description Language (VHDL), and afterwards tested on correct functionality with the Modelsim (v5.5) tool. Finally, the designs will be synthesized with Leonardo spectrum (v2001-1a32) for several FPGA devices.

- In perspective to simulate the MOLEN architecture behavior, the Instruction Set Architecture (ISA) of the SimpleScalar tool-set v3.0c [9] will be extended with the SET and EXECUTE MOLEN instructions. Futhermore, the cycle accurate simulator of the SimpleScalar tool-set will be extended in order to decode and issue the MOLEN instructions. The simulator extension will be based upon a modular structure. In this way, the MOLEN architecture can be easily integrated with an updated version of the SimpleScalar tool-set.

- In addition, an application interface will be built for this extension in order to simulate various applications on the MOLEN platform.

1.2 Framework

The organization of this thesis is as follows. Chapter 2 provides the thesis background information. In this chapter the foundations and principles of cryptography and the MOLEN architecture are overviewed. Furthermore, an overview of related work on the AES cipher is given. Among other, the AES speed performance on different platforms is presented. Chapter 3 overviews the design trajectory of an efficient AES software program that is targeted for the MOLEN architecture. First, in this chapter the algorithm of the AES cipher is analyzed. Second, the design trajectory from the mathematical description of several AES operation to their hardware realization, is described. And finally the AES modification for the MOLEN processor is presented. Chapter 5, describes the simulation of the MOLEN AES. First, in this chapter the design of the simulation framework is described. Second, the conditions and environment settings of the simulations are discussed. And finally MOLEN AES performance results are presented. The final chapter, chapter 6, concludes this thesis and gives some suggestions for future research directions.

2 Background Information

This chapter provides the background information of this thesis. The foundations and principles of cryptography and the MOLEN architecture are overviewed. Furthermore, it also overviews related work on the AES cipher. This chapter is divided in three sections and the organization is as follows. The first section, section 2.1 , describes the basic principles of cryptography. In the second section, section 2.2, related work on the AES cipher is presented. While in the third section, section 2.3, the basic principles and foundations of the MOLEN architecture are described.

2.1 Overview of Cryptography

This section gives an overview of cryptography. More precisely in this section ciphering definitions, methods and techniques are overviewed. This section is included as background information, since this thesis is about cryptography.

2.1.1 Basic terminology and concepts

The term cryptography is derived from the Greek word Kryptos. Kryptos is used to describe anything that is hidden, obscured, veiled, secret or mysterious. Cryptography, over the ages, has been an art practiced by many who have devised ad-hoc techniques to meet various information security requirements. However over the past twenty years, cryptography has moved from an art perspective to a science perspective. The definition of cryptography given by A. Menezes, P. van Oorschot, and S. Vanstone [1] is as follows:

Cryptography is the study of mathematical techniques related to aspects of information security such as privacy, data integrity, entity authentication, and data origin authentication.

The fundamental goal of cryptography is to prevent and detect, cheating and other malicious activities. This goal is achieved by adequately addressing four frameworks in both theory and practice. These four frameworks, that are commonly applied in network services, are described as follows:

- **Confidentiality** is a service used to keep the content of information from all but those authorized to see/understand it. Secrecy is a term synonymous with confidentiality and privacy. There are numerous approaches to provide confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.

- **Data integrity** is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such actions as insertion, deletion, substitution and multiplication.

- **Authentication** is a service related to identification. This service applies to both the sender and the receiver entity. To clarify, two parties initiated into a secure communication should first identify each other.

- **Non-repudiation** is a service which prevents an entity from denying previous commitments or actions. This service is desired in situations where, for example, one entity can authorize the purchase of property to another entity and later denies such authorization was granted. In practice the involvement of a trusted third party is necessary to resolve such disputes.

The mathematical techniques, as the definition states, are known as cryptographic algorithms. These algorithms are the fundamental building blocks for the four frameworks that are described above. Each cryptographic algorithm is classified according to its characteristic features. The next section will overview these classifications and describe the ciphering principles of each classification.

2.1.2 Ciphering Techniques

There are three groups in which cryptographic algorithms can be classified, these are: symmetric cryptographic algorithms, public-key algorithms and hash functions. Algorithms of the first two classifications are key based techniques in which plain-text is transformed into cipher-text or vice versa. Plain-text is a state of data in which information is easily accessible. While, cipher-text is a state of data in which information is hard to reveal. The process of transforming plain-text into cipher-text is called encryption, while the process of transforming cipher-text into plain-text is called decryption. Note that the cipher-text can be transformed back into the plain-text only by using a valid key. The algorithms of the last classification, hash functions, are based on mapping data of arbitrary length to a certain value.

The characteristic feature of symmetric cryptographic algorithms, or also known as ciphers, is that both the encryption and decryption processes are accomplished by using the same key. Note that the AES is a symmetric cryptographic algorithm.Symmetric based algorithms are also classified into two enciphering techniques, namely: stream ciphers and block ciphers. The characteristic of stream ciphers is that the algorithm operates on smaller units of plain-text, usually bits. While block ciphers take a number of bits (known as blocks) and encrypt them as a single unit. In practice, key lengths of at least 128 bits are used in modern block ciphers. Furthermore, There are several operating procedures for block ciphers. These operating procedures are known as modes. A few examples of these modes are: Cipher Block Chaining mode (CBC), Electronic Code Book mode (ECB), Output Feedback (OFB), Cipher Feedback (CFB), Counter mode (CTR). Of all the modes, ECB and CBC are most applied in practice. A detailed description of each mode can be found in [1].

The characteristic feature of public-key algorithms, is that the encryption and decryption processes are accomplished by using different keys. More precisely, the encryption process is based on using a key that is easily available, while the decryption process is based on another key, which is only accessible to a specific entity. The key that is used for the encryption process is known as the public key, while the key that is used for the decryption process is known as the secret key. Both keys are each other complements and they are based upon large prime numbers. The strength of public-key algorithms is based on the fact that factorizing the product of both keys, is a hard mathematical problem. In practice the key sizes variate from 512 bits to 2k bits. RSA is a well known public-key algorithm. The disadvantage of public-key algorithms is that considerable amount of computation capacity is needed for encrypting or decrypting large amounts of data. Therefore, in practice hybrid encryption methods are used, which is based on the combination of symmetric and asymmetric ciphers. The concept of this method is as follows. First the plain-text is encrypted by a symmetric cryptographic algorithm, afterward the symmetric key is encrypted by using a public-key algorithm. For the decryption process, the above procedure is reversed. Public-key algorithms are commonly used in network services involving non-repudiation.

Hash functions are mathematical techniques, that map data of arbitrary length to certain unique value. A hash function operates on a variable input size and returns a fixed-size string, which is called the hash value. One property of hash functions, is that the hash values are relatively easy to compute for any given input. In other words, they do not require an substantial amount of computing capacity. Furthermore, the technique is one-way, which indicates that retrieving the original data from the hash value is not possible. Hash functions are commonly used in network services involving data integrity. The basic application of these algorithms is as follows. First the data is hashed. Afterwards, the hash value along with the data is sent to an entity. That entity also hashes the data and verifies if the retrieved value is equal to the hash value that had been sent.

2.1.3 Cryptanalysis

Cryptanalysis is the study of retrieving the plain-text without knowledge of the valid key. A cipher is said to be breakable if a third party, without prior knowledge of the key, can systematically recover plain-text from the corresponding cipher-text. This all, within a time frame shorter than using the exhaustive search method. With the exhaustive search method, also known as brute force attack, all possible keys are tried in order to reveal the plain-text.

There are three types of cryptanalysis techniques, these are: linear, differential and side-channel techniques. Linear cryptanalysis takes advantage of eventual input-output correlations over a few rounds of the cipher. This technique uses a linear approximation to describe the behavior of the block cipher. Given sufficient pairs of known plain-text and corresponding cipher-text, bits of information about the key can be obtained and increased amounts of data will usually give a higher probability of success.

Differential cryptanalysis, developed by Eli Biham and Adi Shamir, is a type of cryptanalytic technique that appears to be most effective on block ciphers. This technique is

based on the evolution of the differences made in two related plain-texts encrypted with the same key. More information about this technique can be found in [1].

The side-channel cryptanalysis techniques, are based on timing, fault and power analysis of systems. For example, the power consumption of the electrical components is logged to deduce secret information like the encrypting key. In practice, sometimes devices are tampered in order to have it perform some erroneous operations. All these techniques are used within the framework of revealing the secret key or the secret information.

2.2 Related work on the AES cipher

Various aspects of the AES algorithm have been investigated. One is the performance of the algorithm. Several organizations have implemented the AES algorithm on several platforms. Most of the results are published and are available on Internet. Depending on the platform, the AES speed performance varies from several Mbit to a few Gbit per second. Another AES research aspect, is the methodology of breaking the cipher. Nowadays, new cryptanalysis techniques and algorithms are being developed in order to break the AES cipher. This chapter will summarize the results of several publications. The chapter is organized as follows. Section 2.2.1 will evaluate the performances of several AES implementations. While section 2.2.2 will discuss some results of several attacks on the AES cipher.

2.2.1 AES performances on different platforms

Since the performance varies from several Mbit/s to a few Gbit/s, a classification is made. All the published AES implementations can be classified into two groups: software based implementations and hardware based implementations. The software based implementations are designed and coded in programming languages, such as C, C++, Java, and assembly. These implementations are executed, e.g. on general-purpose microprocessors, Digital Signal Processors (DSP), and micro-controllers (such as smart cards). The hardware based implementations are designed and coded in hardware description languages, such as VHDL and Verilog HDL, and finally synthesized into Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGA).

The efficiency of cryptographical implementations in both software and hardware, is generally characterized by several parameters. One of these parameters is speed performance and it is expressed by the metric throughput. Throughput is defined as the number of bits that are processed during a time period. For the encryption process, the throughput is defined as the number of bits encrypted during a time period. Similarly, for the decryption process, the throughput is defined as the number of bits decrypted during a time period. The throughput unit is bit/s (bit per second). Note that in most publications, the initialization process is not included in throughput calculations. Since throughput is depended on the platform environment and therefore it does not always characterize the efficiency of an implementation, it is often accompanied by the parameter latency. Latency is defined as the time that is required to complete the processing of one data block and is usually expressed in number of clock cycles.

Another parameter that characterizes the implementation efficiency is size. In software based designs this parameter is related to the size of the binary code that is compiled for a certain machine. In systems with memory shortage, such as wireless systems and smart cards, this parameter becomes often the most important design criterion. For example, developers who secures systems like wireless phones and personal digital assistants (PDAs), are making often calculated trade-offs between code size and throughput. For hardware based designs, the size parameter is related to the area size of the synthesized circuit. Dependent on what technology is used, there are various ways for expressing the size of a synthesized circuit. In the ASIC based technology, the occupied area, or in short area, is expressed in the terms of equivalent transistors or logic gates. In the FPGA based solutions, area is expressed in the terms of basic building blocks. Dependent on the FPGA vendor, a basic building block is either expressed in Configurable Logic Block (CLB) or in Logic Cells (LC). Various FPGA vendors also give an equivalent logic gate number of the FPGA device.

Another parameter what is used to characterize the efficiency, is the power consumption. This parameter represents the energy that the design consumes and it is usually expressed in Watts or Milli Watts. For low power systems, this parameter is the most critical factor. Note that, since the system environment sets the critical parameter for a design, it is often impossible to compare various designs in a efficient way.

2.2.1.1 Software based implementations

Most software based AES implementations are written in the C,C++, assembler or Java programming languages. Since these implementations are based on different APIs, processors, compilers and various design assumptions, they are hard to compare. NIST however, has tried to compare the AES performance in an efficient way on various platforms [14]. Some of these results are depicted in table 2.1. Note that the results collected in this table, are related only to the encryption process in EBC mode.

One can see from table 2.1 that the designs compiled with the GCC compiler have relative large latencies. This is caused by the fact that GCC does not use the efficiency of the MMX code, while commercial compilers do. Since the testing was only focused on speed performance, other parameters such as code size, are not considered in this work. Paper [2] claims that the fastest software based AES implementation requires 237 cycles to encrypt one data block. This result was obtained on a Pentium II 450 MHZ platform. The implementation was optimized on the basis of several large precalculated tables. Furthermore, it was assumed that all data variables are directly available.

2.2.1.2 Hardware based implementations

The performances of several AES hardware implementations were published recently. In table 2.2 some of these results are presented. One can see, a substantial difference compared to the performance of software based implementations. In most of these implementations, deep pipelines, loop unrolling techniques and other hardware optimizations methods are used to achieve high throughput. For the FPGA based implementations, extra delays are introduced by the routing process. As a result of this speed penalty, the AES implemented in FPGA is typically slower than the same circuit implemented in an

Table 2.1: Performance of software based AES implementations

Platform	**Compiler**	**throughput / latency**
Pentium Pro 200MHz 64MB RAM, windows 95	borland C++ 5.01	704 cycles
Pentium Pro 200MHz 64MB RAM, windows 95	borland C++ 5.01	704 cycles
200MHz Pentium Pro, 64MB RAM, windows 95	visual c 6.0	1149 cycles
600MHz Pentium III, 128MB RAM, windows 98	borland C++ 5.01	757 cycles
600MHz Pentium III, 128MB RAM, windows 98	visual c 6.0	880 cycles
200MHz Pentium Pro, 64MB RAM, Linux	GCC 2.8.1	42.6 Mbit/s
Sun 300MHz UltraSPARC-II w/ 2MB Cache, 128 RAM	GCC 2.95	45.6 Mbit/s
Sun 300MHz UltraSPARC-II w/ 2MB Cache, 128 RAM	Sun Workshop 4.2	20.5 Mbit/s
Pentium II 200 MHz Linux	gcc 2.8.1	862 cycles 4.60 Mbit/s
Alpha EV56 500 MHz Linux	gcc 2.8.1	1313 cycles 5.81 Mbit/s
Pentium 4 3.2 GHz	assembly	257 cycles 1537.9 Mbit/s

ASIC, assuming that both integrated circuits are fabricated using the same semiconductor technology (in particular, using the same transistor size).

Table 2.2: The performance of various AES hardware implementations running in CBC mode

Reference	**Device**	**clock rate** (MHz)	**Area**	**Encryption throughput** (Mbit/s)
Mitsubishi [17]	ASIC 0.23 μ m	-	173K Gates	1950.03
NIST [14]	ASIC 0.25 μ m	-	39K Gates	1280
Elbirt, Yip, Chetwynd, Paar [4]	Xilinx Virtex XCV1000BG560-4	14.1	5302 CLB	300.1

Based on the results presented in table 2.2 and in table 2.1, it should be expected that the relative ranking of the AES algorithms in terms of the hardware efficiency should

be the same or similar for ASICs and FPGAs, and can be substantially different for general-purpose microprocessors and other software environments.

2.2.2 Attacks on the AES

The AES has been designed to resist the classical attacks, used in linear and differential cryptanalysis. However since the AES is based on algebraic operations, new algebraic attacks are developed. In paper [5], Nicolas Courtois and Josef Pieprzyk proved that the AES can be written as an over defined system of multivariate quadratic equations (MQ). Therefore, the problem of recovering the secret key from a single block of plain-text can be written as a system of 8000 quadratic equations with 1600 binary unknowns. In a paper published at Eurocrypt 2000 Shamir et al. described an new algorithm called XL (and FXL)[12] that should solve the problem in sub-exponential time. In practice the algorithm XL fails to break the AES algorithm as shown by Nicolas T. Courtois and Josef Pieprzyk [5].

2.3 The MOLEN architecture

The MOLEN [16] processor is based on a cooperation between a core processor unit(i.e. general purpose processor) and a reconfigurable hardware unit. Extra instructions, also known as MOLEN instructions, are used to control the reconfiguration and execution processes of the reconfigurable hardware unit. In contrast to other reconfigurable architectures, the MOLEN approach can support an infinite number of Custom Computing Unit engines as long as they fit on the reconfigurable hardware unit. In this section the MOLEN machine organization and the MOLEN instruction set will be briefly introduced. A more detailed description of the MOLEN architecture can be found in [19].

2.3.1 The machine organization

The machine organization is shown in figure 2.1. There are several hardware units involved, such as the instruction cache, the arbiter, Core Processor(CP), and the Reconfigurable Processor (RP). The instruction cache is an additional instruction buffer, in which instructions are fetched from the main memory and stored. The arbiter unit fetches instructions from this instruction cache and partially decodes them in order to determine whether they should be issued to the CP or RP. Futhermore, this unit regulates the memory access of both the CP and RP, and distributes control signals. A more precise description of the arbiter control tasks can be found in [18].

The instructions issued to CP are decoded and executed in a general-purpose way. Therefore generally speaking, the tasks and functionalities of the CP are fulfilled by a General Purpose Processor (GPP). Note that the data that is required by this unit are fetched from the General-Purpose Registers (GPRs), the data cache or even the main memory.

There are eight instructions that are issued to the RP, these are: C-SET, P-SET, EXECUTE, set prefetch, execute prefetch, break, movtx and movfx. These instructions will be described in more details in the next section. As illustrated in figure 2.1, the RP

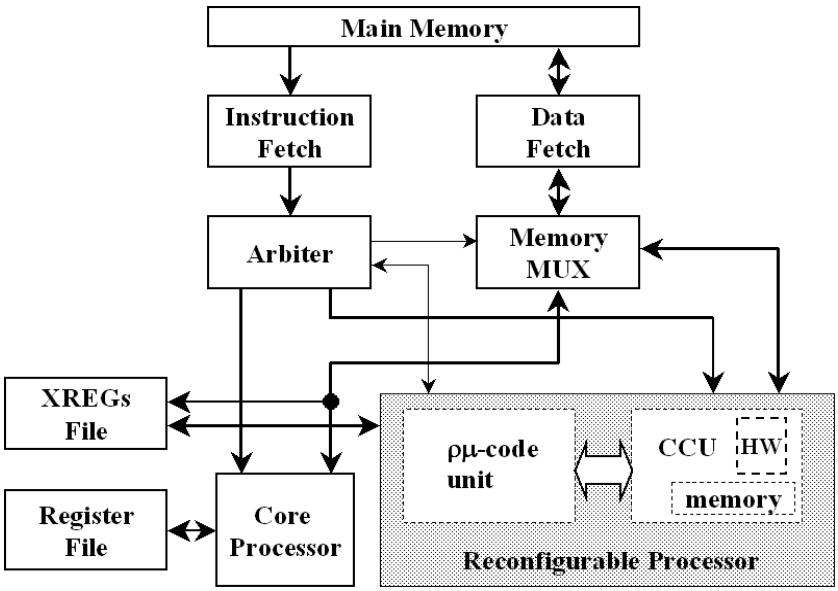


Figure 2.1: The MOLEN machine organization

consists of a Custom Computing Unit (CCU) and a $\rho\mu$-code unit. The CCU consists of a reconfigurable hardware, e.g. FPGA, and memory. Operations that are related to the CCU are divided into two distinct phases: set and execute. In the set phase the configuration process of the CCU is performed by executing reconfiguration microcode, which can be considered as firmware or special hardware description files. In other words, the set phase will enable the CCU to perform a certain operation. In the execute phase, the actual execution process of the configured CCU is performed. The microcode that controls the reconfiguration process will be referred as reconfiguration microcode, while the microcode that controls the CCU execution will be referred as the execution microcode. In section 2.3.3 the processes that are related to these microcodes will be explained in more details.

The $\rho\mu$-code unit provides storage for microcode and regulates the microcode sequence for the execution process. This unit consists of the following parts: ρ-Control Store, $\rho\mu$-code loading unit and the sequencer. The ρ-Control Store provides storage facilities for both fixed microcode as well as pageable microcode. The sequencer regulates the microcode execution sequence. In other words it determines which microcode block will be executed. Futhermore, the sequencer also checks if a certain microcode is stored in the pageable part of the ρ-Control Store. In case that a certain microcode is not present, the $\rho\mu$-code loading unit will load the microcode from the memory. A more detailed description of the $\rho\mu$-code unit can be found in [19].

For the exchange of data between the GPP and the CCU, an input/output interface is provided. This interface is established by some additional registers that are known as Exchange Registers (XREGS). With these registers, arguments are passed to and from the configured CCU and the GPP. Note that in case that the amount of arguments do not exceed the number of XREGS, these arguments are passed by value. Otherwise the

arguments are passed by memory references.

2.3.2 The MOLEN Instruction Set Architecture

The complete MOLEN instruction set consists of eight instructions:

- **two SET instructions**. The two SET instructions, partially set (P-SET) and complete set (C-SET), will actually configure the CCU. Both instructions require a single parameter, which represents the beginning of the memory address that contains the configuration microcode. The P-SET instruction performs partial hardware configuration of the CCU. Since this instruction will cover common and often used functions of an application or set of applications, it will diminish the expensive reconfiguration time. In contrast to this instruction , the C-SET instruction performs a complete configuration of the CCU.
- **EXECUTE instruction**. Once the functionality of the CCU is established, the initiation and regulation of the execution on the CCU is performed by the EXECUTE < address > instruction. The address parameter represents the beginning of the memory address that contains the *execution microcode* (the microcode that will be executed by a certain CCU engine).
- **BREAK**. This instruction is needed in implementing explicit parallel execution between GPP and CCU if it is found to gain substantial performance with simultaneous execution. This operation is used for synchronization to indicate the parallel execution and setting boundaries. In addition BREAK can be used to express parallelism between two or more concurrent CCU units.
- **SET PREFETCH < address >**. In implementations with a reconfigurable hardware of limited size, this instruction can be used by the compiler to prefetch the SET microcode from main memory to a local (much faster) on chip cache or the control store in order to minimize the reconfiguration time penalty.
- **EXECUTE PREFETCH < address >**. The same reasoning as for the SET PREFETCH holding for the EXECUTE microcode.
- **Two MOVE instructions**. Two Move instructions are introduced for data exchange between the GPP register file and XREGS. The instruction MOVTX Rb,XRa (move to X-REG) is used to move the content of general purpose register Rb to XRa. While the instruction MOVFX Ra,XRb: (move from X-REG) is used to move the content of exchange register XRb to GPP register Ra.

The GPP ISA is extended by these instructions in order to control the CCU refconfiguration and execution process. There are three distinctive sets of MOLEN instructions which can be used for the GGP ISA extension, these are known as: the minimal πISA, the preferred πISA and the complete πISA. The minimal πISA consists of the C-SET, the EXECUTE, the MOVTX and the MOVFX instructions. This is the smallest set of MOLEN instructions that provide a working reconfiguration and execution scenario of the CCU. The preferred πISA consists of the P-SET, the C-SET, the EXECUTE, the

MOVTX, the MOVFX, and both PREFETCH instructions. This set supports besides the complete CCU hardware configuration also the partial CCU hardware configuration. Futhermore, by using the PREFETCH instructions the loading time of microcode can be decreased. Note that with these two sets, that is both the minimal πISA and the preferred πISA, parallel execution on the GGP and FPGA is not possible. However, utilizing the complete πISA, parallel execution will be possible. This set consists of all eight MOLEN instructions. A more detailed description of these instruction sets can be found in [11].

2.3.3 Reconfiguration and execution microcode

The reconfiguration microcode for a certain CCU configuration is derived from the synthesis output of the corresponding design. A synthesis output contains a bitstream of data and configuration commands that configure the CCU. Note that the bitstream pattern is dependent of the targeted FPGA device. In other words, a design that is described in a certain hardware language, will result in a different bitsream sequence when targeted for an different FPGA technology. While the reconfiguration microcode consists of a certain bitstream, the execution microcode consists of a sequence of microinstructions, that controls the execution process on the CCU.

There are several processes related to the microcode, these are: initialization, loading, execution and termination. The initialization of microcode refers to storing microcode. In case that it concerns pageable microcode, the microcode is stored in the main memory. In case that it concerns resident microcode, the microcode is stored in the fixed part of the ρ-Control Store. The loading of microcode is related to directing the appropriate microcode from the main memory to the ρ-Control Store in case it is not present there. The execution process of the microcode will actually perform the required operation. Note that the execution process of reconfiguration microcode is related to the CCU reconfiguration process, while the execution process of execution microcode is related to perform a certain functionality. The microcode termination process is related to the process of ending the execution operation. For this process the *end_op* microinstruction is utilized. Note that, since reconfiguration microcode consists of an arbitrary bit sequence, early microcode termination might occur. Therefore, an extra flag bit is utilized, that indicates the length of the reconfiguration microcode or the end address of the reconfiguration microcode. Note that this flag is placed in the beginning of the microprogram. A more detailed discussion of microcode termination can be found in [11].

2.4 Chapter summary

This chapter provided the background information for this thesis. It was divided in three sections. In the first section the principles of cryptography were described. As described in this section, cryptographic algorithms are the fundamental building blocks for utilizing four network services: confidentiality, data integrity, authentication and NON-repudiation. Futhermore, it was described that cryptographic algorithms can be classified in three groups: symmetric algorithms, public-key algorithms and hash functions. As explained, symmetric algorithms are based on one key, public-key algorithms

are based on several keys and hash functions maps data of an arbitrary length to a certain value. It was highlighted that the AES is based on a symmetric algorithm.

In the second section of this chapter, several speed performances of AES implementations were presented. It was shown that the performances varies from several Mbit/s to a few Gbit/s. As discussed, the software AES implementations compiled with the GCC compiler have low speed performance, since GCC does not use the efficiency of the MMX code. For the hardware implementations, it was discussed that deep pipelines, loop unrolling techniques and other hardware optimizations methods are used to achieve high throughput. Futhermore, in this section it was discussed that the AES can be written in an over defined system of equations and therefore it can be possible to recover the secret key. However, in practice this system fails to break the AES [5].

In the third section the MOLEN architecture was described. As described in this section, the architecture is based on a cooperation between one or more GPP and a RP. There are eight instructions that are issued to the RP, these are: C-SET, P-SET, EXECUTE, set prefetch, execute prefetch, break, movtx and movfx. As explained, the RP can be configured to a certain hardware engine by utilizing these instructions and the microcode that is derived from the synthesis output of the corresponding hardware design.

The design of the MOLEN AES

3

This chapter describes the design methodology of an efficient AES software program that is targeted for the MOLEN architecture. The organization of this chapter is as follows. The design methodology is overviewed in the introduction of this chapter. In section 3.2 the AES algorithm scheme is overviewed. In section 3.3 the AES algorithm is analyzed and profiled. In section 3.4 the design steps of the AES CCU engines are overviewed. And finally in section 3.5, the MOLEN AES is presented.

3.1 Introduction

The main characteristic of the MOLEN architecture is the presence of the reconfigurable processor which can be configured to a specific functionality. As stated before, the performance of the MOLEN AES will be established by executing the non time-critical operations of the AES algorithm by the GPP while the time-critical operations will be executed by special hardware engines. The methodology for designing the MOLEN AES is depicted in figure 3.1. First of all, the AES will be implemented in software. Subsequently, it will be analyzed on time-critical operations (indicated as f() in figure 3.1) and non time-critical operations. Note that, in order to find the distinction between these operations, a careful investigation of the AES algorithm is required. Afterwards, the time-critical functions will be described in a certain hardware description language and the description will be synthesized. Note that the CCU reconfiguration microcode will be derived from the synthesis FPGA configuration file. And finally, the AES cipher will be modified with the SET and EXECUTE MOLEN instructions in order to utilize the implemented AES hardware engines.

3.2 The AES algorithm scheme

As stated before, the National Institution of Standards and Technology (NIST) chose the Rijndael algorithm as the Advanced Encryption Standard (AES) in October 2001. The AES is a symmetric block cipher and supports key lengths of 128, 192 or 256 bits and block sizes of 128, 192 or 256 bits. The AES performs the encryption and decryption process on an iterative basis. Each iterative step is known as a round operation or just round. All rounds are identical. The number of rounds that are used, is determined by the key size. For example a 128-bit key requires nine rounds, while a 256-bit key requires thirteen rounds. The ANSI C source code v2.2 of the AES algorithm [6] is used as a reference for the development of the MOLEN AES. This section will only focus on the 128 bit version of the AES, however, the same techniques are applicable for the bigger key sizes.

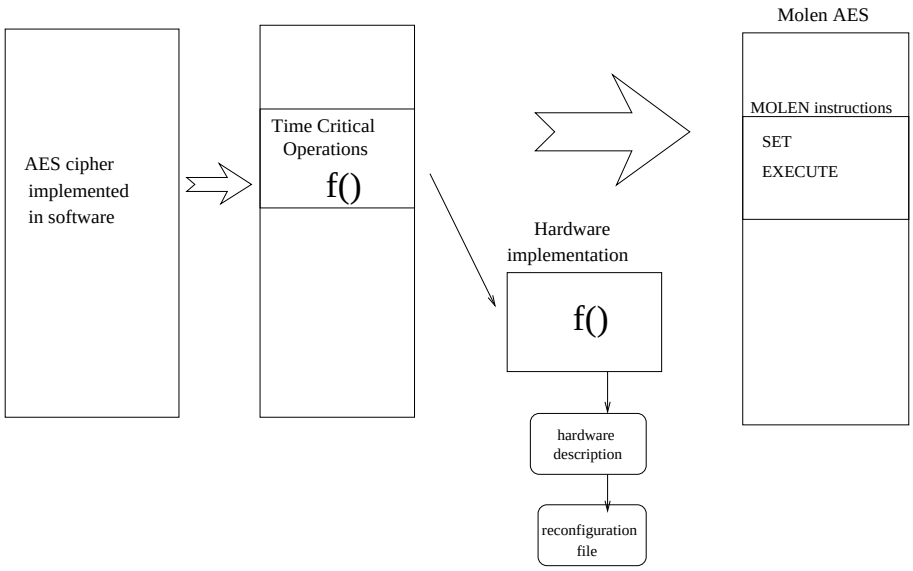


Figure 3.1: The design methodology for the MOLEN AES

The AES algorithm operates on a 2-dimensional four-by-four array that is called the state of the data-block or in short state. The state represents the status of the data-block after a certain transformation. Each element of the state has a length of a byte (8-bits). This section will describe how the state is changed through the cipher.

The AES algorithm diagram flow is depicted in figure 3.2. The encryption algorithm consists of an initial key addition, nine rounds and a final step. Note that the final step is a reduced round operation and therefore in several documents it is stated that the 128-bit AES version has ten rounds. The encryption process is as follows. First, sixteen bytes of the plain-text are copied to the state, which is an empty 2-dimensional buffer in the initial phase. Secondly, the 128-bit key is added to the state. Afterwards, the state enters the first round operation. As stated before, the 128-bit AES version has nine rounds. Each round is composed of the following four transformations:

- **ByteSub** substitutes a byte by using a S-box substitution table.
- **ShiftRow** changes the sequence of the input bytes.
- **Mixcolumn** maps a word (4-bytes) to a single byte.
- **AddRoundKey** performs modulo-2 addition of a key with the input bytes.

In case when all nine rounds are performed, the state enters the final step. This final step is a reduced round operation and consists of the ByteSub, ShiftRow and the AddRoundKey transformations. Afterwards, the state is finally copied to the cipher-text.

