DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

IMPLEMENTATION AND COMPARISON OF ADVANCED ENCRYPTION STANDARD (AES) MODES ON FPGA

by

Murat KARATOPRAK

February, 2011

İZMİR

IMPLEMENTATION AND COMPARISON OF ADVANCED ENCRYPTION STANDARD (AES) MODES ON FPGA

A Thesis Submitted to the

Graduate School of Natural And Applied Sciences of Dokuz Eylül University

In Partial Fulfillment of the Requirements for the Degree of Master of Science

in Electrical and Electronics Engineering

by

Murat KARATOPRAK

February, 2011

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**IMPLEMENTATION AND COMPARISON OF ADVANCED ENCRYPTION STANDARD (AES) MODES ON FPGA**" completed by **MURAT KARATOPRAK** under supervision of **ASST. PROF. DR. ÖZGE ŞAHİN** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Özge ŞAHİN

Supervisor

(Jury Member)

(Jury Member)

Prof. Dr. Mustafa SABUNCU
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEGMENTS

I would like to thank to my advisor Asst. Prof Dr. Özge Şahin for her encouragements throughout this research. I also would like to thank my family for their endless support.

MURAT KARATOPRAK

IMPLEMENTATION AND COMPARISON OF ADVANCED ENCRYPTION STANDARD (AES) MODES ON FPGA

ABSTRACT

System-On-Chip (SoC) is an interesting target platform that includes both hardware and software on a single chip which makes an embedded system a typical development environment. The main idea for this thesis is to study and implement state-of-the-art cryptographic block cipher Advanced Encryption Standard (AES) modes of operation on a SoC development environment.

In this thesis implementation and comparison of AES block cipher algorithm modes of operation on a Xilinx SoC development platform have been accomplished. It consists of two parts, hardware and software and both sections have been developed by using Xilinx licensed Embedded Development Kit (EDK). At the hardware section the hardware input output interfaces are determined according to the requirements of the project and the corresponding hardware is designed. At the second section, the software requirements are determined similar to hardware, AES and modes of operation is developed by using "C" as the programming language and the software is tested by commands entered through serial port. A detailed analysis of AES and modes of operation, MicroBlaze soft processor core architecture is investigated. Implementation is realized on a soft processor core, MicroBlaze and analyzed using mb-gprof profiler (a gprof based profiler). A software intellectual property (IP) that is capable of demonstrating all modes of operation including electronic code book (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB) and counter (CTR) modes is generated and tested with build-in test application commands and each mode is compared in terms of time taken to encrypt-decrypt messages.

Keywords: MicroBlaze, Profiler, AES, Modes of operation.

GELİŞMİŞ ŞİFRELEME STANDARDI MODLARININ FPGA ÜZERİNDE GERÇEKLENMESİ VE KARŞILAŞTIRILMASI

ÖZ

Sistem-On-Chip (SoC) hem donanım hem de yazılımı tek bir çip üzerinde içeren, gömülü bir sistemi tipik bir geliştirme ortamı yapan ilgi çekici bir hedef platformdur. Bu tezin ana fikri en son gelişmeleri yansıtan Gelişmiş Şifreleme Standardı (Advanced Encryption Standard - AES) blok şifreleme algoritması modlarının bir SoC geliştirme ortamında araştırılması ve uygulanmasıdır.

Bu tezde bir Xilinx SoC geliştirme platformu üzerinde Gelişmiş Şifreleme Standardı blok şifreleme algoritması modlarının uygulanması ve karşılaştırılması yapılmıştır. Çalışma donanım ve yazılım olmak üzere iki kısımdan oluşmaktadır. Her iki kısımda Xilinx lisanslı Gömülü Sistem Set (Embedded Development Kit - EDK)'i kullanılarak geliştirilmiştir. Donanım kısmında gerekli giriş-çıkış arayüzleri proje gereksinimlerine uygun şekilde seçilmiştir. İkinci kısımda yani yazılım kısmında ise benzer şekilde yazılım gereksinimleri belirlenmiş, AES ve çalışma modları "C" dili kullanılarak geliştirilmiş ve seri porttan girilen komutlarla test edilmiştir. AES ve çalışma modlarının, MicroBlaze soft-core mimarisinin ayrıntılı bir analizi yapılmıştır. Uygulama MicroBlaze soft-core mimarisi üzerinde gerçekleştirilmiş ve mb-gprof profiler (gprof tabanlı profiler) ile analiz edilmiştir. Elektronik kod kitabı (ECB), zincirleme şifre blok (CBC), şifre gizle (CFB), çıkış gizle (OFB) ve sayaç (CTR) modları dahil olmak üzere tüm çalışma modlarını gösterme yeteneğine sahip bir yazılım fikri mülkiyet (IP) yaratılmış ve dahili test uygulama komutları ile test edilerek her mod mesajları şifreleme-çözme sırasında çektikleri süre açısından karşılaştırılmıştır.

Anahtar sözcükler: MicroBlaze, Profiler, AES, Çalışma Modları.

CONTENTS

Page

CHAPTER ONE

INTRODUCTION

1.1 Introduction

Ever since man developed his communication skills, he has embarked on a journey of technological developments. These communication skills have been developed to such an extent that the information passed must, at times, be secret and authenticable. The new conditions of secrecy, authenticity and integrity have given rise to a new field of science called cryptology. Cryptology is divided into cryptography and cryptanalysis. Cryptography, deals with the art and science of encoding and decoding information, whereas, cryptanalysis deals with breaking the encoded information (Jayavardhan, 2003).

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques. Cryptography describes a number of basic cryptographic tools (primitives) used to provide information security. Figure 1.1 provides a schematic listing of the primitives considered and how they relate. These primitives should be evaluated with respect to various criteria such as:

1. *Level of security*. This is usually difficult to quantify. Often it is given in terms of the number of operations required (using the best methods currently known) to defeat the intended objective. Typically the level of security is defined by an upper bound on the amount of work necessary to defeat the objective. This is sometimes called the work factor.
2. *Functionality*. Primitives will need to be combined to meet various information security objectives. Which primitives are most effective for a given objective will be determined by the basic properties of the primitives.
3. *Methods of Operation*. Primitives, when applied in various ways and with various inputs, will typically exhibit different characteristics; thus, one

primitive could provide very different functionality depending on its mode of operation or usage.

4. *Performance.* This refers to the efficiency of a primitive in a particular mode of operation. (For example, an encryption algorithm may be rated by the number of bits per second which it can encrypt.)
5. *Ease of implementation.* This refers to the difficulty of realizing the primitive in a practical instantiation. This might include the complexity of implementing the primitive in either a software or hardware environment.

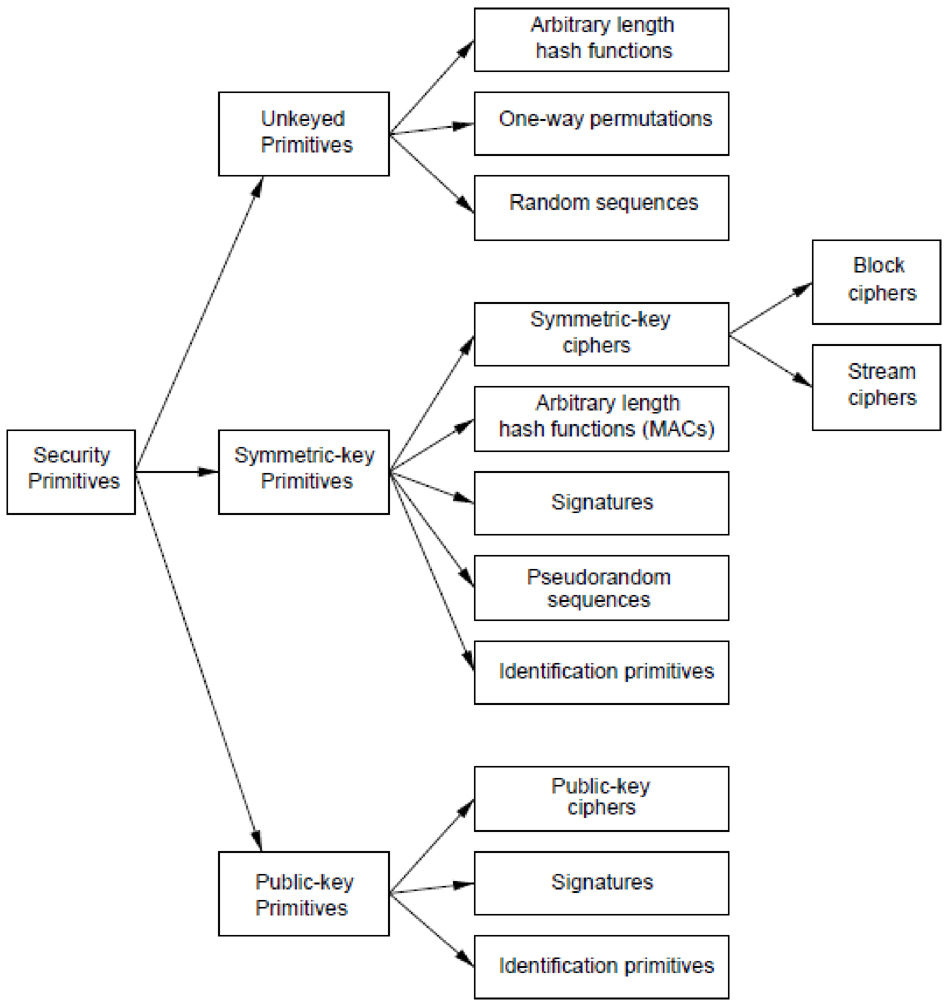


Figure 1.1 A classification of cryptographic primitives (tools) (Schneier, 1996)

Microprocessor obsolescence is a major concern for many companies. Programmable logic can provide a viable solution to this problem. By using soft core microprocessors embedded within a programmable logic device, not only can you own the processor core for use in any future devices and platforms, but the design can be both flexible and scalable to suit different platforms (Parnell & Bryner, 2004).

An emergent trend is to move from bespoke microprocessors to soft-core processors embedded within either FPGAs or ASICs. This trend has been driven by the long- term supply uncertainties of companies that provide bespoke microprocessors. This uncertainty is due to their inability to take advantage of new process technologies and geometries.

Embedded systems have become ubiquitous in recent years stemming from the exponential growth in mobile phones, PDAs, portable multimedia devices and smart cards. This has lead to a need for strong cryptography to protect users' identity, transactions and allow secure billing. This includes security in both wireless communications and authentication. Since embedded systems have limited resources then it is essential that the cryptography overhead is as small as possible. The main drawback with block ciphers like AES (NIST, 2001) is that they are quite costly to implement in software, but have simple hardware realizations using logical bit operations and manipulation. Offloading these operations from software to hardware using user-defined instructions tightly coupled to a processor leads to considerable clock cycle savings. The AES algorithm is specified in many wireless standards as the MAC protocol encryption method ((IEEE, 2007) & (IEEE, 2003)). (EnSilica Ltd, 2010).

As the need for secure data transmission grows, there is a major urgency of integrating cryptography into the embedded systems, in order to enable secure and reliable data transfer. Embedded systems populate the new generation gadgets such as cell phones and smartcards where the encryption algorithms are obviously an integral part of the system. Many conditional access vendors such as Nagravision, Viaccess, Irdeto requires their conditional access kernel libraries are not visible as a plaintext so forces their partners to use encryption systems with an approved mode of operation. Modes of operation enable the repeated and secure use of a block cipher under a single key. A block cipher by itself allows encryption only of a single data block of the cipher's block length. When targeting a variable-length message, the data must first be partitioned into separate cipher blocks. Typically, the last block must also be extended to match the cipher's block length using a suitable padding scheme. A mode of operation describes the process of encrypting each of these blocks, and generally uses randomization based on an additional input value, often called an initialization vector, to allow doing so safely.

This research explored the different cryptographic modes of operation which are approved by National Institute of Standards & Technology (NIST) that would enable an insertion of the cryptography into the embedded system, specifically on a MicroBlaze development environment and analyze time taken on operations with mb-gprof profiler tool, made a comparison between each modes of operation with regard to error properties and computational complexity.

1.2 Literature Overview

In 2001, the NIST selected Rijndael as the replacement for DES (FIPS 197). Flemish for XYZ and pronounced “rain-doll,” Rijndael is an interesting cipher, since it works in a completely different way from the previous ciphers. The algorithm is in some ways similar to shuffling and cutting a deck of cards. The interstate is laid out in a square, and the rows and columns are shifted, mixed, and added in various ways. The entries themselves are also substituted and altered. It has a lot of parallel and symmetric structure because of the mathematics, which provides a lot of flexibility in how it is implemented. However, some have criticized it as having too much structure, which may lead to future attacks. Apparently that didn’t bother the NSA (National Security Agency) or the NIST. No known cryptographic attacks are known, and it works well on a wide variety of processors, doesn’t use bit shifting or rotation, and is very fast (Galbreath, 2002).

A block cipher mode is an algorithm that features the use of a symmetric key block cipher algorithm to provide an information service, such as confidentiality or authentication. Currently, NIST has approved nine modes of the approved block ciphers in a series of special publications and there are six confidentiality modes (ECB, CBC, OFB, CFB, CTR, and XTS-AES), one authentication mode (CMAC), and two combined modes for confidentiality and authentication (CCM and GCM).

There are numerous studies implementing AES algorithm in FPGA and/or PC as crypto processor but with the lake of all modes of operation support.

A reconfigurable processor implementation is proposed by Yongzhi Fu, Lin Hao and Xuejie Zhang. This study is about the implementation of a counter mode AES based on the Xilinx Virtex2 FPGA platform whose difference is using a switch between MixColumns operation and AddRoundKey operation (Fu, Hao & Zhang, 2005).

In another study by Alireza Hodjat, David D. Hwang, Bocheng Lai, Kris Tiri and Ingrid Verbauwhede (Hodjat & Verbauwhede, 2006) an AES crypto processor, which can handle non feedback counter mode of operation is presented. It is reported that this implementation can achieve a throughput of 3.84 Gbps at a 330 MHz clock frequency. For the implementation of the non-feedback modes of the operation the design has a non-pipelined structure. The area efficient AES architecture with throughput rate of over 30 Gbits/s is used in the counter mode of operation for the encryption of data streams in optical networks.

In another study by Melek Dirayet Başkök (Başkök, 2007), a modeling of AES algorithm, which operates in CBC and ECB modes and gives permission to the use of file and text based encryption and decryption, has been implemented. In this modeling, C++ was chosen as the programming language and implementation is realized on PC.

In another study by R. W. Ward, Dr. T. C. A. Molteno (Ward & Molteno, 2002), a microcontroller with a CPLD to perform Rijndael encryption and decryption using the CPLD as a coprocessor for the microcontroller is used. This configuration gives improved throughput/power characteristics over using a microcontroller alone. Microcontrollers and CPLDs are both relatively low power devices, so such an arrangement could be used for encryption and decryption in an embedded device where power consumption is an issue. Such a device is likely to be used in an environment where some information is lost in transmission; in this study only non-feedback mode (ECB (Dworkin, 2001)) for encryption is considered.

This thesis is distinguished from others mentioned above in two ways. First, by studying and implementing all the NIST approved modes of operations using AES algorithm. Second it is generated on a soft processor core, MicroBlaze and "C" is chosen as the programming language so that the algorithm related application segment is portable to any embedded platform. The main concern of this study is to compare and to determine the most efficient mode of operation in terms of efficiency, computational complexity and timing.

1.3 Thesis Outline

This thesis is presented in six chapters. In chapter one, an introduction to the cryptography and soft processor cores, a literature investigation and studies about embedded cryptography together the differences with this thesis is presented. In chapter two theoretical aspects of soft processor core, MicroBlaze is given with information about Xilinx development tools; Integrated Software Environment (ISE) and Embedded Development Kit (EDK). In chapter three AES algorithm is investigated in detail. In chapter four approved modes of operation by NIST are analyzed. In chapter five implementation and experimental results are illustrated. In chapter six conclusion and future work is discussed.

CHAPTER TWO
TECHNOLOGY BACKGROUND & ENVIRONMENT

2.1 Integrated Circuits

2.1.1 System on Chip (SoC)

System on Chip (SoC) refers to devices where all essential parts of a computing system have been integrated in a single circuit. A typical SoC includes one (or many) processor core(s), an arbitrary number of peripherals, some on-chip memory and a bus architecture which interconnects all these devices. The SoC design goal is that only one circuit should required for an application. In practice a SoC may also contain a large set of I/O interfaces to other circuits, for example memory modules, off-chip peripherals, radio transceivers, network interfaces.

As SoCs usually are designed with a limited set of applications in mind, they tend to need less processing power than a general purpose computer. While a modern work-station operates at clock frequencies in the range of 500 MHz – 3 GHz, the SoC CPU might operate at just a few megahertz. An ideal SoC processor core is operating at the minimum clock frequency needed to properly perform the desired task. By utilizing a low clock frequency the power consumption and chip temperature is reduced. This allows SoCs to operate with less cooling devices and better battery/power utilization (Magnusson, 2004).

2.1.2 Application Specific Integration Circuit (ASIC)

ASIC is one of the most common chip types. An ASIC may implement simple designs as well as large designs such SoCs. An ASIC is designed for a specific application therefore it can be customized for reduces power dissipation, less chip area or greater clock frequencies. Normally ASICs have low mass production costs but non-recurring engineering (NRE) cost of ASICs is high.

2.1.3 Field Programmable Gate Array (FPGA)

FPGA is a type of programmable logic devices. FPGA is a generic architecture consisting of configurable logic blocks and programmable interconnections. Several FPGAs contain enough logic to implement SoCs and other large designs. FPGAs are not optimized for a specific application; therefore they may consume more power or implement a design less efficient than an ASIC. Price per chip is high however it is easy to reprogram, which shortens design cycles and allows early real world tests. This makes FPGAs well suited for prototypes and small production volumes. FPGAs may also be used for applications which are not of ASIC production quality such as first generation of manufacturing where standards and specifications are subject to change.

2.2 Processor Cores

A processor core refers to a processor excluding any peripherals it is used with. A traditional processor core resides in a dedicated processor chip. In SoC designs, one or more processor cores are integrated with peripherals on a single chip.

2.2.1 Soft, Firm and Hard Cores

The terms soft, firm and hard cores are originally ASIC manufacturing related words:

- "Soft Core" refers to cores delivered as a technology dependent gate-level netlist or Hardware Description Language (HDL) source code.
- "Firm Core" refers to cores delivered as a library element.
- "Hard Core" refers to cores which has a fixed physical layout and is incorporated into the design as a standard cell.

Firm and hard cores mainly apply to ASIC design. Soft cores are commonly used with programmable logic as well.

2.2.2 Instruction Set Architecture

An Instruction Set Architecture is a definition of how processor should perform an instruction. An instruction is a very short and basic command to the processor. Reduced Instruction Set Computer (RISC) refers to instruction set architectures with all or most of the following properties:

- Rapid execution of a small instruction set with simple instructions
- Uniform instruction length
- All processor registers are general purpose
- Simple addressing modes

RISC architectures are commonly used in microcontrollers and SoC cores.

2.2.3 Soft Processors

A soft processor is a "soft core" processor fully described in software, usually in an HDL, which can be synthesized in programmable hardware, such as FPGAs. A soft-core processor targeting FPGAs is flexible because its parameters can be changed at any time by reprogramming the device. Traditionally, systems have been built using general-purpose processors implemented as Application Specific Integrated Circuits (ASIC), placed on printed circuit boards that may have included FPGAs if flexible user logic was required. Using soft-core processors, such systems can be integrated on a single FPGA chip, assuming that the soft-core processor provides adequate performance. Recently, two commercial soft-core processors have become available: Nios (Altera Corporation, 2004) from Altera Corporation and MicroBlaze (Xilinx Inc., 2008) from Xilinx Inc. Soft processors have recently gained a lot of popularity that appears to be especially strong among FPGA developers. Reasons of this include:

- Performance increases (soft cores utilizes FPGA/ASICs better)
- Increased performance/price ratio on FPGAs
- Increased availability of both commercial and academic cores, as well as open cores.

2.3 Xilinx Development Tools

2.3.1 Integrated Software Environment (ISE)

ISE controls all aspects of the design flow. Through the Project Navigator interface, all of the design entry and design implementation tools can be accessed. The files and documents associated with the projects can also be accessed. Xilinx ISE (Xilinx Inc., 2008) is a software tool for synthesis and analysis of HDL designs, which enables the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

2.3.2 Embedded Development Kit (EDK)

EDK is the development package for building MicroBlaze (and PowerPC) embedded processor systems in Xilinx FPGAs. Hosted in the Eclipse IDE, the project manager consists of two separate environments: XPS and SDK.

Designers use XPS (Xilinx Platform Studio) to configure and build the hardware specification of their embedded system (processor core, memory-controller, I/O peripherals, etc.) The XPS converts the designer's platform specification into a synthesizable RTL description (Verilog or VHDL), and writes a set of scripts to automate the implementation of the embedded system (from RTL to the bit stream-file.) For the MicroBlaze core, the EDK normally generates an encrypted (non human-readable) netlist, but the processor description (written in VHDL) can be purchased from Xilinx.

The Board Support Package (BSP) is a collection of files that defines the hardware elements of your system for each processor. The BSP contains the various embedded software elements, such as software driver files, selected libraries, standard I/O devices, interrupt handler routines, and other related features. Consequently, it is easiest to have SDK generate the BSP after the hardware system is populated with its processors and peripherals and after the address map is defined.

As with the hardware assembly, SDK allows you to specify all aspects of software platform and manage software applications. The SDK handles the software that will execute on the embedded system. Powered by the GNU toolchain (GNU Compiler Collection, GNU Debugger), the SDK enables programmers to write, compile, and debug C/C++ applications for their embedded system. Xilinx includes a cycle-accurate instruction set simulator (ISS), giving programmers the choice of testing their software in simulation, or using a suitable FPGA-board to download and execute on the actual system (Xilinx Inc., 2008).

The tools described in section 2.3.1 and 2.3.2 expedites the design process as in Figure 2.1 which shows the simplified flow for an embedded design.

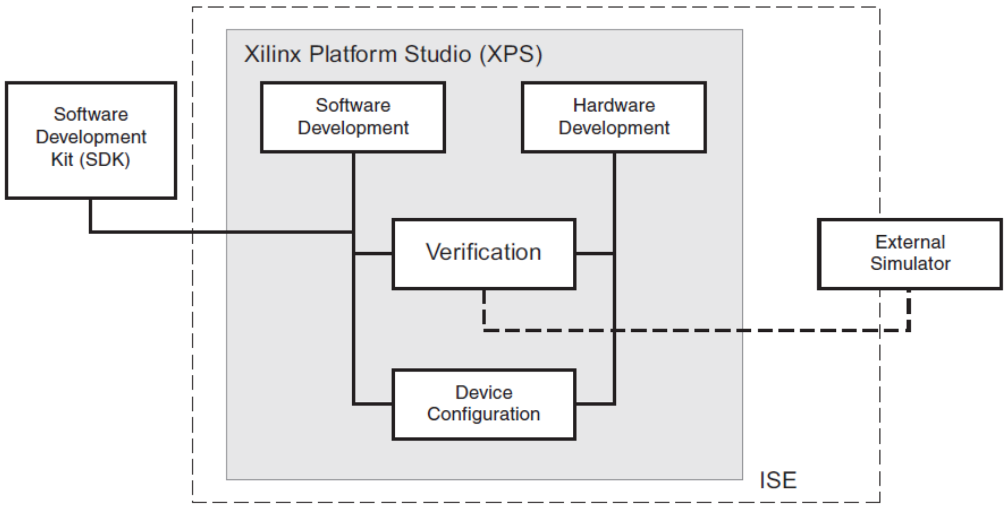


Figure 2.1 Basic Embedded Design Process Flow (Xilinx Inc., 2008)

2.4 The Target System Xilinx MicroBlaze Development Kit Spartan3E 1600E

The target system is a MicroBlaze Development Kit Spartan3E 1600E development board which is a SoC board from Xilinx. It consists of many different peripherals such as memory controllers, general purpose I/O (GPIO) and bus interfaces making it a fitting system in different areas. The MicroBlaze Development Kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing applications. The board highlights these features (Xilinx Inc., 2007):

- Spartan-3E specific features
    - Parallel NOR Flash configuration
    - MultiBoot FPGA configuration from Parallel NOR Flash PROM
    - SPI serial Flash configuration
- Embedded development
    - MicroBlaze 32-bit embedded RISC processor
    - PicoBlaze 8-bit embedded controller
    - DDR memory interfaces
    - 10-100 Ethernet
    - UART

The Spartan3E 1600E has support for two processors; a Xilinx's own soft processor core MicroBlaze RISC processor and a PicoBlaze 8-bit embedded controller. Spartan3E 1600E is no longer available for purchase from Xilinx as of December 2010.

2.4.1 Xilinx MicroBlaze Architecture

The soft-core processor used for this project is Microblaze (Parnell & Bryner, 2004). The MicroBlaze embedded processor soft core is a reduced instruction set computer (RISC), 5 stage pipeline, optimized for implementation in Xilinx field programmable gate arrays (FPGAs). Figure 2.2 shows a functional block diagram of the MicroBlaze core. MicroBlaze uses a big-endian numeric presentation meaning the most significant byte is assigned the lowest byte address. Many aspects of the MicroBlaze can be configured at compile time owing to the configurable nature of FPGAs. Cache structure, peripherals, and interfaces can be customized to the application. In addition, hardware support for certain operations, such as multiplication, division, and floating-point arithmetic, can be added or removed (Barma, 2007).

