Um Cripto-Processador Reconfigurável
Baseado em Algoritmos de
Curvas Elípticas e AES

Roberto Alves Gallo Filho

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Cripto-Processador Reconfigurável Baseado em Algoritmos de Curvas Elípticas e AES

Roberto Alves Gallo Filho[1]

Abril de 2004

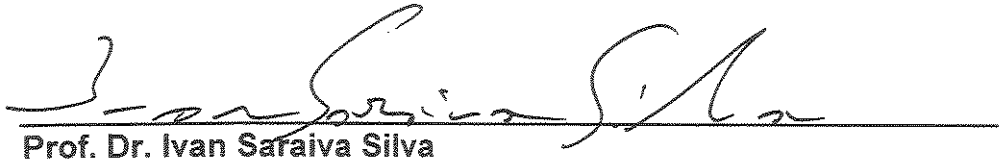
Banca Examinadora:

- Prof. Dr Guido Costa Souza Araújo (Orientador)
- Prof. Dr Ricardo Dahab
  Instituto de Computação – UNICAMP
- Prof. Dr. Ivan Saraiva Silva
  Departamento de Informática e Matemática Aplicada - UFRN
- Prof. Dr. Julio Lopéz (suplente)
  Instituto de Computação – UNICAMP

[1] Auxílio financeiro da FAPESP sob o processo número (01/12156-8)

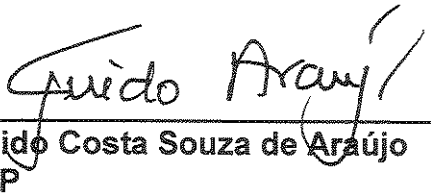
TERMO DE APROVAÇÃO

Tese defendida e aprovada em 20 de abril de 2004, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Ivan Saraiva Silva
DIMA - UFRN

Prof. Dr. Ricardo Dahab
IC - UNICAMP



Prof. Dr. Guido Costa Souza de Araújo
IC - UNICAMP

FICHA CATALOGRÀFICA ELABORADA PELA BIBLIOTECA DA UNICAMP

Gallo Filho, Roberto Alves

G137u Um cripto-processador reconfigurável baseado em algoritmos de curvas elípticas e AES / Roberto Alves Gallo Filho – Campinas, [S.P.:s.n], 2004

Orientador: Guido Costa Souza de Araújo

Dissertação (mestrado) Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Arquitetura de computadores. I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III Título.

Um Cripto-Processador Reconfigurável Baseado em Algoritmos de Curvas Elípticas e AES

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Roberto Alves Gallo Filho e aprovada pela Banca Examinadora.

Campinas, 20 de abril de 2004.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre me Ciência da Computação.

Substitua pela folha com a assinatura da banca

Resumo

Segurança da Informação é fator chave para o desenvolvimento da sociedade. Diariamente quantidades massivas de dados são transportadas em redes de alta velocidade sem o suporte adequado de segurança criptográfica.

Criptografia é infra-estrutura computacional e como tal está migrando para o *hardware*. Neste contexto a criação de um cripto-processador parametrizado utilizando tecnologia reconfigurável é de importância estratégica para o país.

Esta dissertação apresenta estudos acerca da construção de tal cripto-processador, com os estudos e as implementações de vários mecanismos relacionados com dispositivos criptográficos. Foram implementados e testados em VHDL[2] os algoritmos AES[3] – FIPS[4]197 e ECC[5] – IEEE[6]1363, rendendo aceleradores criptográficos de alto desempenho. Além disso, análises sobre a eficiência em hardware reconfigurável do algoritmo SHA-1[7] foram realizadas e seus resultados foram estendidos para toda família FIPS 180-2. O interfaceamento lógico e eletrônico do sistema também recebeu atenção nesta dissertação. No primeiro caso, um estudo de interfaces padrão para dispositivos criptográficos, ou *tokens*, baseados no PKCS[8]#11 foi realizado, rendendo uma implementação simplificada, mas compatível com este padrão. No caso do interfaceamento elétrico, diversas alternativas foram estudadas, resultando na escolha do sistema PCI PCI-SIG r.2.1.

[2] Do inglês: *Very High Speed Integrated Circuit Hardware Description Language*
[3] Do inglês: *Advanced Encryption Standard*
[4] Do inglês: *Federal Processing Information Standard*, padrão publicado pelo NIST – *National Institute of Standards and Technology* do Governo Americano.
[5] Do inglês: *Elliptic Curve Cryptography*
[6] Do inglês: *Institute for Electric and Electronic Engineers.*
[7] Do inglês: *Secure Hashing Algorithm revision 1*
[8] Do inglês: *Public Key Cryptographyc Standard*

Abstract

Information Security is a key factor for de development of modern society. Every day, massive amounts of data flow through high speed networks without proper cryptographic security support.

Cryptography services belong to the computational infrastructure and as such they are migrating into computer's hardware. In this context, the design of a parameterized crypto-processor that uses reconfigurable technology contributes to the development of such technology in the country.

This dissertation presents cases studies on the construction of such crypto-processor, along with de analysis of many mechanisms for cryptographic devices. By using VHDL for both implementation and testing, the AES – FIPS197 and the ECC – IEEE1363 algorithms were implemented yielding high performance cryptographic accelerators. In addition, efficiency analysis of hash functions as SHA-1 and others from its family were conducted for the resulting reconfigurable hardware. The logical and electrical interfaces received some attention too. In the first case, a brief study of cryptographic token interfaces, based on PKCS#11, was done and a simplified compatible implementation was constructed. For the electrical interface many alternatives were analyzed and the PCI PCI-SIG r.2.1 was chosen.

Agradecimentos

Sumário

Resumo vii
Abstract viii
Agradecimentos ix
Sumário x

1 Introdução 1
1.2 Criptografia 2
1.2.2 Objetivos Criptográficos 2
1.3 Arquiteturas Para Criptografia. 3

2 Matemática Preliminar 5
2.1 Pontos em Álgebra Finita 5
2.1.1 Corpos Finitos 5
2.1.2 Corpos Finitos Binários 6
2.1.3 Aritmética no Corpo Binário Utilizando Bases Normais. 8
2.1.4 Corpos Compostos e Mudança de Base 15
2.2 Curvas Elípticas 17
2.2.1 Introdução 17
2.2.2 Álgebra das Curvas Elípticas 17
2.2.3 A Multiplicação Escalar *kP* sobre Curvas Elípticas 20

3 Primitivas Criptográficas 25
3.1 Criptografia Simétrica – Cifradores por Blocos 25
3.1.1 Descrição Geral 25
3.1.2 Advanced Encryption Standard – AES FIPS 197 30
3.1.3 Data Encryption Standard – DES FIPS 46-2 40
3.2 Criptografia Assimétrica 46
3.2.1 Descrição Geral 46
3.2.2 Operações de ECC 46
3.2.3 Escolha dos Tamanhos e Tipos dos Corpos 48
3.3 Funções Hash 53
3.3.1 Introdução 53
3.3.2 Construção Usual, SHA e MD. 53
3.4 Números Aleatórios 56
3.4.1 Introdução 56
3.4.2 Estimadores de Aleatoriedade 57

4 Arquiteturas para Mecanismos Criptográficos 59
4.1 Introdução 59
4.1.1 Mecanismos em Hardware 59
4.1.2 Velocidade e Hardware 59
4.1.3 Caminhos Críticos e Hardware com VHDL 60
4.1.4 Desenrolamento de Iteração, Pipeline e Subpipeline 60

4.1.5 FPGAs e Sintetizadores 61
4.2 Entrada e Saída para dos Mecanismos 62
4.2.1 Escolha do Protocolo 62
4.2.2 O Padrão PCI 63
4.2.3 Implementação da Placa PCI em *Wire-up* 65
4.2.4 Problemas Encontrados 65
4.3 API compatível PKCS#11 66
4.4 Gerador de Números Aleatórios 68
4.5 Arquitetura para o DES 70
4.6 SHA e MD5. 73
4.7 Arquitetura para o AES 76
4.7.1 SubBytes e InvSubBytes 76
4.7.2 MixColumns e InvMixColumns 78
4.7.3 O Datapath 82
4.7.4 A Máquina de Expansão de Chaves 82
4.7.5 O Core AES 84
4.8 Arquiteturas para ECC 85
4.8.1 A Arquitetura Global 86
4.8.2 As Caixas *f* 88
4.8.3 Resultados e Análise. 90
4.9 Práticas de Projeto Derivadas 92

5 Conclusão e Trabalhos Futuros 97

Lista de Tabelas

Tabela 3.1 – Número de iterações para o AES. 33
Tabela 3.2 – Deslocamentos para a transformação ShiftRows do AES. 34
Tabela 3.3 – PC-1, permutação para o DES. 40
Tabela 3.4 – Deslocamentos para a chave do DES em bits. 41
Tabela 3.5 – PC-2, permutação dos bits para as subchaves. 41
Tabela 3.6 – Permutação Inicial IP 42
Tabela 3.7 – Seleção de Bits E 42
Tabela 3.8 – As oito tabelas S 43
Tabela 3.9 – Permutação P 44
Tabela 3.10 – Permutação final IP^{-1} 44
Tabela 3.11 – Segurança e Número e bits 49
Tabela 3.12 – Velocidades de Cálculo 51
Tabela 4.1 – Tabela comparativa de sistemas de IO de uso comercial. O padrão PCI e o Firewire apresentam as melhores características. 63
Tabela 4.2 – Resultados das análises estatísticas para o RNG implementado 69
Tabela 4.3 – Resultados obtidos para o *fitting* em diferentes dispositivos de nossa implementação do DES sem *pipeline*. 72
Tabela 4.4 Performances para algumas implementações do DES. 73
Tabela 4.5 - Performance para a operação de soma *mod* 2^{32} dos cinco elementos de SHA-1 utilizando arquitetura com um somador. 74
Tabela 4.6 - Performance para a operação de soma *mod* 2^{32} dos cinco elementos de SHA-1 utilizando arquitetura com somador em árvore. 74
Tabela 4.7 – SubBytes utilizando tabelas em implementações por LUTs e ROM. 77
Tabela 4.8 – SubBytes utilizando inversões. 77
Tabela 4.9 – Resultados para as transformações MixColumns e InvMixColumns para o *estado* inteiro. 80
Tabela 4.10 – Resultados de algumas configurações para o *core* utilizando a transformação SubBytes implementada por meio de inversão e transformada afim – alto desempenho. 85
Tabela 4.12 – Utilização de recursos e performances para diferentes corpos utilizando a arquitetura sem compartilhamento de *Xorers* em uma FPGA EP1C20C6 do ECCP. 90
Tabela 4.13 – Utilização de recursos com compartilhamento das operações de *xor*, utilizando o mesmo dispositivo, EP1C20C6. 90
Tabela 4.14 – Performances para variações no *pipeline* horizontal – aumento da performance sublinear em uma FPGA EP1C20C6. 91
Tabela 4.15 – Tempos de multiplicação de elementos do corpo para a arquitetura proposta. 91
Tabela 4.16 – Tempos de multiplicação de elementos do corpo para a arquitetura de Orlando e Paar utilizando a FPGA Xilinx Virtex E XCV400E8-BG432. 91

Lista de Figuras

Figura 2.1 – Inversão de elementos de GF(2^8) utilizando decomposição de Wada [52]. 13
Figura 2.2 – Gráfico da curva elíptica $y^2 = x^3 - 7x + 5$. 18
Figura 2.3 – Adição de pontos de curvas elípticas sobre números reais. 18
Figura 2.4 – Ciclos versus *m* para GF(2^{263}) – Comportamento do crescimento de *m*. 23
Figura 3.1 – Modo ECB – Possui nível de segurança inadequado para a maioria das aplicações. Funcionamento não encadeado. 28
Figura 3.2 – Modo CBC – Provavelmente o modo de operação mais utilizado para os cifradores por blocos. Possui muito bom nível de segurança. Funcionamento encadeado. 28
Figura 3.3 – Modo CFB – Adequado para blocos de dados de tamanhos diferentes do mecanismo base. Funcionamento encadeado. 29
Figura 3.4 – Modo OFB – Possíveis ciclos no espaço na *keystream* resultam em um nível de segurança menor que o máximo possível [54]. Funcionamento encadeado. 29
Figura 3.5 – Modo CTR – Bom nível de segurança, utilização sem necessidade de função de decifragem e funcionamento não encadeado. 30
Figura 3.6 – Esquema gráfico do DES. 45
Figura 3.7 – Gráfico representado a tabela 3 52
Figura 3.8 – Modelo genérico de função de *hashing* por blocos retirado de [42]. 54
Figura 4.1 – Uma transação pelo protocolo PCI. Uma transação é composta de um ciclo de endereços e um ou mais ciclos de dados. A transação mostrada é um *burst* de leitura. ... 64
Figura 4.2 – Ilustração do sistema de sessões e principais funções implementadas na API do projeto. 67
Figura 4.3 – Esquema elétrico da parte responsável pela geração de entropia do RNG 68
Figura 4.4 – Arquitetura implementada para o DES. 71
Figura 4.5 – Somador para as arquiteturas voltadas à implementação da função de compressão de SHA-1. 75
Figura 4.6 – Arquitetura para as tabelas S utilizando compartilhamento da inversão. 78
Figura 4.7 – O *Datapath* dual para o AES. A localização do subpipeline se justifica com os atrasos infligidos pela adição de round e pelo multiplexador na figura. A lógica de adição para o round final é apenas sintetizada naqueles rounds que podem servir de saída da arquitetura. Em uma versão *full pipelined* com suporte a apenas um tamanho de chaves, a área demarcada por "Final Round Only" é sintetizada uma única vez. 81
Figura 4.9 – Máquina de expansão de chaves de tamanhos 128 e 256 bits para o AES. 83
Figura 4.10 – Unidade de multiplicação com pipeline horizontal para corpos finitos binário com representação em bases normais. 87
Figura 4.11 – Estrutura interna da Caixa *f*. Capacidade para o cálculo para mais de um tamanho de corpo. 89

Capítulo 1

Introdução

1.1 A Importância da Segurança da Informação

Vivemos a sociedade da Informação. As origens da denominação podem ser duas: na primeira a Informação seria a cada dia que passa o elemento de unidade e aglutinação da sociedade. Corrobora com isso o fato de que Informação e hoje uma das mercadorias mais valiosos na sociedade. Fato concreto é que a Informação é um bem de importância e valor inegável, e como tal, sua Segurança merece especial atenção.

Por outro lado, a crescente utilização de dispositivos computacionais, a maioria deles ligados à Rede, tornou a idéia de computação ubíqua uma realidade. Estes dispositivos operam freqüentemente com, dados sensíveis de seus usuários diretos ou indiretos e portanto demandam mecanismos de segurança.

Através da utilização da Criptografia, em adição a outras técnicas, objetivos como confidencialidade e autenticidade da informação e de entidades podem ser alcançados. Em razão de tais objetivos serem demandas recorrentes em todos os sistemas computacionais modernos, a noção de que a Criptografia se incorporará à estrutura computacional está se tornando um consenso.

Como infra-estrutura básica, os serviços criptográficos mais básicos tendem a migrar para hardware, pois desta forma reduzem a carga do sistema, adicionado confiabilidade e requerendo menos recursos que quando implementados somente em software.

Devida à própria natureza e complexidade dos mecanismos básicos de criptografia, suas implementações tendem a ter certas deficiências em ambientes específicos. Por exemplo, Em dispositivos de baixo poder computacional, como aqueles baseados em microcontroladores, a utilização de mecanismos (ou primitivas) criptográficos assimétricos é muitas vezes impossível.

Já em situações de alta demanda de autenticações ou mesmo de cifragem massiva, muito comum em *backbones*, unidades certificadoras ou mesmo em satélites, processadores de uso geral são normalmente incapazes de oferecer o desempenho necessário.

Esta deficiência tem explicação no fato de que tipicamente as operações básicas relacionadas com criptografia não utilizam aritmética convencional ou utilizam operações em nível de bits, contrário ao padrão centrado em múltiplos de bytes dos processadores modernos. Em ambos os casos, hardware dedicado é a melhor opção.

1.2 Criptografia

1.2.1 Contexto Histórico

Menezes et al [42] apresentam a seguinte *Definição*:

"Criptografia é o estudo de técnicas matemáticas relacionadas com aspectos da segurança da informação, como confidencialidade, integridade, autenticidade de entidades e autenticidade da origem da informação."

A história da criptografia remonta aos egípcios, cerca de 2000 A.C, onde sua utilização era restrita aos escribas dos templos, até os dias modernos onde tem papel fundamental no funcionamento de nossa sociedade da informação. O desenvolvimento desta ciência teve alguns momentos-chave de aceleração. Foram cruciais em ambas Guerras Mundiais, tiveram grande demanda com a proliferação de sistemas computacionais de comunicação nos anos sessenta e na década de setenta teve seu mais profundo salto, com a introdução da criptografia de chaves públicas por Diffie e Hellman [44].

A década de setenta foi testemunha de mais avanços. O desenvolvimento do mais conhecido mecanismo de criptografia de toda a história, o DES – Data Encryption Standard[45], que foi originário dos trabalhos de Feistel, na IBM e a descoberta por Rivest, Shamir e Adleman do primeiro sistema de chaves públicas para cifragem e assinaturas de uso prático, conhecido por RSA[46].

1.2.2 Objetivos Criptográficos

Existem vários objetivos em termos de Segurança da Informação. A maioria deles, no entanto, pode ser derivada de quatro principais objetivos criptográficos que constituem base para os demais. São eles confidencialidade, autenticidade (integridade) dos dados, autenticação e não-repudiação [42]:

1. *Confidencialidade (ou privacidade)*, é o serviço de manter a informação secreta de todos aqueles que não estejam autorizados a tê-la.
2. *Autenticidade (Integridade) de dados*, é o serviço que garante, impede, ou detecta a modificação não autorizada da informação. O termo modificação pode ser entendido como inserção, remoção e substituição do todo ou de pedaços de informação.
3. *Autenticação é relacionada com identificação*. É o serviço que garante tanto a identificação da origem dos dados quanto a autenticidade das entidades envolvidas.
4. *Não-repudiação*, é o serviço que impede que uma entidade negue ações ou compromissos passados.

Cada um destes objetivos criptográficos apóia-se em determinadas classes de primitivas de segurança para sua consecução. Estas primitivas podem ser inicialmente divididas em três classes, baseadas em suas dependências às chaves: (a) as sem chaves como funções hash e seqüências aleatórias; (b) as de chaves simétricas como cifradores de chaves simétricas e MACs (*Message Authentication Codes*) e; (c) primitivas de chaves públicas como cifradores de chaves públicas e assinaturas.

1.3 Arquiteturas Para Criptografia.

Arquiteturas baseadas em *hardware* especializado para serviços criptográficos podem variar de acordo com a sua complexidade, mecanismos e funcionalidades. A especialização de uma arquitetura para criptografia pode ser feita através de: (a) alteração do *instruction set* de um processador a fim de acelerar algum passo de um algoritmo; (b) aceleradores criptográficos completos; ou finalmente (c) na forma de um *Hardware Security Module* – HSM.

Os HSM normalmente têm seus níveis de segurança físico associados ao padrão FIPS 140-2[47]. Este padrão estabelece uma série de práticas a serem seguidas a fim de tornar os módulos seguros, compreendendo características elétricas, físicas e protocolos de operação, dentre outras.

Os *Tokens* criptográficos são dispositivos de *hardware*, *software* ou mistos que implementam alguma função criptográfica. Tokens podem ou não ser portáteis e um único dispositivo pode conter mais de um token lógico [41]. Um HSM, aceleradores de hardware, guardadores de chaves USB e diversos outros dispositivos podem ser vistos como tokens.
A fim de padronizar o acesso lógico a estes dispositivos a RSALabs [41] promoveu e publicou o PKCS#11 – *Cryptographic Token Interface Standard* e o PKCS#15 [48] – *Cryptographic Token Information Format Standard*, atualmente bastante divulgados na indústria.

Do ponto de vista computacional, e mais precisamente criptográfico, existem funções bem claras que um HSM deve desempenhar. E mais, devido ao fato de que tal módulo é um objetivo *bastante* complexo tanto em termos de pesquisa de desenvolvimento, quanto em termos estratégicos de âmbito nacional, as funcionalidades requeridas constituem o objetivo de pesquisa da presente dissertação.

Desta forma, uma série de primitivas e mecanismos criptográficos e aspectos de Engenharia foram estudados nesta dissertação, aqui se propõem soluções para vários problemas associados com projeto de um HSM, principalmente focando na construção de um criptoprocessador para o mesmo. São apresentadas análises de primitivas, tais como: mecanismos simétricos (AES, DES/3DES e MESH[49]), mecanismo assimétrico (ECCs), funções hash (SHA-1 e família), protocolos de comunicação (lógico, PKCS#11 e 15 e eletrônico, PCI PCI-SIG r2.1) e geração de números aleatórios (TRNG[9] pelos ruídos Johnson e avalanche).

Esta dissertação está divida em quadro capítulos da seguinte forma:

[9] Do Inglês: *True random number generator*

- Capítulo 2 – Matemática Preliminar. Neste capítulo são revisadas algumas ferramentas necessárias à implementação e a otimização dos mecanismos criptográficos envolvidos no restante do documento como pontos em álgebra finita e curvas elípticas.
- Capítulo 3 – Primitivas Criptográficas. Lá são apresentados com detalhes os mecanismos criptográficos que foram alvo de implementação nesta dissertação, nominalmente o AES, DES, operações para ECC, funções *hash* e geração de números aleatórios.
- Capítulo 4 – Arquiteturas para Mecanismos Criptográficos. São apresentadas as arquiteturas propostas e implementadas para os mecanismos mencionados anteriormente. Além disso, análises de performance das arquiteturas são realizadas comparando com resultados da literatura e de implementações comerciais.
- Capítulo 5 – Conclusão e Trabalhos Futuros. São apresentados possíveis temas para trabalhos futuros e seus pontos de pesquisa, além das conclusões desta dissertação.

Capítulo 2

Matemática Preliminar

Neste capítulo são introduzidas algumas das bases matemáticas relacionadas com os mecanismos criptográficos abordados nesta dissertação. A seção 2.1 discorre sobre pontos em álgebra finita, enquanto a seção 2.2 apresenta brevemente as curvas elípticas.

2.1 Pontos em Álgebra Finita

2.1.1 Corpos Finitos

Um corpo é uma estrutura matemática composta por um conjunto de elementos e sobre o qual são definidas duas operações chamadas de soma e multiplicação, normalmente denotadas por + e × respectivamente. Estas operações possuem as seguintes propriedades:

- A soma e multiplicação são fechadas, associativas e comutativas;
- Existem identidades aditivas e multiplicativas;
- Qualquer elemento do corpo possui um inverso aditivo;
- Qualquer elemento do corpo possui um inverso multiplicativo, com exceção à identidade aditiva;
- A multiplicação é distributiva com relação à soma.

A característica de um corpo é o número k, tal que k sucessivas somas do elemento identidade da multiplicação resulta no elemento identidade aditivo. Se isso não ocorrer, dizemos que a característica do corpo é 0. Se a característica de um corpo é diferente de 0, então o corpo é dito finito.

A ordem q de um corpo é o número de elementos que ele contém. Se $q > 1$ é um inteiro, então o corpo finito de ordem q existe se, e somente se, q é uma potência de um número primo. Um corpo finito de ordem q é normalmente denotado GF(q).

As propriedades dos corpos finitos independem da forma escolhida para a sua representação, apesar das operações realizadas sobre seus elementos serem sensíveis a essa escolha.

2.1.2 Corpos Finitos Binários

Seja *m* um inteiro positivo, então o corpo finito binário GF(2^m) pode ser representado pelas 2^m diferentes cadeias de bits de tamanho m; assim, para GF(2^3), por exemplo temos:

GF(2^3) = {000, 001, 010, 011, 100, 101, 110, 111},

e *m* é chamado de grau do corpo.

Para *m* igual a 1, o corpo GF(2) é apenas o conjunto {0, 1} módulo 2. A adição e a multiplicação são:

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

2.1.2.1 Adição

Com *m* > 1, a adição de quaisquer dois elementos de GF(2^m) é dada pela adição bit a bit módulo 2. Assim, por exemplo:

(11001) + (10100) = (01101).

Como dito, esta operação é equivalente a:

(11001) + (10100) = (11001) ⊕ (10100) = (01101).

2.1.2.2 Multiplicação

Há várias maneiras de se efetuar eficientemente a multiplicação em corpos finitos binários, dependendo das famílias escolhidas para representação das bases. Estas famílias definem as regras matemáticas da multiplicação e são principalmente três: as polinomiais, as normais e as duais [50].

2.1.2.3 A Representação em Bases Polinomiais.

Em bases polinomiais, cada elemento de GF(2^m) é interpretado como um polinômio binário (i. e., cujos coeficientes pertencem a GF(2)), com grau menor que *m*. Assim, a seqüência de bits (a_{m-1}... a_1 a_0) representa o polinômio em *t* binário:

$a_{m-1}t^{m-1} + ... + a_1t^1 + a_0$.

A adição em bases polinomiais, e nas normais, de corpos binários já foi descrita acima. A multiplicação, por sua vez, é definida em termos de um polinômio binário irredutível (ou primo) de grau *m*, *p(t)*, chamado de polinômio para a representação do corpo. Assim, o produto de dois elementos é simplesmente o produto dos dois polinômios respectivos reduzidos a módulo *p(t)*.

2.1.2.4 A Representação em Bases Normais

Como citado anteriormente é possível descrever elementos de um corpo usando polinômios na variável *t*. Seja β um elemento do corpo GF(p^m), com representação polinomial:

$$\beta = a_{m-1}t^{m-1} + ... + a_1t^1 + a_0. \quad (1)$$

Uma base normal pode ser formada usando-se as potências de β dada em (1):

$$\left\{\beta, \beta^{p}, \beta^{p^2}, ..., \beta^{p^{m-1}}\right\}$$

Tal base sempre existe [4] e pode ser encontrada para qualquer corpo finito. Para o caso de máquinas binárias é de interesse utilizar $p = 2$:

$$\left\{\beta, \beta^{2}, \beta^{2^2}, ..., \beta^{2^{m-1}}\right\}$$

Qualquer elemento *e* do corpo GF(2^m) pode ser representado em bases normais como:

$$e = e_0 + e_1\beta^4 + ... + e_{m-1}\beta^{2^{m-1}} = \sum_{i=0}^{m-1} e_i\beta^{2^i}$$

2.1.2.5 Corpos de Extensão e Polinômios

Um polinômio irredutível P(x) sobre GF(q) de grau *m* pode ser utilizado para construir um corpo de extensão de GF(q). O corpo construído é denotado por GF(q^m) e tem ordem q^m. Todos os elementos deste novo corpo podem ser representados como polinômios com coeficientes em GF(q) de grau máximo m – 1, classes de resíduos de todos os polinômios sobre GF(q) modulo P(x).

Os corpos compostos, um subtipo especial de corpos de extensão, binários, podem ser definidos, veja por exemplo [50].

Os pares de característica 2

$$\left\{GF\left(2^n\right), Q\left(y\right) = y^n + \sum_{i=0}^{n-1} q_i y^i\right\} \text{ e}$$

$$\left\{GF\left(\left(2^n\right)^m\right), P\left(x\right) = x^m + \sum_{i=0}^{m-1} p_i x^i\right\},$$

constituem um corpo composto denotado por GF($(2^n)^m$) se:

- GF(2^n) tem polinômio gerador Q(y) e;
- GF($(2^n)^m$) tem polinômio gerador P(x).

Um corpo composto GF($(2^n)^m$) é isomórfico matematicamente com o corpo binário GF(2^k), com $k = nm$, mas suas operações, × e +, são possivelmente diferentes tanto algebricamente quanto em sua complexidade [4][50].

2.1.3 Aritmética no Corpo Binário Utilizando Bases Normais.

As operações aritméticas usando bases normais são mais elaboradas do que aquelas empregadas em bases polinomiais. No entanto, quando bem empregada, as bases normais podem introduzir grandes ganhos em projetos computacionais, pois suas operações podem ser facilmente descritas em termos de rolagens, somas tipo XOR e produtos AND sobre os bits. A potenciação ao quadrado de um elemento representado em bases normais é um exemplo típico, sendo explicada em seguida.

A multiplicação em bases normais é um processo algebricamente complexo, mas que também pode ser bastante eficiente quando Bases Normais Gaussianas, ou BNG, são utilizadas [4][1].

Por este motivo ANSI X9.62 [1] especifica que as BNG devem ser utilizadas, pois a multiplicação com elas é tanto mais simples como mais eficiente.

2.1.3.1 O Quadrado de um Elemento Finito

Um aspecto positivo das bases normais é que a operação de elevar um elemento ao quadrado pode ser efetuada apenas com uma rotação da seqüência de bits que representa o elemento. Seja

$a = (a_0 a_1 ... a_{m-2} a_{m-1}) \in$ GF(2^m);

então

$$a^2 = \left(\sum_{i=0}^{m-1} a_i \beta^{2^i}\right)^2 = \sum_{i=0}^{m-1} a_i \beta^{2^{i+1}} = \sum_{i=0}^{m-1} a_{i-1} \beta^{2^i} = \left(a_{m-1} a_0 \ldots a_{m-2}\right),$$

com índices reduzidos módulo *m*. A adição por sua vez pode ser efetuada em bases normais da mesma forma que é efetuada em bases polinomiais.

2.1.3.2 A Multiplicação em Bases Normais Gaussianas - BNG

A aritmética da multiplicação em bases normais pode ser bastante complicada. Porém, o seu princípio é o mesmo: devem-se multiplicar os coeficientes e somar os que tiverem a mesma potência. A regra para a multiplicação é dada abaixo:

Seja $p = Tm + 1$ e seja u $\in$ GF(p) um elemento de ordem T. Define-se a seqüência F(1), F(2), ..., F(p-1) por:

$F(2^i u^j \bmod p) = i$ para $0 \le i \le m\text{-}1$, $0 \le j \le T - 1$

Se $a = (a_0 a_1 \ldots a_{m-2} a_{m-1})$ e $b = (b_0 b_1 \ldots b_{m-2} b_{m-1}) \in GF(2^m)$, então $a \,.\, b = c$, onde

$$c_i = \begin{cases} \displaystyle\sum_{k=1}^{p-2} a_{F(k-1)+i} b_{F(p-k)+i} & \text{se T é par} \\ \displaystyle\sum_{k=1}^{\frac{m}{2}} \left(a_{a+i-1} b_{m/2+k+i-1} + a_{m/2+k+i-1} b_{k+i-1}\right) + \sum_{k=1}^{p-2} a_{F(k-1)+i} b_{F(p-k)+i} & \text{caso contrário} \end{cases}$$

A complexidade da multiplicação depende do parâmetro T. Alguns valores de T para os corpos recomendados pelo NIST são 4 e 2 para os corpos com *m* igual a 163 e 233 respectivamente.

