UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Desenvolvimento em Linguagem de Descrição de Hardware de Codificador e Decodificador Reed-Solomon

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a obtenção
do grau de Mestre em Engenharia Elétrica

Tomás Grimm

Orientador: Danilo Silva
Coorientador: Eduardo Augusto Bezerra

Florianópolis, 20 de fevereiro de 2014.

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Grimm, Tomás
Desenvolvimento em Linguagem de Descrição de Hardware de Codificador e Decodificador Reed-Solomon / Tomás Grimm ; orientador, Danilo Silva ; coorientador, Eduardo Augusto Bezerra. - Florianópolis, SC, 2014.
142 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. Códigos Corretores de Erros. 3. Reed-Solomon. 4. Hardware Reconfigurável. 5. FPGA. I. Silva, Danilo. II. Bezerra, Eduardo Augusto. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

Tomás Grimm

DESENVOLVIMENTO EM LINGUAGEM DE DESCRIÇÃO DE HARDWARE DE CODIFICADOR E DECODIFICADOR REED-SOLOMON

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, área de concentração Sistemas Embarcados, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Patrick Kuo Peng, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca examinadora

Prof. Danilo Silva, Ph.D. (Orientador)
Universidade Federal de Santa Catarina

Prof. Bartolomeu Ferreira Uchôa Filho, Ph.D.
Universidade Federal de Santa Catarina

Prof. Raimes Moraes, Ph.D.
Universidade Federal de Santa Catarina

Prof. Letícia Maria Bolzani Poehls, Ph.D.
Pontifícia Universidade Católica do Rio Grande do Sul

Dedico este trabalho à minha esposa
por sempre estar ao meu lado.

Agradecimentos

Agradeço primeiramente aos meus Professores Orientadores Danilo Silva, por me ajudar a construir este trabalho através da sua orientação sempre edificante, e Eduardo Augusto Bezerra, por fornecer os meios e a ajuda necessária para que a parte prática deste trabalho pudesse ser implementada.

Agradeço à minha namorada/noiva/esposa Eliane por sempre estar ao meu lado e nunca me deixar esmorecer e, principalmente, pelo seu amor, carinho e paciência de aprender a teoria dos códigos corretores de erros.

À minha família que sempre acreditou no que eu sou capaz de alcançar e sempre me apoiou.

Agradeço aos Professores Bartolomeu, Raimes e Letícia por fazerem parte desta importante etapa da minha vida, não apenas acadêmica, mas de crescimento em diversos sentidos.

E também ao Professor Djones Lettnin que, durante o curso destes dois anos, me ajudou a crescer e me mostrou diversas oportunidades para a continuidade da minha empreitada.

Aos meus colegas e amigos do LCS, por terem me ajudado de alguma forma, principalmente a desestressar durante todo esse período.

Agradeço especialmente ao aluno João Pedro dos Reis, por me auxiliar com a validação deste trabalho e por me aguentar tanto tempo.

Agradeço à UFSC e ao CNPq por fornecerem os meios e o apoio

financeiro suficiente para realizar e completar esta pesquisa acadêmica.

Finalmente, mas não menos importante, agradeço à Deus a minha vida e esta magnífica oportunidade de conhecer pessoas maravilhosas e auxiliar no crescimento de algumas delas.

"E conhecereis a verdade,
e a verdade vos libertará."
— João 8,32

Resumo

Atualmente, diversos sistemas de comunicação demandam grandes volumes de tráfego de dados para consumo quase instantâneo. Estes dados devem ser entregues aos usuários tal qual foram gerados: sem erros. Por isso, técnicas de controle e correção de erros estão intrinsecamente ligadas aos sistemas que realizam trocas de dados, sejam sistemas de armazenamento, os quais estão sujeitos a falhas durante a leitura, ou sistemas de comunicação, que estão sujeitos às adversidades do meio (radiação, interferência eletromagnética, desvanecimento, entre outros). Neste cenário, os códigos Reed-Solomon representam uma solução viável para inúmeras aplicações, bem como pesquisas acadêmicas, mesmo tanto tempo após sua invenção. Este trabalho realiza um estudo da teoria que embasa os códigos Reed-Solomon, assim como implementa as técnicas do estado-da-arte dos módulos que compõem tanto o codificador quanto o decodificador, as quais são prototipadas em *hardware* reconfigurável.

Palavras-chave: Códigos Corretores de Erros, Reed-Solomon, *Hardware* Reconfigurável, FPGA, VHDL

Abstract

Nowadays, many communication systems demand a high volume of data traffic for instant consumption, and this data must be delivered to the users the same way they were generated: error-free. For this reason, error control coding techniques are so intrinsically connected to communication systems, be it storage systems, which are subject to reading failures, or transmission systems, which are subject to the channel's impairments (radiation, electromagnetic interference, fading, to name a few). In this scenario, Reed-Solomon codes are widely used and still researched by the scientific community, even so much time after their invention. This work performs a study of the theory behind Reed-Solomon codes, as well as implements the state-of-the-art techniques for the modules that are part of both the encoder and the decoder, so that they can be prototyped using reconfigurable hardware.

Keywords: Error Control Coding, Reed-Solomon, Reconfigurable Hardware, FPGA, VHDL

Lista de Figuras

2.1 Ilustração do funcionamento do decodificador de mínima distância 10
2.2 Ilustração do funcionamento do decodificador de distância limitada 11

3.1 Bloco Codificador/Decodificador da Altera 36
3.2 Bloco Codificador do ISE 39
3.3 Bloco Decodificador do ISE 40
3.4 Bloco Codificador do Vivado 41
3.5 Bloco Decodificador do Vivado 42
3.6 Bloco Codificador da Lattice 44
3.7 Bloco Decodificador da Lattice 45
3.8 Bloco Decodificador da Factorial 47
3.9 Bloco Codificador da implementação *open source* da System LSI. 48
3.10 Bloco Decodificador da implementação *open source* da System LSI. 49

4.1 Representação do somador 55
4.2 Representação do multiplicador polinomial 55

4.3 Exemplo de multiplicador polinomial para $m = 3$ e $p(x) = x^3 + x + 1$ 57
4.4 Codificador sistemático 58
4.5 Interface do Codificador 59
4.6 Arquitetura interna do decodificador 60
4.7 Interface do Decodificador 60
4.8 Circuito de cálculo da síndrome 62
4.9 Interface do bloco de cálculo da síndrome 62
4.10 Interface do bloco de determinação dos polinômios localizador e avaliador de erros 63
4.11 Célula de atualização dos coeficientes de $\hat{\delta}^{(j)}(x)$ 64
4.12 Arquitetura sistólica do algoritmo RiBM 65
4.13 Célula de atualização dos coeficientes de $R(x)$ 66
4.14 Célula de atualização dos coeficientes de $\lambda(x)$ 67
4.15 Arquitetura sistólica do algoritmo E-DCME 68
4.16 Arquitetura do bloco de determinação das raízes de $\sigma(x)$ e cálculo da magnitude dos erros 69
4.17 Interface do bloco de determinação das raízes de $\sigma(x)$ e cálculo da magnitude dos erros 69
4.18 Interface do bloco de determinação do polinômio dos apagamentos 70

5.1 Exemplo do processo de codificação 78
5.2 Detalhe dos símbolos de paridade 79
5.3 Exemplo da decodificação sem erros na palavra recebida 80
5.4 Exemplo da decodificação com 8 erros na palavra recebida 81
5.5 Exemplo da decodificação com 10 erros na palavra recebida 82
5.6 Exemplo da decodificação com 16 apagamentos na palavra recebida 84
5.7 Exemplo da decodificação com 17 apagamentos na palavra recebida 85
5.8 Exemplo da decodificação com 4 erros e 8 apagamentos na palavra recebida 86
5.9 Exemplo da decodificação com 7 erros e 2 apagamentos na palavra recebida 87
5.10 Exemplo da decodificação com 1 erros e 16 apagamentos na palavra recebida 88

5.11 Exemplo da decodificação com 4 erros e 9 apagamentos na palavra recebida 89
5.12 Fluxo de comunicação da rotina de validação da arquitetura desenvolvida para o codificador 90
5.13 Fluxo de comunicação da rotina de validação das arquiteturas desenvolvidas para o decodificador 91

6.1 Conceito do funcionamento do decodificador com *pipeline* 101

Lista de Tabelas

3.1 Comparação das funcionalidades oferecidas por cada arquitetura 50
3.2 Sinais de interface de cada arquitetura 50
3.3 Possíveis configurações das arquiteturas comerciais. . . . 51

4.1 Atraso dos submódulos do decodificador 71

5.1 Configuração do código desenvolvido 74
5.2 Ocupação do Codificador 92
5.3 Ocupação de cada submódulo do Decodificador utilizando o algoritmo RiBM 92
5.4 Ocupação de cada submódulo do Decodificador utilizando o algoritmo E-DCME 92
5.5 Frequência de operação estimada para cada arquitetura 93
5.6 Dados de área e frequência para FPGAs da Altera . . . 94
5.7 Dados de área e frequência para FPGAs da Xilinx . . . 94
5.8 Dados de área e frequência para FPGAs da Lattice . . . 95
5.9 Dados de área e frequência para a arquitetura da System LSI 95
5.10 Dados de área e frequência para a arquitetura da Factorial 95

5.11 Parâmetros de entrada da ferramenta de geração automática da arquitetura 97
5.12 Possíveis configurações dos parâmetros do código RS . . 97
5.13 Exemplos de polinômios primitivos para corpos finitos binários 98

Lista de Abreviaturas e Siglas

ASIC	Application Specific Integrated Circuit
BCH	Bose-Chaudhuri-Hocquenghem
BM	Berlekamp-Massey
CAM	Content-addressable Memory
CD	Compact Disc
DCME	Degree Computationless Modified Euclidean algorithm
E-DCME	Enhanced Degree Computationless Modified Euclidean algorithm
FPGA	Field Programmable Gate Array
GAPH	Grupo de Apoio ao Projeto de Hardware
GSE	Grupo de Sistemas Embarcados
iBM	inversionless Berlekamp-Massey
IP	Intellectual Property
LFSR	Linear Feedback Shift Register

LUT	look-up table
ME	Modified Euclidean Algorithm
PUCRS	Pontifícia Universidade Católica do Rio Grande do Sul
RiBM	Reformulated inversionless Berlekamp-Massey
RS	Reed-Solomon
SAGE	System for Algebra and Geometry Experimentation
VHDL	Very High Speed Integrated Circuits Hardware Description Language

Sumário

1 Introdução **1**
1.1 Objetivos 3
1.2 Estrutura do Texto 4

2 Código Reed-Solomon **5**
2.1 Conceitos básicos 5
2.2 Códigos Reed-Solomon 15
2.3 Decodificação 17
2.4 Métodos de determinação dos polinômios localizador e avaliador de erros 24

3 Arquiteturas Comerciais **35**
3.1 Reed-Solomon II MegaCore Function da Altera 35
3.2 Reed-Solomon LogiCORE IP da Xilinx 37
3.3 Reed-Solomon Dynamic Block da Lattice 42
3.4 Reed-Solomon Decoder IP Core da Factorial 44
3.5 IP Core Open Source da System LSI 47
3.6 Sumarização das arquiteturas comerciais 49

4 Arquitetura Proposta **53**
4.1 Metodologia de desenvolvimento 53
4.2 Operações em corpos finitos 54

4.3 Codificador 58
4.4 Decodificador 59
4.5 Atrasos de operação 71

5 Desenvolvimento e Avaliação 73
5.1 Definição do código 73
5.2 Validação da arquitetura 76
5.3 Avaliação 91
5.4 Ferramenta de geração automática da arquitetura 96

6 Conclusão 99

A Álgebra Abstrata 103
A.1 Grupos e Subgrupos 103
A.2 Corpos 104
A.3 Corpos de Extensão 105
A.4 Espaços Vetoriais 109

CAPÍTULO 1

Introdução

Atualmente, é tão simples fazer *upload* de uma fotografia para uma rede social ou consultar as condições de tráfego ou do clima a partir de um *smartphone*. Assim, é difícil associar o conceito de erro de comunicação a estas atividades. Porém, sem técnicas de correção de erros, tais processos seriam impossíveis de serem executados por meio de comunicação sem fio, já que o ambiente está constantemente sujeito a ruídos. Por este motivo, técnicas de codificação para controle de erros (através da adição de informação redundante aos dados originais [26]) são utilizadas em sistemas de comunicação, assim como em dispositivos de armazenamento, para garantir a proteção dos dados e torná-los mais seguros contra erros quando os mesmos necessitarem ser transmitidos, seja de uma antena para outra, ou lida a partir de uma mídia física, como CDs ou discos rígidos.

A partir do primeiro código de correção de erros inventado (o código de Hamming [12], pertencente à classe de códigos lineares), a pesquisa neste campo aumentou significativamente, buscando melhores taxas de transmissão através de técnicas que diminuem o atraso de operação ou que consomem menos energia, objetivo bastante visado em aplicações críticas. Apesar de ser bastante simples de ser configurado, o código

de Hamming é ineficiente para pacotes de dados muito grandes, além de oferecer baixa capacidade de correção. Por outro lado, os códigos cíclicos (subclasse dos códigos lineares) trabalham com o conceito de polinômios, tornando-os mais flexíveis quanto à configuração, além de garantir uma maior capacidade de correção de erros. Os exemplos mais conhecidos de códigos cíclicos polinomiais são o Bose-Chaudhuri-Hocquenghem (BCH) e o Reed-Solomon (RS) [23]. Devido às características similares de codificação e decodificação destes códigos, o código RS é comumente classificado como um caso específico do código BCH, uma vez que este é definido sobre um alfabeto binário e aquele é um código definido em um não binário.

Sendo um código fácil de ser configurado, possuidor de estruturas eficientes de decodificação e capaz de explorar a capacidade de correção ao máximo, o código RS é bastante empregado nos mais diversos fins (principalmente, em comunicação e armazenamento de dados). Por isso, mesmo tanto tempo após sua invenção, sua estrutura ainda é objeto de estudos, tanto acadêmicos quanto comerciais e militares [3, 4, 5, 6, 24, 29, 42].

Porém, mesmo sendo usados em inúmeras aplicações atuais, alguns aspectos dos códigos RS ainda representam desafios em sua arquitetura. Além disso, estudar uma nova técnica e vê-la funcionando em pouco tempo é possível somente com o emprego de *software* ou de *hardware* reprogramável. Além disso, dependendo da aplicação alvo, uma implementação em *software* pode não atender aos requisitos de tempo de operação, uma vez que é necessário utilizar um processador genérico para executar as instruções do programa. Por outro lado, um projeto de um *Application Specific Integrated Circuit* (ASIC) demanda tempo e investimento, requer uma equipe de desenvolvimento e verificação bastante experiente, e não pode ser facilmente testado. Assim, entram em cena os *Field Programmable Gate Arrays* (FPGAs), que podem ser usados como arquitetura alvo de um projeto como plataforma de prototipação, além de serem facilmente reprogramados caso uma alteração tenha que ser feita na arquitetura e possibilitarem altas taxas de operação.

1.1 Objetivos

O presente trabalho visa a implementação de arquiteturas para o codificador e o decodificador de códigos Reed-Solomon utilizando tecnologias reconfiguráveis.

O estudo relacionado a este projeto envolve a análise dos algoritmos do estado da arte para a implementação do módulo de determinação dos polinômios localizador e avaliador de erros, o qual é considerado a parte mais crítica do decodificador dos códigos RS, assim como das demais estruturas (cálculo da síndrome e localização e correção dos erros) para estabelecer o conhecimento acerca do bloco decodificador, e a teoria que embasa o bloco codificador será explorada para estabelecer o cenário completo, onde existe a complementação do processo de codificação com o processo de decodificação. A partir do estudo teórico, arquiteturas modulares serão especificadas, assim como o interfaceamento necessário para tornar este projeto compatível com as arquiteturas comerciais estudadas e possibilitar a geração automática da arquitetura. A partir da especificação, a implementação das técnicas envolvidas no processo de codificação e decodificação será realizada, para posteriormente e ferramenta de geração da arquitetura possa também ser desenvolvida. Com a arquitetura desenvolvida, o processo de validação será realizado em duas etapas, a primeira utilizando uma ferramenta de simulação, a qual exercitará esta arquitetura sem e com erros nos dados, e a segunda utilizando uma plataforma de desenvolvimento para a prototipação. Como passo final desta pesquisa, acontecerá a avaliação dos resultados obtidos para área de ocupação e frequência de operação frente às arquiteturas comerciais estudadas.

Portanto, este trabalho tem como objetivo principal o desenvolvimento de uma arquitetura que descreva os códigos Reed-Solomon. Os objetivos específicos deste trabalho são:

- Estudar as estruturas que compõem os códigos RS e as técnicas de implementação atuais dos algoritmos de determinação dos polinômios localizador e avaliador de erros mais comuns;

- Especificar e implementar as técnicas estudadas para algoritmos em linguagem de descrição de *hardware*, e uma ferramenta para geração automática da arquitetura;

- Validar a arquitetura desenvolvida através de simulações (verificação funcional) e prototipação em *hardware* (sintetização);
- Avaliar os resultados obtidos com relação às arquiteturas comerciais estudadas, frente aos dados obtidos de ocupação e frequência de operação.

1.2 Estrutura do Texto

Esta dissertação está organizada da seguinte forma: o Capítulo 2 faz uma breve apresentação dos conceitos básicos dos códigos corretores de erros mais tradicionais, assim como apresenta a teoria dos códigos Reed-Solomon. No Capítulo 3, é realizado um estudo de algumas ferramentas presentes no mercado, tanto integradas a ambientes de desenvolvimento de *hardware* quanto aplicações específicas, que realizam a autoconfiguração e geração de *Intellectual Properties* (IPs) de blocos codificadores e decodificadores de códigos Reed-Solomon. Já no Capítulo 4, são demonstradas a especificação dos algoritmos assim como a implementação das estruturas e as técnicas escolhidas para este estudo acadêmico. O Capítulo 5 destaca a definição do código implementado, o processo de validação das arquiteturas desenvolvidas e a avaliação dos resultados obtidos a partir dos processos de síntese lógica e física da descrição de *hardware* implementada. Finalmente, o Capítulo 6 apresenta as considerações finais e os trabalhos futuros.

CAPÍTULO 2

Código Reed-Solomon

Neste capítulo serão apresentados a estrutura e o funcionamento do código Reed-Solomon. Para o melhor entendimento deste, primeiramente será feita uma introdução sobre as classes de códigos a partir dos quais o Reed-Solomon foi derivado [25]. Adicionalmente, o Apêndice A apresenta a teoria sobre álgebra abstrata, necessária para o embasamento da teoria dos códigos apresentados neste trabalho.

2.1 Conceitos básicos

A teoria dos códigos lineares e suas subclasses está embasada nos corpos finitos, os quais são conjuntos finitos em que as operações de adição e multiplicação estão definidas e cada elemento possui um elemento inverso correspondente. Simbolizados por $\mathbb{F}_q$, onde q indica a ordem (número de elementos) do corpo em questão, corpos finitos existem quando q é uma potência de primo. Mais detalhes sobre corpos finitos são apresentados no Apêndice A.3.

Quando um corpo finito for definido a partir de uma potência m de um número primo p, este é dito ser um corpo de extensão, sendo sua ordem dada por $q = p^m$ e os símbolos pertencentes a este corpo

compostos por m elementos. Como os corpos de extensão $\mathbb{F}_{p^m}$ são definidos sobre um corpo base $\mathbb{F}_p$, os elementos que compõem os símbolos provêm deste corpo base.

Como o enfoque deste trabalho são sistemas digitais, o corpo base utilizado na definição dos códigos é o corpo binário $\mathbb{F}_2$ e os símbolos dos corpos de extensão são compostos por m *bits*.

2.1.1 Códigos lineares

Em comunicações digitais, as informações são transmitidas usando sequências de zeros e uns, ou seja, sequências binárias. Na construção de códigos em bloco, estas sequências são divididas em mensagens de tamanho k, o que permite a criação de q^k mensagens distintas. Com o intuito de proteger estas mensagens durante sua transmissão, símbolos de redundância são introduzidos para ser possível corrigir possíveis erros adicionados pelo canal de transmissão. Usando-se um codificador, cada uma das q^k mensagens são transformadas em palavras-código de tamanho n, sendo que $n > k$. Assim, define-se um código em bloco com q^k palavras-código, sendo que cada uma das q^k mensagens tem apenas uma correspondência dentre as q^k palavras-código.

Porém, para valores muito grandes de n e k a implementação do código torna-se custosa, beirando a impossibilidade. Tendo isto em mente, busca-se criar códigos que possuam a característica da linearidade, a qual facilita a construção da estrutura envolvida a partir da vantagem do uso de uma função de codificação, tornando desnecessário o armazenamento de todas as q^k mensagens e as palavras-código correspondentes.

Um código é dito linear se todas as suas q^k palavras-código formarem um subespaço vetorial de dimensão k a partir do espaço vetorial de todas as n-uplas do corpo $\mathbb{F}_q^n$.

A eficiência de transmissão de um código pode ser mensurada a partir da sua taxa R, definida como:

$$R = \frac{k}{n} \tag{2.1}$$

ou seja, a taxa do código é a indicação do aproveitamento de cada palavra-código. Quanto mais próxima de "1", maior é a quantidade de informação enviada e menor é a perda por causa dos símbolos redun-

dantes. Porém, para canais onde existe muito ruído, uma taxa elevada pode não ser uma boa saída, já que os símbolos de redundância existentes na palavra-código podem não ser suficientes para recuperá-la quando muitos erros forem inseridos.

Matriz geradora

Pelo fato de um código linear ser um subespaço vetorial de dimensão k criado a partir do espaço vetorial de todas as n-uplas do corpo $\mathbb{F}_q^n$, é possível encontrar k palavras-código linearmente independentes a partir das quais é possível gerar todas as palavras-código deste código linear.

Estas k palavras-código podem ser ordenadas em uma matriz $k \times n$, a qual é chamada matriz geradora e é utilizada na codificação das mensagens.

Sendo u uma mensagem de tamanho k, a palavra-código v correspondente é obtida através da seguinte equação:

$$v = u \cdot G \tag{2.2}$$

Códigos lineares são especificados completamente a partir da sua matriz geradora, o que representa uma grande vantagem para a sua implementação, uma vez que é necessário armazenar somente as k palavras-código que compõem a matriz geradora.

Outra característica que um código deve oferecer é a estrutura sistemática, onde as palavras-código são divididas em duas partes:

- a mensagem, de tamanho k;
- a parte redundante, de tamanho $(n - k)$.

Um código linear sistemático é completamente descrito por sua matriz geradora, a qual possui a seguinte estrutura:

$$G = [P\ I_k] \tag{2.3}$$

onde P é uma matriz de paridade de dimensão $k \times (n - k)$ e I_k é uma matriz identidade de dimensão $k \times k$.

Devido à estrutura sistemática, a mensagem pode ser facilmente recuperada a partir da palavra-código.

Matriz de verificação de paridade

Além da matriz geradora, existe outra matriz também importante para os códigos lineares.

A partir de uma matriz $(k \times n)$ com k linhas linearmente independentes, é possível criar uma segunda matriz $(n-k) \times n$ que consiste de $(n-k)$ vetores linearmente independentes tal que qualquer vetor no espaço vetorial da primeira matriz é ortogonal aos vetores do espaço vetorial da segunda matriz.

Levando em consideração a ideia da ortogonalidade, é possível gerar uma segunda definição para os códigos lineares: uma n-upla é uma palavra-código v de um código linear C se, e somente se

$$v \cdot H^T = 0 \tag{2.4}$$

onde H é a matriz de verificação de paridade do código, e H^T é a sua transposta.

Se a matriz geradora estiver na forma sistemática, a matriz de verificação de paridade pode ser definida como

$$H = [I_{n-k} \ P^T] \tag{2.5}$$

onde I_{n-k} é uma matriz identidade $(n-k) \times (n-k)$ e P^T é a matriz P transposta.

Síndrome e detecção de erros

Um vetor recebido r pode ser representado da seguinte forma:

$$r = v + e \tag{2.6}$$

onde v é uma palavra-código enviada através de um canal e e é o vetor de erros inseridos pelo canal.

O vetor e é representado como

$$e = (e_0, e_1, \cdots, e_{n-1}) \tag{2.7}$$

e cada uma das suas posições indica se o vetor r possui erros.

Porém, o receptor não tem conhecimento sobre e nem v. Por esse motivo, para saber se r é uma palavra-código, o receptor realiza a

seguinte operação:

$$s = r \cdot H^T \tag{2.8}$$

onde s é a síndrome do código e tem tamanho $(n - k)$.

Como visto anteriormente, a matriz de verificação de paridade é ortogonal à matriz geradora. Por isso, para qualquer palavra-código v, $v \cdot H^T = 0$. Assim, se $r \cdot H^T = 0$, então r é uma palavra-código, caso contrário r possui erros, devendo ser corrigida antes de ser utilizada.

Como r é a soma da palavra-código v enviada com o vetor de erros e, a síndrome depende somente do vetor de erros:

$$\begin{aligned} s &= r \cdot H^T \\ r &= v + e \\ s &= (v + e) \cdot H^T = v \cdot H^T + e \cdot H^T = 0 + e \cdot H^T \end{aligned} \tag{2.9}$$

O sistema de equações dado pela Equação 2.9 é utilizado para encontrar o vetor de erros; porém, este sistema possui 2^k soluções.

Decodificação

Quando acontece a recepção de dados, antes de ser possível utilizá-los, é necessário verificar se erros foram inseridos pelo canal de comunicação, e, em caso afirmativo, os mesmos devem ser corrigidos para que então os dados possam ser utilizados. A tarefa de detecção e correção de erros é realizada através do emprego de algoritmos de decodificação.

Códigos lineares binários são compostos por 2^k palavras-código; porém, após uma delas ser transmitida, devido às interferências a que o canal de transmissão está sujeito, qualquer uma das 2^n n-uplas presentes no corpo utilizado pode ser recebida. Por isso, faz-se necessária a existência de um método de particionamento destas 2^k palavras-código para que, recebendo uma sequência qualquer, a mesma seja decodificada para somente uma palavra-código.

