UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Maurício Wurthmann Saad

Implementação do algoritmo AES em hardware reconfigurável - FPGA

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciências em Ciência e Tecnologia da Computação

Área de Concentração:
Sistemas de computação: Hardware e Software Básico

Orientador: Prof. Robson Luiz Moreno, Dr.

Co-orientador: Prof. Leonardo Mesquita, Dr.

Dezembro de 2010

Itajubá - MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

S111i

Saad, Maurício Wurthmann
Implementação do algoritmo AES em hardware reconfigurá_
vel - FPGA / Maurício Wurthmann Saad. -- Itajubá, (MG) : [s.n.],
2010.
143 p. : il.

Orientador: Prof. Dr. Robson Luiz Moreno.
Coorientador: Prof. Dr. Leonardo Mesquita.
Dissertação (Mestrado) – Universidade Federal de Itajubá.

1. Criptografia. 2. AES. 3. VHDL. 4. FPGA. I. Moreno, Rob_
son Luiz, orient. II. Mesquita, Leonardo, coorient. III. Universi_
dade Federal de Itajubá. IV. Título.

Maurício Wurthmann Saad

Implementação do algoritmo criptográfico AES em hardware reconfigurável - FPGA

Dissertação apresentada ao Instituto de Engenharia de Sistemas e Tecnologias da Informação (IESTI) e Instituto de Ciências Exatas (ICE) da Universidade Federal de Itajubá - UNIFEI, como parte dos requisitos para obtenção do título de Mestre em Ciência e Tecnologia da Computação.

BANCA EXAMINADORA:

Prof. Robson Luiz Moreno, Dr.
Orientador – UNIFEI

Prof. Leonardo Mesquita, Dr.
Co-orientador – FEG-UNESP

Prof. Samuel Euzedice de Lucena, Dr.
FEG-UNESP

Prof. Paulo César Crepaldi, Dr.
UNIFEI

Dedico este trabalho aos meus pais, Salvador Saad (in memoriam) e
Ermgarht Luiza Wurthmann Saad, que sempre me incentivaram
a estudar e aos quais devo tudo que tenho.

Agradecimentos

Ao prof. Dr. Leonardo Mesquita que, ao me apresentar ao prof. Dr. Robson Luiz Moreno, abriu-me efetivamente o caminho para iniciar esse trabalho, e se dispôs a ser meu co-orientador, contribuindo de forma decisiva para elaboração desta dissertação, compreendendo minhas limitações e me ajudando a superá-las.

Ao meu orientador, prof. Dr. Robson Luiz Moreno, pela paciência e tolerância com os meus percalços ao logo desta caminhada.

Ao prof. Dr. Galdenoro Botura Júnior que sempre me incentivou a fazer o mestrado.

À minha esposa, Valéria, que mesmo com todos os seus compromissos profissionais, seu curso de especialização e mãe de três filhos, se desdobrou para atender a mim e às crianças. À ela e aos nossos filhos, Gabriel, Beatriz e Luísa, pelo apoio dado e por compreenderem e suportarem minha ausência e minha falta de participação em diversos momentos familiares.

Às minhas amigas e colegas de trabalho, Valéria Cristina Martins Ferraz e Denise Aparecida da Silva Garcia Pereira, pelo empenho ainda maior no serviço, para que eu pudesse frequentar as aulas e os compromissos do mestrado em Itajubá.

Aos demais professores do mestrado, em especial ao prof. Dr. Edmilson Marmo, que apesar das grandes dificuldades encontradas como coordenador do programa de mestrado, desta primeira turma, enfrentou as adversidades e nos fez chegar até o fim.

E a dois novos amigos, Daniela Bretas e Glauco da Silva, pelo companheirismo, incentivo e apoio ao longo desta jornada.

“Se você tem uma maçã e eu tenho uma maçã e nós trocamos nossas maçãs, então eu e você ainda teremos uma maçã. Mas se você tem uma ideia e eu tenho uma ideia e nós trocamos estas ideias, então cada um de nós terá duas ideias.”

(George Bernard Shaw)

Resumo

Neste projeto de pesquisa realizou-se a implementação do algoritmo criptográfico AES em hardware reconfigurável, utilizando-se da linguagem de programação VHDL. Inicialmente, o modelo VHDL de todas as funções constituintes foram desenvolvidas e posteriormente sintetizadas no componente EP2C20F484C7 da família Cyclone II da Altera. A seguir o algoritmo de criptografia AES, tendo uma chave de 128 bits, foi implementado e validado via simulação. Como última etapa de desenvolvimento foram gerados códigos parametrizados que possibilitam ao usuário definir se o modelo criptográfico irá operar com chaves de 128, 192 ou 256 bits.

Palavras-chave: Criptografia, AES, VHDL, FPGA

Abstract

In this research project dealt with on the implementation of AES cryptographic algorithm on reconfigurable hardware, using the VHDL programming language. Initially, the VHDL model of all the constituent functions were developed and subsequently summarized in component EP2C20F484C7 Cyclone II family from the Altera. Then the AES encryption algorithm, with a 256-bit key was implemented and validated by simulation. As the final stage of development were generated parameterized codes that allow the user to define whether the model will operate with cryptographic keys of 128, 192 or 256 bits.

Keywords: Criptography, AES, VHDL, FPGA

Lista de figuras

Figura 2.1 - Esquema geral da criptografia de um texto 4
Figura 2.2 - A Máquina Enigma 6
Figura 2.3 - Esquema geral de criptografia com chave 8
Figura 2.4 - Código de Mary, Rainha da Escócia 8
Figura 2.5 - Código Pigpen 9
Figura 2.6 - Cifra de César 10
Figura 2.7 - Modelo simétrico de criptografia 11
Figura 2.8 - Modelo assimétrico de criptografia 11
Figura 2.9 - Geração de assinatura digital 12
Figura 2.10 - Representação da criação de 52 subchaves do IDEA 21
Figura 3.1 - ***State*** 27
Figura 3.2 - Relação dos blocos de entrada, saída e ***state*** 28
Figura 3.3 - Visão geral do AES 29
Figura 3.4 - Pseudocódigo para o cifrador 31
Figura 3.5 - Transformação ***SubByte*** do AES 32
Figura 3.6 - Transformação ***SubBytes()*** usando a S-Box 32
Figura 3.7 - Transformação ***ShiftRows()*** para ***Nb***=4 34
Figura 3.8 - Transformação baseada na matriz de multiplicação para o AES 35
Figura 3.9 – Transformação ***MixColumns()***... 35
Figura 3.10 - Transformação ***AddRoundKey()***... 36
Figura 3.11 - Vetor de chaves para ***Nb***=4 e ***Nk***=4 36
Figura 3.12 - Pseudocódigo para o decifrador 38
Figura 3.13 - Transformação ***InvShiftRows()*** para ***Nb***=4 39
Figura 3.14 - Transformação ***InvSubBytes()*** usando a S-Box invertida 40
Figura 3.15 - Transformação baseada na matriz de multiplicação para o AES 41
Figura 3.16 - Transformação ***InvMixColumns()*** 41
Figura 4.1 - (a) Circuitos integrados variados. (b) Detalhe da estrutura interna de um circuito integrado 42
Figura 4.2 - Célula padrão utilizada em projeto *semi-custom* 44
Figura 4.3 - Circuito básico de um PLD 45
Figura 4.4 - Detalhe de um esquemático utilizando a simbologia simplificada para PLD’s 46
Figura 5.1 - Fluxograma de execução do AES 50
Figura 5.2 - Componente ***KeyExpansion*** 53
Figura 5.3 - Código VHDL da ***KeyExpansion*** (constante e funções) 54
Figura 5.4 - Código VHDL da ***KeyExpansion*** (chave de 128 bits) 55
Figura 5.5 - Código VHDL da ***KeyExpansion*** (chave de 256 bits) 56
Figura 5.6 - Simulação da expansão da chave de 128 bits 57
Figura 5.7 - Simulação da expansão da chave de 192 bits 58
Figura 5.8 - Simulação da expansão da chave de 256 bits 58
Figura 5.9 - Componente ***AddRoundKey*** 59
Figura 5.10 - Código VHDL da ***AddRoundKey*** 60
Figura 5.11 - Simulação do ***AddRoundKey*** 61
Figura 5.12 - Componente ***SubBytes*** 61
Figura 5.13 - Código VHDL da ***SubBytes*** 62

Figura 5.14 - Simulação do ***SubBytes*** 62
Figura 5.15 - Componente ***ShifRows*** 63
Figura 5.16 - Código VHDL da ***ShiftRows*** 63
Figura 5.17 - Simulação do ***ShiftRows*** 64
Figura 5.18 - Componente ***MixColumns*** 65
Figura 5.19 - Código VHDL das funções da ***MixColumns*** 66
Figura 5.20 - Código VHDL da ***MixColumns*** 67
Figura 5.21 - Simulação do ***MixColumns*** 67
Figura 5.22 - Entradas e saídas 68
Figura 5.23 - Trecho de código VHDL do “Estado_0” 69
Figura 5.24 - Trecho de código VHDL do “Estado_1” 70
Figura 5.25 - Trecho de código VHDL de atualização de sinais internos 71
Figura 5.26 - Resultado de uma criptografia completa com chave de 128 bits 72

Lista de tabelas

Tabela 2.1 - Comparação entre algoritmos quanto à chave simétrica ou assimétrica 13
Tabela 2.2 - Custo para encontrar a chave por força 14
Tabela 2.3 - Principais algoritmos assimétricos 14
Tabela 2.4 - Características de algoritmos de hashing 15
Tabela 2.5 - Características dos algoritmos simétricos mais conhecidos 16
Tabela 2.6 - Comparativo entre alguns algoritmos simétricos e assimétricos 17
Tabela 3.1 - Tabela de logaritmos para multiplicação 26
Tabela 3.2 - Relação de Nr em função de Nb e Nk 28
Tabela 3.3 - Parâmetros do AES 28
Tabela 3.4 - S-Box: substituição dos valores do byte xy (em formato hexadecimal) 33
Tabela 3.5 - Constante de deslocamento de bytes em função do tamanho do bloco ***Nb*** e da linha ***r*** 34
Tabela 3.6 - Rcon(i) – constante de rodada da expansão de chaves 37
Tabela 3.7 - S-Box inversa: substituição dos valores do byte xy (em formato hexadecimal) 40
Tabela 5.1 - Características do FPGA da família Cyclone II 49

Lista de abreviaturas e siglas

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CI	Circuito Integrado
CMOS	Complementary Metal Oxide Silicon
DES	Digital Encryption Standard
DH	Diffie, Hellman
EPLD	Electrically Programmable Logic Device
FBI	Federal Bureau of Investigation
FIPS-197	Federal Information Processing Standards Publication 197 (FIPS-PUB-197)
FIPS-46-3	Federal Information Processing Standards Publication 46-3 (FIPS-PUB-46-3). Data Encryption Standard (DES), October 1999.
FPGA	Field Programmable Gate Array
GF	Galois Finite Field
IBM	International Business Machine
IDEA	International Data Encryption Algorithm
IEEE	Institute of Electrical and Electronic Engineering
ITU	International Telecommunication Union
LSI	Large Scale Integration
MIT	Massachusetts Institute of Technology
MSI	Medium Scale Integration
NBS	National Bureau of Standards
NIST	National Institute of Standards and Technology
NSA	National Security Agency
ONU	Organização das Nações Unidas
PAL	Programmable Array Logic
PGP	Pretty Good Privacy
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PLE	Programmable Logic Element
PROM	Programmable Read-Only Memory
RC3,4,5,6	Ron’s Cipher
RSA	Rivest-Shamir-Adleman
SSI	Small Scale Integration
SSL	Secure Sockets Layer
TTL	Transistor-Transistor Logic
TLS	Transport Layer Security
UIT	União Internacional de Telecomunicações
VHSIC	Very High-Speed Integrated Circuit
VLSI	Very Large Scale Integration
XOR	Exclusive OR

Lista de símbolos

$\oplus$	XOR - ou exclusivo
$\bullet$	multiplicação dentro de um corpo finito (GF).

Sumário

Capítulo 1 - Introdução 1
1.1 - Motivação 1
1.2 - Objetivo 2
1.3 - Estrutura de trabalho 2

Capítulo 2 - Criptografia 4
2.1 - Definição 4
2.2 - Histórico e evolução 5
2.3 - Métodos de criptografia 7
2.3.1 - Importância do tamanho da chave 13
2.3.2 - Principais algoritmos assimétricos 14
2.3.2.1 - Função de hashing 15
2.3.3 - Principais algoritmos simétricos 16
2.3.4 - Comparativo entre alguns algoritmos 16
2.4 - RSA (Rivest, Shamir e Adleman) 18
2.5 - DES (Data Encryption Standard) 19
2.6 - 3DES ou Triple-DES ou TDEA (Triple Data Encryption Algorithm) 20
2.7 - IDEA 20
2.8 - Rivest Ciphers (RC2; RC4 e RC5) 21
2.9 - Advanced Encryption Standard (AES) 22

Capítulo 3 - Algoritmo AES (Advanced Encryption Standard) (Rijndael) 25
3.1 - Operações 25
3.2 - Estrutura do AES 27
3.3 - Cifrador 30
3.3.1 - Transformação SubBytes() 31
3.3.2 - Transformação ShiftRows() 33
3.3.3 - Transformação MixColumns() 34
3.3.4 - Transformação AddRoundKey 36
3.3.5 - Geração de subchaves 36
3.4 - Decifrador 37
3.4.1 - Transformação InvShiftRows() 39
3.4.2 - Transformação InvSubBytes() 39
3.4.3 - Transformação InvMixColumns() 40

Capítulo 4 - Circuitos integrados e linguagem VHDL 42
4.1 - Dispositivos Lógicos Programáveis (PLD) 44
4.2 - Ideia básica de um PLD 45
4.3 - Simbologia 46
4.4 - Características dos dispositivos lógicos programáveis 47
4.5 - Linguagem de programação VHDL 48

Capítulo 5 – Resultados 49
5.1 - Considerações iniciais 49
5.2 - Visão geral do processamento 49
5.3 - Módulos desenvolvidos 52

5.3.1 - Expansão da chave (KeyExpansion) 52
5.3.2 - Adição de chave (AddRoundKey) 59
5.3.3 - Substituição de bytes (SubBytes) 61
5.3.4 - Deslocamento de linhas (ShiftRows) 63
5.3.5 - Mistura de colunas (MixColumns) 64
5.4 - Entradas e saídas 68
5.5 - Unidade de controle 68

Capítulo 6 – Conclusões e trabalhos futuros 73

Referências Bibliográficas 74

Anexos 77

Documento FIPS-197 77

CAPÍTULO 1

1.1 - MOTIVAÇÃO

Conforme estimativa da União Internacional de Telecomunicações (UIT), do inglês *International Telecommunication Union (ITU)*, entidade integrante da Organização das Nações Unidas (ONU), o número de usuários da internet deve superar a marca de 2 bilhões de pessoas até o final deste ano de 2010, o que significa que um terço da população mundial estará *online* [1].

Com a crescente integração dos equipamentos via rede e o uso cada vez maior de dispositivos eletrônicos para comunicações, pesquisas, comércio eletrônico, armazenamento de dados entre outras aplicações, a segurança da informação passa a ter, ainda mais, papel relevante no dia-a-dia.

Muitos problemas podem ocorrer se a privacidade de pessoas ou empresas for violada, como por exemplo, o acesso não autorizado a informações pessoais tais como: demonstrativo de pagamento, faturas de cartão de crédito, diagnósticos de saúde ou senhas bancárias. Para as empresas, os prejuízos podem ser ainda maiores, caso uma concorrente tenha acesso não autorizado a dados estratégicos, como novos lançamentos, detalhes técnicos de produtos, resultados de pesquisas, estimativas de vendas e investimentos ou mesmo informações pessoais dos funcionários, isto pode resultar até mesmo na falência da empresa [2].

Antigamente, a segurança era obtida trancando-se a porta de uma sala ou cofre, pois a grande maioria das informações estava em papéis. Atualmente como grande parte das informações está armazenada em meio eletrônico, e como o volume de informações transmitidas pela internet é cada vez maior, acentuou-se a necessidade de outros mecanismos de proteção. Uma das técnicas para isto é a criptografia.

A criptografia utiliza-se de técnicas e métodos para transformar uma informação original que está num formato compreensível, em outro que é incompreensível, e que posteriormente possa ser convertida novamente em compreensível para os que tiverem direito em acessá-la. O primeiro relato de processos criptográficos data de 1900 A.C [3], no Egito, que consistia na substituição de trechos de um documento, com a intenção de proteger o tesouro de Khnumhotep II. Ao longo dos tempos houve evolução nos processos criptográficos especialmente a partir da 2ª Guerra mundial. Dentre os principais algoritmos da atualidade,

encontra-se o *Advanced Encryption Standard* (AES) que é adotado oficialmente como padrão pelo governo americano, em função do seu alto grau de confiabilidade, e é amplamente utilizado em diversas aplicações em todo mundo, como troca de informações entre os órgãos do governo, criptografia de informações diversas como dados em meio eletrônico, disco rígido de computadores, equipamentos wireless compatíveis com o padrão 802.11i (protocolo WPA2), certificados pela Wi-Fi Alliance, entre outros. Por este motivo, este foi o algoritmo escolhido para estudo nesta dissertação.

1.2 - OBJETIVO

Este trabalho tem como objetivo principal implementar o algoritmo criptográfico AES com possibilidade do usuário programar o tamanho da chave utilizada, dentre as possíveis permitidas para este algoritmo. A implementação foi feita em hardware ao invés de software, utilizando-se como alvo um componente reconfigurável, "*Field Programmable Gate Array* (FPGA), e a linguagem VHDL, originada do projeto "*VHSIC Hardware Description Language*" do Departamento de Defesa do governo Americano. VHSIC é o acrônimo de *Very High Speed Integrated Circuits.*

1.3 - ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta um estudo sobre criptografia, e sobre algumas formas de cifragem desenvolvidas desde a antiguidade até os dias atuais, ressaltando-se os principais algoritmos e métodos criptográficos e os fatos mais importantes historicamente.

O Capítulo 3 descreve a estrutura do algoritmo de criptografia AES, descrevendo as funções que compõem o algoritmo bem como as suas principais características fundamentadas principalmente nos diferentes tamanhos de chave e blocos possíveis de uso no mesmo.

O Capítulo 4 apresenta uma breve revisão de elementos de lógica reconfigurável e características básicas da linguagem de programação VHDL.

No capítulo 5, o projeto dos blocos necessários do algoritmo AES modelados em VHDL é apresentado. O resultado de síntese destes blocos, tendo como alvo o componente EP2C20F484C7 da família Cyclone II da Altera, é apresentado e discutido. O modelo VHDL do algoritmo AES para chave de 128 bits é apresentado juntamente com a simulação, ambos são discutidos.

O Capítulo 6 apresenta as conclusões do trabalho além de sugestões de trabalhos futuros nesta linha de pesquisa.

CAPÍTULO 2

A segurança, antigamente, baseava-se em trancar uma porta ou um cofre. Atualmente, como grande parte da informação está em meio eletrônico, como em computadores, servidores de banco de dados, palmtops, notebooks, agendas eletrônicas, ou trafegando pela internet, como no caso do comércio eletrônico, foram necessários novos procedimentos para garantir o sigilo e segurança destes dados, tendo em vista que a violação das informações, tanto empresarias como pessoais, pode ter consequências gravíssimas, como no caso da violação de contas bancárias, em que um invasor pode transferir fundos indevidamente, a invasão e destruição de sistemas, o acesso a informações sigilosas afetando a privacidade, tais como dados pessoais, faturas de cartões de crédito, diagnósticos de saúde, informações estratégicas de empresas, detalhes técnicos de produtos, etc.

Atualmente a forma mais amplamente utilizada para garantir o sigilo das informações é a criptografia.

2.1 - DEFINIÇÃO

A criptografia é o estudo de técnicas e métodos para transformar uma informação original que está num formato compreensível, em outro que é incompreensível, podendo ser posteriormente convertida em formato compreensível novamente, por seu destinatário, por meio de um processo inverso, decriptografia, que recupera as informações originais [2] (figura 2.1).

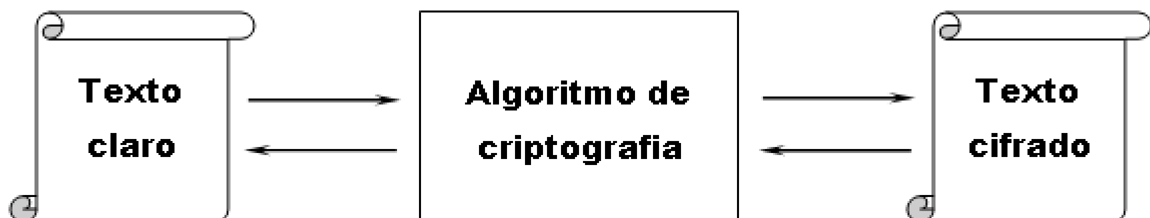

Figura 2.1: *Esquema geral da criptografia/decriptografia de um texto.*

Algoritmos de criptografia nada mais são do que sequências de procedimentos utilizados para cifrar e/ou decifrar dados, podendo ser executados manualmente ou de modo

automático como por exemplo com o auxílio de um programa de computador ou um hardware dedicado para tal tarefa.

A criptoanálise é o estudo das formas de reverter estas técnicas, quando não se é o destinatário, recuperando as informações originais ou forjando informações que serão aceitas como autênticas (no caso de assinatura digital). O criptoanalista procura pelas fraquezas do algoritmo [2, 4].

Alguns termos comuns à criptografia e que serão usados ao longo deste texto, com seus respectivos sinônimos, são apresentados a seguir [2]:

- Texto original, também chamado de texto claro, texto simples, texto legível, texto plano, refere-se à informação original, não codificada;
- Texto codificado, também chamado de texto ilegível, texto cifrado, texto código, texto criptografado ou cifra, refere-se à informação já criptografada;
- Criptografar, codificar, encriptar, cifrar, é o ato de transformar um texto legível em ilegível;
- Decodificar, decriptar, decriptografar, decifrar: é o ato de transformar um texto ilegível em legível, ou seja, representa a função inversa de cifrar;
- Chave é um conjunto único de caracteres que, associado ao algoritmo, permite realizar o processo de criptografia e/ou decriptografia.

2.2 - HISTÓRICO E EVOLUÇÃO

A criptografia é utilizada desde a antiguidade, é tão antiga quanto à própria escrita hieroglífica dos egípcios. Os romanos a utilizaram durante as batalhas para proteger suas mensagens. Ao longo dos anos diversas técnicas foram criadas. Como um breve histórico, podemos citar:

1900 a.C. - o primeiro exemplo documentado de criptografia, segundo Kahn [3], data de 1900 a.C., sendo um documento em que trechos do texto foram substituídos, para proteger o tesouro de Khnumhotep II [2, 4].

50 a.C. - Cifra de César, em que Júlio César alterava letras desviando-as em três posições, “A” se tornava “D”, “B” se tornava “E”, etc. Atualmente, qualquer cifra baseada nesta ideia (substituição cíclica) é denominada de cifra de César. Esta técnica foi utilizada pelos sulistas na Guerra de Secessão americana (1861 a 1865) e pelo exército russo em 1915.

1933 a 1945 - o aperfeiçoamento da Máquina Enigma (Figura 2.2), de Arthur Scherbius, tornou-se a ferramenta criptográfica mais importante da Alemanha nazista. No início da

segunda guerra os poloneses capturaram uma destas máquinas e após a invasão da Polônia pelos alemães, em 1939, o matemático polonês Marian Rejewski fugiu para Inglaterra e juntamente com os britânicos conseguiram quebrar o código da mesma. Os americanos conseguiram determinar o funcionamento das máquinas de codificação japonesa mesmo sem ter a posse de uma delas. Desta forma, todas as informações cifradas por elas ficaram expostas.

Figura 2.2: *A Máquina Enigma, utilizada na cifragem e decifragem de mensagens secretas.*

1943 - Máquina Colossus - projetada para quebrar códigos.

1976 - algoritmo Diffie-Hellman baseado no problema do logaritmo discreto.

1976 - a IBM apresenta a cifra Lucifer**,** desenvolvida por Feistel, que se tornou padrão de criptografia de dados nos EUA (FIPS46-3) [5], conhecida hoje como DES (Data Encryption Standard). O DES foi durante muito tempo a cifra mais utilizada para criptografia até que em julho de 1998 foi quebrada por força bruta (tentativa e erro) em 56 horas por pesquisadores do Vale do Silício [2, 6].

1977 - RSA - baseado na dificuldade da fatoração de números primos.

1990 - IDEA (International Data Encryption Algorithm), com chave de 128 bits; torna as implementações em software mais eficientes.

1991 - PGP (Pretty Good Privacy), disponibilizado como *freeware* em resposta às exigências do FBI de acessar qualquer texto claro da comunicação entre usuários de uma rede de comunicação digital. Tornou-se padrão mundial

1994 - RC5 - rotação dependente dos dados e é parametrizado de modo que o usuário possa variar o tamanho do bloco, o número de estágios e o comprimento da chave;

1994 - Blowfish - cifra de bloco de 64 bits e chave de até 448 bits;

1997 - PGP5 - *freeware*;

1997 - DES de 56 bits é quebrado por uma rede de 14.000 computadores;

1998 - DES é quebrado em 56 horas por pesquisadores do Vale do Silício;

1999 - DES é quebrado em 22 horas e 15 minutos por uma rede de 100.000 computadores pessoais ligado ao DES Cracker pela internet;

2000 - O algoritmo Rijndael, dos autores Rijmen e Daemen, foi oficializado pelo NIST (National Institute of Standards and Technology) como AES (Advanced Encryption Standard) para substituir o DES que havia tornado-se insuficiente para conter ataques de força bruta;

Atualmente - criptografia quântica, que ainda está em fase de testes [7].

2.3 - MÉTODOS DE CRIPTOGRAFIA

Como hoje a maior parte dos dados é digital, portanto representados por bits, o processo de criptografia é basicamente feito por algoritmos que embaralham os bits desses dados a partir de uma determinada chave ou par de chaves, dependendo do sistema criptográfico escolhido.

A maioria dos algoritmos modernos são embaralhadores de bits derivados de deslocamentos, substituições, permutações simples, operações matemáticas básicas e funções lógicas tipo “ou exclusivo”.

Na criptografia com chaves, conforme ilustrado na figura 2.3, o algoritmo utiliza-se de uma chave para cifrar o texto. Estes algoritmos são divulgados à comunidade, cabendo às chaves a garantia do sigilo da informação. Se alguém descobrir uma chave, terá acesso apenas às informações que foram cifradas com esta chave, as demais informações cifradas com o mesmo algoritmo, mas com chaves diferentes, estarão seguras. Como o algoritmo é público, a comunidade pode verificar sua força e sugerir melhorias, ou não utilizá-lo, se for fraco.

Figura 2.3: *Esquema geral de criptografia com chave.*

Um algoritmo que não utilize chaves deve ser mantido em sigilo, pois a quebra do algoritmo expõe todos os dados criptografados pelo mesmo, desta forma a comunidade não pode examiná-lo para identificar possíveis fraquezas que podem comprometer a segurança. É mais difícil manter um algoritmo em segredo do que uma chave e não se tem garantia que a empresa ou pessoa que o desenvolveu nunca o revelará.

A criptografia pode ser feita por meio de códigos ou cifras. A técnica por meio de códigos consiste na troca de parte da informação por códigos predefinidos, e para se ter acesso à informação é necessário conhecer os códigos utilizados.

No século XVI, Mary, a Rainha da Escócia e herdeira legítima do trono da Inglaterra, foi mantida prisioneira por conspirar pelo assassinato da Rainha Elizabeth I. Ela e seus aliados trocavam mensagens enviadas dentro de barris de cerveja inglesa, utilizando-se de uma código, como mostra a figura 2.4, que substituía cada letra e algumas palavras inteiras como "of" e "you", por um símbolo inventado. Utilizavam ainda, por todo o texto, quatro caracteres nulos, sem significado algum para confundir os criptoanalistas [8, 9].

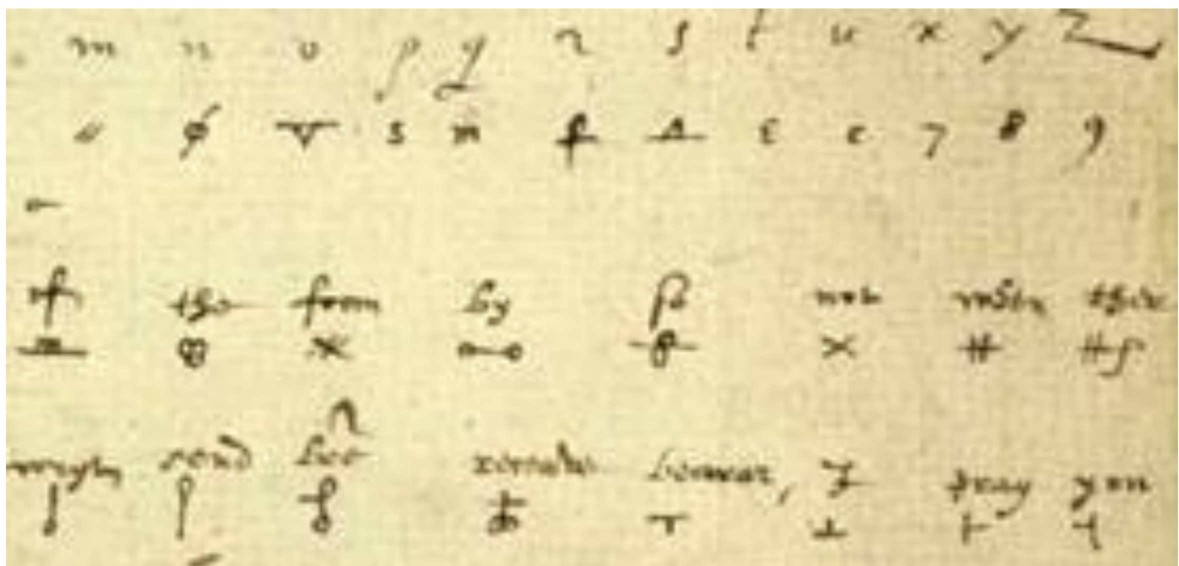

Figura 2.4: *código de Mary, Rainha da Escócia.*

Em 1533, Heinrich Cornelius Agrippa von Nettelsheim publica o “De occulta philosophia”, em Colônia, na Alemanha. No livro 3, capítulo 30, descreve seu código de substituição monoalfabética, hoje conhecida como cifra Pig Pen (“pigpen” é uma caneta usada para marcar porcos). A tradução literal deste código é “Porco no Chiqueiro” e vem do fato de que cada uma das letras (os porcos) é colocada numa "casa" (o chiqueiro). Este código foi utilizado pela maçonaria, no século XVIII, para manter seus registros privados, por isto também é conhecida como “cifra maçônica”. Existem algumas variações deste código considerando o alfabeto da idade média, o alfabeto alemão e o inglês. Consistia na troca de cada letra do alfabeto por um símbolo, como mostra a figura 2.5. Para criptografar uma mensagem basta substituir as letras da mensagem original pelo símbolo formado pelas linhas ao redor da letra, e o ponto, caso esteja desenhado próximo da letra. Assim, a frase:

“encontre me na ponte em quatro horas” seria transformada em:

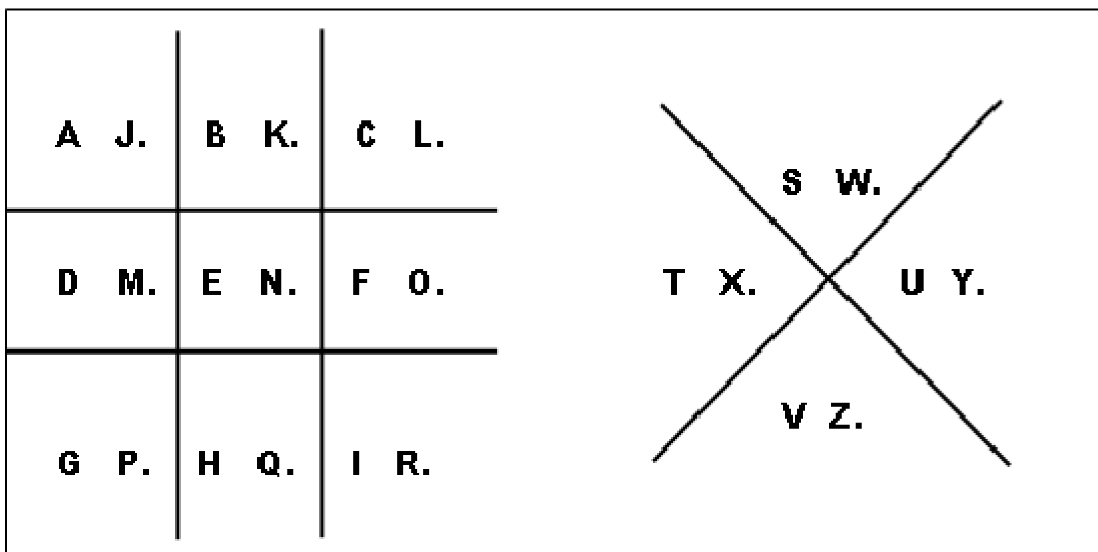

Figura 2.5: *Cifra Pigpen.*

A técnica por cifra consiste na transposição e/ou substituição das letras da informação original. Para se ter acesso à informação é necessário conhecer o processo de cifragem.

Na cifragem por transposição, os caracteres da informação original são misturados. Por exemplo, “CRIPTOGRAFIA” pode ser cifrado como “RPORFACITGAI”. Já na cifragem por substituição, utiliza-se uma tabela predefinida para trocar os caracteres da informação original

[2]. A figura 2.6 ilustra a substituição original do código de César, no qual cada letra da mensagem original é substituída pela letra que a segue em três posições no alfabeto.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Figura 2.6: *Cifra de César.*

Os algoritmos de criptografia podem trabalhar as informações em blocos ou em fluxo. No caso da cifra em blocos, a informação que será cifrada ou decifrada é dividida em blocos de tamanho fixo, normalmente em blocos de 8 a 16 bytes (64 a 128 bits) dependendo do algoritmo utilizado.

Com o avanço das técnicas de criptoanálise e as arquiteturas computacionais de 64 bits, os cifradores de blocos de 64 bits que eram grandes o suficiente para garantirem a segurança do algoritmo e pequenos o bastante para os computadores operarem estão sendo substituídos pelos cifradores de blocos de pelo menos 128 bits como, por exemplo, o AES [10].

Para que os cifradores tivessem uma boa resistência à criptoanálise estatística, Claude Shannon propôs, em sua teoria de 1949, a utilização de cifradores que alternassem substituições e permutações. Esta técnica, difundida por Feistel, consistia em duas características conhecidas como confusão e difusão que se tornaram os pilares dos cifradores de bloco modernos [4].

O princípio da confusão, obtida normalmente através de técnicas de substituição de pedaços da informação de entrada de modo que os bits de saída não possuam nenhuma relação óbvia com os de entrada, obscurece a relação entre o texto claro e o cifrado. Já a difusão, obtida normalmente por permutações e também substituições, elimina a redundância do texto claro, espalhando-a sobre o texto cifrado. A difusão propaga os efeitos de um bit do texto claro para vários bits no texto cifrado.

No caso de cifra de fluxo, os bits são criptografados continuamente, bit a bit através de uma operação XOR entre o bit de dados e o bit gerado pela chave. Um gerador de bits gera, a partir da chave inicial, uma sequência de bits chamados de "*keystream*" e que são utilizados no processo criptográfico.

Os algoritmos criptográficos podem utilizar a mesma chave tanto para cifrar como para decifrar, como pode ser observado na figura 2.7 em que a chave utilizada (chave A) para efetuar a cifragem é a mesma utilizada no processo de decifragem . Neste caso são chamados de algoritmos de chave simétrica ou de algoritmos de chave única.