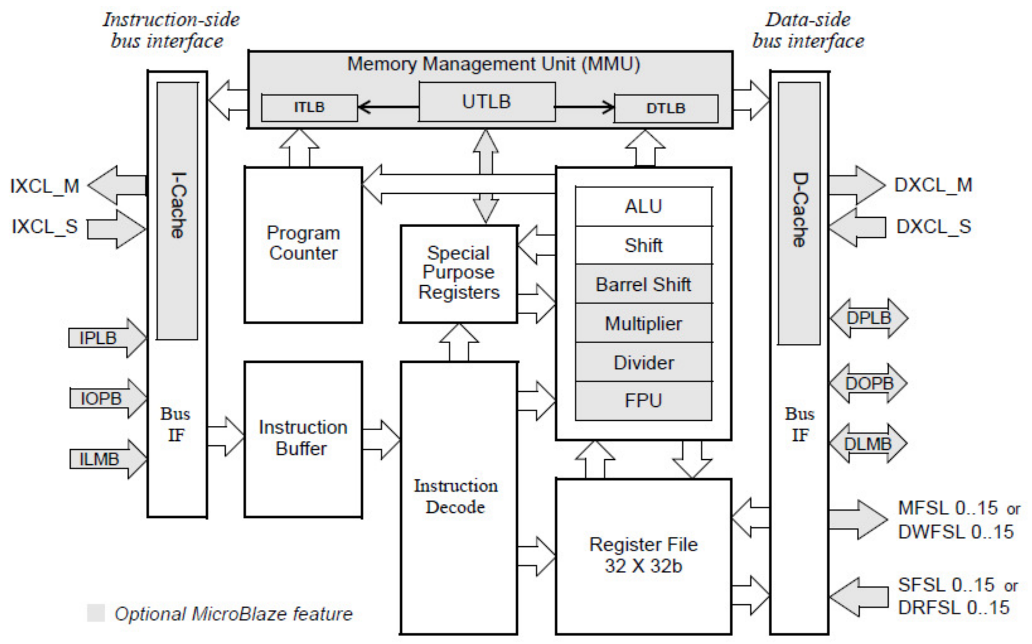


Figure 2.2 MicroBlaze (v7.0d) Core Block Diagram (Xilinx Inc., 2008)

DPLB: Data interface, Processor LocalBus.

DOPB: Data interface, On-chip Peripheral Bus

DLMB: Data interface, Local Memory Bus (BRAM only)

IPLB: Instruction interface, Processor Local Bus

IOPB: Instruction interface, On-chip Peripheral Bus

ILMB: Instruction interface, Local Memory Bus (BRAM only)

MFSL 0...15: FSL master interfaces

DWFSL 0...15: FSL master direct connection interfaces

SFSL 0...15: FSL slave interfaces

DRFSL 0...15: FSL slave direct connection interfaces

IXCL: Instruction side Xilinx CacheLink interface (FSL master/slave pair)

DXCL: Data side Xilinx CacheLink interface (FSL master/slave pair)

Core: Miscellaneous signals for clock, reset, debug, and trace.

General purpose registers, special purpose registers, a 32-bit address bus and a pipeline are all features that are fixed on MicroBlaze. The list below consists of some additional features that can be added to the MicroBlaze (Xilinx Inc., 2008):

- Hardware barrel shifter: A digital circuit that can shift data any number of bits in one operation. A vital component in floating point operations
- Hardware divider: Divide by zero hardware exception can only be enabled if the processor is configured with a hardware divider.
- Instruction and data cache: Consists of both an instruction and a data cache.
- On-chip peripheral bus (OPB)
- Processor Local Bus (PLB)
- Local memory bus (LMB)
- Fast Simplex Link (FSL)
- Xilinx CacheLink

2.4.1.1 Registers

MicroBlaze provides two kinds of registers, general purpose registers and special purpose registers.

General purpose registers; there are 32 general purpose registers divided into three categories. Volatile, non-volatile and dedicated (Xilinx Inc., 2008).

- Volatile registers (caller-save) are temporary registers and do not retain their values across function calls. Volatile registers are registers R3-R12, R3 and R4 are used for returning values to the caller function. R5-R12 are used to pass parameters.
- Non-volatile registers keep their values across function calls (callee-save). Non-volatile register are registers R19-R31.
- Dedicated registers are the other registers. Registers R14-R17 are used to store return addresses from interrupts, sub-routines, traps and exceptions. R0 is always value 0 and R1 is used to store the stack pointer. These register should not be used for anything else.

Special purpose registers; there are five special purpose registers (Xilinx Inc., 2008).

- Program counters (PC) – A read-only register containing the address of the executing instruction.
- Machine Status register (MSR) – The MSR register holds control and status bits for the processor. In the MSR it is possible to enable/disable interrupts, exceptions and data and instruction cache. It also contains bits for errors such as division by zero and FSL errors.
- Exception Address Register (EAR) – Stores the full address that caused the exception.
- Exception Status Register (ESR) – Contains exception status bits for the processor.
- Branch Target Register (BTR) – It only exists if the MicroBlaze processor is configured to use exceptions. The register stores the branch target address for all delay slot branch instructions.
- Floating Point Status Register (FSR) – Contains status bits for the floating point unit.
- Exception Data Register (EDR) – It stores data read on an FSL link that caused an FSL exception.
- Process Identifier Register (PIR) – It is used to uniquely identify a software process during MMU address translation. It is controlled by the C_USE_MMU configuration option on MicroBlaze.
- Zone Protection Register (ZPR) – It is used to override MMU memory protection defined in Translation Look-Aside Buffer entries.
- Translation Look-Aside Registers – It is used to access MMU Unified Translation Look-Aside Buffer (UTLB) entries.
- Translation Look-Aside Buffer Search Index Register – It is used to search for a virtual page number in the Unified Translation Look-Aside Buffer.
- Processor Version Register – It is controlled by the C_PVR configuration option on MicroBlaze and used to detect processor version.

2.4.1.2 Bus Interfaces

MicroBlaze is implemented with Harvard memory architecture; instruction and data accesses are done in separate address spaces. Each address space has a 32-bit range (that is, handles up to 4-GB of instructions and data memory respectively). The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory. The latter is useful for software debugging (Xilinx Inc., 2008).

Both instruction and data interfaces of MicroBlaze are 32 bits wide and use big endian, bit-reversed format. MicroBlaze supports word, halfword, and byte accesses to data memory.

MicroBlaze does not separate data accesses to I/O and memory (it uses memory mapped I/O). The processor has up to three interfaces for memory accesses:

- Local Memory Bus (LMB)
- Processor Local Bus (PLB) or On-Chip Peripheral Bus (OPB)
- Xilinx CacheLink (XCL)

The LMB memory address range must not overlap with PLB, OPB or XCL ranges.

2.4.1.2.1 Local Memory Bus (LMB) The LMB is a synchronous bus used primarily to access on-chip block RAM. It uses a minimum number of control signals and a simple protocol to ensure that local block RAM are accessed in a single clock cycle. All LMB signals are active high (Xilinx Inc., 2008).

2.4.1.2.2 Processor Local Bus (PLB) The PLB is one element of the IBM CoreConnect architecture, and is a high-performance synchronous bus designed for connection of processors to high-performance peripheral devices. The PLB includes the following features (from 64-bit Processor Local Bus, Architecture Specifications):

- Overlapping of read and write transfers allow two data transfers per clock cycle for maximum bus utilization.
- Decoupled address and data buses support split-bus transaction capability for improved bandwidth.
- Address pipelining reduces overall bus latency by allowing the latency associated with a new request to be overlapped with an ongoing data transfer in the same direction.
- Late master request abort capability reduces latency associated with aborted requests.
- Hidden (overlapped) bus request/grant protocol reduces arbitration latency.
- Bus architecture supports sixteen masters and any number of slave devices.
- Four levels of request priority for each master allow PLB implementations with various arbitration schemes.
- Bus arbitration-locking mechanism allows for master-driven atomic operations.
- Support for 16-, 32-, and 64-byte line data transfers.
- Read word address capability allows slave devices to fetch line data in any order (that is, target word-first or sequential).
- Sequential burst protocol allows byte, halfword, and word burst data transfers in either direction.
- Guarded and unguarded memory transfers allow a slave device to enable or disable the pre-fetching of instructions or data.

The PLB is a full-featured bus architecture with many features that increase bus performance. Most of these features map well to the FPGA architecture, however, some can result in the inefficient use of FPGA resources or can lower system clock rates (Xilinx Inc., 2005).

2.4.1.2.3 On-Chip Peripheral Bus (OPB) The OPB is one element of the IBM CoreConnect architecture, and is a general-purpose synchronous bus designed for easy connection of on-chip peripheral devices. The OPB includes the following features:

- 32-bit or 64-bit data bus
- Up to 64-bit address
- Supports 8-bit, 16-bit, 32-bit, and 64-bit slaves
- Supports 32-bit and 64-bit masters
- Dynamic bus sizing with byte, halfword, fullword, and doubleword transfers
- Optional Byte Enable support
- Distributed multiplexer bus instead of 3-state drivers
- Single cycle transfers between OPB master and OPB slaves (not including arbitration)
- Support for sequential address protocol
- 16-cycle bus time-out (provided by arbiter)
- Slave time-out suppress capability
- Support for multiple OPB bus masters
- Support for bus parking
- Support for bus locking
- Support for slave-requested retry
- Bus arbitration overlapped with last cycle of bus transfers

The OPB is a full-featured bus architecture with many features that increase bus performance. However, some features can result in the inefficient use of FPGA resources or can lower system clock rates. Consequently, Xilinx uses an efficient subset of the OPB for Xilinx-developed OPB devices (Xilinx Inc., 2005).

2.4.1.2.4 Xilinx Cache Link (XCL) Xilinx CacheLink (XCL) is a high performance solution for external memory accesses. The MicroBlaze CacheLink interface is designed to connect directly to a memory controller with integrated FSL (Fast Simplex Link bus provides a point-to-point communication channel between an output FIFO and an input FIFO) buffers , for example, the MPMC. This method has the lowest latency and minimal number of instantiations.

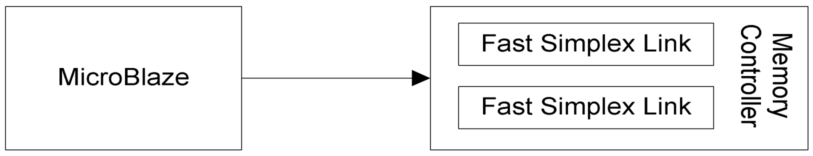


Figure 2.3 CacheLink Connections with Integrated FSL Buffers (Xilinx Inc., 2008)

The interface is only available on MicroBlaze when caches are enabled. It is legal to use a CacheLink cache on the instruction side or the data side without caching the other.

How memory locations are accessed depend on the parameter C_ICACHE_ALWAYS_USED for the instruction cache and the parameter C_DCACHE_ALWAYS_USED for the data cache. If the parameter is 1, the cached memory range is always accessed via the CacheLink. If the parameter is 0, the cached memory range is accessed over PLB or OPB whenever the caches are software disabled (that is, MSR[DCE]=0 or MSR[ICE]=0).

Memory locations outside the cacheable range are accessed over PLB, OPB or LMB (Xilinx Inc., 2008).

2.4.1.2.5 Fast Simplex Link (FSL) MicroBlaze can be configured with up to 16 Fast Simplex Link (FSL) interfaces, each consisting of one input and one output port. The FSL channels are dedicated uni-directional point-to-point data streaming interfaces. The FSL interfaces on MicroBlaze are 32 bits wide. A separate bit indicates whether the sent/received word is of control or data type. Each FSL provides a low latency dedicated interface to the processor pipeline. Thus they are ideal for extending the processors execution unit with custom hardware accelerators (Xilinx Inc., 2008).

CHAPTER THREE
ADVANCED ENCRYPTION STANDARD (AES)

Cryptographic techniques are typically divided into two generic types: symmetric-key and public-key. Symmetric algorithms, sometimes called conventional algorithms, are algorithms where the encryption key can be calculated from the decryption key and vice versa. In most symmetric algorithms, the encryption key and the decryption key are the same. These algorithms, also called secret-key algorithms, single-key algorithms, or one-key algorithms, require that the sender and receiver agree on a key before they can communicate securely. The security of a symmetric algorithm rests in the key; divulging the key means that anyone could encrypt and decrypt messages. As long as the communication needs to remain secret, the key must remain secret.

Symmetric algorithms can be divided into two categories. Some operate on the plaintext a single bit (or sometimes byte) at a time; these are called stream algorithms or stream ciphers. Others operate on the plaintext in groups of bits. The groups of bits are called blocks, and the algorithms are called block algorithms or block ciphers. A block cipher is an encryption scheme which breaks up the plaintext messages to be transmitted into strings (called blocks) of a fixed length and encrypts one block at a time (Schneier, 1996).

Not all the primitives (tools) are explained by looking at Figure 1.1, instead the ones that AES depends on are explained in this thesis.

3.1 The Origins of AES

The most widely used encryption scheme is based on the Data Encryption Standard (DES) adopted in 1977 by the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as Federal Information Processing Standard 46 (NIST, 1999). For DES, data are encrypted in 64 bit blocks using a 56 bit key. The algorithm transforms 64-bit input in a series of steps into a 64-bit output. The same steps, with the same key, are used to reverse the encryption.

In 1999, NIST issued a new version of its DES standard that indicated that DES should only be used for legacy systems and that triple DES (3DES) (NIST, 2008) be used instead. 3DES has two attractions that assure its widespread use over the next few years. First, with its 168-bit key length, it overcomes the vulnerability to brute-force attack of DES. Second, the underlying encryption algorithm in 3DES is the same as in DES. If security were the only consideration, then 3DES would be an appropriate choice for a standardized encryption algorithm for decades to come.

The principal drawback of 3DES is that the algorithm is relatively sluggish in software. The original DES was designed for mid-1970s hardware implementation and does not produce efficient software code. 3DES, which has three times as many rounds as DES, is correspondingly slower. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable.

Because of these drawbacks, 3DES is not a reasonable candidate for long-term use. As a replacement, NIST in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have security strength equal to or better than 3DES and significantly, improved efficiency. In addition to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits.

In a first round of evaluation, 15 proposed algorithms were accepted. A second round narrowed the field to 5 algorithms. NIST completed its evaluation process and published a final standard in November of 2001. NIST selected Rijndael as the proposed AES algorithm. The two researchers who developed and submitted Rijndael for the AES are both cryptographers from Belgium: Dr. Joan Daemen and Dr. Vincent Rijmen (Stallings, 2005).

3.2 Notations and Mathematical Preliminaries

The following parts are mainly derived from (NIST, 2001), (Galbreath, 2002) and (Zabala, 2004). Parts contain the conventions, mathematical preliminaries and overall architecture AES uses.

3.2.1 Inputs and Outputs

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128, 192 or 256 bits. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number "i" attached to a bit is known as its index and will be in one of the ranges $0 \leq i < 128$, $0 \leq i < 192$ or $0 \leq i < 256$ depending on the block length and key length (specified above).

3.2.2 Bytes

The basic unit for processing in the AES algorithm is a byte, a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described in Sec. 3.2.1 are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes (see Sec. 3.2.3). For an input, output or Cipher Key denoted by a, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in one of the following ranges:

Key length = 128 bits, $0 \leq n < 16$; Block length = 128 bits, $0 \leq n < 16$;

Key length = 192 bits, $0 \leq n < 24$;

Key length = 256 bits, $0 \leq n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order {b7, b6, b5, b4, b3, b2, b1, b0}. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0 = \sum_{i=0}^{7} b_i x^i \qquad \text{Eq 3.1}$$

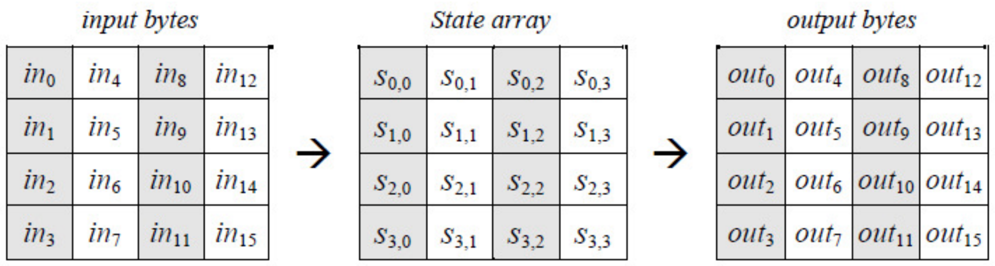
For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$. Some finite field operations involve one additional bit (b8) to the left of an 8-bit byte. Where this extra bit is present, it will appear as '{01}' immediately preceding the 8-bit byte; for example, a 9-bit sequence will be presented as {01} {1b}.

3.2.3 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing *Nb* bytes, where *Nb* is the block length divided by 32.

In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number *r* in the range $0 \leq r < 4$ and its column number *c* in the range $0 \leq c < Nb$. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or *s[r,c]*. For this standard, *Nb*=4, i.e., $0 \leq c < 4$.

At the start of the Cipher and Inverse Cipher described in Sec. 5, the input – the array of bytes in0, in1 … in15 – is copied into the State array as illustrated in Figure 3.1. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes out0, out1 … out15.



input bytes
State array
output bytes
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
out_0 out_4 out_8 out_{12}
out_1 out_5 out_9 out_{13}
out_2 out_6 out_{10} out_{14}
out_3 out_7 out_{11} out_{15}

Figure 3.1 State array input & output.

So at the beginning of the Cipher or Inverse Cipher, the input array, in, is copied to the State array according to the scheme:

$$s[r, c] = in[r + 4c] \text{ for } 0 \le r < 4 \text{ and } 0 \le c < Nb, \qquad \text{Eq3.2}$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array out as follows:

$$out[r + 4c] = s[r, c] \text{ for } 0 \le r < 4 \text{ and } 0 \le c < Nb. \qquad \text{Eq3.3}$$

3.2.3.1 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit words, where the row number *r* provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), *w0...w3*, where the column number *c* provides an index into this array. For the example in Figure 3.3, the State can be considered as an array of four words, as follows:

$$w_0 = s_{0,0} + s_{1,0} + s_{2,0} + s_{3,0} \qquad w_1 = s_{0,1} + s_{1,1} + s_{2,1} + s_{3,1} \qquad \text{Eq3.4}$$

$$w_2 = s_{0,2} + s_{1,2} + s_{2,2} + s_{3,2} \qquad w_3 = s_{0,3} + s_{1,3} + s_{2,3} + s_{3,3} \qquad \text{Eq3.5}$$

3.2.4. Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in Sec. 3.2.2 Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts.

3.2.4.1 Addition

The addition of two elements in a finite field is achieved by “adding” the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$) - i.e., modulo 2 - so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$. Consequently, subtraction of polynomials is identical to addition of polynomials. Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ + $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$ = $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e, $c_7 = a_7 \oplus b_7, c_6 = a_6 \oplus b_6, \ldots c_0 = a_0 \oplus b_0$). For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);

$\{01010111\} \oplus \{10000011\} = \{11010100\}$ (binary notation);

$\{57\} \oplus \{83\} = \{d4\}$ (hexadecimal notation).

3.2.4.2 Multiplication

In the polynomial representation, multiplication in GF (2^8) (denoted by ●) corresponds with the multiplication of polynomials modulo an irreducible polynomial of degree 8. A polynomial is irreducible if its only divisors are one and itself. For the AES algorithm, this irreducible polynomial is $m(x) = x^8 + x^4 + x^3 + x + 1$ or {01}{1b} in hexadecimal notation.

For example, {57} ● {83} = {c1}, because the resultant polynomial is modulo of *m(x)* and appears as: $x^7 + x^6 + 1$.

The modular reduction by *m(x)* ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

3.2.4.2.1 Multiplication by x Multiplying the binary polynomial defined in equation (3.1) with the polynomial x results in

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x^1 \qquad \text{Eq3.6}$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo *m(x)*, irreducible polynomial. If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial *m(x)*. It follows that multiplication by x (i.e., {00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by *xtime()*. Multiplication by higher powers of x can be implemented by repeated application of *xtime()*. By adding intermediate results, multiplication by any constant can be implemented.

For example, {57} ● {13} = {fe} because

{57} ● {02} = *xtime*({57}) = {ae}

{57} ● {04} = *xtime*({ae}) = {47}

{57} ● {08} = *xtime*({47}) = {8e}

{57} ● {10} = *xtime*({8e}) = {07},

thus,

$$\begin{aligned}\{57\} \bullet \{13\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\ &= \{57\} \oplus \{ae\} \oplus \{07\} \\ &= \{fe\}.\end{aligned}$$

3.2.4.3 Polynomials with Coefficients in GF (2^8)

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 \qquad \text{Eq3.7}$$

which will be denoted as a word in the form [a_0, a_1, a_2, a_3]. Note that the polynomials in this section behave somewhat different than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, *x*. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below. The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0 \qquad \text{Eq3.8}$$

define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of *x*. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Multiplication is achieved in two steps. In the first step, the polynomial product *c(x) = a(x)* ● *b(x)* is algebraically expanded, and like powers is collected to give:

$$c(x) = a(x) + b(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \qquad \text{Eq3.9}$$

The result, *c(x)*, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce *c(x)* modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than 4. For the AES algorithm, this is accomplished with the polynomial $x^4 + 1$, so that

$$x^i \bmod(x^4 + 1) = x^{i \bmod(4)} \qquad \text{Eq3.10}$$

3.3 Algorithm Specification

For the AES algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by *Nb* = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm, the length of the Cipher Key, K, is 128, 192, or 256 bits. The key length is represented by *Nk* = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by *Nr*, where *Nr* =10 when *Nk* = 4, *Nr* = 12 when *Nk* = 6, and *Nr* = 14 when *Nk* = 8.

The only Key-Block-Round combinations that conform to this standard are given in Table 3.1.

Table 3.1 AES Parameters

Key Size (Words/Bytes/Bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (Words/Bytes/Bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (Words/Bytes/Bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (Words/Bytes)	44/176	52/208	60/240

Figure 3.2 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit block. In (NIST, 2001), this block is depicted as a square matrix of bytes. This block is copied into the State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix. These operations are depicted in Figure 3.2 (a).

Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is four bytes and the total key schedule is 44 words for the 128-bit key (Figure 3.2 (b)). Note that the ordering of bytes within a matrix is by column.

So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the in matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the w matrix.

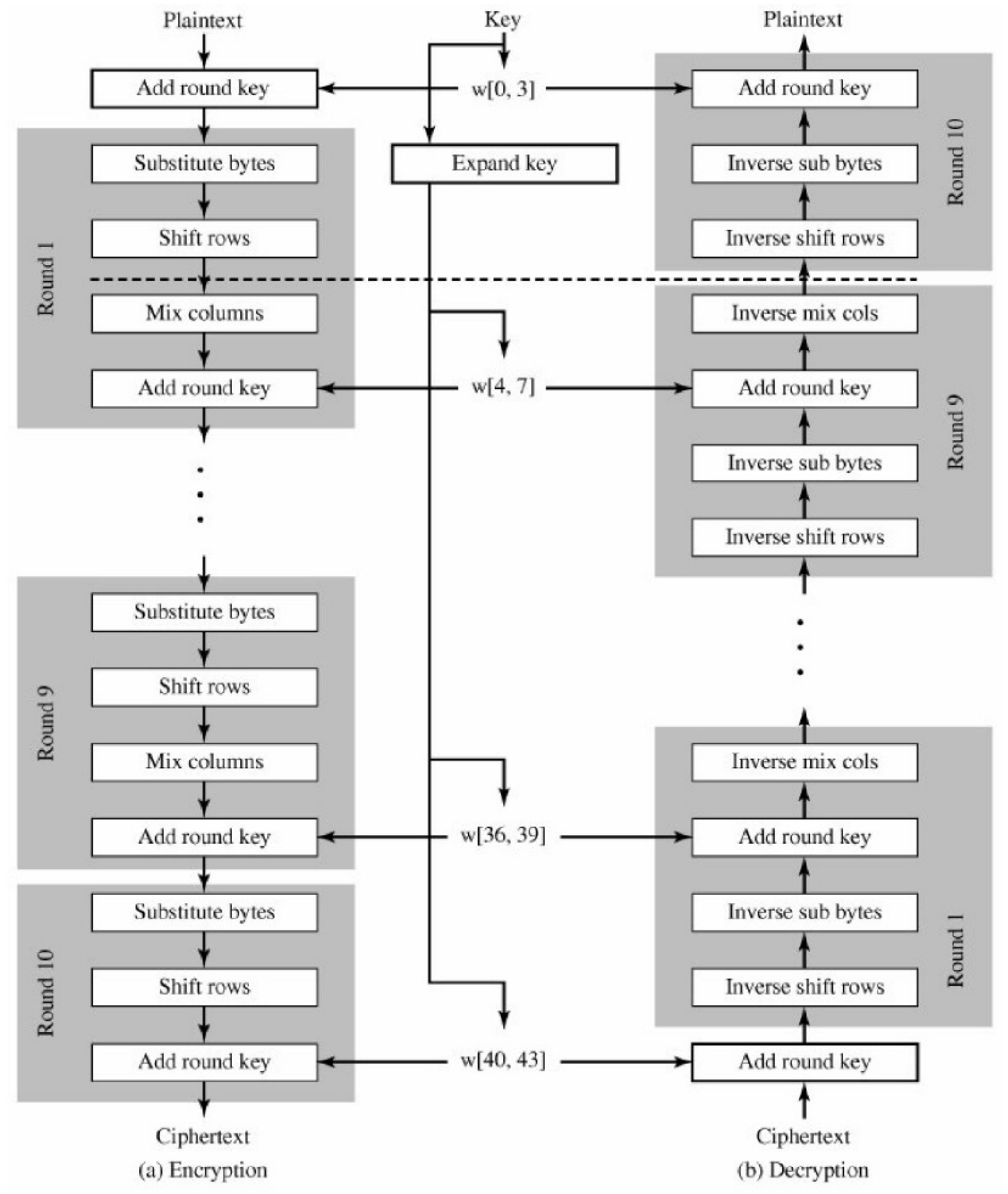


Figure 3.2 AES Encryption (a) and Decryption (b), Overall Structure (Stallings, 2005)

For both its Cipher (Encryption) and Inverse Cipher (Decryption), the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

1. **Substitute bytes**: Uses an S-box to perform a byte-by-byte substitution of the block.
2. **ShiftRows**: A simple permutation.