Particularmente para i = 0 a regra acima retorna a expressão para c_0. Uma propriedade importante é que as expressões para $c_1, c_2 \ldots c_{m-1}$ são obtidas da fórmula de c_0 por uma rotação dos índices de *a* e *b* módulo *m*. Para $m = 53$, por exemplo, a expressão de c_0 é:

```
c0 = a0 b1 + a1 (b0 + b17) + a2 (b17 + b47) + a3 (b43 + b34) + a4 (b11 + b29)  + a5 (b27 + b39)
+ a6 (b8 + b24)  + a7 (b49 + b23)  + a8 (b40 + b6)  + a9 (b23 + b20)  + a10 (b28 + b13)  + a11
(b44 + b4)  + a12 (b32 + b27)  + a13 (b10 + b21)  + a14 (b19 + b44)  + a15 (b41 + b51) + a16
b46 + b52) + a17 (b1 + b2) + a18 (b47 + b43) + a19 (b22 + b14) + a20 (b9 + b41) + a21 (b13 + 33)
+ a22 (b48 + b19) + a23 (b7 + b9) + a24 (b6 + b28) + a25 (b35 + b49) + a26 (b38 + b31) + a27
(b12 + b5) + a28 (b24 + b10) + a29 (b4 + b35) + a30 (b39 + b37) + a31 (b26 + b50) + a32 (b45 +
b12) + a33 (b21 + b42) + a34 (b3 + b48) + a35 (b29 + b25) + a36 (b37 + b38) + a37 (b30 + b36) +
a38 (b36 + b26) + a39 (b5 + b30) + a40 (b50 + b8) + a41 (b20 + b15) + a42 (b33 + b46) + a43
(b18 + b3) + a44 (b14 + b11) + a45 (b51 + b32) + a46 (b42 + b16) + a47 (b2 + b18) + a48 (b34 +
b22) + a49 (b25 + b7) + a50 (b31 + b40) + a51 (b15 + b45) + a52 (b16 + b52)
```

Observa-se, nesta expressão, que ela é função de todos os bits das entradas *a* e *b*. Isso implica que para se determinar cada um dos *m* bits da saída, são necessárias operações sobre todos os 2*m* bits da entrada, ou seja, a complexidade da multiplicação é O(m^2) no número de bits.

2.1.3.3 A Inversão em Bases Binárias

Existem diversas técnicas de inversão em bases binárias, mais ou menos eficientes dependendo da representação utilizada para os elementos do corpo. Paar [50] discute minuciosamente a inversão para corpos binários de tamanho reduzidos, $4 < m < 32$.

Antes de introduzirmos a inversão em bases binárias, é necessário apresentar o *Teorema de Fermat* para GF(p) [3]:

Seja qualquer número inteiro a e qualquer número primo p tal que MMC(a, p) = 1, ou seja, a e p são co-primos. O Teorema de Fermat diz que:

$$a^p = 1 \bmod p$$

ou

$$a^{p-1} = a \bmod p$$

Uma extensão do Teorema de Fermat, para corpos binários GF(2^m), mostra que:

$$b^{-1} = b^{2^m - 2}$$

Algoritmo Para Calcular a^{-1} em GF(2^m)

Neste caso, podemos exponenciar diretamente b e obter o resultado da inversão. Entretanto é fácil notar que esse método é muito ineficiente, pois precisa de m quadrados e $m-1$ multiplicações, de longe a operação algébrica básica mais custosa a ser efetuada, a não ser pela própria inversão.

Pode-se, no entanto, fatorar o expoente de b a fim de se diminuir o número de multiplicações e exponenciações necessárias, fazendo:

$$2^m - 2 = 2\,(2^{m-1} - 1),$$

implica em:

$$b^{-1} = \left(b^{2^{m-1}-1}\right)^2$$

Se chamarmos $m-1 = r_0$, e supusermos que r_0 é par, poderemos fatorar,

$2^{r0} - 1 = (2^{r0/2} - 1)\,(2^{r0/2} + 1)$.

Como computar *b* elevado a $2^{r0/2} + 1$ é fácil, pois se trata apenas de uma rolagem e de uma multiplicação, este método é computacionalmente interessante. Se, por outro lado, r_0 for ímpar, teremos:

$$2^{r_0} - 1 = 2\left(2^{\frac{r_0-1}{2}} - 1\right)\left(2^{\frac{r_0-1}{2}} + 1\right) + 1$$

O que, em termos de exponenciação, significa [88]:

$$\left[\left(b^{\left(2^{\frac{r_0-1}{2}}-1\right)}\right)^{\left(2^{\frac{r_0-1}{2}}+1\right)}\right]^2 b$$

Isto é, se r_0 for impar, nós teremos uma multiplicação e uma inversão a mais do que no caso par. Para m igual a 53, por exemplo, a decomposição se torna:

253 - 2 = 2 (252 - 1)
252 - 1 = (226 - 1) (226 + 1)
226 - 1 = (213 - 1) (213 + 1)
213 - 1 = 2 (26 - 1) (26 + 1) + 1
26 - 1 = (23 - 1) (23 + 1)
23 - 1 = 2(2 + 1) + 1

Ou seja, a inversão pode ser assim obtida em termos de algumas poucas multiplicações e exponenciações. Novamente, para $m = 53$, podem-se utilizar as seguintes operações para a inversão:

Sejam *x*, *y*, *z* e *w* elementos representados em bases binárias:

$x = x^2$,

$x = x^{2^{26}} x$,

$x = x^{2^{13}} x$,

$y = x^2$,

$y = y^{2^6} y$,

$y = y^{2^3} y$,

$z = y^2$,

$z = z^2 z$,

$w = xyz$

onde w é o resultado da inversão.

Uma pergunta natural que pode surgir neste ponto é: "Qual o número de multiplicações necessárias para se inverter um elemento finito no método geral?". Pode-se mostrar que o número necessário destas operações à inversão em corpos bináriss é dado por [85]:

$M(m) = \lfloor \log_2 (m-1) \rfloor + g(m-1) - 1$,

onde

m é o grau do corpo de $GF(2^m)$, e
$g(m-1)$ é uma função que retorna o numero de bits 1 na representação binária de *m-1*.

Exemplo: $GF(2^8)$

Na especificação do algoritmo AES – FIPS 197 [39], as tabelas S, ou S-Boxes, são funções que operam sobre elementos de 8 bits, S-Box : $GF(2^8) \rightarrow GF(2^8)$ e são compostas por duas operações distintas. A primeira é a inversão do elemento de entrada, cuja representação é tida como polinomial com polinômio irredutível $m(x) = x^8 + x^4 + x^3 + x + 1$. Já a segunda é a aplicação de uma transformação afim, descrita mais adiante.

Pode-se utilizar o Método de Inversão Geral para calcular o inverso de um elemento y [52]:

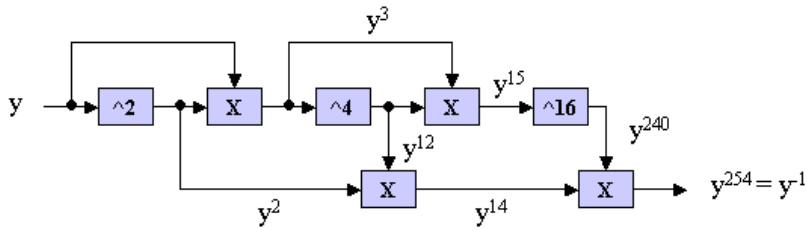


Figura 2.1 – Inversão de elementos de GF(2^8) utilizando decomposição de Wada [52].

A fatoração pode ser realizada de maneiras distintas, a fim de priorizar operações de menor custo. Para a finalizar, é necessário obter uma expressão para a multiplicação de dois elementos de GF(2^8).

Se observarmos que $m(x) \equiv 0 \bmod m(x)$, então podemos fazer:

$$x^8 = x^4 + x^3 + x + 1 \pmod{m(x)}$$

e utilizar este resultado para achar C(x), tal que:

$$A(x) \times B(x) = C(x)$$

Lembrando que $A(x)$ e $B(x)$ são polinômios binários de grau menor do que 8, obtém-se um polinômio $C(x)$ de grau máximo 14, com índices explicitados abaixo:

$$(c_{14}x^{14} + c_{13}x^{13} + c_{12}x^{12} + c_{11}x^{11} + c_{10}x^{10} + c_9x^9 + c_8x^8 +$$
$$+ c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0) =$$
$$(a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \times$$
$$(b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0)$$

$$c_{14} = a_7b_7$$
$$c_{13} = a_7b_6 + a_6b_7$$
$$c_{12} = a_7b_5 + a_6b_6 + a_5b_7$$
$$c_{11} = a_7b_4 + a_6b_5 + a_5b_6 + a_4b_7$$
$$c_{10} = a_7b_3 + a_6b_4 + a_5b_5 + a_4b_6 + a_3b_7$$
$$c_9 = a_7b_2 + a_6b_3 + a_5b_4 + a_4b_5 + a_3b_6 + a_2b_7$$
$$c_8 = a_7b_1 + a_6b_2 + a_5b_3 + a_4b_4 + a_3b_5 + a_2b_6 + a_1b_7$$
$$c_7 = a_7b_0 + a_6b_1 + a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 + a_1b_6 + a_0b_7$$
$$c_6 = a_6b_0 + a_5b_1 + a_4b_2 + a_3b_3 + a_2b_4 + a_1b_5 + a_0b_6$$
$$c_5 = a_5b_0 + a_4b_1 + a_3b_2 + a_2b_3 + a_1b_4 + a_0b_5$$
$$c_4 = a_4b_0 + a_3b_1 + a_2b_2 + a_1b_3 + a_0b_4$$
$$c_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3$$
$$c_2 = a_2b_0 + a_1b_1 + a_0b_2$$
$$c_1 = a_1b_0 + a_0b_1$$

$c_0 = a_0 b_0$

então, usando a relação $x^8 = x^4 + x^3 + x + 1$ e a identidade acima, pode-se terminar cada um dos índices de $D(x)$, o resíduo polinomial de $C(x)$:

$$
\begin{aligned}
d_7 &= c_7 + c_{11} + c_{12} + c_{14} \\
d_6 &= c_6 + c_{10} + c_{11} + c_{13} \\
d_5 &= c_5 + c_9 + c_{10} + c_{12} \\
d_4 &= c_4 + c_8 + c_9 + c_{11} + c_{14} \\
d_3 &= c_3 + c_8 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} \\
d_2 &= c_2 + c_9 + c_{10} + c_{13} \\
d_1 &= c_1 + c_8 + c_9 + c_{12} + c_{14} \\
d_0 &= c_0 + c_8 + c_{12} + c_{13}
\end{aligned}
$$

É importante notar que as operações de adição, "+" e multiplicação acima que são realizadas sobre os coeficientes de $A(x)$, $B(x)$ e $C(x)$ estão em GF(2).

Em Paar [50] uma série de métodos de inversão são investigados. No caso específico do AES Rijmen [24] propõem uma mudança na representação dos elementos de $GF(2^8)$ que ao invés de ser interpretados como classes de resíduos de polinômios de grau máximo sete passam a ser utilizados como polinômios de grau máximo 1 com coeficientes em $GF(2^4)$, um esquema semelhantes daquele empregado em bases compostas.

Assim, qualquer elemento de $GF(2^8)$ pode ser representado na forma $b\,x + c$, com polinômio irredutível $x^2 + \mathrm{A}\,x + \mathrm{B}$ e com coeficientes a e b em $GF(2^4)$. O inverso deste elemento pode ser escrito da seguinte forma:

$$(bx + c)^{-1} = b(b^2B + bcA + c^2)^{-1}\,x + (c + bA)\,(b^2B + bcA + c^2)^{-1}$$

O problema de se calcular o inverso de $GF(2^8)$ pode então ser transformado no de se calcular o de $GF(2^4)$ além de alguma multiplicações, somas e exponenciações. O inverso em $GF(2^4)$ pode ser armazenado em pequenas tabelas (16 *nibbles*). As somas podem ser realizadas com uma simples operação como descrito anteriormente para o caso de bases binárias.

Com a utilização de BNGs para a representação do corpo $GF(2^4)$, as exponenciações podem ser descritas como simples rolagens e as multiplicações podem ser implementadas de maneiras mais ou menos eficientes.

De qualquer forma, é necessário definir um mapeamento isomórfico entre as representações das bases para que a inversão seja representada corretamente.

2.1.4 Corpos Compostos e Mudança de Base

2.1.4.1 Matriz de Mudança de Base

Para que operações realizadas em diferentes representações de bases sejam úteis, muitas vezes é necessário que um mapeamento entre estas representações seja corretamente efetuado. Paar [50] define um algoritmo, que, dada duas representações para um corpo de tamanho $k = mn$ gera uma matriz de mapeamento T entre $GF(2^k)$ e $GF((2^n)^m)$, respectivamente representações polinomiais e compostas.

A matriz T tem tamanho $(k \times k)$ e opera a transformação direta entre as representações anteriores enquanto T^{-1}, sua inversa, opera o mapeamento no sentido contrário. No que se segue, $Q(y)$ representa o polinômio irredutível que gera o subcorpo $GF(2^n)$, com coeficientes em GF(2). Similarmente, $P(x)$ é o polinômio irredutível com coeficientes em $GF(2^n)$, que gera o corpo composto $GF((2^n)^m)$, e R(z) é o polinômio que gera $GF(2^k)$. O símbolo α denota um elemento primitivo de $GF((2^n)^m)$, que é raiz de P(x), de forma que $P(\alpha) = 0$. O símbolo ω denota um elemento primitivo de $GF(2^n)$, de forma que $Q(\omega) = 0$ e β representa uma raiz de R(z).

Cada elemento de $GF(2^k)$ pode ser representado por um vetor que dita a combinação linear dos elementos k da base polinomial $B_2 = (\beta^{k-1}, \beta^{k-2}, ..., \beta, 1)$.

Para se construir a matriz de mapeamento é necessário achar os k elementos base da representação $GF((2^n)^m)$ correspondentes aos k elementos base da representação de $GF(2^k)$, de modo que cada elemento β^i seja mapeado para um elemento primitivo α^{it}, ou seja:

$$T\beta^i = \alpha^{it}, \quad 0 \leq t < k$$

Torna-se então necessário determinar o expoente t de tal sorte que o mapeamento seja homomórfico com respeito a ambas operações do corpo, pois não basta mapear β para qualquer elemento α^t. A condição é a seguinte:

$$R(\alpha^t) = 0 \pmod{Q(y), P(x)}.$$

O onde o par de módulos representa a redução dos coeficientes (mod Q(y)) e posterior redução do polinômio (mod P(x)). O algoritmo seguinte [50], acha a matriz de mapeamento:

1. (Inicialização) Seja α um elemento primitivo em $GF((2^n)^m)$ para o qual $P(\alpha) = 0$. Faça t = 1 e prepare uma lista com $2^k - 1$ endereços de memória de 1 bit com o valor "verdadeiro". Faça a coluna mais a direita de T igual ao vetor coluna $(0, 0, ..., 0, 1)^T$, provendo o mapeamento para o elemento da base 1.
2. Se $R(\alpha^t) \pmod{Q(y), P(x)} = 0$, o valor de t foi achado e portanto o elemento α^t. Vá para o passo 7.

3. Nem α^t, nem seus conjugados $\alpha^{2^j t}$ são elementos para os quais β deva ser mapeado. Então faça com que todos os elementos da lista nas posições $t2^j \pmod{2^k - 1}$ sejam ajustados para "falso".
4. Incremente t (t = t +1) até que na posição t da lista haja um "verdadeiro".
5. Verifique se α^t é um elemento primitivo através do cálculo de MDC(t, 2^k -1). Se não o for, vá para o passo 4.
6. Vá para o passo 2.
7. Faça a segunda coluna mais à esquerda de T igual à representação binária de α^t e consecutivamente, da direita para a esquerda, dos elementos α^{2t}, α^{3t}, ... $\alpha^{(k-1)t}$.

Quanto à inicialização das posições de memória do passo 1 nota-se que para corpos de interesse criptográfico, com 160 < k < 512 [11] tal aproximação é naturalmente inviável. Uma solução de compromisso entre poder de processamento no cálculo dos índices $t2^j$ e espaço de armazenamento faz-se necessária. Nota-se que a exponenciação envolvida pode ser calculada apenas com deslocamentos.

2.1.4.2 Transformação entre outro corpo de extensão e GF(2^8) polinomial.

De forma semelhante ao caso composto-polinomial, é possível achar uma matriz de mudança de representação de bases na qual os coeficientes em GF(2^n) ao invés possuírem uma representação polinomial as têm em bases normais.

Para o caso específico dos S-Boxes do AES, o corpo GF(2^8), cuja a representação original utilizada é aquela de polinômio gerador $R(z) = z^8 + z^4 + z^3 + z + 1$ [39], deve ser mapeado para uma representação de bases cujo polinômio gerador tem forma $x^2 + Ax + B$ [24] e cujos coeficientes, que estão em GF(2^4), formam bases normais. Este mapeamento é motivado pelo método de inversão apresentado na subseção 2.1.3.3 e proposto em [24] e [53].

Uma vez que GF(2^4) forma BNG Tipo 1, esta será a representação utilizada. Baseando-se nesta escolha e na liberdade de escolha dos valores de A e B, o primeiro terá valor {1111}, a identidade multiplicativa, e B = {0001}, valor de baixo *hamming* [38].

Adaptando-se o passo 4 do algoritmo da seção anterior para bases normais, com a condição (mod Q(y)) modificada para a álgebra em BNG Tipo 1 (T = 1), obtém-se uma das 8 possíveis matrizes de mapeamento, junto de sua inversa:

$$T = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \qquad T^{-1} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

É importante notar que a inicialização da matriz T no passo 1 do algoritmo também foi modificada, já que o elemento identidade multiplicativa da nova representação é agora {00001111} ao invés de {00000001}.

2.2 Curvas Elípticas

2.2.1 Introdução

Em 1985 N. Koblitz e V. S. Miller [2][3][4] propuseram, independentemente, que curvas elípticas fossem usadas para esquemas de criptografia de chaves públicas. No entanto, no início, os cálculos envolvidos eram muito difíceis de serem executados, tanto por falta de poder computacional quanto pelo fato de que a matemática envolvida não era muito eficiente. Àquela época tratava-se apenas de um exercício acadêmico interessante.

Desde então, muito trabalho foi realizado a fim de gerar implementações eficientes. No final dos anos noventa, as implementações já eram dezenas de vezes mais rápidas do que as originais, igualando-se às dos algoritmos criptográficos baseados no problema da fatoração de inteiros com o mesmo número de bits.

Como os algoritmos de criptografia baseados em Curvas Elípticas - ECC requerem menos bits para o mesmo nível de segurança do que aqueles baseados em fatoração de números inteiros, há uma redução no tamanho de palavras e nos tempos de execução.

2.2.2 Álgebra das Curvas Elípticas

Uma curva elíptica pode ter várias formas. Iniciamos com a forma de "*Weierstrass*" [2]:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (2)$$

As variáveis x e y cobrem o plano e podem ser números reais, complexos, inteiros ou elementos de um corpo arbitrário.

Para ilustrar, examinamos uma forma simples da equação (2) com coeficientes reais:

$$y^2 = x^3 + a_4x + a_6 \qquad (3)$$

Fixando, por exemplo, a_4 em -7, a_6 em 5, x e y reais, podemos grafar a equação (3), obtendo a curva da figura 2:

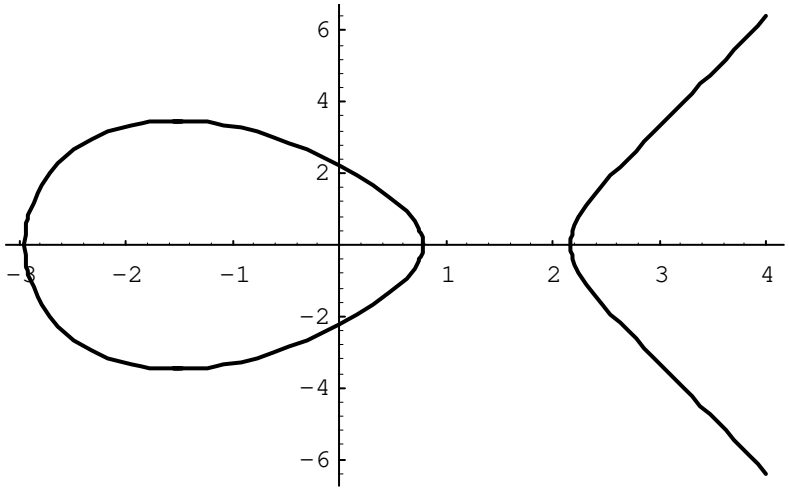


Figura 2.2 – Gráfico da curva elíptica $y^2 = x^3 - 7x + 5$.

É necessário, agora, definir a álgebra associada aos pontos sobre as curvas elípticas. Primeiro define-se "adição" de dois pontos sobre a curva, de modo que, a "soma" resulte em um outro ponto sobre a própria curva. Além disso, é necessário definir um elemento identidade denotado $\boldsymbol{O}_{\infty}$, chamado de ponto no infinito, de forma que para todo ponto P temos:

$$P + \boldsymbol{O}_{\infty} = P \tag{4}$$

Definimos ainda, baseados na equação (3) o "negativo" de um ponto P = (x, y), como sendo –P = (x, $-y$). Pode-se definir a adição P+Q de quaisquer dois pontos sobre a curva elíptica, traçando uma reta entre os dois pontos e encontrando a terceira interseção com a curva, como visto na figura 3. A interseção define o ponto –R e a soma P+Q é o ponto R:

$$R = P + Q \tag{5}$$

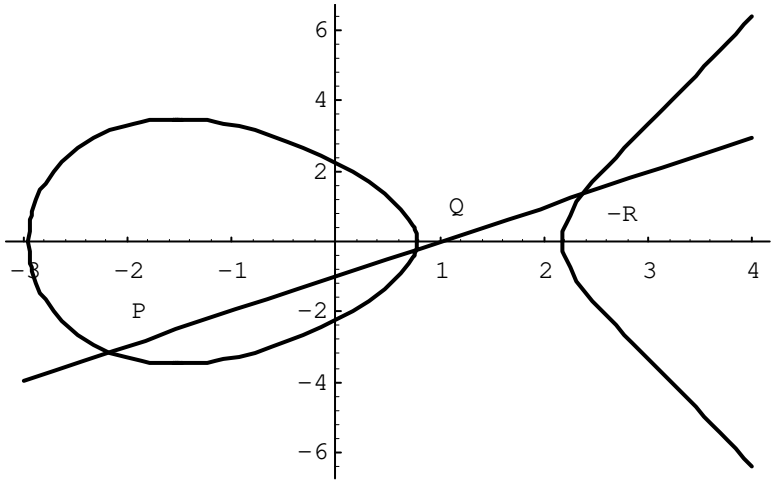


Figura 2.3 – Adição de pontos de curvas elípticas sobre números reais da curva $y^2 = x^3 - 7x + 5$.

Algebricamente, se:

$P = (x_1, y_1)$ e
$Q = (x_2, y_2)$,

então

$$R = P + Q = (x_3, y_3),$$

onde

$$x_3 = \theta^2 - x_1 - x_2,$$
$$y_3 = \theta\,(x_1 + x_3) - y_1,$$
$$\theta = (y_2 - y_1)\,(x_2 - x_1)^{-1} \qquad \text{se } P \neq Q,$$

ou

$$\theta = (3x^2{}_1 + a_4)\,(2y_1)^{-1} \quad \text{se } P = Q,$$

A adição de um ponto com ele mesmo é um caso especial e resulta na reta tangente à curva naquele ponto. O produto de um ponto por um escalar, aP, é definido como a adição de a termos iguais a P.

Analogamente às curvas elípticas sobre números reais, é possível utilizá-las com outros grupos cíclicos, tal qual GF(p)*, o corpo finito dos números inteiros módulo um primo p.

Define-se a adição de dois pontos sobre uma curva elíptica sobre o corpo finito binário GF(2^m) em termos de coordenadas afins:

Sejam os pontos P e Q e a curva E sobre GF(2^m):

E: $y^2 + xy = x^3 + ax^2 + b$,
$P = (x_0, y_0)$ e
$Q = (x_1, y_1)$;

então,

$$P + Q = R = (x_2, y_2),$$

onde

$$y_2 = (x_1 + x_2)\theta + x_2 + y_1 \text{ e}$$

$$x_2 = a + \theta^2 + \theta + x_0 + x_1$$

$$\theta = \begin{cases} \dfrac{y_0 + y_1}{x_0 + x_1}, & \text{se P} = \text{Q} \\ x_1 + \dfrac{y_1}{x_1}, & \text{se P} \neq \text{Q} \end{cases}$$

2.2.3 A Multiplicação Escalar *kP* sobre Curvas Elípticas

2.2.3.1 Introdução

Os esquemas de criptografia utilizando curvas elípticas baseiam-se fortemente na multiplicação escalar de um dado ponto *P* sobre a curva por um inteiro *k*. A multiplicação escalar é definida então como sendo *k* sucessivas adições de *P* sobre ele mesmo, ou seja:

$kP = P + P + ... P.$, k vezes.

A fim de ser útil aos métodos criptográficos, *k* deve ser um inteiro tão grande que sua obtenção por simples verificação de seus possíveis valores torne-se inviável; assim o cálculo de *kP* por sucessivas *k* adições de *P* é um processo impraticável.

No entanto, uma variedade de técnicas torna possível esta operação ser efetuada de maneira muito mais eficiente, enquanto que dado $Q = kP$, o problema de achar *k* continua sem solução sub-exponencial [4]. Dos diversos algoritmos existentes, uma amostra é apresentada a seguir, em ordem crescente de complexidade.

2.2.3.2 Método Binário

O método binário remonta a mais de 2000 anos atrás [14][38] e é derivado de uma observação simples do problema da multiplicação. Este método também é conhecido como método de duplicar e somar, constituindo em calcular *k*P utilizando-se da representação binária de *k,* onde k_i é 0 ou 1:

$$k = \sum_{i=1}^{l-1} 2^i k_i$$

O seguinte algoritmo calcula o *k*P utilizando-se desta observação:

Algoritmo 2.1

1. *Entrada*: $k \in \mathbb{Z}$ na representação binária e P um ponto sobre a curva
2. *Saida*: kP
3. Se $k = 0$ então retorne $\boldsymbol{O}_\infty$
4. $R \leftarrow P$
5. De $i = l - 2$, enquanto $i \geq 0$, com i- -, faça
   5.1 $R \leftarrow 2R$
   5.2 Se $k_i = 1$, então $R \leftarrow R + P$
6. retorne R

Cada iteração o algoritmo acima calcula $R = sP$, onde s é um prefixo da representação binária de k [14]. O algoritmo acima pode ser modificado para calcular o produto kP avaliando k da direita para esquerda. Esta abordagem possui, entretanto, uma desvantagem pois necessita de uma variável para armazenar os valor 2^iP, mostrado no algoritmo 2.2:

Algoritmo 2.2

1. *Entrada*: k na representação binária e P um ponto sobre a curva
2. *Saida*: kP
3. Se $k = 0$ então retorne $\boldsymbol{O}_\infty$
4. $S \leftarrow P$
5. $R \leftarrow \boldsymbol{O}_\infty$
6. De $i = 0$, enquanto $l - 2 \geq i$, com i++, faça
   6.1 Se $k_i = 1$, então $R \leftarrow R + S$
   6.2 $S \leftarrow 2S$
7. retorne $R + P$

Apesar de bastante simples, o método binário requer no pior caso $\log_2 k$ somas elípticas e $\log_2 k - 1$ duplicações. Já no caso médio são necessárias $\log_2 k/2$ somas, pois na média espera-se que metade dos bits sejam 0 e a outra metade 1, e $\log_2 k - 1$ duplicações. Este método representa uma melhora muito grande sobre o método direto.

Algumas considerações sobre implementação em hardware podem ser feitas sobre este método. A primeira é que ele necessita de recursos bastante modestos para ser efetuado, ou seja, é possível implementar este algoritmo dado uma unidade de soma de pontos elípticos, com apenas um registrador de deslocamento de m bits, onde m é o tamanho do corpo, algumas posições de memória para se guardar um valor intermediário e circuitos para controle.

2.2.3.3 O Método *m - ário*

O método *m – ário* é uma generalização natural do método binário. Ao invés de utilizar-se da base 2 na representação de *k*, pode-se utilizar uma base *m* arbitrária. Assim a representação de *k* torna-se:

$$k = \sum k_i m^i, \text{ onde } k_i \in \{0,1,...m-1\}$$

O cálculo de *k*P então pode ser realizada com o algoritmo a seguir:

Algoritmo 2.3

1. *Entrada*: $m = 2^r$, *k* em sua forma apresentada anteriormente e P
2. *Saida: k*P
3. Se $k = 0$ então retorne $\boldsymbol{O}_\infty$
4. $R \leftarrow k_{l-1}P$
5. De $i = l - 2$, enquanto $i \geq 0$, com *i*- -, faça
   5.1. $R \leftarrow mR$
   5.2 Se $k_i \neq 0$ então $R \leftarrow R + k_iP$
6. Retorne *R*

Evidente que com $m = 2$, o algoritmo *m – ário* reduz-se ao binário. Se na linha 5.1, *m* for alguma potência inteira de 2, então *R* pode ser obtido com *r* duplicações de P, fato que sugere a utilização de $m = 2^r$. A maior vantagem deste método sobre o método binário é o aumento da velocidade de multiplicação, através da pré-computação de k_iP para os $m - 1$ valores distintos não nulos de k_i.

Uma análise de seu desempenho mostra que o número de operações deste algoritmo no pior caso, considerando $m = 2^r$, é dado por $\log k - 1$ duplicações e $\log k / r$ somas durante a execução e $m - 2$ adições no pré-cálculo para a construção da tabela de *m* posições.

É fácil notar que o método 2^r *– ário* representa uma solução de compromisso entre espaço e velocidade. Supondo, para o caso específico de implementações em hardware, que a velocidade de *doubling*[10], seja igual a velocidade de adição de dois pontos distintos, temos que o número de operações necessárias *n*, com a adição elíptica sendo a unidade, será:

$$n = \lceil \log_2 k \rceil - 1 + \lceil \log_2 k / \log_2 m \rceil + m - 2$$

Esta função claramente não é monótona, possui um mínimo global, que podemos achar através da resolução:

[10] Chama-se de *doubling* de pontos sobre curvas elípticas a multiplicação de um ponto P da curva por 2, ou seja, 2 P.

$\frac{\partial n}{\partial m} = 0 = 1 - \frac{\log k}{m \log^2 m}$, função transcendental.

O gráfico a seguir mostra o número de operações *n* para *k* definido em um corpo 263 de bits e com *m* variando entre 2 (caso binário) até 64:

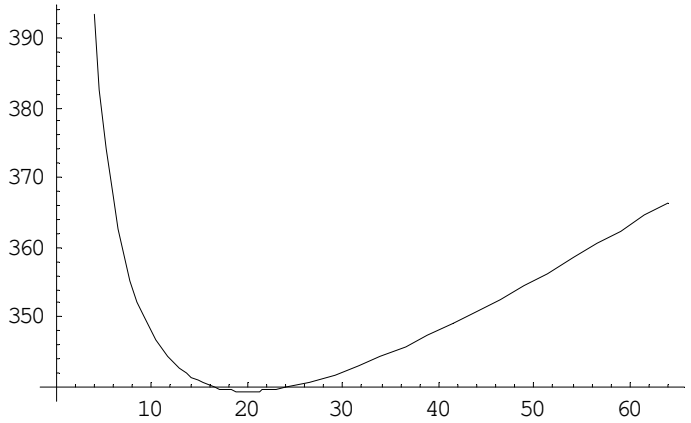


Figura 2.4 – Ciclos versus *m* para GF(2^{263}) – Comportamento do crescimento de *m*.

É possível notar que para este caso, o valor de $m = 16$ é extremamente interessante, permitindo que o número de adições no pior caso caia de 522, no método binário, para 341, ou seja, algo em torno de 66%.

Em termos de sua implementação, o método 2^r – *ário* é interessante, pois aos requisitos do método binário somam-se apenas *m* posições de memória e uma unidade de controle moderadamente mais complicada.