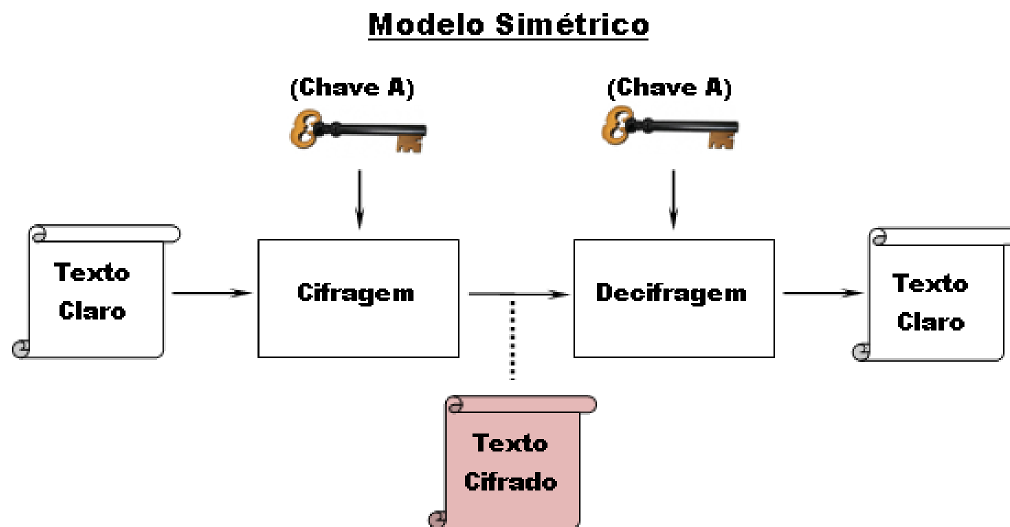

Figura 2.7: *Modelo simétrico de criptografia.*

Nos casos em que as chaves utilizadas são diferentes, como pode ser observado nas figuras 2.8 e 2.9, em que a chave utilizada para cifrar é diferente da chave utilizada na decifragem, o algoritmo é dito de chave assimétrica. Apesar das chaves serem diferentes, elas são matematicamente relacionadas.

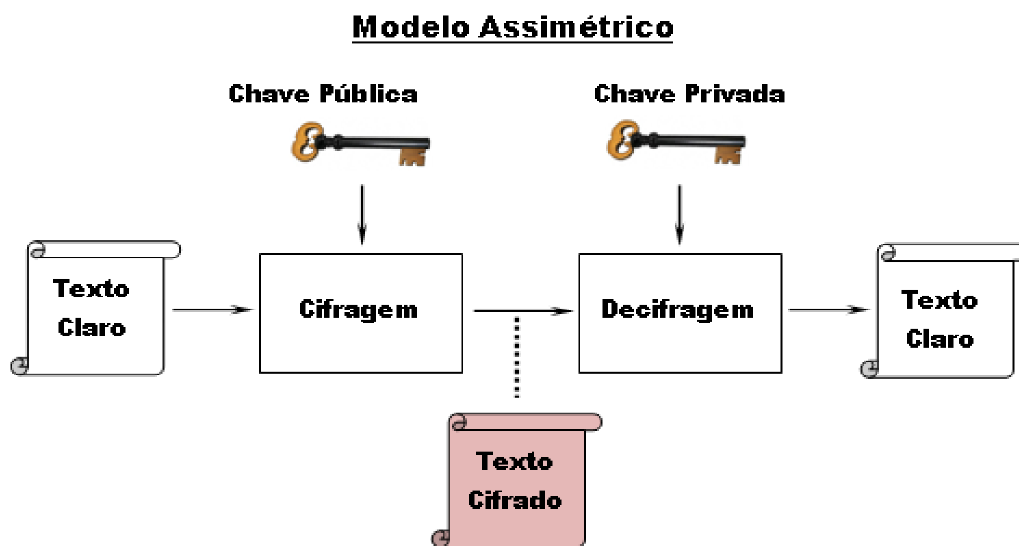

Figura 2.8: *Modelo assimétrico de criptografia.*

Chave simétrica ou chave única é o tipo mais simples de criptografia, já que tanto o emissor quanto o receptor da mensagem possuem a mesma chave, ou seja, a mesma chave é usada tanto na codificação quanto na decodificação. Para ser realizada, é necessário que o emissor antes de enviar a mensagem cifrada, envie a chave privada para todos os usuários que irão, então, utilizá-la para decifrar [2, 11].

A desvantagem, além da impossibilidade de serem usados com fins de autenticação, é a necessidade da troca constante dessas chaves. A exigência de escolher, distribuir e armazenar chaves sem erro e sem perda é conhecida como “gerenciamento de chave”. Os algoritmos de chave simétrica são geralmente muito mais rápidos que os de chave assimétrica.

Chave assimétrica, também conhecida como chave pública, contorna o problema de distribuição de chave através do uso de duas chaves, uma pública que é divulgada e outra privada que é secreta. A criptografia por chave assimétrica cifra os dados utilizando-se de uma das duas chaves e de um algoritmo de criptografia. Para recuperar o texto claro, é necessário utilizar a outra chave e um algoritmo de decriptografia [4].

Se forem iguais a origem está garantida.

A criptografia assimétrica, além de ser utilizada para confidencialidade (sigilo dos dados), também pode ser utilizada para autenticação e/ou assinatura digital. Na assinatura digital, que equivale a uma assinatura no documento, a mensagem é cifrada pela chave privada do indivíduo e pode ser decifrada por qualquer um pela chave pública deste mesmo indivíduo, conforme apresentado na figura 2.9. Se o resultado for o esperado, pode-se ter certeza de que somente o detentor da correspondente chave privada realizou a cifragem. O uso da assinatura digital é uma prova inegável de que uma mensagem veio do emissor. Para verificar este requisito, uma assinatura digital deve ter as seguintes propriedades:

- autenticidade: o receptor deve poder confirmar que a assinatura foi feita pelo emissor;
- integridade: qualquer alteração da mensagem faz com que a assinatura não corresponda mais ao documento;
- não repúdio ou irretratabilidade: o emissor não pode negar a autenticidade da mensagem.

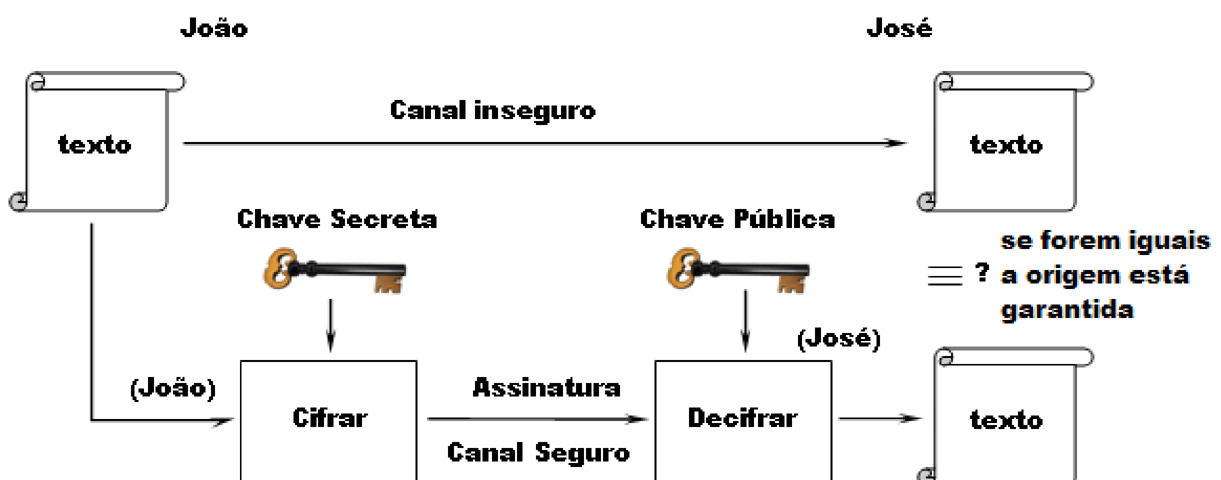

Figura 2.9: *Geração de assinatura digital.*

A tabela 2.1 apresenta de forma sucinta uma comparação entre o processo de criptografia simétrica versus assimétrica.

Criptografia simétrica	**Criptografia assimétrica**
Rápida	Lenta
Gerenciamento e distribuição das chaves são complexas	Gerenciamento e distribuição das chaves são simples
Não oferece assinatura digital	Oferece assinatura digital

Tabela 2.1: *Comparação entre algoritmos quanto à chave simétrica ou assimétrica [2].*

2.3.1 - Importância do tamanho da chave

Para alguém não autorizada conseguir decifrar uma mensagem que foi criptografada com o uso de chaves, terá que descobrir esta chave. Um dos tipos de ataque mais utilizados, conhecido como força bruta, consiste em tentar todas as possíveis combinações de chaves até que a correta seja identificada, portanto a identificação da chave passa a ser uma questão de tempo, o que está diretamente relacionado ao tamanho da mesma. Como a representação é binária, o intervalo dos possíveis valores da chave obedece à relação 2^{TC}, onde TC é o tamanho da chave em bits. Para uma chave de 2 bits o intervalo é de 0 até 4 (2^2), já para uma chave de 56 bits o intervalo é de 0 até 72 quatrilhões (2^{56}). Cada bit adicionado à chave dobra o número de possibilidade, o que dobra o tempo médio para se encontrar a chave por força bruta [2].

O tempo para se quebrar, por força bruta, uma chave em função do tamanho da mesma é apresentado na tabela 2.2. Se considerarmos que a estimativa de existência do universo é de 10^{10} anos, conclui-se que mesmo que seja possível determinar, por força bruta, uma chave, isto pode demorar mais do que o tempo de duração do próprio universo, mesmo com o poder de computação dobrando a cada 1,5 anos [2].

	Tamanho da chave (bits)					
Custo U$	**40**	**56**	**64**	**80**	**112**	**128**
100 mil	2 s	35 h	1 ano	70.000 anos	10^{14} anos	10^{19} anos
1 milhão	200 ms	3,5 h	37 dias	7.000 anos	10^{13} anos	10^{18} anos
10 milhões	20 ms	21 m	4 dias	700 anos	10^{12} anos	10^{17} anos
100 milhões	2 ms	2 m	9 h	70 anos	10^{11} anos	10^{16} anos
1 bilhão	200 us	13 s	1 h	7 anos	10^{10} anos	10^{15} anos
10 bilhões	20 us	1 s	5,4 m	245 dias	10^{9} anos	10^{14} anos
100 bilhões	2 us	100 ms	32 s	24 dias	10^{8} anos	10^{13} anos
1 trilhão	0,2 us	10 ms	3 s	2,4 dias	10^{7} anos	10^{12} anos
10 trilhões	0,02 us	1 ms	300 ms	6 horas	10^{6} anos	10^{11} anos

Tabela 2.2: *Custo x tempo para encontrar a chave por força bruta [2].*
Onde: h = horas; m = minutos; s = segundos; ms = milessegundos (10^{-3} segundos); us = microssegundos (10^{-6} segundos).

2.3.2 - Principais algoritmos assimétricos

Os principais algoritmos assimétricos são apresentados na tabela 2.3, onde podemos observar uma breve descrição do mesmo e de onde advém a sua segurança.

Algoritmo	Descrição	Criptoanálise
DH (Diffie-Hellman) desenvolvido por Whitfield Diffie e Martin Hellman, publicado em 1976.	Baseado no problema do logaritmo discreto. É o sistema mais antigo de chave pública	Intratabilidade do problema do logaritmo discreto em um corpo finito.
ELGAMAL ElGamal, 1985.	Pode ser usado tanto para gerar assinaturas digitais como para cifrar dados. Baseado no problema do logaritmo discreto.	Intratabilidade do problema do logaritmo discreto em um corpo finito.
DSA NIST, 1991.	Algoritmo para gerar assinaturas digitais proposto pelo governo dos EUA [NIST 94]. Esquema de assinaturas baseado no problema do logaritmo discreto. Semelhante ao algoritmo de ELGAMAL.	Intratabilidade do problema do logaritmo discreto em um corpo finito.
ECC - *Elliptic Curve Cryptosystems*.	Definidos sobre corpos compostos de pontos de uma curva elíptica. Oferece o mesmo nível de segurança que os sistemas baseados em corpo de inteiros, mas com tamanho de chave menor. Um ECC de 160 bits equivale a um sistema RSA de 1024 bits	Intratabilidade do problema do logaritmo discreto em grupos aritméticos definidos sobre os pontos de uma curva elíptica.
RSA	O mais usado e fácil de implementar dos algoritmos assimétricos. Diferente do algoritmo DH, que utiliza um único número de 1.024 bits como módulo e realiza o acordo de chave, o módulo de 1.024 bits do RSA é obtido através da multiplicação de números primos de 512 bits.	Problema do logaritmo discreto

Tabela 2.3: *Principais algoritmos assimétricos [10, 12].*

No caso dos algoritmos assimétricos, um dos mais citados e utilizados é sem dúvida o RSA, que através de uma chave pública e outra privativa realiza o processo de cifrar e decifrar a informação. Devido à ineficiência e baixa velocidade dos algoritmos assimétricos, os

métodos de assinatura digital utilizados na prática não cifram os documentos propriamente ditos, mas uma súmula destes, obtidas pelo processamento dos documentos ou informações, por meio de uma função denominada “função de hashing”.

2.3.2.1 - Função de hashing

A função de hashing gera uma saída independente da informação original de tamanho fixo (dependendo do algoritmo de 128 - 256 bits), não importando o tamanho da entrada, assim pode-se ter outro dado separado da informação com o qual se pode utilizar para certificar sua integridade e originalidade. No uso de funções de hashing para assinatura digital, a verificação da informação é feita gerando-se o código hash logo após a informação ser recebida, sendo assim, pode-se comparar este código com o código inicial gerado na origem e em caso positivo confirmar a autenticidade e integridade da informação.

As características mínimas que uma função de hashing deve ter são: ser simples (eficiente e rápido) de se computar; impraticável de se determinar a entrada a partir do mesmo; impraticável de se determinar outra entrada, que resulte no mesmo hash.

Na tabela 2.4 [2], são apresentadas as características de alguns dos algoritmos de hashing mais conhecidos da comunidade de segurança. É possível observar que a maioria deles utiliza tamanho de hashing fixo, especialmente 128 e 160 bits.

Algoritmo de hash	Tamanho do hash gerado	Kbytes/s
Abrreast Bavies-Meyer (c/IDEA)	128 bits	22
Davies-Meyer (c/DES)	64 bits	9
GOST-Hash	256 bits	11
NAVAL (5 passos)	Variável	118
MD4 – Message Digest 4	128 bits	236
MD5 – Message Digest 5	128 bits	174
N-NASH (15 rounds)	128 bits	24
RIPE-MD	128 bits	182
SHA – Secure Hash Algorithm	160 bits	75
SENEFRU (8 passos)	128 bits	23

Tabela 2.4: *Características de algoritmos de hashing.*

Comercialmente os mais utilizados em aplicações para internet, como em servidores e navegadores, são SHA, MD4 e MD5 [2].

2.3.3 - Principais algoritmos simétricos

Na tabela 2.5 são apresentadas as características dos principais algoritmos simétricos. Pode-se observar que em sua grande maioria cifram as informações por bloco, e que o tamanho de bloco mais utilizado é 64bits e o da chave sendo de pelo menos 56 bits. Alguns deles possuem tamanho de chave e/ou bloco variáveis como é o caso do blowfish, RC5, CAST, RC2, RC4, Rinjdael (que foi o vencedor do concurso do AES), MARS, RC6, Serpent e Twofish, que inclusive fizeram parte do projeto AES [2].

Algoritmo	Tipo	Tamanho da chave	Tamanho do bloco
DES	Bloco	56	64
Triple DES (2 chaves)	Bloco	112	64
Triple DES (3 chaves)	Bloco	168	64
IDEA	Bloco	128	64
Blowfish	Bloco	32 a 448	64
RC5	Bloco	0 a 2.040	32, 64, 128
CAST-128	Bloco	40 a 128	64
RC2	Bloco	0 a 1.024	64
RC4	Stream (fluxo)	0 a 256	--
Rinjdael (AES)	Bloco	128, 192, 256	128, 192, 256
MARS	Bloco	Variável	128
RC6	Bloco	Variável	128
Serpent	Bloco	Variável	128
Twofish	Bloco	128, 192, 256	128

Tabela 2.5: *Características dos algoritmos simétricos mais conhecidos.*

2.3.4 - Comparativo entre alguns algoritmos

Na tabela 2.6 é apresentado um comparativo entre os principais algoritmos. O RSA é comumente utilizado para autenticação digital (assinatura). É o algoritmo assimétrico mais popular. Suporta assinatura digital, troca de chaves e cifragem. O DES trabalha com chave simétrica e é um algoritmo rápido mas, em 1999, o NIST lançou uma nova versão do seu padrão DES (FIPS46-3) [5], indicando que o DES só deveria ser utilizado para criptografar dados "não críticos" e em sistemas legados e que o DES triplo (3DES) fosse utilizado em seu lugar [4]. Foi durante muito tempo utilizado pela indústria bancária, atualmente foi substituído pelo AES. O 3DES é o DES aplicado com até 3 chaves distintas, é portanto mais lento que o DES. A aplicação de uma mesma chave no 3DES equivale ao DES em segurança mas é mais lento. O IDEA é uma cifra de blocos, utilizado tanto para cifragem quanto para decifragem. A filosofia que norteou o projeto do IDEA foi "misturar operações de grupos

algébricos diferentes". Estas operações, que podem ser facilmente implementadas via hardware e/ou software, são: XOR; adição módulo 2^{16} (adição ignorando qualquer overflow) e multiplicação módulo $2^{16}+1$ (multiplicação ignorando qualquer overflow). Todas estas operações são feitas com blocos de 16 bits, o que o torna também eficiente em processadores de 16 bits. O IDEA é utilizado no PGP [13].

Algoritmo	**Características**	**Chave**	**Vantagem**	**Desvantagem**
RSA (Rivest-Shamir-Adleman) em 1997	A dificuldade de atacar consiste em encontrar os fatores primos de um número composto [4]	Assimétrica (pública) chegando até 2048 bits	Utilizado como confidencialidade, autenticidade ou ambos. Implementável em hardware.	Capacidade de canal limitado (número de bits que pode transmitir por segundo) [2]
DES	Algoritmo mais utilizado no mundo, até surgir o AES. Blocos de 64 bits	Simétrica de 56 bits	Rápido e o mais utilizado	Já foi quebrado em poucas horas
3DES	Cifragem idêntica ao do DES mas é repetido 3 vezes utilizando 2 ou 3 chaves diferentes, blocos de 64 bits	Simétrica de 112 ou 168 bits	Mais seguro, mas com tamanho de bloco e chave fixos.	Mais lento em relação ao DES
IDEA	Mais utilizado para criptografia de e-mail, PGP e mercado financeiro; blocos de 64 bits	Simétrica de 128 bits	Mais rápido que o DES.	Tamanho de bloco e chave fixos.
RC5	Algoritmo parametrizado onde o usuário pode definir o tamanho do bloco, da chave e o número de iterações. Largamente utilizado em e-mails	Variável, de 8 a 1024 bits, e definida conforme a necessidade do usuário	Adaptar o algoritmo às nossas necessidade	Parâmetros mal escolhidos podem torná-lo muito fraco
AES	Cifrar e decifrar blocos de 128 bits, trabalha com chaves de 128/192/256 bits e é mais rápido que o 3DES	Simétrica de 128, 192 ou 256 bits	Publicamente definido. Projeto permite aumento da chave. Implementável em hardware e em software. Disponibilizado livremente.	Em software, a encriptação e sua inversa empregam códigos diferentes e/ou tabelas. Em hardware a inversa pode usar apenas uma parte do circuito usado no processo de encriptação

Tabela 2.6: *Comparativo entre alguns algoritmos simétricos e assimétricos.*

2.4 - RSA (RIVEST, SHAMIR E ADLEMAN)

RSA é um algoritmo de criptografia inventado por três professores do Instituto MIT, Ron **R**ivest, Adi **S**hamir e Len **A**dleman. Este algoritmo de chave pública (assimétrica) é amplamente utilizado na internet especialmente em mensagens de emails e operações bancárias [2], e foi também o primeiro algoritmo a possibilitar criptografia e assinatura digital [4].

O RSA envolve um par de chaves, uma chave pública que pode ser conhecida por todos e é utilizada geralmente para cifrar os dados e uma chave privada que deve ser mantida em sigilo e utilizada para decifrar os dados. Toda mensagem cifrada usando uma chave pública só pode ser decifrada usando a respectiva chave privada.

O par de chaves é gerado a partir do produto de dois números primos grandes, o que dificulta a obtenção de uma chave a partir de outra.

Sistemas assimétricos, como o RSA, requerem chaves maiores do que os sistemas simétricos para um nível de segurança equivalente. Para gerar as chaves:

1. Escolher dois números primos grandes p e q;
2. Obter $n = p \cdot q$;
3. Escolher um número d, tal que d seja menor que n, e d seja relativamente primo à $(p-1).(q-1)$;
4. Escolher um número e tal que $(ed-1)$ seja divisível por $(p-1).(q-1)$. Utilizar para esse cálculo o algoritmo de Euclides estendido.
5. Os valores e e d são chamados de expoentes público e privado, respectivamente. O par (n,e) é a chave pública e o par (n,d), a chave privada. Os valores p e q devem ser mantidos em segredo ou destruídos.

Para cifrar uma mensagem m, realiza-se a potenciação modular:

$$c = m^e \bmod n$$

Recuperar a mensagem m a partir de c, só é possível utilizando-se a chave privada (**n,d**) e efetuando-se a potenciação modular:

$$m = c^d \bmod n$$

É razoável admitir que não haverá modificação na utilidade desses algoritmos, pois quanto mais avançam as técnicas de fatoração, também avançam as técnicas para se obter números primos maiores [14].

2.5 - DES (DATA ENCRYPTION STANDARD)

Em 1972, o NBS (National Bureau of Standards), órgão de padrões do governo norte-americano da época, atualmente conhecido como NIST (National Institute of Standards and Technology), após uma consulta à NSA, solicitou propostas para um algoritmo de criptografia. Em 1974, a IBM submeteu um algoritmo desenvolvido no período de 1973-1974 baseado num algoritmo mais antigo, o Lucifer de Horst Feistel. O NBS, em novembro de 1976, o adotou como norma federal americana, e assim permaneceu até 2001, quando foi substituído pelo AES. O DES foi durante muito tempo o algoritmo cifrador de blocos mais conhecido e usado e devido à sua importância para o desenvolvimento da criptografia moderna, o DES é também a referência clássica no assunto [2].

O DES é um algoritmo simétrico, composto por operações simples como permutações, substituições, XOR e deslocamentos. Ao fazer estas operações repetidas vezes e de uma maneira não-linear, chega-se a um resultado que não pode ser revertido à entrada original sem o uso da chave. Pequenas alterações na mensagem original geram grandes alterações na mensagem cifrada, o que dificulta conhecer a chave, mesmo que se possa cifrar aquilo que se pretende [15]. O texto cifrado possui o mesmo tamanho do texto plano original.

Este algoritmo cifra as informações em blocos de 64 bits, que são divididos em duas partes de 32 bits cada. O processo principal é repetido 16 vezes, e em cada uma delas utiliza uma chave derivada da chave original. As chaves são armazenadas com 64 bits (8 bytes), sendo que o último bit de cada byte, utilizado para checar a paridade, é depois desprezado, perfazendo o total de 56 bits da chave original [2]. A geração das 16 subchaves, de 48 bits cada, pode ser feita por rotação de bits e permutações de compressão ou permutação direta, esta segunda forma é bastante utilizada na descrição em hardware deste algoritmo.

São utilizadas tabelas, com valores predefinidos, nas permutações de expansão como também nas substituições, sendo que 8 destas tabelas de substituição são conhecidas como S-Boxes.

A operação principal possui sempre a mesma sequência, mudando apenas a ordem em que as subchaves são aplicadas para cifrar ou decifrar o bloco.

2.6 - 3DES OU TRIPLE-DES OU TDEA (TRIPLE DATA ENCRYPTION ALGORITHM)

Com a quebra do DES em 1997, foi proposta por Walter Tuchman uma alteração que aumentou consideravelmente a sua segurança, mas também o tornou bem mais lento, passando então a ser chamado de triple-DES (3DES). Trabalha com chaves de 112 ou 168 bits. A alteração consiste em três cifragens sucessivas, utilizando-se duas ou três chaves diferentes, sendo que a primeira chave é utilizada para cifrar, a segunda decifra a mensagem, mas como a chave é diferente da correta, irá embaralhar mais ainda os dados; a terceira chave (podendo ser a mesma que a primeira) encripta novamente os dados. Como resultado, o espaço de chaves resultante é de 2^{112} [2]. Se as chaves adotadas forem iguais, ele irá funcionar como o DES normal, mas com um custo de processamento maior, tendo em vista que será executado 3 vezes [15].

2.7 - IDEA

Inicialmente chamado de IPES, foi idealizado por Lai e Massey em 1991 para ser eficiente em implementações por software [2]. É o programa para criptografia de e-mail pessoal mais disseminado no mundo e é utilizado principalmente no PGP e no mercado financeiro [16]. Segue a mesma linha geral do DES, mas possui uma implementação mais simples. A implementação por software do IDEA é mais rápida do que a do DES.

Trabalha com blocos de 64 bits e chave simétrica de 128 bits. A criptografia e decriptografia são feitas pelo mesmo algoritmo, primeiramente são feitas 8 iterações com subchaves distintas e depois uma transformação final [2].

As operações básicas do algoritmo são três, facilmente implementadas em hardware, todas sobre 16 bits:

- XOR;
- adição módulo 2^{16} (adição ignorando qualquer overflow);
- multiplicação módulo $(2^{16} + 1)$ (multiplicação ignorando qualquer overflow);

A partir da chave K de 128 bits, são geradas 52 subchaves (K_1 até K_{52}) de 16 bits. K_1 é composta pelos 16 bits mais significativos de K, K_2 é composta pelos próximos 16 bits e assim sucessivamente até K_8 que é composta pelos últimos 16 bits de K (bits menos significativos). A cada 8 subchaves geradas é efetuado um deslocamento de 25 bits para a

esquerda e geram-se novas 8 subchaves. Este processo é repetido até se obter as 52 subchaves (figura 2.10).

Subchaves	Bit de início em K (a partir da direita)							
K1..K8	0	16	32	48	64	80	96	112
K9..K16	25	41	57	73	89	105	121	9
K17..K24	50	66	82	98	114	2	18	34
K25..K32	75	91	107	123	11	27	43	59
K33..K40	100	116	4	20	36	52	68	84
K41..K48	125	13	29	45	61	77	93	109
K49..K52	22	38	54	70				

Figura 2.10: *Representação da criação de 52 subchaves [2].*

Na cifragem, o texto claro é dividido em blocos de 64 bits. Cada um destes blocos é dividido em quatro sub-blocos de 16 bits: B1, B2, B3 e B4. Estes quatro sub-blocos são a entrada da primeira volta ou rodada do algoritmo. No total, são oito rodadas, cada uma utilizando 6 subchaves. Em cada rodada, os quatro sub-blocos são submetidos à operação lógica XOR, somados e multiplicados entre si e com seis sub-blocos de 16 bits oriundos da chave (K1, K2, K3, K4, K5 e K6). Entre cada rodada, o segundo e o terceiro sub-bloco são trocados.

No final, os quatro sub-blocos obtidos (G1, G2, G3 e G4) são concatenados para produzir o texto cifrado [17]

2.8 - RIVEST CIPHERS (RC2; RC4 E RC5)

Criado pelo professor Ron Rivest [2], são uma sucessão de algoritmos bastante usados e que possuem grande flexibilidade e possibilitam maior segurança do que o DES simples. Todos são algoritmos simétricos.

Rivest Cipher 2 (RC2): caracteriza-se por blocos de entrada de 64 bits (8 bytes), as chaves podem ser de vários tamanhos, mas o mais comum é a de 128 bits (16 bytes), por ser considerado mais seguro.

Rivest Cipher 4 (RC4): não trabalha com blocos mas sim com fluxo contínuo e é bastante rápida, cerca de 4 vezes mais rápida do que o DES simples, por estes motivos é

bastante utilizado. Tal como o RC2, permite qualquer comprimento de chave, usando-se normalmente 16 bytes (128 bits). É usado nos padrões SSL/TLS (Secure Sockets Layer/Transport Layer Security) [4]

Rivest Cipher 5 (RC5): criado em 1995, além de muito rápido, o usuário pode escolher o tamanho do bloco (w), o número de interações (r) e o tamanho da chave (b) que desejar, com isto pode ser adaptado às necessidades do usuário, mas uma escolha de parâmetros mal feita pode torná-lo fraco.

Os parâmetros de escolha são:

- tamanho da palavra (w) - o RC5 usa blocos de 2 palavras, portanto um "w" de 16 bits resultará num bloco de 32 bits; um "w" de 32 bits resultará num bloco de 64 bits; valores típicos para w são: 16, 32 e 64
- número de interações (r) - define o número de interações do algoritmo, pode variar de 1 a 255, sendo o ideal um "r" maior que oito. Para aplicar "r" vezes o algoritmo, vai ser gerada a partir da chave uma tabela com t = 2.(r+1) blocos de "w" bits.
- tamanho da chave (b) – define o tamanho da chave a ser usada, o ideal é um "b" de 12 ou 16 bits, que resulta numa chave de 96 e 128 bits respectivamente, pois a chave é igual ao "b" multiplicado por oito.

Para se designar o RC5, usa-se a notação RC5 - w/r/b. O RC5-32/16/7 é equivalente ao DES e, de acordo com Rivest, o RC5 ideal seria o RC5 - 32/12/16, mas cada um usa o que melhor se adapta às suas necessidades.

2.9 - ADVANCED ENCRYPTION STANDARD (AES)

Como o DES havia se tornado insuficiente para conter ataque de força bruta, em 02 de janeiro de 1997 o *National Institute of Standards and Technology* (NIST) manifestou sua intenção de desenvolver o AES [18], em substituição ao DES, com a ajuda da comunidade criptográfica e da indústria e, em setembro do mesmo ano, fez uma chamada oficial para proposta de novos algoritmos, definindo algumas condições que este novo algoritmo deveria ter:

- ser de domínio público, disponível para todo mundo.
- ser um algoritmo de cifra simétrica e suportar blocos de, no mínimo, 128 bits.
- suportar chaves de 128, 192 e 256 bits.
- ser implementável tanto em hardware como em software.

Os critérios utilizados pelo NIST, na proposta original publicada em 1997 (NIST97) para avaliar os candidates ao AES, abrangiam uma gama de preocupações para a aplicação prática de modernas cifras de bloco simétricas, e foram agrupadas em três categorias de critérios [4, 16]:

- segurança: refere-se ao esforço exigido para criptoanalisar um algoritmo, que não fosse a força bruta, pois com uma chave mínima de 128 bits e com a tecnologia da época, mesmo projetada para os dias de hoje, este tipo de ataque foi considerado impraticável.
- custo: a pretensão era de um uso prático numa grande variedade de aplicações, portanto teria que ter alta eficiência computacional, podendo ser utilizado em aplicações de alta velocidade, como links de banda larga.
- algoritmo e características de implementação: uma variedade de considerações dentre as quais a simplicidade para tornar a análise da segurança mais clara e a flexibilidade, a conformidade com uma série de implementações em hardware e software.

A partir dos critérios inicialmente definidos, do grupo inicial de 21 algoritmos candidatos, o NIST, em agosto de 1998, apresentou 15 algoritmos (LOKI97, RIJNDAEL, CAST-256, DEAL, FROG, DFC, MAGENTA, E2, CRYPTON, HPC, MARS, RC6, SAFER+, TWOFISH, SERPENT) na Primeira Conferência dos Candidatos AES, quando solicitou comentários públicos sobre os candidatos, submetendo-os a membros da comunidade criptográfica mundial. Na Segunda Conferência dos Candidatos AES, em março de 1999, o NIST selecionou 5 finalistas: MARS, RC6, Rijndael, Serpent e Twofish. Na Terceira Conferência dos Candidatos AES, além de um fórum de discussões públicas da análise dos finalistas, o NIST convidou os criadores dos algoritmos a participarem, respondendo críticas e comentando seus algoritmos. Em maio de 2000 o NIST iniciou a análise de todas as informações e finalmente em 02 de outubro de 2000, foi anunciado como algoritmo vencedor o Rijndael, dos autores belgas Rijmen e Daemen [2, 4].

O AES não utiliza uma estrutura de Feistel, que consiste basicamente na divisão dos blocos de texto claro em dois sub-blocos esquerdo e direito. Cada sub-bloco direito passa por uma função de encriptação juntamente com uma chave de rodada gerada pelo escalonamento de chaves. O sub-bloco resultante e combinado com o sub-bloco esquerdo, geralmente por meio de uma operação XOR. Depois os blocos, esquerdo e direito, são permutados e o processo é repetido em todas as rodadas, exceto na última, que não possui a permutação. Ao invés disto, no AES, cada rodada consiste de quatro funções distintas: substituição de bytes

(***SubBytes***), deslocamento de linhas (***ShiftRows***), permutações aritméticas sobre um corpo finito $GF(2^8)$ (***MixColumns***) e operação XOR com uma chave (***AddRoundKey***).

Este algoritmo é utilizado neste trabalho e o próximo capítulo será dedicado ao mesmo.

CAPÍTULO 3

3 - AES (ADVANCED ENCRYPTION STANDARD)

3.1 - OPERAÇÕES

O AES trabalha no corpo finito GF(2^8), onde GF é um campo de Galois (Galois Field), onde podemos definir uma operação binária em um conjunto, como um mapeamento de matrizes SxS para S, ou seja, associamos cada par de elemento de S com um elemento de S [2].

Um corpo finito pode ser representado de várias maneiras diferentes, e a equipe de desenvolvimento do AES optou por uma representação clássica de polinômios em que o polinômio b_7x^7, b_6x^6, b_5x^5, b_4x^4, b_3x^3, b_2x^2, b_1x^1, b_0x^0 é representado pelo byte b[7..0] = b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0.

- Adição ⊕:

A adição de dois elementos, na representação polinomial, é um polinômio com os coeficientes resultantes da operação de **soma módulo 2** dos coeficientes dos dois termos, o que corresponde à operação XOR em bits.

Exemplo: soma dos bytes 01010111 e 10000011:

$$01010111 = x^6 + x^4 + x^2 + x^1 + 1$$
$$10000011 = x^7 + x^1 + 1$$
$$(x^6 + x^4 + x^2 + x^1 + 1) + (x^7 + x^1 + 1) = x^7 + x^6 + x^4 + x^2$$

Portanto a operação de soma no AES, em notação binária, equivale à operação XOR:

```
  01010111
⊕ 10000011
------------------
  11010100
```

- **Multiplicação ●**:

A multiplicação em GF(2^8), (denotada por ●), na representação polinomial, corresponde à **multiplicação modular de polinômios** com um polinômio binário irredutível de grau 8, o

que resultará num polinômio binário de grau inferior a 8. Um polinômio é irredutível caso não tenha nenhum divisor diferente de 1 e de si mesmo [2]. Por analogia com inteiros, um polinômio irredutível também é chamado de **polinômio primo** [4]. Para o AES, esse polinômio é: $\mathbf{m(x)= x^8 + x^4 + x^3 + x + 1}$ ou $11B_{16}$ ou 100011011_2 [2].

