UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

# Soft IP para Criptografia usando o Algoritmo Rijndael e Implementação em Lógica Programável

por

MARCELO BOEIRA DE BARCELOS

Dissertação submetida à avaliação
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Porto Alegre, fevereiro de 2002

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Barcelos, Marcelo Boeira de

Soft IP para Criptografia usando o Algoritmo Rijndael e Implementação em Lógica Programável / por Marcelo Boeira de Barcelos. – Porto Alegre: PPGC da UFRGS, 2002.

147p. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Reis, Ricardo Augusto da Luz.

1. Criptografia. 2. VHDL. 3. Rijndael. 4. FPGA. 5. Microeletrônica. I. Reis, Ricardo Augusto da Luz. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fensterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenadora do PPGC: Profa. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# Agradecimentos

Todo este trabalho não teria sido possível sem o apoio e a compreensão de uma pessoa muito especial: a minha esposa. Renata, muito obrigado pelas palavras e gestos de incentivo e força que sempre me foram dados. Com certeza, eu nunca me esquecerei destes momentos.

Eu gostaria de agradecer também à minha mãe, meu pai e minha irmã, pelo incomparável apoio e educação que me deram durante toda a minha vida, possibilitando que mais esse desafio fosse completado.

Quero agradecer também a todo o restante da minha família, pedindo desculpas pelos momentos em que não estive presente.

Ao Tibério, meu grande amigo, pela excelente revisão do texto e pelas conversas tidas durante toda a graduação e pós-graduação.

Ao meu orientador, Ricardo Reis, pela conduta, pela orientação sempre correta, pelo incentivo, pela cobrança e principalmente por acreditar nos resultados deste trabalho. Reis, muito obrigado!

Aos professores de microeletrônica do PPGC, ao pessoal do GME, e a todos aqueles que fazem deste grupo um centro de excelência no Brasil e no exterior. Agradecimento especial ao Alex Panato, bolsista de Iniciação Científica, pelo trabalho desenvolvido nesta dissertação. Obrigado também a todos os funcionários e demais integrantes do Instituto de Informática, pela ajuda e apoio a mim e todos os demais estudantes e pesquisadores.

Aos meus colegas de trabalho durante o período da dissertação, pela compreensão.

Agradeço, finalmente, à Deus, pela vida e por tudo.

Enfim, muito obrigado!

# Sumário

**Lista de Abreviaturas** ..... 7
**Lista de Figuras** ..... 8
**Lista de Tabelas** ..... 10
**Resumo** ..... 11
**Abstract** ..... 12
**1 Introdução** ..... 13
**1.1 Objetivos** ..... 14
**1.2 A Internet** ..... 15
**1.3 Segurança da informação** ..... 17
1.3.1 Confidencialidade ..... 17
1.3.2 Autenticação ..... 17
1.3.3 Não-repúdio ..... 17
1.3.4 Integridade ..... 17
**1.4 Hardware desempenhando funções criptográficas** ..... 17
**1.5 Advanced Encryption Standard (AES)** ..... 19
**1.6 Representação numérica** ..... 21
**2 Conceitos básicos de criptografia** ..... 22
**2.1 Histórico** ..... 22
**2.2 O nível de segurança oferecido pelas técnicas de criptografia** ..... 22
**2.3 A criptoanálise** ..... 23
**2.4 Criptografia como funções matemáticas** ..... 24
2.4.1 Definições: ..... 24
**2.5 Algoritmos simétricos** ..... 25
**2.6 Algoritmos assimétricos** ..... 28
**2.7 Funções hash** ..... 31
**3 Algoritmo Rijndael** ..... 34
**3.1 Fundamentos matemáticos** ..... 34
3.1.1 Operações ao nível de byte ..... 34
3.1.2 Operações ao nível de palavras ..... 36
**3.2 Descrição** ..... 38
**3.3 "State" e "Cipher Key"** ..... 39
**3.4 Número de rodadas** ..... 40
**3.5 Cipher** ..... 41
3.5.1 ByteSub ..... 42
3.5.2 ShiftRow ..... 43
3.5.3 MixColumn ..... 44
3.5.4 AddRoundKey ..... 45
**3.6 Expansion Key** ..... 45
3.6.1 KStran ..... 46
**3.7 Inverse Cipher** ..... 46
3.7.1 InvByteSub ..... 49
3.7.2 InvShiftRow ..... 50
3.7.3 InvMixColumn ..... 51
**4 Implementação do UFIR-128** ..... 52
**4.1 Modos de operação** ..... 52
4.1.1 Electronic Codebook (ECB) ..... 52

4.1.2 Cipher Block Chaining (CBC) ... 53
4.1.3 Cipher Feedback (CFB) ... 54
4.1.4 Output Feedback (OFB) ... 55
4.1.5 Counter (CTR) ... 56
4.1.6 Cipher Block Chaining – Message Authentication Code (CBC-MAC) ... 56
**4.2 Arquiteturas de projeto para cifradores simétricos ... 57**
4.2.1 Arquitetura básica ... 57
4.2.2 Iteração com múltiplas rodadas (Loop unrolling) ... 58
4.2.3 Paralelismo dentro da rodada (Inner-round pipelining) ... 59
4.2.4 Paralelismo fora da rodada (Outer-round pipelining) ... 61
4.2.5 Comparação de desempenho ... 61
**4.3 Decisões de projeto ... 62**
**4.4 Ambiente de desenvolvimento ... 64**
**4.5 Implementação VHDL das transformações básicas do Rijndael ... 64**
4.5.1 ByteSub ... 64
4.5.2 ShiftRow ... 66
4.5.3 MixColumn ... 67
4.5.4 AddRoundKey ... 68
4.5.5 Expansion Key ... 69
4.5.6 Inverse ByteSub ... 71
4.5.7 Inverse ShiftRow ... 72
4.5.8 Inverse MixColumn ... 73
**4.6 Arquitetura e bloco de controle da versão de 128 bits ... 76**
4.6.1 Rijndael na encriptação de 128 bits ... 77
4.6.2 Rijndael na decriptação de 128 bits ... 78
**4.7 Arquitetura e bloco de controle da versão de 32 bits ... 79**
4.7.1 Rijndael na encriptação de 32 em 32 bits ... 81
4.7.2 Rijndael na decriptação de 32 em 32 bits ... 82
**5 Resultados ... 83**
**5.1 Arquitetura de 128 bits ... 83**
5.1.1 Comentários ... 85
**5.2 Arquitetura de 32 bits ... 86**
5.2.1 Comentários ... 87
**5.3 Implementação do AES-128 sobre outras plataformas FPGA ... 87**
**6 Conclusão ... 89**
**Anexo 1 Dispositivo 128 bits ... 92**
**A.1 Interface ... 92**
**A.2 Operação ... 93**
**A.3 Encriptador ... 97**
A.3.1. Diagrama de blocos ... 97
A.3.2 Performance ... 98
A.3.3. Utilização do componente ... 100
A.3.4. Pinagem ... 101
**A.4 Decriptador ... 103**
A.4.1. Diagrama de blocos ... 103
A.4.2. Performance ... 104
A.4.3 Utilização do componente ... 106
A.4.4. Pinagem ... 106
**A.5 Resultados encriptador versus decriptador ... 108**
**Anexo2 Dispositivo 32 bits ... 111**

**B.1 Interface** ..... **111**
**B.2 Operação** ..... **112**
**B.3 Encriptador** ..... **116**
B.3.1. Diagrama de blocos ..... 116
B.3.2. Desempenho ..... 117
B.3.3 Utilização do componente ..... 119
B.3.4. Pinagem ..... 119
**B.4 Decriptador** ..... **121**
B.4.1 Diagrama de blocos ..... 121
B.4.2 Desempenho ..... 123
B.4.3 Utilização do componente ..... 125
B.4.4 Pinagem ..... 125
**Anexo 3 Ambiente de Desenvolvimento** ..... **128**
**Glossário** ..... **133**
**Bibliografia** ..... **135**

# Lista de Abreviaturas

| | |
|---|---|
| AES | Advanced Encryption Standard |
| NIST | National Institute of Standards and Technologies |
| IP | Intellectual Properties |
| DES | Data Encryption Standard |
| NSA | National Security Agency |
| SSH | Secure Shell |
| SSL | Secure Sockets Layer |
| TLS | Transport Layer Security |
| VPN | Virtual Private Network |
| PGP | Pretty Good Privacy |
| FPGA | Field Programmable Gate Array |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| ANSI | American National Standards Institute |
| ASIC | Application Specific Integrated Circuit |
| EDIF | Electronic Design Interchange Format |
| IETF | Internet Engineering Task Force |
| RFC | Request for Comments |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| PDF | Portable Document Format |

# Lista de Figuras

FIGURA 2.1 - Criptografia utilizando chaves....25
FIGURA 2.2 - Algoritmo Simétrico com Chaves Iguais....26
FIGURA 2.3 - Chaves Pública e Privada....29
FIGURA 2.4 - Enviando uma mensagem cifrada para o mensageiro B....30
FIGURA 2.5 - Enviando uma mensagem assinada para o mensageiro B....30
FIGURA 2.6 - O *hash* de uma mensagem equivale à impressão digital do seu conteúdo....32
FIGURA 2.7 - Método de Assinatura Digital de documentos utilizando Função Hash....32
FIGURA 2.8 - Iteração da função *hash*....33
FIGURA 3.1 - Multiplicação de polinômios representados por palavras....37
FIGURA 3.2 - Multiplicação por "x"....38
FIGURA 3.3 - Estrutura do algoritmo Rijndael....39
FIGURA 3.4 - *State* como um array 4 x $N_b$ de bytes....39
FIGURA 3.5 - Cipher Key como um array 4 x $N_k$ de bytes....40
FIGURA 3.6 - Diagrama em Blocos do Cipher....41
FIGURA 3.7 - Transformação BytesSub atuando sobre o *state*....42
FIGURA 3.8 - Transformação ShiftRow atuando sobre o *state*....44
FIGURA 3.9 - Transformação MixColumn atuando sobre o *state*....44
FIGURA 3.10 - Transformação AddRoundKey, tendo como parâmetro a *round key*....45
FIGURA 3.11 - Array unidimensional de palavras....45
FIGURA 3.12 - Transformação KStran....46
FIGURA 3.13 - Diagrama em blocos do Inverse Cipher....47
FIGURA 3.14 - Diagrama em Blocos do Inverse Cipher....49
FIGURA 3.15 - Transformação InvShiftRow atuando sobre o *state*....50
FIGURA 3.16 - Transformação InvMixColumn atuando sobre o *state*....51
FIGURA 4.1 - Encriptação e Decriptação, respectivamente, no modo ECB....53
FIGURA 4.2 - Encriptação no modo CBC....53
FIGURA 4.3 - Decriptação no modo CBC....53
FIGURA 4.4 - Encriptação no modo CFB....54
FIGURA 4.5 - Decriptação no modo CFB....54
FIGURA 4.6 - Encriptação no modo OFB....55
FIGURA 4.7 - Decriptação no modo OFB....55
FIGURA 4.8 - Encriptação no modo CTR....56
FIGURA 4.9 - Decriptação no modo CTR....56
FIGURA 4.10 - Geração do Message Authentication Code a partir de um mensagem....57
FIGURA 4.11 - Arquitetura básica para implementação de criptografia simétrica....58
FIGURA 4.12 - Iteração com múltiplas rodadas....59
FIGURA 4.13 - Paralelismo dentro da rodada....60
FIGURA 4.14 - Diagrama temporal da arquitetura de paralelismo dentro da rodada....60
FIGURA 4.15 - Paralelismo fora da rodada....61
FIGURA 4.16 - Comparação entre velocidade e área para as quatros arquiteturas segundo [GAJ00]....62
FIGURA 4.17 - Blocos funcionais da implementação de 128 bits....76
FIGURA 4.18 - Processo Rijndael no ENC128....78
FIGURA 4.19 - Processo Rijndael no DEC128....79
FIGURA 4.20 - Blocos funcionais da implementação de 32 bits....80
FIGURA 4.21 - Máquina de estados do ENC32....81
FIGURA 4.22 - Máquina de estados do DEC32....82
FIGURA A.1 - Entrada da chave....93
FIGURA A.2 - Entrada do dado inicial....94
FIGURA A.3 - Entrada de dados novos ao longo da operação....95
FIGURA A.4 - DATA_OK indica quando pode-se ler e escrever dados no barramento....95
FIGURA A.5 - Modo de espera do dispositivo....96
FIGURA A.6 - Entrada de dados antes de sair do modo de espera....97
FIGURA A.7 - Diagrama de blocos do encriptador 128 bits....97
FIGURA A.8 - CLK mínimo do encriptador....99
FIGURA A.9 - CLK2 mínimo do encriptador....99
FIGURA A.10 - Primeiro dado pronto para ser lido....100

FIGURA A.11 - Segundo dado pronto para ser lido ..... 100
FIGURA A.12 - Vista inferior do EPF10K250AGC599-1 ..... 101
FIGURA A.13 - Pinagem do ENC128 ..... 102
FIGURA A.14 - Diagrama de blocos do decriptador 128 bits. ..... 103
FIGURA A.15 - CLK mínimo do decriptador ..... 104
FIGURA A.16 - CLK2 mínimo do decriptador ..... 105
FIGURA A.17 - Primeiro dado pronto para ser lido ..... 105
FIGURA A.18 - Segundo dado pronto para ser lido ..... 106
FIGURA A.19 - Vista inferior do EPF10K250AGC599-1 ..... 107
FIGURA A.20 - Pinagem do DEC128 ..... 108
FIGURA A.21 - Novo CLK2 mínimo do encriptador sem o Mix_Column ..... 109
FIGURA A.22 - Saída do encriptador sem o Mix_Column ..... 109
FIGURA A.23 - Novo CLK2 mínimo do decriptador sem o IMix_Column ..... 109
FIGURA A.24 - Saída do decriptador sem o IMix_Column ..... 110
FIGURA B.1 - Entrada da chave ..... 112
FIGURA B.2 - Entrada do dado inicial ..... 113
FIGURA B.3 - Entrada de dados novos ao longo da operação ..... 114
FIGURA B.4 - DATA_OK indica quando pode-se ler e escrever dados no barramento ..... 114
FIGURA B.5 - Modo de espera do dispositivo ..... 115
FIGURA B.6 - Entrada de dados antes de sair do Halt ..... 115
FIGURA B.7 - Diagrama de blocos do encriptador 32 bits ..... 116
FIGURA B.8 - Máquina de estados do encriptador 32 bits ..... 117
FIGURA B.9 - CLK mínimo do encriptador ..... 118
FIGURA B.10 - Primeiro dado pronto para ser lido ..... 118
FIGURA B.11 - Segundo dado pronto para ser lido ..... 119
FIGURA B.12 - Vista inferior do EPF10K50VFC484-1 ..... 120
FIGURA B.13 - Pinagem do ENC32 ..... 121
FIGURA B.14 - Diagrama de blocos do decriptador 128 bits ..... 122
FIGURA B.15 - Máquina de estados do decriptador 32 bits ..... 123
FIGURA B.16 - CLK mínimo do decriptador ..... 124
FIGURA B.17 - Primeiro dado pronto para ser lido ..... 124
FIGURA B.18 - Segundo dado pronto para ser lido ..... 125
FIGURA B.19 - Vista inferior do Vista inferior do EPF10K50VFC484-1 ..... 126
FIGURA B.20 - Pinagem do DEC32 ..... 127
FIGURA C.1 - Codificação VHDL no LeonardoSpectrum ..... 128
FIGURA C.2 - Verificação VHDL no LeonardoSpectrum ..... 129
FIGURA C.3 - Importação de arquivo EDIF no Max+plus II ..... 130
FIGURA C.4 - Compilação no Max+plus II ..... 131
FIGURA C.5 - Simulação e verificação no Max+plus II ..... 132

# Lista de Tabelas

TABELA 1.1 – Quantidade de usuários em diferentes regiões mundiais ..... 15
TABELA 1.2 - Implementação do AES-128 em diferentes plataformas ..... 21
TABELA 1.3 – Representação numérica utilizada na dissertação ..... 21
TABELA 2.1 – Algoritmos de criptografia simétrica ..... 26
TABELA 3.1 - Número de rodadas em função dos tamanhos do bloco e da chave ..... 40
TABELA 3.2 - Número de rodadas definidas pelo AES ..... 40
TABELA 3.3 - S-Box da Encriptação ..... 43
TABELA 3.4 - Deslocamento do ShiftRow em função de $N_b$ ..... 43
TABELA 3.5 - S-Box da Decriptação ..... 50
TABELA 5.1 – Período mínimo dos sinais de relógio na arquitetura de 128 bits ..... 84
TABELA 5.2 – Comparação da taxa de transferência máxima ..... 84
TABELA 5.3 – Utilização do componente EPF10K250AGC599-1 para implementar o AES-128 ..... 85
TABELA 5.4 – ENC128 e DEC128 modificados ..... 85
TABELA 5.5 – Período mínimo dos sinais de relógio na arquitetura de 32 bits ..... 86
TABELA 5.6 – Utilização do componente EPF10K50VFC484-1 para implementar o AES-128 ..... 86
TABELA A.1 – Pinagem dos dispositivos de 128 bits (encriptação/decriptação) ..... 92
TABELA A.2 – Ocupação de área do encriptador 128 bits ..... 101
TABELA A.3 – Ocupação de área do decriptador 128 bits ..... 106
TABELA B.1 – Pinagem para o dispositivo de 32 bits ..... 111
TABELA B.2 – Ocupação de área do encriptador 32 bits ..... 119
TABELA B.3 – Ocupação de área do decriptador 32 bits ..... 125

# Resumo

A criptografia assumiu papel de destaque no cotidiano das pessoas, em virtude da necessidade de segurança em inúmeras transações eletrônicas. Em determinadas áreas, a utilização de hardware dedicado à tarefa de criptografia apresenta vantagens em relação à implementação em software, devido principalmente ao ganho de desempenho.

Recentemente, o *National Institute of Standards and Technology* (NIST) publicou o novo padrão norte-americano de criptografia simétrica, chamado *de Advanced Encryption Standard* (AES). Após um período de aproximadamente 3 anos, no qual várias alternativas foram analisadas, adotou-se o algoritmo Rijndael.

Assim, este trabalho apresenta um Soft IP do padrão AES, codificado em VHDL, visando a implementação em FPGA Altera™. Todo o projeto foi construído com funções e bibliotecas genéricas, a fim de permitir a posterior implementação sobre outras tecnologias.

Foram geradas duas versões: uma priorizando desempenho e outra priorizando a área ocupada nos componentes. Para cada uma das versões, produziu-se um circuito para encriptar e outro para decriptar. O desempenho alcançado em termos de velocidade de processamento superou todos os outros trabalhos publicados na área, sobre a mesma tecnologia.

São apresentados os detalhes de implementação, arquiteturas envolvidas e decisões de projeto, bem como todos os resultados. A dissertação contém ainda conceitos básicos de criptografia e uma descrição do algoritmo Rijndael.

**Palavras-Chave**: Criptografia, VHDL, Rijndael, FPGA, microeletrônica

**TITLE**: "SOFT IP FOR CRYPTOGRAPHY USING THE RIJNDAEL ALGORITHM AND IMPLEMENTATION IN PROGRAMMABLE LOGIC"

# Abstract

The cryptography is part of many people daily life, due to the increasing need of security in electronic transactions. In some areas, the usage of dedicated hardware presents advantages over the cryptography implementations in software, mainly because of the performance.

Recently, the National Institute of Standards and Technologies (NIST) published the new North American symmetric cryptography standard, called Advanced Encryption Standard (AES). After more than 3 years of analyzes, the Rijndael algorithm was adopted.

So, this work presents a Soft IP of the AES, coded in VHDL, aiming an Altera™ FPGA implementation. Generic VHDL functions and libraries were used, producing a standard code that could be used to implement the Rijndael with other technologies.

Two versions were generated: one of then optimized for performance and the other one optimized for area purposes. Each version has a circuit for encryption and other for decryption. The obtained performance overcomes any other published work using the chosen technology.

All the implementation details, architecture and project decisions, as all the results, are presented. The text also presents basic cryptography concepts and a description of the Rijndael algorithm.

**Keywords**: Cryptography, VHDL, Rijndael, FPGA, Microelectronics

# 1 Introdução

Vive-se um contexto mundial no qual a informação desempenha papel estratégico em todas as atividades exercidas pela sociedade. Além disso, o avanço tecnológico possibilitou a integração de sistemas outrora independentes, aumentando a acessibilidade aos dados. No final do ano de 2001, por exemplo, entrará em operação o Sistema de Pagamentos Brasileiro, que interligará através de uma rede TCP/IP [COM 95] todas as instituições financeiras do país [BCB 2001]. Em especial, a Tecnologia da Informação evoluiu e hoje está presente no dia-a-dia de todos. Cidadãos comuns utilizam diariamente serviços bancários eletrônicos, dispensando as filas do passado. Ferramentas de busca de informação multiplicam-se, tornando-se cada vez mais inteligentes. A velocidade de mudança da realidade acompanha o ritmo do avanço tecnológico e exige, cada vez mais, informações atualizadas que sirvam de sustentação para a tomada de decisões.

A informação, como um dos pilares da nova economia, necessita ser cada vez mais confiável, disponível, íntegra, e muitas vezes confidencial. A fim de prover segurança utiliza-se critptografia como uma poderosa ferramenta. Ainda no mesmo contexto, aparece a criptografia em hardware, capaz de oferecer vantagens em situações nas quais a implementação em software não é a melhor alternativa.

O presente trabalho versa sobre a implementação em VHDL e síntese em FPGA de um algoritmo de criptografia chamado Rijndael [DAE 99]. O projeto levou o nome de UFIR-128, sendo um acrônimo para *UFRGS FPGA Implementation of Rijndael*. O parâmetro 128 indica uma implementação com chaves de 128 bits [SCH 96].

Esta dissertação foi dividida em seis (6) capítulos e três (3) anexos. No primeiro capítulo são mostradas as motivações e os objetivos do trabalho. Então, procura-se posicionar o leitor sobre o mercado de segurança da informação, e para tanto é necessário mostrar alguns números atuais sobre o desenvolvimento da Internet, elemento essencial em qualquer análise. Depois, a área de criptografia em hardware é abordada. Conta-se também um pouco de história do desenvolvimento do novo padrão norte-americano para criptografia simétrica.

No segundo capítulo, denominado "Conceitos básicos de criptografia", é fornecido um embasamento teórico mínimo sobre criptografia, mostrando os fundamentos necessários para a compreensão do trabalho. Conceitos como criptografia simétrica e assimétrica, funções de mão única, chamadas funções *hash*, robustez de algoritmos criptográficos, entre outros, são abordados. Não é intuito desta dissertação fornecer conhecimentos mais aprofundados, tampouco explicar as bases matemáticas que fundamentam os algoritmos, mas sim apresentar uma visão prática capaz de permitir uma implementação eficiente.

Em seguida, no capítulo chamado de "Algoritmo Rijndael", considerando que o leitor já possui um maior conhecimento das questões preliminares de criptografia, são explicadas as transformações pertinentes ao Rijndael. Novamente, não são focadas questões relativas aos fundamentos matemáticos, mas sim aos aspectos funcionais para a implementação do algoritmo. Prioriza-se a compreensão das transformações, de maneira que seja possível visualizar a sua implementação em hardware. Todo o algoritmo é mostrado, incluindo a encriptação, a decriptação e geração de chaves utilizadas internamente.

O quarto capítulo apresenta o trabalho de implementação, sendo por isso chamado de Implementação do UFIR-128. Nesta parte, são mostradas as alternativas de arquitetura e os modos de operação possíveis. Em seguida, as decisões de projeto são

justificadas. Parte-se então para a explicação dos diferentes módulos, mostrando todas as metodologias escolhidas para a manipulação dos dados. Este é o capítulo que objetiva a compreensão do desenvolvimento da implementação.

O capítulo denominado "Resultados" é então apresentado, mostrando a produção final em relação aos vários objetivos traçados no início da dissertação. Fatores quantitativos são mostrados, como as performances[1] alcançadas em variados componentes, bem como as taxas de ocupação em termos de lógica utilizada.

Na conclusão, faz-se um levantamento crítico dos resultados mostrados anteriormente, em função do estado-da-arte. Como também, as principais dificuldades, conquistas e desafios ainda presentes são citados, a fim de criar um caminho que possa ser seguido também por outros pesquisadores. Ao final, espera-se apresentar justificativas suficientes para que o projeto UFIR prossiga.

Os anexos são então apresentados. Os dois primeiros abordam a documentação das diferentes versões dos módulos desenvolvidos, de maneira a habilitar a utilização de maneira modular em outros sistemas. O terceiro anexo mostra através de figuras o ambiente de desenvolvimento utilizado

Aos leitores já conhecedores dos conceitos de criptografia e também do algoritmo Rijndael, recomenda-se necessariamente a leitura da introdução, do capítulo que versa sobre a implementação, dos resultados e da conclusão. Tais tópicos são essenciais para o entendimento das motivações e objetivos iniciais, das decisões de projeto do UFIR, dos resultados encontrados e das conclusões e desafios futuros. Àqueles que julgarem necessário reforçar os fundamentos e conhecer em maiores detalhes o funcionamento do Rijndael, sugere-se então, adicionalmente, a leitura dos Capítulos 2 e 3.

## *1.1 Objetivos*

São quatro (4) as constatações e acontecimentos, conforme será mostrado ao longo deste capítulo, que motivaram esta dissertação:

- A Segurança de Informação desempenha papel essencial na nova economia.
- A importância da criptografia como ferramenta para a Segurança da Informação.
- Utilização de Hardware/FPGA para desempenhar funções criptográficas específicas.
- A escolha do novo padrão AES.

Assim, o trabalho apresentado possui os seguintes objetivos:

- Fazer um estudo aprofundado do algoritmo Rijndael, de maneira a identificar suas principais características, a fim de escolher qual o modo mais apropriado de implementação em lógica programável Altera®.

- Produzir duas codificações em VHDL do algoritmo Rijndael, possibilitando a sua implementação em Lógica Programável Altera®. Uma das versões será

[1] A palavra performance, apesar de pertencer à lingua inglesa, figura no Novo Dicionário Aurélio [FER9?]. Por ser muito utilizada na literatura especializada, optou-se por utilizá-la ao longo desta dissertação, com o mesmo significado de desempenho.

otimizada para garantir alta performance, enquanto a outra versão terá como prioridade a baixa ocupação de área nos circuitos lógicos programáveis. Verificar a adequação do projeto aos componentes lógicos Altera®.

- Verificar a funcionalidade requerida do algoritmo de criptografia. Através de simulações, serão analisados todos os blocos funcionais, a fim de garantir a implementação correta do algoritmo Rijndael em linguagem VHDL padrão. O codigo poderá, então, ser implementado em outros componentes utilizando uma grande gama de ferramentas.

- Determinar os limites de desempenho alcançados em termos de taxa de transferência máxima de bits, como também apresentar os números referentes à utilização de área dos circuitos lógicos programáveis, discriminando quais unidades lógicas são necessárias para implementar o projeto.

## *1.2 A Internet*

Com certeza, papel essencial em todas as modificações do cenário mundial coube à Internet, que possibilitou troca de informações em níveis nunca antes conseguidos. A sociedade já reflete a capacidade de mudança inerente à Internet, e o futuro não é projetado sem levar em consideração as novas aplicações que surgem a cada dia. Conceitos como "home-banking", "e-commerce" e "Business-to-Business" já tomaram o seu lugar no cotidiano de várias empresas e pessoas.

Apesar de não ser o escopo deste trabalho, será apresentado um breve histórico sobre a Internet, seguido de alguns dados atuais sobre o panorama atual. Por fim, serão analisados alguns aspectos relevantes, a fim de permitir uma melhor compreensão dos objetivos finais da dissertação.

No final da década de 60 uma rede chamada ARPANET começou o seu desenvolvimento. O objetivo era interligar os computadores de institutos acadêmicos de pesquisa, mas sempre levando em consideração os computadores militares americanos, de maneira eficiente e confiável [TAN 96]. A arquitetura da rede deveria prever que determinados pontos pudessem ser desconectados, sem contudo paralisar as comunicações entre os diversos equipamentos. Assim nasceu o protocolo TCP/IP, base da conhecida Internet [COM 95].

Segundo dados fornecidos pela consultoria Nielsen//NetRatings [NIE 2001] e pelo Ibope eRatings [IBO 2001][IDG 2001a], o número de usuários Internet em algumas regiões pode ser vislumbrada na tabela a seguir:

TABELA 1.1 – Quantidade de usuários em diferentes regiões mundiais

| **País** | **Número de Usuários (milhões)** | **Mês da estimativa** |
|---|---|---|
| EUA | 165,2 | Julho/01 |
| Alemanha | 27,9 | Julho/01 |
| Reino Unido | 23,9 | Julho/01 |
| Itália | 18,7 | Julho/01 |
| Brasil | 11,3 | Junho/01 |

Destacam-se ainda:

- Yankee Group prevê 42,3 milhões de internautas brasileiros em 2006 [IDG 2001b];
- 429 milhões de internautas estão na rede [IDG 2001c];
- 93% das declarações do IRPF 2001 foram feitas pela Web [IDG 2001d];
- estimativas apontam para compras online da ordem de US$ 27,5 bilhões nos últimos 12 meses no mercado norte americano [EMA 2001].

Tais números são somente uma amostra de quão representativa se tornou a Internet desde o seu desenvolvimento inicial para a sociedade moderna. Tanto em termos econômicos, quando são analisados o montante de dinheiro e transações comerciais que circulam na Web, quanto em termos de comportamento, lembrando o percentual de contribuintes que entregaram a sua declaração de imposto de renda eletronicamente, percebe-se a presença constante de uma nova realidade.

Entretanto, o protocolo TCP/IP na versão 4, atualmente difundido, não oferece os requisitos de segurança exigidos pela nova economia, pois não foi construído com essa preocupação[STA 2000]. Os dados que trafegam em uma rede TCP/IP sem técnicas de criptografia podem ser facilmente observados, copiados e até modificados, uma vez que as informações encontram-se em texto puro sobre os pacotes IP. Se um analisador de pacotes for colocado em uma Rede Local padrão Ethernet, na qual se utilizam hubs, todo o tráfego será observado [COM 95]. Ou ainda, se pacotes redirecionadores de rotas forem enviados para roteadores desprotegidos, todo o fluxo de pacotes de um dado canal poderá ser observado [COM 95]. Em aplicações padrão do tipo Telnet, SMTP, POP3 e HTTP, os dados trafegam sem a menor confidencialidade, podendo ser observados pelo analisador [FIE 99][STA 95][STA 2000].

Citando novamente estudos realizados sobre o mercado Internet, mas com foco na questão de segurança da informação que trafega na rede, pode-se citar:

- segurança preocupa 86% dos internautas nos EUA [IDG 2001e][GAR 2001];
- medo de fraude nas compras com cartão de crédito é a principal preocupação para 46% dos internautas [IPS 2001].

Ou seja, a falta de segurança nas transações é um dos fatores inibidores para o crescimento explosivo das compras via Internet pelo grande mercado consumidor. Sendo assim, as empresas ainda deverão investir de forma significativa em segurança, a fim de garantirem o seu lugar neste novo mercado.

Desta maneira, novas aplicações estão sendo desenvolvidas utilizando como base os protocolos TCP/IP, a fim de encriptar os dados sigilosos do acesso aos Bancos, ou de uma sessão de configuração, por exemplo. São exemplos de protocolos criptografados o Secure Sockets Layer (SSL)[FRE 96] e o Transport Layer Security (TLS) [IET 2001c][DIE 99], sobre os quais é construído o HTTPS para acesso a bancos, o Secure Shell (SSH) para sessões de console seguras [IET 2001b], o padrão IPSec para Redes Privativas Virtuais, ou *Virtual Private Network* (VPN)[IET 2001a], entre outros. Todos estes protocolos são melhoramentos sobre a estrutura atual, que garantem a transferência segura de informações.

## *1.3 Segurança da informação*

Diante desta nova realidade, é crescente a necessidade de segurança para o tráfego de informações, sejam elas militares, monetárias, ou ainda particulares. Acima foram citados exemplos de pesquisas feitas em relação à Internet. Entretanto, é claro que os problemas se repetem em outros ambientes. A utilização dos cartões de crédito no seu modo usual, ou seja, nos pontos de venda, também exige segurança na troca de informações, da mesma maneira que transações via Internet.

Neste contexto, a criptografia aparece como uma ferramenta para garantir um ambiente seguro, independente dos meios de transmissão ou ainda do tipo de informação. Para tanto, diferentes métodos e algoritmos de criptografia satisfazem necessidades variadas como forma de garantir não só confidencialidade, mas também autenticação, não-repúdio e integridade dos dados que trafegam via datagramas na Internet [STA 95].

### 1.3.1 Confidencialidade

Como os dados trafegam encriptados fim-a-fim, somente as partes que efetivamente estão se comunicando, e acordaram sobre quais algoritmos e chaves são utilizados, têm acesso às mensagens legíveis. A confidencialidade é então garantida, desde que a técnica de criptografia seja segura.

### 1.3.2 Autenticação

Para garantir que as partes que estão transacionando não estejam blefando em relação às suas identidades, são necessários mecanismos de autenticação. Através da utilização de técnicas apropriadas, é possível utilizar a criptografia para garantir a autenticidade das mensagens.

### 1.3.3 Não-repúdio

Da mesma maneira que a falsa identidade deve ser evitada, também deve-se garantir que se uma mensagem for enviada por alguém, essa mesma pessoa não possa, em outro momento, negar o envio da mensagem em questão. A isso chama-se não-repúdio.

### 1.3.4 Integridade

A integridade visa garantir que os dados enviados cheguem intactos ao destino, não sendo possível modificá-los no caminho. Para tanto, funções de assinatura são utilizadas, provendo os documentos de identificações capazes de comprovar sua originalidade.

## *1.4 Hardware desempenhando funções criptográficas*

A utilização de hardware para desempenhar funções criptográficas, como ferramenta de apoio ao software, pode auxiliar no incremento de desempenho e segurança do sistema. O incremento de performance é obtido pois recursos importantes

de processadores complexos são liberados, uma vez que as funções específicas são processadas por circuitos integrados especialistas ou circuitos lógicos programáveis (*Application Specific Integrated Circuits* – ASICS – ou ainda *Field Programmable Gate Arrays* – FPGA). A melhora do nível de segurança é conseguida através do armazenamento seguro da chave utilizada para encriptar e decriptar os pacotes, uma vez que os Sistemas Operacionais, nos quais as chaves são manipuladas pelas aplicações, possuem vulnerabilidades que podem expor a segurança de todo um sistema [MCC 2001].

O protocolo Secure Sockets Layer (SSL) [FRE 96] tem sido muito utilizado para oferecer segurança em canais originalmente inseguros. O protocolo HTTP, largamente utilizado para acessar conteúdos WWW, ou popularmente páginas Web, não prevê qualquer tipo de criptografia em sua estrutura. Existe suporte para autenticação, mas as credenciais de usuário e senha trafegam em texto puro pela rede [FIE 99]. Desta maneira, faz-se necessário recorrer ao SSL para garantir autenticidade e confidencialidade nas transações. O SSL faz uso de chaves públicas e privadas e pode prover autenticação de ambos os lados da transação, garantindo também o não-repúdio. O mais comum, todavia, é a autenticação acontecer somente no lado do servidor, o qual possui um certificado assinado por alguma entidade certificadora [FAR 2001]. É assim que são feitos os acessos a bancos atualmente, como também transações de comércio eletrônico - *e-commerce*. Quando utilizado sobre SSL, geralmente o protocolo HTTP recebe o nome de HTTPS.

Entretanto, a utilização de criptografia faz com que a performance, por exemplo, de servidores Web caia vertiginosamente [BRA 2000]. Tais servidores possuem alto custo, pois necessitam processar não somente requisições HTTP e HTTPS, mas aplicações dinâmicas que suportam o site, envolvendo acesso ao disco rígido, acesso e consultas a banco de dados, etc. Em Outubro de 1999, um servidor que custava em torno de US$ 36.000,00, com capacidade de processar 2.702 conexões HTTP por segundo, chegou ao máximo de 156 conexões por segundo quando utilizando SSL [CRO 99]. Ou seja, uma queda de quase 18 vezes no desempenho. O mercado oferece opções de dispositivos de hardware e outros de hardware e software, dedicados à tarefa de criptografia [RAI 2001][NCI 2001a].

Plataformas em hardware também são justificáveis em situações nas quais o armazenamento das chaves deve estar em outro dispositivo, que por medida de segurança não esteja em contato com as transações de rede. Utilizam-se equipamentos que recebem dados em texto puro de um servidor e entregam dados criptografados para o mesmo. Tal servidor, por sua vez, interage com a rede de comunicação de dados. Assim, mesmo que o servidor em questão seja de alguma forma comprometido, a chave estará segura. Tais equipamentos podem ter inclusive proteção contra violações físicas que desejem acessar diretamente os circuitos eletrônicos [NCI 2001b][BAL 2001].

Em virtude do espaço de destaque que a criptografia em hardware ocupa nas necessidades de mercado atuais e futuras, decidiu-se pesquisar sobre o assunto. Por isso elaborou-se um Trabalho Individual nesta universidade sobre as possibilidades e viabilidades de implementação de criptografia em hardware, sob o título: Estudo de Implementações de Algoritmos de Criptografia em VHDL. Na oportunidade, concluiu-se que as criptografias simétricas seriam as mais adequadas, pois não apresentam uma grande complexidade, bem como exigem desempenhos muitas vezes não encontrados em processadores mais complexos. Ou ainda, consomem recursos caros em sistemas já encarregados de executar outras funções. O Capítulo 2 desta dissertação aborda conceitos básicos de criptografia, permitindo ao leitor entender a diferença entre a criptografia simétrica e a assimétrica, por exemplo.

Entretanto, a confecção de circuitos integrados específicos (*Application Specific Integrated Circuit* – ASIC) nem sempre é possível, ou muitas vezes não é justificável em termos de relação custo x benefício. Desta maneira, o uso de componentes reconfiguráveis, como por exemplo *Programmable Logic Devices* (PLD) ou *Field Programmable Gate Array* (FPGA), configura-se como uma alternativa viável para garantir uma performance superior ao software, sem contudo dispender os recursos necessários para gerar um ASIC.

Pode-se destacar algumas vantagens inerentes à utilização de FPGA:

- facilidade de desenvolvimento;
- rapidez na prototipação e testes, a fim de garantir um resultado livre de erros e com melhor desempenho;
- possibilidade de modificação do projeto, caso alguns de seus componentes necessitem atualização devido a mudanças nos algoritmos;
- facilidade de reconfiguração dos componentes para desempenhar funções diferentes, ou ainda implementar variações do algoritmo; essa é uma questão específica dos hardwares de criptografia, pois pode-se utilizar diferentes chaves, pode-se necessitar criptografar ou decriptografar, etc;
- performance superior às implementações de software sobre plataformas e sistemas operacionais convencionais.

## 1.5 Advanced Encryption Standard (AES)

Em virtude dos requerimentos atuais de segurança e performance, em relação aos algoritmos criptográficos utilizados pelas aplicações correntes e futuras, padrões anteriormente válidos passaram a apresentar deficiências. É o caso do padrão Data Encryption Standard (DES) [NIS 99], citado no Capítulo 2, o qual foi o algoritmo simétrico adotado oficialmente pelo governo norte-americano em meados da década de 70 [STA 2000][SCH 96]. Durante mais de duas décadas tal padrão foi suficiente para garantir a segurança da maioria das aplicações comerciais. Entretanto, em 1998, mensagens criptografadas com o DES foram decifradas em menos de 3 dias por um projeto feito pela EFF – *Electronic Frontier Foundation* , o qual custou menos do que US$ 250.000,00 [EFF 98]. Seis meses depois, o tempo necessário para decifrar caiu para 22 horas e 15 minutos utilizando aproximadamente 100.000 computadores normais ligados em rede. Ou seja, a criptografia DES, utilizando chaves de 56 bits, não oferecia mais a segurança necessária para as aplicações comerciais [SIL 2001].

Uma variação do DES, na qual aplica-se três (3) vezes a transformação definida no mesmo, foi chamada de 3DES, e adotada também como padrão pelo governo norte-americano através de uma atualização do padrão DES antigo [NIS 99]. Entretanto, o 3DES apresenta problemas de performance [SCH 96][DAI 2000], pois sua arquitetura original data de meados dos anos 70.

Novos algoritmos foram criados a fim de incrementar tanto a segurança quanto o desempenho em relação ao DES e 3DES. É o caso dos algoritmos IDEA, Blowfish, MARS, entre outros [SCH 96][STA 95]. Tais algoritmos são citados também no Capítulo 2.

Entretanto, o governo norte americano necessitava rever o seu padrão de criptografia oficial, a fim de garantir segurança e performance às aplicações [NIS 2001b]. Este processo foi iniciado em 02 de Janeiro de 1997 [NIS 2001d]. Em 12 de Setembro de 1997 o *National Institute of Standards and Technology* (NIST) fez uma

chamada formal, iniciando um processo mundial no qual qualquer pesquisador poderia participar apresentando algoritmos, no intuito de definir o novo padrão de criptografia simétrica norte-americano, chamado Advanced Encryption Standard (AES). O novo padrão deve necessariamente ser utilizado pelo governo norte-americano e voluntariamente por toda a sociedade em geral. O padrão escolhido seria disponibilizado publicamente, livre de quaisquer licenças de utilização e disponível mundialmente [NIS 2001d].

O novo algoritmo utilizaria criptografia simétrica, operando sobre blocos de 128 bits, e utilizando chaves de 128, 192 e 256 bits, sendo muitas vezes mais seguro que o DES, que criptografava blocos de 64 bits utilizando chaves de 56 bits. Uma chave de 128 bits produz aproximadamente $3{,}4*10^{38}$ diferentes possibilidades de combinação.

Em 20 de Agosto de 1998, vários algoritmos foram apresentados inicialmente, reunindo pesquisadores de 12 países diferentes. Quinze (15) algoritmos foram selecionados para o *Round 1* [NIS 2001c]. Depois, cinco (5) foram escolhidos para o *Round 2*, no qual o NIST intensificou a análise de cada proposta, efetivamente incentivando o "ataque" sobre todos os competidores [NIS 2000a]. Os algoritmos selecionados para o *Round 2* foram:

- MARS [IBM 2000]
- RC6 [RSA 2000]
- RIJNDAEL [DAE 2001]
- SERPENT [AND 98]
- TWOFISH [COU 2001]

Ao final, concluiu-se que em termos de segurança os cinco concorrentes finais apresentavam características semelhantes. Em 2 de outubro de 2000, o NIST anunciou o algoritmo Rijndael como o vencedor do concurso, em função da melhor combinação entre segurança, desempenho, eficiência, implementabilidade e flexibilidade [NIS 2000b]. No Capítulo 3 serão detalhadas as características construtivas do Rijndael.

Após a apresentação do Rijndael como padrão AES, é salutar voltar à questão da performance de hardware em comparação com software. A seguir é apresentada uma tabela mostrando o autor, a velocidade atingida e a plataforma testada, tanto em software quanto em hardware.

TABELA 1.2 - Implementação do AES-128 em diferentes plataformas

| Autor | Taxa de transmissão Encriptação | Taxa de transmissão Decriptação | Plataforma |
|---|---|---|---|
| [GLA 99] | 68,4 Mbps | 72,7 Mbps | Código C otimizado, sobre um Pentium Pro 200 MHz. |
| [DAI 2000] | 254 Mbps* | 254 Mbps* | Código C otimizado, sobre um Celeron 850 MHz. |
| [MRO 2000] | 268 Mbps** | 248 Mbps** | Código VHDL sobre FPGA Altera da Família Flex10K. |
| [LOM 2000] | 271 Mbps | 272 Mbps | Código C otimizado, sobre um Pentium III 800 MHz. |
| [GAJ 2000b] | 414 Mbps** | 414 Mbps** | Código VHDL sobre FPGA Xilinx da família Virtex. |
| [NSA 2000] | 605 Mbps** | 605 Mbps** | Código VHDL sobre ASIC de tecnologia 0,5 μm. |

* Os desempenhos separados da encriptação e decriptação não foram fornecidos.
** Desempenho na arquitetura iterativa.

Ou seja, mesmo quando processadores de última geração são utilizados, o desempenho de dispositivos de hardware dedicados é superior.

## *1.6 Representação numérica*

Ao longo do trabalho serão referenciados valores numéricos que seguirão o padrão mostrado abaixo.

TABELA 1.3 – Representação numérica utilizada na dissertação

| Representação | Descrição |
|---|---|
| #10100101b | valor em código binário correspondente ao número decimal 165 |
| #A5h | valor em código hexadecimal correspondente ao número decimal 165 |
| #165d | valor 165 em código decimal |

# 2 Conceitos básicos de criptografia

## 2.1 Histórico

**Criptografia**. S.f. 1. Arte de escrever em cifra ou em código. 2. Conjunto de técnicas que permitem criptografar escritas.[FER 9?]

**Criptografar.** V. t. d. Tornar incompreensível, com observância de normas especiais consignadas numa cifra ou num código, o texto de (uma mensagem escrita com clareza).[FER 9?]

O desejo de tornar incompreensíveis mensagens que possuem caráter sigiloso não surgiu recentemente, tampouco as tentativas de quebrar tais códigos secretos. A arte da criptografia acumula muitas histórias ao longo do seu desenvolvimento, desde a Antiguidade até os dias de hoje. Inúmeras foram as suas aplicações para as mais diversas finalidades.

Durante a Segunda Guerra, por exemplo, os alemães utilizaram um equipamento mecânico chamado Enigma para criptografar suas mensagens. Ao longo do período de conflitos, o algoritmo utilizado foi descoberto, forçando a modificação dos aparatos que codificavam e decodificavam as mensagens [SCH 96][SIN 99]. A bibliografia é farta ao relatar detalhes interessantes das proezas e dificuldades encontradas por ambos os adversários, que ocorria não somente nos campos de batalha.

Mais recentemente, com o emprego intensivo de recursos computacionais, utilizam-se algoritmos criptográficos para tarefas corriqueiras na sociedade moderna. Um simples acesso ao *Internet Banking* utiliza tanto criptografia simétrica quanto assimétrica. O armazenamento das senhas de acesso aos sistemas operacionais mais evoluídos, como é o caso dos vários sistemas UNIX, Linux, Windows NT e 2000, Netware, entre outros, utiliza algoritmos de *Hash* [MCC 2001], os quais fazem parte do universo da criptografia. Tais conceitos serão explicitados a seguir neste Capítulo.

## 2.2 O nível de segurança oferecido pelas técnicas de criptografia

As técnicas de criptografia competem diretamente com as tentativas de ataques disponíveis atualmente. Imagine o que aconteceria se o algoritmo criptográfico utilizado para transferir um grande segredo industrial for quebrado, divulgando os dados sigilosos pertinentes à conexão. Desta maneira, é importante uma avaliação constante do nível de segurança oferecido pela criptografia utilizada, a fim de identificar se os conceitos de confidencialidade, autenticidade, não-repúdio e integridade estão sendo satisfeitos.

Em geral, as técnicas de criptografia constituem-se de um algoritmo e de uma chave. Aceita-se amplamente o conceito de que o algoritmo deve ser de domínio público, de maneira que toda a comunidade científica tenha acesso e comprove sua segurança [SCH 96]. Nunca deve ser escolhido um algoritmo cuja confiabilidade reside no desconhecimento das técnicas criptográficas empregadas. A isso chama-se segurança por obscuridade, e ocasiona uma falsa sensação de segurança.

Por outro lado, não é possível afirmar que um determinado algoritmo é 100% seguro, mesmo que várias técnicas de verificação formal sejam empregadas [SCH 96]. O que se sabe é que até o presente momento ele ainda não foi quebrado.

Então, toda a confidencialidade de uma transmissão encriptada reside na chave utilizada, desde que o algoritmo utilizado seja amplamente conhecido e aceito como seguro. Tal chave deve ser mantida em sigilo, e em muitos casos trocada regularmente. Conforme será mostrado nos tópicos seguintes, em algoritmos de chave simétrica, a chave em geral é trocada a cada nova sessão, de tempos em tempos ou de acordo com o volume de tráfego transmitido. Nos algoritmos de criptografia assimétrica, a chave privada é utilizada por períodos maiores de tempo, podendo inclusive não ser nunca trocada.

Assim, se um atacante possuir somente as mensagens cifradas, mesmo com o conhecimento completo do algoritmo, mas não tiver acesso à chave, não será possível decifrar a mensagem. Por isso, a tentativa de quebra da chave pela técnica da força bruta, em algoritmos confiáveis, ainda é a alternativa mais fácil para se quebrar a confidencialidade de uma mensagem criptografada com um algoritmo seguro [SCH 96]. A força bruta experimenta todo o universo de chaves possíveis de serem utilizadas para decifrar as mensagens. Os textos produzidos pelas diversas chaves criadas devem então ser analisados à procura de padrões reconhecíveis de escrita, ou expressões específicas contidas na mensagem original.

Desta maneira, à medida que os recursos computacionais disponíveis para cifrar e decifrar mensagens evoluem, as chaves devem necessariamente incrementar o tamanho, a fim de aumentar a quantidade de combinações diferentes a serem utilizadas.

A grande questão é escolher o tamanho adequado da chave, levando em consideração a criticidade das informações e os recursos computacionais disponíveis. Existe, para cada caso particular, uma curva Custo X Benefício própria, que deve ser utilizada para determinar o nível de segurança mais satisfatório. Deve-se prever também o princípio de que os recursos computacionais evoluem significativamente com o tempo, aumentando a performance e barateando os equipamentos. Assim, é sempre necessário projetar os níveis atuais levando em consideração o período futuro no qual deseja-se preservar a segurança da informação.

## *2.3 A criptoanálise*

A criptoanálise é a arte de recuperar mensagens decifradas sem o conhecimento da chave [SCH 96]. Algumas vezes é possível decifrar os textos cifrados e também descobrir a chave utilizada.

Existem diversos tipos de ataque utilizados na criptoanálise, de acordo com a pretensão e a quantidade de informação disponível ao atacante [SCH 96]:

- **Acesso restrito ao texto cifrado**: o criptoanalista possui somente uma grande quantidade de mensagens cifradas com um mesmo algoritmo. O objetivo é decifrar algumas mensagens ou ainda descobrir a chave.

- **Acesso ao texto original e ao cifrado**: o criptoanalista possui mensagens originais e cifradas, mas não possui acesso à chave. O desafio é descobrir a chave – ou as chaves – utilizada, ou ainda um algoritmo capaz de decifrar as mensagens cifradas com aquela chave – ou chaves.

- **Escolha do texto original**: o criptoanalista não só possui acesso aos textos originais e aos criptografados, mas pode também escolher quais textos serão cifrados, permitindo a utilização de mensagens que ressaltem as características da chave. O

objetivo é o mesmo do ataque anterior, ou seja, descobrir a chave usada na comunicação.

- **Escolha adaptativa do texto original**: caso especial da escolha do texto cifrado, no qual o criptoanalista possui um controle granular das mensagens cifradas, de maneira a modificá-las de acordo com o resultado obtido.

- **Escolha do texto cifrado**: o criptoanalista pode escolher quais textos cifrados serão decifrados. O objetivo é deduzir a chave.

- **Escolha da chave**: o criptoanalista não pode escolher a chave, mas ele conhece algumas relações entre diferentes chaves. Esse ataque é obscuro e não muito prático.

- **Compra ou roubo da chave**: não se pode deixar de considerar a hipótese da chave ser revelada de alguma maneira trivial para o atacante, tanto via a compra, o roubo ou até a utilização de técnicas violentas.

## *2.4 Criptografia como funções matemáticas*

### 2.4.1 Definições:

| | |
|---|---|
| **M →** | mensagem original, compreensível; |
| **C →** | mensagem codificada, cifrada; |
| **E →** | função responsável por codificar a mensagem original, tornando-a cifrada; |
| **D →** | função responsável pela decodificação da mensagem cifrada, revelando a mensagem original. |

Um algoritmo criptográfico pode ser representado por duas funções – E e D – cujas responsabilidades são cifrar e decifrar, respectivamente, uma mensagem. A função E é uma função matemática capaz de, a partir de uma mensagem M, gerar outra mensagem C. O mesmo algoritmo deve prover a função inversa D, na qual a partir da mensagem C seja possível reconstituir a mensagem M. Em muitos algoritmos, as funções E e D são iguais. Então:

$$E(M) = C \qquad (1)$$

$$D(C) = M \qquad (2)$$

Assim,

$$D(E(M)) = M \qquad (3)$$

Quando E e D são iguais, tem-se ainda que

$$D(E(M)) = E(D(M)) \quad (4)$$

Entretanto, a forma descrita acima cria uma dependência entre a confidencialidade da mensagem cifrada e o conhecimento das funções E e D. Se uma mensagem M foi cifrada pelo algoritmo E, basta conhecer o algoritmo D para que a mensagem cifrada C seja transformada novamente em M.

Conforme descrito anteriormente, os algoritmos devem ser de conhecimento público para que sejam efetivamente testados e aprovados. Assim, atualmente são utilizados conjuntos de bits, chamados de chaves criptográficas, a partir dos quais os algoritmos são parametrizados. As transformações efetuadas dependem do valor da chave, que em conjunto com a mensagem M, gera uma mensagem cifrada C. O sigilo reside na segurança da chave utilizada, uma vez que o algoritmo é público [SCH 96][STA 2000].

$$E_{k1}(M) = C \quad (5)$$

$$D_{k2}(C) = M \quad (6)$$

De acordo com a técnica utilizada, as chaves k1 e k2 podem ser diferentes ou não.

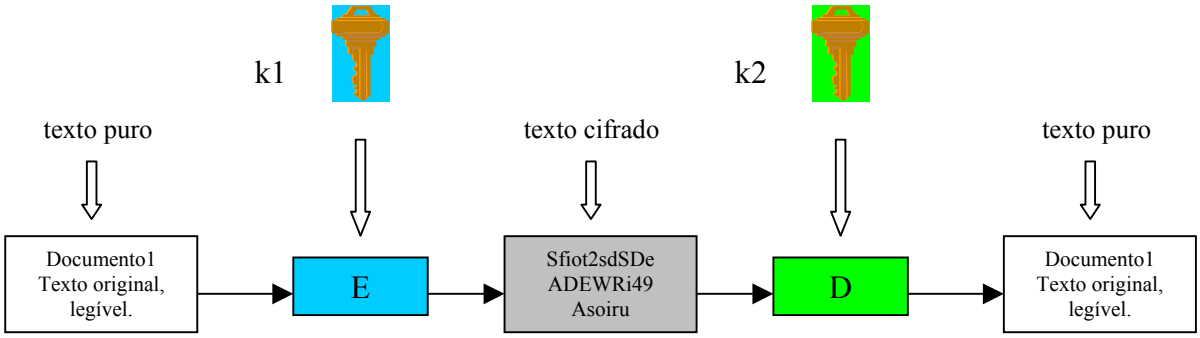


FIGURA 2.1 - Criptografia utilizando chaves

## 2.5 Algoritmos simétricos

Os algoritmos simétricos são aqueles nos quais as chaves k1 e k2 podem ser obtidas uma a partir da outra. No caso mais simples e mais utilizado, as chaves k1 e k2 são iguais, resultando em:

$$E_k(M) = C \quad (7)$$

$$D_k(C) = M \quad (8)$$

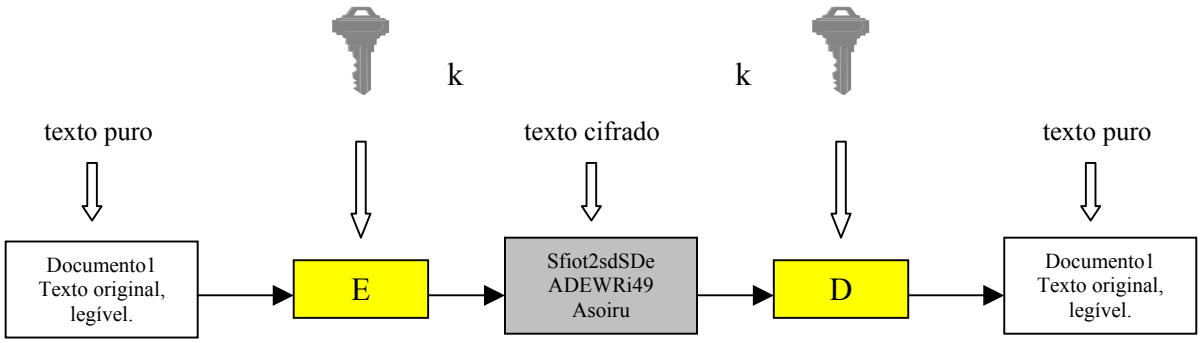


FIGURA 2.2 - Algoritmo Simétrico com Chaves Iguais

Uma vez que as funções E e D são conhecidas, a segurança da criptografia reside na confidencialidade da chave escolhida. As chaves simétricas mais utilizadas variam desde 40 bits até 256 bits nas implementações comerciais.