Visando resolver este problema, algoritmos de decodificação buscam escolher a opção que maximiza a probabilidade de correção de um símbolo. Este método é conhecido como decodificação de mínima distância [25], através do qual a palavra recebida é corrigida para a palavra-código mais próxima, sendo o conceito de distância de Ham-

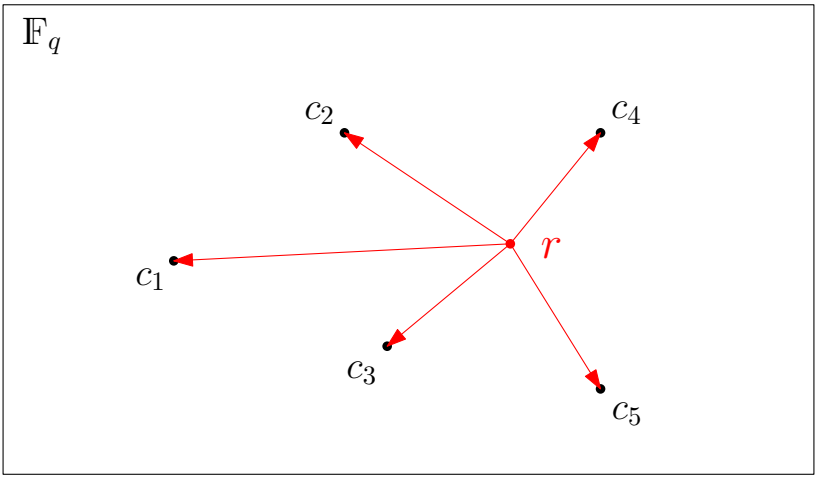

Figura 2.1: Ilustração do funcionamento do decodificador de mínima distância

ming aplicado. Este indica quantas posições diferem entre dois vetores. No entanto, a complexidade deste método o torna impraticável, uma vez que todas as palavras-código pertencentes ao alfabeto devem ser testadas para ser possível estabelecer aquela que possui a menor distância. No exemplo da Figura 2.1, para acontecer a correção da palavra recebida r, deve-se compará-la com as todas as palavras-código do alfabeto (c_1, c_2, c_3, c_4 e c_5) para estabelecer qual é a mais próxima.

Para tornar o problema da decodificação mais tratável, um dos algoritmos mais utilizados para o decodificador é o de distância limitada de raio t, onde uma palavra recebida é corrigida com sucesso quando existir somente uma palavra-código a uma distância menor ou igual à t, caso contrário, uma falha de decodificação é sinalizada. A Figura 2.2 exemplifica o funcionamento deste método; c_1, c_2, c_3, c_4 e c_5 são possíveis palavras-código pertencentes ao espaço $\mathbb{F}_q$ e r_1 e r_2 são palavras recebidas com erros. No caso da palavra recebida r_1, como c_4 se encontra dentro do seu raio de correção, esta será a palavra estimada. Já para r_2, como esta não possui nenhuma palavra-código dentro do raio de correção, ocorrerá uma falha de decodificação.

Aplicando o conceito da distância de Hamming às palavras-código de um alfabeto $\mathbb{F}_q$, pode-se encontrar as palavras-código que possuem

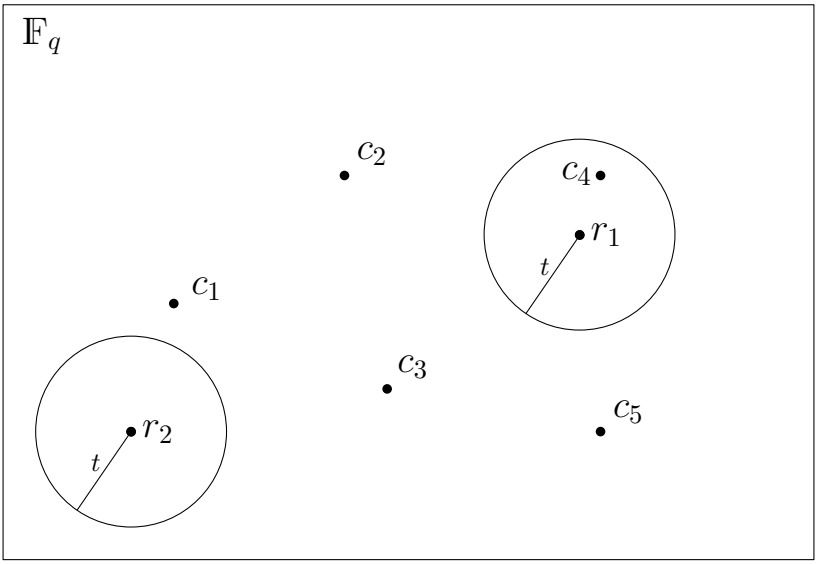

Figura 2.2: Ilustração do funcionamento do decodificador de distância limitada

a menor distância entre si. Esta distância é denotada por d_{min}. A partir desta característica do código são definidos os conceitos de raio de detecção (s) e raio de correção (t) de um código.

Para a situação em que o código é capaz apenas de detectar erros, a medida do raio de detecção s é dada por:

$$s = d_{min} - 1 \tag{2.10}$$

Para que um código seja capaz de corrigir erros, o raio de verificação deve ser reduzido tal que não hajam sobreposições das esferas ao redor de cada palavra-código. Assim, um vetor recebido pode ser corrigido para somente uma palavra-código. O raio de correção t é definido por:

$$t = \frac{d_{min} - 1}{2} \tag{2.11}$$

Porém, para o caso em que d_{min} é par, a operação dada por (2.11) resultará em um número não inteiro, e para garantir que os raios de correção não ficarão sobrepostos e nem se tocarão, faz-se o arredonda-

mento para baixo do resultado. Assim, reescrevendo a Equação (2.11) tem-se:

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \tag{2.12}$$

Finalmente, para que um código seja capaz de detectar e corrigir erros, nem o raio de correção, nem o raio de detecção podem conter sobreposições. Porém, neste último caso, as circunferências podem se tocar. Sendo assim, para um código que detecta e corrige erros, o raio de detecção s é definido como:

$$s = d_{min} - t - 1 \tag{2.13}$$

ou seja, é uma unidade menor que a diferença entre a distância mínima de Hamming e o raio de correção do código.

Quando uma palavra recebida não é envolta pelos raios de detecção e de correção, esta não pode ser corrigida e uma falha deve ser sinalizada.

2.1.2 Códigos cíclicos

Os códigos cíclicos são uma subclasse dos códigos em bloco e possuem duas grandes vantagens:

(*i*) codificação e cálculo da síndrome podem ser implementados usando registradores de deslocamento;

(*ii*) devido à sua estrutura algébrica, é possível criar diversas formas práticas para efetuar a decodificação.

A sua definição formal é dada a seguir.

Definição. Um código linear C é dito ser um código cíclico se, para cada deslocamento cíclico em uma palavra-código de C, o vetor resultante também é uma palavra-código em C [23].

Sendo um vetor $v = (v_0,\ v_1,\ \cdots, v_{n-1})$, fazendo i deslocamentos em v obtém-se

$$v^{(i)} = (v_{n-i},\ v_{n-i+1},\ \cdots,\ v_{n-1},\ v_0,\ v_1,\ \cdots,\ v_{n-i-1}) \tag{2.14}$$

Outra forma de representação bastante utilizada para facilitar as operações é a polinomial, onde existe uma correspondência 1-para-1 com a representação vetorial.

$$v = (v_0,\ v_1,\ \cdots, v_{n-1}) \Rightarrow v(x) = v_0 + v_1 x + \cdots + v_{n-1}x^{n-1} \quad (2.15)$$

onde, neste caso, cada palavra-código é um polinômio de grau menor ou igual à $(n-1)$.

Para a geração de um código cíclico, usa-se um polinômio de grau igual à $(n-k)$, denominado polinômio gerador. Este polinômio é o único com grau $(n-k)$ de todo o código, e não existem polinômios não-nulos com grau inferior.

As palavras-código que compõem um código cíclico são geradas a partir do polinômio gerador através de deslocamentos e combinações lineares. Além disso, um polinômio de grau $(n-1)$ ou menor só é uma palavra-código se for um múltiplo do polinômio gerador.

Sendo $u(x)$ o polinômio que representa a mensagem a ser enviada, de tamanho k, então a palavra-código $v(x)$ pode ser obtida através da operação

$$v(x) = u(x) \cdot g(x) \quad (2.16)$$

Uma questão importante a ser respondida é se um polinômio pode gerar um código cíclico. Para tal, o polinômio $g(x)$ deve atender duas condições:

- o grau do polinômio deve ser igual a $(n-k)$;
- o polinômio deve ser um fator de $x^n - 1$.

Pertencente à classe de códigos lineares, o RS é um código cíclico (conforme o Apêndice A.3, os símbolos que compõem suas mensagens e palavras-código provêm de um corpo de extensão definido para m *bits*, onde $m \geq 2$; por consequência, possui $2^m - 1$ elementos não nulos).

Codificação

A matriz geradora de um código cíclico é gerada a partir de k n-uplas linearmente independentes, ou seja, por todos os k fatores $x^i g(x)$, onde

$0 \leq i < k$. A matriz geradora de um código cíclico (n, k) possui dimensão $k \times n$ e pode ser escrita na forma sistemática através de operações entre as linhas da mesma. No entanto, é mais eficiente utilizar a codificação via polinômios, pois envolve menos operações e é mais simples de ser realizada.

A codificação de códigos cíclicos pode ser feita a partir de um circuito de divisão com $(n - k)$ estágios baseado no polinômio gerador. Para se obter as palavras-código de forma sistemática, onde a mensagem fica separada da paridade, deve-se seguir três passos:

(i) multiplicar a mensagem
$u(x) = u(x)x^{n-k}$

(ii) obter o resto da divisão entre a nova mensagem e o polinômio gerador
$b(x) = u(x)x^{n-k} \bmod g(x)$

(iii) combinar o resto com a nova mensagem para obter a palavra-código
$c(x) = -b(x) + u(x)x^{n-k}$

sendo que $b(x)$ representa os símbolos de paridade e $x^{n-k}u(x)$ representa a mensagem a ser enviada. Pode-se provar que a palavra resultante pertence ao código, ou seja, que é múltipla de $g(x)$ [25].

2.1.3 Modificações em códigos

Além de ser possível a utilização dos códigos conforme a definição apresentada anteriormente, é possível modificá-los de acordo com as especificações de projeto em desenvolvimento. As modificações mais comuns são definidas a seguir.

Encurtamento

Quando um símbolo de mensagem é eliminado, diz-se que o código foi encurtado, o que torna um código (n, k) em $(n-f, k-f)$ após f encurtamentos. Em códigos lineares tradicionais, esta operação consiste na remoção tanto da linha referente ao símbolo removido quanto da coluna referente ao símbolo codificado. Para códigos cíclicos, o encurtamento acontece quando o subconjunto referente às mensagens contendo zeros

nas f posições superiores é removido do conjunto original, e somente as mensagens restantes são consideradas válidas. Por exemplo, para uma palavra-código com n símbolos, sendo k de mensagem:

$$\underbrace{c_1, c_2, \cdots, c_k}_{\text{mensagem}}, \underbrace{c_{k+1}, c_{k+2}, \cdots, c_n}_{\text{paridade}} \tag{2.17}$$

após ser realizado o encurtamento do símbolo c_k, seu comprimento torna-se $n-1$ ($k-1$ símbolos de mensagem) e sua estrutura se torna a seguinte:

$$\underbrace{c_1, c_2, \cdots, c_{k-1}}_{\text{mensagem}}, \underbrace{c_{k+1}, c_{k+2}, \cdots, c_n}_{\text{paridade}} \tag{2.18}$$

Puncionamento

A operação de puncionamento remove símbolos de paridade da palavra-código após a codificação, aumentando a taxa do código. Após a realização desta operação p vezes, um código (n, k) torna-se $(n-p, k)$. Por exemplo, para uma palavra-código de tamanho n contendo os símbolos

$$\underbrace{c_1, c_2, \cdots, c_k}_{\text{mensagem}}, \underbrace{c_{k+1}, c_{k+2}, \cdots, c_n}_{\text{paridade}} \tag{2.19}$$

após ser realizado o puncionamento do símbolo de paridade c_n, seu comprimento torna-se $n-1$ ($2t-1$ símbolos de paridade) e sua estrutura se torna a seguinte:

$$\underbrace{c_1, c_2, \cdots, c_k}_{\text{mensagem}}, \underbrace{c_{k+1}, c_{k+2}, \cdots, c_{n-1}}_{\text{paridade}} \tag{2.20}$$

2.2 Códigos Reed-Solomon

Os códigos Reed-Solomon foram desenvolvidos na década de 1960, mas foi a partir da publicação de [27] que começaram a ganhar espaço em projetos que requerem uma boa capacidade de correção. Atualmente, esta classe de códigos é utilizada principalmente em sistemas de comunicação e armazenamento magnético, devido à sua grande capacidade de correção de erros em sequência (ou erros em rajada). Porém, códigos

RS são aplicados em diversos outros fins.

Uma grande vantagem desta classe de códigos é a sua flexibilidade de especificação de acordo com a aplicação alvo, uma vez que é possível configurar praticamente todos os parâmetros disponíveis, sendo a exceção aumentar seu comprimento.

O código RS é um código cíclico com polinômio gerador definido de acordo com a Equação (2.21).

$$g(x) = (x - \alpha^b)(x - \alpha^{b+1}) \cdots (x - \alpha^{b+2t-1}) \tag{2.21}$$

onde b é um inteiro não negativo, α é um elemento primitivo do corpo $\mathbb{F}_q$, onde $q = 2^m$, e t é a capacidade de correção projetada para o código. Geralmente, o valor escolhido para b é 1, e assim o código gerado é chamado de *narrow sense*. Note que $\alpha^b, \alpha^{b+1}, \cdots, \alpha^{b+2t-1}$ são as $2t$ raízes do polinômio gerador, o qual tem grau igual à $2t$.

Como citado na Seção 2.1, o corpo $\mathbb{F}_q$ utilizado é o binário ($q = 2^m$), e cada elemento do corpo é composto por m *bits*. Diversos padrões utilizados atualmente [2, 8, 9, 13, 14, 15] trabalham com símbolos de um *byte*; portanto, para estes códigos, o valor escolhido para m é 8.

O comprimento n das palavras-código está relacionado com m da seguinte forma

$$n = q - 1 = 2^m - 1 \tag{2.22}$$

A notação utilizada para simbolizar um código Reed-Solomon é

$$RS(n, k, d_{\min}) \tag{2.23}$$

onde n é o tamanho da palavra-código, k é a quantidade de símbolos de informação em cada palavra-código e $d_{\min}$ é a distância mínima do código, estando relacionada com os demais parâmetros da seguinte forma

$$2t = n - k = d_{min} - 1 \tag{2.24}$$

Observa-se que o código RS possui a maior capacidade de correção possível entre todos os códigos com os mesmos parâmetros n e k [17].

A capacidade de correção do código (t) pode ser definida de acordo com os parâmetros do código; seu valor deve ser razoavelmente grande

para oferecer uma boa aplicabilidade, porém não demasiadamente grande a ponto de reduzir drasticamente a taxa de transmissão do código.

2.2.1 Codificação

Por pertencer à classe dos códigos lineares, o método de codificação dos códigos RS segue os mesmos passos descritos na Seção 2.1.2.

2.3 Decodificação

O processo de decodificação é composto por quatro etapas, as quais são:

1. Cálculo da síndrome;
2. Determinação dos polinômios localizador e avaliador de erros;
3. Determinação das raízes do polinômio localizador de erros;
4. Determinação da magnitude dos erros.

Os passos citados são detalhados nas próximas seções.

2.3.1 Cálculo da síndrome

Como primeira etapa do processo, deve-se verificar se o polinômio recebido, $r(x)$, possui erros. O polinômio $r(x)$ é definido como

$$r(x) = c(x) + e(x) \tag{2.25}$$

onde $c(x)$ é a palavra-código transmitida pelo canal e $e(x)$ é o polinômio simbolizando os erros introduzidos pelo canal.

Como $\alpha^b, \alpha^{b+1}, \cdots, \alpha^{b+2t-1}$ são as raízes de $c(x)$, então $c(\alpha^{b+i}) = 0$ para $0 \leq i \leq 2t-1$, então a síndrome depende unicamente de $e(x)$. Assim, tem-se

$$\begin{aligned} s_{b+i} &= r(\alpha^{b+i}) \\ &= r_0 + r_1 \cdot (\alpha^{b+i}) + r_2 \cdot (\alpha^{b+i})^2 + \cdots + r_{n-1} \cdot (\alpha^{b+i})^{n-1} \\ &= c(\alpha^{b+i}) + e(\alpha^{b+i}) \\ &= e(\alpha^{b+i}) \end{aligned} \tag{2.26}$$

Supondo que $r(x)$ possua uma certa quantidade de erros, é possível construir um sistema de equações a partir da definição da síndrome que tenha como objetivo a correção destes erros. Por exemplo, se $r(x)$ possuir v erros, monta-se o seguinte sistema:

$$\begin{aligned} s_1 &= e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + \cdots + e_{j_v}\alpha^{j_v} \\ s_2 &= e_{j_1}(\alpha^{j_1})^2 + e_{j_2}(\alpha^{j_2})^2 + \cdots + e_{j_v}(\alpha^{j_v})^2 \\ &\vdots \\ s_{2t} &= e_{j_1}(\alpha^{j_1})^{2t} + e_{j_2}(\alpha^{j_2})^{2t} + \cdots + e_{j_v}(\alpha^{j_v})^{2t} \end{aligned} \tag{2.27}$$

onde α^{j_l}, com $1 \leq l \leq v$, são chamados de coeficientes de localização dos erros (*error-location numbers*) e são desconhecidos. Assim que os valores de $\alpha^{j_1}, \alpha^{j_2}, \cdots, \alpha^{j_v}$ forem encontrados, as potências $j_1, j_2, \cdots, j_v$ indicarão as posições dos erros em $e(x)$. Neste caso, se o número de erros presentes em $e(x)$ for menor que t, então a solução que tiver o menor número de erros no padrão será aceita como a correta.

Ao final desta etapa, verifica-se o resultado obtido pela operação realizada em (2.27); se $s(x)$ for igual a zero, o polinômio $r(x)$ é uma palavra-código e os próximos passos não são necessários. Caso contrário, sabe-se que existem erros em $r(x)$ e os mesmos devem se corrigidos.

2.3.2 Determinação dos polinômios localizador e avaliador de erros

Após a determinação da síndrome do polinômio recebido e no caso de ser detectada a presença de erros, o próximo passo da decodificação é a determinação do polinômio localizador de erros.

Recapitulando a Equação (2.27), a mesma pode ser reescrita como

$$s_i = e_{j_1}(\alpha^{j_1})^i + e_{j_2}(\alpha^{j_2})^i + \cdots + e_{j_v}(\alpha^{j_v})^i \qquad (i = b, b+1, \cdots, b+2t-1) \tag{2.28}$$

e realizando uma troca de variáveis

$$\beta_l = \alpha^{j_l} \tag{2.29}$$

tem-se

$$s_i = e_{j_1}(\beta_1)^i + e_{j_2}(\beta_2)^i + \cdots + e_{j_v}(\beta_v)^i \qquad (i = b+1, b+2, \cdots, b+2t) \tag{2.30}$$

de onde se pode construir o seguinte sistema

$$\begin{aligned} s_1 &= e_{j_1}(\beta_1) + e_{j_2}(\beta_2) + \cdots + e_{j_v}(\beta_v) \\ s_2 &= e_{j_1}(\beta_1)^2 + e_{j_2}(\beta_2)^2 + \cdots + e_{j_v}(\beta_v)^2 \\ s_3 &= e_{j_1}(\beta_1)^3 + e_{j_2}(\beta_2)^3 + \cdots + e_{j_v}(\beta_v)^3 \\ &\vdots \\ s_{2t} &= e_{j_1}(\beta_1)^{2t} + e_{j_2}(\beta_2)^{2t} + \cdots + e_{j_v}(\beta_v)^{2t} \end{aligned} \tag{2.31}$$

A decodificação de $r(x)$ consiste da resolução deste sistema, sendo que qualquer método capaz de realizar esta tarefa pode ser empregado. Um dos primeiros métodos a surgirem para resolver este sistema de equações foi o de Peterson-Gorenstein-Zeirler [17]; porém, o mesmo só é interessante para decodificadores que consigam corrigir pequenas quantidades de erros, tendo um aumento de complexidade considerável para cada erro a mais que é possível corrigir.

Assim, visando técnicas que possuam uma complexidade mais uniforme de acordo com o aumento da capacidade de correção, busca-se estudar alternativas para esta etapa. Um método normalmente empregado para esta resolução é o polinômio localizador de erros, o qual torna o problema mais tratável.

O polinômio localizador de erros é definido como

$$\begin{aligned} \sigma(x) &= (1-\beta_1 x)(1-\beta_2 x)(1-\beta_3 x)\cdots(1-\beta_v x) \\ \sigma(x) &= \sigma_0 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_v x^v \end{aligned} \tag{2.32}$$

sendo que, por definição $\sigma_0 = 1$. As inversas das raízes deste polinômio indicam quais são as posições de $r(x)$ que devem ser corrigidas, daí o seu nome. Este polinômio é relacionado com a síndrome da seguinte

forma:

$$s_j = -\sum_{i=1}^{v} \sigma_i s_{j-i}, \quad j = v+1, v+2, \cdots, 2t \tag{2.33}$$

Ao mesmo tempo em que a determinação do polinômio localizador de erros ($\sigma(x)$) acontece, o polinômio avaliador de erros ($\Omega(x)$) (utilizado para calcular o valor a ser somado a cada uma das posições em erro) é também determinado. Porém, a versão do algoritmo Berlekamp-Massey apresentada em [6] possui como saída apenas o polinômio localizador de erros, e, para se obter o polinômio avaliador de erros, é utilizada a relação dada pela Equação (2.34).

$$\Omega(x) = s(x)\sigma(x) \bmod x^{2t} \tag{2.34}$$

Existem dois métodos bastante conhecidos para determinar os polinômios localizador e avaliador de erros, os quais são

- Berlekamp-Massey;
- Euclideano.

Ambos possuem complexidade similar, porém o método Euclideano apresenta um algoritmo mais estruturado, enquanto o método Berlekamp-Massey apresenta uma eficiência um pouco maior nas operações realizadas [17]. Ambos algoritmos serão estudados com maior profundidade na Seção 2.4.

2.3.3 Determinação das raízes do polinômio localizador de erros

Após a determinação do polinômio localizador de erros, $\sigma(x)$, é necessário encontrar suas raízes, pois, como já mencionado, suas inversas indicam as posições do polinômio recebido que devem ser corrigidas.

Para este passo do processo, é possível empregar qualquer método capaz de retornar as raízes do polinômio em questão. Porém, como as operações são realizadas em um corpo finito, uma das técnicas mais utilizadas é a busca exaustiva por todos os elementos não nulos do corpo. Tal técnica é chamada de Busca de Chien.

Assim, tem-se um sistema conforme (2.35).

$$\begin{aligned}
\sigma(1) &= \sigma_0 + \sigma_1(1) + \sigma_2(1)^2 + \cdots + \sigma_v(1)^v \\
\sigma(\alpha) &= \sigma_0 + \sigma_1(\alpha) + \sigma_2(\alpha)^2 + \cdots + \sigma_v(\alpha)^v \\
\sigma(\alpha^2) &= \sigma_0 + \sigma_1(\alpha^2) + \sigma_2(\alpha^2)^2 + \cdots + \sigma_v(\alpha^2)^v \\
&\vdots \\
\sigma(\alpha^{q^m-2}) &= \sigma_0 + \sigma_1(\alpha^{q^m-2}) + \sigma_2(\alpha^{q^m-2})^2 + \cdots + \sigma_v(\alpha^{q^m-2})^v
\end{aligned} \tag{2.35}$$

Ao término da computação de todos os termos pertencentes ao corpo finito, tem-se todas as v raízes do polinômio localizador de erros, caso seu grau seja igual ou menor a t. Caso contrário, as raízes encontradas não refletem as posições corretas a serem corrigidas, e assim, ocorre um erro de decodificação.

2.3.4 Cálculo da magnitude dos erros

Como último passo do processo de decodificação, é necessário calcular o valor de correção para cada uma das posições que devem ser corrigidas. Recapitulando o polinômio da síndrome (2.30), agora com as raízes β_l determinadas, é possível montar um sistema de equações lineares para a determinação da magnitude de cada um dos erros.

$$\begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & \cdots & \beta_v \\ \beta_1^2 & \beta_2^2 & \beta_3^2 & \cdots & \beta_v^2 \\ \vdots & & & & \\ \beta_1^{2t} & \beta_2^{2t} & \beta_3^{2t} & \cdots & \beta_v^{2t} \end{bmatrix} \begin{bmatrix} e_{i_1} \\ e_{i_2} \\ \vdots \\ e_{i_v} \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_{2t} \end{bmatrix} \tag{2.36}$$

Como pode ser notado, o primeiro termo de (2.36) é uma matriz de Vandermonde, ou seja, as linhas formam uma progressão geométrica. Para este tipo de sistema existem técnicas que aceleram o processamento, não sendo necessário solucioná-lo da forma tradicional. Especificamente para a decodificação de códigos Reed-Solomon, utiliza-se o Algoritmo de Forney para a resolução deste sistema, o qual utiliza os polinômios localizador e avaliador de erros aplicados à cada uma das inversas das raízes do polinômio localizador de erros.

O Algoritmo de Forney é definido como

$$e_{j_l} = -\frac{\Omega(\alpha^{j_l})}{\sigma'(\alpha^{j_l})} \tag{2.37}$$

sendo que $\Omega(x)$ é definido pela Equação (2.34) e $\sigma'(x)$ é a derivada de primeiro grau de $\sigma(x)$. Cada um dos resultados para e_{j_l} é somado ao termo correspondente no polinômio $r(x)$ para então corrigi-lo. Assim que todos os v erros forem removidos, a palavra-código estimada é disponibilizada para o usuário e o processo termina.

2.3.5 Correção de erros e apagamentos

Além de ser possível corrigir erros, um decodificador pode agregar a capacidade de corrigir apagamentos, ou seja, posições assinaladas como tendo que ser corrigidas. Quando o decodificador possui esta capacidade, o seu poder de correção é dado por:

$$2E + A \leq 2t \tag{2.38}$$

onde E é a quantidade de erros e A é a quantidade de apagamentos presentes no dado recebido.

Para que ocorra a correção dos símbolos apagados, os mesmos devem ser substituídos por algum outro elemento do corpo finito, como zero, para então formar um novo polinômio recebido, $r'(x)$, para então calcular a síndrome desse novo polinômio.

É também necessária a determinação de um polinômio localizador de apagamentos. Da mesma forma que o polinômio localizador de erros, este indicará as posições em que os símbolos apagados se encontram para que posteriormente ocorra a sua correção. Porém, diferentemente do polinômio localizador de erros, o polinômio localizador de apagamentos é construído a partir das posições apagadas, não necessitando de um método de determinação mais complexo, como Berlekamp-Massey ou Euclideano.

Supondo que $r'(x)$ possua A apagamentos, o polinômio localizador de apagamentos $\Gamma(x)$ é definido conforme a Equação (2.39).

$$\Gamma(x) = (1 + y_1 x)(1 + y_2 x) \cdots (1 + y_A x) \tag{2.39}$$

Para o cálculo da magnitude dos erros e dos apagamentos, o polinômio avaliador de erros é redefinido conforme a Equação (2.40)

$$\Omega(x) = \sigma(x)\Gamma(x)s(x) \bmod x^{2t} \tag{2.40}$$

e fazendo

$$\Xi(x) = \Gamma(x)s(x) \tag{2.41}$$

para representar a informação conhecida, é possível aplicar ambos os métodos apresentados na Seção 2.4 para a determinação dos polinômios localizador e avaliador de erros. Usando o algoritmo Berlekamp-Massey, deve-se empregar o novo polinômio (2.41) ao invés do polinômio da síndrome para o cálculo da discrepância. No caso do algoritmo Euclideano, inicializa-se $R(x)$ com x^{2t} e $T(x)$ com $\Xi(x)$, tendo como novo critério de parada

$$deg(R(x)) \leq \begin{cases} t + \frac{A}{2} & \text{se } A \text{ for par} \\ t + \frac{A-1}{2} & \text{se } A \text{ for ímpar} \end{cases} \tag{2.42}$$

e, ao final da operação do algoritmo utilizado para a determinação dos polinômios, tem-se, além do polinômio avaliador de erros, o polinômio localizador de erros e apagamentos, o qual, a partir das suas raízes, informará quais são as posições que devem ser corrigidas.