Exemplo: multiplicação de 73_{16} por $9C_{16}$ ($73_{16} \bullet 9C_{16}$)

$73_{16} = 0111\ 0011 = (x^6 + x^5 + x^4 + x + 1)$ e

$9C_{16} = 1001\ 1100 = (x^7 + x^4 + x^3 + x^2)$

$73_{16} \bullet 9C_{16} = (x^6 + x^5 + x^4 + x + 1)\ (x^7 + x^4 + x^3 + x^2) =$

$x^{13} + x^{12} + x^{11} + x^8 + x^7 \quad + \quad x^{10} + x^9 + x^8 + x^5 + x^4 \quad +$

$x^9 + x^8 + x^7 + x^4 + x^3 \quad + \quad x^8 + x^7 + x^6 + x^3 + x^2 \ =$

$x^{13} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^2$

Como é operação módulo m(x), teremos:

$x^{13} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^2$ módulo $x^8 + x^4 + x^3 + x + 1 =$

$x^7 + x^5 + x^4 + x^2 + x = 1011\ 0110_2 = B6_{16}$

As operações de multiplicação e inversa também podem ser efetuadas por meio da utilização de uma tabela de logaritmos (veja tabela 3.1).

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

Tabela 3.1: *Tabela de logaritmos para multiplicação [19].*

Para efetuar a mesma multiplicação anterior, mas utilizando a tabela de logaritmos, devemos obter na tabela o valor correspondente a cada valor original hexadecimal, sendo que a primeira componente do valor hexadecimal é o índice da linha e a segunda é o índice da coluna. Somar os dois valores encontrados e pesquisar no interior da tabela até encontrar este

valor. O resultado final da multiplicação será um valor hexadecimal composto pelos índices da linha e coluna obtida.

Do exemplo anterior, para a multiplicação de 73_{16} por $9C_{16}$, teremos:

3 para 73_{16} (linha 7 e coluna 3) e 168 para $9C_{16}$ (linha 9 e coluna C);

3 + 168 = 171, que é encontrado na tabela na linha "B" coluna "6", portanto o resultado da multiplicação é $B6_{16}$.= 1011 0110_2.

3.2 - ESTRUTURA DO AES

Na proposta inicial de Rijmen e Daemen para o Rijndael, a cifra poderia ter o tamanho de chave e bloco especificados independentemente como 128, 192 ou 256 bits. A especificação do AES [19] **limita o tamanho do bloco em 128,** mas permite trabalhar com chaves entre 128, 192 ou 256 bits.

A unidade básica para processamento no algoritmo AES é um byte, uma sequência de 8 bits tratadas como uma entidade única.

Internamente, as operações no AES são executadas em uma matriz retangular (array bidimensional) de bytes, denominada ***state***. O ***state*** consiste de 4 linhas, cada uma contendo ***Nb*** bytes, onde ***Nb*** é o tamanho do bloco dividido por 32 bits, portanto na proposta inicial ***Nb*** poderia assumir os valores 4 (=128/32), 6 (=192/32) ou 8 (=256/32), mas o padrão adotado para o AES limitou o bloco em 128 bits, portanto ***Nb***=4 [19].

Na matriz ***state***, figura 3.1, denotada pelo símbolo ***S***, cada byte individual tem dois índices ($S_{r,c}$), um que corresponde ao número da linha $\boldsymbol{r}$ no intervalo $0 \leq \boldsymbol{r} < 4$, e outro que corresponde ao número da coluna $\boldsymbol{c}$ no intervalo $0 \leq \boldsymbol{c} < \boldsymbol{Nb}$, com ***Nb***=4 por padrão do AES. As colunas hachuradas da figura 3.1 fazem parte da proposta original do algoritmo Rijndael, mas não fazem parte do padrão adotado para o AES.

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	$S_{0,4}$	$S_{0,5}$	$S_{0,6}$	$S_{0,7}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,4}$	$S_{1,5}$	$S_{1,6}$	$S_{1,7}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	$S_{2,5}$	$S_{2,6}$	$S_{2,7}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,4}$	$S_{3,5}$	$S_{3,6}$	$S_{3,7}$

Figura 3.1: *State.*

O preenchimento de state é feito por colunas, ou seja, $S_{0,0}$ recebe o primeiro byte do texto plano, $S_{1,0}$ recebe o segundo byte e assim sucessivamente.

Da mesma forma a chave é também organizada em colunas de 4 bytes, representadas pela sigla ***Nk***. A quantidade de rodadas (iterações), representada pela sigla ***Nr*, ** para cifrar/decifrar também está relacionada com o tamanho do bloco e da chave e, portanto, com ***Nb*** e ***Nk***. A tabela 3.2 apresenta esta relação. As colunas hachuradas da tabela 3.2 fazem parte da proposta original do algoritmo Rijndael, mas não fazem parte do padrão adotado para o AES.

Nr	Nb=4	Nb=6	Nb=8
Nk=4	10	12	14
Nk=6	12	12	14
Nk=8	14	14	14

Tabela 3.2: *Relação de **Nr** em função de **Nb** e **Nk**.*

A tabela 3.3 apresenta os parâmetros adotados para o padrão AES [19], considerando que o tamanho de bloco foi fixado em 128 bits

Tamanho do bloco de texto claro em bits	128	128	128
Tamanho da chave em bits	128	192	256
Número de rodadas	10	12	14
Tamanho da chave da rodada em bits	128	128	128
Tamanho da chave expandida em palavras de 32 bits	44	52	60

Tabela 3.3: *Parâmetros do AES [4].*

A figura 3.2 apresenta a relação entre o bloco de entrada, state e o bloco de saída, conforme especificação FIPS-197 **[19]**. O bloco de texto claro ou criptografado, representado por uma matriz quadrada de bytes, é copiado para o vetor ***state***, que é modificado a cada estágio de criptografia ou decriptografia (rodada), e ao final do processo é copiado para uma matriz de saída [4].

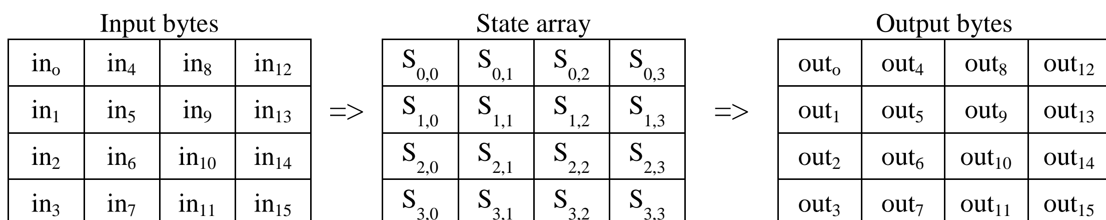

Figura 3.2: *Relação dos blocos de entrada, state e saída.*

A figura 3.3 apresenta uma visão geral do funcionamento do algoritmo AES, tanto na cifragem como na decifragem, com as respectivas transformações, rodadas e utilização das subchaves, resultantes do processo de expansão da chave principal.

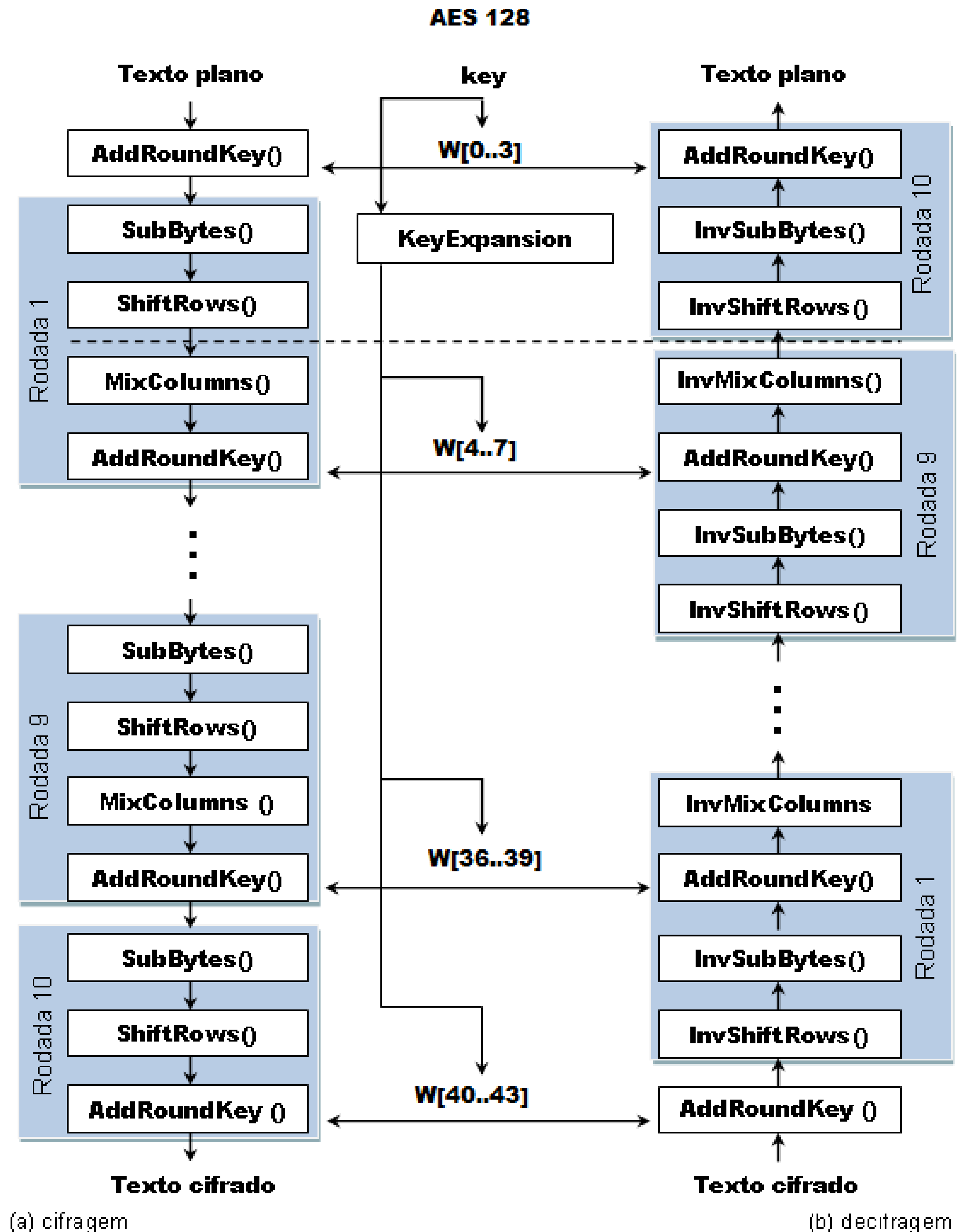

Figura 3.3: *Visão geral do AES para chave de 128 bits.*

O lado esquerdo da figura ilustra o processo de cifragem; no centro a chave original e as subchaves geradas após o processo de expansão, utilizadas em cada uma das rodadas; e do lado direito o processo de decifragem. Observa-se também que as subchaves resultantes da

expansão da chave original são utilizadas, na cifragem, na ordem em que foram produzidas e na decifragem, na ordem inversa. A primeira adição de chave (***AddRoundKey***) utiliza os primeiros 128 bits da chave original, ou seja, as primeiras 4 palavras (w[0..3]), mesmo nos casos de chave com tamanho de 192 ou 256 bits. Esta primeira adição é um XOR com os 128 bits do bloco de texto plano. Podemos considerar que as setas que interligam os blocos, tanto na cifragem quanto na decifragem, equivalem à matriz ***state*** após cada transformação, já que no início do processo o texto plano ou cifrado é copiado para ***state***, sofre as transformações em cada "função" e sai, respectivamente, como texto cifrado ou plano, conforme indicado na figura 3.2.

Podemos também observar as "funções" que compõem cada rodada e a ordem em que as mesmas são executadas. A figura 3.3 apresenta uma visão geral do processo para chaves de 128 bits, portanto são apresentadas 10 rodadas. Para as chaves de 192 ou 256 bits, apenas serão acrescidas mais rodadas, ao invés de 10 serão 12 ou 14 rodadas respectivamente, sendo que sempre a última rodada difere das demais por não possuir "***MixColumns***" ou sua inversa, "***InvMixColumns***" no caso da decifragem. A rodada padrão do AES, para cifragem, é composta por ***SubByte***, ***ShiftRow***, ***MixColumns*** e ***AddRoundKey***, nesta ordem. E para decifragem, é composta por ***InvShifRow***, ***InvSubByte***, ***InvAddRoundKey*** e ***InvMixColumns***.

3.3 - CIFRADOR

No início do processo de cifragem, um bloco de texto plano é copiado para a matriz de estado (state), por colunas. Depois da adição inicial da chave, a matriz de estado é transformada pela implementação da função de rodada 10, 12, ou 14 vezes (dependendo do tamanho da chave), com a última rodada (rodada final) sendo diferente das anteriores. Após a rodada final, a matriz de estado é copiada para a saída e assim termina-se o processo.

A função de rodada é parametrizada usando o escalonamento de chaves que consiste em uma matriz unidimensional de palavras de quatro bytes (32 bits) derivadas da chave original, produzido pela função de expansão de chaves.

O cifrador é descrito em pseudocódigo na figura 3.4. Como o padrão de bloco adotado para o AES é de 128 bits (Nb=4), a matriz de estado possui 16 bytes. Podemos observar na linha 1, da figura 3.4, que o cifrador recebe 16 bytes de texto plano (byte in [4*Nb]), tem como saída 16 bytes de texto criptografado e utiliza as palavras da chave expandida, proporcional ao número de rodadas mais um (w[Nb*(Nr+1)]); o vetor w[ ] contém o escalonamento das chaves que serão usadas na cifragem. Na linha 4, o texto plano é atribuído

à state. Na linha 5, ocorre a primeira adição de chave, transformação ***AddRoundKey()***. Nas linhas 6 até 10, são executadas as transformações ***SubBytes()***, ***ShiftRows()***, ***MixColumns()*** e ***AddRoundKey()***, que compõem uma rodada padrão do AES e que são repetidas 9, 11 ou 13 vezes (linha 6), dependendo do número de rodadas definidas na tabela 3.2. As linhas 12 até 14 representam a última rodada (rodada final), que difere da rodada padrão por não incluir a transformação ***MixColumns()***.

As transformações individuais ***SubBytes()***, ***ShiftRows()***, ***MixColumns()*** e ***AddRoundKey()*** serão detalhadas em outras seções conforme citado na própria figura 3.4.

```
1   Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)]
2   begin
3      byte state [4,Nb]
4      state = in
5      AddRoundKey(state, w[0, Nb-1])                          // Veja seção 3.3.4
6      for round = 1 step 1 to Nr-1
7         SubBytes(state)                                      // Veja seção 3.3.1
8         ShiftRows(state)                                     // Veja seção 3.3.2
9         MixColumns(state)                                    // Veja seção 3.3.3
10        AddRoundKey(state, w[round*Nb,  (round+1)*Nb-1])
11     end for
12     SubBytes(state)
13     ShiftRows(state)
14     AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
15     out = state
16  end
```

Figura 3.4: *Pseudocódigo para o cifrador [5].*

3.3.1 - Transformação SubBytes()

A transformação ***SubBytes***() é uma substituição não linear que modifica, independentemente, cada byte de ***state*** através do uso de uma S-Box ou tabela de substituição. Como esta S-Box é construída por meio de uma operação matemática, a mesma ela pode ser criada em tempo de execução. A transformação é inversível e construída por duas transformações:

1. Opera-se a inversa da multiplicação em GF(2^8), com a representação definida por polinômio com coeficientes {0,1}, onde o elemento {00h} é mapeado em si mesmo [19].
2. Aplica-se uma operação similar definida por:

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$$

para $0 \leq i < 8$, onde b_i é o $i^{ésimo}$ bit do byte, e c_i é o $i^{ésimo}$ bit do byte c com valor {63} ou 01100011_2.

A figura 3.5 mostra a transformação dos elementos na forma de matriz:

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

Figura 3.5: *Transformação **SubBytes()** do AES* [19].

A operação ***SubBytes(State)*** equivale a aplicação desta operação para cada byte de ***state***, figura 3.6. Para a decifragem é utilizada a matriz inversa, que é obtida pela inversa da multiplicação sobre o corpo $GF(2^8)$.

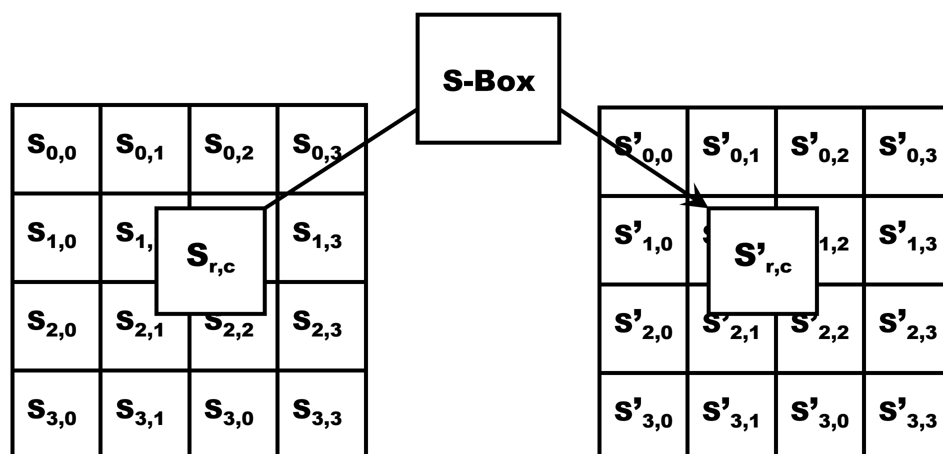

Figura 3.6: *Transformação **SubBytes()** usando a S-Box.*

Pode-se implementar a transformação ***SubBytes()*** através da S-Box de dimensão 16x16 armazenada em memória, o que diminui o tempo de cálculo exigindo entretanto mais recurso

para armazenamento. A transformação inversa é feita através da mesma S-Box. A tabela 3.4 representa a S-Box em notação hexadecimal.

Por exemplo, se $S_{1,1}$ = {9a}, o valor a ser substituído é determinado pela interseção da linha de índice “9” e a coluna de índice “a” na S-Box e portanto $S'_{1,1}$ teria o valor {b8}.

hex		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	08
	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	fb	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabela 3.4: *S-Box: substituição dos valores do byte xy (em formato hexadecimal)* [19].

Neste trabalho a S-Box foi implementada como um vetor unidimensional de 256 posições, de índices 0 até 255 de forma que o valor contido no byte de state, convertido para decimal, corresponda ao índice (endereço) de S-Box em memória.

3.3.2 - Transformação *ShiftRows()*

Na transformação ***ShiftRows()***, os bytes das três últimas linhas de ***State*** são ciclicamente deslocados para esquerda, um número diferente de bytes, em função do tamanho do bloco (***Nb***), conforme mostra a tabela 3.5. Observe que a primeira linha nunca sofre deslocamento, e que no AES os deslocamentos são feitos em bytes, e não em bits como em

muitos outros algoritmos. O objetivo deste deslocamento é aumentar a difusão. As colunas hachuradas da tabela 3.5 fazem parte da proposta original do algoritmo Rijndael, mas não fazem parte do padrão adotado para o AES.

Nb	r=0	r=1	r=2	r=3
4	0	1	2	3
6	0	1	2	3
8	0	1	3	4

Tabela 3.5: *Constante de deslocamento de bytes em função do tamanho do bloco **Nb** e da linha **r**.*

A figura 3.7 apresenta a transformação ***ShiftRows***. Observe os deslocamentos sofridos nas três últimas linhas.

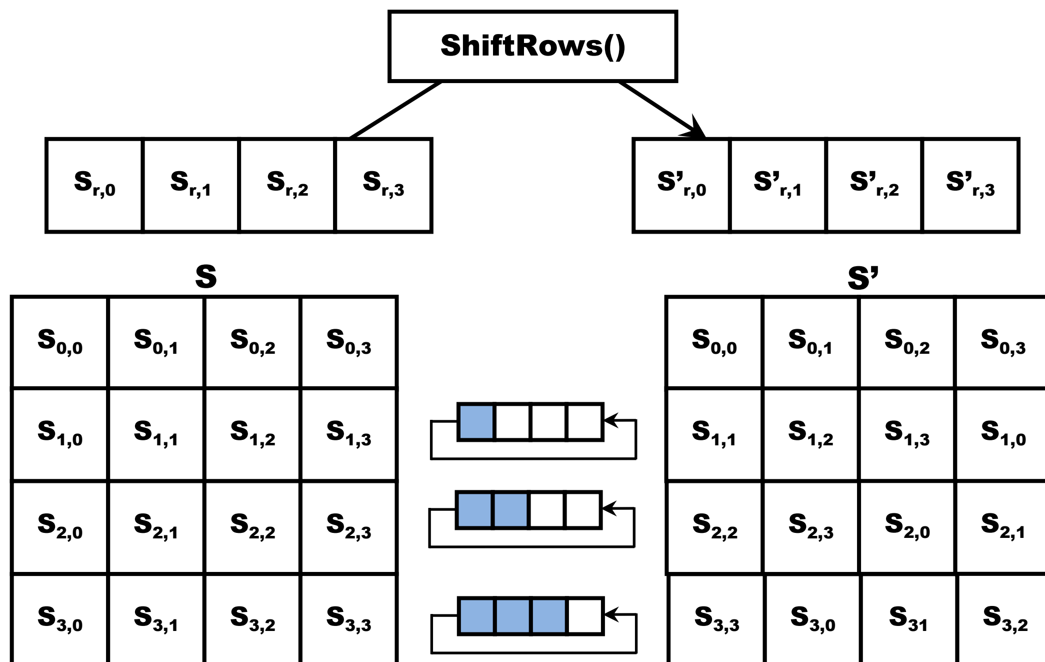

Figura 3.7: *Transformação **ShiftRows()** para **Nb**=4.*

3.3.3 - Transformação *MixColumns()*

A transformação ***MixColumns()*** opera em ***State***, coluna por coluna, conforme figura 3.9, tratando cada coluna como um polinômio de quatro termos sobre $GF(2^8)$ e multiplicadas módulo $x^4 + 1$ com um polinômio fixo a(x), dado por:

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

Este polinômio é reversível o que torna possível o processo de decifrar. Esta transformação pode ser escrita na forma de multiplicação de matriz, conforme figura 3.8.

Seja S'(x) = a(x) ⊗ S(x):

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad \textbf{para } 0 \leq c < Nb$$

Figura 3.8: *Transformação baseada na matriz de multiplicação para o AES.*

Como resultado desta multiplicação, os quatro bytes de uma coluna são substituídos pelos seguintes:

$$S'_{0,c} = (\{02\} \bullet S_{o,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c}$$

$$S'_{1,c} = S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c}$$

$$S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c})$$

$$S'_{3,c} = (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c})$$

A figura 3.9 apresenta a transformação ***MixColumns***. Observe que esta trasnformação trabalha com os 4 bytes de cada coluna por vez.

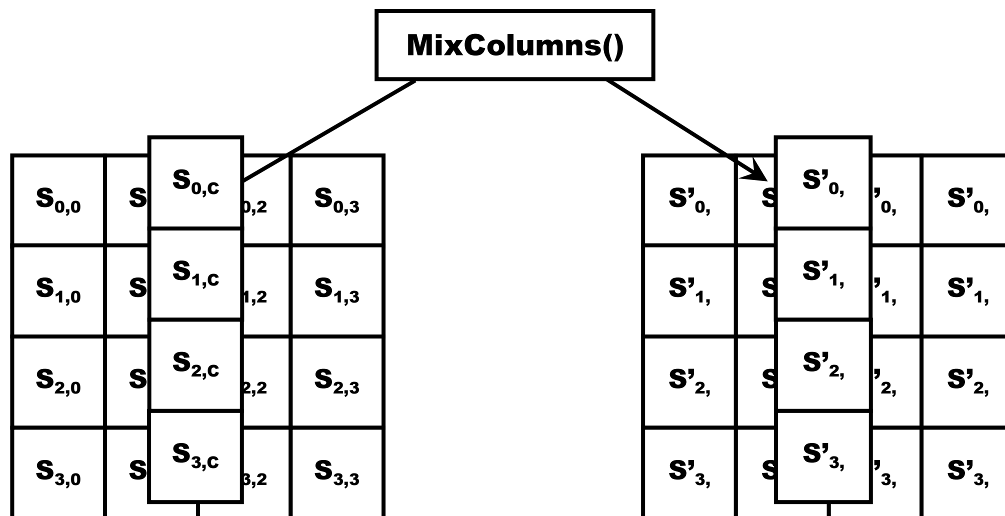

Figura 3.9: *Transformação* ***MixColumns().***

3.3.4 - Transformação *AddRoundKey()*

A transformação ***AddRoundKey()*** realiza uma operação **XOR**, byte a byte, entre ***State*** (matriz de estados) e a chave da rodada. A chave da rodada possui o mesmo tamanho do bloco ***Nb***, ou seja, a chave da rodada é uma matriz com as mesmas dimensões de ***state***.

A Figura 3.10 ilustra a operação XOR entre a matriz de estados (***state***) e a matriz contendo a chave da rodada, resultando na nova matriz de estados.

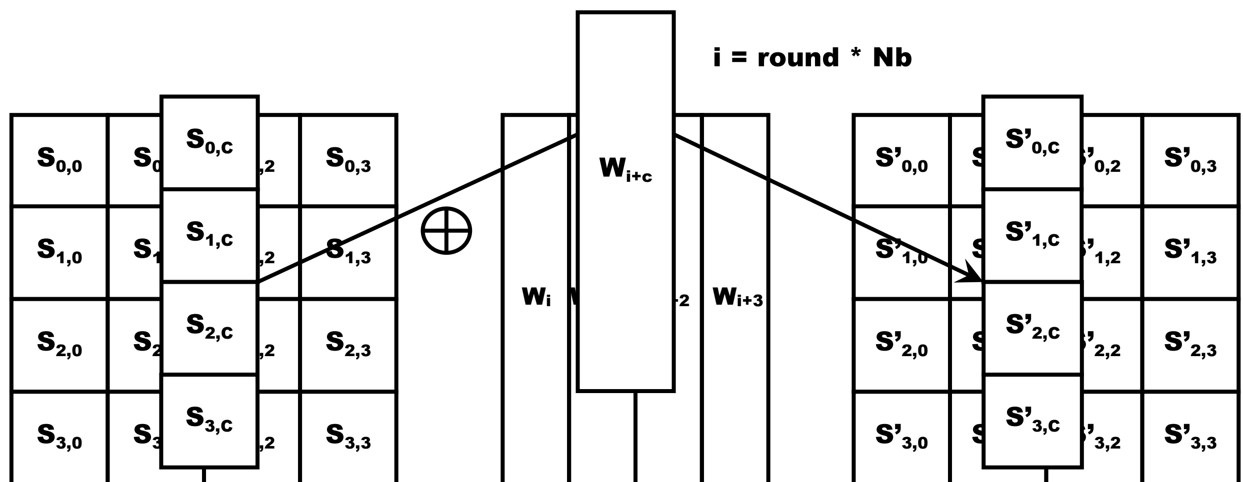

Figura 3.10: *Transformação **AddRoundKey().***

Podemos formular esta operação como sendo:

$$[S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [w_{round*Nb+c}]$$

para $0 \leq c < Nb$ e $0 \leq round < Nr$, onde w_i é um elemento do vetor de chaves com palavras (w) de 4 bytes.

3.3.5 - Geração de subchaves

O AES utiliza na criptografia um total de chaves que é igual ao número de rodadas mais um (***Nr*** + 1), sendo uma chave para cada rodada e uma para o ***AddRoundKey()*** inicial. A geração de subchaves é feita a partir da chave original, que é expandida num vetor unidimensional com ***Nb****(***Nr***+1) palavras (w) de 4 bytes (32 bits). Cada subchave deve possuir o mesmo tamanho da chave original. A figura 3.11 mostra um vetor para ***Nb***=4 e ***Nk***=4

<table>
<tr><td>w_0</td><td>w_1</td><td>w_2</td><td>w_3</td><td>w_4</td><td>w_5</td><td>w_6</td><td>w_7</td><td>...</td><td>w_{40}</td><td>w_{41}</td><td>w_{42}</td><td>w_{43}</td></tr>
<tr><td colspan="4">subchave 0</td><td colspan="4">subchave 1</td><td>...</td><td colspan="4">Subchave 10</td></tr>
</table>

Figura 3.11: *Vetor de chaves para Nb=4 e Nk=4.*

O processo de expansão consiste em preencher os primeiros bytes (primeiras ***Nk*** palavras) do vetor unidimensional com a própria chave original e então gerar as demais palavras do vetor, considerando que a chave da posição "*i*" é obtida pela adição (XOR) da chave da posição anterior com a chave ***Nk*** posições anteriores, ou seja, $\mathbf{w_i = w_{i-1} \oplus w_{i-Nk}}$. Para palavras com índice múltimplo de ***Nk***, antes da adição, a chave w_{i-1} passa por duas transformações, ***RotWord()*** e ***SubWord()***, e por uma adição com uma constante ***Rcon[í]***. Para ***Nk***=8, ou seja, chave de 256 bits, a expansão de chaves é um pouco diferente, pois neste caso, se (i-4) for um múltiplo de ***Nk***, a função ***SubWord()*** é aplicada à palavra $\boldsymbol{w_{i-1}}$ antes da adição de ***Rcon[i]***.

A função ***RotWord()*** faz uma permutação cíclica dos bytes de uma palavra, correspondendo a um byte à esquerda. Por exemplo, seja a palavra [a_o, a_1, a_2, a_3], após a execução da função obtém-se [a_1, a_2, a_3, a_o].

A função ***SubWord()*** aplica a tabela S-Box do AES (tabela 3.4) em cada um dos 4 bytes da palavra.

A função ***Rcon[í]*** contém um valor resultante de [x^{i-1}, {00}, {00}, {00}], com x^{i-1} sendo uma potência de x (x é definido por 02_h) no corpo GF(2^8), e sendo que i começa em 1 e não em zero. Os valores da constante ***Rcon*** são apresentados na tabela 3.6.

Rcon(1)	Rcon(2)	Rcon(3)	Rcon(4)	Rcon(5)	Rcon(6)	Rcon(7)	Rcon(8)	Rcon(9)	Rcon(10)
01	02	04	08	10	20	40	80	1B	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Tabela 3.6: *Rcon(i) – constante de rodada da expansão de chaves.*

A função ***KeyExpansion***, que calcula a expansão das chaves, depende de ***Nk*** e existe uma versão para ***Nk*** ≤ 6 e outra para ***Nk*** > 6.

A partir do vetor unidimensional gerado, é obtida cada uma das subchaves que serão utilizadas no processamento do AES, sendo que para a primeira subchave serão utilizadas as primeiras ***Nb*** palavras (W_0 até W_{Nb-1}) do vetor; a segunda subchave será composta pelas próximas ***Nb*** palavras (W_{Nb} até W_{2*Nb-1}), e assim sucessivamente.

3.4 - Decifrador

O decifrador do AES fará o processo inverso da cifragem (figura 3.4), dando como resultado o texto plano (original). O decifrador é descrito em pseudocódigo na Figura 3.12. As

transformações individuais usadas para decifrar são: ***AddRoundKey()*** (a mesma da cifragem) e ***InvShiftRows()***, ***InvSubBytes()***, ***InvMixColumns()*** que são as funções inversas das transformações utilizadas na cifragem. Como a transformação ***AddRoundKey()*** é a aplicação do OU exclusivo (XOR) entre o ***state*** (matriz de estados) e a subchave, numa primeira aplicação efetua a encriptação e numa nova aplicação a decriptação.

O processo de geração das chaves é o mesmo (***KeyExpansion***), mas a seleção das subchaves, como pode ser observado no loop (linhas 6 até 11) do pseudocódigo da Figura 3.12, obedece a ordem inversa.

Podemos observar na linha 1, da figura 3.12, que o decifrador recebe 16 bytes de texto criptografado (byte in [4*Nb]), tem como saída 16 bytes de texto plano e utiliza as palavras da chave expandida, proporcional ao número de rodadas mais um (w[Nb*(Nr+1)]); o vetor W[ ] contém o escalonamento das chaves que serão usadas na decifragem. Na linha 4, o texto criptografado é atribuído a state. Na linha 5, ocorre a primeira adição de chave, transformação ***AddRoundKey()***, utilizando-se da última chave expandida. Nas linhas 6 até 10, são executadas as transformações ***InvShiftRows()***, ***InvSubBytes()***, ***AddRoundKey()*** e ***InvMixColumns()*** que compõem uma rodada padrão da decriptografia do AES e que são repetidas 9, 11 ou 13 vezes (linha 6), dependendo do número de rodadas definidas na tabela 3.2. As linhas 12 até 14 representam a última rodada (rodada final), que difere da rodada padrão por não incluir a transformação ***InvMixColumns()***.

```
 1   InvCipher(byte in[4*Nb], byte out[4*Nb], word W[Nb*(Nr+1])
 2   begin
 3      byte state [4,Nb]
 4      state = in
 5      AddRoundKey(state, W[Nr*Nb, (Nr+1)*Nb-1])          // Veja seção 3.3.4
 6      for round = Nr-1 step -1 downto 1
 7         InvShiftRows(state)                             // Veja seção 3.4.1
 8         InvSubBytes(state)                              // Veja seção 3.4.2
 9         AddRoundKey(state, w[round*Nb, (Nr+1)*Nb-1])
10         InvMixColumns(state)                            // Veja seção 3.4.3
11      end for
12      InvShiftRows(state)
13      InvSubBytes(state)
14      AddRoundKey(state, w[0, Nb-1])
15      out = state
16   end
```

Figura 3.12: *Pseudocódigo para o decifrador.*

As transformações individuais ***InvSubBytes()***, ***InvShiftRows() e InvMixColumns()*** serão detalhadas em outras seções conforme citado na própria figura 3.12. A transformação e ***AddRoundKey()*** foi detalhada na seção 3.3.4

3.4.1 - Transformação *InvShiftRows()*

A transformação ***InvShiftRows()*** é o inverso da transformação ***ShiftRows()***, e nela os bytes das três últimas linhas de ***State*** são ciclicamente deslocados para a direita, um número diferente de bytes, em função do tamanho do bloco (***Nb***), conforme mostra a tabela 3.5. A primeira linha, r=0, nunca sofre deslocamento, e os deslocamentos são feitos em bytes. Exatamente o mesmo deslocamento do processo de criptografia, mas em sentido inverso. A figura 3.13 apresenta esta transformação.