3. **MixColumns**: A substitution that makes use of arithmetic over GF (2^8).
4. **AddRoundKey**: A simple bitwise XOR of the current block with a portion of the expanded key.

3.3.1 The Cipher (Encryption)

At the start of the Cipher, the input is copied to the State array using the conventions described in Section 3.2. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length - being 128, 192 or 256 bits), with the final round differing slightly from the first *Nr -1* rounds. The final State is then copied to the output as described in Sec. 3.2.

The Cipher is described in the pseudo code in Figure 3.3. The individual transformations - SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() – process the State and are described in the following subsections.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 3.3 Pseudo code for cipher (NIST, 2001). The various transformations (e.g., SubBytes(), ShiftRows(), etc.) act upon the State array that is addressed by the 'state' pointer. AddRoundKey() uses an additional pointer (w[ ]) to address the Round Key.

3.3.1.1 SubBytes Transformation

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box, which is invertible, is constructed by composing two transformations:

- Take the multiplicative inverse in the finite field GF (2^8), the element {00} is mapped to itself.
- Apply the following affine transformation (over GF(2)):

  $$b'_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i \qquad \text{Eq3.11}$$

  for $0 \le i < 8$, where b_i is the i^{th} bit of the byte, and c_i is the i^{th} bit of a byte c with the value {63} or {01100011}. Here and elsewhere, a prime on a variable (e.g., *b`)* indicates that the variable is to be updated with the value on the right.

In matrix form, affine transformation element of the S-box can be expressed as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 3.4 shows the effect of the SubBytes() transformation on the State. AES defines a 16 x 16 matrix of byte values, called an S-box (Figure 3.5) that contains a permutation of all possible 256 8-bit values. Each individual byte of State is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value. For example, the hexadecimal value {95} references row 9, column 5 of the S-box, whcich contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

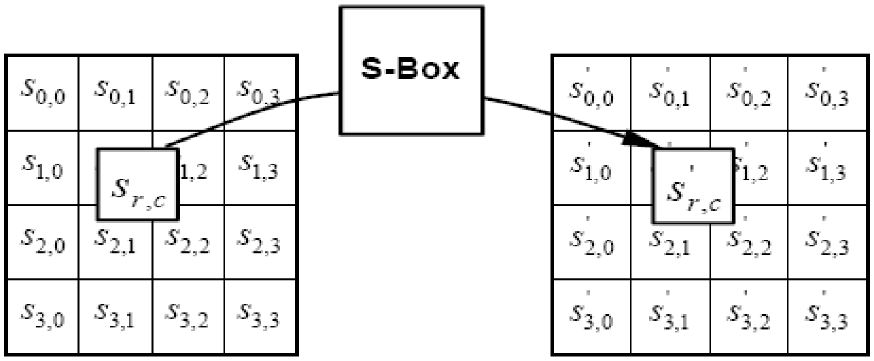

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{r,c}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{r,c}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.4 SubBytes() applies the S-box to each byte of the State. (NIST, 2001)

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.5: S-box: substitution values for the byte xy (in hexadecimal format). (NIST, 2001)

3.3.1.2 ShiftRows Transformation

The ShiftRow operation is depicted in Figure 3.6. The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. Figure 3.7 shows an example of ShiftRows.

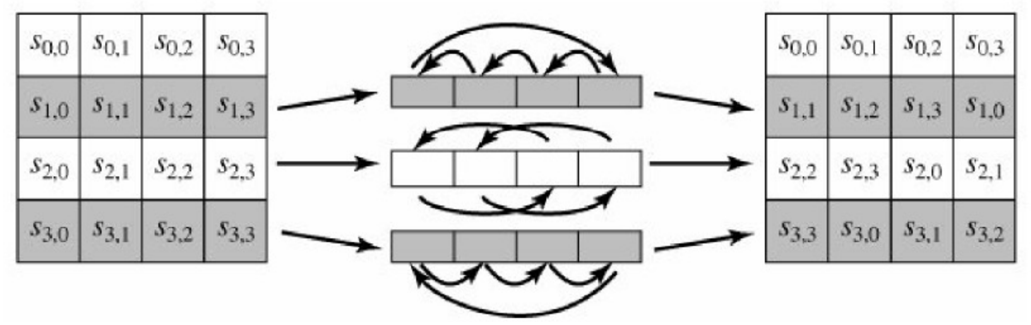
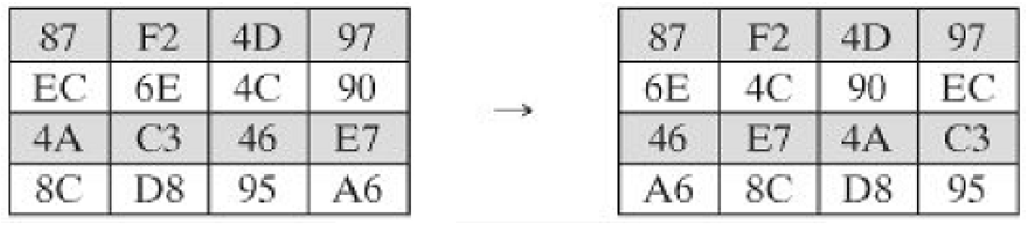


Figure 3.6 ShiftRow transformation



87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

→

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

Figure 3.7 Example of ShiftRow transformations

The shift row transformation is more substantial than it may first appear. This is because the State, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of State, and so on. Further, as will be seen, the round key is applied to State column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Also note that the transformation ensures that the 4 bytes of one column are spread out to four different columns (Stallings, 2005).

3.3.1.3 MixColumns Transformation

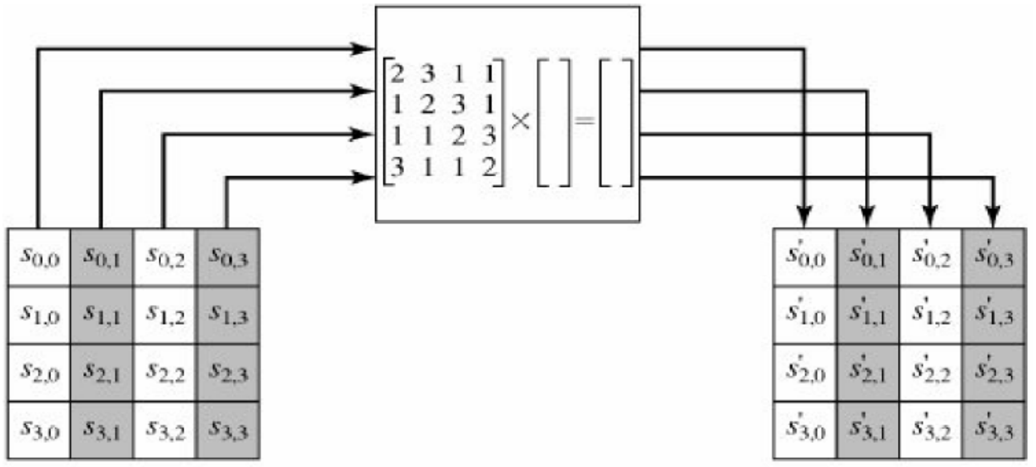
The MixColumns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The columns are considered as polynomials over GF (2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial *a(x)*, given by:

$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ Eq3.12

Eq3.12 can be written as a matrix multiplication, let $s'(x) = a(x) \oplus s(x)$;

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb.$$

Figure 3.8 depicts MixColumn transformation.


2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
×
=
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$
Figure 3.9 MixColumn transformation (Stallings, 2005)

The coefficients of the matrix above are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits.

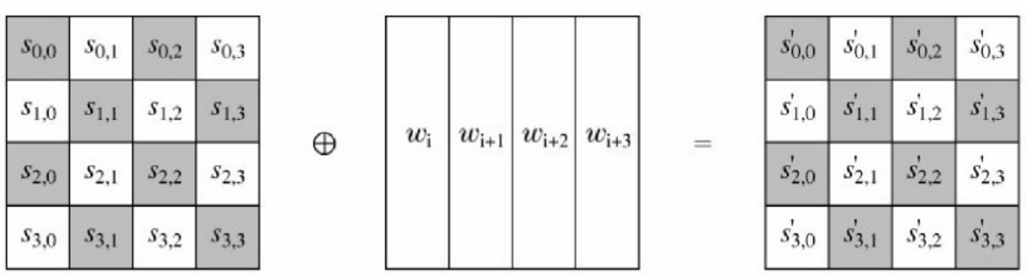
In addition, the choice of coefficients in MixColumns, which are all {01}, {02}, or {03}, was influenced by implementation considerations. As was discussed, multiplication by these coefficients involves at most a shift and an XOR. The coefficients in InvMixColumns are more formidable to implement. However, encryption was deemed more important than decryption for two reasons:

1. For the CFB and OFB cipher modes (described in Chapter 4), only encryption is used.

2. As with any block cipher, AES can be used to construct a message authentication code, and for this only encryption is used (Stallings, 2005).

3.3.1.4 AddRoundKey Transformation

In the AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. As shown in Figure 3.8, the operation is viewed as a columnwise operation between the 4 bytes of a State column and one word of the round key; it can also be viewed as a byte-level operation.



$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$\oplus$
w_i w_{i+1} w_{i+2} w_{i+3}
=
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.10 AddRoundKey XORs each column of the State with a word from the key schedule (Stallings, 2005)

The AddRoundKey transformation is as simple as possible and affects every bit of State. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensures security.

3.3.2 Key Expansion

The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of $Nb\ (Nr + 1)$ words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [w_i], with i in the range $0 \leq i < Nb(Nr + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Figure 3.9. SubWord() is a function that takes a four-byte input word and applies the S-box (Sec. 3.3.1.1, Figure 3.4) to each of the four bytes to produce an output word. The function RotWord() takes a word [a_0,a_1,a_2,a_3] as input, performs a cyclic permutation, and returns the word [a_1,a_2,a_3,a_0]. The round constant word array, *Rcon[i]*, contains the values given by [x_i-1,{00},{00},{00}], with (x_i – 1) being powers of *x* (*x* is denoted as {02}) in the field GF(2^8) (note that *i* starts at 1, not 0).

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 3.11 Pseudo code for AES Key Expansion

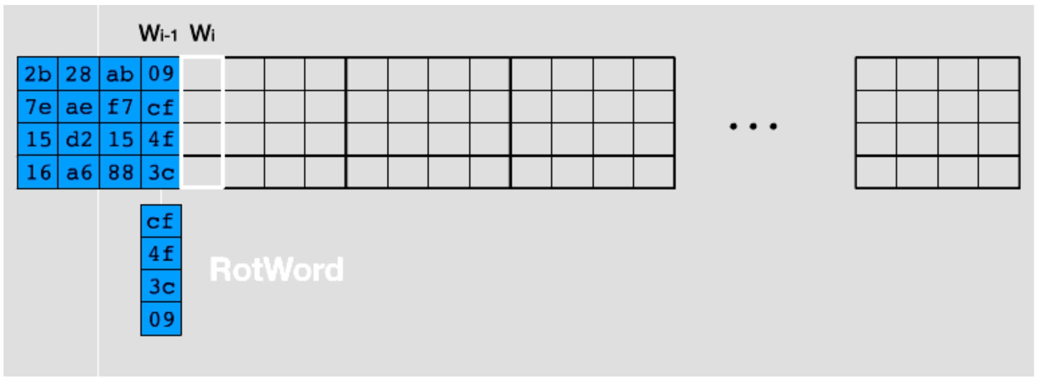
From Figure 3.11, it can be seen that the first *Nk* words of the expanded key are filled with the Cipher Key. Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word *Nk* positions earlier, w[i-*Nk*]. For words in positions that are a multiple of *Nk*, a transformation is applied to w[i-1] prior to the XOR, followed by an XOR with a round constant, Rcon[i]. This transformation consists of a cyclic shift of the bytes in a word (RotWord()), followed by the application of a table lookup to all four bytes of the word (SubWord()).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys (*Nk* = 8) is slightly different than for 128- and 192-bit Cipher Keys. If *Nk* = 8 and *i*-4 is a multiple of *Nk*, then SubWord() is applied to w[*i*-1] prior to the XOR.

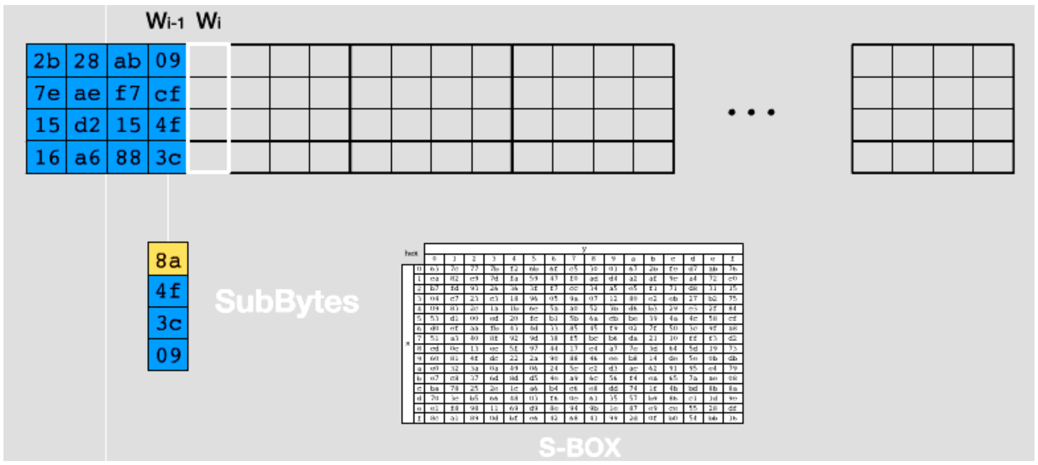
The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as Rcon[j] = (RC[j], 0, 0, 0), with RC[1] = 1, RC[j] = 2 · RC[j - 1] and with multiplication defined over the field GF(2^8) (Table3.1). From Figure 3.12: (a) to (f) AES key expansion is illustrated in graphical form.

Table 3.2 RCon Values

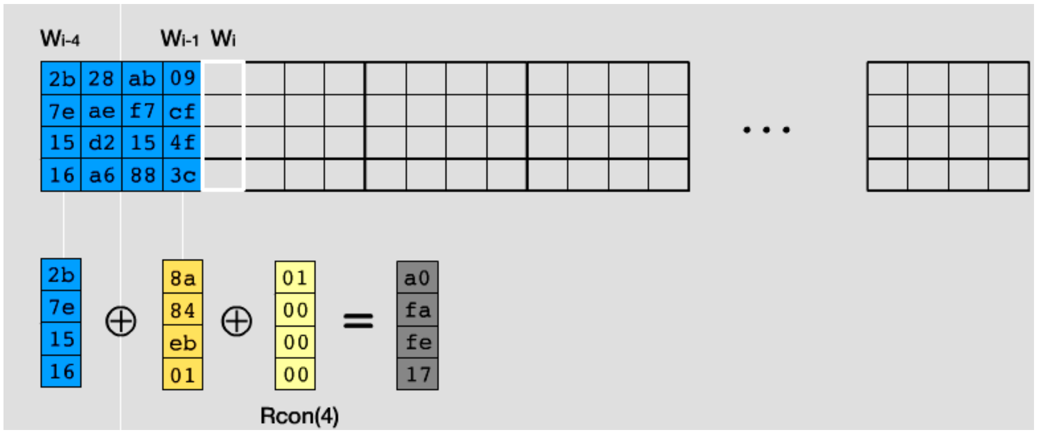
j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36



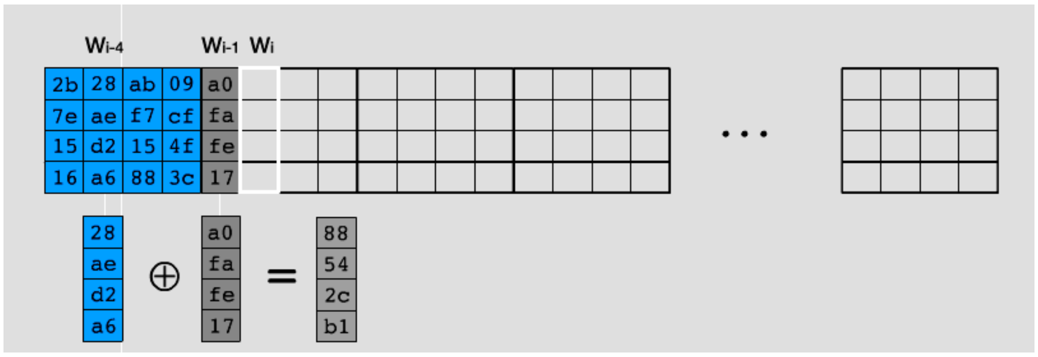
(a)



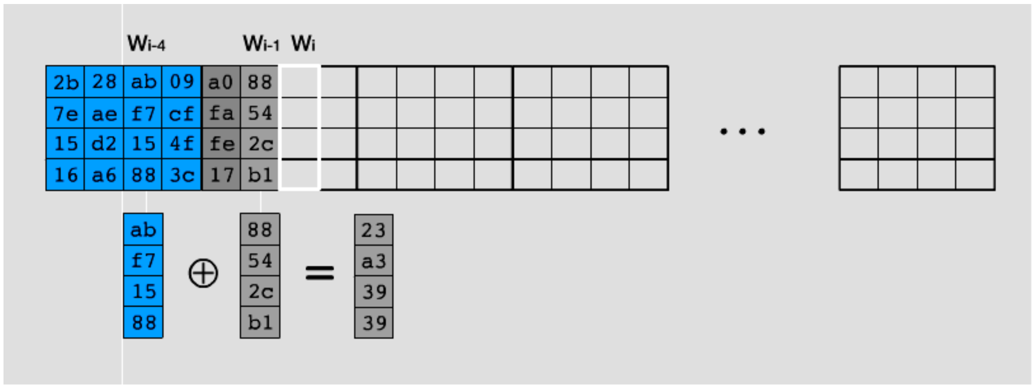
(b)


Wi-4
Wi-1
Wi
2b 28 ab 09
7e ae f7 cf
15 d2 15 4f
16 a6 88 3c
2b 7e 15 16
8a 84 eb 01
01 00 00 00
a0 fa fe 17
Rcon(4)

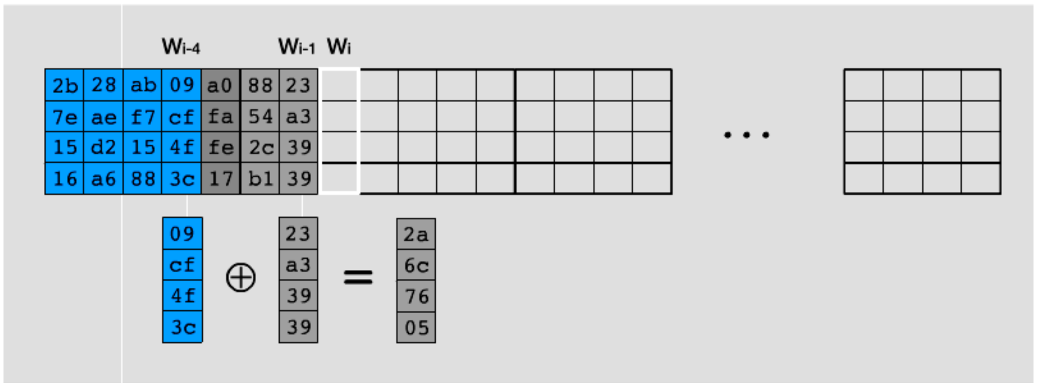
(c)


Wi-4
Wi-1
Wi
2b 28 ab 09 a0
7e ae f7 cf fa
15 d2 15 4f fe
16 a6 88 3c 17
28 ae d2 a6
a0 fa fe 17
88 54 2c b1

(d)


Wi-4
Wi-1
Wi
2b 28 ab 09 a0 88
7e ae f7 cf fa 54
15 d2 15 4f fe 2c
16 a6 88 3c 17 b1
ab f7 15 88
88 54 2c b1
23 a3 39 39

(e)



(f)

Figure 3.12 AES Key Expansion in graphical form, RotWord operation (a), SubWord operation (b), For words in positions that are a multiple of *Nk(=4)*, a transformation is applied to w[i-1] prior to the XOR, followed by an XOR with a round constant, Rcon[i] (c), Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word Nk positions earlier, w[i-Nk] ((d), (e), (f)).

The Rijndael developers designed the expansion key algorithm to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the ways in which round keys are generated in different rounds. The specific criteria that were used are as follows:

- Knowledge of a part of the cipher key or round key does not enable calculation of many other round key bits

- An invertible transformation [i.e., knowledge of any Nk consecutive words of the Expanded Key enables regeneration the entire expanded key (Nk = key size in words)]

- Speed on a wide range of processors

- Usage of round constants to eliminate symmetries

- Diffusion of cipher key differences into the round keys; that is, each key bit affects many round key bits

- Enough nonlinearity to prohibit the full determination of round key differences from cipher key differences only

- Simplicity of description (Daemen & Rijmen, 2003)

The authors do not quantify the first point on the preceding list, but the idea is that if you know less than *Nk* consecutive words of either the cipher key or one of the round keys, then it is difficult to reconstruct the remaining unknown bits. The fewer bits one knows, the more difficult it is to do the reconstruction or to determine other bits in the key expansion (Stallings, 2005).

3.3.3 The Inverse Cipher (Decryption)

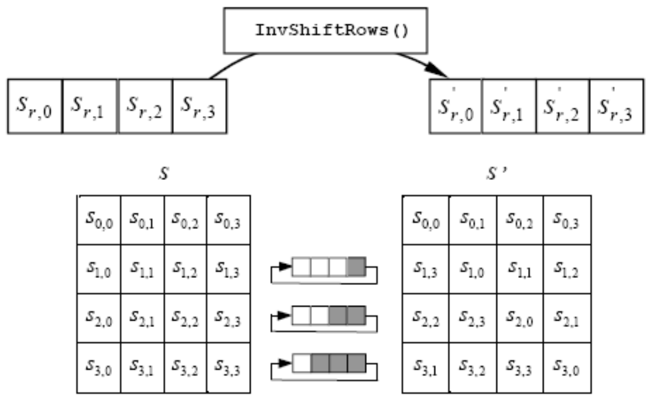
The AES decryption cipher is not identical to the encryption cipher (Figure 3.2). The sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. The decryption algorithm has the same sequence of transformations as the encryption algorithm with transformations replaced by their inverses (Figure 3.13). The individual transformations used in the Inverse Cipher are: InvShiftRows(), InvSubBytes(), InvMixColumns(), and AddRoundKey().

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)
      InvSubBytes(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

Figure 3.13 Pseudo Code for Inverse Cipher (Decryption)

3.3.3.1 InvShiftRows Transformation

InvShiftRows() is the inverse of the ShiftRows() transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, r = 0, is not shifted. The bottom three rows are cyclically shifted by Nb - shift(r, Nb) bytes, where the shift value shift(r,Nb) depends on the row number. InvShiftRows affects the sequence of bytes in State but does not alter byte contents and does not depend on byte contents to perform its transformation. Figure 3.14 illustrates InvShiftRows() transformation.


InvShiftRows()
$s_{r,0}$ $s_{r,1}$ $s_{r,2}$ $s_{r,3}$
$s'_{r,0}$ $s'_{r,1}$ $s'_{r,2}$ $s'_{r,3}$
s
s'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,3}$ $s_{1,0}$ $s_{1,1}$ $s_{1,2}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,1}$ $s_{3,2}$ $s_{3,3}$ $s_{3,0}$
Figure 3.14 InvShiftRows() transformation

3.3.3.2 InvShiftRows Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse Sbox is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (Eq3.11) followed by taking the multiplicative inverse in GF(2^8). InvSubBytes() affects the contents of bytes in State but does not alter byte sequence and does not depend on byte sequence to perform its transformation.

The inverse S-box used in the InvSubBytes() transformation is presented in Figure 3.15.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
x	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 3.15 Inverse S-box substitution values for the byte xy (in hexadecimal format).

3.3.3.3 InvMixColumns Transformation

InvMixColumns() is the inverse of the MixColumns() transformation. InvMixColumns() operates on the State column-by-column, treating each column as a fourterm polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by;

$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$ Eq3.13

This can be written as a matrix multiplication. Let $s'(x) = a^{-1}(x) \oplus s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

3.3.3.4 Inverse of AddRoundKey Transformation

AddRoundKey(), which was described in Section 3.3.1.4, is its own inverse, since it only involves an application of the XOR operation. The transformations InvAddRoundKey and InvMixColumns do not alter the sequence of bytes in State. If we view the key as a sequence of words, then both InvAddRoundKey and InvMixColumns operate on State one column at a time. These two operations are linear with respect to the column input.

3.3.5 Implementation Issues

3.3.5.1 Key Length Requirements

An implementation of the AES algorithm shall support at least one of the three key lengths specified in Section 3.3.1.1: 128, 192, or 256 bits (i.e., $Nk = 4$, 6, or 8, respectively). Implementations may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations.