Assim que o polinômio localizador de erros e apagamentos é determinado, usa-se a Busca de Chien para encontrar as $(E + A)$ raízes de $\sigma(x)$. Como último passo do processo modificado, emprega-se uma versão alterada do Algoritmo de Forney para o cálculo da magnitude dos erros, a qual depende de uma combinação dos polinômios localizadores de erros e apagamentos, dada por

$$\Phi(x) = \sigma(x)\Gamma(x) \tag{2.43}$$

Substituindo (2.43) pelo denominador de (2.37), obtém-se

$$e_{j_l} = -\frac{\Omega(\alpha^{j_l})}{\Phi'(\alpha^{j_l})} \tag{2.44}$$

para encontrar a magnitude dos erros, e

$$\Gamma_{i_l} = -\frac{\Omega(\alpha^{i_l})}{\Phi'(\alpha^{i_l})} \tag{2.45}$$

para encontrar a magnitude dos apagamentos, onde i_l são as l posições apagadas, as quais são conhecidas desde o início do processo.

2.4 Métodos de determinação dos polinômios localizador e avaliador de erros

Como o passo de determinação dos polinômios é o mais complexo do decodificador dos códigos RS, diversas técnicas existem para realizar seu processamento. Porém, conforme citado na Seção 2.3.2, os algoritmos Berlekamp-Massey [6] e Euclideano [32] são os mais conhecidos e possuem complexidade similar. Portanto, nas próximas seções, ambos os algoritmos serão estudados, assim como algumas técnicas derivadas destes, as quais exploram algumas melhorias matemáticas para estruturar de forma mais uniforme estes algoritmos.

2.4.1 Algoritmo Berlekamp-Massey

A Equação (2.33) descreve a saída de um *Linear Feedback Shift Register* (LFSR) com coeficientes σ_l. Para ser possível determinar as posições de $r(x)$ que possuem erro, pode-se empregar o algoritmo Berlekamp-Massey [6, 24] para determinar os coeficientes do polinômio.

O objetivo do algoritmo BM é produzir o menor polinômio capaz de encontrar as posições com erro de $r(x)$, a partir das síndromes calculadas no primeiro passo da decodificação. Assim, o algoritmo começa com um polinômio unitário e constrói o polinômio localizador de erros à medida que se faz necessário aumentá-lo de acordo com as síndromes utilizadas. Neste processo, um polinômio é utilizado para fazer a conexão entre o polinômio sendo construído e os coeficientes da síndrome utilizados até o momento. Este polinômio é chamado de polinômio de conexão.

O Algoritmo 1 descreve os passos do algoritmo Berlekamp-Massey. Inicialmente, na Linha 1 as variáveis principais são inicializadas antes das $2t$ iterações começarem. Para cada uma das iterações, calcula-se

a discrepância do coeficiente atual da síndrome (Linha 3) que indica se o polinômio deve ser atualizado ou não; sinalização esta que é realizada através do valor da variável δ (Linhas 4 a 8). Já na Linha 9, a nova discrepância é incorporada ao polinômio de conexão atual, caso a mesma seja diferente de zero. Finalmente, as variáveis de controle são atualizadas (Linhas 10 a 16).

É importante notar que, neste algoritmo, mais especificamente na Linha 14, é necessário realizar uma operação de inversão, a qual, em termos de álgebra em corpos finitos, não é uma operação trivial.

Algoritmo 1 Berlekamp-Massey

1: $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, L^{(0)} = 0$
2: **for** $j = 1 : 2t$ **do**
3: $\quad \Delta_j = \sum_{i=0}^{L^{(j-1)}} \sigma_i^{(j-1)} s_{j-i}$
4: $\quad$ **if** $\Delta_j \neq 0$ and $2L^{(j-1)} \leq j - 1$ **then**
5: $\quad\quad \delta = 1$
6: $\quad$ **else**
7: $\quad\quad \delta = 0$
8: $\quad$ **end if**
9: $\quad \sigma^{(j)}(x) = \sigma^{(j-1)}(x) - \Delta_j x B^{(j-1)}(x)$
10: $\quad$ **if** $\delta = 0$ **then**
11: $\quad\quad B^{(j)}(x) = xB^{(j-1)}(x)$
12: $\quad\quad L^{(j)} = L^{(j-1)}$
13: $\quad$ **else if** $\delta = 1$ **then**
14: $\quad\quad B^{(j)}(x) = \delta_j^{-1}\sigma^{(j-1)}(x)$
15: $\quad\quad L^{(j)} = j - L^{(j-1)}$
16: $\quad$ **end if**
17: **end for**

2.4.2 Algoritmo Berlekamp-Massey sem inversão (*Inversionless Berlekamp-Massey* - iBM)

Em [42], é apresentada uma variação para o algoritmo Berlekamp-Massey onde a inversão necessária no algoritmo original é substituída pela inserção de um novo coeficiente, (θ), que compensa a operação.

Conforme mostrado no Algoritmo 2, uma das diferenças entre o algoritmo original e a alternativa sem inversão está na atualização do polinômio de conexão atual (Linha 4), onde é possível perceber que

a multiplicação acontece independentemente da discrepância atual ser igual à zero ou não.

A outra diferença entre os procedimentos é o polinômio obtido como saída. Por causa deste novo coeficiente, o polinômio localizador de erros resultante é um múltiplo escalar do polinômio original, tendo, portanto, ambos as mesmas raízes, e garantindo a sua funcionalidade.

Algoritmo 2 Berlekamp-Massey sem inversão

1: $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, \Theta^{(0)} = 1, L^{(0)} = 0$
2: **for** $j = 1 : 2t$ **do**
3: $\quad \Delta_j = \sum_{i=0}^{L^{(j-1)}} \sigma_i^{(j-1)} s_{j-i}$
4: $\quad \sigma^{(j)}(x) = \theta^{(j-1)}\sigma^{(j-1)}(x) - \Delta_j x B^{(j-1)}(x)$
5: $\quad$ **if** $\Delta_j \neq 0$ and $2L^{(j-1)} \leq j - 1$ **then**
6: $\quad\quad B^{(j)}(x) = \sigma^{(j-1)}(x)$
7: $\quad\quad \theta^{(j)} = \Delta_j$
8: $\quad\quad L^{(j)} = j - L^{(j-1)}$
9: $\quad$ **else**
10: $\quad\quad B^{(j)}(x) = xB^{(j-1)}(x)$
11: $\quad\quad \theta^{(j)} = \theta^{(j-1)}$
12: $\quad\quad L^{(j)} = L^{(j-1)}$
13: $\quad$ **end if**
14: **end for**

2.4.3 Algoritmo Berlekamp-Massey sem inversão reformulado (*Reformulated Inversionless Berlekamp-Massey* - RiBM)

Buscando uma arquitetura mais modular para o algoritmo Berlekamp-Massey, chegou-se à estrutura proposta em [29]. A partir de reformulações do algoritmo Berlekamp-Massey sem inversão, obteve-se uma estrutura mais uniforme, o que facilita a implementação em *hardware*, além de ser possível obter os dois polinômios necessários em apenas $2t$ ciclos de relógio, graças ao cálculo da próxima discrepância ao mesmo tempo em que o polinômio de conexão é atualizado.

Analisando o Algoritmo 2 em termos de polinômios, nota-se que, após o cálculo da discrepância, o próximo passo é a atualização do polinômio de conexão e do polinômio auxiliar $B^{(j)}(x)$. É possível perceber que a discrepância calculada no início do laço é o termo multiplicado

por x^j no produto dado por

$$\begin{aligned}\sigma^{(j)}(x)s(x) &= \Delta^{(j)}(x) \\ &= \delta_0^{(j)} + \delta_1^{(j)}x + \cdots + \delta_j^{(j)}x^j + \cdots \end{aligned} \tag{2.46}$$

Embora a solução ótima seja ter todo o polinômio $\Delta(x)$ calculado já no início da operação do algoritmo, somente o termo $\delta_j^{(j)}$ é necessário para realizar as atualizações citadas anteriormente. Então, assumindo que o decodificador possua todos os coeficientes de $\Delta^{(j)}(x)$ e $\Pi^{(j)}(x) = B^{(j)}(x)s(x)$, e por consequência $\delta_j^{(j)}$, disponíveis no início do ciclo de relógio, é possível realizar a atualização de $\sigma^{(j)}(x)$ e de $B^{(j)}(x)$. Além disso, a partir da equação de atualização de $\sigma^{(j)}(x)$ do Algoritmo 2 e da relação dada pela Equação (2.46), tem-se

$$\begin{aligned}\Delta^{(j+1)}(x) &= \sigma^{(j+1)}(x)s(x) \\ &= \left(\theta^{(j)}(x)\sigma^{(j)}(x) - x\delta_j^{(j)}B^{(j)}(x)\right)s(x) \\ &= \theta^{(j)}(x)\Delta^{(j)}(x) - x\delta_j^{(j)}\Pi^{(j)}(x)\end{aligned} \tag{2.47}$$

e assim $\Pi^{(j+1)}(x)$ é atualizado para $\Delta^{(j)}(x) = \sigma^{(j)}(x)s(x)$ ou para $x\Pi^{(j)}(x) = xB^{(j)}(x)s(x)$. A partir daí, percebe-se que $\Delta^{(j+1)}(x)$ e $\Pi^{(j+1)}(x)$ são calculados da mesma forma que $\sigma^{(j+1)}(x)$ e $B^{(j+1)}(x)$ e todos os quatro polinômios são computados ao mesmo tempo a cada iteração junto com $\delta_{j+1}^{(j+1)}$ para então estarem disponíveis no início do próximo ciclo de relógio.

Outro ponto muito importante desta reformulação, é a determinação de ambos os polinômios ao mesmo tempo nas $2t$ iterações do algoritmo, tornando desnecessária a multiplicação entre o polinômio localizador de erros e o polinômio das síndromes ao final do Algoritmo 2. Como o polinômio avaliador de erros pode ser determinado a partir da relação dada pela Equação (2.34), é possível perceber que esta operação acontece a cada iteração com o cálculo de $\Delta^{(j)}(x) = \sigma^{(j)}(x)s(x)$ e, ao final do processamento deste algoritmo, os graus mais baixos de $\Delta(x)$ correspondem ao polinômio $\Omega(x)$. Porém, os graus mais elevados de $\Delta(x)$ também podem ser utilizados para o cálculo do valor dos erros, o que traz mais um benefício arquitetural para este algoritmo. Em

termos polinomiais, $\Delta(x)$ pode ser decomposto em

$$\Delta(x) = \Omega(x) + x^{2t}\Omega^{(h)}(x) \tag{2.48}$$

sendo que $\Omega^{(h)}(x)$ contém os termos de mais alta ordem com grau máximo $v-1$, e, como para $i < j$ os valores de $\delta^{(i)}$ na iteração j não afetam as atualizações subsequentes, não é necessário armazenar estes valores, e a partir desta constatação são definidos os elementos

$$\tilde{\delta}^{(j)} = \delta^{(i+j)} \tag{2.49}$$

$$\tilde{\pi}^{(j)} = \pi^{(i+j)} \tag{2.50}$$

$$\tilde{\Delta}^{(j)}(x) = \sum_{i=0}^{2t-1} \tilde{\delta}_i^{(j)} x^i \tag{2.51}$$

$$\tilde{\Pi}^{(j)}(x) = \sum_{i=0}^{2t-1} \tilde{\pi}_i^{(j)} x^i \tag{2.52}$$

onde $\tilde{\Delta}(x)$ e $\tilde{\Pi}(x)$ são inicializados com o polinômio da síndrome, a atualização de $\tilde{\delta}^{(j)}$ é dada por

$$\begin{aligned} \tilde{\delta}^{(j)} &= \delta_{i+1+j}^{(j)} \\ &= \theta^{(j-1)}\delta_{i+1+j}^{(j-1)} - \delta_j^{(j-1)}\pi_{i+j}^{(j-1)} \\ &= \theta^{(j-1)}\tilde{\delta}_{i+1}^{(j-1)} - \tilde{\delta}_0^{(j-1)}\tilde{\pi}_i^{(j-1)} \end{aligned} \tag{2.53}$$

e $\tilde{\pi}^{(j)}$ é atualizado para $\tilde{\delta}_{i+1}^{(j-1)}$ ou para $\tilde{\delta}_i^{(j-1)}$. Pode-se notar ainda que a discrepância atual $\delta_j^{(j)}$ fica posicionada sempre no termo independente do polinômio ($\tilde{\delta}_0^{(j)}$).

Para $\tilde{\Delta}(x)$, tem-se como resultado final

$$\begin{aligned} \tilde{\Delta}(x) &= \delta_{2t}^{(2t)} + \delta_{2t+1}^{(2t)} x + \cdots \\ &= \Omega^{(h)}(x) \end{aligned} \tag{2.54}$$

ou seja, o polinômio avaliador de erros.

Conforme apresentado na Equação (2.48), o polinômio avaliador de erros que o algoritmo RiBM tem como resultado é multiplicado por um fator de x^{2t}. Para adequar o algoritmo de Forney à esta mudança, o

mesmo deve ser escrito como

$$e_{i_l} = -\frac{\alpha^{b+2t}\Omega^{(h)}(\alpha^{j_l})}{\sigma'(\alpha^{j_l})} \tag{2.55}$$

de tal forma que é possível utilizar ambos os polinômios de saída do algoritmo RiBM na etapa de correção dos erros da palavra recebida.

O Algoritmo 3 exemplifica as operações realizadas nesta nova arquitetura do algoritmo Berlekamp-Massey.

Algoritmo 3 Berlekamp-Massey sem inversão reformulado

$\sigma^{(0)} = 1, B^{(0)} = 1, L^{(0)} = 0, \theta^{(0)} = 1$
$\tilde{\delta}^{(0)} = s(x) + x^{3t}, \tilde{\pi}^{(0)} = s(x) + x^{3t}$
for $j = 1 : 2t$ **do**
    $\sigma^{(j)}(x) = \theta^{(j-1)}\sigma^{(j-1)}(x) - \tilde{\delta}^{(j-1)}(0)B^{(j-1)}(x)$
    $\tilde{\delta}^{(j)} = \theta^{(j-1)}\frac{\tilde{\delta}^{(x)}}{x} - \tilde{\delta}^{(j-1)}(0)\tilde{\pi}^{(j)}$
    if $\tilde{\delta}^{(j-1)}(0) \neq 0$ and $L^{(j-1)} \geq 0$ **then**
        $B^{(j)} = \sigma^{(j-1)}(x)$
        $\tilde{\pi}^{(j)} = \frac{\tilde{\delta}^{(x)}}{x}$
        $\theta^{(j)} = \sigma^{(j-1)}(0)$
        $L^{(j)} = -L^{(j-1)} - 1$
    else
        $B^{(j)} = xB^{(j-1)}$
        $\tilde{\pi}^{(j)} = \tilde{\pi}^{(j-1)}$
        $\theta^{(j)} = \theta^{(j-1)}$
        $L^{(j)} = L^{(j-1)} + 1$
    end if
end for

2.4.4 Algoritmo Euclideano

O algoritmo Euclideano é baseado no método criado pelo matemático Euclides para encontrar o máximo divisor comum entre dois números inteiros positivos [17].

O funcionamento do algoritmo se dá inicialmente pela obtenção do resto da divisão (d_1), entre dois números inteiros a e b. A seguir, b é dividido por d_1 para a obtenção de um novo resto (d_2), e o processo segue sempre dividindo o denominador da divisão anterior pelo resto

obtido até que o resto seja igual à zero, indicando o final do processo. Então, o último resto diferente de zero é o resultado da operação.

Pode ser também observado que, sendo d o máximo divisor comum entre dois inteiros positivos a e b, então existem dois inteiros f e g tal que

$$fa + gb = d \tag{2.56}$$

assim, tendo-se a, b e d, é possível realizar o processo ao contrário para encontrar os valores de f e g.

Este algoritmo pode ser também aplicado a polinômios. Recapitulando a Equação (2.34), a mesma pode ser reescrita como

$$s(x)\sigma(x) + x^{2t}\pi(x) = \Omega(x) \tag{2.57}$$

onde $\pi(x)$ é o quociente de

$$\frac{s(x)\sigma(x)}{x^{2t}} \tag{2.58}$$

para então ser aplicado o algoritmo Euclideano. Deve-se atentar ao fato de que, por definição, $\Omega(x)$ tem grau menor que t. Portanto, não é necessário realizar este procedimento até que $\Omega(x)$ seja igual à zero, e sim até que o grau de $\Omega(x)$ fique menor que t.

O Algoritmo 4 demonstra os passos necessários para a realização desta operação para a obtenção dos polinômios localizador ($\sigma(x)$), e avaliador de erros ($\Omega(x)$). A inicialização dos polinômios acontece na Linha 1. Conforme explicitado na Linha 2, o algoritmo é iterado enquanto o grau do polinômio $T(x)$ for maior que t. Cada iteração é composta pela divisão entre os polinômios $R(x)$ e $T(x)$ para a obtenção do novo resto (Linha 3), atualização da matriz auxiliar $\mathbf{A}(x)$ (Linha 4) e dos polinômios $R(x)$ e $T(x)$ (Linhas 5 e 6), onde acontece a troca dos operandos para a próxima iteração. Na Linha 7, é feita a atualização do iterador j. Quando a condição de parada do laço é atingida, as últimas instruções são executadas, as quais são o cálculo do termo corretor U (Linha 9), e a determinação dos polinômios localizador e avaliador de erros normalizados (Linhas 10 e 11).

Algoritmo 4 Euclideano

1: $R^{(0)}(x) = x^{2t}, T^{(0)}(x) = s(x), \mathbf{A}^{(0)}(x) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, j = 1$
2: **while** $deg(T^{(j)}(x)) \geq t$ **do**
3: $\quad Q^{(j)}(x) = \frac{R^{(j-1)}(x)}{T^{(j-1)}(x)}$
4: $\quad \mathbf{A}^{(j)}(x) = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(j)}(x) \end{bmatrix} \cdot \mathbf{A}^{(j-1)}(x)$
5: $\quad R^{(j)}(x) = T^{(j-1)}(x)$
6: $\quad T^{(j)}(x) = R^{(j-1)}(x) + Q^{(j)}(x)T^{(j-1)}(x)$
7: $\quad j = j + 1$
8: **end while**
9: $U = A_{22}^{(j)}(0)$
10: $\sigma(x) = U^{-1}U_{22}^{j}(x)$
11: $\Omega(x) = U^{-1}T^{(j)}(x)$

2.4.5 Algoritmo Euclideano modificado (*Modified Euclidean Algorithm* - MEA)

Com o intuito de evitar a operação de divisão polinomial presente no algoritmo Euclideano (Linha 3), utiliza-se a modificação introduzida em [7] e otimizada em [30] para o cálculo dos polinômios localizador e avaliador de erros, onde a atualização dos polinômios é realizada apenas com multiplicações e adições, evitando o emprego de divisões e inversões.

A diferença desta modificação para o algoritmo original é, ao invés de efetuar a divisão entre os operadores, é realizada uma normalização entre os polinômios para que aconteça a diminuição do grau de um dos polinômios de acordo com a diferença de graus entre eles. Assim, a cada iteração um dos polinômios tem seu grau diminuído, até que o critério de parada é atingido, ou seja, até que o grau do polinômio $R(x)$ fique menor que t.

O Algoritmo 5 descreve o funcionamento desta modificação baseada no algoritmo Euclideano. Na Linha 1, os polinômios são inicializados para começar o processo. O critério de parada é estabelecido na Linha 2 e o algoritmo itera até que o mesmo seja atendido. A primeira instrução a ser executada a cada iteração (Linha 3) é a atualização da diferença

de graus entre os polinômios, $l^{(j)}$, a qual ditará como será a atualização dos próprios polinômios. De acordo com o valor de $l^{(j)}$ (Linhas 4 a 8), é assertado o valor de $\sigma^{(j)}$, o qual fará a escolha da ordem dos polinômios. As operações entre os polinômios é realizada entre as Linhas 9 e 19. Entre as Linhas 9 e 14, é realizada a atualização caso o grau de $R(x)$ seja igual ou maior ao de $T(x)$, para que então ocorra a eliminação dos $l^{(j)}$ termos de grau maior de $R(x)$. Entre as Linhas 14 e 19, a atualização acontece para o caso do polinômio $T(x)$ ter grau maior, onde então se eliminam os seus $l^{(j)}$ termos de maior grau.

Algoritmo 5 Euclideano modificado

1: $R^{(0)}(x) = x^{2t}, T^{(0)}(x) = s(x), \lambda^{(0)}(x) = 0, \mu^{(0)}(x) = 1$
2: **while** $deg(R^{(j)}(x)) \geq t$ **do**
3: $\quad l^{(j)} = deg(R^{(j-1)}(x)) - deg(T^{(j-1)}(x))$
4: $\quad$ **if** $l^{(j)} \geq 0$ **then**
5: $\quad\quad \sigma^{(j)} = 1$
6: $\quad$ **else**
7: $\quad\quad \sigma^{(j)} = 0$
8: $\quad$ **end if**
9: $\quad$ **if** $\sigma^{(j)} == 1$ **then**
10: $\quad\quad R^{(j)}(x) = b^{(j-1)}R^{(j-1)}(x) - x^{|l_{j-1}|}[a^{(j-1)}T^{(j-1)}(x)]$
11: $\quad\quad \lambda^{(j)}(x) = b^{(j-1)}\lambda^{(j-1)}(x) - x^{|l_{j-1}|}[a^{(j-1)}\mu^{(j-1)}(x)]$
12: $\quad\quad T^{(j)}(x) = T^{(j-1)}(x)$
13: $\quad\quad \mu^{(j)}(x) = \mu^{(j-1)}(x)$
14: $\quad$ **else**
15: $\quad\quad R^{(j)}(x) = a^{(j-1)}T^{(j-1)}(x) - x^{|l^{(j-1)}|}[b^{(j-1)}R^{(j-1)}(x)]$
16: $\quad\quad \lambda^{(j)}(x) = a^{(j-1)}\mu^{(j-1)}(x) - x^{|l_{j-1}|}[b^{(j-1)}\lambda^{(j-1)}(x)]$
17: $\quad\quad T^{(j)}(x) = R^{(j-1)}(x)$
18: $\quad\quad \mu^{(j)}(x) = \lambda^{(j-1)}(x)$
19: $\quad$ **end if**
20: **end while**

As variáveis a e b são os coeficientes dos termos de maior grau dos polinômios $R(x)$ e $T(x)$, respectivamente.

2.4.6 Algoritmo Euclideano modificado sem computação do grau (*Degree Computationless ME algorithm* - DCME)

[5] parte do Algoritmo 5 para propor otimizações em uma versão que não faz o cálculo do grau dos polinômios $R(x)$ e $T(x)$. Tal condição é

alcançada alterando-se os valores iniciais do algoritmo para que os dois polinômios tenham o mesmo grau. Outra vantagem desta modificação é a eliminação da comparação do valor de $l^{(j)}$ e a posterior asserção da variável $\sigma^{(j)}$.

As novas condições iniciais do algoritmo são

$$\begin{aligned} R^{(0)}(x) &= x^{2t} \\ T^{(0)}(x) &= xs(x) \\ \lambda^{(0)}(x) &= 0 \\ \mu^{(0)}(x) &= x \end{aligned} \tag{2.59}$$

as quais garantem que os graus de $R(x)$ e $T(x)$ sejam iguais. Devido a este fato, a variável l é sempre igual à zero. Ao aplicar esta constatação à atualização dos polinômios, percebe-se que o termo $x^{|l^{(j-1)}|}$ pode ser removido das equações. Pode ser também notado que, independentemente do valor de $\sigma^{(j-1)}$, a atualização das equações torna-se

$$\begin{aligned} R^{(j)}(x) &= (xb^{(j-1)}R^{(j-1)}(x) - xa^{(j-1)}T^{(j-1)}(x)) \bmod x^{2t} \\ \lambda^{(j)}(x) &= (xb^{(j-1)}\lambda^{(j-1)}(x) - xa^{(j-1)}\mu^{(j-1)}(x)) \bmod x^{2t} \\ T^{(j)}(x) &= R^{(j-1)}(x) \\ \mu^{(j)}(x) &= \lambda^{(j-1)}(x) \end{aligned} \tag{2.60}$$

Durante a operação deste algoritmo, caso um dos polinômios $R(x)$ ou $T(x)$ fique com o coeficiente de x^{2t} igual a zero, deve-se realizar multiplicações por x até que um novo coeficiente não nulo ocupe esta posição para que então os cálculos possam continuar.

Pela Equação (2.60), nota-se que, ao final dos cálculos, os polinômios $R(x)$ e $\lambda(x)$ têm grau $2t-1$ e t, respectivamente. Porém, caso a quantidade de erros inseridos pelo canal seja inferior a t, ambos os polinômios ficam deslocados de s posições. Para garantir que os cálculos realizados para determinar a magnitude dos erros não sejam alterados devido a este resultado, a demonstração matemática é realizada a seguir.

A nova derivada do polinômio localizador de erros é definida como

$$\begin{aligned}\hat{\sigma}'(x) &= (x^s\sigma(x))' \\ &= sx^{s-1}\sigma(x) + x^s\sigma'(x)\end{aligned} \tag{2.61}$$

e para se ter certeza de que o resultado obtido pelo algoritmo de Forney é o correto, deve-se garantir a seguinte relação

$$\begin{aligned}-\left.\frac{\Omega(x)}{\hat{\sigma}'(x)}\right|_{x=\alpha^{j_l}} &= -\left.\frac{x^s\Omega(x)}{(x^s\sigma(x))'}\right|_{x=\alpha^{j_l}} \\ &= -\left.\frac{x^s\Omega(x)}{sx^{s-1}\sigma(x) + x^s\sigma'(x)}\right|_{x=\alpha^{j_l}} \\ &= -\frac{(\alpha^{j_l})^s\Omega(\alpha^{j_l})}{s(\alpha^{j_l})^{s-1}\cancelto{0}{\sigma(\alpha^{j_l})} + (\alpha^{j_l})^s\sigma'(\alpha^{j_l})} \\ &= -\frac{\cancel{(\alpha^{j_l})^s}\,\Omega(\alpha^{j_l})}{\cancel{(\alpha^{j_l})^s}\,\sigma'(\alpha^{j_l})} \\ &= -\frac{\Omega(\alpha^{j_l})}{\sigma'(\alpha^{j_l})}\end{aligned} \tag{2.62}$$

Assim, prova-se que o funcionamento deste algoritmo é independente da quantidade de deslocamentos dos polinômios em questão.