Por outro lado, a necessidade de *m* posições de memória pode tornar-se um problema em implementações para ambientes restritos em termos de memória. Por exemplo, para um corpo de 173 bits e com $m = 16$ (que representa o melhor desempenho para o algoritmo), são necessários, ao menos, 5 Kbits de memória, valor modesto. Já para um corpo de 263 bits, um valor de $m = 16$ (ainda o melhor valor para, ignorando-se qualquer restrição de espaço), são necessários ao menos 8 Kbits de memória.

Para ambientes restritos, como microcontroladores e pequenas arquiteturas reconfiguráveis tal utilização de memória é proibitiva. Com esta motivação, algumas versões diferentes do algoritmo foram propostas, dentre elas uma que pré-computa apenas os valores k_iP quando k_i for um número ímpar e usa a relação $a_i 2^t h_i$, com t entre 0 e *r* e h_i ímpar se a_i for par. Deste modo, o número de pontos a serem armazenados pode ser reduzido pela metade.

Diversas outras modificações deste método foram propostas [14] e apresentam graus diferentes de melhorias à velocidade em detrimento à complexidade destas implementações. Uma delas por meio do esquema de Janelas Deslizantes, por exemplo, reduz o número de somas de acordo com o valor de *a*, que fica:

$\lceil \log_2 a \rceil / (r + 1)$

e representa, para um corpo de tamanho 173, um tempo de execução de 89% daquele dado pelo método *a – ário.*

Em termos de implementação em hardware, esta opção não parece muito interessante, pois requer uma complexidade consideravelmente maior da unidade de controle, para um ganho não muito expressivo: podemos utilizar estes recursos para aumentar, por meio de paralelismo, a velocidade de outras operações necessárias à multiplicação escalar em algumas vezes. A melhor opção pode ser então o controle dos circuitos que implementam o algoritmo através de micro-programação ou mesmo via software.

Capítulo 3
Primitivas Criptográficas

Este capítulo apresenta as primitivas criptográficas abordadas nesta dissertação. Na seção 3.1 são apresentados os algoritmos AES e DES além de algumas considerações sobre segurança. A seção 3.2 discorre brevemente sobre o uso de criptografia assimetria. Funções hash e noções sobre aleatoriedade são apresentadas nas seções 3.3 e 3.4 respectivamente.

3.1 Criptografia Simétrica – Cifradores por Blocos

3.1.1 Descrição Geral

3.1.1.1 Introdução

Os cifradores de chaves simétricas por blocos são um dos principais tijolos básicos para sistemas criptográficos. Seu objetivo primeiro é o de garantir confidencialidade da informação, mas sua versatilidade permite uma série de outras utilizações. Geradores de seqüências pseudo-aleatórias, cifradores de *streams*, funções *hash* e *message authentication codes – MACs* são apenas algumas de suas possíveis aplicações [42].

Vários *block ciphers* (cifradores por blocos) apresentam diferentes *níveis de segurança* e nem todos estes cifradores são adequados para todas as aplicações: diferentes compromissos entre velocidade, tamanho de implementação e níveis de segurança são possíveis [54].

Se κ denotar o espaço de chaves de k bits, V_n todos os possíveis vetores de n bits, p o texto original e c o texto cifrado então, podemos definir:

Definição [42]: Um cifrador por blocos é uma função um-para-um $E : V_n \times \kappa \rightarrow V_n$, de tal forma que para cada chave $K \in \kappa$, E(K, p) (a função de cifragem por K) é um mapeamento inversível de p de V_n para V_n, denotado $E_K(p)$. Se $c = E_K(p)$, então o mapeamento inverso é a função de decifragem, denotada por $D_K(c)$, de tal sorte que $p = D_K(c)$.

A geração atual de cifradores [39] [55] [56] [57] utiliza blocos de 128 bits [54] enquanto suporta chaves entre 128 e 256. Note que a definição de cifrador por blocos não requer que o tamanho dos blocos sejam iguais ao tamanho das chaves.

Para pequenas mensagens ($\leq n$ bits) um cifrador por blocos pode ser utilizado diretamente[11] para garantir confidencialidade. No entanto, para mensagens maiores a utilização de algum *modo de operação* é necessário. Modos de operação serão discutidos mais adiante.

[11] Naturalmente uma série de cuidados devem ser tomados, como troca de chaves, unicidade da mensagem e outros.

3.1.1.2 Níveis de Segurança e Tipos de Ataques

Primeiro introduziremos um princípio importante, tal qual está em [54]

Princípio de Kerckhoff : o nível de segurança de um sistema criptográfico deve somente depender do desconhecimento por terceiros da chave de cifragem e não dos algoritmos envolvidos.

Com poder computacional suficiente, um sistema criptográfico pode de maneira geral[12] ser quebrado. A quantidade de esforço despendido em tal tentativa é que define o nível de segurança do sistema. Pode-se dizer, que se para quebrar um sistema são necessários 2^n passos, então isto corresponde à procura exaustiva em um espaço de 2^n bit.

Um *passo*, como utilizado anteriormente, pode significar uma cifragem completa ou uma decifragem completa por um dado mecanismo. Algumas vezes um passo pode ser realizado atomicamente, outras vezes este passo pode necessitar de milhares de ciclos de máquina.

Atualmente, e pelo menos pelos próximos 50 anos [54] [6], um sistema com nível de segurança de 128 bits (3.4×10^{38} passos necessários) é adequado. No entanto, não é fácil conseguir tal nível de proteção com mecanismos com chaves de 128 bits, por exemplo, pois existe uma série de possíveis ataques que minam os níveis de segurança, em especial aqueles ligados com o paradoxo do "Dia do Aniversário" [38].

Alguns ataques são listados abaixo:

- *Apenas Texto Cifrado*. Neste ataque somente o texto cifrado está disponível para o invasor, e é o tipo de ataque mais difícil de ter sucesso já que a menor quantidade possível de informação está disponível.
- *Texto Conhecido*. Aqui, o atacante conhece tanto o texto original (ou parte dele) e o texto cifrado. Este tipo de ataque é mais comum do que se possa imaginar, já que em transmissões de dados, os pacotes são comumente compostos de cabeçalhos fixos. Este ataque é mais fácil de ser implementado já que o invasor tem mais informações para serem analisadas.
- *Texto Conhecido Escolhido*. Neste ataque o atacante pode escolher o texto a ser cifrado, tornando o ataque ainda mais poderoso. Se quem analisa ainda puder escolher o valor daquilo que será cifrado baseado em resultados anteriores o ataque será conhecido como de *Texto Conhecido Escolhido Adaptativo*.
- *Texto Original e Cifrados Escolhidos*. Ainda mais poderoso podendo ser *Adaptativo*.
- *Do Paradoxo do "Dia do Aniversário"*. Baseados no paradoxo do Dia do Aniversário, consistem no fato de que *colisões* acontecem muito freqüentemente [42]. Em um sistema utilizando chaves de *n* bits geradas aleatoriamente para cada transação, por exemplo, pode-se esperar que duas transações tenham a mesma chave após apenas $2^{n/2}$ transações terem ocorrido. De maneira geral, se um elemento pode assumir *N* valores, então esco-

[12] Para o *One Time Pad* esta quantidade de poder é infinita.

lhendo-se aleatoriamente entre estes valor (com reposição), esperar-se encontrar o mesmo elemento após algo como $\sqrt{N}$ escolhas.

- *Meet in the Middle (Encontro no Meio)*. Este tipo de ataque é uma variação do ataque pelo paradoxo do Aniversário. Nele, ao invés de esperar que $2^{n/2}$ escolhas sejam feitas, uma tabela de pré-computados ($> 2^{n/2}$) pode ser construída de forma que sejam necessárias menos transações em escuta.

3.1.1.3 Estrutura Básica

Cifradores por blocos são comumente construídos por meio de composições de funções menos seguras e mais simples. Usualmente, cifradores simples utilizando funções menos complexas não possuem segurança satisfatória contra criptoanálise. Tais cifradores mais simples podem ser classificados em termos de suas funções básicas em cifradores por substituição simples, homofônica e polialfabética e cifradores por transposição [42].

Funções de transposição e substituição[13] podem ser combinadas para formar um *round*. Os mais usados e seguros cifradores simétricos da atualidade funcionam por meio da aplicação de *rounds* sucessivos de transformações.

Tais *rounds* são especialmente interessantes em termos de implementações tanto em software quando em hardware. Se em software, eles permitem que menos código seja necessário para a sua implementação; em hardware adicionam *flexibilidade*, pois permitem que os conceitos de paralelismo espacial e temporal sejam explorados através de técnicas como *pipelining*, que será descrita adiante.

Além de estas implementações dependerem claramente do algoritmo em questão, elas ainda dependem de algumas considerações quanto aos seus modos de operação [37].

3.1.1.4 Modos de Operação e MACs

Cifradores por blocos operam sobre blocos de tamanhos fixos (e de tamanho normalmente reduzidos). Para cifrar mensagens de tamanho arbitrário, algum dos *modos de operação* dos cifradores deve ser utilizado, junto de algum mecanismo de *padding* para que a mensagem tenha tamanho múltiplo dos blocos. O modos de operação têm importância significativa no nível de segurança alcançado, uma vez que podem impedir que alguns tipos de ataques sejam efetuados, como aqueles utilizando o paradoxo do "Dia do Aniversário".

O NIST [37] recomenda cinco principais modos de operação, *Electronic Codebook* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB) e *Counter* (CTR). Alguns outros modos de operação que garantam não somente confidencialidade, mas também autenticidade, estão em estudo, mas ainda não foram padronizados.

[13] Uma substituição em um round adiciona o que se chama de *confusão*, enquanto uma transposição adiciona *difusão* ao round. A difusão essencialmente refere-se ao rearranjamento dos bits da mensagem e a confusão torna a relação entre a chave e a mensagem cifrada a mais complexa possível.

Cifradores podem ser utilizados ainda em mecanismos compostos, como o CBC-MAC e O-MAC de forma a constituírem funções de autenticação. Em ambos os casos a operação básica realizada é a de um cifrador em modo CBC, com modificações nos passos finais e ou iniciais.

Os funcionamentos dos cinco modos de operação são representados abaixo de maneira gráfica e foram retirados de [37].

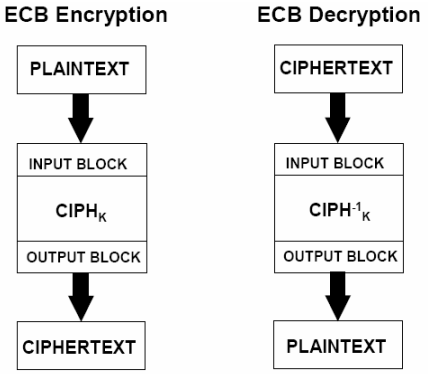


Figura 3.1 – Modo ECB – Possui nível de segurança inadequado para a maioria das aplicações. Funcionamento não encadeado.

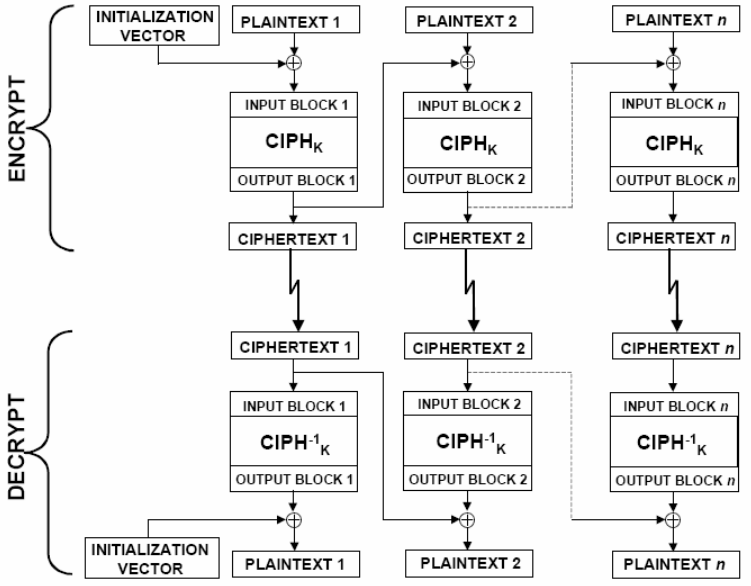


Figura 3.2 – Modo CBC – Provavelmente o modo de operação mais utilizado para os cifradores por blocos. Possui muito bom nível de segurança. Funcionamento encadeado.

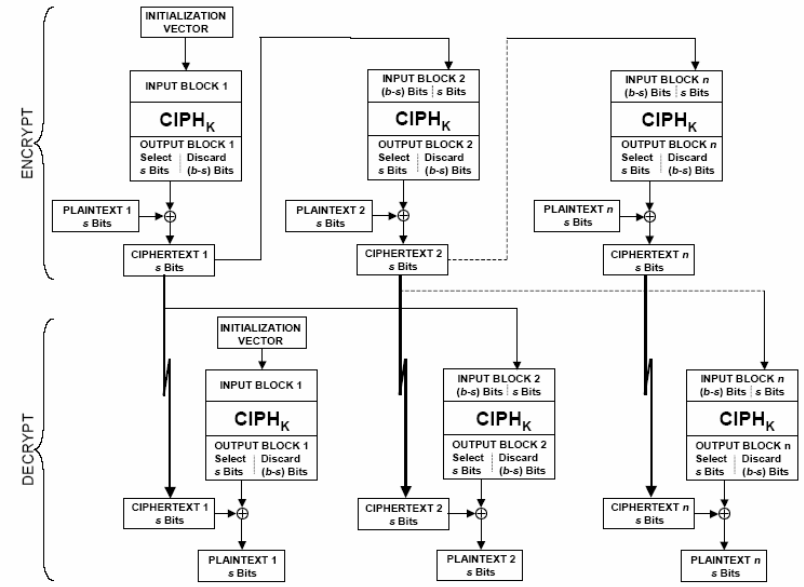


Figura 3.3 – Modo CFB – Adequado para blocos de dados de tamanhos diferentes do mecanismo base. Funcionamento encadeado.

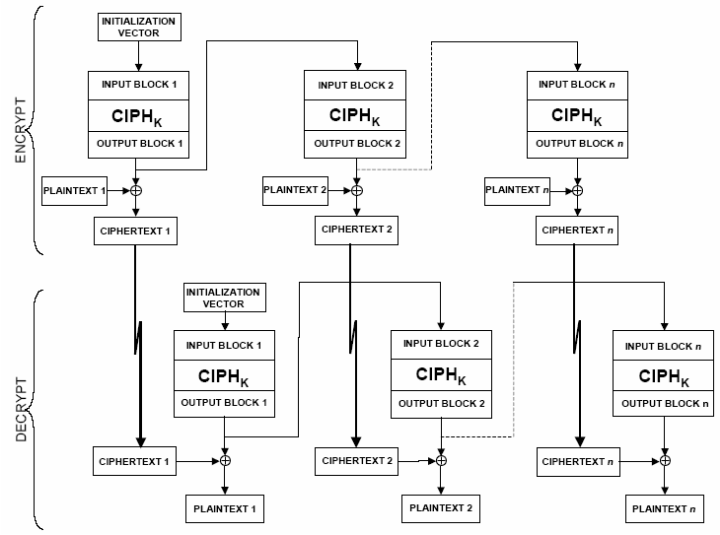


Figura 3.4 – Modo OFB – Possíveis ciclos no espaço na *keystream* resultam em um nível de segurança menor que o máximo possível [54]. Funcionamento encadeado.

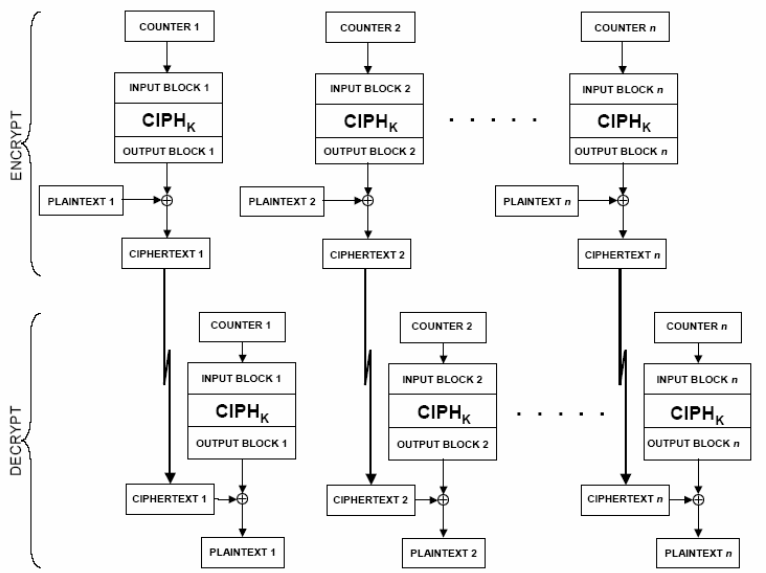


Figura 3.5 – Modo CTR – Bom nível de segurança, utilização sem necessidade de função de decifragem e funcionamento não encadeado.

Como se observa, os modos de operação podem ser subdivididos em dois conjuntos diferentes, aqueles encadeados (*feedback modes*) e os não encadeados (*non-feedback modes*). Na operação dos primeiros, uma mensagem somente pode ser processada após o cálculo do resultado (*ciphertext*) do bloco anterior, reduzindo as margens de paralelização. Já os demais (ECB e CTR) permitem a construção eficiente de *pipelines* e paralelização de seu funcionamento.

O encadeamento é importante fator na avaliação de performance de *hardware* criptográfico, pois influi fortemente na arquitetura utilizada para a construção do cifrador.

A seguir dois cifradores simétricos são introduzidos detalhadamente; o AES e o DES.

3.1.2 Advanced Encryption Standard – AES FIPS 197

3.1.2.1 Introdução

O algoritmo Rijndael [15], vencedor do concurso para o AES, foi proposto pelos belgas Vincent Rijmen e Joan Daemen e foi padronizado sob o padrão FIPS 197. Trata-se de um cifrador simétrico de blocos e de chaves de 128, 192 ou 256 bits. Várias operações do algoritmo são realizadas sobre corpos finitos binários $GF(2^8)$ com representação polinomial e outras sobre polinômios com coeficientes em $GF(2^8)$.

3.1.2.2 Operações Básicas Características

Para os elementos em $GF(2^8)$ do AES, na representação polinomial o polinômio irredutível utilizado é chamado *m(x)* e dado por:

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

ou '11B' em hexa. As duas principais operações realizadas nesta representação pelo algoritmo são a inversão de elementos e a multiplicação pelo polinômio *x*, referenciada como *xtime*. Neste caso se multiplicarmos um polinômio *b(x)* por *x*, teremos:

$$b_7 x^8 + b_6 x^7 + ... + b_1 x^2 + b_0 x.$$

assim, $x \bullet b(x)$ é obtido reduzindo o polinômio acima módulo *m(x)*. Se $b_7 = 0$, então esta redução é a operação identidade. Se $b_7 = 1$, *m(x)* precisa ser subtraído, através de um XOR. Assim a multiplicação por *x*, denotada por *xtime(b(x))* pode ser implementada em termos de byte, com as operações *shift left* e com um XOR condicional.

Alguns elementos do AES são interpretados como polinômios com coeficientes em $GF(2^8)$, em vetores de 4 bytes e representado um polinômio de grau menor que quatro. Polinômios desta forma, podem ser adicionados, pela simples adição modular de seus coeficientes. Como uma adição em $GF(2^8)$, esta operação pode ser realizada apenas com um XOR.
Já a multiplicação de dois termos, *a(x)* e *b(x)* é definida da maneira usual:

$$a(x)=a_3 x^3 + a_2 x^2 + a_1 x + a_0 \text{ e } b(x)=b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

O seu produto $c(x) = a(x)\, b(x)$ é dado por

$$c(x)= c_6 x^6 + c_5 x^5 + a_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \text{ com}$$
$$c_0 = a_0 \bullet b_0$$
$$c_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1$$
$$c_2 = a_2 \bullet b_0 \oplus a_0 \bullet b_2 \oplus a_1 \bullet b_1$$
$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$
$$c_4 = a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$
$$c_5 = a_3 \bullet b_2 \oplus a_2 \bullet b_3$$
$$c_6 = a_3 \bullet b_3$$

Claramente, *c(x)*, não pode ser mais representado com um vetor de 4 bytes. Pela redução de *c(x)* módulo um polinômio de grau 4, o resultado é um polinômio de grau menor que 4. No AES isso é feito com o polinômio $M(x) = x^4 + 1$ (redutível). Como

$$x^j \bmod x^4 + 1 = x^{j \bmod 4},$$

o produto modular de *a(x)* e *b(x)*, denotado por $d(x) = a(x) \otimes b(x)$ é dado por

$$d(x)= d_3 x^3 + d_2 x^2 + d_1 x + d_0 \text{ com}$$
$$d_0 = a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$

$$d_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3$$
$$d_2 = a_2 \bullet b_0 \oplus a_0 \bullet b_2 \oplus a_1 \bullet b_1 \oplus a_3 \bullet b_3$$
$$d_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$

Assim a operação consistindo da multiplicação por um polinômio fixo *a(x)* pode ser representada por uma multiplicação matricial, onde a matriz é uma matriz circulante:

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

Como *M(x)* não é um polinômio irredutível sobre GF(2^8), a multiplicação por uma polinômio fixo *a(x)* não é necessariamente inversível. No Rijndael foi escolhido um polinômio fixo que *tem* um inverso.

3.1.2.3 Especificação do Algoritmo

O Estado, a Chave do Cifrador e o Número de *Rounds*.

As diferentes transformações do AES operam sobre um resultado intermediário chamado *Estado*. O Estado pode ser visualizado como uma matriz retangular de bytes. Este *array* tem quatro linhas, o número de colunas é denotado por Nb e é igual ao tamanho do bloco (128, 192 e 256 bits) dividido por 32 (8 bits x 4 bytes). FIPS 197 requer somente que o bloco de 128 bits seja implementado. A Chave do Cifrador pode ser também visualizada como um array retangular com quatro linhas e com Nk colunas. Nk é igual ao tamanho da chave (128, 192 e 256 bits) divido por 32.

Em algumas instâncias, esses blocos também podem ser considerados vetores unidimensionais de 4 bytes. Os bytes de entrada do cifrador são mapeados nos bytes do Estado na ordem $a_{0,0}\ a_{1,0}\ a_{2,0}\ a_{3,0}\ a_{0,1}\ a_{1,1}$... Os bytes da Chave do Cifrador são mapeados de maneira similar.

Assim, se um índice unidimensional de um byte dentro do bloco é *n*, então os índices bidimensionais (i, j) são:

$$i = n \bmod 4; \quad j = \lfloor n/4 \rfloor; \qquad n = i + 4\,j$$

O número de *rounds*, denotado por Nr, depende tanto de Nb quanto de Nk, e é dado na seguinte tabela seguinte.

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

Tabela 3.1 – Número de iterações para o AES.

A Transformação do Round

A transformação do *round* é composta de quatro diferentes transformações. Em pseudo C, temos:

```
Round(State, RoundKey) {
     ByteSub(State);
     ShiftRow(State);
     MixColumn(State);
     AddRoundKey(State, RoundKey);
};
```

O *round* final é um pouco diferente:

```
FinalRound(State, RoundKey) {
     ByteSub(State);
     ShiftRow(State);
     AddRoundKey(State, RoundKey);
};
```

Nesta notação, cada uma das funções (SubByte, ShiftRow, MixColumn e AddRoundKey) operam sobre os apontadores dados pelo parâmetros. A seguir cada uma destas funções será detalhada.

A transformação ByteSub

A transformação ByteSub é uma substituição não linear, operando em cada byte do Estado independentemente. A tabela de substituições (ou S-box) é inversível e construída pela composição de duas transformações:

1- Primeiro, tomando o inverso multiplicativo do elemento em $GF(2^8)$, com a representação definida anteriormente. O valor ‘00’ é mapeado para ele mesmo.
2 – Então, aplicando a transformação afim em $GF(2^8)$ definida por:

$y = Mx + c$, onde:

y é o vetor coluna $[y_0, ..., y_7]^T$, x é o vetor coluna $[x_0, ..., x_7]^T$,
c é o vetor coluna $[1\ 1\ 0\ 0\ 0\ 1\ 1\ 0]^T$ e

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

A aplicação da S-box sobre todos os bytes do Estado é denotada por SubBytes(State). A inversa de SubBytes é a substituição onde a tabela inversa é aplicada. Esta tabela é obtida pelo inverso da transformada afim, seguido pela inversão do número em $GF(2^8)$.

A transformação ShiftRow

Nesta transformação, as linhas do Estado são roladas à esquerda. A linha 0 não é deslocada, a linha 1 é rolada em C_1 bytes, a linha 2 em C_2 bytes e a linha 3 em C_3 bytes.

Os valores para C_1, C_2 e C_3 dependem do tamanho do bloco Nb. Os valores são especificados na tabela a seguir:

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Tabela 3.2 – Deslocamentos
para a transformação ShiftRows
do AES.

A operação de deslocamento dos bytes do estado é denotada por SfhitRow(State). A operação inversa é a rolagem a direita dos mesmos C_1, C_2 e C_3 bytes das três últimas linhas.

A transformação MixColumn

Em MixColumn, as colunas do Estado são consideradas polinômios de grau menor que 4 sobre $GF(2^8)$, e multiplicadas por um polinômio fixo $c(x)$, módulo $x^4 + 1$, onde

$c(x) = \text{'03'}\ x^3 + \text{'01'}\ x^2 + \text{'01'}\ x + \text{'02'}$.

Este polinômio é primo relativo à $x^4 + 1$, e portanto inversível. Em forma matricial, esta transformação pode ser expressa como uma multiplicação. Seja $b(x) = c(x) \otimes a(x)$, então:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

A aplicação desta operação sobre todas as colunas do Estado é denotada por MixColumn(State). A transformação inversa é similar à direta: cada coluna é transformada pela multiplicação com um polinômio *d(x)* definido por:

$$\text{'03'}\ x^3 + \text{'01'}\ x^2 + \text{'01'}\ x + \text{'02'} \otimes d(x) = \text{'01'}.$$

Assim:

$$d(x) = \text{'0B'}\ x^3 + \text{'0D'}\ x^2 + \text{'09'}\ x + \text{'0E'}$$

A transformação de Adição de Chaves

Nesta transformação, uma chave de *round* é aplicada ao estado pela simples operação bit a bit de um XOR. A chave do *round* é derivada da chave do cifrador por meios da Derivação de Chaves. O tamanho da chave do *round* é igual ao tamanho do bloco do Estado.

A transformação que consiste na adição da chave do *round* ao round é denotada por AddRoundKey(State, RoundKey). E ela é sua própria transformação inversa.

3.1.2.4 A Derivação das Chaves

As chaves de *round* são derivadas da chave do cifrador. Esta derivação é constituída de duas componentes: a expansão da chave e a seleção da chave. O princípio básico é o seguinte: O número total de bits das chaves de round é igual ao tamanho do bloco multiplicado pelo número de *rounds* mais 1.

A chave do cifrador é expandida em uma Chave Expandida.

As chaves dos *rounds* são selecionadas da Chave Expandida da seguinte maneira: a primeira chave de round consiste nas primeiras Nb palavras de memória e assim por diante.

A Expansão da Chave

A chave expandida é um *array* linear de palavras de 4 bytes e é denotada por W[Nb(Nr+1)]. As primeiras Nk palavras contêm a chave do cifrador. Todas as outras palavras

são definidas recursivamente em termos de palavras com índices menores. A função de expansão depende do valor de Nk: há uma versão para Nk menor ou igual a 6 e outra para Nk maior que 6.

No primeiro caso:

```
KeyExpansion(byte Key[4*Nk], word W[Nb*(Nr+1)]) {
      for (i = 0; i < Nk; i++)
            W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]);
      for (i = Nk; i < Nb*(Nr+1); i++) {
            temp = W[i-1];
            if (i % Nk == 0)
                  temp = SubBytes(RotByte(temp)) ^ Rcon[i/k];
            W[i] = W[i-Nk] ^ temp;
      }
}
```

Nessa descrição SubBytes(W), opera a transformação S-box sobre a palavra de 4 bytes. A função RotByte(W) retorna uma palavra no qual os bytes são ciclicamente permutados em relação aos originais, tal que (a, b, c, d) torna-se (b, c, d, a).

Pode-se observar que as primeiras Nk palavras são preenchidas com a chave do cifrado. Cada palavra W[i] seguinte é igual ao XOR das palavras W[i-1] e da palavra Nk posições atrás, W[i-Nk]. Para palavras em posições múltiplas de Nk, uma transformação é aplicada a W[i-1] antes do XOR e uma constante de round é adicionada por XOR. Esta transformação é constituída de uma rolagem (RotByte), seguida da aplicação de uma tabela em todos os quatro bytes da palavra (SubByte).

Para Nk > 6, temos:

```
KeyExpansion(byte Key[4*Nk], word W[Nb*(Nr+1)]) {
      for (i = 0; i < Nk; i++)
            W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]);
      for (i = Nk; i < Nb*(Nr+1); i++) {
            temp = W[i-1];
            if (i % Nk == 0)
                  temp = SubByte(RotByte(temp)) ^ Rcon[i/k];
            else if (i % Nk == 4)
                  temp = SubByte(temp);
            W[i] = W[i-Nk] ^ temp;
      }
}
```

As constantes de round são independentes de Nk e dadas por: Rcon[i] = (RC[i], ‘00’, ‘00’, ‘00’), com RC[i] representando um elemento em GF(2^8) com o valor $x^{(i-1)}$,

RC[1] = 1 = ‘01’

RC[i] = x • RC[i-1]

A Seleção da Chave de Round

A chave de round i é dada pelas palavras de memória do buffer de chaves W, entre os índices Nb*i e Nb*(i+1).

3.1.2.5 O Cifrador e o Decifrador.

O Cifrador AES em pseudo C é constituído de:

```
Rijndael (State, ChiperKey) {
      KeyExpansion(ChiperKey, ExpandedKey);
      AddRoundKey(State, ExpandedKey);
      For (i=1; i<Nr; i++) {
            SubBytes(State);
            ShiftRow(State);
            MixColumn(State);
            AddRoundKey(State, ExpandedKey + Nb*i);
      }
      SubBytes(State);
      ShiftRow(State);
      AddRoundKey(State, ExpandedKey + Nb*Nr);
}
```

Segundo [15], na implementação do cifrador é essencial que a transformação não linear SubBytes seja a primeira transformação em um *round* e que as linhas sejam deslocadas antes que a transformação MixColumn seja aplicada.

No cifrador inverso, a ordem destas transformações é também invertida e conseqüentemente o passo não linear acabará sendo o último. Além disso MixColumn será aplicado antes do deslocamento das linhas. Isso acaba impedindo que o cifrador inverso seja implementado com as tabelas de procura descritas anteriormente.