TABELA 2.1 – Algoritmos de criptografia simétrica

| **Tamanho da chave** | **Algoritmo** | **Descrição** |
|---|---|---|
| 40 bits | FWZ | Proprietário da fabricante de ferramentas de segurança Checkpoint. |
| 56 bits | DES | *Data Encription Standard*, adotado pelo governo dos EUA. |
| 112 bits | 3DES | 3DES com chaves duplas. |
| 168 bits | 3DES | 3DES com chaves triplas. |
| 128 bits | IDEA | *International Data Encription Algorithm*. |
| 128 bits | AES128 | Rijndael com 128 bits de chave. |
| 192 bits | AES192 | Rijndael com 192 bits de chave. |
| 256 bits | AES256 | Rijndael com 256 bits de chave. |

Os algoritmos simétricos são em geral utilizados para transmissões que requerem altas taxas, e por isso as chaves e os algoritmos são projetados para oferecerem alto desempenho. Assim, antes de qualquer transmissão encriptada simetricamente, uma fase de geração e troca de chaves é necessária. Para cada nova conexão, uma nova chave pode ser negociada e trocada. Mesmo durante uma mesma conexão, a chave deve ser trocada dentro de intervalos convenientes, a fim de preservar a confiabilidade da conexão.

Tal necessidade deve-se ao fato de que a maioria dos algoritmos de criptografia permite que uma análise feita a partir de uma grande quantidade de mensagens cifradas revele um determinado padrão. Se esse padrão for relacionado com um tipo de tráfego conhecido, no qual o texto puro pode ser deduzido, existe a possibilidade de que a chave simétrica seja quebrada. A quebra da chave consiste em gerar todas as chaves possíveis e criptografar as mensagens deduzidas à procura dos padrões encontrados nas mensagens cifradas. Se os padrões forem encontrados e o restante das mensagens cifradas for decodificado gerando mensagens legíveis, então a chave foi encontrada.

Esse ataque é chamado de Escolha do texto original, ou *choosen-plaintext attack* [SCH 96], e já foi citado anteriormente.

Desta maneira, o tempo escolhido para a troca de chaves simétricas deve ser sempre menor que o tempo estimado para que a chave seja quebrada. É claro que esse tempo depende dos recursos computacionais disponíveis para quebrar a chave e da informação a ser transmitida.

São exemplos de Algoritmos Simétricos:

- **Data Encryption Standard (DES)**: em meados dos anos 70, o *National Institute of Standards and Technology* (NIST), então chamado de *National Bureau of Standards* (NBS), aprovou como padrão do governo norte-americano um algoritmo inicialmente desenvolvido pela IBM™, e modificado pela *National Security Agency* (NSA). A modificação foi e é motivo de polêmica, uma vez que o tamanho da chave inicialmente proposto de 128 bits foi reduzido para 56 bits [SCH 96]. Apesar da chave DES ser de 56 bits, e do algoritmo possuir aproximadamente 25 anos, ele continua sendo muito utilizado e adequado para a maioria das aplicações comerciais. O DES atualmente é definido no padrão norte-americano FIPS 46-3 [NIS 99], que incorpora também o Triple-DES, descrito abaixo.

- **Triple DES (3DES) ou Triple Data Encryption Algorithm (TDEA)**: a fim de aumentar o tamanho da chave utilizada, foi proposta a utilização em cascata do DES. O padrão divulgado em 1999 *pelo National Institute of Standards and Technology* (NIST), denominado FIPS 46-3, adota a implementação do 3DES segundo a norma do *American National Standards Institute* (ANSI) X9.52. O 3DES é então denominado *Triple Data Encryption Algorithm* – TDEA [NIS 99]. O 3DES consiste em três fases, que são uma encriptação, uma decriptação e outra encriptação, utilizando três chaves k1, k2 e k3 [NIS 99]. Assim,

$$C = E_{k3}(D_{k2}(E_{k1}(M))) \qquad (9)$$

São definidas três opções de utilização das chaves:
  - k1, k2 e k3 são independentes;
  - k1 e k2 são independentes e k3 é igual a k1;
  - k1, k2 e k3 são iguais.

Quando k1=k2=k3, é mantida a compatibilidade com o DES, pois resulta:

$$C = E_k(D_k(E_k(M))) \Rightarrow C = E_k(M) \qquad (10)$$

Existem implementações comerciais do 3DES que utilizam todas as opções possíveis de chaves. Desta forma, o 3DES possui chaves de 56, 112 e 168 bits.

- **International Data Encryption Algorithm (IDEA):** algoritmo bastante utilizado, em virture da sua adoção como um dos padrões utilizados pelo *Pretty Good Privacy* (PGP) [PGP 2001]. Foi inicialmente desenvolvido por Kuejia Lai e James Massey em 1990. A versão final, batizada de IDEA, foi apresentada em 1992 [STA 95]. O algoritmo utiliza operações de Ou-Exclusivo (XOR), adição em módulo 2^16 e multiplicação em módulo 2^16+1. A chave é de 128 bits [SCH 96].

- **RC5:** algoritmo com chave variável desenvolvido por Ron Rivest, da empresa RSA. É baseado em operações de Ou-Exclusivo, adições e rotações [SCH 96].

- **CAST**: desenvolvido no Canadá por Carlisle Adams e Stafford Tavares. O algoritmo é construído sobre o conceito de S-boxes, que são caixas de seleção, dependentes de cada aplicação. O tamanho das chaves é variável, desde 40 bits até 256. A versão de 40 bits até 128 bits, permitindo incrementos de 8 em 8 bits, está descrita no *Request for Comments* 2144 (RFC2144) [ADA 97], e é livre de *Royalties*. A versão de 256 bits competiu na escolha do padrão AES, e é definida na RFC 2612 [ADA 99], sendo também livre de *Royalties*. É um dos algoritmos utilizados no PGP [PGP 2001].

- **AES**: Em 12 de Setembro de 1997 o *National Institute of Standards and Technology* (NIST) requisitou propostas para o que foi chamado *Advanced Encription Standard*. Inúmeros algoritmos foram apresentados inicialmente, reunindo pesquisadores de 12 países diferentes. Quinze (15) algoritmos foram selecionados para o Round 1 [NIS 2001c]. Depois, cinco (5) foram escolhidos para o Round 2, no qual o NIST intensificou a análise de cada proposta, efetivamente incentivando o "ataque" sobre todos os competidores [NIS 2000a]. Os algoritmos selecionados para o Round 2 foram:

  - MARS [IBM 2000]
  - RC6 [RSA 2000]
  - RIJNDAEL [DAE 2001]
  - SERPENT [AND 98]
  - TWOFISH [COU 2001]

  Ao final, concluiu-se que em termos de segurança os cinco concorrentes finais apresentavam características semelhantes. Em 2 de outubro de 2000 o NIST anunciou o algoritmo Rijndael como o vencedor do concurso, em função da melhor combinação entre segurança, desempenho, eficiência, implementabilidade e flexibilidade [NIS 2000b]. No capítulo seguinte serão detalhadas as características construtivas do Rijndael.

## 2.6 Algoritmos assimétricos

Apesar do seu histórico de desenvolvimento, que remonta da Antiguidade, a criptografia sempre baseou-se em um segredo conhecido por ambos os mensageiros autorizados. De posse do segredo era possível decifrar a mensagem secreta. Desta maneira, durante milhares de anos, o que se utilizou foram efetivamente algoritmos simétricos. No início, o segredo era o próprio algoritmo, enquanto que mais recentemente passou-se a utilizar chaves.

A grande inovação conceitual do ponto de vista da criptografia ocorreu em 1976 [SCH 96], quando Diffie e Hellman publicaram os conceitos da criptografia baseada em chaves públicas [DIF 76]. O postulado mencionava a utilização de duas chaves diferentes, uma para cifrar e a outra para decifrar a mensagem. Desde que seja praticamente inviável descobrir uma das chaves a partir da outra, pode-se distribuir livremente uma delas, chamada então de chave pública. A outra chave deve permanecer

em sigilo, por isso chamada de chave privada. Tais algoritmos utilizam duas chaves diferentes para a encriptação e decriptação[2], e são então chamados de Assimétricos.

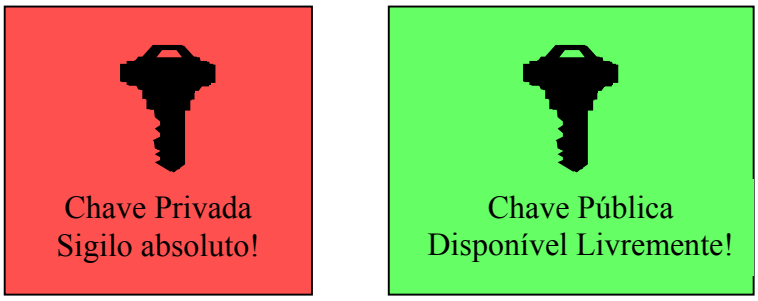


FIGURA 2.3 - Chaves Pública e Privada

Diffie e Hellman definiram condições que deveriam ser seguidas de maneira a implementar um algoritmo de chave pública [STA 95]:

1. é computacionalmente fácil para um mensageiro B gerar um par KUb e KRb de chaves pública e privada, respectivamente;
2. é computacionalmente fácil para um mensageiro A, sabendo a chave pública do mensageiro B, nesse caso a chave KUb, cifrar com a mesma uma mensagem M, gerando um texto cifrado C;

$$C = E_{KUb}(M) \qquad (11)$$

3. é computacionalmente fácil para o mensageiro B decifrar o texto cifrado C utilizando a sua chave privada KRb;

$$M = D_{KRb}(C) \qquad (12)$$

4. é computacionalmente inviável descobrir a chave privada KRb do mensageiro B a partir da sua chave pública KUb;
5. é computacionalmente inviável descobrir a mensagem M a partir do texto cifrado C e da chave pública KUb.

É possível acrescentar-se um outro requerimento, que apesar de útil, não é necessário para todas as implementações de chave pública:

6. as funções de encriptação e decriptação podem ser aplicadas em qualquer ordem.

$$M = E_{KUb}(D_{KRb}(M)) \qquad (13)$$

Através da técnica de chave pública é possível tanto criptografar quanto assinar uma mensagem. Pode-se ainda assinar e também criptografar a mensagem.

Para mandar uma mensagem criptografada para o mensageiro B, basta cifrar a mensagem usando a chave pública de B. Considera-se que a chave privada de B esteja

[2] Existem no Novo Dicionário Aurélio [FER9?] as expressões encriptar, decriptar e decriptação. Assim, utiliza-se encriptação com o mesmo sentido do ato de encriptar.

bem armazenada, de maneira que o único usuário capaz de decifrar a mensagem é B, pois B possui a chave privada que corresponde à chave pública usada para cifrar.

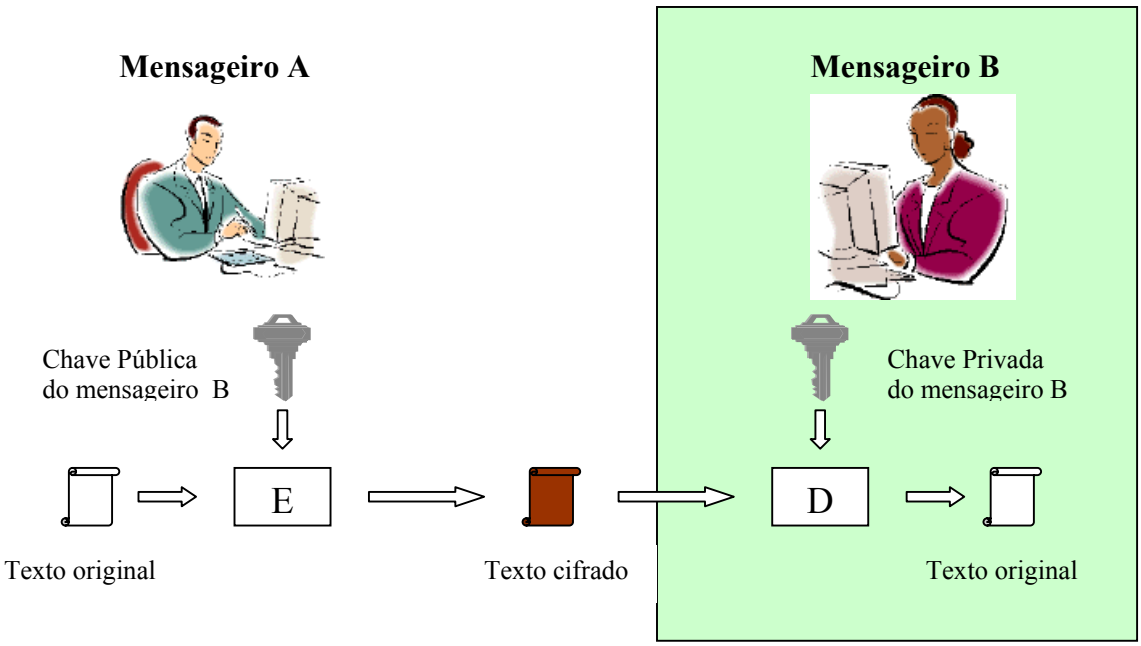


FIGURA 2.4 - Enviando uma mensagem cifrada para o mensageiro B

Analogamente, se o mensageiro A criptografar uma mensagem com a sua chave privada, todos os interessados poderão decifrar a mensagem a partir da chave pública de A. A *assinou* a mensagem, pois somente o portador da chave privada correspondente à chave pública de A poderia produzir uma mensagem corretamente decifrada por tal chave pública.

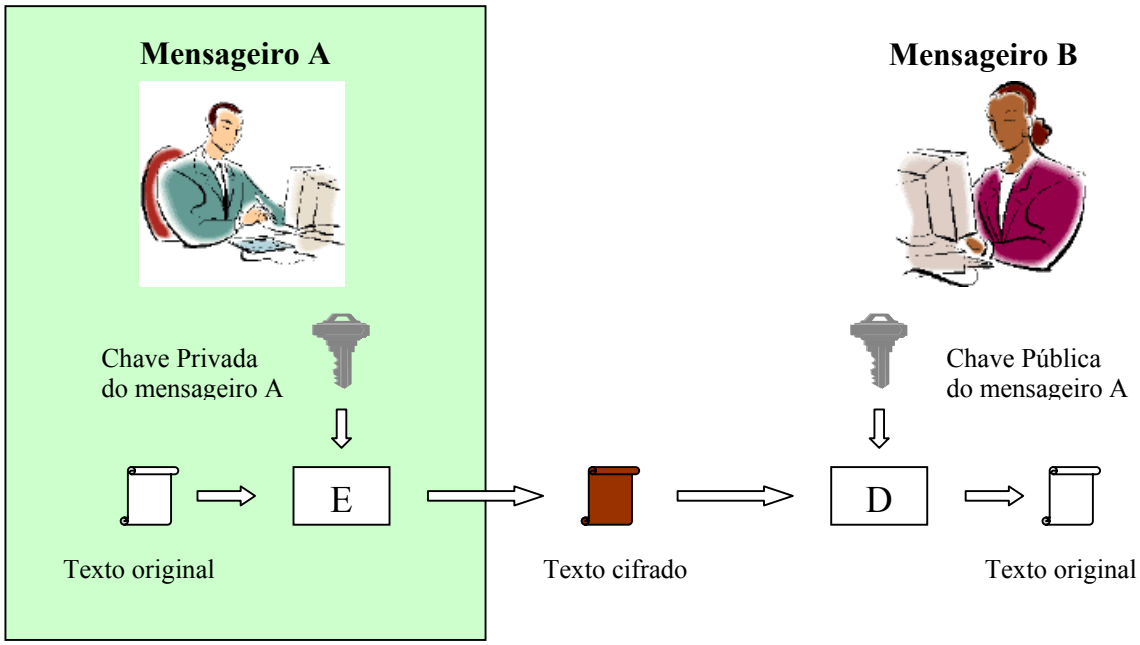


FIGURA 2.5 - Enviando uma mensagem assinada para o mensageiro B

Diversas combinações de encriptação e assinatura são possíveis e amplamente utilizadas. É muito comum que uma mensagem seja encriptada e assinada, garantindo a confidencialidade e autenticidade do documento.

Entretanto, os algoritmos de chave pública e privada em geral trabalham com números grandiosos, de maneira que o desempenho da encriptação e decriptação é lento, consumindo recursos computacionais consideráveis. O tamanho das chaves variam desde 768 bits até 4096 bits [SCH 96]. Em geral utilizam-se chaves de 1024 ou 2048 bits. Por isso, na maioria das vezes os algoritmos de chave pública e privada são utilizados para estabelecer a conexão inicial, de maneira a trocar com segurança as chaves simétricas que serão utilizadas no restante da conexão.

São exemplos de Algoritmos Assimétricos:

- **RSA**: o primeiro algoritmo de chave pública e privada capaz de encriptar e assinar digitalmente uma mensagem, observando todos os requisitos preconizados por Diffie e Hellman [SCH 96]. É o algoritmo de mais fácil compreensão e implementação, e por isso o mais popular. A denominação RSA é advinda dos seus criadores – Ron Rivest, Adi Shamir e Leonard Adleman. Desde a sua públicação em 1978 [STA 95], o algoritmo RSA tem resistido à toda forma de criptoanálise, demonstrando enorme robustez. A chave é variável, sendo que a faixa comumente encontrada vai de 1024 até 3072 bits. A patente norte-americana sobre o algoritmo terminou em Setembro de 2000, liberando quaisquer implementações do pagamento de *Royalties* à RSA, Inc.

- **ElGamal**: capaz de realizar encriptação e assinatura digital. Foi o primeiro algoritmo de chave pública e privada e ter a patente liberada, desde 1997 [SCH 96].

- **DSA**: o *Digital Signature Algorithm* (DSA) foi proposto pelo NIST como o algoritmo utilizado pelo *Digital Signature Standard* (DSS). Desenvolvido pela NSA, o DSA seria livre das patentes requeridas pelo RSA. Originalmente, o DSA suporta somente assinatura digital, entretanto, é possível utilizá-lo para implementar a encriptação de mensagens através dos algoritmos RSA e ElGamal. Quando de sua criação, um grande movimento proveniente de grandes companhias norte-americanas, incluindo a RSA, contestou a adoção do algoritmo DSA no padrão DSS ao invés do RSA [SCH 96].

## *2.7 Funções hash*

Uma função mão única, ou função *hash*, recebe uma entrada de tamanho variável e retorna uma saída de tamanho fixo, chamada *hash*. Muitas funções satisfazem à condição de receber uma entrada variável e oferecer um resultado de tamanho constante. Entretanto, uma função *hash* deve apresentar características adicionais. O ideal é que o *hash* de cada mensagem seja único, de maneira que identifique inequivocamente o conteúdo, servindo analogamente a uma impressão digital da mensagem.

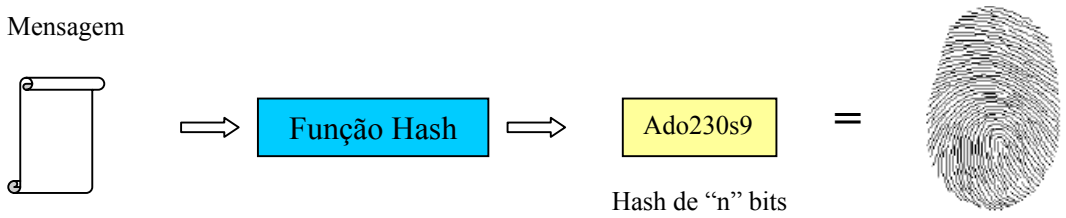


FIGURA 2.6 - O *hash* de uma mensagem equivale à impressão digital do seu conteúdo

As funções *hash* são rápidas e por isso muito utilizadas em assinatura digital. Analogamente à encriptação, aplicar um algoritmo assimétrico a um texto pode demorar muito tempo, ao passo que assinar um *hash* da mensagem economiza processamento. Assim, na maioria das vezes, quando efetivamente se assina um texto, na realidade faz-se a assinatura do *hash* da mensagem.

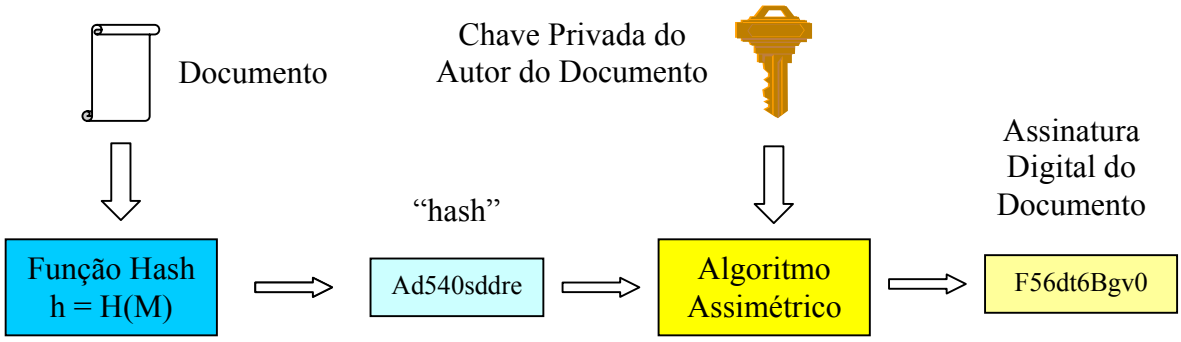


FIGURA 2.7 - Método de Assinatura Digital de documentos utilizando Função Hash

Assim, uma função *hash* com resultado h, na forma

$$h = H(M) \qquad (14)$$

deve apresentar as seguintes propriedades [STA 95]:

1. a função H pode ser aplicada a um bloco de dados M de qualquer tamanho;
2. o resultado h deve ser de tamanho fixo;
3. dado M, é computacionalmente fácil processar a função H(M) e encontrar o resultado h;
4. dado um *hash* h, é computacionalmente improvável que o valor M correspondente seja encontrado de modo que H(M)=h;
5. dada uma mensagem M, é improvável encontrar outra mensagem N diferente de M, cujos resultados da função de *hash* sejam iguais, ou seja, H(M)=H(N);
6. deve ser improvável encontrar um par de mensagens P e Q, que diferem uma da outra, e que produzam o mesmo *hash*, ou seja H(P)=H(Q).

As primeiras três propriedades listadas acima são requisitos práticos de funcionamento de uma função *hash*. Na verdade, a primeira condição muitas vezes não é satisfeita, pois alguns algoritmos trabalham sobre mensagens de tamanhos variáveis,

mas menores que um determinado tamanho. É o caso do algoritmo SHA-1, visto a seguir.

A quarta propriedade refere-se ao caráter de mão única que deve conter uma função *hash*. É fácil a partir de uma mensagem produzir o seu *hash*, mas deve ser praticamente impossível, a partir do *hash*, reproduzir a mensagem original. A quinta propriedade visa garantir que um determinado documento, que produziu um *hash* h, não seja forjado, uma vez que é computacionalmente muito difícil encontrar um documento diferente do original que produza o mesmo *hash* h.

Mesmo que uma função *hash* satistaça todas as 5 primeiras condições, ela somente será considerada forte se a sexta for também aceita. Existe uma diferença sutil entre a quinta e a sexta condições devido ao chamado Ataque do Aniversário. A teoria da probabilidade mostra que dada uma pessoa M, a chance de se encontrar outra pessoa N cuja data do aniversário seja a mesma é X. Entretanto, a probabilidade de que duas pessoas com a mesma data de aniversário sejam encontradas é muito maior do que X [SCH 96]. Ou seja, é mais fácil encontrar previamente duas mensagens M e N que produzam o mesmo *hash*, de maneira a poder escolher qual mensagem utilizar em função das circunstâncias, do que a partir de uma mensagem M procurar uma outra mensagem N que satisfaça a condição de apresentar um *hash* idêntico. É por isso que existe a sexta propriedade.

Não é simples construir uma função *hash*. Na prática, utilizam-se funções de compressão que aceitam entradas de tamanho *i*, produzindo saídas de tamanho *o*, sendo *o* menor do que *i* [SCH 96]. Para computar o *hash* é necessário dividir a mensagem M em vários blocos de tamanho i. A função de compressão recebe como entradas não somente os blocos de tamanho i da mensagem M, mas também o resultado da operação anterior. Temos então a seguinte iteração:

$$h_i = f(M_i, h_{i-1}) \qquad (15)$$

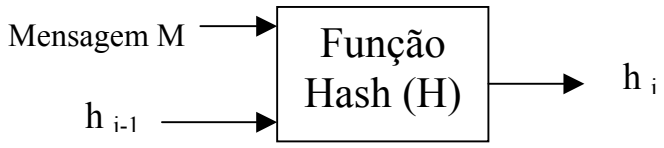


FIGURA 2.8 - Iteração da função *hash*

Inúmeras funções de *hash* são conhecidas, mas as mais utilizadas atualmente são a MD5, que produz uma saída de 128 bits, e a SHA, cuja saída tem 160 bits.

- **MD5**: o Message Digest 5 (RFC 1321) foi desenvolvido por Ron Rivest, o mesmo do RSA, e é um melhoramento do algoritmo MD4, do mesmo autor. O MD5 recebe blocos de 512 bits, tendo como saída um *hash* de 128 bits [SCH 96].

- **SHA-1**: desenvolvido pelo NIST e NSA para ser utilizado no Digital Signature Standard (DSS), o SHA-1 aceita mensagens de até 2^64 bits, trabalhando sobre blocos de 512 bits e produzindo uma saída de 160 bits. O SHA-1 baseou-se no algoritmo MD4 de Ron Rivest [SCH 96].

# 3 Algoritmo Rijndael

## *3.1 Fundamentos matemáticos*

Não é escopo do presente trabalho analisar os conceitos matemáticos que servem de suporte para o algoritmo Rijndael, tampouco discorrer sobre os motivos das escolhas de projeto tomadas pelos criadores. Tais informações podem ser encontradas na proposta do Rijndael ao AES, na qual é descrito o algoritmo [DAE 99]. Entretanto, é salutar apresentar, de maneira concisa, as operações matemáticas básicas e quais as implicações em relação aos circuitos digitais necessários para a sua implementação.

Muitas operações no Rijndael são definidas ao nível de bytes, os quais representam elementos de um *Finite Field* GF($2^8$) [LID 86]. Tais elementos podem ser adicionados e multiplicados, mas as operações são diferentes da aritmética aplicada a números reais. Outras operações são definidas em termos de palavras, que são conjuntos de 4 bytes.

### 3.1.1 Operações ao nível de byte

Um byte pode ser representado por um polinômio de grau 7, no qual cada bit do conjunto {0,1} representa um coeficiente. Abaixo é mostrado um polinômio genérico que representa um byte.

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 \qquad (16)$$

Tem-se, por exemplo:

$$\#10100101b \quad => \quad x7 + x5 + x2 + 1 \qquad (17)$$

#### 3.1.1.1 Adição

Na representação polinomial, a soma no espaço GF($2^8$) de dois elementos é o polinômio cujos coeficientes são o resultado da soma módulo 2 dos coeficientes dos termos [DAE 99]. Desta maneira, pode-se mostrar que a operação da adição é um Ou-Exclusivo feito bit a bit sobre os coeficientes dos termos adicionados.

O Ou-Exclusivo é representado pelo símbolo “⊕”.

$$( x^6 + x^3 + x + 1 )+( x^7 + x^4 + x + 1 ) = x^7 + x^6 + x^4 + x^3 \qquad (18)$$

Ou, transferindo para binário:

$$\#01001011b \oplus \#10010011b = \#11011000b \qquad (19)$$

É importante notar que, no espaço GF($2^8$), não existe diferença entre a adição e a subtração, pois ambas são o resultado de um Ou-Exclusivo bit a bit dos operandos.

### 3.1.1.2 Multiplicação

Na representação polinomial, a multiplicação no $GF(2^8)$ corresponde à multiplicação de polinômios, módulo um polinômio irredutível de grau 8. Um polinômio é irredutível se ele não possui divisores que não sejam 1 ou ele mesmo. No Rijndael, esse polinômio é chamado de m(x) e é dado por [DAE 99]:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (20)$$

Tal multiplicação é representada pelo símbolo "•" .

Tem-se, por exemplo: #57h • #83h = #C1h (21)

$$( x^6 + x^4 + x^2 + x + 1 )( x^7 + x + 1 ) =$$
$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \qquad (22)$$

$$( x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 ) \text{ módulo}$$
$$( x^8 + x^4 + x^3 + x + 1 ) = x^7 + x^6 + 1 = \#C1h \qquad (23)$$

### 3.1.1.3 Multiplicação por "x"

Se o polinômio genérico b(x), mostrado na expressão 16, for multiplicado por "x", o resultado será:

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \qquad (24)$$

Entretanto, o resultado b(x) • x é obtido após encontrar o módulo m(x) da expressão acima. Caso o coeficiente $b_7$ seja igual a zero (0), o módulo m(x) será igual à própria expressão. Se $b_7$ é diferente de zero, então é necessário subtrair da expressão acima o polinômio m(x) [DAE 99].

Nota-se, então, que a multiplicação por "x" ao nível de bit pode ser implementada por um deslocamento de uma posição à esquerda, seguida de uma operação de Ou-Exclusivo com o valor #1Bh. Quando o coeficiente $b_7$ é igual a zero (0), não é necessária a operação de Ou-Exclusivo [DAE 99].

A multiplicação por "x" é usualmente chamada de *xtime*. Então:

$$b(x) \bullet x = xtime(b) \qquad (25)$$

A multiplicação por um polinômio de grau maior do que "x" pode ser implementada através da aplicação repetida da operação xtime e a soma dos resultados intermediários [DAE 99].

Tem-se, por exemplo:

#57h • #83h = #C1h (26)

$$\#57h \bullet \#02h = xtime\ (\#57h) = \#AEh \quad (27)$$
$$\#57h \bullet \#04h = xtime\ (\#AEh) = \#47h \quad (28)$$
$$\#57h \bullet \#08h = xtime\ (\#47h) = \#8Eh \quad (29)$$
$$\#57h \bullet \#10h = xtime\ (\#8Eh) = \#07h \quad (30)$$
$$\#57h \bullet \#20h = xtime\ (\#07h) = \#0Eh \quad (31)$$
$$\#57h \bullet \#40h = xtime\ (\#0Eh) = \#1Ch \quad (32)$$
$$\#57h \bullet \#80h = xtime\ (\#1Ch) = \#38h \quad (33)$$

$$\#57h \bullet \#83h = \#57h \bullet (\ \#01h \oplus \#02h \oplus \#80h\ ) =$$
$$\#57h \oplus \#AEh \oplus \#38h = \#C1h \quad (34)$$

### 3.1.2 Operações ao nível de palavras

Polinômios podem ser também definidos com coeficientes em $GF(2^8)$, ou seja, bytes. Uma palavra, que pode ser representada por um *array* de 4 *bytes*, forma assim um polinômio de grau menor do que quatro.

#### 3.1.2.1 Adição

A adição de polinômios com coeficientes em $GF(2^8)$ é feita através da aplicação da operação de Ou-Exclusivo bit a bit sobre cada um dos coeficientes dos polinômios, de maneira similar à adição em operações sobre bytes [DAE 99]. Novamente, não há diferenças entre a adição e a subtração.

#### 3.1.2.2 Multiplicação

A fim de demonstrar as multiplicações nas operações de palavras, serão utilizados dois polinômios com coeficientes em $GF(2^8)$.

$$a(x) = a_3\, x^3 + a_2\, x^2 + a_1\, x + a_0 \quad (35)$$
$$b(x) = b_3\, x^3 + b_2\, x^2 + b_1\, x + b_0 \quad (36)$$

O produto $c(x) = b(x) \bullet a(x)$ é dado por:

$$c(x) = c_6\, x^6 + c_5\, x^5 + c_4\, x^4 + c_3\, x^3 + c_2\, x^2 + c_1\, x + c_0 \quad (37)$$

sendo

$$c_6 = a_3 \bullet b_3 \quad (38)$$
$$c_5 = a_3 \bullet b_2 + a_2 \bullet b_3 \quad (39)$$
$$c_4 = a_3 \bullet b_1 + a_2 \bullet b_2 + a_1 \bullet b_3 \quad (40)$$
$$c_3 = a_3 \bullet b_0 + a_2 \bullet b_1 + a_1 \bullet b_2 + a_0 \bullet b_3 \quad (41)$$
$$c_2 = a_2 \bullet b_0 + a_1 \bullet b_1 + a_0 \bullet b_2 \quad (42)$$
$$c_1 = a_1 \bullet b_0 + a_0 \bullet b_1 \quad (43)$$
$$c_0 = a_0 \bullet b_0 \quad (44)$$

Contudo, o resultado c(x) não pode mais ser representado em um polinômio de grau menor do que quatro. Por isso, é preciso computar o módulo de c(x) por um outro polinômio de grau quatro, de maneira que o resultado seja necessariamente de grau menor do que quatro. Novamente, o Rijndael define o polinômio M(x) dado por [DAE 99]:

$$M(x) = x^4 + 1 \qquad (45)$$

O resultado de c(x) módulo M(x) é d(x), cujos coeficientes são:

$$d(x) = d_3\, x^3 + d_2\, x^2 + d_1\, x^1 + d_0 \qquad (46)$$

sendo

$$d_0 = a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \qquad (47)$$
$$d_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3 \qquad (48)$$
$$d_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3 \qquad (49)$$
$$d_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 \qquad (50)$$

A operação do polinômio b(x) por a(x) pode ser representada como uma multiplicação de matrizes, na qual a matriz que representa o polinômio a(x) é circulante, conforme visto abaixo:

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

FIGURA 3.1 - Multiplicação de polinômios representados por palavras

Tal multiplicação é representada pelo símbolo “⊗”.

### 3.1.2.3 Multiplicação por “x”

Ao multiplicar-se o polinômio b(x) pelo polinômio x, tem-se:

$$b_3\, x^4 + b_2\, x^3 + b_1\, x^2 + b_0\, x \qquad (51)$$

No entanto, b(x) ⊗ x é obtido através da redução da expressão acima pelo módulo $x^4 + 1$, produzindo:

$$( b_3\, x^4 + b_2\, x^3 + b_1\, x^2 + b_0\, x ) \text{ módulo } ( x^4 + 1 ) = b_2\, x^3 + b_1\, x^2 + b_0\, x + b_3 \qquad (52)$$

Assim, a multiplicação por x é equivalente à multiplicação do polinômio b(x) pela matriz mostrada abaixo, correspondendo a um deslocamento interno dos bytes do vetor do polinômio b(x).

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

FIGURA 3.2 - Multiplicação por "x"

## *3.2 Descrição*

O algoritmo Rijndael foi desenvolvido pelos belgas Joan Daemen da Proton World International e Vincent Rijmen da Katholieke Universiteit Leuven. O Rijndael é um algoritmo simétrico que cifra blocos de tamanho variável, com chaves de tamanho também variável [DAE 99]. A natureza do algoritmo permite que os tamanhos das chaves e dos blocos sejam múltiplos de 32 bits.

A especificação atual do algoritmo permite a utilização de chaves de 128, 192 ou 256 bits, e cifra blocos também de 128, 192 e 256 bits [DAE 99], totalizando nove modos de funcionamento. Entretanto, o padrão AES especificou somente três modos de funcionamento, nos quais o tamanho do bloco é fixado em 128 bits e a chave pode ser de 128, 192 ou 256 bits [NIS 2001a]. Tais implementações podem ser chamadas de AES-128, AES-192 e AES-256 [NIS 2001a]. A partir deste momento, a denominação AES-128 será bastante utilizada, referindo-se ao Rijndael com bloco e chave de 128 bits.

A estrutura do algoritmo baseia-se em sucessivas rodadas, chamadas também de *rounds*, nas quais funções específicas são aplicadas sobre o bloco de bits de entrada. O número de rodadas depende do número de bits de entrada, bem como do tamanho da chave utilizada.

A cada rodada, a transformação aplicada a este bloco de entrada de dados depende de valores derivados da chave principal escolhida. Este conjunto de bits é chamado de "chaves de rodada", ou *round key*, assumindo diferentes valores para cada nova passagem. Desta maneira, o tamanho da chave utilizada pode diferir do bloco de entrada, pois as chaves de rodadas são derivadas, e não iguais à chave principal.

Destacam-se claramente a existência da rotina de cifragem dos dados, chamada também de *cipher*, da rotina de decifragem, ou *inverse cipher*, e de outra responsável pela geração das chaves de rodada utilizadas em cada rodada, chamada de expansão da chave ou *expansion key*.

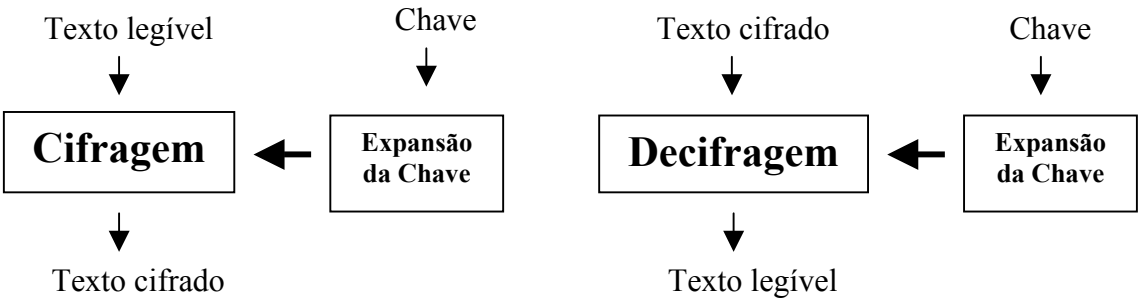


FIGURA 3.3 - Estrutura do algoritmo Rijndael

### *3.3 "State" e "Cipher Key"*

Por definição, o resultado intermediário dos blocos *cipher* ou *inverse cipher*, ou seja, o bloco de bits sobre os quais as funções são sucessivamente aplicadas, é chamado *state* [DAE 99].

O *state* pode ser vislumbrado como um array bidimensional de bytes, formado por quatro linhas e um número variável de colunas $N_b$, de maneira a totalizar o número de bits do bloco de entrada definido. Como são 4 linhas de 8 bits cada, o número de bits por colunas é igual a 32. Assim:

128 bits de entrada => $N_b = 128/32 = 4$ (53)
192 bits de entrada => $N_b = 192/32 = 6$ (54)
256 bits de entrada => $N_b = 256/32 = 8$ (55)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ | $a_{0,4}$ | $a_{0,5}$ | $a_{0,6}$ | $a_{0,7}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,4}$ | $a_{1,5}$ | $a_{1,6}$ | $a_{1,7}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{2,4}$ | $a_{2,5}$ | $a_{2,6}$ | $a_{2,7}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ | $a_{3,4}$ | $a_{3,5}$ | $a_{3,6}$ | $a_{3,7}$ |

FIGURA 3.4 - *State* como um array 4 x $N_b$ de bytes

De maneira análoga, o conjunto de bits da chave é chamado de *cipher key*, e também pode ser representado como um array bidimensional de bytes, formado por quatro linhas e $N_k$ colunas, no qual o número de colunas depende do tamanho da chave utilizada. Então:

Chave de 128 bits => $N_k = 4$ (56)
Chave de 192 bits => $N_k = 6$ (57)
Chave de 256 bits => $N_k = 8$ (58)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $k_{0,0}$ | $k_{0,1}$ | $k_{0,2}$ | $k_{0,3}$ | $k_{0,4}$ | $k_{0,5}$ | $k_{0,6}$ | $k_{0,7}$ |
| $k_{1,0}$ | $k_{1,1}$ | $k_{1,2}$ | $k_{1,3}$ | $k_{1,4}$ | $k_{1,5}$ | $k_{1,6}$ | $k_{1,7}$ |
| $k_{2,0}$ | $k_{2,1}$ | $k_{2,2}$ | $k_{2,3}$ | $k_{2,4}$ | $k_{2,5}$ | $k_{2,6}$ | $k_{2,7}$ |
| $k_{3,0}$ | $k_{3,1}$ | $k_{3,2}$ | $k_{3,3}$ | $k_{3,4}$ | $k_{3,5}$ | $k_{3,6}$ | $k_{3,7}$ |

FIGURA 3.5 - Cipher Key como um array 4 x $N_k$ de bytes

Como também, tanto o *state* quanto a *cipher key* podem ser considerados como um array unidimensional de palavras, no qual o número da coluna define o índice.

## *3.4 Número de rodadas*

O número de rodadas aplicadas ao *state*, utilizando diferentes chaves derivadas da *cipher key*, é função do número de bits do bloco de entrada e da chave escolhida [DAE 99].

Segundo a especificação original do algoritmo [DAE 99], o número de rodadas $N_r$ é definido por:

| **Nr** | **$N_b = 4$** | **$N_b = 6$** | **$N_b = 8$** |
|---|---|---|---|
| $N_k = 4$ | 10 | 12 | 14 |
| $N_k = 6$ | 12 | 12 | 14 |
| $N_k = 8$ | 14 | 14 | 14 |

TABELA 3.1 - Número de rodadas em função dos tamanhos do bloco e da chave

Entretanto, de acordo com o padrão AES, em virtude da característica constante do bloco de entrada, o qual foi definido como $N_b = 4$, tem-se:

| **$N_k$** | **$N_r$** |
|---|---|
| 4 | 10 |
| 6 | 12 |
| 8 | 14 |

TABELA 3.2 - Número de rodadas definidas pelo AES

### 3.5 Cipher

Na figura 3.6 é apresentado o diagrama em blocos do *cipher*, de maneira a mostrar em alto nível as transformações feitas sobre o bloco de entrada a partir da *cipher key*. A idéia do diagrama em blocos é fornecer uma visão geral sobre a metodologia adotada pelo algoritmo.

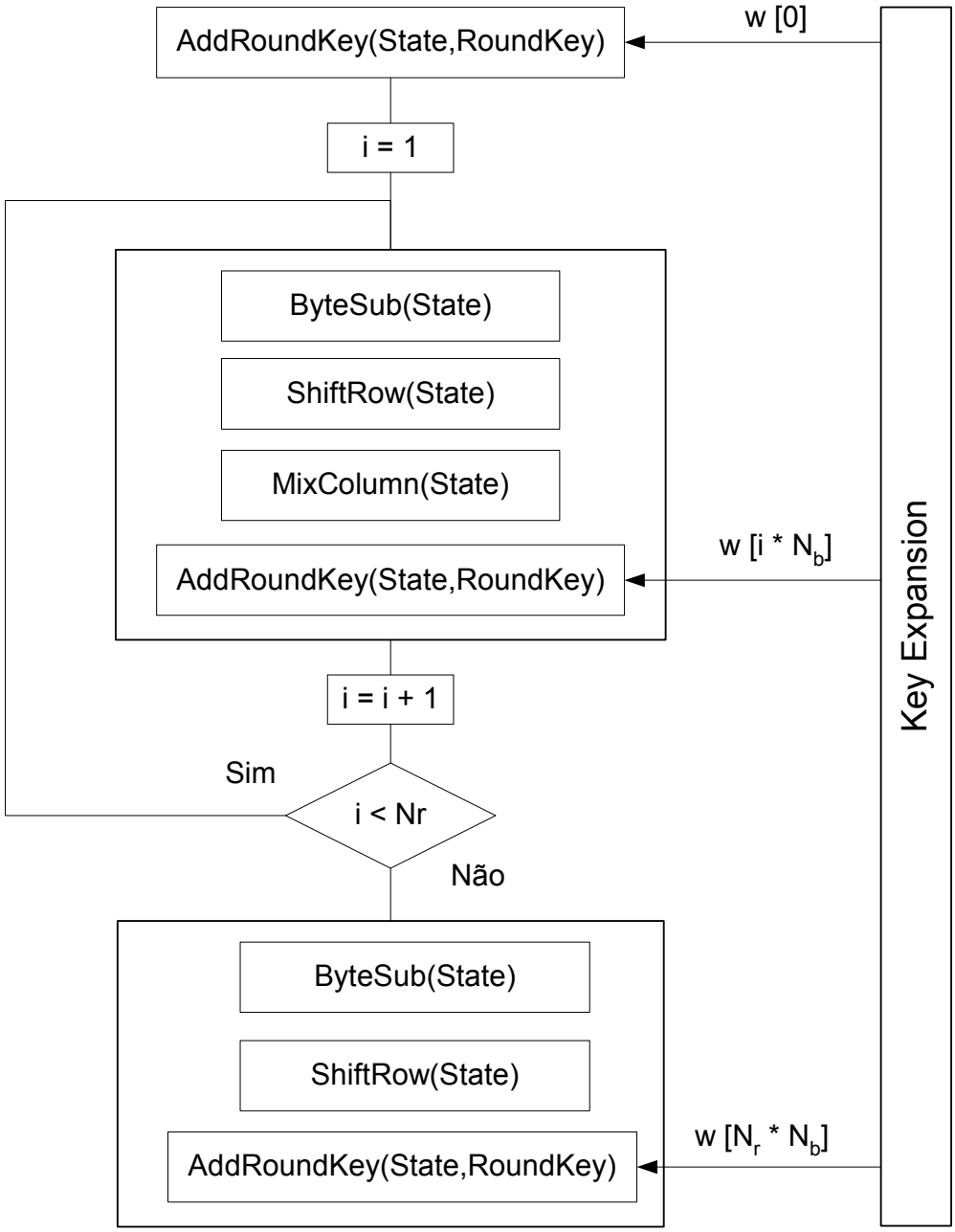


FIGURA 3.6 - Diagrama em Blocos do Cipher

A cifragem é feita sobre o *state*, utilizando um conjunto de *round keys* extraídos da *cipher key*. A primeira transformação feita é a adição do bloco de bits *state* com a primeira *round key*. Em seguida, o *state* é submetido a várias rodadas de operações consecutivas, em função da definição do algoritmo. Note-se que a última rodada é diferente das demais, pois uma das transformações não está mais presente.

É interessante notar que devem existir $N_{r+1}$ *round keys*, pois são $N_r$ rodadas, além da adição inicial. Cada *round key* possui o mesmo tamanho do bloco de bits do *state*, a fim de permitir a adição bit a bit dos seus elementos.

As *round keys* são geradas pela função *key expansion*, sendo que os primeiros bits são formados pela própria *cipher key*, enquanto que os seguintes seguem uma regra de formação em função dos antecedentes. No caso do AES, apesar da *cipher key* poder assumir valores de 128, 192 ou 256 bits, as *round keys* serão sempre de 128 bits.

As transformações utilizadas na cifragem são:

- ByteSub
- ShiftRow
- MixColumn
- AddRoundKey

### 3.5.1 ByteSub

A transformação ByteSub é uma substituição não linear, que opera independentemente sobre cada byte do *state*, utilizando uma tabela previamente construída, chamada S-box.

Tal tabela é encontrada a partir de duas operações:

1. encontra-se o Inverso Multiplicativo no Finite Field $GF(2^8)$. O elemento {00} é mapeado para ele mesmo;
2. aplica-se uma Transformação Afim, ou *Affine Transformation*, definida em [NIS 2001a].

Desta maneira, todo o *state* é submetido a uma transformação byte a byte, produzindo um novo conjunto de elementos. Abaixo é ilustrada a transformação sobre o *state* definido no padrão AES, onde $S_{r,c}$ representa o byte posicionado na linha r e coluna c.

| | | | |
|---|---|---|---|
| $S_{0,0}$ | $S_{0,1}$ | $S_{0,2}$ | $S_{0,3}$ |
| $S_{1,0}$ | $S_{1,1}$ | $S_{1,2}$ | $S_{1,3}$ |
| $S_{2,0}$ | $S_{2,1}$ | $S_{2,2}$ | $S_{2,3}$ |
| $S_{3,0}$ | $S_{3,1}$ | $S_{3,2}$ | $S_{3,3}$ |

$\Rightarrow$ S-Box $\Rightarrow$

| | | | |
|---|---|---|---|
| $S'_{0,0}$ | $S'_{0,1}$ | $S'_{0,2}$ | $S'_{0,3}$ |
| $S'_{1,0}$ | $S'_{1,1}$ | $S'_{1,2}$ | $S'_{1,3}$ |
| $S'_{2,0}$ | $S'_{2,1}$ | $S'_{2,2}$ | $S'_{2,3}$ |
| $S'_{3,0}$ | $S'_{3,1}$ | $S'_{3,2}$ | $S'_{3,3}$ |

FIGURA 3.7 - Transformação BytesSub atuando sobre o *state*

A S-Box é apresentada em hexadecimal na tabela a seguir.

| ***X\Y*** | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** | ***9*** | ***A*** | ***B*** | ***C*** | ***D*** | ***E*** | ***F*** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***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*** | 4 | C7 | 23 | C3 | 18 | 96 | 05 | 9A | 07 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| ***4*** | 9 | 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 | 16 | 87 | E9 | CE | 55 | 28 | DF |
| ***F*** | 8C | A1 | 89 | 0D | BF | E6 | 42 | 68 | 41 | 99 | 2D | 0F | B0 | 54 | BB | 16 |

TABELA 3.3 - S-Box da Encriptação

### 3.5.2 ShiftRow

A transformação ShiftRow opera sobre as linhas do *state*, representando um deslocamento cíclico para a esquerda dos bytes das últimas três linhas do *state*. A posição dos bytes nas colunas é modificada de acordo com a linha e com o tamanho de colunas do *state*, que reflete o tamanho do bloco de bits de entrada de dados. De acordo com o tópico 3.3, o tamanho do bloco de bits de entrada é representado pelo valor $N_b$. A tabela a seguir define valores para C1, C2 e C3, os quais representam o deslocamento para as linhas 1, 2 e 3, respectivamente.

| **Nb** | **C1** | **C2** | **C3** |
|---|---|---|---|
| 4 | 1 | 2 | 3 |
| 6 | 1 | 2 | 3 |
| 8 | 1 | 3 | 4 |

TABELA 3.4 - Deslocamento do ShiftRow em função de $N_b$

Como o AES foi definido apenas para blocos de 128 bits de entrada por enquanto, tem-se que $N_b = 4$. Assim, o deslocamento é ilustrado a seguir.

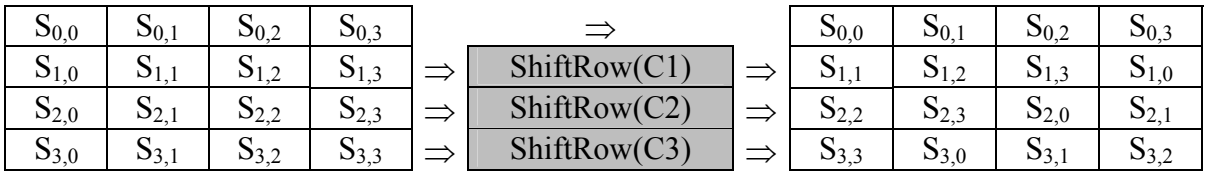


FIGURA 3.8 - Transformação ShiftRow atuando sobre o *state*

### 3.5.3 MixColumn

A transformação MixColumn opera sobre as colunas do *state*, tratando-as com um polinômio de grau menor do que quatro, cujos coeficientes são bytes que encontram-se no Finite Field $GF(2^8)$. Tais polinômios são multiplicados, em $GF(2^8)$, por um polinômio a(x) definido por:

$$a(x) = \{\#03h\}\ x^3 + \{\#01h\}\ x^2 + \{\#01h\}\ x + \{\#02h\} \qquad (59)$$

Então,

$$S'(x) = a(x) \otimes S(x) \qquad (60)$$

Efetuando a operação definida no tópico 3.1.2.2, chega-se às seguintes expressões, onde $N_b \geq c \geq 0$ :

$$S'_{0,c} = (\{\#02h\}\bullet S_{0,c}) \oplus (\{\#03h\}\bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \qquad (61)$$
$$S'_{1,c} = S_{0,c} \oplus (\{\#02h\}\bullet S_{1,c}) \oplus (\{\#03h\}\bullet S_{2,c}) \oplus S_{3,c} \qquad (62)$$
$$S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus (\{\#02h\}\bullet S_{2,c}) \oplus (\{\#03h\}\bullet S_{3,c}) \qquad (63)$$
$$S'_{3,c} = (\{\#03h\}\bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{\#02h\}\bullet S_{3,c}) \qquad (64)$$

Abaixo é ilustrada a transformação aplicada ao *state* definido no AES.

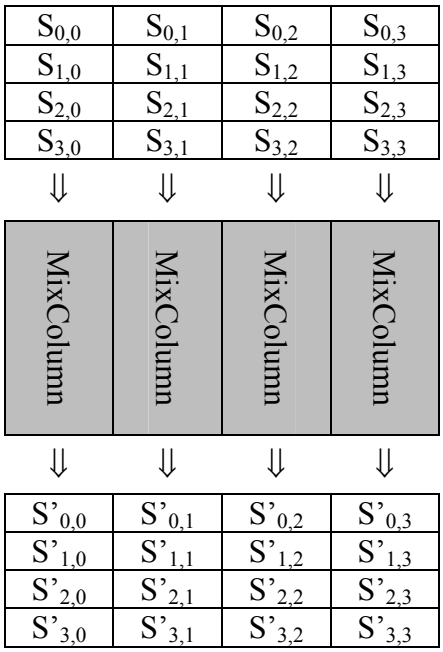


FIGURA 3.9 - Transformação MixColumn atuando sobre o *state*

### 3.5.4 AddRoundKey

A transformação AddRoundKey é a adição do *state* com a *round key*. Conforme citado anteriormente, ambos os conjuntos de bits possuem o mesmo tamanho, de maneira que a adição torna-se um Ou-Exclusivo bit-a-bit entre os dois operandos.

Abaixo é ilustrada a transformação, sendo que a *round key* é representada por $R_{r,c}$, analogamente ao *state* $S_{r,c}$.

| | | | |
|---|---|---|---|
| $S_{0,0}$ | $S_{0,1}$ | $S_{0,2}$ | $S_{0,3}$ |
| $S_{1,0}$ | $S_{1,1}$ | $S_{1,2}$ | $S_{1,3}$ |
| $S_{2,0}$ | $S_{2,1}$ | $S_{2,2}$ | $S_{2,3}$ |
| $S_{3,0}$ | $S_{3,1}$ | $S_{3,2}$ | $S_{3,3}$ |

$\oplus$

| | | | |
|---|---|---|---|
| $R_{0,0}$ | $R_{0,1}$ | $R_{0,2}$ | $R_{0,3}$ |
| $R_{1,0}$ | $R_{1,1}$ | $R_{1,2}$ | $R_{1,3}$ |
| $R_{2,0}$ | $R_{2,1}$ | $R_{2,2}$ | $R_{2,3}$ |
| $R_{3,0}$ | $R_{3,1}$ | $R_{3,2}$ | $R_{3,3}$ |

=

| | | | |
|---|---|---|---|
| $S'_{0,0}$ | $S'_{0,1}$ | $S'_{0,2}$ | $S'_{0,3}$ |
| $S'_{1,0}$ | $S'_{1,1}$ | $S'_{1,2}$ | $S'_{1,3}$ |
| $S'_{2,0}$ | $S'_{2,1}$ | $S'_{2,2}$ | $S'_{2,3}$ |
| $S'_{3,0}$ | $S'_{3,1}$ | $S'_{3,2}$ | $S'_{3,3}$ |

FIGURA 3.10 - Transformação AddRoundKey, tendo como parâmetro a *round key*

## *3.6 Expansion Key*

Conforme mostrado anteriormente, a cada novo round do Rijndael, uma das transformações, chamada AddRoundKey, faz uma adição no espaço $GF(2^8)$ entre o *state* e a *round key*. O valor da *round key* muda a cada nova rodada.

Desta maneira, são necessárias $N_r + 1$ *round keys* para garantir a primeira adição, e depois as $N_r$ restantes. Cada *round key*, a fim de satisfazer a transformação de AddRoundKey, deve apresentar o mesmo tamanho do *state*, pois a transformação em questão é na prática um Ou-Exclusivo bit-a-bit.

Entretanto, o tamanho da chave não necessariamente corresponde ao tamanho do bloco de bits de entrada, que acaba definindo o tamanho do *state*. Segundo definições do próprio algoritmo, é possível, por exemplo, utilizar blocos de entrada de 128 bits e chaves de 256 bits. O Rijndael compatibiliza as possíveis diferenças através do processo de Expansion Key [DAE 99].

É gerado um array unidimensional de $N_b(N_r + 1)$ palavras (cada palavra é composta por quatro bytes, correspondendo ao tamanho da coluna). Cada *round key* é composta por $N_b$ palavras, formando estruturas matriciais de mesma ordem que o *state*. A figura abaixo ilustra a matriz de *round keys* gerada pelo procedimento.

| | | | | |
|---|---|---|---|---|
| w[0] | w[1] | w[2] | ... | w[Nb(Nr+1)] |

FIGURA 3.11 - Array unidimensional de palavras

As primeiras $N_k$ palavras são preenchidas pela chave de entrada. As demais palavras são definidas recursivamente em função das entradas anteriores. Cada nova palavra é resultado de um Ou-Exclusivo entre a palavra anterior, w[i-1], e a palavra $N_k$ posições anteriores, w[i-$N_k$]. Para palavras cuja posição i é múltipla de Nk, aplica-se a transformação definida por KStran. Quando a chave de entrada é de 256 bits, o procedimento é um pouco diferente, pois parte da transformação KStran é aplicada também quando i-4 é múltiplo de $N_k$ [DAE 99].