3.3.5.2 Keying Restrictions

No weak or semi-weak keys have been identified for the AES algorithm, and there is no restriction on key selection.

3.3.5.3 Parameterization of Key Length, Block Size, and Round Number

AES explicitly defines the allowed values for the key length (Nk), block size (Nb), and number of rounds (Nr). However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind.

3.3.5.4 Implementation Aspects

Implementation variations are possible that may, in many cases, offer performance or other advantages. Given the same input key and data (plaintext or ciphertext), any implementation that produces the same output (ciphertext or plaintext) as the algorithm specified in this standard is an acceptable implementation of the AES.

The Rijndael proposal (Stallings, 2005) provides some suggestions for efficient implementation on 8-bit processors, typical for current smart cards, and on 32-bit processors, typical for PCs. MicroBlaze is a 32-bit soft processor and suggestions on (Stallings, 2005) is strictly followed to meet best performance on implementations.

CHAPTER FOUR

BLOCK CIPHER MODES OF OPERATION

A block cipher algorithm is a basic building block for providing data security. To apply a block cipher in a variety of applications, four "modes of operation" have been defined by NIST (Dworkin, 2001). In essence, a mode of operation is a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application, such as applying a block cipher to a sequence of data blocks or a data stream. A cryptographic mode usually combines the basic cipher, some sort of feedback, and some simple operations. The operations are simple because the security is a function of the underlying cipher and not the mode. Even more strongly, the cipher mode should not compromise the security of the underlying algorithm.

There are other security considerations: Patterns in the plaintext should be concealed, input to the cipher should be randomized, manipulation of the plaintext by introducing errors in the ciphertext should be difficult, and encryption of more than one message with the same key should be possible.

Efficiency is another consideration. The mode should not be significantly less efficient than the underlying cipher. In some circumstances it is important that the ciphertext be the same size as the plaintext.

A third consideration is fault-tolerance. Some applications need to parallelize encryption or decryption, while others need to be able to preprocess as much as possible. In still others it is important that the decrypting process be able to recover from bit errors in the ciphertext stream, or dropped or added bits (Schneier, 1996).

A fourth consideration is using an *initialization vector.* Many modes require an initialization block (also known as an initialization vector or salt) to get started. This generally adds security but at a cost for both storage and speed.

The four modes are intended to cover virtually all the possible applications of encryption for which a block cipher could be used. As new applications and requirements have appeared, NIST has expanded the list of recommended modes to five in Special Publication 800-38A. It specifies five confidentiality modes of operation for symmetric key block cipher algorithms, such as the algorithm specified in FIPS Pub. 197, the Advanced Encryption Standard (AES) (NIST, 2001). The modes may be used in conjunction with any symmetric key block cipher algorithm that is approved by a Federal Information Processing Standard (FIPS). The five modes—the Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes—can provide data confidentiality (Dworkin, 2001). The modes are summarized in Table 4.1 at the end of the section and described briefly in the remainder of this section.

4.1 Underlying Block Cipher Algorithm

Special Publication 800-38A (Dworkin, 2001) assumes that a FIPS-approved symmetric key block cipher algorithm has been chosen as the underlying algorithm, and that a secret, random key, denoted K has been established among all of the parties to the communication. The cryptographic key regulates the functioning of the block cipher algorithm and, thus, by extension, regulates the functioning of the mode. The specifications of the block cipher and algorithms and the modes are public, so the security of the mode depends on the secrecy of the key.

A confidentiality mode of operation of the block cipher algorithm consists of two processes that are inverses of each other: *encryption* and *decryption*. Encryption is the transformation of a usable message, called the plaintext, into an unreadable form, called the ciphertext; decryption is the transformation that recovers the plaintext from the ciphertext (Dworkin, 2001).

4.2 Initialization Vectors

Most modes (CBC, CFB, and OFB) require an initialization vector (IV) – that is, a random byte array with the length of the cipher's block. Depending on how the mode works, this vector is used as an alternate start to the message or as a dummy ciphertext block. Following are some guidelines for using IVs: (Galbreath, 2002)

- The same IV must be used in the decryption as was used for the encryption.
- The IV does not have to be generated by a secure random source (although it certainly can be); timestamps or other semi-unique sources can be used.
- The IV is not a key and can be transmitted or stored in the clear.
- The same IV can be used for multiple messages, although for transient messages a different IV should be used.
- The null IV—that is, an IV with all zeros—is commonly used to minimize bookkeeping, storage, or transmission costs, especially in database applications.

The IV need not be secret; however, for the CBC and CFB modes, the IV for any particular execution of the encryption process must be unpredictable, and, for the OFB mode, unique IVs must be used for each execution of the encryption process.

4.3 Electronic Codebook (ECB)

The simplest mode is the electronic codebook (ECB) mode, in which plaintext is handled one block at a time and each block of plaintext is encrypted using the same key (Figure 4.1). The term codebook is used because, for a given key, there is a unique ciphertext for every b–bit block of plaintext. Therefore, we can imagine a gigantic codebook in which there is an entry for every possible b-bit plaintext pattern showing its corresponding ciphertext.

For a message longer than b bits, the procedure is simply to break the message into b-bit blocks, padding the last block if necessary. Decryption is performed one block at a time, always using the same key. In Figure 4.1, the plaintext consists of a sequence of b-bit blocks, P_1, P_2,..., PN; the corresponding sequence of ciphertext blocks is C_1, C_2,..., C_N.

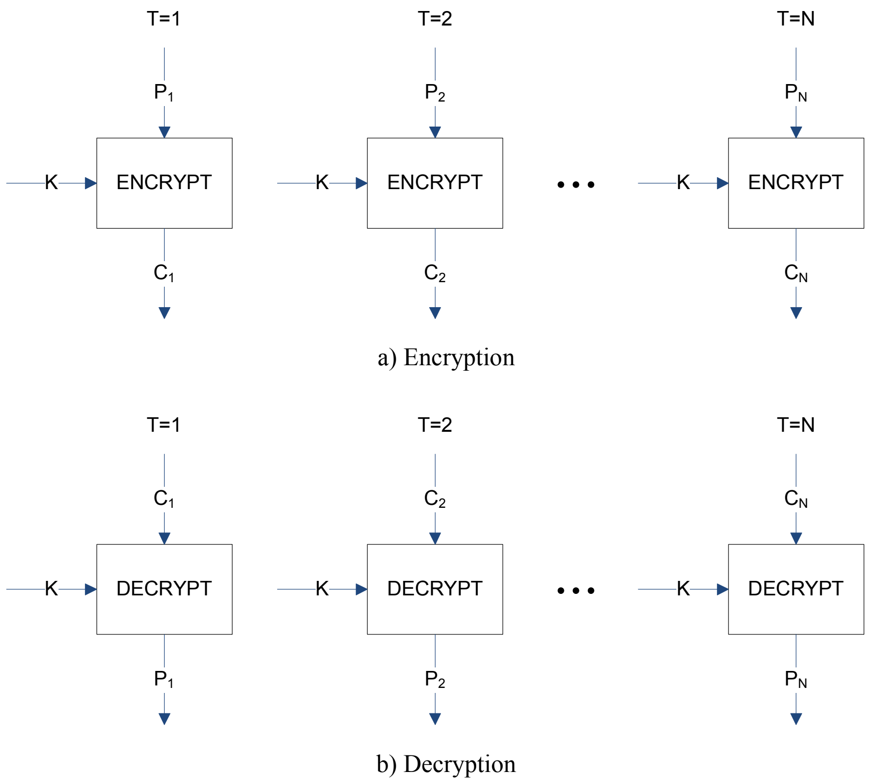


Figure 4.1 Electronic Codebook (ECB) Mode; a) Encryption, b) Decryption

The most significant characteristic of ECB is that the same b-bit block of plaintext, if it appears more than once in the message, always produces the same ciphertext. For lengthy messages, the ECB mode may not be secure. If the message is highly structured, it may be possible for a cryptanalyst to exploit these regularities. For example, if it is known that the message always starts out with certain predefined fields, then the cryptanalyst may have a number of known plaintext-ciphertext pairs to work with. If the message has repetitive elements, with a period of repetition a multiple of b bits, then these elements can be identified by the analyst. This may help in the analysis or may provide an opportunity for substituting or rearranging blocks. This is a fundamental problem of ECB mode; to overcome this problem encryption key should be changed regularly and fast. The solution is a technique called chaining and applied to CBC mode described in section 4.3.

AES block length is constant, 128 bits, so the code book will have 2^{128} entries—much too large to precompute and store. Also every key has a different code book.

In ECB mode, each plaintext block is encrypted independently. It is not mandatory to encrypt a file linearly; encryption could be performed the 10 blocks in the middle first, then the blocks at the end, and finally the blocks in the beginning. This is important for encrypted files that are accessed randomly, like a database. If a database is encrypted with ECB mode, then any record can be added, deleted, encrypted, or decrypted independently of any other record—assuming that a record consists of a discrete number of encryption blocks (Schneier, 1996).

Processing is parallizeable; if multiple encryption processors are valid, they can encrypt or decrypt different blocks without regard for each other.

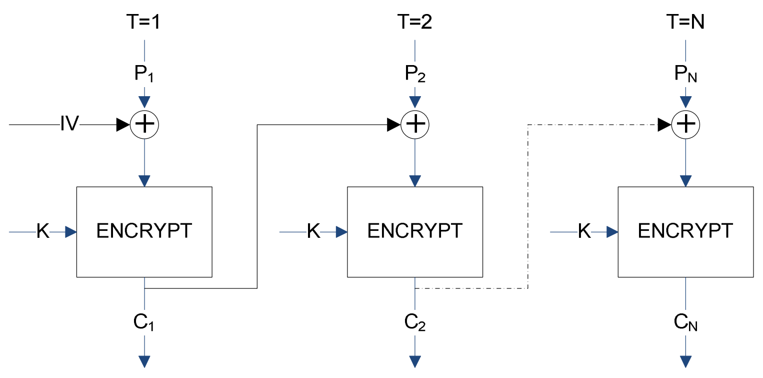
4.4 Cipher Block Chaining (CBC)

CBC mode eliminates security deficiencies of ECB (the dictionary attack) by using the contents of the previous block to encrypt the current block. This extra overhead adds about 20 to 30 percent to the running time over ECB mode. During encryption, each plaintext block is XORed with the previous ciphertext block, and the IV is used as the first ciphertext block (Figure 4.2). In this way the last block depends on all blocks previous to it. The input to the encryption function for each plaintext block bears no fixed relationship to the plaintext block. Therefore, repeating patterns of *b* bits are not exposed.

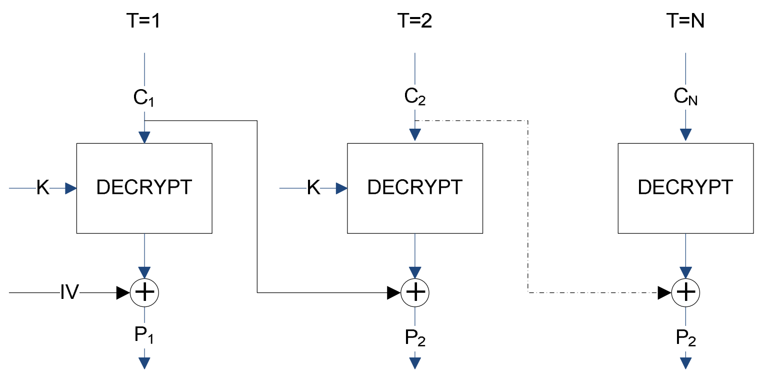
Chaining adds a feedback mechanism to a block cipher: The results of the encryption of previous blocks are fed back into the encryption of the current block. Each ciphertext block is dependent not just on the plaintext block that generated it but on all the previous plaintext blocks.

In CBC encryption, the first input block is formed by XORing the first block of the plaintext with the IV. The forward cipher function is applied to the first input block, and the resulting output block is the first block of the ciphertext. This output block is also XORed with the second plaintext data block to produce the second input block, and the forward cipher function is applied to produce the second output block. This output block, which is the second ciphertext block, is XORed with the next plaintext block to form the next input block. Each successive plaintext block is XORed with the previous output/ciphertext block to produce the new input block. The forward cipher function is applied to each input block to produce the ciphertext block.

In CBC decryption, the inverse cipher function is applied to the first ciphertext block, and the resulting output block is XORed with the initialization vector to recover the first plaintext block. The inverse cipher function is also applied to the second ciphertext block, and the resulting output block is XORed with the first ciphertext block to recover the second plaintext block. In general, to recover any plaintext block (except the first), the inverse cipher function is applied to the corresponding ciphertext block, and the resulting block is XORed with the previous ciphertext block. Decryption is similar to encryption, but each block only depends on the previous block, not all previous blocks. This way decryption can be done in parallel, possibly making it much faster than encryption.



a) Encryption



a) Decryption

Figure 4.2 Cipher Block Chaining (CBC) Mode; a) Encryption, b) Decryption

CBC mode has some interesting error properties. First, it is self-synchronizing: A dropped block or a block with errors will result in only two plaintext blocks being corrupted but this doesn't extend to bits. If a bit is dropped, the remaining message will be completely corrupted. The other property is that a single bit error will corrupt the current block but will only change the corresponding bit in the next block; thereafter the message is intact. Finally, The IV must be known to both the sender and receiver but be unpredictable by a third party. For maximum security, the IV should be protected against unauthorized changes. This could be done by sending the IV using ECB encryption. One reason for protecting the IV is as follows: If an opponent is able to fool the receiver into using a different value for IV, then the opponent is able to invert selected bits in the first block of plaintext (Schneier, 1996).

In conclusion, because of the chaining mechanism of CBC, it is an appropriate mode for encrypting messages of length greater than b bits. In addition to its use to achieve confidentiality, the CBC mode can be used for authentication.

4.5 Cipher Feedback (CFB)

The AES scheme is essentially a block cipher technique that uses b-bit blocks. However, it is possible to convert AES into a stream cipher, using either the cipher feedback (CFB) or the output feedback mode. A stream cipher eliminates the need to pad a message to be an integral number of blocks. It also can operate in real time. Thus, if a character stream is being transmitted, each character can be encrypted and transmitted immediately using a character-oriented stream cipher.

One desirable property of a stream cipher is that the ciphertext be of the same length as the plaintext. Thus, if 8-bit characters are being transmitted, each character should be encrypted to produce a cipher text output of 8 bits. If more than 8 bits are produced, transmission capacity is wasted.

Figure 4.3 depicts the CFB scheme. In the figure, it is assumed that the unit of transmission is s bits; a common value is $s = 8$. The value of s is sometimes incorporated into the name of the mode, e.g., the 1-bit CFB mode, the 8-bit CFB mode, the 64-bit CFB mode, or the 128-bit CFB mode. As with CBC, the units of plaintext are chained together, so that the ciphertext of any plaintext unit is a function of all the preceding plaintext. In this case, rather than units of b bits, the plaintext is divided into segments of s bits.

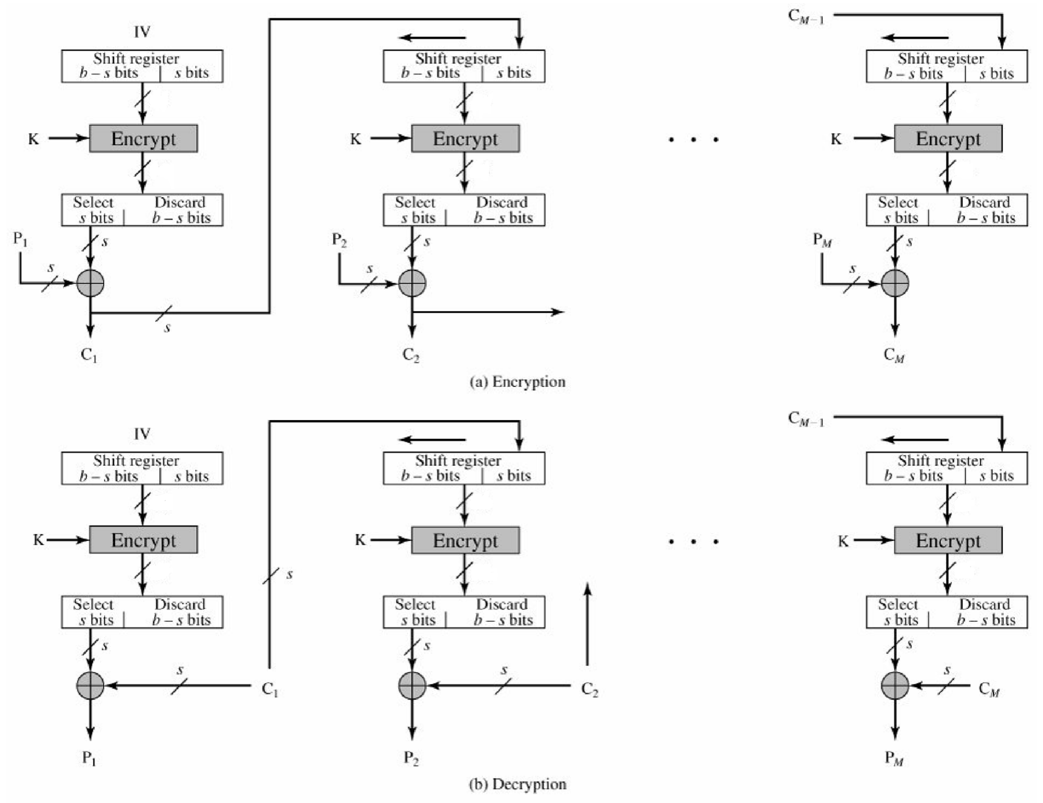


Figure 4.3 s-bit Cipher Feedback (CFB) Mode; a) Encryption, b) Decryption (Schneier, 1996)

Considering encryption, the input to the encryption function is a *b-bit* shift register that is initially set to some initialization vector (IV). The leftmost (most significant) s bits of the output of the encryption function are XORed with the first segment of plaintext P_1 to produce the first unit of ciphertext C_1, which is then transmitted. In addition, the contents of the shift register are shifted left by *s bits* and C1 is placed in the rightmost (least significant) *s bits* of the shift register. This process continues until all plaintext units have been encrypted.

Considering decryption, the same scheme is used, except that the received ciphertext unit is XORed with the output of the encryption function to produce the plaintext unit. Note that it is the encryption function that is used, not the decryption function.

Encryption and decryption steps are virtually identical and that the cipher is only used to encrypt. The cipher algorithm is only used to generate a sequence of pseudorandom bits to XOR against the plaintext. Since encryption is only used, this mode cannot be used with public key ciphers, because this would allow anyone to decrypt in this mode.

CFB mode does not have the same sensitivity to the IV that CBC mode has. A single bit change in the IV will cause random bit errors in the first block. Therefore, the IV need not be kept secret. Otherwise, the properties for CFB mode are similar to CBC mode, except scaled by a factor of b/s. Following are further guidelines for CFB mode:

- Decryption requires b/s previous blocks.
- Bit error in a ciphertext block will cause corresponding bit errors in the plaintext and produce random bit errors in the subsequent b/s blocks.
- CFB can recover from a dropped block in b/s blocks.

In CFB encryption, like CBC encryption, the input block to each forward cipher function (except the first) depends on the result of the previous forward cipher function; therefore, multiple forward cipher operations cannot be performed in parallel. In CFB decryption, the required forward cipher operations can be performed in parallel if the input blocks are first constructed (in series) from the IV and the ciphertext.

4.6 Output Feedback (OFB)

The output feedback (OFB) mode is similar in structure to that of CFB, as illustrated in Figure 4.4. As can be seen, it is the output of the encryption function that is fed back to the shift register in OFB, whereas in CFB the ciphertext unit is fed back to the shift register.

In OFB encryption, the IV is transformed by the forward cipher function to produce the first output block. The first output block is XORed with the first plaintext block to produce the first ciphertext block. The forward cipher function is then invoked on the first output block to produce the second output block. The second output block is XORed with the second plaintext block to produce the second ciphertext block, and the forward cipher function is invoked on the second output block to produce the third output block. Thus, the successive output blocks are produced from applying the forward cipher function to the previous output blocks, and the output blocks are XORed with the corresponding plaintext blocks to produce the ciphertext blocks. For the last block, which may be a partial block of *s* bits, the most significant *s* bits of the last output block are used for the XOR operation; the remaining *b-s* bits of the last output block are discarded.

In OFB decryption, the IV is transformed by the forward cipher function to produce the first output block. The first output block is XORed with the first ciphertext block to recover the first plaintext block. The first output block is then transformed by the forward cipher function to produce the second output block. The second output block is XORed with the second ciphertext block to produce the second plaintext block, and the second output block is also transformed by the forward cipher function to produce the third output block. Thus, the successive output blocks are produced from applying the forward cipher function to the previous output blocks, and the output blocks are XORed with the corresponding ciphertext blocks to recover the plaintext blocks. For the last block, which may be a partial block of *s* bits, the most significant *s* bits of the last output block are used for the XOR operation; the remaining *b-s* bits of the last output block are discarded.

OFB mode is similar to CFB mode, except that *s bits* of the previous output block are moved into the rightmost positions of the queue. Decryption is the reverse of this process. This is called *s-bit* OFB. On both the encryption and the decryption sides, the block algorithm is used in its encryption mode. This is sometimes called internal feedback, because the feedback mechanism is independent of both the plaintext and the ciphertext streams (Campbell, 1978).

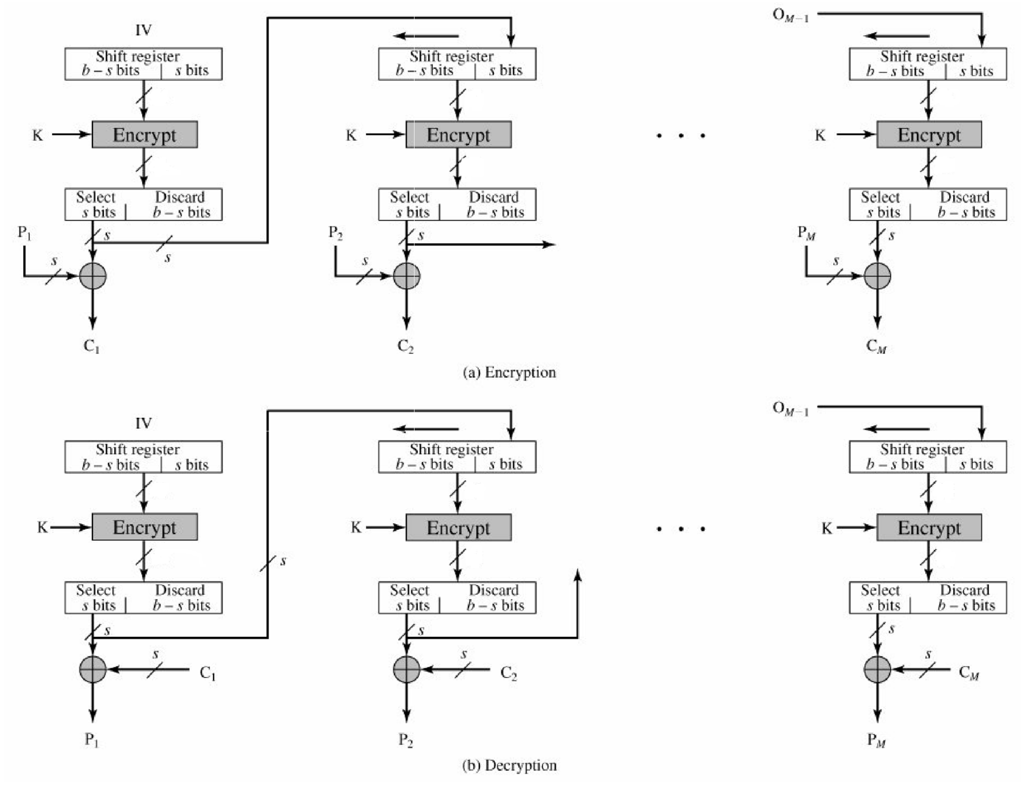


Figure 4.4 s-bit Output Feedback (OFB) Mode; a) Encryption, b) Decryption (Schneier, 1996)

OFB mode has no error extension. A single-bit error in the ciphertext causes a single-bit error in the recovered plaintext. This can be useful in some digitized analog transmissions, like digitized voice or video, where the occasional single-bit error can be tolerated but error extension cannot. On the other hand, a loss of synchronization is fatal. If the shift registers on the encryption end and the decryption end are not identical, then the recovered plaintext will be gibberish. Any system that uses OFB mode must have a mechanism for detecting a synchronization loss and a mechanism to fill both shift registers with a new (or the same) IV to regain synchronization.

An analysis of OFB mode ((Gait, 1977), (Davies & Parkin, 1983) & (Jueneman, 1983)) demonstrates that OFB should be used only when the feedback size is the same as the block size. For example 128 – bit AES cipher algorithm in 128 – bit OFB mode should be used.

OFB mode XORs a keystream with the text. This keystream will eventually repeat. It is important that it does not repeat with the same key; otherwise, there is no security. When the feedback size equals the block size, the block cipher acts as a permutation of m-bit values (where m is the block length) and the average cycle length is $2^m - 1$. For a 128-bit block length, this is a very long number. When the feedback size *s* is less than the block length, the average cycle length drops to around $2^{m/2}$. For a 128 -bit block cipher, this is only 2^{64} - not long enough (Schneier, 1996).