O AES^{-1} foi planejado de forma que, através de algumas transformações algébricas, seja possível torna-lo similar em estrutura ao cifrador direto. Em uma versão do Rijndael inverso de dois *rounds*, temos:

```
InvRound(State, RoundKey) {
      AddRoundKey(State, RoundKey);
      InvMixColumn(State);
      InvShiftRow(State);
      InvSubBytes(State);
}
```

e o round final invertido:

```
InvFinalRound(State, RoundKey) {
      AddRoundKey(State, RoundKey);
      InvShiftRow(State);
      InvSubBytes(State);
}
```

O inverso de uma variante de dois *rounds* do AES consiste do inverso do *round* final seguido por um *round* invertido e finalmente uma adição de chave. Assim:

```
AddRoundKey(State, ExpandedKey +  2*Nb);
InvShiftRow(State);
InvSubBytes(State);
AddRoundKey(State, I_ExpandedKey + Nb);
InvMixColumn(State);
InvShiftRow(State);
InvSubBytes(State);
AddRoundKey(State, ExpandedKey);
```

Mas pode-se fazer melhor. Primeiro observe que as ordens de ShiftRow e SubBytes são indiferentes, uma vez que a primeira simplesmente transporta bytes de um lugar para outro e a segunda age sobre os valores individuais dos bytes, independente de sua posição.
Ainda mais, a seqüência:

```
AddRoundKey(State, RoundKey);
InvMixColumn(State);
```

pode ser mudada para:

```
InvMixColumn(State);
AddRoundKey(State, InvRoundKey);
```

com *InvRoundKey* sendo obtida de *RoundKey* por meio da aplicação de MixColumn. Isso é baseado no fato que estas transformações são lineares.

Usando as propriedades descritas é possível reescrever a versão de dois *rounds* do Rijndael:

```
AddRoundKey(State, ExpandedKey +  2*Nb);

InvByteSub(State);
InvShiftRow(State);
InvMixColumn(State);
```

```
AddRoundKey(State, I_ExpandedKey + Nb);

InvByteSub(State);
InvShiftRow(State);
AddRoundKey(State, ExpandedKey);
```

Pode-se observar que o cifrador inverso é similar em estrutura com o cifrador direto. Estas modificações podem ser generalizadas para qualquer número de *rounds*. Desta forma o cifrador inverso fica, em pseudo-C:

```
I_Round(State, I_RoundKey) {
      InvSubBytes(State);
      InvShiftRow(State);
      InvMixColumn(State);
      AddRoundKey(State, I_RoundKey);
}

I_FinalRound(State, I_RoundKey) {
      InvSubBytes(State);
      InvShiftRow(State);
      AddRoundKey(State, I_RoundKey);
}

I_AES(State, CipherKey) {
      I_KeyExpansion(CipherKey, I_ExpandedKey);
      AddRoundKey(State, I_ExpandedKey + Nb*Nr);
      for (i = Nr-1; i > 0; i--)
            I_Round(State, I_ExpandedKey + Nb*i);
      I_FinalRound(State, I_ExpandedKey);
}

I_KeyExpansion(CipherKey, I_ExpandedKey) {
      KeyExpansion(ChiperKey, I_ExpandedKey);
      for (i = 0; i < Nr; i++)
            InvMixColumn(I_ExpandedKey + Nb * i);
}
```

Uma implementação do cifrador AES não realiza automaticamente seu inverso, uma vez que os inversos das transformações envolvidas são normalmente distintas. Em InvMixColumn, por exemplo, a utilização das caixas *tms* fica impedida, pois os coeficientes necessários à transformação são diferentes.

Mesmo as transformações S-box não podem ser diretamente aproveitadas, uma vez que não são auto inversíveis. A implementação do cifrador inverso é de maneira geral mais lenta e consome mais tempo em *software* e mais recursos em *hardware*. Este problema foi adiantado no *design* do algoritmo, levando em conta que em diversas aplicações para um cifrador em bloco, seu inverso não é utilizado, como por exemplo no modo de operação CTR.

3.1.3 Data Encryption Standard – DES FIPS 46-2

3.1.3.1 Introdução

O algoritmo de cifragem DES (*Data Encryption Standard*) [35] já foi (e talvez ainda seja) o algoritmo de criptografia mais amplamente utilizado no mundo. Durante muitos anos criptografia e DES foram sinônimos. Atualmente, apesar da existência de máquinas especialmente criadas para quebrá-lo por força bruta, o DES continua sendo usado em sua versão mais segura, o Triple-DES [45].

O DES nasceu nos laboratórios da IBM com o nome de LUCIFER mas somente após uma requisição da NBS (National Bureau of Standards) por algoritmos de criptografia e após testes e algumas modificações sugeridas pela NSA (National Security Agency), ele foi tornado o novo algoritmo de criptografia padrão em 15 de julho de 1977, tornando-se o atual DES [42].

O DES trabalha bit a bit, sobre mensagens de 64 bits (16 hexa). Para o seu funcionamento ele utiliza uma chave com 64 bits dos quais apenas 56 são utilizados.

A seguir, o DES será detalhadamente apresentado. Para tanto, diversas tabelas serão utilizadas, que refletem a concepção do algoritmo.

3.1.3.2 A Geração de Chaves

O primeiro passo usado pelo DES no processamento é o da criação de 16 subchaves de round derivadas da chave inicial K. Assim, dos 64 bits de K, 56 são escolhidos de acordo com a tabela PC-1 para formar a chave permutada K+, como mostrado na Tabela 3.3. O primeiro bit a esquerda de K torna-se o qüinquagésimo sétimo de K+, o segundo bit de K o quadragésimo nono de K+ e assim por diante.

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Tabela 3.3 – PC-1, permutação para o DES.

Então dividimos K+ em duas metades, C_0 e D_0, de 28 bits cada. Definidas as duas metades, rolamos cada uma à esquerda de acordo com a tabela de rolagem (Tabela 3.4), criando 16 pares $\{C_1, D_1\}$, $\{C_2, D_2\}$, ..., $\{C_{16}, D_{16}\}$. A tabela seguinte mostra o número de deslocamentos (s) para cada número de iteração (r).

r	*s*
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Tabela 3.4 – Deslocamentos
para a chave do DES em bits.

Isto significa que o par C_1, D_1 é obtido partindo-se de C_0, D_0 através de uma rolagem à esquerda, e que C_3, D_3 é obtido de C_2, D_2 após duas rolagens. Assim teremos 16 pares C_n, D_n.

Cada par então é justaposto para formar um bloco C_nD_n que depois tem seus bits permutados de acordo com a Tabela PC2 para formar as 16 subchaves de 48 bits. Deve-se notar que apesar de cada par C_nD_n ter 56 bits, PC2 utiliza apenas 48. Assim o primeiro bit de à esquerda de K_n é igual o décimo quarto de C_nD_n e assim por diante:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Tabela 3.5 – PC-2, permutação
dos bits para as subchaves.

3.1.3.3 A Mensagem

O segundo passo no entendimento do DES é a codificação de um bloco de 64 bits de dados, a mensagem M. Em primeiro lugar a mensagem M de 16 dígitos hexa passa por uma permutação inicial, com o objetivo de embaralhá-la e transformá-la na mensagem permutada IP,

de acordo com a Tabela 3.6. De acordo com esta tabela o qüinquagésimo oitavo bit de M torna-se o primeiro bit de IP. O qüinquagésimo bit de M torna-se o segundo bit de IP e assim por diante.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Tabela 3.6 – Permutação Inicial IP

Em seguida dividimos IP em duas metades de 32 bits L_0 e R_0, e então procedemos através de 16 interações, usando uma função *f* cujas variáveis são um bloco de dados de 32 bits e uma subchave K_n de 48, que produz um resultado *f* de 32 bits. Assim com *n* indo de 1 a 16 fazemos:

$L_n = R_{n-1}$
$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$, onde $\oplus$ denota XOR.

Isto resulta que em cada interação o lado esquerdo L_n é igual ao lado direito R_{n-1} anterior e que o lado R_n é igual ao resultado do XOR entre o lado esquerdo anterior e o resultado da função *f*.

Resta-nos explicar o funcionamento de *f*. Seja E a seguinte tabela (Tabela 3.7), que a partir de uma entrada de 32 bits, utilizando repetição de alguns, fornece uma saída de 48. Então para a obtenção de R_n fazemos $E(R_{n-1})$, obtendo um bloco 48 bits.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Tabela 3.7 – Seleção de Bits E.

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Tabela 3.8 – As oito tabelas S

Como próxima etapa, calculamos o valor $K_n \oplus E(R_{n-1})$, obtendo um resultado de 48 bits. Neste instante precisamos reduzi-los à 32, utilizando as tabelas E e S: dividimos os 48 bits em oito grupos de seis e para cada grupo *n*, da esquerda para a direita, utilizamos uma das tabelas S_n (3.8).

As tabelas S_n tem 6 bits de entrada e fornecem 4 bits de saída cada, que concatenados na forma $S_1S_2S_3S_4S_5S_6S_7S_8$ resultam em um bloco de 32 bits.

Para utilizá-las deve-se proceder da seguinte maneira: sejam B os seis bits de entrada da tabela S_n. Então concatenando-se o primeiro e o ultimo bit e lendo-os na sua forma decimal correspondente, temos um número, no intervalo de 0 a 3, que indica a linha de S_n a utilizar. Já os quatro bits internos, indo de 0 a 15, informam a coluna selecionada. Importante lembrar que a primeira linha e a primeira coluna são a linha zero e a coluna zero.

O Estágio final para o cálculo de *f* consiste na permutação, dada pela tabela P, dos bits fornecidos pelas tabelas S:

$f = P(S_1S_2S_3S_4S_5S_6S_7S_8)$.

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Tabela 3.9 – Permutação P

Finalmente após as 16 interações obtemos os dois blocos L_{16} e R_{16} que concatenados na forma $R_{16}L_{16}$ fornecem um bloco de 64 bits. Estes então sofrem uma permutação final IP^{-1}, definida pela Tabela 3.10.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Tabela 3.10 – Permutação final IP^{-1}

O Resultado é o texto cifrado C. Para decifrá-lo, basta proceder os mesmos passos descritos acima, mas em cada uma das interações utilizar as chaves K_n de forma decrescente.

3.1.3.4 Decifragem e a Análise do DES

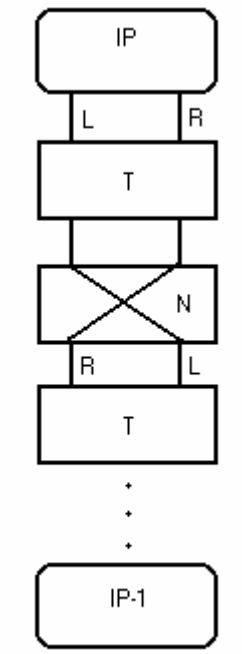


Figura 3.6 – Esquema gráfico do DES

A Figura 3.6 confere-nos uma visão gráfica do DES e permite-nos escrever:

DES = IP^{-1} . T_{K16} . N . T_{K15} . ... . N . T_{K1} . IP, onde N é simplesmente a troca de lados das metades e T é a transformação pela qual as metades passam antes de N. A combinação N . T_{Kn} então resulta em:

$L_n = R_{n-1}$

$R_n = L_{n-1} \oplus f(R_{-n-1}, K_n)$

É fácil notar que N é uma involução, assim N^2 é uma identidade.

A transformação T opera $f \oplus$ L para o lado esquerdo, mas não altera o conteúdo da metade do lado direito. Independente do que seja f podemos mostrar que T também é uma involução:

$T(L, R) = (L \oplus f(R + K), R)$ e assim

$T^2(L, R) = T(L \oplus f(R, K), R) = (L \oplus f(R, K) \oplus f(R, K), R) = (L, R)$

Isso nos permite escrever $DES^{-1} = IP^{-1}$. T_{K1} . N . T_{K2} . ... . N . T_{K16} . IP, ou seja, o decifragem pelo DES é basicamente igual à cifragem, apenas invertendo-se a ordem no qual as chaves são utilizadas.

3.2 Criptografia Assimétrica

3.2.1 Descrição Geral

Em sistemas de criptografia por chaves assimétricas cada entidade "A" tem um par de chaves composto de uma chave pública *e* e de uma chave privada *d*, onde a chave pública é derivada da privada. Em um sistema seguro a chave *d* não pode ser derivada da chave *e*.

A chave pública *e* define a transformação de cifragem E_e enquanto a chave privada *d* define a transformação inversa D_d, a decifragem. O principal objetivo dos mecanismos de criptografia de chaves públicas é a confidencialidade, mas em conjunto com outras técnicas, ela pode ser utilizada em serviços de autenticação e usada na construção de protocolos para estabelecimento de chaves.

A chave pública pode ser amplamente divulgada enquanto a chave privada deve ser mantida em segredo. Esta característica permite que um número linear de chaves precise ser compartilhado em um ambiente onde todos podem se comunicar com os demais. Em contraposição, no caso simétrico, um número exponencial de chaves é necessário [42][54].

Dois dos principais representantes de esquemas criptográficos assimétricos são o RSA [46] que se baseia no problema da Fatoração de Números Inteiros e o ElGamal Generalizado [44] que se baseia no problema do Logaritmo Discreto sobre algum grupo G finito e cíclico. Este grupo pode ser dentre outros o usual $\mathbb{Z}_p^*$ ou aquele representado pelos pontos de uma curva elíptica sobre um corpo finito [42]. Operações nestes grupos são usualmente lentas para serem realizadas em processadores de uso geral, uma vez que a aritmética envolvida raramente pode ser completamente expressa em termos de operações inteiras de processadores de uso geral.

Desta forma, esquemas assimétricos são tipicamente dezenas de vezes mais lentos que mecanismos simétricos e normalmente requerem chaves consideravelmente maiores. Por este motivo são comumente utilizadas para o transporte das chaves que serão posteriormente utilizadas pelos mecanismos simétricos.

3.2.2 Operações de ECC

Os algoritmos de ECC podem ser usados para criptografia assimétrica utilizando o esquema de ElGamal Generalizado[42]. Assim, o primeiro passo necessário ao estabelecimento da comunicação é o da geração das chaves públicas e privadas daqueles que querem participar do esquema. Em um primeiro momento cada uma das entidades cria um par composto de uma chave pública e uma privada, executando os seguintes passos:

1. Selecionar uma curva elíptica apropriada E sobre um corpo finito definido, formando um grupo de ordem n, com um ponto gerador sobre a curva;

2. Selecionar um número inteiro aleatório a entre 1 e n-1 inclusive.

3. Calcular aP, ou seja o produto do número a e o ponto P sobre a curva, de acordo com a definição de multiplicação escalar dada anteriormente;

4. A chave pública é (E, P e aP), e a chave privada da entidade é então o número a. Neste ponto, a tripla pode ser publicada em um diretório confiável.

Os algoritmos de Cifragem e Decifragem propriamente ditos ficam então: Sejam duas entidades "A" e "B", onde "B" quer se comunicar com "A". Considere ainda a mensagem m a ser enviada. Então:

(a) - Cifragem:

1. A mensagem m de "B" deve ser dividida em um par m_1, m_2, cujos elementos formam um par ordenado e pertencem ao corpo finito escolhido. Existem técnicas para dividir a mensagem m em duas partes a fim de que seus tamanhos tenham o mesmo tamanho do corpo e que isso ainda não implique no enfraquecimento do esquema;

2. "B" deve obter a chave pública de "A", possivelmente através de um diretório público certificado, composta de (E, P, a P);

3. Então "B" escolhe um número k aleatório entre 1 e n-1 inclusive;

4. Em seguida, "B" deve calcular kP, a multiplicação escalar de P por k e $k(a\text{P}) = (x, y)$;

5. Finalmente "B" envia o texto cifrado, composto de $c = (k\text{P}, xm_1, ym_2)$ para A. O primeiro elemento da tripla é um ponto na curva e os dois seguintes são elementos definidos sobre o corpo finito escolhido.

(b) - Decifragem:

1. "A" deve usar a sua chave privada a e calcular $a(k\text{P}) = ka\text{P} = (x, y)$;

2. "A" calcula então os elementos inversos x^{-1} e y^{-1};

3. Finalmente "A" recupera a mensagem original, através do calculo de xm_1x^{-1} e ym_2y^{-1}, finalizando o processo de decifragem.

Protocolos diferentes podem ser implementados utilizando ECC. No entanto, como trata-se da tarefa de mais alto nível dentro do esquema criptográfico, e que portanto envolve mais atividades de controle do que de processamento algébrico, estes protocolos podem ser implementados em software com as primitivas acelerada por hardware criptográfico. Esta abordagem é a que foi adotada nas arquiteturas implementadas.

As duas operações mais evidentes no protocolo acima são a multiplicação escalar aP e a multiplicação de elementos do corpo. Como anteriormente observado a primeira operação é inteiramente expressa em termos de adições e multiplicações no corpo finito base.

Desta forma foram alvo de estudo de arquiteturas as operações básicas envolvidas com evidente ênfase nos cálculos dos produtos em $GF(2^m)$.

3.2.3 Escolha dos Tamanhos e Tipos dos Corpos

3.2.3.1 Considerações

Da escolha do tipo e interpretação do corpo finito utilizado para a criptografia de curvas elípticas dependem mais técnicas de multiplicação, inversão e velocidade de processamento do que segurança propriamente dita.

Por sua vez, a escolha do tamanho do corpo é fundamental pois define fortemente a segurança do esquema, sua velocidade de funcionamento e finalmente o tamanho da implementação. Corpos maiores significam, de maneira geral, mais segurança, menor velocidade de cifragem e maiores circuitos, de maneira mais ou menos proporcional.

No entanto, características importantes dos corpos, como por exemplo, o número necessário de multiplicações para inversão (se usado o método geral descrito anteriormente) e a possibilidade de economia no cálculo dos bits c do produto $c = a.b$, motivam a escolha cuidadosa do tamanho m do corpo.

Além disso, a escolha da representação utilizada é importante. Devido ao fato de que a representação em Bases Normais rende implementações fixas para os coeficientes do produto para um dado tamanho de corpo, e devido ao fato de que tamanhos diferentes de corpos podem aproveitar se da mesma estrutura, utilizaremos esta representação. Se por um lado bases normais podem não ser a maneira mais eficiente para cálculos completamente paralelizados em pequenos corpos (4 – 32 bits), elas são aquelas de viabilidade mais evidente para corpos de interesse criptográfico [50].

3.2.3.2 Segurança e o Tamanho *m*

Não faz sentindo estabelecer um canal seguro utilizando-se um nível de segurança muito menor no processo de estabelecimento do que aquele que será utilizado ao longo do tempo na transmissão massiva de dados, pois se um invasor for capaz de mais tarde decifrar as mensagens transmitidas na fase inicial, ele detém também a chave do algoritmo de cifragem simétrico, rendendo a segurança do sistema.

Por isso, o nível de segurança nesta fase deve ser equivalente àquele oferecido pelo algoritmo de criptografia por chaves simétricas. Neste ponto surge uma pergunta bastante natural. Qual seria o nível de segurança aceitável para o último?

A resposta para esta questão é difícil de se precisar e depende claramente da vida útil daquilo que se quer proteger e da motivação do adversário. O DES, o antigo padrão de criptografia, ao menos em sua forma normal, apresenta-se não seguro. Sua chave de 64 bits (dos quais apenas 56 são usados) não é mais um desafio considerável; foi quebrado diversas vezes por ataques de força bruta em períodos pequenos de tempo [59].

No entanto, sua versão modifica, o Triple-DES, apresenta chaves de criptografia de 112 bits e é considerado uma opção forte (ainda que pelo paradoxo do dia do aniversário possa haver colisões em 2^{56} passos). O AES, por sua vez, tem chaves de três possíveis tamanhos, 128, 192 e 256 bits [15]. Apesar de ser teoricamente possível, com suficiente trabalho computacional, e informações, quebrar este tipo de criptografia, podemos construí-la de forma que o trabalho computacional necessário seja na prática inviável. Um "algoritmo de criptografia que tenha nível de segurança de 128-bits, necessita de um computador de US$ 7.5 x 10^{25} para ser quebrado em um ano" [6] por meio de força bruta.

Baseando-se em considerações semelhantes, podemos dizer que um nível de segurança para os algoritmos de ECC deve ser ao menos de 128. Por outro lado, números maiores que aquele equivalente ao de 256 bits para chaves simétricas podem render lentidão ao sistema.

A tabela abaixo compara os níveis de segurança para alguns cifradores e foi retirada de [11]:

Tamanho da Chave do Algoritmo de Chaves Simétricas	Exemplo de Algoritmo	Tamanho da chave Para ECC para Segurança semelhante	Tamanho da chave para RSA/DSA para segurança semelhante
80 bits	SKIPJACK	160	1024
112	Triple – DES	224	2048
128	128 – AES	256	3072
192	192 – AES	384	7680
256	256 – AES	512	15360

Tabela 3.11 – Segurança e Número e bits

Esta tabela é bastante ilustrativa. Mostra claramente a vantagem dos esquemas de ECC sobre o RSA e ainda mostra quão forte é a criptografia utilizando-se chaves simétrica. Ainda observado a tabela podemos chegar à seguinte conclusão: o tamanho *m* deve ser maior que 160 bits e não muito maior do que 512, admitindo que 128 bits seja o nível de segurança almejado. Por outro lado, implementações comerciais em hardware para computadores pessoais, utilizando interface PCI, usam corpos menores; o *Certicom Certilock* tem corpos na faixa de 113 a 239 bits.

3.2.2.3 As BNGs e a Facilidade dos Cálculos

Uma vez definida a faixa de números para no qual devemos procurar o tamanho de *m*, entre 160 e 512, precisamos observar que o algoritmo eficiente para multiplicação de elementos do corpo para bases normais binárias, requer que exista uma BNG para o corpo GF(2^m), por sua facilidade de cálculo. É possível mostrar que existem Bases Normais Gaussianas para todo *m* natural tal que $m \bmod 8 \neq 0$.

O último ponto a considerar antes da escolha do tamanho de m tem relação direta com a velocidade de processamento. O algoritmo IEEE P1363 A.3.8, a seguir calcula o produto de $c = a.b$. É fácil notar que o tempo de multiplicação é $O(m^2)$.

Algoritmo 5

Entrada: $a = (a_0 a_1 ... a_{m-1})$, $b = (b_0 b_1 ... b_{m-1})$
Saida: $c = (c_0 c_1 ... c_{m-1})$

Seja $F(a, b)$ a expressão com

$c_0 = F(a, b)$ (da seção 2.1.3.2)

Então o produto a x $b = c$ pode ser computado como se segue:

1. Para k de 0 até m-1 do
   1.1 Compute
   $c_k = F(a, b)$
   1.2 Faça $a \leftarrow$ LeftShift(a) e $b \leftarrow$ LeftShift(b), onde LeftShift denota o deslocamento circular a esquerda

2. Retorne $c = (c_0 c_1 ... c_{m-1})$.

A operação básica do esquema ECC é a multiplicação de pontos, a qual requer operações sobre os de elementos finitos. Felizmente todas estas operações, excetuando-se a soma, podem ser descritas em termos de multiplicações, inclusive a inversão.

Por outro lado, a inversão, operação obrigatória na soma de dois pontos elípticos, é composta de um número variável de multiplicações, dependente de m, mas nada linear, e determinado pela formula dada em 2.1.3.3. Se cada bit c_i do produto $c = a.b$ for calculado por ciclo, é possível estimar o tempo necessário para a inversão de um elemento do corpo. A Tabela 3.12 mostra estes custos para as mesmas BNGs.

m	M(m)	Ciclos	Tipo
146	10	1460	2
148	11	1628	1
155	11	1705	2
158	12	1896	2
162	10	1620	1
172	12	2064	1
173	11	1903	2
174	12	2088	2
178	11	1958	1
179	11	1969	2
180	12	2160	1
183	12	2196	2
186	12	2232	2
189	12	2268	2
191	13	2483	2
194	10	1940	2
196	11	2156	1
209	10	2090	2
210	11	2310	1
221	12	2652	2
251	13	3263	2
254	14	3556	2
261	10	2610	2
268	12	3216	1
375	14	5250	2
378	14	5292	1
386	11	4246	2
388	12	4656	1
393	11	4323	2
398	13	5174	2
508	16	8128	1
509	15	7635	2
515	11	5665	2
519	12	6228	2
522	12	6264	1
530	12	6360	2
531	12	6372	2
540	14	7560	1

Tabela 3.12 – Velocidades de Cálculo

3.2.2.4 O Valor para *m*.

Reunindo todas as considerações feitas nesta seção, temos finalmente conhecimento do problema para escolher valores para o tamanho *m.* Se grafarmos os valores obtidos na tabela anterior, obtemos uma boa visão do problema:

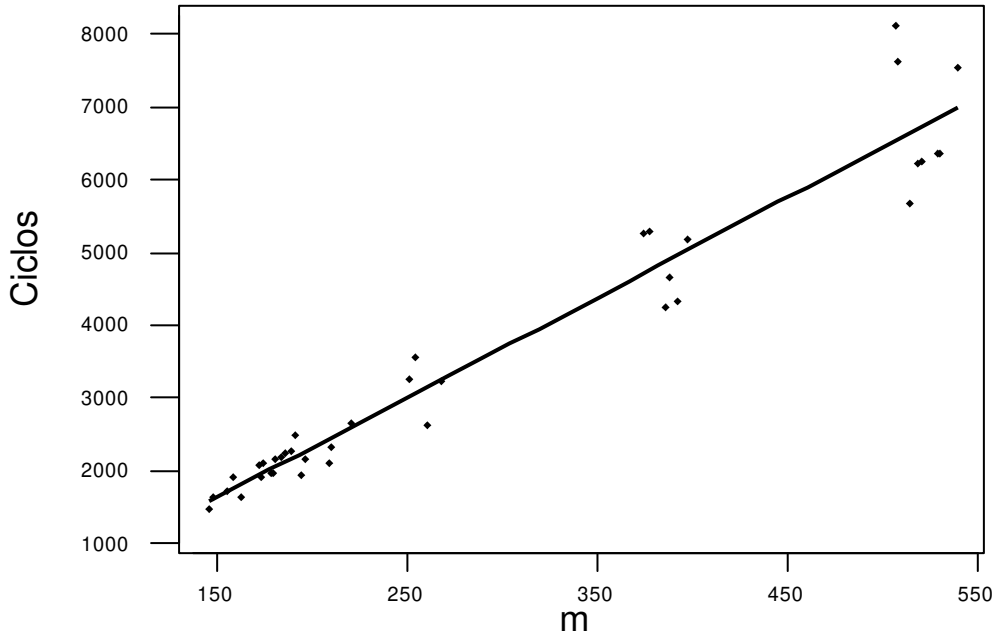


Figura 3.7 – Gráfico representado a tabela 3

A linha reta neste gráfico representa a regressão linear para os pontos considerados. Utilizamos esta regressão linear não para justificar qualquer modelo, mas para mostrar os corpos cujos tempos de inversão estejam abaixo ou acima de uma "média esperada". É fácil notar que os corpos de tamanho 173, 261, 393 e 515 apresentam performance além da média para as regiões que representam níveis de segurança compatíveis com aqueles da Tabela 3.11

3.3 Funções Hash

3.3.1 Introdução

Funções *hash* seguras são uma das mais versáteis primitivas criptográficas. Podem ser usadas para cifragem, autenticação, assinaturas digitais simples [54] [6] e geração de *streams* pseudo-aleatórias .

Estas funções usualmente trabalham sobre mensagens de tamanho arbitrário, quebrando-as em blocos. Elas produzem um resultado, chamado de *hash* de tamanho fixo através de uma função de *compressão*.

As principais características de uma função hash seguras são [42]:

1. Resistência de Pré-Imagem – Requer que dada uma saída $y = h(m_0)$ seja inviável achar *qualquer* mensagem m_i (pré-imagem) que gere aquele hashing *y* específico.
2. Resistência de 2ª Pré-Imagem – Requer que dada uma pré-imagem, seja inviável achar uma segunda pré-imagem que resulte em um mesmo valor de *hashing* que a primeira.
3. Resistência à Colisão – Requer que seja inviável achar *qualquer duas* pré-imagens que resultem no mesmo valor de *hashing*.

Funções hash podem ser construídas a partir de cifradores por blocos ou através de algoritmos especificamente projetados para este fim. Nesta segunda categoria enquadram-se aqueles das famílias MD [61] e SHA – FIPS 180-2 [40].

Nem todos os algoritmos destas famílias são usados. Da primeira, o MD5 é o mais amplamente utilizado e da segunda SHA-1. SHA-256 e SHA-512 são cada vez mais utilizados à medida que o *digest* de 160 bits de SHA-1 torna-se inadequado[14].

3.3.2 Construção Usual, SHA e MD.

Funções de *hash* por iterações têm um modelo como aquele apresentado abaixo. Neste, o maior poder de processamento é consumido na função de compressão *f*. O pré-processamento inicial, composto de *padding* da mensagem e algumas outras operações são de custo muito baixo para as famílias anteriormente citadas.

[14] Pelo paradoxo do dia do aniversário, o seu nível de segurança é de apenas 80 bits.

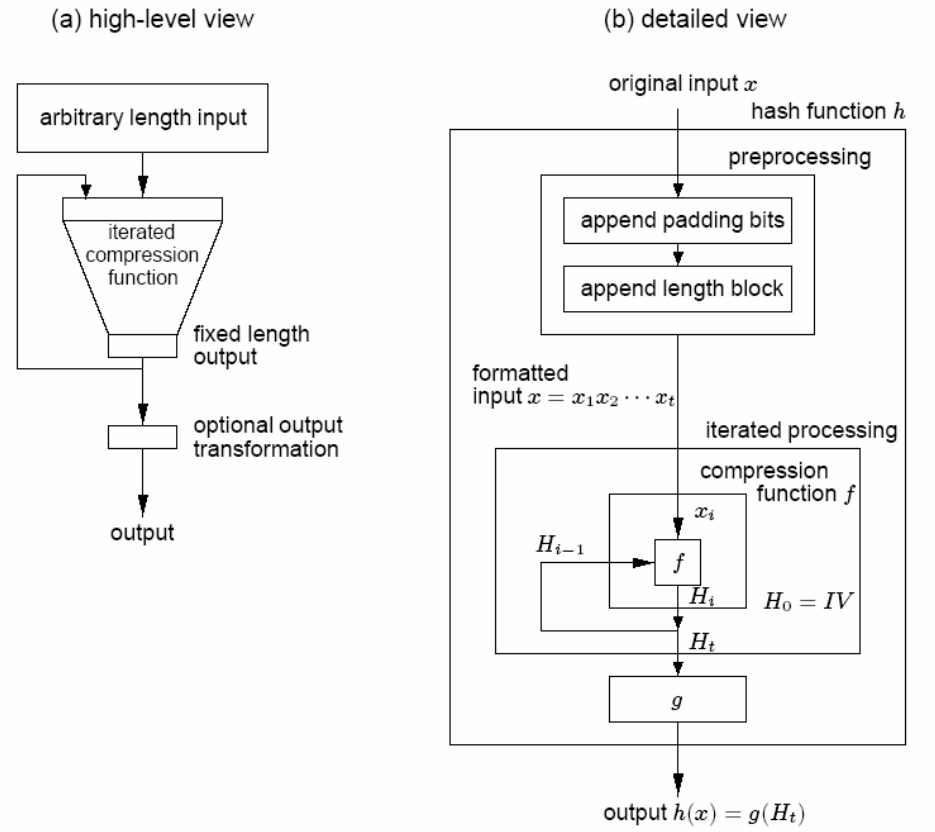


Figura 3.8 – Modelo genérico de função de *hashing*
por blocos retirado de [42].

As funções destas famílias têm uma estrutura em comum, baseada principalmente na construção do MD4, hoje reconhecidamente não seguro. Nesta construção, blocos de mensagem da ordem de 512 a 1024 bits são processados em *rounds* (tipicamente da ordem de 64 a 80 deles) onde são aplicadas transformações sobre a mensagem utilizando-se variáveis temporárias de 32 ou 64 bits.

Estas transformações podem ser divididas entre três grupos: as de rolagem e deslocamento; as de operação lógica bit a bit (*ands*, *ors*, *xors* e *not*) e as aritméticas.

Todas estas operações são realizadas com palavras de 32 ou 64 bits, dependendo do algoritmo. A única operação aritmética presente é da adição de variáveis cujos tamanhos são alguns dos dois anteriores.