### 3.6.1 KStran

A transformação KStran se aplica à palavra anterior w[i-1] quando i é múltiplo de $N_k$, antes do Ou-Exclusivo com a palavra $N_k$ posições anteriores. Ela consiste de uma rotação para a esquerda da palavra, seguida da mesma transformação SubBytes definida anteriormente. Após, ainda é feito um Ou-Exclusivo com uma constante determinada em função de i. A figura abaixo ilustra a transformação.

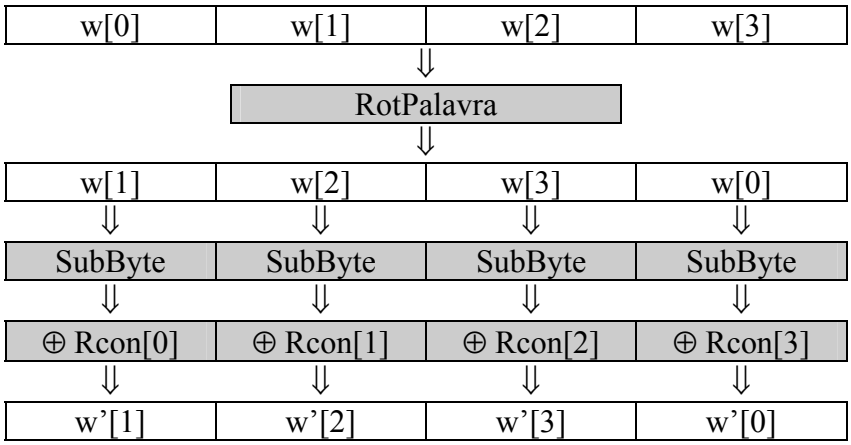


FIGURA 3.12 - Transformação KStran

## *3.7 Inverse Cipher*

A operação de cifragem pode ser invertida e implementada na ordem inversa, produzindo o *Inverse Cipher*. A seguir pode ser observado o diagrama em blocos do *Inverse Cipher*. Uma das caracterísiticas marcantes da operação inversa é a necessidade de gerar todo o conjunto de round keys antecipadamente, uma vez que o valor final é necessário logo na primeira transformação.

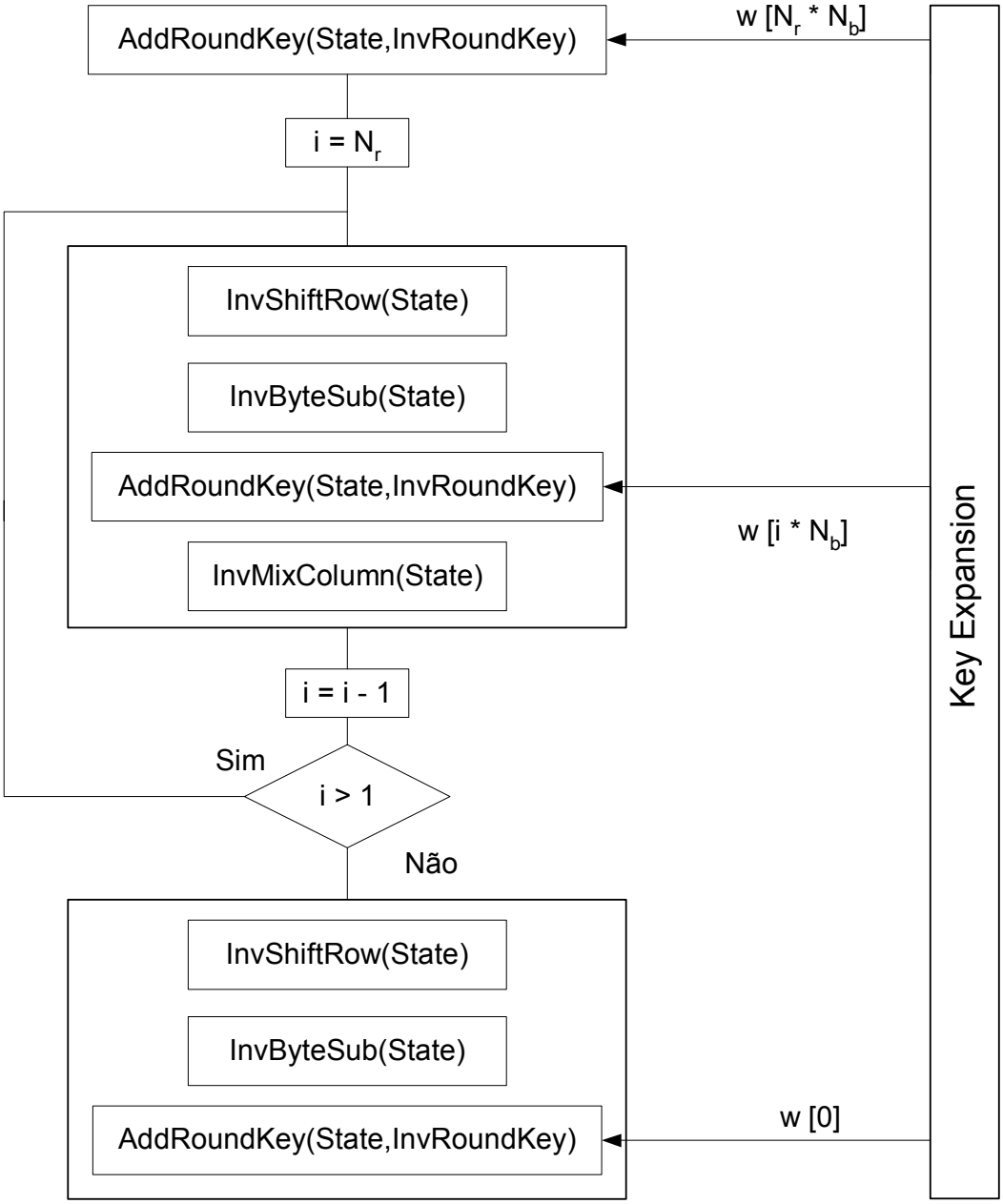


FIGURA 3.13 - Diagrama em blocos do Inverse Cipher.

Entretanto, em função de algumas propriedades particulares das transformações utilizadas pelo Rijndael, pode-se construir uma seqüência de operações melhor distribuídas. Tais propriedades são [DAE 99][NIS 2001a]:

- a ordem de execução das transformações ByteSub e ShiftRow, como também das suas funções inversas InvByteSub e InvShiftRow, não altera o resultado. Tais transformações operam e movem bytes individuais;

- as operações MixColumn e InvMixColumn são lineares em relação à sua entrada, de maneira que:

InvMixColumn(*State* XOR Round Key) =
InvMixColumn(*State*) XOR InvMixColumn(Round Key) (65)

A figura 3.14 ilustra o que se chama *Inverse Cipher* equivalente. As modificações atingem também a geração das *round keys*, que agora são denominadas *round keys* modificadas.

As transformações necessárias ao *Inverse Cipher* ou ao *Equivalent Inverse Cipher* representam o inverso das transações feitas no *Cipher*. Assim, tem-se:

- AddRoundKey
- InvByteSub
- InvShiftRow
- InvMixColumn

A transformação AddRoundKey é o inverso de si mesma, pois a adição e a subtração no espaço GF($2^8$) são iguais.

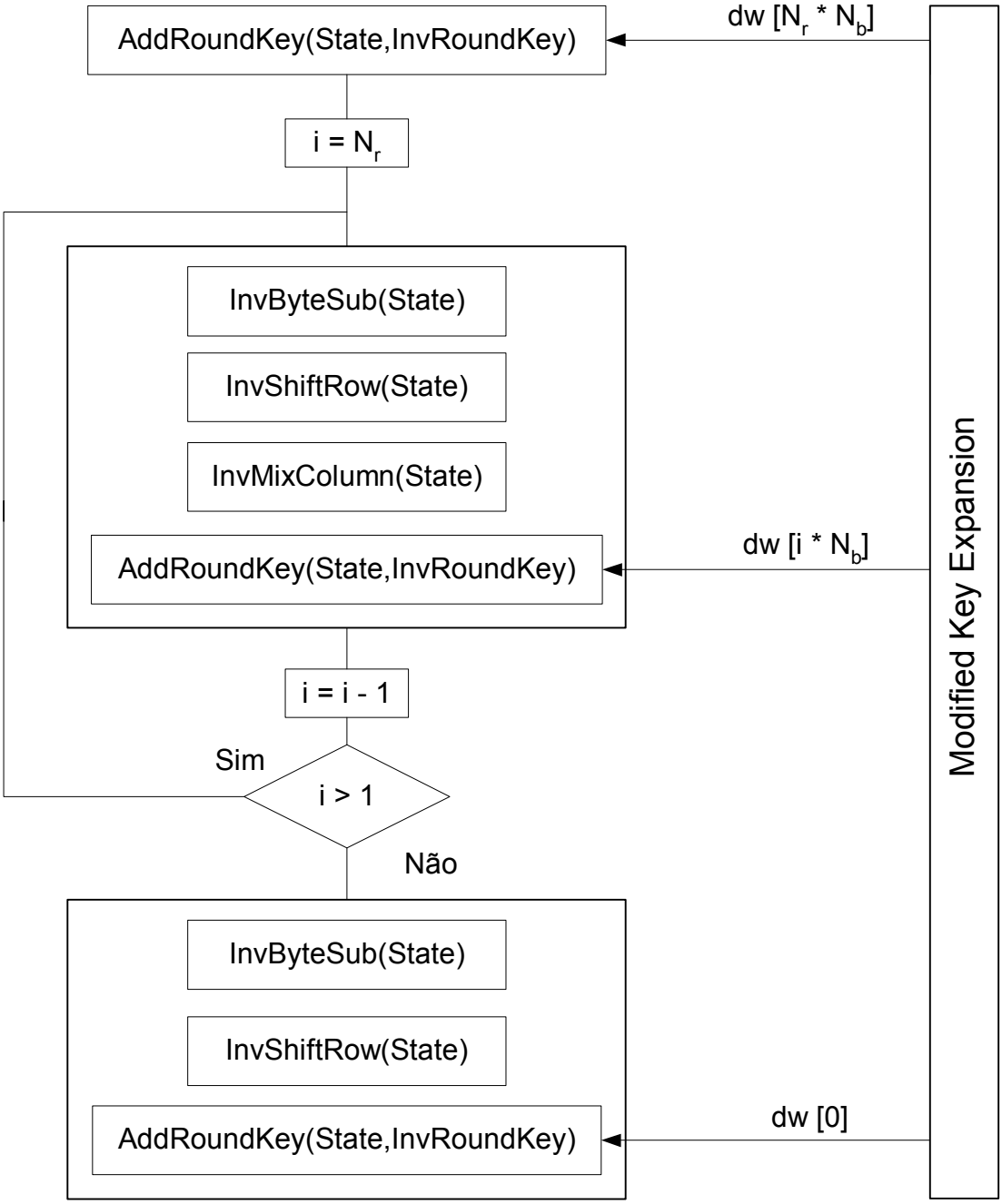


FIGURA 3.14 - Diagrama em Blocos do Inverse Cipher

### 3.7.1 InvByteSub

A transformação InvByteSub, a exemplo da ByteSub, é uma substituição não linear, que opera independentemente sobre cada byte do *state*, revertendo a operação anteriormente efetuada. Utiliza-se outra tabela previamente construída, chamada S-box Inversa. A S-Box Inversa é apresentada em hexadecimal na tabela a seguir.

| ***X\Y*** | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** | ***9*** | ***A*** | ***B*** | ***C*** | ***D*** | ***E*** | ***F*** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***0*** | 52 | 09 | 6A | D5 | 30 | 36 | A5 | 38 | BF | 40 | A3 | 9E | 81 | F3 | D7 | FB |
| ***1*** | 7C | E3 | 39 | 82 | 9B | 2F | FF | 87 | 34 | 8E | 43 | 44 | C4 | DE | E9 | CB |
| ***2*** | 54 | 7B | 94 | 32 | A6 | C2 | 23 | 3D | EE | 4C | 95 | 0B | 42 | FA | C3 | 4E |
| ***3*** | 08 | 2E | A1 | 66 | 28 | D9 | 24 | B2 | 76 | 5B | A2 | 49 | 6D | 8B | D1 | 25 |
| ***4*** | 72 | F8 | F6 | 64 | 86 | 68 | 98 | 16 | D4 | A4 | 5C | CC | 5D | 65 | B6 | 92 |
| ***5*** | 6C | 70 | 48 | 50 | FD | ED | B9 | DA | 5E | 15 | 46 | 57 | A7 | 8D | 9D | 84 |
| ***6*** | 90 | D8 | AB | 00 | 8C | BC | D3 | 0A | F7 | E4 | 58 | 05 | B8 | B3 | 45 | 06 |
| ***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 3.5 - S-Box da Decriptação

### 3.7.2 InvShiftRow

A transformação InvShiftRow reverte o deslocamento dos bytes nas linhas feito pelo ShiftRow. O deslocamento segue a mesma regra da tabela 3.4 ilustrada na transformação ShiftRow, tendo, entretanto, sentido inverso.

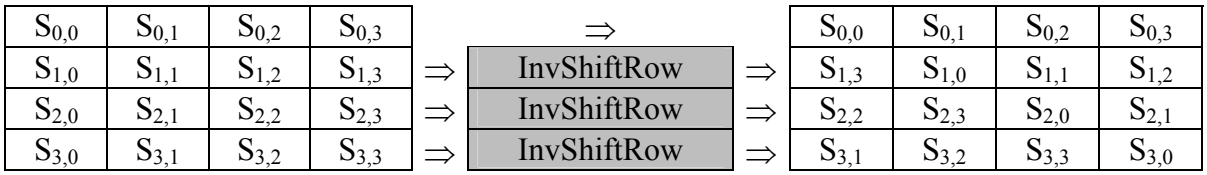


FIGURA 3.15 - Transformação InvShiftRow atuando sobre o *state*

### 3.7.3 InvMixColumn

A transformação InvMixColumn é o inverso da MixColumn, operando sobre as colunas do *state*. Novamente, trata-se cada coluna como um polinômio de grau menor do que quatro, cujos coeficientes são bytes que encontram-se no Finite Field $GF(2^8)$. Tais polinômios são multiplicados, em $GF(2^8)$, por um polinômio $a^{-1}(x)$ definido por:

$$a^{-1}(x) = \{\#0Bh\}\ x^3 + \{\#0Dh\}\ x^2 + \{\#09h\}\ x + \{\#0Eh\} \qquad (66)$$

Então,

$$S'(x) = a^{-1}(x) \otimes S(x) \qquad (67)$$

Efetuando a operação definida no tópico 3.1.2.2, chega-se às seguintes expressões, onde $N_b \geq c \geq 0$ :

$$S'_{0,c}=(\{\#0Eh\}\bullet S_{0,c})\oplus(\{\#0Bh\}\bullet S_{1,c})\oplus(\{\#0Dh\}\bullet S_{2,c})\oplus(\{\#09h\}\bullet S_{3,c}) \quad (68)$$
$$S'_{1,c}=(\{\#09h\}\bullet S_{0,c})\oplus(\{\#0Eh\}\bullet S_{1,c})\oplus(\{\#0Bh\}\bullet S_{2,c})\oplus(\{\#0Dh\}\bullet S_{3,c}) \quad (69)$$
$$S'_{2,c}=(\{\#0Dh\}\bullet S_{0,c})\oplus(\{\#09h\}\bullet S_{1,c})\oplus(\{\#0Eh\}\bullet S_{2,c})\oplus(\{\#0Bh\}\bullet S_{3,c}) \quad (70)$$
$$S'_{3,c}=(\{\#0Bh\}\bullet S_{0,c})\oplus(\{\#0Dh\}\bullet S_{1,c})\oplus(\{\#09h\}\bullet S_{2,c})\oplus(\{\#0Eh\}\bullet S_{3,c}) \quad (71)$$

Abaixo é ilustrada a transformação aplicada ao *state* definido no AES.

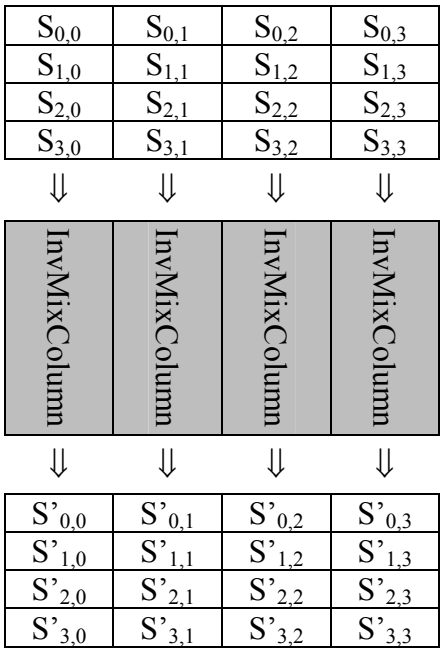


FIGURA 3.16 - Transformação InvMixColumn atuando sobre o *state*

# 4 Implementação do UFIR-128

O NIST definiu o algoritmo Rijndael como o novo padrão *Advanced Encryption Standard* em 28 de Fevereiro de 2001. Nessa oportunidade, foi publicada uma versão preliminar do padrão, na qual definiu-se o bloco de entrada de dados com 128 bits [NIS 2001b]. As chaves podem assumir tamanhos de 128, 192 ou 256 bits, chamando-se respectivamente AES-128, AES-192 e AES-256 [NIS 2001a].

O período de comentários para o novo padrão encerrou-se 90 dias após, precisamente no dia 29 de Maio de 2001. Espera-se a conclusão do padrão até o final do ano de 2001 [NIS 2001b].

Desta maneira, esta foram as condições de contorno que definiram os principais aspectos da implementação.

## *4.1 Modos de operação*

Paralelamente à divulgação do padrão do AES, o NIST publicou uma versão prévia das recomendações dos modos de utilização do algoritmo de criptografia simétrica [NIS 2001e]. As recomendações independem do algoritmo escolhido e são amplamente utilizadas nos algoritmos de segurança. Tais modos de operação instruem como utilizar um algoritmo simétrico para prover autenticação, assinatura de mensagens, confidencialidade em canais de comunicação, entre outras aplicações.

Foram então recomendados os seguintes modos de operação:

- *Electronic Codebook*
- *Cipher Block Chaining*
- *Cipher Feedback*
- *Output Feedback*
- *Counter*
- *Cipher Block Chaining – Message Authentication Code*

### 4.1.1 Electronic Codebook (ECB)

No modo *Electronic Codebook*, as funções de encriptação e decriptação são aplicadas aos blocos de maneira independente, ou seja, divide-se a mensagem em blocos de tamanho definido e aplica-se as transformações. Por isso esse modo é chamado de livro de código eletrônico, pois dada uma determinada chave, a cada bloco de entrada encontra-se somente um valor de saída.

Nesse modo de operação pode-se processar tantos blocos simultaneamente quantos forem possíveis, permitindo uma arquitetura totalmente paralela.

FIGURA 4.1 - Encriptação e Decriptação, respectivamente, no modo ECB

### 4.1.2 Cipher Block Chaining (CBC)

No modo *Cipher Block Chaining* as operações de encriptação e decriptação não são aplicadas de maneira independente aos blocos da mensagem.

É feita uma operação de Ou-Exclusivo entre o bloco de entrada e a saída encriptada do bloco anterior, antes de efetivamente submeter ao bloco de encriptação. Na primeira rodada, como não há ainda uma saída encriptada, utiliza-se um vetor de inicialização, o qual deve ser mantido em sigilo. A figura abaixo ilustra a encriptação no modo CBC.

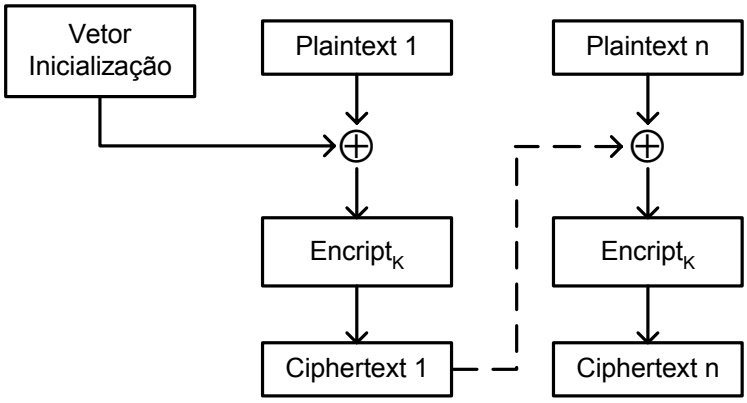


FIGURA 4.2 - Encriptação no modo CBC

Na decriptação, o texto cifrado – *ciphertext* – é decriptado, e é então feito um Ou-Exclusivo com o texto cifrado anterior a fim de encontrar o texto original. Na primeira rodada, deve-se utilizar novamente o vetor de inicialização. A figura abaixo ilustra a operação.

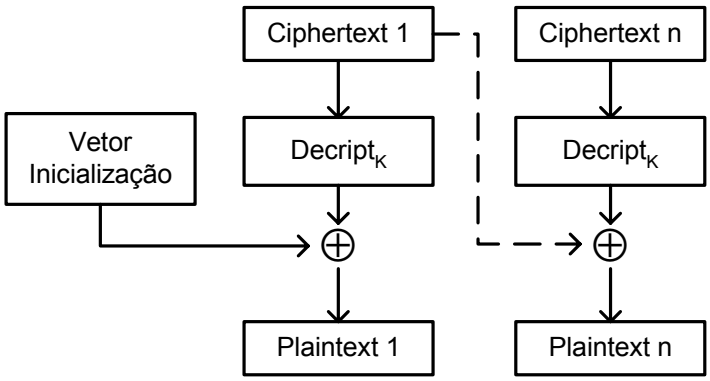


FIGURA 4.3 - Decriptação no modo CBC

É interessante notar a impossibilidade de construir uma arquitetura paralela para a encriptação no modo CBC, pois a entrada de cada bloco depende do resultado do bloco anterior, não permitindo a transformação simultânea de vários blocos. Entretanto, na decriptação é possível paralelizar, operando sobre vários blocos ao mesmo tempo.

### 4.1.3 Cipher Feedback (CFB)

No modo *Cipher Feedback* pode-se utilizar blocos de bits de entrada menores do que o número determinado pelo algoritmo simétrico. A definição detalhada desse modo pode ser encontrada em [NIS 2001e]. A descrição feita a seguir assume que o tamanho dos blocos de entrada e saída sejam iguais aos definidos pelo algoritmo simétrico.

No processo de encriptação, o resultado da rodada anterior é encriptado novamente, e no final é feito um Ou-Exclusivo com o bloco de entrada, produzindo a saída da rodada em questão. Novamente, na primeira rodada é utilizado o vetor de inicialização. Abaixo ilustra-se a encriptação no modo CFB.

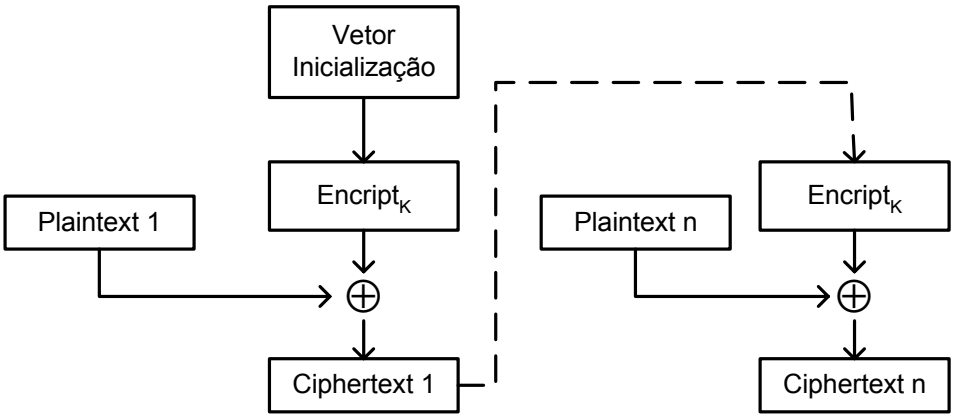


FIGURA 4.4 - Encriptação no modo CFB

Na decriptação os textos cifrados – *ciphertext* – sofrem as transformações das funções de decriptação, e depois são submetidos a um Ou-Exclusivo com o bloco cifrado posterior, a fim de produzir o texto original – *plaintext*. Novamente, na primeira rodada deve-se utilizar o vetor de inicialização. A figura abaixo ilustra o processo.

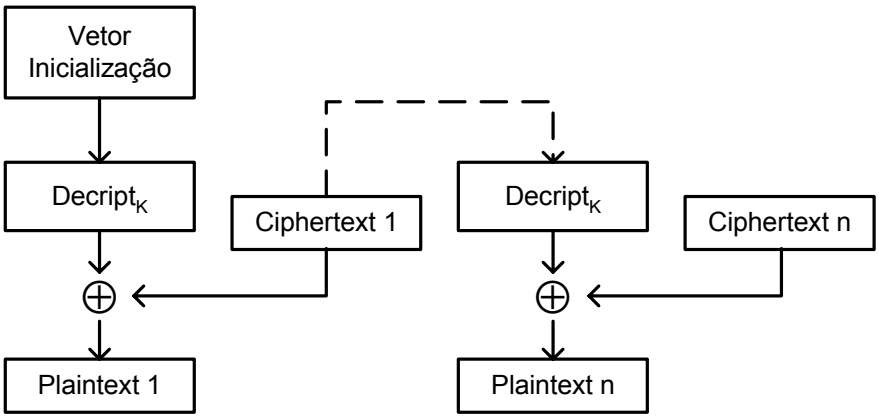


FIGURA 4.5 - Decriptação no modo CFB

Novamente, não é possível paralelizar a operação de encriptação, mas somente a de decriptação.

### 4.1.4 Output Feedback (OFB)

No modo *Output Feedback,* um vetor de inicialização é transformado pela função de encriptação, gerando saídas que são novamente submetidas à encriptação, produzindo um série determinada de vetores de saída. O bloco de bits de entrada é então submetido a um Ou-Exclusivo com o conjunto de vetores de saída, a fim de produzir a saída de blocos cifrados. A figura abaixo ilustra este modo de operação.

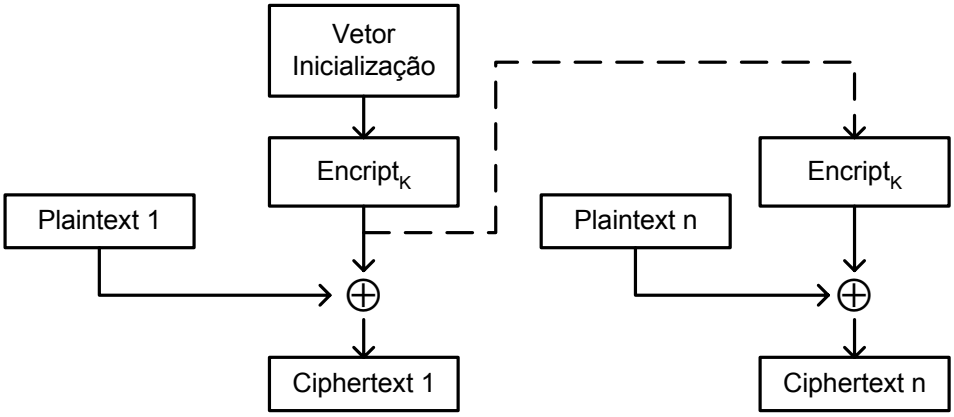


FIGURA 4.6 - Encriptação no modo OFB

Na decriptação gera-se o mesmo conjunto de vetores de saída a partir do vetor de inicialização, mas faz-se um Ou-Exclusivo com o texto cifrado, produzindo o texto original. A figura abaixo ilustra o conceito.

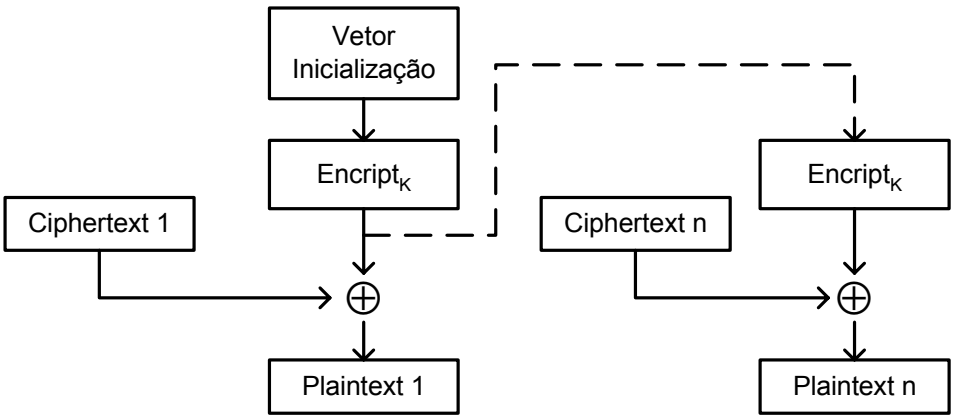


FIGURA 4.7 - Decriptação no modo OFB

Se por um lado não é possível paralelizar nenhuma das operações, pode-se computar previamente o conjunto de vetores de saída a partir do vetor de entrada. Desta maneira, tanto a encriptação quanto a decriptação poderiam ser feitas através de Ou-Exclusivos paralelos entre a entrada e o conjunto de vetores previamente armazenados. No entanto, por questões de confidencialidade, recomenda-se a troca do vetor de inicialização a cada nova mensagem [NIS 2001e].

### 4.1.5 Counter (CTR)

No modo *Counter* é necessária a presença de uma seqüência de blocos, com a propriedade de que cada bloco é diferente um do outro. Tal condição deve ser satisfeita enquanto a mesma chave for utilizada, independentemente do número de mensagens processadas. Tanto na encriptação quanto na decriptação, submete-se tais contadores à função de encriptação do algoritmo simétrico e ao final aplica-se um Ou-Exclusivo. Abaixo ilustra-se tais transformações.

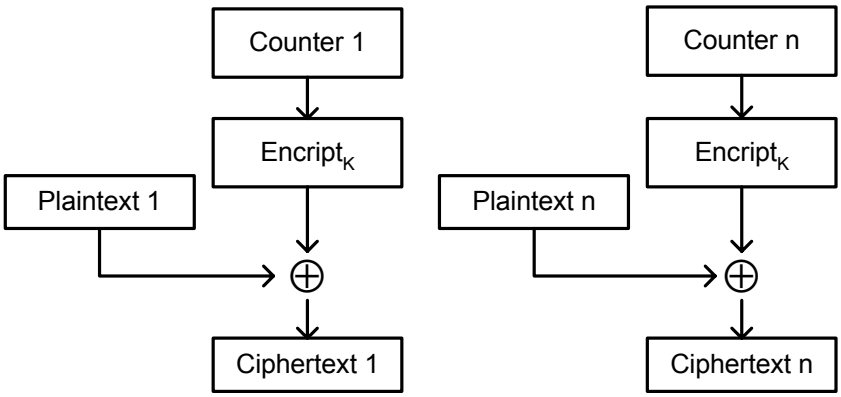


FIGURA 4.8 - Encriptação no modo CTR

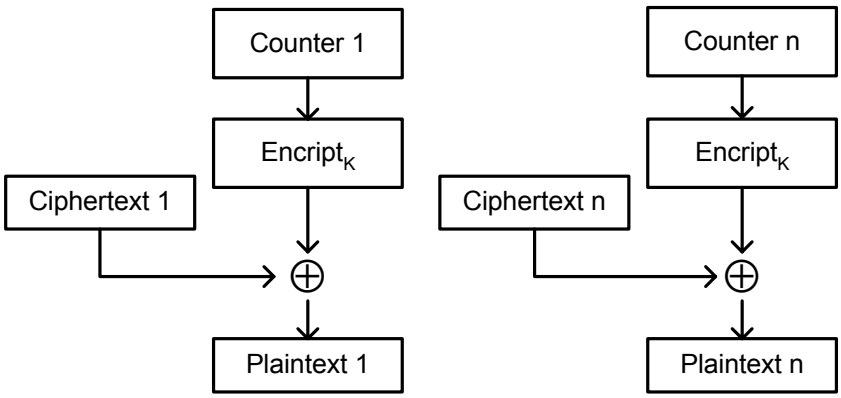


FIGURA 4.9 - Decriptação no modo CTR

A clara vantagem no modo CTR é a possibilidade de efetuar de maneira totalmente paralela a encriptação e a decriptação, bem como a alternativa de pré-processamento.

### 4.1.6 Cipher Block Chaining – Message Authentication Code (CBC-MAC)

O modo *Cipher Block Chaining – Message Authentication Code (CBC-MAC)* é uma função *hash*, explicada anteriormente neste trabalho. A figura a seguir ilustra a metodologia utilizada para gerar uma assinatura a partir de uma mensagem qualquer.

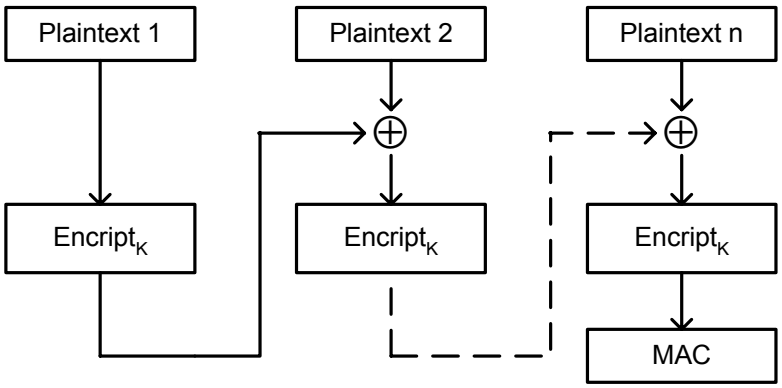


FIGURA 4.10 - Geração do Message Authentication Code a partir de um mensagem

## *4.2 Arquiteturas de projeto para cifradores simétricos*

Em termos de arquitetura de projeto para os blocos de encriptação e decriptação, pode-se ressaltar basicamente quatro metodologias [GAJ00]. Tais modelos são apresentados abaixo.

São mostradas também as velocidades esperadas para cada tipo de implementação. O cálculo leva em conta a quantidade de bits que pode ser processada em determinado período de tempo. Em geral nomeia-se tal velocidade de taxa de transmissão, ou *throughput*, nome comumente encontrado na literatura da área de comunicação de dados.

### 4.2.1 Arquitetura básica

Cada rodada é implementada como uma lógica combinacional, sendo registrada ao final. Implementa-se então um bloco de controle que garanta a passagem pelo número definido de rodadas. A figura a seguir ilustra esse modelo.

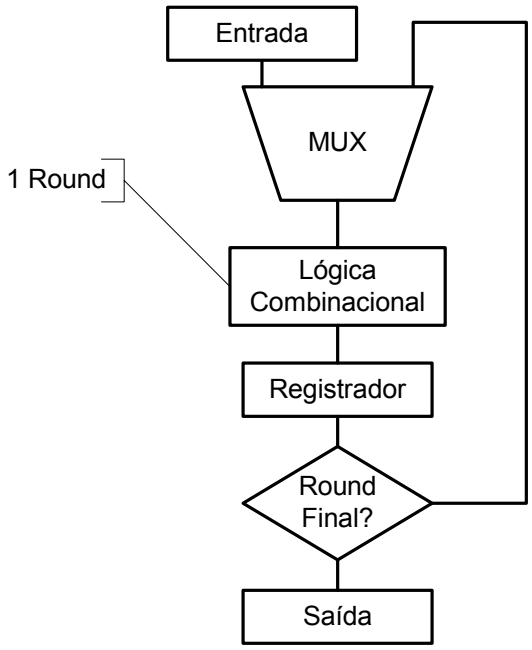


FIGURA 4.11 - Arquitetura básica para implementação de criptografia simétrica

A taxa de transmissão deste tipo de arquitetura pode ser calculada por:

$$t = (\#\ bits \bullet freq\ clock\ max)/(\#\ rodadas) \qquad (72)$$

Ou seja, toma-se o número de bits processados, multiplica-se pela freqüência máxima de operação do circuito, mas divide-se pelo número de rodadas.

Deve-se observar também que não há neste modelo a aplicação do conceito de paralelismo. É claro que pode-se utilizar vários blocos construídos com a arquitetura básica para processar as entradas de maneira simultânea, mas serão mostradas a seguir outras arquiteturas mais eficientes para implementação de processamento paralelo.

### 4.2.2 Iteração com múltiplas rodadas (Loop unrolling)

Segue-se o modelo da arquitetura básica, mas acrescenta-se mais lógica combinacional de maneira a implementar outras rodadas. Assim, a cada passagem pela iteração, “n” rodadas são processadas. É interessante que o número total de rodadas seja um múltiplo do número de rodadas implementadas em cada iteração, facilitando a construção da lógica de controle. A figura a seguir ilustra este método.

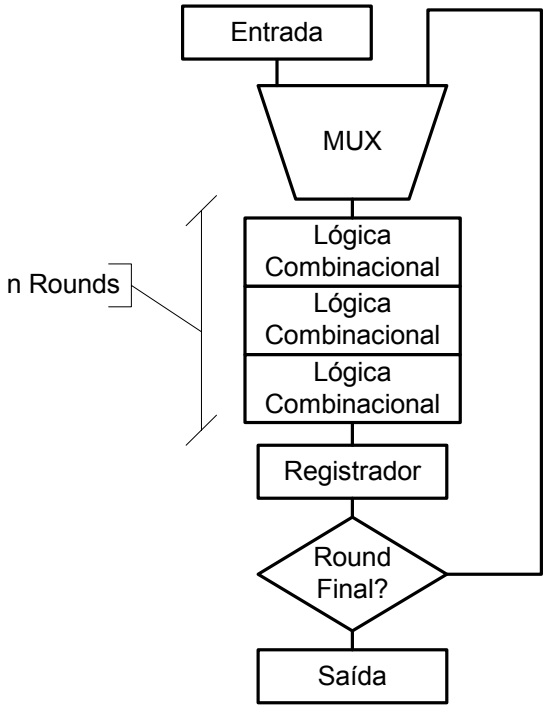


FIGURA 4.12 - Iteração com múltiplas rodadas

Neste modelo, consegue-se um ganho de velocidade, segundo mostrado em [GAJ00], pois a inclusão de novos módulos de lógica combinacional diminui o número de rodadas da expressão 72. Entretanto, a freqüência máxima do circuito diminui, como também aumenta consideravelmente a área ocupada.

Novamente, não há aplicação de paralelismo neste modelo.

### 4.2.3 Paralelismo dentro da rodada (Inner-round pipelining)

Nesta arquitetura são colocados registradores para armazenar resultados temporários dentro da rodada. Desta maneira consegue-se diminuir o tempo necessário para processar cada bloco, ao mesmo tempo que outros blocos também são processados.

As duas figuras a seguir ilustram esta técnica aplicada ao processo de encriptação do Rijndael, no padrão AES-128.

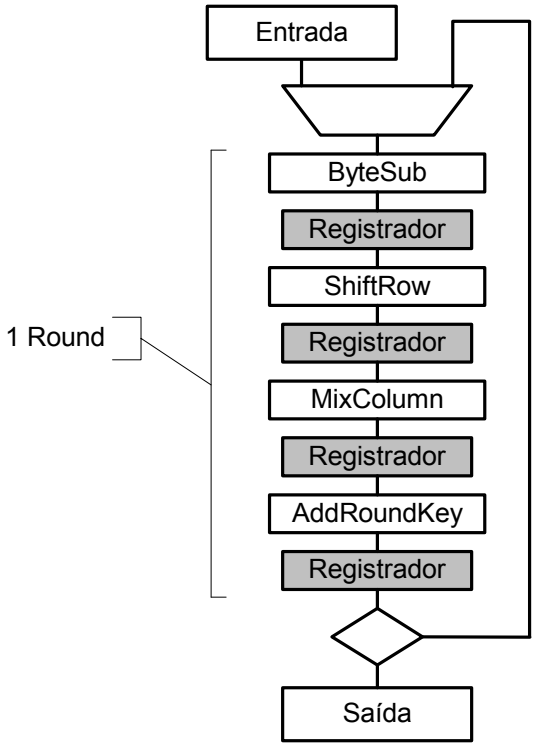


FIGURA 4.13 - Paralelismo dentro da rodada

| **Função \ Clock** | **1** | **2** | **3** | **4** | **5** | **...** | **37** | **38** | **39** | **40** | **41** | **42** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ByteSub | B1 | B2 | B3 | B4 | B1 | | B1 | B2 | B3 | B4 | B5 | B6 |
| ShiftRow | | B1 | B2 | B3 | B4 | | B4 | B1 | B2 | B3 | B4 | B2 |
| MixColumn | | | B1 | B2 | B3 | | B3 | B4 | B1 | B2 | B3 | B4 |
| AddRoundKey | | | | B1 | B2 | | B2 | B3 | B4 | B1 | B2 | B3 |

FIGURA 4.14 - Diagrama temporal da arquitetura de paralelismo dentro da rodada

A cada pulso de relógio, somente uma das transformações do Rijndael é aplicada a cada bloco de bits, aumentando assim a freqüência máxima do circuito em relação à arquitetura básica, pois a lógica combinacional envolvida é menor. Da mesma forma, sempre existem 4 blocos de bits sendo processados ao mesmo tempo. Desta maneira, apesar da necessidade de 4 pulsos para completar uma única rodada, ganha-se velocidade em relação à arquitetura básica, conforme mostra a expressão abaixo.

$$t = (4 \bullet \#bits) \bullet (freq\ clock\ max)/(4 \bullet \#rounds) \qquad (73)$$

Ou seja,

$$t = (\#\ bits \bullet freq\ clock\ max)/(\#\ rodadas) \qquad (74)$$

onde a freqüência máxima de operação do circuito é superior à freqüência alcançada pela arquitetura básica.

Conforme o próprio nome indica, esta arquitetura utiliza técnicas de paralelismo no processamento dos blocos de entrada.

### 4.2.4 Paralelismo fora da rodada (Outer-round pipelining)

A partir da conjunção dos conceitos das arquiteturas de *Loop unrolling* e *Inner-round pipelining*, chega-se à utilização de registradores após cada rodada, dentro de uma iteração onde várias rodadas são agrupadas. A figura abaixo ilustra o conceito.

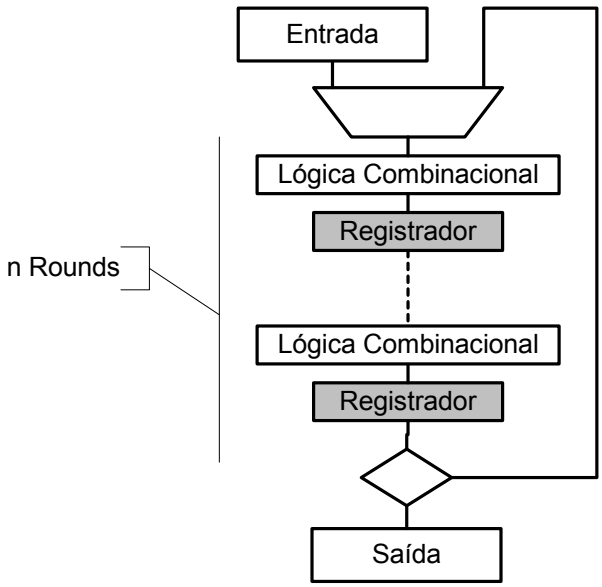


FIGURA 4.15 - Paralelismo fora da rodada

Com um pequeno acréscimo de área em relação ao Loop unrolling, devido ao acréscimo de registradores intermediários, consegue-se processar múltiplos blocos ao mesmo tempo. Entretanto, o tempo total requerido para cada interação aumenta um pouco, em virtude da necessidade de armazenamento nos registradores. Mesmo assim, consegue-se um ganho considerável de velocidade.

### 4.2.5 Comparação de desempenho

Em [GAJ00] pode-se visualizar o gráfico de comparação, levando em consideração velocidade alcançada e área ocupada, entre as quatro arquiteturas descritas anteriormente. O mesmo gráfico é mostrado a seguir..

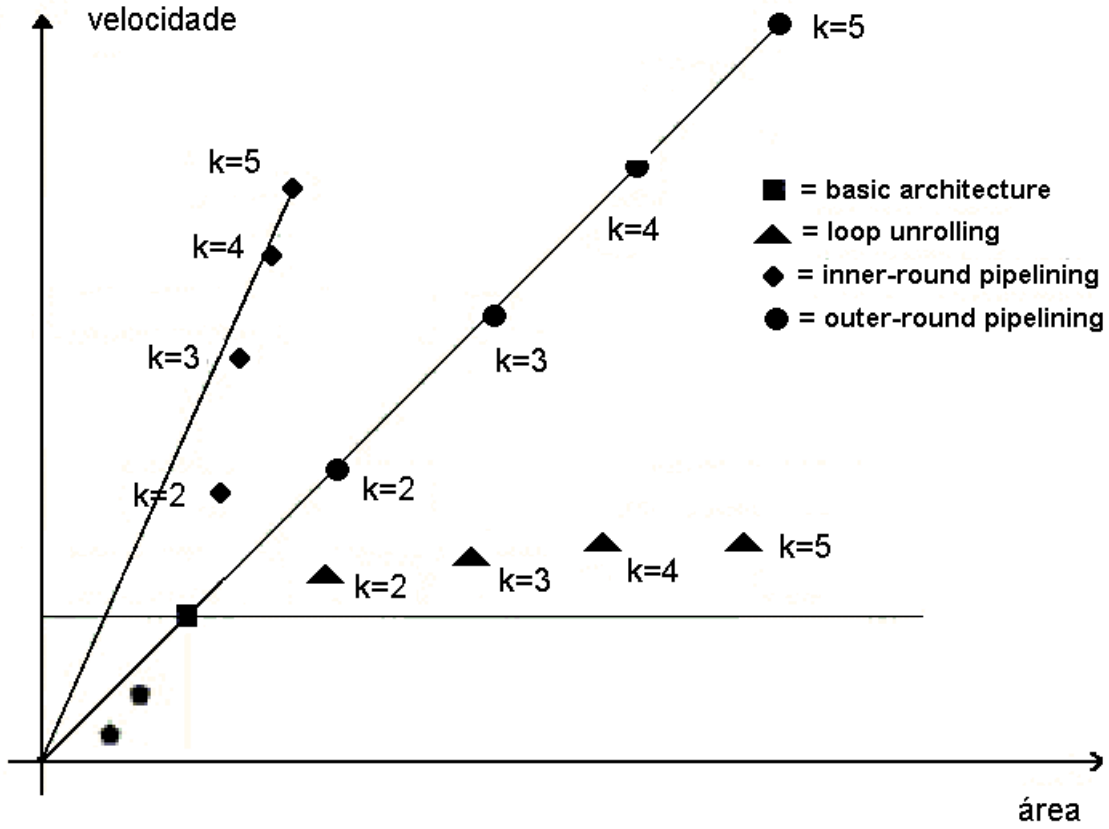


FIGURA 4.16 - Comparação entre velocidade e área para as quatros arquiteturas segundo [GAJ00]

## *4.3 Decisões de projeto*

Os principais objetivos da dissertação, segundo já apresentado na introdução, consistem em construir um conhecimento sólido sobre criptografia em hardware, desenvolver um Soft IP do AES-128 (codificação em VHDL), de maneira a permitir a utilização em outros projetos, como também produzir resultados que permitissem a comparação com outros trabalhos desenvolvidos em diferentes centros de pesquisas. A implementação será efetuada sobre FPGA Altera™.

A decisão do ambiente de desenvolvimento escolhido, mais especificamente quais ferramentas de software e quais componentes utilizar, seguiu a continuidade de trabalhos já desenvolvidos comercialmente em empresas de comunicação de dados do Brasil [PAR 2001], como também trabalhos desenvolvidos ao longo do mestrado em questão. Assim, escolheu-se componentes da Altera™, da família Flex10K [ALT 2000]. Como plataforma de desenvolvimento e verificação VHDL, optou-se pelo software LeonardoSpectrum da Mentor™ [EXE 2001]. A versão utilizada é a fornecida pela Altera, em acordo com a Mentor, chamada de LeonardoSpectrum Level 1 Altera [ALT 2001c]. Para implementação nos componentes, testes de funcionalidade e desempenho, utilizou-se o software MAX+plus II 10.0 da Altera™ [ALT 2001d].

Baseado nos trabalhos de [FIS 2000] e [MRO 2000], decidiu-se implementar o padrão AES-128 em componentes da família Flex10k, pois seria possível comparar resultados. Posteriormente, pode-se adaptar o código para os padrões AES-192 e AES-256.

No momento em que iniciou-se o projeto, não havia ainda trabalhos sobre o algoritmo Rijndael utilizando os novos componentes da Altera™, da família APEX. Utilizando a família Flex10k seria possível comparar os resultados alcançados ao final do projeto com os mostrados nos dois trabalhos acima citados, efetivamente avaliando a qualidade do código produzido. Tal código pode, então, ser utilizado para sintetizar o Rijndael em componentes de outras famílias, ou ainda de outros fabricantes, como Xilinx, por exemplo.

Desde o início, percebeu-se que a concepção de implementação utilizada em [MRO 2000] exigia a utilização do maior componente da família Flex10K, em virtude da quantidade de memória exigida para implementar a ROM que armazena as tabelas S-Box requeridas na transformação ByteSub e na expansão da chave. São 40 kbytes de memória ROM, a qual utiliza a totalidade das *Embedded Array Blocks* (EAB) do componente EPF10k250.

A arquitetura básica, mostrada no tópico anterior, foi a escolhida neste caso. A arquitetura escolhida para implementar o algoritmo Rijndael depende intrinsicamente dos modos de utilização. Segundo mostrado no tópico anterior, as arquiteturas que utilizam o conceito de paralelismo não podem ser utilizadas em modos de operação com realimentação.

Entretanto, a maioria dos algoritmos criptográficos utiliza o modo realimentado [GAJ00]. É o caso das definições do padrão de Redes Privativas Virtuais (VPN) definidas pelo *Internet Engineering Task Force* (IETF),na forma das *Request for Comments* (RFC). Os documentos RFC-1829 [KAR 95], RFC-2405 [MAD 98], RFC-2406 [KEN 98], RFC-2451 [PER 98], padrões "de facto" da indústria, definem sempre algoritmos simétricos no modo CBC.

Desta maneira, priorizou-se a implementação de arquiteturas que permitissem a realimentação, em especial o modo de operação CBC. Das arquiteturas citadas anteriormente, sobraram a arquitetura básica e a iteração com múltiplas rodadas. Contudo, para implementar cada rodada é necessário reservar 40 kbytes de memórias em EAB's. Como esse número é a totalidade de memória disponível no maior componente da família Flex10K, caso fosse utilizada a técnica de iteração com múltiplas rodadas seria necessário um novo componente a cada rodada acrescentada à iteração.

Em virtude do demonstrado acima, as decisões de projeto foram:

- ambiente de desenvolvimento MAX+plus II 10.0 e LeonardoSpectrum;
- implementar o padrão AES-128, que utiliza blocos de entrada e chaves de 128 bits;
- utilizar a arquitetura básica para a implementação do algoritmo Rijndael;
- utilizar a família Flex10K, permitindo comparação de resultados;
- codificação em VHDL, permitindo a reutilização do código;
- geração de uma versão visando performance (*throughput*), na qual os blocos são sempre tratados de 128 em 128 bits; serão gerados dois circuitos: um para encriptação e outro para decriptação;
- geração de uma versão visando uma boa relação custo x benefício entre a área ocupada e a performance atingida; serão gerados dois circuitos: um para encriptação e outro para decriptação.

## 4.4 Ambiente de desenvolvimento

Toda a codificação do circuito utilizou *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) [ASH 96], como linguagem de alto nível, adotada largamente na indústria, e compatível com a grande maioria das ferramentas de síntese do mercado. Para verificação da sintaxe e compilação do projeto utilizou-se então o aplicativo LeonardoSpectrum, gerando uma saída no padrão *Electronic Design Interchange Format* (EDIF). A saída EDIF era então importada pelo software MAX+plus II para síntese nos componentes escolhidos. O MAX+plus II foi também utilizado para simulação dos resultados e verificação das funcionalidades de cada circuito. No Anexo 3 existem figuras ilustrando as diferentes etapas de desenvolvimento do trabalho.

## 4.5 Implementação VHDL das transformações básicas do Rijndael

Todas as implementações, tanto dos encriptadores e decriptadores, quanto das versões de 128 e 32 bits, utilizaram a mesma metodologia. O conceito de biblioteca, através da estrutura *package* em VHDL, foi amplamente utilizado [ALT 96][ASH 96][CAR 90].

A utilização da estrutura de *package* permite que um conjunto de funções possa ser construído, sendo utilizado dinamicamente pelos circuitos implementados. Assim, existia somente um *package*, que era utilizado para a encriptação e decriptação. Nele existiam funções construídas para todas as versões. Como algumas funções não eram utilizadas na implementação de determinado circuito, as ferramentas de síntese simplesmente ignoravam tais construções.

Os circuitos de encriptação e decriptação que visavam velocidade utilizavam funções que processam os blocos de 128 bits de maneira paralela. No caso dos circuitos cuja preocupação era a área ocupada, muitas vezes dividia-se o bloco de 128 bits em sub-blocos de 32 bits, a fim de reaproveitar a lógica construída, mesmo que com um custo em relação à performance. Durante o restante do texto, ao referenciar-se a arquitetura de 128 bits e a arquitetura de 32 bits, deve-se considerar as versões visando performance e área, respectivamente.

A seguir serão mostrados os códigos VHDL utilizados para implementar cada funcionalidade do Rijndael, as quais foram examinadas no Capítulo 3. Nesta codificação algumas regras básicas de construção do VHDL serão ignoradas, a fim de permitir uma melhor visualização.

### 4.5.1 ByteSub

A função ByteSub, descrita em 3.5.1, opera sobre cada byte, sendo na realidade um conjunto de *Read Only Memory* (ROM) de 8 bits de entrada e 8 bits de saída, ou seja de 256 posições X 8 bits. Para cada byte processado é produzida uma ROM de 2048 bits.

Foi criada uma variável que varia de 0 até 255, em decimal, para representar os 8 bits armazenados. Depois criou-se um *array* de 256 posições. Assim, construiu-se um espaço de 2048 bits, arranjados de 8 em 8 bits, perfazendo 256 posições. Após, bastou declarar uma constante de tipo igual ao *array* e cujo conteúdo se igualava à tabela 3.3. Em VHDL essa codificação torna-se:

```
subtype        I256           is integer range 0 to 255; -- 8 bits
type           SBOX_T         is array (0 to 255) of I256; -- array de 256 posições

-- Constante que define a tabela de substituição de bytes --

constant SBOX : SBOX_T := (

 99, 124, 119, 123, 242, 107, 111, 197,  48,   1, 103,  43, 254, 215, 171, 118,
202, 130, 201, 125, 250,  89,  71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
183, 253, 147,  38,  54,  63, 247, 204,  52, 165, 229, 241, 113, 216,  49,  21,
  4, 199,  35, 195,  24, 150,   5, 154,   7,  18, 128, 226, 235,  39, 178, 117,
  9, 131,  44,  26,  27, 110,  90, 160,  82,  59, 214, 179,  41, 227,  47, 132,
 83, 209,   0, 237,  32, 252, 177,  91, 106, 203, 190,  57,  74,  76,  88, 207,
208, 239, 170, 251,  67,  77,  51, 133,  69, 249,   2, 127,  80,  60, 159, 168,
 81, 163,  64, 143, 146, 157,  56, 245, 188, 182, 218,  33,  16, 255, 243, 210,
205,  12,  19, 236,  95, 151,  68,  23, 196, 167, 126,  61, 100,  93,  25, 115,
 96, 129,  79, 220,  34,  42, 144, 136,  70, 238, 184,  20, 222,  94,  11, 219,
224,  50,  58,  10,  73,   6,  36,  92, 194, 211, 172,  98, 145, 149, 228, 121,
231, 200,  55, 109, 141, 213,  78, 169, 108,  86, 244, 234, 101, 122, 174,   8,
186, 120,  37,  46,  28, 166, 180, 198, 232, 221, 116,  31,  75, 189, 139, 138,
112,  62, 181, 102,  72,   3, 246,  14,  97,  53,  87, 185, 134, 193,  29, 158,
225, 248, 152,  17, 105, 217, 142, 148, 155,  30, 135, 233, 206,  85,  40, 223,
140, 161, 137,  13, 191, 230,  66, 104,  65, 153,  45,  15, 176,  84, 187,  22

);
```

Segundo comentado anteriormente, a implementação destas ROM's em EAB's da família Flex10K foi o grande limitador em termos de arquitetura de projeto. A partir do apresentado acima conclui-se que para cada 8 bits processados faz-se necessário 2048 bits de memória ROM. Na iteração principal do Rijndael, a totalidade dos 128 bits do *state* são processados pela função ByteSub. Além disso, na função de expansão da chave, 32 bits também são submetidos às tabelas ROM. Desta maneira, em caso de processamento paralelo e não reaproveitamento de recursos, deve-se prover um total de 40 kbit de memória ROM, pois:

$$(128 + 32) \text{ bits} / 8 \text{ bits} = 20 \qquad (75)$$

$$20 * 2048 \text{ bits} = 40960 \text{ bits} = 40 \text{ kbit} \qquad (76)$$

As funções que implementam o ByteSub são diferentes para as versões de 128 e de 32 bits. Na versão de 128 bits, o *state* é processado paralelamente, conforme pode ser visto no código VHDL abaixo.

```
-- Ao STATE, com suas 4 linhas por 4 colunas, posição por posição, aplica-se a
-- substituição de bytes a cada conjunto de 8 bits, de maneira paralela.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function BYTE_SUB_F ( state : STATE_T )
                      return STATE_T is

variable a      : STATE_T;

begin

 for row in 0 to 3 loop
  for column in 0 to 3 loop
   a(row)(column):=SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(UNSIGNED(state(row)(column)))),8));
  end loop;
 end loop;

 return a;

end BYTE_SUB_F;
```

Na versão de 32 bits, a aplicação é feita de 32 em 32 bits, ou seja, de linha em linha do *state*, conforme mostra a codificação abaixo.

```
-- A cada linha do STATE, com 4 colunas, posição por posição, aplica-se a
-- substituição de bytes a cada conjunto de 8 bits, de maneira paralela.

function BYTE_SUB_F ( state : STATE_ROW_T )
                        return STATE_ROW_T is

variable a      : STATE_ROW_T;

begin

   for column in 0 to 3 loop
      a(column) := SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(unsigned(state(column)))),8));
   end loop;

   return a;

end BYTE_SUB_F;
```

### 4.5.2 ShiftRow

A função ShiftRow, descrita em 3.5.2, é na realidade um deslocamento dos bytes dentro do *state*, seguindo regras dependentes da cada linha que se está processando.