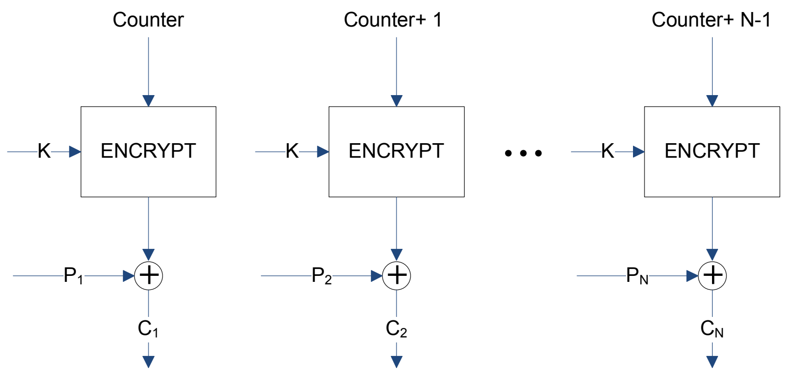
4.7 Counter Mode (CTR)

Although interest in the counter mode (CTR) has increased recently, with applications to ATM (asynchronous transfer mode) network security and IPSec (IP security), this mode was proposed early on (Diffie & Hellman, 1979).

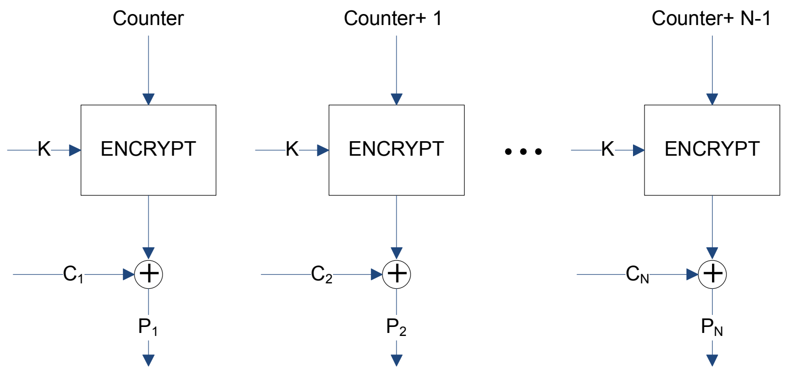
Block ciphers in counter mode use sequence numbers as the input to the algorithm ((Kent, 1976) & (Diffie & Hellman, 1979)). Instead of using the output of the encryption algorithm to fill the register, the input to the register is a counter. After each block encryption, the counter increments by some constant, typically one. The sequence of counters must have the property that each block in the sequence is different from every other block. This condition is not restricted to a single message: across all of the messages that are encrypted under the given key, all of the counters must be distinct (Figure 4.5).

In CTR encryption, the forward cipher function is invoked on each counter block, and the resulting output blocks are XORed with the corresponding plaintext blocks to produce the ciphertext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the last output block are used for the XOR operation; the remaining b-u bits of the last output block are discarded.

In CTR decryption, the forward cipher function is invoked on each counter block, and the resulting output blocks are XORed with the corresponding ciphertext blocks to recover the plaintext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the last output block are used for the XOR operation; the remaining b-u bits of the last output block are discarded.



a) Encryption



b) Decryption

Figure 4.5 Counter (CTR) Mode; a) Encryption, b) Decryption

In both CTR encryption and CTR decryption, the forward cipher functions can be performed in parallel; similarly, the plaintext block that corresponds to any particular ciphertext block can be recovered independently from the other plaintext blocks if the corresponding counter block can be determined. Moreover, the forward cipher functions can be applied to the counters prior to the availability of the plaintext or ciphertext data (Dworkin, 2001). (Lipmaa, Rogaway & Wagner, 2000) lists the following advantages of CTR mode:

- *Hardware efficiency*: Unlike the three chaining modes, encryption (or decryption) in CTR mode can be done in parallel on multiple blocks of plaintext or ciphertext. For the chaining modes, the algorithm must complete the computation on one block before beginning on the next block. This limits the maximum throughput of the algorithm to the reciprocal of the time for one execution of block encryption or decryption. In CTR mode, the throughput is only limited by the amount of parallelism that is achieved.

- *Software efficiency*: Similarly, because of the opportunities for parallel execution in CTR mode, processors that support parallel features, such as aggressive pipelining, multiple instruction dispatch per clock cycle, a large number of registers, and SIMD instructions, can be effectively utilized.

- *Preprocessing*: The execution of the underlying encryption algorithm does not depend on input of the plaintext or ciphertext. Therefore, if sufficient memory is available and security is maintained, preprocessing can be used to prepare the output of the encryption boxes that feed into the XOR functions in Figure 4.5. When the plaintext or ciphertext input is presented, then the only computation is a series of XORs. Such a strategy greatly enhances throughput.

- *Random access*: The *i*th block of plaintext or ciphertext can be processed in random-access fashion. With the chaining modes, block C_i cannot be computed until the *i* - 1 prior block are computed. There may be applications in which a ciphertext is stored and it is desired to decrypt just one block; for such applications, the random access feature is attractive.

- *Provable security*: It can be shown that CTR is at least as secure as the other modes discussed in this section.

- *Simplicity*: Unlike ECB and CBC modes, CTR mode requires only the implementation of the encryption algorithm and not the decryption algorithm. This matters most when the decryption algorithm differs substantially from the encryption algorithm, as it does for AES. In addition, the decryption key scheduling need not be implemented.

Table 4.1 summarizes the various modes of the operations. RBE stands for random bit errors, SBE is single-bit errors, where a single bit is altered.

Table 4.1 Block Cipher Modes of Operation ((Stallings, 2005), (Schneier, 1996) & (Galbreath, 2002))

Mode	Description	Typical Application	Requires IV	Bit Errors in Block	Bit Errors in IV
ECB	Each block of 128 plaintext bits is encoded independently using the same key.	Secure transmission of single values (e.g., an encryption key)	No	RBE in block	Not applicable
CBC	The input to the encryption algorithm is the XOR of the next 128 bits of plaintext and the preceding 128 bits of ciphertext.	General-purpose block-oriented transmission, authentication	Yes	RBE in block, SBE in next block	SBE in first block only
CFB	Input is processed *s* bits at a time, preceding ciphertext is used as input to the encryption algorithm, which is XORed with plaintext to produce next unit of ciphertext.	General-purpose stream-oriented transmission, authentication	Yes	SBE in block, RBE in subsequent block	RBE of first few blocks
OFB	Similar to CFB, except that the input to the encryption algorithm is the preceding output.	Stream-oriented transmission over noisy channel (e.g., satellite communication)	Yes	SBE in block	RBE in every block
CTR	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	General-purpose block-oriented transmission, useful for high-speed requirements	Yes	SBE in block	Bit error in counter block will result in RBE

CHAPTER FIVE

IMPLEMENTATION OF AES MODES OF OPERATION ON MICROBLAZE

The implementation of creating, testing and analyzing of AES modes of operation is performed on target platform described in section 2.4, Spartan3E 1600E Microblaze Edition board.

EDK (specifically XPS (Xilinx Inc., 2008)) is used for generation of complete hardware embedded processor system project. During generation any hardware related changes such as constraints entry, timing analysis, logic placement and routing, and device programming have all been done in EDK environment on behalf of ISE, which issues mentioned utilities in the background. XPS maintains hardware platform description in a high-level form, known as the Microprocessor Hardware Specification (MHS) file. The MHS, an editable text file, and is the principal source file representing the hardware component of the embedded system. XPS synthesizes the MHS source file into Hardware Description Language (HDL) netlists ready for FPGA place and route. The MHS file defines the configuration of the embedded processor system and includes information on the bus architecture, peripherals, processor, connectivity, and address space (Xilinx Inc., 2008).

XPS creates an analogous software system description in the Microprocessor Software Specification (MSS) file. The MSS file, together with software applications, are the principal source files representing the software elements of the embedded system generated. This collection of files, used in conjunction with EDK installed libraries and drivers, and any custom libraries and drivers for custom peripherals generated for this project provide allows SDK to compile applications generated. The compiled software routines are available as an Executable and Linkable Format (ELF) file. Figure 5.1 shows the files and flow stages that generate the ELF file. A SDK project is generated for each software application. The project directory contains C/C++ source files, executable output file, and associated utility files such as the make files used to build the project. Each SDK project directory is typically located under the XPS project directory tree for the embedded system that the application targets. Each SDK project produces just one executable file (Xilinx Inc., 2008).

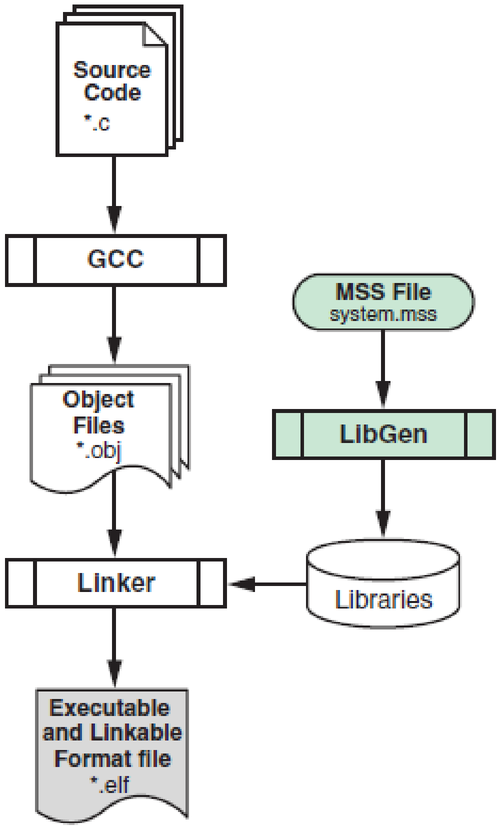


Figure 5.1 Stages of ELF File Generation

5.1 Generating the Hardware Platform & XPS Project

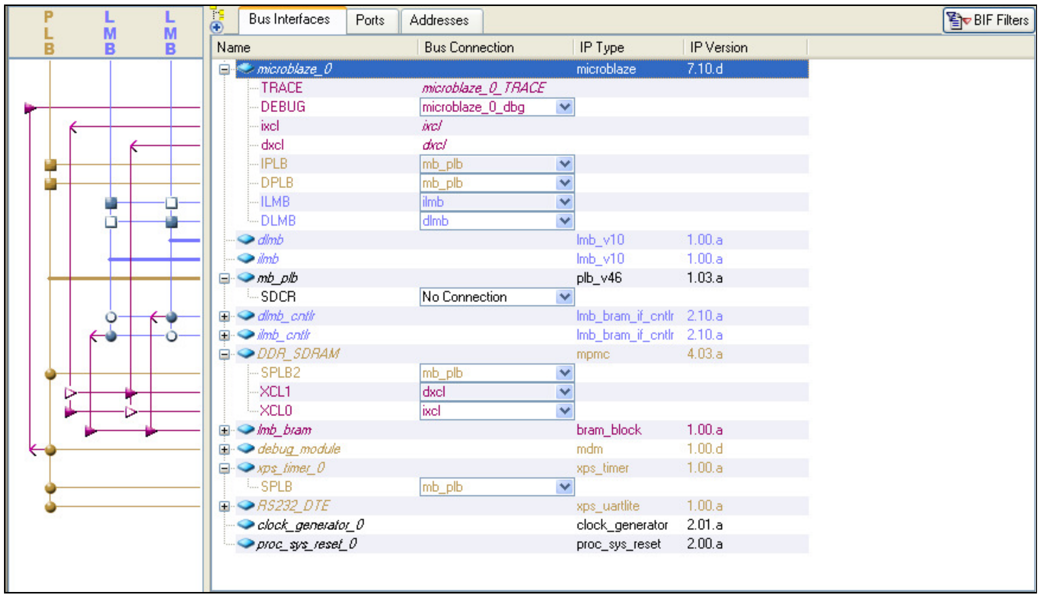
Creating an XPS project is described in (Xilinx Inc., 2008) in detail. A project is generated at 100MHz DDR clock frequency with cache and barrel shifter enabled. Project named "aes_modes_20101031_100MHzDDR_cache_barrel_shifter" and can be found in CD-ROM delivery, in the folder "projects". In the same directory there is a system block diagram as in the name "system.png" which is generated by XPS and illustrates overall structure of hardware design and there is design report summarizes all the hardware features of the project as in the name "system.html".

All the files generated by XPS e.g., MHS, MSS, User Constraints File (UCF) for the projects can be found in the "projects" directory, in the specific project directories.

Through the hardware development an XPS project is created from scratch with the Base System Builder (BSB) following steps described below:

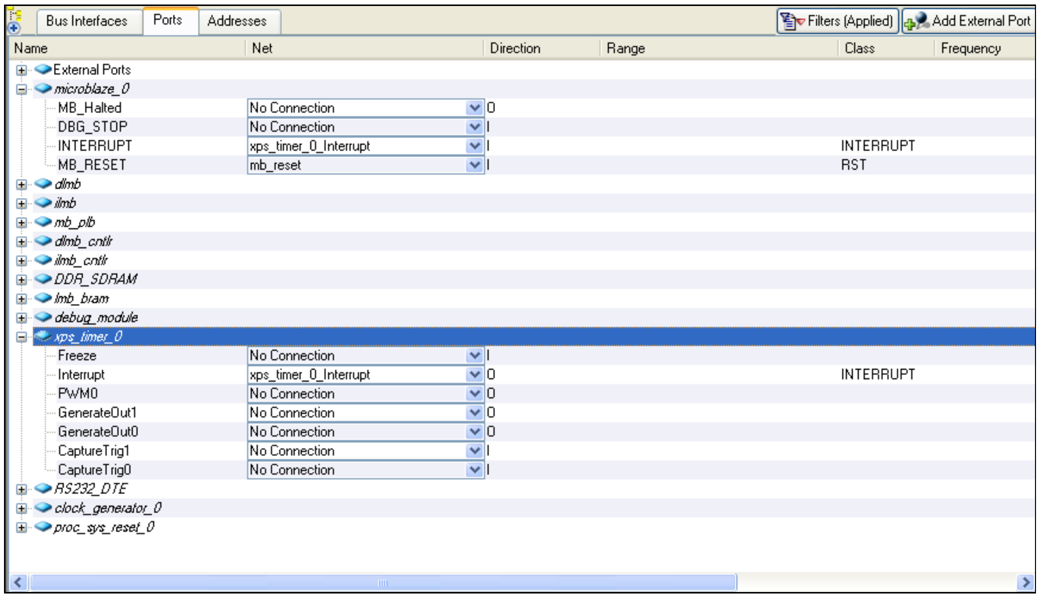
- Specify the target FPGA device : Spartan-3E 1600E MicroBlaze Dev Board
- Choose "Reference Clock Frequency" and "Processor Clock Frequency", 50MHz and 100MHz respectively, choose "Debug I/F" with "XMD with S/W debug stub", choose 16KB BRAM memory.
- Configure cache setup and enable both instruction and data cache allocating 2KB cache size.
- Disable all peripheral interfaces other than "RS232_DTE" and "DDR_SDRAM".
- Add "Timer" for further used for profiling (described in the next section). Connect timer's slave PLB bus (SPLB) to MicroBlaze's SPLB bus. Connect timer's interrupt pin with MicroBlaze interrupt pin.
- Configure "MicroBlaze" so that it has "Enable Barrel Shifter" in addition to default settings. Barrel shifter improves system response by means of time since AES algorithm uses shift operations in ShiftRows() part.(see Chapter 3).

Bus interface, port configuration and address mapping of project "aes_modes_20101031_100MHzDDR_cache_barrel_shifter" is depicted in Figure 5.2, Figure 5.3 and Figure 5.4.



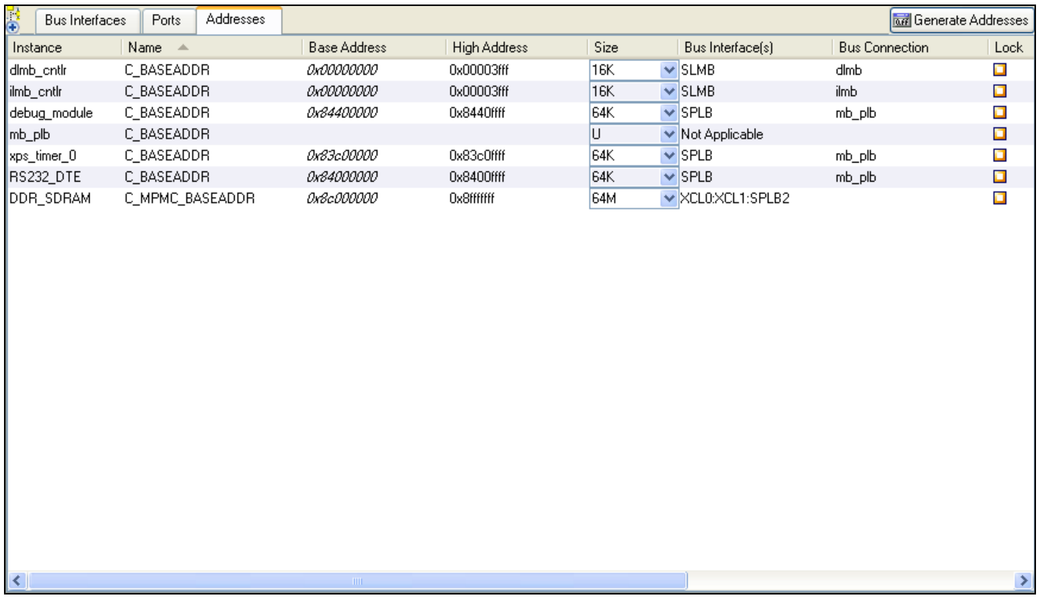
Name	Bus Connection	IP Type	IP Version
microblaze_0		microblaze	7.10.d
TRACE	microblaze_0_TRACE		
DEBUG	microblaze_0_dbg		
ixcl	ixcl		
dxcl	dxcl		
IPLB	mb_plb		
DPLB	mb_plb		
ILMB	ilmb		
DLMB	dlmb		
dlmb		lmb_v10	1.00.a
ilmb		lmb_v10	1.00.a
mb_plb		plb_v46	1.03.a
SDCR	No Connection		
dlmb_cntlr		lmb_bram_if_cntlr	2.10.a
ilmb_cntlr		lmb_bram_if_cntlr	2.10.a
DDR_SDRAM		mpmc	4.03.a
SPLB2	mb_plb		
XCL1	dxcl		
XCL0	ixcl		
lmb_bram		bram_block	1.00.a
debug_module		mdm	1.00.d
xps_timer_0		xps_timer	1.00.a
SPLB	mb_plb		
RS232_DTE		xps_uartlite	1.00.a
clock_generator_0		clock_generator	2.01.a
proc_sys_reset_0		proc_sys_reset	2.00.a

Figure 5.2 Bus Interfaces of project “aes_modes_20101031_100MHzDDR_cache_barrel_shifter”



Name	Net	Direction	Range	Class	Frequency
External Ports					
microblaze_0					
MB_Halted	No Connection	O			
DBG_STOP	No Connection	I			
INTERRUPT	xps_timer_0_Interrupt	I		INTERRUPT	
MB_RESET	mb_reset	I		RST	
dlmb					
ilmb					
mb_plb					
dlmb_cntlr					
ilmb_cntlr					
DDR_SDRAM					
lmb_bram					
debug_module					
xps_timer_0					
Freeze	No Connection	I			
Interrupt	xps_timer_0_Interrupt	O		INTERRUPT	
PWM0	No Connection	O			
GenerateOut1	No Connection	O			
GenerateOut0	No Connection	O			
CaptureTrig1	No Connection	I			
CaptureTrig0	No Connection	I			
RS232_DTE					
clock_generator_0					
proc_sys_reset_0					

Figure 5.3 Ports of project “aes_modes_20101031_100MHzDDR_cache_barrel_shifter”



Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	Lock
dlmb_cntlr	C_BASEADDR	*0x00000000*	0x00003fff	16K	SLMB	dlmb	☐
ilmb_cntlr	C_BASEADDR	*0x00000000*	0x00003fff	16K	SLMB	ilmb	☐
debug_module	C_BASEADDR	*0x84400000*	0x8440ffff	64K	SPLB	mb_plb	☐
mb_plb	C_BASEADDR			U	Not Applicable		☐
xps_timer_0	C_BASEADDR	*0x83c00000*	0x83c0ffff	64K	SPLB	mb_plb	☐
RS232_DTE	C_BASEADDR	*0x84000000*	0x8400ffff	64K	SPLB	mb_plb	☐
DDR_SDRAM	C_MPMC_BASEADDR	*0x8c000000*	0x8fffffff	64M	XCL0:XCL1:SPLB2		☐

Figure 5.4 Address map of project "aes_modes_20101031_100MHzDDR_cache_barrel_shifter"

After hardware platform design entry is setup, the next step is to set up User Constraints File (UCF) (Xilinx Inc., 2005). Since a specific development board is selected in BSB, the UCF file contains a complete pinout specification for connections to the on-board peripherals specified for the design. BSB automatically generated UCF file and since no external peripheral is used in the design UCF file should not be modified.

The last step is to generate the bitstream (BIT) file that represents the completed hardware platform. Hardware generation consists of the following steps:

- Generating the Netlist
- Generating the Bitstream.

Generating Netlist calls the platform building tool, Platgen (Xilinx Inc.., 2005), which does the following:

- Reads the design platform configuration Microprocessor Hardware Specification (MHS) file
- Generates a VHDL representation

- Runs the Xilinx Synthesis Technology (XST)
- Produces the netlist file in Xilinx NGC format

Generating the bitstream runs Platgen to produce the netlist. It then runs the ISE implementation tools, which read the UCF file and produce the BIT file containing hardware design. Software patterns are not included.

All the steps above is automated by XPS and can be performed from menu item "Hardware > Generate Bitstream". The changes already been made to the timer, added its interrupt pin and bus to MicroBlaze's interrupt pin and bus (SPLB). After generating bitstream file, software project is generated.

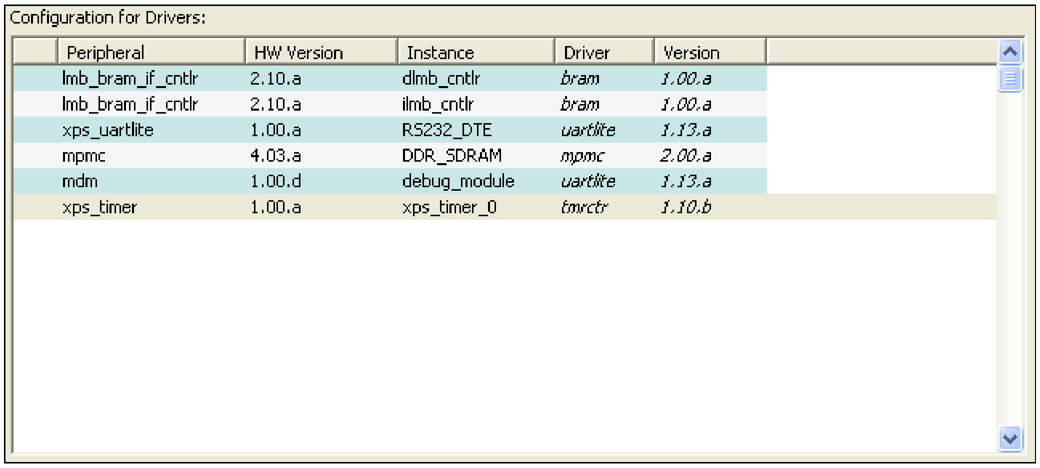
5.2 Generating the Software Platform & SDK Project

A software platform is a collection of software drivers and the operating system on which to build the application. The embedded software platform defines, for each processor, the drivers associated with the peripherals included in the hardware platform (the board support package), selected libraries, standard input/output devices, interrupt handler routines, and other related software features.

For the software platform generation SDK tool is used, SDK provides an interactive development environment that allows specify all aspects of the software platform and manage software applications. SDK maintains software platform description in a high-level form in the Microprocessor Software Specification (MSS) file. The MSS file represents the software component of the embedded system. SDK compiles applications, including software components specified in the MSS, into Executable and Linkable Format (ELF) files.

For purposes in AES modes of operation comparison a project is created as in the name "aes_modes_20101031_100MHzDDR_cache_barrel_shifter" in SDK environment.

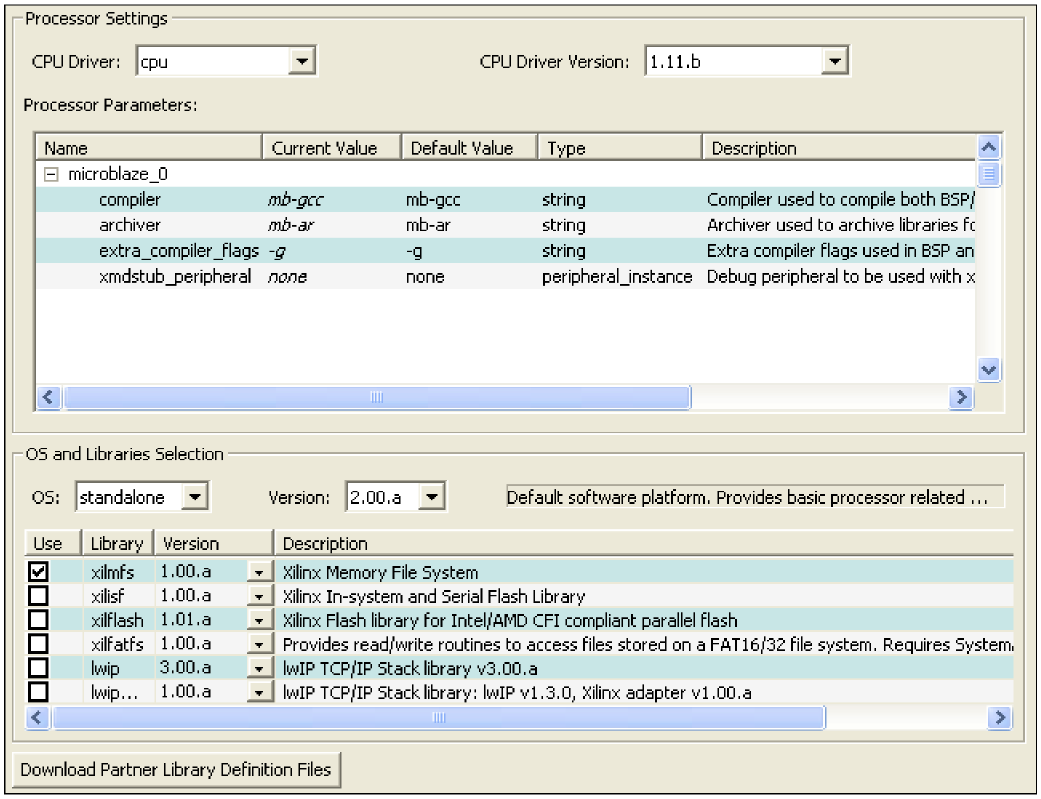
SDK is launched in XPS window from “Software > Launch Platform Studio SDK”. After Eclipse-based SDK is launched the first thing to do is to assign drivers, libraries, and operating systems to the software project. Figure 5.5, Figure 5.6 and Figure 5.7 depict software features of the projects.