A seguir é mostrada uma implementação em C do SHA-1. É fácil notar que o algoritmo tem uma estrutura fortemente seqüencial.

```
U32 K0 = 0x5a827999;
U32 K20 = 0x6ed9eba1;
U32 K40 = 0x8f1bbcdc;
U32 K60 = 0xca62c1d6;

void SHA1(U32 * Mt, U32 * Ht) {

        U32 * W;
        U32 a, b, c, d, e;
```

```
        U32 T;
        unsigned char s, t;

        W = Mt;
        a = Ht[0];
        b = Ht[1];
        c = Ht[2];
        d = Ht[3];
        e = Ht[4];

        for (t = 0; t < 16; t++) {
                T = roll(a, 5) + ((b & c) ^ (~b & d)) + e + K0 + W[t];
                e = d;
                d = c;
                c = rolr(b, 2);
                b = a;
                a = T;
        }

        for (t = 16; t < 20; t++) {
                s = t & MASK;
                W[s] = roll(W[(s+13) & MASK] ^ W[(s+8) & MASK] ^
W[(s+2) & MASK] ^ W[s], 1);
                T = roll(a, 5) + ((b & c) ^ (~b & d)) + e + K0 + W[s];
                e = d;
                d = c;
                c = rolr(b, 2);
                b = a;
                a = T;
        }
        for (t = 20; t < 40; t++) {
                s = t & MASK;
                W[s] = roll(W[(s+13) & MASK] ^ W[(s+8) & MASK] ^
W[(s+2) & MASK] ^ W[s], 1);
                T = roll(a, 5) + (b ^ c ^ d) + e + K20 + W[s];
                e = d;
                d = c;
                c = rolr(b, 2);
                b = a;
                a = T;
        }
        for (t = 40; t < 60; t++) {
                s = t & MASK;
                W[s] = roll(W[(s+13) & MASK] ^ W[(s+8) & MASK] ^
W[(s+2) & MASK] ^ W[s], 1);
                T = roll(a, 5) + ((b & c) ^ (b & d) ^ (c & d)) + e + K40 + W[s];
                e = d;
                d = c;
                c = rolr(b, 2);
                b = a;
                a = T;
        }
        for (t = 60; t < 80; t++) {
                s = t & MASK;
                W[s] = roll(W[(s+13) & MASK] ^ W[(s+8) & MASK] ^
W[(s+2) & MASK] ^ W[s], 1);
                T = roll(a, 5) + (b ^ c ^ d) + e + K60 + W[s];
                e = d;
                d = c;
                c = rolr(b, 2);
                b = a;
                a = T;
        }

        Ht[0] += a;
        Ht[1] += b;
        Ht[2] += c;
        Ht[3] += d;
        Ht[4] += e;
}
```

Os demais algoritmos destas famílias não divergem muito em estrutura da apresentada pelo SHA-1. As funções *roll* e *rolr* executam rolagens à esquerda e à direita respectivamente, pela quantidade dada no segundo argumento.

3.4 Números Aleatórios

3.4.1 Introdução

Seqüências de números aleatórios são importantes em uma série de funções e primitivas criptográficas [6] [42]. São especialmente necessárias para a geração de material para chaves e na construção de *nonces* estatisticamente únicos.

A noção de aleatoriedade é bastante complexa e aberta, mas está ligada ao conceito de entropia, e incerteza sobre os resultados que serão gerados e densidade de informação.

Sistemas digitais são naturalmente determinísticos e portanto incapazes de gerar qualquer quantidade de incerteza. Nem por isso uma fonte de dados de entropia real não é necessária nos mesmos.

Entropia pode ser coletada por um sistema digital através de fontes reais de aleatoriedade que nascem das interfaces destes mesmos sistemas com o mundo físico. Não raro são os sistemas que utilizam variações nos tempos de acessos aos discos rígidos e tempo entre teclas pressionadas para trazer entropia para dentro do sistema.

Ainda que sejam de uma fonte real, a quantidade de dados que pode ser obtida por um método destes é reduzida [54]. Em um sistema onde dezenas de kbytes de material aleatório devem ser gerados, estas fontes não inadequadas.

Para contornar este problema, geradores pseudo-aleatórios são usados em comunhão com sementes realmente aleatórias para gerar *streams* úteis. Naturalmente, geradores pseudo-aleatórios não são nada aleatórios dado que conhecida a semente (e o algoritmo usado) é possível recuperar e prever toda a seqüência.

A maioria dos geradores pseudo-aleatórios foi construída para satisfazer critérios estatísticos de qualidade e não de segurança. Funções construídas com este objetivo, em adição ao primeiro, são ditas criptoseguras (CSPRNGs) [42] e são possivelmente construídas com mecanismos de segurança reconhecida, como cifradores por blocos ou funções hash. De qualquer forma são difíceis de implementar-se, tanto que Knuth [38] aponta que um gerador aleatório não deve ser escolhido aleatoriamente.

TRNGs, ou *True Random Number Generators* podem ser construídos por meio de sistemas que explorem fenômenos físicos e os capturem. Alguns deles são: o ruído térmico dos resistores (ruído Johnson), ruídos de quebra e tunelamento em diodos Zeners e o ruído do efeito Avalanche de junções semicondutoras.

Este último fenômeno foi explorado nesta dissertação através da construção de circuitos que implementam a captura de ruídos dos mesmos e cujos resultados foram avaliados por meio de alguns indicadores de aleatoriedade.

3.4.2 Estimadores de Aleatoriedade

Knuth [38] apresenta uma série de estimadores de aleatoriedade. Alguns deles são brevemente comentados a seguir.

3.4.2.1 Entropia

Uma das definições clássicas de entropia é a dada a seguir:

$$H(X) = -\sum_{x} P(X = x) \log_2 P(X = x)$$

Ela expressa a densidade de informação contida em uma distribuição. Se nos referirmos mais especificamente a sua interpretação computacional (bytes e bits), expressa por exemplo a quantidade média bits necessários para a representação de todos os bytes de um dado bloco.
Esta propriedade pode ser explorada por exemplo por compressores de dados. Dados de alta entropia não são muito compressíveis – uma fonte de dados com 8 bits de entropia por byte não pode ser comprimida em qualquer quantidade.

3.4.2.2 A Distribuição Chi-Quadrado

O teste da distribuição Chi-Quadrado é um dos mais utilizados para o teste de aleatoriedade pois é altamente sensível a erros em fontes pseudo-aleatórias. As distribuições das amostras são comparadas com aquela esperada na distribuição Chi-Quadrado de um conjunto de valores aleatórios [38]. Mesmo dados muito densos, mas não aleatórios, costumam cair no teste.

3.4.2.3 Média Aritmética

É simplesmente a média entre todos os valores da amostra. Em temos computacionais, mais especificamente para um conjunto de bytes que sejam aleatórios, a média deve ser de 127,5.

Ainda que seqüências realmente aleatórias precisem ter médias bem definidas, algumas construções permitem que *streams* não balanceadas de dados sejam transformadas em ruído branco [28]. Suponha que em uma *stream* de bits, gerados por eventos independentes, a probabilidade de um bit *B* ser 0, seja P(B = 0) = p e a probabilidade do mesmo ser igual a 1 seja P(B = 1) = 1-p. Dado que:

$P(B_0B_1 = 00) = p^2$
$P(B_0B_1 = 11) = (1-p)^2$
$P(B_0B_1 = 01) = p-p^2 = P(B_0B_1 = 10)$

Então se observarmos a amostra como pares ordenados (possivelmente agrupados cronologicamente na geração), podemos atribuir o valor 0 para o par "01" e o valor 1 para o par "10", enquanto ignoramos os pares "00" e "11". Desta forma é construída uma segunda amostra balanceada e com bits independentes.

A principal desvantagem deste método é que o número de bits gerados na segunda amostra pode ser muito reduzido caso a original sofra de um desbalanceamento mais forte.

Uma outra possibilidade é a utilização de uma função de compressão simples, como o da soma em GF(2) entre duas metades da amostra original. Sua principal desvantagem ou vantagem está ligada à estimativa de entropia da amostra original. Se a densidade de dados for muito baixa talvez seja necessário repetir o processo, diminuindo consistentemente o tamanho da amostra final.

3.4.2.4 O Valor Monte Carlo para π

Um teste relativamente simples e que converge lentamente para uma amostra de elementos aleatórios é o do cálculo do valor de π de maneira estatística. Neste teste os elementos da amostra são agrupados formando pares ordenados de um plano.
O percentual de elementos cujas coordenadas caírem dentro de um dado raio fixo podem ser utilizado para o cálculo do valor de π.

3.4.2.5 Coeficiente de Correlação Serial

O coeficiente de correlação serial indica em que medida os valores de uma amostra são dependentes serialmente dos anteriores, isto é, como que a observação de um elemento pode permitir a previsão dos posteriores. Idealmente este valor deve ser 0 [38].

Capítulo 4
Arquiteturas para Mecanismos Criptográficos

4.1 Introdução

4.1.1 Mecanismos em Hardware

Neste capítulo serão apresentadas as arquiteturas propostas para três mecanismos criptográficos importantes, o DES (seção 4.5), o AES (seção 4.7) e aceleradores de ECC (seção 4.8) e suas implementações. Além disso, será analisada a viabilidadea das implementações de algoritmos de hash (seção 4.6) padrão e análises sobre os dados do gerador de números aleatórios construído (seção 4.4).

4.1.2 Velocidade e Hardware

Hardware especializado é na pior das hipóteses tão rápido quanto hardware de uso geral na implementação de qualquer algoritmo sendo normalmente bem mais rápido. Estas altas velocidades são conseguidas basicamente através de dois mecanismos; o paralelismo e a implementação de funções de custo reduzido em hardware (quando comparadas às implementações em software), como as operações orientadas a bit.

O paralelismo pode ser explorado normalmente replicando-se unidades de processamento que são utilizadas recorrentemente na execução do algoritmo. Esta abordagem será exemplificada no projeto, arquitetura e implementação do AES, ECC e SHA-1 descritos mais adiante neste texto.

É possível ainda encontrar operações de algoritmos criptográficos cujas implementações em hardware não têm qualquer impacto sobre o número de portas lógicas, e portanto área e freqüência, dos circuitos implementados[15]. Estas funções são principalmente aquelas de deslocamento, rolagem e permutação (*shift*, *roll* e *permutation*). Operações orientadas a bit, por sua vez, podem ser implementadas com baixíssimo custo tanto em termos de área quanto em termos de atraso. Esta característica será evidenciada na implementação em hardware do DES.

[15] Com o aumento das interconexões em FPGAs existe o aumento nos atrasos de propagação dos sinais.

4.1.3 Caminhos Críticos e Hardware com VHDL

A freqüência de operação de qualquer circuito está diretamente ligada ao conceito de caminho crítico. Cada tecnologia de construção de circuitos lógicos rende diferentes implementações de portas lógicas e por conseguinte diferentes valores de atrasos nos sinais que passam através mesmas.

De maneira simplificada, o caminho crítico de um circuito é aquele que apresenta o maior atraso na estabilização do sinal de saída dadas todas a possíveis variações nos sinais de entrada. Tal atraso, em conjunto com outros fatores[16], determina a freqüência máxima de operação do circuito em questão.

O projeto e implementação de arquiteturas devem sempre levar em conta este fator e o projetista deve tentar minimizá-lo e balanceá-lo em relação aos demais circuitos que compõem o sistema.

O VHDL –VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language* é uma linguagem que descreve a maneira pela qual um circuito lógico opera [7] [73], ou seja, com operações sendo feitas em paralelo. Isso é bastante claro na linguagem, onde as linhas de código são normalmente concorrentes, a não ser que se especifique o contrário. Este é um aspecto muito importante de sistemas de hardware, dado que a possibilidade de se realizar operações em paralelo permite que a complexidade de alguns algoritmos possa não ser apenas refletida no número de estágios para a sua conclusão, mas também possa ser reduzida através do compartilhamento operações executadas simultaneamente.

Por ser uma linguagem de descrição, o VHDL assimila além das características mais básicas de um circuito, a características de seu comportamento. Desta forma, existem duas maneiras principais de se descrever hardware utilizando a linguagem. A primeira maneira é a descrição estrutural. A segunda maneira de se descrever circuitos em VHDL é através das descrições comportamentais, sendo possível apenas descrever as saídas como funções das entradas e fazer com que, para o caso de síntese, o compilador ache a melhor implementação para o modelo.

Como nem sempre o compilador ou sintetizador VHDL é capaz de gerar circuitos para descrições comportamentais muito complexas [7][73], deve-se procurar utilizar descrições estruturais e comportamentais de baixo nível dos circuitos.

4.1.4 Desenrolamento de Iteração, Pipeline e Subpipeline

Paar et al [72], Zhang e Parhi [76], Lin et al [77] e Elbirt el al [78] descrevem para diferentes cifradores por blocos, três possíveis otimizações de arquiteturas e seus resultados. Uma vez que cifradores por blocos como AES e DES operam por *rounds* de transformações semelhantes, as técnicas de otimização das arquiteturas se apóiam nesta estrutura regular para

[16] Freqüências mais altas de operação causam maior consumo de energia e por conseguinte aquecimento. Uma vez que as temperaturas máximas de operação de semicondutores devem ser observadas, as freqüências de operação podem ser limitadas.

tentar otimizar algum critério. Estes critérios são área, freqüência de operação, taxa de processamento e outros derivados destes três primeiros.

Cada uma das técnicas (a) Desenrolamento de Iteração – *Loop Unrolling*, (b) *Pipeline* e (c) *Subpipeline* afeta de maneira diferente os critérios mencionados no parágrafo anterior e podem ser combinadas para conseguir nuances de resultados [72] [78].

A técnica de desenrolamento de iterações consiste na implementação de mais de uma *transformação de round*[17] em cada iteração, isto é, mais de um *round* (e possivelmente todos) de transformações são calculados para um mesmo bloco de dados do cifrador por ciclo do relógio.

Enquanto esta técnica permite que uma operação do cifrador seja realizada em somente um período do relógio, por outro lado implica uma menor freqüência de operação. As principais vantagens deste método são: (a) uma cifragem por ciclo e (b) uso reduzido de registradores, que são elementos de alto custo. Já as principais desvantagens são: (a) baixa freqüência de operação, em função do aumento do caminho crítico, e (b) consumo maior de elementos lógicos pois, todas as transformações de rounds são instanciadas mais de uma vez.

A técnica de *Pipeline*, por sua vez, permite a sobreposição da cifragem de diferentes blocos nos circuitos da arquitetura. Com esta técnica é possível cifrar mais de um bloco de dados ao mesmo tempo[18]. Um pipeline em cifradores por blocos é construído com a replicação da transformação de iteração e com a inclusão de registradores entre estas. As principais vantagens deste método são: (a) freqüência de operação elevada e (b) maior taxa de cifragem. As desvantagens são: (a) uso intensivo de recursos, em função das transformações de *round* e registradores replicados e (b) impossibilidade de aproveitamento do *pipeline* por modos de operação com retro-alimentação (CBC, OFB e etc).

O *Subpipeline*, por sua vez, é uma variação da técnica de *pipeline*, onde uma transformação de iteração é dividida em etapas, por meio da inclusão de registradores para os sinais. As vantagem desta abordagem são: (a) aumento da freqüência de operação e (b) pequena adição ao número de elementos lógicos da arquitetura. As principais desvantagens são: (a) aumento da latência para o processamento e (b) não garantia de possibilidade de utilização da técnica, pois, existem transformações que não podem ser satisfatoriamente divididas de modo a balancear o caminho crítico na arquitetura.

4.1.5 FPGAs e Sintetizadores

Field Programmable Gate Arrays – FPGAs são dispositivos que podem ser configurados pelo usuário de forma a implementar uma variedade de funções lógicas, das mais simples às mais avançadas. São ao menos três os principais recursos e elementos de uma FPGA, os lógicos (LEs), os de interconexões e os de memória. Os LEs são compostos de *look-up tables* e registradores e têm acesso a uma variedade de elementos de conexão. Os elementos de memó-

[17] O número de transformações de *round* deve dividir o número de *rounds* total.
[18] Um número de blocos, que divida o número de *rounds*, pode ser calculado simultaneamente.

ria possuem usualmente geometria variável e apresentam freqüências máximas de funcionamento menores, além de serem recursos usualmente escassos.

FPGAs modernas contêm tipicamente de 1.000 a 200.000 elementos lógicos (tipicamente de 25 mil a 5 milhões de portas lógicas), são fabricadas em tecnologias de 130 a 90 nanômetros e têm freqüências máximas de operação da ordem 300 a 500 MHz, com memória embutida de 100 Kbits a 10 Mbits.

Nesta dissertação, as FPGAs utilizadas para a síntese e simulação das arquiteturas construídas são da Altera [82], das linhas Stratix II, Cyclone e APEX20KC. A linha Stratix contém as FPGAs de maior desempenho e tamanho da indústria e são dispositivos de topo de linha. As FPGAs Cyclone possuem uma tecnologia moderna, mas são de baixo custo. A linha APEX20KC representa FPGAs da geração imediatamente anterior à atual e são dispositivos de bom desempenho e muito utilizados. Neste projeto usamos as FPGAs EP2S15C3, EP1C20C6 e EP20KC200C7 das três linhas respectivamente.

A FPGAs Stratix II utilizam elementos lógicos em configuração diferente das demais linhas da Altera, portanto a comparação de utilização entre as linhas não é adequada.

Como compilador, sintetizador e *fitter* para as arquiteturas construídas utilizamos o software Quartus II 4.0 [83] da própria Altera.

4.2 Entrada e Saída para dos Mecanismos

4.2.1 Escolha do Protocolo

Um sistema de entrada e saída (I/O) é muito sujeito às tecnologias de mercado, já que a interface com dispositivos computacionais existentes é bem definida.

Para um cartão de aceleramento criptográfico, utilizando o cripto-processador construído, um padrão como PCI parece o mais adequado à aplicação, enquanto para dispositivos autocontidos, como um HSM com uma Autoridade Certificadora (AC), o cripto-processador pode ser utilizado diretamente como co-processador.

Os critérios de escolha do sistema de I/O são muitos. O primeiro deles refere-se à velocidade de comunicação do dispositivo criptográfico com o sistema *host*; uma vez que altas taxas de cifragem conseguidas necessitam de altas velocidades de transmissão.

Um segundo critério a ser utilizado é a compatibilidade e facilidade de construção dos *drivers* deste sistema de I/O com as plataformas já existentes, para os sistemas operacionais existentes (Intel-Linux, Intel-Windows, Sun-Solaris e Apple-MacOS). Outro critério importante é sem dúvidas a carga do sistema *host* com a interface. É importante que seja baixa, pois caso contrário uma das vantagens da criptografia em hardware, que é liberar a CPU principal, estará comprometida.

Outros critérios relevantes são ainda o custo de construção ou utilização da interface, interfaceamento elétrico, interfaceamento mecânico, compatibilidade passada e futura e finalmente segurança quanto à remoção do dispositivo.

As seguintes interfaces padrão foram consideradas na análise do acelerador criptográfico: Serial, Paralela, ISA, PCI, USB 1.1, USB 2.0 e Firewire 800. Na Tabela 4.1 estão listadas algumas características importantes de cada uma destas interfaces [21] [62] [63].

Critério	ISA (PC104)	PCI	USB 1.1	USB 2.0	Firewire 800 (IEEE 1394b)	Serial	Paralela
Velocidade (MBps)	Média (5,0)	Alta (132 - 528)	Média Baixa (1,5)	Média Alta (60)	Alta (100)	Baixa	Média Baixa (2,0)
Carga do host	Média	Baixa	Média Baixa	Baixa	Média Baixa	Baixa	Média Baixa
Multi Plataforma	Não	Sim	Sim	Sim	Sim	Sim	Sim
Custos	Baixo	Médio Alto	Médio Baixo	Médio	Médio	Baixo	Baixo
Interfaceamento mecânico	Médio	Médio	Fácil	Fácil	Fácil	Fácil	Fácil
Interfaceamento elétrico	Fácil	Fácil	Médio	Médio	Médio	Médio	Fácil
Compatibilidade Futura prevista	Baixa	Boa	Boa	Boa	Boa	Baixa	Baixa
Compatibilidade Passada	Alta	Média Alta	Média Alta	Média (PCI)	Baixa (PCI)	Alta	Alta

Tabela 4.1 – Tabela comparativa de sistemas de IO de uso comercial. O padrão PCI e o Firewire apresentam as melhores características.

Levando-se em conta os critérios mostrados na Tabela 4.1, e em especial a taxa de transmissão, as opções PCI e USB 2.0 parecem as melhores, mas devido novamente ao critério da velocidade, optou-se pelo protocolo PCI.

4.2.2 O Padrão PCI

O Padrão PCI é mantido pelo PCI-SIG [29], um comitê formado pela indústria para a normalização do padrão. Ele cobre todos os detalhes elétricos, lógicos, mecânicos e de protocolo do padrão PCI Local Bus.

O PCI Local Bus foi criado para contornar o gargalo que se formou entre o microprocessador dos sistemas com a interface de vídeo nos primórdios dos sistemas operacionais gráficos (GUI) [29].

Atualmente, o padrão PCI especifica alguns modos de operação para o Bus; comunicação de 32 ou 64 bits, velocidade de barramento de 33 MHz ou 66 Mhz e finalmente tensão de operação de 5 ou 3,3V. A combinação da velocidade do barramento com sua largura permite três velocidades de transmissão de dados de 132 MBps, 264 MBps e 528 MBps (respectivamente 1.056 Gbps, 2.112 Gbps e 4.224 Gbps). Na plataforma PC algumas limitações às taxas de transferência são observadas em função de problemas com *burst* no modo *escravo* do barramento, o que implicam em taxas máximas de 50 MBps para escrita e 30-40 MBps para leitura [20].

O barramento (*bus*) PCI é conectado aos sistemas através de uma *bridge* que gerencia o acesso do processador e *bus* à memória, e permite concorrência total destes dispositivos. Isso implica numa pequena carga sobre o processador principal.

O PCI é síncrono e tem pequenos tempos de latência, de 60ns a 33MHz e 30ns a 66MHz, não necessita de circuitos adicionais para a conexão de dispositivos e permite operação de leitura e escrita nos modos normal ou *burst*.

Um periférico, de acordo com o padrão [29], "pode operar no modo *Master* ou *Slave*, é independente do processador utilizado no sistema e permite um porte relativamente simples dos *drivers* de software entre plataformas, tornando-se ideal como sistema de I/O para os sistemas em questão."

O PCI Local Bus estabelece linhas de dados e linhas de endereçamento para os dispositivos. A unidade básica para a transferência do barramento é o *burst*. Ele é composto de uma fase de endereço e uma ou mais fases de dados. A Figura 4.1 [20] ilustra uma operação de leitura.

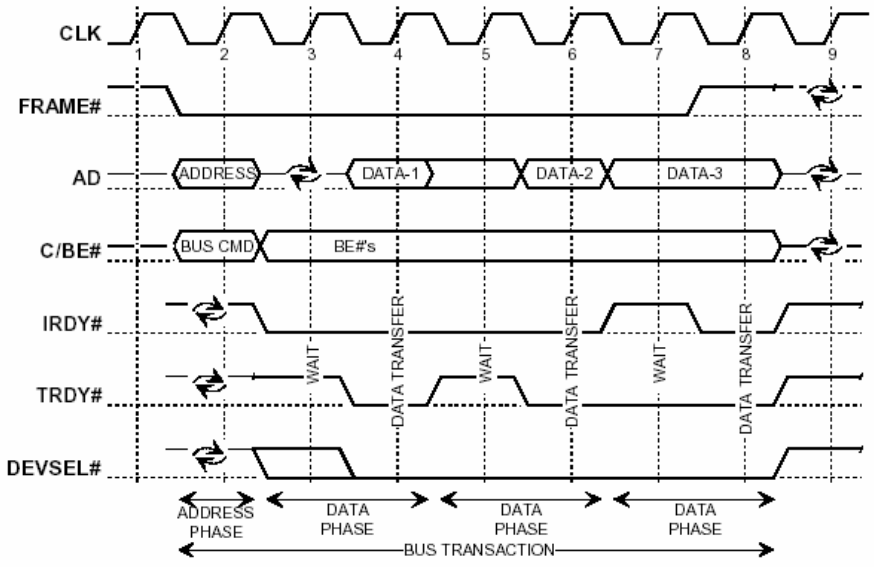


Figura 4.1 – Uma transação pelo protocolo PCI. Uma transação é composta de um ciclo de endereços e um ou mais ciclos de dados. A transação mostrada é um *burst* de leitura.

Se por um lado o padrão PCI permite altas taxas de transmissão e um baixo comprometimento da CPU (tornando-a o padrão de barramento *de facto*), sua especificação é um tanto ampla e complexa, o que faz com que desenvolvedores de dispositivos de uma maneira geral concentrem-se em seus projetos e utilizem blocos prontos de *bridging* (PCI *chips*).

Existem duas maneiras de se fazer isso. A primeira é através da utilização de *semiconductor IP*s (ou IP *cores*), blocos de circuitos descritos utilizando alguma HDL[19] ou através de circuitos integrados que fazem este papel. Em nosso caso utilizaremos a segunda opção, pelos motivos apresentados na secção a seguir.

4.2.3 Implementação da Placa PCI em *Wire-up*[20]

Uma placa de referência foi utilizada na construção do sistema de I/O para os dispositivos criptográfico complementando o *framework* de tecnologias e sistemas necessários a utilização de um cripto-processador. Tal placa de referência permite o *wire-up* de componentes aos seus circuitos bem como a soldagem, de forma que é possível construir circuitos de complexidade mediana no próprio kit de desenvolvimento.

O subsistema de controle foi implementado e exercitado através de uma API especialmente escrita para o acesso à placa pelo sistema operacional. Depois de alguns meses de ajustes o sistema passou a funcionar de maneira estável e com tempo de resposta relativamente baixo.

A placa foi construída utilizando-se o kit de desenvolvimento PLX de outro projeto do LSC-IC-Unicamp[21] e contou com a participação fundamental do engenheiro eletrônico Henrique M. Kawakami.

A construção de tal sistema permite a incorporação relativamente simples de uma gama de componentes criptográfico. Os sistemas incorporados foram: o gerador de números aleatórios, guarda segura de chaves e unidade de controle para o cripto-processador. A estabilização do sistema tomou algum tempo.

4.2.4 Problemas Encontrados

Devido à alta freqüência que opera o barramento PCI, as linhas de sinal passam a comportar-se como antenas, gerando problemas de interferência eletromagnética. No *wire-up* tais problemas são maximizados, devido tanto à inexistência de plano terra próximo, quanto à proximidade das linhas.

---

[19] Do inglês: *Hardware Description Language*. Dentre as mais utilizadas estão o VHDL, Verilog e SystemC.

[20] O *wire-up* é uma técnica de montagem de circuitos através da conexão dos componentes por meio de *fios* condutores que são presos aos terminais dos mesmos sem soldagem.

[21] Laboratório de Sistemas de Computação do Instituto de Computação da Universidade Estadual de Campinas.

Quando uma trilha de dados interfere e induz níveis lógicos diferentes daqueles esperados nos barramentos, acontece o problema chamado de *crosstalking*. Tal problema é de difícil predição e análise e demorou certo tempo para ser identificado e solucionado.

A solução veio através da aproximação dos fios do circuito com o plano terra da placa, inclusão de resistores a fim de ajustar a impedância das linhas e finalmente novo roteamento destas. A identificação de tais problemas somente foi possível com o uso de equipamentos como osciloscópios e software com capacidade de cálculo de impedância de *crosstalking*.

Resolvido este problema, evidenciou-se então um segundo: o travamento completo do sistema *host* nas operações endereçando à placa de desenvolvimento. Tal problema teve solução mais rápida e era fruto da natureza do próprio barramento PCI.

Quando uma operação começa no barramento ela é encarada como uma transação – precisa ter um prólogo, envio ou recebimento dos dados e finalmente sua fase de término. Tal fase de término é essencial ao protocolo, pois o barramento principal do *host* fica à espera desta finalização.

Sob certas condições de temporizarão, onde interrupções eram perdidas pelo sistema operacional, uma leitura do barramento pelo *host* era disparada. Com o subsistema de controle fora de operação, tal transação não era finalizada e o *host* bloqueava.

Para acelerar o *debuging*, uma API para o sistema foi construída, seguindo o modelo proposto pelo PKCS#11 [41].

4.3 API compatível PKCS#11

A constituição da API[22] merece especial atenção no desenvolvimento de um framework para hardware criptográficos. De maneira geral, dispositivos criptográficos podem implementar em diferentes graus uma série de serviços criptográficos.

Enquanto um HSM deve dar suporte a uma vasta gama de operações necessárias a uma AC, um guardador de chaves USB usualmente carrega apenas um certificado. Para que aplicações possam utilizar com facilidade dispositivos de características variadas, uma API genérica padrão foi criada pelo RSALabs em conjunto com a Indústria e a Academia [41].

O resultado foi o PKCS#11 – *Public Key Cryptographic Standard* 11, que define a *criptoki*, uma interface genérica e expansível entre *tokens* criptográficos e aplicações. O PKCS#11 é o padrão *de facto*, sendo adotado por todas grandes empresas da área como IBM, Diebold, Thales e Rainbow.

A *criptoki* tem por objetivo permitir a abstração do *hardware* utilizado no fornecimento de serviços criptográficos e define uma série de *métodos* e *propriedades* que são expansíveis e implementam sistemas de propriedade (*ownership*). De maneira simplificada, a utilização de um dispositivo através da API requer que o usuário abra uma *sessão*.

Sessões de dois tipos são possíveis, as de SuperUsuário (SU) e as de Usuário. As sessões de superusuário são necessárias para que uma sessão normal possa ser aberta. Objetos

[22] Do inglês: *Application Programming Interface.*

podem ser criados nos módulos; estes podem ser características públicas e privadas, ou seja, objetos de propriedade de um usuário podem ser restritos somente àquele usuário.

Serviços criptográficos podem ou não estar disponíveis para cada usuário. Mais de uma sessão, a critério do fabricante, pode ser aberta no dispositivo. A maioria das primitivas criptográficas padrão existentes hoje têm referencias na API definida em [41] e quaisquer novas podem ser inclusas.

Se por um lado tal flexibilidade permite fácil integração e a utilização em uma série de dispositivos diferentes, ela pode levar a uma série de ataques de segurança [64] [65]. Estes podem ser de níveis diferentes de gravidade, chegando até ao controle completo do dispositivo e passando pela recuperação de chaves.

Os ataques mencionados em [64] e [65] podem ser evitados em dois pontos distintos; na própria implementação da *critoki* (que em geral é um produto por si só) e ou na implementação do dispositivo. Naturalmente a segunda opção apesar de ser menos flexível e de ter que ser realizada para cada módulo é a mais segura pois impede que ataques ao software da API sejam realizados a fim de explorar fragilidades dos módulos [65].

Um dispositivo que seja integrável à criptoki definida no PKCS#11 não precisa implementar todas as funções do padrão e igualmente não precisa ter exatamente sua estrutura. É papel do fabricante da criptoki a integração entre a API do token com a sua [41]. Esta integração pode ser em muito facilitada, no entanto, se a construção do dispositivo criptográfico levar em conta a integração necessária.