Na arquitetura de 128 bits, a implementação utiliza integralmente o *state*, deslocando de uma só vez todo o conjunto de bytes, conforme mostrado a seguir.

```
-- A troca de posicionamento dos bytes dentro do STATE segue uma regra definida --- em
função da linha que tal byte ocupa. Esta regra é representada pela
-- constante SHIFTR. Note-se que para a primeira linha o deslocamento é zero,
-- para a segunda o deslocamento é 1, e assim por diante.

subtype I4            is integer range 0 to 3;
subtype SLV_8         is std_logic_vector(7 downto 0);
type SHIFT_ROW_T      is array (0 to 3) of I4;
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

constant SHIFTR :     SHIFT_ROW_T := (0,1,2,3);

function SHIFT_ROW_F ( state : STATE_T )
                        return STATE_T is

variable a      : STATE_T;

begin
       for row in 0 to 3 loop
              for column in 0 to 3 loop
                     a (row) (column) := state (row) ((column + SHIFTR(row)) mod 4);
              end loop;
       end loop;

       return a;

end SHIFT_ROW_F;
```

Na arquitetura de 32 bits, o processamento é feito linha a linha do *state*. É importante ressaltar, que diferentemente da transformação ByteSub, na qual a posição da linha não importava, agora faz-se necessário informar qual a posição da linha a ser processada, a fim de definir o deslocamento que deve ser aplicado.

```
-- O deslocamento dos bytes dentro da linha é determinado pela posição da linha
-- no STATE. A informação de posicionamento da linha é dada através do parâmetro
-- inc.

subtype I4            is integer range 0 to 3;
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function SHIFT_ROW_F (        inc : I4;
                              state : STATE_ROW_T )
                              return STATE_ROW_T is

variable a      : STATE_ROW_T;

begin
for column in 0 to 3 loop
               a(column) := state((column + inc) mod 4);
       end loop;

       return a;

end SHIFT_ROW_F;
```

### 4.5.3 MixColumn

A função MixColumn trabalha sobre as colunas do *state*, aplicando a tranformação XTIME sobre alguns dos bytes e fazendo um Ou-Exclusivo ao final. A descrição completa desta função encontra-se em 3.5.3.

```
-- O valor de cada elemento dentro do state, na transformação MixColumn, depende
-- dos demais elementos da coluna. Aplica-se a transformação XTIME_F aos
-- elementos de entrada da mesma posição e da posição imediatamente posterior na
-- coluna. Depois faz-se um Ou-Exclusivo com o restante dos elementos, incluindo
-- também o elemento posterior.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function MIX_COLUMN_F (       state : STATE_T )
                        return STATE_T is

variable b      : STATE_T;

begin

   for row in 0 to 3 loop
      for column in 0 to 3 loop

         b(row)(column) :=
            XTIME_F ( state(row)(column) ) xor
            XTIME_F ( state((row + 1) mod 4)(column) ) xor
             state((row + 1) mod 4)(column) xor
             state ((row + 2) mod 4)(column) xor
            state ((row + 3) mod 4)(column);

      end loop;  -- column
   end loop; -- row

   return b;

end MIX_COLUMN_F;
```

A transformação XTIME representa a multiplicação pelo polinômio x, explicada na seção 3.1.1.3. A seguir mostra-se a codificação VHDL que implementa tal operação.

```
-- A multiplicação pelo polinômio x representa um deslocamento para a esquerda de
-- uma posição. Entretanto, quando o coeficiente de x7 vale 1, significa que ao
-- resultado deve ser aplicado a operação de módulo com o polinômio definido pelo
-- Rijndael, a fim de que a resultante seja ainda um polinômio de grau 7.

subtype SLV_8         is std_logic_vector(7 downto 0);

function XTIME_F (    a : SLV_8 )
                    return SLV_8 is
begin
       if a(7)='1' then
              return SLV_8(unsigned(a) sll 1) xor "00011011";
       else
              return SLV_8(unsigned(a) sll 1);
       end if;
end XTIME_F;
```

Na arquitetura de 32 bits, a transformação é feita coluna por coluna do *state*, conforme é mostrado abaixo.

```
-- O valor de cada elemento dentro do state, na transformação MixColumn, depende
-- dos demais elementos da coluna. Aplica-se a transformação XTIME_F aos
-- elementos de entrada da mesma posição e da posição imediatamente posterior na
-- coluna. Depois faz-se um Ou-Exclusivo com o restante dos elementos, incluindo
-- também o elemento posterior.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function MIX_COLUMN_F (       state : STATE_ROW_T )
                        return STATE_ROW_T is

variable b      : STATE_ROW_T;

begin

       for row in 0 to 3 loop
              b(row):=
              XTIME_F ( state(row)) xor
              XTIME_F ( state((row + 1) mod 4)) xor
              state((row + 1) mod 4)xor
              state ((row + 2) mod 4)xor
              state ((row + 3) mod 4);
       end loop;
       return b;

end MIX_COLUMN_F;
```

### 4.5.4 AddRoundKey

Conforme já foi descrito na seção 3.5.4, a função de AddRoundKey é um Ou-Exclusivo entre o *state* e a chave específica de cada rodada, a *round key*. Assim, a codificação VHDL para a arquitetura de 128 bits é mostrada abaixo.

```
-- A transformação AddRoundKey é um Ou-Exclusivo entre o state e a round key.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function ADD_KEY_F (  state : STATE_T;
                      key : STATE_T )
                      return STATE_T is

variable a      : STATE_T;

begin
   for row in 0 to 3 loop
      for column in 0 to 3 loop
```

```
         a(row)(column) := state (row) (column) XOR key (row) (column);
      end loop;
   end loop;
   return a;
end ADD_KEY_F;
```

Na arquitetura de 32 bits, as entradas são de 32 em 32 bits.

```
-- A transformação AddRoundKey é um Ou-Exclusivo (XOR) entre o state e a round key.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function ADD_KEY_F (  state : STATE_ROW_T;
                      key : STATE_ROW_T )
                      return STATE_ROW_T is

variable a      : STATE_ROW_T;

begin
   for column in 0 to 3 loop
      a(column) := state(column) XOR key(column);
   end loop;
   return a;
end ADD_KEY_F;
```

### 4.5.5 Expansion Key

A partir do bloco de bits da chave inicial são gerados outros conjuntos, que serão utilizados a cada rodada na transformação AddRoundKey. A explicação detalhada da função de expansão de chave pode ser encontrada na seção 3.6.

Como no AES-128, tanto a chave quanto o bloco de bits possuem 128 bits, e a fim de realizar a transformação AddRoundKey de maneira simples e rápida, optou-se por gerar as chaves de rodada já na mesma estrutura do *state*. Tal organização diferencia-se um pouco da estrutura definida pelo Rijndael, na qual as chaves estariam armazenadas em um *array* unidimensional de 32 bits. No UFIR-128 são utilizadas estruturas bidimencionais que armazenam 8 bits a cada posição, a exemplo do *state*. Entretanto, tais estruturas são equivalentes, sendo que a última possui a vantagem de estar pronta para interagir com o bloco de bits *state*. É importante salientar que tal solução foi aplicada para o padrão AES-128, e pode sofrer alterações nas outras versões.

Foi implementada uma função ROUND_KEY, responsável por, a partir de uma dada chave de rodada, gerar a próxima. É necessário fornecer também o posicionamento das chaves dentro da expansão, pois a próxima chave não depende somente do valor da chave anterior, mas também da posição que será ocupada dentro do conjunto de chaves expandidas.

É importante notar que antes de efetuar qualquer operação, como também ao sair da função, o posicionamento de linhas e colunas é invertido. Isto é feito para manter a compatibilidade com a definição do tipo *STATE*_T, utilizado em todo o desenvolvimento, e permitir operar sobre as colunas da estrutura, pois as operações da expansão da chave são assim definidas. Com exceção da primeira coluna, que sofre outra transformação chamada KS_TRAN, que foi também explicada na seção 3.6.1, as colunas restantes são encontradas a partir de um Ou-Exclusivo entre a coluna anterior e a coluna 4 posições atrás.

```
-- Para gerar uma nova round key é preciso fornecer como parâmetros da função a
-- round key anterior e a posição da round key anterior. Quando a round anterior
-- for a chave inicial a posição a ser colocada é zero.

subtype I16           is integer range 0 to 15;
subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function ROUND_KEY_F ( i : I16;
                       rkey : STATE_T )
                       return STATE_T is

variable a      : STATE_T;
variable b      : STATE_T;

begin

        for row in 0 to 3 loop
                for column in 0 to 3 loop
                        b(row)(column) := rkey(column)(row);
                end loop;
        end loop;
        a(0) := KS_TRAN_F2(i,b(3),b(0));
        for column in 0 to 3 loop
                a(1)(column) := a(0)(column) xor b(1)(column);
                a(2)(column) := a(1)(column) xor b(2)(column);
                a(3)(column) := a(2)(column) xor b(3)(column);
        end loop;
        for row in 0 to 3 loop
                for column in 0 to 3 loop
                        b(row)(column) := a(column)(row);
                end loop;
        end loop;
        return b;

end ROUND_KEY_F;
```

A função KS_TRAN utilizada no bloco acima, implementa também substituições de bytes usando a transformação ByteSub. Entretanto, tais substituições são aplicadas somente a quatro (4) conjuntos de 8 bits, totalizando uma memória ROM de 8192 bits, ou 8 Kbit. Outra questão a ser ressaltada é que a posição da chave na estrutura da expansão, representada pelo índice i, modifica o valor final.

```
-- Para gerar uma nova round key é preciso fornecer como parâmetros da função a round --
-- key anterior e a posição da round key anterior. Quando a round anterior for a chave -
-- inicial a posição a ser colocada é zero.

subtype I16           is integer range 0 to 15;
subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function KS_TRAN_F2 (    i      : I16;
                         w1          : STATE_ROW_T;
                         w4          : STATE_ROW_T )
                         return STATE_ROW_T is

variable a : SLV_8;
variable b : STATE_ROW_T;

begin

        case i is
                when 0 =>
                        a := X"01";
                when 1 =>
                        a := X"02";
                when 2 =>
                        a := X"04";
                when 3 =>
                        a := X"08";
                when 4 =>
                        a := X"10";
```

```
                when 5 =>
                        a := X"20";
                when 6 =>
                        a := X"40";
                when 7 =>
                        a := X"80";
                when 8 =>
                        a := X"1B";
                when 9 =>
                        a := X"36";
                when others =>
                        a := X"00";
        end case;
        b(0) := SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(unsigned(w1(1)))), 8)) xor w4(0) xor a;
        b(1) := SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(unsigned(w1(2)))), 8)) xor w4(1);
        b(2) := SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(unsigned(w1(3)))), 8)) xor w4(2);
        b(3) := SLV_8(TO_UNSIGNED(SBOX(TO_INTEGER(unsigned(w1(0)))), 8)) xor w4(3);
        return b;

end KS_TRAN_F2;
```

### 4.5.6 Inverse ByteSub

A função Inverse ByteSub, descrita em 3.7.1, é o inverso da função ByteSub. Novamente, a função reflete uma *Read Only Memory* (ROM) de 2048 bits para cada byte processado. Todas as questões apresentadas anteriormente são novamente aplicáveis.

De maneira semelhante à função ByteSub, faz-se necessário criar uma ROM que contenha a tabela inversa. A codificação VHDL utilizada pode ser observada a seguir.

```
subtype        I256           is integer range 0 to 255; -- 8 bits
type           SBOX_T         is array (0 to 255) of I256; -- array de 256 posições

-- Constante que define a tabela inversa de substituição de bytes --

constant ISBOX : SBOX_T := (

 82,   9, 106, 213,  48,  54, 165,  56, 191,  64, 163, 158, 129, 243, 215, 251,
124, 227,  57, 130, 155,  47, 255, 135,  52, 142,  67,  68, 196, 222, 233, 203,
 84, 123, 148,  50, 166, 194,  35,  61, 238,  76, 149,  11,  66, 250, 195,  78,
  8,  46, 161, 102,  40, 217,  36, 178, 118,  91, 162,  73, 109, 139, 209,  37,
114, 248, 246, 100, 134, 104, 152,  22, 212, 164,  92, 204,  93, 101, 182, 146,
108, 112,  72,  80, 253, 237, 185, 218,  94,  21,  70,  87, 167, 141, 157, 132,
144, 216, 171,   0, 140, 188, 211,  10, 247, 228,  88,   5, 184, 179,  69,   6,
208,  44,  30, 143, 202,  63,  15,   2, 193, 175, 189,   3,   1,  19, 138, 107,
 58, 145,  17,  65,  79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115,
150, 172, 116,  34, 231, 173,  53, 133, 226, 249,  55, 232,  28, 117, 223, 110,
 71, 241,  26, 113,  29,  41, 197, 137, 111, 183,  98,  14, 170,  24, 190,  27,
252,  86,  62,  75, 198, 210, 121,  32, 154, 219, 192, 254, 120, 205,  90, 244,
 31, 221, 168,  51, 136,   7, 199,  49, 177,  18,  16,  89,  39, 128, 236,  95,
 96,  81, 127, 169,  25, 181,  74,  13,  45, 229, 122, 159, 147, 201, 156, 239,
160, 224,  59,  77, 174,  42, 245, 176, 200, 235, 187,  60, 131,  83, 153,  97,
 23,  43,   4, 126, 186, 119, 214,  38, 225, 105,  20,  99,  85,  33,  12, 125

);
```

As funções que implementam o Inverse ByteSub também são diferentes para as versões de 128 e de 32 bits, como era de se esperar. A seguir ilustra-se a codificação para a arquitetura de 128 bits.

```
-- Ao STATE, com suas 4 linhas por 4 colunas, posição por posição, aplica-se a
-- substituição inversa de bytes a cada conjunto de 8 bits, de maneira paralela.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function IBYTE_SUB_F ( state : STATE_T )
                       return STATE_T is

variable a      : STATE_T;

begin

 for row in 0 to 3 loop
  for column in 0 to 3 loop
   a(row)(column) :=
SLV_8(TO_UNSIGNED(ISBOX(TO_INTEGER(unsigned(state(row)(column)))),8));
  end loop;
 end loop;

 return a;

end IBYTE_SUB_F;
```

Na versão de 32 bits, a aplicação da função inversa é feita de 32 em 32 bits, ou seja, de linha em linha do *state*, conforme mostra a codificação abaixo.

```
-- A cada linha do STATE, com 4 colunas, posição por posição, aplica-se a
-- substituição invesa de bytes a cada conjunto de 8 bits, de maneira paralela.

function IBYTE_SUB_F (        state : STATE_ROW_T )
                        return STATE_ROW_T is

variable a      : STATE_ROW_T;

begin

   for column in 0 to 3 loop
      a(column) := SLV_8(TO_UNSIGNED(ISBOX(TO_INTEGER(unsigned(state(column)))),8));
   end loop;

   return a;

end IBYTE_SUB_F;
```

### 4.5.7 Inverse ShiftRow

A função Inverse ShiftRow, descrita em 3.7.2, desloca os bytes dentro do *state*, de maneira a desfazer as mudanças provocadas pela função ShiftRow. Para tanto, os deslocamentos seguem as regras definidas pelo algoritmo, as quais são dependentes da posição da linha no *state*.

Na arquitetura que prioriza a performance, a implementação utiliza integralmente o *state*, deslocando de uma só vez todo o conjunto de bytes, conforme mostrado a seguir.

```
-- A troca de posicionamento dos bytes dentro do STATE segue uma regra definida
-- em função da linha que tal byte ocupa. Esta regra é representada pela
-- constante ISHIFTR. Note-se agora o inverso da função SHIFT_ROW.

subtype I4            is integer range 0 to 3;
subtype SLV_8         is std_logic_vector(7 downto 0);
type SHIFT_ROW_T      is array (0 to 3) of I4;
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

constant ISHIFTR : SHIFT_ROW_T := (0,3,2,1);
```

```
function ISHIFT_ROW_F ( state : STATE_T )
                        return STATE_T is

variable a      : STATE_T;

begin
       for row in 0 to 3 loop
              for column in 0 to 3 loop
                 a (row) (column) := state (row) ((column + ISHIFTR(row)) mod 4);
              end loop;
       end loop;

       return a;

end ISHIFT_ROW_F;
```

Na arquitetura que prioriza a área, o processamento é feito linha a linha do *state*. Novamente, como o deslocamento é variável dependendo da posição da linha dentro do *state*, é necessário fornecer tal informação através do parâmetro inc.

```
-- O deslocamente dos bytes dentro da linha é determinado pela posição da linha
-- no STATE. A informação de deslocamento é dada através do parâmetro inc.

subtype I4            is integer range 0 to 3;
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function ISHIFT_ROW_F ( inc : I4;
                                              state : STATE_ROW_T )
                        return STATE_ROW_T is

variable a      : STATE_ROW_T;

begin
       for column in 0 to 3 loop
              a (column) := state ((column + 4 - inc) mod 4);
       end loop;
       return a;

end ISHIFT_ROW_F;
```

### 4.5.8 Inverse MixColumn

A função Inverse MixColumn é responsável por reverter a transformação MixColumn. Várias transformações são necessárias, de maneira que a função Inverse MixColumn é uma das mais trabalhosas em todo o Rijndael. A descrição completa desta função encontra-se em 3.7.3.

Na função Inverse MixColumn utilizam-se quatro (4) transformações: ETIME, BTIME, DTIME e NINETIME. Tais transformações implementam, respectivamente, a multiplicação no espaço $GF2^8$ pelos polinômios representados pelos coeficientes #0Eh, #0Bh , #0Dh e #09h.

Abaixo é mostrada a codificação VHDL para a arquitetura de 128 bits, na qual o *state* é manipulado de uma só vez.

```
-- O valor de cada elemento dentro do state, na transformação MixColumn, depende
-- dos demais elementos da coluna. Aplicam-se as transformações ETIME_F, BTIME_F,
-- DTIME_F, NINETIME_F sobre os elementos da coluna e faz-se um Ou-Exclusivo
-- entre eles.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;
type STATE_T          is array (0 to 3) of STATE_ROW_T;

function IMIX_COLUMN_F ( state : STATE_T )
                         return STATE_T is
```

```
variable b      : STATE_T;

begin

   for row in 0 to 3 loop
      for column in 0 to 3 loop

         b(row)(column) :=
            ETIME_F ( state(row)(column) ) xor
            BTIME_F ( state((row + 1) mod 4)(column) ) xor
            DTIME_F ( state((row + 2) mod 4)(column) ) xor
            NINETIME_F ( state((row + 3) mod 4)(column) );

      end loop;  -- column
   end loop; -- row

   return b;

end IMIX_COLUMN_F;
```

Na arquitetura de 32 bits a transformação é feita coluna por coluna do *state*, conforme é mostrado abaixo.

```
-- O valor de cada elemento dentro do state, na transformação MixColumn, depende
-- dos demais elementos da coluna. Aplicam-se as transformações ETIME_F, BTIME_F,
-- DTIME_F, NINETIME_F sobre os elementos da coluna e faz-se um Ou-Exclusivo
-- entre eles.

subtype SLV_8         is std_logic_vector(7 downto 0);
type STATE_ROW_T      is array (0 to 3) of SLV_8;

function IMIX_COLUMN_F ( state : STATE_ROW_T )
                         return STATE_ROW_T is

variable b      : STATE_ROW_T;

begin

        for row in 0 to 3 loop
                b(row):=
                        ETIME_F (state(row)) xor
                        BTIME_F (state((row + 1) mod 4)) xor
                        DTIME_F (state((row + 2) mod 4)) xor
                        NINETIME_F (state((row + 3) mod 4));
        end loop; -- row
        return b;

end IMIX_COLUMN_F;
```

Abaixo é mostrado o código VHDL que implementa as transformações ETIME, BTIME, DTIME e NINETIME. Tais funções são implementadas a partir da função XTIME,utilizando-se as propriedades definidas em 3.1.1.3.

```
-- A multiplicação pelos polinômios representados por 0x0E, 0x0B, 0x0D e 0x09 são
-- feitos utilizando a técnica de adição de resultados intermediários.

subtype SLV_8         is std_logic_vector(7 downto 0);

-- ****************************************************************
-- * Funcao    : ETIME_F                                          *
-- * Descricao : Multiplicacao pelo polinomio "X^3 + X^2 + X"      *
-- ****************************************************************
function ETIME_F ( a : SLV_8 )
                   return SLV_8 is

variable a1 : SLV_8;  -- Multiplica por X
```

```
variable a2 : SLV_8;  -- Multiplica por X^2
variable a3 : SLV_8;  -- Multiplica por X^3

begin
       a1 := XTIME_F(a);
       a2 := XTIME_F(a1);
       a3 := XTIME_F(a2);
       return SLV_8(a1 xor a2 xor a3); -- multip 0x0E
end ETIME_F;

-- ******************************************************************
-- * Funcao    : BTIME_F                                            *
-- * Descricao : Multiplicacao pelo polinomio "X^3 + X + 1"          *
-- ******************************************************************
function BTIME_F (    a : SLV_8 )
                    return SLV_8 is

variable a1 : SLV_8;  -- Multiplica por X
variable a2 : SLV_8;  -- Multiplica por X^2
variable a3 : SLV_8;  -- Multiplica por X^3

begin
       a1 := XTIME_F(a);
       a2 := XTIME_F(a1);
       a3 := XTIME_F(a2);
       return SLV_8(a xor a1 xor a3); -- multip 0x0B
end BTIME_F;

-- ******************************************************************
-- * Funcao    : DTIME_F                                            *
-- * Descricao : Multiplicacao pelo polinomio "X^3 + X2 + 1"         *
-- ******************************************************************
function DTIME_F (    a : SLV_8 )
                    return SLV_8 is

variable a1 : SLV_8;  -- Multiplica por X
variable a2 : SLV_8;  -- Multiplica por X^2
variable a3 : SLV_8;  -- Multiplica por X^3

begin
       a1 := XTIME_F(a);
       a2 := XTIME_F(a1);
       a3 := XTIME_F(a2);
       return SLV_8(a xor a2 xor a3); -- multip 0x0D
end DTIME_F;

-- ******************************************************************
-- * Funcao    : NINETIME_F (9TIME_F)                               *
-- * Descricao : Multiplicacao pelo polinomio "X^3 + 1"              *
-- ******************************************************************
function NINETIME_F ( a : SLV_8 )
                    return SLV_8 is

variable a1 : SLV_8;  -- Multiplica por X
variable a2 : SLV_8;  -- Multiplica por X^2
variable a3 : SLV_8;  -- Multiplica por X^3

begin
       a1 := XTIME_F(a);
       a2 := XTIME_F(a1);
       a3 := XTIME_F(a2);
       return SLV_8(a xor a3); -- multip 0x0D
end NINETIME_F;
```

## 4.6 Arquitetura e bloco de controle da versão de 128 bits

A implementação da versão de 128 bits pode ser compreendida partindo-se do diagrama em blocos mostrado abaixo, que serve tanto para o encriptador quanto decriptador. Maiores detalhes podem ser encontrados no Anexo 1, que aborda os circuitos ENC128 e DEC128.

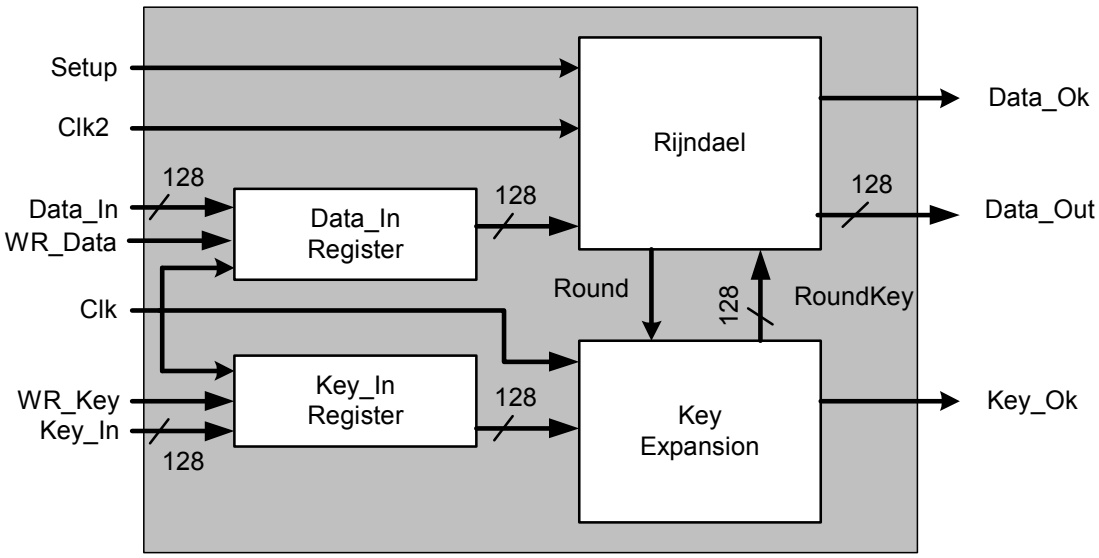


FIGURA 4.17 - Blocos funcionais da implementação de 128 bits

O processo principal, denominado Rijndael, executa uma rodada inteira do algoritmo homônimo, perfazendo as quatro (4) transformações em um único ciclo, determinado pelo pulso de relógio Clk2. Dependendo da função, se encriptador ou decriptador, o processo é alterado em rodadas específicas para satisfazer o algoritmo. Desta maneira, esse processo difere nas versões de encriptação e decriptação. A idéia principal deste módulo é trabalhar continuamente, com velocidade determinada por Clk2.

Na primeira rodada, o dado de entrada já é processado e armazenado internamente. A chaves específicas de cada rodada são retiradas diretamente da memória interna, criada pela função Key Expansion. A cada dez (10) rodadas, 128 bits são transformados utilizando o padrão AES-128 e entregues na interface de saída. Tanto a sinalização de que a saída está pronta, quanto a de que deve-se colocar uma nova entrada, é feita pelo sinal Data_OK. Ou seja, a cada dezena de pulsos em Clk2 um novo dado deve ser escrito e a saída transformada do dado anterior estará pronta.

Por isso, o processo Data_In Register é independente, trabalhando paralelamente ao processo Rijndael. É responsabilidade do processador externo escrever novos dados que serão utilizados para a próxima transformação. Este processo é idêntico para a encriptação e decriptação. Os dados são escritos em blocos de 128 bits.

Desta maneira, quando não é necessário trocar a chave utilizada, o mecanismo funciona sem interrupções e com a maior velocidade possível. Ciclos inteiros de relógio não são perdidos para carregar novos dados.

Entretanto, em determinados instantes é preciso carregar a chave principal, como por exemplo no início da operação. Para tanto, utiliza-se o processo Key_In Register, o

qual armazena a chave principal e automaticamente gera toda a expansão da chave, armazenando internamente os valores que serão utilizados pelo processo Rijndael. Esses processos também são independentes e são feitos paralelamente ao processo principal. Como também, não diferem entre a encriptação e decriptação.

Dois sinais, Clk e Clk2, foram utilizados, pois o período mínimo de operação do processo Rijndael era menor do que o período mínimo do processo de expansão da chave. Como a expansão da chave ocorre somente em determinados intervalos, optou-se por separar os sinais de relógio a fim de garantir uma maior velocidade de operação quando o processo Rijndael é executado em modo contínuo, o que acontece na maior parte do tempo.

Contudo, a fim de sincronizar as operações quando a chave é trocada, faz-se necessário acionar o sinal de Setup, o qual irá forçar o retorno do processo Rijndael para o seu começo. O processamento começará então com um conjunto conhecido de chaves expandidas e com o dado correto a ser transformado.

### 4.6.1 Rijndael na encriptação de 128 bits

Na encriptação de 128 bits o processo Rijndael recebe os 128 bits do dado de entrada, como também a chave de rodada específica, que é indexada por um contador interno que controla a rodada corrente. Depois de dez (10) iterações completas, são apresentados os 128 bits de saída e o sinal Data_OK sofre um pulso.

As transformações são aplicadas segundo o diagrama a seguir.

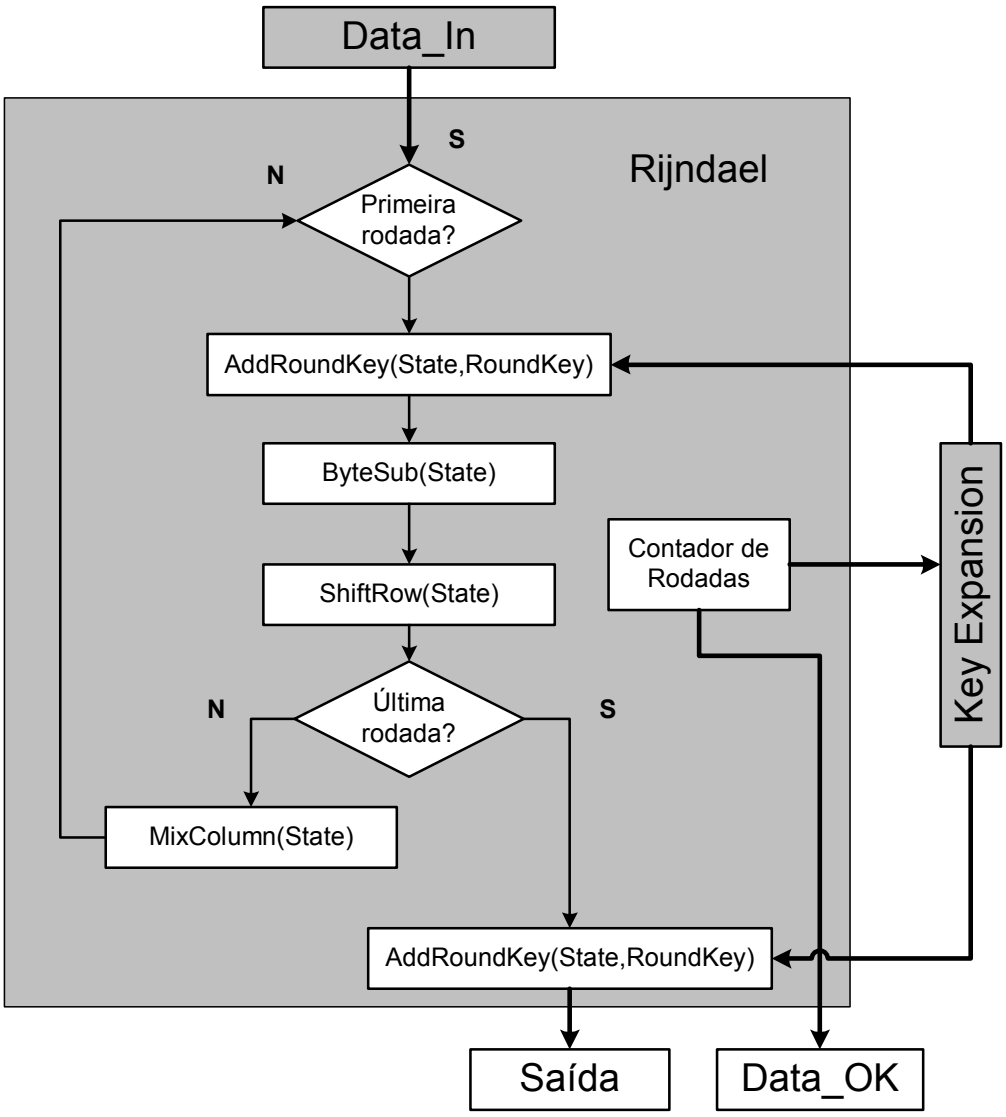


FIGURA 4.18 - Processo Rijndael no ENC128

### 4.6.2 Rijndael na decriptação de 128 bits

Na decriptação de 128 bits, a interface e o funcionamento do processo Rijndael são muito semelhantes aos da encriptação. Modificam-se somente a ordem e as transformações utilizadas.

As transformações na decriptação são aplicadas segundo o diagrama abaixo.

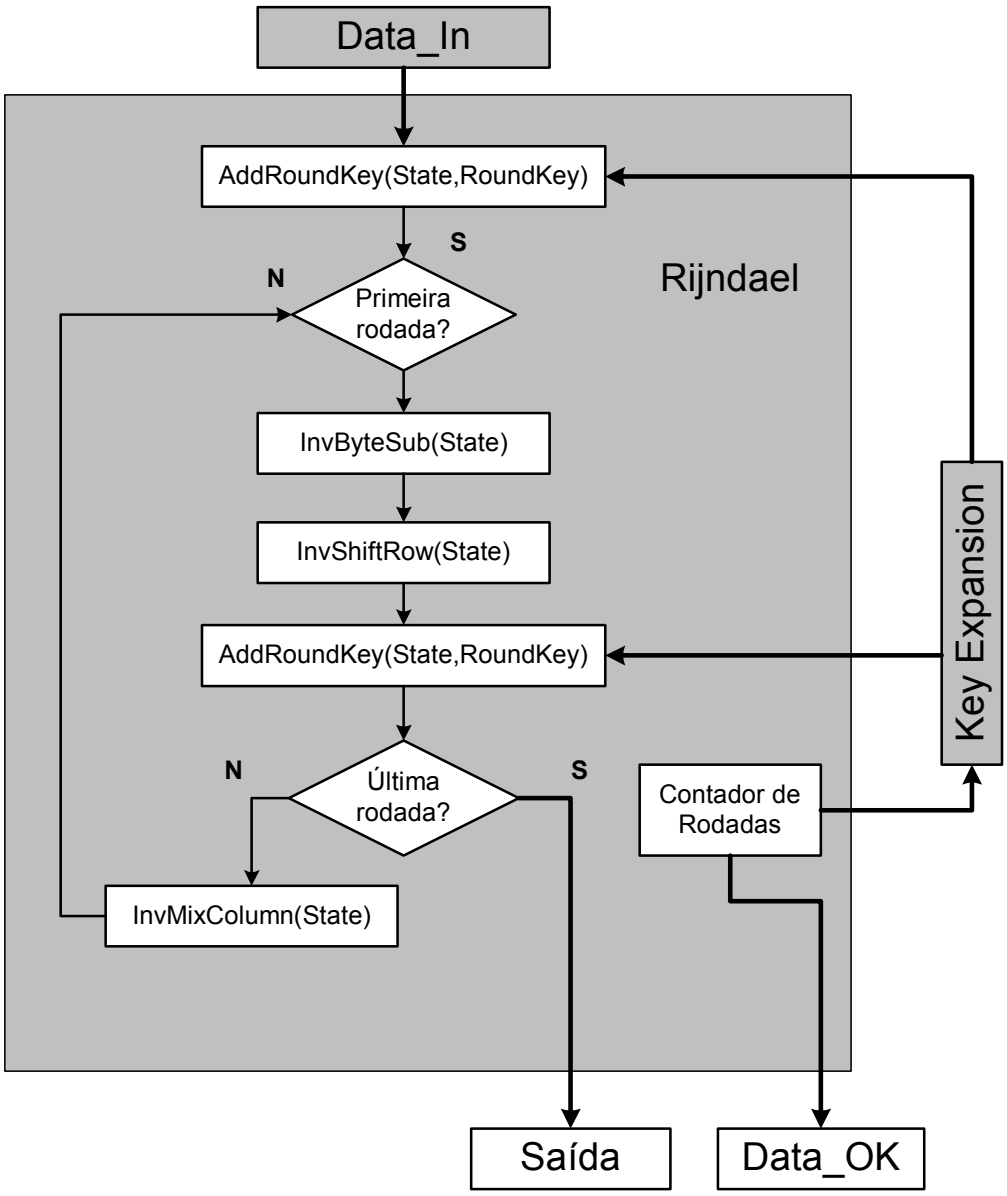


FIGURA 4.19 - Processo Rijndael no DEC128

## *4.7 Arquitetura e bloco de controle da versão de 32 bits*

A implementação da versão de 32 bits difere da codificação anterior, podendo ser compreendida partindo-se do diagrama em blocos mostrado abaixo, que serve tanto para o encriptador quanto decriptador. Maiores detalhes podem ser encontrados no Anexo 2, que aborda os circuitos ENC32 e DEC32.

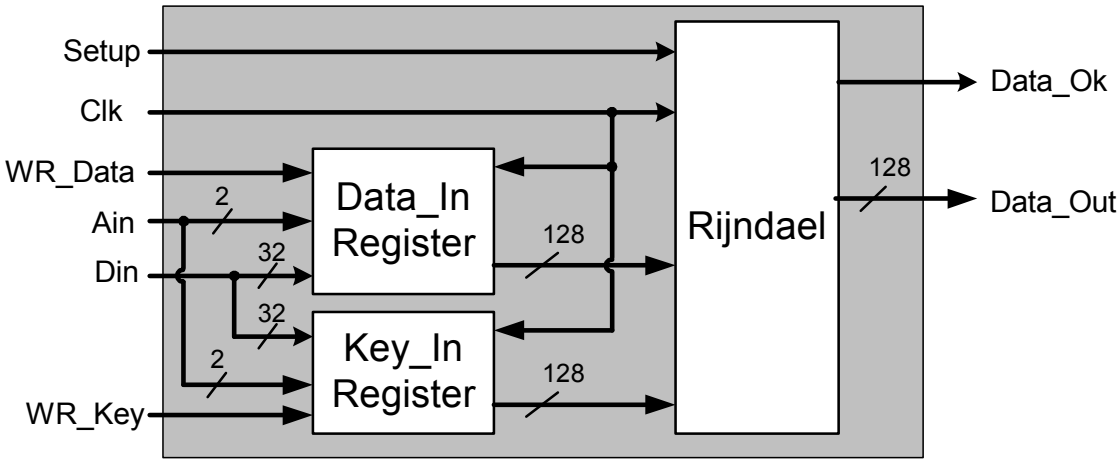


FIGURA 4.20 - Blocos funcionais da implementação de 32 bits

O processo principal continua sendo denominado Rijndael. Entretanto, agora são necessários vários ciclos de relógio para executar uma rodada inteira do algoritmo, uma vez que as transformações necessitam ser feitas de 32 em 32 bits. Apesar da necessidade de múltiplos ciclos, a idéia principal deste módulo continua inalterada, ou seja, trabalhar continuamente, com velocidade determinada por Clk.

Uma grande diferença em relação à implementação de 128 bits é a geração das várias *round keys*. Naquele caso, as chaves eram geradas independentemente do processo Rijndael e armazenadas em memória interna específica. O Rijndael então indexava tal memória a fim de utilizar a *round key* correta. Na arquitetura de 32 bits, as *round keys* são geradas pelo próprio processo Rijndael, de maneira seqüencial. A cada iteração do algoritmo, uma nova round key é gerada a partir da atual.

Desta maneira é necessário fornecer a última chave de rodada no caso da decriptação, pois internamente a chave anterior será gerada. Tal tarefa fica como responsabilidade do controle central. Na encriptação, a chave normal é fornecida, ou seja, a *cipher key* definida usualmente.

Cada rodada necessita doze (12) ciclos de relógio nas rodadas normais. A primeira e última rodadas requerem números diferentes de ciclos de relógio em virtude da função desempenhada, se encriptação ou decriptação. A cada dez (10) rodadas, 128 bits são transformados utilizando o padrão AES-128 e entregues na interface de saída. Tanto a sinalização de que a saída está pronta, quanto de que deve-se colocar uma nova entrada é feita pelo sinal Data_OK.

Os processos Data_In Register e Key_In Register são novamente independentes, trabalhando paralelamente ao processo Rijndael. É responsabilidade do processador externo escrever novos dados que serão utilizados para a próxima transformação. Estes processos são idênticos para a encriptação e decriptação. Os dados, contudo, são escritos em blocos de 32 bits, através da interface Din, sendo que o posicionamento é determinado pelos bits de endereçamento Ain. Quando o sinal WR_Data é acionado, o barramento escreve um novo dado. Da mesma forma, quando WR_Key é acionado, o barramento escreve uma nova chave.

Como o processo Rijndael gera dinamicamente cada nova *round key* a partir da atual, não se faz necessária a geração antecipada de todas as chaves utilizadas nas diversas rodadas. Desta maneira, a cada dez (10) rodadas, que representam uma iteração

completa do Rijndael sobre 128 bits, uma nova chave, bem como um novo dado, podem ser utilizados.

Mesmo assim, a exemplo da arquitetura de 128 bits, pode-se acionar o sinal de Setup, o qual irá forçar o retorno do processo Rijndael para o seu começo.

### 4.7.1 Rijndael na encriptação de 32 em 32 bits

Na encriptação de 32 em 32 bits o processo Rijndael utiliza uma máquina de estados de quatro (4) posições. Cada um destes estados executa uma transformação específica do algoritmo.

Entretanto, como cada transformação atua sobre o *state*, que é por natureza de 128 bits, é necessária outra máquina de estados interna, também de quatro (4) posições, a fim de processar os 128 bits em blocos de 32 bits.

Por questões de facilidade de entendimento, a figura abaixo irá representar somente a primeira máquina de estados citada. Deve-se, todavia, lembrar que cada um dos estados possui outra máquina de estados interna, e que portanto quatro (4) ciclos de relógio são necessários a cada estado visualizado na figura.

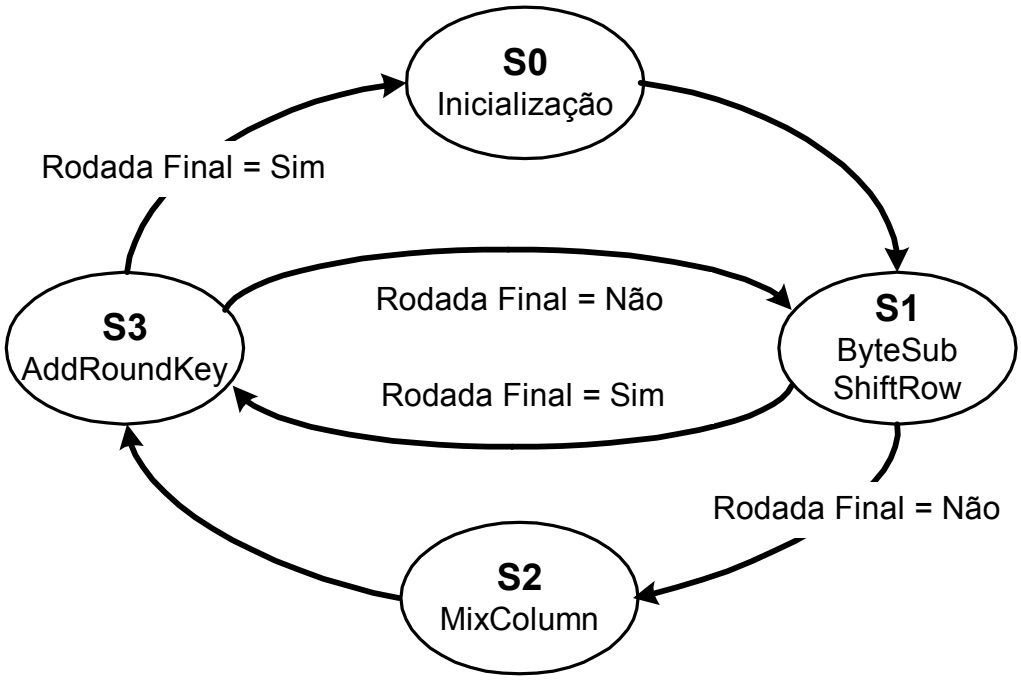


FIGURA 4.21 - Máquina de estados do ENC32

No estado S0, além da inicialização de dados e chaves, já é feita a primeira transformação AddRoundKey.

Na primeira rodada são necessários 16 pulsos de relógios para percorrer todos os estados – S0, S1, S2 e S3. No restante do processo somente os estados S1, S2 e S3 são utilizados, perfazendo 12 ciclos. Na rodada final somente 8 ciclos de relógio são necessários, pois o estado S2 é ignorado. Desta maneira, todo o processo de encriptação de 128 bits usando o padrão AES-128 consome 120 ciclos de relógio.

### 4.7.2 Rijndael na decriptação de 32 em 32 bits

Na decriptação de 32 em 32 bits o processo Rijndael também utiliza uma máquina de estados de quatro (4) posições, onde as transformações inversas vão sendo aplicadas. De maneira semelhante, são necessárias outras máquinas de estados internas, também de quatro (4) posições, a fim de processar os 128 bits em blocos de 32 bits.

Analogamente, a figura abaixo irá representar somente a primeira máquina de estados. Deve-se lembrar que quatro (4) ciclos de relógio são necessários a cada estado visualizado na figura.

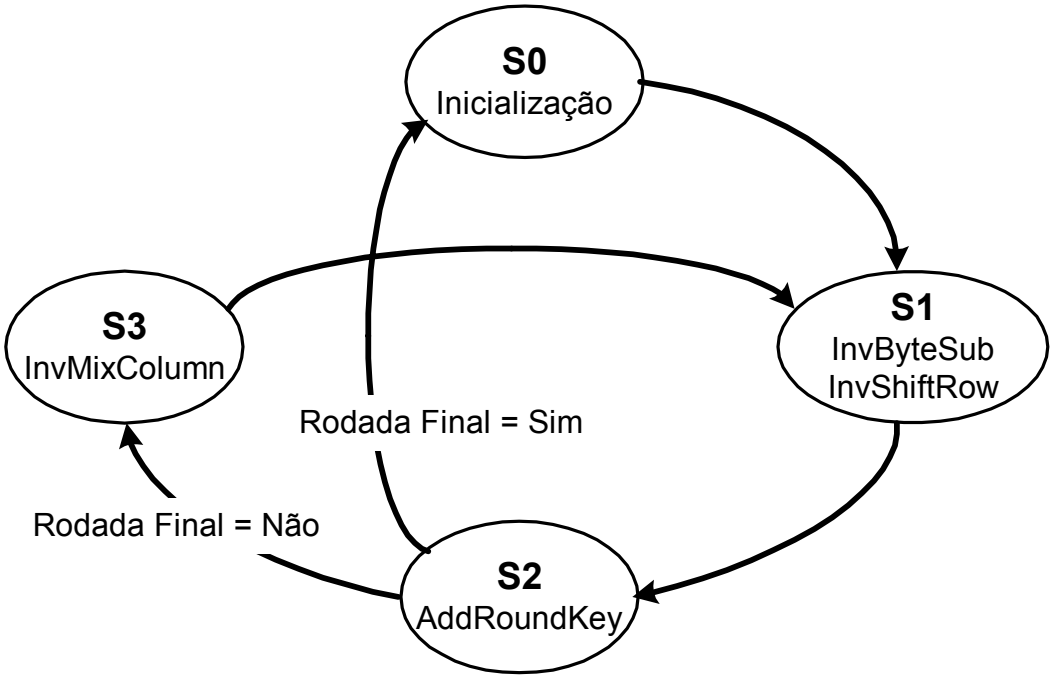


FIGURA 4.22 - Máquina de estados do DEC32

Novamente, no estado S0, além da inicialização de dados e chaves, já é feita a primeira transformação AddRoundKey.

Na primeira rodada da decriptação, são necessários 8 pulsos de relógio para percorrer os estados S0 e S1. As demais rodadas utilizam os estados S2, S3 e S1, perfazendo 12 ciclos. Na rodada final, o estado S2 repete-se, somando 16 ciclos de relógio. Desta maneira, todo o processo de decriptação de 128 bits usando o padrão AES-128 também consome 120 ciclos de relógio, igual ao processo de encriptação.

# 5 Resultados

Este capítulo apresenta os resultados das duas (2) arquiteturas implementadas: versão 128 bits e versão 32 bits. No caso da arquitetura de 128 bits, na qual privilegia-se a performance em detrimento da área, pode-se comparar os resultados obtidos, pois foram encontrados trabalhos semelhantes. Na versão de 32 bits são mostrados os valores absolutos, atestando a efetiva economia de área em componentes.

Apesar do desenvolvimento do código VHDL ter sido feito no software Leonardo Spectrum, toda a simulação de funcionalidade do código utilizou o software Max+plus II. Para tanto, utilizam-se vetores de testes disponíveis na definição do algoritmo [NIS 2001a]. Tais vetores de testes definem entradas e saídas padrão para a verificação de resultados, que podem assim ser comparados bit a bit. No Anexo 3 estão figuras que ilustram as telas dos simuladores.

Depois de efetivamente implementar a codificação VHDL nos componentes Altera™, o Max+plus II fornece relatórios detalhados de ocupação. Destes relatórios, foram retirados os dados mostrados a seguir em termos de utilização.

Não serão detalhados os elementos que constituem os componentes Altera™, especificamente a família Flex10K utilizada no UFIR-128. Serão citados os valores absolutos de utilização dos elementos, bem como a sua utilização percentual em relação à totalidade dos elementos disponíveis em cada componente.

Além disso, utilizou-se também o Max+plus II a fim de produzir todos os resultados de freqüência máxima de operação. Os valores foram confirmados através das ferramentas de Timing Analyzer e Waveform Generator. Os períodos de relógio mínimos eram encontrados através do Timing Analyzer e confirmados funcionalmente com o Waveform Generator.

Este fato é de especial importância, pois não foram obtidos resultados sobre hardware em laboratório, os quais demandariam a utilização de componentes, equipamentos de geração e verificação de vetores e, principalmente, um ambiente de testes específico. Ou seja, os resultados aqui apresentados para a encriptação e decriptação em hardware baseiam-se em ferramentas de software para simulação providas pelo fabricante dos componentes, no caso a Altera™.

## *5.1 Arquitetura de 128 bits*

Os desenvolvimentos ENC128 e DEC128 podem ser conferidos em maiores detalhes no Anexo 1. A seguir, serão mostrados os dados de performance e utilização de área sobre componentes Altera™ EPF10K250AGC599-1. Esse dados são comparados com dois outros trabalhos.

O primeiro destes trabalhos foi públicado por Viktor Fischer [FIS 2000], e apresenta 3 versões de implementação para o AES-128. Entretanto, mesmo a mais veloz das implementações não utiliza os mesmos conceitos utilizados nesta dissertação, incluindo a síntese em componentes diferentes. Mesmo assim, os resultados serão apresentados para efeito de conhecimento.

O segundo trabalho foi feito por Piotr Mroczkowski [MRO 2000]. Neste caso a comparação é válida, pois a arquitetura utilizada também manipula todos os 128 bits paralelamente, como também utiliza o mesmo componente da Altera.

Os resultados apresentados na tabela abaixo ilustram o período mínimo de clock conseguido na encriptação e decriptação da arquitetura de 128 bits.

TABELA 5.1 – Período mínimo dos sinais de relógio na arquitetura de 128 bits

| Projeto | Sinal de Relógio | Tempo mínimo |
|---|---|---|
| ENC128 | CLK | 80 ns |
| ENC128 | CLK2 | 40 ns |
| DEC128 | CLK | 80 ns |
| DEC128 | CLK2 | 50 ns |

Os períodos mínimos levam a dois cálculos distintos e importantes para a avaliação de desempenho em circuitos de encriptação e decriptação [NSA 2000]. O primeiro deles é a taxa de transferência máxima conseguida quando em operação contínua, e a segunda é o atraso devido ao tempo de inicialização de uma nova chave.

Conforme explicado no capítulo anterior, tópico 4.6, quando em operação contínua, tanto a encriptação quanto a decriptação necessitam de dez (10) ciclos de relógio CLK2 para aplicar a transformação AES-128. Assim,

t (taxa de transmissão ENC128) = 128 bits / 400 ns (77)
t (taxa de transmissão ENC128) = 320 Mbit/s (78)

e

t (taxa de transmissão DEC128) = 128 bits / 500 ns (79)
t (taxa de transmissão DEC128) = 256 Mbit/s (80)

Como também, quando uma nova chave é colocada, são necessários 12 ciclos para gerar as chaves de rodadas e sinalizar como pronto o processo. Desta maneira, tanto para a encriptação quanto para a decriptação, conclui-se que,

d (tempo de inicialização) = 12 x 80 ns = 960 ns (81)

A tabela abaixo mostra a comparação dos resultados obtidos nesta dissertação com os outros trabalhos citados acima.

TABELA 5.2 – Comparação da taxa de transferência máxima

| Projeto | Encriptação (AES-128) | Decriptação (AES-128) |
|---|---|---|
| UFIR-128 | 320 Mbit/s | 256 Mbit/s |
| [MRO 2000] | 268 Mbit/s | 248 Mbit/s |
| [FIS 2000] | 232,7 Mbit/s | 211,5 Mbit/s |

A seguir, outra tabela mostra a utilização dos componentes Altera, já fornecendo em conjunto os dados de outro projeto, para efeitos de comparação.

TABELA 5.3 – Utilização do componente EPF10K250AGC599-1 para implementar o AES-128

| Projeto | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|
| ENC128 | 261 | 130 | 0 | 40960(100%) | 4805(39,5%) |
| DEC128 | 261 | 130 | 0 | 40960(100%) | 5016(41,3%) |
| Encript. [MRO 2000] | 258 | 129 | 0 | 40960(100%) | 1032(8,5%) |
| Decript. [MRO 2000] | 258 | 129 | 0 | 40960(100%) | 2885(23,7%) |

### 5.1.1 Comentários

Apesar do desempenho superior em termos de performance em relação aos outros trabalhos, a diferença entre a encriptação e a decriptação é grande. Deve-se então analisar as funções utilizadas na encriptação e suas inversas utilizadas na decriptação. Conforme mostrado anteriormente, são quatro (4) as transformações utilizadas pelo Rijndael na operação contínua da arquitetura de 128 bits:

- SubByte e InvSubByte => ambas as transformações são implementadas como tabelas ROM, e ,portanto, não podem apresentar diferenças significativas na implementação.
- ShiftRow e InvShiftRow => são implementadas como roteamento interno dos componentes, e portanto não devem apresentar variações significativas de performance.
- AddRoundKey => não há função inversa.
- MixColumn e InvMixColumn => as transformações são realmente diferentes, sendo que a MixColumn é nitidamente mais simples que sua inversa. Vide tópicos 3.5.3 e 3.7.3.

Por isso, simulou-se a encriptação e a decriptação com a mesma estrutura original, porém sem as funções MixColumn e InvMixColumn respectivamente. Os resultados são mostrados abaixo.

TABELA 5.4 – ENC128 e DEC128 modificados

| **Projeto** | **Sinal de Relógio** | **Tempo mínimo** |
|---|---|---|
| ENC128[1] | CLK | 80 ns |
| ENC128[1] | CLK2 | 32 ns |
| DEC128[2] | CLK | 80 ns |
| DEC128[2] | CLK2 | 34 ns |

[1] - Encriptação 128 bits sem a transformação MixColumn
[2] - Decriptação 128 bits sem a transformação InvMixColumn

Pode-se observar que realmente as funções MixColumn e InvMixColumn apresentam uma grande diferença de performance, pois a partir da retirada de ambas as funções os desempenhos quase se equivaleram.

Outro fator muito importante a ser destacado é a utilização de 100% da memória disponível de *Embedded Array Block's* dos componentes Altera EPF10K250AGC599-1. Do total, 32768 bits foram utilizados para implementar a transformação ByteSub

paralelamente sobre os 128 bits do *State*. Outros 8192 bits foram utilizados na geração das chaves de rodada, onde faz-se necessário também uma transformação ByteSub sobre 32 bits.

Tal componente é o maior da família Flex10K. Apesar da total ocupação dos blocos EAB, ainda restam mais do que 50% de Logical Cels nos componentes para implementar outras funções complementares.

## 5.2 Arquitetura de 32 bits

Os desenvolvimentos ENC32 e DEC32 podem ser conferidos em maiores detalhes no Anexo 2. A seguir, são mostrados os dados de performance e utilização de área sobre componentes Altera™ EPF10K50VFC484-1.

Os resultados apresentados na tabela abaixo ilustram o período mínimo de clock conseguido na encriptação e decriptação da arquitetura de 32 bits.