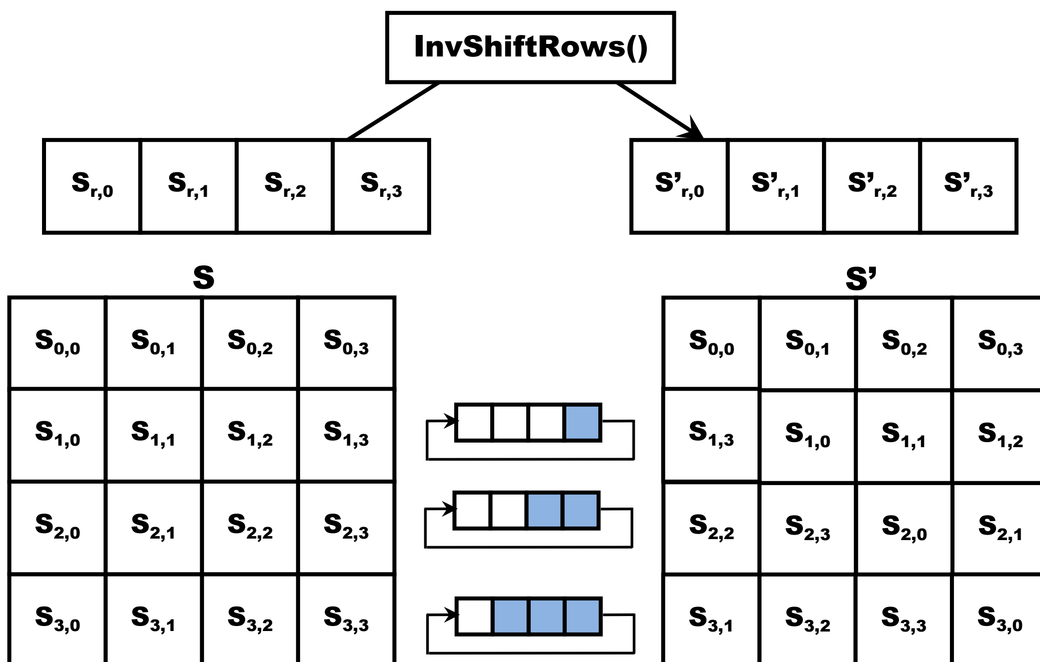

Figura 3.13: *Transformação **InvShiftRows()** para **Nb**=4.*

3.4.2 - Transformação *InvSubBytes()*

A transformação ***InvSubBytes()*** é o inverso da transformação ***SubBytes***(). Nesta transformação, cada byte de ***state*** é substituído por outro byte pelo uso de uma S-Box inversa, conforme tabela 3.7.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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
	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

Tabela 3.7: *S-Box inversa: substituição dos valores do byte xy (em formato hexadecimal).*

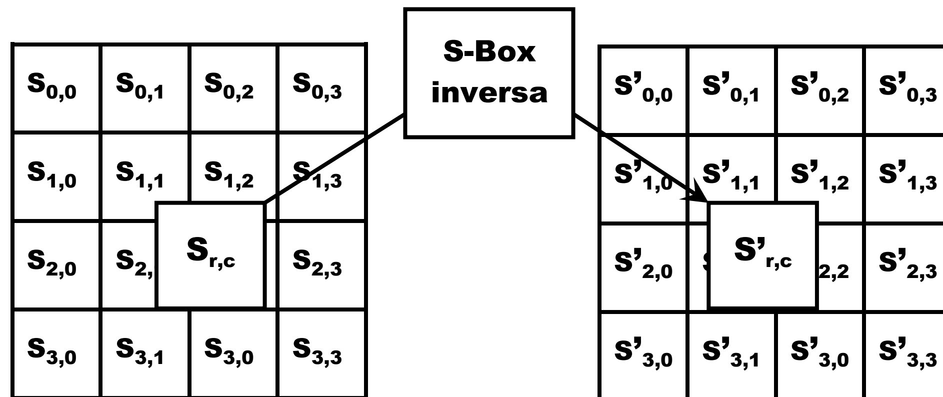

Figura 3.14: *Transformação **InvSubBytes()** usando a S-Box invertida.*

3.4.3 - Transformação *InvMixColumns()*

A transformação ***InvMixColumns()*** é o inverso da transformação ***MixColumns()***. Esta transformação opera em ***State*** (matriz de estados), coluna por coluna, tratando cada coluna como um polinômio de quatro termos sobre $GF(2^8)$ e multiplicadas módulo $x^4 + 1$ com um polinômio fixo $a^{-1}(x)$, dado por:

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

Esta transformação pode ser escrita na forma de multiplicação de matrizes, conforme figura 3.15.

Seja $S'(x) = a^{-1}(x) \otimes S(x)$:

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad \textbf{para } 0 \leq c < Nb$$

Figura 3.15: *Transformação baseada na matriz de multiplicação para o AES.*

Como resultado desta multiplicação, os quatro bytes de uma coluna são substituídos pelos seguintes:

$$S'_{0,c} = (\{0e\} \bullet S_{o,c}) \oplus (\{0b\} \bullet S_{1,c}) \oplus (\{0d\} \bullet S_{2,c}) \oplus (\{09\} \bullet S_{3,c})$$

$$S'_{1,c} = (\{09\} \bullet S_{o,c}) \oplus (\{0e\} \bullet S_{1,c}) \oplus (\{0b\} \bullet S_{2,c}) \oplus (\{0d\} \bullet S_{3,c})$$

$$S'_{2,c} = (\{0d\} \bullet S_{o,c}) \oplus (\{09\} \bullet S_{1,c}) \oplus (\{0e\} \bullet S_{2,c}) \oplus (\{0b\} \bullet S_{3,c})$$

$$S'_{3,c} = (\{0b\} \bullet S_{o,c}) \oplus (\{0d\} \bullet S_{1,c}) \oplus (\{09\} \bullet S_{2,c}) \oplus (\{0e\} \bullet S_{3,c})$$

A figura 3.16 apresenta a transformação ***InvMixColumns***. Observa-se que esta transformação trabalha com os 4 bytes de cada coluna por vez.

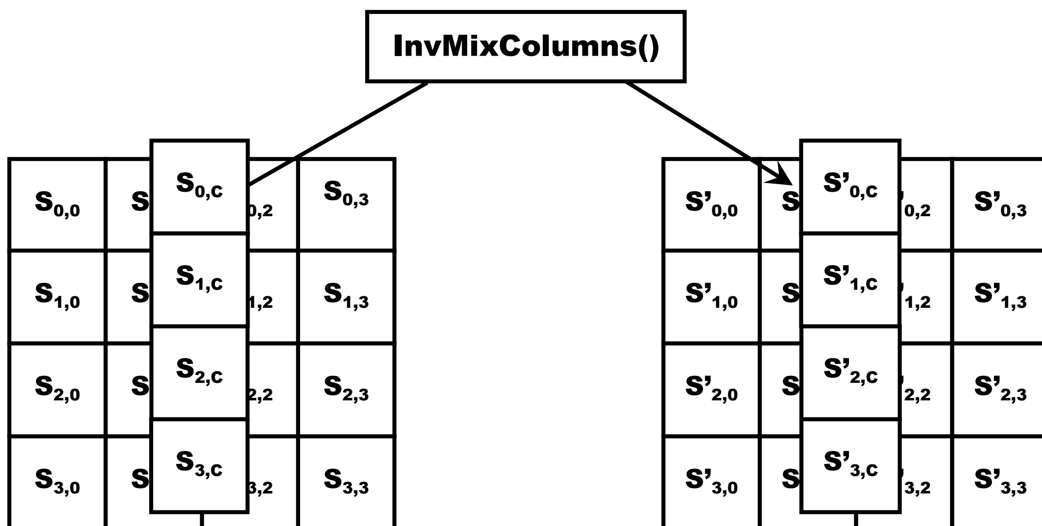

Figura 3.16: *Transformação* ***InvMixColumns().***

CAPÍTULO 4

4 – CIRCUITOS INTEGRADOS E LINGUAGEMVHDL

Os primeiros CI's (circuitos integrados) começaram a ser fabricados no início da década de 70, implementando funções lógicas básicas (AND, OR, NOT, etc.). Esses primeiros CI's foram denominados SSI (*Small Scale Integration*, ou integração em pequena escala), e possuíam até dez portas lógicas. Passado algum tempo, surgiram os circuitos MSI (*Medium Scale Integration*, integração em média escala), e com esses dispositivos um maior número de portas podia ser implementado na pastilha de silício, disponibilizando no mercado circuitos que desempenhavam funções lógicas mais complexas, incluindo blocos lógicos, contadores, registradores e outros. Com a LSI (*Large Scale Integration*, integração em larga escala), surgiram os primeiros microprocessadores em um único chip; mais atualmente, a tecnologia VLSI (*Very Large Scale Integration*, integração em escala muito larga) possibilitou a fabricação de microprocessadores de 64 bits com milhares de transistores [20].

A medida usual de escala de integração de um CI é o número de portas equivalentes por área de pastilha de silício, onde uma porta equivalente é representada por uma porta NAND de duas entradas.

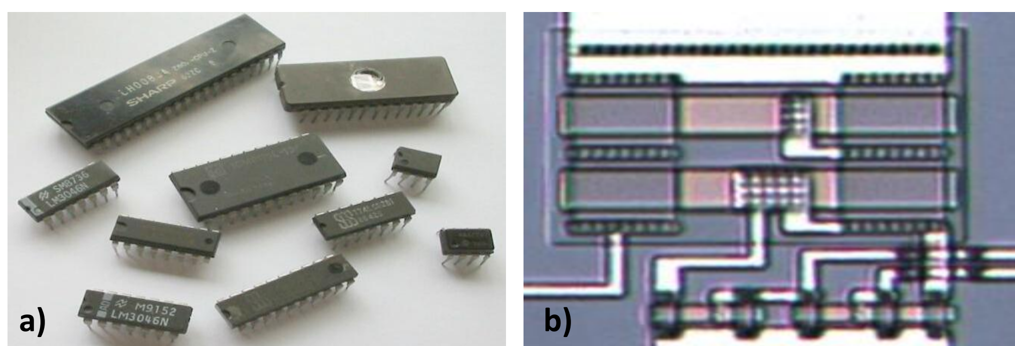

Figura 4.1: *(a) Circuitos integrados variados. (b) Detalhe da estrutura interna de um circuito integrado. Os circuitos integrados com tecnologia mais moderna podem ter até alguns milhões de transistores integrados na sua área de pastilha.*

A implementação de um projeto digital pode ser realizada utilizando-se circuitos integrados padronizados (figura 4.1). Assim, o projeto poderia resumir-se a elaborar a lógica do circuito a ser implementado e esquematizar as interligações entre os CI's padronizados

disponíveis no mercado. Entretanto, projetos mais complexos exigiriam o emprego de centenas ou milhares de CI's padronizados, o que traria muitas desvantagens, tanto de viabilidade econômica do projeto como em aspectos técnicos. Desta forma, a redução do número de CI's utilizados em um projeto traria várias vantagens, como menor espaço ocupado na placa de circuito impresso, menor quantidade de solda a ser empregada, menor consumo de energia, processo de montagem mais simplificado e com menor custo, maior confiabilidade do sistema, etc.

Tendo em vista esta necessidade, os fabricantes de semicondutores desenvolveram dispositivos capazes de englobar em uma pastilha de silício inúmeras funções para um determinado projeto, surgindo então os dispositivos ASIC (*Application Specific Integrated Circuit*). O ASIC é um dispositivo cujas funções são implementadas para uma aplicação específica, produzido sob encomenda do usuário. As principais metodologias para implementação de um dispositivo ASIC são:

- *Full-Custom*: metodologia de projeto de circuitos integrados no nível de dispositivo transistor, onde o projetista caracteriza cada célula do dispositivo. Esta metodologia é o oposto ao uso de bibliotecas pré-definidas de componentes. O projeto *full-custom* exige habilidade e experiência do projetista, e é geralmente viável apenas para circuitos pouco mais simples, especialmente aqueles em que uma estrutura se repete muito, como por exemplo, em dispositivos de memória, onde um pequeno ganho em espaço ocupado na pastilha e em consumo de potência do componente proporcionará uma grande economia total.

- *Semi-custom*: projeto de ASIC's baseado na caracterização de dispositivos a partir de blocos lógicos pré-projetados, conhecidos como células padrão (*standard cells*), e também com auxílio de uma ferramenta computacional. Assim, no software de auxílio ao projeto existem bibliotecas que podem conter centenas de diferentes células padrão, incluindo desde portas lógicas, até *latches* e *flip-flops* com diferentes combinações de reset, *preset* e opções de *clock*. O desenvolvimento segundo a metodologia *semi-custom* se fundamenta em definir e efetuar as combinações e interligações dessas várias células padrão, alocando os blocos de forma a otimizar a área do circuito e reduzindo os tempos de atraso na propagação dos sinais. A figura 4.2 apresenta como exemplo o *layout* de uma célula padrão de uma porta NAND de 3 entradas [21]

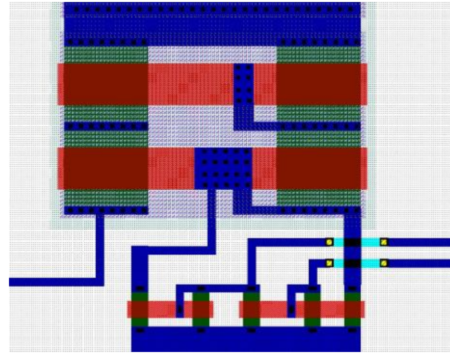

Figura 4.2: *Célula padrão utilizada em projeto semi-custom. Essas células padronizadas (standard cells) estão disponíveis em bibliotecas dos softwares utilizados na caracterização de circuitos integrados.*

Dispositivos de Lógica Programável: metodologia de projeto de ASIC's baseada na utilização de dispositivos lógicos programáveis. Diferentemente das metodologias *full-custom* e *semi-custom*, onde o projeto ASIC é definido antes da fabricação da máscara do circuito integrado, o projeto usando dispositivos de lógica programável é feito a partir de componentes já encapsulados. O desenvolvimento do ASIC se resume à elaboração do projeto, entrada deste projeto a partir de um software de desenvolvimento, definição do dispositivo que comporta as funções as quais se deseja implementar e a gravação do dispositivo lógico.

4.1 - Dispositivos Lógicos Programáveis (PLD)

Um PLD (*Programmable Logic Device*) é um dispositivo que dispõe em sua arquitetura de um grande número de portas lógicas, *flip-flops*, *latches* e registradores interconectados, e que permite ao próprio usuário realizar a programação das funções de saída desejadas para sua aplicação.

Os primeiros dispositivos lógicos programáveis surgiram em meados da década de 70, a partir do desenvolvimento de elementos PROM (fusíveis) de programação, fabricados com tecnologia bipolar TTL (*Transistor-Transistor Logic*). Posteriormente foi desenvolvida a tecnologia CMOS (*Complementary Metal Oxide Silicon*), que então passou a ser a mais utilizada na fabricação de circuitos integrados em geral, por apresentar várias vantagens em relação à tecnologia bipolar, tais como maior densidade de integração, menor consumo de energia, maior imunidade a ruídos, processo de fabricação da pastilha mais simples, entre outras. A princípio, podemos dividir os PLD's em dois grupos: os de primeira geração e os de segunda geração.

Os PLD's de primeira geração têm arquiteturas mais simples, baseadas no conceito de somas de produtos (veja figura 4.3). São capazes de implementar funções de baixa

complexidade, com até mil portas equivalentes. Podemos citar como PLD's de primeira geração a PROM (ou PLE), a PAL e o PLA.

Os PLD's de segunda geração surgiram no final da década de 80, e já dispõem de arquiteturas mais avançadas, com um grande número de blocos lógicos e elementos de memória interconectados, de forma a permitir a implementação de funções lógicas bem mais complexas. Entre os PLD's de segunda geração estão o EPLD, o FPGA e o *Folded-array*.

4.2 - Ideia básica de um PLD

A figura 4.3 traz o esquema de um circuito que representa a ideia básica para o desenvolvimento dos dispositivos lógicos programáveis.

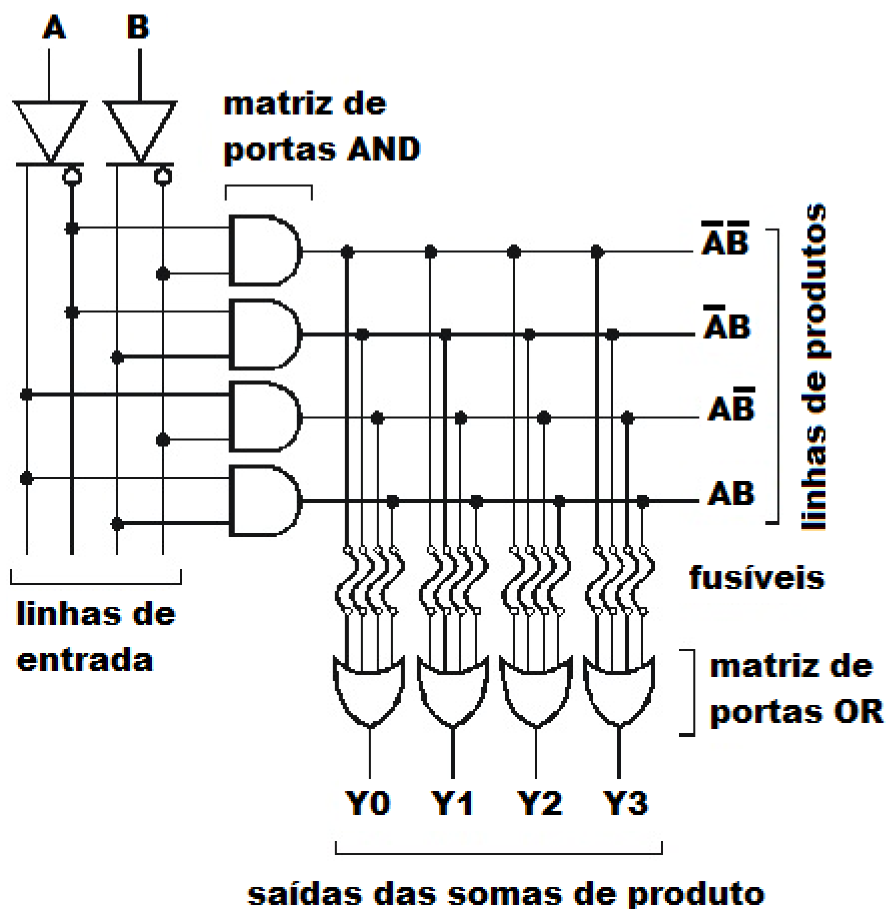

Figura 4.3: *Circuito básico de um PLD.*

O circuito deste exemplo consiste de duas variáveis de entrada A e B, uma matriz de portas AND e a uma matriz de portas OR. As entradas A e B estão ligadas a buffers inversores e não-inversores, de forma a se ter linhas de entrada com as próprias variáveis A e B e também $\overline{A}$ e $\overline{B}$. Cada porta AND é conectada a duas linhas de entrada diferentes, tendo

em cada saída um produto diferente. As saídas da porta AND são chamadas linhas de produto [22]. As linhas de produto são ligadas às entradas das portas OR através de elementos fusíveis. Nas saídas das portas OR têm-se as somas de produto.

Com todos os fusíveis intactos, as saídas das portas OR estarão em nível lógico “1”. A programação de funções no PLD é feita através de um gravador próprio, que faz passar uma corrente suficiente para efetuar a queima de determinados fusíveis. O elemento fusível foi desenvolvido de forma a fornecer nível lógico “0” quando está queimado. Desta forma, com a queima de fusíveis de posições definidas, interrompe-se a ligação com produtos de variáveis que não fazem parte da função lógica que se deseja implementar, restando as ligações de produtos que constituem uma função lógica.

4.3 - Simbologia

O esquemático de um PLD para muitas variáveis apresentado da forma como está o da figura 4.3 ficaria um tanto complicado pela quantidade de linhas e ligações a serem representadas, dificultando a própria realização do desenho do circuito e sua interpretação. Por este motivo, os fabricantes adotaram uma simbologia que simplifica bastante a representação do circuito de um PLD. A figura 4.4 ilustra esta simbologia.

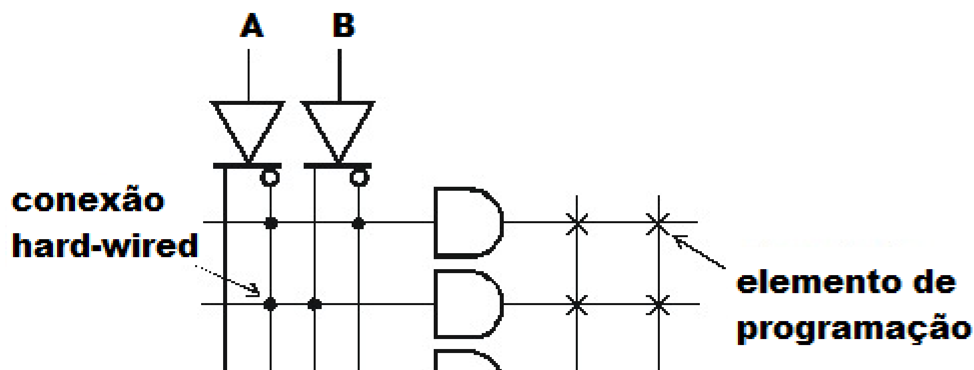

Figura 4.4: *Detalhe de um esquemático utilizando a simbologia simplificada para PLD’s.*

Basicamente, tem-se na figura 4.4 uma representação de entrada unifilar para portas lógicas, onde as conexões das variáveis de entrada são representadas pelos símbolos $\bullet$ ou $\times$. O símbolo $\bullet$ representa uma conexão hard-wired, que não pode ser alterada, já o símbolo $\times$ representa uma conexão feita através de um fusível ou outro elemento de programação. Evidentemente, a ausência de um símbolo de conexão indica que duas linhas não estão ligadas.

4.4 – Características dos dispositivos lógicos programáveis

O uso de PLD's para a implementação de ASIC's é crescente devido às vantagens apresentadas para estas aplicações, como mencionado anteriormente. Tais vantagens se justificam através de certas características dos dispositivos lógicos programáveis. As principais características apresentadas pelos PLD's:

- Alto desempenho: o desempenho do dispositivo está relacionado com a sua arquitetura e sua tecnologia de fabricação. Como mencionado anteriormente, a tecnologia mais utilizada atualmente na fabricação de circuitos integrados é CMOS. Além de menor consumo de potência, dispositivos CMOS mais atuais já conseguem operar em velocidades próximas às de dispositivos TTL. Os PLD's apresentam em suas arquiteturas uma estrutura de interconexão denominada interconexão contínua, que usa uma única trilha de metal para realizar a conectividade entre as células lógicas existentes no dispositivo. Deste modo, os atrasos de propagação interna dos sinais são minimizados, permitindo aos PLD's operar em altas velocidades. Também se consegue redução do tamanho do die do PLD quando comparado com outros dispositivos programáveis [20].
- Alta densidade de integração: dispositivos CMOS ocupam uma área bem menor na pastilha de silício que dispositivos com tecnologia TTL, permitindo uma alta densidade de integração, variando de 300 a 1 milhão de portas equivalentes. Com isso, aumenta-se a confiabilidade, assim como seu desempenho, acarretando ainda a redução nos custos do projeto.
- Baixo custo: basicamente, o custo de um projeto de ASIC usando PLD se resume à aquisição do dispositivo e sua gravação.
- Flexibilidade: dependendo da tecnologia de fabricação dos elementos de programação, é possível efetuar alterações na programação do PLD. Isto permite flexibilidade para o projetista, que pode fazer alterações no projeto conforme sua conveniência sem a necessidade de alteração dos componentes do sistema, tendo somente que reprogramar as funções a serem implementadas no PLD.
- Tempo de projeto reduzido: o uso de ferramentas computacionais facilita em muito a realização de projetos com dispositivos lógicos programáveis.

4.5 – A linguagem VHDL

No início da década de 80, o Departamento de Defesa dos Estados Unidos estava trabalhando no programa de seu circuito integrado de maior velocidade (Very High-Speed Integrated Circuit – VHSIC) e durante o processo desenvolveu uma linguagem de descrição de hardware para especificar e simular esses sistemas bastante complexos. Essa linguagem envolveu um método padronizado pelo IEEE (Institute of Electrical and Electronic Engineering) de descrição de circuitos lógicos para finalidades de projeto, simulação e documentação. Essa linguagem é conhecida como VHDL (VHSIC Hardware Description Language – Linguagem de descrição de hardware para VHSIC) [22].

Assim, a linguagem VHDL e também outras linguagens descritivas vêm sendo uma tendência atual para o desenvolvimento de projetos ASIC. Algumas das vantagens apresentadas no desenvolvimento de projetos digitais usando VHDL:

- Descrição da estrutura de um projeto, sua decomposição em submódulos (hierárquica) e como os submódulos se comunicam;
- Especificação com mais precisão dos parâmetros de circuitos sem a necessidade de saber exatamente como este será construído;
- Especificação de funções em projetos usando formas de linguagens de programação familiares.
- Possibilidade de simulação do projeto antes de sua fabricação, de forma a permitir comparações entre alternativas de soluções e a realização de testes e ajustes sem a necessidade de implementar um protótipo de hardware, representando uma economia de tempo e de dinheiro.

CAPÍTULO 5

RESULTADOS

5.1 - Considerações iniciais

Conforme citado anteriormente, um dos requisitos para aceitação do algoritmo criptográfico AES pelo NIST era a possibilidade de implementação em software e hardware. Neste trabalho focamos a implementação em hardware.

O objetivo do trabalho foi sintetizar o AES, permitindo ao usuário escolher o tamanho da chave que iria utilizar, 128, 192 ou 256 bits, no momento da criptografia ou decriptografia, ou seja, durante o uso do componente, diferentemente de outros trabalhos em que a síntese é feita já com o tamanho da chave definido, restringindo o uso do algoritmo àquele tamanho de chave.

O componente escolhido para o desenvolvimento dos trabalhos foi o componente EP2C20F484C7 da família um Ciclone II do fabricante ALTERA, por este componente ser parte constituinte do kit DE2 usado no desenvolvimento do trabalho. A tabela 5.1 mostra as características dos componentes da família Cyclone II.

Table 1-1. ***Cyclone II FPGA Family Features***

Feature	EP2C5	EP2C8	EP2C15	EP2C20	EP2C35	EP2C50	EP2C70
LEs	4,608	8,256	14,448	18,752	33,216	50,528	68,416
M4K RAM *blocks (e Kbits plus 512 parity bits*)	26	36	52	52	105	129	250
Total RAM bits	119,808	165,888	239,616	239,616	483,840	594,432	1,152,000
Embedded multipliers	13	18	26	26	35	86	150
PLLs	2	2	4	4	4	4	4
Maximum user I/O pins	158	182	315	315	475	450	622

Tabela 5.1: *Características do FPGA da família Cyclone II [23].*

5.2 – Visão geral do processamento

O primeiro passo foi definir um fluxograma de execução do AES, apresentando a ordem em que as funções deveriam ocorrer para o funcionamento adequado ao processo de criptografia. A operação detalhada de cada módulo será apresentada nas próximas seções.

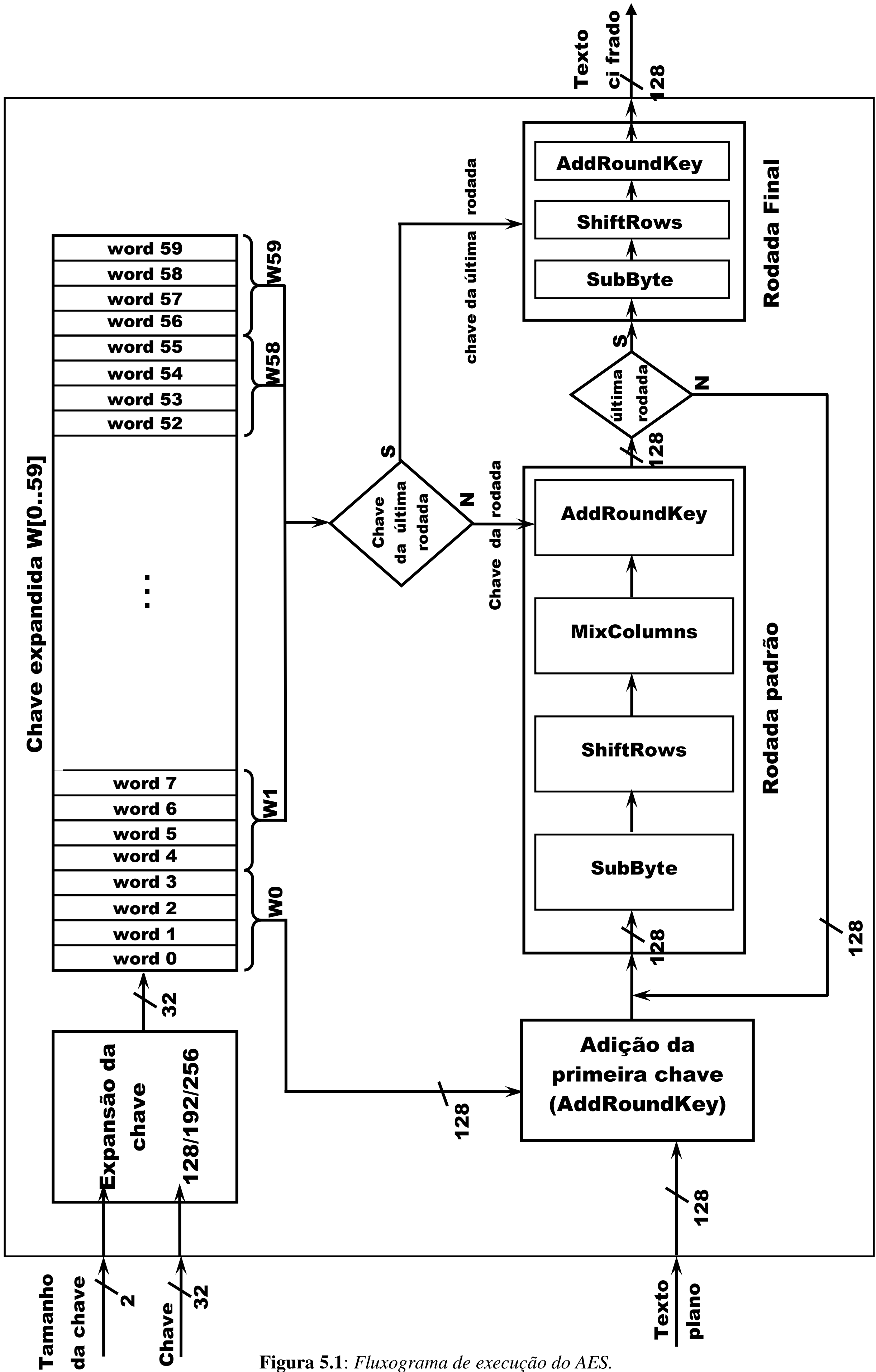

Figura 5.1: *Fluxograma de execução do AES.*

Na figura 5.1 foram excluídos os sinais de entrada de clock (**clock_pino**), reset (**reset_pino**) e de fim de processamento (**Fim_pino**), para permitir uma maior clareza do fluxograma.

Os sinais de entrada são:

- tamanho da chave: define o tamanho da chave desejada pelo usuário, podendo ser 1, 2 ou 3, respectivamente para as chaves de tamanho 128, 192 ou 256 bits;

- chave: chave informada pelo usuário para ser utilizada no processo de criptografia, deverá ter o tamanho de 128, 192 ou 256 bits, conforme valor definido no sinal de entrada para o "tamanho da chave". A entrada da chave, no módulo principal, será feita em blocos de 32 bits, isto devido a uma limitação de pinos disponíveis no componente escolhido. Optou-se por fragmentar a entrada da chave, ao invés do texto plano ou da saída do texto criptografado, para que tivéssemos um ganho no processamento, ou seja, como a chave será recebida apenas uma única vez durante o processo, e o texto plano será criptografado continuamente em blocos de 128 bits. Se fragmentássemos o texto plano ou o texto de saída, criptografado, o processo teria que, a cada novo bloco, esperar a montagem completa do mesmo. Para criptografar um único bloco de texto, utilizando-se de uma chave de 128 bits, não haveria diferença entre uma escolha e outra. Para um único bloco mas utilizando-se chaves de 192 ou 256 bits, seria mais vantajoso a fragmentação do texto ao invés da chave, mas como o objetivo é a criptografia, não apenas de um único bloco de 128 bits, torna-se mais vantajoso, por conta da limitação do componente, fragmentar a chave. Desta forma, a chave informada pelo usuário será montada totalmente após 4, 6 ou 8 pulsos de relógio, dependendo do tamanho da mesma, respectivamente para chaves de 128, 192 ou 256 bits.

- texto plano: texto a ser criptografado. Serão utilizados, por definição do padrão AES, blocos de 128 bits;

- texto criptografado: texto final após o processo de criptografia. O texto criptografado possui o mesmo tamanho do bloco de texto plano de entrada, portanto serão blocos de 128 bits.

O módulo denominado de "expansão da chave" irá expandir a chave informada pelo usuário até obter o total de palavras necessárias para atender todas as chaves de rodada do processo criptográfico. Como o bloco do AES é fixado em 128 bits e cada palavra possui 32 bits, teremos a necessidade de 4 palavras (128 bits / 32 bits de cada palavra) para cada chave de rodada. Como o processamento com chaves de 128, 192 e 256 bits necessita respectivamente de 10, 12 e 14 rodadas (ver tabela 3.2) e que ainda há uma adição de chave

inicial, serão necessárias, portanto, 44, 52 ou 60 palavras [(número de rodadas + 1) * 4 palavras] respectivamente para estes tamanhos de chave.

Como foi implementado, neste trabalho, o AES para os três tamanhos de chave, o vetor de saída, deste módulo, foi definido com 60 palavras (w[0..59]), para atender as três possibilidades; entretanto, o módulo de "expansão da chave" irá expandir a chave somente até atingir o número necessário de palavras para o processo, considerando o tamanho da chave informada.

O bloco de texto plano (128 bits) sofre a primeira transformação pela adição da primeira chave, que corresponde às primeiras 4 palavras da chave expandida (w[0..3]). Depois disto, entrará numa rotina de processamento, chamada de rodada padrão, que envolve as quatro transformações ***SubByte***, ***ShiftRows***, ***MixColumns*** e ***AddRoundKey***, nesta ordem. A cada rodada padrão é utilizada uma chave de rodada a partir da chave de índice 1. A primeira rodada utiliza a chave W1 (w[4..7]), a segunda rodada, a chave W2 (w[8..11]), a terceira rodada a chave W3 (w[12..15]) e assim sucessivamente, até atingir o número de rodadas necessárias, definida pelo padrão AES conforme tabela 3.2.

Ao final da penúltima rodada, o bloco que está sendo criptografado é direcionado para o módulo "rodada final", e sofrerá três transformações: ***SubByte***, ***ShiftRows*** e ***AddRoundKey***. A última rodada difere da rodada padrão por não conter a transformação ***MixColumns***. A saída da rodada final será o bloco criptografado.

5.3 – Módulos desenvolvidos

Posteriormente à definição do fluxo geral da criptgrafia, foram detalhados e desenvolvidos cada um dos módulos em VHDL.

5.3.1 – Expansão da chave (KeyExpansion)

O componente de expansão de chaves (***KeyExpansion***), como já descrito anteriormente, irá efetuar a expansão da chave original, informada pelo usuário, gerando o número necessários de palavras de 32 bits para suprir a necessidade de todas as chaves do processo.

Como pode ser observado na figura 5.2, o componente possui como entradas os sinais:

- HabilitaExpansao: irá ou não permitir a expansão das chaves. Este sinal é inicializado com zero (0) e será alterado para um (1) após o término da montagem completa da chave fornecida pelo usuário, lembrando que neste projeto a chave é inserida em blocos de 32 bits. Após o período dado para o processo de expansão

ocorrer, um pulso de clock, este sinal é alterado novamente para zero (0) (linha 100).

- TamanhoDaChave: informa qual o tamanho da chave escolhida pelo usuário (linha 101).
- KeyIn: chave original informada pelo usuário com tamanho total de 128, 192 ou 256 bits; o sinal foi definido com tamanho de 256 bits para atender o caso da maior chave (linha 102).
- KeyExpandida: é um vetor de 60 palavras de 32 bits que conterá o resultado final da expansão das chaves (linha 103).

```
98    -- componente que efetua a expansão da chave
99    component KeyExpansion is port (
100      HabilitaExpansao : in std_logic; -- =1, habilita o cálculo da expansão de chaves; =0, não
         habilita (a expansão não é efetuada)
101      TamanhoDaChave : in integer range 1 to 3; -- opção pela utilização de chave de 128, 192
         e 256 bits, respectivamente =1,2 ou 3;
102      KeyIn : in std_logic_vector(0 to 255); -- chave original informada
103      KeyExpandida : out KeyExpansion_type); -- matriz com 60 palavras de 32 bits (contém
         todas as chaves de rodada) expandida da chave original
104   end component KeyExpansion;
```

Figura 5.2: *Componente KeyExpansion.*

Na figura 5.3, figura 5.4 e figura 5.5, são apresentados trechos do código VHDL da expansão da chave descrita no seção 3.3.5. Pode-se observar na figura 5.3, linhas 55 até 58, que a constante da expansão de chaves (***Rcon)*** foi definida internamente no código. Nas linhas 63 até 75, é definida a função ***funcRotWord***(), que efetua a rotação dos bytes da palavra de posição anterior à palavra da chave que está sendo calculada. Nas linhas 79 até 91, é definida a função ***funcSubWord***(), que efetua a substituição dos bytes utilizando-se da tabela S-Box (ver tabela 3.4) .