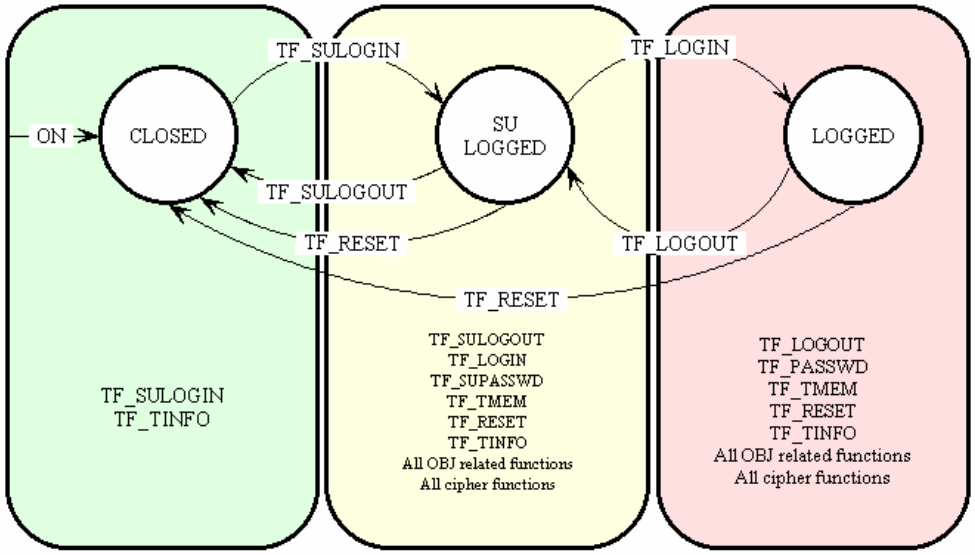


Figura 4.2 – Ilustração do sistema de sessões e principais funções implementadas na API do projeto.

Para a placa PCI montada neste projeto, a construção de *drivers* e de uma API foi claramente necessária a fim de se fornecer às aplicações as primitivas criptográficas criadas. Tais bibliotecas foram construídas tendo em mente a fácil integração ao padrão PKCS#11 e foram feitas de forma a resistir a vários tipos de ataques conhecidos [65], sendo que aqueles de

recuperação de chaves, por exemplo, são simplesmente impossíveis por construção, já que de maneira alguma é possível exportar chaves.

O resultado foi uma API própria não tão permissiva porém flexível e que junto do sistema de controle implementado na placa têm as características de *propriedade* de objetos, serviços criptográficos e mecanismos de *sessão* definidos no padrão.

Cerca de meia centena de funções foram criadas na API própria de forma a permitir a utilização das primitivas criptográficas implementadas e descritas neste documento. Esta API teve ainda papel fundamental no processo de depuração dos mesmos mecanismos.

A Figura 4.2 ilustra alguns métodos disponíveis e seu funcionamento, com estrutura de sessões e com algumas funções disponíveis em cada tipo de sessão.

4.4 Gerador de Números Aleatórios

A fim de testar a qualidade da geração de números aleatórios pelo efeito avalanche, foi construído um dispositivo físico para a aquisição de dados. A concepção dos circuitos foi realizada com o auxílio do engenheiro eletrônico Henrique M. Kawakami, baseado em diversos projetos da literatura [79] [80].

A Figura 4.3 apresenta o esquema elétrico do circuito tal qual ele foi construído e representa a fonte de entropia real do gerador. A *stream* de bits gerada necessitou ainda de algum processamento para produzir ruído aproximadamente branco.

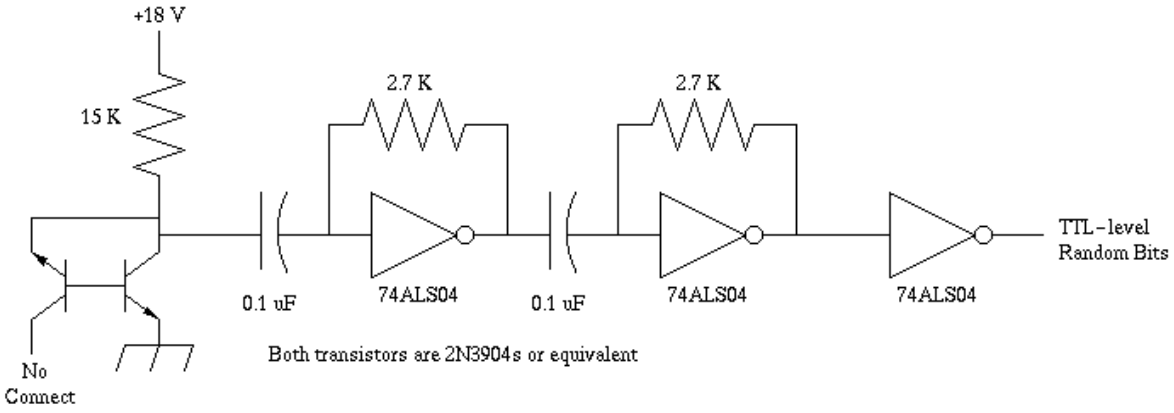


Figura 4.3 – Esquema elétrico da parte
responsável pela geração de entropia do RNG

A análise da saída do gerador mostrou dois efeitos importantes da implementação. O primeiro deles tem relação com os níveis de tensão necessários para a interpretação de níveis lógicos pelas portas inversoras comerciais 74ALS04 e implicou em um desbalanceamento dos bits de saída. A probabilidade de bits com valor ‘1’ serem gerados foi de ~ 0.6 nas três instâncias construídas do circuito.

O segundo efeito observado foi o de aumento do coeficiente de correlação serial dos dados com o aumento da taxa de amostragem da saída do circuito. Enquanto os inversores

utilizados respondiam até freqüências de centenas de MHz [66], a capacitância das linhas junto com uma distribuição não plana do período típico de uma *avalanche* dos semicondutores conduziram o circuito a uma situação onde amostragem a mais de 500 Kbps começavam a apresentar coeficientes de correlação maiores que 0.001, valor escolhido de *threshold* [38].

Indicador-Balanceamento	Por Soma	Não Determinístico	Ideal
Entropia [bits/byte]	7,999645	7,999223	8
Compressão Teórica [%]	0	0	0
Chi-Quadrado [%]	50	75	50
Média Aritmética	127,496	127,5128	127,5
Pi por Monte Carlo (erro %)	3.138525391 (0.10%)	3.142987192 (0.05%)	0
Correlação Serial	0.000291	0.000188	0

Tabela 4.2 – Resultados das análises estatísticas
para o RNG implementado.

A análise de amostras típicas capturadas a 8 Kbps neste circuito, com uma função de balanceamento por *soma* conforme introduzido em 3.4.2, resulta nos dados da Tabela 4.2. Na mesma tabela estão os resultados das análises da utilização do balanceamento não determinístico [28] para a mesma amostra típica.

As análises dos dados gerados pelo *hardware* foram feitas utilizando-se o software ENT [68] e Diehard [69], e mostram uma boa qualidade, com altos níveis de entropia e compressão nula, dentre outros. Uma comparação entre os modelos de balanceamento sugere que ambos têm qualidades similares. Já do ponto de vista de taxa de geração, o segundo modelo apresenta taxa média de geração 4.17 vezes menor que aquela da fonte de entropia real enquanto o primeiro modelo foi somente 2 vezes mais lento.

Taxas mais altas de geração de números aleatórios podem ser facilmente alcançadas utilizando múltiplas instâncias do circuito proposto já que sua construção com tecnologia SMD[23] permite alta densidade.

Alguns testes elementares de dependência térmica e imersão em campos eletromagnéticos foram realizados para se estudar a viabilidade de alguns ataques. No teste de dependência térmica, nenhuma relação clara foi encontrada. Os circuitos foram colocados sob operação em temperaturas variando de 0 ºC a 80 ºC sem qualquer prejuízo verificável pelos testes estatísticos.

Nos testes de imersão em campos eletromagnéticos foram utilizados campos de freqüências de 60 Hz, 100 KHz, 25 Mhz e 1.8 GHz. Ainda que nenhuma interferência imediata tenha sido observada, possivelmente pela ação do plano terra do circuito, um estudo mais detalhado constitui tema interessante de pesquisa. Tal estudo deveria incluir testes com faixas de freqüências amplas (na procura de ressonâncias) e campos com diferentes intensidades.

[23] Do inglês: *Surface Mount Device* – Tecnologia de encapsulamento de componentes que permite solda automatizada e componentes de tamanho extremamente reduzidos.

4.5 Arquitetura para o DES

O *Data Encryption Standard* é um exemplo expressivo de como uma implementação em hardware de um algoritmo pode ser muitas vezes mais eficiente do que aquela feita em software.

Nesta dissertação apresentamos uma arquitetura para o DES que, utilizando tecnologia atual, é tipicamente muito mais rápida que aquela observada em software [70], e como esperado [78], mais rápida que aquelas em *hardware* por Paar et al [72].

O DES apresenta uma série de dificuldades para ser implementado em software pois envolve muitas permutações realizadas bit a bit, de forma bastante irregular. A transformações representadas pelas tabelas IP, PC-1, PC-2, E e IP^{-1} são basicamente todas operações de permutação de bits, custosas de serem implementadas em software.

Por outro lado, essas mesmas transformações apresentam custo zero[24] quando implementadas em *hardware*. Das operações do algoritmo sobram somente as consultas às tabelas S_n, a derivação de chaves e as somas *xor* para adição de chaves e da estrutura de Feistel.

A derivação das subchaves (ou chaves) de *round* pode ser realizada uma única vez para cada diferente chave do algoritmo. Esta pré-derivação, ou expansão, é especialmente interessante para implementações em *software* pois neste caso a utilização de chaves pré-calculadas é mais eficiente do que o cálculo das mesmas sob demanda.

Já para implementações em *hardware*, a derivação de chaves pode ser realizada em paralelo com as transformações de *rounds* do algoritmo. Este paralelismo não adiciona atrasos ao caminho crítico e utiliza menos recursos que a opção de pré-derivação já que não precisa de armazenamento para as chaves de round.

As adições *xor* do algoritmo são simples de serem realizadas em ambas implementações, ao contrário das consultas às tabelas S_n. Em *software* estas tabelas podem ser eficientemente reagrupadas para gerar tabelas maiores mas de menor tempo de busca. Mesmo assim, Corella [70] somente foi capaz de produzir uma implementação que consome 192 ciclos para um cifragem DES.

Estas mesmas tabelas podem ser implementadas em hardware utilizando-se funções lógicas ou elementos de memória das FPGAs e em ambos os casos a utilização das tabelas pode ser feita completamente em paralelo. Nos dois casos os recursos necessários são modestos quando comparados com a capacidade das FPGAs atuais.

A Figura 4.4 apresenta a arquitetura implementada para o DES. Nela são facilmente identificáveis todas as transformações do algoritmo.

[24] Em termos de portas lógicas. O roteamento destes sinais por outro lado pode implicar em atrasos na transmissão, especialmente custosos em FPGAs. Além disso, elementos de interconexão são bastante consumidos como mostraram testes realizados. Os atrasos na transmissão são porém tipicamente bem menores que aqueles das portas lógicas propriamente.

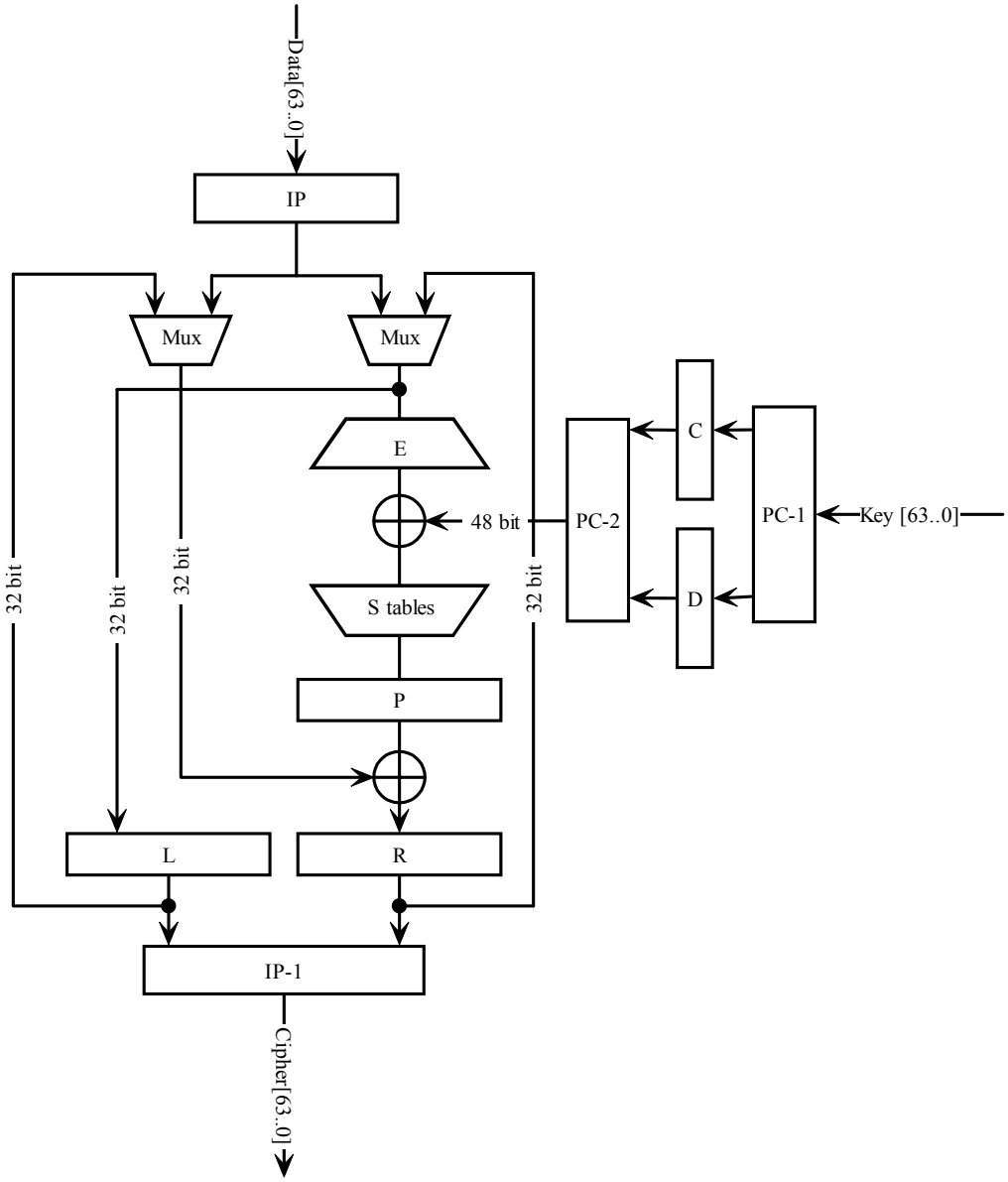


Figura 4.4 – Arquitetura implementada para o DES.

Nesta arquitetura um bloco de dados é admitido a cada 16 ciclos de relógio, o tempo necessário para que se complete a cifragem do bloco anterior. Para simplificação do diagrama, tanto a máquina como os sinais de controle foram omitidos.

Os registradores L e R guardam os estado entre ciclos do relógio. O caminho crítico do circuito é aquele que o sinal de saída de R perfaz até a a chegada novamente no registrador, passando através da expansão E, a adição da chave de round e pelas tabelas de substituição S.

A assinatura VHDL da *core* criado é a seguinte:

```
entity DES is
       port (DnK             :      in            Std_Logic;
             Load            :      in            Std_Logic;
               Go            :      in            Std_Logic;
               Clock         :      in            Std_Logic;
               Reset         :      in            Std_Logic;
               CnD           :      in            Std_Logic;
               Ready         :      out           Std_Logic;
               Key           :      in            Std_Logic_Vector(1 to 64);
               DataIn  :      in           Std_Logic_Vector(1 to 64);
             DataOut   :      out          Std_Logic_Vector(1 to 64));
end DES;
```

A Tabela 4.3 apresenta os resultados obtidos na síntese de nossa arquitetura para os dispositivos anteriormente mencionados, Cyclone, Stratix II e APEX20KC. Nenhuma mudança no código VHDL foi realizada entre as sínteses. A diferença de performance deve-se exclusivamente às características do dispositivo utilizado.

Dispositivo/Item	Interconexões	El. Lógicos	Registradores	Taxa	F_{Max}
EP1C20C6	2%	598 LEs	126	804Mbps	201MHz
EP2S15C3	1%	401 ALUTs	126	1260Mbps	315MHz
EP20K200C7	n.d.	658 LEs	126	472Mbps	118MHz

Tabela 4.3 – Resultados obtidos para o *fitting* em diferentes dispositivos de nossa implementação do DES sem *pipeline*.

A síntese foi realizada utilizando somente elementos lógicos para as tabelas S_n pois rendem melhor performance do que aquela com elementos e memória das FPGAs, como pode ser observado durante o desenvolvimento através experimentações. Isto deve-s ao fato que essas tabelas são simples e com profundidade pequena (*fan in* de 6 sinais, menor que a LUT das Stratix II, por exemplo). Entretanto, para implementações sensíveis à área pode-se utilizar a aproximação da implementação em ROM, através da configuração do sintetizador ou simplesmente utilizando uma biblioteca de funções parametrizadas (LPM).

A especificação do DES permite uma fácil implementação utilizando técnicas como desenrolamento de iterações e *pipeline*, principalmente em sua versão completa (contendo todos os *rounds*), pois a máquina de expansão de chaves nestes casos é mais simples do que aquela para a implementação básica – não é necessário implementar registradores de deslocamento na derivação dos conjuntos C_nD_n. Já subpipeline não aparenta ser uma boa opção de implementação, uma vez que nas transformações de round, a utilização das tabelas S_n é bem mais lenta que a operação de *xor* envolvida.

A Tabela 4.4 contém os resultados de performance para algumas implementações rápidas conhecidas [70] [72] além daquele produzida nessa dissertação. É importante notar que o *core* implementado aceita decifragem e cifragem.

A configuração descrita como "Nossa Full Pipelined" é tão somente a replicação da "Nossa Básica", da figura 4.4, onde 16 instâncias da segunda são seqüencialmente ligadas formando o *pipeline*. É importante notar, entretanto, que esta aproximação pode não ser inte-

ressante se o sistema de IO não for capaz de dar vazão aos dados processados ou mesmo se o modo de operação do cifrador for encadeado.

Implementação	Ciclos/Bloco	Taxa	Plataforma
Paar el al Básica	16	99 Mbps	XC4009E-3-PG223
Paar et al Pipeline de 4 estágios	4	400 Mbps	XC4028EX-3-PG299
Nossa Básica	16	1260 Mbps	EP2S15C3
Nossa Full Pipelined	1	18,9 Gbps	EP2S15C3
Corella Software	192	183 Mbps	PA-RISC 2.0 Alpha (64 bits) 550 MHz.

Tabela 4.4 Performances para algumas implementações do DES.

Ainda que o design de nosso *core* seja enxuto, a diferença de performance entre as versões de Paar e as apresentadas nesta dissertação é provavelmente fruto de melhoria nos sintetizadores para VHDL e na evolução das FPGAs.

A implementação em VHDL produzida está em conformidade com boas práticas para a linguagem [73] e foi liberada em LGPL [87] para posterior integração ao IP Brazil [30].

4.6 SHA e MD5.

Os algoritmos da família SHA [40] e o MD5 [42] são primitivas criptográficas importantes para quase qualquer sistema que utiliza criptografia para serviços de autenticidade. Estes mecanismos compartilham uma estrutura similar baseada em uma função de compressão iterada.

A implementação em ANSI C apresentada em 3.3.2 mostra claramente alguns pontos fundamentais desta função de compressão. Primeiro pode-se facilmente notar que os passos envolvidos nos *rounds* são altamente seqüenciais, com pouca margem a paralelização.

Em segundo lugar, nota-se que a operação crítica do algoritmo, soma *mod* 2^{32}, é repetida para cinco termos, e mais, não permite muito paralelismo. Isso é um problema relativamente grave uma vez que a operação de soma aritmética usual com *carry look-ahead* é custosa em termos de recursos lógicos de ser realizada em um ciclo para grandes palavras [10], principalmente em virtude do cálculo dos próprios *carries*.

Se levarmos em conta que funções de *hash* são utilizadas em modo retro-alimentado (*feedback mode*), podemos por em dúvida a vantagem de desempenho de uma arquitetura especializada para algoritmos como SHA e MD5, já que o paralelismo não pode ser muito bem explorado.

Nesta seção, não se tentou implementar os algoritmos SHA-1 e MD5 por inteiro, mas sim determinar qual a performance atual de FPGAs para a operação crítica do algortimo, a da soma *mod* 2^{32}.

Desta forma foram investigadas as performances para duas arquiteturas que implementam a operação em questão. A primeira arquitetura utiliza uma unidade de soma de 32 bits, que em cinco ciclos de máquina é capaz de processar uma iteração do algoritmo. A segunda implementação processa uma iteração do algoritmo por ciclo e utiliza uma estrutura de árvore para as somas de 32 bits do *round*.

Dispositivo/Item	LEs	Taxa	F_{Max}
EP1C20C6	64	399 Mbps	312 MHz
EP2S15C3	32 (ALUT)	540 Mbps	422 MHz
EP20K200C7	64	253 Mbps	195 MHz

Tabela 4.5 - Performance para a operação de soma *mod* 2^{32} dos cinco elementos de SHA-1 utilizando arquitetura com um somador.

Dispositivo/Item	LEs	Taxa	F_{Max}
EP1C20C6	128	1209 Mbps	189 MHz
EP2S15C3	69 (ALUT)	1683 Mbps	263 MHz
EP20K200C7	128	902 Mbps	141 MHz

Tabela 4.6 - Performance para a operação de soma *mod* 2^{32} dos cinco elementos de SHA-1 utilizando arquitetura com somador em árvore.

Nas tabelas 4.5 e 4.6 são apresentados os resultados para as arquiteturas implementadas. O número de elementos lógicos utilizados refere-se somente aos utilizados pela unidade de soma e um acumulador. Na arquitetura da Figura 4.5 (a), que utiliza somente um somador, é utilizado um esquema de acumulador para armazenar os dados intermediários da transformação. Já na outra arquitetura, uma árvore de soma é utilizada, tomando-se o cuidado de fazer o caminho crítico o menor possível (Figura 4.5 (b)), pois os sinais W[s] e f(b, c, d) apresentam atraso considerável em função das operações que são realizadas.

Para a implementação, algumas técnicas de soma foram utilizadas, mas a biblioteca da Altera *lpm_add_sub*, que utiliza toda a especialização do hardware, foi a que apresentou o melhor resultado tanto em termos de LEs como de freqüência de operação.

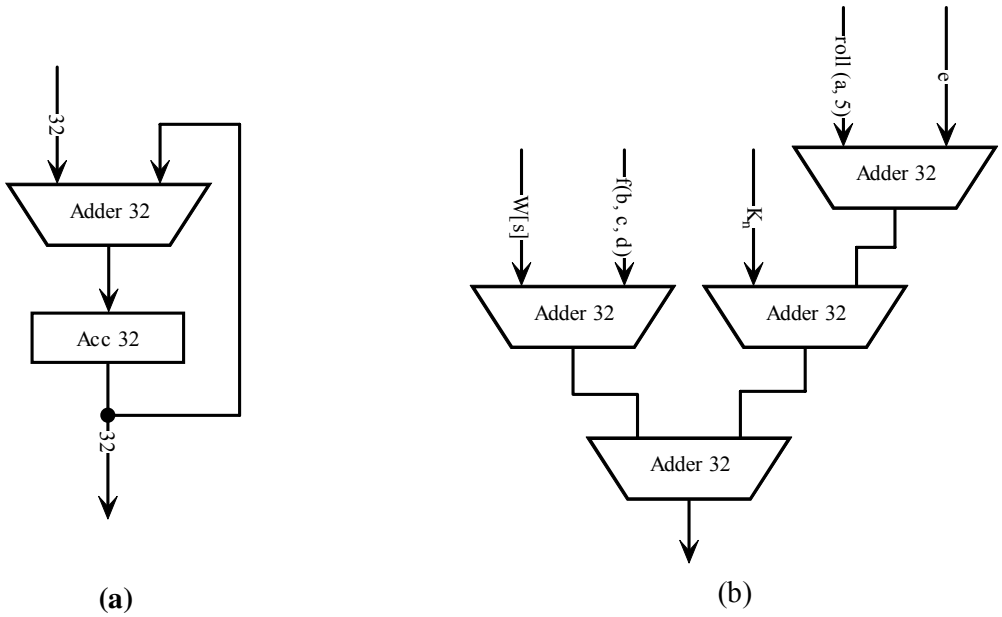


Figura 4.5 – Somador para as arquiteturas voltadas à implementação da função de compressão de SHA-1.

Os resultados de performance obtidos são equivalentes àqueles apresentados em implementações comerciais do algoritmo, que também utilizam a técnica de árvore para a implementação [74] e podem ser facilmente estendidos para os demais algoritmos da família SHA e o MD5, que utilizam somas *mod* 2^{32} como operações críticas.

O resultados obtidos de freqüências de operação são similares aqueles para o DES, o que permite uma integração relativamente simples dos algoritmos no mesmo dispositivo sem o uso de PLLs.

Uma alternativa à implementação completa em hardware reconfigurável destes algoritmos é o da inclusão de instruções especiais nos processadores de uso geral, como deslocamentos circulares de mais de uma posição (2 e 5) , operações especiais com 3 operandos para o cálculo em um só ciclo da função *f* descrita no padrão [40]. A inclusão de instruções especiais em processadores de uso geral, como feito no projeto ChameLeon [81], para funções criptográficas constitui um tema rico e aberto para outros projetos de pesquisa.

4.7 Arquitetura para o AES

O AES é um algoritmo que têm boa performance em implementações em *software* [15] e cujas implementações em *hardware* também apresentam bom desempenho [78]. Ao contrário do DES que opera principalmente funções descritas em nível de bit, as operações do AES podem ser facilmente descritas com operações sobre bytes.

Das transformações de *round* do algoritmo, as duas mais complexas são *MixColumns* e *SubBytes* sendo que a segunda merece uma atenção especial. A transformação *ShiftRows* é de custo zero pois apenas realiza um deslocamento fixo dos bytes do Estado. *AddRoundKey* é simples e não tem espaço para otimizações pois implementa simplesmente um *xor* bit a bit.

Nesta secção serão apresentadas com detalhes as arquiteturas construídas para a máquina de expansão de chaves e para o *datapath* e para as transformações *MixColumns* e *SubBytes*.

4.7.1 SubBytes e InvSubBytes

A transformação SubBytes é a que causa maior impacto no número de LEs utilizadas pelo mecanismo como um todo. Como descrito na seção 3.1.2, esta transformação opera sobre uma entrada de 1 byte e produz um resultado de igual tamanho. A aproximação mais direta para a sua implementação, utilizada em [78], usa um tabela de 256 bytes (1 x 256) para a transformação direta e outra de igual tamanho para a transformação inversa *InvSubBytes*.

Em uma FPGA, estas tabelas podem ser diretamente implementadas por meio de funções lógicas (usando LUTs[25]) ou através da utilização de ROM assíncrona. No primeiro caso cabe ao sintetizador minimizar o número de elementos utilizados.

Uma alternativa para esta abordagem é construir uma arquitetura que de fato calcule o resultado da transformação, que grosseiramente pode ser dividida em duas etapas: (a) a inversão do elemento em $GF(2^8)$ e (b) a aplicação da transformação afim. Já a transformação inversa *InvSubBytes* é calculada por meio da aplicação da transformação afim inversa e posteriormente da inversão em $GF(2^8)$.

A vantagem deste segundo método é que se pudermos fazer a inversão eficientemente os circuitos gerados devem ser reduzidos. Além disso, as transformações direta e inversa compartilham uma operação em comum, a inversão em $GF(2^8)$. Resta verificar qual o melhor desempenho para a inversão, utilizando a inversão binária geral ou a inversão por mudança de bases.

Nas Tabela 4.7 e 4.8 estão apresentados os resultados de variações para a implementação da transformação SubBytes.

[25] Do inglês: *look-up tables*

	ROM		LUT	
Dispositivo/Item	Recursos	Atraso	Recursos	Atraso[26]
EP1C20C6	2048 bits	5,83 ns	208 LEs	~7,2 (12,86) ns
EP2S15C3	2048 bits	4,26 ns	60 ALUTs	~4,4 (8,26) ns
EP20K200C7	2048 bits	5,22 ns	226 LEs	n/d

Tabela 4.7 – SubBytes utilizando tabelas em implementações por LUTs e ROM.

	Exponenciação		Mudança de Base	
Dispositivo/Item	Recursos	Atraso	Recursos	Atraso
EP1C20C6	252 LEs	18,93 ns	73 LEs	9,61 ns
EP2S15C3	203 ALUTs	12,73 ns	61 ALUTs	6,21 ns
EP20K200C7	279 LEs	26,63 ns	83 LEs	12,49 ns

Tabela 4.8 – SubBytes utilizando inversões.

Os resultados mostram que a implementação mais rápida da transformação SubBytes é aquela que utiliza ROM para o cálculo. Se levarmos em conta que são necessárias 16 instâncias de S-Box na transformação de round e mais 4 para a derivação de chaves, o consumo de bits de memória será de 40960, para uma versão somente de cifragem ou decifragem e 73728 bits para uma versão dual, que dependendo do dispositivo são valores consideráveis. Considerações semelhantes podem ser feitas acerca dos elementos lógicos utilizados (LEs e ALUTs).

A Tabela 4.8 mostra também que a implementação utilizando inversão por exponenciação é a pior nos dois parâmetros. Já a implementação utilizando inversão dos elementos por mudança de base, apesar de não ser a mais rápida (de 25 a 30% mais lenta), é seguramente aquela que utiliza menor quantidade de elementos lógicos (de 63 a 65%) para a implementação.

Como é possível instruir o sintetizador a implementar uma função lógica em ROM, é factível recriar as implementações por tabelas partindo da implementação que inverte e aplica a transformada afim nos elementos de entrada.

Rijmen [24] aponta que um esquema de inversão por mudança de bases poderia ser mais eficiente do que aquele utilizando outro método, mas também aponta que o sintetizador poderia ser capaz de criar uma implementação já reduzida partindo da tabela pré-calculada. Pode-se entretanto observar que a situação é parcialmente verdadeira, pois somente para o dispositivo da linha Stratix II foi possível uma implementação menor e mais rápida do que aquela conseguida pela inversão.

A implementação escolhida foi por inversão com mudança de bases, pois adiciona flexibilidade à arquitetura, já que com ela é possível recriar as versões por tabelas e ainda é possível utilizar o núcleo de inversão para a transformação *InvSubBytes*, com a arquitetura mostrada na Figura 4.6.

[26] Atraso estimado através dos tempos de propagação pino a pino, entre parênteses.

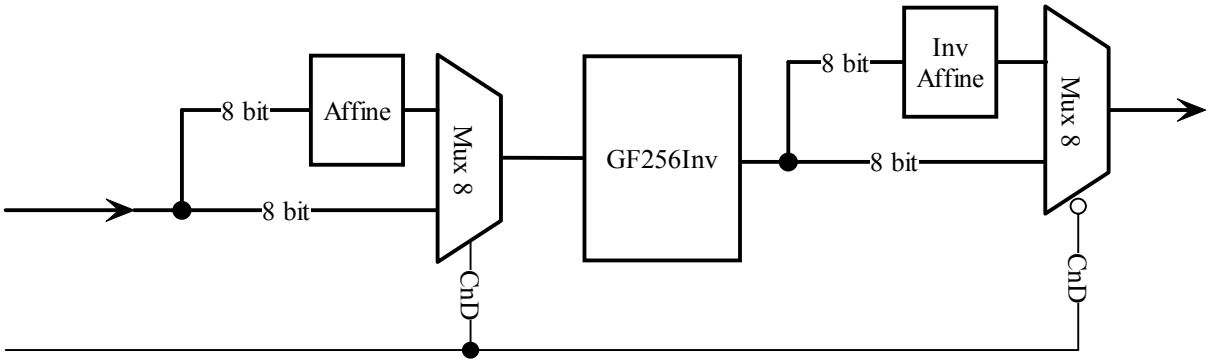


Figura 4.6 – Arquitetura para as tabelas S utilizando compartilhamento da inversão.