TABELA 5.5 – Período mínimo dos sinais de relógio na arquitetura de 32 bits

| Projeto | Sinal de Relógio | Tempo mínimo |
|---|---|---|
| ENC32 | CLK | 40 ns |
| DEC32 | CLK | 40 ns |

Os períodos mínimos levam ao cálculo da taxa de transferência máxima conseguida quando em operação contínua. Conforme explicado no capítulo anterior, tópico 4.7, quando em operação contínua, tanto a encriptação quanto a decriptação necessitam de cento e vinte (120) ciclos de relógio CLK para aplicar a transformação AES-128. Assim, tanto para a encriptação quanto para a decriptação tem-se:

t (taxa de transmissão) = 128 bits / 4800 ns (82)
t (taxa de transmissão) = 26,67 Mbit/s (83)

Diferentemente da arquitetura de 128 bits, na versão de 32 bits as chaves de rodadas são geradas dinamicamente. Desta maneira, não há tempo de inicialização necessário para processar o algoritmo com uma nova chave.

A tabela a seguir mostra a utilização dos componentes Altera.

TABELA 5.6 – Utilização do componente EPF10K50VFC484-1 para implementar o AES-128

| Projeto | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|
| ENC32 | 38 | 129 | 0 | 16384(80%) | 1980(68,8%) |
| DEC32 | 38 | 129 | 0 | 16384(80%) | 2473(85,9%) |

Apesar desta arquitetura poder utilizar um componente EPF10K40, pois 16384 bits de RAM estão disponíveis, optou-se por um componente ligeiramente superior, permitindo que outras funções complementares sejam inseridas no futuro.

### 5.2.1 Comentários

É claro que o ganho em termos de área necessária para a implementação teve custos em relação ao desempenho. Através de uma utilização de área 2,5 vezes menor do que a arquitetura de 128 bits, conseguiu-se uma performance aproximadamente doze (12) vezes menor para a encriptação e dez (10) vezes menor para a decriptação.

O número de bits de memória que utilizam EAB's, fator determinante para a escolha do componente da família Flex10K para implementação do Rijnadel, foi de 16384. Duas transformações SubBytes para manipular 32 bits foram construídas. Umas delas processa parcialmente o total de 128 bits do *state*, enquanto a outra gera a próxima chave rodada, que requer uma transformação SubByte sobre 32 bits.

Entretanto, ao contrário da arquitetura de 128 bits, na qual não há muito o que se fazer para melhorar o desempenho, na arquitetura de 32 bits inúmeras possibilidades são viáveis.

Em virtude da utilização de 80% dos blocos EAB, não é possível processar mais do que 32 bits em paralelo, pois isto requeriria outros 8192 bits de memória RAM, não disponíveis no componente, cujo máximo é de 20480 bits. Todavia, todas as demais operações, como demandam somente *Logical Cells*, poderiam ser feitas paralelamente sobre os 128 bits. Desta maneira, ao invés do ciclo normal do Rijndael consumir 12 ciclos de relógio, seriam consumidos somente seis (6), quatro (4) para a transformação SubByte e somente um (1) ciclo para os demais estados. O ganho neste caso é de 100% sobre o que foi mostrado acima, pois uma transformação completa AES-128 levaria sessenta (60) ciclos, contra cento e vinte (120) mostrados acima. Assim,

t (taxa de transmissão) = 128 bits / 2400 ns (84)

t (taxa de transmissão) = 53,33 Mbit/s (85)

No cálculo acima estima-se que o período de clock mínimo permanece inalterado, com valor de 40 ns.

Outra alternativa seria gerar separadamente as chaves de rodadas, através de um processo semelhante ao utilizado na arquitetura de 128 bits. Depois, reutilizar a transformação SubByte para processar 64 bits do *state* por vez, através de duas transformações SubByte de 32 bits. Esta arquitetura aumentaria com certeza o tamanho do bloco de controle, bem como faria com que o projeto necessitasse também de um tempo de inicialização da chave.

## *5.3 Implementação do AES-128 sobre outras plataformas FPGA*

A decisão pela utilização da família Flex10K já foi justificada na parte 4.3 desta dissertação. Contudo, desde o início do projeto, a implementação do AES-128 sobre outras plataformas evoluiu, alcançando resultados muito interessantes.

Através de pesquisa feita na Internet no período entre Agosto e Outubro de 2001, foram encontradas três soluções que se destacaram. A primeira delas é resultado do trabalho do grupo de criptografia em hardware da George Mason University, do qual fazem parte Kris Gaj e Pawel Chodowiec, autores de inúmeros trabalhos já citados nesta dissertação [GAJ 2000a][GAJ 2000b]. O outro trabalho é da própria Altera, que oferece códigos prontos para funções conhecidas na forma de Propriedade Intelectual [ALT 2001b]. O terceiro trabalho também encontra-se no site da Altera sob a forma de Propriedade Intelectual, mas foi desenvolvido pela empresa D'Crypt. Pte. Ltd.

No trabalho da George Mason University [GMU 2001], são citadas taxas de transmissão de 16 Gbit/s quando modos realimentados não são utilizados, e 743 Mbit/s na arquitetura básica, também utilizada nesta dissertação. Tais resultados foram possíveis utilizando-se componentes da família Virtex E, da Xilinx [XIL 2000].

Os dois projetos oferecidos pela Altera na forma de Propriedade Intelectual foram feitos pelos grupos HammerCores, agora uma empresa da própria Altera [ALT 99], e D'Crypt Pte. Ltd. O projeto da Hammercores alcança 907 Mbit/s para a encriptação e 698 Mbit/s para a decriptação, utilizando a arquitetura básica que permite realimentação, sobre os novos componentes da família APEX [ALT 2001b]. O outro trabalho, feito pela empresa D'Crypt, divulga taxa de transmissão da ordem de 2,5 Gbit/s, mas não são fornecidos maiores detalhes sobre a arquitetura implementada.

# 6 Conclusão

Várias eram as propostas a serem concluídas ao final do trabalho desta dissertação, e que foram sempre perseguidas ao longo de todo o período. Todos os objetivos traçados inicialmente são discutidos agora de maneira crítica, a fim de identificar resultados positivos e negativos.

Claramente, uma das tarefas seria o estudo do algoritmo Rijndael para posterior implementação em lógica programável Altera. Para tanto, foi necessário aprofundar o conhecimento sobre criptografia, em especial criptografia em hardware. Atualmente, e espera-se que o trabalho escrito tenha mostrado este conhecimento, domina-se a técnica de caracterizar uma implementação Rijndael sobre componentes FPGA Altera. A utilização de cada componente já pode ser prevista com os resultados mostrados.

Além disto, outras pessoas foram envolvidas no projeto, de maneira que o conhecimento foi absorvido por uma equipe, e não somente o mestrando e o seu orientador. Diretamente no projeto do Rijndael sobre Altera, trabalhou também um bolsista de iniciação científica, que agora é capaz de evoluir nas questões que ainda persistem, buscando novas soluções. Como também, na disciplina de CMP119, Projeto de um Sistema VLSI, os alunos estão trabalhando também sobre a codificação produzida, bem como criando novas versões, a fim de implementá-la sobre outras plataformas.

Tamanha atividade em torno do assunto mostra a atualidade dos seus desafios e o desejo de continuidade sobre um tema promissor. Se o conhecimento acumulado por somente umas das pessoas não for suficiente, com certeza o poder de uma equipe de trabalho pode superar inúmeras dificuldades, alcançando resultados excelentes. Estas questões expõem aspectos qualitativos do trabalho, tão importantes quanto os resultados quantitativos conseguidos.

Contudo, esta dissertação possuia um aspecto muito prático, no qual duas codificações para o Rijndael deveriam ser produzidas. Quando o trabalho parte para a implementação de projetos, envolvendo fases de desenvolvimento, testes e certificação de resultados, muitas vezes é difícil produzir resultados definitivos. Isto acontece pois a produtividade da tarefa de criação não é linear ao longo do tempo, levando em muitos casos mais tempo do que o planejado. Entretanto, cabe ao desenvolvedor identificar momentos nos quais, apesar do trabalho não estar completo do ponto de vista do teste de todas as alternativas, resultados satisfatórios já podem ser apresentados.

As duas codificações do Rijndael em VHDL foram produzidas, uma preconizando a performance e a outra a utilização de área. As versões da UFIR-128 foram implementadas em VHDL, e funcionalmente testadas incessantemente, de maneira a comprovar a validade da descrição. Ambas as versões atendem às especificações originais do algoritmo e comprovam a geração com sucesso de todos os vetores de testes sugeridos pelos autores e ratificados pelo NIST na definição oficial do padrão.

Tais implementações foram simuladas com sucesso sobre componentes Altera na família Flex10K. Para a arquitetura de 128 bits, utilizou-se o componente EPF10K250, o maior da família Flex10k, enquanto que na arquitetura de 32 bits utilizou-se o componente EPF10K50. Todos os detalhes em termos de utilização dos componentes foram fornecidos no capítulo anterior.

Todavia, além do desafio de construir uma versão funcional do algoritmo Rijndael sobre lógica programável Altera, existia também a questão de produzir um circuito rápido, capaz de trabalhar com taxas de transmissão adequadas para um projeto

em hardware. Não se pode esquecer que um projeto em FPGA cujo desempenho fosse menor que implementações comuns em software provavelmente não seria justificável. Como também, é óbvio que superar as marcas até então divulgadas era uma das pretensões do projeto desde o seu início.

Pois o desempenho da arquitetura de 128 bits sobre a família Flex10K da Altera superou, em simulações do software do fabricante dos componentes, quaisquer outros trabalhos semelhantes. Todos os detalhes da implementação foram mostrados no Capítulo 4, e os resultados no Capítulo 5. Tais resultados gratificam os desenvolvedores, premiando o trabalho desenvolvido, e certificam que as decisões de projeto em relação à arquitetura estavam corretas.

Como também, em virtude da utilização de codificação VHDL padrão, é possível implementar o UFIR-128 em outros componentes da Altera, ou até mesmo em outras plataformas FPGA, como Xilinx, por exemplo. A partir dos resultados conseguidos na família Flex10K, espera-se também dados significativos nas outras alternativas, sendo capaz de competir com os últimos trabalhos públicados na área e disponíveis no tópico 5.3. Ou seja, um código VHDL de alta qualidade e rápido foi produzido, podendo agora ser utilizado para inúmeras outras aplicações.

Por outro lado, a versão de 32 bits mostrou uma sensível economia de área utilizada, para os casos nos quais desempenho não é o fator mais importante. Conforme mostrado no tópico 5.2.1, é ainda possível melhorar consideravelmente a relação área x desempenho para esta versão, utilizando-se o conhecimento acumulado neste trabalho. Contudo, em virtude dos prazos exíguos para a entrega da dissertação, não foi possível desenvolver muitas das idéias disponíveis. Optou-se por apresentar resultados sobre uma codificação comprovadamente funcional, ao invés de prosseguir o desenvolvimento e apresentar resultados sobre uma codificação instável, com funcionalidades não totalmente testadas.

O tempo também limitou o desenvolvimento de uma decriptação com maior desempenho na arquitetura de 128 bits. Foi possível identificar a enorme influência da função InvMixColumn no período mínimo do circuito, justificando em parte a grande diferença entre a encriptação e a decriptação naquela versão. Entretanto não foram testadas outras soluções.

Ainda restam muitos desafios em relação à implementação do Rijndael sobre hardware, e não somente sobre FPGA. Pode-se citar:

- implementação da arquitetura de 128 bits sobre outros componentes da Altera, principalmente sobre a família APEX, comparando os resultados com os trabalhos já existentes;
- implementação da arquitetura de 128 bits sobre outras plataformas de hardware ou FPGA, utilizando o mesmo código gerado, ou ainda modificando-o visando agregar eficiência;
- prosseguir no desenvolvimento da versão de 32 bits a fim de melhorar a relação área x desempenho;
- codificar as demais versões do Rijndael, para produzir os padrões AES-192 e AES-256;
- integrar o algoritmo de criptografia simétrica Rijndael, agora dominado, em outros projetos, a fim de produzir sistemas completos para criptografia e segurança da informação.

Desta forma, chega-se ao final desta dissertação com a convicção de que todos os objetivos iniciais foram alcançados, e com satisfação, em virtude da qualidade dos

resultados finais. É claro que existem ainda atividades a serem desenvolvidas no futuro, mas que somente reforçam o acerto nas decisões e motivações deste trabalho.

# Anexo 1 Dispositivo 128 bits

## *A.1 Interface*

O encriptador/decriptador Rijndael de 128 bits recebe os dados a serem encriptados/decriptados, bem como a chave, em 128 bits. Sua saída apresenta dados criptografados diretamente em 128 bits. É utilizado o chip da Altera EPF10K250AGC599-1, com 470 pinos de I/O, 12160 LCs e 40960 bits de RAM embutida. Decriptador e encriptador usam um FPGA cada.

O objetivo deste projeto é produzir um processamento paralelo de 128 bits, aumentando a performance, com o custo de uma maior utilização de memória. Os componentes FLEX10K250 são os únicos componentes possíveis da família FLEX para implementação, pois apenas eles tem memória suficiente para esta aplicação.

Existem entradas para dados e chave, onde cada uma toma 128 bits por vez, armazenando-os em uma variável matricial de 4 por 4 bytes (state_t), totalizando 128 bits. Todo o processamento é feito inteiramente sobre os 128 bits da variável.

A operação de leitura armazena a entrada de dados em uma variável, que é lida pela variável de trabalho do processo Rijndael. A saída é feita transferindo-se os dados da variável de trabalho para uma variável que armazena o dado de saída, garantindo a estabilidade para leitura dos dados para o processador.

As chaves de rodada expandidas são armazenadas em uma variável de onze vetores de 4 linhas por 4 colunas de 8 bits, ou seja, um vetor de onze componentes state_t. A chave de entrada é armazenada no primeiro dos componentes e os demais são gerados através dele.

TABELA A.1 – Pinagem dos dispositivos de 128 bits (encriptação/decriptação)

| Nome | Tipo | Descrição |
|---|---|---|
| CLK | IN | Relógio da entrada de dados e geração das chaves de rodada. |
| CLK2 | IN | Relógio do processo Rijndael. |
| SETUP | IN | Determina período de configuração/operação. |
| WR_KEY | IN | Indica que a chave de entrada está no barramento. |
| WR_DATA | IN | Indica que os dados de entrada estão no barramento. |
| KEYIN | IN | Entrada da chave em 128 bits. |
| DATAIN | IN | Entrada de dados em 128 bits. |
| KEY_OK | OUT | Indica que Expansion Key já gerou todas as chaves. |
| DATA_OK | OUT | Indica permissão de leitura/escrita no barramento. |
| DATAOUT | OUT | Saída de dados em 128 bits. |

## *A.2 Operação*

Há dois modos distintos de operação no dispositivo, coordenados pelo sinal SETUP. Estando em lógica zero, o sinal permite o processamento do algoritmo Rijndael. Ao ser acionado, este processamento é interrompido, e é feita a escrita da chave e do primeiro dado a ser encriptado/decriptado.

Para o inicio da operação do dispositivo ou troca do valor da chave, deve-se, inicialmente, colocar o sinal de SETUP em lógica 1, entrando com a chave, a partir da qual serão geradas todas as chaves de rodada. Deve-se colocar também o primeiro bloco de bits a ser transformado, aproveitando que o proceso Rijndael não é executado e suas variáveis foram todas ajustadas.

Para escrever a chave deve-se acionar o sinal WR_KEY na borda de subida do pulso de relógio CLK. O valor da chave deve ser escrito em 128 bits diretamente. Após a leitura da chave, deve-se proceder com a leitura dos dados. Diferentemente dos dispositivos 32 bits, nos dispositivos 128 bits a entrada da chave deve ser feita usando, sempre, a chave inicial, não a última chave expandida. Logo após a escrita da chave, todas as chaves de rodadas (round keys) são geradas, tomada como relógio o sinal CLK. Enquanto a expansão da chave estiver sendo feita, o sinal KEY_OK terá valor 1. A figura abaixo ilustra esta operação.

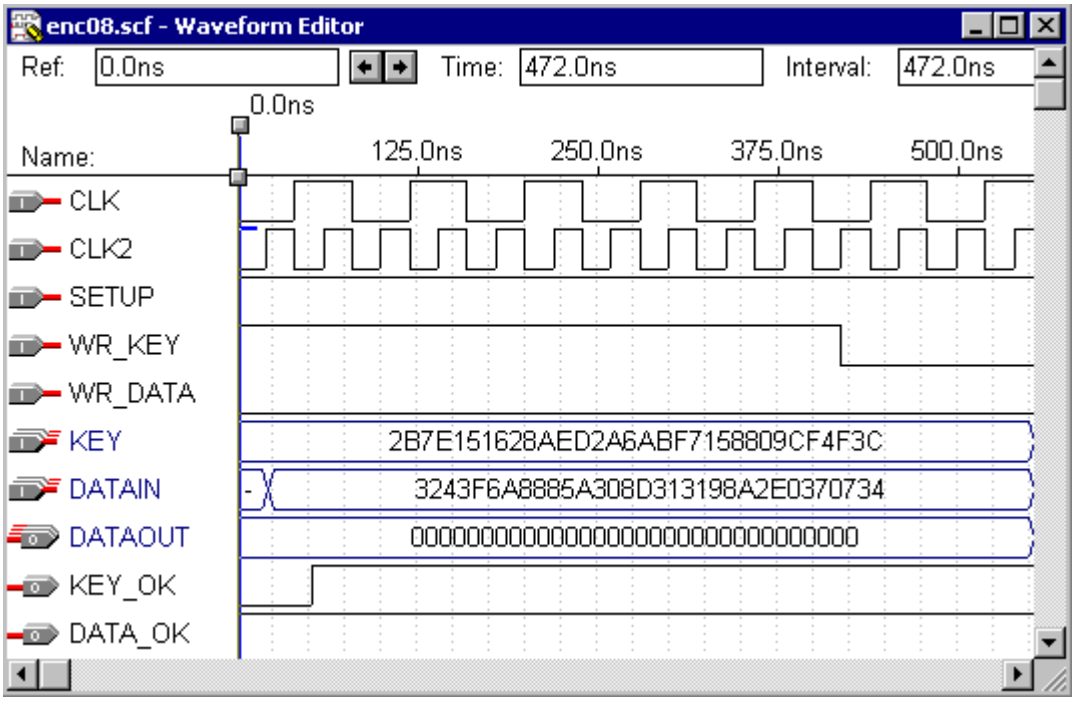


FIGURA A.1 - Entrada da chave

O dado incial deve ser escrito mantendo-se SETUP ativado e acionando-se WR_DATA para valor lógico 1 na borda de subida do relógio CLK. O valor da chave deve ser escrito diretamente em 128 bits, conforme ilustra a figura A.2. A escrita do valor inicial independe da escrita da chave.

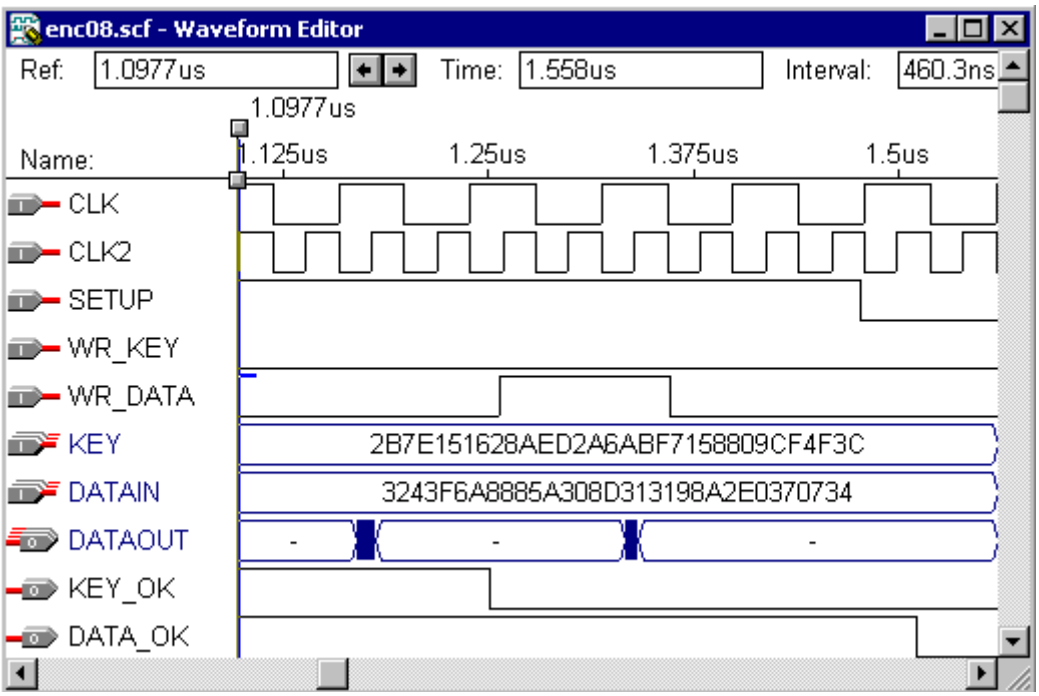


FIGURA A.2 - Entrada do dado inicial

No segundo momento, é feito o processamento do algoritmo e a leitura de novos dados. Após a escrita do dado inicial, da chave e da sua completa expansão, o sinal SETUP deve ser colocado em zero lógico, para permitir o processamento do Rijndael. O processo do Rijndael funciona continuamente, com velocidade determinada pelo sinal de relógio CLK2, utilizando o dado inicial e as chaves de rodadas previamente geradas. A cada 10 sinais de relógio um novo dado de entrada é processado, gerando uma saída paralela de 128 bits.

Como o Rijndael armazena internamente o dado inicial, tão logo seja possível ao dispositivo de controle escrever um dado novo, ele acionará o sinal DATA_OK, informando que deve ser alimentado por um novo dado de entrada. Para escrever cada dado novo, basta acionar o sinal WR_DATA na borda de subida do relógio CLK, com o novo dado escrito no barramento dos dados. A figura A.3 ilustra o processo.

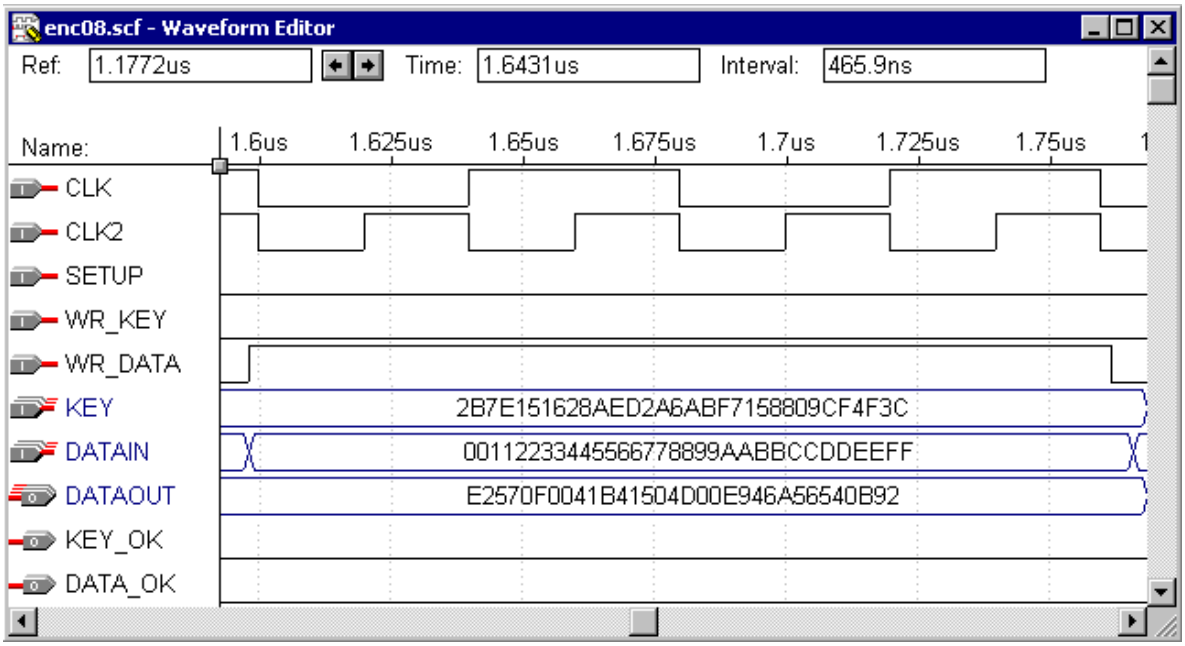


FIGURA A.3 - Entrada de dados novos ao longo da operação

Como o Rijndael opera continuamente, a escrita dos dados novos e leitura dos resultados possuem uma temporização determinada. Um novo dado no barramento colocado em um período incorreto pode resultar em valores incorretos, bem como a leitura dos resultados também. O dispositivo informa, através do sinal DATA_OK, os momentos adequados para leitura e escrita de dados no barramento. A escrita e a leitura devem ser efetuadas sempre a partir da borda descendente do referido sinal. Vide figura A.4.

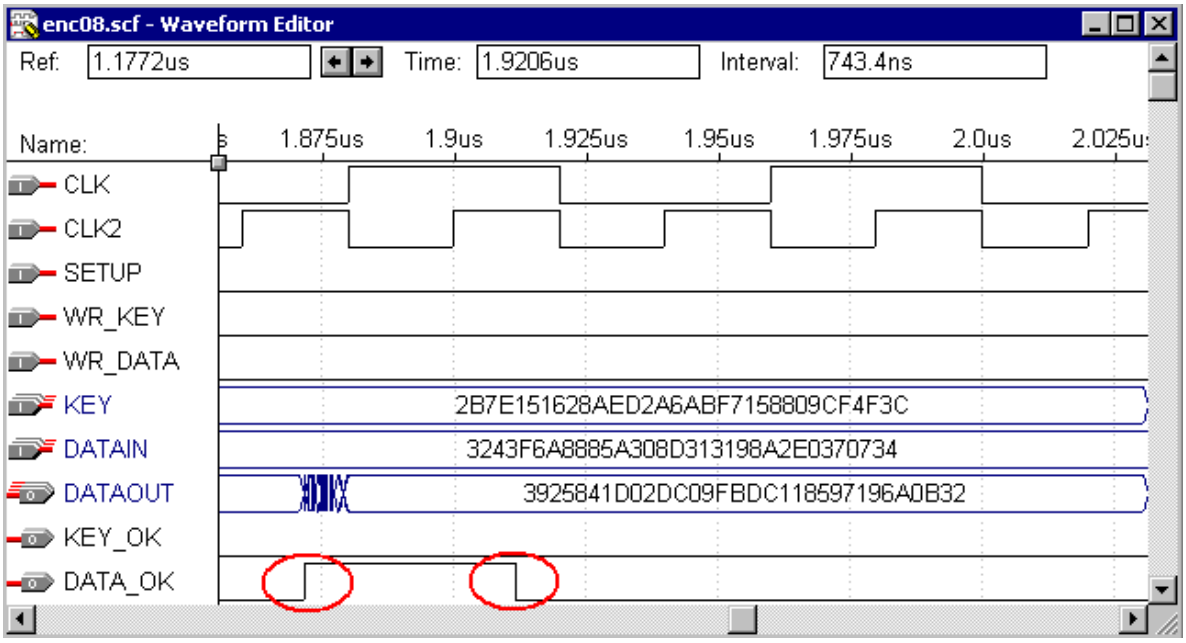


FIGURA A.4 - DATA_OK indica quando pode-se ler e escrever dados no barramento

É importante notar que o novo dado somente será utilizado na próxima rodada, e portanto o controle central possui algum tempo para efetuar a escrita. Entretanto, tal

operação deve ocorrer necessariamente antes da borda de subida de DATA_OK, pois então será tarde demais. Os resultados disponíveis no pinos de DATAOUT permanecem no barramento desde a descida do sinal DATA_OK. Analogamente, se o controle central não efetuar a leitura antes da próxima borda de subida de DATA_OK, resultados errôneos serão apresentados.

O sinal SETUP serve, também, para interromper o processamento por qualquer motivo, deixando o dispositivo em modo de espera. Assim, quando não existirem dados a serem encriptados/decriptados, deve-se ler o último dado encriptado/decriptado e então levantar o sinal SETUP, conforme mostrado na figura A.5. Cabe observar que o sinal SETUP trabalha conjuntamente com o sinal CLK2, logo não deve ser ativado em sua borda de subida.

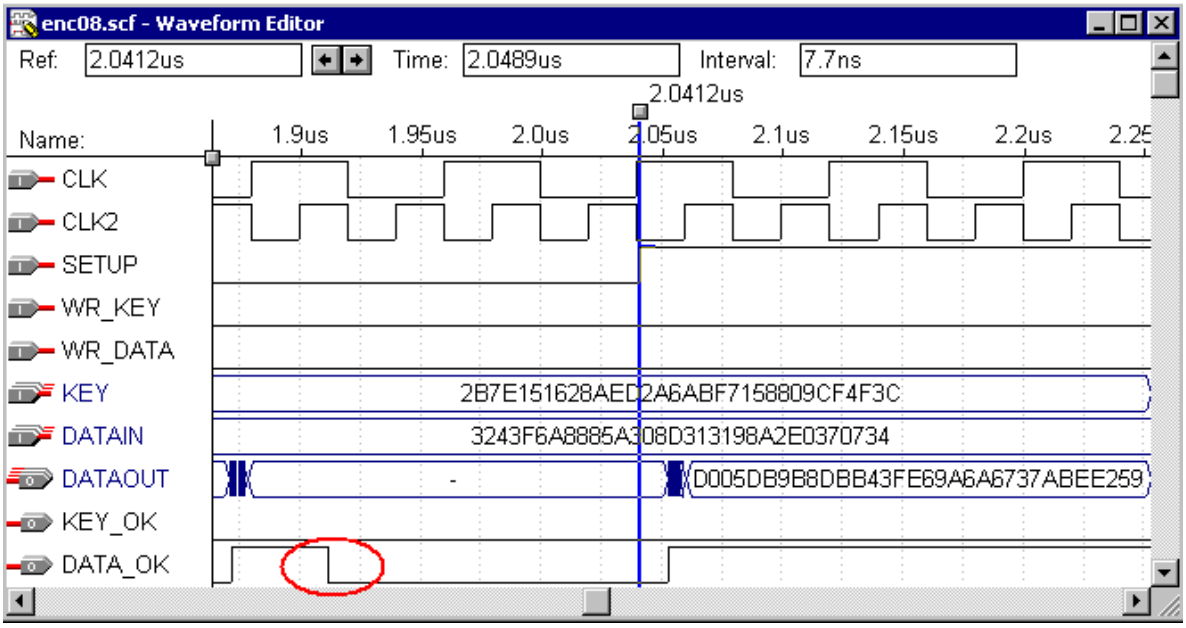


FIGURA A.5 - Modo de espera do dispositivo

Para reiniciar o dispositivo sem trocar a chave, deve-se, inicialmente, colocar o novo dado no barramento com o sinal WR_DATA ativado. Isso feito, o sinal de SETUP deve ser desativado e o dispositivo operado normalmente. Vide figura A.6.

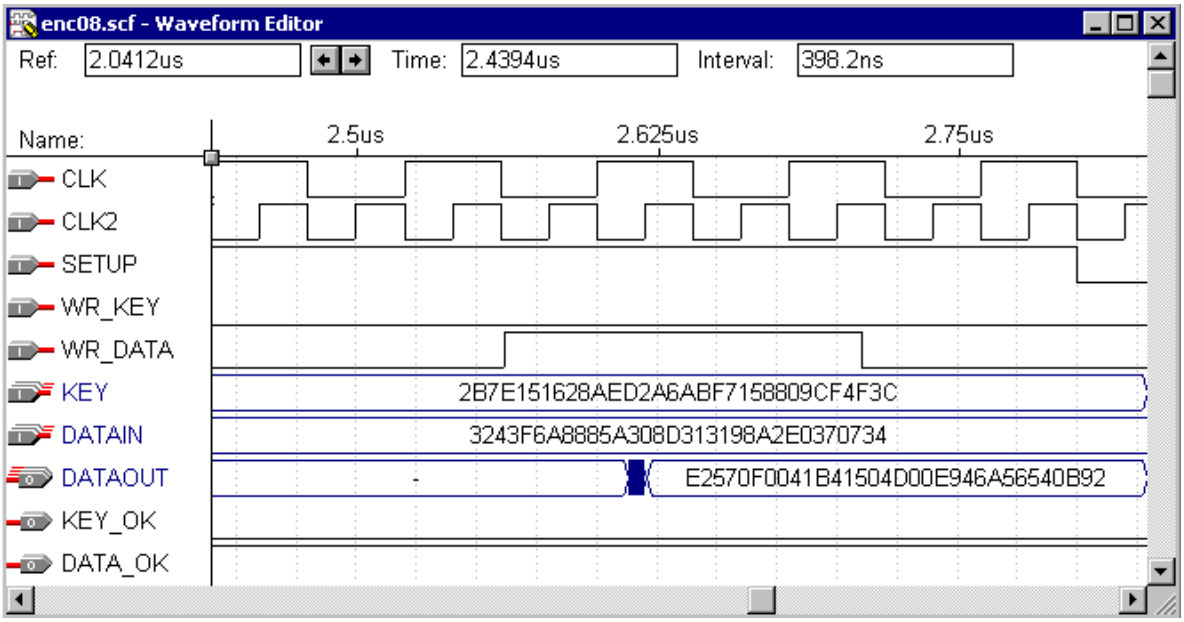


FIGURA A.6 - Entrada de dados antes de sair do modo de espera

## *A.3 Encriptador*

### A.3.1. Diagrama de blocos

Há quatro processos realizados em paralelo na arquitetura do dispositivo, como pode ser visto na figura abaixo.

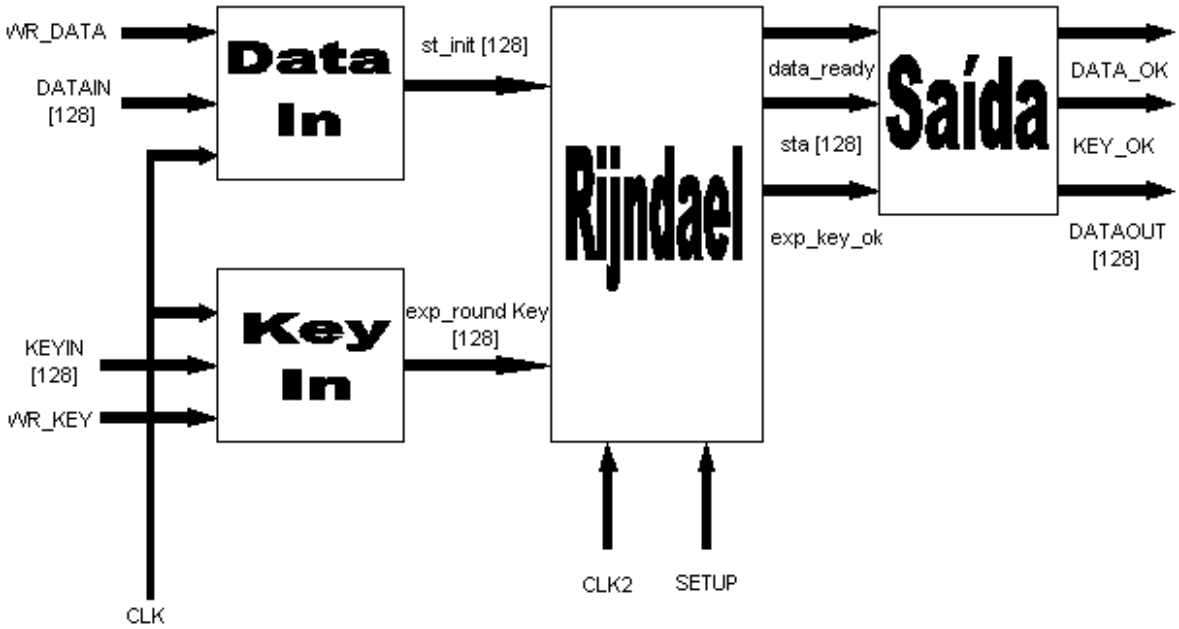


FIGURA A.7 - Diagrama de blocos do encriptador 128 bits

O processo Data_In é responsável pela recepção dos dados de entrada. Ele toma os 128 bits de entrada de dados e os armazena em uma variável específica. Este processo é ativado sempre que o sinal WR_DATA é acionado na borda de subida do

sinal CLK, e permite que o processador escreva um novo dado durante a encriptação do dado anterior. Para tanto, o processador deve esperar a borda descendente do sinal DATA_OK, como explicado na seção anterior.

O processo Key_In é responsável pela recepção da chave inicial, seu armazenamento e geração das *expansion_round_keys* que serão usadas ao longo de todos os *rounds*. Para tanto, ele trabalha com 11 componentes compostos por uma matriz de 4 linhas por 4 colunas de 8 bits. Cada um destes componentes é uma variável do tipo state_t, usada como tipo padrão ao longo do processamento. A chave inicial é armazenada no primeiro componente e os demais são gerados a partir dela. Este processo é ativado pelo sinal WR_KEY na borda de subida do sinal CLK, e o final do processamento das chaves é assinalado pelo sinal KEY_OK.

O processo Rijndael implementa o algoritmo em 128 bits. Há duas formas de funcionamento. Se o sinal SETUP estiver acionado, ele ajusta valores das variáveis para o início do funcionamento e coloca o dispositivo em modo de espera. Se o sinal não estiver acionado o algoritmo é executado.

A execução consiste na realização de um Add_Key_F, seguido de um Byte_Sub, um Shift_Row e, excetuando-se o *round* 9, quando não é executado, o Mix_Column. No *round* 9, ao invés da execução do Mix_Column, é transferida para a variável de trabalho o novo dado de entrada (já armazenado pelo processo Data_In) e indicado o acionamento do sinal DATA_OK.

O processo Saída simplesmente realiza a interface de saída de dados. Ele aciona os sinais DATA_OK e KEY_OK quando solicitado. Sua função mais importante, contudo, é garantir a estabilidade da saída ao longo de todo o processo, ou seja, ela será alterada somente quando existir uma nova saída para ser exibida.

### A.3.2 Performance

Há dois sinais de clock no dispositivo. O primeiro (CLK) é usado para a escrita do dado de entrada, da chave e processamento das chaves de rodada. Sua velocidade máxima conseguida é um período de 80ns (12,5MHz).

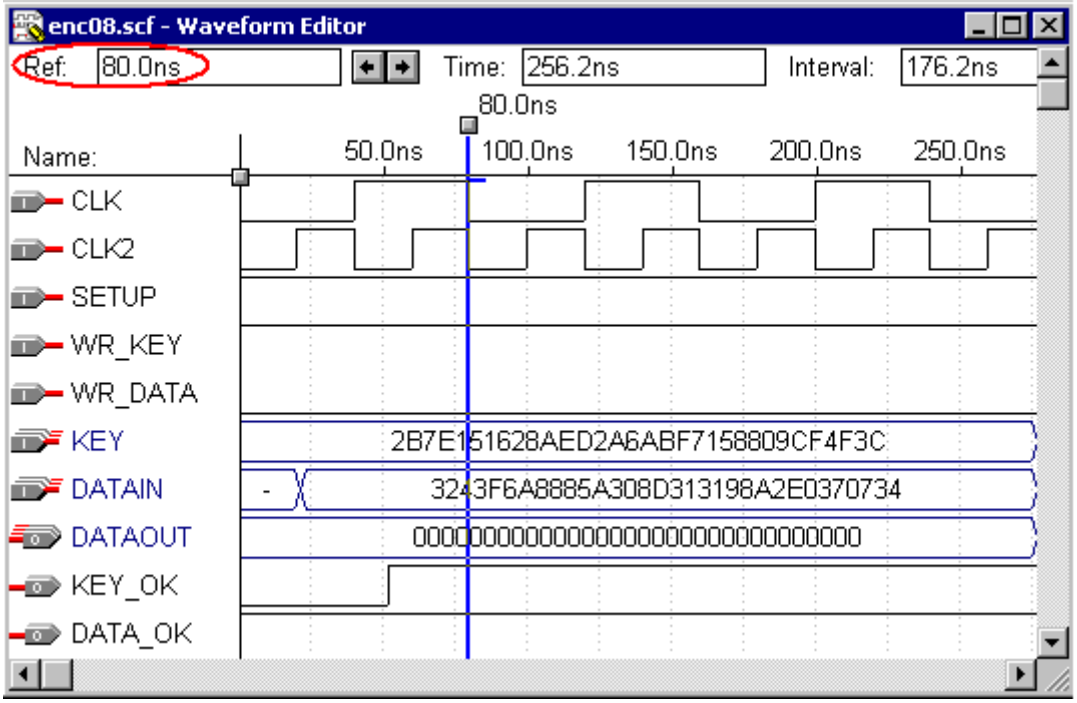


FIGURA A.8 - CLK mínimo do encriptador

A velocidade de processamento do dado no processo Rijndael depende do período de CLK2, tendo em vista que há um número pré-estabelecido de *rounds* e que cada *round* requer um ciclo de CLK2. O melhor resultado obtido foi o uso de um período de 40 ns (25MHz), como pode ser visto na Figura A.9.

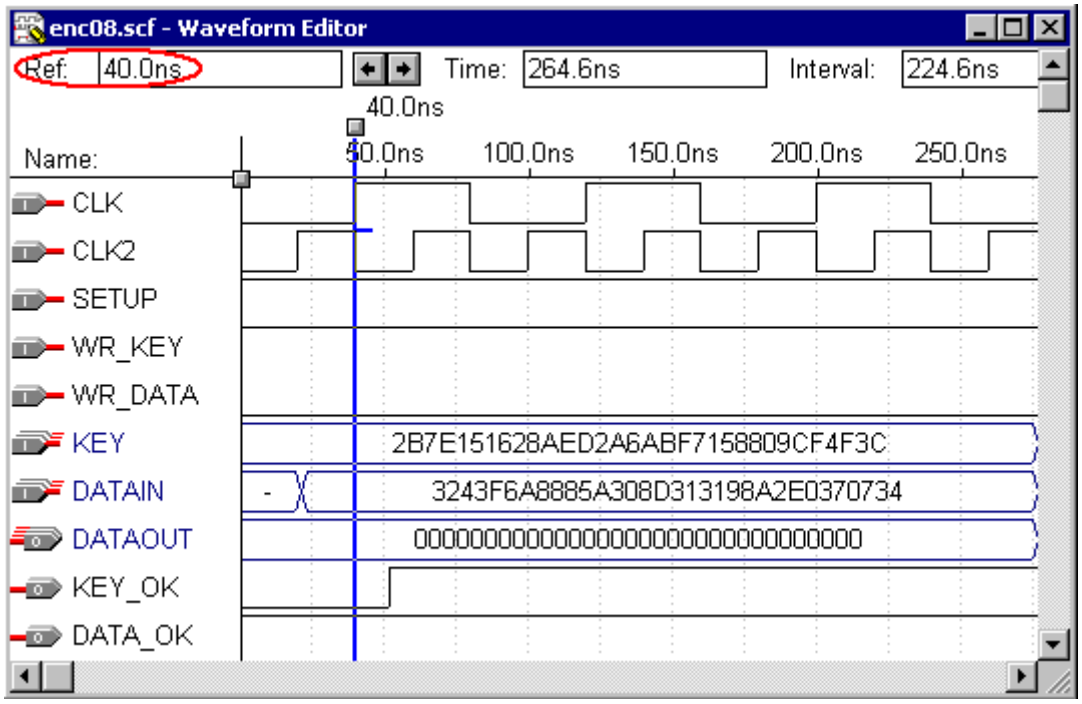


FIGURA A.9 - CLK2 mínimo do encriptador.

Com o valor sugerido de CLK2, leva-se 400ns (nas figuras, 2,3118us menos 1,9118us) para uma encriptação completa de 128 bits. Estes valores permitem uma taxa de saída de dados de aproximadamente 320 Mbit/s, conforme mostrado nas figuras A.10 e A.11.

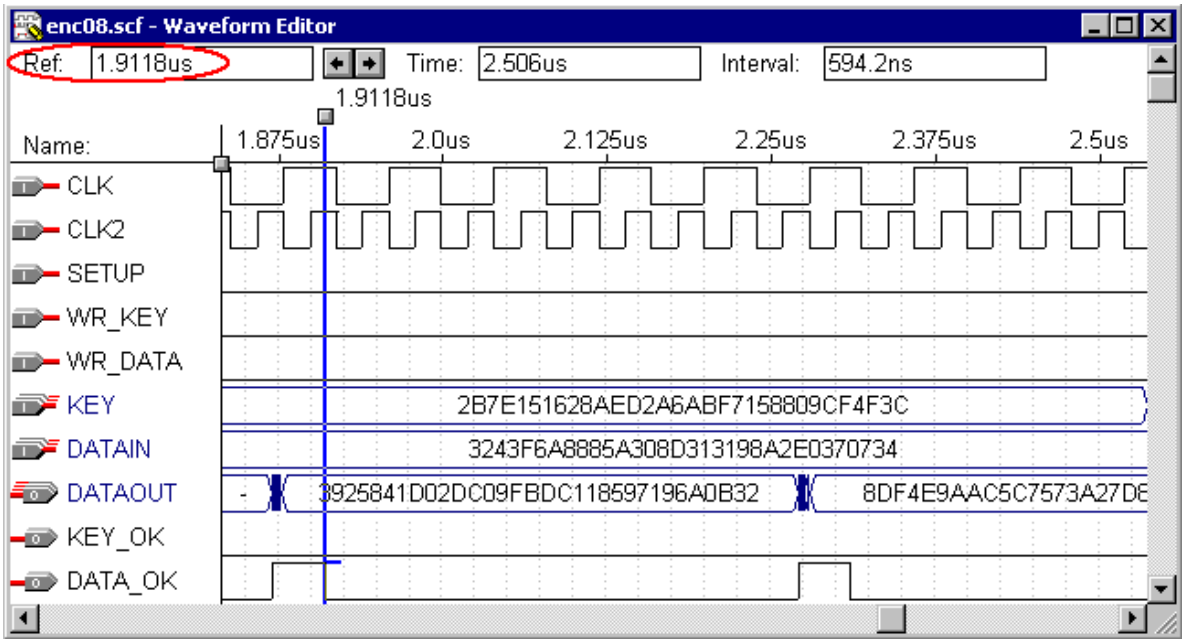


FIGURA A.10 - Primeiro dado pronto para ser lido

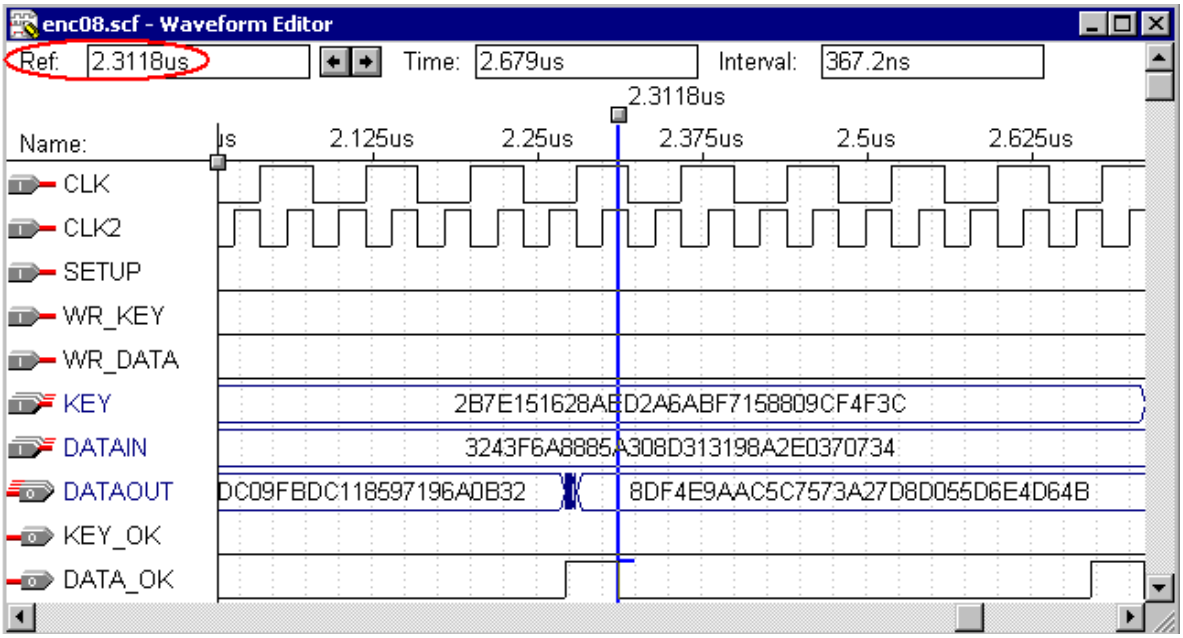


FIGURA A.11 - Segundo dado pronto para ser lido

### A.3.3. Utilização do componente

A implementação para o componente da Altera EPF10K250AGC599-1 apresenta a ocupação mostrada na tabela A.2. A memória do dispositivo é usada em sua

totalidade, a pinagem é bastante exigida, mas há pouco uso de Logic Cells (LCs). A ocupação é a seguinte:

TABELA A.2 – Ocupação de área do encriptador 128 bits

| Design | Componente | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|---|
| ENC128 | EPF10K250AGC599-1 | 261 | 130 | 0 | 40960 | 4805 |

## A.3.4. Pinagem

O dispositivo usa 391 pinos de I/O. Mostra-se, a seguir, a vista inferior do EPF10K250AGC599-1 (com seu diagrama de pinos) e a função de cada um.