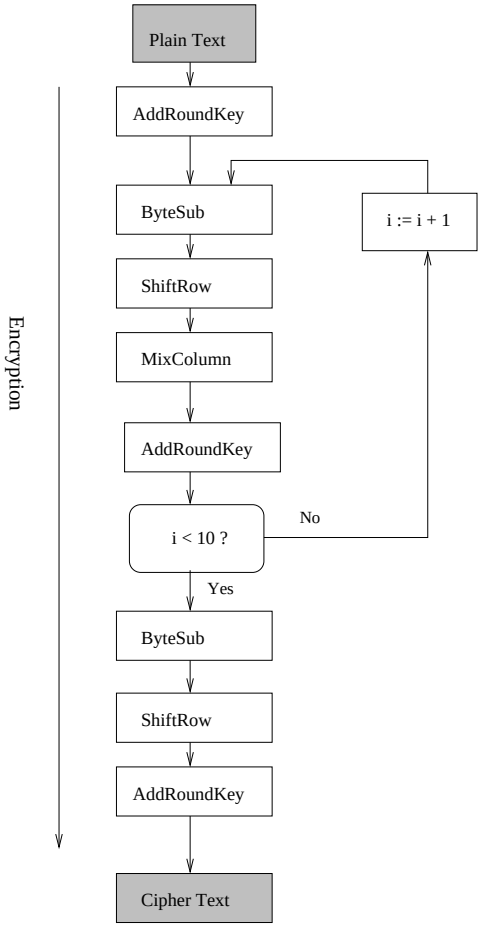


Figure 3.2: The AES encryption diagram flow

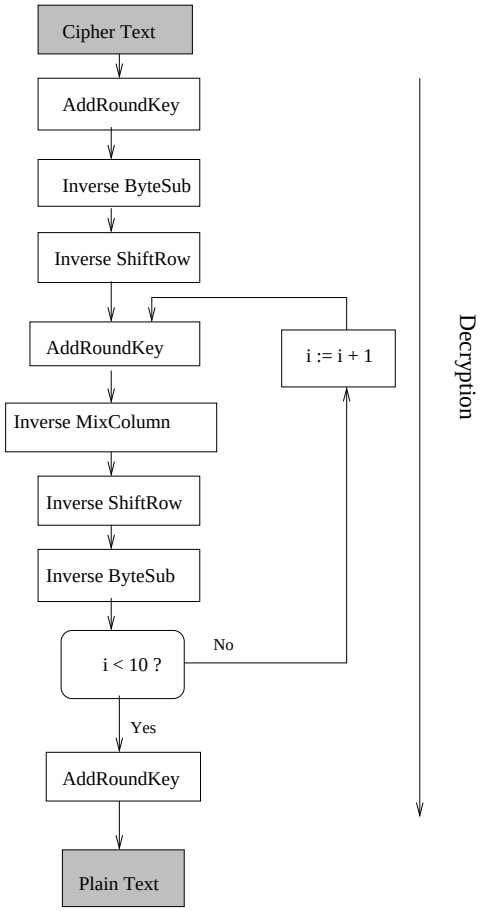


Figure 3.3: The AES decryption diagram flow

Unlike the DES [1] algorithm, the decryption process of the AES cipher consists of different transformations then those used during the encryption process. These transformations are based on the inverse properties of the transformations that are used during the encryption process. The decryption scheme is illustrated in figure 3.3. The figure indicates that the decryption process consists of three initial transformations, nine rounds and a key addition. The rounds are identical and consist of the following four transformations:

- **AddRoundKey** performs modulo-2 addition of a key with the input bytes.
- **Inverse Mixcolumn** maps a word (4-bytes) to a single byte.
- **Inverse ShiftRow** changes the sequence of the input bytes.
- **Inverse ByteSub** substitutes a byte by using a S-box substitution table.

The Inverse Mixlcoumn, the Inverse Shiftrow and the Inverse Bytesub transformations are based on the inverse properties of respectively the Mixcolumn, the Shiftrow

and the Bytesub transformations. In the next sections these transformations will be described and analyzed in more details.

3.2.1 ByteSub and Inverse ByteSub

ByteSub, which stands for byte substitution, is a non-linear transformation that replaces each element of the state. For the replacement strategy a 256-bytes substitution table is used. Note that substitution tables, or better known as S-boxes, are commonly used in block ciphers. The S-box is a sixteen-by-sixteen table. The replacement strategy is as follows. The four most significant bits of each state element are used for the row index, while the rest are used for the column index. Without going into much detail, the creation of this S-box, is based on the composition of two transformations in the Galois field (GF) for 2^8 [10]. The first transformation is that every byte is replaced by its reciprocal. While the second transformation is based on an affine transformation. A more detailed description can be found in [6].

The inverse of the ByteSub transformation is known as Inverse ByteSub. In this transformation the same replacement strategy is used as in the ByteSub transformation. The main difference is that a different substitution table is used, which is based on the inverse property of the ByteSub's S-box.

3.2.2 ShiftRow and Inverse ShiftRow

The ShiftRow transformation is based on changing the sequence of the bytes within the state's row. This transformation shifts the bytes of each row with a certain offset. The first row is shifted with offset zero, in other words, the first row remains intact. The second row, however, is cyclically shifted with offset one. In practice this implies that the state's second byte obtains the fourth position, the sixth byte obtains the second position, the tenth byte obtains the sixth position and the sixteenth byte obtains position ten. Likewise, the bytes of the third row of the state is cyclically shifted with offset two. And finally the state's last row, which is row four, is cyclically shifted with offset three.

During the decryption process, the Inverse ShiftRow transformation reverses the ShiftRow operation. All rows, except the state's first row, are cyclically shifted with a certain offset. The offsets used during this operation are three, two and one for respectively the second, third and the state's fourth row. Table 3.1 gives an overview of these shifts for both the encryption and decryption directions.

Table 3.1: The offsets that are used in the ShiftRow and Inverse ShiftRow transformations

	Encryption	Decryption
Row 1	0	0
Row 2	1	3
Row 3	2	2
Row 4	3	1

3.2.3 MixColumn and Inverse Mixcolumn

The MixColumn transformation is based on multiplication of the state with a certain matrix. This matrix is derived from the property that multiplication of a polynomial with a fixed polynomial over GF(2^8), results in a constant matrix multiplication. The derivations of this matrix is as follows. In GF(2^8), columns of the state can be considered as polynomials. These polynomials are multiplied over GF(2^8) with a fixed polynomial c(x), which is defined as:

$$c(x) = 03x^3 + 01x^2 + 01x + 02$$

As described in [10], multiplication with a fixed polynomial will result in a constant matrix multiplication. Using this property, the polynomial c(x) will result in equation 3.1, which represents a constant matrix multiplication. In this equation, $\bar{a}$ represents a column of the state, while $\bar{b}$ represents a column of the new state. Note that the matrix elements are denoted as hexadecimal values. For example, the first byte of the new state column is generated as follows. The first two bytes of $\bar{a}$, are multiplied by respectively hex-value 2 and 3 over GF(2^8). The output of this operation is added to the third and fourth byte of $\bar{a}$, what will produce the first byte of the column of the new state .

$$\bar{b} = \bar{a} \otimes C \tag{3.1}$$

with C defined as:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

The Inverse Mixcolumn transformation reverses the Mixcolumn operation and therefore it is used during the decryption process. This transformation is based on the same procedures that are used within the Mixcolumn transformation. The main difference is, that the Inverse Mixcolumn transformation uses a different constant matrix. This matrix is derived from the following property:

$$(03x^3 + 01x^2 + 01x + 02) \otimes d(x) = 01 \tag{3.2}$$

Solving equation 3.2 will result in :

$$d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$$

Using the techniques as described in [10], the polynomial d(x) can be represented by the matrix:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$$

3.2.4 AddRoundKey

The AddRoundKey transformation adds a temporary key to the state. This is done by bite-wise xoring the key with the state elements. Equation 3.3 depicts the mathematical description of AddRoundKey. The K-matrix represents the elements of the key, while the B-matrix represents the elements of the state. Note that the depicted elements in equation 3.3 represents four bytes.

Since there are eleven AddRoundKey transformations during both the encryption and decryption process, there are in total eleven different keys. These keys consist of the initial key and ten temporary keys, which are also known as round keys. These round keys are obtained by expanding the initial (16 bytes) key by using a special routine. The next section provides a detailed description of this expansion process, which is known as the key scheduler.

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} \bigoplus \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \end{pmatrix} \tag{3.3}$$

3.2.5 The Key Scheduler

In the key scheduler, ten round keys are generated from the initial key. Each round key is generated by several operations on the previous round key. The routine of generating a new round key is as follows:

- First, the last 4-bytes (word) of the previous round key (or the initial key), are circularity shifted, in the following way:

$$A0 \quad A1 \quad A2 \quad A3 -> \quad A1 \quad A2 \quad A3 \quad A0 \tag{3.4}$$

  Note that, "A0", "A1", "A2" and "A3" represent respectively the thirteenth, fourteenth, fifteenth and sixteenth byte of the key. In the literature, this operation is also known as the ShiftByte transformation.

- Second, the ByteSub transformation is applied on the new byte sequence.

- Third, the first byte of the sequence is xored with a variable called Rcon. Rcon represents the round number. For example for generating the first round key, Rcon has the hex-value 01.

- For the fourth step, the obtained word of the previous step is xored with the first byte of the previous roundkey, what will produce the first byte of the new round key.

- Final, the second, third and fourth byte of the previous round key are xored with the first byte of the new round key. This will produce respectively the second, third and fourth byte of the new round key.

This routine is repeated until all the ten round keys are obtained. In figure 3.4 the procedure of generating a new round key is illustrated. Note that k3 represents the last four bytes of the initial round key.

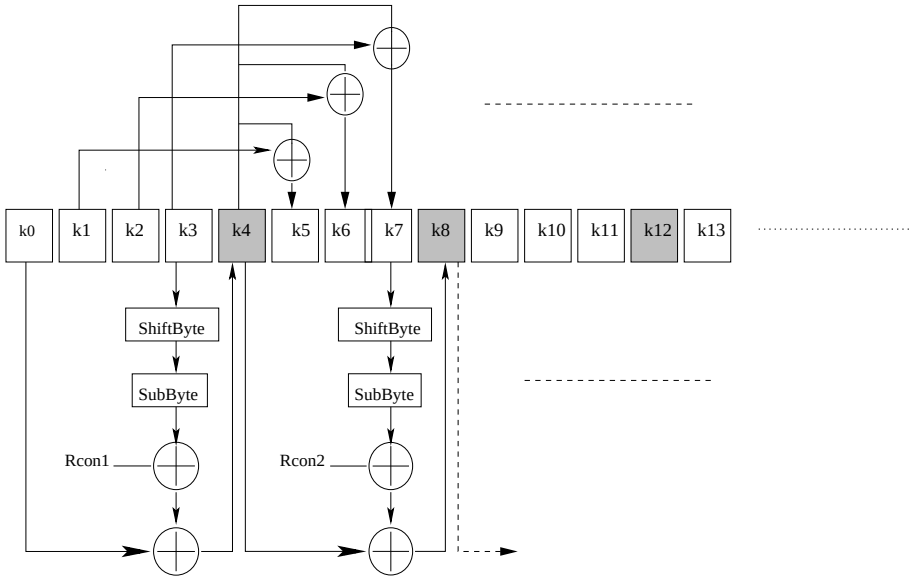


Figure 3.4: The procedure for obtaining a new round key

3.3 Time critical operations

In this section two methods will be used in order to find the distinction between time-critical operations and non time-critical operations. The first method is based on analyzing AES transformations on arithmetical level, while the second method is based on analyzing the transformations on the amount of executed instructions. The basis of the first method is: what are the arithmetical operations and how many clock cycles will these operations require. The second method will be performed by using a simulator that will give a detailed profile-ing information of the executed instructions. For the second method, the sim-profiler simulator from the SimpleScalar tool-set [9] will be used. This simulator displays a detailed description of the executed instructions.

3.3.1 Analysis on arithmetical level

The first transformation of the encryption process is AddRoundKey. As described in the previous section, in this transformation a 16-bytes round key is added to the state. Since there is no dependency between the state's bytes, this transformation can be performed by using sixteen simple bitwise XOR operations. For the whole encryption process, there are eleven AddRoundKey transformations which are associated with eleven round key additions . The first RoundKey addition involves the initial step, which is the addition of the input key with the plain-text. The second till the tenth AddRoundKey transformations involve adding the appropriate round key to the MixColumn output state. The final AddRoundKey transformation involves adding the output state of the ByteSub transformation with the last round key. Thus, considering only the effect of the

AddRoundKey transformation, 176 byte-wise xor operations are required for the encryption process. This will cost a limited amount of clock cycles. Therefore, AddRoundKey is not a time-critical transformation and will be performed on software basis.

As described in the previous section, the ShiftRow and the InvShiftRow transformations manipulate the order of bytes. Reordering of the positions of bytes, does not require any use of the ALU or other functional units. Therefore, these transformations are non time-critical operations.

In the ByteSub and Inverse ByteSub transformations, the sixteen bytes of the state are substituted using the S-box or inverse S-box. These transformations are based on fetching data out of the memory or inside the register file. Since no computational power is required, these transformations will consume limited amount of clock cycles. Furthermore, implementing these transformations in hardware resources would require substantial amount of logic cells, since a 8-bytes 256 Look Up Table (LUT) is required. Therefore realizing these transformations in hardware will not be efficient.

As described in the previous section, the Mixcolumn and Inverse Mixcolumn transformation are based on matrix multiplication. In general, a matrix multiplication consists of two arithmetical operations: multiplication and addition. Since with these transformations four-by-four matrix are involved, both the Mixcolumn and Inverse Mixcolumn transformation consist of 88 multiplications and 48 additions. In practice, an 8-bit addition operation uses at least 1 clock cycle, while a fast 8-bit multiplication algorithm uses at least 31 clock cycles [3]. Using these numbers, one single Mixcolumn or Inverse Mixcolumn transformation will be estimated on 2776 clock cycles. If only the effect of these transformations are considered for the ciphering process, 27760 cock cycles will be required for processing each data block. This makes both the Mixcolumn and Inverse Mixcolumn transformation time-critical operations and therefore they are suitable candidates for hardware implementation.

As described in the previous section, the KeyScheduler is based on substitutions and additions of bytes. As estimated these operations are non time-critical. Furthermore, since the KeyScheduler is executed only once during a cipher process, it requires a fixed cycle time. Therefore, the KeyScheduler will be performed in software.

3.3.2 Analysis on instruction level

The Simple Scalar tool-set [9] comes with a functional simulator that produces various profile information on instruction level of a computer program. This simulator known as the sim-profiler, is based on the 32bits MIPS ISA. The generated profile is actually the statistics of instruction groups and events that occurs during the execution of a program. The sim-profiler, can generate detailed profiles on instruction classes, addresses, text symbols, memory accesses, branches, and data segment symbols.

For our simulation, the reference ANSI C source code v2.2 of the AES algorithm was used. This source code is freely available at the Rijndael website [6], which makes it more suitable for this thesis purpose. In this reference code every transformation as described in chapter 3, is represented by a separate function, what makes the code more transparent. As an addition to this code, an user interface was build. This user interface provides the possibility to encrypt or decrypt data in EBC mode, independently of the

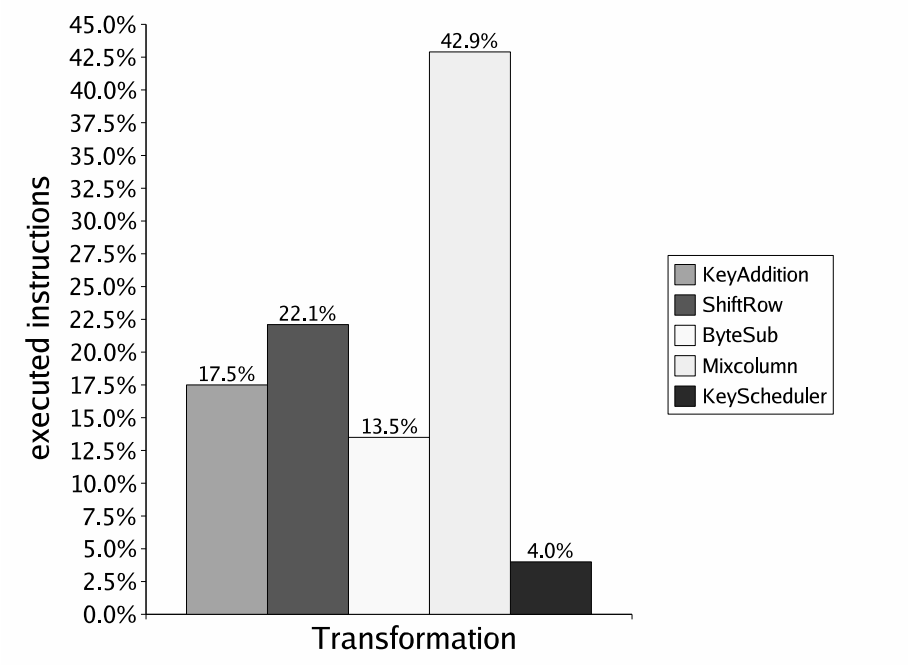


Figure 3.5: The number of executed instruction in terms of percentage for the encryption process

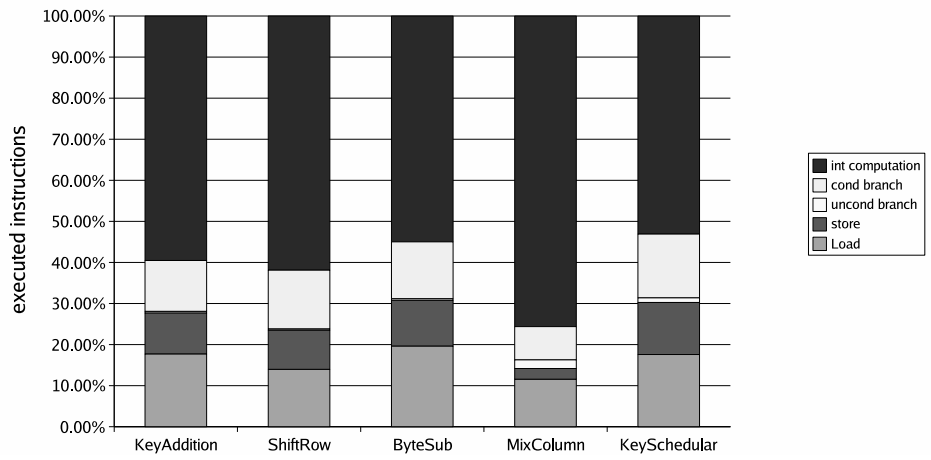


Figure 3.6: Instruction type that are executed during the encryption process

input size. The source code of this interface is depicted in Appendix C.

For the simulation process, a 4kb ASCII text was used as plain-text with a 16-bytes

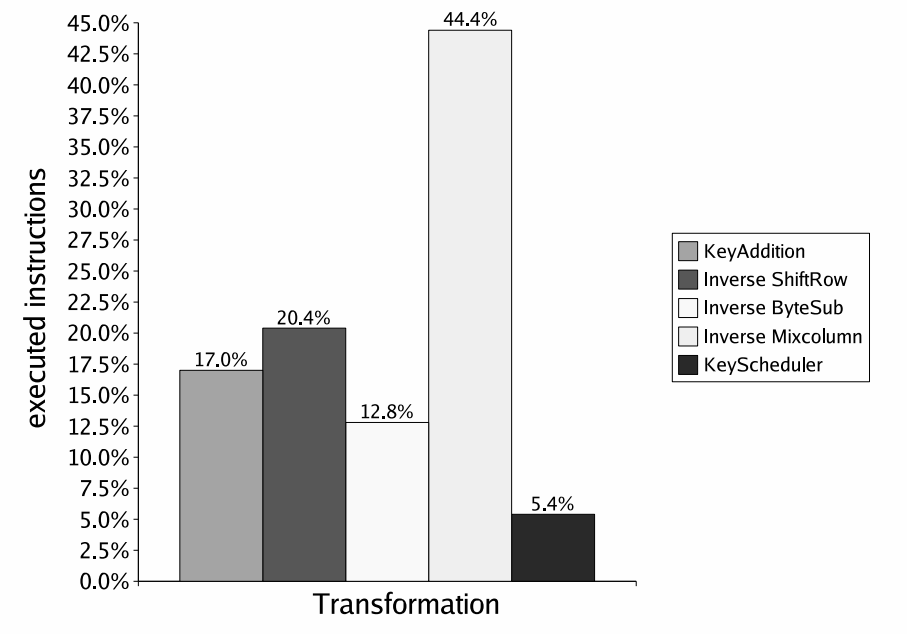


Figure 3.7: The number of executed instruction in terms of percentage for the decryption process

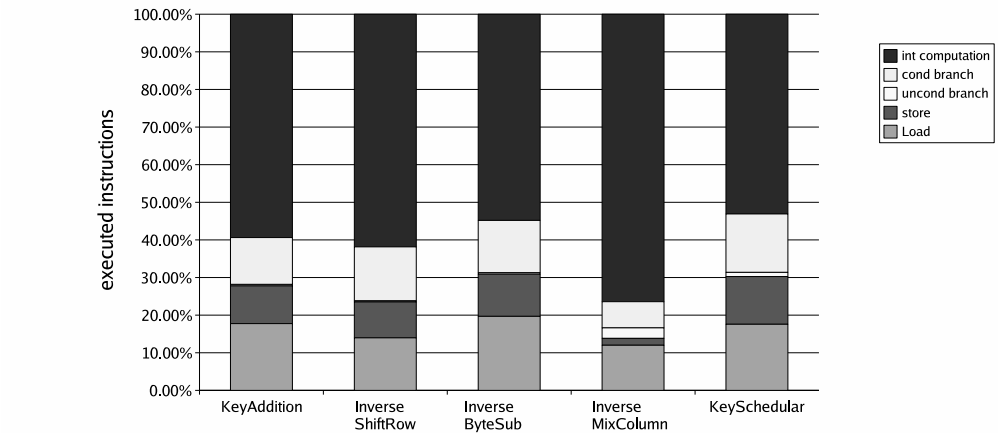


Figure 3.8: Instruction type that are executed during the decryption process

key. The simulation results for the encryption process are depicted in figure 3.5. In this figure 3.5, the number of executed instructions for a certain transformation in terms of percentage is illustrated. One can see that there are at least two times more instructions executed for the Mixcolumn transformation then for any other transformation. With

sim-profiler it can be analyzed which instruction types are executed. In figure 3.6, it is illustrated that Mixcolumn contains the most integer computations. Therefore one can conclude that the Mixcolumn transformation is indeed the most time-critical operation for the encryption process.

The simulation results of the decryption process are illustrated in figure 3.7 and in figure 3.8. As expected, this figure illustrates that the Inverse Mixcolumn transformation is the most time-critical transformation.

3.3.3 Conclusion

In this section, two methods were used in order to find the distinction between time-critical operations and non time-critical operations. The first method, which was based on analyzing AES transformations on arithmetical level, showed that the MixColumn and the Inverse MixColumn transformation are the most time-critical operation. The second method, which was based on the profile simulator, showed that the MixColumn and the Inverse MixColumn transformation contain the most executed instructions, and that most of these instructions are related to integer computations. Therefore one can conclude that the Mixcolumn and Inverse MixColumn transformation are indeed the most time-critical operations.

3.4 The design of the AES Custom Computing Units

By using two different analysis methods, it is shown that the Mixcolumn and the Inverse Mixcolumn transformations are the only time-critical transformations of the AES algorithm. Therefore realizing these transformations in hardware will probably give the highest performance gain of the MOLEN AES. However one assumption is made. As described in the previous section, the ShiftRow is based on reordering bytes and therefore it is a non time-critical transformation. However, implementing this transformation in hardware will require no or at least minimum hardware resources and therefore it can be efficiently implemented together with the Mixcolumn transformation. This assumption also applies for the Inverse ShiftRow and the Inverse Mixcolumn.

The design goal for the hardware implementation, is to get the highest throughput with a small area size of the CCU. This section describes the design of the AES CCU engines. The organization of this section is as follows. In the first section, various design strategies are discussed with the perspective on high speed performance and CCU area occupation. In section 3.4.2 and 3.4.3, hardware is designed from the mathematical description of matrix multiplication over GF(2^8). The hardware is described in VHDL. In section 3.4.4 the circuit is presented. While in section 3.4.5, the design is verified on functional correctness. Finally, in section 3.4.6, the synthesis results are presented and the design of the CCU configuration file is discussed.

3.4.1 Matrix multiplication

As pointed out in section 3.2.3, the Mixcolumn and Inverse Mixcolumn transformations are based on constant matrix multiplication over GF(2^8). Matrix multiplication is a fre-

quently used operation in a wide variety of applications such as, graphic, image, robotics, cryptography, etc. There are several design approaches for implementing constant matrix multiplication in hardware. However most of these approaches are based on an optimized algorithm(e.g booth) of the arithmetic operation: multiplication.

As described in the previous section, matrix multiplication is based on two arithmetical operations that are applied on matrix elements, these are: multiplication and addition. Based on these operations, it was estimated that both the Mixcolumn transformation, and the Inverse transformation, will at least require 2776 clock cycles. However, since the Mixcolumn and Inverse Mixcolumn transformations are performed in the (2^8) Galois field, the amount of operations that is required for achieving a multiplication can be significantly reduced. The next sections will overview both the addition and multiplication operation over GF(2^8) and will present an efficient hardware implementation of the Mixcolumn and InvMixcolumn transformations.

3.4.2 Addition over Galois field

Each matrix element, that is used within the Mixcolumn and Inverse Mixcolumn transformation, is represented by a byte. In GF(2^8), these bytes can be considered as polynomials of degree 8. For example, a byte B existing of b7 b6 b5 b4 b3 b2 b1 b0 bits, is represented by the following polynomial:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

The coefficients of b_0 up to b_7 are elements of the {0,1} Galois field. The addition of two polynomials in this Galois field, is defined by equation 3.5. Note that the polynomials a(x) and b(x), are polynomials over GF(2^8), while coefficients are elements over GF(2). The equation describes that a new polynomial is obtained by adding the corresponding coefficients of two polynomials, modulo 2 (i.e., 1 + 1 = 0). Since all the coefficients are elements of GF(2), this operation is the equal to the exclusive-or $\oplus$ operation. Therefore, in hardware logic perspective, the addition of two polynomials of degree 8 can be established by using eighth 2-bit XOR gates.

$$C(x) = (a(x) + b(x))\ mod2 = \sum_{i=0}^{m}(a_i + b_i)x^i\ mod2 \tag{3.5}$$

3.4.3 Multiplication over Galois field

In contrast to the addition operation, the math of multiplying polynomials over GF(2^8) is more complicated. Equation 3.6 defines the multiplication of polynomials over GF(2^8).

$$C(x) = a(x) \cdot b(x) = \sum_{i=0}^{m+n}\left(\sum_{j=max(0,i-m)}^{min(i,n)} a_jb_{i-j}\right)x^i\ mod(x8 + x4 + x3 + x + 1) \tag{3.6}$$

$$\text{with } a(x)= \textstyle\sum_{i=0}^{n} a_ix^i,\ b(x)= \sum_{i=0}^{n} b_ix^i$$

Multiplications of polynomials over GF(2^8) introduces an extra polynomial, which is called the irreducible polynomial. This polynomial is used as the modulo-argument of the polynomial product. A polynomial is called irreducible if it has no divisors other than 1 and itself. For the AES cipher, the following irreducible polynomial is chosen:

$$x^8 + x^4 + x^3 + \mathrm{x} + 1.$$

As described in section 3.2.3, the following matrix elements are used for Mixcolumn: 01, 02, and 03. Please note that for the multiplication with value 01, no operation will be performed. As described in the previous section, each 8-bit value can be represented by a polynomial of degree 8. Multiplication over GF(2^8) with value 2 will result as follows. Since value 2 is represented by "000 0010" in byte representation, the polynomial representation of value 2 is: $(0x^7 + 0x^6 + 0x^5 + 0x^4 + 0x^3 + 0x^2 + 1x + 0) = \mathrm{x}$. Using equation 3.6, multiplication of a polynomial a(x) with polynomial x, can be presented as:

$$C(x) = c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \; with \tag{3.7}$$

$$\begin{aligned} c_0 &= a_7 \\ c_1 &= a_0 + a_7 \\ c_2 &= a_1 \\ c_3 &= a_2 + a_7 \\ c_4 &= a_3 + a_7 \\ c_5 &= a_4 \\ c_6 &= a_5 \\ c_7 &= a_6 \end{aligned}$$

The above polynomial shows that multiplication of polynomial a(x) with value 2, will result in specific dependencies between the coefficients of a(x). In byte representation, the above dependencies indicate that multiplication with value 2 wilt result in a predefined bit dependency pattern of the input byte. For example, equation 3.7 implies that the third output bit c_3 is generated by adding the second bit a_2 with the seventh a_7 bit. As discussed in the previous section, the addition operation can be efficiently implemented by the XOR operation. Translating the above dependency into a VHDL description will result in the following code:

```
entity mul_02 is
port (
input: in std_logic_vector(7 downto 0);
output: out std_logic_vector(7 downto 0)
);
end mul_02;

architecture mul_02 of mul_02
begin
```

```
output(7) <= input (6);
output(6) <= input (5);
output(5) <= input (4);
output(4) <= input (3) xor input(7);
output(3) <= input (2) xor input(7);
output(2) <= input (1);
output(1) <= input (0) xor input(7);
output(0) <= input (7);
end mul_02;
```

Note that the above description shows that the complexity of the multiplication with value 2 is reduced to three xor operations. Likewise, multiplication of a polynomial a(x) with value 3 will result in the following polynomial:

$$C(x) = c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \; with \tag{3.8}$$

$$\begin{aligned} c_0 &= a_0 + a_7 \\ c_1 &= a_1 + a_0 + a_7 \\ c_2 &= a_2 + a_1 \\ c_3 &= a_3 + a_2 + a_7 \\ c_4 &= a_4 + a_3 + a_7 \\ c_5 &= a_5 + a_4 \\ c_6 &= a_6 + a_5 \\ c_7 &= a_7 + a_6 \end{aligned}$$

The above dependency shows that the complexity of multiplication over GF(2^8) with value 3 is reduced to eleven xor operations. Just like for value 2, translating the above dependency into a VHDL description is a straightforward process. The documented VHDL source files of all the values that are used within both transformations, are depicted in appendix D. The primary difference of the Inverse Mixcolumn transformation compared to the Mixcolumn transformation, is the larger hexadecimal values of the matrix coefficients. Therefore, the Inverse Mixcolumn transformation results in a more complex bit dependency pattern for every input byte.

3.4.4 The circuit

As described in section 3.2.3, the Mixcolumn transformation operates on each 4-bytes of the state. The block scheme of the design is depicted in figure 3.9. In figure 3.9, the blocks "X2" and "X3" represent the multiplication with value 2 respectively value 3 over GF(2^8). Given the fact that the state consists of 16 bytes and that these bytes are uncorrelated, some parallelizations methods can be applied. Therefore the circuit depicted in figure 3.9 can be repeated four times, in such a way that all the state's bytes can be processed at the same time.

The CCU engine for the encryption process is depicted in figure 3.10. This CCU engine will be referred as the Mixcolumn hardware unit. Two control pins are added for data-flow synchronization, these are "enable" and "data_valid". In case that, the

enable pin is high, the input data will be processed. When the Mixcolumn hardware unit completes its operation, the "data_valid" signal becomes high. Since the MOLEN arbiter distributes control signals, the "data_valid" and the "enable" control signals will be directed to the MOLEN arbiter. The CCU engine for the decryption process will be referred as the Inverse Mixcolumn hardware unit. This unit has similar structure to the Mixcolumn hardware unit. The VHDL source files of both the Mixcolumn and Inverse Mixcolumn hardware units are depicted in appendix D.