4.7.2 MixColumns e InvMixColumns

Outro conjunto de transformações que causa bastante impacto de área e de freqüência de operação no AES é o das operações MixColumns e InvMixColumns. Estas transformações são por construção [24] desbalanceadas quanto ao caminho crítico. Enquanto MixColumns é bastante simples e envolve apenas alguns *xors* entre elementos da entrada de 32 bits, InvMix-Columns utiliza muito mais operações de *xors*. Os dois trechos de código VHDL a seguir mostram a implementação das transformações.

```
library IEEE;
        use IEEE.Std_Logic_1164.all;

entity MixColumns is
        port(A          :       in              Std_Logic_Vector(31 downto 0);
             B          :       out             Std_Logic_Vector(31 downto 0));
end MixColumns;

architecture MixColumns_a of MixColumns is

        subtype Width8_Typ is Integer range 7 downto 0;
        subtype Vect8_Typ is Std_Logic_Vector(Width8_Typ);

        function XTime (s : Vect8_Typ) return Vect8_Typ is
                variable r_v : Vect8_Typ;
                begin
                        r_v(7) := s(6);
                        r_v(6) := s(5);
                        r_v(5) := s(4);
                        r_v(4) := s(3) xor s(7);
                        r_v(3) := s(2) xor s(7);
                        r_v(2) := s(1);
                        r_v(1) := s(0) xor s(7);
                        r_v(0) := s(7);
                        return r_v;
                end XTime;

        signal a0x_s, a1x_s, a2x_s, a3x_s     :       Vect8_Typ;
        signal A0_s, A1_s, A2_s, A3_s         :       Vect8_Typ;
begin
```

```
        A0_s <= A(31 downto 24);
        A1_s <= A(23 downto 16);
        A2_s <= A(15 downto 8);
        A3_s <= A(7 downto 0);

        a0x_s <= XTime (A0_s);
        a1x_s <= XTime (A1_s);
        a2x_s <= XTime (A2_s);
        a3x_s <= XTime (A3_s);

        B(31 downto 24) <= a0x_s xor a1x_s xor A1_s xor A2_s xor A3_s;
        B(23 downto 16) <= a1x_s xor a2x_s xor A0_s xor A2_s xor A3_s;
        B(15 downto 8)  <= a2x_s xor a3x_s xor A0_s xor A1_s xor A3_s;
        B(7 downto 0)   <= a3x_s xor a0x_s xor A0_s xor A1_s xor A2_s;

end MixColumns_a;
```

```
library IEEE;
        use IEEE.Std_Logic_1164.all;

entity InvMixColumns is
        port(A          :       in              Std_Logic_Vector(31 downto 0);
             B          :       out             Std_Logic_Vector(31 downto 0));
end InvMixColumns;

architecture InvMixColumns_a of InvMixColumns is

        subtype Width8_Typ is Integer range 7 downto 0;
        subtype Vect8_Typ is Std_Logic_Vector(Width8_Typ);


        function H9Time (s : Vect8_Typ) return Vect8_Typ is
                variable r_v : Vect8_Typ;
                begin
                        r_v(7) := s(4) xor s(7);
                        r_v(6) := s(3) xor s(6) xor s(7);
                        r_v(5) := s(2) xor s(5) xor s(6) xor s(7);
                        r_v(4) := s(1) xor s(4) xor s(5) xor s(6);
                        r_v(3) := s(0) xor s(3) xor s(5) xor s(7);
                        r_v(2) := s(2) xor s(6) xor s(7);
                        r_v(1) := s(1) xor s(5) xor s(6);
                        r_v(0) := s(0) xor s(5);
                        return r_v;
                end H9Time;

        function HbTime (s : Vect8_Typ) return Vect8_Typ is
                variable r_v : Vect8_Typ;
                begin
                        r_v(7) := s(4) xor s(6) xor s(7);
                        r_v(6) := s(3) xor s(5) xor s(6) xor s(7);
                        r_v(5) := s(2) xor s(4) xor s(5) xor s(6) xor s(7);
                        r_v(4) := s(1) xor s(3) xor s(4) xor s(5) xor s(6) xor s(7);
                        r_v(3) := s(0) xor s(2) xor s(3) xor s(5);
                        r_v(2) := s(1) xor s(2) xor s(6) xor s(7);
                        r_v(1) := s(0) xor s(1) xor s(5) xor s(6) xor s(7);
                        r_v(0) := s(0) xor s(5) xor s(7);
                        return r_v;
                end HbTime;


        function HdTime (s : Vect8_Typ) return Vect8_Typ is

                variable r_v : Vect8_Typ;
                begin
                        r_v(7) := s(4) xor s(5) xor s(7);
                        r_v(6) := s(3) xor s(4) xor s(6) xor s(7);
                        r_v(5) := s(2) xor s(3) xor s(5) xor s(6);
                        r_v(4) := s(1) xor s(2) xor s(4) xor s(5) xor s(7);
                        r_v(3) := s(0) xor s(1) xor s(3) xor s(5) xor s(6) xor s(7);
                        r_v(2) := s(0) xor s(2) xor s(6);
```

```
                        r_v(1) := s(1) xor s(5) xor s(7);
                        r_v(0) := s(0) xor s(5) xor s(6);
                        return r_v;
                end HdTime;

        function HeTime (s : Vect8_Typ) return Vect8_Typ is
                variable r_v : Vect8_Typ;
                begin
                        r_v(7) := s(4) xor s(5) xor s(6);
                        r_v(6) := s(3) xor s(4) xor s(5) xor s(7);
                        r_v(5) := s(2) xor s(3) xor s(4) xor s(6);
                        r_v(4) := s(1) xor s(2) xor s(3) xor s(5);
                        r_v(3) := s(0) xor s(1) xor s(2) xor s(5) xor s(6);
                        r_v(2) := s(0) xor s(1) xor s(6);
                        r_v(1) := s(0) xor s(5);
                        r_v(0) := s(5) xor s(6) xor s(7);
                        return r_v;
                end HeTime;

        signal A0_s, A1_s, A2_s, A3_s                :       Vect8_Typ;

begin

        A0_s <= A(31 downto 24);
        A1_s <= A(23 downto 16);
        A2_s <= A(15 downto 8);
        A3_s <= A(7 downto 0);



        B(31 downto 24) <= HeTime(A0_s) xor HbTime(A1_s) xor HdTime(A2_s) xor H9Time(A3_s);
        B(23 downto 16) <= H9Time(A0_s) xor HeTime(A1_s) xor HbTime(A2_s) xor HdTime(A3_s);
        B(15 downto 8)  <= HdTime(A0_s) xor H9Time(A1_s) xor HeTime(A2_s) xor HbTime(A3_s);
        B(7 downto 0)   <= HbTime(A0_s) xor HdTime(A1_s) xor H9Time(A2_s) xor HeTime(A3_s);


end InvMixColumns_a;
```

Observado a implementação de InvMixColumns fica claro que a responsabilidade pelo desbalanceamento é o intensivo uso das operações de *xor* entre os sinais. Parhi et al [76] sugere algumas possíveis otimizações na transformação, basicamente de agrupamento de operações, na esperança de ao menos minimizar o número de elementos lógicos utilizados.

Otimizações por aproveitamento de resultados intermediários também podem ser realizadas na implementação compondo as funções HeTime, H9Time, HdTime e HbTime por meio de somas de resultados e reaplicações da função XTime.

Ganhos de compartilhamento de estruturas também foram propostos por [76]. Entretanto, os resultados da síntese das transformações com qualquer uma das otimizações propostas por [76] ou tentadas nesta dissertação, como a por composição de XTimes, produziram o mesmo resultado.

	MixColumns		InvMixColumns	
Dispositivo/Item	Recursos	Atraso	Recursos	Atraso
EP1C20C6	192 LEs	2,68 ns	376 LEs (+96%)	3,68 ns (+37%)
EP2S15C3	152 ALUTs	2,37 ns	312 ALUTs (+105%)	3,01 ns (+27%)
EP20K200C7	216 LEs	3,24 ns	412 LEs (+91%)	4,44 ns (+37%)

Tabela 4.9 – Resultados para as transformações MixColumns e InvMixColumns para o *estado* inteiro.

O sintetizador foi capaz de simplificar, de acordo com a ênfase de área ou velocidade, as modificações tentadas para a mesma implementação. A Tabela 4.9 mostra os atrasos e consumo de elementos lógicos para a transformação de uma coluna por MixColumns e InvMixColumns para os três dispositivos alvo.

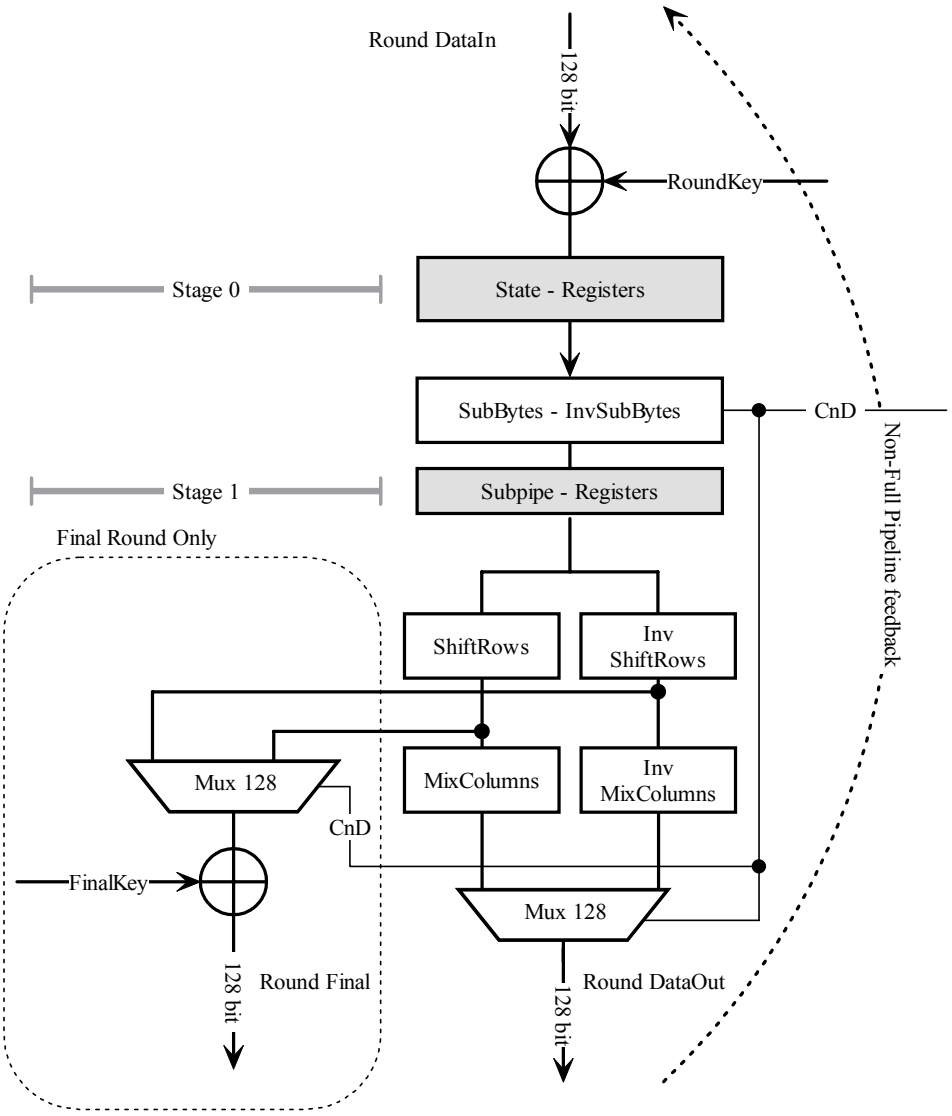


Figura 4.7 – O *Datapath* dual para o AES. A localização do subpipeline se justifica com os atrasos infligidos pela adição de round e pelo multiplexador na figura. A lógica de adição para o round final é apenas sintetizada naqueles rounds que podem servir de saída da arquitetura. Em uma versão *full pipelined* com suporte a apenas um tamanho de chaves, a área demarcada por "Final Round Only" é sintetizada uma única vez.

Os atrasos nas transformações MixColumn e InvMixColumns são consideravelmente menores do que aquelas de SubBytes, ao contrário do que [77] nos faz crer. Esta diferença de

atraso torna menos atrativa a utilização de um estágio de *subpipeline* na transformação de round quando somente o cifrador é sintetizado na arquitetura.

4.7.3 O Datapath

O *datapath* da arquitetura pode espelhar-se em basicamente dois modelos quando o modo de decifragem é implementado; a do decifrador ou do decifrador equivalente [24]. A principal vantagem de se utilizar o decifrador equivalente é que sua estrutura é mais parecida com a do cifrador, conferindo homogeneidade aos circuitos. O problema porem, é que a chave de round deve ser seletivamente utilizada com uma transformação MixColumns sobre todos os seus bytes, resultando em custos um pouco maiores de área. Estes são, no entanto, compensados pelo aumento de multiplexadores que a implementação conjunta do cifrador e decifrador original requer. A arquitetura implementada para o AES é mostrada na Figura 4.7.

Quando instanciado na versão *pipelined*, o *datapath* é replicado 2, 5, 7, 10 ou 14 vezes, dependo do tamanho da chave que a implementação aceite. O registrador de *subpipeline* da figura pode ou não ser sintetizado, assim como os circuitos de cifragem ou os decifragem, de acordo com opções de instanciação.

As transformações MixColumns e InvMixColumns não foram fundidas no projeto porque os testes realizados mostraram que não resulta em ganho.

4.7.4 A Máquina de Expansão de Chaves

Apesar de existirem vários estudos sobre implementações do AES em hardware poucos deles tratam da máquina de expansão de chaves, e menos ainda tratam de uma máquina para chaves maiores de 128 bits.

Paar et al [78], em um estudo comparativo sobre de candidatos ao AES, deixa de fora a análise de desempenho para a expansão de chaves do Rijndael. No entanto, como se pode verificar, a máquina de expansão pode conter o caminho crítico para o algoritmo do AES na decifragem em uma implementação não cuidadosa, o que pode resultar, para uma implementação com geração de chaves *on-the-fly*, menores taxas de cifragem do que aquelas reportadas.

Uma modificação simples de arquitetura, no entanto, permite que a unidade geração de chaves tenha o caminho crítico bastante reduzido. Mesmo que esta alteração possivelmente não afete o caminho crítico geral do AES, dá maior grau de liberdade para o sintetizador equalizar os demais atrasos do circuito.

A unidade de expansão de chaves do AES é comparativamente mais complexa e consome mais recursos do que a do DES, por exemplo. Isso se deve principalmente a não homogeneidade das transformações para derivação de subchaves para a versão de 256 bits e principalmente a de 192 bits.

A derivação para a versão de 192 bits utiliza quatro diferentes transformações de entrada e saída de 128 bits para as subchaves, dependendo do número da iteração, enquanto as versões de 128 e 256 bits usam somente uma e duas transformações respectivamente. Observa-

do que chaves de 192 não são muito utilizadas. O *core* VHDL produzido implementa as versões de 128 e 256 bits.

A arquitetura para a unidade de geração de chaves é mostrada na Figura 4.8. Fato importante é que os circuitos são capazes de gerar, com pequeno aumento de complexidade, chaves tanto para decifragem como para cifragem.

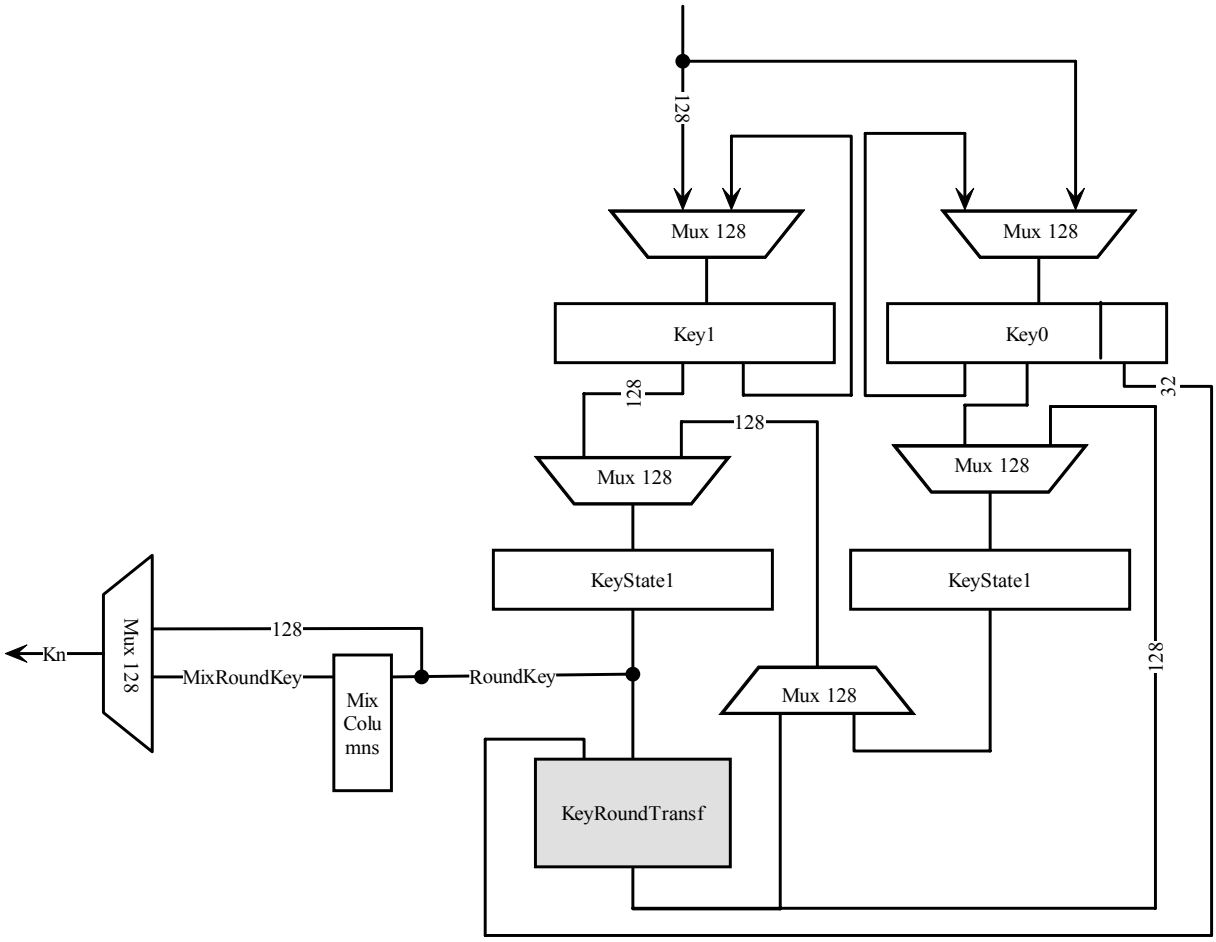


Figura 4.9 – Máquina de expansão de chaves de tamanhos 128 e 256 bits para o AES.

O componente KeyRoundTransf, na Figura 4.9, calcula a transformação de Round dependendo do tamanho da chave selecionada e da operação (cifragem ou decifragem). Ele pode conter um *subpipeline* dependendo da instanciação. Key1, Key0, KeyState1 e KeyState0 são registradores de 128 bits.

Quando o mecanismo é instanciado para um só tamanho de chaves, através do mecanismo de "GENERICS" do VHDL, as partes necessárias ao outro tamanho de chave são auto-

maticamente descartados pelo sintetizador. Diferentes combinações de instanciação do *core* resultam em performances variadas.

4.7.5 O Core AES

O paralelismo em nossa implementação em *hardware* do AES é explorado de duas maneiras diferentes, através do cálculo simultâneo das transformações sobre os 128 bits do bloco de dados e através de mecanismos de *pipeline* e *subpipeline*.

O resultado é um *core* altamente flexível através de parametrização, voltado para o desempenho extremo em sua versão de *pipeline* completo, mas com baixo consumo de recursos na versão básica. A declaração de componente do AES é dada a seguir.

```
component AES
        generic (Pipe_g              :       Natural := 14;
                 KeySize_g           :       String  := "BOTH";
                 SubPipe_g           :       String  := "YES";
                 Mode_g         :       String  := "DUAL");
        port (Clock                  :       in              Std_Logic;
                  Reset              :       in              Std_Logic;
                  CnD                :       in              Std_Logic;
                  KeySize            :       in              Std_Logic;
                  LoadData           :       in              Std_Logic;
                  LoadKey            :       in              Vect2_Typ;
                  DataIn             :       in              Vect128_Typ;
                  DataOut            :       out             Vect128_Typ);
end component;
```

O componente aceita qualquer uma das combinações dos "GENERICS" que tem como parâmetro: Pipe_g controla a profundidade do pipeline, em 1, 2, 5, 7, 10 ou 14 estágios. As profundidades de 10 e 14 representam *full pipelines* para os cifradores com chaves de 128 e 256 bits.

KeySize_g controla os tamanhos de chave que o *core* aceitará. O valor "BOTH" permite o sintetizador produzir circuitos para chaves de 128 e 256 bits. Para limitar para somente um valor de tamanho de chave, deve-se utilizar "128" ou "256".

SubPipe_g regula a síntese de um estágio de subpipeline no meio da transformação de round, afim de aumentar a freqüência máxima de operação do circuito. Os valores possíveis são "YES" e "NO".

Finalmente, Mode_g indica quais funções devem ser sintetizadas, "CIPHER" – cifragem apenas, "DECIPHER" – decifragem apenas, ou "DUAL" – ambas. As possibilidades de instanciação permitem um total de 108 diferentes configurações do *core*. Algumas destas combinações têm seus resultados de síntese mostrados nas Tabelas 4.10 e 4.11.

Na primeira coluna à esquerda de cada uma das tabelas está a configuração utilizada na síntese. A primeira posição reflete Mode_g, a segunda SubPipe_g, a terceira KeySize_g e a quarta Pipe_g.

Como observado, a freqüência máxima de operação dos circuitos diminui com o aumento do número de estágios do *pipeline* e com a inclusão do modo de decifragem. O desempenho para as arquiteturas, especialmente utilizando memória ROM para a transformação

SubBytes mostra resultados com altíssimo desempenho, resultando em taxas próxima de 7 Gbps para uma versão com *pipeline* de apenas 2 estágios mais subpipeline. Estes resultados são bem melhores que aqueles encontrados em [78], por exemplo.

Disp.	EP1C20C6		EP2S15C3		EP20K200C7	
Config	Recursos	F_{max} Taxa	Recursos	F_{max} Taxa	Recursos	F_{max} Taxa
C-Y-128-1	2956 LEs	103 MHz 2636 Mbps	2488 ALUTs	146 MHz 3738 Mbps	3368 LEs	70 MHz 1792 Mbps
C-Y-128-2	4159 LEs	103 MHz 5272 Mbps	4479 ALUTs	133 MHz 6808 Mbps	4601 LEs	60 MHz 3072 Mbps
Du-Y-B-1	3897 LEs	85 MHz 2176 Mbps	3587 ALUTs	130 MHz 3328 Mbps	4730 LEs	60 MHz 1536 Mbps
Du-Y-B-2	6789 LEs	87 MHz 4454 Mbps	5970 ALUTs	127 MHz 6468 Mbps	7881 LEs	58 MHz 2968 Mbps

Tabela 4.10 – Resultados de algumas configurações para o *core* utilizando a transformação SubBytes implementada por meio de inversão e transformada afim – alto desempenho.

Config.	LEs	RAM bits	F_{max} – Taxa
C-Y-128-1	1040	40960	150 MHz – 3840 Mbps
C-Y-128-2	1680	73728	136 MHz –6962 Mbps
C-N-128-1	912	40960	109 MHz – 1395 Mbps
C-N-128-2	1424	73728	98 MHz – 2509 Mbps
De-Y-128-1	1256	40960	140 MHz – 3584 Mbps
De-N-128-1	1128	40960	90 MHz – 1152 Mbps
Du-Y-B-1	2717	73728	124 MHz – 3168 Mbps
Du-N-B-1	2599	73728	70 MHz – 1792 Mbps

Tabela 4.11 – Resultados para o dispositivo EP20K200C7 utilizando para a implementação das tabelas S elementos de memória.

Freqüências bem mais altas de funcionamento podem ser observadas para os dispositivos da Tabela 4.10 se a implementação por ROM foi empregada também. Esta modificação não requer qualquer esforço adicional de codificação da arquitetura mas sim ajustes no sintetizador.

4.8 Arquiteturas para ECC

O custo da multiplicação escalar de pontos elípticos está diretamente relacionada com a operação de multiplicação e inversão de elementos do corpo base. Utilizando representação em bases normais, a adição de elementos do corpo pode ser realizada com um simples *xor* (soma

mod 2), e a exponenciação, por sua vez, pode ser calculada apenas com deslocamentos. Uma vez que com o método de multiplicação repetida [85] é possível expressar de maneira mais ou menos eficiente a inversão. Arquiteturas para ECC sobre bases normais devem focar nesta operação.

Desta forma, foi desenvolvido um co-processador para esta operação (ECCP), que pode ser instanciado junto ao *core* de um, processador RISC de 32 bits, como o NIOS [31], capaz de efetuar as demais operações e cuidar do controle da unidade e possivelmente dos demais mecanismos do cripto-processador.

Orlando e Paar [84] construíram um cripto-processador para curvas elípticas sobre corpos binários utilizando representação padrão, que apresenta um ótimo desempenho para hardware reconfigurável. Eles justificam a escolha de representação pelo fato que bases normais ótimas não existem para qualquer tamanho de corpo, e que, a utilização de bases de Tipos maiores representa aumento considerável de complexidade.

Entretanto, pode-se facilmente observar que na faixa de corpos de interesse criptográfico (173 a 519) 304 deles (87%) formam bases normais, dos quais menos de 23% representam corpos de grau maior que 7 [4]. O multiplicador serial descrito na seção 2.2.3 contém *m* termos operando sobre no máximo T + 1 dígitos binário, o que resulta em *m* termos operando sobre o máximo de 8 bits, justamente o tamanho de uma *look-up table* de FPGAs de última geração.

Isto permite que uma vasta gama de tamanhos de corpos sejam utilizados. Somando-se a esta questão tecnológica a possibilidade de paralelização e inclusão de pipeline de vários estágios no multiplicador, pode-se construir um acelerador para a multiplicação que opere em altas freqüências e a taxas arbitrárias.

4.8.1 A Arquitetura Global

Arquitetura construída para a unidade de multiplicação está ilustrada na Figura 4.10. A sua entrada e a sua saída são de 32 bits, para facilitar a integração com processadores e barramentos modernos. As unidades A e B são registradores de deslocamento circulares com carregamento paralelo e independente de 32 bits e de *m* posições, onde *m* é o tamanho do maior corpo a ser multiplicado pela unidade. Valores típicos de *m* são entre 173 e 519.

O registrador C é também um registrador de deslocamento, mas com carga serial em 2^n pontos, onde 2^n, é o número de réplicas da função de multiplicação serial *f*, definida matematicamente na seção 2.2.3.

A inclusão de mais cópias da função *f* implica em um aumento arbitrário do número de bits do produto $C = A \times B$, ($A, B, C \in GF(2^m)$) produzidos por ciclo de execução. Evidentemente tais bits devem ser inseridos uniformemente no registrador C. Chamaremos a estrutura de replicação das caixas *f* de *pipeline horizontal* – PH. Para um *pipeline horizontal* de tamanho 1, por exemplo, somente a caixa *f* de índice de saída c(0) ser utilizada. Para PH = 4, por exemplo, as caixas de índices c(0), c(m/4), c(m/2) e c(3m/4) são utilizadas. A assinatura VHDL do co-processador é dada a seguir:

```
component ECCP
        generic (HorizontalPipe_g      :       Natural := 8;
                 VerticalPipe_g        :       String := 3;
                 FieldSize_g           :       String := "ALL");
        port ( Clock            :       in              Std_Logic;
               Reset            :       in              Std_Logic;
               Go               :       in              Std_Logic;
               Reload           :       in              Std_Logic;
               NextWord         :       in              Std_Logic;
               FieldSize        :       in              Vect2_Typ;
               LoadData         :       in              Vect8_Typ;
               DataIn           :       in              Vect32_Typ;
               DataOut          :       out             Vect32_Typ);
end component;
```

O número de componentes *f* é controlado no tempo de instanciação do co-processador, por meio do "GENERIC" HorizontalPipe_g. Os valores possíveis para este parâmetro são 1, 2, 4, 8 e 16. O componente *f* é instanciado com os indicies dos sinais de entrada deslocados, para que forneçam corretamente os bits de saída, em concordância com a formula definida em 2.2.3.

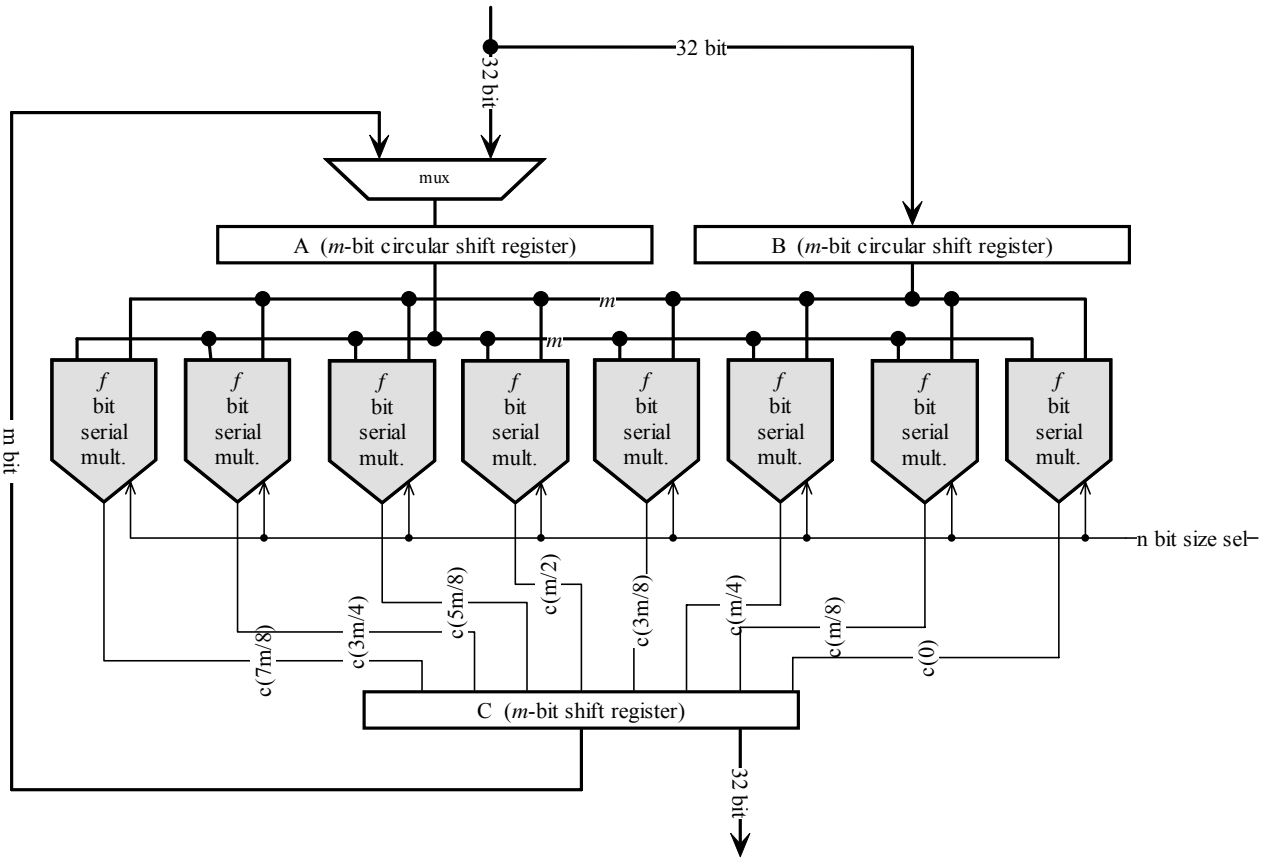


Figura 4.10 – Unidade de multiplicação com pipeline horizontal para corpos finitos binário com representação em bases normais.