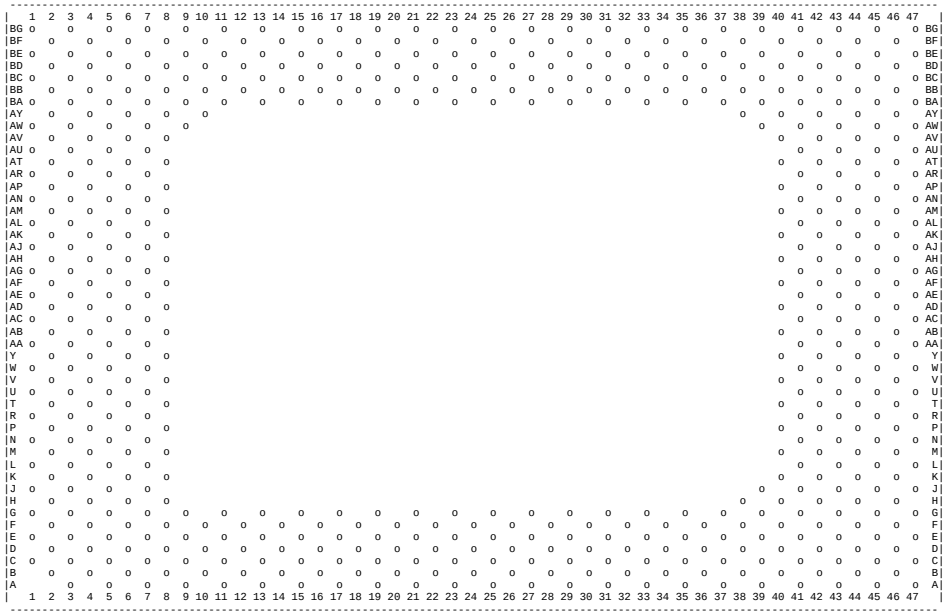

FIGURA A.12 - Vista inferior do EPF10K250AGC599-1

A3 VCCINT
A5 DATAIN62
A7 RESERVED
A9 RESERVED
A11 DATAOUT104
A13 RESERVED
A15 DATAIN94
A17 RESERVED
A19 KEY126
A21 DATAIN108
A23 DATAIN24
A25 CLK2
A27 DATAOUT81
A29 RESERVED
A31 RESERVED
A33 DATAIN76
A35 RESERVED
A37 KEY24
A39 DATAOUT103
A41 RESERVED
A43 RESERVED
A45 VCCINT
A47 GNDINT
B2 GNDINT
B4 ^nCONFIG
B6 RESERVED
B8 KEY30
B10 DATAIN67
B12 DATAIN84
B14 DATAOUT21
B16 DATAIN77
B18 KEY124
F8 DATAIN110
F10 RESERVED
F12 RESERVED
F14 DATAIN45
F16 DATAIN101
F18 DATAIN25
F20 RESERVED
F22 DATAIN63
F24 GNDIO
F26 RESERVED
F28 RESERVED
F30 RESERVED
F32 DATAIN103
F34 RESERVED
F36 RESERVED
F38 KEY95
F40 DATAIN81
F42 #TMS
F44 DATAOUT39
F46 DATAOUT74
G1 KEY51
G3 KEY14
G5 GNDIO
G7 VCCIO
G9 DATAIN17
G11 KEY67
G13 DATAIN78
G15 DATAIN29
G17 KEY55
G19 RESERVED
G21 DATAIN23
G23 GNDINT
U5 DATAOUT114
U7 KEY62
U41 DATAOUT91
U43 DATAOUT78
U45 DATAIN98
U47 DATAIN106
V2 DATAOUT86
V4 KEY16
V6 DATAOUT62
V8 KEY46
V40 KEY115
V42 KEY45
V44 DATAIN115
V46 KEY106
W1 KEY49
W3 VCCINT
W5 GNDIO
W7 DATAOUT115
W41 DATAOUT46
W43 GNDIO
W45 VCCINT
W47 DATAIN86
Y2 DATAOUT106
Y4 KEY52
Y6 DATAOUT110
Y8 DATA_OK
Y40 DATAOUT98
Y42 DATAIN16
Y44 DATAOUT52
Y46 KEY53
AA1 DATAOUT17
AA3 GNDINT
AM6 DATAIN74
AM8 DATAIN3
AM40 DATAOUT70
AM42 KEY41
AM44 KEY44
AM46 KEY102
AN1 DATAOUT102
AN3 KEY12
AN5 VCCIO
AN7 KEY4
AN41 DATAIN70
AN43 VCCIO
AN45 DATAOUT10
AN47 DATAIN102
AP2 DATAOUT31
AP4 DATAOUT27
AP6 DATAOUT60
AP8 DATAIN58
AP40 DATAIN57
AP42 DATAIN49
AP44 DATAOUT20
AP46 DATAOUT40
AR1 DATAOUT8
AR3 GNDINT
AR5 GNDIO
AR7 DATAOUT101
AR41 DATAIN52
AR43 GNDIO
AR45 GNDINT
AR47 DATAOUT44
AT2 KEY75
AT4 DATAOUT57
BB46 RESERVED
BC1 DATAIN125
BC3 DATAIN104
BC5 ^DATA0
BC7 GNDIO
BC9 VCCIO
BC11 DATAIN123
BC13 GNDIO
BC15 VCCIO
BC17 RESERVED
BC19 GNDIO
BC21 VCCIO
BC23 RESERVED
BC25 KEY81
BC27 VCCIO
BC29 GNDIO
BC31 DATAIN127
BC33 VCCIO
BC35 GNDIO
BC37 KEY20
BC39 VCCIO
BC41 GNDIO
BC43 ^CONF_DONE
BC45 DATAIN34
BC47 DATAOUT82
BD2 DATAIN18
BD4 DATAIN83
BD6 DATAOUT90
BD8 DATAIN113
BD10 RESERVED
BD12 RESERVED
BD14 DATAIN21

B20 KEY117
B22 KEY125
B24 WR_DATA
B26 KEY64
B28 DATAIN33
B30 RESERVED
B32 RESERVED
B34 RESERVED
B36 DATAIN14
B38 KEY120
B40 DATAIN73
B42 RESERVED
B44 RESERVED
B46 #TRST
C1 VCCINT
C3 ^MSEL1
C5 DATAIN42
C7 DATAIN124
C9 KEY89
C11 VCCINT
C13 GNDINT
C15 KEY25
C17 RESERVED
C19 VCCINT
C21 GNDINT
C23 RESERVED
C25 KEY39
C27 GNDINT
C29 VCCINT
C31 DATAIN48
C33 KEY68
C35 GNDINT
C37 VCCINT
C39 KEY85
C41 DATAIN64
C43 KEY57
C45 GNDINT
C47 VCCINT
D2 KEY109
D4 GNDINT
D6 RESERVED
D8 DATAIN122
D10 RESERVED
D12 DATAIN119
D14 KEY2
D16 KEY77
D18 RESERVED
D20 RESERVED
D22 KEY94
D24 VCCIO
D26 KEY9
D28 RESERVED
D30 KEY11
D32 KEY15
D34 RESERVED
D36 KEY59
D38 KEY103
D40 RESERVED
D42 DATAIN4
D44 DATAOUT71
D46 KEY108
E1 KEY42
E3 DATAIN71
E5 VCCINT
E7 GNDIO
E9 VCCIO
E11 DATAIN54
E13 GNDIO
E15 VCCIO
E17 DATAIN7
E19 GNDIO
E21 VCCIO
E23 DATAIN41
E25 KEY27
E27 VCCIO
E29 GNDIO
E31 RESERVED
E33 VCCIO
E35 GNDIO
E37 RESERVED
E39 VCCIO
E41 GNDIO
E43 ^nSTATUS
E45 DATAOUT95
E47 DATAOUT72
F2 KEY19
F4 DATAOUT54
F6 ^MSEL0

G25 VCCINT
G27 RESERVED
G29 RESERVED
G31 KEY8
G33 DATAIN79
G35 DATAOUT87
G37 RESERVED
G39 DATAIN95
G41 VCCIO
G43 GNDIO
G45 KEY80
G47 DATAOUT6
H2 DATAIN6
H4 DATAIN0
H6 DATAOUT42
H8 DATAOUT77
H38 RESERVED
H40 GNDIO
H42 KEY38
H44 DATAOUT35
H46 DATAOUT88
J1 DATAIN69
J3 KEY88
J5 VCCIO
J7 KEY54
J39 DATAIN72
J41 KEY72
J43 VCCIO
J45 RESERVED
J47 DATAIN111
K2 DATAOUT16
K4 DATAOUT0
K6 DATAIN43
K8 DATAOUT14
K40 DATAIN80
K42 DATAOUT80
K44 DATAIN19
K46 DATAIN117
L1 DATAOUT117
L3 VCCINT
L5 DATAOUT55
L7 DATAOUT19
L41 KEY35
L43 DATAIN51
L45 VCCINT
L47 DATAOUT79
M2 DATAIN37
M4 DATAIN120
M6 DATAIN35
M8 KEY6
M40 DATAOUT51
M42 DATAOUT120
M44 DATAIN109
M46 DATAIN118
N1 DATAIN11
N3 GNDINT
N5 GNDIO
N7 KEY36
N41 DATAIN88
N43 GNDIO
N45 GNDINT
N47 KEY91
P2 DATAIN91
P4 KEY92
P6 KEY69
P8 KEY111
P40 DATAOUT11
P42 DATAOUT69
P44 DATAOUT111
P46 KEY78
R1 DATAOUT45
R3 KEY114
R5 VCCIO
R7 DATAOUT26
R41 KEY37
R43 VCCIO
R45 KEY60
R47 DATAOUT47
T2 KEY47
T4 DATAOUT118
T6 DATAIN26
T8 DATAOUT36
T40 DATAOUT15
T42 DATAOUT37
T44 DATAIN114
T46 DATAIN47
U1 DATAIN46
U3 DATAOUT109

AA5 VCCIO
AA7 KEY7
AA41 KEY86
AA43 VCCIO
AA45 GNDINT
AA47 KEY107
AB2 DATAOUT13
AB4 DATAIN13
AB6 DATAOUT92
AB8 DATAIN36
AB40 KEY17
AB42 DATAOUT9
AB44 DATAIN53
AB46 DATAIN9
AC1 DATAOUT53
AC3 DATAOUT1
AC5 DATAIN87
AC7 KEY119
AC41 DATAOUT94
AC43 DATAOUT66
AC45 DATAIN56
AC47 DATAIN1
AD2 KEY87
AD4 VCCIO
AD6 GNDIO
AD8 DATAOUT119
AD40 KEY1
AD42 GNDIO
AD44 VCCIO
AD46 KEY66
AE1 DATAOUT56
AE3 DATAOUT113
AE5 DATAOUT121
AE7 DATAIN105
AE41 DATAOUT33
AE43 DATAOUT67
AE45 KEY40
AE47 DATAOUT116
AF2 DATAIN116
AF4 DATAOUT105
AF6 KEY116
AF8 KEY48
AF40 DATAOUT30
AF42 DATAIN89
AF44 KEY121
AF46 RESERVED
AG1 DATAIN121
AG3 GNDINT
AG5 VCCIO
AG7 KEY29
AG41 DATAOUT25
AG43 VCCIO
AG45 GNDINT
AG47 DATAOUT48
AH2 DATAOUT29
AH4 DATAOUT24
AH6 DATAIN22
AH8 DATAOUT97
AH40 DATAIN65
AH42 DATAOUT65
AH44 DATAOUT89
AH46 DATAOUT3
AJ1 DATAOUT123
AJ3 VCCINT
AJ5 GNDIO
AJ7 KEY99
AJ41 DATAIN99
AJ43 GNDIO
AJ45 VCCINT
AJ47 KEY123
AK2 KEY90
AK4 KEY63
AK6 DATAOUT5
AK8 KEY31
AK40 DATAOUT50
AK42 DATAIN12
AK44 KEY101
AK46 DATAOUT100
AL1 KEY23
AL3 DATAOUT99
AL5 DATAIN10
AL7 DATAOUT96
AL41 DATAOUT75
AL43 DATAOUT12
AL45 KEY3
AL47 DATAIN97
AM2 KEY79
AM4 DATAOUT41

AT6 DATAOUT58
AT8 DATAOUT124
AT40 DATAIN61
AT42 DATAOUT126
AT44 DATAOUT59
AT46 DATAIN28
AU1 DATAOUT112
AU3 VCCINT
AU5 DATAIN126
AU7 DATAOUT61
AU41 DATAIN82
AU43 KEY43
AU45 VCCINT
AU47 DATAIN20
AV2 DATAIN27
AV4 DATAOUT125
AV6 KEY18
AV8 DATAOUT18
AV40 KEY122
AV42 DATAOUT122
AV44 KEY50
AV46 DATAOUT4
AW1 DATAOUT7
AW3 DATAIN59
AW5 VCCIO
AW7 DATAIN32
AW9 DATAOUT84
AW39 DATAIN107
AW41 KEY83
AW43 VCCIO
AW45 DATAOUT64
AW47 KEY84
AY2 DATAIN60
AY4 DATAOUT108
AY6 DATAOUT34
AY8 GNDIO
AY10 DATAIN31
AY38 DATAIN40
AY40 GNDIO
AY42 DATAOUT83
AY44 DATAOUT76
AY46 DATAOUT43
BA1 DATAOUT38
BA3 DATAOUT93
BA5 GNDIO
BA7 VCCIO
BA9 RESERVED
BA11 DATAOUT22
BA13 RESERVED
BA15 RESERVED
BA17 KEY26
BA19 DATAIN90
BA21 DATAIN38
BA23 KEY93
BA25 RESERVED
BA27 KEY76
BA29 RESERVED
BA31 DATAIN44
BA33 RESERVED
BA35 DATAIN96
BA37 RESERVED
BA39 RESERVED
BA41 VCCIO
BA43 GNDIO
BA45 DATAOUT73
BA47 DATAOUT49
BB2 KEY_OK
BB4 DATAIN68
BB6 ^nCE
BB8 DATAIN93
BB10 RESERVED
BB12 DATAIN15
BB14 DATAIN2
BB16 RESERVED
BB18 DATAOUT28
BB20 KEY28
BB22 KEY113
BB24 GNDIO
BB26 RESERVED
BB28 RESERVED
BB30 RESERVED
BB32 KEY0
BB34 DATAIN75
BB36 KEY33
BB38 DATAIN66
BB40 RESERVED
BB42 #TDO
BB44 DATAOUT107

BD16 RESERVED
BD18 KEY112
BD20 RESERVED
BD22 DATAOUT2
BD24 VCCIO
BD26 DATAOUT85
BD28 DATAOUT68
BD30 RESERVED
BD32 DATAIN50
BD34 DATAIN39
BD36 RESERVED
BD38 KEY56
BD40 RESERVED
BD42 DATAIN85
BD44 GNDINT
BD46 KEY32
BE1 VCCINT
BE3 GNDINT
BE5 ^DCLK
BE7 DATAIN92
BE9 DATAIN55
BE11 VCCINT
BE13 GNDINT
BE15 KEY61
BE17 KEY82
BE19 VCCINT
BE21 GNDINT
BE23 DATAIN100
BE25 KEY96
BE27 GNDINT
BE29 VCCINT
BE31 DATAIN30
BE33 RESERVED
BE35 GNDINT
BE37 VCCINT
BE39 KEY70
BE41 KEY73
BE43 #TCK
BE45 GNDINT
BE47 VCCINT
BF2 KEY104
BF4 #TDI
BF6 DATAOUT23
BF8 RESERVED
BF10 DATAOUT63
BF12 RESERVED
BF14 KEY118
BF16 RESERVED
BF18 KEY100
BF20 KEY97
BF22 RESERVED
BF24 CLK
BF26 KEY21
BF28 KEY10
BF30 KEY98
BF32 DATAOUT127
BF34 KEY5
BF36 RESERVED
BF38 DATAOUT32
BF40 KEY13
BF42 KEY65
BF44 ^nCEO
BF46 KEY127
BG1 GNDINT
BG3 VCCINT
BG5 KEY74
BG7 RESERVED
BG9 RESERVED
BG11 KEY34
BG13 DATAIN112
BG15 RESERVED
BG17 DATAIN5
BG19 RESERVED
BG21 KEY71
BG23 SETUP
BG25 WR_KEY
BG27 RESERVED
BG29 RESERVED
BG31 KEY110
BG33 RESERVED
BG35 KEY105
BG37 KEY22
BG39 KEY58
BG41 DATAIN8
BG43 RESERVED
BG45 VCCINT
BG47 GNDINT

FIGURA A.13 - Pinagem do ENC128

## *A.4 Decriptador*

### A.4.1. Diagrama de blocos

Há quatro processos realizados em paralelo na arquitetura do dispositivo de 128 bits, como pode ser visto na figura abaixo.

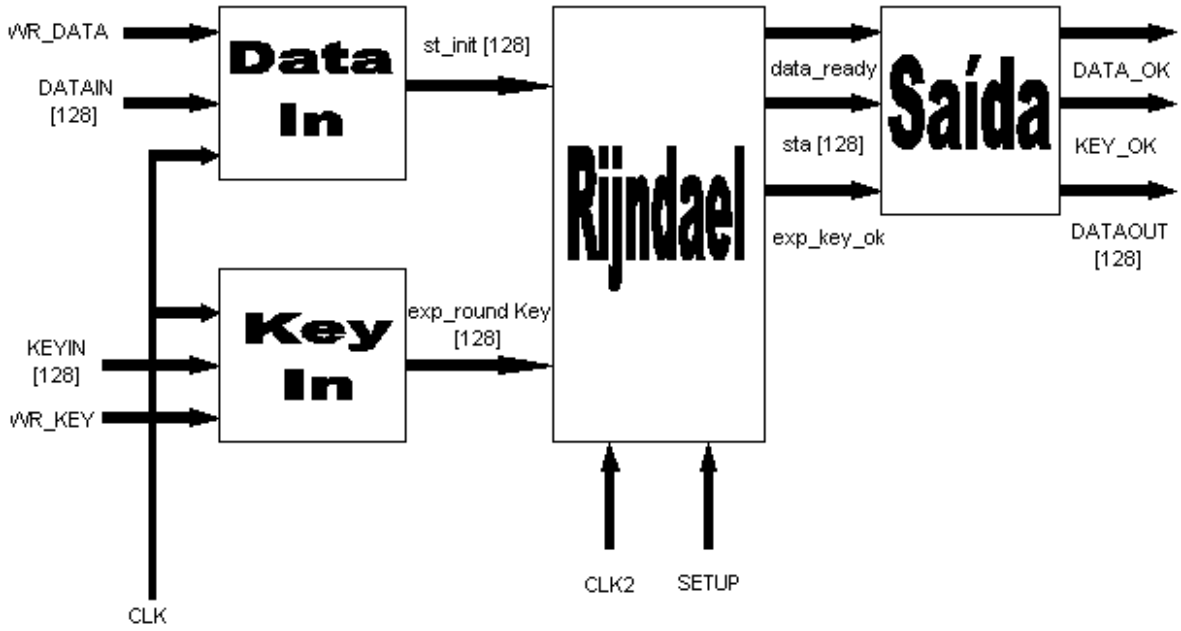


FIGURA A.14 - Diagrama de blocos do decriptador 128 bits.

O processo Data_In é responsável pela recepção dos dados de entrada. Ele toma os 128 bits de entrada de dados e os armazena em uma variável específica. Este processo é ativado sempre que o sinal WR_DATA é acionado e permite que o processador escreva um novo dado durante a encriptação do dado anterior. Para tanto, o processador deve esperar que o sinal DATA_OK seja acionado, como explicado anteriormente. Este processo executa, também, a primeira Add_Key_F.

O processo Key_In é responsável pela recepção da chave inicial, seu armazenamento e geração das chaves expandidas de rodada que serão usadas ao longo de todas as rodadas restantes. Para tanto, ele trabalha com uma variável de 11 componentes compostos por uma matriz de 4 linhas por 4 colunas de 8 bits. Cada um destes componentes é uma variável do tipo state_t, usada como tipo padrão ao longo do processamento. A chave inicial é armazenada no primeiro componente e os demais são gerados a partir dela. Este processo é ativado pelo sinal WR_KEY.

O processo Rijndael roda o algoritmo paralelamente em 128 bits. Há duas formas de funcionamento. Se o sinal SETUP estiver acionado, ele ajusta valores das variáveis para o início do funcionamento e coloca o dispositivo em modo de espera. Se o sinal não for acionado, o algoritmo é executado.

A decriptação consiste na realização de um IByte_Sub, seguido de um IShift_Row e, excetuando-se o *round* 0, quando não são executados, um Add_Key_F e um IMix_Column. No *round* 0, ao invés da execução do Mix_Column, é transferida para a variável de trabalho o novo dado de entrada (já armazenado pelo processo Data_In) e indicado o acionamento através do sinal DATA_OK.

O processo de saída simplesmente realiza a interface de saída de dados. Ele aciona os sinais DATA_OK e KEY_OK quando solicitado. Sua função mais importante, contudo, é garantir a estabilidade da saída ao longo de todo o processo, ou seja, ela será alterada somente quando existir uma nova saída para ser exibida.

### A.4.2. Performance

Há dois sinais de clock no dispositivo. O primeiro (CLK) é usado para a leitura do dado de entrada, da chave e expansão da chave. Sua velocidade máxima corresponde a um período de 80ns (12,5MHz), como mostra a Figura A.15.

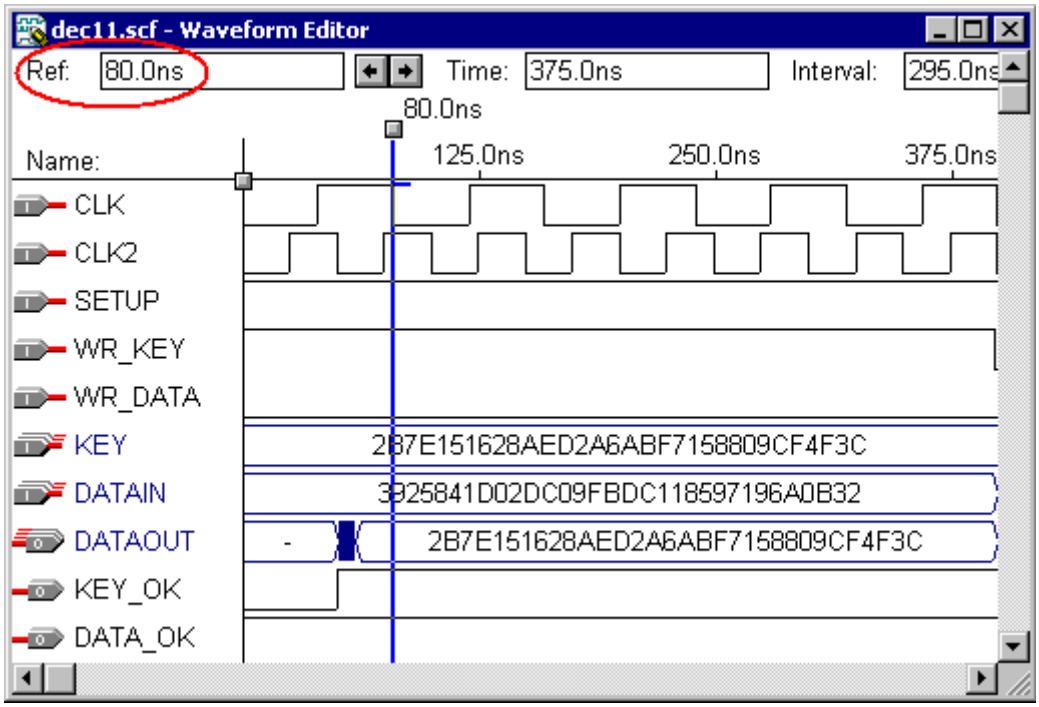


FIGURA A.15 - CLK mínimo do decriptador

A velocidade de processamento do dado depende do período de CLK2, tendo em vista que há um número pré estabelecido de rounds e que cada round requer um ciclo de CLK2. O melhor resultado obtido foi o uso de um período de 50 ns (20MHz), como mostra a figura A.16.

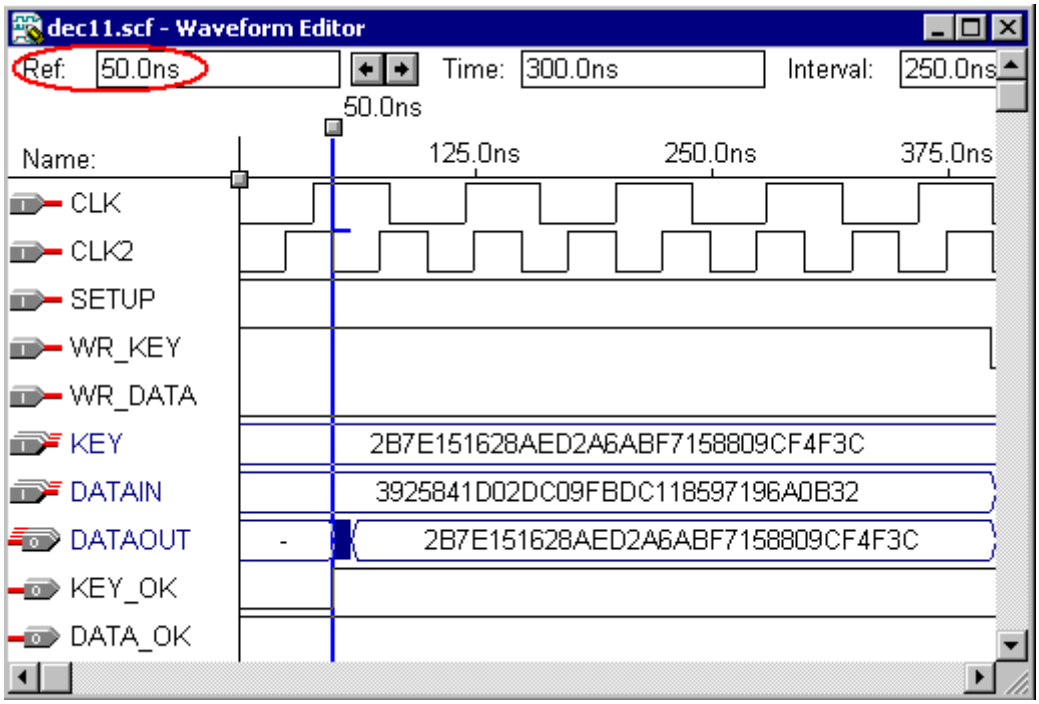


FIGURA A.16 - CLK2 mínimo do decriptador

Com o valor sugerido de CLK2, leva-se 420ns (nas figuras, 3,1866us menos 2,6866us) para uma decriptação completa de 128 bits. Estes valores permitem uma saída de dados de aproximadamente 256 MB/s. Vide figuras A.17 e A.18.

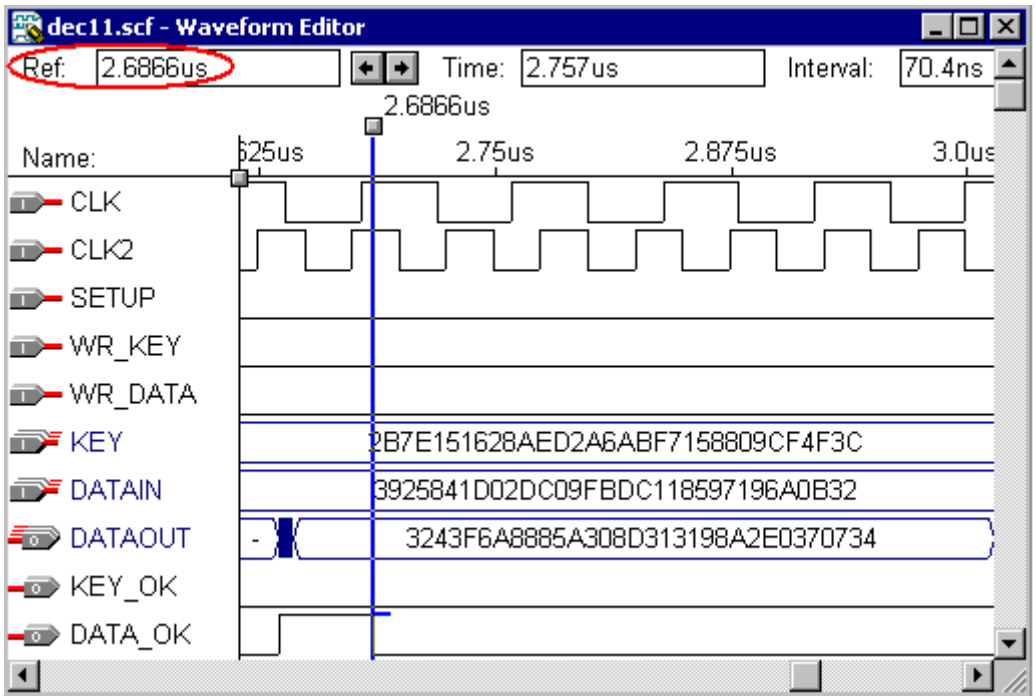


FIGURA A.17 - Primeiro dado pronto para ser lido

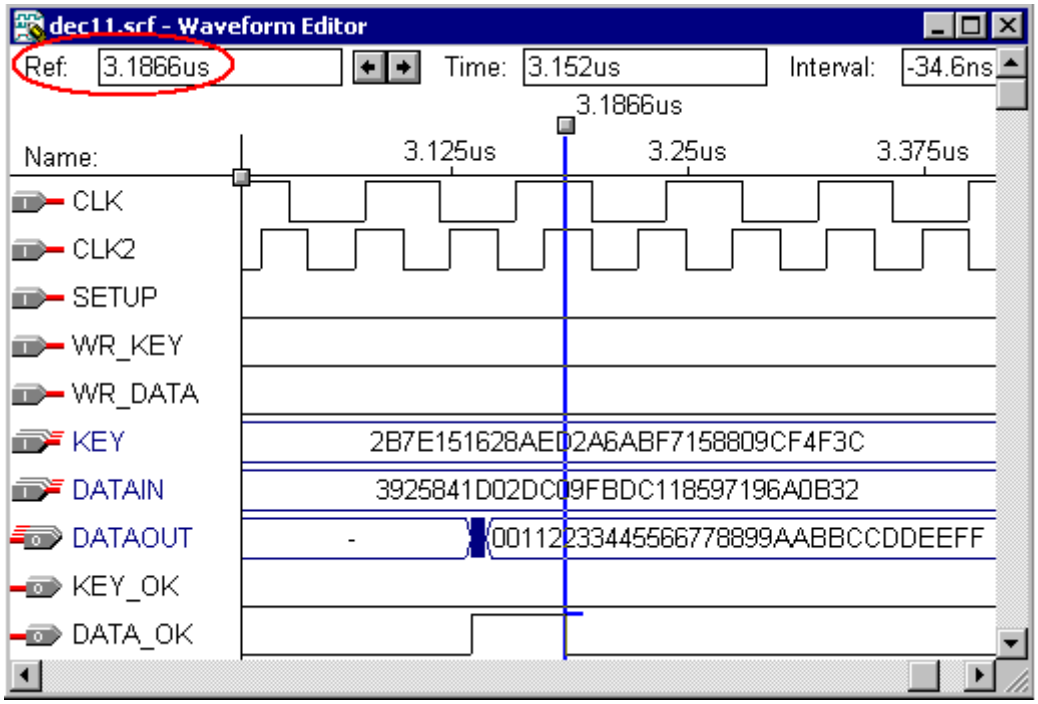


FIGURA A.18 - Segundo dado pronto para ser lido

### A.4.3 Utilização do componente

A implementação para o componente da Altera EPF10K250AGC599-1 apresenta a ocupação mostrada na tabela A.3. A memória do dispositivo é usada em sua totalidade, a pinagem é bastante exigida, mas há muito pouco uso de Logic Cells (LCs).

TABELA A.3 – Ocupação de área do decriptador 128 bits

| Design | Componente | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|---|
| DEC128 | EPF10K250AGC599-1 | 261 | 130 | 0 | 40960 | 5016 |

### A.4.4. Pinagem

O dispositivo usa 391 pinos de I/O. Mostra-se, a seguir, a vista inferior do EPF10K250AGC599-1 (com seu diagrama de pinos) e a função de cada um.

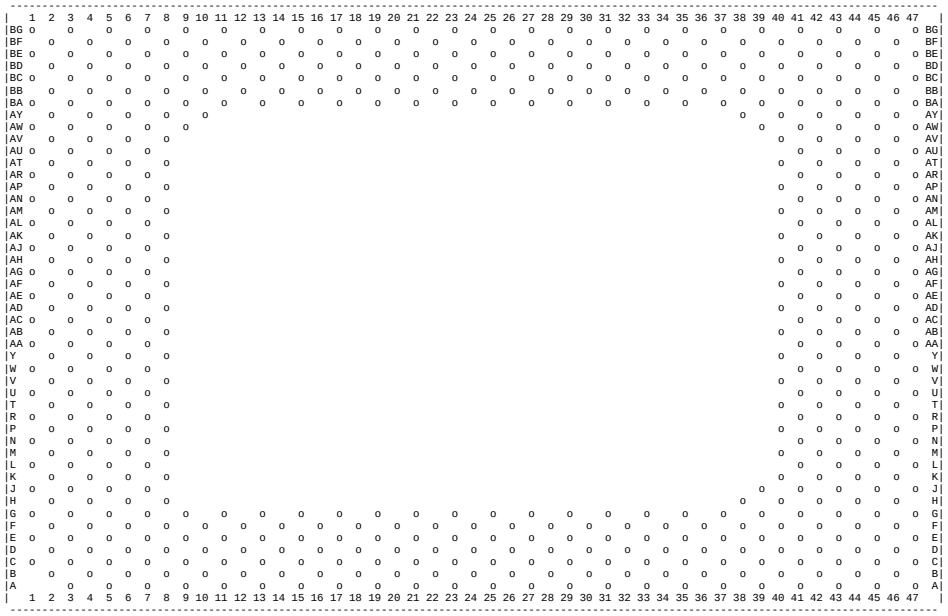

FIGURA A.19 - Vista inferior do EPF10K250AGC599-1

A3 VCCINT
A5 KEY91
A7 DATAIN14
A9 KEY34
A11 DATAIN19
A13 RESERVED
A15 DATAOUT61
A17 DATAIN114
A19 DATAIN84
A21 RESERVED
A23 RESERVED
A25 CLK2
A27 DATAIN55
A29 KEY33
A31 RESERVED
A33 RESERVED
A35 KEY47
A37 DATAIN116
A39 RESERVED
A41 DATAIN59
A43 RESERVED
A45 VCCINT
A47 GNDINT
B2 GNDINT
B4 ^nCONFIG
B6 DATAIN6
B8 DATAIN2
B10 KEY125
B12 RESERVED
B14 RESERVED
B16 KEY77
B18 DATAIN63
B20 KEY106
B22 DATAOUT23
B24 WR_DATA
B26 RESERVED
B28 RESERVED
B30 DATAIN120
B32 RESERVED
B34 RESERVED
B36 DATAOUT127
B38 KEY111
B40 KEY41
B42 RESERVED
B44 RESERVED
B46 #TRST
C1 VCCINT
C3 ^MSEL1
C5 DATAIN82
C7 DATAOUT63
C9 RESERVED
C11 VCCINT
C13 GNDINT
C15 DATAIN45
C17 DATAIN13
C19 VCCINT
C21 GNDINT
C23 RESERVED
C25 DATAIN0
C27 GNDINT
C29 VCCINT

F8 RESERVED
F10 DATAIN109
F12 RESERVED
F14 KEY121
F16 KEY57
F18 RESERVED
F20 DATAIN89
F22 KEY5
F24 GNDIO
F26 RESERVED
F28 DATAIN51
F30 DATAOUT58
F32 DATAOUT96
F34 RESERVED
F36 RESERVED
F38 RESERVED
F40 KEY78
F42 #TMS
F44 KEY68
F46 DATAOUT65
G1 DATAIN70
G3 DATAIN64
G5 GNDIO
G7 VCCIO
G9 KEY114
G11 DATAIN81
G13 DATAIN12
G15 DATAIN90
G17 RESERVED
G19 KEY81
G21 DATAIN101
G23 GNDINT
G25 VCCINT
G27 DATAIN44
G29 DATAIN65
G31 RESERVED
G33 DATAOUT62
G35 DATAIN105
G37 DATAIN29
G39 RESERVED
G41 VCCIO
G43 GNDIO
G45 DATAIN71
G47 DATAIN87
H2 KEY87
H4 KEY71
H6 DATAOUT27
H8 DATAOUT29
H38 RESERVED
H40 GNDIO
H42 DATAOUT26
H44 DATAOUT0
H46 KEY102
J1 DATAOUT76
J3 KEY108
J5 VCCIO
J7 DATA_OK
J39 DATAIN30
J41 DATAOUT15
J43 VCCIO
J45 DATAIN76

U5 DATAIN113
U7 DATAOUT106
U41 KEY116
U43 DATAIN91
U45 KEY88
U47 DATAIN67
V2 KEY124
V4 DATAIN21
V6 DATAOUT94
V8 DATAOUT14
V40 KEY90
V42 KEY113
V44 DATAIN48
V46 DATAOUT35
W1 KEY89
W3 VCCINT
W5 GNDIO
W7 DATAOUT90
W41 RESERVED
W43 GNDIO
W45 VCCINT
W47 DATAIN35
Y2 KEY59
Y4 KEY27
Y6 DATAIN22
Y8 KEY35
Y40 KEY67
Y42 KEY21
Y44 DATAIN124
Y46 DATAOUT78
AA1 KEY1
AA3 GNDINT
AA5 VCCIO
AA7 RESERVED
AA41 DATAOUT124
AA43 VCCIO
AA45 GNDINT
AA47 RESERVED
AB2 KEY74
AB4 KEY127
AB6 DATAIN74
AB8 DATAOUT1
AB40 DATAOUT74
AB42 KEY42
AB44 DATAIN78
AB46 DATAIN1
AC1 DATAOUT42
AC3 KEY85
AC5 KEY82
AC7 DATAOUT46
AC41 DATAIN46
AC43 KEY46
AC45 DATAOUT85
AC47 KEY109
AD2 DATAOUT109
AD4 VCCIO
AD6 GNDIO
AD8 DATAOUT5
AD40 DATAIN85
AD42 GNDIO
AD44 VCCIO

AM6 RESERVED
AM8 KEY10
AM40 KEY103
AM42 DATAIN26
AM44 KEY61
AM46 DATAOUT33
AN1 DATAOUT36
AN3 DATAIN38
AN5 VCCIO
AN7 DATAOUT10
AN41 KEY100
AN43 VCCIO
AN45 DATAOUT55
AN47 KEY44
AP2 DATAOUT59
AP4 KEY112
AP6 DATAOUT92
AP8 DATAOUT18
AP40 KEY92
AP42 DATAIN18
AP44 DATAOUT119
AP46 DATAOUT53
AR1 KEY26
AR3 GNDINT
AR5 GNDIO
AR7 DATAOUT8
AR41 KEY8
AR43 GNDIO
AR45 GNDINT
AR47 KEY96
AT2 RESERVED
AT4 KEY18
AT6 DATAIN9
AT8 RESERVED
AT40 DATAOUT44
AT42 DATAIN25
AT44 DATAIN8
AT46 KEY120
AU1 DATAIN53
AU3 VCCINT
AU5 DATAOUT69
AU7 DATAIN10
AU41 KEY93
AU43 KEY69
AU45 VCCINT
AU47 DATAIN119
AV2 KEY17
AV4 DATAIN69
AV6 DATAOUT4
AV8 DATAOUT67
AV40 KEY4
AV42 KEY65
AV44 DATAIN79
AV46 DATAOUT123
AW1 DATAIN103
AW3 DATAOUT125
AW5 VCCIO
AW7 DATAIN32
AW9 DATAOUT12
AW39 DATAOUT54
AW41 DATAIN98

BB46 DATAOUT32
BC1 DATAOUT93
BC3 DATAOUT98
BC5 ^DATA0
BC7 GNDIO
BC9 VCCIO
BC11 KEY72
BC13 GNDIO
BC15 VCCIO
BC17 DATAIN121
BC19 GNDIO
BC21 VCCIO
BC23 RESERVED
BC25 RESERVED
BC27 VCCIO
BC29 GNDIO
BC31 KEY56
BC33 VCCIO
BC35 GNDIO
BC37 DATAIN33
BC39 VCCIO
BC41 GNDIO
BC43 ^CONF_DONE
BC45 KEY28
BC47 DATAOUT115
BD2 DATAOUT79
BD4 KEY110
BD6 RESERVED
BD8 KEY13
BD10 KEY_OK
BD12 RESERVED
BD14 DATAIN72
BD16 DATAIN86
BD18 DATAIN57
BD20 RESERVED
BD22 KEY63
BD24 VCCIO
BD26 DATAIN40
BD28 RESERVED
BD30 DATAIN66
BD32 KEY122
BD34 DATAIN36
BD36 KEY60
BD38 RESERVED
BD40 DATAIN37
BD42 DATAIN107
BD44 GNDINT
BD46 KEY32
BE1 VCCINT
BE3 GNDINT
BE5 ^DCLK
BE7 RESERVED
BE9 KEY15
BE11 VCCINT
BE13 GNDINT
BE15 RESERVED
BE17 KEY24
BE19 VCCINT
BE21 GNDINT
BE23 RESERVED
BE25 DATAOUT113

| | | | | |
|---|---|---|---|---|
| C31 DATAIN104 | J47 DATAOUT126 | AD46 KEY62 | AW43 VCCIO | BE27 GNDINT |
| C33 KEY115 | K2 DATAIN108 | AE1 DATAOUT82 | AW45 RESERVED | BE29 VCCINT |
| C35 GNDINT | K4 DATAOUT64 | AE3 KEY31 | AW47 DATAIN92 | BE31 RESERVED |
| C37 VCCINT | K6 KEY3 | AE5 KEY43 | AY2 KEY9 | BE33 RESERVED |
| C39 DATAIN112 | K8 DATAOUT88 | AE7 DATAIN43 | AY4 DATAIN54 | BE35 GNDINT |
| C41 RESERVED | K40 DATAOUT6 | AE41 DATAOUT75 | AY6 KEY12 | BE37 VCCINT |
| C43 RESERVED | K42 KEY64 | AE43 KEY7 | AY8 GNDIO | BE39 RESERVED |
| C45 GNDINT | K44 DATAOUT68 | AE45 DATAOUT43 | AY10 KEY98 | BE41 KEY52 |
| C47 VCCINT | K46 DATAOUT105 | AE47 DATAIN75 | AY38 RESERVED | BE43 #TCK |
| D2 DATAOUT31 | L1 DATAOUT83 | AF2 DATAOUT7 | AY40 GNDIO | BE45 GNDINT |
| D4 GNDINT | L3 VCCINT | AF4 KEY75 | AY42 KEY30 | BE47 VCCINT |
| D6 DATAOUT39 | L5 DATAOUT73 | AF6 KEY38 | AY44 KEY54 | BF2 DATAIN28 |
| D8 DATAIN68 | L7 DATAOUT89 | AF8 DATAOUT95 | AY46 DATAIN24 | BF4 #TDI |
| D10 RESERVED | L41 KEY70 | AF40 KEY80 | BA1 DATAOUT103 | BF6 RESERVED |
| D12 RESERVED | L43 KEY0 | AF42 DATAOUT49 | BA3 DATAIN93 | BF8 RESERVED |
| D14 DATAIN27 | L45 VCCINT | AF44 DATAOUT100 | BA5 GNDIO | BF10 KEY36 |
| D16 KEY39 | L47 DATAIN127 | AF46 DATAIN7 | BA7 VCCIO | BF12 KEY86 |
| D18 KEY105 | M2 DATAOUT25 | AG1 DATAOUT11 | BA9 RESERVED | BF14 RESERVED |
| D20 RESERVED | M4 DATAOUT87 | AG3 GNDINT | BA11 KEY14 | BF16 KEY94 |
| D22 KEY104 | M6 DATAOUT86 | AG5 VCCIO | BA13 RESERVED | BF18 DATAIN88 |
| D24 VCCIO | M8 KEY73 | AG7 DATAOUT84 | BA15 KEY16 | BF20 DATAIN99 |
| D26 KEY40 | M40 DATAOUT41 | AG41 DATAIN95 | BA17 KEY117 | BF22 KEY50 |
| D28 KEY119 | M42 KEY58 | AG43 VCCIO | BA19 DATAIN102 | BF24 CLK |
| D30 RESERVED | M44 DATAIN100 | AG45 GNDINT | BA21 DATAOUT108 | BF26 KEY123 |
| D32 DATAIN126 | M46 DATAOUT97 | AG47 KEY95 | BA23 DATAIN34 | BF28 RESERVED |
| D34 RESERVED | N1 DATAOUT104 | AH2 KEY45 | BA25 KEY66 | BF30 RESERVED |
| D36 RESERVED | N3 GNDINT | AH4 DATAIN49 | BA27 RESERVED | BF32 KEY51 |
| D38 RESERVED | N5 GNDIO | AH6 KEY20 | BA29 RESERVED | BF34 DATAIN41 |
| D40 RESERVED | N7 DATAOUT107 | AH8 DATAOUT117 | BA31 DATAIN115 | BF36 DATAOUT70 |
| D42 KEY6 | N41 KEY107 | AH40 DATAOUT101 | BA33 RESERVED | BF38 DATAIN97 |
| D44 DATAOUT30 | N43 GNDIO | AH42 DATAIN77 | BA35 DATAOUT56 | BF40 RESERVED |
| D46 DATAOUT2 | N45 GNDINT | AH44 DATAOUT80 | BA37 DATAIN106 | BF42 KEY37 |
| E1 DATAOUT24 | N47 KEY118 | AH46 DATAIN80 | BA39 KEY2 | BF44 ^nCEO |
| E3 DATAOUT17 | P2 DATAOUT91 | AJ1 DATAOUT45 | BA41 VCCIO | BF46 DATAIN111 |
| E5 VCCINT | P4 DATAOUT102 | AJ3 VCCINT | BA43 GNDIO | BG1 GNDINT |
| E7 GNDIO | P6 DATAOUT81 | AJ5 GNDIO | BA45 DATAIN4 | BG3 VCCINT |
| E9 VCCIO | P8 KEY23 | AJ7 DATAOUT77 | BA47 DATAOUT9 | BG5 DATAIN15 |
| E11 KEY49 | P40 DATAIN83 | AJ41 DATAOUT19 | BB2 DATAOUT72 | BG7 DATAIN16 |
| E13 GNDIO | P42 KEY76 | AJ43 GNDIO | BB4 DATAOUT66 | BG9 KEY83 |
| E15 VCCIO | P44 DATAIN23 | AJ45 VCCINT | BB6 ^nCE | BG11 RESERVED |
| E17 DATAIN17 | P46 DATAIN94 | AJ47 KEY84 | BB8 DATAIN73 | BG13 DATAIN61 |
| E19 GNDIO | R1 KEY99 | AK2 DATAOUT50 | BB10 RESERVED | BG15 KEY11 |
| E21 VCCIO | R3 DATAOUT99 | AK4 DATAIN56 | BB12 DATAIN50 | BG17 DATAIN5 |
| E23 DATAIN125 | R5 VCCIO | AK6 KEY55 | BB14 KEY79 | BG19 RESERVED |
| E25 DATAIN118 | R7 DATAOUT51 | AK8 DATAOUT34 | BB16 DATAIN39 | BG21 DATAIN110 |
| E27 VCCIO | R41 DATAOUT112 | AK40 RESERVED | BB18 DATAIN31 | BG23 SETUP |
| E29 GNDIO | R43 VCCIO | AK42 DATAOUT47 | BB20 RESERVED | BG25 WR_KEY |
| E31 RESERVED | R45 KEY29 | AK44 DATAOUT57 | BB22 DATAIN3 | BG27 DATAIN122 |
| E33 VCCIO | R47 KEY48 | AK46 DATAOUT20 | BB24 GNDIO | BG29 DATAOUT40 |
| E35 GNDIO | T2 DATAOUT22 | AL1 KEY19 | BB26 DATAOUT37 | BG31 RESERVED |
| E37 KEY97 | T4 DATAOUT114 | AL3 DATAIN20 | BB28 DATAIN60 | BG33 DATAIN62 |
| E39 VCCIO | T6 DATAOUT120 | AL5 DATAOUT13 | BB30 KEY126 | BG35 DATAOUT38 |
| E41 GNDIO | T8 DATAOUT121 | AL7 DATAOUT110 | BB32 DATAOUT116 | BG37 DATAIN52 |
| E43 ^nSTATUS | T40 DATAOUT122 | AL41 RESERVED | BB34 DATAIN123 | BG39 RESERVED |
| E45 DATAOUT3 | T42 KEY25 | AL43 DATAOUT52 | BB36 DATAIN42 | BG41 DATAIN47 |
| E47 DATAIN11 | T44 DATAOUT118 | AL45 DATAOUT16 | BB38 DATAIN58 | BG43 DATAIN96 |
| F2 DATAOUT71 | T46 KEY22 | AL47 KEY101 | BB40 DATAOUT60 | BG45 VCCINT |
| F4 DATAOUT28 | U1 DATAOUT21 | AM2 DATAIN117 | BB42 #TDO | BG47 GNDINT |
| F6 ^MSEL0 | U3 DATAOUT48 | AM4 KEY53 | BB44 DATAOUT111 | |

FIGURA A.20 - Pinagem do DEC128

## *A.5 Resultados encriptador versus decriptador*

Nos ensaios, constatou-se uma diferença significativa entre os resultados obtidos pelo decriptador e encriptador, mesmo ambos possuindo a mesma arquitetura. Após algumas simulações, verificou-se que a diferença vinha, basicamente, da diferença entre as funções Mix_Column e IMix_Column.

Para demonstrar isto, implementou-se o mesmo código excluindo-se as funções citadas. Com esta modificação, o encriptador passou a funcionar com um período de 32ns (figura A.21), apresentando a saída ainda correta para um algoritmo sem Mix_Column(figura A.22).

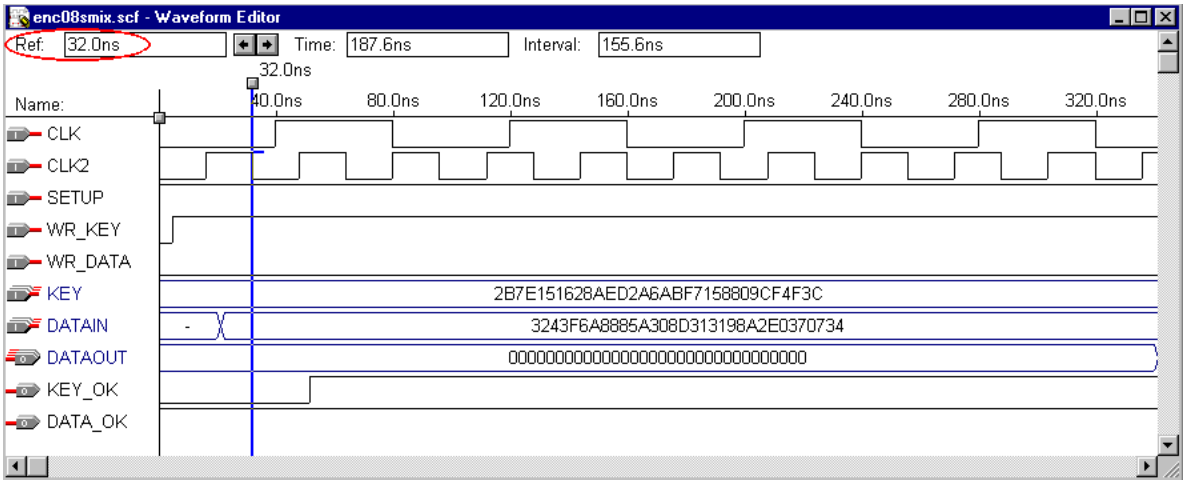


FIGURA A.21 - Novo CLK2 mínimo do encriptador sem o Mix_Column

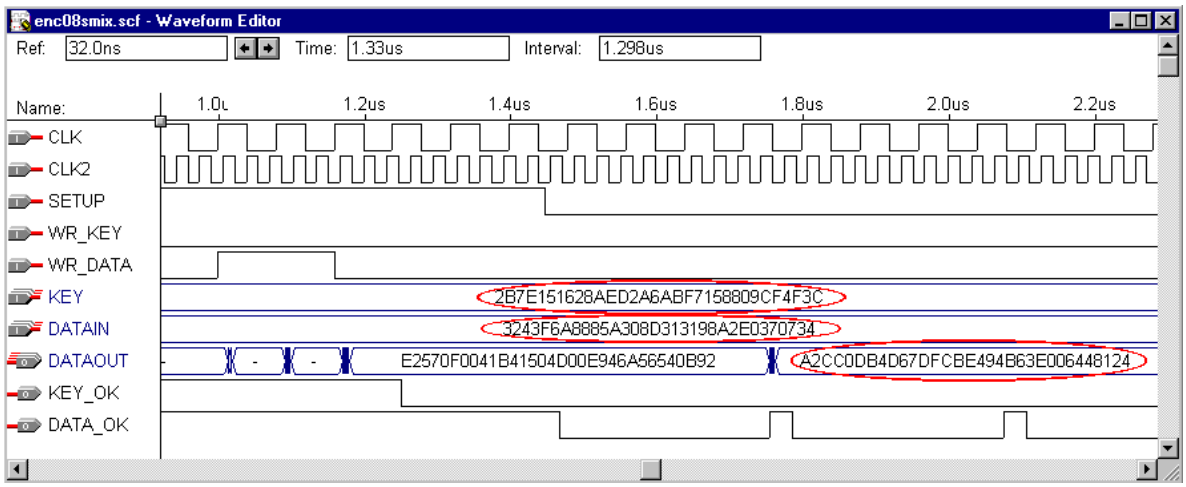


FIGURA A.22 - Saída do encriptador sem o Mix_Column

Já o decriptador passou a funcionar com um período mínimo de 34ns (figura A.23), apresentando resultados corretos, para um algoritmo sem Imix_Column, que validam os resultados do encriptador (figura A.24).

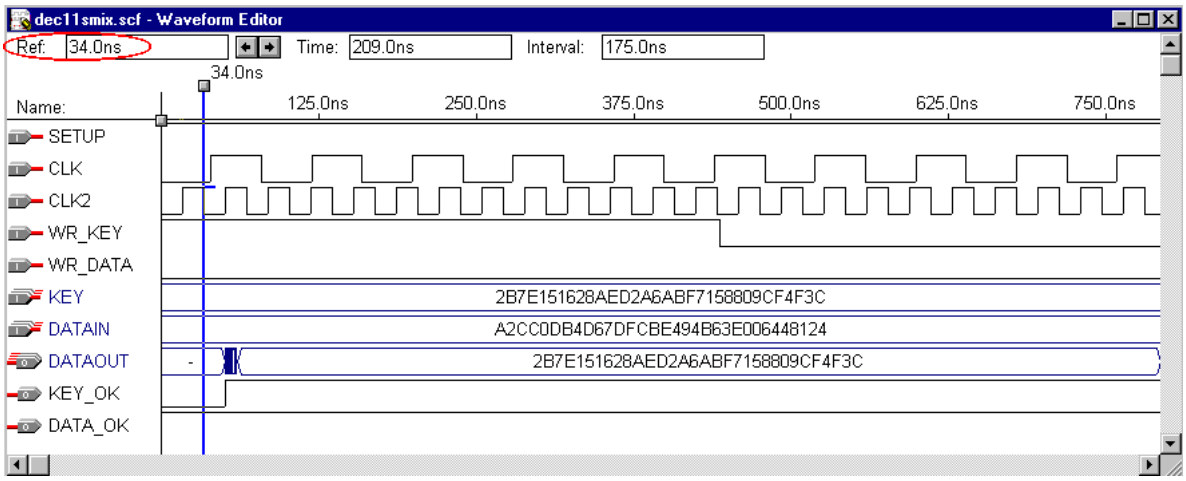


FIGURA A.23 - Novo CLK2 mínimo do decriptador sem o IMix_Column

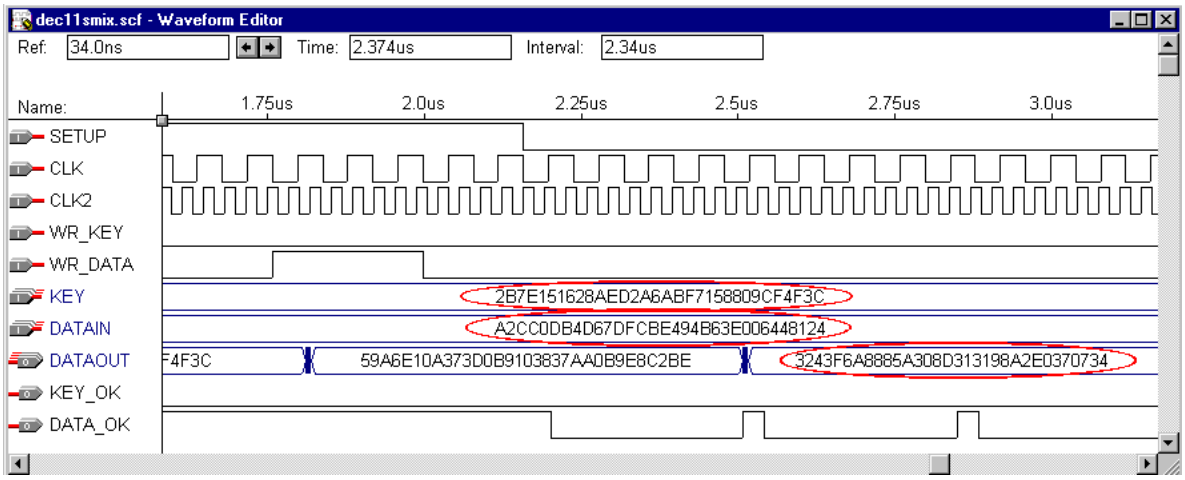


FIGURA A.24 - Saída do decriptador sem o IMix_Column

# Anexo2 Dispositivo 32 bits

## *B.1 Interface*

O encriptador/decriptador Rijndael de 32 bits recebe os dados a serem criptografados/decriptografados, bem como a chave, em 32 bits. Sua saída apresenta dados encriptados/decriptados diretamente em 128 bits. É utilizado o componente da Altera EPF10K50VBC484-1, com 291 pinos de I/O, 2880 LCs e 20480 bits de RAM embutida. Decriptador e encriptador usam um (1) FPGA cada.

O objetivo deste projeto é produzir um processamento de 32 bits, pois, desta forma, uma quantidade menor de memória é necessária para as operações da função ByteSub, permitindo o uso de componentes mais simples e baratos. Os componentes FLEX10KA50 possuem a quantidade de memória necessária para permitir a implementação da função. Conforme será mostrado a seguir, neste dispositivo há sobra de células lógicas, e de uma pequena parte de memória RAM, mas um número pequeno de pinos. Por este motivo, foi escolhida uma entrada de dados em 32 bits, mas uma saída de 128 bits.

A entrada de dados toma 32 bits por vez, armazenando-os em uma variável matricial de 4 por 4 bytes, totalizando 128 bits. Toda a operação, tanto de leitura quanto de processamento de dados, é executada sobre as linhas desta variável, exceto pelas funções Mix_Column e InvMix_Column, que agem sobre as colunas.

A leitura da entrada é composta por quatro estados, cada um responsável pela leitura e armazenamento de uma etapa de 32 bits dos 128 bits totais. A entrada é compartilhada entre dados e chave, permitindo uma redução ainda maior no número de pinos de I/O utilizados. A saída utiliza 128 bits diretamente, evitando uma necessária troca de sinais de controle entre processador e dispositivo.

TABELA B.1 – Pinagem para o dispositivo de 32 bits

| Nome | Tipo | Descrição |
|---|---|---|
| CLK | IN | Relógio comum a todos os blocos. |
| SETUP | IN | Determina período de configuração/operação. |
| WR_KEY | IN | Indica que a chave de entrada está no barramento. |
| WR_DATA | IN | Indica que os dados de entrada estão no barramento. |
| AIN | IN | Indica qual das palavras de 32 bits de entrada está no barramento. |
| DIN | IN | Entrada de dados. Serve tanto para a chave quanto para o dado. |
| DATA_OK | OUT | Indica permissão de leitura/escrita no barramento. |
| DATAOUT | OUT | Saída de dados em 128 bits. |

### *B.2 Operação*

Neste dispositivo não há a geração de todas as chaves expandidas a partir da chave inicial. Ao contrário, a cada rodada utiliza-se a chave atual para gerar a próxima. Assim, deve ser preocupação do controle central escrever o novo dado e a nova chave antes do início da próxima rodada. Ainda existem dois modos distintos de operação nos dispositivos, coordenados pelo sinal SETUP. Estando em lógica zero, o sinal permite o processamento contínuo do algoritmo Rijndael. Mesmo quando em operação contínua é possível trocar a chave de encriptação, respeitando-se somente os tempos permitidos de escrita, sinalizados pelo sinal DATA_OK, sem a necessidade do sinal SETUP ser acionado. Ao ser acionado o sinal SETUP, o processamento é interrompido, fazendo com que todas as variáveis internas e máquinas de estado sejam inicializadas. Em geral é feita a escrita da chave e do primeiro dado a ser encriptado/decriptado neste momento.

A fim de efetuar a escrita da chave deve-se acionar o sinal WR_KEY, mantendo o sinal WR_DATA em valor zero, pois ambos utilizam o mesmo barramento de dados e endereço, conforme mostrado na figura B.1. O valor da chave deve ser escrito em palavras de 32 bits. Para indicar qual das quatro palavras serão escritas, a entrada AIN deve ser convenientemente assinalada, ou seja, caso deseje-se colocar no barramento o valor da primeira palavra, deve-se indicar em AIN o valor 00.

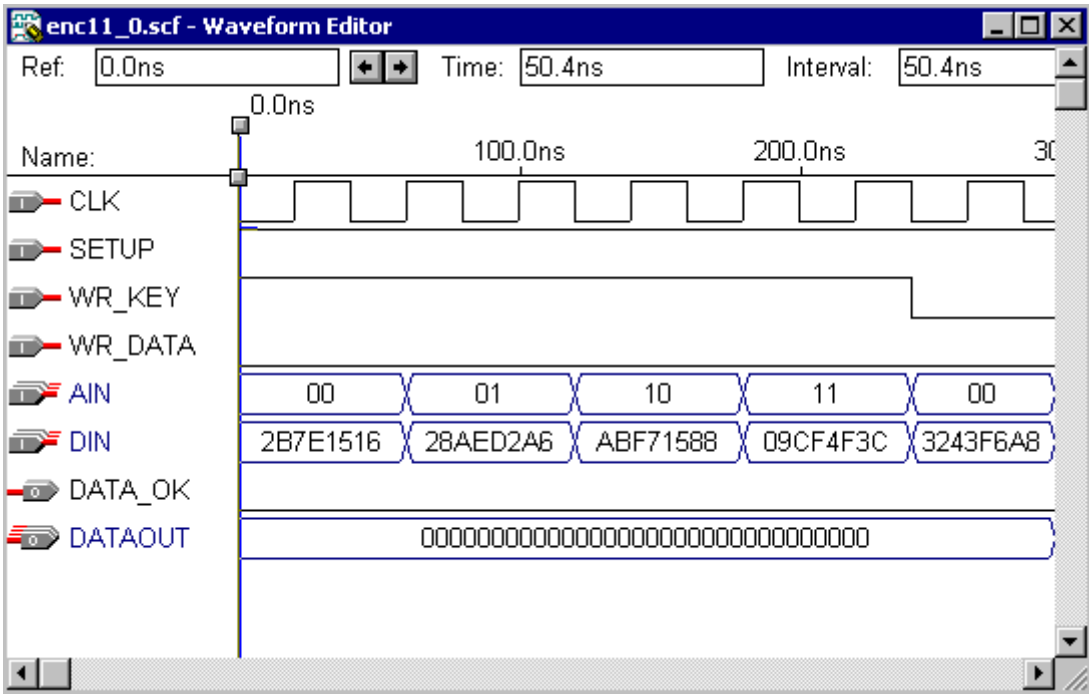


FIGURA B.1 - Entrada da chave

É importante clarificar a única diferença de funcionamento entre encriptador e decriptador. No primeiro, o conjunto de bits da chave é realmente a chave definida para o processo de encriptação e decriptação. Já no segundo, a chave inicial a ser enviada pelo processador deve ser, na verdade, a última chave produzida pela expansão da chave, ou seja, é tarefa do controle central gerar a expansão. Isto é necessário, pois as chaves de rodada são calculadas iterativamente. Caso não fosse colocada a chave final, seria necessário para a decriptação esperar o tempo de geração das chaves, diferindo substancialmente da arquitetura da encriptação.