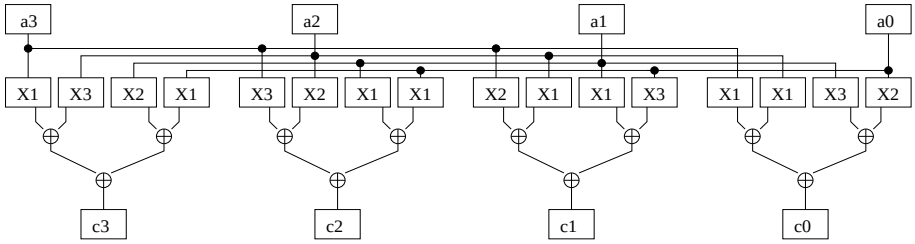


Figure 3.9: The hardware scheme of processing four bytes

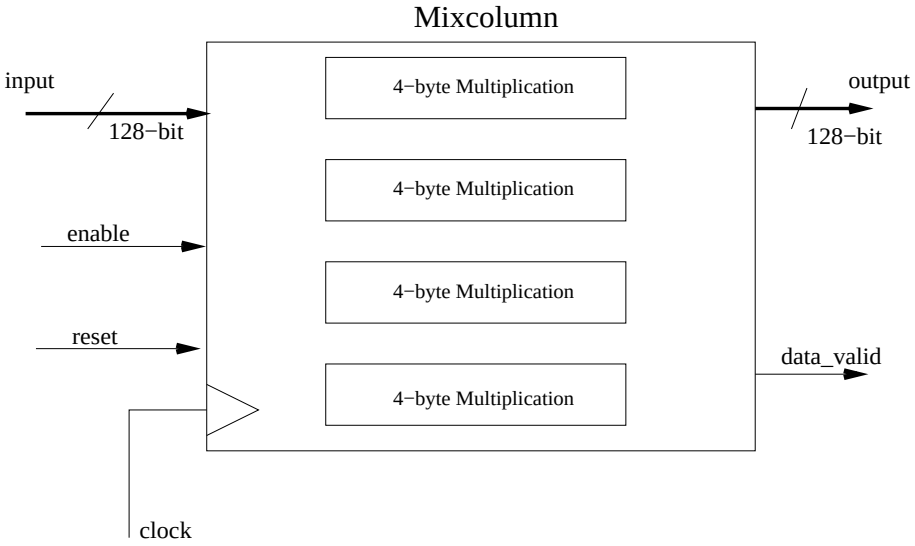


Figure 3.10: The Mixcolumn block scheme

3.4.5 The design verification

In order to verify the design on functional correctness, the Modelsim (v5.5) CAD tool is used. This tool allows fast and accurate simulation of the design using a test bench file. In this file signals are assigned to each input pin. Note that only the functional

behavior and not the timing behavior of the design was tested with this tool. The testing procedure was as follows. Several random 128-bit test vectors were generated. The outcome of these vectors were calculated by hand and verified by a small self written program that was based on the Mixcolumn and the Inverse Mixcolumn transformations. Afterwards, the test vectors were applied to the design. For example, according to the calculation by hand and the verification program, test vector "D4 BF 5D 30 E0 B4 52 AE B8 41 11 F1 1E 27 98 E5" will result in the following outcome: " 04 66 81 E5 E0 CB 19 9A 49 F8 D3 7A 28 06 26 4C". Likewise the inverse operation on this outcome vector will result into the input test vector. The simulation result of this test vector is depicted in figure 3.11. As shown, the input for the Mixcolumn hardware unit will result in the precalculated outcome. Likewise, the outcome of the Inverse Mixcolumn hardware unit will result in the input test vector. Using other test vectors will also result in the expected values. Therefore, it is assumed that the design is functionally correct.

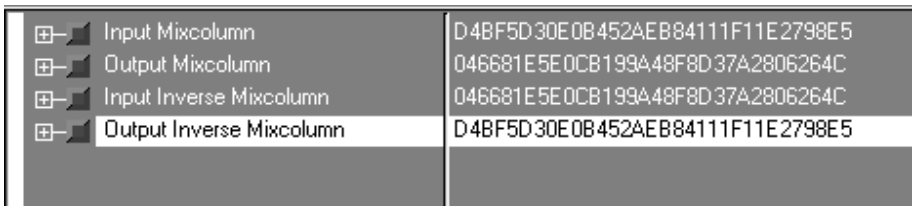


Figure 3.11: Design verification of the arithmetical implementation

3.4.6 The synthesis results

In order to obtain the desired reconfiguration files, both the Mixcolumn and the Inverse Mixcolumn hardware units are synthesized with Leonardo spectrum (v2001-1a32) CAD tool. Leonardo Spectrum is a synthesis tool which accepts either VHDL or Verilog as its input and generates and output that consists of net-list of library parts for the chosen FPGA device. All the designs were synthesized for two devices Altera Apex EP20K1000 and Xilinx Virtex II pro 2VP20.

Leonardo spectrum reports for each synthesized design, various circuit characteristics. There are three important parameters that characterizes the circuit, these are: area, critical path and throughput. The required FPGA area is an important factor of the total design costs. First of all, the costs are directly related to size of the FPGA device. The larger the required FPGA is, the higher the cost will be. In general, it is also assumed that the cost of an FPGA device is directly proportional to the area capacity. The second relation that is coupled to the size of the integrated circuit is the power consumption. The larger the required area, the more energy will be consumed. It also has been shown that power consumption grows linearly by increasing the clock frequency [20].

Unfortunately, different FPGA vendors publish their FPGA sizes using different terminology and counting schemes, which complicates the comparison of synthesis results.

The size of a design which is synthesized for a certain FPGA device is expressed in either Logic Cells (LC) or in Configurable Logic Blocks (CLB) Slices. The basic building block for the Altera Apex 20KE devices is the LC. Each LC consists of a 4-input look-up table (LUT), a register, and additional carry and cascade logic. The basic building block of Xilinx Virtex II pro devices is the CLB Slice. According to the Xilinx Virtex II pro data sheet [20], a CLB Slice contains two registers, a 4-input function generator that can be used as a 16-bit distributed RAM, a 4-input LUT or a 16-bit variable tap shift register and additional carry and cascade logic. In other words, the basic building block of the Xilinx FPGA device, compared to the Altera's basic building block, has an extra register.

The second parameter, critical path, is defined as the amount of time required by the circuit to process a single data-block. This parameter is established by the slowest part of the circuit and it is expressed in seconds. Note that, the process time of a circuit can also be expressed in clock cycles, what is called the latency of the circuit. The throughput, on the other hand, is expressed by the amount of bits and the time that is required for the process. The maximum of the throughput is established by slowest part of the circuit and therefore this parameter is directly related to the critical path parameter. More preciously, the maximum throughput is defined as follows:

$$Maximum\,throughput = \frac{number\,of\,processed\,bits}{critical\,path}\,[bits/s] \tag{3.9}$$

Table 3.2: The maximum operating frequency for the Mixcolumn and Inverse Mixcolumn hardware units

FPGA device	**Mixcolumn**	**Inverse Mixcolumn**
Altera Apex EP20KE1000	132.6 MHz	145.9 MHz
Xilinx Virtex II pro 2VP20	145.9 MHz	182.7 MHz

The maximum operating frequency for the hardware units are calculated with Leonardo Spectrum. These results are depicted in table 3.2. For our thesis it is assumed that the MOLEN processor will operate at 1 GHz, therefore the hardware designs are synthesized for 1 GHz. The synthesis results of the designs are presented in table 3.3 and table 3.4. The results are expressed in critical path, maximum throughput and occupied FPGA cells. Leonardo also reports the utilization of a design, which is the occupation in terms of percentage of the total FPGA size.

The results in table 3.3 and table 3.4 show that the critical path of the Mixcolumn hardware unit is longer compared to the critical path of the Inverse MixColumn hardware unit. However the occupied area of the Inverse MixColumn hardware unit is larger. As described in section 3.4.1, the Inverse Mixcolumn transformation resulted in more complex bit dependency patterns then of the Mixcolumn transformation, therefore, the larger area occupation was expected. Note that, since only a small part of the FPGA area is utilized, the design can also be realized in a smaller low cost FPGA.

Table 3.3: Mixcolumn hardware unit synthesis parameters

FPGA device	**Critical path delay** (ns)	**Maximum Throughput** (Gbit/s)	**Occupied Area**	**Utilization** (%)
Altera Apex EP20KE	6.54	19.6	368 LCs	0.96
Xilinx Virtex II pro 2VP20	5.18	24.7	144 CLB slices	0.26

Table 3.4: Inverse Mixcolumn hardware unit synthesis parameters

FPGA device	**Critical Path delay** (ns)	**Maximum Throughput** (Gbit/s)	**Occupied Area**	**Utilization** (%)
Altera Apex EP20KE	5.85	21.9	632 LCs	1.65
Xilinx Virtex II pro 2VP20	4.47	28.6	328 CLB slices	3.53

Table 3.3 and table 3.4 show that the designs are faster on the Xilinx device. This is explained by the fact that the Xilinx Virtex II pro 2VP20 device is based on a newer technology. From the data sheets one can see that the Xilinx device is based on 0.13-Micron technology, while the Altera device is based on 0.22-Micron technology. Another possible explanation for the differences, is that Xilinx utilizes a different synthesize technique, which results in a better optimization for our designs. Due to the fact that the first MOLEN processor prototype was build on the Virtex II pro device [18], the synthesis results of this device will be used for the simulation of the MOLEN AES.

As described in chapter 2 the MOLEN reconfiguration microcode is generated from the FPGA configuration file that is produced by the synthesis tool. During this generation process, the FPGA configuration file is augmented with microinstructions such as the end_op microinstruction that was discussed earlier. Afterwards, it is required to store the microcode in the systems main memory. The process of preparing the microcode for its final alignment into the memory is referred as microcode finalization. Since the MOLEN architecture will only be simulated in our case, the generation of the set microcode is not carried out. However, a detailed description of the generation procedures can be found in [11] and [18].

The critical path values, that are depicted in table 3.4 and table 3.3, are important parameters for the simulation of the MOLEN AES. The fact is that these parameters will set the execution time of the two hardware engines which is required in order to the mimic these hardware engines. Another important parameter for the simulation process is the *reconfiguration latency*. This parameter refers to the reconfiguration time of the FPGA. According to the *Virtex-II Pro Platform User guide* [20] the reconfiguration time

of the Virtex device is 164.29 ms. In other words, it will take 164.29 ms to configure the whole FPGA. Since the MOLEN processor will operate at 1 GHz, the *reconfiguration load latency* parameter will be set to 164,290,000 cycles. Note that a worst case scenario is assumed, therefore partial configuration will be not utilized. In the next chapter, these parameters will be described and discussed in more details.

Another parameter that can be important for simulating the MOLEN AES is the size of the reconfiguration file. In case that partial reconfiguration is utilized, there is latency accompanied to loading to the *rho*-Control store and a latency that is accompanied to the reconfiguration of the CCU itself. Note that in case of complete reconfiguration, this loading latency can be ignored, since the reconfiguration process itself is takes long time. In the next chapter this will be discussed in more details. The synthesis process showed that the size of the reconfiguration file is 178 Kbytes and 247 Kbytes for respectively the Mixcolumn hardware unit and the Inverse Mixcolumn hardware unit.

3.5 The MOLEN AES

The design methodology of the MOLEN AES was illustrated in figure 3.1. The last design step is to modify the AES cipher with the SET and EXECUTE MOLEN instructions in order to utilize the implemented AES hardware engines. As discussed before, the AES ANSI C source code v2.2 [6] is used as the reference for the developing the MOLEN AES. Note that the MOLEN AES will be compiled using the compiler of the SimpleScalar tool-set. In this section, the modification of the reference code is described.

3.5.1 Function descriptions of the Transformations

As concluded in the previous sections, the KeyScheduler, AddRoundKey, and the Byte-Sub transformations will be performed on the GPP, while the MixColumn, Shiftrow, Inverse MixColumn and the Inverse Shiftrow transformations are performed on the RP during the round operations. Note that in case all the round operations are performed, the Shiftrow transformation and the Inverse ShiftRow transformation are also performed on the GPP. As described before, the KeyScheduler generates ten round keys from the initial key. The function description of the KeyScheduler is as follows.

Prototype:

```
void rijndaelKeySched (word8 k[4][4], word8 W[11][4][4])
```

Description:

This function generates ten round keys.

Arguments:

- k[4][4] = the initial key;
- word8 W[11][4][4] = a three dimensional buffer.

Changed variables:

- W[11][4][4] = buffer containing the initial key and ten round keys

As described in section 3.2, the AddRoundKey transformation adds a roundkey to the cipher state. The function description of AddRoundKey is as follows.

Prototype:

```
void KeyAddition(word8 a[4][4], word8 rk[4][4])
```

Description:

This function adds a round key to the cipher state.

Arguments:

- a[4][4] = the cipher state;
- rk[4][4] = a round key;

Changed variables:

- a[4][4] = the cipher state;

The ByteSub transformation replaces each element of the state by using a certain S-box. As stated before the S-box is a 256-bytes substitution table. The function description of the ByteSub transformation is as follows:

Prototype:

```
void ByteSub(word8 a[4][MAXBC], word8 box[256])
```

Description: This function replaces every byte of the cipher state by a nonlinear S-box.

Arguments:

- a[4][4] = the cipher state;
- box[256] = the S-Box.

Changed variables:

- a[4][4] = the cipher state;

As described in section 3.2, the ShiftRow and the Inverse ShiftRow transformations shift the cipher state row. Both transformation are implemented by the function *ShiftRow*. An extra argument d is utilized to indicate the cipher direction. The function description of *ShiftRow* is as follows:

Prototype:

```
void ShiftRow(word8 a[4][4], word8 d)
```

Description: This function replaces every byte of the cipher state by a nonlinear S-box.

Arguments:

- a[4][4] = the cipher state;
- d = cipher direction (0 for encryption direction, 1 for the decryption direction)

Changed variables:

- a[4][4] = the cipher state;

3.5.2 The reconfiguration and execution microcode

As described in section 3.4.6, the reconfiguration microcode contains the FPGA device reconfiguration procedure. As discussed in the previous section, reconfiguration microcode is generated from the FPGA configuration file, that is produced by the synthesis tool. However for our case, since the MOLEN AES will be only simulated, this generation process is not carried out.

In order to mimic the reconfiguration microcode, two variables are utilized, these are:*reconfbin1* and *reconfbin2*. The reconfiguration microcode for the Mixcolumn hardware unit is represented by *reconfbin1*, while the reconfiguration microcode for the Inverse Mixcolumn is represented by *reconfbin2*. Both are declared as an integer array. From the synthesis results, it was shown that reconfiguration consists of 178000 bytes and 247000 bytes for respectively the Mixcolumn and the Inverse Mixcolumn hardware unit. Therefore, *reconfbin1* contains 178000 array elements and *reconfbin2* contains 247000 array elements. Note that since the real content of these buffer is not known, all the array elements are set to zero.

As described in chapter 2, the execution microcode is utilized for the regulation of the execution process on the CCU. Since no-regulation of the execution process for the MOLEN AES is required, utilizing the end_op microinstruction is sufficient. This microinstruction is represented by an array, that contains one array element, which is set the value zero.

Both type microcode, that is reconfiguration microcode and execution microcode, are extended with an extra flag, that is aligned at the beginning of the microcode. This flag, represents to which hardware engine the microcode is related. In the next chapter, this flag will be described in more details.

3.5.3 Scheduling the SET and the EXECUTE MOLEN instructions

In order to minimize the effect of the reconfiguration latency, the SET instruction will be scheduled far ahead of the EXECUTION instruction. In figure 3.12 the flow diagram of the MOLEN AES is illustrated for the encryption direction. As illustrated, the hardware reconfiguration process is scheduled before the ciphering process takes place. In order to analyze our MOLEN AES on its lowest performance, a worst case scenario is assumed and therefore only the C-SET instruction is scheduled. Note that the parameter of the SET instruction is the memory address of the beginning of the reconfiguration microcode. Executing the SET instruction will lead to fetching the reconfiguration microcode from the main memory to the ρ-Control Store and be executed afterwards. After the set

phase is completed, the AddRoundKey transformation is performed. Subsequently, the cipher will enter the first round operation, by performing the ByteSub transformation. After the ByteSub transformation, the cipher state will be loaded to the encryption hardware engine. Since the SimpleScalar compiler does not support additional registers (without modifying the compiler), the XREGS are not utilized. For our implementation the argument passing is done as follows. Note that for the simulation process it assumed that the EXECUTION instruction has more than one parameter. The beginning of the cipher state memory address, is one of the parameters of the EXECUTION instruction. The other parameter is the beginning of the microcode memory address. In the next chapter, argument passing is described in more details. The execution of the EXECUTE MOLEN instruction will load the cipher state to the hardware engine, perform the appropriate calculation and store the results back to the memory. Note that for the MOLEN architecture the results are stored by utilizing the XREGS, however, for our implementation the store memory address is also an argument of the EXECUTE MOLEN instruction. After the execute phase is finished, the AddRoundKey transformation is performed, which is the last transformation of the round operation. As described before there are nine round operations. In case that all these round operations are performed, the ByteSub, the ShiftRow and the AddRoundKey will be consecutively performed. The reference source code that is modified for the MOLEN platform is depicted in Appendix C.

3.6 Chapter summary

In this chapter, the design of the MOLEN AES was described. As discussed in this chapter, the performance of the MOLEN AES will be established by executing the non time-critical operations of the AES algorithm by the GPP while the time-critical operations will be executed by special hardware engines. In order to obtain the time-critical AES operations, two methods were used. The first method, which was based on analyzing AES transformations on arithmetical level, showed that the MixColumn and the Inverse MixColumn transformation are the most time-critical operation. The second method, which was based on a profile simulator, showed that the MixColumn and the Inverse MixColumn transformation contain the most executed instructions, and that most of these instructions are related to integer computations. Therefore it was concluded that the Mixcolumn and Inverse MixColumn transformation are the most time-critical operations. Both transformation were described in VHDL and verified on functional correctness with the Modelsim CAD tool. In order to obtain the desired reconfiguration files, both the Mixcolumn and the Inverse Mixcolumn hardware units were synthesized with Leonardo spectrum (v2001-1a32) CAD tool. The synthesis results showed that the critical path of the Mixcolumn hardware unit is longer compared to the critical path of the Inverse MixColumn hardware unit. However the occupied area of the Inverse MixColumn hardware unit is larger. Since, the VHDL description of the Inverse Mixcolumn transformation resulted in more complex bit dependency patterns then of the Mixcolumn transformation, the larger area occupation was expected. Since the MOLEN AES running on the MOLEN processor will only be simulated, the generation of the set microcode was not carried out. Finally, in this chapter the the AES cipher was

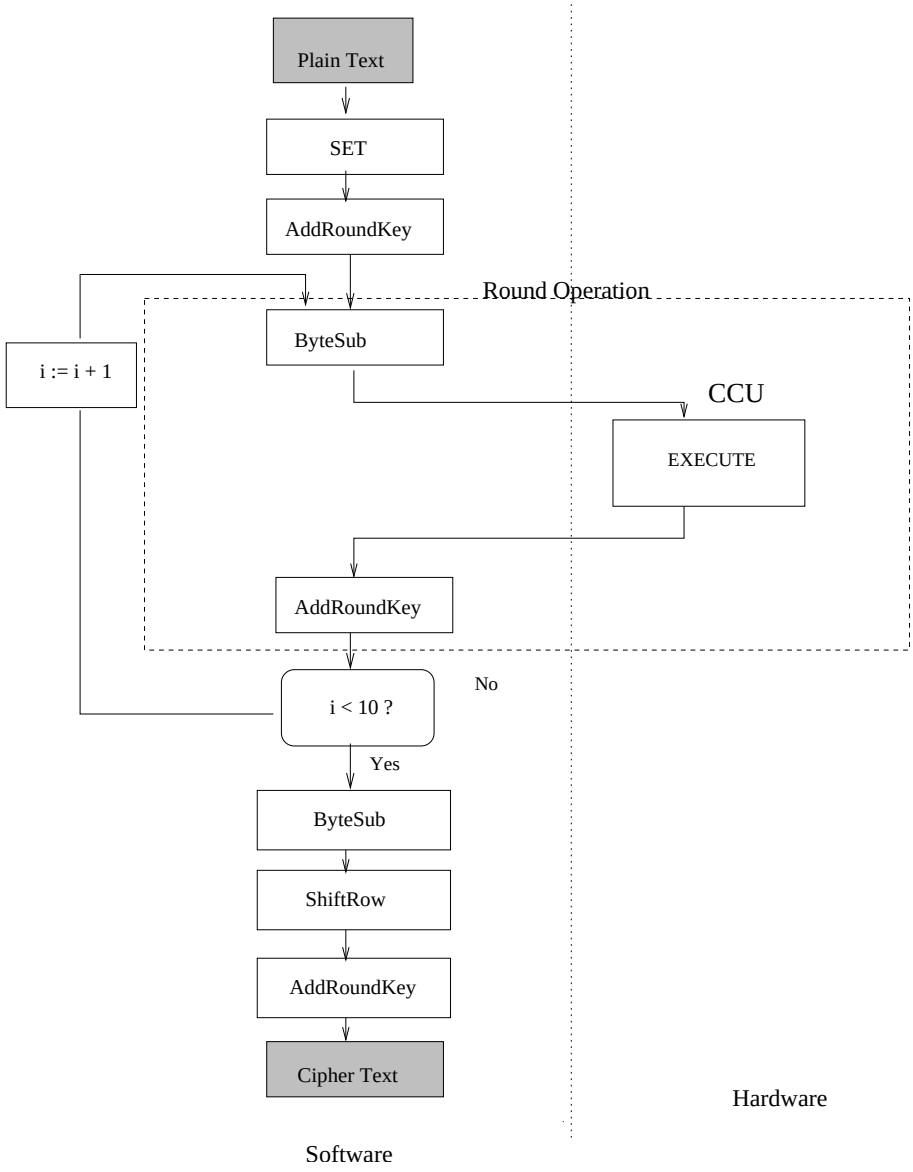


Figure 3.12: The diagram flow of the MOLEN AES

modified with the SET and EXECUTE MOLEN instructions. In order to minimize the effect of the reconfiguration latency, the SET instruction was scheduled far ahead of the EXECUTION instruction.

4 The Simulation Framework

This chapter introduces a simulator that will cycle-accurately mimic the MOLEN processor. This simulator is based on a modular extension of the SimpleScalar tool-set and in particularly the sim-outorder tool. This extension is based on the work of Stephan Wong [16], who modified the sim-outorder simulator of SimpleScalar(v2.0) for the MOLEN architecture. This chapter describes the construction of this simulator and it is organized as follows. Section 4.1 gives an introduction to simulation in general, highlights the features of the SimpleScalar tool-set and discusses why the sim-outorder simulator has been chosen above the other simulators that come with the tool-set. Section 4.2, discusses the extension of the SimpleScalar ISA and defines the MOLEN instructions that were introduced. Section 4.3 describes the modular structure of sim-outorder and introduces a new module for simulating the MOLEN processor. Section 4.4 describes how the timing characteristics of the GPP, RP and the arbiter are simulated, while section 4.5 describes how these units on functional level are simulated. And finally, section 4.6 overviews the variables and functions that are related to the application interface of the new module.

4.1 The SimpleScalar tool-set

The MOLEN AES performance results, which will be presented in the next chapter, strongly depend on the simulator. Since the word simulator has a large scope, the definition of the word simulator used in the context of this thesis is as follows: a *simulator* is a software program which mimics the behavior of a computing system.

Performance analysis of modern processors often requires sophisticated simulators written in high level program languages, e.g. C, C++ and Java. SimpleScalar [9], designed at the University of Wisconsin, is a tool-set of such simulators. Besides several simulators, this tool-set consists of a modified gcc compiler, assembler, linker and a profiling tool. The simulators, implemented in program language C, can model a variety of hardware platforms ranging from not pipelined processors to detailed dynamically scheduled micro architectures. In practice, this tool-set is used for various purposes e.g. program performance analysis, detailed micro architectural modeling, and hardware-software simulation.

SimpleScalar supports the following program code languages: C, Fortran or SimpleScalar's assembly which is a super set of MIPS assembly language. The compiler/assembler is a GNU C Compiler (GCC) compiler that is modified for the SimpleScalar architecture. SimpleScalar's binaries can be executed using one of the following simulators:

- **Sim-fast** is a fast instruction interpreter, optimized for speed. It does not simulate

pipeline stages, caches, or any other part of the micro architecture.

- **Sim-safe** is a slightly different instruction interpreter than sim-fast. It checks for memory alignment and memory access permission on all memory operations. This simulator can be used for diagnostics, in case that a simulated program causes sim-fast to crash without clear explanation.
- **Sim-cheetah** simulates ranges of single level set-associative and fully associative caches.
- **Sim-cache** implements a functional cache simulator. Cache statistics are generated for a user-selected cache and the Table Lookup Buffer (TLB) configuration, which may include up to two level of instruction and data cache (with any level unified), and one level of instruction and data TLBs. No timing information is generated.
- **Sim-profile** provides detailed profiles on instruction classes and addresses, text symbols, memory accesses, branches, and data segment symbols. The profile information is actually the frequency of events that occur during the execution of a program. One common example of profile data is a count of how often each type of instruction (e.g. branch, load, store, ALU operation, etc.) is performed during the program execution.
- **Sim-outorder** is a detailed performance simulator. It simulates five pipeline stages, memory and cache accesses, and other internal processor functional units, cycle accurate. With this simulator, timing statistics are generated for a detailed out-of-order issue super scalar processor.

The goal of our simulator is to cycle-accurately simulate the execution of the AES cipher that is optimized for the MOLEN architecture. Therefore, the simulator has to approximate the real hardware very close. Since Sim-outorder is the only cycle-accurate simulator of the tool-set, this simulator is chosen for our simulator framework.

Sim-outorder has several advantage features. One of them is, that it utilizes a dynamic trace-driven simulation technique. This technique, also known as execution-driven simulation technique, generates traces on speculative basis. A sim-outorder trace is actually a record of the instructions sequence that would be generated by an application that is executed on the MIPS architecture. Sim-outorder controls the trace by directing the Program Counter (PC). Therefore, this simulator provides access to all data produced and consumed during program execution. Some other features of sim-outorder are the ability to extend the Instruction Set Architecture (ISA)[8] and the possibility to integrate new functional units. Note that for these features no modification is required of the compiler. Another nice feature is that, every pipeline stage is characterized by a separate function, what makes this simulator well-organized. Therefore sim-outorder permits design variant or even completely different designs to be modeled with ease. Futhermore, the SimpleScalar tool-set is freely distributed as open source for noncommercial purposes. This all, makes sim-outorder a suitable simulation environment for approaching the behavior of the MOLEN processor. A complete description of the SimpleScalar tool-set can be found in [9].

4.2 Extending SimpleScalar ISA with MOLEN instructions

The ISA of sim-outorder is derived from the MIPS-IV ISA [8]. One feature of SimpleScalar ISA is that the semantics has an extended 16 bit annotation field. This annotation field provides the possibility to extend the ISA without the need to modify the compiler. In this field, bits are allocated for the decoding of custom instructions. Sim-outorder of the SimpleScalar(V3.0c) tool-set, supports bit annotations from "/a" through "/p", what will set bits from 0 through 15, respectively. This implies that 2^{16} new instructions can be added to the ISA. Using this feature, the MIPS-IV ISA can be easily extended with the MOLEN instructions. The detection of MOLEN instructions is done by evaluating the annotation field bits.

As introduced in chapter 2, the complete MOLEN instruction set consists of eight instructions. For our simulator, the minimal πISA is implemented. In order to simulate the CCU's partially reconfiguration process, the P-SET instruction is also implemented. The bit annotations reserved for the MOLEN instructions are overviewed in table 4.1. To allow permanent and pageable reconfiguration and execution microcode to coexists, both the SET and EXECUTE instruction are extended with a pageable part and a resident part.

As described in 2, the exchange of data between the GPP and the CCU is established using the Exchange Registers (XREGS). However SimpleScalar compiler does not support XREGS. Therefore the registers fields of the annotated instruction are utilized to exchange parameters. Since, there are instructions with the maximum of three register-fields, three different memory location can be appointed. The register-fields are assigned as follows. The first register-field is assigned to exchange the arguments, in other words, the register-field refers to the beginning of the memory location that contains the arguments. The second register-field is assigned for the microcode. It refers to the beginning of the memory address that contains the reconfiguration microcode or the execution microcode. Note that, as discussed before, the reconfiguration microcode consists of a bitstream that will configure the CCU, however for our simulation, this bitstream is modeled by a large arbitrary chosen buffer. The third register-field is assigned for the results that is to be stored. Since three register-fields are utilized, the simulator requires an annotated instruction with three register fields, for example the addition instruction "ADD".

4.3 Simoutorder's structure and extension of this structure

According to the SimpleScalar hackers guide [9], sim-outorder is structured into two cores. These cores are known as the Emulation core and the Performance core. The Emulation core consists of instruction interpreters for the Advanced RISC Machines (ARM), x86, Power-PC (PPC) and Alpha instruction sets. More precisely, in this core, the host instructions, on binary level, are translated into a new instruction trace. Afterwards, the new trace is processed by the Performance core.

The Performance core correctly accounts the behavior of each instruction type.

Table 4.1: The MOLEN SET and EXECUTE instruction

Bit Annotation	Description	example
/b	the C-SET instruction (resident)	(ADD/b)
/a/b	the C-SET instruction (pageable)	(ADD/a/b)
/c/b	the P-SET instruction (resident)	(ADD/c/b)
/c/b/a	the P-SET instruction (pageable)	(ADD/c/b/a)
/c	the EXECUTE instruction (resident)	(ADD/c)
/a/c	the EXECUTE instruction (pageable)	(ADD/a/c)

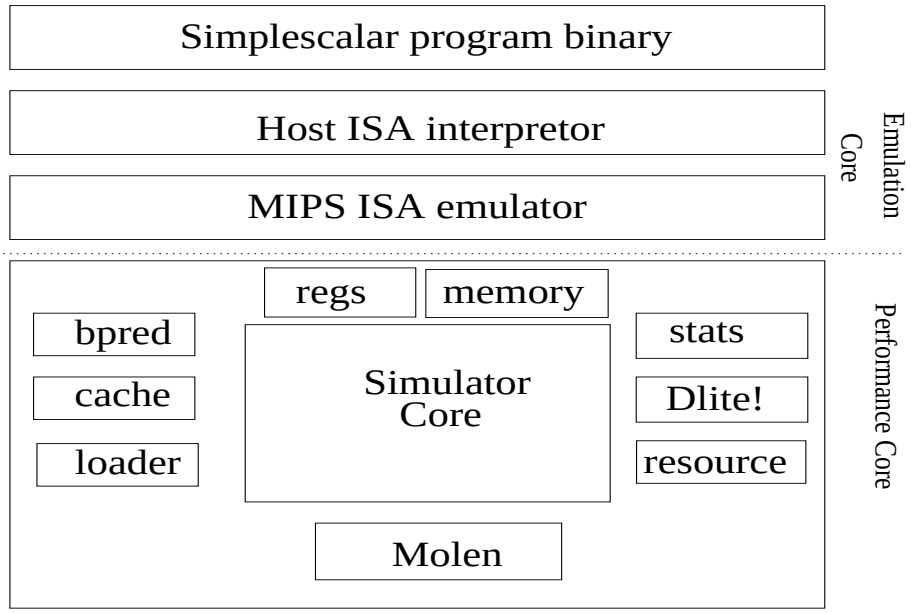


Figure 4.1: The structure of the sim-outorder simulator extended for the MOLEN architecture

Futhermore, the Performance core exists of a modular structure of several functional modules, such as cache simulator, branch predictor, simulator core etc. Note that, all the simulators within the tool-set are based on this structure, only some of them do not include all the functional modules.

The main module of sim-outorder is the simulator core: sim-outorder.c. This module exists of a resource pool definition, several function definitions that simulates various pipeline stages and the main function. The resource pool definition models Functional Units (FU), such as adders and ALUs, in terms of latencies. In sections 4.4, the resource pool definition will be overviewed in more details. There are five pipeline stages that sim-outorder simulates, these are: fetch, decode, issue, write-back and commit. Each

stage is implemented by a separate function. In section 4.5.4 these functions will be described in more details. The main function, sim-main(), consists of a loop that is organized as follows:

```
                    ruu_init();
                    for (;;) {
                            ruu_commit();
                            ruu_writeback();
                            ruu_issue();
                            ruu_dispatch();
                            ruu_fetch();
                            sim_cycle++;
                            if (max_insts && sim_num_insn >= max_insts)
return;
                            }
```

As depicted the loop consists of calls to the five pipeline functions. Each time that the loop is walked through, the simulation-cycle counter is incremented. In case that all the instructions are processed, the loop will be interrupted and the simulation results will be displayed.