Configuration for Drivers:

Peripheral	HW Version	Instance	Driver	Version
lmb_bram_if_cntlr	2.10.a	dlmb_cntlr	bram	1.00.a
lmb_bram_if_cntlr	2.10.a	ilmb_cntlr	bram	1.00.a
xps_uartlite	1.00.a	RS232_DTE	uartlite	1.13.a
mpmc	4.03.a	DDR_SDRAM	mpmc	2.00.a
mdm	1.00.d	debug_module	uartlite	1.13.a
xps_timer	1.00.a	xps_timer_0	tmrctr	1.10.b

Figure 5.5 Drivers



Processor Settings

CPU Driver: cpu   CPU Driver Version: 1.11.b

Processor Parameters:

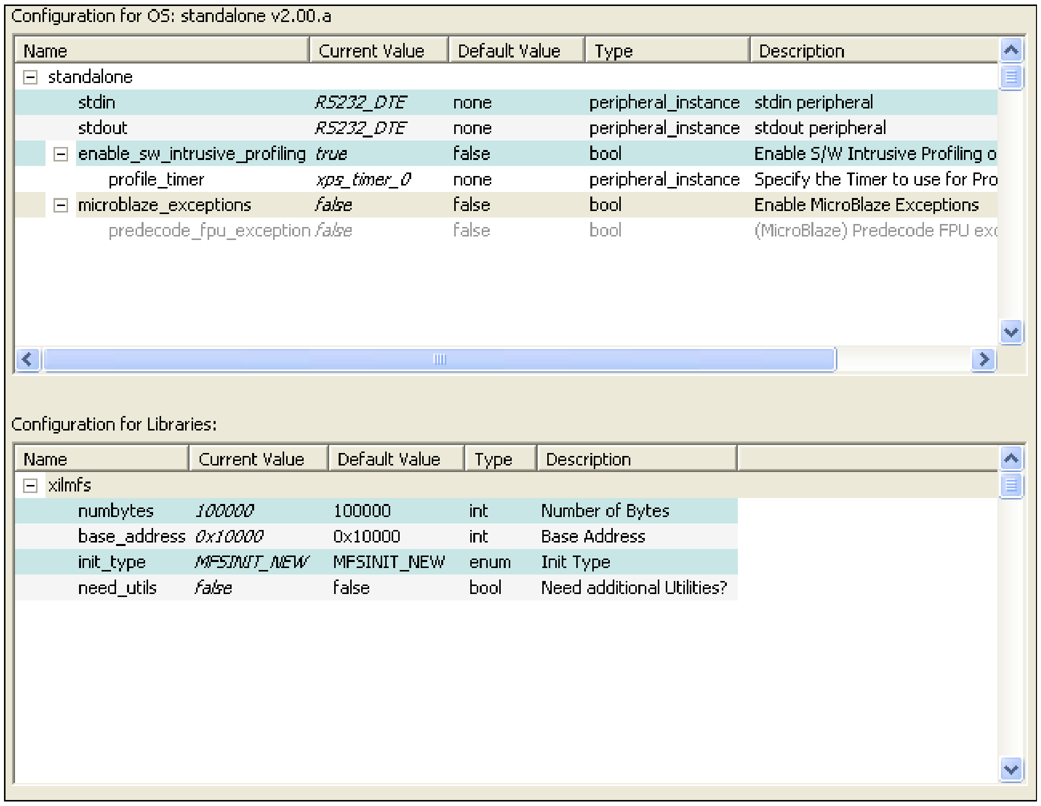
Name	Current Value	Default Value	Type	Description
⊟ microblaze_0				
compiler	mb-gcc	mb-gcc	string	Compiler used to compile both BSP/
archiver	mb-ar	mb-ar	string	Archiver used to archive libraries fc
extra_compiler_flags	-g	-g	string	Extra compiler flags used in BSP an
xmdstub_peripheral	none	none	peripheral_instance	Debug peripheral to be used with x

OS and Libraries Selection

OS: standalone   Version: 2.00.a   Default software platform. Provides basic processor related ...

Use	Library	Version	Description
☑	xilmfs	1.00.a	Xilinx Memory File System
☐	xilisf	1.00.a	Xilinx In-system and Serial Flash Library
☐	xilflash	1.01.a	Xilinx Flash library for Intel/AMD CFI compliant parallel flash
☐	xilfatfs	1.00.a	Provides read/write routines to access files stored on a FAT16/32 file system. Requires System
☐	lwip	3.00.a	lwIP TCP/IP Stack library v3.00.a
☐	lwip...	1.00.a	lwIP TCP/IP Stack library: lwIP v1.3.0, Xilinx adapter v1.00.a

Figure 5.6 OS and Libraries



Configuration for OS: standalone v2.00.a

Name	Current Value	Default Value	Type	Description
⊟ standalone				
stdin	*RS232_DTE*	none	peripheral_instance	stdin peripheral
stdout	*RS232_DTE*	none	peripheral_instance	stdout peripheral
⊟ enable_sw_intrusive_profiling	*true*	false	bool	Enable S/W Intrusive Profiling o
profile_timer	*xps_timer_0*	none	peripheral_instance	Specify the Timer to use for Pro
⊟ microblaze_exceptions	*false*	false	bool	Enable MicroBlaze Exceptions
predecode_fpu_exception	*false*	false	bool	(MicroBlaze) Predecode FPU exc

Configuration for Libraries:

Name	Current Value	Default Value	Type	Description
⊟ xilmfs				
numbytes	*100000*	100000	int	Number of Bytes
base_address	*0x10000*	0x10000	int	Base Address
init_type	*MFSINIT_NEW*	MFSINIT_NEW	enum	Init Type
need_utils	*false*	false	bool	Need additional Utilities?

Figure 5.7 Software Platform

Software Features:

- Standalone OS (since XilKernel OS does not support profiler (Xilinx Inc., 2007))
- XilMFS, Xilinx Memory File System to be used for file I/O (Xilinx Inc., 2006)
- XMD stub, Debug peripheral to be used for debugging
- XPS Timer, Timer to be used for profiling
- XPS UARTLITE, stdin/stdout peripheral, standard RS232 terminal

The software project is generated on the environment described above.

5.2.1 Implementing AES modes of Operation

In the software project an implementation of AES modes of operation is performed. (Bertoni, Breveglieri, Fragneto, Macchetti, & Marchesin, 2002) and (Gladman, 2002) details some software enhancements based on the standard AES algorithm and they all have been integrated into this design although our primary concern is modes of operation instead of algorithm itself. Generated software project for modes of operation has the following software structure as in Figure 5.8.

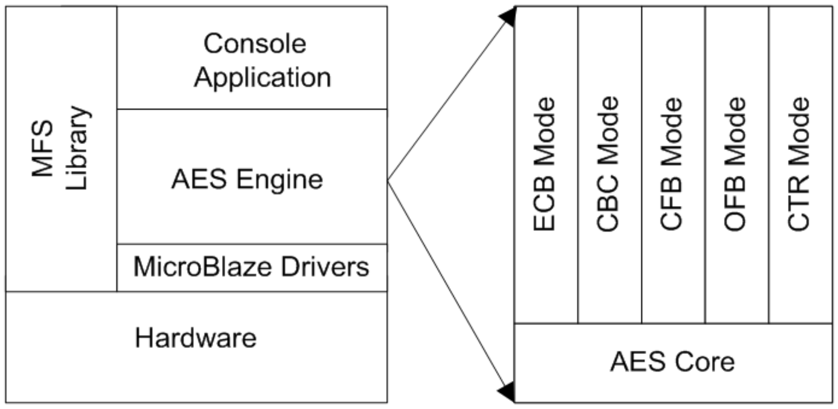


Figure 5.8 Software structure of project "aes_modes_20101031_100MHzDDR_cache_barrel_shifter"

Software project files are illustrated in Figure 5.10.

"Hardware" corresponds to FPGA device, is not related with software applications and/or drivers. In the project it consists of MicroBlaze soft processor core and other hardware intellectual properties (IP) like RS232_DTE, multiport memory controller (MPMC), timer, clock and reset generation circuits, debug module instantiated on Spartan-3E 1600E MicroBlaze Development Board. MicroBlaze and other hardware related parts configured as in section 5.1.

"MicroBlaze Drivers" includes:

- xilmfs_v1_00_a; "Xilinx Memory File System (MFS)" driver.
- uartlite_v1_13_a; RS232_DTE driver.

- tmrctr_v1_10_b; timer-counter driver.
- standalone_v2_00_a; standalone OS provides basic processor related drivers like interrupt handler, cache setup, exception setup.
- mpmc_v2_00_a; Xilinx MPMC driver supports Error Correction Code (ECC) capability, performance monitoring if one of them is enabled in the MPMC device. Default settings are used for the project.
- lldma_v1_00_a; controls DMA settings and transaction. No DMA transfer is used.
- common_v1_00_a; controls version management, basic types, assertions, system parameters like driver peripheral addresses.
- bram_v1_00_a; is not used, this driver is used when application runs on BRAM totally instead of DDR SDRAM.

"MFS Library"; is xilmfs_v1_00_a – Xilinx memory file system driver. It is drawn separately since there are calls from "AES Engine" and "Console Application" directly to MFS Library functions (Xilinx Inc., 2006).

"AES Engine"; implements AES block cipher algorithm and five confidentiality modes of operation (ECB, CBC, CFB, OFB and CTR – described in chapter 4). For CFB and OFB modes only 128-bit versions are implemented. For the initial testing test vectors from (Dworkin, 2001) are used and the results compared for consistency. Specific files can also be created and encrypted/decrypted. Section 5.3 has details about the results. The software flowchart is depicted in Figure 5.9.

"Console Application"; enables interrupts, initializes "MFS Library" and "AES Engine", and then waits for predefined commands which can be seen by running the application and typing "help" (Figure 5.11). Appendix – A has details about command usage and samples.

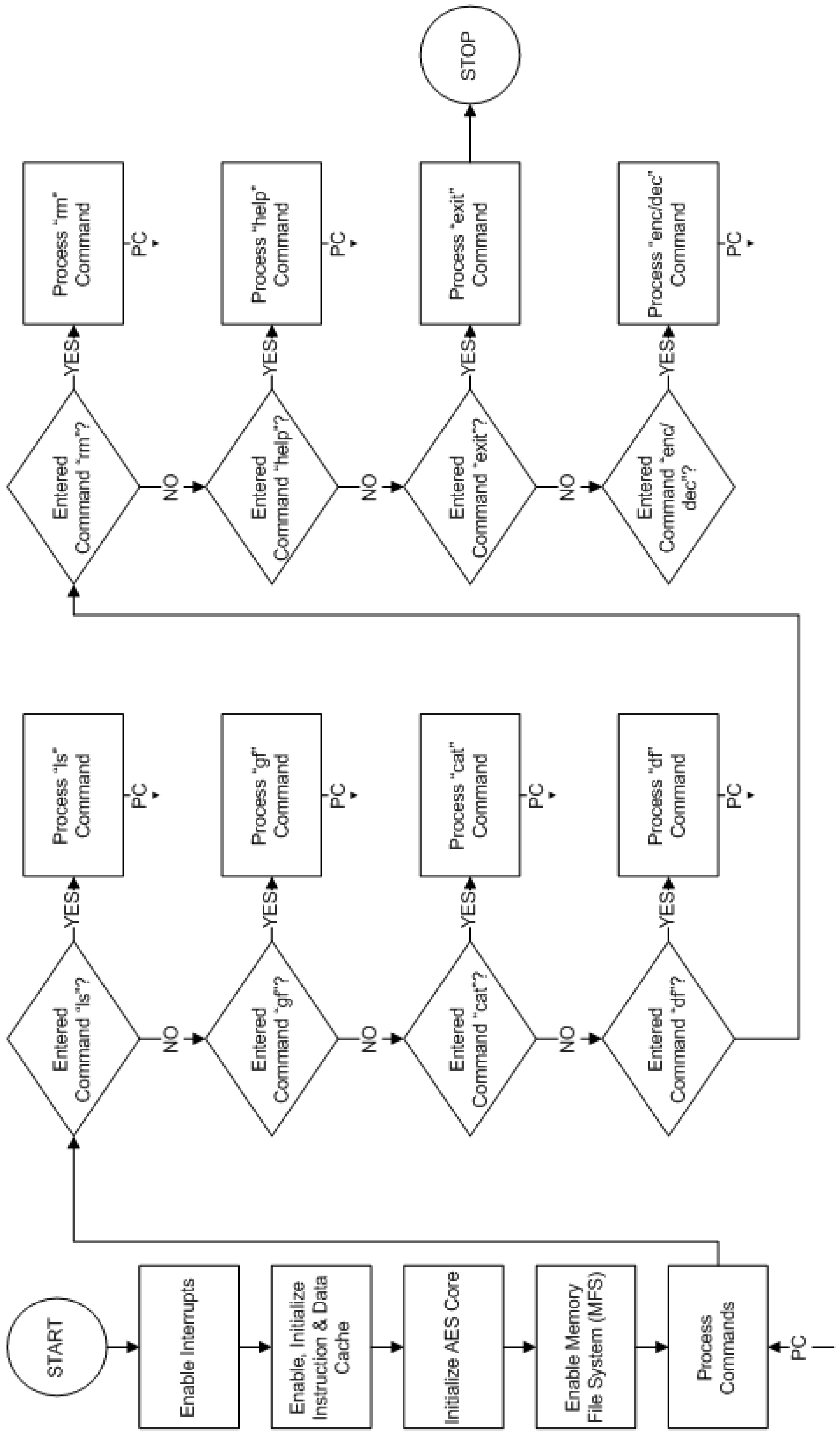
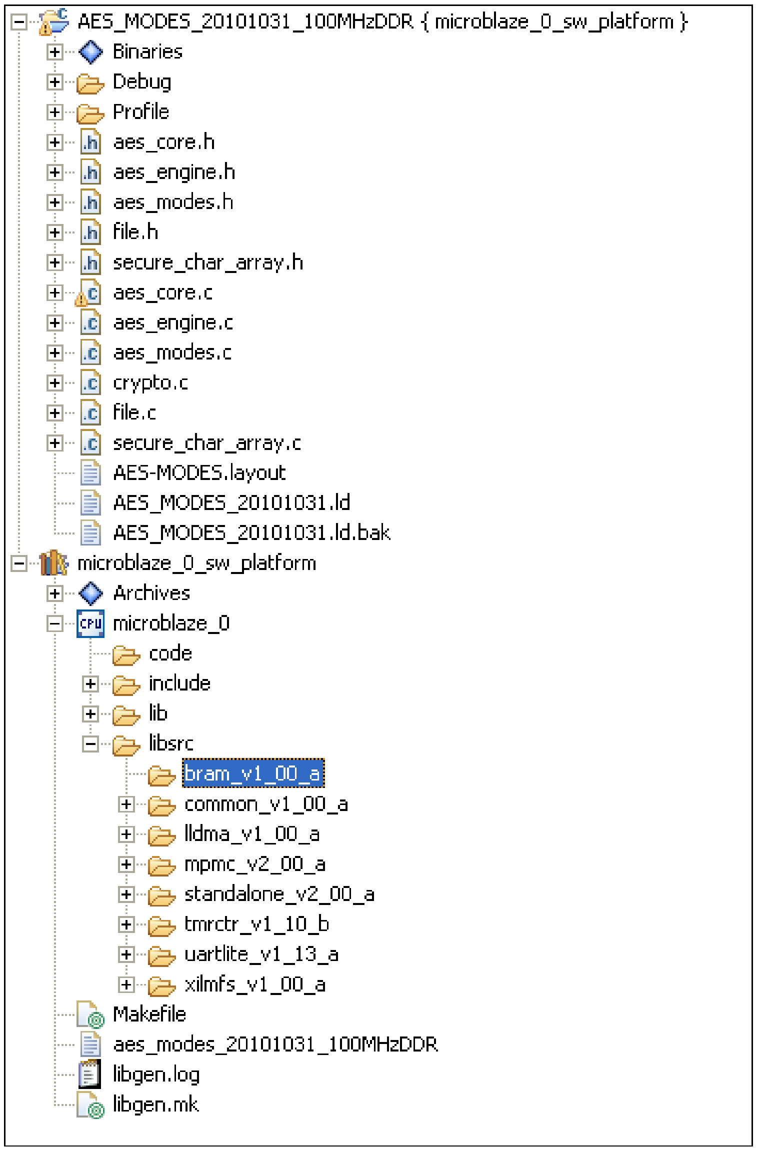
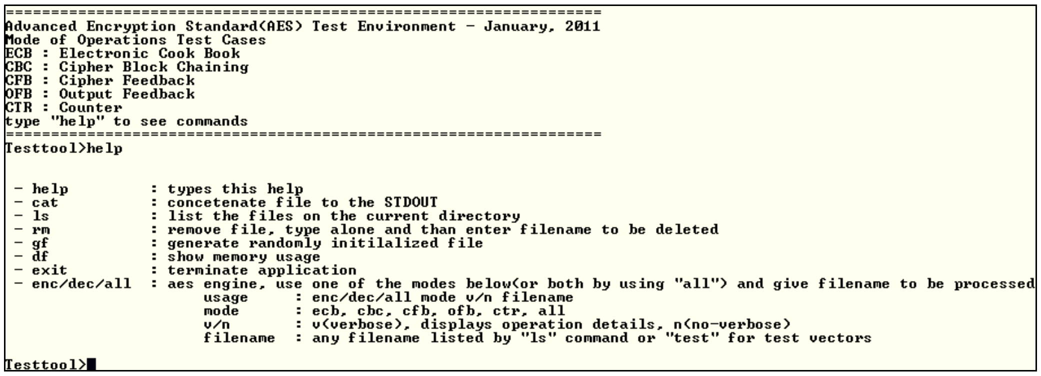


Figure 5.9 Software flowchart of the SDK project



```
AES_MODES_20101031_100MHzDDR { microblaze_0_sw_platform }
    Binaries
    Debug
    Profile
    aes_core.h
    aes_engine.h
    aes_modes.h
    file.h
    secure_char_array.h
    aes_core.c
    aes_engine.c
    aes_modes.c
    crypto.c
    file.c
    secure_char_array.c
    AES-MODES.layout
    AES_MODES_20101031.ld
    AES_MODES_20101031.ld.bak
microblaze_0_sw_platform
    Archives
    microblaze_0
        code
        include
        lib
        libsrc
            bram_v1_00_a
            common_v1_00_a
            lldma_v1_00_a
            mpmc_v2_00_a
            standalone_v2_00_a
            tmrctr_v1_10_b
            uartlite_v1_13_a
            xilmfs_v1_00_a
    Makefile
    aes_modes_20101031_100MHzDDR
    libgen.log
    libgen.mk
```

Figure 5.10 Software project files view.



```
=================================================================
Advanced Encryption Standard(AES) Test Environment - January, 2011
Mode of Operations Test Cases
ECB : Electronic Cook Book
CBC : Cipher Block Chaining
CFB : Cipher Feedback
OFB : Output Feedback
CTR : Counter
type "help" to see commands
=================================================================
Testtool>help


 - help          : types this help
 - cat           : concetenate file to the STDOUT
 - ls            : list the files on the current directory
 - rm            : remove file, type alone and than enter filename to be deleted
 - gf            : generate randomly initilalized file
 - df            : show memory usage
 - exit          : terminate application
 - enc/dec/all   : aes engine, use one of the modes below(or both by using "all") and give filename to be processed
                       usage     : enc/dec/all mode v/n filename
                       mode      : ecb, cbc, cfb, ofb, ctr, all
                       v/n       : v(verbose), displays operation details, n(no-verbose)
                       filename  : any filename listed by "ls" command or "test" for test vectors

Testtool>█
```

Figure 5.11 Commands to be used for AES modes of operation test environment.

5.2.2 Setting up Profiler in SDK

To profile an application in EDK hardware and software should be configured accordingly. Generating and viewing profile data can be done from SDK in visual form by issuing commands described below. Profiling restrictions can be found in Appendix – B.

5.2.2.1 Setting up the Hardware for Profiling

To profile a software application, interrupts are raised periodically to sample the program counter (PC) value. To do this, a timer is programmed and the timer interrupt handler is used to collect and store the PC. The profile interrupt handler requires full access to the timer, so a separate timer that is not used by the application itself must be available in the system. The timer interrupt signal is connected directly to the processor. For the system profiler is activated in hardware by adding a separate timer (xps_timer_0) to be used for profiling and interrupt signal (Figure 5.3) is directly connected to MicroBlaze interrupt pin.

5.2.2.2 Setting up the Software for Profiling

There are three steps involved in setting up the software application for profiling:

1. Specify the Board Support Package (BSP) settings from software platform settings and set the *enable_sw_intrusive_profiling* field to *true* and select the timer to be used for profiling (xps_timer_0, see Figure 5.7). Issue command “Generate libraries and the BSP” to configures the profiling libraries to be part of the standalone BSP (libxil.a).

2. Modify the software application code to enable interrupts. Since profile timer is directly connected to the processor without an interrupt controller, interrupts must be enabled in the processor. This is done by adding “microblaze_enable_interrupts()” call at the beginning of the application.

3. Build the application with the Profile build configuration using the Profile Configuration setting in the C/C++ Build configuration options tab. This step appends a -pg option to the compiler flags. An example compile command is: "mb-gcc -c -mxl-soft-mul -mxl-pattern-compare -mcpu=v7.10.d –pg -I ../../microblaze_0_sw_platform/microblaze_0/include -xl-mode-executable -g -O2 -oaes_modes.o ../aes_modes.c"

When libraries are generated, code required for profiling is automatically configured by the standalone BSP and becomes a part of the libxil.a library. The compiler inserts a call to the _mcount function after every software application function call. The _mcount function then gathers data on how often each of these software application functions are called. This function is also provided in the profiling library, and it handles collection of call graph data. The profiling timers initialize during software initialization, and the timer interrupt handlers collect information to provide the histogram data (Xilinx Inc., 2007).

5.2.2.3 Generating and Viewing Profile Data

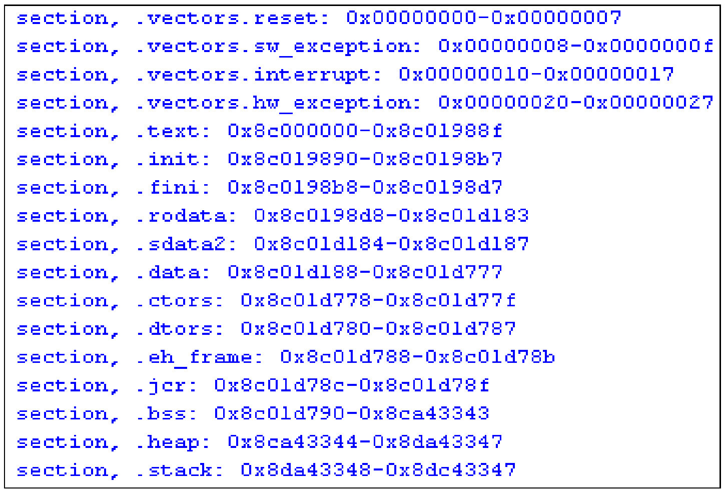
After compiling the application for profiling, it must be run once to obtain profile data. By enabling profiler support at the "run configuration" settings, three parameters must be configured (Xilinx Inc., 2007):

- *Sampling Frequency*: The sampling frequency determines the frequency at which timer interrupts are generated. When a higher frequency is selected, more samples are obtained. This provides more accuracy but is highly software-intrusive because of the number of interrupts. More calls are inserted to collect data.
- *Bin Size*: The program text region is divided into multiple bins. When a program is interrupted because of the sampling frequency, the bin size determines how accurate the PC location is in the sample. When a smaller bin size is selected, the program text region is divided into a large number of small bins. This allows a more accurate sample because profile data can be

attributed to a specific area of the text region. The disadvantage to using a smaller bin size is that it requires a large number of bins to cover the entire text region, so a large amount of memory space is required for storing profile data. When a larger bin size is selected, the program text region is divided into a small number of large bins. This requires less memory space for storing profile data. However, it is much more difficult to identify specific text regions for the sample because of the larger bin size.