Para que haja um controle sobre a atualização dos polinômios $T(x)$ e $\mu(x)$ (uma vez que ambos não são atualizados a cada iteração do algoritmo), é necessário adicionar um sinal que simule a diferença que existiria entre os polinômios $R(x)$ e $T(x)$. Este sinal é inicializado com o valor "2", uma vez que pelo algoritmo original inicialmente a operação de atualização de $R(x)$ é realizada por duas iterações para que então ocorra a primeira atualização de $T(x)$, e decrementado a cada iteração. Quando o valor desta variável chega a zero, os polinômios $T(x)$ e $\mu(x)$ são atualizados de acordo com as respectivas equações em (2.60).

CAPÍTULO 3

Arquiteturas Comerciais

Com a popularização dos FPGAs e a facilidade na implementação em *hardware* de algoritmos complexos a partir de sua introdução no mercado, tornou-se viável criar rotinas em linguagens de alto nível de descrição de *hardware* para configurar e gerar automaticamente uma arquitetura tanto para o codificador quanto para o decodificador de código Reed-Solomon, eliminando assim, a necessidade de realizar uma nova especificação de código do início. Essa possibilidade permite um aumento na reusabilidade do código e também na confiabilidade do código, uma vez que possibilita a eliminação da intervenção humana durante a especificação dos blocos. Portanto, nas próximas seções são apresentadas algumas soluções disponíveis no mercado, tanto pagas quanto gratuitas, onde buscou-se analisar as vantagens e desvantagens de cada uma delas.

3.1 Reed-Solomon II MegaCore Function da Altera

A solução apresentada em [1] demonstra uma função própria para trabalhar em conjunto com os FPGAs da Altera, os quais são programados a partir da suíte de desenvolvimento proprietária do fabricante.

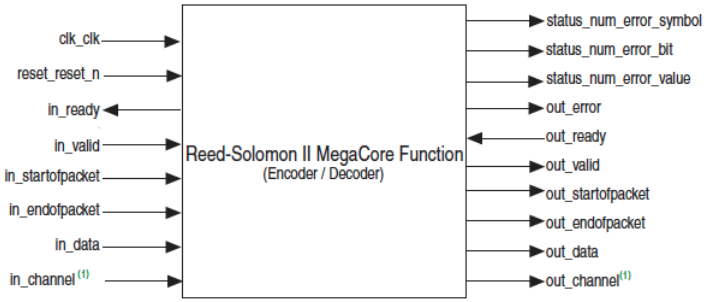

Figura 3.1: Bloco Codificador/Decodificador da Altera

Por meio da ferramenta Qsys, presente na suíte Quartus II, faz-se a configuração dos parâmetros para a geração do bloco desejado, já que não é possível gerar o codificador e o decodificador ao mesmo tempo. O tamanho de símbolo é fixo em 8 *bits*, podendo gerar palavras-código com um número de símbolos dentro do intervalo entre 204 e 255, podendo também variar a quantidade de símbolos de paridade, que pode ficar entre 2 e 66 símbolos. Outro ponto importante a ser salientado é o suporte a 16 polinômios primitivos, garantindo uma boa cobertura sobre diversos padrões que utilizam códigos Reed-Solomon de 8 *bits*.

Está presente nesta solução também, a possibilidade de configurar mais de um canal de entrada e saída. Assim, é possível codificar mais de uma mensagem, e, da mesma forma, decodificar mais de um polinômio recebido, ambos de forma entrelaçada.

Ainda é oferecido suporte para códigos encurtados, através da configuração da quantidade de símbolos tanto da mensagem quanto da palavra-código, ou seja, esta configuração não pode ser alterada durante o uso do bloco.

Esta implementação é composta por três interfaces (Figura 3.1):

- *clock* e *reset* - sinais de sincronização e inicialização do sistema, respectivamente;

- Entrada e Saída Avalon-ST - compreende sinais ligados às mensagens e às palavras-código e o barramento dos símbolos. O codificador apresenta os seguintes sinais:

  - *startofpacket* e *endofpacket*: indicam o início e o fim de cada mensagem ou palavra-código;
  - *valid*: indica dados válidos;
  - *ready*: indicação do processamento do codificador que informe quando a paridade é deslocada para fora do módulo;
  - *data*: barramento dos dados, com largura de 8 *bits*.

  O decodificador apresenta os mesmos sinais que o codificador para esta interface, com a adição de um sinal que indica quando acontece falha na decodificação.

- Status - sinais presentes no bloco decodificador, indicativos do resultado de cada processo de decodificação. Os sinais pertencentes a esta interface são:
  - *status_num_error_symbol*: indica a quantidade de erros em uma palavra-código;
  - *status_num_error_bit*: indica a quantidade de *bits* errados em uma palavra-código;
  - *status_error_value*: possui o valor de correção para cada símbolo errado.

Outro ponto que deve ser observado é que o atraso do codificador é de dois ciclos de relógio quando se usa apenas um canal.

3.2 Reed-Solomon LogiCORE IP da Xilinx

Os módulos especificados em [36] e [35] compõem uma solução completa para uma vasta gama de aplicações, idealizada para trabalhar em conjunto com projetos desenvolvidos utilizando a suíte de desenvolvimento ISE, da empresa Xilinx.

Na configuração básica dos blocos, pode-se escolher um tamanho do símbolo entre 3 e 12 *bits*, a palavra-código pode ser composta por até 4095 símbolos e possuir até 256 símbolos de paridade, configuração esta para o caso de símbolos de 12 *bits*. O circuito gerado é totalmente síncrono, usando somente um sinal de relógio.

É suportado o uso de diversos canais de entrada e saída, além do tamanho do bloco e a quantidade de símbolos de paridade poderem ser

variados entre blocos. Esta solução ainda apresenta suporte à códigos encurtados e suporte à qualquer polinômio primitivo para um dado tamanho de símbolo.

Especificamente para o bloco codificador, é possível ter saída de dados sem lacunas, ou seja, fluxo contínuo de codificação e o polinômio gerador pode ser definido pelo usuário.

Conforme pode ser visto na Figura 3.2, a interface de comunicação deste bloco é composta pelos seguintes sinais:

- *DATA_IN* e *DATA_OUT* - barramentos de dados, com largura entre 3 e 12 *bits*;
- *START* - indicação do início de uma nova mensagem;
- *N_IN* - sinal utilizado quando o tamanho de cada bloco for variável;
- *R_IN* - sinal utilizado quando a quantidade de símbolos de paridade for variável;
- *BYPASS* - quando este sinal está ativo, os símbolos na entrada são passados diretamente para a saída sem serem utilizados no cálculo da paridade;
- *ND* - quando este sinal está ativo, os dados de entrada são amostrados;
- *CLK*, *SCLR* e *CE* - sinais para sincronização, inicialização e habilitação do módulo, respectivamente;
- *INFO* - sinal indicativo de informação no barramento de saída;
- *RDY* - indicação de dados válidos na saída;
- *RFD* - sinal que indica o estado do codificador, o qual fica ativo quando nenhum processamento está acontecendo;
- *RFFD* - este sinal indica quando o módulo está pronto para iniciar um novo processo de codificação.

Já no bloco decodificador, tem-se suporte a correção de erros e apagamentos. A capacidade de correção é parametrizável, sendo também

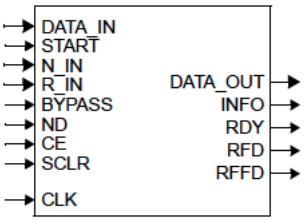

Figura 3.2: Bloco Codificador do ISE

possível a entrada de dados sem lacunas; ou seja, fluxo contínuo de decodificação. Ainda é possível optar por suporte a puncionamento.

Conforme pode ser visto na Figura 3.3, o decodificador apresenta alguns sinais em comum com o codificador. Os demais sinais de interfaceamento deste módulo são:

- *MARK_IN* e *MARK_OUT* - marcação de símbolos;
- *SYNC* - temporização de controle da entrada;
- *ERASE* - indicação de apagamento do símbolo de entrada;
- *PUNC_SEL* - seleção de um padrão de puncionamento;
- *DATA_DEL* - saída de dados não corrigidos;
- *ERASE_CNT* - apresenta a contagem de símbolos indicados como apagados;
- *ERR_CNT* - apresenta a quantidade de erros, apagamentos e puncionamentos corrigidos;
- *ERR_FOUND* - indica se algum erro, apagamento ou puncionamento foi encontrado em um dado polinômio recebido;
- *FAIL* - indica quando o polinômio recebido não pôde ser corrigido;
- *BLK_STRT* e *BLK_END* - sinalização de início e término de um bloco no barramento de saída;
- *INFO_END* - indicação do término dos símbolos de informação no barramento de saída;

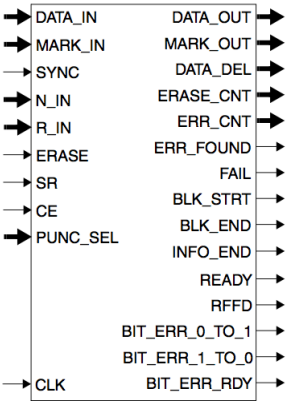

Figura 3.3: Bloco Decodificador do ISE

- *BIT_ERR_1_TO_0* e *BIT_ERR_0_TO_1* - contabilizam as inversões de *bit* nos símbolos que apresentam erro;
- *BIT_ERR_RDY* - quando ativo, indica que são válidos os sinais *BIT_ERR_1_TO_0* e *BIT_ERR_0_TO_1*.

Além de ser possível configurar uma combinação de acordo com a necessidade do usuário, são também oferecidas configurações prontas para padrões utilizados para os mais diversos fins.

Como desvantagem destes blocos, deve-se salientar que o atraso mínimo do codificador é de três ciclos de relógio, ficando maior de acordo com o padrão escolhido para ser implementado.

Para a suíte mais atual da Xilinx, o Vivado, novas versões para o codificador e o decodificador são apresentadas em [39] e [38].

O funcionamento destes blocos é similar à versão anterior, com a diferença de que, para estes, alguns sinais que antes eram explícitos, agora fazem parte de um barramento próprio.

Como visto na Figura 3.4, o sinal *s_axis_input_tdata* é o barramento de entrada de dados e é preenchido com zeros para que a sua largura seja um múltiplo de oito *bits*. O mesmo acontece com o barramento *s_axis_ctrl_tdata*, o qual compreende os sinais *N_IN* e *R_IN*, também preenchido com zeros. O sinal de saída *m_axis_output_tdata* é formado pelo barramento de dados de saída e o sinal indicativo de que naquele pacote existe um símbolo de mensagem ou de paridade.

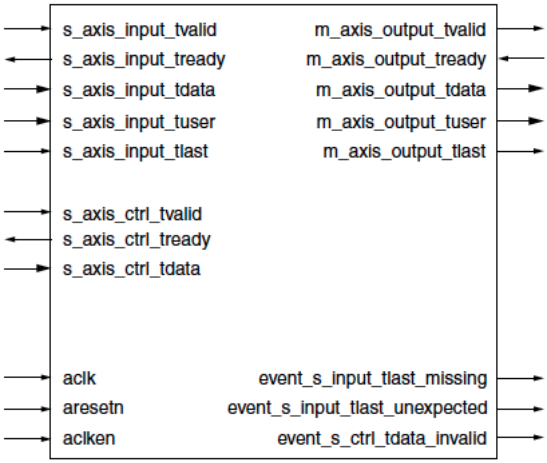

Figura 3.4: Bloco Codificador do Vivado

Igualmente, o bloco decodificador apresenta alguns barramentos que são agregações de vários sinais. Conforme a Figura 3.5, o barramento *s_axis_input_tdata* é composto pelo barramento de entrada de dados e pelo sinal que indica se o símbolo daquele pacote deve ser apagado. A agregação *s_axis_ctrl_tdata* indica o tamanho do bloco, a quantidade de símbolos de paridade e o padrão de puncionamento, caso seja utilizado. Na saída, o barramento *m_axis_output_tdata* é composto pelo símbolo corrigido e o original e um sinal indicando se o símbolo em questão é de informação ou de paridade. Por fim, como barramento de status, *m_axis_stat_tdata* possui sinais que indicam a quantidade de *bits* recebidos como '1' e corrigidos para '0', como também a quantidade de *bits* recebidos como '0' e corrigidos para '1'; se qualquer erro, apagamento ou puncionamento foi detectado no bloco atual e se o processo de decodificação falhou ou não. Apresenta também, a quantidade de apagamentos e erros corrigidos. Todos os barramentos citados acima são preenchidos com zeros para que o tamanho total de cada um seja em múltiplos de um *byte*.

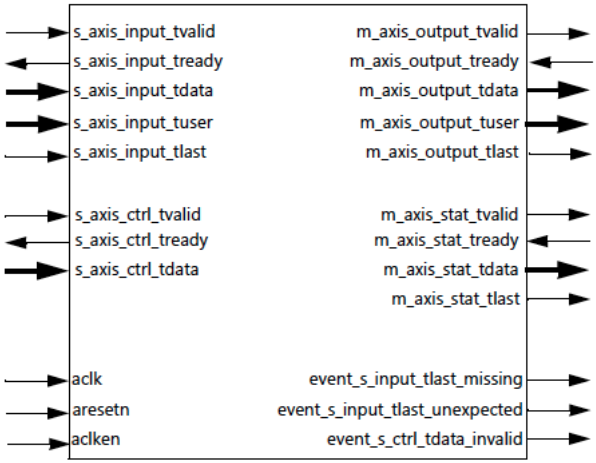

Figura 3.5: Bloco Decodificador do Vivado

3.3 Reed-Solomon Dynamic Block da Lattice

A proposta apresentada pelo conjunto de [21] e [20] é outra solução completa que atende os mais diversos projetos, apresentando uma flexibilidade bastante grande. Porém, fica dependente das ferramentas disponibilizadas pela empresa Lattice.

Os principais pontos desta implementação são a possibilidade de parametrização do tamanho do símbolo (entre 3 e 12 *bits*), configuração dos polinômios primitivo e gerador, assim como do comprimento das palavras-código e a quantidade de símbolos de paridade. Além de possibilitar a parametrização conforme a necessidade, esta ferramenta apresenta configurações prontas para diversos padrões industriais. Esta arquitetura trabalha com sincronia sobre todo o circuito. Como último ponto, existe ainda a possibilidade de trabalhar com tamanho variável para a palavra-código, assim como quantidade variável de símbolos de paridade.

Além disso, especificamente para o decodificador, é possível obter modos de correção de erros, apagamentos e puncionamentos e medidas de desempenho de correção.

Como pode ser visto na Figura 3.6, os sinais que fazem o interfaceamento do módulo codificador são:

- *din* e *dout* - barramentos de entrada e saída de dados;
- *clk*, *rstn* e *enable* - sinais de sincronização, inicialização e habilitação do circuito, respectivamente;
- *byp* - indicação de que, quando ativo, os símbolos na entrada não farão parte do cálculo da paridade;
- *ibstart* - sinal que indica o início de uma nova mensagem;
- *blocksize* - quantidade de símbolos por palavra-código, quando tamanho variável for configurado;
- *numchks* - quantidade de símbolos de paridade, quando puncionamento ou variação do tamanho da paridade forem selecionados;
- *status* - indicação de que os símbolos de informação estão no barramento de saída ou que *byp* está ativo;
- *outvalid* - sinal de indicação de que sinais na saída são válidos;
- *rfi* - ativo quando o módulo está pronto para iniciar um novo processo de codificação;
- *obstart* - indicação de que o primeiro símbolo da palavra-código está no barramento de saída;
- *obend* - indicação de que o último símbolo da palavra-código está no barramento de saída;
- *rfib* - sinal que indica que o codificador está pronto para receber uma nova mensagem;
- *ibend* - indicação de que o bloco está processando o último símbolo da mensagem.

O decodificador possui alguns sinais em comum com o codificador, como mostrado na Figura 3.7. Os demais sinais que fazem parte da interface do decodificador são:

- *ce* - habilitação do sinal de relógio;
- *sr* - inicialização síncrona do circuito;

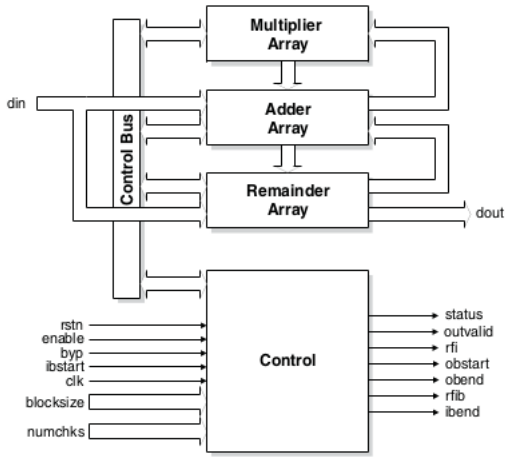

Figura 3.6: Bloco Codificador da Lattice

- *ers* - sinal que indica que o símbolo atual no barramento de entrada deve ser apagado;
- *puncsel* - seleção do padrão de puncionamento;
- *ddel* - dado original de entrada não corrigido;
- *errfnd* - indicação de que pelo menos um erro foi encontrado durante o processo;
- *fail* - indicação de falha no processo de decodificação;
- *errcnt* - contabilização da quantidade de erros corrigidos;
- *erscnt* - contabilização da quantidade de apagamentos corrigidos.

3.4 Reed-Solomon Decoder IP Core da Factorial

A especificação de [10] apresenta um bloco decodificador que pode ser configurado para ser utilizado em diversas aplicações.

Esta ferramenta foi desenvolvida utilizando a linguagem de programação Java, e, portanto, depende da disponibilidade do *Sun Java Runtime Environment Standard Edition* 6. O módulo gerado é descrito

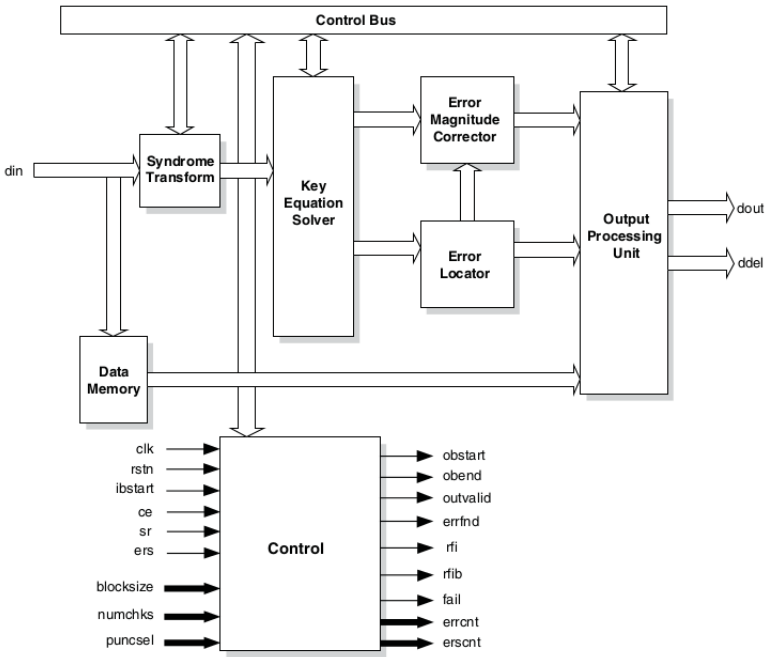

Figura 3.7: Bloco Decodificador da Lattice

em VHDL, e é independente de tecnologia e de dispositivo, podendo ser agregado imediatamente a processos que tenham como alvo tanto FPGAs quanto ASICs.

As principais características desta implementação são alto desempenho e sincronicidade utilizando apenas um sinal de relógio, configuração a partir da especificação dos parâmetros do código ou de padrões predefinidos, suporte a entrada e saída contínuas de dados sem lacunas, um símbolo de entrada e saída por ciclo de relógio, configuração do tamanho do símbolo entre 3 e 12 *bits* e utilização de qualquer polinômio primitivo válido, indicação de erros e falha na decodificação e contagem dos erros corrigidos, e ainda suporte à marcação de símbolos.

Conforme pode ser visto na Figura 3.8, os sinais de entrada e saída do bloco são:

- *clk*, *reset* e *ce* - sinais de sincronização, inicialização e habilitação do circuito, respectivamente;
- *sync_in* e *sync_out* - indicações do início de um novo bloco;
- *data_in* e *data_out* - barramentos de entrada e saída de dados;
- *marker_in* e *marker_out* - marcação de símbolos;
- *erasure_in* - indicação de que um símbolo de entrada deve ser apagado;
- *delayed_out* - símbolo original não corrigido;
- *error_found* - sinal de indicação de que pelo menos um erro foi encontrado no processamento da última palavra recebida;
- *error_count* - quantidade de erros corrigidos da última palavra recebida;
- *fail* - indicação de falha na decodificação;
- *sync_end* - sinalização do último símbolo de uma palavra-código;
- *info_end* - sinalização do último símbolo de informação de uma palavra-código.

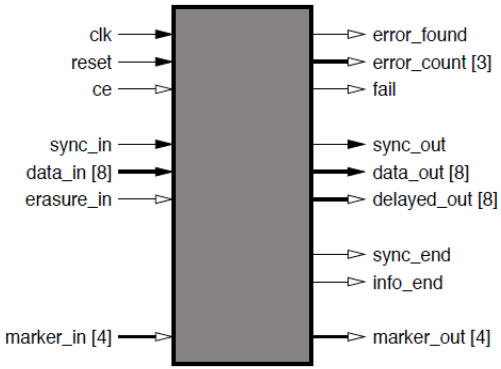

Figura 3.8: Bloco Decodificador da Factorial

Alguns dos sinais são opcionais e dependentes de outros sinais; ou seja, caso seja necessário um determinado sinal, outros sinais devem ser adicionados para que o mesmo possa ser incluído na arquitetura.

O atraso mínimo desta implementação é dado por

$$L = n + \frac{(n-k)}{2} + t + 2 \tag{3.1}$$

quando o suporte a apagamentos não é utilizado.

3.5 IP Core Open Source da System LSI

A solução de [34] apresenta uma plataforma *open source* para gerar blocos parametrizáveis. O código gerado por esta ferramenta é uma descrição em Verilog do comportamento do codificador ou do decodificador, ou ainda de ambos, o qual pode ser importado para qualquer suíte de desenvolvimento, uma vez que o resultado desta ferramenta é o próprio código fonte, diferentemente de *softwares* proprietários onde a descrição dos blocos não permite a portabilidade entre eles.

Esta ferramenta gera codificadores e/ou decodificadores para símbolos com tamanho entre 3 e 12 *bits*. De acordo com o tamanho do símbolo, trabalha com os intervalos entre 1 e 4093 símbolos de mensagem e entre 3 e 4095 símbolos de palavra-código.

Para a configuração do codificador é possível escolher o polinômio

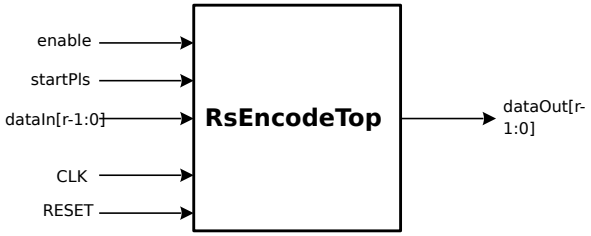

Figura 3.9: Bloco Codificador da implementação *open source* da System LSI.

primitivo a ser usado, bem como inserir atraso para o dado de entrada, e conforme pode ser visto na Figura 3.9, a interface é dada pelos sinais:

- *enable* - habilitação do sistema;
- *startPls* - indicação do início de uma nova mensagem;
- *dataIn* e *dataOut* - barramentos de entrada e saída dos símbolos, respectivamente;
- *CLK* e *RESET* - sinais de sincronização e inicialização do circuito, respectivamente.

Já na configuração do decodificador, é possível acrescentar correção de apagamentos e sinalização tanto de que erros foram encontrados para a palavra recebida quanto de falha na decodificação, conforme visto na Figura 3.10. Os sinais de entrada e saída deste bloco são:

- *enable* - habilitação do circuito;
- *startPls* - indicação de início do processo de decodificação;
- *erasureIn* - sinalização de que o símbolo no barramento de entrada deve ser apagado;
- *dataIn* e *dataOut* - barramentos de entrada e saída de dados;
- *CLK* e *RESET* - sinais de sincronização e inicialização do circuito;
- *outEnable* - indicação de dados válidos no barramento de saída;
- *outStartPls* - indicação do início dos dados decodificados;

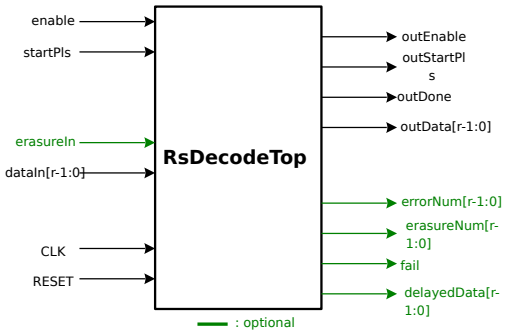

Figura 3.10: Bloco Decodificador da implementação *open source* da System LSI.

- *outDone* - sinalização do término do processamento;
- *errorNum* - contabiliza a quantidade de erros corrigidos;
- *erasureNum* - aponta a quantidade de apagamentos;
- *fail* - indica o resultado da decodificação, sendo ele êxito ou impossibilidade de correção da palavra recebida;
- *delayedData* - dados de entrada sem alteração.

Uma desvantagem desta ferramenta é que a mesma foi desenvolvida utilizando bibliotecas próprias de um sistema operacional, impossibilitando seu uso em outras plataformas.

3.6 Sumarização das arquiteturas comerciais

Analisando as arquiteturas comerciais escolhidas, é possível sumarizar as características comuns a todas. A Tabela 3.1 lista as características consideradas mais importantes que se encontram disponibilizadas pelas arquiteturas estudadas.

A Tabela 3.2 apresenta os sinais de interface que cada arquitetura apresenta.

[1] Alguns fabricantes se referem a *pipeline* como fluxo contínuo de entrada de símbolos.

Funcionalidades	Arquiteturas				
	3.1	3.2	3.3	3.4	3.5
Ferramenta de autoconfiguração	sim	sim	sim	sim	sim
Codificador	sim	sim	sim	não	sim
Decodificador	sim	sim	sim	sim	sim
Correção de apagamentos	não	sim	sim	sim	sim
Puncionamento	não	sim	sim	não	não
Múltiplos canais	sim	sim	não	não	não
Marcação de símbolos	não	sim	não	sim	não
Ignorar símbolos	não	não	sim	não	não
Diferente tamanho de bloco por iteração	não	sim	sim	não	não
Diferente capacidade de correção por iteração	não	sim	sim	não	não
Pipeline	não	sim	não	sim	sim

Tabela 3.1: Comparação das funcionalidades oferecidas por cada arquitetura

Sinais de interface	Arquiteturas				
	3.1	3.2	3.3	3.4	3.5
Falha	sim	sim	sim	sim	sim
Início de pacote na entrada	sim	sim	sim	sim	sim
Final de pacote na entrada	sim	não	não	não	não
Início de pacote na saída	sim	sim	sim	sim	sim
Final de pacote na saída	sim	sim	sim	sim	não
Número de erros corrigidos	sim	sim	sim	sim	sim
Número de apagamentos corrigidos	não	sim	sim	não	sim
Tamanho do bloco	não	sim	sim	não	não
Número de *bytes* de paridade	não	sim	sim	não	não
Apagar símbolo atual	não	sim	sim	sim	sim
Palavra recebida original	não	sim	sim	sim	sim

Tabela 3.2: Sinais de interface de cada arquitetura

Arquiteturas	Parâmetros		
	m	n	$2t$
3.1	8	$204 - 255$	$2 - 66$
3.2	$3 - 12$	$4 - 4095$	$2 - 256$
3.3	$3 - 12$	$5 - 4095$	$4 - 256$
3.4	$3 - 12$	$4 - 4095$	nd[3]
3.5	$3 - 12$	$3 - 4095$	nd[3]

Tabela 3.3: Possíveis configurações das arquiteturas comerciais.