The simulation of the MOLEN processor can be achieved by extending the resource pool definition and the five pipeline stage functions. All the source code that is related to the MOLEN architecture is placed in a new module called the MOLEN module. Figure

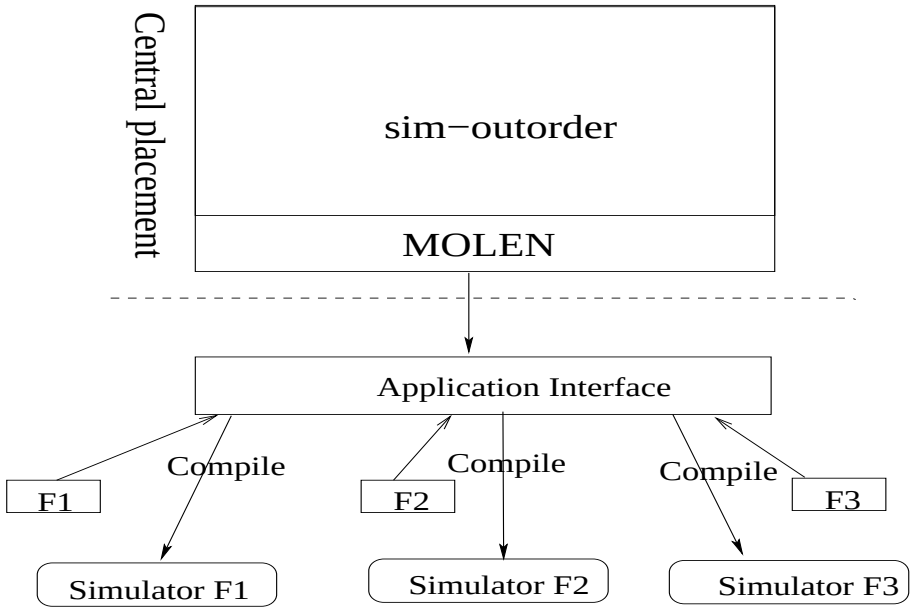


Figure 4.2: The utilization of the application interface

4.1 illustrates the structure of the sim-outorder's Performance core extended with this module. The MOLEN module exists of several functions that decodes and executes MOLEN instructions. Futhermore, it contains various functions related to the CCU and ρ-coded store unit. In section 4.5, this module will be described in more details. The choice for an extra module, instead of integrating the MOLEN architecture into the simulator core is based on the fact that an extra module can be easily installed with an updated version of the sim-outorder simulator.

In addition to this module, an application interface is build in order to simulate various applications on the MOLEN processor. In the application interface, various predefined parameters that are related to the MOLEN module are set for a certain application. In figure 4.2, the utilization of the application interface is illustrated. The application interface inserts a certain application code (indicated as F1, F2 and F3 in the figure) to the MOLEN module at compile time. Therefore, for each different application an unique simulator will be generated. Since only modification is made at compile time, the not installed simulator package (including the MOLEN module) can be centrally placed. The advantage of central placement is that various users can work on the same simulator package. In case that an update is done, every user can benefit from this update.

4.4 The timing characteristics of MOLEN hardware units

As described in chapter 2, the MOLEN processor consists of three units: the arbiter, the GPP, and the RP. The timing characteristics of the GPP are modeled by the resource pool definition of the sim-outorder.c file. In this resource pool the GPP functional units, such as ALUs and adders, are modeled in terms of two parameters: operation latency and the issue latency. The former, indicates when operation results, generated by a certain FU, will become available, while the latter determines how frequently instructions can be sent to the FU. The default definition of the resource pool, contains 4 integer ALUs, 1 integer MULT/DIV-unit, 4 FP adders, 1 FP MULT/DIV unit and 2 memory ports(R/W).

The timing characteristics of the RP can be modeled by extending the resource pool. As described in chapter 2, the RP consists of the $\rho\mu$-code unit and the CCU. In order to load microcode code, the $\rho\mu$-code unit must be ready to accept microcode. Therefore the timing characteristics of the $\rho\mu$-code is modeled by unit called the Rho_Control_Store. The timing characteristics of the CCU is modeled as follows. As discussed before, the operations of the RP can be divided into the set phase and the execute phase. Therefore, the CCU is modeled in terms of two FUs. One FU, which is called SetLogic, is related the set phase and the other FU, which is called ExecuteLogic, is related to the execute phase. The SetLogic represents the CCU during the set phase, the ExecuteLogic represents the CCU timing characteristics during the execute phase. The RP is added in the resource pool as follows:

```
{
    "Reconfigurable Processor",
    1, /* total instances of this unit */
    0, /* non-zero if this unit is busy */
```

```
    {
      { SetLogic, SET_OPLAT, SET_ISSUELAT },
      { ExecuteLogic, EXEC_OPLAT, EXEC_ISSUELAT },
      { Rho_Control_Store, OPLAT, ISSUE}
    }
  },
```

As described in chapter 2, the arbiter operation is based on directing instructions to either the GPP or RP. Futhermore, in order to control the state of the GPP, the arbiter also generates instruction sequence, which are known as arbiter emulation instructions. In [18], it is stated that arbiter emulation instructions can be reduced to one instruction. Therefore, the overhead that is produced by the arbiter is minimal, and therefore, the arbiter timing characteristics are not modeled.

4.5 The MOLEN module

The MOLEN module, which is represented by the molen.c file, consists of several functions that model the MOLEN units and extends sim-outorder pipeline stages. In the next sections these functions will be overviewed.

4.5.1 The arbiter

As discussed earlier, the timing characteristics of the arbiter are not modeled for our simulator. However, one important arbiter task is, directing the appropriate microcode to the $\rho\mu$-code unit. Since the arbiter is not modeled in our simulator, the following solution is used for directing microcode. Both type microcode, that is reconfiguration microcode and execution microcode, are extended with an extra flag, that is aligned at the beginning of the microcode. This flag, represents to which hardware engine the microcode is related. More precisely, for each application, all the hardware engines that are utilized during the application are defined in a variable called *configuration*. This variable is defined as an enumeration of hardware engines. Therefore, the microcode flag is actually the position of a hardware engine in this enumeration array. The *configuration* variable will be described in section 4.6 in more details. For the MOLEN microcodes the flag is set as follows:

```
reconfbin1[0] = 2;
reconfbin2[0] = 3;
```

4.5.2 The Custom Computing Unit

As described in [19] the CCU consists of reconfiguration blocks. For our simulation framework, it is assumed that these reconfiguration blocks are of equal size. The number of CCU blocks that are utilized for the simulator is fixed by the variable *CCU_BLOCKS*. The configuration status of these CCU blocks is modeled by the *CCU_config_status* variable. This variable is declared as one dimensional global array, of which each element

represents a certain hardware engine. In section 4.6, both the *CCU_config_status* variable and the *CCU_BLOCKS* variable will be discussed in more details.

During the simulation it is essential to verify whether the CCU is configured to a certain hardware engine. This verification is performed by the *check_CCU* function. The verification, that the CCU is configured to a specific hardware engine, is done by analyzing the *CCU_config_status* variable. In case that a certain configuration is present in this variable, this function will return the value one, otherwise it will return the value zero. The function description of check_CCU is as follows:

Prototype:

```
int check_CCU(configuration config)
```

Description: This function determines whether the CCU is configured to a certain hardware engine.

Arguments:

- conf = CCU configuration

Return values:

1 (in case that the CCU is configured to a certain hardware engine).

0 (in case that the CCU is not configured to a certain hardware engine).

Changed variables:

- none

The reconfiguration of the CCU itself, is performed by the *reconfigure_CCU* function. This function has one argument, which is a certain hardware engine. The reconfiguration process is as follows. The function first checks whether a not configured CCU block is present in the CCU. Note that a not configured CCU block is indicated as NOCONFIG. In case that a not configured CCU block is found, that block will be configured to the hardware engine. In case that all the CCU blocks are already configured, a randomly chosen old hardware engine is replaced by the new hardware engine. The function description is as follows:

Prototype:

```
void reconfigure_CCU(configuration conf)
```

Description: This function reconfigures the CCU by modifying the CCU_config_status parameter.

Arguments:

- conf = a certain CCU configuration.

Return values:
NONE

Changed variables:

- CCU_config_status

4.5.3 The $\rho\mu$-code unit

As described in chapter 2, the $\rho\mu$-code unit provides storage facilities to the microcode and determines which microcode block will be next executed. For simplification reasons, the determination for the next microcode block will not be simulated.

The storage facility will be simulated as follows. As described in [19], the ρ-Control Store consists of two storage sections for the reconfiguration microcode and two storage sections for the execute microcode. More precisely, for the reconfiguration microcode, a fixed part and a pageable part is allocated. Likewise, for the execute microcode a fixed part and a pageable part is allocated. For simulating these storage sections, four global variables are utilized, these are known as: *set_fixed*, *set_pageable*, *exec_fixed*, *exec_pageable*. The *set_fixed* variable represents the status of the fixed storage section for the reconfiguration microcode, while the *set_pageable* variable represents the status of the pageable storage section for the reconfiguration microcode. Likewise for the execute microcode, *exec_fixed* represents the status of the fixed storage section, while *exec_pageable* represents the status of the pageable storage section. In section 4.6, these variables will be discussed in more details. Note that since the ρ-Control Store is not simulated at functional level, but only on timing level, the microcode is actually not loaded.

During the simulation it is essential to verify whether the microcode of a certain hardware engine is present in the ρ-Control Store. This operation is performed by the function: *microcode_present*. The arguments of this function are: the instruction type (SET or EXECUTE), the microcode location (pageable or resident), the reference memory address of the microcode and the related CCU configuration. The function verifies if the set microcode or the execute microcode is present in the ρ-Control Store, by analyzing the *set_fixed*, *set_pageable*, *exec_fixed* and *exec_pageable* variables. The function returns the following output. In case that the microcode is present in either the fixed or pageable part, it returns value 1 otherwise it returns the value 0. The function description is as follows:

Prototype:

```
int microcode_present(instruction_type type,
       code_location loc,
       int memory_address,
```

```
configuration conf)
```

Description: This function checks if the microcode is present in the rho-Control Store.

Arguments:

- type = instruction type (SET or EXECUTE),
- loc = location of microcode (resident or pageable),
- memory_address = memory address of the pageable microcode,
- conf = a certain CCU configuration.

Return values:

1 (in case that the microcode is present in the rho-Control Store).

0 (in case that the microcode is not present in the rho-Control Store).

Changed variables:

- none

The loading of microcode into the pageable part of the ρ-Control Store, is performed by the function: *insert_microcode*. The arguments of this function are: the instruction type (SET or EXECUTE), the related CCU configuration the microcode store location (pageable or resident) and the memory address of the pageable microcode. This function first check whether there is an empty spot in the storage place. Note that an empty spot is represented by the element EMPTY. If this is the case, it modifies either the *set_pageable* or the *exec_pageable* variable. In case that no empty spot is found in this variable, a random replacement strategy is performed. The function description is as follows:

Prototype:

```
void insert_microcode(instruction_type type,
 code_location loc,
 configuration conf)
```

Description: This function loads the reconfiguration code or microcode in the pageable part of, respectively, the set- or execute sections of the rho-Control Store.

Arguments:

- type = instruction type
- loc = location of microcode
- conf = CCU configuration

Return values:
NONE

Changed variables:

- set_pageable (in case of an SET instruction)
- exec_pageable (in case of an EXECUTE instruction)

4.5.4 Sim-outorder's pipeline extension

As stated before, sim-outorder simulates five pipeline stages, these are: fetch, decode, issue, write-back and commit. Every stage is characterized by a separate function. These functions modify both a central component known as the Register Update Unit (RUU) and the system memory. The RUU is a reordered buffer of records that are known as reservation stations. Each reservation station in the RUU holds the results of a pending instruction. The structure of the reservation station, "struct RUU_station", is defined in sim-outorder.c. In order to save the decoding information of MOLEN instructions, this structure is extended with the following flags:

- **instruction_type inst_type** represents the SET or EXECUTE Instruction type;
- **code_location location** indicates if the microcode is pageable or resident;
- **configuration config** represents a certain CCU configuration;
- **int loading** indicates if it is required to load the microcode. Note that, since there is no boolean type in the C programing language, the value zero is used in case that no microcode is required to be loaded and the value one is used in case that microcode is required to be loaded;
- **int reconfigure** indicates if it is required to reconfigure the CCU. Value zero indicates that it is not required to reconfigure the CCU, while value one indicates that it is required to reconfigure the CCU;
- **int load_data** indicates if it is required to load data or not to load data. Value zero indicates that it is not required to load data, while value one indicates that it is not required to load data;
- **int store_data** indicates if it is required to store data or not store data. Value zero indicates that it is not required to store data, while value one indicates that it is required to store data;
- **int counter** represents a delay counter that indicates when the next operation can begin;

- **int reconf_address** contains the reference address of reconfiguration microcode;
- **int micro_address** contains the reference address of execution microcode;
- **int load_address** contains the reference address of the data that will be loaded to the CCU;
- **int load_address2** contains an extra reference address of the data which will be loaded to a certain hardware engine;
- **int store_address** contains the reference address where the calculated results will be stored.

Since every pipeline stage is characterized by a separate function, the pipeline can be extended in a transparent way. Therefore, the MOLEN module contains four functions that extend sim-outorder's pipeline. These functions are known as: *dispatch_molen*, *issue_molen*, *writeback_molen* and *commit_molen*. Figure 4.3 illustrates the extension of sim-outorder's pipeline stages. In every pipeline stage, besides the fetch stage, it is checked if the instruction is annotated. In case that it concerns a MOLEN instruction, the corresponding MOLEN "pipeline" stage is invoked. The next sections gives a detailed description of sim-outorder's pipeline stages and the extension of it.

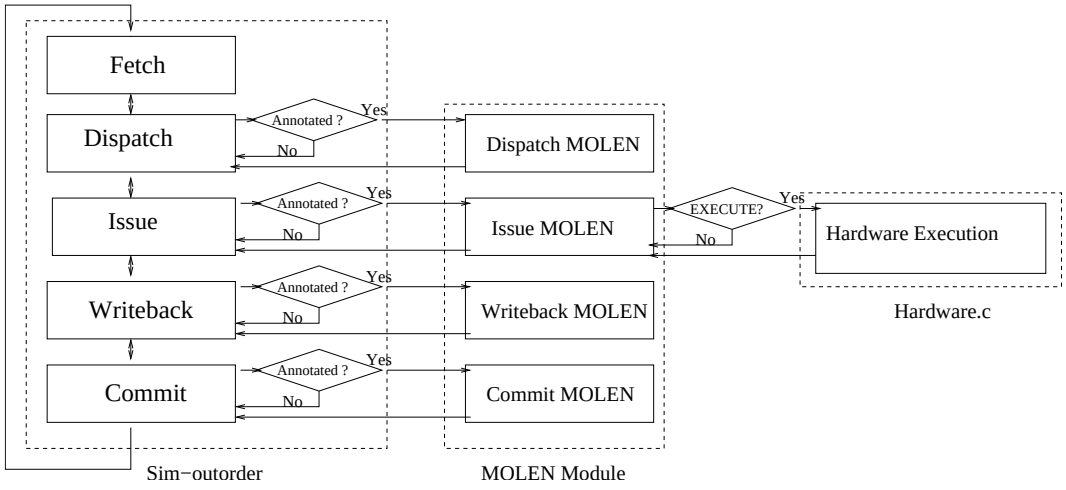


Figure 4.3: The pipeline stages of the simulation framework

4.5.4.1 Fetching instructions

The fetch stage is modeled by the function ruu_fetch. This function is based on a for loop in which as many instruction as the DISPATCH stage can decode are fetched from the Instruction Cache (I-cache). The instruction are placed into the dispatch queue for later decoding. In the simulator's default configuration, four instruction per cycle are

fetched. Note that in case the dispatch queue becomes full this routine will stop fetching instructions from the I-cache.

4.5.4.2 Dispatching MOLEN instructions

The dispatch stage of sim-outorder's pipeline is modeled by the *ruu_dispatch* function. This function is based on a for loop. Note that the amount of times this loop is walked through, is set by the instruction decode bandwidth parameter. In sim-outorder default configuration, four instruction are decoded for each simulation cycle. In the dispatch stage, instruction decoding and register renaming is performed. More precisely, the dispatch routine retrieves an instruction from the dispatch queue, allocates an entry into the RUU (and LSQ if required), decodes the instruction, updates the reservation station and finally places the decoded instruction into the scheduler queue.

For the MOLEN instructions decoding, *ruu_dispatch* is extended by the MOLEN dispatch function, *dispatch_molen*. The decoding of MOLEN instructions is done as follows. The *ruu_dispatch* retrieves an instruction from the dispatch queue, allocates an entry into the RUU and checks if the bits in the annotation fields are set. In case of an annotated instruction, the *dispatch_molen* function will be invoked. Several variables such as the stack recovery address, the reservation station, opcode operands and the annotation value are passed from *ruu_dispatch* to the MOLEN dispatch routine. This routine checks if the annotated instruction, is a MOLEN instruction. In case if it concerns not a MOLEN instruction, the routine will be terminated. Otherwise, it decodes the MOLEN instruction and updates the reservation station of that MOLEN instruction. In case that the MOLEN instruction is decoded and no errors are reported, *ruu_dispatch* will place the reservation station of the MOLEN instruction into the scheduler queue and marks it as ready to be issued.

In order to synchronize the execution sequence on the GPP and the RP, and to regulate the bus access, an extra flag is introduced for the dispatch stage. This flag is called *stop_dispatch* and it is set to value 1 in case of a detected MOLEN instruction. Setting this flag will freeze the Program Counter (PC) and will stop future dispatching of instructions. Therefore, the risk of system instability is minimized. In case that the results related to the MOLEN instruction are written back, this flag is set to value 0. For the synchronizing of the execution sequence, this method is not completely flawless, since the RUU can contain pending instructions in case of a detected MOLEN instruction. Therefore, it is possible that one of the pending instruction can be completed during the decoding of the MOLEN instruction, what will result in data inconsistency.

As stated before, the decoding of MOLEN instruction is done by *dispatch_molen*. For the SET instruction the decoding procedure is as follows. Since the arbiter is not simulated, it is first checked if the reconfiguration microcode is related to a defined hardware engine. Note that, as discussed before, all the hardware engines for a certain application are defined by a variable called *configuration*. In case that the reconfiguration microcode is not related to any defined hardware engine, a fatal error will be displayed and the simulation will be terminated. Otherwise, it will be verified if the CCU is already configured to the hardware engine. This is done by executing the function *CCU_check*. As stated before, this function determines whether the CCU is configured

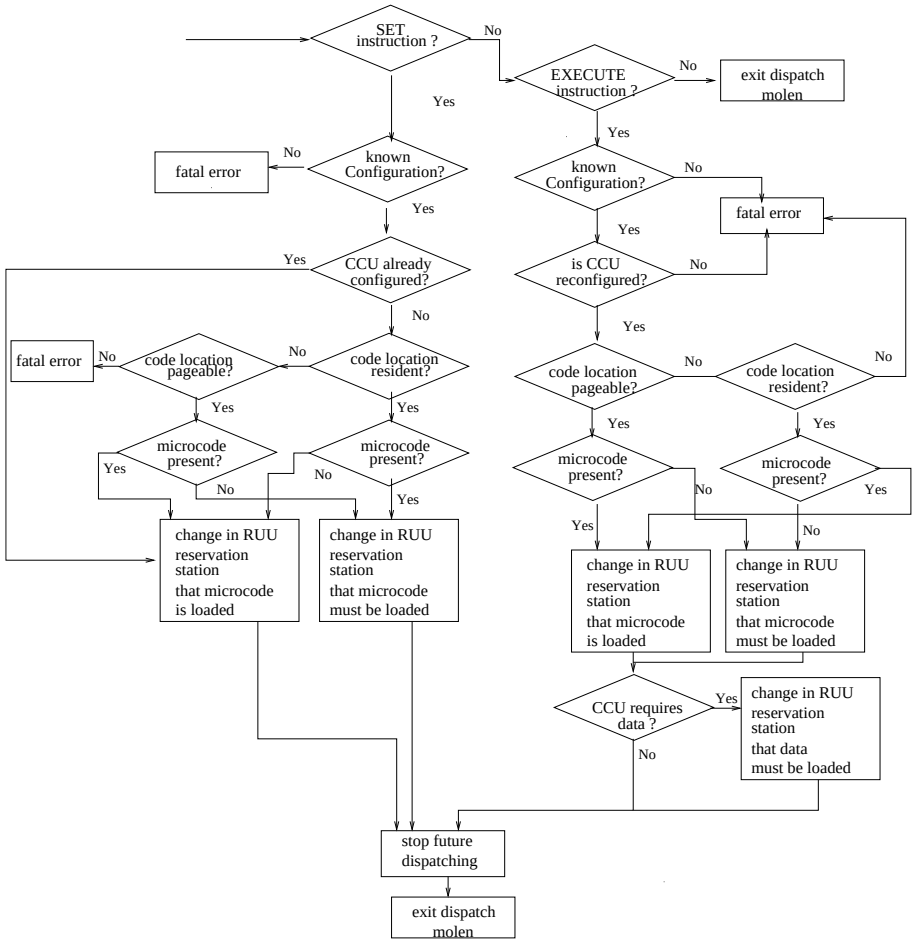


Figure 4.4: The diagram flow of the MOLEN dispatch routine.

to a certain hardware engine. In case that value one is returned, the reservation station of the SET instruction will be updated, future dispatching will be stopped (by setting the top_dispatch flag to 1) and the routine will be terminated. In case of a zero value, it can be concluded that the reconfiguration of the CCU is required, therefore, the *reconfigure* reservation flag is set to one. Afterwards, it is verified if the reconfiguration microcode is already loaded in the ρ-Control store unit. This is performed by the function *microcode_present*. In case that the reconfiguration microcode is not loaded, what is represented by a return , the *loading* reservation flag will be set to one, which indicates that the loading of the reconfiguration microcode is required. Otherwise, the *loading* reservation flag will be set to zero, which indicated that the reconfiguration microcode is present in the ρ-Control store unit. Afterwards, the stop_dispatch flag will be set to 1 and the routine will be terminated. The flow diagram of the *dispatch_molen* routine is depicted in figure 4.4.

In case of a MOLEN EXECUTE instruction, the decoding procedure is as follows.

First, it will be checked if the execution microcode is related to a known hardware engine. As described before, all hardware engines are defined by the configuration variable. In case that the execution microcode is not related to a known engine, a fatal error will be displayed and the simulation will be terminated. Second, it is verified that the CCU contains the hardware engine that is related to the execution microcode. This is performed by the function *CCU_check*. In case a zero value is returned, indicating that the CCU is not configured to the appropriate hardware engine, a fatal error will be displayed and the simulation will be terminated. Third, it is verified if the execution microcode is already loaded in the ρ-Control store unit. This is performed by the function *microcode_present*. In case that the execution microcode is not loaded, the *loading* reservation flag will be set to one, which indicates that the loading of the execution microcode is required. Otherwise, the *loading* reservation flag will be set to zero, which indicates that the execution microcode is already loaded into the ρ-Control store unit. Fourth, it is verified if the hardware engine (that is related to the execution microcode) requires the loading and the storing of data from and to the memory. This verification is done by, analyzing the *data_load_size* and *data_store_size* variables. These variables represents the size of the data that is to be loaded, respectively, stored, and are a part of the application interface. In case that these parameters are not zero, it can be concluded that it is required to load or store data. These variables are discussed in section 4.6 in more details. Finally, the stop_dispatch flag will be set to 1 and routine will be terminated. The function description of *dispatch_molen* is as follows:

Prototype:

```
void dispatch_molen(enum md_opcode op,
                    md_inst_t inst,
                    unsigned int pseq,
                    struct bpred_update_t *dir_update_ptr,
                    struct RUU_station *rs,
                    int stack_recover_idx,
                    int annotated_field
                    )
```

Description:

This function decodes the SET and EXECUTE MOLEN instruction.

Arguments:

- op = opcode
- inst = instruction code
- pseq = pipe-trace sequence number
- dir_update_ptr = branch predictor direction update pointer
- rs = an entry of the Register Update Unit (RUU reservation station)

- stack_recover_idx = branch prediction return stack recovery index
- annotated_field = Annotated value

Return values:
None

Changed variables:

- rs→inst_type;
- rs→location;
- rs→config;
- rs→loading;
- rs→reconfigure;
- rs→load_data;
- rs→store_data;
- rs→reconf_address;
- rs→micro_address;
- rs→load_address;
- rs→load_address2;
- rs→stor_address.

4.5.4.3 Issuing MOLEN instructions

The issue stage routine is modeled by the function *ruu_issue*. This function issues ready instructions from the scheduler queue to the appropriate Functional Units (FU). In case of an annotated instruction, the *issue_molen* will be invoked. This function is divided into a pre-issue part and the issue part. In the pre-issue part, the loading of microcode and data is performed, while the issue part, issues ready MOLEN instructions from the scheduler queue. The reason that the loading procedure is performed by this function instead of the *dispatch_molen* function, is that the *ruu_dispatch* function (which invokes *dispatch_molen*) can not stall the pipeline.

The issue-ing of MOLEN instructions is done as follows. The *ruu_issue* retrieves an instruction from the scheduler queue and checks if the annotation field is set. In case of a MOLEN instruction the *ruu_dispatch* will be invoked. This routine verifies if it is required to load microcode. If this is the case, the pipeline will be stalled and the microcode will be loaded. Afterwards, it is analyzed if it is required to load data. In case that it is required to load data, the pipeline will be stalled and the data will be loaded. And finally the MOLEN instructions will be issued.

For the loading process of both the microcode, the *counter* of the reservation station is utilized. Note that for simplification reasons, it is assumed that the loading of microcode is not performed through the caches. The flow diagram of the the microcode loading process is depicted in figure 4.5. The loading process for microcode is as follows. The *counter* is set to the loading latency of the microcode which is declared by

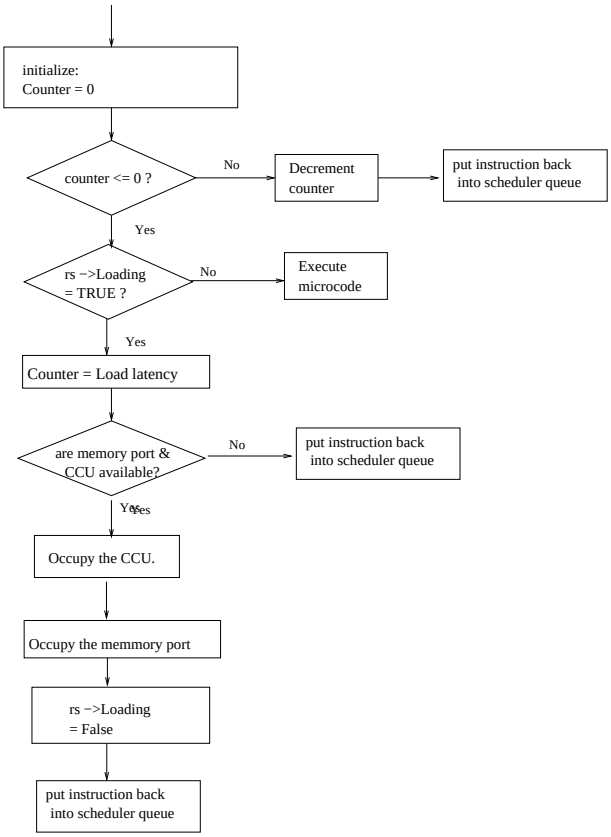


Figure 4.5: The flow diagram of loading microcode to the ρ-Control Store.

the *reconf_load_latency* variable. The *reconf_load_latency* variable represents the latency that is accompanied by the process of loading reconfiguration microcode to the ρ-Control store. This variable is a part of the application interface, which is discussed in 4.6 in more details. Each simulation cycle, the counter is decremented and the instruction is placed back into the scheduler queue. When the *counter* is zero, it will be checked if the ρ-Control store is available for loading the microcode. In case the ρ-Control store is already occupied, the instruction will be placed back into the scheduler queue. Otherwise, the microcode will be loaded by invoking the function *insert_microcode*. Afterwards, the *loading* flag is set to zero and the microcode loading process is accomplished.

Since the loading of data is going through caches, the data loading process is performed in a different way than that of the microcode loading process. The procedure is as follows. First, it is analyzed if it is required to load data, by checking the status of the *load_data* reservation flag. If this is not the case, the instruction will be placed back into the scheduler queue. Second, it will be checked if the memory port and the CCU are available. If this is not the case, the instruction will be placed back into the scheduler queue. Otherwise, the memory port and the CCU will be occupied. Third,

since the data is going through caches, the cache access latency is calculated by invoking the *cache_access* function, which is defined in SimpleScalar cache.c module. And finally, The calculated latency is scheduled into SimpleScalar event queue.

The issue process of the MOLEN instructions, is as follows. For the SET instruction, it is checked if the CCU is available. If this is not the case, the instruction will be placed back into the scheduler queue. Otherwise, the CCU unit will be occupied with the *reconfiguration_latency*. This variable is related to the latency that is accompanied with the reconfiguration process of the CCU. In section 4.6, this parameter will be described in more details. Note that this variable is also a part of the application interface. The reconfiguration process itself, is performed by invoking the *reconfigure_CCU* function. Afterwards, it is indicated that the SET instruction is issued, by setting the *issued* flag to value one.

For the EXECUTE instruction the issue procedure is as follows. First, it is checked if the CCU is available. If this is not the case, the instruction will be placed back into the scheduler queue. Otherwise, the CCU will be occupied. Second, the *execution_latency* will scheduled into SimpleScalar event queue. The *execution_latency* variable indicates when the calculation results of a certain CCU hardware engine becomes available. Note that this variable is also a part of the application interface. Third, the actual hardware algorithm of a certain CCU configuration is performed. This is done by invoking the *hardware_execution* function. Since the *hardware_execution* function is part of the application interface, this function is described in section 4.6. And finally, it is indicated that a certain instruction is retired, by setting the *issued* to value one. The function description of *issue_molen* is as follows:

Prototype:

```
void issue_molen(int annotated_field, struct RUU_station *rs)
```

Description:

This function issues MOLEN instructions from the scheduler queue to the appropriate hardware engines.

Arguments:

- annotated_field = Annotated value
- rs = an entry of the Register Update Unit.

Return values:
None

Changed variables:

- rs → reconfigure = 0 (indicates that CCU is configured);
- rs → loading = 0 (indicates that microcode is loaded);
- rs → counter (stalls the pipeline for certain amount of cycles);
- rs → load_data = 0 (indicates that data is loaded);
- rs → issued = 1 (indicates that instruction is issued).
- n_issued++

4.5.4.4 Write-back MOLEN instructions

The write-back stage is implemented in *ruu_writeback*. Each cycle, the event queue is scanned for both completed and not completed instruction. In case of a detected MOLEN instruction the function *writeback_molen* is invoked. In this stage, it can be concluded that all the results are written back. Therefore, this function sets the "stop_dispatch" flag to value 0, which unfreezes the Program Counter (PC) and allows future dispatch of instructions. The function description is as follows:

Prototype:

```
void writeback_molen(annotated_field)
```

Description:

This function unfreezes the PC and enables the decoding of instructions.