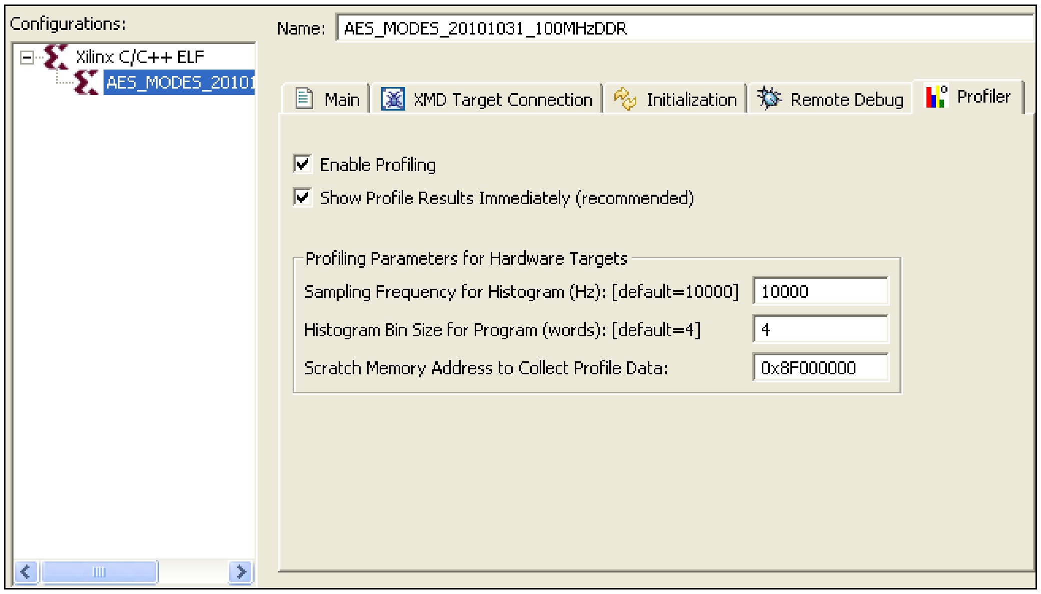
- *Profile Memory*: The profile memory parameter indicates where in memory the profile data must be stored. This memory needs to lie outside the program memory area (including the text, data, heap and stack) and should not be overwritten.

The software project has the following memory area (Figure 5.12) and profiler settings (Figure 5.13). All code and data sections, heap and stack are located in DDR SDRAM memory area.



```
section, .vectors.reset: 0x00000000-0x00000007
section, .vectors.sw_exception: 0x00000008-0x0000000f
section, .vectors.interrupt: 0x00000010-0x00000017
section, .vectors.hw_exception: 0x00000020-0x00000027
section, .text: 0x8c000000-0x8c01988f
section, .init: 0x8c019890-0x8c0198b7
section, .fini: 0x8c0198b8-0x8c0198d7
section, .rodata: 0x8c0198d8-0x8c01d183
section, .sdata2: 0x8c01d184-0x8c01d187
section, .data: 0x8c01d188-0x8c01d777
section, .ctors: 0x8c01d778-0x8c01d77f
section, .dtors: 0x8c01d780-0x8c01d787
section, .eh_frame: 0x8c01d788-0x8c01d78b
section, .jcr: 0x8c01d78c-0x8c01d78f
section, .bss: 0x8c01d790-0x8ca43343
section, .heap: 0x8ca43344-0x8da43347
section, .stack: 0x8da43348-0x8dc43347
```

Figure 5.12 Memory area


Configurations:
Xilinx C/C++ ELF
AES_MODES_20101
Name: AES_MODES_20101031_100MHzDDR

☑ Enable Profiling
☑ Show Profile Results Immediately (recommended)

Profiling Parameters for Hardware Targets
Sampling Frequency for Histogram (Hz): [default=10000] 10000
Histogram Bin Size for Program (words): [default=4] 4
Scratch Memory Address to Collect Profile Data: 0x8F000000

Figure 5.13 Profiler Settings

5.3 Testing the Application

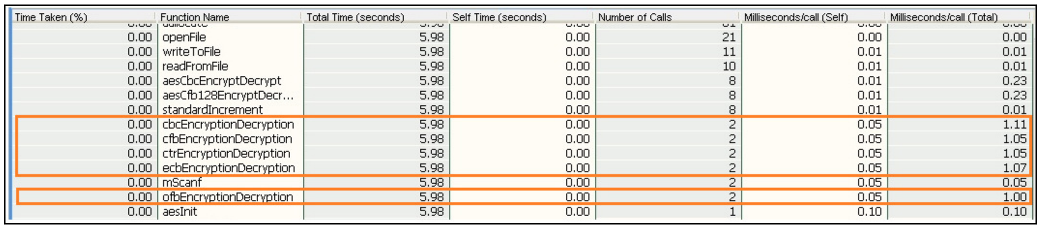
Application is tested in two ways:

- Test vectors from NIST (Dworkin, 2001) are demonstrated under all cipher modes and profile data is examined accordingly.
- Randomly generated data is examined under all cipher modes and profile data is examined accordingly.

5.3.1 Test Vectors

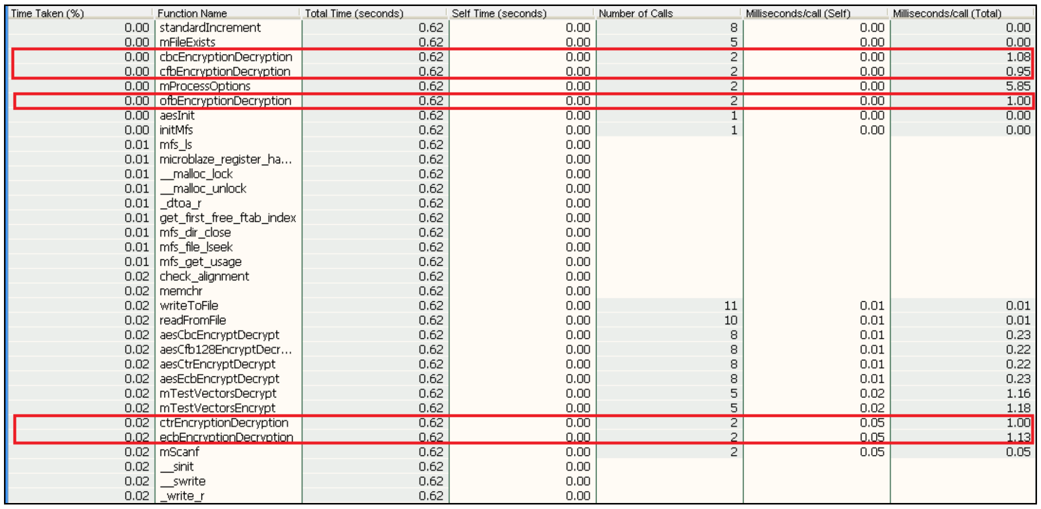
Test application can be called to test AES modes of operation on test vectors defined by (Dworkin, 2001) and command usage is supplied in Appendix – A. “all all n test” command creates 10 files that are encrypted and decrypted forms of original plaintext message using AES modes of operation: ECB, CBC, CFB, OFB and CTR. By default, 128 bit AES key is used; other key lengths (192, 256) are also tested but are not illustrated in the delivery.

Profiler results of the test application after testing test vectors and exited using "exit" command is illustrated in Figure 5.14. After application termination profiler data is loaded from memory 0x8F000000 to the file "gmon.out". SDK processes this file and creates figures "Time Spent in Functions (Self Time)", "3D Bar Chart (Number of Calls)", "3D Pie Chart (Percentage of Time)", "Call Table View" and "Flat Profile View" (Xilinx Inc., 2007). Only results from "Flat Profile View" is illustrated as this can give us time spent in each function effectively. Functions used for modes of operation and time spent in them are showed in rectangular selection boxes. Results are discussed in chapter six.



Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Milliseconds/call (Self)	Milliseconds/call (Total)
0.00	openFile	5.98	0.00	21	0.00	0.00
0.00	writeToFile	5.98	0.00	11	0.01	0.01
0.00	readFromFile	5.98	0.00	10	0.01	0.01
0.00	aesCbcEncryptDecrypt	5.98	0.00	8	0.01	0.23
0.00	aesCfb128EncryptDecr...	5.98	0.00	8	0.01	0.23
0.00	standardIncrement	5.98	0.00	8	0.01	0.01
0.00	cbcEncryptionDecryption	5.98	0.00	2	0.05	1.11
0.00	cfbEncryptionDecryption	5.98	0.00	2	0.05	1.05
0.00	ctrEncryptionDecryption	5.98	0.00	2	0.05	1.05
0.00	ecbEncryptionDecryption	5.98	0.00	2	0.05	1.07
0.00	mScanf	5.98	0.00	2	0.05	0.05
0.00	ofbEncryptionDecryption	5.98	0.00	2	0.05	1.00
0.00	aesInit	5.98	0.00	1	0.10	0.10

(a)



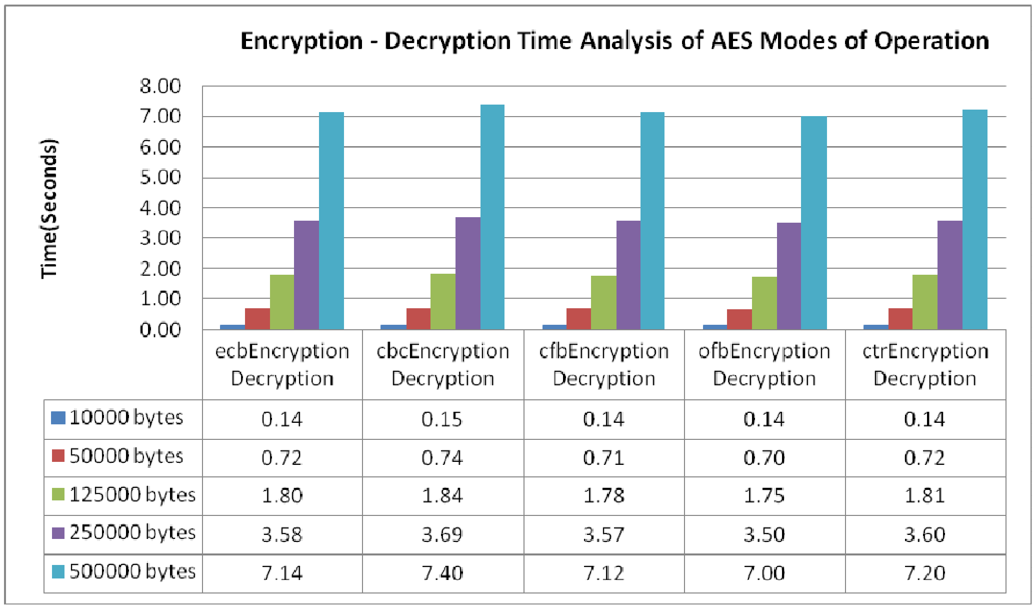
Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Milliseconds/call (Self)	Milliseconds/call (Total)
0.00	standardIncrement	0.62	0.00	8	0.00	0.00
0.00	mFileExists	0.62	0.00	5	0.00	0.00
0.00	cbcEncryptionDecryption	0.62	0.00	2	0.00	1.08
0.00	cfbEncryptionDecryption	0.62	0.00	2	0.00	0.95
0.00	mProcessOptions	0.62	0.00	2	0.00	5.85
0.00	ofbEncryptionDecryption	0.62	0.00	2	0.00	1.00
0.00	aesInit	0.62	0.00	1	0.00	0.00
0.00	initMfs	0.62	0.00	1	0.00	0.00
0.01	mfs_ls	0.62	0.00			
0.01	microblaze_register_ha...	0.62	0.00			
0.01	__malloc_lock	0.62	0.00			
0.01	__malloc_unlock	0.62	0.00			
0.01	_dtoa_r	0.62	0.00			
0.01	get_first_free_ftab_index	0.62	0.00			
0.01	mfs_dir_close	0.62	0.00			
0.01	mfs_file_lseek	0.62	0.00			
0.01	mfs_get_usage	0.62	0.00			
0.02	check_alignment	0.62	0.00			
0.02	memchr	0.62	0.00			
0.02	writeToFile	0.62	0.00	11	0.01	0.01
0.02	readFromFile	0.62	0.00	10	0.01	0.01
0.02	aesCbcEncryptDecrypt	0.62	0.00	8	0.01	0.23
0.02	aesCfb128EncryptDecr...	0.62	0.00	8	0.01	0.22
0.02	aesCtrEncryptDecrypt	0.62	0.00	8	0.01	0.22
0.02	aesEcbEncryptDecrypt	0.62	0.00	8	0.01	0.23
0.02	mTestVectorsDecrypt	0.62	0.00	5	0.02	1.16
0.02	mTestVectorsEncrypt	0.62	0.00	5	0.02	1.18
0.02	ctrEncryptionDecryption	0.62	0.00	2	0.05	1.00
0.02	ecbEncryptionDecryption	0.62	0.00	2	0.05	1.13
0.02	mScanf	0.62	0.00	2	0.05	0.05
0.02	__sinit	0.62	0.00			
0.02	__swrite	0.62	0.00			
0.02	_write_r	0.62	0.00			

(b)

Figure 5.14 Flat Profile View, results in issuing command "all all n test". (a) Represents results cache and barrel shifter enabled, (b) represents results only cache enabled.

5.3.2 Randomly Generated Data

Another usage of test application is on randomly generated files. This is accomplished by issuing “gf” command and by determining name and size of the generated file.A file is generated as in the name “deneme.txt” five times and stored 500000, 250000, 125000, 50000 and 10000 bytes respectively. All the commands usage can be found in Appendix A. Profiler results are illustrated in Appendix C. A much more informative figure is created in Figure 5.15. Results are discussed in chapter six.



	ecbEncryption Decryption	cbcEncryption Decryption	cfbEncryption Decryption	ofbEncryption Decryption	ctrEncryption Decryption
🟦 10000 bytes	0.14	0.15	0.14	0.14	0.14
🟥 50000 bytes	0.72	0.74	0.71	0.70	0.72
🟩 125000 bytes	1.80	1.84	1.78	1.75	1.81
🟪 250000 bytes	3.58	3.69	3.57	3.50	3.60
🟦 500000 bytes	7.14	7.40	7.12	7.00	7.20

Figure 5.15 Encryption – decryption time analysis of each AES modes of operation

CHAPTER SIX
CONCLUSION AND FUTURE WORK

In this thesis, NIST approved cryptographic block cipher algorithm Advanced Encryption Standard (AES) and most importantly modes of operation are discussed and implemented in a Xilinx MicroBlaze SoC development platform. A mode of operation is a technique for enhancing the effect of a cryptographic algorithm such as applying a block cipher to a sequence of data blocks or a data stream. The security is a function of underlying block cipher not the modes. The aim of this thesis was to analyze the modes of operation in an embedded SoC environment and to determine the differences between them in terms of process time taken.

Two tests are performed with two different configurations; first with test vectors delivered by NIST and second with randomly generated data using with cache and with/without barrel shifter. Test vectors are 64 bytes long and from profiling data obtained from chapter five showed that total time for encryption and decryption CFB and OFB mode took the least time; for OFB 1.05ms – 1.00ms, for CFB 0.95ms – 1.00ms, for CTR 1.05ms – 1.00ms, for CBC 1.11ms – 1.08ms and for ECB 1.07ms – 1.13ms. Time pairs represent when cache and barrel shifter enabled versus only cache enabled. Randomly generated data could be any data length restricted to 1MB (memory restriction because of restricted file system capacity) and random data is generated that contains 500000, 250000, 125000, 50000 and 10000 bytes. From Figure 5.15 it is obtained that total time for encryption and decryption OFB mode took the least time, and other modes enumerated CFB, ECB, CTR and CBC respectively. Implementation showed that feedback modes (OFB, CFB) are much more efficient than non-feedback modes (ECB, CTR) in terms of encryption – decryption time.

Barrel shifter is a hardware shifter and it is expected that including it in the design would increase overall performance but using efficient implementations developed by (Bertoni, Breveglieri, Fragneto, Macchetti & Marchesin, 2002) eliminates barrel shifter performance addition. Results proved that with/without barrel shifter did not increase performance at all.

It is learned that ECB mode is vulnerable to the dictionary attack and decryption time takes more than encryption time so normally it is not used other than special purposes such as sending only encryption key or IV to the communication partners. CBC mode eliminates the dictionary attack by using the contents of the previous block to encrypt the current block but time to encrypt and decrypt is the worst among all modes of operation. In CFB, OFB and CTR mode encryption is same as decryption and actually no decryption block is used, the same encryption block is used. This results feedback mode is faster than ECB and CBC modes since decryption takes much more time than encryption. Encryption in feedback modes is the same as decryption thus memory overhead is less compared to CBC and ECB modes.

Modes of operation have primarily been defined for encryption and authentication. Modes of operation is already studied for encryption purposes but some modern modes of operation combine encryption and authentication in an efficient way, and are known as authenticated encryption modes. The results of combined encryption and authentication may be investigated in embedded platforms.

REFERENCES

Altera Corporation (2004). Nios embedded processor system development. Retrieved August 8, 2010, from http://www.altera.com/products/ip/processors/nios/nio-index.html.

Barma S. (2007). Design, development and performance evaluation of multiprocessor systems on FPGA. Master Thesis, Department of Computer Science and Engineering, Indian Institute of Technology Delhi.

Başkök M. D. (2007). The modeling of AES encryption algorithm. Master Thesis, Electrical & Electronics Engineering, Gazi University.

Bertoni G., Breveglieri L., Fragneto P., Macchetti M., & Marchesin S. (2002). Efficient software implementation of AES on 32-Bit Platforms. CHES 2002, 2523, 159–171.

Campbell C. M. (1978). Design and specification of cryptographic capabilities. IEEE Computer Society Magazine, 16, 15–19.

Daemen J. & Rijmen V. (2003). AES proposal: Rijndael, AES algorithm submission. Retrieved October 21, 2010 from http://csrc.nist.gov/publications/fips/.Xilinx Inc. (2008). MicroBlaze processor reference guide. A distribution within EDK 10.1.

Davies D.W. & Parkin G.I.P. (1983). The average size of the key stream in output feedback mode. Proceedings of Crypto 82, 97–98.

Diffie W. & Hellman M.E. (1979). Privacy and authentication: An introduction to cryptography. Proceedings of the IEEE, 67, 397–427.

Dworkin M. (2001). Recommendation for block cipher modes of operation. Retrieved August 10, 2010, from http://csrc.nist.gov/publications/nistpubs/80038a/sp80038a.pdf.

EnSilica Ltd (2010). A study of AES and its efficient implementation on eSi-RISC. Retrieved October 21, from, http://www.ensilica.com/pdfs/A_study_of_aes_and_its_efficient_implementation_on_eSi_RISC_r1.0.pdf .

Fu Y., Hao L. & Zhang X. (2005). Design of an extremely high performance counter mode AES reconfigurable processor. IEEE Computer Society.

Gait J. (1977). A new nonlinear pseudorandom number generator. IEEE Transactions on Software Engineering, SE–3, 359–363

Galbreath N. (2002). Cryptography for internet and database applications. Indianapolis: Wiley.

Gladman B. (2002). A specification for Rijndael, the AES algorithm. Retrieved October 21, 2010, from http://www.gladman.me.uk/cryptography_technology.

Hodjat A. & Verbauwhede Ingrid (2006). Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. IEEE Transactions on Computers, 55, 366-372.

IEEE (2003). IEEE Std 802.15.3-2003, Part 15.3: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for High Rate Wireless Personal Area Networks (WPANs).

IEEE (2007). IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

Jayavardhan R. K. (2003). An analysis and evaluation of performance and code optimization techniques for encryption and decryption in embedded systems. Master Thesis, Electrical Engineering, University of South Florida.

Jueneman R.R. (1983). Analysis of certain aspects of output–feedback mode. Proceedings of Crypto 82, 99–127.Xilinx Inc. (2008). *ISE 10.1 in depth tutorial.* Retrieved September 20, 2010, from www.xilinx.com/direct/ise10_tutorials/ise10tut.pdf.

Kent S.T. (1976). Encryption–based protection protocols for interactive user–computer communications. MIT/LCS/TR–162, MIT Laboratory for Computer Science.

Lipmaa H., Rogaway P. & Wagner D. (2000). CTR mode encryption. NIST First Modes of Operation Workshop.

Magnusson P. (2004). Evaluating Xilinx MicroBlaze for network SoC solutions. Master Thesis, Computer Engineering, Luleå University of Technology.

National Institute of Standards and Technology [NIST] (2001). Advanced Encryption Standard FIPS PUB 197.

NIST (1999). Data encryption standard FIPS PUB 46-3.

NIST (2008). Special publication 800-67, recommendation for the triple data encryption algorithm (TDEA) block cipher.

Parnell K., & Bryner R. (2004). Comparing and contrasting FPGA and microprocessor system design and development. Retrieved October 21, 2010, from http://www.xilinx.com/support/documentation/white_papers/wp213.pdf.

Schneier B. (1996). Applied cryptography (2nd ed.). Indianapolis: Wiley.

Stallings W. (2005). Cryptography and network security principles and practices (4th ed.). USA: Printice Hall.

Ward R. W. & Molteno T. C. A. (2002). A CPLD coprocessor for embedded cryptography. Physics Department, University of Otago.

Xilinx Inc. (2005). *PLB usage in Xilinx FPGAs.* A distribution within EDK 10.1.

Xilinx Inc. (2005). *OPB usage in Xilinx FPGAs.* A distribution within EDK 10.1.

Xilinx Inc. (2005). Constraints guide. A distribution within EDK 10.1.

Xilinx Inc. (2005). Embedded system tools reference manual. A distribution within EDK 10.1.

Xilinx Inc. (2006). LibXil memory file system (MFS). A distribution within EDK 10.1.

Xilinx Inc. (2007). EDK profiling user guide. A distribution within EDK 10.1.

Xilinx Inc. (2007). MicroBlaze development kit Spartan-3E 1600E edition user guide. Not available from Xilinx anymore, supplied in CD-ROM delivery.

Xilinx Inc. (2008). *Platform specification format reference manual.* A distribution within EDK 10.1.

Xilinx Inc. (2008). EDK Concepts, Tools, and Techniques. A distribution within EDK 10.1.