Finalmente, a Tabela 3.3 demonstra as possíveis configurações de cada uma das arquiteturas estudadas, onde m é a largura dos símbolos, n é o comprimento das palavras-código e $2t$ é a quantidade de símbolos de redundância em cada palavra-código.

A partir da análise da Tabela 3.1, percebe-se que todas as arquiteturas estudadas possuem uma ferramenta de autoconfiguração e apresentam a descrição do decodificador. Além disso, o codificador está presente em quase todas as arquiteturas, assim como a opção por correção de apagamentos. Como as demais funcionalidades citadas não são compartilhadas por todos os fabricantes, limitou-se o escopo do desenvolvimento deste trabalho àquelas mais comuns, e que envolvem a teoria estudada no Capítulo 2.

Da mesma forma, a Tabela 3.2 apresenta os sinais que as arquiteturas tem em comum além dos sinais básicos de controle (*clock* e *reset* na entrada e *ready* na saída), tanto no codificador quanto no decodificador. O sinal **falha** é utilizado para indicar uma falha de decodificação, e está presente apenas no decodificador. O sinal **início de pacote na entrada** indica que uma nova iteração, tanto do processo de codificação quanto de decodificação, está iniciando. Já o sinal **início de pacote na saída** indica que os símbolos da palavra-código, no caso da codificação, ou da palavra estimada, no caso da decodificação, serão disponibilizados na saída. Opcionalmente, nos casos em que informações sobre o processo de decodificação sejam necessários, os sinais **número de erros corrigidos** e **número de apagamentos corrigidos** podem ser

[2]Sinais disponíveis nas arquiteturas que disponibilizam alteração deste parâmetro entre iterações.

[3]Dados não disponibilizados pelo fabricante.

adicionados à interface do decodificador. E, no caso de ser adicionado suporte à correção de apagamentos, o sinal **apagar símbolo atual** é adicionado à interface do decodificador. Outro sinal opcional que pode ser adicionado é **palavra recebida original**, onde, ao mesmo tempo em que os símbolos estimados são deslocados para fora, os símbolos originais não corrigidos também são deslocados para a saída. Os sinais **tamanho do bloco** e **número de *bytes* de paridade**, por estarem ligados à possibilidade de utilização de diferentes tamanhos de pacotes de dados por iteração, não são necessários nesta arquitetura. E o sinal **final de pacote na saída** tem a mesma funcionalidade do sinal **ready**, e, portanto, não tem a necessidade de ser adicionado.

CAPÍTULO 4

Arquitetura Proposta

Este capítulo apresenta a metodologia utilizada para a implementação das estruturas aritméticas necessárias para os cálculos dos processos de codificação e decodificação, bem como a estruturação geral dos blocos codificador e decodificador de códigos Reed-Solomon.

4.1 Metodologia de desenvolvimento

Como passo inicial do desenvolvimento da arquitetura dos códigos RS, a especificação dos algoritmos a serem desenvolvidos deve ser realizada. Neste caso, a especificação formal, ou seja, a descrição matemática do funcionamento dos algoritmos envolvidos está exposta nas Seções 2.2 e 2.3.

A partir da especificação do funcionamento do codificador e do decodificador, o segundo passo do desenvolvimento é a implementação desses blocos, sendo que o resultado desta etapa é apresentado neste capítulo. A estratégia utilizada para abordar esta implementação baseia-se na divisão em três etapas, a saber:

1. Implementação das estruturas que realizam as operações em corpos finitos;

2. Implementação do módulo do codificador;

3. Implementação do módulo do decodificador.

Como as operações matemáticas envolvidas são aplicadas tanto na codificação quanto na decodificação, buscou-se estabelecer a sua arquitetura primeiramente, para então começar a construção dos demais blocos.

A seguir, como a operação do codificador é mais simples do que os algoritmos empregados no decodificador, implementou-se primeiramente aquele, para então concluir esta etapa com os algoritmos pertencentes à arquitetura do decodificador.

Tendo a arquitetura implementada, para garantir que seu funcionamento está de acordo com a especificação, é realizado o passo de validação, onde a arquitetura é simulada e os resultados obtidos são comparados com modelos pré-estabelecidos. Posteriormente, a arquitetura é sintetizada fisicamente em *hardware* para garantir que a mesma pode ser utilizada em uma aplicação real.

O último passo deste processo de desenvolvimento é a avaliação dos resultados obtidos após a síntese física e a comparação destes com as arquiteturas estudadas no Capítulo 3, de onde se pode tirar as vantagens da utilização desta arquitetura sobre as demais.

Os passos de validação da arquitetura e a sua avaliação frente às arquiteturas pesquisadas estão descritos no Capítulo 5.

4.2 Operações em corpos finitos

Conforme consta no Apêndice A.2, corpos finitos são estruturas em que as operações de adição e multiplicação estão definidas. No entanto, como o escopo deste trabalho são códigos definidos em um corpo de extensão composto por m bits estendido do corpo finito binário, estas operações devem estar definidas para tais elementos.

A operação de adição em corpos finitos binários pode ser implementada em *hardware* utilizando portas XOR (Figura 4.1), uma vez que nesta operação não existe o "vai um" como na soma de inteiros. Outro ponto importante de ser mencionado é que, como em corpos binários existem somente dois elementos possíveis, as operações de adição e subtração são equivalentes.

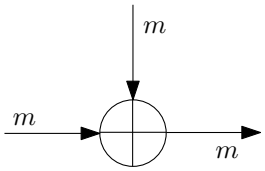

Figura 4.1: Representação do somador

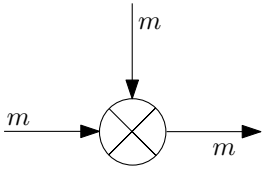

Figura 4.2: Representação do multiplicador polinomial

Diferentemente da operação de adição e subtração, as demais operações não são triviais e serão discutidas a seguir.

4.2.1 Multiplicação

A implementação do multiplicador segue a descrição polinomial presente em [17], de onde se tira a relação

$$w(x) = u(x)v(x) \bmod p(x) \tag{4.1}$$

para a multiplicação polinomial. Neste caso, cada elemento do corpo é visto como um polinômio de grau máximo $m - 1$; cada coeficiente pode ter valor "0" ou "1". A Figura 4.2 demonstra a interface desta operação.

Como pode ser visto na Equação (4.1), a multiplicação tradicionalmente deve ser realizada em dois passos

(i) multiplicação entre os dois polinômios $u(x)$ e $v(x)$

(ii) divisão pelo polinômio $p(x)$ para obtenção do resto

Esta divisão gera uma degradação da *performance* do circuito. Porém, como o polinômio $p(x)$ é conhecido, é possível gerar um circuito onde ambos os passos ocorrem paralelamente, aumentando a eficiência deste bloco.

Dado que $u(x)$ e $v(x)$ são da forma

$$u(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_{m-1} x^{m-1} \tag{4.2}$$

$$v(x) = v_0 + v_1 x + v_2 x^2 + \cdots + v_{m-1} x^{m-1} \tag{4.3}$$

aplicando-os em (4.1), tem-se

$$\begin{aligned} w(x) &= (d_0 + d_1 x + \cdots + d_{m-1} x^{m-1} \\ &\quad + d_m x^m + \cdots + d_{2m-2} x^{2m-2}) \bmod p(x) \\ &= d_0 + d_1 x + \cdots + d_{m-1} x^{m-1} \\ &\quad + d_m [x^m \bmod p(x)] + \cdots + d_{2m-2} [x^{2m-2} \bmod p(x)] \end{aligned} \tag{4.4}$$

onde

$$d_k = \begin{cases} \sum_{i=0}^{k} u_i v_{k-i} & k = 0, 1, 2, \cdots, m-1 \\ \sum_{i=k}^{2m-2} u_{k-i+(m-1)} v_{i-(m-1)} & k = m, m+1, \cdots, 2m-2 \end{cases} \tag{4.5}$$

Devido ao fato de que $p(x)$ é conhecido, os termos com grau acima de m podem ser pré-computados. Sendo assim, tendo

$$\begin{aligned} r_i(x) = r_{i,0} + r_{i,1} x + \cdots + r_{i,(m-1)} x^{m-1} = x^i \bmod p(x) \\ m \leq i \leq 2m-2 \end{aligned} \tag{4.6}$$

obtém-se os coeficientes do produto $w(x)$ de acordo com

$$w(x) = w_0 + w_1 x + \cdots + w_{m-1} x^{m-1} \tag{4.7}$$

onde

$$w_j = d_j + \sum_{i=m}^{2m-2} d_i r_{i,j} \qquad (j = 0, 1, \cdots, m-1) \tag{4.8}$$

Esta arquitetura pode ser implementada utilizando apenas portas AND e XOR [28], como no exemplo apresentado na Figura 4.3.

Segundo [28], esta arquitetura apresenta atraso inferior às outras

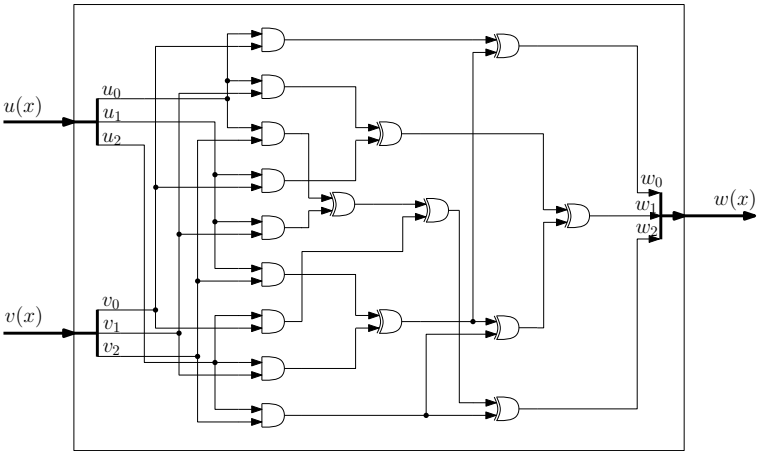

Figura 4.3: Exemplo de multiplicador polinomial para $m = 3$ e $p(x) = x^3 + x + 1$

arquiteturas utilizando quantidade semelhante de componentes internos.

4.2.2 Inversão

A operação de divisão é necessária no processo de decodificação, mais especificamente, na etapa de correção da palavra recebida (Equação (2.37)). Porém, como a operação de divisão não é tão facilmente implementada, opta-se pela operação de inversão seguida por uma operação de multiplicação.

Dentre as formas de implementação da operação de inversão, existe a possibilidade de definir uma tabela em memória no modelo *Content-addressable Memory* (CAM) [19], onde o elemento a ser invertido é utilizado como o endereço desta tabela, e o elemento invertido correspondente é o conteúdo deste endereço. Assim, a operação de inversão nada mais é do que uma tabela fixa em *hardware* composta por n elementos.

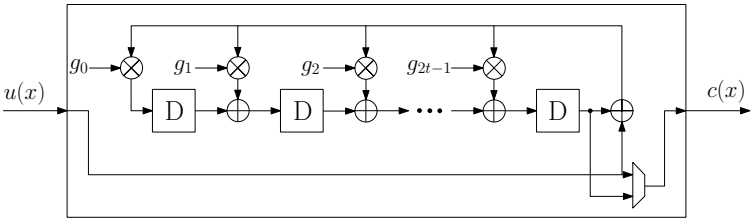

Figura 4.4: Codificador sistemático

4.3 Codificador

O próximo passo do processo de implementação foi o bloco do codificador, o qual necessita das operações de adição e multiplicação.

Para facilitar o processo de codificação, este bloco foi implementado visando a codificação sistemática, conforme descrito na Seção 2.1.2. Como a mensagem fica deslocada $2t$ posições para então adicionar os símbolos de paridade nas posições de menor grau do polinômio a ser enviado, a abordagem utilizada neste projeto foi a de um circuito do tipo *Linear Feedback Shift Register* (LFSR), ilustrado na Figura 4.4.

O funcionamento deste circuito ocorre da seguinte forma:

1. Recepção de um símbolo da mensagem;
2. Multiplicação deste símbolo pelos coeficientes do polinômio gerador do código;
3. Adição de cada coeficiente com o coeficiente do estágio anterior.

Seguindo estes passos, acontece a realização da operação estabelecida na Seção 2.1.2, uma vez que os símbolos são recebidos em ordem decrescente de grau.

Ao mesmo tempo em que os k símbolos de mensagem são utilizados para o cálculo da paridade, estes são disponibilizados para serem enviados na saída do bloco; após k ciclos de relógio, os $2t$ símbolos de paridade são deslocados serialmente para a saída, totalizando n ciclos de relógio para esta operação, com atraso de apenas um ciclo de relógio entre a entrada de um símbolo de mensagem e sua disponibilização na saída.

A interface projetada para este bloco possui quatro sinais de entrada: *clock* (sinal de sincronismo do circuito), *reset* (realiza a inicia-

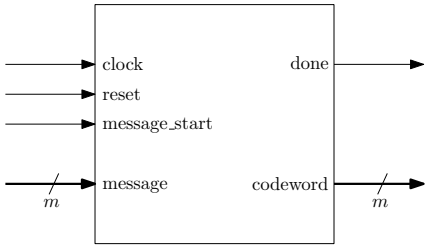

Figura 4.5: Interface do Codificador

lização do circuito), *message_start* (sinaliza que nova mensagem deve ser codificada) e *message* (o barramento composto por m *bits* por onde os k símbolos de mensagem que serão codificados são recebidos serialmente). Os sinais de saída são *done* (indica o término do processo de codificação) e *codeword* (o barramento de m *bits* por onde os n símbolos da palavra-código são disponibilizados serialmente). A Figura 4.5 esquematiza esta interface.

4.4 Decodificador

Como visto na Seção 2.3, o processo de decodificação é composto por quatro passos. A implementação deste bloco foi dividida em submódulos, os quais serão vistos a seguir.

A arquitetura interna do decodificador seguirá o modelo da Figura 4.6.

Ao final do processo de implementação, obter-se-á a interface mostrada na Figura 4.7, que abriga os submódulos.

4.4.1 Cálculo da síndrome

A síndrome de uma palavra recebida é dada pela Equação (2.27). Como citado anteriormente, cada coeficiente pode ser calculado separadamente utilizando as raízes do polinômio gerador do código. Assim, é possível implementar uma estrutura em que cada coeficiente é atualizado quando um novo símbolo da palavra recebida fica disponível na

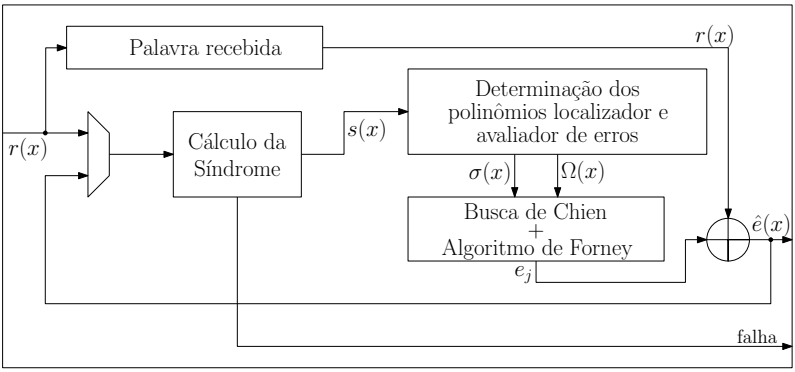

Figura 4.6: Arquitetura interna do decodificador

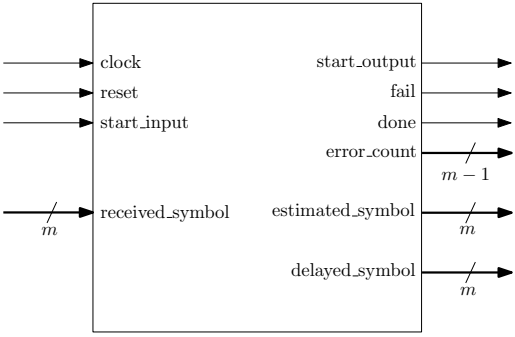

Figura 4.7: Interface do Decodificador

entrada do circuito, ou seja,

$$s_i = r(\alpha^i) = r_0 + r_1\alpha^i + r_2(\alpha^i)^2 + \cdots + r_{n-1}(\alpha^i)^{n-1} \quad (i = 1, 2, \cdots, 2t) \tag{4.9}$$

Para tal, emprega-se a técnica conhecida como regra de Horner à Equação (4.9) [17], que pode, então, ser reescrita como

$$s_i = (((r_{n-1}\alpha^i + r_{n-2})\alpha^i + r_{n-3})\alpha^i + \cdots) + r_0 \quad (i = 1, 2, \cdots, 2t) \tag{4.10}$$

Desta forma, a Equação (4.10) começa a ser determinada a partir do termo mais significativo da palavra recebida (primeiro dado recebido), e cada novo símbolo é usado, paralelamente, para efetuar a multiplicação por todas as $2t$ raízes do polinômio gerador.

Em *hardware*, esta operação é realizada com uma porta XOR para cada *bit*, um multiplicador polinomial e um registrador onde, a cada ciclo de relógio, um novo símbolo recebido é somado em módulo 2 com o resultado do ciclo anterior. O circuito resultante segue o modelo constante na Figura 4.8.

As saídas deste módulo consistem em cada um dos coeficientes de $s(x)$ em paralelo.

A interação deste submódulo com os demais é realizada através dos sinais mostrados na Figura 4.9.

4.4.2 Determinação dos polinômios localizador e avaliador de erros

Com o intuito de realizar uma implementação que tenha mínimo atraso de operação, optou-se pelos algoritmos apresentados nas Seções 2.4.3 e 2.4.6; estes possuem diversas vantagens sobre as respectivas versões anteriores de cada algoritmo.

No caso do algoritmo Berlekamp-Massey a sua versão mais atual [29] elimina as operações de inversão em duas operações do Algoritmo 1; além disto, simplifica a operação de atualização do polinômio de conexão e disponibiliza todos os coeficientes necessários para a atualização já no início da iteração posterior. Outro grande ganho desta versão é obter os dois polinômios necessários para a correção em $2t$ ciclos de relógio, diferentemente do Algoritmo 2, que requer $3t$ ciclos para alcançar

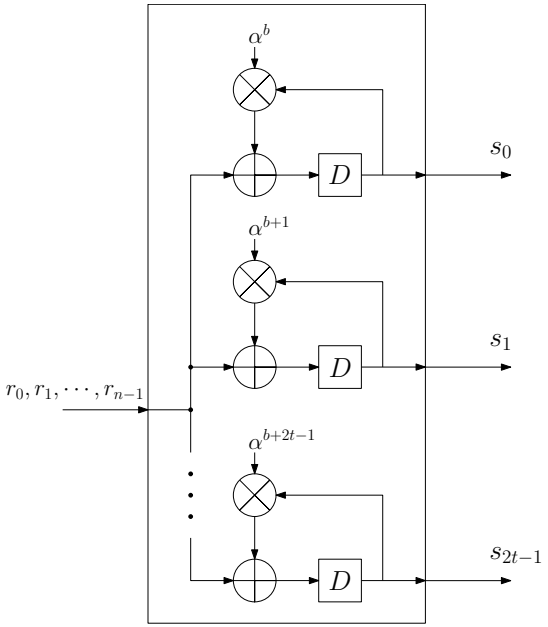

Figura 4.8: Circuito de cálculo da síndrome

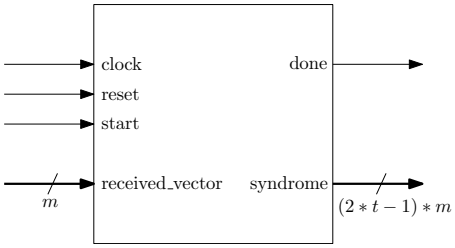

Figura 4.9: Interface do bloco de cálculo da síndrome

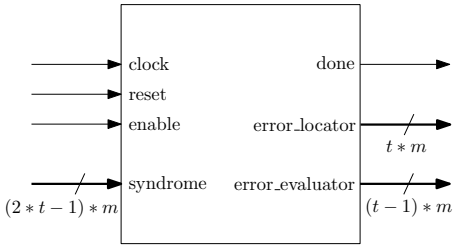

Figura 4.10: Interface do bloco de determinação dos polinômios localizador e avaliador de erros

o mesmo resultado.

Já o algoritmo Euclideano [4] proporciona a eliminação da operação de inversão do Algoritmo 4 e a eliminação do cálculo da diferença dos graus dos polinômios $R(x)$ e $T(x)$. Isto reduz a ocupação da malha de um FPGA o que por sua vez, ajuda a diminuir o atraso da operação. Mais detalhes são oferecidos nas próximas seções.

Embora dois algoritmos sejam explorados neste trabalho, sua interface de interação com os demais blocos é comum a ambos, de acordo com o esquema mostrado na Figura 4.10.

Algoritmo RiBM

O aprimoramento matemático introduzido em [29] possibilita uma arquitetura mais uniforme para o algoritmo Berlekamp-Massey sem inversão. Como a etapa de atualização dos polinômios $\sigma(x)$ e $\tilde{\delta}(x)$ acontece em todas as iterações, e sendo seu processamento idêntico, é possível criar um vetor $\hat{\delta}(x)$ de tamanho $3t+1$ para acomodar os dois polinômios e atualizá-los ao mesmo tempo com o mínimo de atraso, uma vez que a quantidade de coeficientes para cada um é de $t+1$ e $2t$, respectivamente. A condição inicial para este novo vetor é

$$\hat{\delta}^{(j)}(x) = s(x) + x^{3t} \tag{4.11}$$

ou seja, as $2t-1$ posições menos significativas são preenchidas com o polinômio da síndrome; a posição mais significativa do vetor é inicializado com o valor correspondente ao α^0. A região das $2t-1$ posições menos significativas é referente ao polinômio $\sigma(x)$ e as demais posições são referentes ao polinômio $\Omega^{(h)}(x)$; conforme as atualizações são cal-

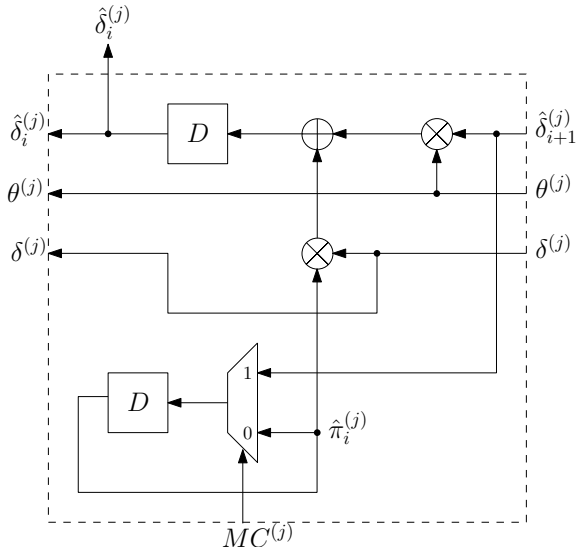

Figura 4.11: Célula de atualização dos coeficientes de $\hat{\delta}^{(j)}(x)$

culadas, ambas as regiões são deslocadas para a esquerda. Ao final das $2t$ iterações, as $t-1$ células menos significativas contêm os coeficientes do polinômio $\sigma(x)$ e as próximas t células contêm o polinômio $\Omega^{(h)}$. Deve-se salientar que as operações realizadas em cada uma das regiões não interferem entre si, garantindo assim, o cálculo correto de cada polinômio.

A partir deste novo vetor, cria-se uma arquitetura sistólica[1] composta por $3t+1$ células que atualizam cada coeficiente deste polinômio, com o auxílio de um polinômio $\hat{\pi}(x)$ de mesmo tamanho. Como mostrado na Figura 4.11, cada uma destas células é composta por dois registradores, dois multiplicadores, um somador e um multiplexador; portanto, todo o conjunto é composto por $6t+2$ registradores, $6t+2$ multiplicadores, $3t+1$ multiplexadores e $3t+1$ somadores. O sinal $MC^{(j)}$ é o controle da atualização das variáveis auxiliares.

A arquitetura completa deste algoritmo com as condições iniciais e a indicação dos sinais de saída segue o esquema proposto na Figura 4.12.

[1]Estrutura em que as unidades de processamento de dados (células) são organizadas em linhas e o resultado do processamento de uma célula é encaminhado imediatamente às células subsequentes.

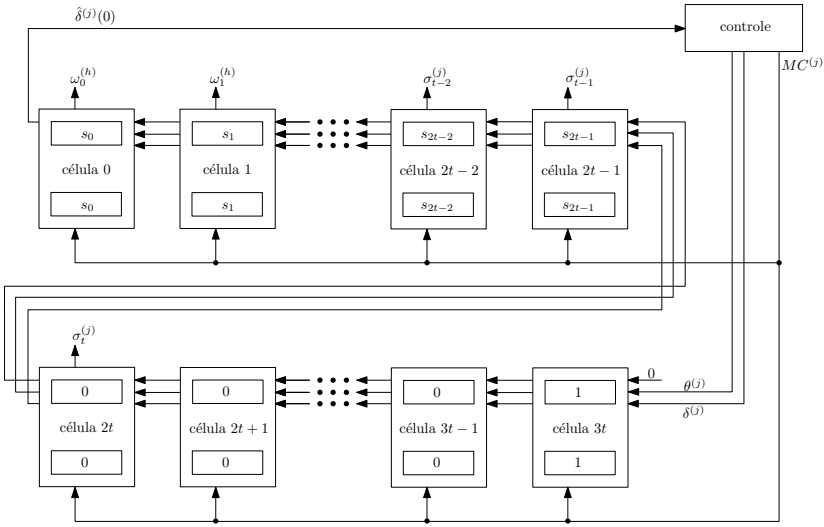

Figura 4.12: Arquitetura sistólica do algoritmo RiBM

Algoritmo E-DCME

Conforme apresentado na Seção 2.4.6, a nova formulação do algoritmo Euclideano modificado [5] dispensa as operações de deslocamento do Algoritmo 5, uma vez que os graus dos polinômios $R(x)$ e $T(x)$ são considerados iguais durante todas as iterações. Este novo paradigma dispensa o uso do circuito de detecção dos graus dos polinômios, sendo substituído por dois sinais de controle: *controle_R* (verifica quando o elemento mais significativo do vetor $R(x)$ é nulo para então deslocar os polinômios $R(x)$ e $\lambda(x)$ até que um novo elemento não nulo seja carregado na posição mais significativa de $R(x)$) e *controle_T* (faz o controle da atualização dos polinômios $T(x)$ e $\mu(x)$).