Arguments:

- annotated_field = Annotated value

Return values:
None

Changed variables:

- stop_dispatch = 0

4.5.4.5 Commit MOLEN instructions

The commit routine, implemented in *ruu_commit*, consists of a loop, in which the data caches (or memory) are updated with store values. Furthermore, in this routine the RUU entries that are ready to be committed, are removed from the RUU. In case of detected MOLEN EXECUTE instruction, the *commit_molen* function is invoked. This function updates the data cache with the store values that are related to the EXECUTE MOLEN instruction. More precisely, first, the memory port is occupied. Second, data is committed to data cache. And finally the data TLB misses are handled. In case that the memory port is not available, the routine will return a 1, which indicates that the RUU entry can not be removed. Otherwise the routine will return a 0, which indicates that the RUU entry can be removed. Not that the instructions are removed by the *ruu_commit* function. The function description of *commit_molen* is as follows:

Prototype:

```
int commit_molen(int annotated_field, struct RUU_station *rs)
```

Description:

This function stores the result from the EXECUTE instruction to the data cache and indicates if the entry can be removed from the RUU.

Arguments:

- annotated_field = Annotated value
- rs = Element of the Register Update Unit

Return values:
return value 0 (indicates that the entry can be removed from the CCU).
return value 1 (indicates that the entry can not be removed from the CCU).

Changed variables:

- none

4.6 The application interface

The application interface consists of predefined (global) variables that are set in the following two files: app_config.c and molen_config.c. The variables that are set in the app_config.c file are related to the user application, while the variables of the molen_config.c file are related to the initial state of the CCU and the ρ-Control store unit. Besides predefined variables, the application interface consists of a function called hardware_execution, in which all the execution algorithms of the hardware engines are described. This function is defined in the hardware.c file. In the next sections, the app_config.c, the molen_config.c and the hardware.c files will be described.

4.6.1 the app_config.c file

The variables of the app_config.c file can be distinguished into a definition part and a latency related part. In the definition part, all the CCU configurations that is required for a certain application are defined. The latency part is related to these CCU configurations. More precisely, the latencies of several process, such as reconfiguration latency, are set for each CCU configuration, in this part. This section will overview both parts in more details.

The definition part consists of one variable: *configuration*. This variable defines all the possible CCU configurations that are utilized for a certain application. As discussed before, this variable is utilized in order to analyze which hardware engine is related to a certain microcode. The *configuration* variable is defined as an enumeration of CCU configuration names. The first two positions of this enumeration are reserved by respectively "NOCONFIG" and "EMPTY". As described before, the former is actually not a CCU configuration, but it indicates that the CCU is not configured. The latter is also not a CCU configuration, but it indicates that there is an empty spot in the ρ-Control store. For the MOLEN AES, the *configuration* parameter is defined as follows:

```
enum configuration {NOCONFIG, EMPTY, MIX, IMIX};
```

As described before, the first two elements of the definition are reserved. The third element, "MIX", models the Mixcolumn hardware unit, which was synthesized in the

previous chapter. The "IMIX" configuration, models the Inverse MixColumn hardware unit, which was also synthesized in the previous chapter.

The latency part of the app_config.c file, consists of the following variables: *reconf_load_latency*, *micro_load_latency*, *execution_latency*, *reconf_latency*, *data_load_size* and *data_store_size*. All these variables are defined as one dimensional integer array. Its elements are related to the *configuration* definition. More precisely, each variable defines the latency of a certain process for all the CCU configurations. For example, *reconf_latency* is related to the latency that is accompanied with the reconfiguration process of the CCU. As discusses in the previous chapter a worst case scenario is assumed. Therefore, the CCU will be "completely" configured. As presented in section 3.4.6 of the previous chapter, the Virtex II pro 2VP20 device, which represents the CCU, has a configuration latency of 164,290,000 cycles. Therefore, this variable will be defined for the MOLEN AES as follows:

```
int reconf\_latency[4] = {0, 0, 164290000 , 164290000};
```

The above declaration indicates that the CCU configuration process to either the MIX or IMIX engine will take 164,290,000 cycles. Note that the first two array elements are set to zero, since the two element of the *configuration* were not related to a CCU configuration, what was discussed before.

The *reconf_load_latency* variable is related to the process of loading reconfiguration microcode to the ρ-Control store. In practice, this loading process takes place through the memory and caches, what is related to a variable latency. However, for simplification reasons, it is assumed that this process has a fixed loading latency. Note that the loading of reconfiguration microcode and the CCU reconfiguration process are consecutive processes and that the configuration CCU process is a very slow process. In case that only the complete set is utilized, the reconfiguration microcode loading latency effect can be ignored, since the bottleneck is not the loading of reconfiguration microcode to ρ-Control store, but the "loading" of the reconfiguration microcode into the CCU. In case that partial CCU configuration is utilized, the reconf_load_latency variable is of importance, since partial CCU configuration is done much faster. For the MOLEN AES, *reconf_load_latency* is declared as follows:

```
int reconf_load_latency[4] = {0, 0, 1, 1};
```

As described before, only the complete set will be utilized for our simulation, and therefore the latency of loading reconfiguration microcode can be set 1. Note that setting this variable to zero can result in system instability, therefore a latency of 1 cycle is chosen.

The *micro_load_latency* variable, is related to the process of loading execution microcode to the ρ-Control store. For the MOLEN AES, there is only one *end_op* microinstruction, therefore this variable is declared as follows:

```
int micro_load_latency[4] = {0, 0, 1, 1};
```

The *execution_latency* variable, indicates when the calculation results of a certain CCU hardware engine becomes available. The synthesis results, that were presented in

the previous chapter, shows that the critical path of the MIX hardware engine is 5.18 ns, while the critical path of the IMIX hardware engine is 4.47 ns. Since the MOLEN processor will operate at 1 GHz, it will take 6 cycles for the MIX hardware engine and 5 cycles for the IMIX hardware engine, before the result is available. For the MOLEN AES this variable is declared as follows:

```
int execution_latency[4] = {0, 0, 6, 5};
```

For some CCU hardware engines, it is required to load data from the memory or even to store data to the memory. Since the data is going through the caches, the loading and storing from respectively to the memory is accompanied with variable latencies. These latencies are calculated during simulation time, however they are dependent on the amount of bytes that is to be loaded or stored. The data size represented in bytes, is indicated by the variable "data_load_size" for the loading process, and by the variable "data_store_size" for the storing process. For the MOLEN AES these parameters are declared as follows:

```
int data_load_size[4] = {0, 0, 16, 16};
int data_store_size[4] = {0, 0, 16, 16};
```

As described in the previous chapter, it is required to load and store the state of the AES cipher to and respectively from the CCU. Since the state of the AES cipher is 16 bytes, both for the MIX and IMIX hardware engine these variables are set to 16 bytes. In appendix C the app_config.c file for the modified AES is depicted.

4.6.2 the hardware.c file

As stated before the hardware.c file consist of the function: *hardware_execution*. In this function the execution algorithm of all the CCU engines, which are defined by the *configuration* definition, are described. The arguments of this function are: the reference address of the execute microcode, the reference address of the data that is to be loaded to the CCU engine and the reference address of the results that is to be stored. The body of this function consists of a switch statement and several case labels. Each case label corresponds to a certain CCU engine. As described before, an extra flag is added to execution microcode in order to indicate to which CCU configuration the microcode belongs to. During the simulation, in case that the microcode flag matches one of the case labels, the corresponding algorithm code will be executed. Otherwise, the simulation will be stopped and an error message will be displayed. The function description of hardware_execution is as follows:

Prototype:

```
void hardware_execution (int field1_content,
                         int field2_content,
                         int field3_content)
```

Description:

This function executes the MOLEN instruction.

Arguments:

- field1_content = the reference address of where the results will be stored to,
- field2_content = the reference address of the execute microcode,
- field3_content = the reference address of the data that is required to be loaded to the CCU.

Return values:
None

Changed variables:

- field2_content

In appendix C the hardware.c file for the modified AES is depicted.

4.6.3 the molen_config.c file

The variables of the molen_config.c file are distinguished in to variables that are related to the initial state of the CCU configuration blocks and variables that are related to the ρ-Control Store Unit. As described before, the number of CCU blocks that is utilized for the simulator is defined by the variable *CCU_BLOCKS*. For the MOLEN AES this variable is declared as 4, what is arbitrary chosen. The *CCU_config_status* variable represents the status of the CCU blocks. For MOLEN AES this variable is declared as follows:

```
int config_status[4] = {NOCONFIG, NOCONFIG, NOCONFIG, NOCONFIG};
```

The above declarations indicates that the CCU blocks are not configured for the initial state.

As described before, the ρ-Control Store is modeled by the following parameters: *set_fixed*, *set_pageable*, *exec_fixed*, *exec_pageable* and *CCU_config_status*. As seen before these variables represent the status of the storage sections in the ρ-Control Store. The first two variables are related to the state of the set section of theρ-Control Store Unit, while the last two variables are related to state of the execute section of theρ-Control Store Unit. For the MOLEN AES the initial stage of these storage sections are declared as follows:

```
int set_fixed[4][2] = {{EMPTY,0},{EMPTY,0},{EMPTY,0},{EMPTY,0}};
int set_pageable[4][2] = {{EMPTY,0},{EMPTY,0},{EMPTY,0},{EMPTY,0}};
int execute_fixed[4][2] = {{EMPTY,0},{EMPTY,0},{EMPTY,0},{EMPTY,0}};
int execute_pageable[4][2] = {{EMPTY,0},{EMPTY,0},{EMPTY,0},{EMPTY,0}};
```

As depicted, the arrays are two dimensional global arrays. The first dimension represents the status of a certain section, while the section dimension represents the address where the microcode is located. As depicted, there are four storage sections for each microcode type. Note that the amount of sections are arbitrary chosen. The array element "EMPTY" indicates that the storage section is empty. In other words, the above declarations indicate that both the fixed and the pageable storage sections for both microcode type, are empty at the initial stage.

4.7 Chapter summary

In this chapter a simulator that will cycle-accurately mimic the MOLEN processor was presented. As discussed in this chapter, the SimpleScalar ISA was extended with the minimal πISA MOLEN instruction set, by using the semantics annotation field. Furthermore, in order to simulate the CCU's partially reconfiguration process, the P-SET instruction was also implemented. It was also discussed that the registers fields of the annotated instruction are utilized to exchange parameters. Since sim-outorder is the only cycle-accurate simulator of the SimpleScalar tool-set, this simulator was chosen for our simulator framework. As described in this chapter, sim-outorder consists of a modular structure of several functional modules, such as cache simulator, branch predictor, simulator core etc. The simulator for the MOLEN processor was achieved by extending the simulator core with an extra module, which was called the MOLEN module. The choice for an extra module, instead of integrating the MOLEN architecture into the simulator core was based on the fact that an extra module can be easily installed with an updated version of the sim-outorder simulator. The MOLEN module, which is represented by the molen.c file, consists of several functions that model the MOLEN units and extends sim-outorder pipeline stages. In addition to this module, an application interface was build in order to simulate various applications on the MOLEN processor. The application interface consists of predefined (global) variables that are set in the app_config.c and molen_config.c files. Futhermore, the application interface consists of a function called hardware_execution, in which all the execution algorithms of the hardware engines are described. As described in this chapter, this function is defined in the hardware.c file.

The MOLEN AES performance results

5

This chapter presents the simulation results of the MOLEN AES. Futhermore, these results are compared to other AES implementations. The organization of this chapter is as follows. In section 5.1, it is described under what conditions the simulations are performed. In section 5.2 the simulations results are presented and discussed.

5.1 The simulation conditions

Speed performance results are dependent of several aspects, such as the platform on which the simulations are performed, the used test-vectors and several assumptions that are made for the simulation process. In the next sections these aspects will be discussed in regard to the MOLEN AES simulation process.

5.1.1 The simulator platform

The machine that is used for the simulations, is based on an IBM compatible machine with an AMD XP 2400+ processor, running Linux RedHat (v9.0) operating System (OS). The memory that is installed on the machine is 256 MB (Mega Bytes). Furthermore, the machine is not a standalone, but it is part of a network system. Since, sim-out-order is stated as a cycle-accurate simulator, all produced numbers should be independent of the platform. However, there is a small aberration. The SimpleScalar Frequently Asked Questions (FAQ) document states the following:

"It is very difficult to produce the same exact execution each time a program executes on the SimpleScalar simulators. Many variations in any particular execution are possible, including:

- *calls to time() and getrusage() will produce different results;*
- *redirecting output will cause subtle changes in printf() execution;*
- *the size of your environment, which is imported into the simulated virtual memory space, affects the starting location of a programs stack pointer;*
- *small variations in floating point across platforms can effect execution.*

Fortunately, all variations are very small, on the order of a few thousand instructions at the most. This is insignificant compared to millions of instruction executed during a program."

Since the fact that the simulation results are dependent of the platform environment, a tolerance factor is introduced. As stated in the SimpleScalar FAQ, the variations are

in the order of a few thousand instructions. Since for the MOLEN AES the amount of executed instructions is related to the test-vectors sizes, the tolerance factor is defined as follows:

$$Tolerance factor = \frac{1000}{the\ number\ of\ executed\ instructions} * 100\ [\%] \qquad (5.1)$$

5.1.2 The simulator configuration

In Sim-outorder it is possible to configure several options, for example the bandwidth of the pipeline stages, the amount of Arithmetical Logic Units (ALUs) and other parameters. For this thesis, the following simulated machine configuration is used:

- bandwidth: 32 bits;
- 16KB L1 instruction cache, 32 bits line size, 1 way associative, Least Recently Used (LRU) replacement strategy;
- 16KB L1 data cache, 32 bits line size, 4 way associative, LRU replacement strategy;
- 256kb l2 unified cache, 64 bits line size, 4 way associative, LRU replacement strategy;
- 2 memory ports(R/W);
- 4 integer ALUs;
- 4 floating point ALUs;
- 1 integer multiplier/dividers available;
- 1 floating multiplier/dividers available.

Note that this configuration is the default machine configuration of sim-outorder. Futhermore, it assumed that the simulated processor operates at 1 GHz.

As discussed in the previous chapter, it is assumed that there is one CCU block and that it is not configured for the initial state. Further, it is assumed that the ρ-Control store is empty, in other words, the reconfiguration microcode is assumed to be resident. The exact files that are related to the application interface, can be overviewed in appendix C.2.

5.1.3 Compile options and testvector

The MOLEN AES software is compiled using the SimpleScalar's compiler. As stated before, this compiler is a modified version the GCC version 2.7.2.3 compiler. Since this thesis is focused on speed performance and not the code size, the compiler option "-03" is used. Furthermore, from all encryption mode, only the ECB mode will be used as reference. The testvector, which is arbitrary chosen, is a Joint Photographic Experts Group (JPG) picture. The picture has a resolution of 225 x 300 pixels and contains

29.276 bytes. The picture is depicted in figure 5.1. This picture was downloaded from the Rijndael homepage [6]. It is said that the picture symbolizes the DES cipher (bird) being replaced by the AES cipher (skull).

Figure 5.1: The testvector, represented by 29.276 bytes

5.1.4 The performance gain metric

As discussed in chapter 2, cipher speed performance is expressed by the metric throughput, which is defined as the number of bits that are processed during a time period. Since our processor operates at 1 GHz, throughput can be defined as follows:

$$Throughput = \frac{number\ of\ processed\ bits}{executiontime} * 1GHz\ \ [bits/s] \tag{5.2}$$

The *execution time* is the minimal amount of clock cycles that is required to perform the cipher process, without considering the operations that are required by the KeyScheduler and by the MOLEN AES API. Futhermore, the loading of microcode and the CCU configuration time, is also not included in this calculation.

In order to analyze the MOLEN AES performance in an efficient way, an extra metric is introduced. This metric is known as speedup and it is defined for this thesis as follows:

$$Speedup = \frac{execution\ time\ of\ the\ MOLEN\ AES}{execution\ time\ of\ reference\ AES} \pm Tolerancefactor \tag{5.3}$$

Speedup is defined as the ratio between the MOLEN AES execution time and the reference AES v2.0 execution time.

5.1.5 Dealing with overhead

As depicted in chapter 3, the AES API produces overhead. To the calculate the speedup gain, it is necessary to filter out the effect of the overhead. Therefore, the simulator is extended with an extra function that is called *trace_routine*. This function displays the amount of cycles that a certain routine is performed in. For this function the "/a" bit annotation is reserved. The flow diagram of this function is illustrated in figure 5.2. In case that the "/a" annotation is detected, it will be verified if a global variable that is called *sim_cycle_save1* is set to zero. If this is the case, the value of the simulation cycle counter will be stored in the sim_cycle_save1 variable. Otherwise, the sim_cycle_save1 will be extracted from the simulation cycle counter and the sim_cycle_save1 will be set to zero. The result is stored in the func1_cycle variable and displayed in the simulation numbers overview. Note that for this function any instruction, even a NOP instruction, can be annotated.

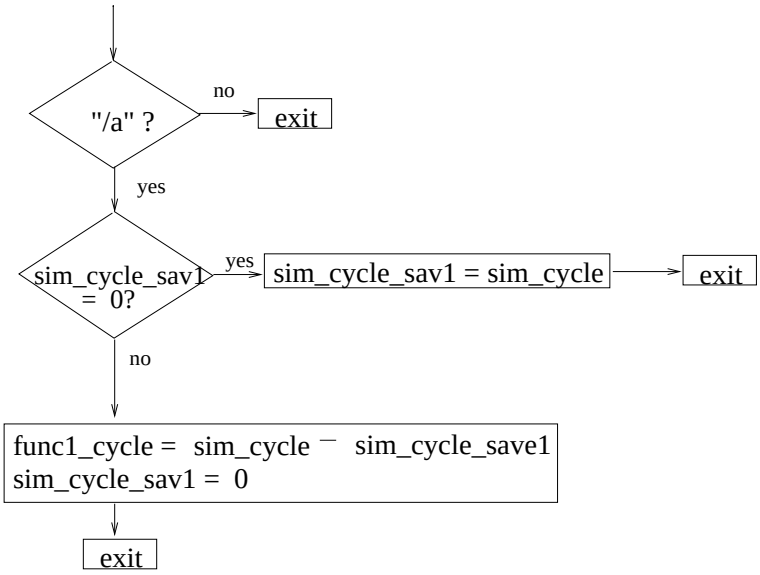


Figure 5.2: The trace routine function

For the MOLEN AES, there is only one important routine: the cipher process on the data blocks. Note that the KeyScheduler is not included in this routine, since it does not operate on data blocks but on the 128-bit key. The procedure for tracking the amount of cycles during the encryption process is as follows. Before the first cipher transformation (KeyAddition) starts, the "NOP/a" annotated instruction is executed. At this point, the simulation cycle counter will be saved by the function trace_routine. After the last Keyaddition is performed, the "NOP/a" annotated instruction will be

executed again. At this point, the amount of sim_cycle_save1 will be extracted from the simulation cycle counter, what will result in the amount of cycles that is elapsed during the encryption process. The result is stored in func1_cycle variable and displayed in the simulation numbers overview. Note that this procedure is also applicable for simulating the decryption direction.

5.2 The performance results

In order to determine the speedup gain, both the MOLEN AES and the reference AES code v2.2 are simulated. First, both sources are compiled by using SimpleScalar's compiler. Second, the binaries are simulated using the simulator that was described in the previous chapter. The performance results are overviewed in table 5.1 for the encryption direction. Due to a low tolerance factor, the effect of the platform environment can be disregarded. It is depicted that the speedup gain is 4.9. In other words, it can be concluded that the AES encryption process can be 4.9 times speeded up on the MOLEN platform.

Table 5.1: The performance results for the encryption direction

	MOLEN AES	Reference Code
Input Data size	29,276 bytes	29,276 bytes
Processor Clock Rate	1 GHz	1 GHz
Number of executed instructions	13,299,996	38,977,457
Tolerance factor	0.0075 %	0.0025 %
execution time	3,405,814 cycles	16,601,869 cycles
Speedup	4.9	1

Table 5.2: The performance results for the decryption direction

	MOLEN AES	Reference Code
Input Data size	29,276 bytes	29,276 bytes
Processor Clock Rate	1 GHz	1 GHz
Number of executed instructions	13,314,131	51,075,628
Tolerance factor	0.0075 %	0.0020 %
execution time	3,381,644 cycles	19,797,636 cycles
Speedup	5.9	1

For the decryption direction the performance results are presented in table 5.2. As depicted the decryption process can be speedup up to 5.9 times. From the results presented in table 5.1 and in table 5.2, one can see that the MOLEN architecture speeds up the decryption process up to 5.9 times.

Using equation 5.2, the throughput performances are calculated. In Figure 5.3 the cipher throughput performance is shown for both the MOLEN AES as well as for the reference code. As illustrated the throughput for the MOLEN AES is about 69 Mbit/s

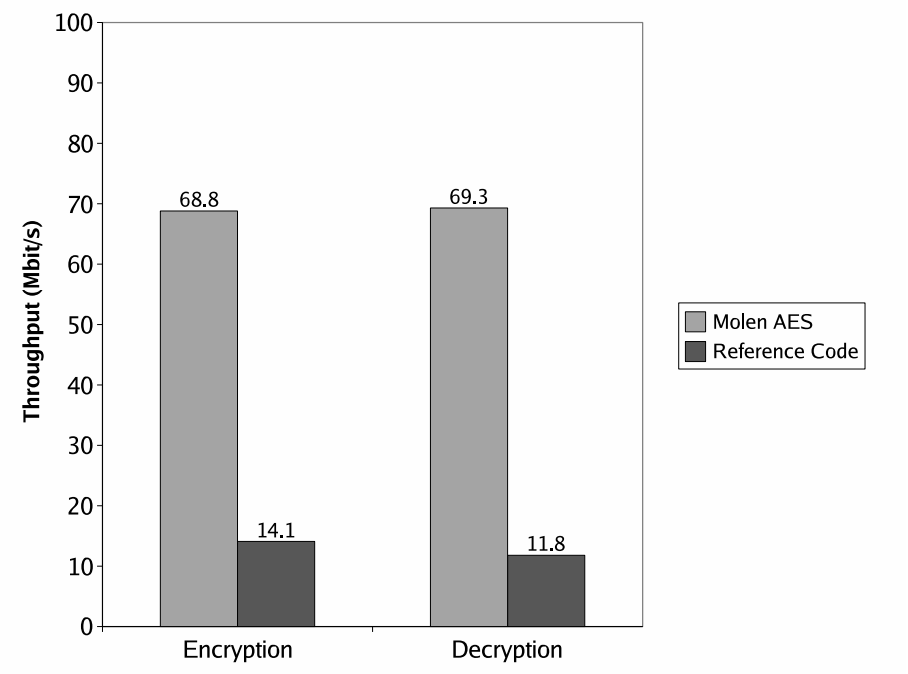


Figure 5.3: The throughput performances on the MOLEN platform

for both cipher directions. The throughput of the reference code are 14.1 Mbit/s and 11.8 Mbit/s for respectively the encryption and the decryption process.

In chapter 2, various AES implementation were overviewed. As depicted in 5.1 and 5.2, the throughputs of the AES that are implemented using higher programing language, varies from 4.60 Mbit/s to 42.6 Mbit/s. One can note, that the speed performance of some of these implementations are relatively high that compared to our 14.1 Mbit/s of our reference AES implementation. The fact is that most of these implementations were optimized by utilizing large precalculated lookup tables and that for some implementation even the MMX instructions were utilized.

5.2.1 The MOLEN AES performance

In order to analyze the MOLEN AES performance, several questions were formulated as thesis objective. The first question was formulated as: *How much is the gain in speed performance, with respect to pure software based implementations?* The simulation results show that the the AES cipher is speedup 4.9 times and 5.9 times for respectively the encryption and the decryption process. Therefore, one can state that the gain in speed performance is 4.9.

The second question was related to size of the AES hardware engines and was formulated as: *What is the minimum number of logical cells of the reconfigurable hardware unit that is required by this solution?* The AES hardware engines were synthesized for

the Altera Apex EP20KE FPGA and the Xilinx Virtex II pro 2VP20 FPGA device. The synthesis results, that were presented in chapter 3, show that "the minimum number of logical cells that is required", depends on the targeted FPGA device. Therefore, the statement, "the minimum number of logical cells that is required", can not be generalized. For our solution, the Xilinx Virtex II pro 2VP20 FPGA device was used as reference. The synthesis results show that there are 144 Configurable Logic Blocks (CLB) utilized for the encryption process and that there are 328 CLBs utilized for the decryption process. Futhermore, the results also show that only 3.53% of the area of the Xilinx device is utilized. Therefore, a smaller FPGA device from Virtex II pro family can be utilized, what will reduce costs.

The third question was formulated as: *What is the reconfiguration time of the platform?* Since reconfiguration time of a FPGA device, depends on the size and structure of the FPGA device, this statement can also not be generalized. For our solution, one can see from the datasheet that the reconfiguration time of the Virtex device is 164.29 ms. Since it is assumed that the processor will be operate at 1 GHz, the MOLEN platform will be idle for 164,290,000 cycles for the reconfiguration process.

The final question was formulated as : *For what applications could this solution be of interest?* As illustrated in figure 5.3, the throughput for the MOLEN AES is about 69 Mbit/s for both cipher directions. For high speed networks, that require throughputs of several Gbit/s, the MOLEN AES is not suitable. In networks where a throughput of a few Mbit/s is sufficient, the MOLEN AES will provide good perspectives. Besides the gain in speed performance, the MOLEN AES provides good flexibility, since reconfiguration is done run-time.

5.3 Chapter summary

In this chapter the simulation results of the MOLEN AES were presented. Since the fact that the simulation results are dependent of the platform environment, a tolerance factor was introduced. Futhermore, it was discussed how the effect of overhead was filtered out. The MOLEN AES software was compiled using the SimpleScalar's compiler and a 29.276 bytes picture was used as test-vector. The result showed that the AES cipher can be speeded up 4.9 times using the MOLEN architecture. Futhermore, in this chapter the performance results were discussed and it was concluded that the MOLEN AES will provide good perspectives.

Conclusion and Future Research Directions

6

The choice for a certain cipher within a communication process, depends on several factors such as cipher speed performance, company policies on encryption strength and government restrictions on encryption export. Considering these facts and the fact that cryptographic algorithms are relative frequently upgraded, cryptographic flexibility and high speed performance are requirements in network systems. The $\rho\mu$-coded processor, also known as the MOLEN processor[16], provides flexibility and high speed performance for various applications. The flexibility of this processor relies in the fact that system reconfiguration is done run-time, while the gain in speed performance is based on executing time-critical operations in hardware. In this thesis, the 128-bit version of the Advanced Encryption Standard [7] cipher was implemented for the MOLEN processor. The main goal of this thesis was to analyze the speed performance of the AES cipher that was targeted on the MOLEN processor.

In order to analyze the speed performance, the AES reference ANSI C source code v2.2 was profiled on time critical operations. Afterwards, the time critical operation were implemented in hardware and synthesized for both the Altera Apex EP20K1000 and Xilinx Virtex II pro 2VP20 FPGA devices. The Xilinx device was used as the reference for the simulation process. The reference ANSI C source code was modified for the SET and EXECUTE MOLEN instructions, afterwards, compiled by using the SimpleScalar compiler [9] and finally, simulated. The simulator that was utilized, is based on SimpleScalar v3.0c sim-outorder tool [9], which was extended for the MOLEN architecture. More precisely, an extra module and an application interface was build in order to simulate various application running on the MOLEN processor. For the simulation process, it was assumed that the MOLEN processor operates at 1 GHz.

The following questions were used for analyzing the modified AES running on the MOLEN processor:

- How much is the gain in speed performance, with respect to pure software based implementations?
- What is the minimum number of logical cells of the reconfigurable hardware unit that is required by this solution?
- What is the reconfiguration time of the platform?
- For what applications could this solution be of interest?

The answer of the first question is obtained by the simulation results. The simulation results showed that using the MOLEN processor, the AES cipher can be substantially speeded up. For the decryption ciphering process a speedup of 5.9 times was calculated, while for the encryption ciphering process a speedup of 4.9 times was calculated. Therefore, one can state that the gain in speed performance is 4.9 times.

The answer of the second question is obtained by the synthesis results. The AES hardware engines were synthesized for the Altera Apex EP20KE FPGA and the Xilinx Virtex II pro 2VP20 FPGA device. The synthesis results showed that "the minimum number of logical cells that is required", depends on the targeted FPGA device. Therefore, the statement, "the minimum number of logical cells that is required", can not be generalized. For our solution, the Xilinx Virtex II pro 2VP20 FPGA device was used as reference. The synthesis results show that there are 144 Configurable Logic Blocks (CLB) utilized for the encryption process and that there are 328 CLBs utilized for the decryption process. Futhermore, the results also show that only 3.53% of the area of the Xilinx device is utilized. Therefore, a smaller FPGA device from Virtex II pro family can be used, what will reduce costs.

The answer of the third question is obtained by the data sheet of the Xilinx Virtex II pro 2VP20 FPGA device. One can see from the datasheet that the reconfiguration time of the Virtex device is 164.29 ms. Since it was assumed that the processor will be operate at 1 GHz, the MOLEN platform will be idle for 164,290,000 cycles for the reconfiguration process. Note that, for our simulation process a worst case scenario was assumed.

The answer of the final question is obtained by calculating the throughputs of the MOLEN AES. For the encryption process the throughput is 68.8 Mbit/s, while for the decryption process the throughput is 69.3 Mbit/s. One can conclude that, the MOLEN AES is not suitable for high speed networks, since often throughputs of several Gbit/s is required in these networks. However, utilizing AES code that is targeted for high speed networks, it is expected that it will result in high speed performance gain, in case that is targeted for the MOLEN platform. In networks where a throughput of a few Mbit/s is sufficient, our MOLEN AES will provide good perspectives. Besides the gain in speed performance, the MOLEN AES provides good level of flexibility, since reconfiguration is done on line.

Future Research Directions

This thesis was focused on analyzing the speed performance of the AES cipher that was targeted for the MOLEN processor. However various aspects, such as the security aspect related to the MOLEN processor were not addressed in this thesis. These aspects can be addressed for future work. Here are some suggestions for future work:

- One aspect of the MOLEN AES, that was not addressed in this thesis, is the security aspect of the MOLEN processor. Since the configuration of the CCU can be done on line, the content of the microcode can be tampered. However methods, such as utilizing the reconfiguration microcode into the read-only memory, (e.g. EPROM) will increase the security level of the system;

- In order to utilize all eight MOLEN instructions, the simulator that was presented here must be extended . Also note that for the utilizing the XREGS, both the SimpleScalar compiler and the MOLEN module must be modified.

- In order to utilize the flexibility aspect of the MOLEN processor in cryptographical network systems, various ciphers have to be targeted for the MOLEN processor.

Bibliography

[1] P.C. Van Oorschot A.J. Menezes and S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, Waterloo, Ontario, Canada, 2001.

[2] K. Aoki1 and H. Lipmaa, *Fast implementations of aes candidates*, Third AES Candidate Conference (New York City, USA), April 2000.

[3] R. Berdin, *Very fast 8x8 multiply result in 8 bits no carry*, http://www.korlanda.com/reggie/snippets.asp?ID=72.

[4] A.J. Elbirt W. Yip B. Chetwynd and C. Paar, *An fpga implementation and performance evaluation of the aes block cipher candidate algorithm finalists*, Third AES Candidate Conference (AES3), April 2000.

[5] N. Courtois and J. Pieprzyk, *Cryptanalysis of block ciphers with overdefined systems of equations*, Asiacrypt 2002, December 2002.

[6] J. Daemen and V. Rijmen, *The block cipher rijndael*, http://www.esat.kuleuven.ac.be/ rijmen/rijndael/.

[7] ———, *The design of rijndael*, Springer-Verlag, Leuven, Belgie, 2002.