Em ambientes, onde a utilização de mais de um tamanho de corpo é usada recorrentemente, até um limite de implementação de 8 tamanhos diferentes, as caixas *f* podem ser instanciadas de forma a aceitar o cálculo dos bits do produto para todos os corpos escolhidos.

Esta habilidade é controlada em tempo de síntese por meio da cláusula FieldSize_g, que pode ser “1..1”, “1..2”, “1..3”..., “1..7”, “ALL”. Desta forma, os corpos definidos como primeiro até o *n*-ésimo (“1..n”) serão inclusos na síntese. A correspondência entre corpos e índices é realizada através da cláusula VHDL de *configuração* [73].

4.8.2 As Caixas *f*

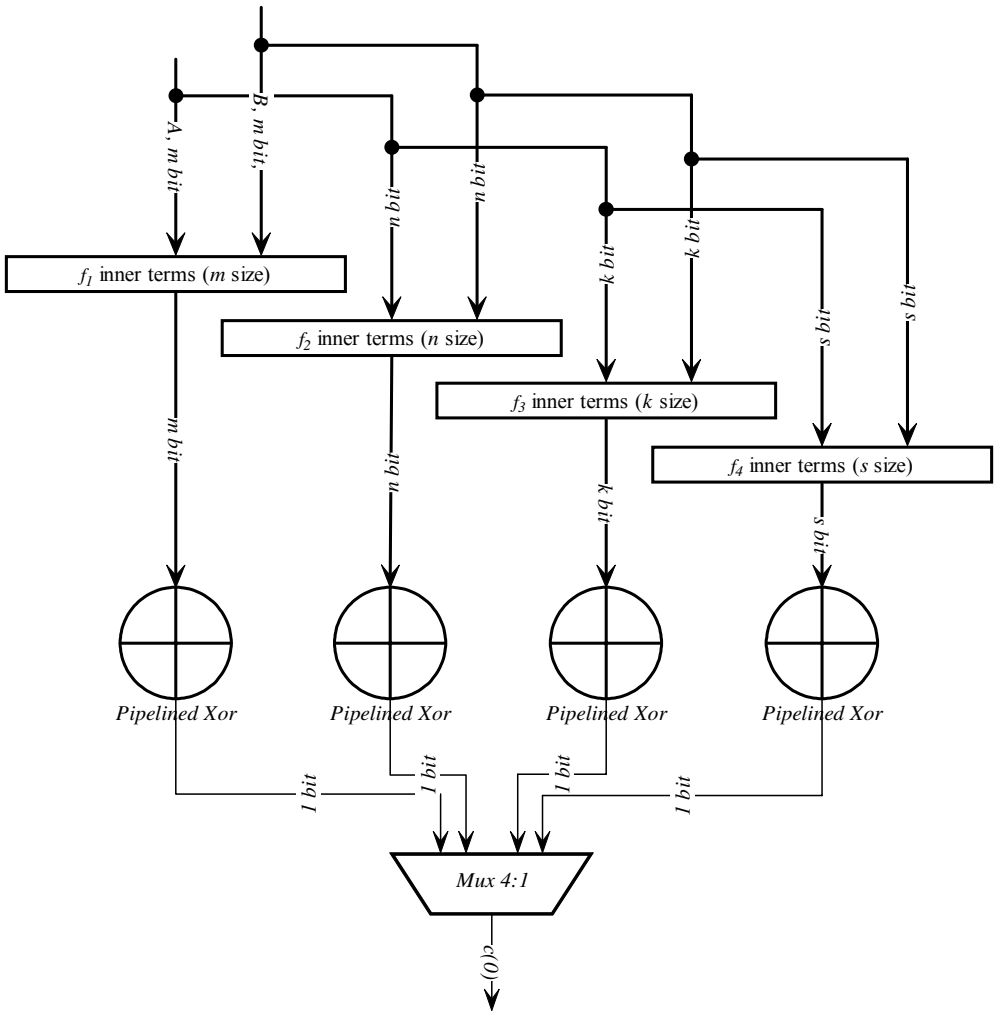


Figura 4.11 – Estrutura interna da Caixa *f*. Capacidade para o cálculo para mais de um tamanho de corpo.

A arquitetura para a *caixa f* é mostrada na figura 4.11, onde quatro tamanhos de corpos são incluídos. Nela, $m > n$, k e s. Os componentes "f_n *inner terms*" calculam os termos intermediários derivados da fórmula de 2.2.3, que tem forma a_i and (b_j xor b_l) para bases normais Tipos I e II. O número de corpos aceitos é definido pelo genérico FieldSize_g. Os termos calculados por "f_i *inner terms*" são então adicionados *mod 2* para gerar um bit do produto $C = A \times B$.

A unidade responsável por esta adição *xor* de todos os bits de entrada é "*Pipelined Xor*". Esta unidade perfaz a soma dos termos com uma estrutura de árvore. No interior desta arvore são registrados valores intermediários, constituindo o que chamaremos de *Pipeline Vertical*, PV. O número de estágios permitidos são 0, 1 e 3, regulados em tempo de síntese através do genérico VerticalPipe_g.

Uma análise superficial da Figura 4.11 pode nos levar a crer que a composição dos *Xor* não é interessante, uma vez que é uma estrutura completamente repetida, e que portanto poderia ser compartilhada, por meio de um multiplexador, pelas "f_i *inner terms*".

Em um projeto ASICs tal observação seguramente é verdadeira, já que além da replicação das portas *xor*, o número de registradores, elementos de alto custo [86], aumentaria em demasiado.

Ocorre que em FPGAs existe uma relação fixa entre funções lógicas e registradores. Na estrutura de adição em arvore o número de registradores é favorecido de tal sorte que a ativação dos estágios do *pipeline vertical* é quase gratuita. Os dados das Tabelas 4.12 e 4.13 ilustram o ocorrido.

Corpos	Config	LEs	IC	F_{max}	[bit KHz/LEs]
261 e 515	PV = 0	2448	7%	129 MHz	53
261 e 515	PV = 1	2553	7%	226 MHz	89
261 e 515	PV = 2	2706	7%	335 MHz	124
173, 261, 393 e 515	PV = 0	3109	12%	106 MHz	34
173, 261, 393 e 515	PV = 2	3529	11%	271 MHz	77

Tabela 4.12 – Utilização de recursos e performances para diferentes corpos utilizando a arquitetura sem compartilhamento de *Xorers* em uma FPGA EP1C20C6 do ECCP.

Corpos	Config	LEs	IC	F_{max}	[bit KHz/LEs]
261 e 515	PV = 0	2576	6%	132 MHz	51
261 e 515	PV = 1	2598	7%	228 MHz	88
261 e 515	PV = 2	2548	7%	306 MHz	120
173, 261, 393 e 515	PV = 0	3567	11%	108 MHz	30
173, 261, 393 e 515	PV = 2	3911	11%	258 MHz	66

Tabela 4.13 – Utilização de recursos com compartilhamento das operações de *xor*, utilizando o mesmo dispositivo, EP1C20C6.

4.8.3 Resultados e Análise.

As Tabelas 4.12 e 4.13 permitem observar que o aumento de elementos consumidos com a utilização de mais estágios de *pipeline* vertical é bastante pequeno e facilmente compensado pelo aumento da freqüência de operação dos circuitos. A relação entre geração de bits do resultado, freqüência de operação e elementos utilizados é claramente favorável à arquitetura escolhida como final para as caixas *f*.

Outro fator importante é a variação de recursos com o aumento do *pipeline* horizontal. A Tabela 4.14 apresenta os dados de uma instanciação do ECCP utilizando somente corpos de

173 bits, com *pipeline* vertical de um estágio e números variados de instancias da função *f*, para a FPGA EP1C20C6.

Corpo	Config	LEs	IC	F_{max}	[bit KHz/LEs]
173	PH = 1	776	1%	296 MHz	381
173	PH = 2	1033	2%	274 MHz	530
173	PH = 4	1544	4%	255 MHz	660
173	PH = 8	2562	8%	225 MHz	702

Tabela 4.14 – Performances para variações no *pipeline* horizontal – aumento da performance sublinear em uma FPGA EP1C20C6.

Para efeitos de comparação com o cripto-processador produzido por Orlando e Paar [84] e os dados de outras implementações, sintetizou-se o circuito para ECC para os corpos de 163 e 173 bits em uma FPGA de geração tecnológica semelhante àquela utilizada em [84], para 4 e 8 réplicas da função *f*. Os dados observados estão na Tabela 4.15, enquanto que na Tabela 4.16 pode-se observar os tempos de multiplicação de elementos do corpo de [84], cuja representação, é importante notar, é polinomial.

Os tempos conseguidos por nossa implementação em corpos de 163 e 173 bits são um dos melhores documentados no tempo de revisão desta dissertação.

Corpo	Config	LEs	F_{max}	[bit KHz/LEs]	Mult no Corpo
163	PH = 4, PV = 3	2148	197 MHz	367	207 ns
163	PH = 8, PV = 3	3807	178 MHz	374	114 ns
173	PH = 4, PV = 3	1584	238 MHz	601	183 ns
173	PH = 8, PV = 3	2652	224 MHz	676	98 ns

Tabela 4.15 – Tempos de multiplicação de elementos do corpo para a arquitetura proposta.

Corpo	Config	F_{max}	Mult no Corpo
167	D = 4	86 MHz	536 ns
167	D = 8	75 MHz	339 ns

Tabela 4.16 – Tempos de multiplicação de elementos do corpo para a arquitetura de Orlando e Paar utilizando a FPGA Xilinx Virtex E XCV400E8-BG432.

As diferenças das velocidades de multiplicação de elementos do corpo entre as arquiteturas podem ser explicadas se observamos que a estrutura do multiplicador construído é muito mais simples em termos de arquitetura do que aquela encontrada em [84] e desta forma possibilitou a inclusão de vários estágios de *pipeline* e minimização simples do caminho crítico.

É importante ressaltar que apesar da FPGA utilizada nos nossos testes ser de geração semelhante àquela empregada na outra implementação, diferenças tecnológicas entre as duas podem causar variações nas performances relativas.

Nesta dissertação, foi focada a implementação sobre corpos finitos binários com representações em bases normais. A experimentação com outras representações e transformações pode servir como possível extensão deste trabalho.

4.9 Práticas de Projeto Derivadas

Durante as pesquisas necessárias a esta dissertação, diversas práticas acerca da implementação de algoritmos de criptografia em hardware reconfigurável foram derivadas. As principais são apresentadas nesta seção, acompanhadas de sua motivação.

1. O Conhecimento da Estrutura do Dispositivo é Importante

Apesar de FPGAs ou CPLDs terem funcionalidades semelhantes entre si, cada geração de dispositivos e cada fabricante implementam de maneiras distintas os recursos disponíveis. É essencial para o projetista de hardware reconfigurável, que as particularidades de cada recurso das FPGAs ou CPLDs-alvo sejam dominados afim de que a implementação possa atingir os resultados desejados em todos os dispositivos.

Um aspecto muito importante de cada FPGA é a constituição de seus elementos lógicos. Usualmente os elementos lógicos dos dispositivos variam na quantidade máxima de bits de entrada das LUTs – *look-up tables* (tipicamente 4, 6 ou 8) e número de registradores que às compõem (normalmente 1 ou 2).

Em um dispositivo com LUTs de 4 bits, como os das famílias APEX e Cyclone, a implementação de qualquer função lógica de 4 ou menos bits de entrada consome apenas uma LE. Já a implementação de uma função de 5 a 7 variáveis consome dois LEs e tipicamente resulta em um atraso duas vezes maior do que no primeiro caso. Ter ciência destas propriedades é portanto algo importante durante o projeto.

Como outro exemplo podemos tomar a implementação das S-Boxes do AES nos dispositivos das famílias APEX, Stratix e Cyclone. Na primeira família, a existência de memória estática assíncrona interna à FPGA permite implementar estas tabelas inteiramente em RAM, como LUTs, independentemente do Estado, definido pelo algoritmo.

Esta tática de implementação, por outro lado, não pode ser imediatamente utilizada nas outras duas famílias de FPGAs já que elas somente possuem memórias síncronas: a opção é a implementação das tabelas junto aos bits do Estado ou a utilização dos próprios elementos lógicos.

2. Parametrização: Diferentes FPGAs, diferentes otimizações

O conhecimento da estrutura dos dispositivos-alvo permite que implementações mais genéricas sejam obtidas e que a compatibilidade entre dispositivos possa ser mantida com desempenho adequado.

No entanto, esta aproximação pode não aproveitar justamente o detalhe tecnológico que faz com uma FPGA seja particularmente melhor do que outra. Estes detalhes, como descritos no item anterior, permitem diferentes otimizações para diferentes dispositivos e podem fazer toda a diferença em uma implementação, a ponto de torná-la algumas vezes mais rápidas do que uma não otimizada.

Uma maneira de possibilitar que apenas um *core* otimizado seja construído para múltiplos dispositivos é através da *parametrização*. Em VHDL a parametrização pode ser conseguida através da utilização da clausula de GENERICS como feito nos *cores* produzidos nesta dissertação.

Além disso, o conhecimento avançado do sintetizador é importante, pois permite que através do ajuste de diversos parâmetros, a síntese mova recursos entre aqueles disponíveis no dispositivo e privilegie aquele de maior interesse (usualmente entre LEs e elementos de memória)

Como exemplo, tomemos a implementação das S-Box do AES para a família APEX de FPGAs. Através do uso de configurações especiais do sintetizador é possível, com relativa facilidade, mudar a implementação entre elementos lógicos (LEs) e blocos de memória, privilegiando o consumo de elementos ou freqüência de operação[27].

3. O Domínio Detalhado do Algoritmo Implementado Permite Flexibilidade

Algoritmos criptográficos são usualmente complexos. O conhecimento detalhado das transformações operadas pelo algoritmo pode permitir que diferentes táticas de implementação sejam utilizadas. Algumas destas são: (a) privilegiar operações que são mais baratas de serem realizadas em hardware reconfigurável do que em processadores de uso geral e; (b) explorar as formas de paralelismo.

Como exemplo, tomemos a análise do algoritmo de multiplicação serial para elementos dos corpos em ECC que permitiu observar que pela implementação padrão, os bits de saída do produto não dependem uns dos outros. Ou seja, não existe relação serial temporal entre eles. Isso encorajou a inserção de estágio de pipelines nas funções *f* apresentadas, diminuindo em muito o tempo de multiplicação e custando muito pouco em termos de elementos lógicos adicionais.

No caso do AES, o conhecimento do algoritmo permitiu que mais de dois tipos de otimizações algorítmicas fossem tentadas na implementação das S-Box, nominalmente na composição da inversão em $GF(2^8)$.

4. É Possível Explorar o Paradoxo das FPGAs

[27] Dependendo do tamanho da tabela, a utilização de elementos de memória para a implementação pode trazer benefícios em termos de atraso.

Em ASICs, registradores são elementos caros em termos de área, consomem algumas vezes mais transistores que portas lógicas simples. O roteamento de sinais, por sua vez, é usualmente barato e introduz poucos atrasos – apenas o de propagação nos fios.

A tecnologia de FPGAs e CPLDs pode introduzir por sua vez aquilo que podemos chamar do paradoxo das FPGAs: a adição de registradores pode ter custo ínfimo enquanto o roteamento pode ser crítico.

Essa situação é extrema é fruto da estrutura fixa das LEs destes dispositivos. Uma vez que a relação entre LUTs e registradores é fixa, a utilização maciça de lógica combinacional pode deixar diversos registradores não utilizados e inutilizáveis. Se a não utilização é fruto da própria função implementada, a impossibilidade de uso é resultado da estrutura da própria FPGA.

No entanto, com otimizações algorítmicas e arquiteturais é possível fazer uso destes registradores, com impacto baixo ou nulo nos recursos consumidos. Estes registradores podem ser especialmente utilizados na introdução de *pipelines* naqueles algoritmos sem *feedback*, tal qual feito para o ECCP.

Já o roteamento dos sinais em dispositivos reconfiguráveis, ao contrário do caso dos ASICs, passa por junções semicondutoras, em virtude destes canais de roteamento não serem fixos. Esta construção introduz atrasos apreciáveis e deve ser levado em conta no design.

O atraso no roteamento é especialmente difícil de se prever pelo projetista e é um problema NP-Completo: o sintetizador usualmente não é capaz de encontrar solução ótima. Felizmente a introdução de *pipelines* tende a equalizar os atrasos dos caminhos e por conseqüência facilitar o roteamento.

5. O Grau de Liberdade Alto no Roteamento Melhora a Performance

O problema de roteamento é complexo e deve ser re-executado a cada síntese. Alguns compiladores utilizam técnicas não determinísticas para o roteamento e que geram resultados diferentes a cada vez que é executada.

Para melhorar a qualidade do roteamento, a diminuição do caminho crítico deve ser um objetivo local de *design*. Em outras palavras, mesmo caminhos que não representam o caminho crítico global do circuito, devem ser minimizados pois ajudam a concentrar os esforços do sintetizador no caminho crítico global através do aumento do grau de liberdade para a ferramenta.

No AES, por exemplo, o *datapath* apresenta o caminho crítico. No entanto, a máquina de expansão de chaves contém um atraso que é quase tão grande quanto o primeiro. Durante a construção do *core* foi testado a otimização do atraso da máquina de expansão de chaves, rendendo resultados até 10% maiores na freqüência de operação, mesmo com este subsistema não fazendo parte do caminho crítico.

Outra maneira de se manter o grau de liberdade alto é através da utilização de arquiteturas de baixo *fan-in* e *fan-out* nos componentes, que diretamente influem no número de caminhos a serem analisados pela ferramenta.

6. A Experimentação é uma Ferramenta Importante

O resultado exato da síntese em dispositivos reconfiguráveis é bastante imprevisível em tempo de projeto. Enquanto o conhecimento da estrutura básica das FPGAs é importante, a experimentação é essencial. Muitas vezes, soluções que parecem ser melhores sob o ponto de vista do design tradicional para ASICs podem não ser boas no caso específico do hardware reconfigurável.

A experimentação deve ser uma prática de otimização. Quando acompanha por boas práticas de projeto, como modularização, permitem grandes ganhos de performance e recursos nas FPGAs de maneira descomplicada.

7. Confiar Pouco ao Sintetizador

Os sintetizadores podem operar diversas otimizações entre estados registrados, isto é, sobre lógica combinacional, mas quase nenhuma através de estados. A tarefa de otimização de funções lógicas pode então ser inteiramente deixada a cargo do sintetizador para em funções de baixo *fan-in.* Ele normalmente é capaz de achar o resultado ótimo nestes casos.

Este fato pôde ser comprovado na implementação das diversas otimizações propostas para o AES nas transformações MixColumns e InvMixColumns por [76]. Funções de alto *fan-in* e *fan-out*, por sua vez, devem ser observadas de perto pelo projetista, como aquelas representas pelas S-Box do AES.

Otimizações através de estados registrados devem ser inteiramente acompanhas pelo projetista pois não podem ser otimizadas pelo sintetizador.

Capítulo 5
Conclusão e Trabalhos Futuros

Nesta dissertação foram apresentados os resultados de pesquisa e desenvolvimento para diversos itens necessário à construção de um cripto-processador com tecnologia nacional.

Uma ampla pesquisa bibliográfica foi realizada cobrindo detalhes de mecanismos criptográficos, protocolos para interfaces e arquiteturas para dispositivos reconfiguráveis.

Vários dos temas pesquisados foram implementados utilizando técnicas documentadas na literatura. Algumas destas técnicas se mostraram inócuas, enquanto outras, de otimização de arquiteturas, foram propostas e testadas, em especial para curvas elípticas.

Dos mecanismos de criptografia abordados, o AES, o DES e o Co-Processador de Curvas Elípticas – ECCP receberam especial atenção e para eles foram produzidos componentes VHDL parametrizados de alto desempenho que serão publicados sob licença LGPL no repositório Brazil IP [30]. Os resultados para o ECCP representam alguns dos melhores documentados no momento.

A implementação de uma API facilmente integrável ao padrão PKCS#11 permitiu que diversas funcionalidades fossem testadas e simuladas em uma placa com interface PCI. O gerador de números aleatórios construído, por exemplo, foi testado e utilizado através desta interface, o que permitiu sua análise estatística.

Algumas sugestões, ao longo desta dissertação, foram evidenciadas como possíveis temas de pesquisa futuros, como a utilização de adaptações nas instruções de processadores de uso geral para comportar algoritmos criptográficos.

No que se refere à criptografia de curvas elípticas, as arquiteturas desenvolvidas devem passar a incluir a multiplicação escalar kP e posteriormente até protocolos completos, para os corpos atualmente recomendados.

Um outro caminho para trabalhos futuros é o de integração do cripto-processador projetado com a placa PCI utilizada, tendo por objetivo a construção de um protótipo de *Hardware Security Module* com tecnologia nacional.

Evidentemente, nem todos as tecnologias necessárias à construção desse HSM foram abordadas. Algumas delas simplesmente por não pertencerem ao escopo de pesquisa, como resistência a ataques físicos e interferências eletromagnéticas. Outras, como a integração do sistema e as implementações de algoritmos de multiplicação e adição modular de inteiros não foram tratadas por falta de tempo.

Referências Bibliográficas

[1] ANSI X9.62-1999. *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, January 1998. Approved January 7, 1999.
[2] Micahel Rosing – *Implementing Elliptic Curve Cryptography* – Manning, 1999
[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest - *Introduction to Algorithms* The MIT Electrical Engineering and Computer Science Series MIT Press - Cambridge, Mass.
[4] IEEE, *IEEE P1363. Standard Specifications for Public Key Cryptography*, IEEE press, 1998.
[5] J. Orlin Grabbe - *The DES Illustrated* em www.zolatimes.com/V2.28/DES.html, Agosto de 1999.
[6] Bruce Schneier, - *Applied Cryptography, Second Edition*, John Wiley & Sons, New York, 1996.
[7] Douglas L. Perry - *VHDL*, 2nd ed.McGraw-Hill, 1994
[9] Altera Corporation - *University Program Design Laboratory Package - User Guide*. ver. 11997
[10] John F. Wakerly - *Digital Design, Principle & Practices, Second Edition,* Pretice Hall, 1994
[11] Julio Lopéz e Ricardo Dahab, – An *Overview of Elliptic Curve Cryptography*, Unicamp, Maio 2000
[12] Julio Lopéz e Ricardo Dahab – *High Speed Software Multiplication in* F_{2m}, Unicamp, Maio 2000
[13] Julio Lopéz e Ricardo Dahab – *Performance of Elliptic Curve Cryptosystens*, Unicamp, Maio 2000
[14] Rogério Miranda e Ricardo Dahab –*Dissertação de Mestrado*, Unicamp, Maio 2001-06-10
[15] Joan Daemen e Vincent Rijmen – *AES Proposal : Rijndael*
[16] V.K. Dandalis A., J.D. Rolim , *A Comparative Study of Performance of AES Final Candidates Using FPGAs*, Proc. Cryptographic Hardware and Embedded Systems Workshop CHES 2000, 2000.
[17] P. Mroczkowski, *Implementation of the block chiper Rijndael using Altera FPGA*, Public Commensts on AES Candidate Algorithms – Round 2, 2000.
[20] PLX 9052 Datasheet http://www.plxtech.com/products/9052/briefs/9052.pdf
[21] USB 2.0 Specification http://www.usb.org/developers/usb20
[22] David Becker – *Group expands USB Standard*, CNET Newas.com, dezembro de 2001.
[23] Apple Firewire brief, http://www.apple.com/firewire/
[24] Vincent Rijmen - *Efficient Implementation of the Rijndael S-box*, Katholieke Universiteit Leuven, Belgica, Janeiro 2003
[25] P. Dusart, G. Letourneux and O. Vivolo, *Differential Fault Analysis on A.E.S.*, Janeiro 2003 - Cryptology ePrint
[26] Christophe Giraud, *DFA on AES*, Janeiro 2003 - Cryptology ePrint

[27] Vipul Gupta, Sumit Gupta, Sheueling Chang, *Performance Analysis of Elliptic Curve Cryptography for SSL*, Sun Microsystems
[28] Scott Durrant, Ra*ndom Numbers in Data Security Systems*, Intel Platform Security Division
[29] PCI-SIG PCI *Local Bus Specification, revision 2.1*, Dezembro de 1998.
[30] Motivations for the Brazilian IP Repository - www.brazilip.org
[31] Altera Corporation, *Nios Datasheet*, Altera Corporation
http://www.altera.com/literature/ds/ds_nioscpu.pdf
[32] RFC 3278, *Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)*
[33] Simon Blake-Wilson, *SEC 1: Elliptic Curve Cryptography*, Certicom Research
[35] NIST FIPS 46-2, *Data Encryption Standard*.
[36] NIST FIPS 186, *Digital Signature Standard.*
[37] NIST, *SP 800-38A 2001 ED, Recommendation for Block Cipher Modes of Operation*, Dezembro 2001.
[38] Donald E. Knuth, *The Art of Computer Programming, Volume 2-/ Seminumerical Algorithms*. Reading MA: Addison-Wesley, 1969.
[39] NIST FIPS 197, *Advanced Encryption Standard*
[40] NIST FIPS 180-2, *Secure Hash Algorith.*
[41] RSALabs PKCS#11, *Cryptographic Token Interface Standard.*
[42] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press 5th ed., 1996.
[44] Whitfield Diffie, Martin E. Hellman*, New Directions in Cryptography*, IEEE Transactions on Information Theory, 1976.
[45] NIST FIPS 46-3, *Data Encryption Standard.*
[46] Ron Rivest, Adi Shamir and Len Adleman, *A method for obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, 1978 pag. 120-126.
[47] NIST FIPS 140-2, *Security Requirements for Cryptography Modules.*
[48] RSALabs PKCS#15, *Cryptographic Token Information Format Standard*.
[49] Jorge Nakahara Jr, Vicent Rijmen, Bart Preneel e Joos Vanderwalle, *The MESH block ciphers*.
[50] Christof Paar, *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*, PhD Thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.
[52] Tom Wada, *SubBytes Transform circuit for AES Cipher*, University of the Ryukyus, Information Engineering Dept, notas de classe.
[53] Hanner Brunner, Andreas Curiger e Max Hostetter, *On Computing Multiplicatives Inverses on GF(2^m)*, IEEE Transactions on Cumputers, Vol 42, No 2, agosto de 1993, pag 1010 a 1015.
[54] Niels Ferguson and Bruce Schneier, *Practical Cryptography*, John Wiley & Sons, 2003.
[55] Ross Anderson, Eli Biham e Lars Knudsen, *The Serpent Block Cipher*, original submission.

[56] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, *Twofish: A 128-Bit Block Cipher*, 15 de junho de 1998.
[57] Ronald L. Rivest1, M.J.B. Robshaw2, R. Sidney2, and Y.L. Yin2, *The RC6 Block Cipher*, 20 de agosto de 1998.
[59] Electronic Frontier Foundation, *Cracking DES Secrets of Encryption Research, Wiretap Politics & Chip Design,* julho de 1998.
[61] R. Rivest, *MD5 Algorithm*, MIT Laboratory for Computer Science e RSA Data Security, Inc., Abril de 1992.
[62] IEEE Standard 1394b, *"High-Performance Serial Bus"*, abril de 2002.
[63] PC/104 Embedded Consortium, *PC/104 Specification Version 2.4*, agosto de 2001
[64] Eli Biham. *New types of cryptanalytic attacks using related keys.* In Advances in Cryptology EUROCRYPT '93, volume 675, pages 398–409, 1994.
[65] *On the Security of PKCS #11.* Jolyon Clulow. University of Natal, Department of Mathematical and Statistical Sciences, Durban.
[66] Texas Instruments, *Hex Inverters Datasheet*, Dezembro de 1994.
[68] John Walker, *ENT, A Pseudorandom Number Sequence Test Program*, http://www.fourmilab.ch/random/, 1998.
[69] George Marsaglia, *DIEHARD: a battery of tests for random number generators*, http://stat.fsu.edu/~geo/diehard.html
[70] Francisco Corella, *A fast implementation of DES and Triple-DES on PA-RISC 2.0*, Hewlett Packard Co.
[72] Jens-Peter Kaps, Christof Paar, *Fast DES Implementation for FPGAs and Its Application to a Universal Key-Search Machine*, Selected Areas in Cryptography 1998: 234-247
[73] Ben Cohen, *Vhdl Coding Styles and Methodologies*, Kluwer Academic Publishers; 2nd Book and CD-ROM edition, abril de 1999.
[74] Helion Tech, *High Performance SHA-256 Hash Core for Xilinx FPGA*.
[76] Xinmiao Zhang e Keshab K. Parhi, *Implementation Approaches Implementation Approaches*, IEEE Circuits and Systems Magazine Volume 2, Number 4, Fourth Quarter 2002.
[77] Tsung-Fu Lin, Chih-Pin Su, Chih-Tsun Huang, and Cheng-Wen Wu, *A High-Throughput Low-Cost AES Cipher Chip*, Laboratory for Reliable Computing, Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan.
[78] AJ Elbirt, W Yip, B Chetwynd, C Paar, *An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*, Electrical and Computer Engineering Department Worcester Polytechnic Institute.
[79] Zanger, H. 1984. *Semiconductor Devices and Circuits.* John Wiley and Sons.
[80] Vergers, C. 1987. *Handbook of Electrical Noise.* TAB Books, Blue Ridge Summit, PA
[81] Projeto ChameLeon – http://www.lsc.ic.unicamp.br
[82] Altera Corporation, http://www.altera.com
[83] Quartus II Software, http://www.altera.com/products/software/pld/products/q2/qts-index.html
[84] Gerardo Orlando and Christof Paar, *A High-Performance Reconfigurable Elliptic*

Curve Processor for GF(2^m), Workshop on Cryptographic Hardware and Embedded Systems (CHES) 2000, Springer-Verlag, Lecture Notes in Computer Science 1965, 2000.
[85] T. Itoh and S. Tsujii. *A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases.* Information and Computation, 78:171{177, 1988}.
[86] David Patterson e John Hennessy, *Computer Organization and Design Second Edition: The Hardware/Software Interface*, Morgan Kaufmann; 2a edição, agosto de 1997.
[87] GNU Lesser General Public License, http://www.gnu.org/copyleft/lesser.html, Version 2.1, fevereiro de 1999.
[88] Jorge Guajardo e Christof Paar, *Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes*. Kluwer Academic Publishers, 2001.