Como as novas operações de atualização dos quatro polinômios envolvidos neste algoritmo são as mesmas para todas as iterações em que não são realizados deslocamentos, pode-se construir uma estrutura que faça a atualização de cada coeficiente dos polinômios $R(x)$ e $\lambda(x)$. Inicialmente, o projeto de tal célula levava em consideração deslocamentos realizados para todos os polinômios; porém, com melhorias na estrutura do algoritmo apresentado na Seção 2.4.6, foi possível diminuir seu tamanho e especializar as células. Foi criada uma célula que atualiza os

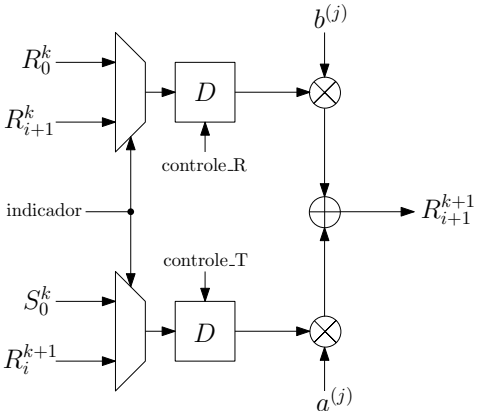

Figura 4.13: Célula de atualização dos coeficientes de $R(x)$

coeficientes de $R(x)$ com o auxílio de $T(x)$ (Figura 4.13), e uma célula que atualiza os coeficientes de $\lambda(x)$ com o auxílio de $\mu(x)$, conforme o modelo da Figura 4.14.

Sabendo que a atualização do polinômio $\lambda(x)$ depende somente dos valores $a^{(j)}$ e $b^{(j)}$, a única entrada da célula além destes coeficientes é o próprio polinômio.

Assim, cada célula é composta por dois registradores, dois multiplicadores, um somador e, no caso das células para $R(x)$, dois multiplexadores para seleção do valor de entrada. A arquitetura resultante é composta por $2t - 2$ células "superiores" (atualização de $R(x)$), e $t + 1$ células "inferiores" (atualização de $\lambda(x)$). Este arranjo pode ser visualizado na Figura 4.15.

As condições iniciais também sofreram alterações, pois a utilização das condições iniciais estabelecidas na Seção 2.4.6 geram como resultado da primeira iteração:

$$\begin{aligned} R^{(1)}(x) &= x^2 s(x) \bmod x^{2t} \\ T^{(1)}(x) &= x s(x) \\ \lambda^{(1)}(x) &= x^2 \\ \mu^{(1)}(x) &= x \end{aligned} \tag{4.12}$$

Assim, optou-se por inicializar os polinômios com os valores de

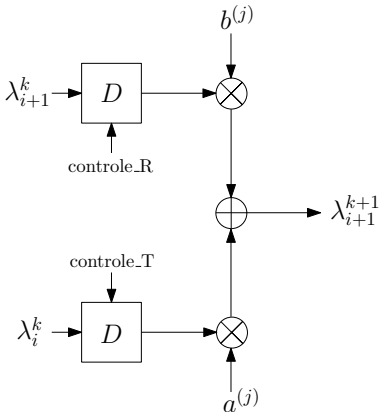

Figura 4.14: Célula de atualização dos coeficientes de $\lambda(x)$

(4.12), tendo como resultado um atraso total de $2t - 1$ ciclos de relógio.

Ao final de cada iteração, o coeficiente mais significativo de $R(x)$ deve ser igual à zero. Porém, como os elementos são deslocados para a direita durante a operação de atualização, este elemento é eliminado, deixando o termo correspondente a $a^{(j)}$ (o coeficiente do grau mais alto de $R(x)$) na posição mais significativa. Portanto, como a operação objetiva eliminar o termo mais significativo, não é necessário calcular o coeficiente resultante. Desta forma, os elementos correspondentes aos termos $a^{(j)}$ e $b^{(j)}$ são armazenados no bloco de controle, onde são necessários somente dois registradores.

4.4.3 Determinação das raízes do polinômio localizador de erros e cálculo da magnitude dos erros

Os passos finais do processo de decodificação são a determinação das raízes do polinômio localizador de erros e o cálculo da magnitude dos erros indicados pelas raízes. Porém, não se deve, necessariamente, realizar o primeiro para então realizar o próximo. Como é demonstrado em [17], pode-se construir uma arquitetura em que os polinômios $\sigma(x)$, $\sigma'(x)$ e $\Omega(x)$ são avaliados paralelamente; a magnitude é calculada usando os valores gerados e, no caso de $\sigma(x) = 0$, o valor determinado é então utilizado na correção da posição correspondente.

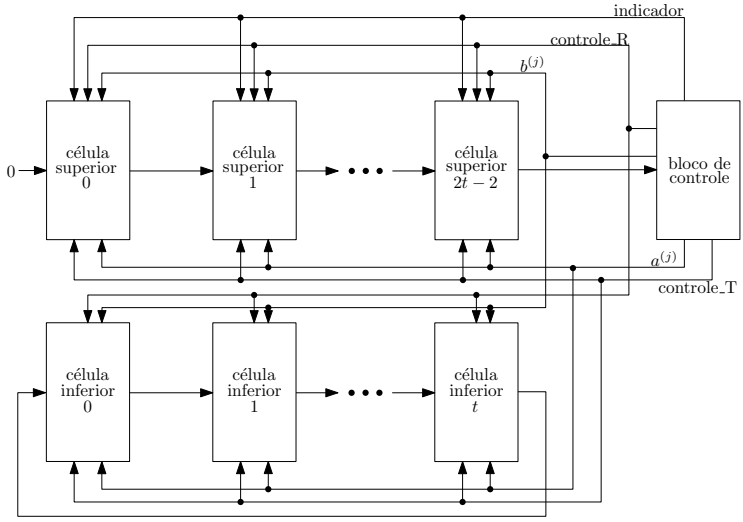

Figura 4.15: Arquitetura sistólica do algoritmo E-DCME

A arquitetura em questão segue o modelo presente na Figura 4.16.

Inicialmente, os registradores são carregados com os coeficientes dos polinômios $\sigma(x)$ e $\Omega(x)$. A cada iteração cada coeficiente é multiplicado pela potência correspondente de $x = \alpha^j$. Para diminuir o atraso do processamento, cada coeficiente é multiplicado pela mesma potência em todas as iterações para não ser necessário realizar j multiplicações antes de determinar o valor dos polinômios. Após a realização das multiplicações dos coeficientes, os mesmos são somados para que o resultado da determinação de cada polinômio esteja disponível. Faz-se, então, a verificação do resultado obtido para $\sigma(\alpha^j)$ ao mesmo tempo que ocorre a operação de inversão de $\sigma'(\alpha^j)$ e a posterior multiplicação de $\Omega(\alpha^j)$ por $\sigma'(\alpha^j)^{-1}$. Logo, caso $\sigma(\alpha^j) = 0$, o valor correspondente à correção da posição j da palavra recebida estará disponível para ser somado ao símbolo recebido.

O funcionamento deste bloco permite o deslocamento dos símbolos recebidos ao mesmo tempo em que a correção é realizada. Portanto, como todos os elementos do corpo devem ser testados, o atraso deste bloco é de n ciclos de relógio. A Figura 4.17 demonstra a interface deste bloco que permite sua comunicação com os demais submódulos.

O sinal "*error_locator*" (polinômio localizador de erros) é utilizado

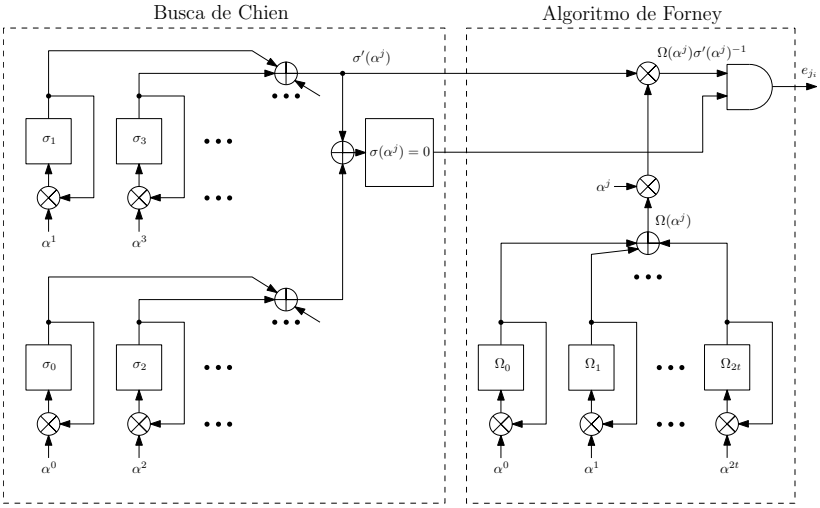

Figura 4.16: Arquitetura do bloco de determinação das raízes de $\sigma(x)$ e cálculo da magnitude dos erros

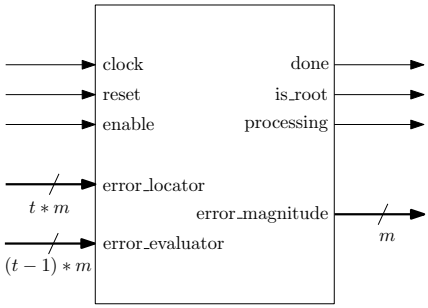

Figura 4.17: Interface do bloco de determinação das raízes de $\sigma(x)$ e cálculo da magnitude dos erros

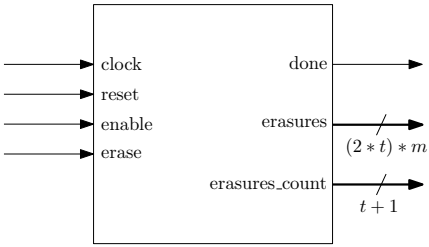

Figura 4.18: Interface do bloco de determinação do polinômio dos apagamentos

para verificar quais são as posições que devem ser corrigidas. O sinal "*error_evaluator*" (polinômio avaliador de erros) é utilizado para calcular o valor a ser somado a cada posição que deve ser corrigida. O barramento "*error_magnitude*" apresenta o valor que deve ser somado ao símbolo atual para efetuar sua correção, sendo que esta operação é sinalizada por "*is_root*", que indica que a posição atual deve ser corrigida. O sinal "*processing*" indica que este submódulo está ativo e que os símbolos devem ser passados para a saída, além de manter a sincronia com o módulo superior.

4.4.4 Correção de erros e apagamentos

Para ser possível a correção de erros e apagamentos, é necessária a adição de um submódulo que determine o polinômio localizador de apagamentos ($\Gamma(x)$) (Equação (2.39)). Este bloco opera paralelamente à determinação da síndrome; nos ciclos de relógio em que o sinal "*erasures*" está ativo, a posição correspondente ao símbolo é adicionada ao polinômio $\Gamma(x)$.

A interface idealizada para este submódulo segue o modelo constante na Figura 4.18.

O sinal "*erasures_count*" é utilizado para acompanhar o processo, pois, caso sejam marcados $2t$ ou $2t-1$ apagamentos, não será possível corrigir nenhum erro, já que estes erros estariam fora do raio de correção do decodificador, e os polinômios $s(x)$ e $\Gamma(x)$ são passados diretamente ao submódulo de determinação das raízes do polinômio localizador de erros e cálculo da magnitude dos erros. É também verificado se mais de $2t$ apagamentos existem; neste caso, o processo de decodificação é

Submódulo	Ciclos de relógio
Cálculo da síndrome	n
Determinação dos polinômios	$2t$
Determinação das raízes e correção dos símbolos	n

Tabela 4.1: Atraso dos submódulos do decodificador

interrompido e o sinal "*fail*" é acionado. Já no caso de ocorrerem menos que $2t-1$ apagamentos, o polinômio $\Gamma(x)$ é utilizado na inicialização do algoritmo de determinação dos polinômios localizador e avaliador de erros para agregar a informação da síndrome e assim, corrigir os erros que possam existir na palavra recebida.

4.5 Atrasos de operação

Após a implementação de cada um dos módulos (codificador e decodificador) e dos submódulos do decodificador (cálculo da síndrome, determinação dos polinômios e determinação das raízes e cálculo da magnitude dos erros), pode-se estabelecer o atraso de operação de cada módulo a partir das arquiteturas resultantes.

Como o codificador processa um símbolo por ciclo de relógio e desloca um símbolo de paridade para a saída por ciclo de relógio, seu atraso de operação é de n ciclos. Além disso, a palavra-código resultante fica totalmente disponível $2n+1$ ciclos de relógio após o início do processo.

Já o decodificador possui dois casos distintos. Quando uma palavra recebida não possui erros, os módulos de determinação dos polinômios e de determinação das raízes e cálculo da magnitude dos erros não são utilizados, o que causa um atraso de n ciclos de relógio para o cálculo da síndrome desta palavra recebida. E quando possui erros, todos os módulos são utilizados, aumentando significativamente o atraso do decodificador. O atraso de cada um dos submódulos é apresentado na Tabela 4.1.

Para o decodificador, no caso de não existirem erros na palavra recebida, a palavra-código fica completamente disponível após $2n+1$ ciclos de relógio. Para o caso de serem detectados erros, a palavra estimada fica completamente disponível após $2n+2t+2$ ciclos de relógio.

CAPÍTULO 5

Desenvolvimento e Avaliação

Este capítulo descreve o processo de validação do circuito desenvolvido como estudo de caso para a posterior avaliação das estruturas implementadas (Capítulo 4) em relação às arquiteturas estudadas no Capítulo 3, e apresenta também a descrição da ferramenta de geração automática da arquitetura.

Tanto o processo de validação funcional quanto a implementação da ferramenta de geração automática contaram com o auxílio do aluno de graduação João Pedro dos Reis.

5.1 Definição do código

Conforme citado no Capítulo 2, existem diversos padrões industriais que empregam os códigos Reed-Solomon (RS) para realizar a correção de erros, sendo que a configuração mais utilizada é a que tem tamanho de símbolo igual a 8 *bits* (1 *byte*), o que possibilita uma palavra-código de até 255 símbolos de comprimento.

Levando em consideração os padrões estudados, a configuração mais utilizada tem capacidade de correção de 8 símbolos, o que se traduz em 16 símbolos de paridade. Assim, o código resultante é o RS(255, 239)

Parâmetro	Valor
Tamanho de cada símbolo	$m = 8$
Comprimento da palavra-código	$n = 255$
Comprimento da mensagem	$k = 239$
Expoente da primeira raiz	$b = 0$
Capacidade de correção	$t = 8$
Polinômio primitivo	$p(x) = x^8 + x^4 + x^3 + x^2 + 1$
Polinômio gerador	$g(x) = \prod_{i=0}^{15}(x - \alpha^i)$

Tabela 5.1: Configuração do código desenvolvido

[2, 9, 13, 15].

Mesmo existindo aplicações onde diferentes configurações para o código RS são utilizadas (seja do tamanho do símbolo, da palavra-código ou da mensagem) [16, 18, 22, 30, 33, 40, 41], optou-se por desenvolver um estudo de caso utilizando a configuração mais comumente empregada em aplicações reais para que se possa avaliar os pontos principais do circuito após sua síntese lógica e física, sendo eles, a área e a frequência de operação.

A configuração escolhida para esta implementação de estudo de caso é apresentada na Tabela 5.1.

Após a finalização do desenvolvimento das arquiteturas, tanto para o codificador quanto para o decodificador, os arquivos fonte desenvolvidos foram:

- RS_top.vhd - descrição da máquina de controle da comunicação entre computador e FPGA
- serialinterface.vhd - interface de comunicação serial com *autobaud*
- ReedSolomon.vhd - pacote de definição de constantes e tipos de dados utilizados nas arquiteturas
- field_element_multiplier.vhd - descrição do multiplicador polinomial

- RS_coder.vhd - descrição do codificador
- RS_decoder.vhd - módulo topo do decodificador e agregador dos seus submódulos
- Syndrome.vhd - algoritmo de cálculo da síndrome
- Erasure.vhd - determinação do polinômio dos apagamentos
- KES.vhd - descrição da interface entre o algoritmo de determinação dos polinômios e o módulo topo
- E_DCME.vhd - algoritmo E-DCME
- RiBM.vhd - algoritmo RiBM
- Chien_Forney.vhd - descrição da interface entre o algoritmo de correção de erros e o módulo topo
- inversion_table.vhd - descrição da tabela de inversão
- CF_EDCME.vhd - algoritmo de correção de erros adaptado para o algoritmo E-DCME
- CF_RiBM.vhd - algoritmo de correção de erros adaptado para o algoritmo RiBM

5.2 Validação da arquitetura

Nesta seção são apresentadas as etapas realizadas para efetuar a validação funcional da arquitetura implementada a partir do código definido, assim como a sua sintetização em *hardware* para garantir sua usabilidade em projetos que demandem proteção dos dados.

5.2.1 Etapa 1: Validação funcional

A validação funcional da arquitetura implementada foi realizada utilizando os *softwares* MatLab versão 2012a (MathWorks), e ModelSim ASE 12 *Service Pack* 2 (Altera), onde uma rotina foi desenvolvida a fim de comparar os resultados gerados por cada um dos blocos.

A estruturação da rotina é exemplificada pelo pseudo-código apresentado no Algoritmo 6. Inicialmente (Linha 1), dois objetos (um que descreve o codificador e um que descreve o decodificador), são inicializados com os parâmetros apresentados na Tabela 5.1 para realizar as operações do laço entre as Linhas 2 e 11. Este laço é composto pelas operações de geração de uma nova mensagem (Linha 3), codificação desta mensagem (Linha 4), para então simular a inserção de erros (Linha 5) e, finalmente, decodificar esta palavra recebida simulada (Linha 6). A mensagem gerada e a palavra recebida simulada são escritas em arquivos separados (Linha 7), para serem usados como entrada da simulação (Linha 8). A quantidade de erros inseridos na geração da palavra recebida variou entre 0 e 15, para exercitar a arquitetura tanto sem erros quanto com erros, e ainda assim, dentro da capacidade de correção do código (pela definição do código escolhida, $t = 8$) e fora da capacidade de correção. As saídas geradas pelas operações no MatLab são também armazenadas para ser possível a comparação destas com as saídas das simulações, sendo elas a palavra-código gerada pelo codificador e a palavra estimada gerada pelo decodificador (Linha 9). Finalmente, na Linha 10, realiza-se a comparação dos resultados para garantir que a funcionalidade implementada está de acordo com a teoria dos algoritmos estudados no Capítulo 2.

Durante o processo de validação funcional, algumas imagens foram geradas a fim de ilustrar o funcionamento dos blocos codificador e decodificador. A Figura 5.1 apresenta o processo completo de codificação de uma mensagem, mostrando os sinais internos do codificador. A Fi-

Algoritmo 6 Rotina de simulação

1: Configuração do código Reed-Solomon
2: **loop**
3: Geração de uma mensagem
4: Codificação
5: Inserção de erros e geração da palavra recebida
6: Decodificação
7: Geração de arquivos com a mensagem e a palavra recebida
8: Simulação da arquitetura com o ModelSim usando os arquivos gerados como entrada
9: Leitura dos arquivos de saída (palavra-código e palavra estimada) da simulação com ModelSim
10: Comparação dos resultados da simulação com os resultados obtidos no MatLab
11: **end loop**

gura 5.2 apresenta o mesmo processo, porém dando ênfase à etapa final, quando, após o último *byte* de mensagem ser processado e deslocado para a saída do circuito, os *bytes* de paridade são também deslocados para a saída.

As Figuras 5.3, 5.4 e 5.5 demonstram o funcionamento do decodificador com capacidade de correção de erros apenas, sendo que neste caso $t = 8$. A Figura 5.3 demonstra o resultado da decodificação quando nenhum erro é detectado na palavra recebida, quando acontece o deslocamento da palavra recebida para a saída, já que a síndrome é igual à zero. Já a Figura 5.4 demonstra o decodificador corrigindo exatamente oito erros, e a Figura 5.5 apresenta o sinal "fail" acionado quando o decodificador tenta corrigir mais erros do que a sua capacidade, neste caso, são dez erros presentes na palavra recebida.

As Figuras 5.6, 5.7, 5.8, 5.9, 5.10 e 5.11 demonstram o decodificador com capacidade de correção de erros e apagamentos.

A Figura 5.6 mostra o decodificador exercitando toda sua capacidade de correção de apagamentos, ao corrigir 16 posições marcadas durante a recepção da palavra. Já a Figura 5.7 tem por objetivo demonstrar o comportamento do decodificador quando a capacidade de correção de apagamentos é excedida, sendo que neste exemplo são marcadas 17 posições. Quando são marcadas mais de $2t$ posições como apagamentos, o sistema automaticamente levanta o sinal "fail" após

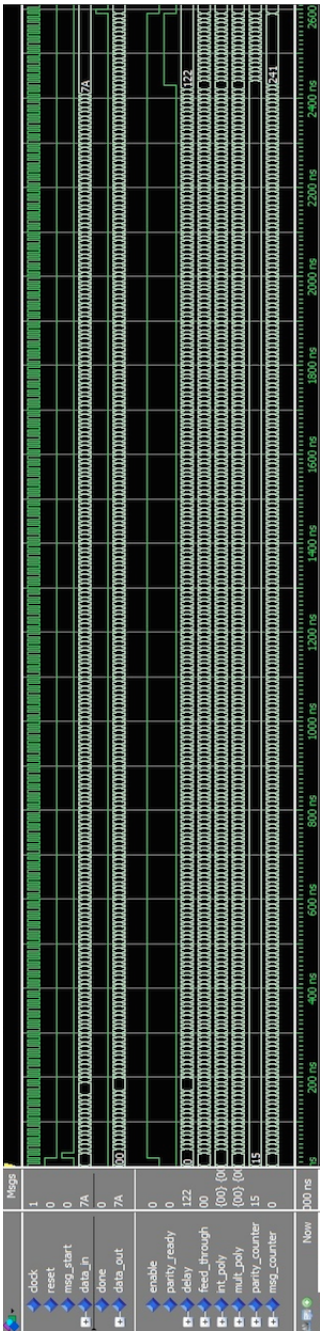

Figura 5.1: Exemplo do processo de codificação

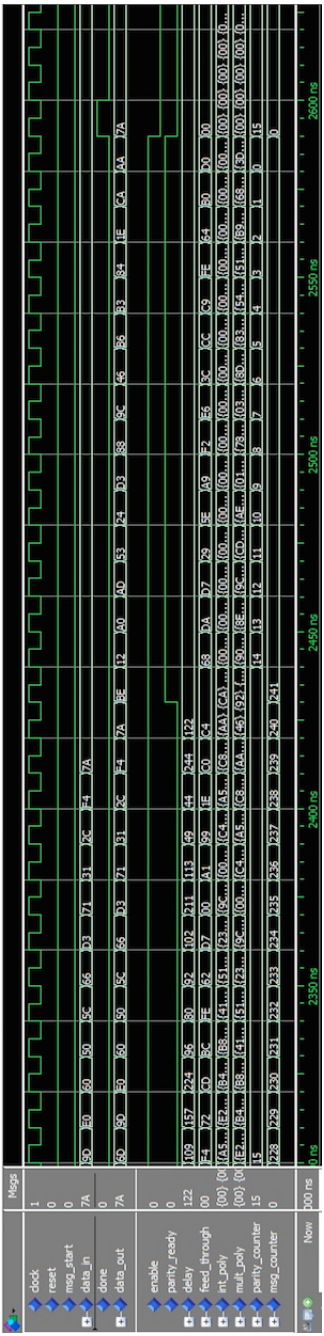

Figura 5.2: Detalhe dos símbolos de paridade

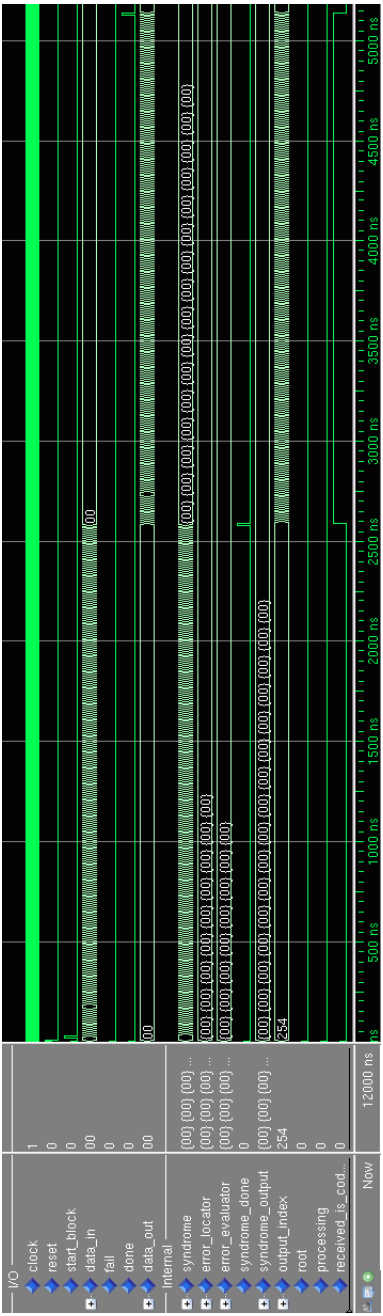

Figura 5.3: Exemplo da decodificação sem erros na palavra recebida

Figura 5.4: Exemplo da decodificação com 8 erros na palavra recebida

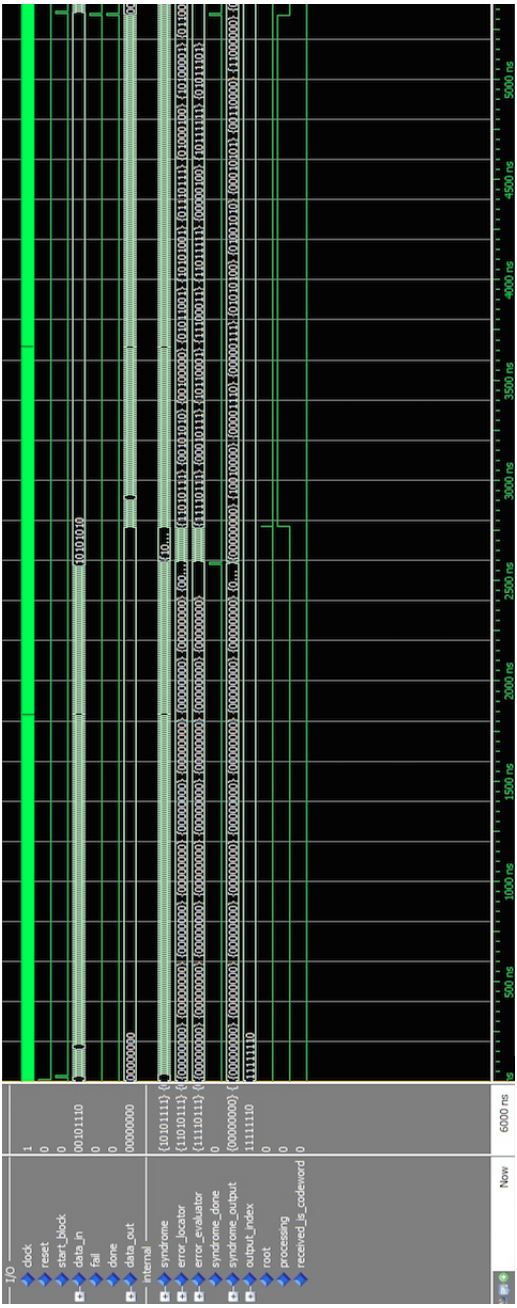

Figura 5.5: Exemplo da decodificação com 10 erros na palavra recebida

o recebimento do último símbolo da palavra recebida, uma vez que a capacidade de correção do código foi excedida.