A função ***funcRotWord*** aceita como entrada uma palavra de 32 bits (padrão *std_logic_vector*) e fornece como saída uma palavra de 32 bits (*std_logic_vector*). Esta função executa um deslocamento cíclico para a esquerda, de um byte da palavra, conforme pode ser observado nas linhas 66 até 75 da figura 5.3.

A função ***funcSubWord*** está descrita nas linhas 82 até 91 da figura 5.3. Esta função possui como sinal de entrada um vetor binário no formato *std_logic_vector* de 32 bits, retornando também um vetor binário no formato *std_logic_vector* de 32 bits. Esta operação é realizada de forma a substituir cada byte da palavra de entrada pelo conteúdo previamente

armazenado em um registrador interno (S-Box do AES, ver tabela 3.4). O endereço da memória é obtido baseado em cada byte da palavra de entrada. Isto porque neste trabalho a S-Box foi implementada como um vetor unidimensional de 256 bits, conforme já descrito na seção 3.3.1.

```
55 -- a constante Rcon definida pelo algoritmo AES
56 constant Rcon : Rcon_type:= (
57 x"01000000", x"02000000", x"04000000", x"08000000", x"10000000", -- 1, 2, 3, 4, 5
58 x"20000000", x"40000000", x"80000000", x"1b000000", x"36000000"); -- 6, 7, 8, 9, 10

63 -- função funcRotWord
64 -- objetivo: Rotacionar a palavra em 1 byte para esquerda
65 -- Parâmetros: recebe uma palavra de 32 bits e retorna uma palavra de 32 bits com rotação de
1 byte à esquerda
66 function funcRotWord(signal W : in std_logic_vector(31 downto 0))
67        return std_logic_vector is
68        variable auxW : std_logic_vector(31 downto 0); -- variável auxiliar para efetuar a rotação
69 begin
70        auxW(31 downto 24) := W(23 downto 16);
71        auxW(23 downto 16) := W(15 downto 8);
72        auxW(15 downto 8) := W( 7 downto 0);
73         auxW( 7 downto 0) := W(31 downto 24);
74        return auxW;
75 end funcRotWord;

79 -- função funcSubWord
80 -- objetivo: Efetuar a substituição dos bytes utilizando-se a tabela SBox
81 -- Parâmetros: recebe a palavra de 32 bits e retorna uma palavra de 32 bits após a substituição
em SBox
82 function funcSubWord(signal W : in std_logic_vector(31 downto 0))
83        return std_logic_vector is
84        variable auxW : std_logic_vector(31 downto 0); -- variável auxiliar para efetuar a rotação
85 begin
86        auxW(31 downto 24) := SBox(conv_integer(W(31 downto 24)));
87        auxW(23 downto 16) := SBox(conv_integer(W(23 downto 16)));
88        auxW(15 downto 8) := SBox(conv_integer(W(15 downto 8)));
89         auxW( 7 downto 0) := SBox(conv_integer(W( 7 downto 0)));
90         return auxW;
91 end funcSubWord;
```

Figura 5.3: *Código VHDL da KeyExpansion (constante e funções).*

Na figura 5.4, linhas 112 até 116, são preenchidas as primeiras quatro palavras do vetor de saída, com os primeiros 128 bits da chave informada. Este procedimento é comum à expansão das chaves de 128, 192 e 256 bits. Se a chave for de 192 bits, serão preenchidas as próximas duas palavras com o restante da chave original; da mesma forma se a chave for de 256 bits, mais outras duas palavras serão preenchidas. As demais palavras que não forem

preenchidas diretamente com o conteúdo da chave original serão calculadas pelo processo de expansão.

```
112 -- atualizar as primeiras 4 palavras da chave expandida, com a chave informada
113 KeyExpandida_reg(0) <= KeyIn_pino( 0 to 31);
114 KeyExpandida_reg(1) <= KeyIn_pino( 32 to 63);
115 KeyExpandida_reg(2) <= KeyIn_pino( 64 to 95);
116 KeyExpandida_reg(3) <= KeyIn_pino( 96 to 127);
117
118 case TamanhoDaChave_pino is
119 when 1 => -- chave de 128 bits - obter as demais 40 palavras de 32 bits
121   KeyExpandida_reg( 4) <= funcSubWord(funcRotWord(KeyExpandida_reg( 3))) xor
        KeyExpandida_reg( 0) xor RCon(1); -- i é múltiplo de Nk
122   KeyExpandida_reg( 5) <= KeyExpandida_reg( 4) xor KeyExpandida_reg( 1);
123   KeyExpandida_reg( 6) <= KeyExpandida_reg( 5) xor KeyExpandida_reg( 2);
124   KeyExpandida_reg( 7) <= KeyExpandida_reg( 6) xor KeyExpandida_reg( 3);
125   KeyExpandida_reg( 8) <= funcSubWord(funcRotWord( KeyExpandida_reg( 7))) xor
        KeyExpandida_reg( 4) xor RCon(2); -- i é múltiplo de Nk
```

Figura 5.4: *Código VHDL da KeyExpansion (chave de 128 bits).*

O procedimento de expansão das chaves é semelhante para chaves de 128 e 192 bits, ou seja, sempre que o índice da palavra que está sendo calculado for múltiplo de **Nk**, serão efetuadas as transformações **RotWord** (funcRotWord), **SubWord** (funcSubWord) e a adição da constante da expansão de chaves **Rcon**. Na figura 5.4, linhas 118 até 129, é apresentado um trecho de código da expansão da chave de 128 bits, portanto **Nk**=4. Pode-se observar nas linhas 122 a 124 que o cálculo da palavra de índice "***i***" é efetuado através de uma operação XOR da palavra de posição anterior (índice "***i-1***") com a palavra **Nk** posições anteriores (índice "***i-Nk***"). No caso das palavras com índice "***i***", múltiplo de **Nk**, linhas 121 e 125, a palavra de posição anterior (índice "***i-1***") sofre duas transformações **RotWord** e **SubWord**, respectivamente pelas funções funcRotWord e funcSubWord, antes das operações XOR com a palavra **Nk** posições anteriores (índice "***i-Nk***"), e com a constante da expansão de chaves, ***Rcon***.

Na figura 5.5, linhas 220 até 240, é apresentado um trecho de código da expansão da chave de 256 bits, portanto **Nk**=8. Nas linhas 222 até 225 são preenchidos os vetores da chave expandida com os últimos 128 bits da chave informada, lembrando que os primeiros 128 bits já foram preenchidos anteriormente, ver figura 5.4 linhas 113 até 116.

```
220 when others => -- chave de 256 bits
221   -- atualizar as primeiras 8 palavras da chave expandida, com a chave informada
222   KeyExpandida_reg(4) <= KeyIn_pino(128 to 159);
223   KeyExpandida_reg(5) <= KeyIn_pino(160 to 191);
224   KeyExpandida_reg(6) <= KeyIn_pino(192 to 223);
225   KeyExpandida_reg(7) <= KeyIn_pino(224 to 255);
226
227   -- efetuar a expansão da chave propriamente dita para obter as demais 52 palavras de 32
        bits
228   KeyExpandida_reg( 8) <= funcSubWord(funcRotWord( KeyExpandida_reg( 7))) xor
        KeyExpandida_reg( 0) xor RCon(1); -- i é múltiplo de Nk
229 KeyExpandida_reg( 9) <= KeyExpandida_reg( 8) xor KeyExpandida_reg( 1);
230 KeyExpandida_reg(10) <= KeyExpandida_reg( 9) xor KeyExpandida_reg( 2);
231 KeyExpandida_reg(11) <= KeyExpandida_reg(10) xor KeyExpandida_reg( 3);
232 KeyExpandida_reg(12) <= funcSubWord(KeyExpandida_reg(11)) xor KeyExpandida_reg( 4); -
        - (i-4) é múltiplo de Nk
233 KeyExpandida_reg(13) <= KeyExpandida_reg(12) xor KeyExpandida_reg( 5);
234 KeyExpandida_reg(14) <= KeyExpandida_reg(13) xor KeyExpandida_reg( 6);
235 KeyExpandida_reg(15) <= KeyExpandida_reg(14) xor KeyExpandida_reg( 7);
236 KeyExpandida_reg(16) <= funcSubWord(funcRotWord (KeyExpandida_reg(15))) xor
        KeyExpandida_reg( 8) xor RCon(2); -- i é múltiplo de Nk
237 KeyExpandida_reg(17) <= KeyExpandida_reg(16) xor KeyExpandida_reg( 9);
238 KeyExpandida_reg(18) <= KeyExpandida_reg(17) xor KeyExpandida_reg(10);
239 KeyExpandida_reg(19) <= KeyExpandida_reg(18) xor KeyExpandida_reg(11);
240 KeyExpandida_reg(20) <= funcSubWord(KeyExpandida_reg(19)) xor KeyExpandida_reg(12);
        -- (i-4) é múltiplo de Nk
```

Figura 5.5: *Código VHDL da KeyExpansion (chave de 256 bits).*

Pode-se observar nas linhas 229, 230, 231, 233, 234, 235, 237, 238 e 239 que o cálculo da palavra de índice "*i*" é efetuado através de uma operação XOR da palavra anterior (índice "***i-1***") com a palavra **Nk** posições anteriores (índice "***i-Nk***"). No caso da palavra que está sendo calculada ser de índice "*i*", múltiplo de **Nk**, linhas 228 e 236, a palavra anterior (índice "***i-1***") sofre duas transformações ***RotWord*** e ***SubWord***, respectivamente pelas funções funcRotWord e funcSubWord, antes das operações XOR com a palavra **Nk** posições anteriores (índice "***i-Nk***") e com a constante de rodada da expansão de chaves, ***Rcon.*** Estes procedimentos de expansão são semelhantes aos das chaves de 128 e 192 bits. A diferença existente na expansão das chaves de 128 ou 192 bits e a expansão da chave de 256 bits reside no fato de que para esta última, as palavras que estão sendo calculadas, cujo índice "*i*" menos quatro ("***i-4***") forem múltiplas de **Nk**, terão como processo de cálculo uma transformação ***SubWord*** na palavra anterior (índice "***i-1***"), antes da operação XOR com a palavra **Nk** posições anteriores (índice "***i-Nk***"), como pode ser observado nas linhas 232 e 240 da figura 5.5.

Na simulação de expansão das chaves de 128, 192 e 256 bits, o sinal de saída "**KeyExpandida_pino**", que representa um vetor de 60 palavras com 32 bits cada uma, foi

alterado para uma palavra de 32 bits, isto devido ao fato do número de pinos de entrada e saída, disponíveis no componente alvo, estar limitado em 315, conforme tabela 5.1. Quando sintetizado em conjunto com os demais componentes, este sinal será considerado sinal interno, não apresentando mais esta limitação. Adotou-se também como padrão, nas três simulações, apresentar o valor da palavra da posição 43 (w[43]) da chave expandida.

No resultado da simulação de expansão de chave de 128 bits (figura 5.6), o sinal de entrada "**TamanhoDaChave_pino**" foi setado com valor 1, indicando que o módulo de expansão de chave irá operar com chave de 128 bits; a chave de entrada do sistema "**KeyIn_pino**" foi inserida com o valor x"2b7e151628aed2a6abf7158809cf4f3c" , conforme anexo A1 do FIPS-197, e os 128 bits menos significativos foram preenchidos com zeros, pois como informado anteriormente, o tamanho de "**KeyIn_pino**" é de 256 bits. Observa-se, inicialmente nesta simulação, que o sinal "**HabilitaExpansao_pino**" possui o valor zero, indicando que durante este intervalo o usuário poderá atribuir um novo valor de chave. Quando este sinal for colocado em nível lógico alto (t=10us) indica que o processo de expansão de chave é inicializado e o valor da chave é calculado. O sinal de saída "**KeyExpandida_pino**" representando o valor do resultado da expansão, para a palavra w[43], de 32 bits, e chave de 128 bits, conforme o vetor de testes do FIPS-197 deve ser x"b6630ca6", como indicado no resultado da simulação.

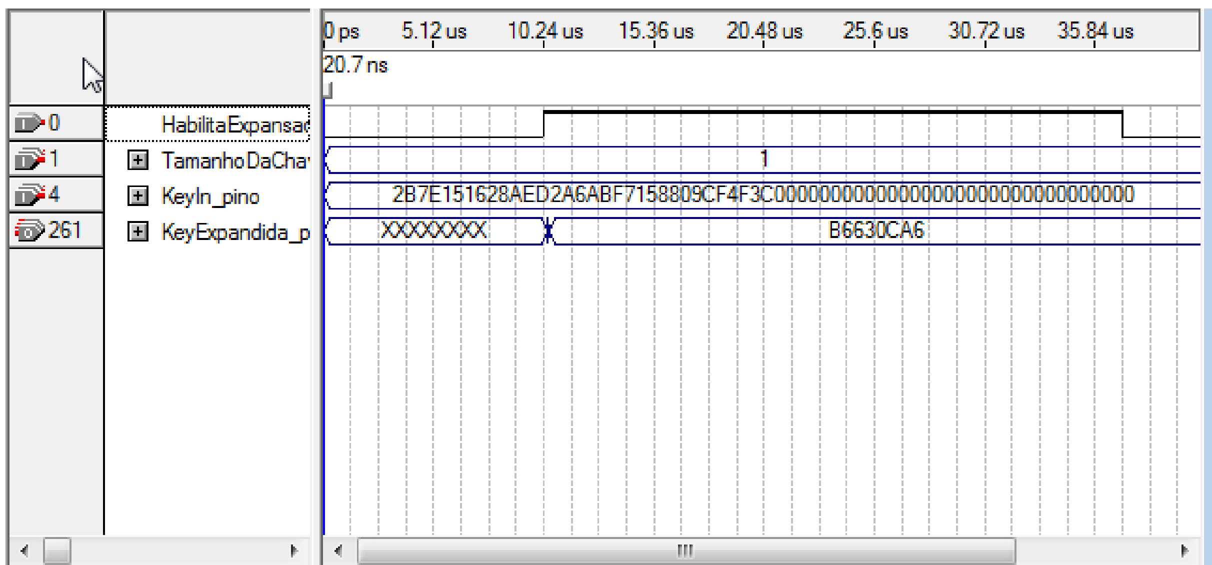

Figura 5.6: Simulação da expansão da chave de 128 bits.

No resultado da simulação de expansão de chave de 192 bits (figura 5.7), o sinal de entrada "**TamanhoDaChave_pino**" foi setado com valor 2, indicando que o módulo de expansão de chave irá operar com chave de 192 bits; a chave de entrada do sistema

“**KeyIn_pino**” foi inserida com o valor, conforme anexo A2 do FIPS-197, de x”8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b”. Os 64 bits menos significativos foram preenchidos com zeros, pois, como informado anteriormente, o tamanho de “**KeyIn_pino**” é de 256 bits. O resultado para a palavra w[43] expandida, conforme FIPS-197, deve ser x“ad07d753”, sendo o resultado obtido nesta simulação.

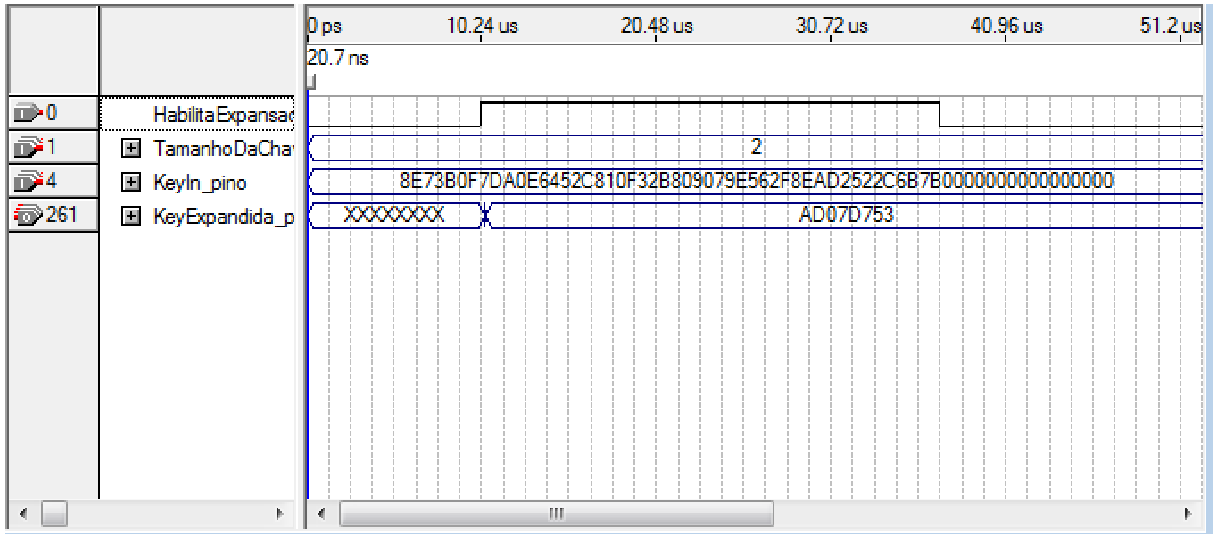

Figura 5.7: *Simulação da expansão da chave de 192 bits.*

No resultado da simulação de expansão de chave de 256 bits (figura 5.8), o sinal de entrada “**TamanhoDaChave_pino**” foi inicializada com o valor 3, indicando que o módulo de expansão de chave irá operar com chave de 256 bits. Tanto o valor x”603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4”, atribuído à chave de entrada “**KeyIn_pino**”, quanto o resultado x“9674ee15”, obtido nesta simulação para a palavra expandida w[43], constam no anexo A2, que é parte integrante do anexo FIPS-197 deste documento.

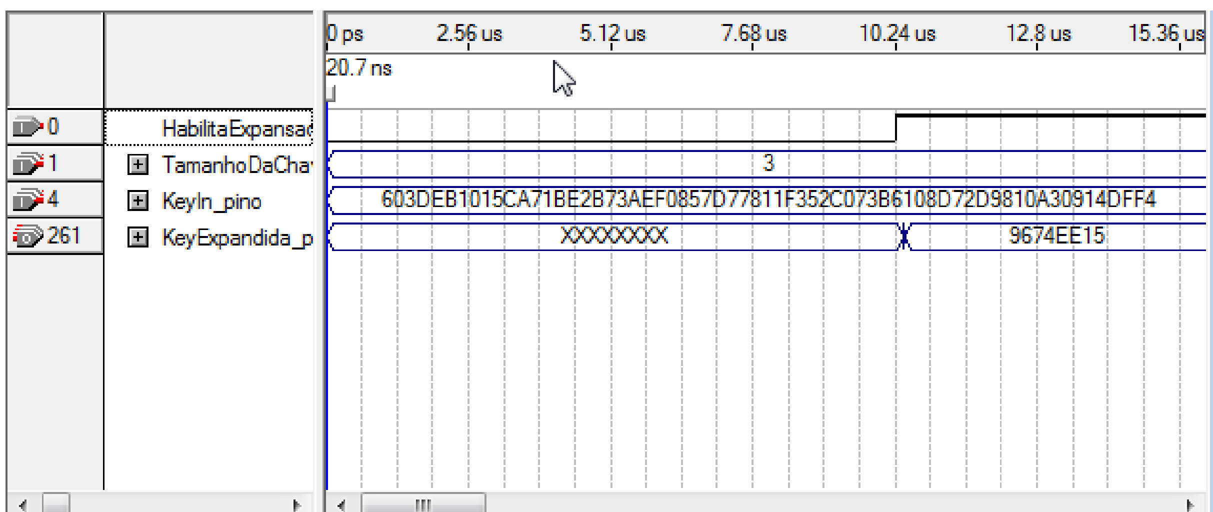

Figura 5.8: *Simulação da expansão da chave de 256 bits.*

5.3.2 – Adição de chave (AddRoundKey)

O componente de adição de chave (**AddRoundKey**), como já descrito anteriormente, irá efetuar uma operação XOR bit a bit, da chave da rodada com o bloco de texto que está sendo processado, e que está contido em *state*.

Como pode ser observado na figura 5.9, o componente possui como entradas os sinais:

- **StateIn**: (linha 108) é um vetor bidimensional (4x4) de bytes (std_logic_vector (7 downto 0)), que contém o bloco de dados que está sendo processado. Representa o state conforme definido na seção 3.2, figura 3.1.
- **StateOut**: (linha 109) possui a mesma estrutura que **StateIn**, e que conterá o bloco de dados após a adição da chave de rodada.
- **KeyRoundWord0:** contém a primeira palavra da chave de rodada (linha 101).
- **KeyRoundWord1:** contém a segunda palavra da chave de rodada (linha 102).
- **KeyRoundWord2:** contém a terceira palavra da chave de rodada (linha 103).
- **KeyRoundWord3:** contém a quarta palavra da chave de rodada (linha 104).

```
106 -- componente que efetua a adição da chave de rodada
107 component AddRoundKey is port (
108       StateIn : in state_type; -- state de entrada (array 4x4 de bytes)
109       StateOut : out state_type; -- state de saída (array 4x4 de bytes)
110       KeyRoundWord0 : in std_logic_vector(31 downto 0); -- contém a 1a palavra de 32 bits da
chave da rodada
111       KeyRoundWord1 : in std_logic_vector(31 downto 0); -- contém a 2a palavra de 32 bits da
chave da rodada
112       KeyRoundWord2 : in std_logic_vector(31 downto 0); -- contém a 3a palavra de 32 bits da
chave da rodada
113       KeyRoundWord3 : in std_logic_vector(31 downto 0)); -- contém a 4a palavra de 32 bits da
chave da rodada
114 end component AddRoundKey;
```

Figura 5.9: *Componente AddRoundKey.*

Na figura 5.10 é apresentado parte do código VHDL da adição de chave de rodada (AddRoundKey). A adição é processada conforme descrito na seção 3.3.4. Observar pelas linhas 42 até 45 que a operação XOR é feita, bit a bit, entre os 32 bits da primeira coluna de *state* com os 32 bits da primeira palavra da chave de rodada. Nas linhas 48 até 51 observa-se esta operação para a segunda coluna de *state* com a segunda palavra da chave de rodada. Este procedimento é repetido para as demais colunas, até a operação ser finalizada.

```
37 architecture MWSaad of AddRoundKey is
38 signal sigStateOut : state_type;
39
40 begin
41 -- efetuar a adição da "chave de rodada" com a 1a coluna de state
42 sigStateOut(0,0) <= StateIn(0,0) xor KeyRoundWord0(31 downto 24);
43 sigStateOut(1,0) <= StateIn(1,0) xor KeyRoundWord0(23 downto 16);
44 sigStateOut(2,0) <= StateIn(2,0) xor KeyRoundWord0(15 downto 8);
45 sigStateOut(3,0) <= StateIn(3,0) xor KeyRoundWord0( 7 downto 0);
46
47 -- efetuar a adição da "chave de rodada" com a 2a coluna de state
48 sigStateOut(0,1) <= StateIn(0,1) xor KeyRoundWord1(31 downto 24);
49 sigStateOut(1,1) <= StateIn(1,1) xor KeyRoundWord1(23 downto 16);
50 sigStateOut(2,1) <= StateIn(2,1) xor KeyRoundWord1(15 downto 8);
51 sigStateOut(3,1) <= StateIn(3,1) xor KeyRoundWord1( 7 downto 0);
```

Figura 5.10: *Código VHDL da AddRoundKey.*

No resultado da simulação (figura 5.11), da adição da chave de rodada (**AddRoundKey**), os sinais de entrada “**StateIn_W0**”, “**StateIn_W1**”, “**StateIn_W2**” e “**StateIn_W3**”, que representam, respectivamente, as quatro colunas de *state*, e que armazenam, portanto, os 4 bytes de cada uma das colunas, foram setados com os valores de *state* conforme “*Appendix B – Cipher Example*” do FIPS-197, considerando que no intervalo de 0 a 10us utilizou-se o valor x"3243f6a8885a308d313198a2e0370734” que representa o próprio bloco de texto plano. No intervalo de 10us até 20us utilizou-se o valor de x”046681e5e0cb199a48f8d37a2806264c” que é o valor de state da primeira rodada imediatamente antes da adição da chave de rodada. Nos intervalos de 20us até 30us e 30us até 40us foram utilizados, respectivamente, os valores da 9ª (nona) e 10ª (décima) rodadas. Os sinais de entrada “**KeyRounWord0**”, “**KeyRounWord1**”, “**KeyRounWord2**” e “**KeyRounWord3**”, que representam, respectivamente, as quatro palavras da chave de rodada, cada uma com 32 bits, foram setadas com os valores da chave de rodada conforme também o “*Appendix B – Cipher Example*” do FIPS-197, sendo que para o intervalo de 0 a 10us utilizou-se o valor x" 2b7e151628aed2a6abf7158809cf4f3c” da primeira chave de rodada, que corresponde inclusive ao valor original da chave. No intervalo de 10us até 20us utilizou-se o valor de x”046681e5e0cb199a48f8d37a2806264c” que corresponde ao valor da primeira chave expandida. Nos intervalos de 20us até 30us e 30us até 40us foram utilizados, respectivamente, os dois últimos valores de chave, da chave expandida, referentes às chaves da 9ª (nona) e 10ª (décima) rodadas. Os sinais de saída “**StateOut_W0**”, “**StateOut_W1**”, “**StateOut_W2**” e “**StateOut_W3**” representam, respectivamente, as quatro colunas de *state*,

após a transformação AddRoundKey. Os valores obtidos nesta simulação são os mesmos constantes no anexo C1 do FIPS-197.

	Name	0 ps – 10.0 us	10.0 us – 20.0 us	20.0 us – 30.0 us	30.0 us – 40.0 us
0	StateIn_W0	32, 43, F6, A8	04, 66, 81, E5	47, 37, 94, ED	E9, 31, 7D, B5
37	StateIn_W1	88, 5A, 30, 8D	E0, CB, 19, 9A	40, D4, E4, A5	CB, 32, 2C, 72
74	StateIn_W2	31, 31, 98, A2	48, F8, D3, 7A	A4, 70, 3A, A6	3D, 2E, 89, 5F
111	StateIn_W3	E0, 37, 07, 34	28, 06, 26, 4C	4C, 9F, 42, BC	AF, 09, 07, 94
148	KeyRoundWord0	2B7E1516	A0FAFE17	AC7766F3	D014F9A8
181	KeyRoundWord1	28AED2A6	88542CB1	19FADC21	C9EE2589
214	KeyRoundWord2	ABF71588	23A33939	28D12941	E13F0CCB
247	KeyRoundWord3	09CF4F3C	2A6C7605	575C006E	B6630CA6
280	StateOut_W0	19, 3D, E3, BE	A4, 9C, 7F, F2	EB, 40, F2, 1E	39, 25, 84, 1D
317	StateOut_W1	A0, F4, E2, 2B	68, 9F, 35, 2B	59, 2E, 38, 84	02, DC, 09, FB
354	StateOut_W2	9A, C6, 8D, 2A	6B, 5B, EA, 43	8C, A1, 13, E7	DC, 11, 85, 94
391	StateOut_W3	E9, F8, 48, 08	02, 6A, 50, 49	1B, C3, 42, D2	19, 6A, 0B, 32

Figura 5.11: *Simulação do AddRoundKey.*

5.3.3 – Substituição de bytes (SubBytes)

O componente de substituição de bytes (**SubBytes**), como já descrito na seção 3.3.1, efetua a substituição de cada byte do *state* de entrada "StateIn" por um outro byte constante da tabela S-Box.

Como pode ser observado na figura 5.12, o componente possui como entrada os sinais:

- **StateIn**: (linha 39) é um vetor bidimensional (4x4) de bytes (std_logic_vector (7 downto 0)), que contém o bloco de dados que está sendo processado. Representa o *state* conforme definido na seção 3.2, figura 3.1.
- **StateOut**: (linha 40) possui a mesma estrutura que **StateIn**, e que conterá o bloco de dados após a substituição dos bytes.

```
36 -- componente que efetua a substituição dos bytes de state
37 component SubBytes
38 port (
39       StateIn : in state_type; -- state de entrada (array 4x4 de bytes) de std_logic_vector(7
         downto 0)
40       StateOut: out state_type); -- state de saída (array 4x4 de bytes) de std_logic_vector(7
         downto 0)
41 end component SubBytes;
```

Figura 5.12: *Componente SubBytes.*

A figura 5.13 apresenta parte do código VHDL da substituição de bytes (***SubBytes***). Esta operação (linha 39) é realizada de forma a substituir cada byte do *state* de entrada

(“***StateIn***”) pelo conteúdo previamente armazenado em uma memória interna (S-Box do AES, ver tabela 3.4).

```
30 architecture MWSaad of SubBytes is
31 signal StateOut_reg : state_type;
32 begin
33
34 TratarLinha_SBox:
35 for row in 0 to 3 generate
36    TratarColuna_SBox:
37    for col in 0 to 3 generate
38       -- efetuar a substituição de bytes propriamente dita
39       StateOut_reg(row, col) <= SBox(conv_integer(StateIn(row, col)));
40
41     end generate TratarColuna_SBox;
42 end generate TratarLinha_SBox;
43
44 -- atualizar "state" após processar os deslocamentos de bytes
45 StateOut <= StateOut_reg;
46
47 end architecture MWSaad;
```

Figura 5.13: *Código VHDL da SubBytes.*

No resultado da simulação (figura 5.14) da substituição de bytes (**SubBytes**), os sinais de entrada “**StateIn_W0**”, “**StateIn_W1**”, “**StateIn_W2**” e “**StateIn_W3**”, que representam, respectivamente, as quatro colunas de *state*, e que armazenam, portanto, os 4 bytes de cada uma das colunas, foram setados com os valores de *state* conforme “*Appendix B – Cipher Example*” do FIPS-197. Utilizou-se nos intervalos de 0us a 10us, 10us a 20us, 20us a 30us e 30 a 40us, respectivamente os valores de state, da primeira até a quarta rodada, imediatamente anteriores da transformação SubBytes. Os sinais de saída “**StateOut_W0**”, “**StateOut_W1**”, “**StateOut_W2**” e “**StateOut_W3**” representam, respectivamente, as quatro colunas de *state*, após a transformação **SubBytes**. Os valores obtidos nesta simulação são os mesmos constantes no “*Appendix B – Cipher Example*” do FIPS-197.

Coverage		0 ps – 10.0 us	10.0 us – 20.0 us	20.0 us – 30.0 us	30.0 us – 40.0 us
0	StateIn_W0	19, 3D, E3, BE	A4, 9C, 7F, F2	AA, 8F, 5F, 03	48, 6C, 4E, EE
37	StateIn_W1	A0, F4, E2, 2B	68, 9F, 35, 2B	61, DD, E3, EF	67, 1D, 9D, 0D
74	StateIn_W2	9A, C6, 8D, 2A	6B, 5B, EA, 43	82, D2, 4A, D2	4D, E3, B1, 38
111	StateIn_W3	E9, F8, 48, 08	02, 6A, 50, 49	68, 32, 46, 9A	D6, 5F, 58, E7
148	StateOut_W0	D4, 27, 11, AE	49, DE, D2, 89	AC, 73, CF, 7B	52, 50, 2F, 28
185	StateOut_W1	E0, BF, 98, F1	45, DB, 96, F1	EF, C1, 11, DF	85, A4, 5E, D7
222	StateOut_W2	B8, B4, 5D, E5	7F, 39, 87, 1A	13, B5, D6, B5	E3, 11, C8, 07
259	StateOut_W3	1E, 41, 52, 30	77, 02, 53, 3B	45, 23, 5A, B8	F6, CF, 6A, 94

Figura 5.14: *Simulação do SubBytes.*

5.3.4 – Deslocamento de linhas (ShiftRows)

O componente de deslocamento de linhas (**ShiftRows**), como já descrito na seção 3.3.2, efetua um deslocamento cíclico de bytes, para esquerda, nas três últimas linhas de state. O número de bytes deslocados é apresentado na tabela 3.5.

Como pode ser observado na figura 5.15, o componente possui como entrada os sinais:

- **StateIn**: (linha 45) é um vetor bidimensional (4x4) de bytes (std_logic_vector (7 downto 0)), que contém o bloco de dados que está sendo processado. Representa o *state* conforme definido na seção 3.2, figura 3.1.
- **StateOut**: (linha 48) possui a mesma estrutura que **StateIn**, e que conterá o bloco de dados após a substituição dos bytes.

```
45 component ShiftRows
46 port (
47      StateIn : in state_type; -- state de entrada (array 4x4 de bytes) de std_logic_vector(7
        downto 0)
48      StateOut: out state_type); -- state de saída (array 4x4 de bytes) de std_logic_vector(7
        downto 0)
49 end component ShiftRows;
```

Figura 5.15: *Componente ShiftRows.*

Na figura 5.16 é apresentado parte do código VHDL do componente ***ShiftRows***.

```
49 -- 1a linha não sofre deslocamento
50 StateOut_reg(0, 0) <= StateIn(0, 0);
51 StateOut_reg(0, 1) <= StateIn(0, 1);
52 StateOut_reg(0, 2) <= StateIn(0, 2);
53 StateOut_reg(0, 3) <= StateIn(0, 3);
54
55 -- 2a linha sofre deslocamento de 1 byte
56 StateOut_reg(1, 0) <= StateIn(1, 1);
57 StateOut_reg(1, 1) <= StateIn(1, 2);
58 StateOut_reg(1, 2) <= StateIn(1, 3);
59 StateOut_reg(1, 3) <= StateIn(1, 0);
60
61 -- 3a linha sofre deslocamento de 2 bytes
62 StateOut_reg(2, 0) <= StateIn(2, 2);
63 StateOut_reg(2, 1) <= StateIn(2, 3);
64 StateOut_reg(2, 2) <= StateIn(2, 0);
65 StateOut_reg(2, 3) <= StateIn(2, 1);
66
67 -- 4a linha sofre deslocamento de 3 bytes
68 StateOut_reg(3, 0) <= StateIn(3, 3);
69 StateOut_reg(3, 1) <= StateIn(3, 0);
70 StateOut_reg(3, 2) <= StateIn(3, 1);
71 StateOut_reg(3, 3) <= StateIn(3, 2);
```

Figura 5.16: *Código VHDL da ShiftRows.*

Observa-se que a primeira linha de *state* não sofre alteração (linhas 50 a 53); a segunda linha sofre um deslocamento de um byte para a esquerda (linhas 56 a 59); a terceira linha sofre um deslocamento de dois bytes para esquerda (linhas 62 a 65); e a terceira linha sofre um deslocamento de 3 bytes esquerda (linhas 68 a 71).

No resultado da simulação (figura 5.17) do deslocamento de linhas (**ShiftRows**), os sinais de entrada "**StateIn_W0**", "**StateIn_W1**", "**StateIn_W2**" e "**StateIn_W3**", que representam, respectivamente, as quatro colunas de *state*, e que armazenam, portanto, os 4 bytes de cada uma das colunas, foram setados com os valores de *state* conforme "*Appendix B – Cipher Example*" do FIPS-197. Utilizou-se nos intervalos de 0us a 10us, 10us a 20us, 20us a 30us e 30 a 40us, respectivamente os valores de state, da primeira até a quarta rodada, imediatamente anteriores à transformação **ShiftRows**. Os sinais de saída "**StateOut_W0**", "**StateOut_W1**", "**StateOut_W2**" e "**StateOut_W3**" representam, respectivamente, as quatro colunas de *state*, após a transformação **ShiftRows**. Os valores obtidos nesta simulação são os mesmos constantes no "*Appendix B – Cipher Example*" do FIPS-197.