[8] M. Dworkin, *"mips technologies" nist special publication 800-38a, december 2001*, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[9] T.M. Austin E. Larson and D. Erns, *Simplescalar llc*, http://www.simplescalar.com.

[10] N. Jacobson, *Theory of fields & galois theory*, Springer-Verlag, USA, 1964.

[11] G.K. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, *Loading ρμ-code: Design considerations*, Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03), July 2003, pp. 8–11.

[12] J. Patarin N. Courtois, A. Klimov and A. Shamir:, *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*, Eurocrypt 2000, May 2000.

[13] NIST, *Advanced encryption standard, questions and answers*, http://csrc.nist.gov/CryptoToolkit/aes/aesfact.html.

[14] ———, *National institute of standards and technology*, www.nist.gov.

[15] Institute of Electrical and Electronics Engineers, *Ieee standard computer dictionary: A compilation of ieee standard computer glossaries*, New York, 1990.

[16] S. Vassiliadis S.Cotofana and S. Wong, *The molen ρμ-coded processor*, The 11th International Conference on Field Programmable Logic and Application (FPL), Augustus 2001.

[17] T. KASUYA T. ICHIKAWA and M. MATSUI, *Hardware evaluation of the aes finalists*, Third AES Candidate Conference (AES3), April 2000.

[18] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G.K. Kuzmanov, and E. Moscu Panainte, *The molen polymorphic processor*, February 2005, p. to appear.

[19] J.S.S.M. WONG, *Microcoded reconfigurable embedded processors*, Delft, The Netherlands, December 2002.

[20] XILINX, *Virtex-ii pro platform user guide*, www.xilinx.com.

Sim-outorder extended for the MOLEN module

The following code represents code parts of sim-outorder.c, which are extended for the MOLEN module.

```
#define EXECUTE_CHANGE
#ifdef EXECUTE_CHANGE
  #include "./molen/app_config.c"
  #include "./molen/molen_config.c"
#endif
.
.
.
.

  "FP-MULT/DIV",
    1,
    0,
    {
      { FloatMULT, 4, 1 },
      { FloatDIV, 12, 12 },
      { FloatSQRT, 24, 24 }
    }
  },
#ifdef EXECUTE_CHANGE
  {
    "Custom Configured Unit",
    1,
    0,
    {
      { SetLogic, SET_OPLAT, SET_ISSUELAT },
      { Execute, EXEC_OPLAT, EXEC_ISSUELAT },
      { LoadPrt, 1, 1 } /*In order to load reconfiguration code*/
    }
  },
#endif
};
.
.
.
```

```
.
sim_reg_stats(struct stat_sdb_t *sdb)   /* stats database */
{
  int i;
#ifdef EXECUTE_CHANGE
  stat_reg_counter(sdb, "special_nops",
   "total number of special NOPs",
   &num_special_nops, 0, NULL);
  stat_reg_counter(sdb, "num_of_set_insn",
   "total of set instructions",
   &num_set_insn, 0, NULL);
  stat_reg_counter(sdb, "num_of_exe_insn",
   "total number of execute insn",
   &num_exe_insn, 0, NULL);
  stat_reg_counter(sdb, "load_port_usage",
   "no. times of loading reconf. or microcode",
   &load_port_usage, 0, NULL);
  stat_reg_counter(sdb, "set_pageable_times",
   "total number of the SET section is updated",
   &set_pageable_times, 0, NULL);
  stat_reg_counter(sdb, "exe_pageable_times",
   "total number of the EXE section is updated",
   &exe_pageable_times, 0, NULL);
  stat_reg_counter(sdb, "squashed_exe_insn",
   "total number of squashed execute insn",
   &squashed_exe_insn, 0, NULL);
  stat_reg_counter(sdb, "av_load_latency",
   "average latency (through caches) of loading one data block to the FPGA",
   &av_load_latency, 0, NULL);
  stat_reg_counter(sdb, "func1_cycle",
   "Function time in cycles",
   &func1_cycle, 0, NULL);
  stat_reg_counter(sdb, "func2_cycle",
   "Function time in cycles",
   &func2_cycle, 0, NULL);

#endif
  stat_reg_counter(sdb, "sim_num_insn",
   "total number of instructions committed",
   &sim_num_insn, sim_num_insn, NULL);
.
.
.
.
```

```
struct RUU_station {
  /* inst info */
  md_inst_t IR; /* instruction bits */
  enum md_opcode op; /* decoded instruction opcode */
  md_addr_t PC, next_PC, pred_PC; /* inst PC, next PC, predicted PC */
  int in_LSQ; /* non-zero if op is in LSQ */
  int ea_comp; /* non-zero if op is an addr comp */
  int recover_inst; /* start of mis-speculation? */
  int stack_recover_idx; /* non-speculative TOS for RSB pred */
  struct bpred_update_t dir_update; /* bpred direction update info */
  int spec_mode; /* non-zero if issued in spec_mode */
  md_addr_t addr; /* effective address for ld/st's */
  INST_TAG_TYPE tag; /* RUU slot tag, increment to
   squash operation */
  INST_SEQ_TYPE seq; /* instruction sequence, used to
   sort the ready list and tag inst */
  unsigned int ptrace_seq; /* pipetrace sequence number */
  int slip;

  #ifdef EXECUTE_CHANGE

  enum instruction_type inst_type;      /* Instruction type */
  enum code_location location;          /* Conf./Micro-code location */
  enum configuration config;            /* Needed configuration */
  int loading;                          /* To load or not to load */
  int reconfigure;                      /* To reconfigure or not the
   FPGA */
  int load_data;                        /* To load data or not? */
  int store_data;                       /* To store data or not? */
  int counter;                          /* Until next operation may
   begin */
  int reconf_address;                   /* Address of reconf. code */
  int micro_address;                    /* Address of microcode */
  int load_address;                     /* Address to which to load */
  int load_address2;
  int store_address;                    /* Address to which to store
   */
#endif

  /* instruction status */
  int queued; /* operands ready and queued */
  int issued; /* operation is/was executing */
  int completed; /* operation has completed execution */
.
.
```

```
.
.
  unsigned int ptrace_seq; /* pipetrace sequence number */
  int slip;

  #ifdef EXECUTE_CHANGE

  enum instruction_type inst_type;      /* Instruction type */
  enum code_location location;          /* Conf./Micro-code location */
  enum configuration config;            /* Needed configuration */
  int loading;                          /* To load or not to load */
  int reconfigure;                      /* To reconfigure or not the
   FPGA */
  int load_data;                        /* To load data or not? */
  int store_data;                       /* To store data or not? */
  int counter;                          /* Until next operation may
   begin */
  int reconf_address;                   /* Address of reconf. code */
  int micro_address;                    /* Address of microcode */
  int load_address;                     /* Address to which to load */
  int load_address2;
  int store_address;                    /* Address to which to store
   */
#endif

  /* instruction status */
  int queued; /* operands ready and queued */
  int issued; /* operation is/was executing */
  int completed;
.
.
.
.

/* the last operation that ruu_dispatch() attempted to dispatch, for
   implementing in-order issue */
static struct RS_link last_op = RSLINK_NULL_DATA;

#ifdef EXECUTE_CHANGE
#include "./molen/hardware.c"
#include "./molen/molen.c"
#endif
/*********************************************************************/
/*
 *  RUU_COMMIT() - instruction retirement pipeline stage
```

```
 */
/*****************************************************************/
static void ruu_commit(void)
{
  int i, lat, events, committed = 0;
  static counter_t sim_ret_insn = 0;


#ifdef EXECUTE_CHANGE
    int annotated_field = 0;
#endif
  /* all values must be retired to the architected reg file in program order */
  while (RUU_num > 0 && committed < ruu_commit_width)
    {
      struct RUU_station *rs = &(RUU[RUU_head]);

      if (!rs->completed)
{
  /* at least RUU entry must be complete */
  break;
}

      /* default commit events */
      events = 0;

 /* Added by rmas5*/
/* retire the annotated instruction */
#ifdef EXECUTE_CHANGE
     annotated_field = MD_ANNOTATION(RUU[RUU_head].IR);
int remove = 0;
      if (annotated_field != 0){

  if (commit_molen(annotated_field, rs, events)){
  break;
}

      annotated_field =0;
      }
     else{
#endif
      if (RUU[RUU_head].ea_comp)

.
.
.
```

```
.

static void ruu_writeback(void)
{
  int i;
  struct RUU_station *rs;


#ifdef EXECUTE_CHANGE
/* added by rmas5 */
 int annotated_field = 0;
#endif

  /* service all completed events */
  while ((rs = eventq_next_event()))
    {
      /* RS has completed execution and (possibly) produced a result */
      if (!OPERANDS_READY(rs) || rs->queued || !rs->issued || rs->completed)
panic("inst completed and !ready, !issued, or completed");

      /* operation has completed */
      rs->completed = TRUE;

      /* does this operation reveal a mis-predicted branch? */
      if (rs->recover_inst)
{
  if (rs->in_LSQ)
    panic("mis-predicted load or store?!?!?");

  /* recover processor state and reinit fetch to correct path */
  ruu_recover(rs - RUU);
  tracer_recover();
  bpred_recover(pred, rs->PC, rs->stack_recover_idx);

  /* stall fetch until I-fetch and I-decode recover */
  ruu_fetch_issue_delay = ruu_branch_penalty;

  /* continue writeback of the branch/control instruction */
}

      /* if we speculatively update branch-predictor, do it here */
      if (pred
  && bpred_spec_update == spec_WB
  && !rs->in_LSQ
  && (MD_OP_FLAGS(rs->op) & F_CTRL))
```

```
{
  bpred_update(pred,
       /* branch address */rs->PC,
       /* actual target address */rs->next_PC,
       /* taken? */rs->next_PC != (rs->PC +
   sizeof(md_inst_t)),
       /* pred taken? */rs->pred_PC != (rs->PC +
sizeof(md_inst_t)),
       /* correct pred? */rs->pred_PC == rs->next_PC,
       /* opcode */rs->op,
       /* dir predictor update pointer */&rs->dir_update);
}

      /* entered writeback stage, indicate in pipe trace */
      ptrace_newstage(rs->ptrace_seq, PST_WRITEBACK,
      rs->recover_inst ? PEV_MPDETECT : 0);


#ifdef EXECUTE_CHANGE
    /* Added by rmas5*/
 annotated_field = MD_ANNOTATION(rs->IR);
      if (annotated_field != 0){
        writeback_molen(annotated_field);
}
#endif
       /* broadcast results to consuming operations, this is more efficiently
         accomplished by walking the output dependency chains of the
 completed instruction */
      for (i=0; i<MAX_ODEPS; i++)
{


.
.
.
.

static void ruu_issue(void)
{
  int i, load_lat, tlb_lat, n_issued;
  struct RS_link *node, *next_node;
  struct res_template *fu;

  /* FIXME: could be a little more efficient when scanning the ready queue */
```

```
  /* copy and then blow away the ready list, NOTE: the ready list is
     always totally reclaimed each cycle, and instructions that are not
     issue are explicitly reinserted into the ready instruction queue,
     this management strategy ensures that the ready instruction queue
     is always properly sorted */
  node = ready_queue;
  ready_queue = NULL;

#ifdef EXECUTE_CHANGE
  /* Added by rmas5 */
  int annotated_field = 0;
#endif

  /* visit all ready instructions (i.e., insts whose register input
     dependencies have been satisfied, stop issue when no more instructions
     are available or issue bandwidth is exhausted */
  for (n_issued=0;
       node && n_issued < ruu_issue_width;
       node = next_node)
    {
      next_node = node->next;

      /* still valid? */
      if (RSLINK_VALID(node))
{
  struct RUU_station *rs = RSLINK_RS(node);

  /* issue operation, both reg and mem deps have been satisfied */
  if (!OPERANDS_READY(rs) || !rs->queued
      || rs->issued || rs->completed)
    panic("issued inst !ready, issued, or completed");

  /* node is now un-queued */
  rs->queued = FALSE;

#ifdef EXECUTE_CHANGE
   /**** Line added by rmas5 ******/
   /* Get the annotated field of the instruction. */
         annotated_field = MD_ANNOTATION(rs->IR);
#endif

  if (rs->in_LSQ
      && ((MD_OP_FLAGS(rs->op) & (F_MEM|F_STORE)) == (F_MEM|F_STORE)))
    {
      /* stores complete in effectively zero time, result is
```

```
 written into the load/store queue, the actual store into
 the memory system occurs when the instruction is retired
 (see ruu_commit()) */
      rs->issued = TRUE;
      rs->completed = TRUE;
      if (rs->onames[0] || rs->onames[1])
panic("store creates result");

      if (rs->recover_inst)
panic("mis-predicted store");

      /* entered execute stage, indicate in pipe trace */
      ptrace_newstage(rs->ptrace_seq, PST_WRITEBACK, 0);

      /* one more inst issued */
      n_issued++;
    }
  else
    {
      /* issue the instruction to a functional unit */
      if (MD_OP_FUCLASS(rs->op) != NA)
{
#ifdef EXECUTE_CHANGE
                /********* Added by rmas5 ********/
if (annotated_field) {
              issue_molen(annotated_field, rs, n_issued);
annotated_field = 0;
}
                else {
#endif
.
.
.
.


#ifdef EXECUTE_CHANGE
   /* Line added by rmas5 */
   int annotated_field = 0;
#endif

  made_check = FALSE;
  n_dispatched = 0;
  while (/* instruction decode B/W left? */
    n_dispatched < (ruu_decode_width * fetch_speed)
```

```
 /* RUU and LSQ not full? */
 && RUU_num < RUU_size && LSQ_num < LSQ_size
 /* insts still available from fetch unit? */
 && fetch_num != 0
 /* on an acceptable trace path */
 && (ruu_include_spec || !spec_mode)
#ifdef EXECUTE_CHANGE
 && stop_dispatch == 0
#endif
 )


    {
      /* if issuing in-order, block until last op issues if inorder issue */
      if (ruu_inorder_issue
  && (last_op.rs && RSLINK_VALID(&last_op)
      && !OPERANDS_READY(last_op.rs)))
{
  /* stall until last operation is ready to issue */
  break;
}

      /* get the next instruction from the IFETCH -> DISPATCH queue */
      inst = fetch_data[fetch_head].IR;
      regs.regs_PC = fetch_data[fetch_head].regs_PC;
      pred_PC = fetch_data[fetch_head].pred_PC;
      dir_update_ptr = &(fetch_data[fetch_head].dir_update);
      stack_recover_idx = fetch_data[fetch_head].stack_recover_idx;
      pseq = fetch_data[fetch_head].ptrace_seq;

      /* decode the inst */
      MD_SET_OPCODE(op, inst);

#ifdef EXECUTE_CHANGE
       /* line added by rmas5*/
      annotated_field = MD_ANNOTATION(inst);
      /*printf("annotated_field = %d\n",annotated_field);*/
#endif

      /* compute default next PC */
      regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);

      /* drain RUU for TRAPs and system calls */
      if (MD_OP_FLAGS(op) & F_TRAP)
{
```

```
  if (RUU_num != 0)
    break;

  /* else, syscall is only instruction in the machine, at this
     point we should not be in (mis-)speculative mode */
  if (spec_mode)
    panic("drained and speculative");
}

      /* maintain $r0 semantics (in spec and non-spec space) */
      regs.regs_R[MD_REG_ZERO] = 0; spec_regs_R[MD_REG_ZERO] = 0;
#ifdef TARGET_ALPHA
      regs.regs_F.d[MD_REG_ZERO] = 0.0; spec_regs_F.d[MD_REG_ZERO] = 0.0;
#endif /* TARGET_ALPHA */

      if (!spec_mode)
{
  /* one more non-speculative instruction executed */
  sim_num_insn++;
}

      /* default effective address (none) and access */
      addr = 0; is_write = FALSE;

      /* set default fault - none */
      fault = md_fault_none;

#ifdef EXECUTE_CHANGE
if (annotated_field)

{
dispatch_molen(op,
inst,
pseq,
dir_update_ptr,
rs,
stack_recover_idx,
annotated_field);


/* Execption made for Trace operation counter*/
if ((annotated_field == 1) || (annotated_field == 6) )
{
trace_routine(annotated_field, sim_cycle);
op =  MD_NOP_OP;
```

```
rs = NULL;
}

annotated_field = 0;
       }


       else {

#endif
```

The MOLEN module

B

The following code represents the MOLEN module.

```
 /* Molen.c, extension for the sim-outorder of simplescalar v3.0c
 *
 *  The reserved bit annotations:
 *
 *  - annotated_field = 2 is the SET instruction (resident)(ADD/b);
 *  - annotated_field = 3 is the SET instruction (pageable)(ADD/a/b);
 *  - annotated_field = 4 is the EXECUTE instruction (resident)(ADD/c);
 *  - annotated_field = 5 is the EXECUTE instruction (pageable)(ADD/a/c).
 *
 *
 */


/***********************/
/* microcode_present */
/***********************/
/* The following function checks if the microcode is present in the
fixed or pageable part of the rho-Control Store. */


int microcode_present(enum instruction_type type, /* conf./micro-code */
       enum code_location loc,  /* location of code */
       int memory_address,      /* memory address of
   code */
       enum configuration conf) /* this configuration?
 */

{
  int *p;
  int size;
  int i;
  int test_address = memory_address;
```

```
  if (type == NOTYPE)
    return FALSE;
  else {
    if (type == SET) {
      if (loc == UNDETERMINED)
return FALSE;
      else {
if (loc == RESIDENT){
  size = NO_FIXED_CONFIG_CODE;
  p = &set_fixed[0][0];
  test_address = 0;
}
if (loc == PAGEABLE){
  size = NO_PAGEABLE_CONFIG_CODE;
  p = &set_pageable[0][0];
}
      }
    }
    else {
      if (loc == UNDETERMINED)
return FALSE;
      else {
if (loc == RESIDENT){
  size = NO_FIXED_MICRO_CODE;
  p = &exec_fixed[0][0];
  test_address = 0;
}
if (loc == PAGEABLE){
  size = NO_PAGEABLE_MICRO_CODE;
  p = &exec_pageable[0][0];
}
      }
    }
  }

  if (p) {
    for (i = 0; i < size; i++) {
      if ((p[i] == conf) && (p[(i + size)] == test_address))
return TRUE;
    }
  }

  return FALSE;
}
```

```
/****************/
/* insert_code */
/****************/
/* This function loads the reconfiguration code or microcode in the
   pageable part of the set- or execute-sections, repectively, of the
   rho-Control Store. */
int insert_code(enum instruction_type type, /* conf./micro-code */
/*enum code_location loc,*/     /* location of code */
int memory_address,         /* memory address of code */
enum configuration conf)    /* this configuration?
     */
{
  int size;
  int i;
  int *p;
  int random_location;

 /* if (type == NOTYPE) {
    printf ("Type : %d, Location : %d, Config : %d\n", type, loc, conf);
    fatal ("Undetermined instruction type detected!!");
  }

  if (loc == UNDETERMINED)
    fatal ("Undetermined location detected!!");

  if (loc == RESIDENT)
    fatal ("Cannot write to RESIDENT part!!");
*/

  if (type == SET) {
    size = NO_PAGEABLE_CONFIG_CODE;
    p = &set_pageable[0][0];
  }

  if (type == EXECUTE) {
    size = NO_PAGEABLE_MICRO_CODE;
    p = &exec_pageable[0][0];
  }

  /* Insert here some replacement method. */
  /* In this case, if full, then random. */

  /* Determine if exists */
  for (i = 0; i < size; i++)
```

```
    if ((p[i] == conf) && (p[(i + size)] == memory_address))
      return 0;

  /* It did not exist, thus insert. */

#ifdef TESTING
  printf ("Type: %d, ", type);
  printf ("Location: %d, ", loc);
  printf ("Memory address: %d, ", memory_address);
  printf ("Configuration: %d ;\n", conf);


  /* Find empty spot. */
  for (i = 0; i < size; i++)
    if (p[(i*2)+0] == NOTHING){
      p[i] = conf;
      p[(i + size)] = memory_address;

      if (type == SET)
set_pageable_times++;
      else
exe_pageable_times++;

      return 0;
    }

  /* No empty spot found. */
  /* Implementing a random replacement methodology. */
  random_location = random()%size;
  p[(random_location*2)] = conf;
  p[((random_location*2) + size)] = memory_address;

  if (type == SET)
    set_pageable_times++;
  else
    exe_pageable_times++;

  return 0;
}


/***********************/
/* check_CCU */
/***********************/
/*  This function determines whether the CCU is
```

```
   configured to a certain configuration. */
int check_CCU(enum configuration config)
{
  int size = no_configurations;
  int i;
  int *p;
  p = &CCU_config_status[0];

  for (i = 0; i < size; i++) {
    if (p[i] == config)
      return TRUE;
  }

  return FALSE;
}

/****************/
/* reconfigure_CCU */
/****************/
/* This function updates the CCU telling it that it has been
   reconfigured to perform a certain operation. */

int reconfigure_CCU(enum configuration conf)
{
  int size;
  int i;
  int *p;

  size = no_configurations;
  p = &CCU_config_status[0];

  /* Determine if already exists */
  for (i = 0; i < size; i++)
    if (p[i] == conf)
return 0;

  /* Find empty spot. */
  for (i = 0; i < size; i++)
    if (p[i] == NOCONFIG)
      p[i] = conf;

  /* No empty spot found. */
  /* Implementing a random replacement methodology */
  p[random()%size] = conf;
  return 0;
```

```
}


/* The following function traces the simulation clock cycle */
/* Use annotation "/a" and "/a/c"*/

void trace_routine(int annotated_field)
{
/* Trace the amount of clock cycles during an operation*/
if (annotated_field == 1){



if (sim_cycle_save1 == 0) {
sim_cycle_save1 = sim_cycle;
//printf("* start function1 at cycle : %d \n", sim_cycle);
}
else {
func1_cycle = func1_cycle + (sim_cycle - sim_cycle_save1);
// printf("* stop function1 at cycle : %d \n", sim_cycle);
//printf(" function 1 cycles is : %d \n\n", func1_cycle);
sim_cycle_save1 = 0;
}


}

/* Trace operation counter*/
if ( annotated_field == 6){



if (sim_cycle_save2 == 0) {
sim_cycle_save2 = sim_cycle;
//printf("* start function1 at cycle : %d \n", sim_cycle);
}
else {
func2_cycle = func2_cycle + (sim_cycle - sim_cycle_save2);
//printf("* stop function1 at cycle : %d \n", sim_cycle);
//printf(" function 1 cycles is : %d \n\n", func1_cycle);
sim_cycle_save2 = 0;
}


}
```

```
}


/******************/
/* dispatch_molen */
/******************/
void dispatch_molen(enum md_opcode op,
                    md_inst_t inst,
                    unsigned int pseq,
    struct bpred_update_t *dir_update_ptr,
                    struct RUU_station *rs,
                    int stack_recover_idx,
                    int annotated_field)
{        enum instruction_type inst_type = NOTYPE;

        md_addr_t addr; /* effective address, if load/store */
       byte_t temp_byte = 0; /* temp variable for spec mem access */
        half_t temp_half = 0; /* " ditto " */
    word_t temp_word = 0; /* " ditto " */
        int out1, out2, in1, in2, in3; /* output/input register names */
        enum configuration config = NOCONFIG;
        int loading = FALSE;
        int reconfigure = FALSE;              /* To reconfigure or not
                                                 reconfigure the FPGA. */
        int load_data = FALSE;                 /* Need to load data? */
        int store_data = FALSE;                /* Need to store data? */
        int reconf_address = 0;                /* Address of reconf. */
        int micro_address = 0;                 /* Address of microcode */
        int load_address = 0;                  /* Load data at address */
        int store_address = 0;                 /* Store data at address */
        int load_address2 = 0;
        int conf_position = 0;



  int field1_content = 0;
  int field2_content = 0;
  int field3_content = 0;

  enum md_fault_type fault;
  enum code_location location = UNDETERMINED;
```

```
/* Trace operation counter*/
if ((annotated_field == 1) || (annotated_field == 6) )
{
trace_routine(annotated_field);
op =  MD_NOP_OP;
rs = NULL;
annotated_field =0;
}

if (annotated_field !=0 ){

 if ((annotated_field == 2) || (annotated_field == 3)  ||
     (annotated_field == 5) || (annotated_field == 7))
{ /* SET instruction */
      /* == 2     --> resident */
      /* == 3     --> pageable */

      num_set_insn++;   /* To track how many times
                           set instruction is detected */

      /* Not setting any dependencies with other instructions
 through registers. */
      out1 = NA; out2 = NA;
      in1 = NA; in2 = NA; in3 = NA;

      /* Storing the contents of the instruction fields. */
      field1_content = GPR(RD);
      field2_content = GPR(RS);
      field3_content = GPR(RT);

      /* This is the address where the reconfiguration
 microcode is located. */
      reconf_address = field2_content;

      /* Determining the exact info on this instruction. */
      inst_type = SET;
      location = LOCATION;
      /* Now: everytime a SET is issued, the configuration is
 determined from looking in the memory. However, it
 should be that the memory address is noted and that
 one is checked. Due to time constraints this was not
 changed anymore since it did not affect the results
 of the simulation as being inside the simulator. If
 you want to change this to better reflect the MOLEN
```

```
 architecture then trace the variable 'config'. */
      config = READ_WORD(field2_content,fault);
      /* printf ("field2_content = %d\n", field2_content); */

      /* Check whether the configuration is known. */
      if (config > KNOWN_CONFIGURATIONS)
fatal ("The SET instruction specified an unknown configuration method");

      /* Check whether the FPGA is already configured for the
 required configuration. */
      if (check_CCU(config) == FALSE) {
/* Reconfiguration of the FPGA is required. */
reconfigure = TRUE;

/* Determine the location of the reconfiguration code. */
if ((location == UNDETERMINED) || (location == PAGEABLE))
  { /* Reconfiguration code must be loaded from memory. */

    /* Determine whether is has been loaded before in
       the pageable part. */
    if (microcode_present(inst_type, location,
   reconf_address, config)
== TRUE) {
      /* Configuration code is still present on-chip */
      /* This is the easiest, because now we do not have
 to consider the loading time of reconfiguration
 code. */
      loading = FALSE;
    }
    else {
      /* Configuration code is not present on-chip
 anymore. */
      /* Initiate somehow the loading of the
 reconfiguration code. */
      loading = TRUE;
      location = PAGEABLE;
    }
  }
else
  { /* Reconfiguration code is on-chip. */

    /* Determine if this is true. */
    /* For now, we do not support that when code that is
       specified to be on-chip is not to be loaded in
       the pageable part. */
```

```
    if (microcode_present(inst_type, location, 0, config)
== FALSE)
      fatal ("Reconfiguration code was NOT on-chip!!");
  }
      }
      else {
/* The FPGA is still reconfigured to the correct
   implementation. */
loading = FALSE;
reconfigure = FALSE;
      }
 }/************end of SET annotation**********************/

 if ((annotated_field == 4) || (annotated_field == 5) )
    { /* EXECUTE instruction */
      /* == 4     --> resident */
      /* == 5     --> pageable */

      num_exe_insn++;   /* To track how many times
                           execute instruction is detected */



      /* Not setting any dependencies with other instructions
 through registers. */
      out1 = NA; out2 = NA;
      in1 = NA; in2 = NA; in3 = NA;

      /* Storing the contents of the instruction fields.
        This is done through the GPRs.*/
      field1_content = GPR(RD);
      field2_content = GPR(RS);
      field3_content = GPR(RT);

      /* Tconf_positionhis is the address where the microprogram is
 located. */
      micro_address = field2_content;


      /* Determining the exact info on this instruction. */
      inst_type = EXECUTE;
      location = LOCATION;
      /* Now: everytime an EXECUTE instruction is issued, the
 memory is checked to determine the correct
 microcode. However, it should be that only the
```

```
 address is checked. Due to time constraints this was
 changed since it did not affect the simulation
 results as it is inside the simulator. If you do want
 to change this to better reflect the MOLEN
 architecture, then you probably have to trace the
 variable config. */
      config = READ_WORD(micro_address,fault);
      conf_position = READ_WORD(micro_address,fault);

#if 0
      printf ("----------------------> %d, ", micro_address);
      printf ("  %d  \n", config);


      /* Check whether the configuration is known. */
      /* In real life situations this will create an
 exception, but for the simulators sake we did this
 check, otherwise the simulator will crash. */
      if ((config > KNOWN_CONFIGURATIONS) ||
  (config <= NOTHING)) {
#if 1
printf ("Config  ===> %d \n", config);
printf ("FPGA    ===> %d \n", CCU_config_status[0]);
printf ("Address ===> %d \n", micro_address);

fatal ("The EXECUTE instruction specified an unknown configuration!!");
      }

      /* Check whether the FPGA is already reconfigured to the
 correct implementation. If not, then check whether
 the SET instruction has been issued or not --OR-- the
 distance between SET and EXECUTE is large enough. */
      if (check_CCU(config) == FALSE){
#if 1
printf ("Config ===> %d \n", config);
printf ("FPGA   ===> %d \n", CCU_config_status[0]);

fatal ("The FPGA was not correctly reconfigured! (CHECK SET exists or distance large
      }
      /* Determine the location of the microcode. */
      if ((location == UNDETERMINED) || (location == PAGEABLE))
{ /* Microcode must be loaded from memory. */

  /* Determine whether is has been loaded before in
     the pageable part. */
```

```
  if (microcode_present(inst_type, location,
 micro_address, config)
      == TRUE) {
    /* Configuration code is still present on-chip */
    /* This is the easiest, because now we do not have
       to consider the loading time of reconfiguration
       code. */
    loading = FALSE;
  }
  else {
    /* Configuration code is not present on-chip
       anymore. */
    /* Initiate somehow the loading of the
       reconfiguration code. */
    loading = TRUE;
    location = PAGEABLE;
  }
}
      else
{ /* Microcode is on-chip. */

  /* Determine if this is true. */
  /* For now, we do not support that when code that is
     specified to be on-chip is not to be loaded in
     the pageable part. */
  if (microcode_present(inst_type, location, 0, config) == FALSE)
    fatal ("Microcode was NOT on-chip!!");
}

      /* Perform the actual execution of the needed
 operation. */
      /* This is done by checking which operation is needed
 and then performing it. */

      /* If there is a data_load_latency & a data_store_latency:
        load_data & store_data are set to TRUE.
        */

if (data_load_size[config] != 0)
                load_data = TRUE;

                if (data_store_size[config] != 0)
                store_data = TRUE;

load_address = field3_content;
```

```
store_address = field3_content;
load_address2 = field1_content;

/*hardware_execution(load_address2, micro_address, load_address);
*/


}/* end of execute annotation*/




  /* fill in RUU reservation station */
   rs = &RUU[RUU_tail];
                rs->slip = sim_cycle - 1;
   rs->IR = inst;
   rs->op = op;
  rs->PC = regs.regs_PC;
   rs->next_PC = regs.regs_NPC; rs->pred_PC = pred_PC;
   rs->in_LSQ = FALSE;
   rs->ea_comp = FALSE;
   rs->recover_inst = FALSE;
                rs->dir_update = *dir_update_ptr;
   rs->stack_recover_idx = stack_recover_idx;
   rs->spec_mode = spec_mode;
   rs->addr = 0;
   /* rs->tag is already set */
    rs->seq = ++inst_seq;
   rs->queued = rs->issued = rs->completed = FALSE;
   rs->ptrace_seq = pseq;

                rs->inst_type = inst_type;
     rs->location = location;
     rs->config = config;
     rs->loading = loading;
     rs->reconfigure = reconfigure;
     rs->load_data = load_data;
     rs->store_data = store_data;
     rs->reconf_address = reconf_address;
     rs->micro_address = micro_address;
     rs->load_address = load_address;
     rs->load_address2 = load_address2;
     rs->store_address = store_address;
     rs->counter = 0;
```

```
       /* link onto producing operation */
    ruu_link_idep(rs, /* idep_ready[] index */0, in1);
      ruu_link_idep(rs, /* idep_ready[] index */1, in2);
      ruu_link_idep(rs, /* idep_ready[] index */2, in3);

      /* install output after inputs to prevent self reference */
      ruu_install_odep(rs, /* odep_list[] index */0, out1);
      ruu_install_odep(rs, /* odep_list[] index */1, out2);

       /* install operation in the RUU */

       RUU_tail = (RUU_tail + 1) % RUU_size;
       RUU_num++;


readyq_enqueue(rs);
   /* issue may continue */
last_op = RSLINK_NULL;




stop_dispatch = 1;
//ruu_fetch_issue_delay++;

}







} /* end of dispatch_molen()*/


/***************/
/* issue_molen */
/***************/


void issue_molen(int annotated_field, struct RUU_station *rs, int n_issued)
{
  int load_lat;
```

```
  int latency = 0;
  int memory_address = 0;
  int address = 0;
  int access_address = 0;
  int real_latency = 0;
  struct res_template *fu;
  int ii;
  int data_size = 0;
  int load_steps = 0;
  int microcode_size = 0; /* Size of the microcode */
  int reconfiguration_code = 0; /* Size of the recofiguration microcode */

       /* There is no use in checking whether the
       annotated field is 1, because the nop's are
       squashed in the dispatch stage. */
    /* if (annotated_field == 1)
       {
       } */
    if ((annotated_field == 2) || (annotated_field == 3)) {
      /* SET instruction detected. */
      /* This means:
 - Load reconfiguration code;
 - Reconfigure the FPGA. */

      /* Determine whether we can start the next
 operation. */
      if (rs->counter <= 0) {
/* Determine whether memory operations are
   needed. */
if (rs->loading) {
  /* Loading of reconfiguration code is
     required. */

  /* In order to load reconfiguration code, the CCU
     must be ready to accept code and the memory unit
     must be ready to provide the code. */
  if (res_get(fu_pool, LoadPrt)) {
    /* Try to get the memory port. */
    fu = res_get(fu_pool, RdPort);
    if (fu) {
      /* Determine the loading latency. */
       reconfiguration_code = reconf_micro_size[rs->config];
latency = reconfiguration_code / rcs_buswidth;

      /* reserve the functional unit */
```

```
      if (fu->master->busy)
panic("functional unit already in use");

      /* Occupy the memory unit. */
      fu->master->busy = latency;

      /* If we want to simulate that these codes also go
 through the caches, then we have to access the
 caches at this point. */
      /* How this must be done is shown in the code of
 loading data from caches in the EXECUTE
 instruction, below. */

      /* Schedule that the current
 instruction starts the
 reconfiguration after the
 reconf. code has been loaded. */
      /* In order to parallize, we can
 lower the value of rs->counter. */
      rs->counter = latency;

      /* Now that we have scheduled the
 loading of the reconf. code, it not
 necessary anymore in the future. */

      /* Loading operation is no longer
 required. */
      rs->loading = FALSE;

      {
/* Occupy the CCU. */
fu = res_get(fu_pool, LoadPrt);

/* reserve the functional unit */
if (fu->master->busy)
  panic("functional unit already in use (T)");

/* Occupy the memory unit. */
fu->master->busy = latency;

/* Update some stats. */
load_port_usage++;
      };

      /* Put the rs back into the queue. */
```

```
      readyq_enqueue(rs);
    }
    else {
      /* It was not possible to find a
 functional unit. */
      readyq_enqueue(rs);
    }
  }
  else {
    /* CCU is still occupied. */
    readyq_enqueue(rs);
  }
}
else {
  /* Insert the reconf. code into the
     pageable SET part of the rCS. */

     if (rs->location != RESIDENT) {
    /* It concerns pageable microcode,
    therefore it is required to load
    microcode into the rho-Control Store.*/

    /* This function automatically checks whether the
       microcode is already present or not. */
    insert_code(rs->inst_type,
/*rs->location,*/
rs->reconf_address,
rs->config);
  }

  /* Try to occupy the CCU. */
  if (rs->reconfigure) {
    fu = res_get(fu_pool, SetLogic);
    if (fu) {
      /* Now the instruction is truly
 issued. */
      rs->issued = TRUE;

      /* reserve the functional unit */
      if (fu->master->busy)
panic("functional unit already in use");

      /* Determine the loading latency. */
      latency = reconf_latency[rs->config];
```

```
      /* Occupy the CCU. */
      fu->master->busy = latency;

      /* Remove the reconfigure field. */
      rs->reconfigure = FALSE;

      /* The instruction is released one cycle after
this one, because it is of no need to hold
the instruction longer in the queues as the
reconfigutation code is now already loaded
into memory. If we want to hold the
instruction longer in the queue, then put
"..._cycle + latency);". */

 /* Now reconfigure the CCU*/
       reconfigure_CCU(rs->config);

      /* schedule the event. */
      eventq_queue_event(rs, sim_cycle + 1);

      /* entered execute stage, indicate in
pipe trace */
      ptrace_newstage(rs->ptrace_seq, PST_EXECUTE,
      rs->ea_comp ? PEV_AGEN : 0);
      n_issued++;
    }
    else {
      /* No functional unit found. */
      readyq_enqueue(rs);
    }
 }
 else {
    /* Somehow no reconfiguration was
       needed. */

    rs->issued = TRUE;

    eventq_queue_event(rs, sim_cycle + 1);

    /* entered execute stage, indicate in
       pipe trace */
    ptrace_newstage(rs->ptrace_seq, PST_EXECUTE,
    rs->ea_comp ? PEV_AGEN : 0);
    n_issued++;
```

```
  }
}
      }
      else {
/* Counter was greater than zero. */
rs->counter--;

/* Put the rs back into the queue. */
readyq_enqueue(rs);
      }
    } /* *****************end execution set **************/

       if ((annotated_field == 4) || (annotated_field == 5))
      {
/* EXECUTE instruction detected. */
/* This means:
   - Load the microcode;
   - Load the data;
   - Execute the microcode;
   - Store the results. (do this in
   ruu_commit) */

if (rs->counter <= 0) {
    /* Is loading of microcode needed? */
  if (rs->loading) {
    /* Loading of microcode is required! */

    /* In order to load reconfiguration code, the CCU
       must be ready to accept code and the memory unit
       must be ready to provide the code. */
    if (res_get(fu_pool, LoadPrt)) {
      /* Try to get a memory-port. */
      fu = res_get(fu_pool, RdPort);
      if (fu) {

/* Determine the loading latency. */
microcode_size = micro_size[rs->config];
latency = microcode_size / rcs_buswidth;

/* reserve the functional unit */
if (fu->master->busy)
  panic("functional unit already in use");

/* Occupy the memory unit. */
fu->master->busy = latency;
```

```
/* Schedule that the current
   instruction starts the loading of
   data (when needed) and execution
   after the microcode has been
   loaded. */
/* In order to parallize, we can lower
   the value of rs->counter. Be careful, this
   one is TRICKIER. */
rs->counter = latency;

/* Loading is not necessary anymore in
   the future. */
rs->loading = FALSE;

{
  /* Occupy the CCU. */
  fu = res_get(fu_pool, LoadPrt);

  /* reserve the functional unit */
  if (fu->master->busy)
    panic("functional unit already in use (T)");

  /* Occupy the memory unit. */
  fu->master->busy = latency;

  /* Update some stats. */
  load_port_usage++;
};

readyq_enqueue(rs);
      }
      else {
/* No memory port found. */
readyq_enqueue(rs);
      }
    }
    else {
      /* CCU is not free. */
      readyq_enqueue(rs);
    }
  }
  else {
    /* No loading of MICROCODE is required!
     */
```

```
    /* The setting of the reconf. code into
       the pageable part is only done when
       loading is required as checked in the
       dispatch stage. */
    if (rs->location != RESIDENT)
      /* This is done here, because we now
 know that the loading of microcode
 (if necessary) must be completed. */
    /* This function automatically checks whether the
       microcode is already present or not. */
      insert_code(rs->inst_type,
  /*rs->location,*/
         rs->micro_address,
  rs->config);

    /* Determine if loading of DATA is
       required. */
    if (rs->load_data) {
      /* Loading is required. */

      if (res_get(fu_pool, Execute)) { /* check if execute unit is available */
fu = res_get(fu_pool, RdPort); /* check if read memory is available */
if (fu) {
  /* Re-occupy the memory port. Only now
     we have to determine more precisely
     the latency, because the data is
     going through the caches. */

          data_size = data_load_size[rs->config];
  load_steps = data_size /LSQ_size;
                latency  =  load_steps;


  /* reserve the functional unit */
  if (fu->master->busy)
    panic("functional unit already in use");

  /* Occupy the memory unit. */
  fu->master->busy = latency;

  /* The determination when to
     continue execution of the EXECUTE
     instruction must be determined by
     the number of cycles it really
```

```
     takes to load from the caches. */

  address = rs->load_address;
  real_latency = 0;

          /* The following lines are based on calculating the latency of reading
        (4 * load_steps) bytes from the cache */

    for (ii = 0; ii < load_steps; ii++) {
if (access_address != ((address+ii) & ~3) )
{
/* Next address line*/
access_address = (address+ii) & ~3;
load_lat = 0;
/* Latency for reading 4 bytes through caches*/
load_lat =  cache_access(cache_dl1, Read,
      access_address, NULL, 4,
      sim_cycle, NULL, NULL);

real_latency = +load_lat;

}


    }

                            rs->load_data = FALSE;

   fu = res_get(fu_pool, Execute);

/* Perform the execution*/
  hardware_execution(rs->load_address2,
        rs->micro_address,
     rs->load_address);


  /* Now the instruction is truly
     issued. */
  rs->issued = TRUE;

  /* Update statistics. */
  exec_times_issued[rs->config]++;

  /* reserve the functional unit */
  if (fu->master->busy)
```

```
    panic("functional unit already in use (S)");

  /* Determine the execution latency. */
  latency = execution_latency[rs->config];

  /* Occupy the CCU. */
  fu->master->busy = real_latency + latency;

/*printf ("Latency : %d,%d\n",real_latency, latency);
  */
  /* If we are not going to store
     anything, why should we schedule
     the writeback event after
     'latency' cycles? It could be
     done more quickly. */

  /* 'real_latency' is the memory latency that
     is needed to load the data. */



  /* schedule the event */
  if (rs->store_data)
    eventq_queue_event(rs, sim_cycle +
       real_latency +
       latency + 1);
  else
    eventq_queue_event(rs, sim_cycle +
       real_latency +
       1);

  /* entered execute stage, indicate in
     pipe trace */
  ptrace_newstage(rs->ptrace_seq, PST_EXECUTE,
  rs->ea_comp ? PEV_AGEN : 0);

  n_issued++;


}
else {
  /* No memory port found. */
  readyq_enqueue(rs);
}
      }
```

```
      else {
/* CCU is not free. */
readyq_enqueue(rs);
      }
    }
    else {
      /* No loading is required. */

      /* We now start the execution on the
 FPGA. */
      fu = res_get(fu_pool, Execute);
      if (fu) {


      /* Perform the execution*/
  hardware_execution(rs->load_address2,
        rs->micro_address,
     rs->load_address);

/* Now the instruction is truly
   issued. */
rs->issued = TRUE;

/* Update statistics. */
exec_times_issued[rs->config]++;

/* reserve the functional unit */
if (fu->master->busy)
  panic("functional unit already in use");

/* Determine the execution latency. */
latency = execution_latency[rs->config];

/* Occupy the CCU. */
fu->master->busy = latency;

/* If we are not going to store
   anything, why should we schedule
   the writeback event after
   'latency' cycles? It could be
   done more quickly. */

/* schedule the event */
if (rs->store_data)
  eventq_queue_event(rs, sim_cycle + latency);
```

```
else
  eventq_queue_event(rs, sim_cycle + 1);

/* entered execute stage, indicate in
   pipe trace */
ptrace_newstage(rs->ptrace_seq, PST_EXECUTE,
rs->ea_comp ? PEV_AGEN : 0);

n_issued++;

      }
      else {
/* No functional unit found. */
readyq_enqueue(rs);
      }
    }
  } /*  */
}/*    */
else {
  /* Counter is not zero. */
  rs->counter--;

  /* Put the rs back into the queue. */
  readyq_enqueue(rs);
}

}



//return n_issued;

}/* end of issue_molen() */


/****************/
/* Commit_molen */
/****************/
int commit_molen(int annotated_field,
                 struct RUU_station *rs,
  int events
                 )
{
  int lat, latency, ii = 0;
  int memory_address, address, access_address = 0;
```

```
  struct res_template *fu;
  int data_size;
  int store_steps;

 /* number of cycles needed to resevere the FU*/
 latency  =  store_steps;

  if ((annotated_field == 4) || (annotated_field == 5)) {
    if (rs->store_data) {

    /* Occupy the memory port */
       fu = res_get(fu_pool, WrPort);
         if (fu) {


   data_size = data_store_size[rs->config];
     store_steps = data_size /LSQ_size;


 /* number of cycles needed to resevere the FU*/
 latency  =  store_steps;

/*Now calculate the store latency of writing
data to the chaches*/

               /* reserve the functional unit */
        if (fu->master->busy)
           panic("functional unit already in use");

           /* schedule functional unit release event */
           fu->master->busy = latency;

   /* address for storing data*/
   memory_address = rs->store_address;







        /*writing 4 Bytes each step to data cache */
    for (ii = 0; ii < store_steps; ii++) {

address = memory_address + 4*ii;
```

```
   /*set access address to store address*/
 if (access_address != (address & ~3)){

   access_address = address & ~3;

   }

   /* calculate cache access latency */
   /* go to the data cache */
           if (cache_dl1)
            {
            /* commit store value to Dl1-cache */
            lat =  cache_access(cache_dl1, Write, (access_address),
                      NULL, 4, sim_cycle, NULL, NULL);

       }

                              /* all loads and stores must to access D-TLB */
            if (dtlb)
            {
             /* access the D-TLB */
            lat = cache_access(dtlb, Read, (access_address),
       NULL, 4, sim_cycle, NULL, NULL);


    }
        } /* 4 bytes are written to the cache */

          return 0; /* all data are written to the cache.
        Now, the RUU entry can be removed*/

       }
       else{

              /* no store ports left, do not remove entry */
      return  1;
       }
}
else{
   return 0; /* no dat needed to be stored. Entry can be removed */
}
    }
    else{
       return  0; /* No execute Molen instruction, entery can be removed */
```

```
      }
}
```

```
void writeback_molen(int annotated_field)
{
     if ((annotated_field == 4) || (annotated_field == 5)
     || (annotated_field == 2) || (annotated_field == 3))
    stop_dispatch = 0;


}
```