As Figuras 5.8 e 5.9 demonstram a capacidade de correção tanto de erros quanto de apagamentos em uma mesma iteração, uma vez que esta é uma funcionalidade inerente ao código (Equação (2.38)).

Já as Figuras 5.10 e 5.11 demonstram a capacidade de correção sendo extrapolada. No primeiro caso, como são marcados 16 apagamentos, o código não é capaz de corrigir nenhum erro. No segundo caso, conforme a Equação (2.38), a quantidade de símbolos necessitando correção ultrapassa a capacidade do decodificador.

5.2.2 Etapa 2: Validação utilizando FPGA

Após a finalização da etapa das simulações, o próximo passo foi a prototipação da arquitetura desenvolvida utilizando um FPGA.

Visando a validação desta arquitetura utilizando uma plataforma que oferecesse uma frequência mais elevada de funcionamento, foi utilizada a placa de desenvolvimento XUPV5-LX110T da empresa Xilinx [37] disponível para uso em pesquisa no Grupo de Sistemas Embarcados (GSE). Esta alcança uma frequência de operação de até 100 MHz, possui 17280 *slices* programáveis (cada *slice* é composto por 4 registradores e 4 *look-up tables* (LUTs) que totalizam 69120 registradores e 69120 LUTs). Para a realização da síntese física, utilizou-se o *software* ISE versão 13.2 da empresa Xilinx.

A rotina desenvolvida para a validação das arquiteturas usando a placa de desenvolvimento com o FPGA citado seguiu a mesma estrutura descrita no Algoritmo 6. No entanto, visando a facilidade de operação envolvendo corpos finitos, optou-se por utilizar a plataforma SAGE [31], que, assim como o *software* MatLab, faz uso de *scripts* para automatizar a execução das operações desejadas. Outra vantagem desta plataforma é que, por estar integrada ao sistema operacional, diminui a latência de operação caso a quantidade de iterações de execução do *script* desenvolvido seja grande.

Para realizar a comunicação entre a rotina executada no computador e a placa de desenvolvimento, foi empregado o protocolo serial no modo 8N1 (8 *bits* de informação, sem paridade e somente um *bit* de parada), com a utilização do módulo para comunicação serial desenvolvido pelo laboratório GAPH [11] da PUCRS. Este módulo possui

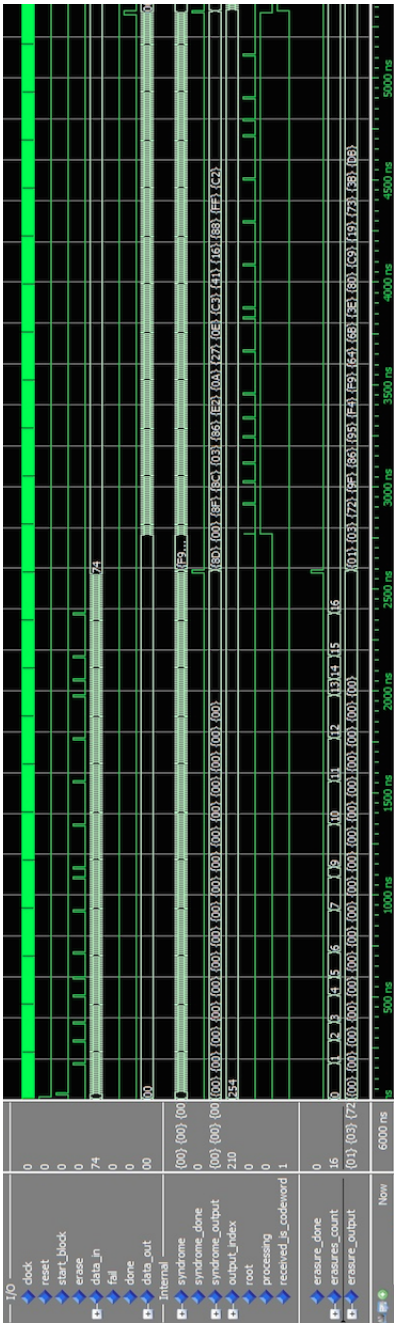

Figura 5.6: Exemplo da decodificação com 16 apagamentos na palavra recebida

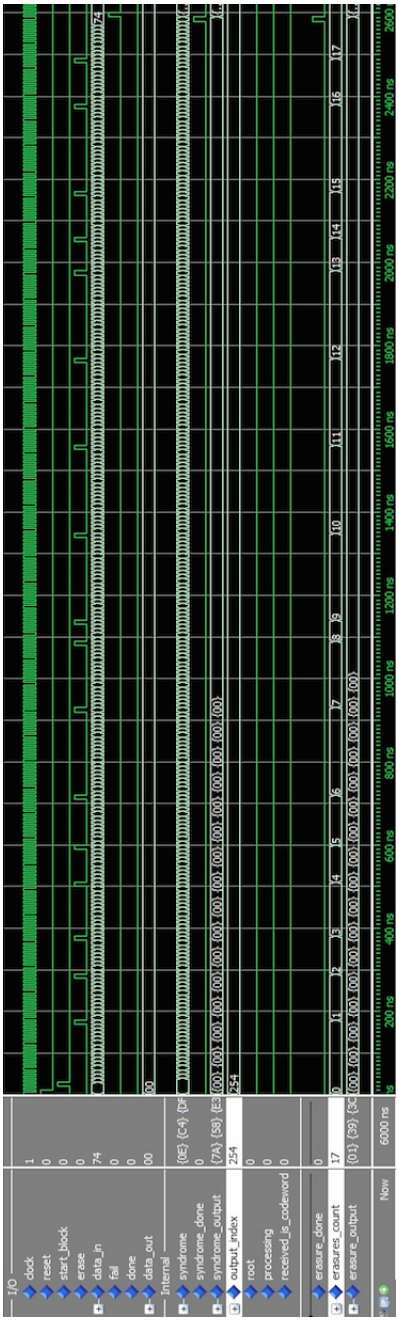

Figura 5.7: Exemplo da decodificação com 17 apagamentos na palavra recebida

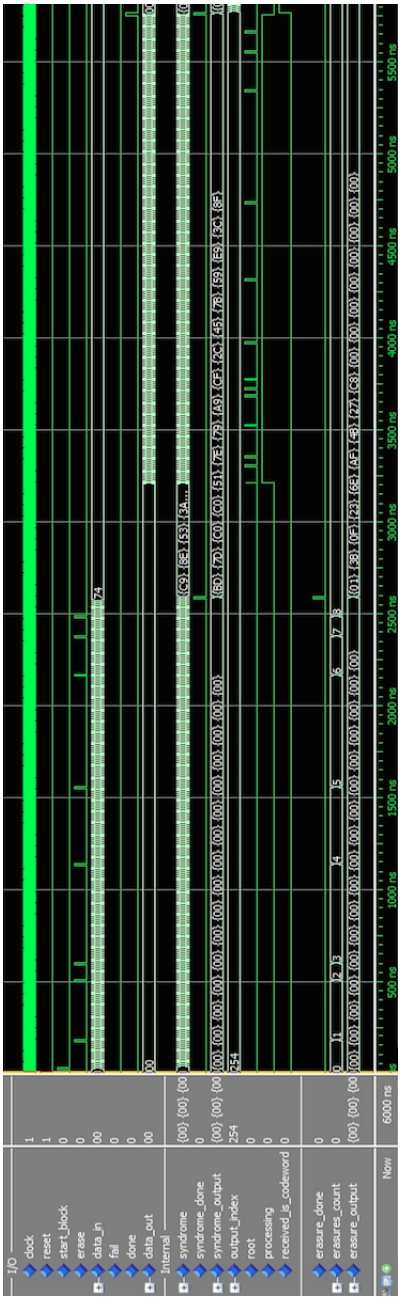

Figura 5.8: Exemplo da decodificação com 4 erros e 8 apagamentos na palavra recebida

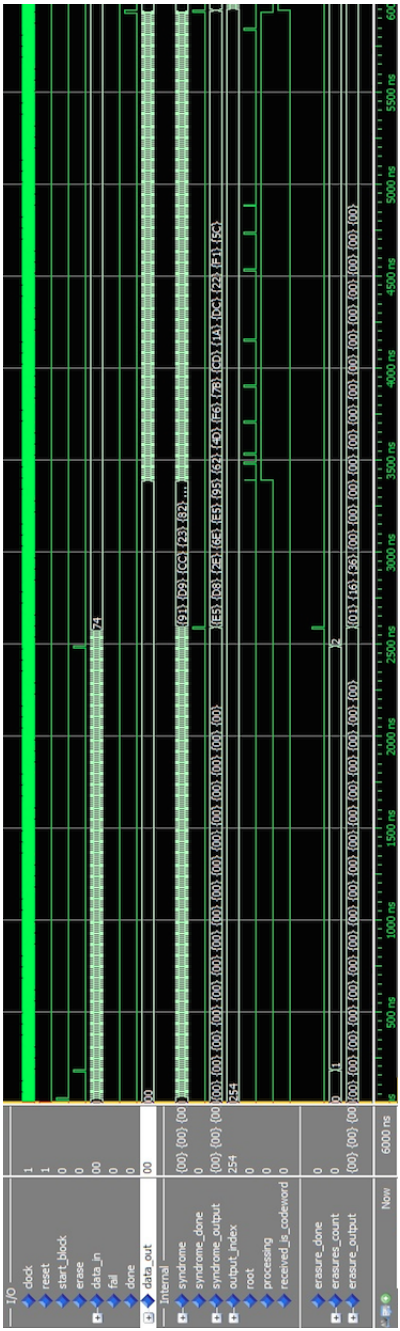

Figura 5.9: Exemplo da decodificação com 7 erros e 2 apagamentos na palavra recebida

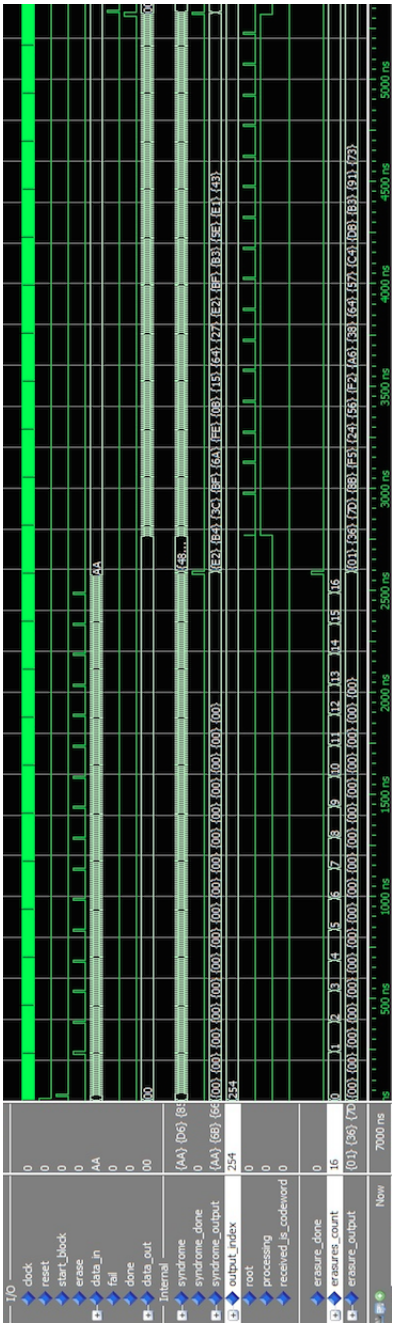

Figura 5.10: Exemplo da decodificação com 1 erros e 16 apagamentos na palavra recebida

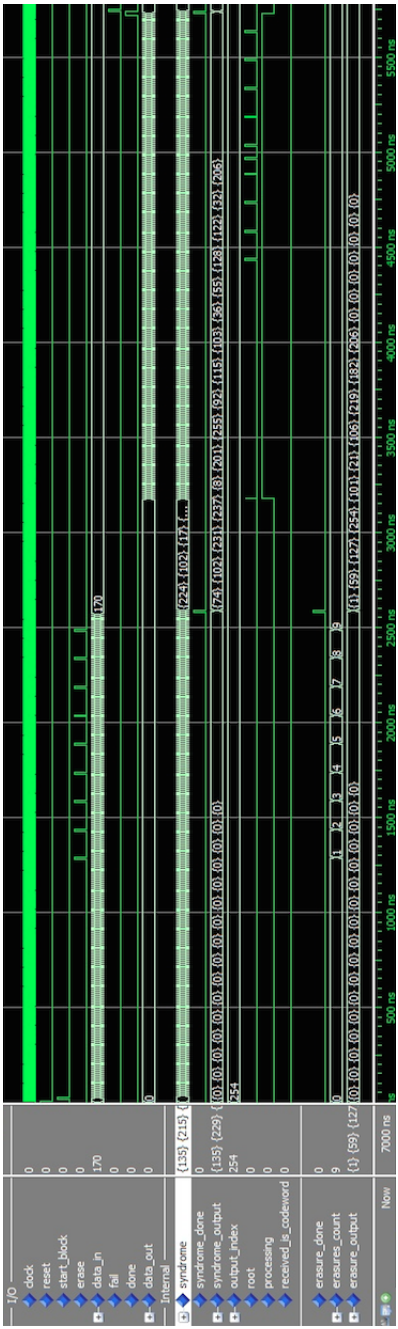

Figura 5.11: Exemplo da decodificação com 4 erros e 9 apagamentos na palavra recebida

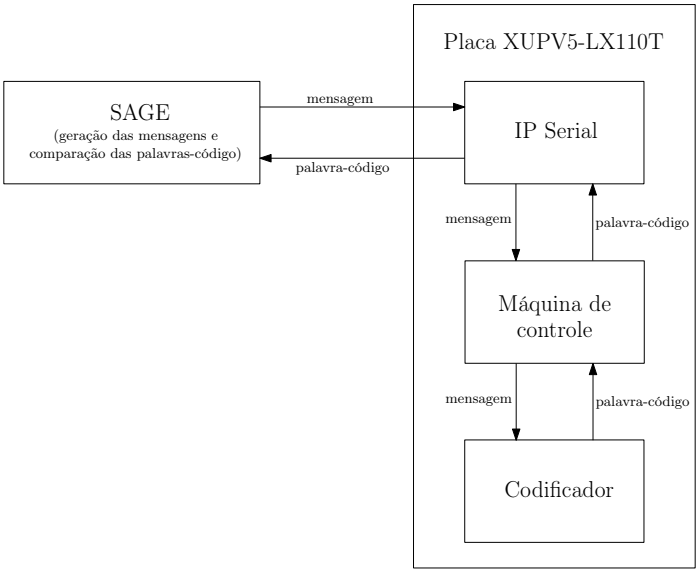

Figura 5.12: Fluxo de comunicação da rotina de validação da arquitetura desenvolvida para o codificador

o recurso de *autobaud* para realizar, automaticamente, a configuração da velocidade da comunicação. Juntamente com as arquiteturas sendo validadas, uma máquina de estados foi desenvolvida para controlar o recebimento e o envio dos vetores de teste.

O fluxo estabelecido para a validação, utilizando a plataforma SAGE e a placa de desenvolvimento XUPV5-LX110T, está ilustrado nas Figuras 5.12 e 5.13.

Para a validação da rotina, realiza-se, inicialmente, a configuração do módulo serial do computador utilizado e envia-se o primeiro comando (0x55) para que aconteça a autoconfiguração do módulo de comunicação do FPGA. A seguir, inicia-se o laço de teste, onde são geradas as mensagens ou as palavras com erros, para então realizar o processo de codificação ou decodificação. Assim que a palavra-código ou a palavra estimada for obtida, acontece o envio da mensagem ou da palavra com erros para o FPGA realizar a mesma operação. Quando o FPGA finalizar a operação, o resultado obtido é enviado para o computador e os resultados são comparados a fim de garantir o correto

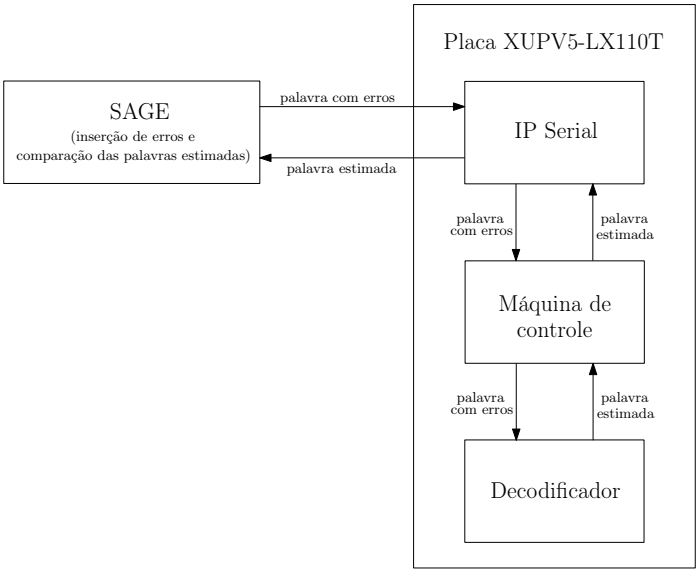

Figura 5.13: Fluxo de comunicação da rotina de validação das arquiteturas desenvolvidas para o decodificador

funcionamento das arquiteturas implementadas.

5.3 Avaliação

A etapa final do desenvolvimento da arquitetura dos códigos RS é a avaliação dos resultados obtidos frente às arquiteturas comerciais estudadas. Os dados obtidos a partir da síntese física, assim como a comparação desses com os dados fornecidos pelos fabricantes estudados são apresentados nesta seção.

5.3.1 Resultados da síntese

Após a síntese física de cada uma das arquiteturas implementadas, obteve-se os dados de ocupação de área e frequência máxima de operação. A Tabela 5.2 apresenta a quantidade de elementos utilizados pelo codificador desenvolvido para o código Reed-Solomon (255, 239).

Pela análise da tabela, nota-se a ocupação de 0,22% dos registradores e 0,33% dos LUTs disponíveis no FPGA. Já as Tabelas 5.3 e 5.4

Módulo	Registradores [#]	LUTs [#]
Codificador	151	227

Tabela 5.2: Ocupação do Codificador

Módulo	Registradores [#]	LUTs [#]
Síndrome	137	169
RiBM	412	2201
Busca de Chien + Algoritmo de Forney	154	470
Decodificador	1127	2985

Tabela 5.3: Ocupação de cada submódulo do Decodificador utilizando o algoritmo RiBM

demonstram a ocupação total dos decodificadores implementados assim como a ocupação de cada um de seus submódulos.

Pela análise da Tabela 5.3, nota-se que o grau de ocupação do decodificador implementado com o algoritmo RiBM é de 1,63% dos registradores e 4,32% das LUTs.

O decodificador implementado com o algoritmo E-DCME (Tabela 5.4) ocupa 1,61% dos registradores e 4,38% das LUTs.

Comparando as Tabelas 5.3 e 5.4, percebem-se diferenças entre os módulos de determinação dos polinômios localizador e avaliador de erros (RiBM e E-DCME), assim como nos módulos de correção de erros (Busca de Chien + Algoritmo de Forney).

Módulo	Registradores [#]	LUTs [#]
Síndrome	137	172
E-DCME	403	2307
Busca de Chien + Algoritmo de Forney	146	403
Decodificador	1110	3027

Tabela 5.4: Ocupação de cada submódulo do Decodificador utilizando o algoritmo E-DCME

Arquitetura	Frequência de operação [MHz]
Codificador	294,38
Decodificador com RiBM	268,03
Decodificador com E-DCME	270,86

Tabela 5.5: Frequência de operação estimada para cada arquitetura

Conforme descrito no Capítulo 4, as células básicas que compõem os módulos dos algoritmos RiBM e E-DCME possuem aproximadamente a mesma quantidade de elementos, porém como o algoritmo E-DCME necessita cobrir algumas situações específicas na sincronização dos polinômios [5], sua ocupação fica maior do que a do algoritmo RiBM.

Já de acordo com a Equação (2.55), a implementação do submódulo **Busca de Chien + Algoritmo de Forney**, o qual realiza a correção dos erros, no caso do algoritmo RiBM, consome uma multiplicação a mais do que a versão implementada para o algoritmo E-DCME, o que acaba aumentando sua área ocupada.

A Tabela 5.5 apresenta a frequência máxima de operação estimada pela ferramenta de síntese física utilizada.

5.3.2 Comparação com arquiteturas comerciais

Analisando a documentação disponibilizada pelos fabricantes estudados, é possível obter dados compatíveis com a arquitetura implementada como estudo de caso para este trabalho, para então compará-la com as arquiteturas comerciais selecionadas.

A documentação disponibilizada pelos fabricantes Altera [1], Xilinx [36, 35, 39, 38], Lattice [21, 20] e System LSI [34] apresentam dados referentes a mesma definição utilizada neste trabalho. Além disso, é possível verificar que o bloco decodificador disponibilizado pela Altera utiliza o algoritmo Berlekamp-Massey na determinação dos polinômios, e pela análise do código da System LSI percebe-se que a mesma trabalha com o algoritmo Euclideano. Os demais fabricantes não revelam qual é o algoritmo utilizado neste submódulo.

As Tabelas 5.6, 5.7 e 5.8 apresentam os dados de área e frequência para FPGAs da Altera, da Xilinx e da Lattice, respectivamente. A Tabela 5.9 apresenta os dados de área e frequência para a arquitetura

FPGA	Bloco	LUTs [#]	Regs [#]	f [MHz]
Arria GX II	Codificador	167	156	463
Arria GX II	Decodificador	1537	1748	278
Cyclone III	Codificador	202	156	353
Cyclone III	Decodificador	2292	934	183
Stratix III	Codificador	166	156	521
Stratix III	Decodificador	1519	937	365

Tabela 5.6: Dados de área e frequência para FPGAs da Altera

FPGA	Bloco	LUTs [#]	Regs [#]	f [MHz]
Virtex-6	Codificador	192	186	441
Virtex-6	Decodificador	754	781	287
Virtex-7	Codificador	225	197	388
Virtex-7	Decodificador	765	811	294

Tabela 5.7: Dados de área e frequência para FPGAs da Xilinx

da System LSI após a síntese utilizando o mesmo FPGA deste trabalho (XUPV5-LX110T).

A informação disponibilizada pela Factorial [10] possui a definição de um código para o padrão CCSDS, onde seu polinômio primitivo e a quantidade de símbolos de paridade diferem dos estabelecidos no estudo de caso deste trabalho. O polinômio primitivo utilizado é $p(x) = x^8 + x^7 + x^2 + x + 1$, e são utilizados 32 símbolos de paridade. A Tabela 5.10 apresenta os dados de área e frequência para esta arquitetura.

Embora apenas a solução disponibilizada pela System LSI apresenta dados para o mesmo FPGA que foi utilizado neste trabalho, pode-se perceber que, para as arquiteturas disponibilizadas pelos fabricantes de FPGA (Altera, Xilinx e Lattice), a área e a frequência conseguem ser consideravelmente mais otimizados do que a solução desenvolvida neste trabalho. Porém, uma vantagem deste trabalho sobre essas arquiteturas é a sua portabilidade, ou seja, como a arquitetura desenvolvida é independente de tecnologia, ela pode ser utilizada independentemente do FPGA utilizado, e mesmo que seja necessário alterar o FPGA alvo em um projeto, a arquitetura continua a mesma.

FPGA	Bloco	LUTs [#]	Regs [#]	f [MHz]
EC e ECP/XP	Codificador	260	207	179/185
EC e ECP/XP	Decodificador	1171	795	123/110
ECP2	Codificador	262	201	320
ECP2	Decodificador	1117	791	175
ECP2S/ECP2M	Codificador	262	201	257
ECP2S/ECP2M	Decodificador	1117	791	169
ECP2MS/XP2	Codificador	262	201	291
ECP2MS/XP2	Decodificador	1117	791	140
ECP3, SC e SCM	Codificador	251	201	400
ECP3	Decodificador	1062	791	148
SC e SCM	Decodificador	1113	803	267

Tabela 5.8: Dados de área e frequência para FPGAs da Lattice

FPGA	Bloco	LUTs [#]	Regs [#]	f [MHz]
Virtex-5 LX110T	Codificador	160	221	254,87
Virtex-5 LX110T	Decodificador	1218	2806	237,87

Tabela 5.9: Dados de área e frequência para a arquitetura da System LSI

FPGA	Bloco	LUTs [#]	Regs [#]	f [MHz]
Xilinx XC5VLX	Decodificador	4208	212	135

Tabela 5.10: Dados de área e frequência para a arquitetura da Factorial

Diferentemente dos fabricantes de FPGA, as arquiteturas disponibilizadas por empresas desenvolvedoras apenas do IP (System LSI e Factorial) possuem dados inferiores dos obtidos nesta pesquisa, o que torna este trabalho mais atrativo do que aqueles.

5.4 Ferramenta de geração automática da arquitetura

Após a finalização do processo de validação das arquiteturas, a descrição de *hardware* gerada foi analisada com o intuito de identificar quais pontos deveriam ser reescritos caso uma nova configuração de código RS fosse especificada. Como resultado, os arquivos que necessitam alterações são: "ReedSolomon.vhd"[1], "field_element_multiplier.vhd"[2], "inversion_table.vhd"[2], "RS_decoder.vhd", "Erasure.vhd", "RiBM.vhd", "E_DCME.vhd", "CF_RiBM.vhd" e "CF_EDCME.vhd".

Com os pontos que necessitam alteração identificados, criou-se um *script* de configuração utilizando a linguagem de programação Python e a plataforma SAGE. Como o SAGE disponibiliza toda a parte de cálculos polinomiais e de corpos finitos, a tarefa de gerar os arquivos necessários foi simplificada.

Por utilizar a plataforma SAGE para os cálculos em corpos finitos, ela deve estar instalada na máquina para ser possível utilizar esta ferramenta.

A invocação da ferramenta deve ser realizada em um terminal de linha de comando, utilizando a seguinte sintaxe:

sage gerador.sage n k t m $p(x)$ **apagamento**

Os parâmetros de entrada da ferramenta são definidos na Tabela 5.11.

Os valores possíveis para os parâmetros m, n e t são apresentados na Tabela 5.12. Estes valores são resultado da automatização possibilitada pela ferramenta de geração da arquitetura e pelo suporte oferecido pela plataforma SAGE. A partir da escolha dos valores de n e t, tem-se o

[1] Como as constantes de configuração são declaradas neste arquivo, sua estrutura permanece igual, porém os valores são alterados de acordo com os parâmetros de entrada.

[2] Por dependerem do polinômio primitivo que define o corpo finito, estes arquivos precisam ser completamente reescritos.