O dado incial deve ser escrito mantendo-se WR_DATA ativado, com o sinal WR_KEY em zero lógico. O valor da chave deve ser escrito em palavras de 32 bits. Para indicar qual das quatro palavras serão escritas, a entrada AIN deve ser convenientemente assinalada, ou seja, caso deseje-se colocar no barramento o valor da primeira palavra, deve-se indicar em AIN o valor 00, mostrado na figura B.2.

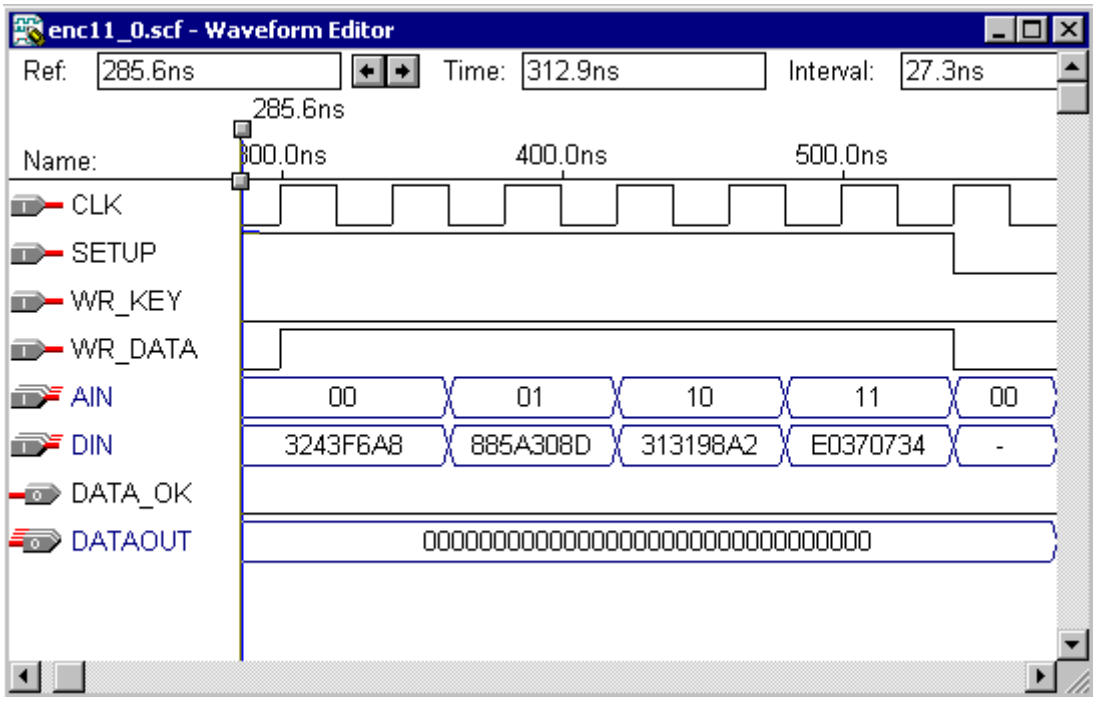


FIGURA B.2 - Entrada do dado inicial

.

Quando o sinal SETUP está em zero lógico, permite-se o processamento contínuo do processo Rijndael. Tão logo seja possível ao dispositivo operar com um dado novo, ele acionará o sinal DATA_OK, cuja borda descendente indica que é possível a escrita de um novo dado. É importante que não ocorra muita demora do processador central para responder, pois na próxima borda de subida o novo dado já deve estar escrito. Para escrever cada dado novo, basta repetir o procedimento descrito acima. É bom lembrar que uma nova chave pode também ser escrita. Entretanto, tanto a escrita do novo dado, quanto da nova chave, devem acontecer no intervalo definido por DATA_OK.

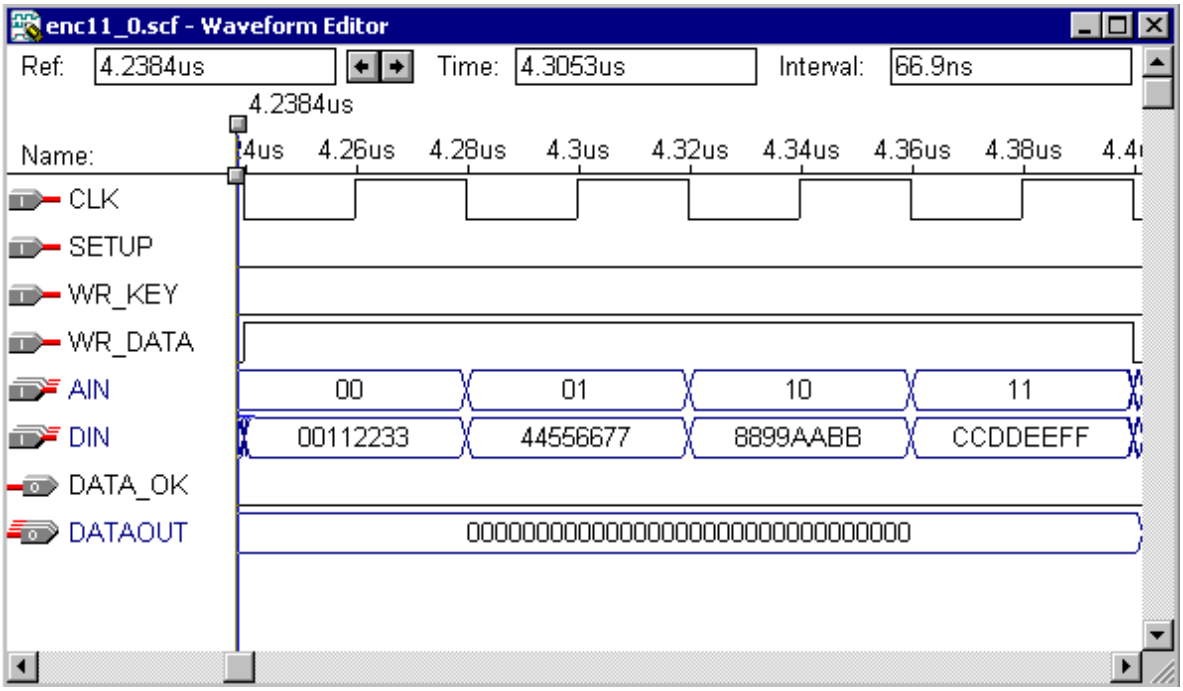


FIGURA B.3 - Entrada de dados novos ao longo da operação

Uma nova saída no barramento é colocada também após a borda descendente do sinal CLK, conforme mostrado na figura B.4.

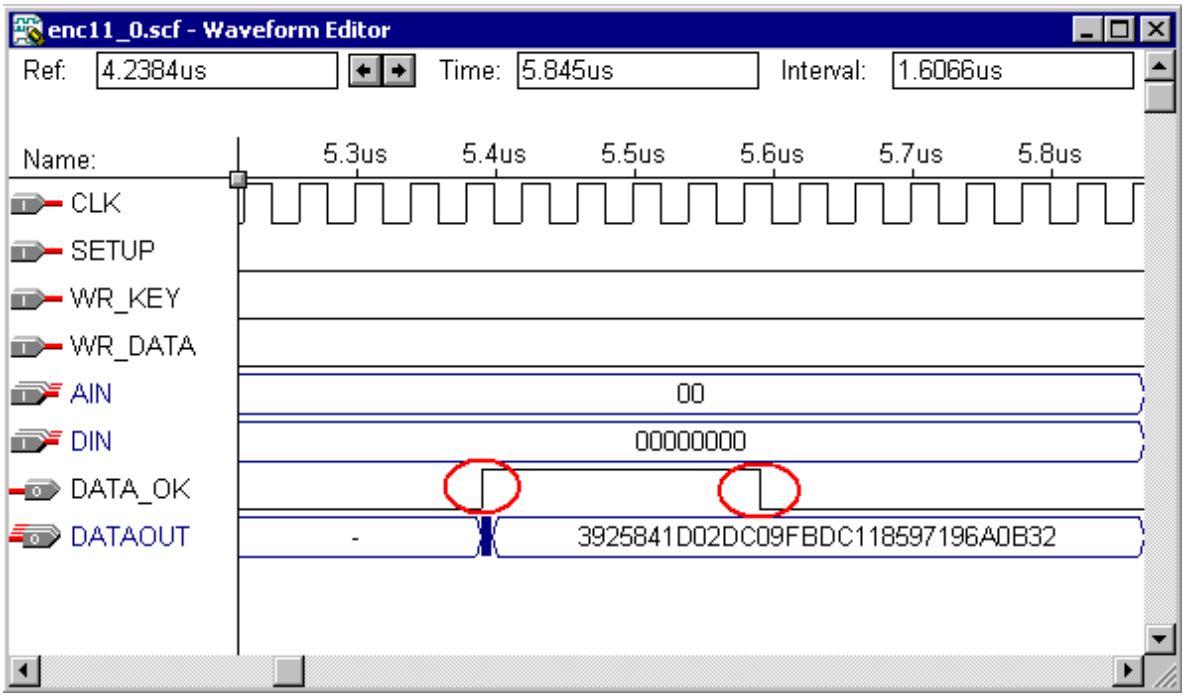


FIGURA B.4 - DATA_OK indica quando pode-se ler e escrever dados no barramento

O sinal SETUP serve, também, para interromper o processamento por qualquer motivo, deixando o dispositivo em modo de espera. Assim, quando não existirem dados a serem encriptados/decriptados, deve-se levantar o sinal SETUP, após a leitura do último dado de interesse, conforme mostrado na figura B.5.

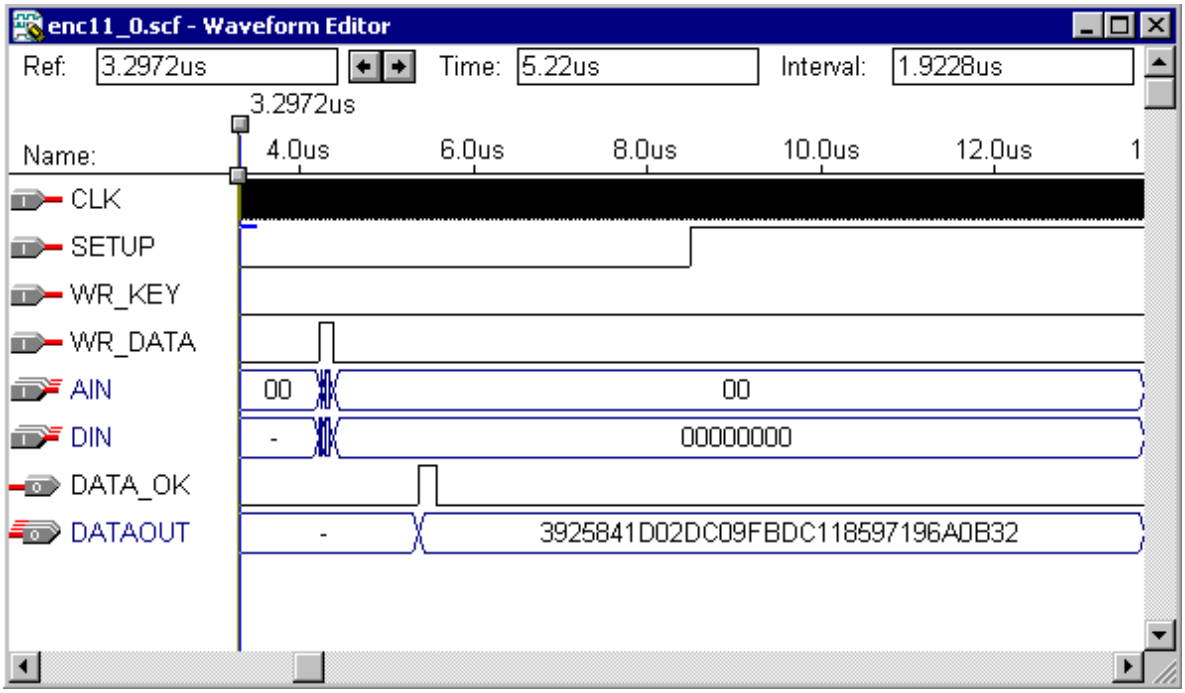


FIGURA B.5 - Modo de espera do dispositivo

Para reiniciar o dispositivo, deve-se manter o SETUP ativado, permitindo a escrita da chave e do dado iniciais. Isso feito, o sinal de SETUP deve ser desativado e o dispositivo operado normalmente, conforme mostra a figura B.6.

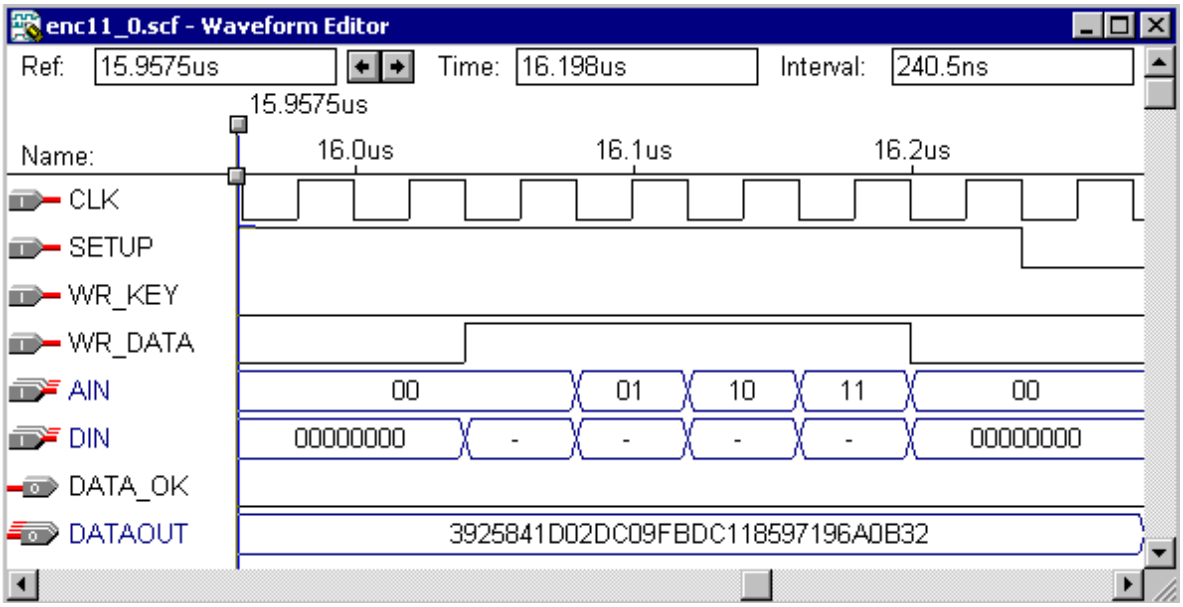


FIGURA B.6 - Entrada de dados antes de sair do Halt

## *B.3 Encriptador*

### B.3.1. Diagrama de blocos

Há quatro processos realizados em paralelo na arquitetura dos dispositivos, como pode ser visto na figura B.7.

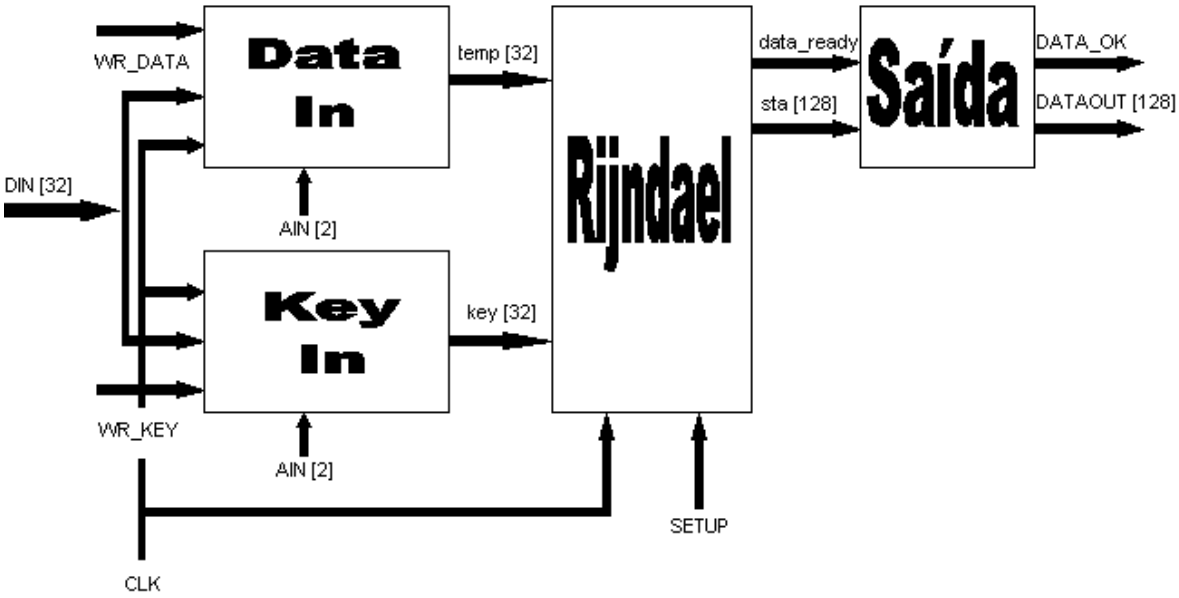


FIGURA B.7 - Diagrama de blocos do encriptador 32 bits

O processo Data_In é responsável pela recepção dos dados de entrada. Ele toma 32 bits por vez, na ordem estabelecida pelo processador, através do sinal AIN, e armazena-os em uma variável temporária de 128 bits. Este processo é ativado sempre que o sinal WR_DATA é acionado e permite que o processador escreva um novo dado durante a encriptação do dado anterior.

O processo Key_In é semelhante ao Data_in. Armazena a chave de entrada em uma variável fixa para a chave. Isto ocorre, pois a chave não é trocada a todo o momento e, assim sendo, fica armazenada no dispositivo até que deseje-se trocá-la. Toma 32 bits por vez, na ordem estabelecida pelo processador, através do sinal AIN. Este processo é ativado ao se acionar o sinal WR_KEY.

O processo Rijndael processa o algoritmo em 32 bits. Há duas formas de funcionamento. Se o sinal SETUP estiver acionado, ele ajusta valores das variáveis para o início do funcionamento. Se o sinal não for acionado, o algoritmo é executado. Nessa segunda opção, existem quatro estados de operação para processamento em 32 bits. O primeiro estado é executado apenas na primeira rodada de funcionamento, sendo responsável pela Add_Key_F inicial, entre as variáveis temporária de dados e a chave inicial, armazenando o resultado na variável de trabalho. Além disso, ajusta valores corretos da rodada atual e alimenta a variável round_key com seu valor inicial. Na verdade, este processo é, por si só, todo o *round* 0.

O segundo estado é executado em todas as rodadas. Consiste em realizar as funções Byte_Sub e Shift_Row sobre a variável de trabalho. O resultado da operação é armazenado na variável de trabalho. Este estado, ainda, executa a formação da round_key para a rodada seguinte e aciona, via processo Saída, o sinal DATA_OK, para

avisar ao processador que um novo dado pode ser enviado ao barramento e que o dado processado pode ser lido.

O terceiro estado também é executado em todas as rodadas, a exceção da última. Consiste em realizar a função Mix Column. O quarto estado é executado em todas as rodadas e consiste, simplesmente, na execução da função Add Round Key.

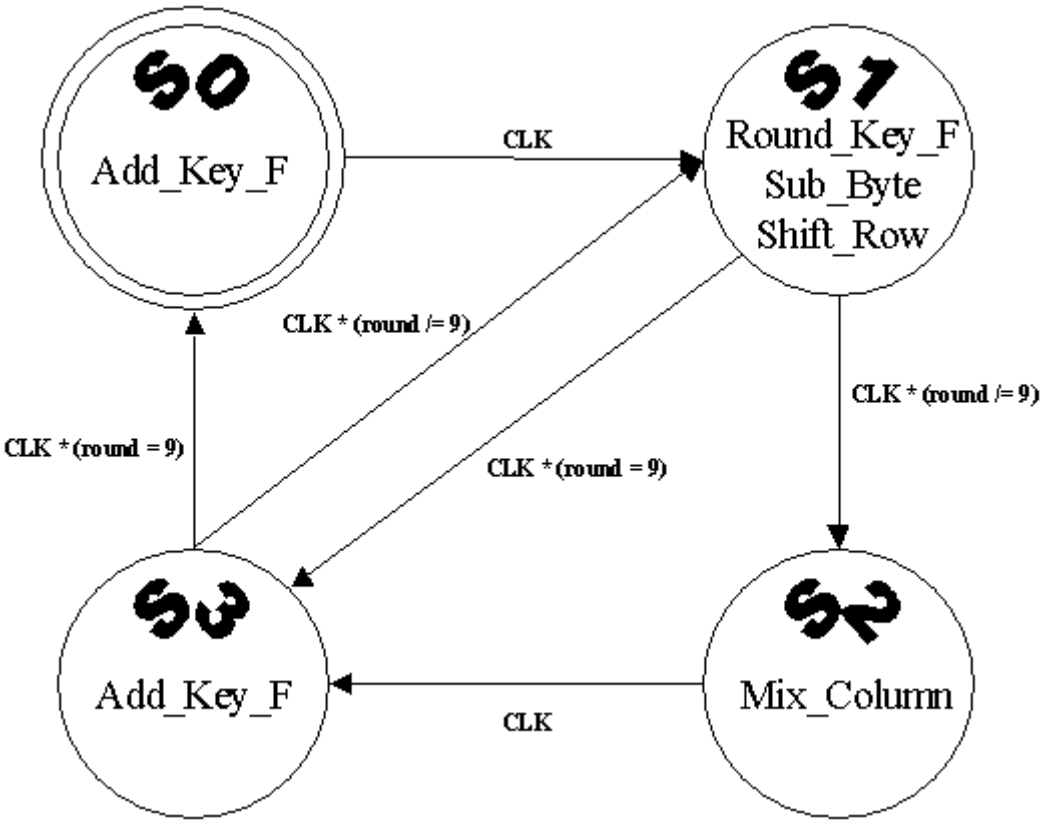


FIGURA B.8 - Máquina de estados do encriptador 32 bits

## B.3.2. Desempenho

A velocidade de processamento do dado depende do período de CLK, tendo em vista que há um número pré estabelecido de rodadas e que cada rodada leva um período de clock. O melhor resultado obtido foi o uso de um período de 40 ns, conforme mostrado na figura B.9.

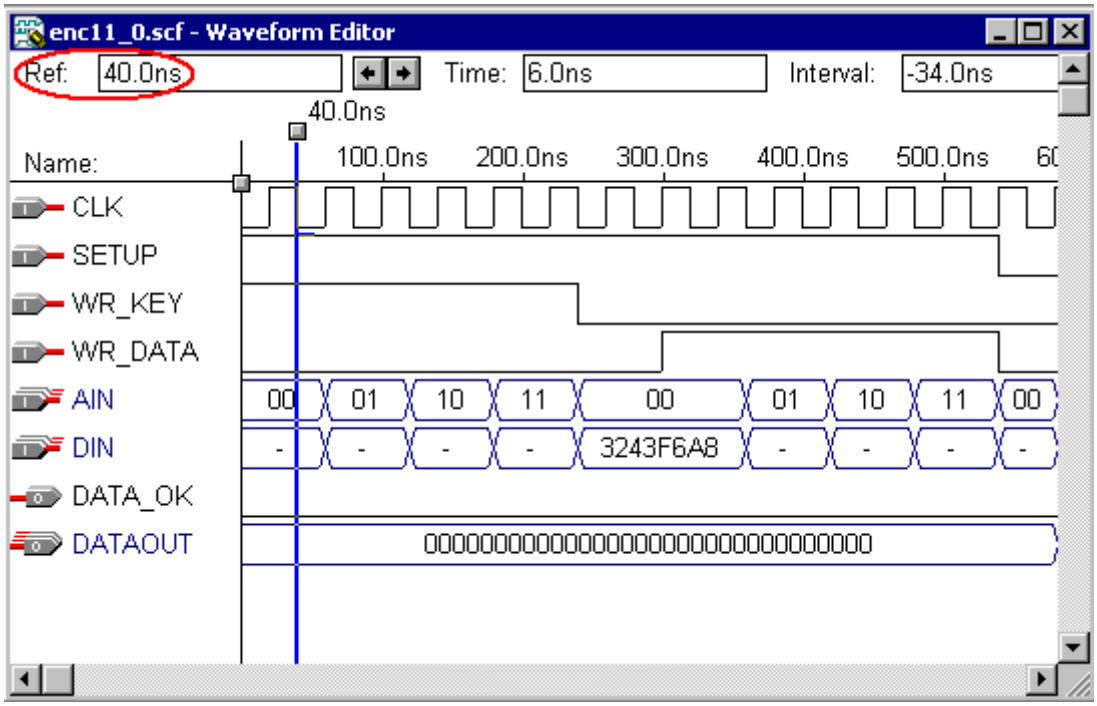


FIGURA B.9 - CLK mínimo do encriptador

Com o valor sugerido de CLK, leva-se 4,8us (10,4us menos 5,6us) para uma encriptação completa de 128 bits. Estes valores permitem uma saída de dados de aproximadamente 26,67 MB/s, conforme figuras B.10 e B.11.

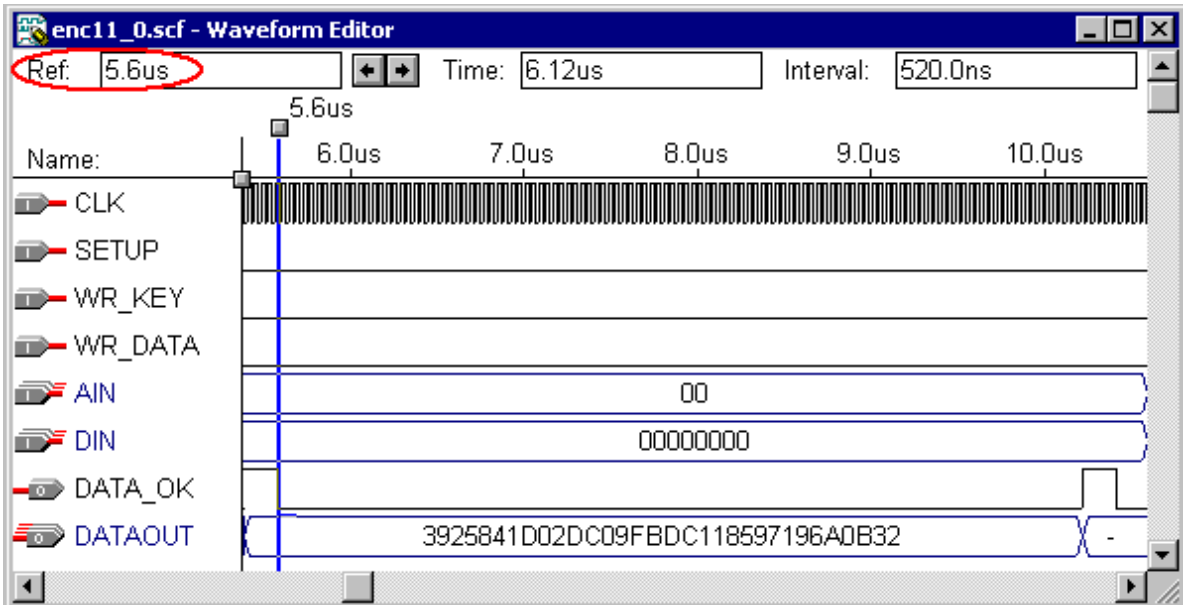


FIGURA B.10 - Primeiro dado pronto para ser lido

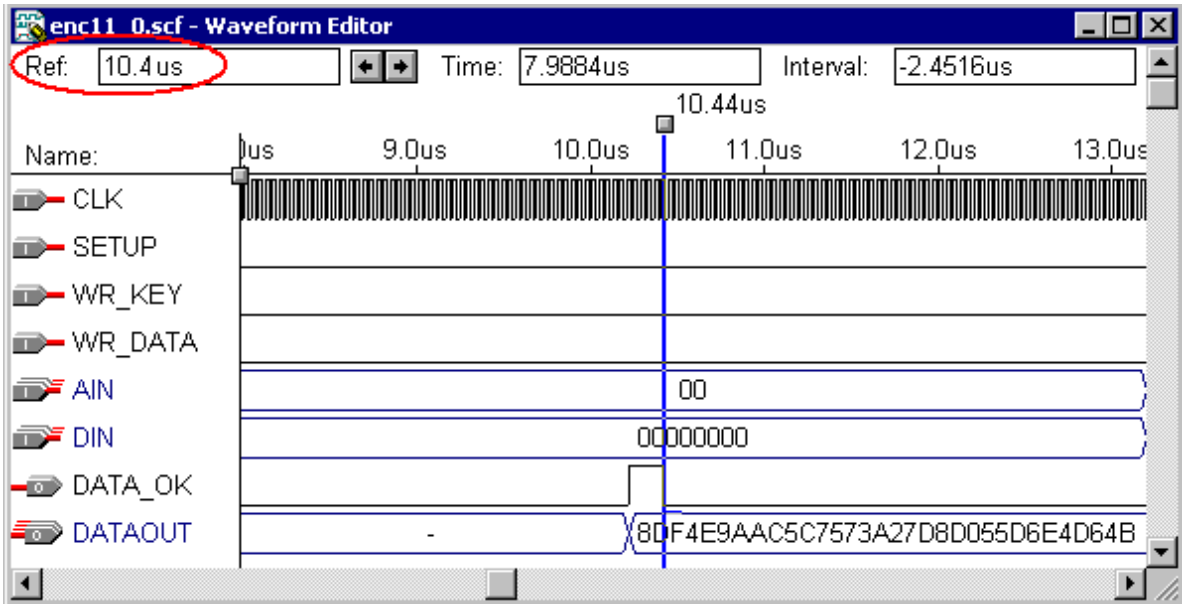


FIGURA B.11 - Segundo dado pronto para ser lido

### B.3.3 Utilização do componente

A implementação para o componente da Altera EPF10K50VFC484-1 apresenta a ocupação mostrada na tabela B.2.

TABELA B.2 – Ocupação de área do encriptador 32 bits

| Design | Componente | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|---|
| ENC32 | EPF10K50VFC484-1 | 38 | 129 | 0 | 16384 | 1980 |

### B.3.4. Pinagem

O dispositivo usa 167 pinos de I/O. Mostra-se, a seguir, a vista inferior do EPF10K50VFC484-1 (com seu diagrama de pinos) e a função de cada um.

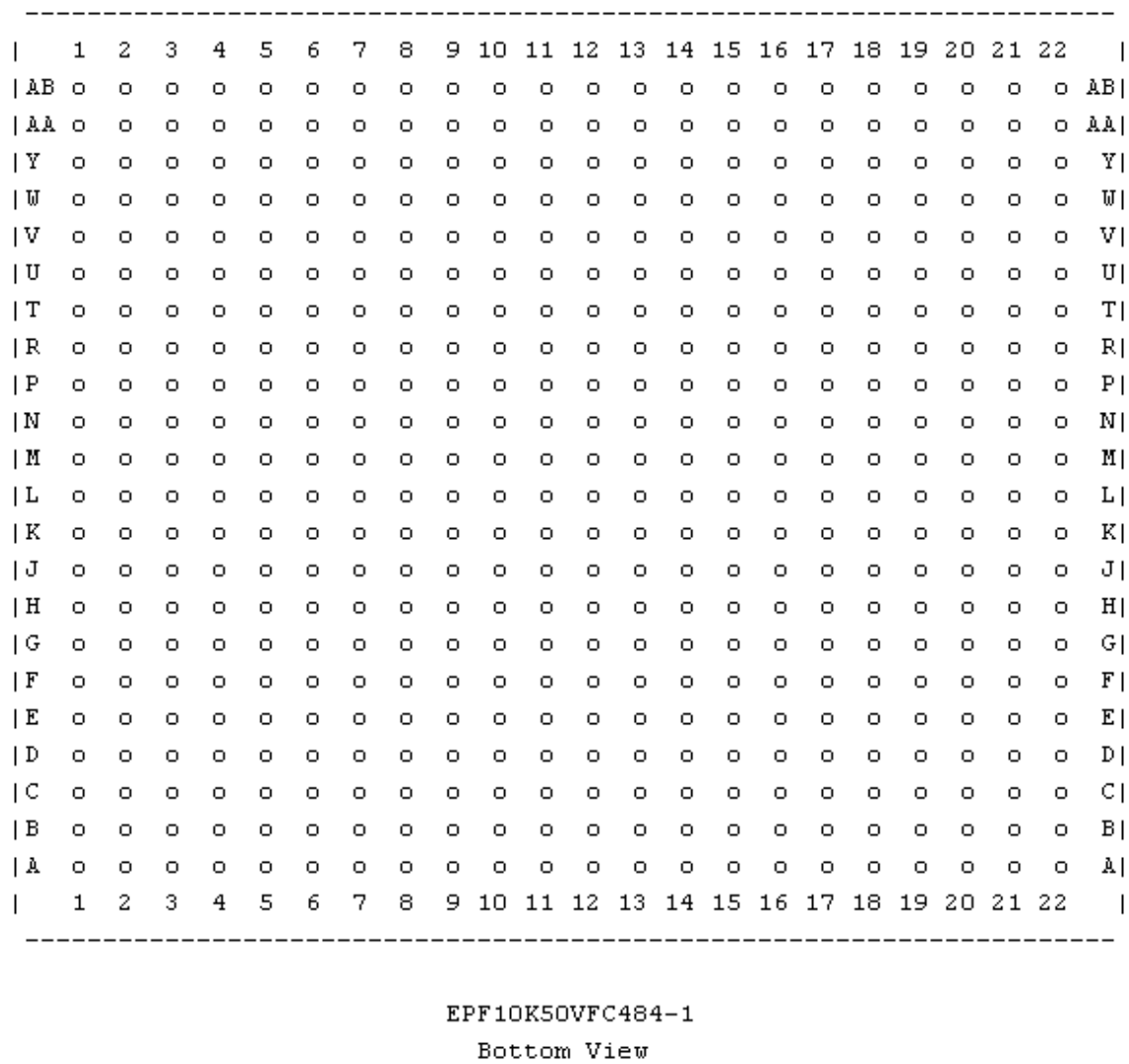


FIGURA B.12 - Vista inferior do EPF10K50VFC484-1

| | | | | |
|---|---|---|---|---|
| A1 GND | E10 RESERVED | J19 DATAOUT46 | P6 DATAOUT122 | V15 RESERVED |
| A2 N.C. | E11 DATAOUT112 | J20 GND | P7 DATAOUT125 | V16 RESERVED |
| A3 N.C. | E12 SETUP | J21 DATAOUT95 | P8 VCCINT | V17 RESERVED |
| A4 N.C. | E13 RESERVED | J22 DATAOUT63 | P9 GND | V18 DATAOUT93 |
| A5 N.C. | E14 DATAOUT89 | K1 RESERVED | P10 VCCINT | V19 #TRST |
| A6 VCCIO | E15 RESERVED | K2 N.C. | P11 DIN12 | V20 RESERVED |
| A7 N.C. | E16 DATAOUT39 | K3 GND | P12 VCCINT | V21 RESERVED |
| A8 GND | E17 RESERVED | K4 DATAOUT35 | P13 VCCIO | V22 VCCIO |
| A9 N.C. | E18 #TCK | K5 DIN25 | P14 GND | W1 RESERVED |
| A10 RESERVED | E19 ^nCEO | K6 DATAOUT105 | P15 VCCINT | W2 DATAOUT18 |
| A11 N.C. | E20 N.C. | K7 DATAOUT126 | P16 DATAOUT72 | W3 RESERVED |
| A12 N.C. | E21 GND | K8 DATAOUT54 | P17 DATAOUT74 | W4 RESERVED |
| A13 VCCIO | E22 DATAOUT86 | K9 VCCIO | P18 RESERVED | W5 DATAOUT41 |
| A14 N.C. | F1 DATAOUT25 | K10 GND | P19 DATAOUT98 | W6 RESERVED |
| A15 N.C. | F2 N.C. | K11 VCCIO | P20 N.C. | W7 RESERVED |
| A16 RESERVED | F3 DATAOUT108 | K12 GND | P21 RESERVED | W8 RESERVED |
| A17 RESERVED | F4 DIN28 | K13 GND | P22 N.C. | W9 RESERVED |
| A18 RESERVED | F5 #TDI | K14 VCCIO | R1 VCCIO | W10 DATAOUT40 |
| A19 RESERVED | F6 RESERVED | K15 DATAOUT55 | R2 DATAOUT30 | W11 GND |
| A20 N.C. | F7 RESERVED | K16 DATAOUT81 | R3 N.C. | W12 RESERVED |
| A21 N.C. | F8 RESERVED | K17 RESERVED | R4 DATAOUT118 | W13 RESERVED |
| A22 GND | F9 RESERVED | K18 DIN26 | R5 DATAOUT101 | W14 RESERVED |
| B1 GND | F10 RESERVED | K19 DIN1 | R6 DIN6 | W15 RESERVED |
| B2 GND | F11 DATAOUT1 | K20 VCCIO | R7 DATAOUT109 | W16 RESERVED |
| B3 N.C. | F12 RESERVED | K21 RESERVED | R8 GND | W17 RESERVED |
| B4 N.C. | F13 RESERVED | K22 DATAOUT45 | R9 VCCIO | W18 RESERVED |
| B5 VCCIO | F14 DATAOUT73 | L1 DIN23 | R10 RESERVED | W19 ^nSTATUS |
| B6 DATAOUT59 | F15 RESERVED | L2 N.C. | R11 RESERVED | W20 GND |
| B7 DATAOUT100 | F16 DATAOUT61 | L3 AIN0 | R12 DIN2 | W21 VCCINT |
| B8 RESERVED | F17 DATAOUT79 | L4 AIN1 | R13 RESERVED | W22 N.C. |
| B9 N.C. | F18 ^CONF_DONE | L5 DIN7 | R14 VCCINT | Y1 GND |
| B10 N.C. | F19 #TDO | L6 WR_DATA | R15 GND | Y2 GND |
| B11 RESERVED | F20 DATAOUT49 | L7 DIN17 | R16 RESERVED | Y3 GND |
| B12 N.C. | F21 DATAOUT47 | L8 DIN9 | R17 DIN21 | Y4 N.C. |
| B13 RESERVED | F22 DATAOUT123 | L9 VCCINT | R18 DIN5 | Y5 RESERVED |
| B14 RESERVED | G1 VCCIO | L10 VCCINT | R19 DIN14 | Y6 RESERVED |
| B15 RESERVED | G2 DATAOUT11 | L11 GND | R20 DIN13 | Y7 RESERVED |
| B16 N.C. | G3 GND | L12 GND | R21 DATAOUT26 | Y8 VCCINT |

| | | | | |
|---|---|---|---|---|
| B17 GND | G4 DATAOUT44 | L13 VCCINT | R22 GND | Y9 N.C. |
| B18 RESERVED | G5 DATAOUT60 | L14 VCCIO | T1 GND | Y10 RESERVED |
| B19 N.C. | G6 DATAOUT97 | L15 DIN10 | T2 DATAOUT27 | Y11 RESERVED |
| B20 N.C. | G7 DATAOUT43 | L16 DIN15 | T3 VCCIO | Y12 N.C. |
| B21 GND | G8 RESERVED | L17 DATAOUT103 | T4 DATAOUT20 | Y13 N.C. |
| B22 GND | G9 RESERVED | L18 DATAOUT53 | T5 DATAOUT9 | Y14 RESERVED |
| C1 N.C. | G10 RESERVED | L19 DATAOUT42 | T6 RESERVED | Y15 RESERVED |
| C2 GND | G11 RESERVED | L20 DATAOUT78 | T7 ^nCONFIG | Y16 N.C. |
| C3 DATAOUT62 | G12 DIN11 | L21 DATAOUT111 | T8 RESERVED | Y17 RESERVED |
| C4 RESERVED | G13 RESERVED | L22 DATAOUT119 | T9 RESERVED | Y18 RESERVED |
| C5 RESERVED | G14 DATAOUT19 | M1 DATAOUT120 | T10 RESERVED | Y19 N.C. |
| C6 N.C. | G15 VCCIO | M2 DATAOUT88 | T11 RESERVED | Y20 N.C. |
| C7 RESERVED | G16 RESERVED | M3 DATAOUT104 | T12 RESERVED | Y21 GND |
| C8 RESERVED | G17 RESERVED | M4 DATAOUT70 | T13 RESERVED | Y22 GND |
| C9 N.C. | G18 DATAOUT64 | M5 DATAOUT29 | T14 RESERVED | AA1 GND |
| C10 N.C. | G19 DATAOUT37 | M6 DATAOUT87 | T15 VCCIO | AA2 N.C. |
| C11 VCCINT | G20 RESERVED | M7 DATAOUT32 | T16 RESERVED | AA3 N.C. |
| C12 N.C. | G21 GND | M8 DIN31 | T17 DIN22 | AA4 N.C. |
| C13 N.C. | G22 DATAOUT113 | M9 VCCIO | T18 DATA_OK | AA5 RESERVED |
| C14 N.C. | H1 DATAOUT3 | M10 VCCINT | T19 DATAOUT23 | AA6 GND |
| C15 VCCINT | H2 GND | M11 GND | T20 N.C. | AA7 RESERVED |
| C16 N.C. | H3 DATAOUT69 | M12 GND | T21 N.C. | AA8 RESERVED |
| C17 N.C. | H4 DATAOUT115 | M13 VCCINT | T22 VCCIO | AA9 N.C. |
| C18 RESERVED | H5 DATAOUT34 | M14 VCCINT | U1 N.C. | AA10 RESERVED |
| C19 RESERVED | H6 DATAOUT124 | M15 DATAOUT65 | U2 DATAOUT2 | AA11 N.C. |
| C20 DIN16 | H7 RESERVED | M16 DIN24 | U3 N.C. | AA12 VCCINT |
| C21 GND | H8 GND | M17 DATAOUT96 | U4 ^MSEL0 | AA13 N.C. |
| C22 DATAOUT36 | H9 VCCIO | M18 DATAOUT102 | U5 RESERVED | AA14 RESERVED |
| D1 N.C. | H10 RESERVED | M19 DATAOUT110 | U6 RESERVED | AA15 N.C. |
| D2 N.C. | H11 DIN8 | M20 GND | U7 RESERVED | AA16 RESERVED |
| D3 N.C. | H12 DATAOUT94 | M21 DATAOUT48 | U8 RESERVED | AA17 RESERVED |
| D4 ^DATA0 | H13 RESERVED | M22 DATAOUT85 | U9 DATAOUT10 | AA18 DIN0 |
| D5 DATAOUT57 | H14 VCCINT | N1 N.C. | U10 RESERVED | AA19 DIN20 |
| D6 RESERVED | H15 GND | N2 DATAOUT14 | U11 DATAOUT58 | AA20 RESERVED |
| D7 RESERVED | H16 DATAOUT92 | N3 VCCIO | U12 RESERVED | AA21 N.C. |
| D8 DATAOUT121 | H17 DATAOUT68 | N4 DATAOUT15 | U13 WR_KEY | AA22 GND |
| D9 RESERVED | H18 DIN3 | N5 DATAOUT38 | U14 RESERVED | AB1 N.C. |
| D10 DATAOUT4 | H19 DATAOUT51 | N6 DATAOUT5 | U15 RESERVED | AB2 N.C. |
| D11 RESERVED | H20 VCCIO | N7 DATAOUT8 | U16 RESERVED | AB3 N.C. |
| D12 CLK | H21 DATAOUT7 | N8 DATAOUT71 | U17 DATAOUT90 | AB4 N.C. |
| D13 DIN19 | H22 DATAOUT107 | N9 VCCIO | U18 #TMS | AB5 N.C. |
| D14 DATAOUT17 | J1 DIN18 | N10 GND | U19 DATAOUT31 | AB6 RESERVED |
| D15 RESERVED | J2 N.C. | N11 VCCIO | U20 DIN29 | AB7 N.C. |
| D16 RESERVED | J3 DATAOUT12 | N12 VCCINT | U21 DIN30 | AB8 N.C. |
| D17 RESERVED | J4 DATAOUT33 | N13 GND | U22 RESERVED | AB9 N.C. |
| D18 RESERVED | J5 DATAOUT82 | N14 VCCIO | V1 DATAOUT21 | AB10 RESERVED |
| D19 DATAOUT77 | J6 DATAOUT75 | N15 DATAOUT0 | V2 DATAOUT117 | AB11 GND |
| D20 N.C. | J7 DIN27 | N16 DATAOUT13 | V3 GND | AB12 N.C. |
| D21 DATAOUT76 | J8 VCCINT | N17 DATAOUT24 | V4 ^MSEL1 | AB13 VCCIO |
| D22 DATAOUT52 | J9 GND | N18 DATAOUT16 | V5 VCCINT | AB14 RESERVED |
| E1 VCCIO | J10 VCCINT | N19 DATAOUT22 | V6 RESERVED | AB15 N.C. |
| E2 DATAOUT67 | J11 VCCIO | N20 VCCIO | V7 DATAOUT116 | AB16 GND |
| E3 DATAOUT84 | J12 VCCINT | N21 DATAOUT80 | V8 RESERVED | AB17 N.C. |
| E4 ^nCE | J13 VCCIO | N22 DATAOUT127 | V9 RESERVED | AB18 N.C. |
| E5 ^DCLK | J14 GND | P1 DATAOUT66 | V10 RESERVED | AB19 N.C. |
| E6 RESERVED | J15 VCCINT | P2 DATAOUT28 | V11 DIN4 | AB20 N.C. |
| E7 RESERVED | J16 DATAOUT99 | P3 DATAOUT6 | V12 RESERVED | AB21 N.C. |
| E8 DATAOUT56 | J17 DATAOUT91 | P4 DATAOUT106 | V13 RESERVED | AB22 N.C. |
| E9 RESERVED | J18 DATAOUT50 | P5 DATAOUT114 | V14 DATAOUT83 | |

FIGURA B.13 - Pinagem do ENC32

## *B.4 Decriptador*

### B.4.1 Diagrama de blocos

Há quatro processos realizados em paralelo na arquitetura dos dispositivos, como pode ser visto na figura abaixo.

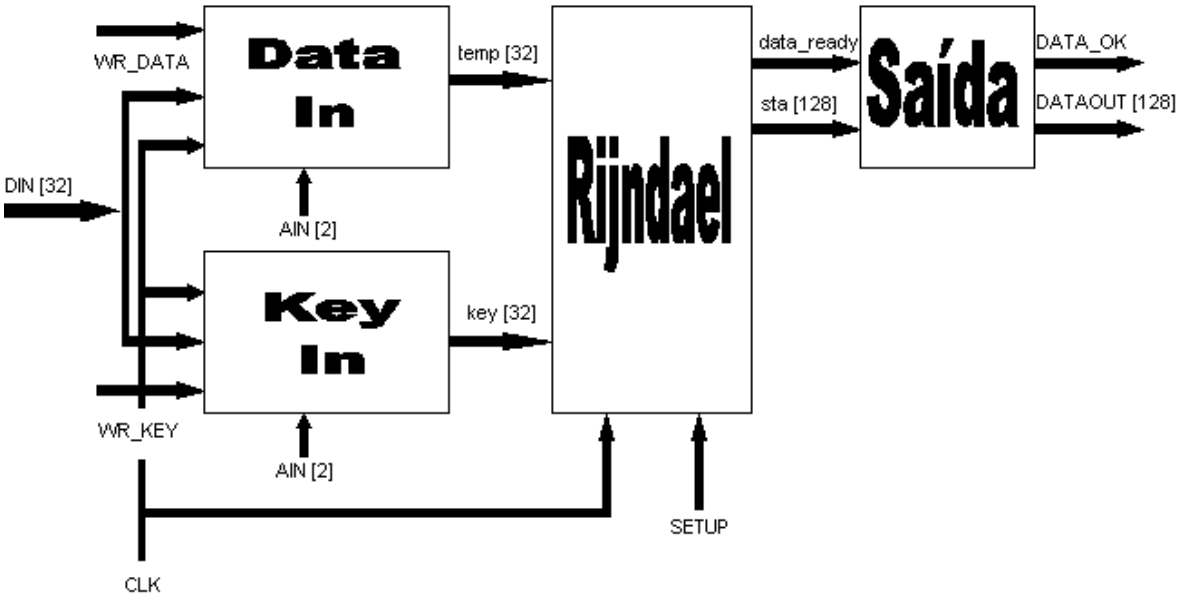


FIGURA B.14 - Diagrama de blocos do decriptador 128 bits

O processo Data_In é responsável pela recepção dos dados de entrada. Ele toma 32 bits por vez, na ordem estabelecida pelo processador, através do sinal AIN, e armazena-os em uma variável temporária de 128 bits. Este processo é ativado sempre que o sinal WR_DATA é acionado e permite que o processador escreva um novo dado durante a encriptação do dado anterior.

O processo Key_In é semelhante ao Data_in. Armazena a chave de entrada em uma variável fixa para a chave. Isto ocorre, pois a chave não é trocada a todo o momento e, assim sendo, fica armazenada no dispositivo até que deseje-se trocá-la. Toma 32 bits por vez, na ordem estabelecida pelo processador, através do sinal AIN. Este processo é ativado ao se acionar o sinal WR_KEY.

O processo Rijndael roda o algoritmo em 32 bits. Há duas formas de funcionamento. Se o sinal SETUP estiver acionado, ele ajusta valores das variáveis para o início do funcionamento. Se o sinal não for acionado, o algoritmo é executado continuamente. Nessa segunda opção, existem quatro estados, divididos em mais quatro estados cada, cada um responsável por processar uma palavra de 32 bits. O primeiro estado é executado apenas na primeira rodada de funcionamento. É responsável pela Add_Key_F inicial entre as variáveis temporária de dados e da chave inicial, e por armazenar o resultado na variável de trabalho. Além disso, ajusta valores corretos da rodada atual e alimenta a variável round_key com seu valor inicial. Na verdade, este processo é, por si só, toda a rodada 0.

O segundo estado é executado em todas as rodadas. Consiste em realizar as funções IByte_Sub (inversa do Byte_Sub) e IShift_Row (inversa do Shift_Row) sobre a variável de trabalho. O resultado da operação é armazenado na variável de trabalho. Este estado, ainda, executa a formação da round_key para a rodada seguinte e aciona, através do processo Saída, o sinal DATA_OK, para avisar ao processador que um novo dado pode ser enviado ao barramento, e que a saída pode ser lida.

O terceiro estado é executado em todas os rodadas também. Consiste em realizar a função Add Round Key. O quarto estado também é executado em todas as rodadas e consiste na realização da função Imix_Column (inversa da Mix_Column). Este resultado é válido, analisando-se o conjunto de ações de todo o processo de decriptação. Primeiramente, é realizado um Add_Key_F, que equivale ao primeiro da decriptação. A

seguir, não é realizada a IMix_Column, como se deveria, mas sim a ISub_Byte e IShif_Row. Desta forma, é executado o primeiro round da decriptação. A Add_Key_F do estado três (o quarto a ser executado) é, na verdade, o primeiro estágio de uma das rodadas. Na última rodada, na qual deveria ser realizado apenas a Add_Key_F, isto acontecerá sem a necessidade de um controle extra, dada a ordem das operações.

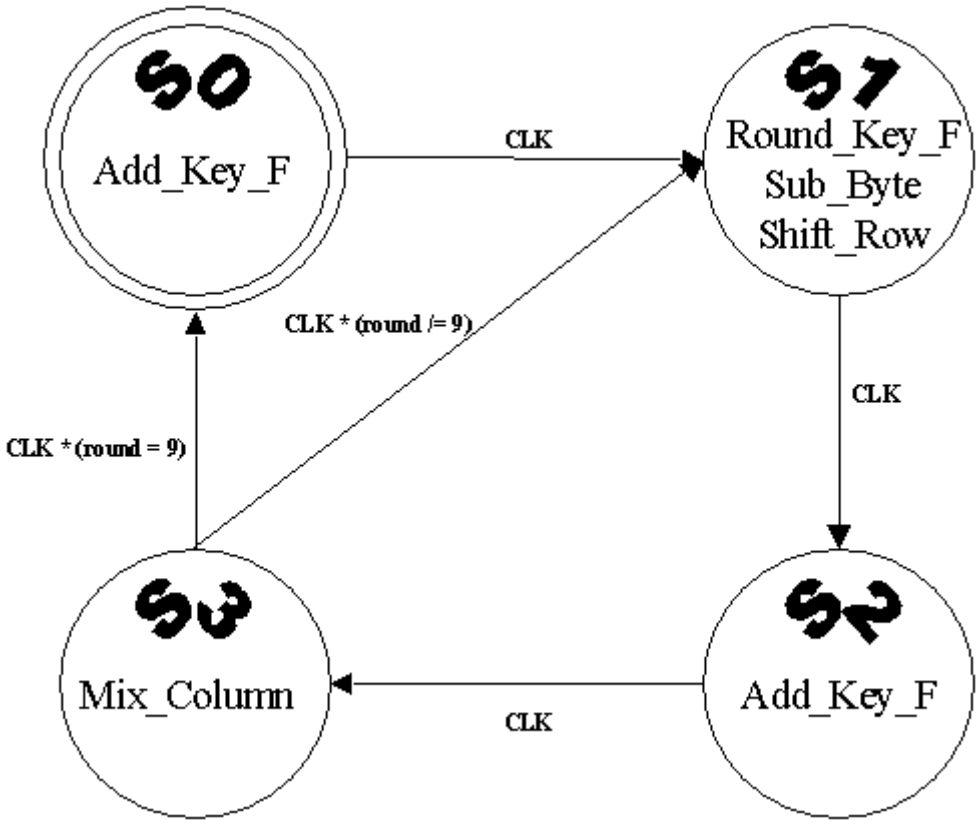


FIGURA B.15 - Máquina de estados do decriptador 32 bits

## B.4.2 Desempenho

A velocidade de processamento do dado depende do período de CLK, tendo em vista que há um número pré estabelecido de rodadas e que cada rodada leva um período de relógio. O melhor resultado obtido foi o uso de um período de 40 ns, mostrado na figura B.16.

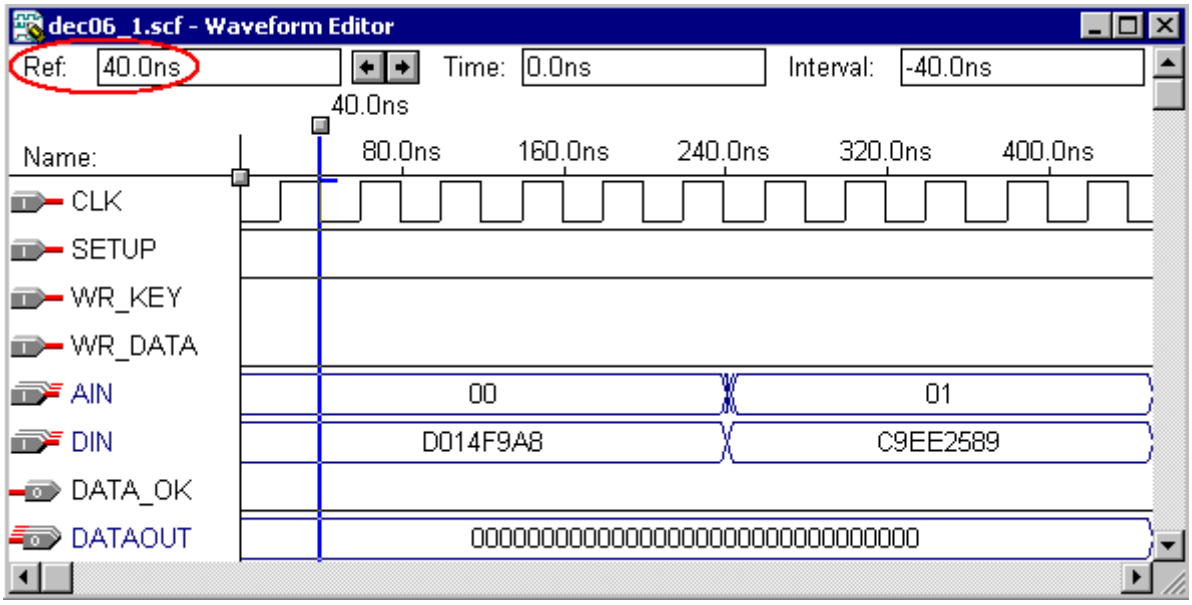


FIGURA B.16 - CLK mínimo do decriptador

Com o valor sugerido de CLK, leva-se 4,8us (nas figuras, 21,08us menos 16,28us) para um decriptação completa de 128 bits. Estes valores permitem uma saída de dados de aproximadamente 26,67 MB/s, conforme figuras B.17 e B.18.