C The MOLEN AES

In the following section all the code related to the MOLEN AES are depicted.

C.1 rijndael.c

```
#include <stdio.h>
#include <stdlib.h>
#include "rijndael-alg-ref.h"
#include "boxes-ref.dat"

static word8 shifts[4][2] = {{0,0},{1,3},{2,2},{3,1}};

int microbin1[4] = {2};
static int micro_addr1 = (int) &microbin1[0];

int finished1 = 0;
static int finished1_addr = (int) &finished1;

int finished2 = 0;
static int finished2_addr = (int) &finished2;

int microbin2[4] = {3};
static int micro_addr2 = (int) &microbin2[0];

int data_addr;


int reconfbin1[228000];
int reconf_addr1 = (int) &reconfbin1[0];


int reconfbin2[276000];
int reconf_addr2 = (int) &reconfbin2[0];



word8 mul(word8 a, word8 b) {
   /* multiply two elements of GF(2^m)
```

```
    * needed for MixColumn and InvMixColumn
    */
if (a && b) return Alogtable[(Logtable[a] + Logtable[b])%255];
else return 0;
}

void KeyAddition(word8 a[4][4], word8 rk[4][4]) {
/* Exor corresponding text input and round key input bytes
 */
int i, j;

for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++) a[i][j] ^= rk[i][j];
}

void ShiftRow(word8 a[4][4], word8 d) {
/* Row 0 remains unchanged
 * The other three rows are shifted a variable amount
 */
word8 tmp[4];
int i, j;

for(i = 1; i < 4; i++) {
for(j = 0; j < 4; j++) tmp[j] = a[i][(j + shifts/*[SC]*/[i][d]) % 4];
for(j = 0; j < 4; j++) a[i][j] = tmp[j];
}
}

void Substitution(word8 a[4][4], word8 box[256]) {
/* Replace every byte of the input by the byte at that place
 * in the nonlinear S-box
 */
int i, j;

for(i = 0; i < 4; i++)
for(j = 0; j < 4; j++) a[i][j] = box[a[i][j]] ;
}

void MixColumn(word8 a[4][4]) {
        /* Mix the four bytes of every column in a linear way
 */
word8 b[4][4];
int i, j;
```

```
for(j = 0; j < 4; j++)
for(i = 0; i < 4; i++)
b[i][j] = mul(2,a[i][j])
^ mul(3,a[(i + 1) % 4][j])
^ a[(i + 2) % 4][j]
^ a[(i + 3) % 4][j];
for(i = 0; i < 4; i++)
  for(j = 0; j < 4; j++) {
    a[i][j] = b[i][j];
  }


}

void InvMixColumn(word8 a[4][4]) {
        /* Mix the four bytes of every column in a linear way
 * This is the opposite operation of Mixcolumn
 */
word8 b[4][4];
int i, j;

for(j = 0; j < 4; j++)
for(i = 0; i < 4; i++)
b[i][j] = mul(0xe,a[i][j])
^ mul(0xb,a[(i + 1) % 4][j])
^ mul(0xd,a[(i + 2) % 4][j])
^ mul(0x9,a[(i + 3) % 4][j]);
for(i = 0; i < 4; i++)
for(j = 0; j < 4; j++) a[i][j] = b[i][j];
}


int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits, word8 W[MAXROUND
/* Calculate the necessary round keys
 * The number of calculations depends on keyBits and blockBits
 */

int i, j, t, rconpointer = 0;
word8 tk[4][MAXKC];


for(j = 0; j < 4; j++)
for(i = 0; i < 4; i++)
tk[i][j] = k[i][j];
t = 0;
```

```
/* copy values into round key array */
for(j = 0; (j <4) && (t < 44); j++, t++)
for(i = 0; i < 4; i++) W[t / 4][i][t % 4] = tk[i][j];

while (t < 44) { /* while not enough round key material calculated */
/* calculate new values */
for(i = 0; i < 4; i++)
tk[i][0] ^= S[tk[(i+1)%4][3]];
tk[0][0] ^= rcon[rconpointer++];


for(j = 1; j <2; j++)
for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
for(i = 0; i < 4; i++) tk[i][2] ^= S[tk[i][1]];
for(j = 2 + 1; j <4; j++)
for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];

/* copy values into round key array */
for(j = 0; (j <4) && (t < 44); j++, t++)
for(i = 0; i < 4; i++) W[t / 4][i][t % 4] = tk[i][j];
}

return 0;
}

int rijndaelEncrypt (word8 a[4][4], int keyBits, int blockBits, word8 rk[MAXROUNDS+1][4][4])
{
/* Encryption of one block.
 */
int r,j,i;



/* begin with a key addition
 */
KeyAddition(a,rk[0]);

        /* ROUNDS-1 ordinary rounds
 */
for(r = 1; r < 10; r++) {
Substitution(a,S);
ShiftRow(a,0);
/*MixColumn(a);*/

data_addr=(int) &a[0][0];
```

```
#ifdef TEST
printf("Input Application: \n");
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_load : = %d\n",a[i][j]);
}
}
#endif

__asm__("lw $18, data_addr");
__asm__("lw $2, micro_addr1");
__asm__("lw $3, finished1_addr");
__asm__("add/a/c $3, $2, $18");

/*load_address = field3_content;
        store_address = field3_content;
        load_address2 = field1_content;
*/
#ifdef TEST
printf("Ouput Application: \n");
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_store : = %d\n",a[i][j]);
}
}

#endif
KeyAddition(a,rk[r]);
}

/* Last round is special: there is no MixColumn
 */
Substitution(a,S);
ShiftRow(a,0);
KeyAddition(a,rk[10]);

return 0;
}


/*
int rijndaelEncryptRound (word8 a[4][4], int keyBits, int blockBits,
```

```
word8 rk[MAXROUNDS+1][4][4], int rounds)

{
int r,
int BC = 4;
int ROUNDS = 10;

switch (blockBits) {
case 128: BC = 4; break;
case 192: BC = 6; break;
case 256: BC = 8; break;
default : return (-2);
}

switch (keyBits >= blockBits ? keyBits : blockBits) {
case 128: ROUNDS = 10; break;
case 192: ROUNDS = 12; break;
case 256: ROUNDS = 14; break;
default : return (-3);
}


if (rounds > 10) rounds = 10;


KeyAddition(a,rk[0]);


for(r = 1; (r <= rounds) && (r < 10); r++) {
Substitution(a,S);
ShiftRow(a,0);
MixColumn(a);
KeyAddition(a,rk[r]);
}


if (rounds == 10) {
Substitution(a,S);
ShiftRow(a,0);
KeyAddition(a,rk[10]);
}

return 0;
}
```

```
*/
int rijndaelDecrypt (word8 a[4][4], int keyBits, int blockBits, word8 rk[MAXROUNDS+1]
{

 int r;


KeyAddition(a,rk[10]);
Substitution(a,Si);
ShiftRow(a,1);

/* ROUNDS-1 ordinary rounds
 */
for(r = 9; r > 0; r--) {
KeyAddition(a,rk[r]);


/*InvMixColumn(a);*/
data_addr=(int) &a[0][0];

__asm__("lw $7, data_addr");
__asm__("lw $2, micro_addr2");
__asm__("lw $3, finished2_addr");

__asm__("add/a/c $3, $2, $7");

Substitution(a,Si);
ShiftRow(a,1);
}

/* End with the extra key addition
 */

KeyAddition(a,rk[0]);

return 0;
}

/*
int rijndaelDecryptRound (word8 a[4][4], int keyBits, int blockBits,
word8 rk[MAXROUNDS+1][4][4], int rounds)

{
int r, BC, ROUNDS;
```

```
switch (blockBits) {
case 128: BC = 4; break;
case 192: BC = 6; break;
case 256: BC = 8; break;
default : return (-2);
}

switch (keyBits >= blockBits ? keyBits : blockBits) {
case 128: ROUNDS = 10; break;
case 192: ROUNDS = 12; break;
case 256: ROUNDS = 14; break;
default : return (-3);
}


if (rounds > 10) rounds = 10;


KeyAddition(a,rk[10]);
Substitution(a,Si);
ShiftRow(a,1);


for(r = 10-1; r > rounds; r--) {
KeyAddition(a,rk[r]);
InvMixColumn(a);
Substitution(a,Si);
ShiftRow(a,1);
}

if (rounds == 0) {

KeyAddition(a,rk[0]);
}

return 0;
}

static void usage() {
printf("\n * Encrypts & decrypts data files.\n");
  printf(" * Only 16 bytes keys are supported.\n");
  printf( "\n rijn X [key] [file] [output] \n");
  printf( " X:     -e for encyrption mode \n        -d for decryption mode\n");

}
```

```
static void proc (const char *key,
                  const char *data,
  const char *output,
  const char *dir)
{
        /* lets assume we this API only support 16 bytes in Hex values*/

        word8 d[4][MAXBC];
word8 k[4][MAXKC];
        FILE *fdata, *fkey, *fp;
word8 bufKey[16];
word8 block[16];
        word8 out[16];
int i, j, r, h,t = 0;
        word8 W[11][4][MAXBC];
word8 direction;
        int fileLength;
int size;




      if (strcmp ( dir, "-e" ) == 0 ){
      direction = 0;
    }
    else if ( strcmp (dir, "-d" ) == 0){
      direction = 1;
    }
    else { usage();
    exit(1);
    }

   /* key processing*/
    fkey = fopen(key, "r" );


    fread(bufKey, 1, 16, fkey);


    for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
```

```
 k[j][i]=bufKey[i+j*4];
}
    }

    __asm__("NOP/b/c");
    r = rijndaelKeySched (k, (int) 128, (int) 128, W);
   __asm__("NOP/b/c");

     /*for(j = 0; t < 44; t++)
{
printf("Key:\n");
for(j = 0; j < 4; j++){
for(i = 0; i < 4; i++){
printf("%d",W[t][j][i]);
}
}

printf("\n");
}
*/




     fdata = fopen(data, "r" );
     fp = fopen (output, "w");
     fflush (fp);









  if (direction == 0){


/* load reconfiguration data */
reconfbin1[0] = 2; /*MIX*/
for (i = 1; i < 228000; i++)
{
reconfbin1[i] = 0;
}

__asm__("lw $8, reconf_addr1");
```

```
        __asm__("add/a/b $8, $8, $8"); /* Set intstruction*/

/* this routine is for encrypting bigger file sizes than 12 bits*/
        fseek(fdata, 0L, SEEK_END);
fileLength  = ftell( fdata );
rewind(fdata);

h= (fileLength - ((int)(fileLength / 16) * 16));
if ( fputc ( h, fp ) != h ) {
printf( "** ERROR writing data\n" );
exit( 1 );
}

/* Fill the first block with zeros if it is not eqaul to 128 bits.
And cipher it.*/
 if (h > 0){
    fread( block, 1, h, fdata );
            for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
d[j][i]=block[i+j*4];
}
     }
            __asm__("NOP/a");
    rijndaelEncrypt (d, 128, 128, W);
    __asm__("NOP/a");
    for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
out[i+j*4]=d[j][i];
}
     }
    if ( fwrite( out, 16, 1, fp ) != 1 ) {
printf( "** ERROR writing data\n" );
exit( 1 );
}


      }
      /* Now process the rest of the data*/
      size = fread( block, 1, 16, fdata );
      while (size > 0){
  for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
d[j][i]=block[i+j*4];
}
     }
```

```
__asm__("NOP/a");
         rijndaelEncrypt (d, 128, 128, W);
__asm__("NOP/a");

  for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
out[i+j*4]=d[j][i];
}
     }

  if ( fwrite( out, 16, 1, fp ) != 1 ) {
    printf( "** ERROR writing data\n" );
    exit( 1 );
  }
  size = fread( block, 1, 16, fdata );
       }
      fclose (fp);
      fclose (fdata);
   }
/* this is the decryption routine*/
    else {

/* load reconfiguration data */
reconfbin2[0] = 3; /*MIX*/
for (i = 1; i < 276000; i++)
{
reconfbin2[i] = 0;
}

     __asm__("lw $8, reconf_addr2");
        __asm__("add/a/b $8, $8, $8"); /* Set intstruction*/
/* decode the first block*/
      h = fgetc (fdata);
      /*printf ("%d\n",i);*/
      if ( h > 0 ){
fread( block, 1, 16, fdata );
for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
d[j][i]=block[i+j*4];
}
        }
__asm__("NOP/a");

        rijndaelDecrypt (d, 128, 128, W);
__asm__("NOP/a");
```

```
        for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
out[i+j*4]=d[j][i];
}
        }
if ( fwrite( out, h, 1, fp ) != 1 ) {
  printf( "** ERROR writing data\n" );
  exit( 1 );
}
      }
      size = fread( block, 1, 16, fdata );
      /* now the header is away lets decrypt */
      while (size > 0)
{
        /*fread( block, 1, 16, fdata );*/
for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
d[j][i]=block[i+j*4];
}
     }
     __asm__("NOP/a");
        r = rijndaelDecrypt (d, 128, 128, W);
__asm__("NOP/a");
        for (j=0;j<4;j++)  {
for (i=0;i<4;i++){
out[i+j*4]=d[j][i];
}
     }
  if ( fwrite( out, 16, 1, fp ) != 1 ) {
    printf( "** ERROR writing data\n" );
    exit( 1 );
  }
  size = fread( block, 1, 16, fdata );
}
      fclose (fp);
      fclose (fdata);
    }
  }


int main(int argc, char* argv[]){

  /*struct tms before, after;
        */
  if (argc == 1){
```

```
    usage();
    exit(1);
  }

  /* checking on arguments */

  if (argc == 5){
    proc (argv[2], argv[3], argv[4], argv[1]);

  }
  else {
    usage();
    exit(1);
  }
}
```

C.2 app_config.c

```
/*****************************************************/
/* The MOLEN Application Interface variables for the MOLEN architecture */
/*****************************************************/

/*#define TEST
 */
enum configuration {NOCONFIG, EMPTY, MIX, IMIX};

#define KNOWN_CONFIGURATIONS  4


#define MIX_RECONF_LOAD     228000 /*228000 bytes reconfiguration code */
#define MIX_MICRO_LOAD      1 /*microcode*/
#define MIX_RECONF_LAT      164290000 /*Reconfiguration time*/
#define MIX_EXEC_LAT        6        /*1 cycles*/
#define MIX_DATA_LOAD       16       /*16 bytes, size of the cipher state */
#define MIX_DATA_STORE      16 /*16 bytes*/

#define IMIX_RECONF_LOAD    276000 /*276 Kbytes*/
#define IMIX_MICRO_LOAD     1 /* microcode*/
#define IMIX_RECONF_LAT     164290000 /*Reconfiguration time*/
#define IMIX_EXEC_LAT       3 /*2 cycles*/
#define IMIX_DATA_LOAD      16 /*16 bytes*/
#define IMIX_DATA_STORE     16 /*16 bytes*/
```

```
/* If no value, use: FALSE */int
reconf_micro_size[4] =
{0,
 0,
   MIX_RECONF_LOAD,
   IMIX_RECONF_LOAD};

int micro_size[4] =
  {0,
   0,
   MIX_MICRO_LOAD,
   IMIX_MICRO_LOAD};

int reconf_latency[4] =
  {0,
   0,
   MIX_RECONF_LAT,
   IMIX_RECONF_LAT};

int execution_latency[4] =
  {0,
   0,
   MIX_EXEC_LAT,
   IMIX_EXEC_LAT};

int data_load_size[4] =
  {0,
   0,
   MIX_DATA_LOAD,
   IMIX_DATA_LOAD};

int data_store_size[4] =
  {0,
   0,
   MIX_DATA_STORE,
   IMIX_DATA_STORE};

/************************************************/
/*********** Additional functions ***************/
/************************************************/


/* Inorder to deal with overhead, the following function
*  traces the simulation clock counter .
*  The annotation "/a" and "/a/c" are used
```

```
*/
static tick_t func1_cycle  = 0;
static tick_t func2_cycle  = 0;
int sim_cycle_save1 = 0;
int sim_cycle_save2 = 0;

void trace_routine(int annotated_field, tick_t sim_cycle)
{
/* Trace the amount of clock cycles during an operation*/
if (annotated_field == 1){



if (sim_cycle_save1 == 0) {
sim_cycle_save1 = sim_cycle;
//printf("* start function1 at cycle : %d \n", sim_cycle);
}
else {
func1_cycle = func1_cycle + (sim_cycle - sim_cycle_save1);
// printf("* stop function1 at cycle : %d \n", sim_cycle);
//printf(" function 1 cycles is : %d \n\n", func1_cycle);
sim_cycle_save1 = 0;
}


}

/* Trace operation counter*/
if ( annotated_field == 6){



if (sim_cycle_save2 == 0) {
sim_cycle_save2 = sim_cycle;
//printf("* start function1 at cycle : %d \n", sim_cycle);
}
else {
func2_cycle = func2_cycle + (sim_cycle - sim_cycle_save2);
//printf("* stop function1 at cycle : %d \n", sim_cycle);
//printf(" function 1 cycles is : %d \n\n", func1_cycle);
sim_cycle_save2 = 0;
}


}
```

```
}
```