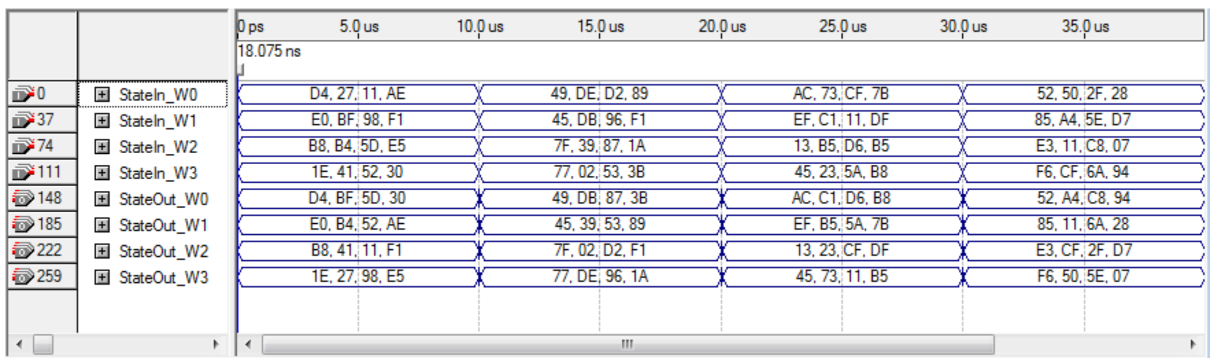

Figura 5.17: *Simulação do ShiftRows.*

5.3.5 - Mistura de colunas (MixColumns)

O componente mistura de colunas (**MixColumns**), como já descrito na seção 3.3.3, efetua uma operação com os quatro bytes de cada coluna de state, tratando cada coluna como um polinômio de quatro termos sobre $GF(2^8)$ e multiplicadas módulo $x^4 + 1$ com um polinômio fixo a(x), dado por: $\boldsymbol{a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}}$.

Como pode ser observado na figura 5.18, o componente possui como entrada os sinais:

- StateIn: (linha 51) é um vetor bidimensional (4x4) de bytes (std_logic_vector (7 downto 0)), que contém o bloco de dados que está sendo processado. Representa o *state* conforme definido na seção 3.2, figura 3.1.

- StateOut: (linha 55) possui a mesma estrutura que StateIn, e que conterá o bloco de dados após a substituição dos bytes.

```
51 -- componente que efetua o Mix (embaralhamento) dos bits dos bytes das colunas, segundo
      operações no GF2^8
52 component MixColumns
53 port (
54       StateIn : in state_type; -- state de entrada (array 4x4 de bytes) de std_logic_vector(7
         downto 0)
55       StateOut: out state_type); -- state de saída (array 4x4 de bytes) de std_logic_vector(7
         downto 0)
56 end component MixColumns;
```

Figura 5.18: *Componente MixColumns.*

Na figura 5.19 é apresentado parte do código VHDL do componente ***MixColumns***. Conforme a definição do AES, a multiplicação no Corpo de Galois $GF2^8$ (Galois Field 2^8) pode ser implementado como sendo uma multiplicação de matrizes e como resultado desta multiplicação, os quatro bytes de uma coluna são substituídos pelos seguintes:

$$\mathbf{S'_{0,c} = (\{02\} \bullet S_{0,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c}}$$

$$\mathbf{S'_{1,c} = S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c}}$$

$$\mathbf{S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c})}$$

$$\mathbf{S'_{3,c} = (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c})}$$

onde: $\boldsymbol{c}$ representa a coluna de *state* ($0 \leq \boldsymbol{c} \leq 3$); $\boldsymbol{S}$ representa o byte antes da transformação e $\boldsymbol{S'}$ representa o byte após a transformação.

Como a multiplicação de {02} por um número Y[7..0] ({02}.Y) equivale a um deslocamento de um bit para esquerda dos bits de Y, sendo que se o bit mais significativo (antes do deslocamento) for ‘1’ (Y[7] = ‘1’), além do deslocamento deve-se efetuar uma operação XOR (ou exclusivo bit a bit) com a constante x"1B" (x”0001 1011”). Uma solução para não ser necessário implementar testes para verificar se Y[7] = ‘1’ é efetuar um XOR com Z onde Z é obtido por:

Z = x"1B" and ("000" & Y[7] & "0" & Y[7] & Y[7])

Desta forma se Y[7] for igual a ‘0’, Z resultará em "00000000" em binário (x"00" em hexadecimal), o que não altera a operação; se Y[7] for igual a ‘1’, Z resultará em "00011011" em binário (x"1B" em hexadecimal) conforme desejado.

A função "**GFMult02**" está descrita nas linhas 74 a 81 da figura 5.19. Esta função possui como sinal de entrada um vetor binário no formato *std_logic_vector* de 8 bits, representando um byte de state, retornando também um vetor binário no formato *std_logic_vector* de 8 bits. A multiplicação do byte por {02} está implementada na linha 78.

A função "**GFMult03**", descrita nas linhas 95 a 102, possui os sinais de entrada e saída iguais aos da função "**GFMult02**" descrita no parágrafo anterior. Esta função efetua a multiplicação do byte por {03}, que é implementada na forma da multiplicação do byte por {02} xor o próprio byte, como pode ser observado na linha 99

```
54 -- Objetivo: Efetuar uma multiplicação por {02} no GF2^8

74 function GFMult02(signal ByteIn : in Byte_type) -- byte de entrada
75      return Byte_type is -- byte de saída após a multiplicação por {02} no GF2^8
76      variable varByte : Byte_type; -- variável auxiliar para efetuar a operação
77 begin
78      varByte := (ByteIn(6 downto 0) & '0') xor (x"1B" and ("000" & ByteIn(7) & ByteIn(7) & '0' &
        ByteIn(7) & ByteIn(7)));
79      -- retornar o byte após a múltiplicação por {02} no GF2^8
80      return varByte;
81 end GFMult02;


88 -- Objetivo: Efetuar uma multiplicação por {03} no GF2^8

95 function GFMult03(signal ByteIn : in Byte_type) -- byte de entrada
96       return Byte_type is -- byte de saída após a multiplicação por {02} no GF2^8
97       variable varByte : Byte_type; -- variável auxiliar para efetuar a operação
98 begin
99       varByte := GFMult02(ByteIn) xor  ByteIn;
100      -- retornar o byte após a múltiplicação por {03} no GF2^8
101     return varByte;
102 end GFMult03;
```

Figura 5.19: *Código VHDL das funções da MixColumns.*

Na figura 5.20, linhas 109 a 112, pode-se observar a transformação de cada um dos bytes da primeira coluna de *state*. Nas linhas 109, 110, 111 e 112 são efetuadas as transformações, respectivamente, para o 1º, 2º, 3º e 4º bytes da primeira coluna. Nas linhas 115 a 118 são efetuadas as transformações para a segunda coluna de *state*. O procedimento é o mesmo para as demais colunas.

```
107 begin
108 -- 1a coluna (índice 0)
109 StateOut_reg(0, 0) <= GFMult02(StateIn(0, 0)) xor GFMult03(StateIn(1, 0)) xor StateIn(2, 0)
      xor StateIn(3,0);
110 StateOut_reg(1, 0) <= StateIn(0, 0) xor GFMult02(StateIn(1, 0)) xor GFMult03(StateIn(2, 0))
      xor StateIn(3, 0);
111 StateOut_reg(2, 0) <= StateIn(0, 0) xor StateIn(1, 0) xor GFMult02(StateIn(2, 0)) xor
      GFMult03(StateIn(3,0));
112 StateOut_reg(3, 0) <= GFMult03(StateIn(0, 0)) xor StateIn(1, 0) xor StateIn(2, 0) xor
      GFMult02(StateIn(3,0));
113
114 -- 2a coluna (índice 1)
115 StateOut_reg(0, 1) <= GFMult02(StateIn(0, 1)) xor GFMult03(StateIn(1, 1)) xor StateIn(2, 1)
      xor StateIn(3,1);
116 StateOut_reg(1, 1) <= StateIn(0, 1) xor GFMult02(StateIn(1, 1)) xor GFMult03(StateIn(2, 1))
      xor StateIn(3,1);
117 StateOut_reg(2, 1) <= StateIn(0, 1) xor StateIn(1, 1) xor GFMult02(StateIn(2, 1)) xor
      GFMult03(StateIn(3,1));
118 StateOut_reg(3, 1) <= GFMult03(StateIn(0, 1)) xor StateIn(1, 1) xor StateIn(2, 1) xor
      GFMult02(StateIn(3,1));
119
```

Figura 5.20: *Código VHDL da MixColumns.*

No resultado da simulação (figura 5.21) da transformação **MixColumns**, os sinais de entrada "**StateIn_W0**", "**StateIn_W1**", "**StateIn_W2**" e "**StateIn_W3**", que representam, respectivamente, as quatro colunas de *state*, e que armazenam, portanto, os 4 bytes de cada uma das colunas, foram setados com os valores de *state* conforme "*Appendix B – Cipher Example*" do FIPS-197. Utilizou-se nos intervalos de 0us a 10us, 10us a 20us, 20us a 30us e 30 a 40us, respectivamente os valores de *state*, da primeira até a quarta rodada, imediatamente anteriores à transformação **MixColumns**. Os sinais de saída "**StateOut_W0**", "**StateOut_W1**", "**StateOut_W2**" e "**StateOut_W3**" representam, respectivamente, as quatro colunas de *state*, após a transformação **MixColumns**. Os valores obtidos nesta simulação são os mesmos constantes no "*Appendix B – Cipher Example*" do FIPS-197.

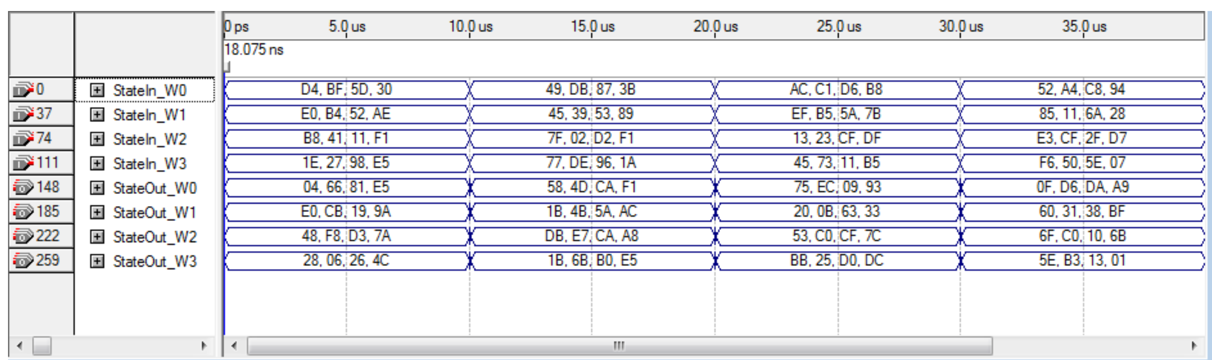

Figura 5.21: *Simulação do MixColumns.*

5.4 – Entradas e saídas

Na figura 5.22, linhas 19 até 30, são definidos os pinos de entrada e saída do circuito:

- *reset_pino*: reset ativo alto. Quando acionado, reinicializa todos os sinais de entrada e saída e reinicializa o processamento (linha 22);
- *clk_pino*: sinal de clock (linha 23);
- *Fim_pino*: controla o final do processamento criptográfico, podendo ser 0 ou 1 respectivamente para manter ou finalizar o processo de criptografia (linha 24);
- *TamanhoDaChave_pino*: definição do tamanho da chave desejada, podendo ser 1, 2 ou 3, respectivamente para as chaves de tamanho 128, 192 ou 256 bits (linha 25);
- *KeyIn_pino*: chave informada pelo usuário. A chave será processada em blocos de 32 bits, podendo ter um total de 128, 192 ou 256 bits (linha 26);
- *TextoPlano_pino*: o texto plano (claro) a ser criptografado, com tamanho de 128 bits (linha 27);

E como saída:

- *TextoCifrado_pino*: o texto criptografado, com tamanho de 128 bits (linha 28).

```
19 entity AES is
20 port
21 (
22   reset_pino : in STD_LOGIC;
23   clk_pino : in STD_LOGIC;
24   Fim_pino : in std_logic;
25   KeyIn_pino : in std_logic_vector(0 to 31);
26   TamanhoDaChave_pino : in integer range 1 to 3; -- opção pela utilização de chave de 128,
     192 e 256 bits, respectivamente =1,2 ou 3
27   TextoPlano_pino : in std_logic_vector(0 to 127); -- texto original (plano), antes de ser
     criptografado
28   TextoCifrado_pino : out std_logic_vector(0 to 127) -- texto criptografado
29 );
30 end AES;
```

Figura 5.22: *Entradas e saídas.*

5.5 – Unidade de controle

A gerência do processamento é feita por uma unidade de controle, utilizando-se de um sinal de relógio externo e do tamanho da chave escolhida. Para implementação foi definida uma máquina de estados com quatro estados (“Estado_0”, “Estado_1”, “Estado_2” e “Estado_3”).

Na figura 5.23, pode-se observar que no “Estado_0” é realizado o controle de entrada das chaves de forma que, a cada pulso do relógio, são lidos 32 bits da chave.

```
416 case EstadoAtual is
417 when Estado_0 =>
418     ProximoEstado <= Estado_0;
419     HabilitaExpansao <= '0'; -- desabilita a expansão das chaves
420     -------------------------------------------------
421     -- carrregar a chave em blocos de 32 bits
422     case ctPulsos is
423     when 1 =>
424         KeyIn256bits_reg( 0 to 31) <= KeyIn32bits_reg; -- carregar 32bits da chave (a chave será
        carregada em blocos de 32bits)
425     when 2 =>
426         KeyIn256bits_reg( 32 to 63) <= KeyIn32bits_reg; -- carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
427     when 3 =>
428         KeyIn256bits_reg( 64 to 95) <= KeyIn32bits_reg; -- carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
429     when 4 =>
430         KeyIn256bits_reg( 96 to 127) <= KeyIn32bits_reg; -- carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
431          -- neste ponto, se a chave for de 128 bits, já foi carregada completamente
432         if (TamanhoDaChave_reg = 1) then
433             HabilitaExpansao <= '1'; -- habilita a expansão das chaves
434             ProximoEstado <= Estado_1; -- definir o próximo estado (no Estado_1 ocorrerá a
            expansão da chave)
435         else
436             ProximoEstado <= Estado_0; -- manter o Estado_0 (instrução implementada para não
            criar latch)
437         end if;
438     when 5 =>
439         KeyIn256bits_reg(128 to 159) <= KeyIn32bits_reg; --carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
440     when 6 =>
441         KeyIn256bits_reg(160 to 191) <= KeyIn32bits_reg; --carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
442         -- se já carregou a chave completamente com 128 bits
443         if (TamanhoDaChave_reg = 2) then
444             HabilitaExpansao <= '1'; -- habilita a expansão das chaves
445             ProximoEstado <= Estado_1; -- definir o próximo estado (no Estado_1 ocorrerá a
            expansão da chave)
446         else
447             ProximoEstado <= Estado_0; -- manter o Estado_0 (instrução implementada para não
            criar latch)
448         end if;
449      when 7 =>
450         KeyIn256bits_reg(192 to 223) <= KeyIn32bits_reg; --carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
451     when others => -- quando chegar nesta opção, irá carregar o último bloco (de 32bits) da
    chave e mudará o estado
452         KeyIn256bits_reg(224 to 255) <= KeyIn32bits_reg; --carregar 32bits da chave (a chave
        será carregada em blocos de 32bits)
453         --HabilitaExpansaoDaChave <= '1'; -- habilita o cálculo da expansão de chaves
454         HabilitaExpansao <= '1'; -- habilita a expansão das chaves
455         ProximoEstado <= Estado_1; -- definir o próximo estado (no Estado_1 ocorrerá a
        expansão da chave)
456 end case; --case ctBlocosDaChave is
```

Figura 5.23: *Trecho de código VHDL do "Estado_0".*

O controle permanecerá no “Estado_0” até que a chave seja carregada totalmente, o que pode variar entre 4, 6 ou 8 pulsos de relógio (linhas 429, 440 e 451), para as chaves de 128, 192 e 256 bits respectivamente. Após o carregamento total da chave, é habilitada a expansão da mesma (linhas 630, 643 e 655), como também o controle é atualizado para o “Estado_1” (linhas 434, 445 e 455).

No “Estado_1” (figura 5.24), que tem a duração de um pulso de relógio, é processada a expansão da chave. O controle passa de forma incondicional para o “Estado_2” e desabilita a expansão das chaves.

```
459 when Estado_1 =>
460     -- neste estado será efetuado o cálculo da expansão da chave de forma concorrente
        (enquanto HabilitaExpansao = 1)
461     ProximoEstado <= Estado_2;
462     -- desabilitar a expansão da chaves
463     HabilitaExpansao <= '0';
```

Figura 5.24: *Trecho de código VHDL do “Estado_1”.*

No “Estado_2” ocorrerá efetivamente a criptografia. O processamento permanecerá neste estado até que seja setado, através de sinal externo, o sinal “Fim_reg” com o valor ‘1’.

A cada pulso do relógio (borda de subida) a unidade de controle irá atualizar os sinais de entrada de cada módulo com os sinais de saída do módulo anterior (figura 5.25). Desta forma, o processamento ocorrerá de modo “enfileirado”, ou seja, num primeiro pulso o primeiro bloco de texto plano é transferido para um sinal interno que é a entrada do primeiro módulo de adição de chave de rodada (**AddRoundKey**); num próximo pulso de relógio, o sinal de saída deste módulo é transferido para o sinal de entrada do próximo módulo (1ª Rodada Padrão) e novamente um novo bloco de texto plano (próximo bloco de texto a ser criptografado) é transferido para o sinal interno que é a entrada do primeiro módulo de adição de chave. No próximo pulso de relógio, o sinal de saída da “1ª Rodada Padrão” é transferido para o sinal de entrada do módulo “2ª Rodada Padrão”, a saída da primeira adição de chave é transferida para a entrada da “1ª Rodada Padrão” e um novo bloco de texto plano é transferido para a entrada da primeira adição de chave, e assim sucessivamente até o módulo de “Rodada Final”.

A unidade de controle irá direcionar o sinal de saída do módulo correspondente a penúltima rodada, em função do tamanho da chave, ou seja, saída da 9ª, 11ª ou 13ª “Rodada Padrão”, respectivamente para chaves de 128, 192 ou 256 bits, para a entrada do módulo da “Rodada Final” (linhas) e irá “habilitar” a saída do texto criptografado. Desta forma, ao final

de 17, 21 ou 25 pulsos de relógio, respectivamente para as chaves de 128, 192 ou 256 bits, sairá o primeiro bloco criptografado, referente ao primeiro bloco de texto plano que entrou. A partir deste ponto, cada novo pulso dará saída a um novo bloco de texto criptografado.

```
358 TextoPlano_reg <= TextoPlano_pino;
359 StateIn01_reg <= StateOut00_reg; -- entrada da 1a rodada recebe saída do 1o AddRoundKey
360 StateIn02_reg <= StateOut01_reg; -- entrada da 2a rodada recebe saída da 1a rodada
361 StateIn03_reg <= StateOut02_reg; -- entrada da 3a rodada recebe saída da 2a rodada
362 StateIn04_reg <= StateOut03_reg; -- entrada da 4a rodada recebe saída da 3a rodada
363 StateIn05_reg <= StateOut04_reg; -- entrada da 5a rodada recebe saída da 4a rodada
364 StateIn06_reg <= StateOut05_reg; -- entrada da 6a rodada recebe saída da 5a rodada
365 StateIn07_reg <= StateOut06_reg; -- entrada da 7a rodada recebe saída da 6a rodada
366 StateIn08_reg <= StateOut07_reg; -- entrada da 8a rodada recebe saída da 7a rodada
367 StateIn09_reg <= StateOut08_reg; -- entrada da 9a rodada recebe saída da 8a rodada
368 StateIn10_reg <= StateOut09_reg; -- para as chaves de 192 e 256bits
369 StateIn11_reg <= StateOut10_reg; -- para as chaves de 192 e 256bits
370 StateIn12_reg <= StateOut11_reg; -- somente para chaves 256bits
371 StateIn13_reg <= StateOut12_reg; -- somente para chaves 256bits

. . .

489 -- O texto criptografado será a saída da RodadaFinal. Os sinais da rodada final dependem do
tamanho da chave
490 case TamanhoDaChave_reg is
491 when 1 => --chave de 128 bits
492       StateInFinal_reg <= StateOut09_reg; -- state da penúltima rodada para chaves de 128 bits
493        FinalKeyRoundW0_reg <= KeyExpandida_reg(40); -- palavra de 32 bits da rodada final
494       FinalKeyRoundW1_reg <= KeyExpandida_reg(41); -- palavra de 32 bits da rodada final
495       FinalKeyRoundW2_reg <= KeyExpandida_reg(42); -- palavra de 32 bits da rodada final
496       FinalKeyRoundW3_reg <= KeyExpandida_reg(43); -- palavra de 32 bits da rodada final
497
498 when 2 => --chave de 192 bits
499       StateInFinal_reg <= StateOut11_reg; -- state da penúltima rodada para chaves de 192 bits
500       FinalKeyRoundW0_reg <= KeyExpandida_reg(48); -- palavra de 32 bits da rodada final
501       FinalKeyRoundW1_reg <= KeyExpandida_reg(49); -- palavra de 32 bits da rodada final
502       FinalKeyRoundW2_reg <= KeyExpandida_reg(50); -- palavra de 32 bits da rodada final
503       FinalKeyRoundW3_reg <= KeyExpandida_reg(51); -- palavra de 32 bits da rodada final
504
505 when others => --chave de 256 bits
506       StateInFinal_reg <= StateOut13_reg; -- state da penúltima rodada para chaves de 256 bits
507       FinalKeyRoundW0_reg <= KeyExpandida_reg(56); -- palavra de 32 bits da rodada final
508       FinalKeyRoundW1_reg <= KeyExpandida_reg(57); -- palavra de 32 bits da rodada final
509       FinalKeyRoundW2_reg <= KeyExpandida_reg(58); -- palavra de 32 bits da rodada final
510       FinalKeyRoundW3_reg <= KeyExpandida_reg(59); -- palavra de 32 bits da rodada final
511 end case; -- TamanhoDaChave_reg
```

Figura 5.25: *Trecho de código VHDL de atualização de sinais internos.*

Na figura 5.26 é apresentado o resultado da simulação do algoritmo com os vetores de teste do FIPS-197 para tamanho de chave de 128 bits e bloco de texto plano também de 128 bits, sendo que os valores obtidos nesta simulação são os mesmos do FIPS-197.

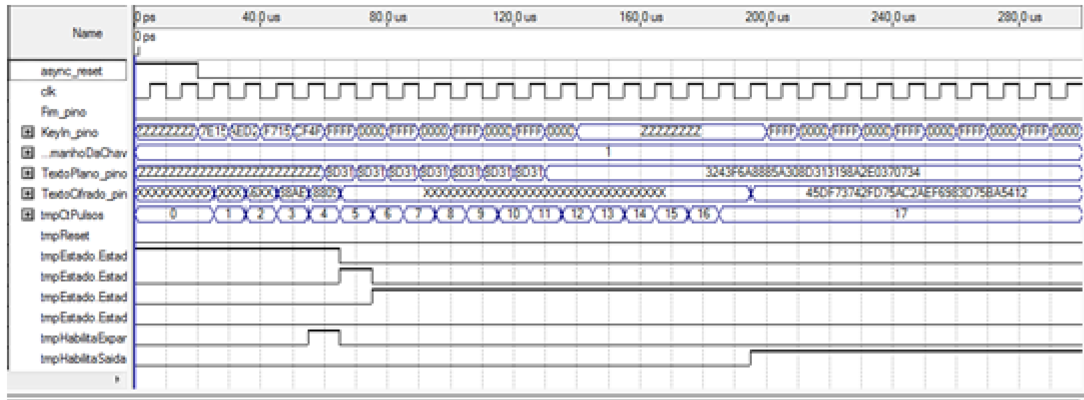

Figura 5.26: *Resultado de uma criptografia completa com chave de 128 bits.*

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação realizou-se a implementação do algoritmo de criptografia AES. Inicialmente foi apresentada a implementação de todos os módulos funcionais necessários para este algoritmo. Todos estes módulos foram sintetizados e o seu desempenho foi avaliado via simulação e posteriormente via implementação no FPGA. A análise dos módulos do algoritmo realizadas no FPGA foi baseada na utilização da ferramenta SignalTap disponível no software Quartus II do ALTERA.

Numa segunda etapa do trabalho, avaliou-se o desempenho global do algoritmo AES com tamanhos de chaves distintos, 128, 192 e 256 bits. Ressaltamos que o código desenvolvido neste trabalho flexibiliza a escolha do tamanho de chave pelo usuário. Para que o sistema em hardware possa realizar tal tarefa, foi necessária a inclusão de rotinas distintas que são função do tamanho da chave do algoritmo e que, portanto, tal flexibilidade teve como custo um aumento de área programável do componente.

Outro ponto que merece destaque na implementação do algoritmo é que o sistema executa de forma completamente paralela o algoritmo da criptografia. Portanto, após o primeiro ciclo completo do algoritmo criptográfico, a cada novo pulso de relógio um novo texto é criptografado e disponibilizado para o usuário.

Validou-se o algoritmo criptográfico através do uso de um *bench mark* fornecido na documentação FIPS-197. Com estes dados gerou-se um vetor de testes que foi usado na simulação em que validamos o funcionamento do projeto deste algoritmo, usando-se VHDL para implementá-lo num dispositivo reconfigurável.

Como trabalho futuro, propomos que o modelo desenvolvido para o algoritmo AES seja sintetizado/gravado em um elemento reconfigurável, para que se possa verificar o desempenho real deste quando implementado fisicamente. Outro aspecto a ser trabalhado é com relação à uma interface de entrada e saída, pois se trabalharmos com os dados de forma completamente paralelo, temos uma impossibilidade de utilizar um componente programável, tendo em vista que o número necessário de pinos ficaria aquém quando comparado ao tamanho da chave, do texto claro e do texto criptografado do algoritmo AES.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] International Telecommunication Union (em 25/10/2010) (http://www.itu.int/net/pressoffice/press_releases/2010/39.aspx)

[2] Edward David Moreno, Fábio Dacêncio Pereira e Rodolfo Barros Chiaramonte - Criptografia em Software e Hardware - Editora Novatec - 2005.

[3] David Kahn - The Codebreakers - The Story of Secret Writing - 1967, 1973

[4] William Stallings - "Criptografia e segurança de redes - Princípios e práticas" - 4a edição - Pearson Prentice Hall – 2007

[5] Federal Information Processing Standards Publication (FIPS PUB 46-3)
DATA ENCRYPTION STANDARD (DES)
October 25, 1999
http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[6] Criptographi Research, DES Key Search Machine.
http://www.cryptography.com/resources/whitepapers/DES.html. Acesso em 2010.

[7] História da Criptografia
http://www.gta.ufrj.br/grad/07_1/ass-dig/HistriadaCriptografia.html

[8] Breno Guimarães de Oliveira
Fundamentos da Criptologia Parte I – Introdução e Histórias
http://www.gris.dcc.ufrj.br/artigos/GRIS-2005-A-003.pdf
Fundamentos da Criptologia Parte II – Criptografia Simétrica
http://www.gris.dcc.ufrj.br/artigos/GRIS-2005-A-004.pdf
Fundamentos da Criptologia Parte III – Criptografia Simétrica (Continuação)
http://www.gris.dcc.ufrj.br/artigos/GRIS-2005-A-005.pdf

[9] Victoria Tkotz – “Criptografia – Segredos Embalados para Viagem” – Novatec – 2005.

[10] Dissertação de Mestrado
Keesje Duarte Pouw
Segurança na arquitetura TCP/IP: de firewalls a canais seguros

http://www.las.ic.unicamp.br/paulo/teses/19990208-MSc-Keesje.Duarte.Pouw-Seguranca.na.arquitetura.TCPIP-De.firewalls.a.canais.seguros.pdf

[11] Tipos de Criptografia - http://www.gta.ufrj.br/grad/07_1/ass-dig/TiposdeCriptografia.html

[12] Monografia de Aliane Veloso Mendes - 2007 - UNIMONTES - Estudo de criptografia com chave pública baseada em curvas Elípticas http://www.ccet.unimontes.br/arquivos/monografias/261.pdf

[13] The International PGP Home Page - http://www.pgpi.org/

[14] Monografia: Estudo de criptografia com chave pública baseada em curvas elípticas – Aliane Veloso Mendes – Montes Claros - 2007

[15] Criptografia
André Moreira (ASC)
Departamento de Engenharia Informática (DEI)
Instituto Superior de Engenharia do Porto (ISEP)
http://www.dei.isep.ipp.pt/~andre/documentos/criptografia.html

[16] National Institute of Standards and Technology. "Request for candidate algorithm nominations for the Advanced Encryption Standard". Federal Register, 12 de setembro de 1997 (NIST97).

[17] O algoritmo IDEA ilustrado - http://www.numaboa.com/criptografia/bloco/336-idea - acesso em junho de 2009 e outubro de 2010.

[18] Advanced Encryption Standard - ITL Security Bulletin - ITL February 1997 (itl99-02.txt). FEBRUARY 1999 - Enhancements To Data Encryption And Digital Signature Federal Standards (http://csrc.nist.gov/publications/nistbul/html-archive/feb-99.html) (obtido em dez/2010).

[19] Federal Information Processing Standards Publication 197 (FIPS PUB 197) Announcing the ADVANCED ENCRYPTION STANDARD (AES) November 26, 2001
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[20] Mesquita, L.; Botura Jr., G; Rocha, P. S., Curso Introdutório sobre Dispositivos Lógicos Programáveis, UNESP Campus Guaratinguetá – DEE, 2002, Guaratinguetá.

[21] Smith, M. J. S., Application Specific Integrated Circuits, 1997.

[22] TOCCI, R, Sistemas Digitais e Aplicações, 8ª Edição, 2002, São Paulo.

[23] Cyclone II PFGA Starter Development kit – User Guide – ALTERA – October 2006 (www.altera.com)

ANEXOS

Documento FIPS-197 (documento original em pdf).

Federal Information

Processing Standards Publication 197

November 26, 2001

Announcing the

ADVANCED ENCRYPTION STANDARD (AES)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

1. Name of Standard. Advanced Encryption Standard (AES) (FIPS PUB 197).

2. Category of Standard. Computer Security Standard, Cryptography.

3. Explanation. The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

4. Approving Authority. Secretary of Commerce.

5. Maintenance Agency. Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. Applicability. This standard may be used by Federal departments and agencies when an agency determines that sensitive (unclassified) information (as defined in P. L. 100-235) requires cryptographic protection.

Other FIPS-approved cryptographic algorithms may be used in addition to, or in lieu of, this standard. Federal agencies or departments that use cryptographic devices for protecting classified information can use those devices for protecting sensitive (unclassified) information in lieu of this standard.

In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

7. Specifications. Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES) (affixed).

8. Implementations. The algorithm specified in this standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc. The algorithm shall be used in conjunction with a FIPS approved or NIST recommended mode of operation. Object Identifiers (OIDs) and any associated parameters for AES used in these modes are available at the Computer Security Objects Register (CSOR), located at http://csrc.nist.gov/csor/ [2].

Implementations of the algorithm that are tested by an accredited laboratory and validated will be considered as complying with this standard. Since cryptographic security depends on many factors besides the correct implementation of an encryption algorithm, Federal Government employees, and others, should also refer to NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, for additional information and guidance (NIST SP 800-21 is available at http://csrc.nist.gov/publications/).

9. Implementation Schedule. This standard becomes effective on May 26, 2002.

10. Patents. Implementations of the algorithm specified in this standard may be covered by U.S. and foreign patents.

11. Export Control. Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Exports of cryptographic modules implementing this standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Export Administration of the U.S. Department of Commerce. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

12. Qualifications. NIST will continue to follow developments in the analysis of the AES algorithm. As with its other cryptographic algorithm standards, NIST will formally reevaluate this standard every five years.

Both this standard and possible threats reducing the security provided through the use of this standard will undergo review by NIST as appropriate, taking into account newly available analysis and technology. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate this standard and provide necessary revisions.

13. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The heads of such agencies may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with this standard would

a. adversely affect the accomplishment of the mission of an operator of Federal computer system or

b. cause a major adverse financial impact on the operator that is not offset by government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision that explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decision, Information Technology Laboratory, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

14. Where to obtain copies. This publication is available electronically by accessing http://csrc.nist.gov/publications/. A list of other available computer security publications, including ordering information, can be obtained from NIST Publications List 91, which is available at the same web site. Alternatively, copies of NIST computer security publications are available from: National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, VA 22161.

Federal Information

Processing Standards Publication 197

November 26, 2001

Specification for the

ADVANCED ENCRYPTION STANDARD (AES)

Table of Contents

1. INTRODUCTION 5

2. DEFINITIONS 5

2.1 GLOSSARY OF TERMS AND ACRONYMS 5

2.2 ALGORITHM PARAMETERS, SYMBOLS, AND FUNCTIONS 6

3. NOTATION AND CONVENTIONS 7

3.1 INPUTS AND OUTPUTS 7

3.2 BYTES 8

3.3 ARRAYS OF BYTES 8

3.4 THE STATE 9

3.5 THE STATE AS AN ARRAY OF COLUMNS 10

4. MATHEMATICAL PRELIMINARIES 10

4.1 ADDITION 10

4.2 MULTIPLICATION 10

4.2.1 Multiplication by x 11

4.3 POLYNOMIALS WITH COEFFICIENTS IN $GF(2^8)$ 12

5. ALGORITHM SPECIFICATION 13

5.1 CIPHER 14

5.1.1 `SubBytes()` Transformation 15

5.1.2 `ShiftRows()` Transformation 17

5.1.3 `MixColumns()` Transformation 17

5.1.4 `AddRoundKey()` Transformation 18

5.2 KEY EXPANSION 19

5.3 INVERSE CIPHER 20

5.3.1 `InvShiftRows()` Transformation *21*

5.3.2 `InvSubBytes()` Transformation *22*

5.3.3 `InvMixColumns()` Transformation *23*

5.3.4 Inverse of the `AddRoundKey()` Transformation *23*

5.3.5 Equivalent Inverse Cipher *23*

6. IMPLEMENTATION ISSUES 25

6.1 KEY LENGTH REQUIREMENTS 25

6.2 KEYING RESTRICTIONS 26

6.3 PARAMETERIZATION OF KEY LENGTH, BLOCK SIZE, AND ROUND NUMBER 26

6.4 IMPLEMENTATION SUGGESTIONS REGARDING VARIOUS PLATFORMS 26

APPENDIX A - KEY EXPANSION EXAMPLES 27

A.1 EXPANSION OF A 128-BIT CIPHER KEY 27

A.2 EXPANSION OF A 192-BIT CIPHER KEY 28

A.3 EXPANSION OF A 256-BIT CIPHER KEY 30

APPENDIX B – CIPHER EXAMPLE 33

APPENDIX C – EXAMPLE VECTORS 35

C.1 AES-128 (*Nk*=4, *Nr*=10) 35

C.2 AES-192 (*Nk*=6, *Nr*=12) 38

C.3 AES-256 (*Nk*=8, *Nr*=14) 42

APPENDIX D - REFERENCES 47

Table of Figures

Figure 1. Hexadecimal representation of bit patterns. 8
Figure 2. Indices for Bytes and Bits. 9
Figure 3. State array input and output. 9
Figure 4. Key-Block-Round Combinations. 14
Figure 5. Pseudo Code for the Cipher. 15
Figure 6. `SubBytes()` applies the S-box to each byte of the State. 16
Figure 7. S-box: substitution values for the byte `xy` (in hexadecimal format). 16
Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State 17
Figure 9. `MixColumns()` operates on the State column-by-column. 18
Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule. 19
Figure 11. Pseudo Code for Key Expansion. 20
Figure 12. Pseudo Code for the Inverse Cipher. 21
Figure 13. `InvShiftRows()`cyclically shifts the last three rows in the State. 22
Figure 14. Inverse S-box: substitution values for the byte `xy` (in hexadecimal format). 22
Figure 15. Pseudo Code for the Equivalent Inverse Cipher. 25

1. Introduction

This standard specifies the **Rijndael** algorithm ([3] and [4]), a symmetric block cipher that can process **data blocks** of **128 bits**, using cipher **keys** with lengths of **128**, **192**, and **256 bits**. Rijndael was designed to handle additional block sizes and key lengths, however they are not adopted in this standard.