Zabala E. (2004). An excellent way to gain an understanding of the inner workings of AES, Retrieved October 21, 2010, from http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf

APPENDIX A – AES MODES OF OPERATION COMMAND USAGE

It is assumed that EDK project is opened in XPS as in the name "aes_modes_20101031_100MHzDDR_cache_barrel_shifter" and then SDK is called from XPS menu; "Software > Launch Platform Studio SDK". After SDK is launched SDK project "aes_modes_20101031_100MHzDDR_cache_barrel_shifter" is opened. Spartan3E 1600E MicroBlaze board is programmed from SDK menu item "Device Configuration>Program FPGA" and software project is run on the board with profiler settings done as described in Chapter 5, section 5.2.2. Following is an output of the terminal after program runs and shows command usage.

```
==================================================================
Advanced Encryption Standard(AES) Test Environment - January, 2011
Mode of Operations Test Cases
ECB : Electronic Cook Book
CBC : Cipher Block Chaining
CFB : Cipher Feedback
OFB : Output Feedback
CTR : Counter
type "help" to see commands
==================================================================
Testtool>help

 - help        : types this help
 - cat         : concetenate file to the STDOUT
 - ls          : list the files on the current directory
 - rm          : remove file, type alone and than enter filename to be deleted
 - gf          : generate randomly initilalized file
 - df          : show memory usage
 - exit        : terminate application
 - enc/dec/all  : aes engine, use one of the modes below(or both by using "all") and give filename
to be processed
                usage    : enc/dec/all mode v/n filename
                mode     : ecb, cbc, cfb, ofb, ctr, all
                v/n      : v(verbose), displays operation details, n(no-verbose)
                filename : any filename listed by "ls" command or "test" for test vectors

Testtool>ls
Testtool>all all n test
non-verbose mode

Testing NIST delivered test vector(s) using command(all) and mode(all)
```

```
Encrypting & Decrypting...
[MK] : Using Mode of Operation : ElectronicCodebook
[MK] : Using Mode of Operation : ElectronicCodebook
Encrypting & Decrypting...
[MK] : Using Mode of Operation : CipherBlockChaining
[MK] : Using Mode of Operation : CipherBlockChaining
Encrypting & Decrypting...
[MK] : Using Mode of Operation : CipherFeedback
[MK] : Using Mode of Operation : CipherFeedback
Encrypting & Decrypting...
[MK] : Using Mode of Operation : OutputFeedback
[MK] : Using Mode of Operation : OutputFeedback
Encrypting & Decrypting...
[MK] : Using Mode of Operation : Counter
[MK] : Using Mode of Operation : Counter

Testtool>ls
test_ecb_enc.txt 00000040
test.txt 00000040
test_ecb_enc_dec.txt 00000040
test_cbc_enc.txt 00000040
test_cbc_enc_dec.txt 00000040
test_cfb_enc.txt 00000040
test_cfb_enc_dec.txt 00000040
test_ofb_enc.txt 00000040
test_ofb_enc_dec.txt 00000040
test_ctr_enc.txt 00000040
test_ctr_enc_dec.txt 00000040

Testtool>cat
est_ecb_enc.txt 00000040
test.txt 00000040
test_ecb_enc_dec.txt 00000040
test_cbc_enc.txt 00000040
test_cbc_enc_dec.txt 00000040
test_cfb_enc.txt 00000040
test_cfb_enc_dec.txt 00000040
test_ofb_enc.txt 00000040
test_ofb_enc_dec.txt 00000040
test_ctr_enc.txt 00000040
test_ctr_enc_dec.txt 00000040

Enter filename to be concetenated :test.txt
file length = 64(bytes)
0x6B 0xC1 0xBE 0xE2 0x2E 0x40 0x9F 0x96 0xE9 0x3D 0x7E 0x11 0x73 0x93 0x17 0x2A
0xAE 0x2D 0x8A 0x57 0x1E 0x03 0xAC 0x9C 0x9E 0xB7 0x6F 0xAC 0x45 0xAF 0x8E 0x51
0x30 0xC8 0x1C 0x46 0xA3 0x5C 0xE4 0x11 0xE5 0xFB 0xC1 0x19 0x1A 0x0A 0x52 0xEF
0xF6 0x9F 0x24 0x45 0xDF 0x4F 0x9B 0x17 0xAD 0x2B 0x41 0x7B 0xE6 0x6C 0x37 0x10

Testtool>cat
```

```
test_ecb_enc.txt 00000040
test.txt 00000040
test_ecb_enc_dec.txt 00000040
test_cbc_enc.txt 00000040
test_cbc_enc_dec.txt 00000040
test_cfb_enc.txt 00000040
test_cfb_enc_dec.txt 00000040
test_ofb_enc.txt 00000040
test_ofb_enc_dec.txt 00000040
test_ctr_enc.txt 00000040
test_ctr_enc_dec.txt 00000040

Enter filename to be concetenated :test_ctr_enc_dec.txt
file length = 64(bytes)

0x6B 0xC1 0xBE 0xE2 0x2E 0x40 0x9F 0x96 0xE9 0x3D 0x7E 0x11 0x73 0x93 0x17 0x2A
0xAE 0x2D 0x8A 0x57 0x1E 0x03 0xAC 0x9C 0x9E 0xB7 0x6F 0xAC 0x45 0xAF 0x8E 0x51
0x30 0xC8 0x1C 0x46 0xA3 0x5C 0xE4 0x11 0xE5 0xFB 0xC1 0x19 0x1A 0x0A 0x52 0xEF
0xF6 0x9F 0x24 0x45 0xDF 0x4F 0x9B 0x17 0xAD 0x2B 0x41 0x7B 0xE6 0x6C 0x37 0x10

Testtool>df
Number of Blocks Used : 12
Number of Blocks Free : 19988

Testtool>gf
Generating File ...:
Enter filename to be created :deneme.txt
Enter file size (decimal) :2000

Testtool>all all n deneme.txt
non-verbose mode
Testing file(deneme.txt) using command(all) and mode(all)
Encrypting & Decrypting...
Filename : test_ctr_enc.txt
Concatenated filename [deneme.txt_ecb_enc]
[MK] : Using Mode of Operation : ElectronicCodebook
Filename : deneme.txt_ecb_enc
Concatenated filename [deneme.txt_ecb_enc_dec]
[MK] : Using Mode of Operation : ElectronicCodebook
Encrypting & Decrypting...
Filename : deneme.txt_ecb_enc
Concatenated filename [deneme.txt_cbc_enc]
[MK] : Using Mode of Operation : CipherBlockChaining
Filename : deneme.txt_cbc_enc
Concatenated filename [deneme.txt_cbc_enc_dec]
[MK] : Using Mode of Operation : CipherBlockChaining
Encrypting & Decrypting...
Filename : deneme.txt_cbc_enc
Concatenated filename [deneme.txt_cfb_enc]
[MK] : Using Mode of Operation : CipherFeedback
```

```
Filename: deneme.txt_cfb_enc
Concatenated filename [deneme.txt_cfb_enc_dec]
[MK] : Using Mode of Operation : CipherFeedback
Encrypting & Decrypting...
Filename : deneme.txt_cfb_enc
Concatenated filename [deneme.txt_ofb_enc]
[MK] : Using Mode of Operation : OutputFeedback
Filename : deneme.txt_ofb_enc
Concatenated filename [deneme.txt_ofb_enc_dec]
[MK] : Using Mode of Operation : OutputFeedback
Encrypting & Decrypting...
Filename : deneme.txt_ofb_enc
Concatenated filename [deneme.txt_ctr_enc]
[MK] : Using Mode of Operation : Counter
Filename: deneme.txt_ctr_enc
Concatenated filename [deneme.txt_ctr_enc_dec]
[MK] : Using Mode of Operation : Counter

Testtool>ls
test_ecb_enc.txt 00000040
test.txt 00000040
test_ecb_enc_dec.txt 00000040
test_cbc_enc.txt 00000040
test_cbc_enc_dec.txt 00000040
test_cfb_enc.txt 00000040
test_cfb_enc_dec.txt 00000040
test_ofb_enc.txt 00000040
test_ofb_enc_dec.txt 00000040
test_ctr_enc.txt 00000040
test_ctr_enc_dec.txt 00000040
deneme.txt 000007d0
deneme.txt_ecb_enc 000007d0
deneme.txt_ecb_enc_dec 000007d0
deneme.txt_cbc_enc 000007d0
deneme.txt_cbc_enc_dec 000007d0
deneme.txt_cfb_enc 000007d0
deneme.txt_cfb_enc_dec 000007d0
deneme.txt_ofb_enc 000007d0
deneme.txt_ofb_enc_dec 000007d0
deneme.txt_ctr_enc 000007d0
deneme.txt_ctr_enc_dec 000007d0

Testtool>df
Number of Blocks Used : 57
Number of Blocks Free : 19943

Testtool>exit
-- Exiting main() –
```

Figure A.1 An example of commands usage in AES modes of operation project.

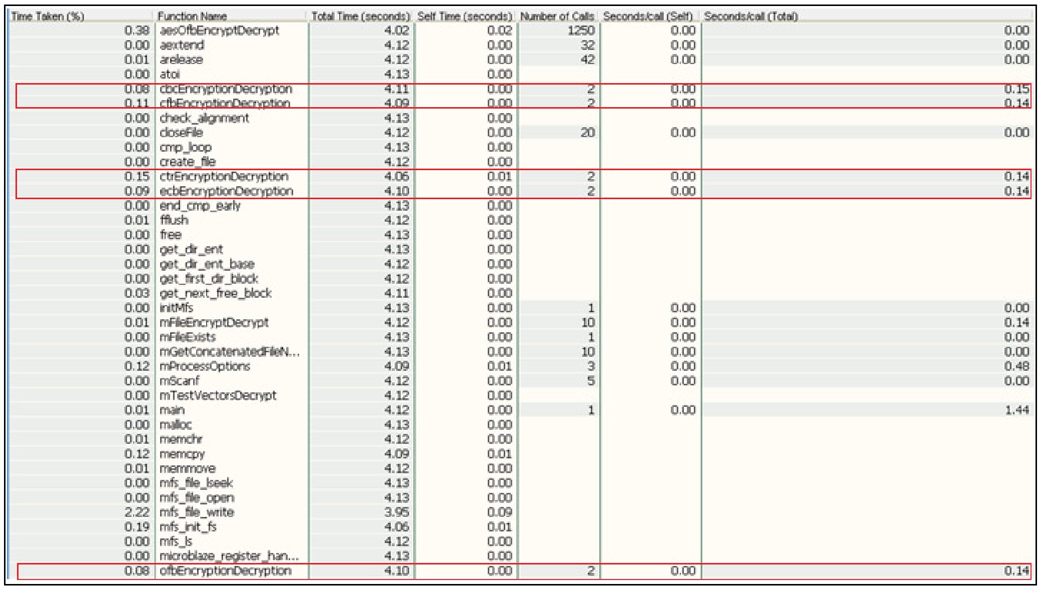
APPENDIX B – PROFILING RESTRICTIONS

The following restrictions apply when profiling in EDK:

- Profiling does not measure the time spent in interrupt handlers because interrupt handlers typically disable further interrupts from occurring. Therefore, it is impossible for profiling interrupts to occur when the program is executing an interrupt handler.
- Profiling can only be done with the standalone platform; it cannot be done in the presence of an OS. This is because the profiling libraries are only available in the standalone BSP.
- Recursive functions are not supported.
- If the timer is directly connected to the processor (for example, when there is no interrupt controller), the software application requires additional setup to support profiling.
- The call graph for functions inside C and Math libraries (libc and libm) are not generated because these libraries are not compiled with the -pg compiler profiling option.
- Ensure that memory used for collecting profile data is not used by any other function in the application.
- Profiling cannot be done while debugging. Enable profiling only when selecting the Run configuration in SDK (Xilinx, 2008).

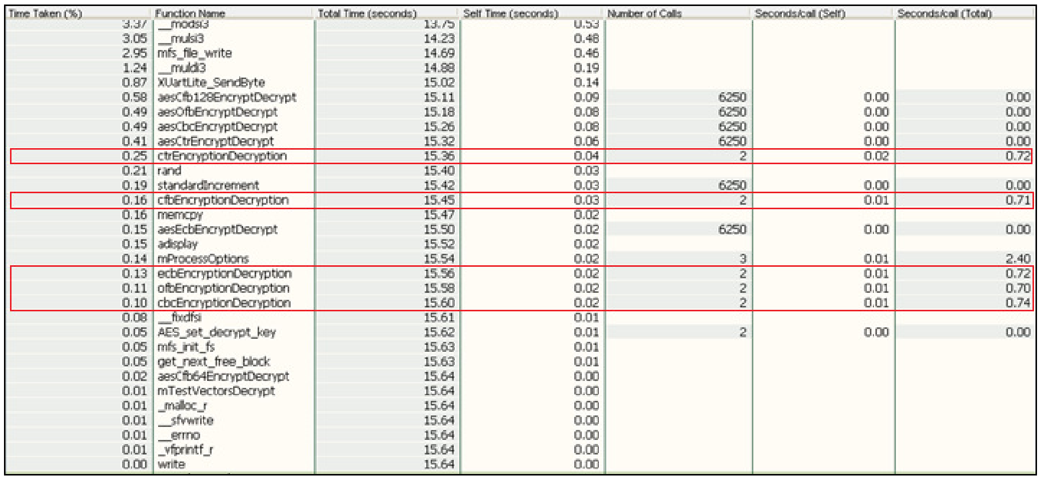
APPENDIX C – PROFILER RESULTS OF RANDOMLY GENERATED DATA

In chapter five section 5.3.2, the informative graph is constructed from "Flat Profile View" results shown below:



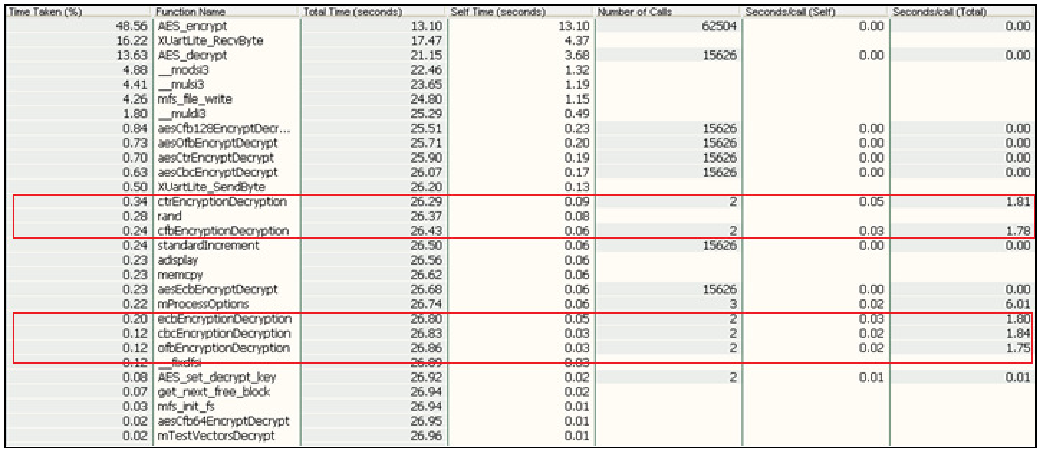
Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Seconds/call (Self)	Seconds/call (Total)
0.38	aesOfbEncryptDecrypt	4.02	0.02	1250	0.00	0.00
0.00	aextend	4.12	0.00	32	0.00	0.00
0.01	arelease	4.12	0.00	42	0.00	0.00
0.00	atoi	4.13	0.00			
0.08	cbcEncryptionDecryption	4.11	0.00	2	0.00	0.15
0.11	cfbEncryptionDecryption	4.09	0.00	2	0.00	0.14
0.00	check_alignment	4.13	0.00			
0.00	closeFile	4.12	0.00	20	0.00	0.00
0.00	cmp_loop	4.13	0.00			
0.00	create_file	4.12	0.00			
0.15	ctrEncryptionDecryption	4.06	0.01	2	0.00	0.14
0.09	ecbEncryptionDecryption	4.10	0.00	2	0.00	0.14
0.00	end_cmp_early	4.13	0.00			
0.01	fflush	4.12	0.00			
0.00	free	4.13	0.00			
0.00	get_dir_ent	4.13	0.00			
0.00	get_dir_ent_base	4.12	0.00			
0.00	get_first_dir_block	4.12	0.00			
0.03	get_next_free_block	4.11	0.00			
0.00	initMfs	4.13	0.00	1	0.00	0.00
0.01	mFileEncryptDecrypt	4.12	0.00	10	0.00	0.14
0.00	mFileExists	4.13	0.00	1	0.00	0.00
0.00	mGetConcatenatedFileN...	4.13	0.00	10	0.00	0.00
0.12	mProcessOptions	4.09	0.01	3	0.00	0.48
0.00	mScanf	4.12	0.00	5	0.00	0.00
0.00	mTestVectorsDecrypt	4.12	0.00			
0.01	main	4.12	0.00	1	0.00	1.44
0.00	malloc	4.13	0.00			
0.01	memchr	4.12	0.00			
0.12	memcpy	4.09	0.01			
0.01	memmove	4.12	0.00			
0.00	mfs_file_lseek	4.13	0.00			
0.00	mfs_file_open	4.13	0.00			
2.22	mfs_file_write	3.95	0.09			
0.19	mfs_init_fs	4.06	0.01			
0.00	mfs_ls	4.12	0.00			
0.00	microblaze_register_han...	4.13	0.00			
0.08	ofbEncryptionDecryption	4.10	0.00	2	0.00	0.14

Figure C.1 Randomly generated data, 10000 bytes.



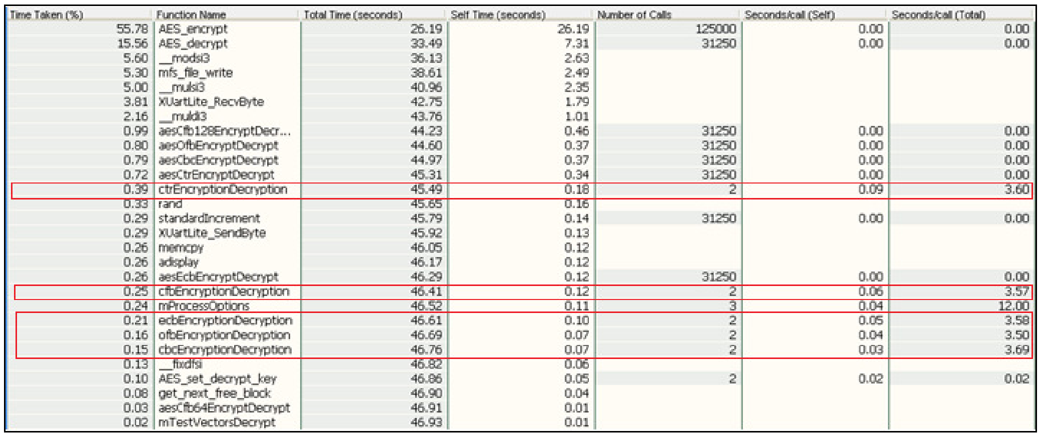
Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Seconds/call (Self)	Seconds/call (Total)
3.37	__modsi3	13.75	0.53			
3.05	__mulsi3	14.23	0.48			
2.95	mfs_file_write	14.69	0.46			
1.24	__muldi3	14.88	0.19			
0.87	XUartLite_SendByte	15.02	0.14			
0.58	aesCfb128EncryptDecrypt	15.11	0.09	6250	0.00	0.00
0.49	aesOfbEncryptDecrypt	15.18	0.08	6250	0.00	0.00
0.49	aesCbcEncryptDecrypt	15.26	0.08	6250	0.00	0.00
0.41	aesCtrEncryptDecrypt	15.32	0.06	6250	0.00	0.00
0.25	ctrEncryptionDecryption	15.36	0.04	2	0.02	0.72
0.21	rand	15.40	0.03			
0.19	standardIncrement	15.42	0.03	6250	0.00	0.00
0.16	cfbEncryptionDecryption	15.45	0.03	2	0.01	0.71
0.16	memcpy	15.47	0.02			
0.15	aesEcbEncryptDecrypt	15.50	0.02	6250	0.00	0.00
0.15	adisplay	15.52	0.02			
0.14	mProcessOptions	15.54	0.02	3	0.01	2.40
0.13	ecbEncryptionDecryption	15.56	0.02	2	0.01	0.72
0.11	ofbEncryptionDecryption	15.58	0.02	2	0.01	0.70
0.10	cbcEncryptionDecryption	15.60	0.02	2	0.01	0.74
0.08	__fixdfsi	15.61	0.01			
0.05	AES_set_decrypt_key	15.62	0.01	2	0.00	0.00
0.05	mfs_init_fs	15.63	0.01			
0.05	get_next_free_block	15.63	0.01			
0.02	aesCfb64EncryptDecrypt	15.64	0.00			
0.01	mTestVectorsDecrypt	15.64	0.00			
0.01	_malloc_r	15.64	0.00			
0.01	__sfvwrite	15.64	0.00			
0.01	__errno	15.64	0.00			
0.01	_vfprintf_r	15.64	0.00			
0.00	write	15.64	0.00			

Figure C.2 Randomly generated data, 50000 bytes.



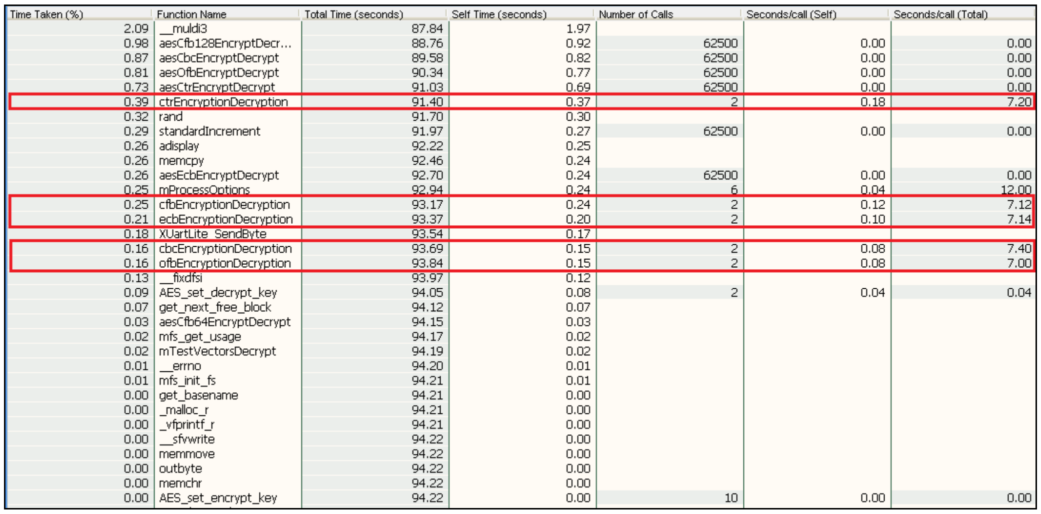
Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Seconds/call (Self)	Seconds/call (Total)
48.56	AES_encrypt	13.10	13.10	62504	0.00	0.00
16.22	XUartLite_RecvByte	17.47	4.37			
13.63	AES_decrypt	21.15	3.68	15626	0.00	0.00
4.88	__modsi3	22.46	1.32			
4.41	__mulsi3	23.65	1.19			
4.26	mfs_file_write	24.80	1.15			
1.80	__muldi3	25.29	0.49			
0.84	aesCfb128EncryptDecr...	25.51	0.23	15626	0.00	0.00
0.73	aesOfbEncryptDecrypt	25.71	0.20	15626	0.00	0.00
0.70	aesCtrEncryptDecrypt	25.90	0.19	15626	0.00	0.00
0.63	aesCbcEncryptDecrypt	26.07	0.17	15626	0.00	0.00
0.50	XUartLite_SendByte	26.20	0.13			
0.34	ctrEncryptionDecryption	26.29	0.09	2	0.05	1.81
0.28	rand	26.37	0.08			
0.24	cfbEncryptionDecryption	26.43	0.06	2	0.03	1.78
0.24	standardIncrement	26.50	0.06	15626	0.00	0.00
0.23	adisplay	26.56	0.06			
0.23	memcpy	26.62	0.06			
0.23	aesEcbEncryptDecrypt	26.68	0.06	15626	0.00	0.00
0.22	mProcessOptions	26.74	0.06	3	0.02	6.01
0.20	ecbEncryptionDecryption	26.80	0.05	2	0.03	1.80
0.12	cbcEncryptionDecryption	26.83	0.03	2	0.02	1.84
0.12	ofbEncryptionDecryption	26.86	0.03	2	0.02	1.75
0.12	__fixdfsi	26.89	0.03			
0.08	AES_set_decrypt_key	26.92	0.02	2	0.01	0.01
0.07	get_next_free_block	26.94	0.02			
0.03	mfs_init_fs	26.94	0.01			
0.02	aesCfb64EncryptDecrypt	26.95	0.01			
0.02	mTestVectorsDecrypt	26.96	0.01			

Figure C.3 Randomly generated data, 125000 bytes.



Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Seconds/call (Self)	Seconds/call (Total)
55.78	AES_encrypt	26.19	26.19	125000	0.00	0.00
15.56	AES_decrypt	33.49	7.31	31250	0.00	0.00
5.60	__modsi3	36.13	2.63			
5.30	mfs_file_write	38.61	2.49			
5.00	__mulsi3	40.96	2.35			
3.81	XUartLite_RecvByte	42.75	1.79			
2.16	__muldi3	43.76	1.01			
0.99	aesCfb128EncryptDecr...	44.23	0.46	31250	0.00	0.00
0.80	aesOfbEncryptDecrypt	44.60	0.37	31250	0.00	0.00
0.79	aesCbcEncryptDecrypt	44.97	0.37	31250	0.00	0.00
0.72	aesCtrEncryptDecrypt	45.31	0.34	31250	0.00	0.00
0.39	ctrEncryptionDecryption	45.49	0.18	2	0.09	3.60
0.33	rand	45.65	0.16			
0.29	standardIncrement	45.79	0.14	31250	0.00	0.00
0.29	XUartLite_SendByte	45.92	0.13			
0.26	memcpy	46.05	0.12			
0.26	adisplay	46.17	0.12			
0.26	aesEcbEncryptDecrypt	46.29	0.12	31250	0.00	0.00
0.25	cfbEncryptionDecryption	46.41	0.12	2	0.06	3.57
0.24	mProcessOptions	46.52	0.11	3	0.04	12.00
0.21	ecbEncryptionDecryption	46.61	0.10	2	0.05	3.58
0.16	ofbEncryptionDecryption	46.69	0.07	2	0.04	3.50
0.15	cbcEncryptionDecryption	46.76	0.07	2	0.03	3.69
0.13	__fixdfsi	46.82	0.06			
0.10	AES_set_decrypt_key	46.86	0.05	2	0.02	0.02
0.08	get_next_free_block	46.90	0.04			
0.03	aesCfb64EncryptDecrypt	46.91	0.01			
0.02	mTestVectorsDecrypt	46.93	0.01			

Figure C.4 Randomly generated data, 250000 bytes.



Time Taken (%)	Function Name	Total Time (seconds)	Self Time (seconds)	Number of Calls	Seconds/call (Self)	Seconds/call (Total)
2.09	__muldi3	87.84	1.97			
0.98	aesCfb128EncryptDecr...	88.76	0.92	62500	0.00	0.00
0.87	aesCbcEncryptDecrypt	89.58	0.82	62500	0.00	0.00
0.81	aesOfbEncryptDecrypt	90.34	0.77	62500	0.00	0.00
0.73	aesCtrEncryptDecrypt	91.03	0.69	62500	0.00	0.00
0.39	ctrEncryptionDecryption	91.40	0.37	2	0.18	7.20
0.32	rand	91.70	0.30			
0.29	standardIncrement	91.97	0.27	62500	0.00	0.00
0.26	adisplay	92.22	0.25			
0.26	memcpy	92.46	0.24			
0.26	aesEcbEncryptDecrypt	92.70	0.24	62500	0.00	0.00
0.25	mProcessOptions	92.94	0.24	6	0.04	12.00
0.25	cfbEncryptionDecryption	93.17	0.24	2	0.12	7.12
0.21	ecbEncryptionDecryption	93.37	0.20	2	0.10	7.14
0.18	XUartLite_SendByte	93.54	0.17			
0.16	cbcEncryptionDecryption	93.69	0.15	2	0.08	7.40
0.16	ofbEncryptionDecryption	93.84	0.15	2	0.08	7.00
0.13	__fixdfsi	93.97	0.12			
0.09	AES_set_decrypt_key	94.05	0.08	2	0.04	0.04
0.07	get_next_free_block	94.12	0.07			
0.03	aesCfb64EncryptDecrypt	94.15	0.03			
0.02	mfs_get_usage	94.17	0.02			
0.02	mTestVectorsDecrypt	94.19	0.02			
0.01	__errno	94.20	0.01			
0.01	mfs_init_fs	94.21	0.01			
0.00	get_basename	94.21	0.00			
0.00	_malloc_r	94.21	0.00			
0.00	_vfprintf_r	94.21	0.00			
0.00	__sfvwrite	94.22	0.00			
0.00	memmove	94.22	0.00			
0.00	outbyte	94.22	0.00			
0.00	memchr	94.22	0.00			
0.00	AES_set_encrypt_key	94.22	0.00	10	0.00	0.00

Figure C.5 Randomly generated data, 500000 bytes.