C.3 molen_config.c

```
/*************************************************************************/
/*** This is configuration file that is related to the MOLEN processor */
/*************************************************************************/

/* Line included in order to detect annotated instructions*/
#define MD_ANNOTATION(INST) (INST.a >> 16)

/* Enumeration of the instruction type. */
enum instruction_type {NOTYPE, SET, EXECUTE};

/*****************************************/
/*The Custum Configured Unit configurations*/
/*****************************************/
/*The total number of functional untis
configured at the same moment on the CCU*/
#define NO_CONFIGURATIONS     2

/* This variable represents the amount of possible configuration on the CCU*/
int no_configurations = NO_CONFIGURATIONS;

/*This array represents what is actually configured on the CCU*/
int CCU_config_status[4] = {NOCONFIG, NOCONFIG, NOCONFIG, NOCONFIG};



#define SET_OPLAT        1
#define SET_ISSUELAT     1
#define EXEC_OPLAT       10
#define EXEC_ISSUELAT    10

/*************************************/
/*The rho-Control Store configurations*/
/*************************************/

/* The rho-Control Store width */
#define rcs_buswidth 4
```

```
/* The following enumeration determines whether location of
   reconfiguration code or microcode. */
enum code_location {RESIDENT, PAGEABLE, UNDETERMINED};


/* The content of the set store facility*/
#define NO_FIXED_CONFIG_CODE        4
#define NO_PAGEABLE_CONFIG_CODE     4
int set_fixed[NO_FIXED_CONFIG_CODE][2] = { {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}};
int set_pageable[NO_PAGEABLE_CONFIG_CODE][2] = { {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}, {EMPTY,


/* The content of the xecute store facility*/
#define NO_FIXED_MICRO_CODE        4
#define NO_PAGEABLE_MICRO_CODE     4
int exec_fixed[NO_FIXED_MICRO_CODE][2] = { {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}};
int exec_pageable[NO_PAGEABLE_MICRO_CODE][2] = { {EMPTY, 0}, {EMPTY, 0}, {EMPTY, 0}, {EMPTY,


#define LOCATION     (((annotated_field > 1) && (annotated_field < 6)) ? \
      ((annotated_field & 1) ? PAGEABLE : RESIDENT) \
      : UNDETERMINED )




/*****************************/
/* Various global counters  */
/*****************************/
/* The following keeps track of how many times a specific operation is
   being executed by the execute instruction. */
counter_t exec_times_issued[KNOWN_CONFIGURATIONS];
static counter_t set_pageable_times = 0;
static counter_t exe_pageable_times = 0;
static counter_t num_special_nops = 0;
static counter_t num_set_insn = 0;
static counter_t num_exe_insn = 0;
static counter_t squashed_exe_insn = 0;
static counter_t load_port_usage = 0;
static counter_t av_load_latency = 0;
static int stop_dispatch;
```

C.4 hardware.c

```
void hardware_execution (int field1_content, int field2_content, int field3_content)
  {
  int load_address = 0;
  int load_address2 = 0;
  int store_address = 0;
  md_addr_t addr;
  enum md_fault_type fault;
  byte_t temp_byte = 0; /* temp variable for spec mem access */
  half_t temp_half = 0; /* " ditto " */
  word_t temp_word = 0;
  enum configuration config = NOCONFIG;

  config = READ_WORD(field2_content,fault);

switch (config){
case MIX:
                load_address = field3_content;
        store_address = field3_content;
        load_address2 = field1_content;
/*algorithm of Mixcolumn*/

byte_t Logtable[256] = {
0,   0,  25,   1,  50,   2,  26, 198,  75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,  76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,   9, 120,
101,  47, 138,   5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
102, 221, 253,  48, 191,   6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,
151, 178, 135, 144,  97, 190, 220, 252, 188, 149, 207, 205,  55,  63,  91, 209,
83,  57, 132,  60,  65, 162, 109,  71,  20,  42, 158,  93,  86, 242, 211, 171,
68,  17, 146, 217,  35,  32,  46, 137, 180, 124, 184,  38, 119, 153, 227, 165,
103,  74, 237, 222, 197,  49, 254,  24,  13,  99, 140, 128, 192, 247, 112,   7,
};

byte_t Alogtable[256] = {
1,   3,   5,  15,  17,  51,  85, 255,  26,  46, 114, 150, 161, 248,  19,  53,
95, 225,  56,  72, 216, 115, 149, 164, 247,   2,   6,  10,  30,  34, 102, 170,
```

```
229,  52,  92, 228,  55,  89, 235,  38, 106, 190, 217, 112, 144, 171, 230,  49,
83, 245,   4,  12,  20,  60,  68, 204,  79, 209, 104, 184, 211, 110, 178, 205,
76, 212, 103, 169, 224,  59,  77, 215,  98, 166, 241,   8,  24,  40, 120, 136,
131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206,  73, 219, 118, 154,
181, 196,  87, 249,  16,  48,  80, 240,  11,  29,  39, 105, 187, 214,  97, 163,
254,  25,  43, 125, 135, 146, 173, 236,  47, 113, 147, 174, 233,  32,  96, 160,
251,  22,  58,  78, 210, 109, 183, 194,  93, 231,  50,  86, 250,  21,  63,  65,
195,  94, 226,  61,  71, 201,  64, 192,  91, 237,  44, 116, 156, 191, 218, 117,
159, 186, 213, 100, 172, 239,  42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
155, 182, 193,  88, 232,  35, 101, 175, 234,  37, 111, 177, 200,  67, 197,  84,
252,  31,  33,  99, 165, 244,   7,   9,  27,  45, 119, 153, 176, 203,  70, 202,
69, 207,  74, 222, 121, 139, 134, 145, 168, 227,  62,  66, 198,  81, 243,  14,
18,  54,  90, 238,  41, 123, 141, 140, 143, 138, 133, 148, 167, 242,  13,  23,
57,  75, 221, 124, 132, 151, 162, 253,  28,  36, 108, 180, 199,  82, 246,   1,
};



int i, j;
int Data_addr;
byte_t a[4][4];
byte_t b[4][4];
byte_t temp[4];

/*load_address = field3_content;
store_address = field3_conten
load_address2 = field1_content;
*/


Data_addr = load_address; /* GPR(RT) */

/*
for (i=0;i<4;i++)
{
for (j=0;j<4;j++)
{
a[i][j]=READ_BYTE(Data_addr,fault);
Data_addr+=1;
}
}
*/

/* Read Data */
for (j=0;j<4;j++)
```

```
{
a[0][j]=READ_BYTE(Data_addr,fault);
Data_addr+=1;
}

Data_addr = Data_addr + 4;

for (j=0;j<4;j++)
{
a[1][j]=READ_BYTE(Data_addr,fault);
Data_addr+=1;
}

Data_addr = Data_addr + 4;

for (j=0;j<4;j++)
{
a[2][j]=READ_BYTE(Data_addr,fault);
Data_addr+=1;
}



Data_addr = Data_addr + 4;

for (j=0;j<4;j++)
{
a[3][j]=READ_BYTE(Data_addr,fault);
Data_addr+=1;
}




/*for (i=0;i<4;i++)
{
for (j=0;j<4;j++)
{
a[i][j]=READ_BYTE($2,fault);
Data_addr+=1;
}
}
*/
/* printf("*********DataLoadaddres : = %d\n",Data_addr);
*/
```

```
/*
printf("*********MOLENLoadaddres : = %d\n",Data_addr);
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_load_molen : = %d\n",a[i][j]);
}
}
*/



for(j = 0; j < 4; j++){
for(i = 0; i < 4; i++){



temp[0] = Alogtable[( Logtable[(a[i][j])] + Logtable[2]) %255];

temp[1] = Alogtable[ ( Logtable[3] +
Logtable[ (a[(i + 1) % 4 ][j]) ]
)%255];


b[i][j] = temp[0]
^ temp[1]
^ a[(i + 2) % 4][j]
^ a[(i + 3) % 4][j];
}
}



Data_addr = store_address;
/*
printf("*********Data Store addres : = %d\n",Data_addr);
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_store_molen : = %d\n",b[i][j]);
}
}
*/
```

```
/* writing the data to the memory*/

for (i=0;i<4;i++){
WRITE_BYTE(b[0][i],Data_addr,fault);

Data_addr += 1;
}


Data_addr = Data_addr + 4;

for (i=0;i<4;i++){
WRITE_BYTE(b[1][i],Data_addr,fault);

Data_addr += 1;
}

Data_addr = Data_addr + 4;

for (i=0;i<4;i++){
WRITE_BYTE(b[2][i],Data_addr,fault);

Data_addr += 1;
}

Data_addr = Data_addr + 4;

for (i=0;i<4;i++){
WRITE_BYTE(b[3][i],Data_addr,fault);

Data_addr += 1;
}



WRITE_BYTE(1,load_address2,fault);


break;
case IMIX:
load_address = field3_content;
store_address = field3_content;
load_address2 = field1_content;
/* Inverse MIxcolumn */
```

```
byte_t Logtable_inverse[256] = {
0,   0,  25,   1,  50,   2,  26, 198,  75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,  76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,   9, 120,
101,  47, 138,   5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
102, 221, 253,  48, 191,   6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,
151, 178, 135, 144,  97, 190, 220, 252, 188, 149, 207, 205,  55,  63,  91, 209,
83,  57, 132,  60,  65, 162, 109,  71,  20,  42, 158,  93,  86, 242, 211, 171,
68,  17, 146, 217,  35,  32,  46, 137, 180, 124, 184,  38, 119, 153, 227, 165,
103,  74, 237, 222, 197,  49, 254,  24,  13,  99, 140, 128, 192, 247, 112,   7,
};

byte_t Alogtable_inverse[256] = {
1,   3,   5,  15,  17,  51,  85, 255,  26,  46, 114, 150, 161, 248,  19,  53,
95, 225,  56,  72, 216, 115, 149, 164, 247,   2,   6,  10,  30,  34, 102, 170,
229,  52,  92, 228,  55,  89, 235,  38, 106, 190, 217, 112, 144, 171, 230,  49,
83, 245,   4,  12,  20,  60,  68, 204,  79, 209, 104, 184, 211, 110, 178, 205,
76, 212, 103, 169, 224,  59,  77, 215,  98, 166, 241,   8,  24,  40, 120, 136,
131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206,  73, 219, 118, 154,
181, 196,  87, 249,  16,  48,  80, 240,  11,  29,  39, 105, 187, 214,  97, 163,
254,  25,  43, 125, 135, 146, 173, 236,  47, 113, 147, 174, 233,  32,  96, 160,
251,  22,  58,  78, 210, 109, 183, 194,  93, 231,  50,  86, 250,  21,  63,  65,
195,  94, 226,  61,  71, 201,  64, 192,  91, 237,  44, 116, 156, 191, 218, 117,
159, 186, 213, 100, 172, 239,  42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
155, 182, 193,  88, 232,  35, 101, 175, 234,  37, 111, 177, 200,  67, 197,  84,
252,  31,  33,  99, 165, 244,   7,   9,  27,  45, 119, 153, 176, 203,  70, 202,
69, 207,  74, 222, 121, 139, 134, 145, 168, 227,  62,  66, 198,  81, 243,  14,
18,  54,  90, 238,  41, 123, 141, 140, 143, 138, 133, 148, 167, 242,  13,  23,
57,  75, 221, 124, 132, 151, 162, 253,  28,  36, 108, 180, 199,  82, 246,   1,
};


int k, n = 0 ;
int d_addr =0;

/*int Datai_addr ;
*/
byte_t c[4][4];
```

```
byte_t d[4][4];
byte_t tmp[4];


d_addr = load_address; /* GPR(RT) */

/*
for (n=0;n<4;n++)
{
for (k=0;k<4;k++)
{
c[n][k]=READ_BYTE(d_addr,fault);
d_addr+=1;
}
}
*/

/* Read Data */
for (j=0;j<4;j++)
{
c[0][j]=READ_BYTE(d_addr,fault);
d_addr+=1;
}

d_addr = d_addr + 4;

for (j=0;j<4;j++)
{
c[1][j]=READ_BYTE(d_addr,fault);
d_addr+=1;
}

d_addr = d_addr + 4;

for (j=0;j<4;j++)
{
c[2][j]=READ_BYTE(d_addr,fault);
d_addr+=1;
}



d_addr = d_addr + 4;

for (j=0;j<4;j++)
```

```
{
c[3][j]=READ_BYTE(d_addr,fault);
d_addr+=1;
}



#ifdef  TEST
printf("*********MOLENLoadaddres : = %d\n",d_addr);
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_load_molen : = %d\n",c[i][j]);
}
}

#endif




for(n = 0; n < 4; n++){
for(k = 0; k < 4; k++) {

tmp[0] = Alogtable_inverse[(Logtable_inverse[0xe] + Logtable_inverse[(c[k][n])])%255];


tmp[1] = Alogtable_inverse[(Logtable_inverse[0xb] + Logtable_inverse[ (c[(k + 1) % 4 ][n]) ]


tmp[2] = Alogtable_inverse[(Logtable_inverse[0xd] + Logtable_inverse[ (c[(k + 2) % 4 ][n])  ]


tmp[3] = Alogtable_inverse[(Logtable_inverse[0x9] + Logtable_inverse[ (c[(k + 3) % 4 ][n])  ]



d[k][n] = tmp[0]
^ tmp[1]
^ tmp[2]
^ tmp[3];
```

```
}

}

d_addr = store_address;

/* writing the data to the memory*/

#ifdef  TEST
printf("*********Data Store addres : = %d\n",d_addr);
for (i=0;i<4;i++)
{
for (j=0;j<4;j++){
printf("data_store_molen : = %d\n",d[i][j]);
}
}

#endif


for (i=0;i<4;i++){
WRITE_BYTE(d[0][i],d_addr,fault);

d_addr += 1;
}


d_addr = d_addr + 4;

for (i=0;i<4;i++){
WRITE_BYTE(d[1][i],d_addr,fault);

d_addr += 1;
}

d_addr = d_addr + 4;

for (i=0;i<4;i++){
WRITE_BYTE(d[2][i],d_addr,fault);

d_addr += 1;
}

d_addr = d_addr + 4;
```

```
for (i=0;i<4;i++){
WRITE_BYTE(d[3][i],d_addr,fault);

d_addr += 1;
}



WRITE_BYTE(1,load_address2,fault);



break;
default :
printf ("CONFIG NUMBER: %d!!\n", config);
fatal ("Unknown operation encountered!!");
break;
}
}
```

AES hardware engines VHDL source code

D

```
--------------------------------------------------------------------------------
-- File Name    : mixc.vhd
-- Author       : R.M. Ashruf
-- Date         : May 2002
-- Project      : The Crypto Engine
-- purpose      : Building a fast mixcolumn for the Rijndael Algorithm
-- bugs         : None
-- system : Big Endian
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- Multiplication by Constants that are needed in encryption mode.
--------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--  Mulitplication by constant 2

entity mul_02 is
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;


  input: in bit_vector( 0 to 7);  -- 8-bit input

  output: out bit_vector( 0 to 7)  -- 8-bit output
  );
end mul_02;



architecture behavior of mul_02 is
begin
```

```
process(clk)
begin
if reset = '1' then
       output <= (others => '0');


elsif(clk'event and clk='1') then

     output(0) <= input (1);
     output(1) <= input (2);
     output(2) <= input (3);
    output(3) <= input (4) xor input(0);
      output(4) <= input (5) xor input(0);
     output(5) <= input (6);
     output(6) <= input (7) xor input(0);
     output(7) <= input (0);

end if;
end process;
end behavior;




library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--  Mulitplication by constant 3

entity mul_03 is
  port (
    clk: in std_logic;
reset: in std_logic;

    input: in bit_vector(0 to 7);  -- 8-bit input
    output: out bit_vector(0 to 7) -- 8-bit output
    );
end mul_03;

architecture behavior of mul_03 is
  begin
  process(clk)
begin
```

```
if reset = '1' then
       output <= (others => '0');
elsif(clk'event and clk='1') then

output(0) <= input (0) xor input (1);
output(1) <= input (1) xor input (2);
output(2) <= input (2) xor input (3);
output(3) <= input (3) xor input (4) xor input(0);
output(4) <= input (4) xor input (5) xor input(0);
output(5) <= input (5) xor input (6);
output(6) <= input (6) xor input (7) xor input(0);
output(7) <= input (7) xor input (0);
end if;
end process;
  end behavior;




library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity column is
  port (
   clock :  in std_logic;
   reset :  in std_logic;

    c_input : in bit_vector( 0 to 31);    -- 128-bit input
--done : out std_logic;
        c_output: out bit_vector( 0 to 31 )   -- processed ouput 128-bit
    );
end column;


-- ============================================================================
-- ========================== Component Definition ============================
-- ============================================================================
architecture behavior of column is


component mul_02
    port (
```

```
        clk: in std_logic;
reset: in std_logic;

      input : in bit_vector(0 to 7);
      output : out bit_vector(0 to 7)
      );
  end component;

  component mul_03
    port (
     clk: in std_logic;
reset: in std_logic;
     input : in bit_vector(0 to 7);
     output : out bit_vector(0 to 7)
      );
  end component;

-- ================================================================================
-- ================ Type, Signal, & Constant Definitions ======================
-- ================================================================================


signal V1 : bit_vector(0 to 7);
signal V2 : bit_vector(0 to 7);
signal V3 : bit_vector(0 to 7);
signal V4 : bit_vector(0 to 7);
signal V5 : bit_vector(0 to 7);
signal V6 : bit_vector(0 to 7);
signal V7 : bit_vector(0 to 7);
signal V8 : bit_vector(0 to 7);
 begin

-- W1 -> 2A_1 + 3A_2 + A_3 + A_4
  U0:  mul_02 port map (clock, reset, c_input(0 to 7), V1);
   U1:  mul_03 port map (clock, reset, c_input(8 to 15), V2);
    c_output(0 to 7) <= V1 xor V2 xor c_input(16 to 23) xor c_input(24 to 31);

-- W2 -> A_1 + 2A_2 + 3A_3 + A_4
U2:  mul_02 port map (clock, reset, c_input(8 to 15), V3);
   U3:  mul_03 port map (clock, reset, c_input(16 to 23), V4);
    c_output(8 to 15) <= c_input(0 to 7) xor V3 xor V4 xor c_input( 24 to 31);

-- W3 -> A_1 + A_2 + 2A_3 + 3A_4
U4:  mul_02 port map (clock, reset, c_input(16 to 23), V5);
   U5:  mul_03 port map (clock, reset, c_input(24 to 31), V6);
```

```
    c_output(16 to 23) <= c_input(0 to 7) xor c_input(8 to 15)  xor V5 xor V6;

--W4 -> 3A_1 + A_2 + A_3 + 2A_4
U6:  mul_02 port map (clock, reset, c_input(24 to 31), V7);
   U7:  mul_03 port map (clock, reset, c_input(0 to 7), V8);
    c_output(24 to 31) <= c_input(8 to 15) xor V7 xor V8 xor c_input( 16 to 23);

end behavior;
```

```
-------------------------------------------------------------------------------
---- The Mixcolumn operation
-------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


-- ============================================================================
-- ========================== Interface Description ===========================
-- ============================================================================

entity mixcolumn is
  port (
   clock :  in std_logic;
   reset :  in std_logic;

        c_input : in bit_vector( 0 to 127);    -- 128-bit input
--done : out std_logic;
        c_output: out bit_vector( 0 to 127 )   -- processed ouput 128-bit
    );
end mixcolumn;

architecture behavior of mixcolumn is

-- ============================================================================
```

```
-- ========================== Component Definition ==========================
-- ============================================================================

component column
  port (
   clock :  in std_logic;
   reset :  in std_logic;

    c_input : in bit_vector( 0 to 31);    -- 128-bit input
--done : out std_logic;
        c_output: out bit_vector( 0 to 31 )   -- processed ouput 128-bit
    );
end component;




-- ============================================================================
-- ================ Type, Signal, & Constant Definitions ======================
-- ============================================================================

--signal V1 : bit_vector(0 to 32);
--signal V2 : bit_vector(0 to 32);
--signal V3 : bit_vector(0 to 32);
--signal V4 : bit_vector(0 to 32);


-- ============================================================================
-- ====================== The Mixcolumn Structure ============================
-- ============================================================================
-- The following generate statements create a the following matrix multiplication
--
--

----- [ A1 B1 C1 D1] [ 2 3 1 1]   [W1 X1 Y1 Z1]
----- [ A2 B2 C2 D2] [ 1 2 3 1]   [W2 X2 Y2 Z2]
----- [ A3 B3 C3 D3]  x [ 1 1 2 3] = [W3 X3 Y3 Z3]
----- [ A4 B4 C4 D4]  [ 3 1 1 2]   [W4 X4 Y4 Z4]
-----

   --if mode = "0" then -- encryption mode
 begin
-- the first 32 bytes
-- W1 -> 2A_1 + 3A_2 + A_3 + A_4
  U0:  column port map (clock, reset, c_input(0 to 31), c_output ( 0 to 31 ));
```

```
   U1:  column port map (clock, reset, c_input(32 to 63), c_output ( 32 to 63 ));
U2:  column port map (clock, reset, c_input(64 to 95), c_output ( 64 to 95 ));
   U3:  column port map (clock, reset, c_input(96 to 127), c_output(96 to 127 ));


end behavior;


library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Multiplication needed for Decryption

entity mul_0E is
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;

    input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end mul_0E;

architecture mul_0E of mul_0E is
begin
process(clk,reset)
begin
if reset = '1' then
       output <= (others => '0');
elsif(clk'event and clk='1') then
       output(0) <= input (1) xor input (2) xor input (3);
     output(1) <= input (2) xor input (3) xor input (4) xor input (0);
     output(2) <= input (3) xor input (4) xor input (5) xor input (1);
     output(3) <= input (4) xor input (5) xor input (6) xor input (2);
     output(4) <= input (5) xor input (6) xor input (7) xor input (1) xor input
(2);
    output(5) <= input (6) xor input (7) xor input (1);
    output(6) <= input (7) xor input(2);
     output(7) <= input (0) xor input (1) xor input (2);
end if;
end process;
end mul_0E;
```

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

 entity mul_0B is
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;

    input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end mul_0B;

architecture mul_0B of mul_0B is
  begin
process(clk,reset)
begin
if reset = '1' then
       output <= (others => '0');
elsif(clk'event and clk='1') then
    output(0) <= input (3) xor input (0) xor input (1);
    output(1) <= input (0) xor input (2) xor input (1) xor input (4);
    output(2) <= input (3) xor input (0) xor input (2) xor input (1) xor input
(5);
    output(3) <= input (4) xor input (3) xor input (2) xor input (1) xor input
(0) xor input (6);
    output(4) <= input (2) xor input (5) xor input (4) xor input (7);
    output(5) <= input (1) xor input (0) xor input (5) xor input (6);
    output(6) <= input (2) xor input (1) xor input (0) xor input (7) xor input
(6);
    output(7) <= input (0) xor input (7) xor input (2);
   end if;
end process;
end mul_0B;

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mul_0D is
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;
```

```
    input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end mul_0D;

architecture mul_0D of mul_0D is
begin
process(clk,reset)
begin
if reset = '1' then
       output <= (others => '0');
elsif(clk'event and clk='1') then
    output(0) <= input (3) xor input (2) xor input (0);
    output(1) <= input (4) xor input (3) xor input (1) xor input (0);
    output(2) <= input (1) xor input (4) xor input (5) xor input (2) ;
    output(3) <= input (6) xor input (5) xor input (3) xor input (2) xor input
(0);
    output(4) <= input (7) xor input (6) xor input (4) xor input (4) ;
    output(5) <= input (7) xor input (5) xor input (4);
    output(6) <= input (6) xor input (3) xor input (2) xor input (1);
    output(7) <= input (7) xor input (6) xor input (4);

end if;
end process;
end mul_0D;

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mul_09 is
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;

    input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end mul_09;

architecture mul_09 of mul_09 is
  begin
```

```
process(clk,reset)
begin
if reset = '1' then
       output <= (others => '0');
elsif(clk'event and clk='1') then
    output(0) <= input (3) xor input (0);
    output(1) <= input (0) xor input (1) xor input (4);
    output(2) <= input (1) xor input (2) xor input (0) xor input (5);
    output(3) <= input (2) xor input (1) xor input (3) xor input (6);
    output(4) <= input (2) xor input (4) xor input (0) xor input (7);
    output(5) <= input (1) xor input (0) xor input (5);
    output(6) <= input (2) xor input (1) xor input (6) ;
    output(7) <= input (7) xor input (2) ;
   end if;
end process;
end mul_09;



library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity column is
  port (
   clock :  in std_logic;
    reset :  in std_logic;

    d_input : in bit_vector( 0 to 31);    -- 128-bit input
--done : out std_logic;
        d_output: out bit_vector( 0 to 31 )   -- processed ouput 128-bit
    );
end column;


-- ================================================================================
-- ========================== Component Definition ==========================
-- ================================================================================
architecture behavior of column is

component mul_0E
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;
```

```
    input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end component;

component mul_0D
  port (
      clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;
input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );

end component;

component mul_0B
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;
input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );
end component;

component mul_09
  port (
  clk       : in STD_LOGIC;
  reset       : in STD_LOGIC;
input: in bit_vector (0 to 7);
    output: out bit_vector (0 to 7)
    );
end component;

-- ============================================================================
-- ================ Type, Signal, & Constant Definitions ======================
-- ============================================================================


signal V1 : bit_vector(0 to 7);
signal V2 : bit_vector(0 to 7);
signal V3 : bit_vector(0 to 7);
signal V4 : bit_vector(0 to 7);
signal V5 : bit_vector(0 to 7);
signal V6 : bit_vector(0 to 7);
```

```
signal V7 : bit_vector(0 to 7);
signal V8 : bit_vector(0 to 7);
signal X1 : bit_vector(0 to 7);
signal X2 : bit_vector(0 to 7);
signal X3 : bit_vector(0 to 7);
signal X4 : bit_vector(0 to 7);
signal X5 : bit_vector(0 to 7);
signal X6 : bit_vector(0 to 7);
signal X7 : bit_vector(0 to 7);
signal X8 : bit_vector(0 to 7);

 begin
q0:  mul_0E port map (clock, reset, d_input(0 to 7), V1);
   q1:  mul_0B port map (clock, reset, d_input(8 to 15), V2);
Y0: mul_0D port map ( clock, reset, d_input(16 to 23), X1);
Y1:  mul_09 port map ( clock, reset, d_input(24 to 31), X2);
    d_output(0 to 7) <= V1 xor V2 xor X2 xor X1;

-- W2 -> 0b A_1 + 0e A_2 + 09 A_3 + 0d A_4
Y2: mul_09 port map (clock, reset, d_input(0 to 7), X3);
q2:  mul_0E port map (clock, reset, d_input(8 to 15), V3);
   q3:  mul_0B port map ( clock, reset,d_input(16 to 23), V4);
   Y3:  mul_0D port map (clock, reset, d_input(24 to 31), X4);
d_output(8 to 15) <= X3 xor X4 xor V3 xor V4;

-- W3 -> 0D A_1 + 0B A_2 + 0E A_3 + 09 A_4
Y4: mul_0D port map (clock, reset, d_input(0 to 7), X5);
q4:  mul_09 port map (clock, reset, d_input(8 to 15), V6);
   q5:  mul_0E port map (clock, reset, d_input(16 to 23), V5);
   Y5:  mul_0B port map (clock, reset, d_input(24 to 31), X6);
    d_output(16 to 23) <= X5 xor X6  xor V5 xor V6;

--W4 -> 09xA_1 + 0dxA_2 + 0bxA_3 + 0exA_4
Y6: mul_0B port map ( clock, reset, d_input(0 to 7), X7);
q6:  mul_0D port map ( clock, reset, d_input(8 to 15), V7);
   q7:  mul_09 port map ( clock, reset, d_input(16 to 23), V8);
   Y7:  mul_0E port map ( clock, reset, d_input(24 to 31), X8);
    d_output(24 to 31) <= X7 xor X8 xor V7 xor V8;

end behavior;


------------------------------------------------------------------------------------------
----
---- Inverse mixcolumn
```

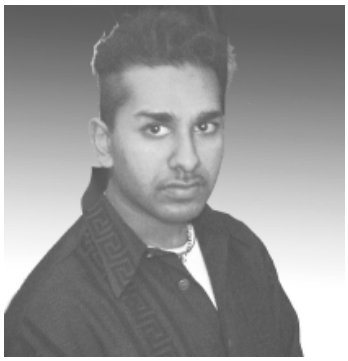
```
--------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Invmixcolumn is
  port (
   clock : in std_logic; -- rising edgde clock
   reset : in std_logic; -- active high reset
   --mode : in std_logic;  -- "0" encryption or "1" decryption mode
   --st_job  : in std_logic; -- "1" start doing your job
        d_input: in bit_vector( 0 to 127);    -- 128 bits input

        --done : out std_logic;  -- "0" computation is done, "1" still busy
     d_output:  out bit_vector( 0 to 127 ) -- processed ouput 128 bits data
    );
end Invmixcolumn;

architecture behaviorinv of Invmixcolumn is

-------------------------------------------------
---- component declaration
-------------------------------------------------


component column
  port (
   clock :  in std_logic;
   reset :  in std_logic;

    d_input : in bit_vector( 0 to 31);
    d_output: out bit_vector( 0 to 31 )
    );
end component;


-------------------------------------------------
--- signal definition
-------------------------------------------------



  begin
```

```
-- the first 32 bytes
-- W1 -> 2A_1 + 3A_2 + A_3 + A_4
  U0:  column port map (clock, reset, d_input(0 to 31), d_output ( 0 to 31 ));
   U1:  column port map (clock, reset, d_input(32 to 63), d_output ( 32 to 63 ));
U2:  column port map (clock, reset, d_input(64 to 95), d_output ( 64 to 95 ));
   U3:  column port map (clock, reset, d_input(96 to 127), d_output(96 to 127 ));



end behaviorinv;
```

```
-------------------------------------------------------------------------------------------
----
---- Inverse mixcolumn
-------------------------------------------------------------------------------------------

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Invmixcolumn is
  port (
   clock : in std_logic; -- rising edgde clock
   reset : in std_logic; -- active high reset
   --mode : in std_logic;  -- "0" encryption or "1" decryption mode
   --st_job  : in std_logic; -- "1" start doing your job
        d_input: in bit_vector( 0 to 127);    -- 128 bits input

        --done : out std_logic;  -- "0" computation is done, "1" still busy
     d_output:  out bit_vector( 0 to 127 ) -- processed ouput 128 bits data
    );
end Invmixcolumn;

architecture behaviorinv of Invmixcolumn is

-------------------------------------------------
---- component declaration
-------------------------------------------------
```

```
component column
  port (
   clock :  in std_logic;
   reset :  in std_logic;

    d_input : in bit_vector( 0 to 31);
    d_output: out bit_vector( 0 to 31 )
    );
end component;


--------------------------------------------------
--- signal definition
--------------------------------------------------




  begin
-- the first 32 bytes
-- W1 -> 2A_1 + 3A_2 + A_3 + A_4
  U0:  column port map (clock, reset, d_input(0 to 31), d_output ( 0 to 31 ));
   U1:  column port map (clock, reset, d_input(32 to 63), d_output ( 32 to 63 ));
U2:  column port map (clock, reset, d_input(64 to 95), d_output ( 64 to 95 ));
   U3:  column port map (clock, reset, d_input(96 to 127), d_output(96 to 127 ));



end behaviorinv;
```

Curriculum Vitae



Raoel Ashruf was born in Paramaribo, Suriname on the 9th of August 1977. In 1978, he emigrated with his family to the Netherlands. He have done both primary and secondary school (basisonderwijs and VWO) in Amsterdam. In 1996, he continued his education at the Faculty of Electrical Engineering, Delft Universtiy of Technology. In December 2001, he started his Master project at the Computer Engineering (CE) Laboratory, under the supervision of ir. Georgi N. Gaydadjiev.