Throughout the remainder of this standard, the algorithm specified herein will be referred to as "the AES algorithm." The algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".

This specification includes the following sections:

2. Definitions of terms, acronyms, and algorithm parameters, symbols, and functions;
3. Notation and conventions used in the algorithm specification, including the ordering and numbering of bits, bytes, and words;
4. Mathematical properties that are useful in understanding the algorithm;
5. Algorithm specification, covering the key expansion, encryption, and decryption routines;
6. Implementation issues, such as key length support, keying restrictions, and additional block/key/round sizes.

The standard concludes with several appendices that include step-by-step examples for Key Expansion and the Cipher, example vectors for the Cipher and Inverse Cipher, and a list of references.

2. Definitions

2.1 Glossary of Terms and Acronyms

The following definitions are used throughout this standard:

AES	Advanced Encryption Standard
Affine Transformation	A transformation consisting of multiplication by a matrix followed by the addition of a vector.
Array	An enumerated collection of identical entities (e.g., an array of bytes).
Bit	A binary digit having a value of 0 or 1.
Block	Sequence of binary bits that comprise the input, output, State, and Round Key. The length of a sequence is the number of bits it contains. Blocks are also interpreted as arrays of bytes.
Byte	A group of eight bits that is treated either as a single entity or as an array of 8 individual bits.

Cipher	Series of transformations that converts plaintext to ciphertext using the Cipher Key.
Cipher Key	Secret, cryptographic key that is used by the Key Expansion routine to generate a set of Round Keys; can be pictured as a rectangular array of bytes, having four rows and ***Nk*** columns.
Ciphertext	Data output from the Cipher or input to the Inverse Cipher.
Inverse Cipher	Series of transformations that converts ciphertext to plaintext using the Cipher Key.
Key Expansion	Routine used to generate a series of Round Keys from the Cipher Key.
Plaintext	Data input to the Cipher or output from the Inverse Cipher.
Rijndael	Cryptographic algorithm specified in this Advanced Encryption Standard (AES).
Round Key	Round keys are values derived from the Cipher Key using the Key Expansion routine; they are applied to the State in the Cipher and Inverse Cipher.
State	Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and ***Nb*** columns.
S-box	Non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform a one-for-one substitution of a byte value.
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

2.2 Algorithm Parameters, Symbols, and Functions

The following algorithm parameters, symbols, and functions are used throughout this standard:

`AddRoundKey()`	Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for *Nb* = 4, the Round Key length equals 128 bits/16 bytes).
`InvMixColumns()`	Transformation in the Inverse Cipher that is the inverse of **`MixColumns()`**.
`InvShiftRows()`	Transformation in the Inverse Cipher that is the inverse of **`ShiftRows()`**.
`InvSubBytes()`	Transformation in the Inverse Cipher that is the inverse of **`SubBytes()`**.
K	Cipher Key.

`MixColumns()`	Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.
Nb	Number of columns (32-bit words) comprising the State. For this standard, ***Nb*** = 4. (Also see Sec. 6.3.)
Nk	Number of 32-bit words comprising the Cipher Key. For this standard, ***Nk*** = 4, 6, or 8. (Also see Sec. 6.3.)
Nr	Number of rounds, which is a function of ***Nk*** and ***Nb*** (which is fixed). For this standard, ***Nr*** = 10, 12, or 14. (Also see Sec. 6.3.)
Rcon[]	The round constant word array.
`RotWord()`	Function used in the Key Expansion routine that takes a four-byte word and performs a cyclic permutation.
`ShiftRows()`	Transformation in the Cipher that processes the State by cyclically shifting the last three rows of the State by different offsets.
`SubBytes()`	Transformation in the Cipher that processes the State using a non-linear byte substitution table (S-box) that operates on each of the State bytes independently.
`SubWord()`	Function used in the Key Expansion routine that takes a four-byte input word and applies an S-box to each of the four bytes to produce an output word.
XOR	Exclusive-OR operation.
$\oplus$	Exclusive-OR operation.
$\otimes$	Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$.
$\bullet$	Finite field multiplication.

3. Notation and Conventions

3.1 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 $0 \le i < 128$, $0 \le i < 192$ or $0 \le i < 256$ depending on the block length and key length (specified above).

3.2 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. 3.1 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. 3.3). 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, $0 \le n < 16$; Block length = 128 bits, $0 \le n < 16$;

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

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

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 $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i \; . \qquad (3.1)$$

For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character as in Fig. 1.

Bit Pattern	Character
0000	0
0001	1
0010	2
0011	3

Bit Pattern	Character
0100	4
0101	5
0110	6
0111	7

Bit Pattern	Character
1000	8
1001	9
1010	a
1011	b

Bit Pattern	Character
1100	c
1101	d
1110	e
1111	f

Figure 1. Hexadecimal representation of bit patterns.

Hence the element {01100011} can be represented as {63}, where the character denoting the four-bit group containing the higher numbered bits is again to the left.

Some finite field operations involve one additional bit (b_8) 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}.

3.3 Arrays of Bytes

Arrays of bytes will be represented in the following form:

$$a_0 a_1 a_2 ... a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

$$input_0 \; input_1 \; input_2 \; ... \; input_{126} \; input_{127}$$

as follows:

$$a_0 = \{input_0, input_1, \ldots, input_7\};$$

$$a_1 = \{input_8, input_9, \ldots, input_{15}\};$$

$$\vdots$$

$$a_{15} = \{input_{120}, input_{121}, \ldots, input_{127}\}.$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, \ldots, input_{8n+7}\}. \quad (3.2)$$

Taking Sections 3.2 and 3.3 together, Fig. 2 shows how bits within each byte are numbered.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…
Byte number	0								1								2								…
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	…

Figure 2. Indices for Bytes and Bits.

3.4 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 $0 \le r < 4$ and its column number *c* in the range $0 \le c < \boldsymbol{Nb}$. 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., $0 \le c < 4$ (also see Sec. 6.3).

At the start of the Cipher and Inverse Cipher described in Sec. 5, the input – the array of bytes $in_0, in_1, \ldots in_{15}$ – is copied into the State array as illustrated in Fig. 3. 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 $out_0, out_1, \ldots out_{15}$.

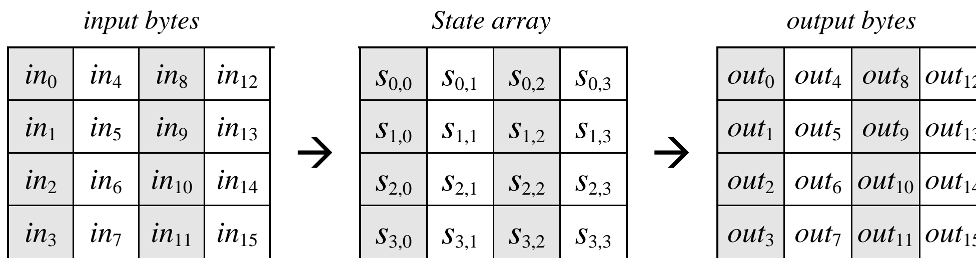

Figure 3. State array input and output.

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

$$s[r, c] = in[r + 4c] \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb, \quad (3.3)$$

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

$$out[r + 4c] = s[r, c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb. \qquad (3.4)$$

3.5 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), $w_0...w_3$, where the column number c provides an index into this array. Hence, for the example in Fig. 3, the State can be considered as an array of four words, as follows:

$$w_0 = s_{0,0}\ s_{1,0}\ s_{2,0}\ s_{3,0} \qquad w_2 = s_{0,2}\ s_{1,2}\ s_{2,2}\ s_{3,2}$$
$$w_1 = s_{0,1}\ s_{1,1}\ s_{2,1}\ s_{3,1} \qquad w_3 = s_{0,3}\ s_{1,3}\ s_{2,3}\ s_{3,3}\ . \qquad (3.5)$$

4. Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in Sec. 3.2. 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 needed for Sec. 5.

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 $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, the sum is $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6$, $...c_0 = a_0 \oplus b_0$).

For example, the following expressions are equivalent to one another:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 \qquad \text{(polynomial notation);}$$

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

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

4.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) (denoted by $\bullet$) 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 <u>irreducible</u> polynomial is**

$$m(x) = x^8 + x^4 + x^3 + x + 1, \qquad (4.1)$$

or {01}{1b} in hexadecimal notation.

For example, {57} • {83} = {c1}, because

$$(x^6+x^4+x^2+x+1)\ (x^7+x+1) = x^{13}+x^{11}+x^9+x^8+x^7+$$
$$x^7+x^5+x^3+x^2+x+$$
$$x^6+x^4+x^2+x+1$$
$$= x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1$$

and

$$x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1 \ \texttt{modulo}\ (x^8+x^4+x^3+x+1)$$
$$= x^7+x^6+1.$$

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.

The multiplication defined above is associative, and the element {01} is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than 8, the multiplicative inverse of $b(x)$, denoted $b^{-1}(x)$, can be found as follows: the extended Euclidean algorithm [7] is used to compute polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x)+m(x)c(x)=1. \tag{4.2}$$

Hence, $a(x)\bullet b(x) \bmod m(x)=1$, which means

$$b^{-1}(x)=a(x) \bmod m(x). \tag{4.3}$$

Moreover, for any $a(x)$, $b(x)$ and $c(x)$ in the field, it holds that

$$a(x)\bullet(b(x)+c(x))=a(x)\bullet b(x)+a(x)\bullet c(x).$$

It follows that the set of 256 possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field GF(2^8).

4.2.1 Multiplication by *x*

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

$$b_7x^8+b_6x^7+b_5x^6+b_4x^5+b_3x^4+b_2x^3+b_1x^2+b_0x. \tag{4.4}$$

The result $x\bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation (4.1). 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

$$\{57\} \bullet \{02\} = \texttt{xtime}(\{57\}) = \{ae\}$$

$$\{57\} \bullet \{04\} = \texttt{xtime}(\{ae\}) = \{47\}$$

$$\{57\} \bullet \{08\} = \texttt{xtime}(\{47\}) = \{8e\}$$

$$\{57\} \bullet \{10\} = \texttt{xtime}(\{8e\}) = \{07\},$$

thus,

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

4.3 Polynomials with Coefficients in GF(2^8)

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

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \qquad (4.5)$$

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 differently 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

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \qquad (4.6)$$

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.

Thus, using the equations of (4.5) and (4.6),

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0) \qquad (4.7)$$

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

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \qquad (4.8)$$

where

$$\begin{aligned} c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\ c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\ c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_6 &= a_3 \bullet b_3 \end{aligned} \qquad (4.9)$$

$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 .$$

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

$$x^i \bmod(x^4 + 1) = x^{i \bmod 4} . \tag{4.10}$$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \otimes b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \tag{4.11}$$

with

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3) \end{aligned} \tag{4.12}$$

When $a(x)$ is a fixed polynomial, the operation defined in equation (4.11) can be written in matrix form as:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{4.13}$$

Because $x^4 + 1$ is not an irreducible polynomial over GF(2^8), multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that *does* have an inverse (see Sec. 5.1.3 and Sec. 5.3.3):

$$a(x) = \{\texttt{03}\}x^3 + \{\texttt{01}\}x^2 + \{\texttt{01}\}x + \{\texttt{02}\} \tag{4.14}$$

$$a^{-1}(x) = \{\texttt{0b}\}x^3 + \{\texttt{0d}\}x^2 + \{\texttt{09}\}x + \{\texttt{0e}\}. \tag{4.15}$$

Another polynomial used in the AES algorithm (see the **`RotWord()`** function in Sec. 5.2) has $a_0 = a_1 = a_2 = \{\texttt{00}\}$ and $a_3 = \{\texttt{01}\}$, which is the polynomial x^3. Inspection of equation (4.13) above will show that its effect is to form the output word by rotating bytes in the input word. This means that $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.

5. 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 $\boldsymbol{Nb} = 4$, 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 Fig. 4. For implementation issues relating to the key length, block size and number of rounds, see Sec. 6.3.

	Key Length (*Nk words*)	Block Size (*Nb words*)	Number of Rounds (*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 4. Key-Block-Round Combinations.

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations: 1) byte substitution using a substitution table (S-box), 2) shifting rows of the State array by different offsets, 3) mixing the data within each column of the State array, and 4) adding a Round Key to the State. These transformations (and their inverses) are described in Sec. 5.1.1-5.1.4 and 5.3.1-5.3.4.

The Cipher and Inverse Cipher are described in Sec. 5.1 and Sec. 5.3, respectively, while the Key Schedule is described in Sec. 5.2.

5.1 Cipher

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec. 3.4. 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), 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. 3.4.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine described in Sec. 5.2.

The Cipher is described in the pseudo code in Fig. 5. The individual transformations - **`SubBytes()`**, **`ShiftRows()`**, **`MixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections. In Fig. 5, the array **`w[]`** contains the key schedule, which is described in Sec. 5.2.

As shown in Fig. 5, all ***Nr*** rounds are identical with the exception of the final round, which does not include the **`MixColumns()`** transformation.

Appendix B presents an example of the Cipher, showing values for the State array at the beginning of each round and after the application of each of the four transformations described in the following sections.

```
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])            // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                        // See Sec. 5.1.1
      ShiftRows(state)                       // See Sec. 5.1.2
      MixColumns(state)                      // See Sec. 5.1.3
      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 5. Pseudo Code for the Cipher.[1]

5.1.1 `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 (Fig. 7), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field $GF(2^8)$, described in Sec. 4.2; the element {00} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):

$$b_i^{'} = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i \quad (5.1)$$

   for $0 \le i < 8$, where b_i is the ith bit of the byte, and c_i is the ith 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, the affine transformation element of the S-box can be expressed as:

[1] 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 to address the Round Key.

$$
\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} +
\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{5.2}
$$

Figure 6 illustrates the effect of the **`SubBytes()`** transformation on the State.

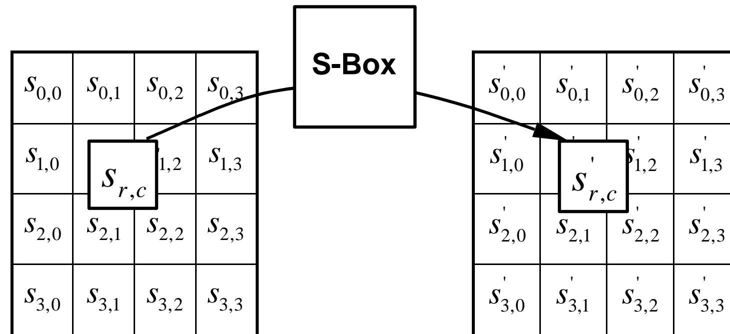

Figure 6. `SubBytes()` applies the S-box to each byte of the State.

The S-box used in the **`SubBytes()`** transformation is presented in hexadecimal form in Fig. 7. For example, if $s_{1,1} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Fig. 7. This would result in $s'_{1,1}$ having a value of {ed}.

		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 7. S-box: substitution values for the byte `xy` (in hexadecimal format).

5.1.2 `ShiftRows()` Transformation

In 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.

Specifically, the **`ShiftRows()`** transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \bmod Nb} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb}, \tag{5.3}$$

where the shift value $shift(r,Nb)$ depends on the row number, r, as follows (recall that $\boldsymbol{Nb} = 4$):

$$shift(1,4) = 1\,; \quad shift(2,4) = 2\,; \quad shift(3,4) = 3\,. \tag{5.4}$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of c in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of c in a given row).

Figure 8 illustrates the **`ShiftRows()`** transformation.

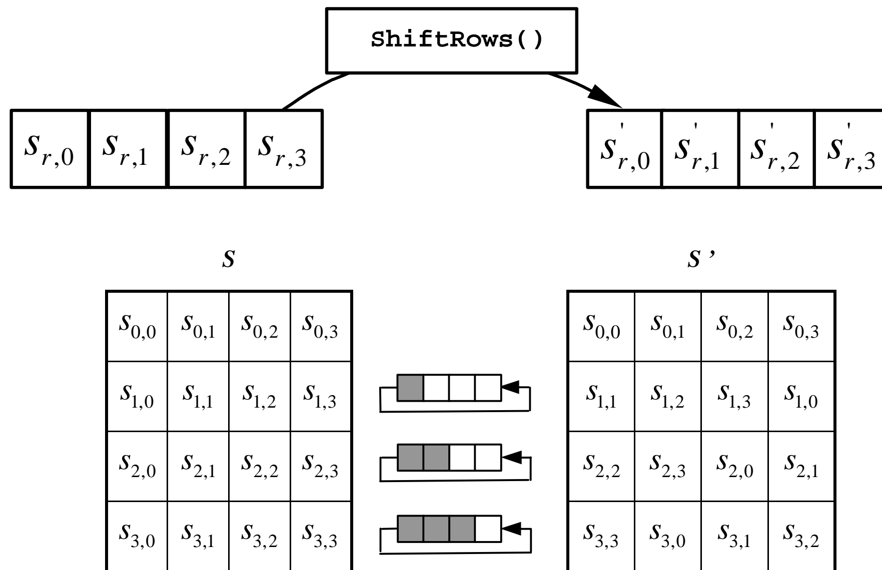

Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State.

5.1.3 `MixColumns()` Transformation

The **`MixColumns()`** transformation operates on the State column-by-column, treating each column as a four-term 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 $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}\,. \tag{5.5}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}. \tag{5.6}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Figure 9 illustrates the **`MixColumns()`** transformation.

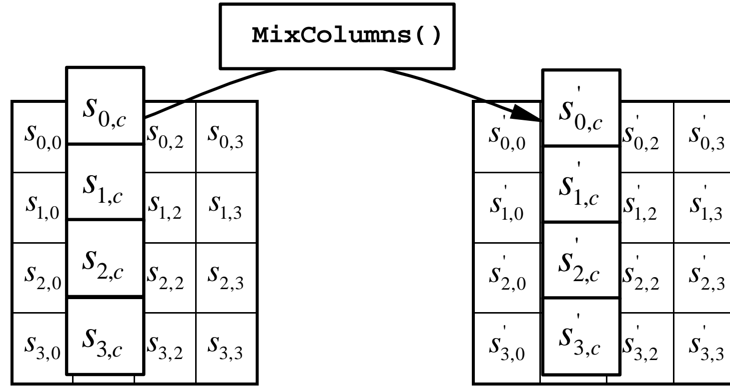

Figure 9. `MixColumns()` operates on the State column-by-column.

5.1.4 `AddRoundKey()` Transformation

In the **`AddRoundKey()`** transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of ***Nb*** words from the key schedule (described in Sec. 5.2). Those ***Nb*** words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad \text{for } 0 \le c < \boldsymbol{Nb}, \tag{5.7}$$

where $[w_i]$ are the key schedule words described in Sec. 5.2, and *round* is a value in the range $0 \le round \le \boldsymbol{Nr}$. In the Cipher, the initial Round Key addition occurs when *round* = 0, prior to the first application of the round function (see Fig. 5). The application of the **`AddRoundKey()`** transformation to the ***Nr*** rounds of the Cipher occurs when $1 \le round \le \boldsymbol{Nr}$.

The action of this transformation is illustrated in Fig. 10, where $l = round * \boldsymbol{Nb}$. The byte address within words of the key schedule was described in Sec. 3.1.

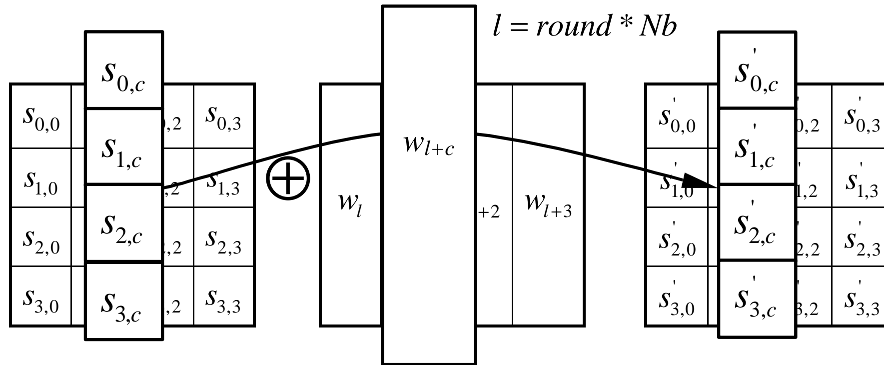

Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule.

5.2 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 $0 \le i < Nb(Nr + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Fig. 11.

`SubWord()` is a function that takes a four-byte input word and applies the S-box (Sec. 5.1.1, Fig. 7) 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), as discussed in Sec. 4.2 (note that i starts at 1, not 0).

From Fig. 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.

```
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

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.  Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 11. Pseudo Code for Key Expansion.[2]

Appendix A presents examples of the Key Expansion.

5.3 Inverse Cipher

The Cipher transformations in Sec. 5.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - **`InvShiftRows()`**, **`InvSubBytes()`**,**`InvMixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections.

The Inverse Cipher is described in the pseudo code in Fig. 12. In Fig. 12, the array **`w[]`** contains the key schedule, which was described previously in Sec. 5.2.

[2] The functions **`SubWord()`** and **`RotWord()`** return a result that is a transformation of the function input, whereas the transformations in the Cipher and Inverse Cipher (e.g., **`ShiftRows()`**, **`SubBytes()`**, etc.) transform the State array that is addressed by the '`state`' pointer.

```
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]) // See Sec. 5.1.4

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)                    // See Sec. 5.3.1
      InvSubBytes(state)                     // See Sec. 5.3.2
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)                   // See Sec. 5.3.3
   end for

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

   out = state
end
```

Figure 12. Pseudo Code for the Inverse Cipher.[3]

5.3.1 `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 $\boldsymbol{Nb} - shift(r, Nb)$ bytes, where the shift value *shift(r,Nb)* depends on the row number, and is given in equation (5.4) (see Sec. 5.1.2).

Specifically, the **`InvShiftRows()`** transformation proceeds as follows:

$$s'_{r,(c+shift(r,Nb)) \bmod Nb} = s_{r,c} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb} \qquad (5.8)$$

Figure 13 illustrates the **`InvShiftRows()`** transformation.

[3] The various transformations (e.g., **`InvSubBytes()`**, **`InvShiftRows()`**, etc.) act upon the State array that is addressed by the '`state`' pointer. **`AddRoundKey()`** uses an additional pointer to address the Round Key.

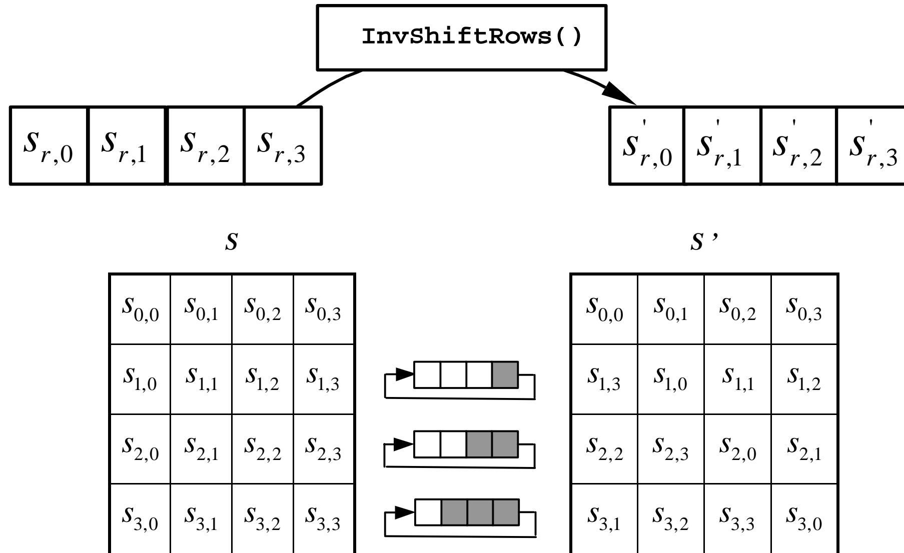

Figure 13. `InvShiftRows()` cyclically shifts the last three rows in the State.

5.3.2 `InvSubBytes()` Transformation

`InvSubBytes()` is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (5.1) followed by taking the multiplicative inverse in GF(2^8).

The inverse S-box used in the **`InvSubBytes()`** transformation is presented in Fig. 14:

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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
	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 14. Inverse S-box: substitution values for the byte `xy` (in hexadecimal format).

5.3.3 `InvMixColumns()` Transformation

`InvMixColumns()` is the inverse of the **`MixColumns()`** transformation. **`InvMixColumns()`** operates on the State column-by-column, treating each column as a four-term 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

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \tag{5.9}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a^{-1}(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}. \tag{5.10}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

5.3.4 Inverse of the `AddRoundKey()` Transformation

`AddRoundKey()`, which was described in Sec. 5.1.4, is its own inverse, since it only involves an application of the XOR operation.

5.3.5 Equivalent Inverse Cipher

In the straightforward Inverse Cipher presented in Sec. 5.3 and Fig. 12, the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

The two properties that allow for this Equivalent Inverse Cipher are as follows:

1. The **`SubBytes()`** and **`ShiftRows()`** transformations commute; that is, a **`SubBytes()`** transformation immediately followed by a **`ShiftRows()`** transformation is equivalent to a **`ShiftRows()`** transformation immediately followed buy a **`SubBytes()`** transformation. The same is true for their inverses, **`InvSubBytes()`** and **`InvShiftRows.`**

2. The column mixing operations - **MixColumns()** and **InvMixColumns()** - are linear with respect to the column input, which means

```
InvMixColumns(state XOR Round Key) =
               InvMixColumns(state) XOR InvMixColumns(Round Key).
```

These properties allow the order of **InvSubBytes()** and **InvShiftRows()** transformations to be reversed. The order of the **AddRoundKey()** and **InvMixColumns()** transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the **InvMixColumns()** transformation.

The equivalent inverse cipher is defined by reversing the order of the **InvSubBytes()** and **InvShiftRows()** transformations shown in Fig. 12, and by reversing the order of the **AddRoundKey()** and **InvMixColumns()** transformations used in the "round loop" after first modifying the decryption key schedule for *round* = 1 to *Nr*-1 using the **InvMixColumns()** transformation. The first and last *Nb* words of the decryption key schedule shall *not* be modified in this manner.

Given these changes, the resulting Equivalent Inverse Cipher offers a more efficient structure than the Inverse Cipher described in Sec. 5.3 and Fig. 12. Pseudo code for the Equivalent Inverse Cipher appears in Fig. 15. (The word array **dw[]** contains the modified decryption key schedule. The modification to the Key Expansion routine is also provided in Fig. 15.)

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

   state = in

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

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

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

   out = state
end
```

For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine (Sec. 5.2):

```
   for i = 0 step 1 to (Nr+1)*Nb-1
      dw[i] = w[i]
   end for

   for round = 1 step 1 to Nr-1
      InvMixColumns(dw[round*Nb, (round+1)*Nb-1])     // note change of type
   end for
```

Note that, since InvMixColumns operates on a two-dimensional array of bytes while the Round Keys are held in an array of words, the call to InvMixColumns in this code sequence involves a change of type (i.e. the input to InvMixColumns() is normally the State array, which is considered to be a two-dimensional array of bytes, whereas the input here is a Round Key computed as a one-dimensional array of words).

Figure 15. Pseudo Code for the Equivalent Inverse Cipher.

6. Implementation Issues

6.1 Key Length Requirements

An implementation of the AES algorithm shall support *at least one* of the three key lengths specified in Sec. 5: 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.

6.2 Keying Restrictions

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

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

This standard explicitly defines the allowed values for the key length (***Nk***), block size (***Nb***), and number of rounds (***Nr***) – see Fig. 4. 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.

6.4 Implementation Suggestions Regarding Various Platforms

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.

Reference [3] and other papers located at Ref. [1] include suggestions on how to efficiently implement the AES algorithm on a variety of platforms.

Appendix A - Key Expansion Examples

This appendix shows the development of the key schedule for various key sizes. Note that multi-byte values are presented using the notation described in Sec. 3. The intermediate values produced during the development of the key schedule (see Sec. 5.2) are given in the following table (all values are in hexadecimal format, with the exception of the index column (i)).

A.1 Expansion of a 128-bit Cipher Key

This section contains the key expansion of the following cipher key:

`Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c`

for $\boldsymbol{Nk} = 4$, which results in

$w_0 =$ **2b7e1516** $w_1 =$ **28aed2a6** $w_2 =$ **abf71588** $w_3 =$ **09cf4f3c**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

A.2 Expansion of a 192-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =       8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
                   80 90 79 e5 62 f8 ea d2 52 2c 6b 7b
```

for $Nk = 6$, which results in

w_0 = **8e73b0f7** w_1 = **da0e6452** w_2 = **c810f32b** w_3 = **809079e5**

w_4 = **62f8ead2** w_5 = **522c6b7b**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e

9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb095386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
19	485af057					69b54118	21efb14f
20	21efb14f					85a74796	a448f6d9
21	a448f6d9					e92538fd	4d6dce24
22	4d6dce24					e75fad44	aa326360
23	aa326360					bb095386	113b30e6
24	113b30e6	3b30e611	e2048e82	08000000	ea048e82	485af057	a25e7ed5
25	a25e7ed5					21efb14f	83b1cf9a
26	83b1cf9a					a448f6d9	27f93943
27	27f93943					4d6dce24	6a94f767
28	6a94f767					aa326360	c0a69407
29	c0a69407					113b30e6	d19da4e1
30	d19da4e1	9da4e1d1	5e49f83e	10000000	4e49f83e	a25e7ed5	ec1786eb
31	ec1786eb					83b1cf9a	6fa64971
32	6fa64971					27f93943	485f7032
33	485f7032					6a94f767	22cb8755
34	22cb8755					c0a69407	e26d1352
35	e26d1352					d19da4e1	33f0b7b3
36	33f0b7b3	f0b7b333	8ca96dc3	20000000	aca96dc3	ec1786eb	40beeb28
37	40beeb28					6fa64971	2f18a259
38	2f18a259					485f7032	6747d26b
39	6747d26b					22cb8755	458c553e
40	458c553e					e26d1352	a7e1466c
41	a7e1466c					33f0b7b3	9411f1df
42	9411f1df	11f1df94	82a19e22	40000000	c2a19e22	40beeb28	821f750a
43	821f750a					2f18a259	ad07d753

44	ad07d753					6747d26b	ca400538
45	ca400538					458c553e	8fcc5006
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

A.3 Expansion of a 256-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
                  1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for ***Nk*** = 8, which results in

w_0 = 603deb10 w_1 = 15ca71be w_2 = 2b73aef0 w_3 = 857d7781

w_4 = 1f352c07 w_5 = 3b6108d7 w_6 = 2d9810a3 w_7 = 0914dff4

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229

23	98312229					b75d5b9a	2f6c79b3
24	2f6c79b3	6c79b32f	50b66d15	04000000	54b66d15	d59aecb8	812c81ad
25	812c81ad					5bf3c917	dadf48ba
26	dadf48ba					fee94248	24360af2
27	24360af2					de8ebe96	fab8b464
28	fab8b464		2d6c8d43			b5a9328a	98c5bfc9
29	98c5bfc9					2678a647	bebd198e
30	bebd198e					98312229	268c3ba7
31	268c3ba7					2f6c79b3	09e04214
32	09e04214	e0421409	e12cfa01	08000000	e92cfa01	812c81ad	68007bac
33	68007bac					dadf48ba	b2df3316
34	b2df3316					24360af2	96e939e4
35	96e939e4					fab8b464	6c518d80
36	6c518d80		50d15dcd			98c5bfc9	c814e204
37	c814e204					bebd198e	76a9fb8a
38	76a9fb8a					268c3ba7	5025c02d
39	5025c02d					09e04214	59c58239
40	59c58239	c5823959	a61312cb	10000000	b61312cb	68007bac	de136967
41	de136967					b2df3316	6ccc5a71
42	6ccc5a71					96e939e4	fa256395
43	fa256395					6c518d80	9674ee15
44	9674ee15		90922859			c814e204	5886ca5d
45	5886ca5d					76a9fb8a	2e2f31d7
46	2e2f31d7					5025c02d	7e0af1fa
47	7e0af1fa					59c58239	27cf73c3
48	27cf73c3	cf73c327	8a8f2ecc	20000000	aa8f2ecc	de136967	749c47ab
49	749c47ab					6ccc5a71	18501dda
50	18501dda					fa256395	e2757e4f
51	e2757e4f					9674ee15	7401905a
52	7401905a		927c60be			5886ca5d	cafaaae3
53	cafaaae3					2e2f31d7	e4d59b34
54	e4d59b34					7e0af1fa	9adf6ace
55	9adf6ace					27cf73c3	bd10190d
56	bd10190d	10190dbd	cad4d77a	40000000	8ad4d77a	749c47ab	fe4890d1
57	fe4890d1					18501dda	e6188d0b

58	**e6188d0b**					**e2757e4f**	**046df344**
59	**046df344**					**7401905a**	**706c631e**

Appendix B – Cipher Example

The following diagram shows the values in the State array as the Cipher progresses for a block length and a Cipher Key length of 16 bytes each (i.e., $Nb = 4$ and $Nk = 4$).

```
Input =      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

The Round Key values are taken from the Key Expansion example in Appendix A.

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns		Round Key Value	
	32 88 31 e0					2b 28 ab 09	
	43 5a 31 37					7e ae f7 cf	
input	f6 30 98 07				⊕	15 d2 15 4f	=
	a8 8d a2 34					16 a6 88 3c	
	19 a0 9a e9	d4 e0 b8 1e	d4 e0 b8 1e	04 e0 48 28		a0 88 23 2a	
	3d f4 c6 f8	27 bf b4 41	bf b4 41 27	66 cb f8 06		fa 54 a3 6c	
1	e3 e2 8d 48	11 98 5d 52	5d 52 11 98	81 19 d3 26	⊕	fe 2c 39 76	=
	be 2b 2a 08	ae f1 e5 30	30 ae f1 e5	e5 9a 7a 4c		17 b1 39 05	
	a4 68 6b 02	49 45 7f 77	49 45 7f 77	58 1b db 1b		f2 7a 59 73	
	9c 9f 5b 6a	de db 39 02	db 39 02 de	4d 4b e7 6b		c2 96 35 59	
2	7f 35 ea 50	d2 96 87 53	87 53 d2 96	ca 5a ca b0	⊕	95 b9 80 f6	=
	f2 2b 43 49	89 f1 1a 3b	3b 89 f1 1a	f1 ac a8 e5		f2 43 7a 7f	
	aa 61 82 68	ac ef 13 45	ac ef 13 45	75 20 53 bb		3d 47 1e 6d	
	8f dd d2 32	73 c1 b5 23	c1 b5 23 73	ec 0b c0 25		80 16 23 7a	
3	5f e3 4a 46	cf 11 d6 5a	d6 5a cf 11	09 63 cf d0	⊕	47 fe 7e 88	=
	03 ef d2 9a	7b df b5 b8	b8 7b df b5	93 33 7c dc		7d 3e 44 3b	
	48 67 4d d6	52 85 e3 f6	52 85 e3 f6	0f 60 6f 5e		ef a8 b6 db	
	6c 1d e3 5f	50 a4 11 cf	a4 11 cf 50	d6 31 c0 b3		44 52 71 0b	
4	4e 9d b1 58	2f 5e c8 6a	c8 6a 2f 5e	da 38 10 13	⊕	a5 5b 25 ad	=
	ee 0d 38 e7	28 d7 07 94	94 28 d7 07	a9 bf 6b 01		41 7f 3b 00	
	e0 c8 d9 85	e1 e8 35 97	e1 e8 35 97	25 bd b6 4c		d4 7c ca 11	
	92 63 b1 b8	4f fb c8 6c	fb c8 6c 4f	d1 11 3a 4c		d1 83 f2 f9	
5	7f 63 35 be	d2 fb 96 ae	96 ae d2 fb	a9 d1 33 c0	⊕	c6 9d b8 15	=
	e8 c0 50 01	9b ba 53 7c	7c 9b ba 53	ad 68 8e b0		f8 87 bc bc	

	f1	c1	7c	5d	a1	78	10	4c	a1	78	10	4c	4b	2c	33	37		6d	11	db	ca	
6	00	92	c8	b5	63	4f	e8	d5	4f	e8	d5	63	86	4a	9d	d2	⊕	88	0b	f9	00	=
	6f	4c	8b	d5	a8	29	3d	03	3d	03	a8	29	8d	89	f4	18		a3	3e	86	93	
	55	ef	32	0c	fc	df	23	fe	fe	fc	df	23	6d	80	e8	d8		7a	fd	41	fd	
	26	3d	e8	fd	f7	27	9b	54	f7	27	9b	54	14	46	27	34		4e	5f	84	4e	
7	0e	41	64	d2	ab	83	43	b5	83	43	b5	ab	15	16	46	2a	⊕	54	5f	a6	a6	=
	2e	b7	72	8b	31	a9	40	3d	40	3d	31	a9	b5	15	56	d8		f7	c9	4f	dc	
	17	7d	a9	25	f0	ff	d3	3f	3f	f0	ff	d3	bf	ec	d7	43		0e	f3	b2	4f	
	5a	19	a3	7a	be	d4	0a	da	be	d4	0a	da	00	b1	54	fa		ea	b5	31	7f	
8	41	49	e0	8c	83	3b	e1	64	3b	e1	64	83	51	c8	76	1b	⊕	d2	8d	2b	8d	=
	42	dc	19	04	2c	86	d4	f2	d4	f2	2c	86	2f	89	6d	99		73	ba	f5	29	
	b1	1f	65	0c	c8	c0	4d	fe	fe	c8	c0	4d	d1	ff	cd	ea		21	d2	60	2f	
	ea	04	65	85	87	f2	4d	97	87	f2	4d	97	47	40	a3	4c		ac	19	28	57	
9	83	45	5d	96	ec	6e	4c	90	6e	4c	90	ec	37	d4	70	9f	⊕	77	fa	d1	5c	=
	5c	33	98	b0	4a	c3	46	e7	46	e7	4a	c3	94	e4	3a	42		66	dc	29	00	
	f0	2d	ad	c5	8c	d8	95	a6	a6	8c	d8	95	ed	a5	a6	bc		f3	21	41	6e	
	eb	59	8b	1b	e9	cb	3d	af	e9	cb	3d	af						d0	c9	e1	b6	
10	40	2e	a1	c3	09	31	32	2e	31	32	2e	09					⊕	14	ee	3f	63	=
	f2	38	13	42	89	07	7d	2c	7d	2c	89	07						f9	25	0c	0c	
	1e	84	e7	d2	72	5f	94	b5	b5	72	5f	94						a8	89	c8	a6	
	39	02	dc	19																		
output	25	dc	11	6a																		
	84	09	85	0b																		
	1d	fb	97	32																		

Appendix C – Example Vectors

This appendix contains example vectors, including intermediate values – for all three AES key lengths (***Nk*** = 4, 6, and 8), for the Cipher, Inverse Cipher, and Equivalent Inverse Cipher that are described in Sec. 5.1, 5.3, and 5.3.5, respectively. Additional examples may be found at [1] and [5].

All vectors are in hexadecimal notation, with each pair of characters giving a byte value in which the left character of each pair provides the bit pattern for the 4 bit group containing the higher numbered bits using the notation explained in Sec. 3.2, while the right character provides the bit pattern for the lower-numbered bits. The array index for all bytes (groups of two hexadecimal digits) within these test vectors starts at zero and increases from left to right.