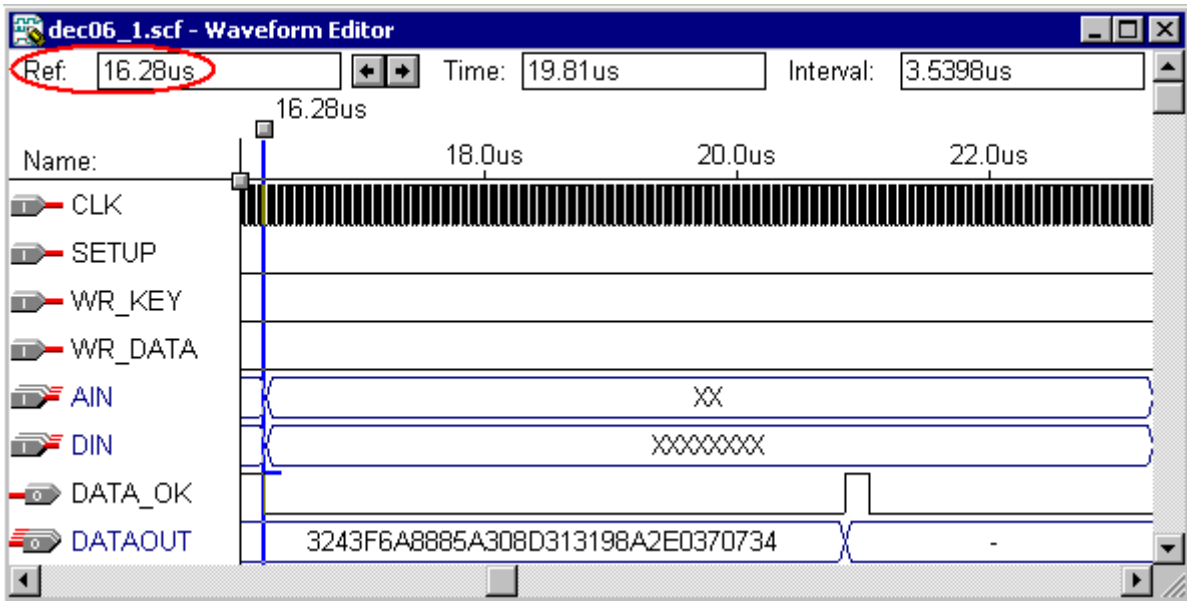


FIGURA B.17 - Primeiro dado pronto para ser lido

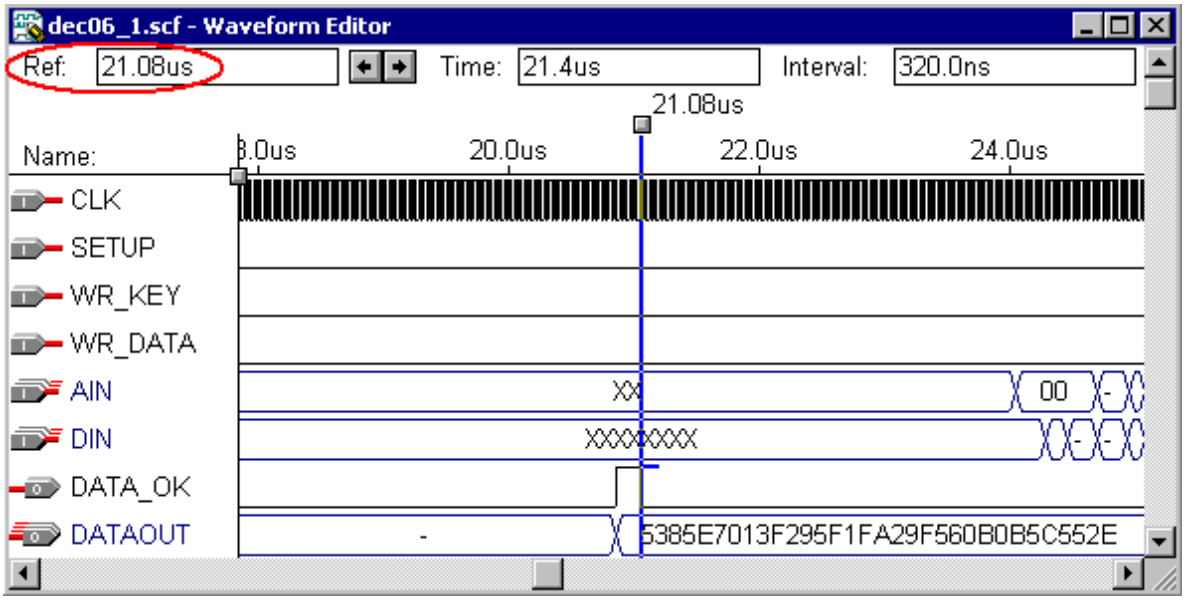


FIGURA B.18 - Segundo dado pronto para ser lido

### B.4.3 Utilização do componente

A implementação para o componente da Altera EPF10K50VFC484-1 é apresentada na tabela B.3.

TABELA B.3 – Ocupação de área do decriptador 32 bits

| Design | Componente | Input Pins | Output Pins | Bidir Pins | Memory Bits | LCs |
|---|---|---|---|---|---|---|
| DEC32 | EPF10K50VFC484-1 | 38 | 129 | 0 | 16384 | 2473 |

### B.4.4 Pinagem

O dispositivos usam 167 pinos de I/O. Mostra-se, a seguir, a vista inferior do EPF10K50VFC484-1 (com seu diagrama de pinos) e a função de cada um.

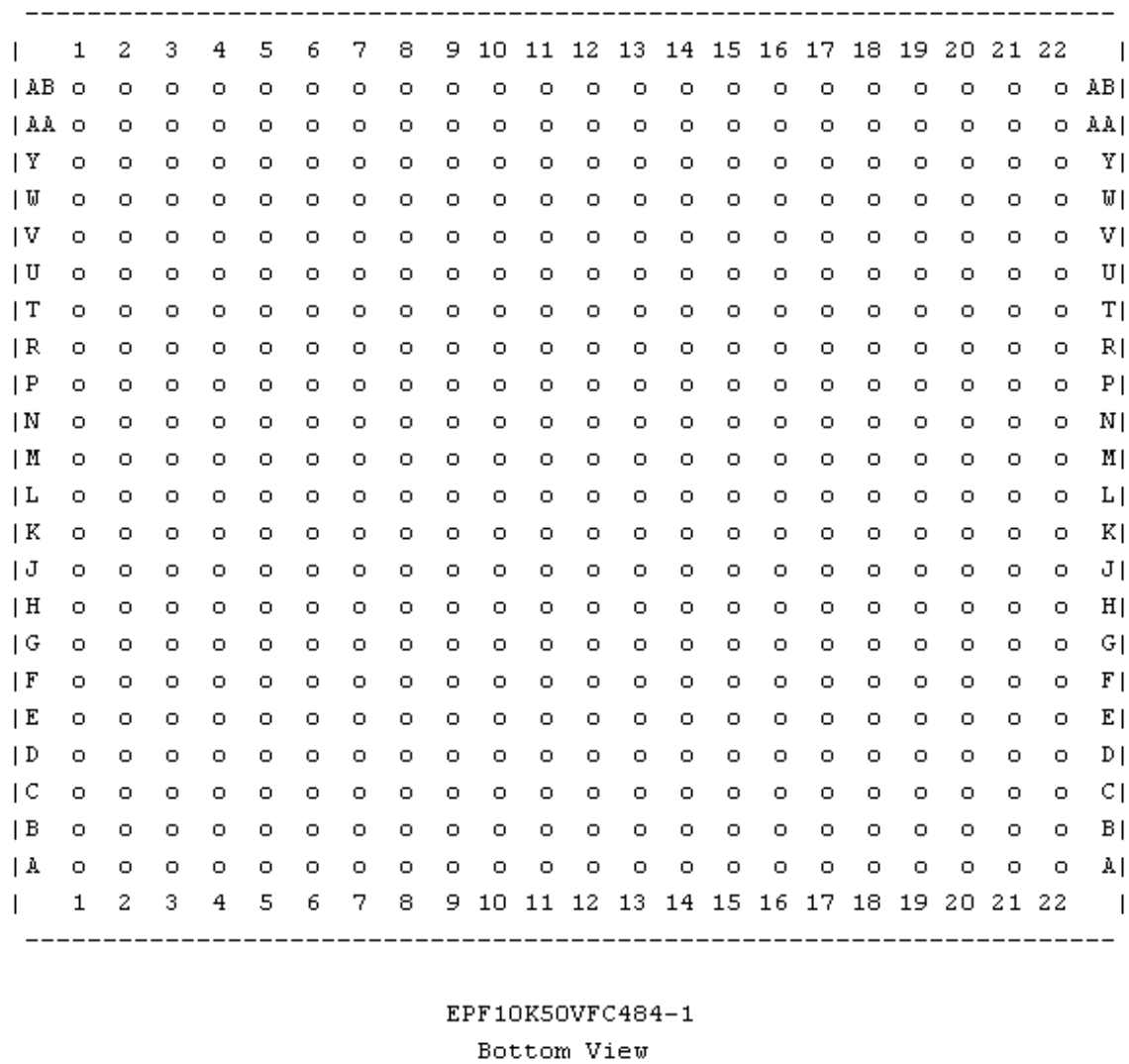


FIGURA B.19 - Vista inferior do Vista inferior do EPF10K50VFC484-1

| | | | | |
|---|---|---|---|---|
| A1 GND | E10 RESERVED | J19 DATAOUT19 | P6 DATAOUT121 | V15 DATAOUT72 |
| A2 N.C. | E11 DIN2 | J20 GND | P7 RESERVED | V16 DATAOUT21 |
| A3 N.C. | E12 SETUP | J21 DATAOUT62 | P8 VCCINT | V17 RESERVED |
| A4 N.C. | E13 RESERVED | J22 DATAOUT75 | P9 GND | V18 DATAOUT115 |
| A5 N.C. | E14 RESERVED | K1 DATAOUT54 | P10 VCCINT | V19 #TRST |
| A6 VCCIO | E15 RESERVED | K2 N.C. | P11 DIN29 | V20 RESERVED |
| A7 N.C. | E16 RESERVED | K3 GND | P12 VCCINT | V21 DATAOUT86 |
| A8 GND | E17 DIN20 | K4 DATAOUT22 | P13 VCCIO | V22 VCCIO |
| A9 N.C. | E18 #TCK | K5 DATAOUT45 | P14 GND | W1 DATAOUT70 |
| A10 DATAOUT91 | E19 ^nCEO | K6 DATAOUT30 | P15 VCCINT | W2 DATAOUT118 |
| A11 N.C. | E20 N.C. | K7 DATAOUT9 | P16 DATAOUT112 | W3 DIN27 |
| A12 N.C. | E21 GND | K8 DATAOUT7 | P17 DATAOUT80 | W4 RESERVED |
| A13 VCCIO | E22 RESERVED | K9 VCCIO | P18 DATAOUT73 | W5 DIN5 |
| A14 N.C. | F1 RESERVED | K10 GND | P19 DATAOUT97 | W6 RESERVED |
| A15 N.C. | F2 N.C. | K11 VCCIO | P20 N.C. | W7 RESERVED |
| A16 RESERVED | F3 RESERVED | K12 GND | P21 DATAOUT0 | W8 DIN7 |
| A17 RESERVED | F4 DATAOUT58 | K13 GND | P22 N.C. | W9 RESERVED |
| A18 RESERVED | F5 #TDI | K14 VCCIO | R1 VCCIO | W10 DATAOUT87 |
| A19 RESERVED | F6 DATAOUT110 | K15 RESERVED | R2 DATAOUT48 | W11 GND |
| A20 N.C. | F7 DIN30 | K16 DATAOUT35 | R3 N.C. | W12 RESERVED |
| A21 N.C. | F8 RESERVED | K17 DATAOUT38 | R4 RESERVED | W13 DIN16 |
| A22 GND | F9 DIN10 | K18 DATAOUT39 | R5 RESERVED | W14 DATAOUT4 |
| B1 GND | F10 RESERVED | K19 DATAOUT37 | R6 DATAOUT125 | W15 RESERVED |
| B2 GND | F11 RESERVED | K20 VCCIO | R7 RESERVED | W16 DATAOUT55 |
| B3 N.C. | F12 RESERVED | K21 DATAOUT26 | R8 GND | W17 DATAOUT29 |
| B4 N.C. | F13 RESERVED | K22 RESERVED | R9 VCCIO | W18 DATAOUT123 |
| B5 VCCIO | F14 RESERVED | L1 RESERVED | R10 RESERVED | W19 ^nSTATUS |
| B6 RESERVED | F15 RESERVED | L2 N.C. | R11 DATAOUT17 | W20 GND |
| B7 RESERVED | F16 RESERVED | L3 RESERVED | R12 DIN1 | W21 VCCINT |
| B8 RESERVED | F17 DATAOUT43 | L4 RESERVED | R13 DATAOUT108 | W22 N.C. |
| B9 N.C. | F18 ^CONF_DONE | L5 RESERVED | R14 VCCINT | Y1 GND |
| B10 N.C. | F19 #TDO | L6 RESERVED | R15 GND | Y2 GND |
| B11 RESERVED | F20 RESERVED | L7 DATAOUT10 | R16 DATAOUT114 | Y3 GND |
| B12 N.C. | F21 DATAOUT67 | L8 DATAOUT14 | R17 RESERVED | Y4 N.C. |
| B13 DATAOUT105 | F22 DATAOUT85 | L9 VCCINT | R18 RESERVED | Y5 RESERVED |
| B14 RESERVED | G1 VCCIO | L10 VCCINT | R19 RESERVED | Y6 RESERVED |
| B15 DATAOUT107 | G2 DATAOUT51 | L11 GND | R20 RESERVED | Y7 RESERVED |
| B16 N.C. | G3 GND | L12 GND | R21 DATAOUT113 | Y8 VCCINT |

B17 GND
B18 RESERVED
B19 N.C.
B20 N.C.
B21 GND
B22 GND
C1 N.C.
C2 GND
C3 RESERVED
C4 RESERVED
C5 RESERVED
C6 N.C.
C7 DIN25
C8 DATAOUT41
C9 N.C.
C10 N.C.
C11 VCCINT
C12 N.C.
C13 N.C.
C14 N.C.
C15 VCCINT
C16 N.C.
C17 N.C.
C18 DATAOUT27
C19 DATAOUT81
C20 DATAOUT93
C21 GND
C22 RESERVED
D1 N.C.
D2 N.C.
D3 N.C.
D4 ^DATA0
D5 DIN8
D6 AIN1
D7 RESERVED
D8 DIN17
D9 RESERVED
D10 RESERVED
D11 DIN26
D12 CLK
D13 DATAOUT2
D14 RESERVED
D15 RESERVED
D16 DATAOUT61
D17 RESERVED
D18 RESERVED
D19 RESERVED
D20 N.C.
D21 RESERVED
D22 DATAOUT104
E1 VCCIO
E2 DATAOUT101
E3 RESERVED
E4 ^nCE
E5 ^DCLK
E6 DIN9
E7 RESERVED
E8 DATAOUT25
E9 RESERVED

G4 RESERVED
G5 RESERVED
G6 RESERVED
G7 RESERVED
G8 RESERVED
G9 DIN18
G10 DATAOUT127
G11 RESERVED
G12 RESERVED
G13 RESERVED
G14 RESERVED
G15 VCCIO
G16 RESERVED
G17 RESERVED
G18 DIN21
G19 DATAOUT84
G20 DATAOUT18
G21 GND
G22 RESERVED
H1 DATAOUT1
H2 GND
H3 DATAOUT77
H4 DATAOUT69
H5 DIN13
H6 RESERVED
H7 RESERVED
H8 GND
H9 VCCIO
H10 DIN31
H11 DIN0
H12 DIN12
H13 RESERVED
H14 VCCINT
H15 GND
H16 DATAOUT103
H17 DATAOUT99
H18 DATAOUT40
H19 DATAOUT109
H20 VCCIO
H21 DATAOUT3
H22 DATAOUT92
J1 DATAOUT46
J2 N.C.
J3 DATAOUT49
J4 DATAOUT33
J5 DATAOUT8
J6 DATAOUT76
J7 DATAOUT53
J8 VCCINT
J9 GND
J10 VCCINT
J11 VCCIO
J12 VCCINT
J13 VCCIO
J14 GND
J15 VCCINT
J16 DATAOUT13
J17 DATAOUT117
J18 DATAOUT83

L13 VCCINT
L14 VCCIO
L15 DATAOUT6
L16 RESERVED
L17 RESERVED
L18 DATAOUT34
L19 DATAOUT11
L20 DATAOUT88
L21 RESERVED
L22 RESERVED
M1 RESERVED
M2 DIN15
M3 DATAOUT36
M4 DATAOUT15
M5 RESERVED
M6 DATAOUT60
M7 DATAOUT95
M8 RESERVED
M9 VCCIO
M10 VCCINT
M11 GND
M12 GND
M13 VCCINT
M14 VCCINT
M15 RESERVED
M16 DATAOUT79
M17 DATAOUT31
M18 DATAOUT90
M19 DATAOUT126
M20 GND
M21 DATAOUT23
M22 DATAOUT52
N1 N.C.
N2 DATAOUT56
N3 VCCIO
N4 DATAOUT32
N5 DATAOUT106
N6 DATAOUT16
N7 RESERVED
N8 DATAOUT89
N9 VCCIO
N10 GND
N11 VCCIO
N12 VCCINT
N13 GND
N14 VCCIO
N15 RESERVED
N16 DATAOUT82
N17 DATAOUT64
N18 DATAOUT98
N19 DATAOUT66
N20 VCCIO
N21 DATAOUT44
N22 DATAOUT24
P1 DATAOUT50
P2 RESERVED
P3 DATAOUT74
P4 DATAOUT65
P5 DATAOUT42

R22 GND
T1 GND
T2 DATAOUT96
T3 VCCIO
T4 DATAOUT122
T5 DATAOUT124
T6 RESERVED
T7 ^nCONFIG
T8 DATAOUT102
T9 RESERVED
T10 RESERVED
T11 DATAOUT119
T12 DATAOUT63
T13 RESERVED
T14 DIN28
T15 VCCIO
T16 RESERVED
T17 RESERVED
T18 DATAOUT111
T19 DATAOUT94
T20 N.C.
T21 N.C.
T22 VCCIO
U1 N.C.
U2 DATAOUT71
U3 N.C.
U4 ^MSEL0
U5 DATAOUT78
U6 DATAOUT57
U7 WR_KEY
U8 DIN22
U9 DIN6
U10 DIN14
U11 RESERVED
U12 DATAOUT20
U13 RESERVED
U14 RESERVED
U15 RESERVED
U16 DATAOUT12
U17 DATAOUT47
U18 #TMS
U19 DATAOUT59
U20 DATAOUT120
U21 RESERVED
U22 DATAOUT28
V1 DATAOUT100
V2 RESERVED
V3 GND
V4 ^MSEL1
V5 VCCINT
V6 RESERVED
V7 RESERVED
V8 RESERVED
V9 RESERVED
V10 DIN19
V11 DIN24
V12 RESERVED
V13 RESERVED
V14 RESERVED

Y9 N.C.
Y10 DIN23
Y11 RESERVED
Y12 N.C.
Y13 N.C.
Y14 DATAOUT5
Y15 RESERVED
Y16 N.C.
Y17 RESERVED
Y18 RESERVED
Y19 N.C.
Y20 N.C.
Y21 GND
Y22 GND
AA1 GND
AA2 N.C.
AA3 N.C.
AA4 N.C.
AA5 RESERVED
AA6 GND
AA7 RESERVED
AA8 DIN11
AA9 N.C.
AA10 WR_DATA
AA11 N.C.
AA12 VCCINT
AA13 N.C.
AA14 RESERVED
AA15 N.C.
AA16 RESERVED
AA17 DATAOUT116
AA18 DATAOUT68
AA19 DIN4
AA20 RESERVED
AA21 N.C.
AA22 GND
AB1 N.C.
AB2 N.C.
AB3 N.C.
AB4 N.C.
AB5 N.C.
AB6 DIN3
AB7 N.C.
AB8 N.C.
AB9 N.C.
AB10 AIN0
AB11 GND
AB12 N.C.
AB13 VCCIO
AB14 DATA_OK
AB15 N.C.
AB16 GND
AB17 N.C.
AB18 N.C.
AB19 N.C.
AB20 N.C.
AB21 N.C.
AB22 N.C.

FIGURA B.20 - Pinagem do DEC32

# Anexo 3 Ambiente de Desenvolvimento

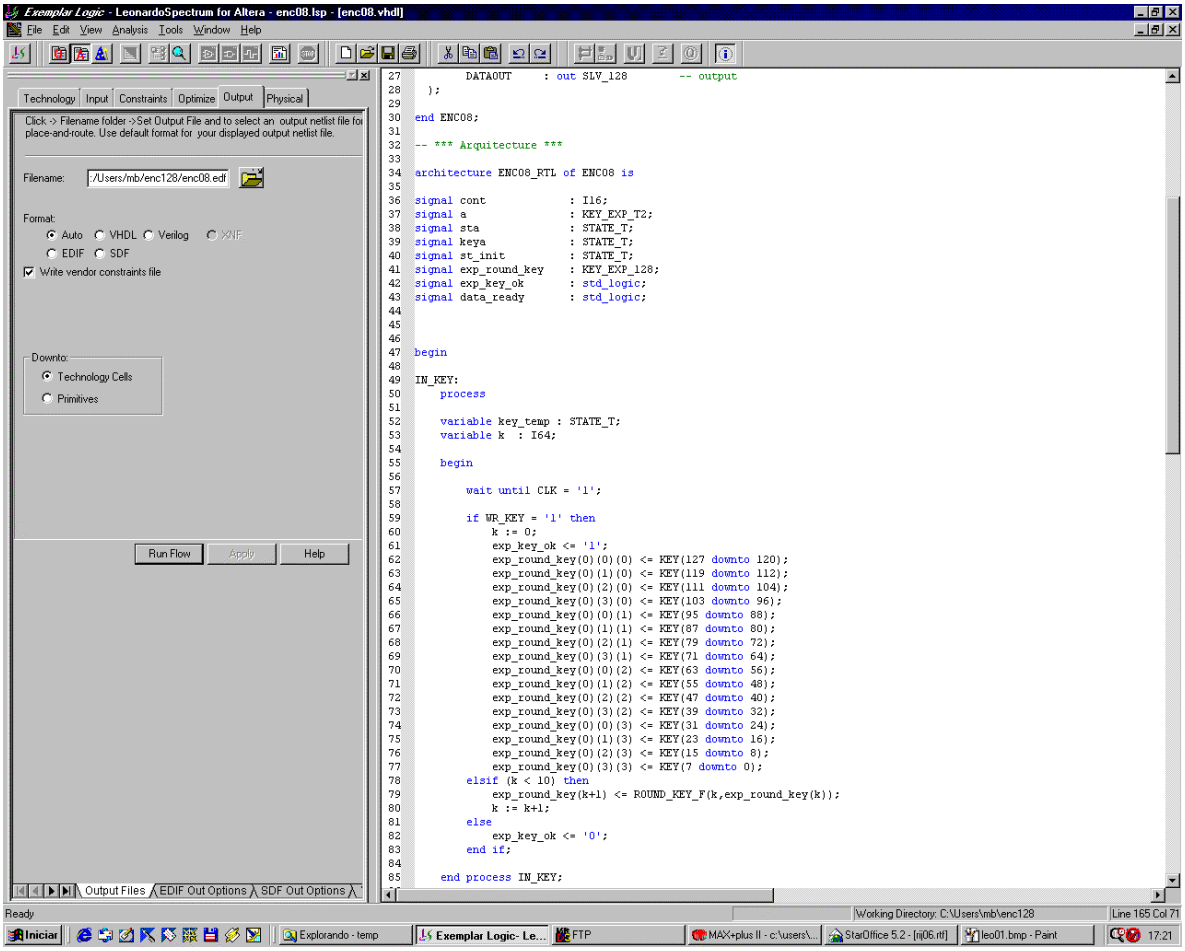


FIGURA C.1 - Codificação VHDL no LeonardoSpectrum

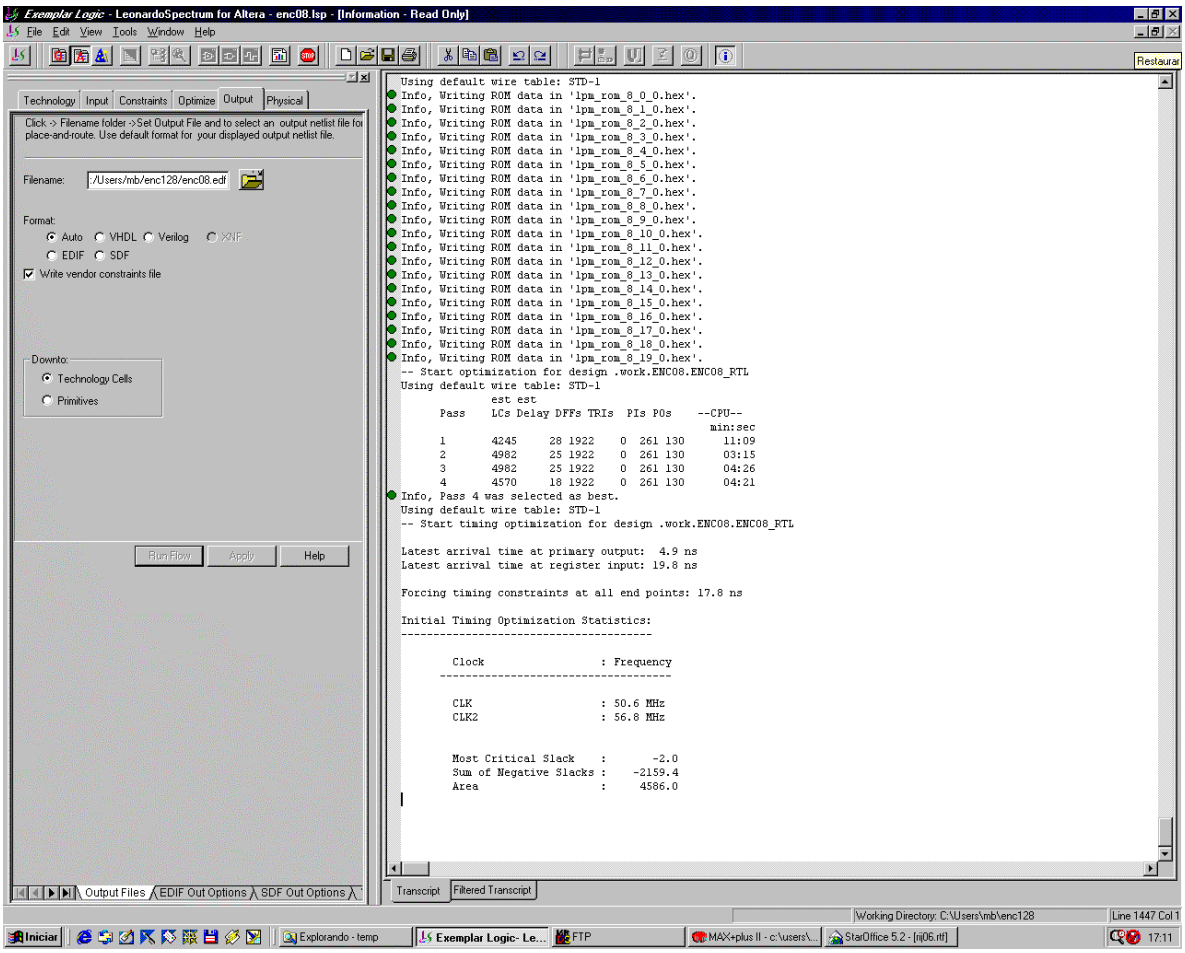


FIGURA C.2 - Verificação VHDL no LeonardoSpectrum

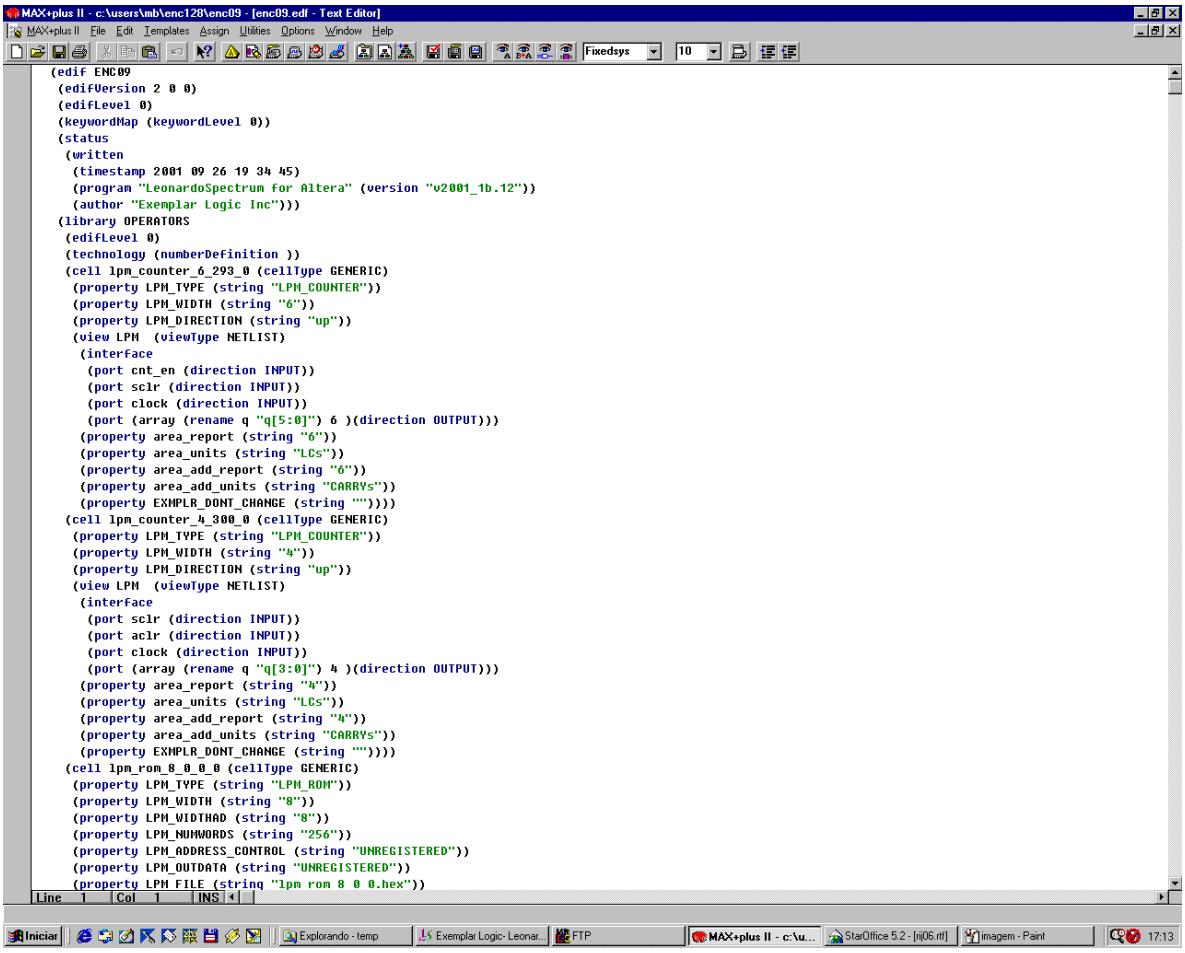


FIGURA C.3 - Importação de arquivo EDIF no Max+plus II

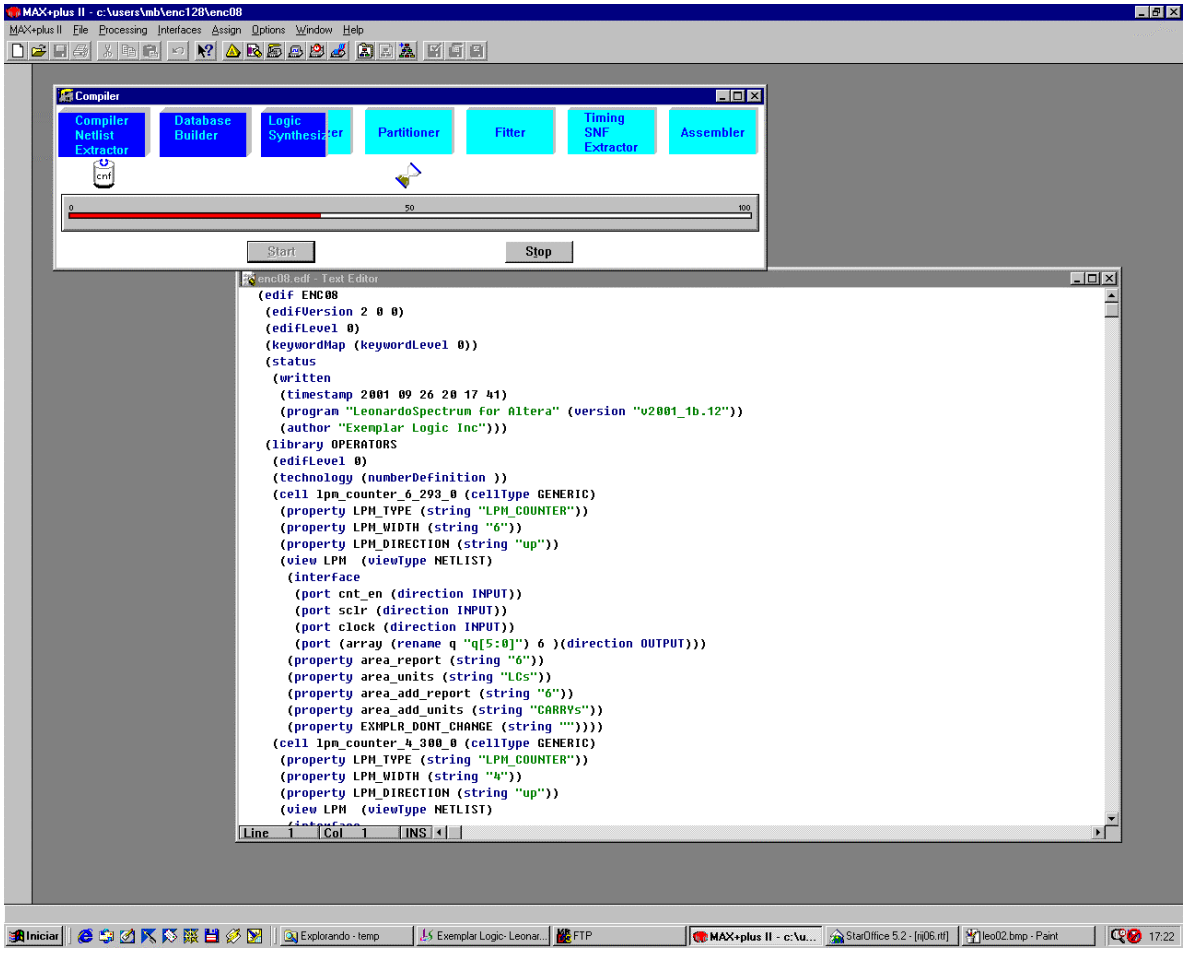


FIGURA C.4 - Compilação no Max+plus II

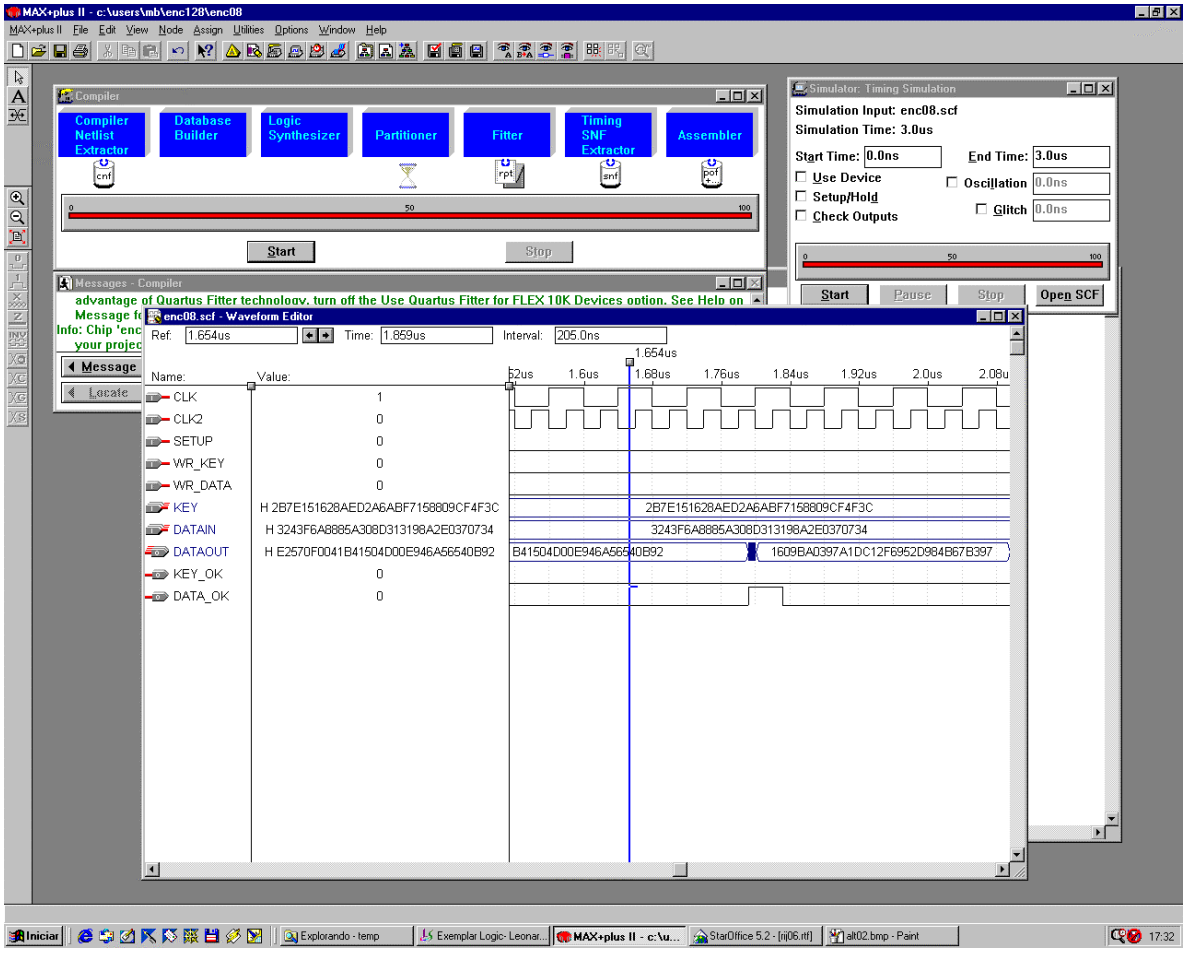


FIGURA C.5 - Simulação e verificação no Max+plus II

# Glossário

| | |
|---|---|
| AES | Advanced Encryption Standard. Padrão de Criptografia Avançada adotado pelo governo dos Estados Unidos da América. |
| Affine Transformation | Transformação Afim. |
| Array | Conjunto enumerável de entidades de mesmo tipo. |
| Bit | Dígito binário com valor 0 ou 1. |
| Bloco | Sequência de bits ou conjunto de bytes. |
| Byte | Conjunto de oito (8) bits, que pode ser tratado como uma única entidade, ou como um *array* de bits. |
| Cipher | Conjunto de transformações que converte plaintext em ciphertext utilizando uma chave. |
| Cipher Key | Chave secreta utilizada pelo mecanismo de Key Expansion para gerar as Round Keys utilizadas para criptografar os passos intermediários da cifragem. |
| Ciphertext | Bloco de saída do Cipher ou entrada no Inverse Cipher. |
| Criptografar | Tornar incompreensível, com observância de normas especiais consignadas numa cifra ou código, o texto de (uma mensagem escrita com clareza). [FER 9?] |
| Criptografia | Arte de escrever em cifra ou em código. Conjunto de técnicas que permitem criptografar escritas. |
| Hash | Algoritmo criptográfico de mão única, no qual a operação não pode, teoricamente, ser revertida. Ou ainda, resultado de uma função hash. |
| Inverse Cipher | Conjunto de transformações que converte ciphertext em plaintext. |
| Internet Banking | Acesso bancário via Internet para movimentações de conta e outros serviços. |
| Key Expansion | Rotina utilizada para gerar uma série de Round Keys a partir de uma Cipher Key. |
| NIST | National Institute of Standards and Technology |
| NSA | National Security Agency |
| Palavra | Conjunto de 32 bits, ou array de 4 bytes. |
| Plaintext | Bloco de saída do Inverse Cipher ou entrada no Cipher. |
| Rijndael | Algoritmo especificado no padrão AES norte americano. |
| Round | Rodadas do algoritmo em questão. |
| Round Key | Chaves intermediárias geradas pelas rotinas de Key Expansion a partir da Cipher Key. |

| | |
|---|---|
| Royalties | Licença de utilização de um algoritmo, metodologia, esquema, entre outros. |
| State | Resultados intermediários do processo de cifragem dos dados. |
| S-box | Tabela não linear de substituição de bytes utilizada em várias partes do processo. |

# Bibliografia


[ADA 97] ADAMS, C. **The CAST-128 Encryption Algorithm**. 1997. RFC 2144. Disponível em:<http://www.ietf.org/rfc/rfc2144.txt>.Acesso em: jul. 2001.

[ADA 99] ADAMS, C.; GILCHRIST, J. **The CAST-256 Encryption Algorithm**. 1999. RFC 2612. Disponível em: <http://www.ietf.org/rfc/rfc2612.txt>. Acesso em: jul. 2001.

[ALT 96] ALTERA. **Max+plus II VHDL**. San Jose, 1996.

[ALT 99] ALTERA. **Altera Acquires IP provider Hammercores, Inc**. 1999. Disponível em: <http://www.altera.com/corporate/press_box/releases/pr-hammercores.html>. Acesso em: out. 2001.

[ALT 2000] ALTERA. **FLEX 10K Embedded Programmable Logic Family Data Sheet**. 2000. Disponível em: <http://www.altera.com/literature/ds/dsf10k.pdf>. Acesso em: set. 2001.

[ALT 2001a] ALTERA. **High-Speed AES Encryption Cores**. 2001. Disponível em: <http://www.altera.com/products/ip/ampp/dcrypt/m-dcr-aes_enc.html>. Acesso em: out. 2001.

[ALT 2001b] ALTERA. **High-Speed Rijndael Encryption/Decryption Processors**. 2001. Disponível em: <http://www.altera.com/literature/wp/wp_hcores_rijnfast.pdf>. Acesso em: set. 2001.

[ALT 2001c] ALTERA. **Leonardo Spectrum Altera**. 2001. Disponível em: <http://www.altera.com/products/software/oem/oem-ls.html>. Acesso em: nov. 2001.

[ALT 2001d] ALTERA. **Max+plus II Software Overview**. 2001. Disponível em: <http://www.altera.com/products/software/maxplus2/mp2-index.html>. Acesso em: nov. 2001.

[AND 98] ANDERSON, Ross; BIHAM, Eli; KNUDSEN, Lars. **Serpent - A Candidate Block Cipher for the Advanced Encryption Standard**. 1998. Disponível em: <http://www.cl.cam.ac.uk/~rja14/serpent.html>. Acesso em: jul. 2001.

[ASH 96] ASHENDEN, Peter. **The Designer's Guide to VHDL**. San Francisco: Morgan Kaufmann, 1996.

[ASH 98] ASHENDEN, Peter. **The Student's Guide to VHDL**. San Francisco: Morgan Kaufmann, 1998.

[BAL 2001] BALTIMORE. **SureWare e-security hardware**. 2001. Disponível em:

<http://www.baltimore.com/products/sureware/keyper.html>. Acesso em: set. 2001.

[BCB 2001] BCB. **Sistema de Pagamentos Brasileiro**. 2001. Disponível em: <http://www.bcb.gov.br/mPag.asp?Perfil=1&cod=194&codP=53>. Acesso em: nov. 2001.

[BRA 2000] BRACCO, Tere. **Tales from the Crypto**. 2000. Disponível em: <http://www.nwfusion.com/reviews/2000/0522rev3.html>. Acesso em: out. 2001.

[BRO 2000] BROWN, Stephen. **Fundamentals of digital logic with VHDL design**. Boston: McGraw-Hill, 2000.

[CAR 90] CARLSON, Steve. **Introduction to HDL-Based design using VHDL**. Mountain View: Synopsys, 1990.

[COM 95] COMER, Douglas E. **Internetworking with TCP/IP**. Upper Saddle River: Prentice Hall, 1995. v.1.

[COU 2001] COUNTERPANE. **Twofish:** A New Block Cipher. 2001. Disponível em: <http://www.counterpane.com/twofish.html>. Acesso em: jun. 2001.

[CRO 99] CROLL, Alistair. **SSL Crunch Time**. 1999. Disponível em: <http://www.rainbow.com/cryptoswift/InfoSecurity/239202.html>. Acesso em: set. 2001.

[DAE 99] DAEMEN, Joan; RIJMEN, Pawel. **AES Proposal:** Rijndael. 1999. Disponível em: <http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip>. Acesso em: jul. 2001.

[DAE 2001] DAEMEN, Joan; RIJMEN, Pawel. **The block cipher Rijndael**. 2001. Disponível em: <http://www.esat.kuleuven.ac.be/~rijmen/rijndael/index.html>. Acesso em: jul. 2001.

[DAI 2000] DAI, Wei. **Speed Comparation of Popular Crypto Algorithms**. 2000. Disponível em: <http://www.eskimo.com/~weidai/benchmarks.html>. Acesso em: out. 2001.

[DIE 99] DIERKS, T; ALLEN, C. **The TLS Protocol Version 1.0**. 1999. Disponível em: <http://www.ietf.org/rfc/rfc2246.txt>. Acesso em: jun 2000.

[DIF 76] DIFFIE, W.; HELLMAN, M.E.. New Directions in Cryptography. **IEEE Transactions on Information Theory**, New York, v.IT-22, Nov. 1976.

[EFF 98] EFF. **EFF DES Cracker Project**. 1998. Disponível em: <http://www.eff.org/descracker.html>. Acesso em: out. 2001.

[EMA 2001] EMARKETER. **Q1 Sales Dip for US eCommerce**. 2001. Disponível em: <http://www.emarketer.com/estatnews/estats/ecommerce_b2c/20010522_census_retail.html>. Acesso em: set. 2001.

[EXE 2001] EXEMPLAR. **LeonardoSpectrum**. 2001. Disponível em: <http://www.exemplar.com/products/leonardospectrum.html>. Acesso em: nov. 2001.

[FAR 2001] FARROW, Rik. **Secure Sockets Layer Is Not A Magic Bullet**. 2001. Disponível em: <http://www.networkmagazine.com/article/NMG20001219S0006>. Acesso em: set. 2001.

[FER 9?] FERREIRA, Aurélio Buarque de Holanda. **Novo Dicionário da Língua Portuguesa**. Rio de Janeiro: Nova Fronteira. [199?].

[FIE 99] FIELDING, R. et al. **Hypertext Transfer Protocol - HTTP/1.1**. 1999. Disponível em: <http://www.faqs.org/rfcs/rfc2616.html>. Acesso em: ago. 2001.

[FIS 2000] FISCHER, Viktor. **Realization of the Round 2 AES Candidates using Altera FPGA**. 2000. Disponível em: <http://csrc.nist.gov/encryption/aes/round2/conf3/papers/24-vfischer.pdf>. Acesso em: ago. 2001.

[FRE 96] FREIER, Alan; KARLTON, Philip; KOCHER, Paul C. **The SSL Protocol Version 3.0**. 1996. Disponível em: <http://home.netscape.com/eng/ssl3/draft302.txt>. Acesso em: set. 2001.

[GAJ 2000a] GAJ, Kris; CHODOWIEC, Vincent. **Comparison of the hardware performance of the AES candidates using reconfigurable hardware**. 2000. Disponível em: <http://csrc.nist.gov/encryption/aes/round2/conf3/papers/22-kgaj.pdf>. Acesso em: ago. 2001.

[GAJ 2000b] GAJ, Kris; CHODOWIEC, Pawel. **Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays**. 2000. Disponível em: <http://bass.gmu.edu/crypto/AES_non_feedback.pdf>. Acesso em: out. 2001.

[GAR 2001] GARTNER. **Gartner, Inc.** 2001. Disponível em: <http://www.gartner.com>. Acesso em: set. 2001.

[GLA 99] GLADMAN, Brian. **AES Algorithm Efficiency**. 1999. Disponível em: <http://fp.gladman.plus.com/cryptography_technology/aes>. Acesso em: ago. 2001.

[GMU 2001] GMU. **Hardware IP Cores of Advanced Encryption Standard AES-Rijndael**. 2001. Disponível em: <http://ece.gmu.edu/crypto/rijndael.htm>. Acesso em: out. 2001.

[IBM 2000] IBM. **The MARS cipher - IBM submission to AES**. 2000. Disponível em: <http://www.research.ibm.com/security/mars.html>. Acesso em: jul. 2001.

[IBO 2001] IBOPE. **IBOPE eRatings**. 2001. Disponível em: <http://www.ibope.com.br/eratings/index.htm>. Acesso em: set. 2001.

[IDG 2001a] IDG. **IDG Now!** 2001. Resultados do IBOPE eRatings. Disponível em: <http://idgnow.terra.com.br/idgnow/internet/2001/07/0031>. Acesso em: set. 2001.

[IDG 2001b] IDG. **IDG Now!**. 2001. Previsões do Yankee Group para o Brasil. Disponível em: <http://idgnow.terra.com.br/idgnow/internet/2001/07/0028>. Acesso em: set. 2001.

[IDG 2001c] IDG. **IDG Now!**. 2001. Estudo feito no primeiro trimestre de 2001 pela Nielsen//NetRatings. Disponível em: <http://idgnow.terra.com.br/idgnow/internet/2001/06/0035>. Acesso em: set. 2001.

[IDG 2001d] IDG. **IDG Now!**. 2001. Informações prestadas pela Receita Federal. Disponível em: <http://idgnow.terra.com.br/idgnow/internet/2001/06/0025>. Acesso em: set. 2001.

[IDG 2001e] IDG. **IDG Now!**. 2001. Pesquisas do Gartner Group sobre segurança online. Disponível em: <http://idgnow.terra.com.br/idgnow/internet/2001/08/0023>. Acesso em: set. 2001.

[IET 2001a] IETF. **IP Security Protocol**. 2001. Disponível em: <http://www.ietf.org/html.charters/ipsec-charter.html>. Acesso em: jul. 2001.

[IET 2001b] IETF. **Secure Shell**. 2001. Disponível em: <http://www.ietf.org/html.charters/secsh-charter.html>. Acesso em: jul. 2001.

[IET 2001c] IETF. **Transport Layer Security**. 2001. Disponível em: <http://www.ietf.org/html.charters/tls-charter.html>. Acesso em: jul. 2001.

[IPS 2001] IPSOS-REID. **Online Security Concerns Have Chilling Effect on Consumers Even If Actual Fraud Rare**. 2001. Disponível em: <http://www.angusreid.com/us/media/content/displaypr.cfm?id_to_view=1253>. Acesso em: set. 2001.

[JAI 91] JAIN, Raj. **The art of computer systems performance analysis** : techniques for experimental design, measurement, simulation, and modeling. New York: John Wiley, 1991.

[KAR 95] KARN, P.; METZGER, P.; SIMPSON, W. **The ESP DES-CBC Transform**. 1995. RFC-1829. Disponível em:

<http://www.ietf.org/rfc/rfc1829.txt>. Acesso em: set. 2001.

[KEN 98] KENT, S; ATKINSON, R. **IP Encapsulating Security Payload (ESP)**. 1998. RFC-2406. Disponível em: <http://www.ietf.org/rfc/rfc2406.txt>. Acesso em: set. 2001.

[LID 86] LIDL, R.; NIEDERREITER, H. **Introdution to finite fields and their applications**. Cambridge: Cambridge Univerty Press, 1986.

[LOM 2000] LOMONT, Chris. **AES – Advanced Encryption Standard**. 2000. Disponível em: <http://www.math.purdue.edu/~clomont/software/AES/AES.htm>. Acesso em: out. 2001.

[MCC 2001] McCLURE, S.; SCAMBRAY, J.; KURTZ, G. **Hacking Exposed**: Network Security Secrets and Solutions. 3rd ed. Berkeley: McGraw-Hill, 2001.

[MAD 98] MADSON, C.; DORASWAMY, N. **The ESP DES-CBC Cipher Algorithm With Explicit IV** . 1998. RFC-2405. Disponível em: <http://www.ietf.org/rfc/rfc2405.txt>. Acesso em: set. 2001.

[MRO 2000] MROCZKOWSKI, Piotr. **Implementation of the block cipher Rijndael using Altera FPGA**. 1998. Disponível em: <http://csrc.nist.gov/encryption/aes/round2/comments/20000510-pmroczkowski.pdf>. Acesso em: ago. 2001.

[NCI 2001a] NCIPHER. **nForce Secure e-Commerce Accelerators**. 2001. Disponível em: <http://www.ncipher.com/products/nforce/index.html>. Acesso em: set. 2001.

[NCI 2001b] NCIPHER. **nShield Hardware Security Modules for PKI**. 2001. Disponível em: <http://www.ncipher.com/products/nshield/index.html>. Acesso em: set. 2001.

[NIE 2001] NIELSEN//NETRATINGS. **Global Internet Usage**. 2001. Disponível em: <http://www.nielsen-netratings.com/hot_of_the_net_i.htm>. Acesso em: set. 2001.

[NIS 95] NIST. **Secure Hash Standard**. 1995. FIPS PUB 180-1. Disponível em: <http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.pdf>. Acesso em: nov. 2001.

[NIS 99] NIST. **Data Encryption Standard**. 1999. FIPS PUB 46-3. Disponível em: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>. Acesso em: nov. 2001.

[NIS 2000a] NIST. **AES Round 2 Information**. 2000. Disponível em: <http://csrc.nist.gov/encryption/aes/round2/round2.htm#algorithms>. Acesso em: ago. 2001.

[NIS 2000b] NIST. **Commerce Department Announces Winner of Global Information Security Competition**. 2000. Disponível em: <http://www.nist.gov/public_affairs/releases/g00-176.htm>. Acesso em: ago. 2001.

[NIS 2000c] NIST. **Digital Signature Standard**. 2000. Disponível em: <http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf>. Acesso em: nov. 2001.

[NIS 2001a] NIST. **Advanced Encryption Standard (AES)**. 2001. Disponível em: <http://csrc.nist.gov/publications/drafts/dfips-AES.pdf>. Acesso em: ago. 2001.

[NIS 2001b] NIST. **Advanced Encryption Standard (AES)**. 2001. Disponível em: <http://csrc.nist.gov/encryption/aes>. Acesso em: ago. 2001.

[NIS 2001c] NIST. **AES Round 1 Information**. 2001. Disponível em: <http://csrc.nist.gov/encryption/aes/round1/round1.htm#algorithms>. Acesso em: ago. 2001.

[NIS 2001d] NIST. **Overview of the AES Development Effort**. 2001. Disponível em: <http://csrc.nist.gov/encryption/aes/index2.html#overview>. Acesso em: ago. 2001.

[NIS 2001e] NIST. **Recommendation for Block Cipher Modes of Operation**. 2001. Disponível em: <http://csrc.nist.gov/encryption/modes//Modes01.pdf>. Acesso em: ago. 2001.

[NSA 2000] NSA. **Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms**. 2000. Disponível em: <http://csrc.nist.gov/encryption/aes/round2/NSA-AESfinalreport.pdf>. Acesso em: jul. 2001.

[PAR 2001] PARKS. **UP 2048 HDSL**. 2001. Disponível em: <http://www.parks.com.br/up2048.html>. Acesso em: nov. 2001.

[PER 98] PEREIRA, R.; ADAMS, R. **The ESP CBC-Mode Cipher Algorithms**. 1998. RFC-2451. Disponível em: <http://www.ietf.org/rfc/rfc2451.txt>. Acesso em: set. 2001.

[PGP 2001] PGPI. **The International PGP Home Page**. 2001. Disponível em: <http://www.pgpi.org>. Acesso em: set. 2001.

[RAI 2001] RAINBOW. **Secure Network Perfomance**. 2001. Disponível em: <http://www.rainbow.com/cryptoswift/index.html>. Acesso em: set. 20011.

[RSA 2000] RSA. **RC6® Block Cipher**. 2000. Disponível em: <http://www.rsa.com/rsalabs/rc6/index.html>. Acesso em: ago. 2001.

[SCH 96] SCHNEIER, Bruce. **Applied cryptography** : protocols, algorithms, and source code in c. 2nd. ed. New York: John Wiley, 1996.

[SCH 2000] SCHNEIER, Bruce. **Secrets and Lies**: Digital Security in a Networked World**.** New York: John Wiley, 2000.

[SIL 2001] SILVEIRA, Christopher. **AES**: the new key on the block. 2001. Disponível em: <http://www.sans.org/infosecFAQ/firewall/AES.htm>. Acesso em: set. 2001.

[SIN 99] SINGH, Simon. **The Code Book**: the evolution of secrecy from Mary Queen of Scots to quantum cryptography. New York: Anchor Books, 1999.

[STA 95] STALLINGS, William. **Network and internetwork security principles and practice**. New Jersey: Prentice Hall, 1995.

[STA 2000] STALLINGS, William. **Network Security Essentials**: applications and standards. New Jersey: Prentice Hall, 2000.

[TAN 96] TANENBAUM, Andrew S. **Computer Networks**. 3rd ed. New Jersey: Prentice Hall, 1996.

[XIL 2000] XILINX. **VIRTEX E**. 2000. Disponível em: <http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=ss_vi-out>. Acesso em: 2001.