```
Legend for CIPHER (ENCRYPT) (round number r = 0 to 10, 12 or 14):

   input:   cipher input
   start:   state at start of round[r]
   s_box:   state after SubBytes()
   s_row:   state after ShiftRows()
   m_col:   state after MixColumns()
   k_sch:   key schedule value for round[r]
   output:  cipher output


Legend for INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12 or 14):
   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   ik_sch:  key schedule value for round[r]
   ik_add:  state after AddRoundKey()
   ioutput: inverse cipher output


Legend for EQUIVALENT INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12
   or 14):

   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   im_col:  state after InvMixColumns()
   ik_sch:  key schedule value for round[r]
   ioutput: inverse cipher output
```

C.1 AES-128 (*Nk*=4, *Nr*=10)

```
PLAINTEXT:          00112233445566778899aabbccddeeff
KEY:                000102030405060708090a0b0c0d0e0f

CIPHER (ENCRYPT):
```

```
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 2].start    89d810e8855ace682d1843d8cb128fe4
round[ 2].s_box    a761ca9b97be8b45d8ad1a611fc97369
round[ 2].s_row    a7be1a6997ad739bd8c9ca451f618b61
round[ 2].m_col    ff87968431d86a51645151fa773ad009
round[ 2].k_sch    b692cf0b643dbdf1be9bc5006830b3fe
round[ 3].start    4915598f55e5d7a0daca94fa1f0a63f7
round[ 3].s_box    3b59cb73fcd90ee05774222dc067fb68
round[ 3].s_row    3bd92268fc74fb735767cbe0c0590e2d
round[ 3].m_col    4c9c1e66f771f0762c3f868e534df256
round[ 3].k_sch    b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 4].start    fa636a2825b339c940668a3157244d17
round[ 4].s_box    2dfb02343f6d12dd09337ec75b36e3f0
round[ 4].s_row    2d6d7ef03f33e334093602dd5bfb12c7
round[ 4].m_col    6385b79ffc538df997be478e7547d691
round[ 4].k_sch    47f7f7bc95353e03f96c32bcfd058dfd
round[ 5].start    247240236966b3fa6ed2753288425b6c
round[ 5].s_box    36400926f9336d2d9fb59d23c42c3950
round[ 5].s_row    36339d50f9b539269f2c092dc4406d23
round[ 5].m_col    f4bcd45432e554d075f1d6c51dd03b3c
round[ 5].k_sch    3caaa3e8a99f9deb50f3af57adf622aa
round[ 6].start    c81677bc9b7ac93b25027992b0261996
round[ 6].s_box    e847f56514dadde23f77b64fe7f7d490
round[ 6].s_row    e8dab6901477d4653ff7f5e2e747dd4f
round[ 6].m_col    9816ee7400f87f556b2c049c8e5ad036
round[ 6].k_sch    5e390f7df7a69296a7553dc10aa31f6b
round[ 7].start    c62fe109f75eedc3cc79395d84f9cf5d
round[ 7].s_box    b415f8016858552e4bb6124c5f998a4c
round[ 7].s_row    b458124c68b68a014b99f82e5f15554c
round[ 7].m_col    c57e1c159a9bd286f05f4be098c63439
round[ 7].k_sch    14f9701ae35fe28c440adf4d4ea9c026
round[ 8].start    d1876c0f79c4300ab45594add66ff41f
round[ 8].s_box    3e175076b61c04678dfc2295f6a8bfc0
round[ 8].s_row    3e1c22c0b6fcbf768da85067f6170495
round[ 8].m_col    baa03de7a1f9b56ed5512cba5f414d23
round[ 8].k_sch    47438735a41c65b9e016baf4aebf7ad2
round[ 9].start    fde3bad205e5d0d73547964ef1fe37f1
round[ 9].s_box    5411f4b56bd9700e96a0902fa1bb9aa1
round[ 9].s_row    54d990a16ba09ab596bbf40ea111702f
round[ 9].m_col    e9f74eec023020f61bf2ccf2353c21c7
round[ 9].k_sch    549932d1f08557681093ed9cbe2c974e
round[10].start    bd6e7c3df2b5779e0b61216e8b10b689
round[10].s_box    7a9f102789d5f50b2beffd9f3dca4ea7
round[10].s_row    7ad5fda789ef4e272bca100b3d9ff59f
round[10].k_sch    13111d7fe3944a17f307a78b4d2b30c5
round[10].output   69c4e0d86a7b0430d8cdb78070b4c55a

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
```

```
round[ 1].is_row   7a9f102789d5f50b2beffd9f3dca4ea7
round[ 1].is_box   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].ik_sch   549932d1f08557681093ed9cbe2c974e
round[ 1].ik_add   e9f74eec023020f61bf2ccf2353c21c7
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_row   5411f4b56bd9700e96a0902fa1bb9aa1
round[ 2].is_box   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].ik_sch   47438735a41c65b9e016baf4aebf7ad2
round[ 2].ik_add   baa03de7a1f9b56ed5512cba5f414d23
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_row   3e175076b61c04678dfc2295f6a8bfc0
round[ 3].is_box   d1876c0f79c4300ab45594add66ff41f
round[ 3].ik_sch   14f9701ae35fe28c440adf4d4ea9c026
round[ 3].ik_add   c57e1c159a9bd286f05f4be098c63439
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_row   b415f8016858552e4bb6124c5f998a4c
round[ 4].is_box   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].ik_sch   5e390f7df7a69296a7553dc10aa31f6b
round[ 4].ik_add   9816ee7400f87f556b2c049c8e5ad036
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_row   e847f56514dadde23f77b64fe7f7d490
round[ 5].is_box   c81677bc9b7ac93b25027992b0261996
round[ 5].ik_sch   3caaa3e8a99f9deb50f3af57adf622aa
round[ 5].ik_add   f4bcd45432e554d075f1d6c51dd03b3c
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_row   36400926f9336d2d9fb59d23c42c3950
round[ 6].is_box   247240236966b3fa6ed2753288425b6c
round[ 6].ik_sch   47f7f7bc95353e03f96c32bcfd058dfd
round[ 6].ik_add   6385b79ffc538df997be478e7547d691
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_row   2dfb02343f6d12dd09337ec75b36e3f0
round[ 7].is_box   fa636a2825b339c940668a3157244d17
round[ 7].ik_sch   b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 7].ik_add   4c9c1e66f771f0762c3f868e534df256
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_row   3b59cb73fcd90ee05774222dc067fb68
round[ 8].is_box   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].ik_sch   b692cf0b643dbdf1be9bc5006830b3fe
round[ 8].ik_add   ff87968431d86a51645151fa773ad009
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_row   a761ca9b97be8b45d8ad1a611fc97369
round[ 9].is_box   89d810e8855ace682d1843d8cb128fe4
round[ 9].ik_sch   d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 9].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_row   63cab7040953d051cd60e0e7ba70e18c
round[10].is_box   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
round[ 1].is_box   bdb52189f261b63d0b107c9e8b6e776e
round[ 1].is_row   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].im_col   4773b91ff72f354361cb018ea1e6cf2c
```

```
round[ 1].ik_sch   13aa29be9c8faff6f770f58000f7bf03
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_box   fde596f1054737d235febad7f1e3d04e
round[ 2].is_row   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].im_col   2d7e86a339d9393ee6570a1101904e16
round[ 2].ik_sch   1362a4638f2586486bff5a76f7874a83
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_box   d1c4941f7955f40fb46f6c0ad68730ad
round[ 3].is_row   d1876c0f79c4300ab45594add66ff41f
round[ 3].im_col   39daee38f4f1a82aaf432410c36d45b9
round[ 3].ik_sch   8d82fc749c47222be4dadc3e9c7810f5
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_box   c65e395df779cf09ccf9e1c3842fed5d
round[ 4].is_row   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].im_col   9a39bf1d05b20a3a476a0bf79fe51184
round[ 4].ik_sch   72e3098d11c5de5f789dfe1578a2cccb
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_box   c87a79969b0219bc2526773bb016c992
round[ 5].is_row   c81677bc9b7ac93b25027992b0261996
round[ 5].im_col   18f78d779a93eef4f6742967c47f5ffd
round[ 5].ik_sch   2ec410276326d7d26958204a003f32de
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_box   2466756c69d25b236e4240fa8872b332
round[ 6].is_row   247240236966b3fa6ed2753288425b6c
round[ 6].im_col   85cf8bf472d124c10348f545329c0053
round[ 6].ik_sch   a8a2f5044de2c7f50a7ef79869671294
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_box   fab38a1725664d2840246ac957633931
round[ 7].is_row   fa636a2825b339c940668a3157244d17
round[ 7].im_col   fc1fc1f91934c98210fbfb8da340eb21
round[ 7].ik_sch   c7c6e391e54032f1479c306d6319e50c
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_box   49e594f755ca638fda0a59a01f15d7fa
round[ 8].is_row   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].im_col   076518f0b52ba2fb7a15c8d93be45e00
round[ 8].ik_sch   a0db02992286d160a2dc029c2485d561
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_box   895a43e485188fe82d121068cbd8ced8
round[ 9].is_row   89d810e8855ace682d1843d8cb128fe4
round[ 9].im_col   ef053f7c8b3d32fd4d2a64ad3c93071a
round[ 9].ik_sch   8c56dff0825dd3f9805ad3fc8659d7fd
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_box   0050a0f04090e03080d02070c01060b0
round[10].is_row   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff
```

C.2 AES-192 (*Nk*=6, *Nr*=12)

```
PLAINTEXT:   00112233445566778899aabbccddeeff
KEY:         000102030405060708090a0b0c0d0e0f1011121314151617

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
```

```
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161758 46f2f95c43f4fe
round[ 2].start    4f63760643e0aa85aff8c9d041fa0de4
round[ 2].s_box    84fb386f1ae1ac977941dd70832dd769
round[ 2].s_row    84e1dd691a41d76f792d389783fbac70
round[ 2].m_col    9f487f794f955f662afc86abd7f1ab29
round[ 2].k_sch    544afef55847f0fa4856e2e95c43f4fe
round[ 3].start    cb02818c17d2af9c62aa64428bb25fd7
round[ 3].s_box    1f770c64f0b579deaaac432c3d37cf0e
round[ 3].s_row    1fb5430ef0accf64aa370cde3d77792c
round[ 3].m_col    b7a53ecbbf9d75a0c40efc79b674cc11
round[ 3].k_sch    40f949b31cbabd4d48f043b810b7b342
round[ 4].start    f75c7778a327c8ed8cfebfc1a6c37f53
round[ 4].s_box    684af5bc0acce85564bb0878242ed2ed
round[ 4].s_row    68cc08ed0abbd2bc642ef555244ae878
round[ 4].m_col    7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 4].k_sch    58e151ab04a2a5557effb5416245080c
round[ 5].start    22ffc916a81474416496f19c64ae2532
round[ 5].s_box    9316dd47c2fa92834390a1de43e43f23
round[ 5].s_row    93faa123c2903f4743e4dd83431692de
round[ 5].m_col    aaa755b34cffe57cef6f98e1f01c13e6
round[ 5].k_sch    2ab54bb43a02f8f662e3a95d66410c08
round[ 6].start    80121e0776fd1d8a8d8c31bc965d1fee
round[ 6].s_box    cdc972c53854a47e5d64c765904cc028
round[ 6].s_row    cd54c7283864c0c55d4c727e90c9a465
round[ 6].m_col    921f748fd96e937d622d7725ba8ba50c
round[ 6].k_sch    f501857297448d7ebdf1c6ca87f33e3c
round[ 7].start    671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 7].s_box    8572a1542fe5727b9e86c8df27bc1404
round[ 7].s_row    85e5c8042f8614549ebca17b277272df
round[ 7].m_col    e913e7b18f507d4b227ef652758acbcc
round[ 7].k_sch    e510976183519b6934157c9ea351f1e0
round[ 8].start    0c0370d00c01e622166b8accd6db3a2c
round[ 8].s_box    fe7b5170fe7c8e93477f7e4bf6b98071
round[ 8].s_row    fe7c7e71fe7f807047b95193f67b8e4b
round[ 8].m_col    6cf5edf996eb0a069c4ef21cbfc25762
round[ 8].k_sch    1ea0372a995309167c439e77ff12051e
round[ 9].start    7255dad30fb80310e00d6c6b40d0527c
round[ 9].s_box    40fc5766766c7bcae1d7507f09700010
round[ 9].s_row    406c501076d70066e17057ca09fc7b7f
round[ 9].m_col    7478bcdce8a50b81d4327a9009188262
round[ 9].k_sch    dd7e0e887e2fff68608fc842f9dcc154
round[10].start    a906b254968af4e9b4bdb2d2f0c44336
round[10].s_box    d36f3720907ebf1e8d7a37b58c1c1a05
round[10].s_row    d37e3705907a1a208d1c371e8c6fbfb5
round[10].m_col    0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[10].k_sch    859f5f237a8d5a3dc0c02952beefd63a
round[11].start    88ec930ef5e7e4b6cc32f4c906d29414
round[11].s_box    c4cedcabe694694e4b23bfdd6fb522fa
round[11].s_row    c494bffae62322ab4bb5dc4e6fce69dd
round[11].m_col    71d720933b6d677dc00b8f28238e0fb7
round[11].k_sch    de601e7827bcdf2ca223800fd8aeda32
round[12].start    afb73eeb1cd1b85162280f27fb20d585
round[12].s_box    79a9b2e99c3e6cd1aa3476cc0fb70397
round[12].s_row    793e76979c3403e9aab7b2d10fa96ccc
```

```
round[12].k_sch    a4970a331a78dc09c418c271e3a41d5d
round[12].output   dda97ca4864cdfe06eaf70a0ec0d7191

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_row   79a9b2e99c3e6cd1aa3476cc0fb70397
round[ 1].is_box   afb73eeb1cd1b85162280f27fb20d585
round[ 1].ik_sch   de601e7827bcdf2ca223800fd8aeda32
round[ 1].ik_add   71d720933b6d677dc00b8f28238e0fb7
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_row   c4cedcabe694694e4b23bfdd6fb522fa
round[ 2].is_box   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].ik_sch   859f5f237a8d5a3dc0c02952beefd63a
round[ 2].ik_add   0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_row   d36f3720907ebf1e8d7a37b58c1c1a05
round[ 3].is_box   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].ik_sch   dd7e0e887e2fff68608fc842f9dcc154
round[ 3].ik_add   7478bcdce8a50b81d4327a9009188262
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_row   40fc5766766c7bcae1d7507f09700010
round[ 4].is_box   7255dad30fb80310e00d6c6b40d0527c
round[ 4].ik_sch   1ea0372a995309167c439e77ff12051e
round[ 4].ik_add   6cf5edf996eb0a069c4ef21cbfc25762
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_row   fe7b5170fe7c8e93477f7e4bf6b98071
round[ 5].is_box   0c0370d00c01e622166b8accd6db3a2c
round[ 5].ik_sch   e510976183519b6934157c9ea351f1e0
round[ 5].ik_add   e913e7b18f507d4b227ef652758acbcc
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_row   8572a1542fe5727b9e86c8df27bc1404
round[ 6].is_box   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].ik_sch   f501857297448d7ebdf1c6ca87f33e3c
round[ 6].ik_add   921f748fd96e937d622d7725ba8ba50c
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_row   cdc972c53854a47e5d64c765904cc028
round[ 7].is_box   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].ik_sch   2ab54bb43a02f8f662e3a95d66410c08
round[ 7].ik_add   aaa755b34cffe57cef6f98e1f01c13e6
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_row   9316dd47c2fa92834390a1de43e43f23
round[ 8].is_box   22ffc916a81474416496f19c64ae2532
round[ 8].ik_sch   58e151ab04a2a5557effb5416245080c
round[ 8].ik_add   7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_row   684af5bc0acce85564bb0878242ed2ed
round[ 9].is_box   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].ik_sch   40f949b31cbabd4d48f043b810b7b342
round[ 9].ik_add   b7a53ecbbf9d75a0c40efc79b674cc11
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_row   1f770c64f0b579deaaac432c3d37cf0e
round[10].is_box   cb02818c17d2af9c62aa64428bb25fd7
round[10].ik_sch   544afef55847f0fa4856e2e95c43f4fe
round[10].ik_add   9f487f794f955f662afc86abd7f1ab29
round[11].istart   84e1dd691a41d76f792d389783fbac70
```

```
round[11].is_row   84fb386f1ae1ac977941dd70832dd769
round[11].is_box   4f63760643e0aa85aff8c9d041fa0de4
round[11].ik_sch   101112131415161758 46f2f95c43f4fe
round[11].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_row   63cab7040953d051cd60e0e7ba70e18c
round[12].is_box   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   000102030405060708090a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_box   afd10f851c28d5eb62203e51fbb7b827
round[ 1].is_row   afb73eeb1cd1b85162280f27fb20d585
round[ 1].im_col   122a02f7242ac8e20605afce51cc7264
round[ 1].ik_sch   d6bebd0dc209ea494db073803e021bb9
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_box   88e7f414f532940eccd293b606ece4c9
round[ 2].is_row   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].im_col   5cc7aecce3c872194ae5ef8309a933c7
round[ 2].ik_sch   8fb999c973b26839c7f9d89d85c68c72
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_box   a98ab23696bd4354b4c4b2e9f006f4d2
round[ 3].is_row   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].im_col   b7113ed134e85489b20866b51d4b2c3b
round[ 3].ik_sch   f77d6ec1423f54ef5378317f14b75744
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_box   72b86c7c0f0d52d3e0d0da104055036b
round[ 4].is_row   7255dad30fb80310e00d6c6b40d0527c
round[ 4].im_col   ef3b1be1b9b0e64bdcb79f1e0a707fbb
round[ 4].ik_sch   1147659047cf663b9b0ece8dfc0bf1f0
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_box   0c018a2c0c6b3ad016db7022d603e6cc
round[ 5].is_row   0c0370d00c01e622166b8accd6db3a2c
round[ 5].im_col   592460b248832b2952e0b831923048f1
round[ 5].ik_sch   dcc1a8b667053f7dcc5c194ab5423a2e
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_box   672ab1304edc9bfddf78f1033d1e1eef
round[ 6].is_row   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].im_col   0b8a7783417ae3a1f9492dc0c641a7ce
round[ 6].ik_sch   c6deb0ab791e2364a4055fbe568803ab
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_box   80fd31ee768c1f078d5d1e8a96121dbc
round[ 7].is_row   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].im_col   4ee1ddf9301d6352c9ad769ef8d20515
round[ 7].ik_sch   dd1b7cdaf28d5c158a49ab1dbbc497cb
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_box   2214f132a896251664aec94164ff749c
round[ 8].is_row   22ffc916a81474416496f19c64ae2532
round[ 8].im_col   1008ffe53b36ee6af27b42549b8a7bb7
round[ 8].ik_sch   78c4f708318d3cd69655b701bfc093cf
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_box   f727bf53a3fe7f788cc377eda65cc8c1
round[ 9].is_row   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].im_col   7f69ac1ed939ebaac8ece3cb12e159e3
```

```
round[ 9].ik_sch   60dcef10299524ce62dbef152f9620cf
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_box   cbd264d717aa5f8c62b2819c8b02af42
round[10].is_row   cb02818c17d2af9c62aa64428bb25fd7
round[10].im_col   cfaf16b2570c18b52e7fef50cab267ae
round[10].ik_sch   4b4ecbdb4d4dcfda5752d7c74949cbde
round[11].istart   84e1dd691a41d76f792d389783fbac70
round[11].is_box   4fe0c9e443f80d06affa76854163aad0
round[11].is_row   4f63760643e0aa85aff8c9d041fa0de4
round[11].im_col   794cf891177bfd1d8a327086f3831b39
round[11].ik_sch   1a1f181d1e1b1c194742c7d74949cbde
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_box   0050a0f04090e03080d02070c01060b0
round[12].is_row   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   000102030405060708090a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff
```

C.3 AES-256 (*Nk*=8, *Nr*=14)

```
PLAINTEXT:   00112233445566778899aabbccddeeff
KEY:         000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161718191a1b1c1d1e1f
round[ 2].start    4f63760643e0aa85efa7213201a4e705
round[ 2].s_box    84fb386f1ae1ac97df5cfd237c49946b
round[ 2].s_row    84e1fd6b1a5c946fdf4938977cfbac23
round[ 2].m_col    bd2a395d2b6ac438d192443e615da195
round[ 2].k_sch    a573c29fa176c498a97fce93a572c09c
round[ 3].start    1859fbc28a1c00a078ed8aadc42f6109
round[ 3].s_box    adcb0f257e9c63e0bc557e951c15ef01
round[ 3].s_row    ad9c7e017e55ef25bc150fe01ccb6395
round[ 3].m_col    810dce0cc9db8172b3678c1e88a1b5bd
round[ 3].k_sch    1651a8cd0244beda1a5da4c10640bade
round[ 4].start    975c66c1cb9f3fa8a93a28df8ee10f63
round[ 4].s_box    884a33781fdb75c2d380349e19f876fb
round[ 4].s_row    88db34fb1f807678d3f833c2194a759e
round[ 4].m_col    b2822d81abe6fb275faf103a078c0033
round[ 4].k_sch    ae87dff00ff11b68a68ed5fb03fc1567
round[ 5].start    1c05f271a417e04ff921c5c104701554
round[ 5].s_box    9c6b89a349f0e18499fda678f2515920
round[ 5].s_row    9cf0a62049fd59a399518984f26be178
round[ 5].m_col    aeb65ba974e0f822d73f567bdb64c877
round[ 5].k_sch    6de1f1486fa54f9275f8eb5373b8518d
round[ 6].start    c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 6].s_box    2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 6].s_row    2e6e7a2dafc6eef83a86ace7c25ba934
round[ 6].m_col    b951c33c02e9bd29ae25cdb1efa08cc7
round[ 6].k_sch    c656827fc9a799176f294cec6cd5598b
round[ 7].start    7f074143cb4e243ec10c815d8375d54c
round[ 7].s_box    d2c5831a1f2f36b278fe0c4cec9d0329
```

```
round[ 7].s_row    d22f0c291ffe031a789d83b2ecc5364c
round[ 7].m_col    ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 7].k_sch    3de23a75524775e727bf9eb45407cf39
round[ 8].start    d653a4696ca0bc0f5acaab5db96c5e7d
round[ 8].s_box    f6ed49f950e06576be74624c565058ff
round[ 8].s_row    f6e062ff507458f9be50497656ed654c
round[ 8].m_col    5174c8669da98435a8b3e62ca974a5ea
round[ 8].k_sch    0bdc905fc27b0948ad5245a4c1871c2f
round[ 9].start    5aa858395fd28d7d05e1a38868f3b9c5
round[ 9].s_box    bec26a12cfb55dff6bf80ac4450d56a6
round[ 9].s_row    beb50aa6cff856126b0d6aff45c25dc4
round[ 9].m_col    0f77ee31d2ccadc05430a83f4ef96ac3
round[ 9].k_sch    45f5a66017b2d387300d4d33640a820a
round[10].start    4a824851c57e7e47643de50c2af3e8c9
round[10].s_box    d61352d1a6f3f3a04327d9fee50d9bdd
round[10].s_row    d6f3d9dda6279bd1430d52a0e513f3fe
round[10].m_col    bd86f0ea748fc4f4630f11c1e9331233
round[10].k_sch    7ccff71cbeb4fe5413e6bbf0d261a7df
round[11].start    c14907f6ca3b3aa070e9aa313b52b5ec
round[11].s_box    783bc54274e280e0511eacc7e200d5ce
round[11].s_row    78e2acce741ed5425100c5e0e23b80c7
round[11].m_col    af8690415d6e1dd387e5fbedd5c89013
round[11].k_sch    f01afafee7a82979d7a5644ab3afe640
round[12].start    5f9c6abfbac634aa50409fa766677653
round[12].s_box    cfde0208f4b418ac5309db5c338538ed
round[12].s_row    cfb4dbedf4093808538502ac33de185c
round[12].m_col    7427fae4d8a695269ce83d315be0392b
round[12].k_sch    2541fe719bf500258813bbd55a721c0a
round[13].start    516604954353950314fb86e401922521
round[13].s_box    d133f22a1aed2a7bfa0f44697c4f3ffd
round[13].s_row    d1ed44fd1a0f3f2afa4ff27b7c332a69
round[13].m_col    2c21a820306f154ab712c75eee0da04f
round[13].k_sch    4e5a6699a9f24fe07e572baacdf8cdea
round[14].start    627bceb9999d5aaac945ecf423f56da5
round[14].s_box    aa218b56ee5ebeacdd6ecebf26e63c06
round[14].s_row    aa5ece06ee6e3c56dde68bac2621bebf
round[14].k_sch    24fc79ccbf0979e9371ac23c6d68de36
round[14].output   8ea2b7ca516745bfeafc49904b496089

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_row   aa218b56ee5ebeacdd6ecebf26e63c06
round[ 1].is_box   627bceb9999d5aaac945ecf423f56da5
round[ 1].ik_sch   4e5a6699a9f24fe07e572baacdf8cdea
round[ 1].ik_add   2c21a820306f154ab712c75eee0da04f
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_row   d133f22a1aed2a7bfa0f44697c4f3ffd
round[ 2].is_box   516604954353950314fb86e401922521
round[ 2].ik_sch   2541fe719bf500258813bbd55a721c0a
round[ 2].ik_add   7427fae4d8a695269ce83d315be0392b
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_row   cfde0208f4b418ac5309db5c338538ed
round[ 3].is_box   5f9c6abfbac634aa50409fa766677653
round[ 3].ik_sch   f01afafee7a82979d7a5644ab3afe640
round[ 3].ik_add   af8690415d6e1dd387e5fbedd5c89013
```

```
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_row   783bc54274e280e0511eacc7e200d5ce
round[ 4].is_box   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].ik_sch   7ccff71cbeb4fe5413e6bbf0d261a7df
round[ 4].ik_add   bd86f0ea748fc4f4630f11c1e9331233
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_row   d61352d1a6f3f3a04327d9fee50d9bdd
round[ 5].is_box   4a824851c57e7e47643de50c2af3e8c9
round[ 5].ik_sch   45f5a66017b2d387300d4d33640a820a
round[ 5].ik_add   0f77ee31d2ccadc05430a83f4ef96ac3
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_row   bec26a12cfb55dff6bf80ac4450d56a6
round[ 6].is_box   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].ik_sch   0bdc905fc27b0948ad5245a4c1871c2f
round[ 6].ik_add   5174c8669da98435a8b3e62ca974a5ea
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_row   f6ed49f950e06576be74624c565058ff
round[ 7].is_box   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].ik_sch   3de23a75524775e727bf9eb45407cf39
round[ 7].ik_add   ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_row   d2c5831a1f2f36b278fe0c4cec9d0329
round[ 8].is_box   7f074143cb4e243ec10c815d8375d54c
round[ 8].ik_sch   c656827fc9a799176f294cec6cd5598b
round[ 8].ik_add   b951c33c02e9bd29ae25cdb1efa08cc7
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_row   2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 9].is_box   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].ik_sch   6de1f1486fa54f9275f8eb5373b8518d
round[ 9].ik_add   aeb65ba974e0f822d73f567bdb64c877
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_row   9c6b89a349f0e18499fda678f2515920
round[10].is_box   1c05f271a417e04ff921c5c104701554
round[10].ik_sch   ae87dff00ff11b68a68ed5fb03fc1567
round[10].ik_add   b2822d81abe6fb275faf103a078c0033
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_row   884a33781fdb75c2d380349e19f876fb
round[11].is_box   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].ik_sch   1651a8cd0244beda1a5da4c10640bade
round[11].ik_add   810dce0cc9db8172b3678c1e88a1b5bd
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_row   adcb0f257e9c63e0bc557e951c15ef01
round[12].is_box   1859fbc28a1c00a078ed8aadc42f6109
round[12].ik_sch   a573c29fa176c498a97fce93a572c09c
round[12].ik_add   bd2a395d2b6ac438d192443e615da195
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_row   84fb386f1ae1ac97df5cfd237c49946b
round[13].is_box   4f63760643e0aa85efa7213201a4e705
round[13].ik_sch   101112131415161718191a1b1c1d1e1f
round[13].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_row   63cab7040953d051cd60e0e7ba70e18c
round[14].is_box   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

EQUIVALENT INVERSE CIPHER (DECRYPT):

```
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_box   629deca599456db9c9f5ceaa237b5af4
round[ 1].is_row   627bceb9999d5aaac945ecf423f56da5
round[ 1].im_col   e51c9502a5c1950506a61024596b2b07
round[ 1].ik_sch   34f1d1ffbfceaa2ffce9e25f2558016e
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_box   5153862143fb259514920403016695e4
round[ 2].is_row   516604954353950314fb86e401922521
round[ 2].im_col   91a29306cc450d0226f4b5eaef5efed8
round[ 2].ik_sch   5e1648eb384c350a7571b746dc80e684
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_box   5fc69f53ba4076bf50676aaa669c34a7
round[ 3].is_row   5f9c6abfbac634aa50409fa766677653
round[ 3].im_col   b041a94eff21ae9212278d903b8a63f6
round[ 3].ik_sch   c8a305808b3f7bd043274870d9b1e331
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_box   c13baaeccae9b5f6705207a03b493a31
round[ 4].is_row   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].im_col   638357cec07de6300e30d0ec4ce2a23c
round[ 4].ik_sch   b5708e13665a7de14d3d824ca9f151c2
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_box   4a7ee5c9c53de85164f348472a827e0c
round[ 5].is_row   4a824851c57e7e47643de50c2af3e8c9
round[ 5].im_col   ca6f71058c642842a315595fdf54f685
round[ 5].ik_sch   74da7ba3439c7e50c81833a09a96ab41
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_box   5ad2a3c55fe1b93905f3587d68a88d88
round[ 6].is_row   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].im_col   ca46f5ea835eab0b9537b6dbb221b6c2
round[ 6].ik_sch   3ca69715d32af3f22b67ffade4ccd38e
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_box   d6a0ab7d6cca5e695a6ca40fb953bc5d
round[ 7].is_row   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].im_col   2a70c8da28b806e9f319ce42be4baead
round[ 7].ik_sch   f85fc4f3374605f38b844df0528e98e1
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_box   7f4e814ccb0cd543c175413e8307245d
round[ 8].is_row   7f074143cb4e243ec10c815d8375d54c
round[ 8].im_col   f0073ab7404a8a1fc2cba0b80df08517
round[ 8].ik_sch   de69409aef8c64e7f84d0c5fcfab2c23
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_box   c345bdfa1bc799e1a2dcaab0a857b728
round[ 9].is_row   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].im_col   3225fe3686e498a32593c1872b613469
round[ 9].ik_sch   aed55816cf19c100bcc24803d90ad511
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_box   1c17c554a4211571f970f24f0405e0c1
round[10].is_row   1c05f271a417e04ff921c5c104701554
round[10].im_col   9d1d5c462e655205c4395b7a2eac55e2
round[10].ik_sch   15c668bd31e5247d17c168b837e6207c
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_box   979f2863cb3a0fc1a9e166a88e5c3fdf
round[11].is_row   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].im_col   d24bfb0e1f997633cfce86e37903fe87
round[11].ik_sch   7fd7850f61cc991673db890365c89d12
```

```
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_box   181c8a098aed61c2782ffba0c45900ad
round[12].is_row   1859fbc28a1c00a078ed8aadc42f6109
round[12].im_col   aec9bda23e7fd8aff96d74525cdce4e7
round[12].ik_sch   2a2840c924234cc026244cc5202748c4
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_box   4fe0210543a7e706efa476850163aa32
round[13].is_row   4f63760643e0aa85efa7213201a4e705
round[13].im_col   794cf891177bfd1ddf67a744acd9c4f6
round[13].ik_sch   1a1f181d1e1b1c191217101516131411
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_box   0050a0f04090e03080d02070c01060b0
round[14].is_row   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

Appendix D - References

[1] AES page available via http://www.nist.gov/CryptoToolkit.[4]

[2] Computer Security Objects Register (CSOR): http://csrc.nist.gov/csor/.

[3] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available at [1].

[4] J. Daemen and V. Rijmen, *The block cipher Rijndael*, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[5] B. Gladman's AES related home page http://fp.gladman.plus.com/cryptography_technology/.

[6] A. Lee, NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, National Institute of Standards and Technology, November 1999.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997, p. 81-83.

[8] J. Nechvatal, et. al., *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, October 2, 2000, available at [1].

[4] A complete set of documentation from the AES development effort – including announcements, public comments, analysis papers, conference proceedings, etc. – is available from this site.