Parâmetro	Função
gerador.sage	a ferramenta de geração automática
n	tamanho da palavra-código
k	tamanho da mensagem
t	capacidade de correção
m	tamanho dos símbolos
$p(x)$	polinômio primitivo do corpo finito
apagamento	adicionar correção de apagamentos

Tabela 5.11: Parâmetros de entrada da ferramenta de geração automática da arquitetura

Sinal	Resultado
Tamanho do símbolo (m)	$3 - 12$ *bits*
Tamanho do bloco (n)	$7 - 4095$ símbolos
Símbolos de paridade ($2t$)	$4 - 256$ símbolos

Tabela 5.12: Possíveis configurações dos parâmetros do código RS

valor de k.

Os valores disponíveis para o parâmetro **apagamento** são "0" (sem correção de apagamentos) e "1" (adicionar suporte para correção de apagamentos).

Já os polinômios primitivos possíveis de serem utilizados, são os mesmos que podem ser utilizados no SAGE para configuração de um corpo finito. Porém, como argumento da ferramenta, deve-se utilizar a representação decimal do polinômio primitivo. Alguns exemplos de polinômios primitivos que podem ser utilizados são apresentados na Tabela 5.13.

Os arquivos fonte necessários para a utilização desta ferramenta de geração automática da arquitetura está disponibilizada no seguinte endereço:

`https://github.com/TomasGrimm/RS_Codes`

Diferentemente desta ferramenta, as fabricantes de FPGA (Altera, Xilinx e Lattice) oferecem as suas em conjunto com as respectivas suítes de desenvolvimento. Tanto essas ferramentas quanto a da empresa

m	$p(x)$	Representação decimal
3	$x^3 + x + 1$	11
4	$x^4 + x + 1$	19
5	$x^5 + x^2 + 1$	37
6	$x^6 + x + 1$	67
7	$x^7 + x^3 + 1$	137
8	$x^8 + x^4 + x^3 + x^2 + 1$	285
9	$x^9 + x^4 + 1$	529
10	$x^{10} + x^3 + 1$	1033
11	$x^{11} + x^2 + 1$	2053
12	$x^{12} + x^6 + x^4 + x + 1$	4179

Tabela 5.13: Exemplos de polinômios primitivos para corpos finitos binários

Factorial oferecem uma interface gráfica, o que torna o processo de configuração do código mais interativo. Já a ferramenta disponibilizada pela empresa System LSI deve ser invocada pela linha de comando, assim como esta ferramenta.

CAPÍTULO 6

Conclusão

Esse trabalho apresentou o estudo e o desenvolvimento de uma descrição em *hardware* dos códigos Reed-Solomon para ser aplicada a plataformas reconfiguráveis. O foco deste trabalho foi o desenvolvimento de uma arquitetura para o codificador baseada em um esquema de LFSR e duas arquiteturas para o decodificador, cada uma utilizando um dos algoritmos estudados para a determinação dos polinômios localizador e avaliador de erros. Em ambos os casos, tanto para o codificador quanto para o decodificador, objetivou-se disponibilizar uma arquitetura que possa ser integrada a outros projetos onde se faça necessário o uso de códigos de correção de erros.

Primeiramente, realizou-se o estudo teórico necessário para a fundamentação deste trabalho ao mesmo tempo em que a especificação formal necessária para a implementação foi definida. Em seguida algumas arquiteturas comerciais foram estudadas com o intuito de sumarizar os pontos em comum e definir quais deveriam constar em uma arquitetura básica.

O próximo passo do desenvolvimento foi a implementação da arquitetura, que, a partir de um padrão industrial, aproveitou-se das técnicas do estado da arte para os algoritmos selecionados, buscando

assim estruturas mais otimizadas.

Como terceiro passo do processo de desenvolvimento, a validação, tanto por simulação quanto por prototipação, foi realizada com o intuito de garantir o funcionamento e a aplicabilidade da arquitetura desenvolvida, além de possibilitar o colhimento dos dados necessários para a última etapa do desenvolvimento: a avaliação. Neste passo, a partir destes dados, foi possível ter uma melhor ideia sobre as técnicas utilizadas, além de tornar possível a comparação da arquitetura implementada com aquelas estudadas.

Como resultado deste trabalho, tem-se uma ferramenta capaz de gerar automaticamente uma configuração dos códigos Reed-Solomon capaz de atender diversas aplicações e projetos, uma vez que, após a conclusão das arquiteturas alvo, foram identificados os pontos que necessitavam retrabalho a fim de gerar uma arquitetura com configuração diferente. Assim, através da configuração automatizada de alguns trechos de código nos arquivos-fonte, é possível gerar diversas configurações de códigos RS tendo como base os mesmos algoritmos implementados neste trabalho.

A partir da análise dos resultados obtidos após a implementação das arquiteturas, foi possível ter uma melhor noção dos algoritmos estudados, uma vez que os resultados apresentados pelas fontes [3, 29] não levavam em conta as estruturas de controle, mas apenas as estruturas desenvolvidas para realizar os cálculos necessários para o processo de decodificação.

Em relação aos objetivos estabelecidos para este trabalho, nota-se que todos foram atendidos:

- a teoria necessária para o entendimento do funcionamento dos códigos RS foi estudada e as técnicas de implementação foram exploradas;
- a arquitetura foi modularizada de modo que, com pouco retrabalho, é possível gerar novas configurações com os mesmos algoritmos implementados;
- a interface criada é compatível com as interfaces apresentadas pelos IPs gerados utilizando as ferramentas comerciais estudadas;
- uma ferramenta de geração automática foi implementada utilizando a plataforma SAGE, a qual é *open source* e baseada na

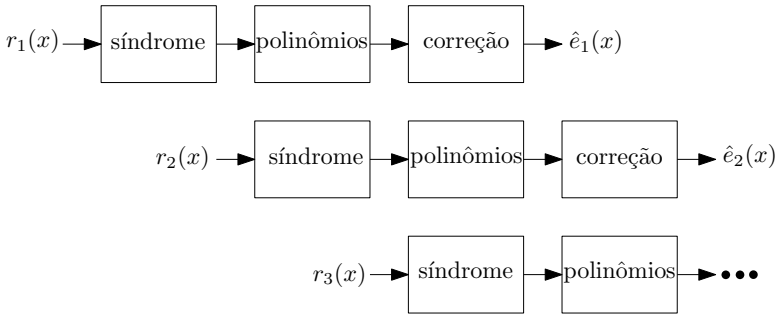

Figura 6.1: Conceito do funcionamento do decodificador com *pipeline*

linguagem Python.

Em se tratando dos pontos escolhidos para a implementação de códigos Reed-Solomon, recapitulando os itens salientados nas Tabelas 3.1, 3.2 e 3.3, as funcionalidades básicas e o interfaceamento necessário de um conjunto codificador/decodificador para códigos Reed-Solomon estão presentes nas arquiteturas aqui apresentadas. Pode-se também salientar que os pontos escolhidos para implementação são os mesmos oferecidos pela arquitetura de [34], com a vantagem de que esta implementação utiliza uma plataforma *open source* que é independente de sistema operacional, aumentando a portabilidade da ferramenta aqui desenvolvida.

Sugere-se como trabalho futuro, a adição de mais um submódulo para o cálculo da síndrome dedicado à verificação da palavra estimada, uma vez que a arquitetura implementada neste trabalho contém apenas um módulo e a mesma é reutilizada quando acontece a etapa de correção dos símbolos da palavra recebida. Tal adição facilitaria a introdução do conceito de *pipeline* ao decodificador, de tal forma que seu funcionamento seguisse o esquema demonstrado na Figura 6.1.

Tendo em vista a ampliação das funcionalidades disponibilizadas neste trabalho, pode-se sugerir como trabalhos futuros, a implementação das funcionalidades apresentadas nas ferramentas comerciais estudadas, mas que, devido ao escopo deste trabalho, foram excluídas destas arquiteturas. As funcionalidades que trariam mais benefícios à este trabalho são:

- Encurtamento;

- Puncionamento;
- Diferente tamanho de bloco por iteração;
- Diferente capacidade de correção por iteração.

Como consequência dos dois últimos itens, dois novos sinais de entrada devem ser criados para indicar tamanho de bloco e quantidade de símbolos de paridade a serem utilizados.

Também como trabalho derivado deste, pode-se melhorar a ferramenta de geração automática da arquitetura, tornando-a mais interativa e adicionando configurações prontas de acordo com padrões utilizados na indústria.

Como aprimoramento das estratégias para a validação das arquiteturas, pode-se buscar alternativas para o modo como é realizada a comunicação entre o computador e a plataforma de testes. A plataforma utilizada neste trabalho, em particular, possui as interfaces USB, Ethernet e PCI Express, as quais trabalham com velocidades muito superiores às da interface serial.

APÊNDICE A

Álgebra Abstrata

Conforme [25], códigos lineares, e por consequência códigos cíclicos, formam grupos e espaços vetoriais. Porém, para ser possível compreender a sua construção, tais estruturas devem, primeiramente, ser formalizadas, assim como as demais estruturas que são utilizadas como base para os códigos não-binários.

A.1 Grupos e Subgrupos

Definição. Uma operação binária $*$ aplicada sobre um par de elementos de um conjunto atribui um terceiro elemento, também pertencente ao conjunto, a este par. Esta é a definição da operação binária fechada.

Definição. Um grupo $(G, *)$ é um conjunto definido sob a operação *ast* tal que

- A operação binária $*$ é associativa.

$$a * (b * c) = (a * b) * c$$

- G contém um elemento e tal que, para qualquer a em G,

$$a * e = e * a = a$$

onde e é chamado elemento identidade, o qual é único para o grupo

- Para qualquer elemento a em G existe um outro elemento a' em G tal que

$$a * a' = a' * a = e$$

onde o elemento a' é chamado de inverso de a (a também é o inverso de a'). Cada elemento possui apenas um inverso.

Definição. Um grupo de ordem finita é chamado de grupo finito. O número de elementos em um grupo é chamado de ordem do grupo. Para qualquer inteiro positivo m é possível definir um grupo finito de ordem m sob uma operação binária.

Definição. Um grupo G é dito comutativo se

$$a * b = b * a, \ \forall a, b \in G$$

Definição. Um subconjunto não nulo H de G será também um subgrupo se, o mesmo for definido sob a operação de G e satisfizer todas as condições de grupo.

A.2 Corpos

Definição. Seja F um conjunto de elementos onde duas operações, adição ("+") e multiplicação ("·"), estão definidas, então, o conjunto F, juntamente com as operações "+" e "·", é um corpo se as seguintes condições forem satisfeitas:

- F é comutativo na adição.
  - O elemento identidade com respeito à adição é chamado elemento zero ou identidade aditiva de F e é denotado por 0;

  - O elemento aditivo inverso de a é $-a$.

- O conjunto de elementos não nulos em F é um grupo comutativo para a multiplicação.

  - O elemento identidade da multiplicação é chamado elemento unitário ou identidade multiplicativa de F e é denotado por 1;
  - O elemento multiplicativo inverso é a^{-1}, desde que $a \neq 0$.

- A multiplicação é distributiva sobre a adição. Ou seja, para quaisquer três elementos a, b e c em F, $a \cdot (b + c) = a \cdot b + a \cdot c$.

DEFINIÇÃO. O número de elementos em um corpo define a sua ordem, e um corpo com número finito de elementos é chamado de corpo finito.

A.3 Corpos de Extensão

A.3.1 Corpos finitos

Seja p um número primo.

DEFINIÇÃO. A operação de adição módulo-p é definida como sendo o resto da divisão realizada após a soma de dois elementos pertencentes a um grupo G de ordem p, ou seja,

$$(a + b) \bmod p, \forall a, b \in G$$

EXEMPLO. Adição módulo-2 em $\mathbb{F}_2$

+	0	1
0	0	1
1	1	0

□

DEFINIÇÃO. A operação de multiplicação módulo-p é definida como sendo o resto da divisão após a multiplicação de dois elementos pertencentes a um grupo G de ordem p, ou seja,

$$(a \cdot b) \bmod p, \forall a, b \in G$$

EXEMPLO. Multiplicação módulo-2 em $\mathbb{F}_2$

$\cdot$	0	1
0	0	0
1	0	1

□

Se um conjunto de ordem p for um corpo sob as operações de adição módulo-p e multiplicação módulo-p, então este será um corpo primo e será denotado por $\mathbb{F}_p$.

Para qualquer inteiro positivo m é possível criar um corpo estendido ($\mathbb{F}_{p^m}$), sendo comumente chamado de Corpos de Extensão.

Como o número de elementos em um corpo finito é limitado, deve existir um momento em que as potências de a começam a repetir. Ou seja, devem existir dois inteiros m e k, onde $m > k$ e $a^k = a^m$.

Seja a^{-1} o inverso multiplicativo de a, então $(a^{-1})^k = a^{-k}$ é o inverso multiplicativo de a^k. Multiplicando os dois lados de $a^k = a^m$ por a^{-k} obtém-se

$$1 = a^{m-k}$$

Esta igualdade implica na existência de um inteiro mínimo positivo n tal que $a^n = 1$. Este inteiro é chamado de ordem do elemento a.

Em um corpo finito, um elemento não nulo a é dito primitivo se a sua ordem for $q - 1$, e portanto as potências do termo primitivo geram os elementos do corpo finito.

A.3.2 Elementos

Para estes corpos, além dos elementos "0" e "1", é introduzido um novo símbolo, α. E também é definida a multiplicação, "$\cdot$" para introduzir a

sequência das potências de α:

$$
\begin{aligned}
&0 \cdot 0 = 0 \\
&0 \cdot 1 = 1 \cdot 0 = 0 \\
&1 \cdot 1 = 1 \\
&0 \cdot \alpha = \alpha \cdot 0 = 0 \\
&1 \cdot \alpha = \alpha \cdot 1 = \alpha \\
&\alpha^2 = \alpha \cdot \alpha \\
&\alpha^3 = \alpha \cdot \alpha \cdot \alpha \\
&\vdots \\
&\alpha^j = \alpha \cdot \alpha \cdot \cdots \cdot \alpha \ (j \text{ vezes}) \\
&\vdots
\end{aligned}
$$

Da definição anterior é possível definir:

$$
\begin{aligned}
&0 \cdot \alpha^j = \alpha^j \cdot 0 = 0 \\
&1 \cdot \alpha^j = \alpha^j \cdot 1 = \alpha^j \\
&\alpha^i \cdot \alpha^j = \alpha^j \cdot \alpha^i = \alpha^{i+j}
\end{aligned}
$$

E assim é possível obter o seguinte conjunto definido sob a multiplicação:

$$F = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^j, \ldots\right\}$$

A.3.3 Polinômios

Um polinômio $f(x)$ com uma variável x e coeficientes de $\mathbb{F}_2$ se dá pela forma:

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_n x^n$$

onde $f_i = 0$ ou 1 para $0 \leq i < n$. O grau de um polinômio é a maior potência de x com um expoente não nulo.

Em geral, existem 2^n polinômios em $\mathbb{F}_2$ com grau n.

Para um polinômio $f(x)$ em $\mathbb{F}_2$, se o mesmo consiste de um número par de termos, ele é divisível por $X + 1$.

Um polinômio $p(x)$ em $\mathbb{F}_2$ de grau m é dito irredutível em $\mathbb{F}_2$ se

$p(x)$ não for divisível por nenhum polinômio em $\mathbb{F}_2$ de grau menor que m e maior que zero.

Um polinômio irredutível $p(x)$ de grau m é dito primitivo se o menor inteiro positivo n pelo qual $p(x)$ divide $x^n + 1$ é $n = 2^m - 1$.

A.3.4 Condições sobre m

A seguir, deve-se colocar uma condição sobre α para que o conjunto F contenha somente 2^m elementos e seja definido para a multiplicação ("·").

Seja $p(x)$ um polinômio primitivo de grau m em $\mathbb{F}_2$. Assume-se que $p(\alpha) = 0$, ou seja, α é uma raiz de $p(x)$. Uma vez que $p(x)$ divide $x^{2^m-1} + 1$, tem-se

$$x^{2^m-1} + 1 = q(x)p(x)$$

Substituindo x por α, e sabendo que $p(\alpha) = 0$:

$$\alpha^{2^m-1} + 1 = q(\alpha) \cdot 0$$

Finalmente, somando 1 (usando adição modulo-2) em ambos os lados, tem-se:

$$\alpha^{2^m-1} = 1$$

Portanto, sob a condição de $p(\alpha) = 0$, o conjunto F é composto pelos elementos

$$F = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\right\}$$

A.3.5 Representação

Existem três modos de representar um elemento em um Corpo de Extensão:

- Potência
- Polinômio
- m-upla ou vetor

EXEMPLO. Representação dos elementos de $\mathbb{F}_{2^3}$ com polinômio primitivo $p(x) = x^3 + x + 1$

Potência	Polinômio	3-upla $(b_0\ b_1\ b_2)$
0	0	(0 0 0)
1	1	(1 0 0)
α	α	(0 1 0)
α^2	α^2	(0 0 1)
α^3	$1 + \alpha$	(1 1 0)
α^4	$\alpha + \alpha^2$	(0 1 1)
α^5	$1 + \alpha + \alpha^2$	(1 1 1)
α^6	$1 + \alpha^2$	(1 0 1)

□

Para operações de multiplicação, a representação em potência facilita o cálculo:

$$\alpha^i \cdot \alpha^j = \alpha^{i+j}$$

Já para operações de adição, as representações em polinômio e m-upla são mais adequadas.

EXEMPLO.

$$(1 + \alpha + \alpha^3) + (\alpha + \alpha^2) = 1 + \alpha^2 + \alpha^3$$
$$(1\ 1\ 0\ 1) + (0\ 1\ 1\ 0) \Rightarrow \Big((1+0)\ (1+1)\ (0+1)\ (1+0)\Big) = (1011)$$

□

A.4 Espaços Vetoriais

Seja V um conjunto de elementos em que a operação binária adição "+" está definida, seja F um corpo finito e uma operação de multiplicação, "$\cdot$", entre os elementos de F e V, também é definida.

DEFINIÇÃO. Assim, o conjunto V é chamado de espaço vetorial sobre F se as seguintes condições forem satisfeitas:

(*i*) V é comutativo para a adição

(*ii*) Para qualquer elemento a em F e qualquer elemento v em V, $a \cdot v$ é um elemento em V

(*iii*) Para quaisquer elementos u e v em V e a e b em F:

$$a \cdot (u + v) = a \cdot u + a \cdot v$$
$$(a + b) \cdot v = a \cdot v + b \cdot v$$

(*iv*) Para qualquer v em V e quaisquer a e b em F:

$$(a \cdot b) \cdot v = a \cdot (b \cdot v)$$

(*v*) Seja 1 o elemento unitário de F, então, para qualquer v em V, $1 \cdot v = v$

Definição. Os elementos de V são chamados vetores e os elementos do campo F são chamados escalares.

Seja S um subconjunto não nulo de um espaço vetorial V sobre um corpo F, então, S é um subespaço de V se as seguintes condições forem satisfeitas:

(*i*) Para quaisquer dois vetores u e v em S, $u+v$ também é um vetor em S

(*ii*) Para qualquer elemento a em F e qualquer vetor v em S, $a \cdot v$ também está em S.

Sejam $v_1, v_2, \ldots, v_k$ k vetores em um espaço vetorial V em um corpo F. O conjunto de todas as combinações lineares de $v_1, v_2, \ldots, v_k$ formam um subespaço de V.

Seja S um subespaço k-dimensional do espaço vetorial V_n de todas as n-uplas de $\mathbb{F}_2$. A dimensão do espaço nulo S_d é $n - k$.

A adição em V é chamada adição vetorial e a multiplicação que combina um escalar de F e um vetor de V em um vetor em V é chamada de multiplicação escalar ou produto escalar.

A.4.1 Adição vetorial

Para quaisquer n-uplas $u = (u_0, u_1, \ldots, u_{i-1})$ e $v = (v_0, v_1, \ldots, v_{i-1})$ em V_n:

$$u + v = (u_0 + v_0, u_1 + v_1, \ldots, u_{i-1} + v_{i-1})$$

O resultado $u + v$ também será uma n-upla.

A n-upla que consiste somente de zeros é a identidade aditiva:

$$0 = (0, 0, \ldots, 0)$$

E para qualquer v em V_n:

$$v + v = (v_0 + v_0, v_1 + v_1, \ldots, v_{i+1} + v_{i+1})$$

Ou seja, a inversa aditiva de cada elemento em V_n é o próprio elemento.

A.4.2 Multiplicação escalar

A multiplicação de um vetor v de V_n por um escalar a é dada por:

$$a \cdot v = (a \cdot v_0, a \cdot v_1, \ldots, a \cdot v_{i-1})$$

E, se $a = 1$, então:

$$a \cdot v = 1 \cdot v = (1 \cdot v_0, 1 \cdot v_1, \ldots, 1 \cdot v_{i-1}) = (v_0, v_1, \ldots, v_{i-1})$$

A.4.3 Produto interno

Sejam $u = (u_0, u_1, \ldots, u_{i-1})$ e $v = (v_0, v_1, \ldots, v_{i-1})$ duas n-uplas em V_n, então o produto interno é definido por:

$$u \cdot v = u_0 \cdot v_0 + u_1 \cdot v_1 + \cdots + u_{n-1} \cdot v_{n-1}$$

onde o resultado da operação será um escalar.

Caso $u \cdot v = 0$, então u e v são ditos ortogonais.

O produto interno tem as seguintes propriedades:

- Comutatividade

$$u \cdot v = v \cdot u$$

- Associatividade Vetorial

$$u \cdot (v + w) = u \cdot v + u \cdot w$$

- Associatividade Escalar

$$(au) \cdot v = a(u \cdot v)$$

Referências bibliográficas

[1] *Reed-Solomon II MegaCore Function*, UG-01090-3.0, Altera Corporation, 2012. [Online]. Available: http://www.altera.com/literature/ug/ug_rsii.pdf

[2] *Network and Customer Installation Interfaces – Asymmetric Digital Subscriber Line (ADSL) Metallic Interface*, American National Standard For Telecommunications Standards Project for Interfaces Relating to Carrier to Customer Connection of Asymmetrical Digital Subscriber Line (ADSL) Equipment T1E1.4/98-007R3, 1998.

[3] J. Baek and M. H. Sunwoo, "Simplified Degree Computationless Modified Euclid's Algorithm and its Architecture," in *2007 IEEE International Symposium on Circuits and Systems*. IEEE, May 2007, pp. 905–908. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4252782

[4] ——, "Enhanced degree computationless modified Euclid's algorithm for Reed-Solomon decoder," *2006 IEEE International Symposium on Circuits and Systems*, p. 4, 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1693394

[5] J. Baek and M. Sunwoo, "New degree computationless modified euclid algorithm and architecture for Reed-Solomon

decoder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 915–920, Aug. 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1664912

[6] E. Berlekamp, "Nonbinary BCH decoding," *IEEE Transactions on Information Theory*, vol. 14, no. 2, pp. 242–242, Mar. 1968. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1054109

[7] R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Polynomial GCD Computation," *IEEE Transactions on Computers*, vol. C-33, no. 8, pp. 731–736, Aug. 1984. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5009358

[8] *Telemetry Channel Coding*, ser. Blue Book, No. 6, Consultative Committee for Space Data Systems (CCSDS) Recommendation for Space Data System Standards 101.0-B-6, October 2002. [Online]. Available: http://public.ccsds.org/publications/archive/101x0b6s.pdf

[9] *Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television DVB-H PHY addition*, European Broadcasting Union Update of performance figures for DVB-T. ETSI EN 300 744 V1.6.1 (2009-01), 2009.

[10] *Reed-Solomon decoder IP Core*, Product Datasheet, Factorial Consulting Ltd., 2008. [Online]. Available: http://factorial.hu/system/files/datasheet.pdf

[11] *RS-232 compatible serial interface with autobaud*, Open Source, Grupo de Apoio ao Projeto de Hardware (GAPH), PUCRS, 2003. [Online]. Available: https://corfu.pucrs.br/gaph_public/Projects/RS-232%20Compatible%20Serial%20Interface%20with%20Autobaud/RS-232%20Compatible%20Serial%20Interface%20with%20Autobaud.html

[12] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160,

Apr. 1950. [Online]. Available: http://doi.wiley.com/10.1002/j.1538-7305.1950.tb00463.x

[13] *IEEE Standard for Air Interface for Broadband Wireless Access Systems*, IEEE Computer Society and the IEEE Microwave Theory and Techniques Society Revision of IEEE Std 802.16-2009 IEEE Std 802.16-2012, 2012.

[14] *Asymmetric digital subscriber line transceivers 2 (ADSL2)*, International Telecommunication Union - Telecommunication Standardization Sector of ITU Recommendation ITU-T G.992.3 ITU-T G.992.3, 2009.

[15] *Interfaces for the optical transport network*, International Telecommunication Union - Telecommunication Standardization Sector of ITU Recommendation ITU-T G.709/Y.1331 ITU-T G.709/Y.1331, 2012.

[16] C. Jezierski and J. Modelski, "FPGA transceiver for space environment," in *IEEE Africon '11*. IEEE, Sep. 2011, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6072187

[17] Y. Jiang, *A Practical Guide to Error-Control Coding Using MATLAB*. 685 Canton Street Norwood, MA 02062: Artech House, 2010.

[18] C. Kim, S. Rhee, J. Kim, and Y. Jee, "Product Reed-Solomon Codes for Implementing NAND Flash Controller on FPGA Chip," *2010 Second International Conference on Computer Engineering and Applications*, pp. 281–285, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5445821

[19] T. Kohonen, *Content-Addressable Memories*, 2nd ed., ser. Springer Series in Information Sciences. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 1. [Online]. Available: http://link.springer.com/10.1007/978-3-642-83056-3

[20] *Dynamic Block Reed-Solomon Decoder User's Guide*, IPUG52_01.6, Lattice Semiconductor Corporation, 2010. [Online].

Available: http://www.latticesemi.com/~/media/Documents/UserManuals/DynamicBlockReedSolomonDecoderUsersGuide.ashx

[21] *Dynamic Block Reed-Solomon Encoder User's Guide*, IPUG40_03.6, Lattice Semiconductor Corporation, 2010. [Online]. Available: http://www.latticesemi.com/~/media/Documents/UserManuals/DynamicBlockReedSolomonEncoderIPCoreUsersGuide.ashx

[22] S. S. Lee and M. K. Song, "An efficient recursive cell architecture of modified euclid's algorithm for decoding reed-solomon codes," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 4, pp. 845–849, Nov. 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1196411

[23] S. Lin and D. Costello, *Error Control Coding*, 2nd ed. Prentice Hall, 2004.

[24] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, Jan. 1969. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1054260

[25] T. Moon, *Error Correction Coding Mathematical Methods and Algorithms*, 1st ed. Wiley-Interscience, 2005.

[26] D. K. Pradhan, Ed., *Fault-tolerant Computer System Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[27] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, Jun. 1960. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0108018

[28] A. Reyhani-Masoleh and M. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over GF(2m)," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 945–959, Aug. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1306989

[29] D. Sarwate and N. Shanbhag, "High-speed architectures for Reed-Solomon decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 641–655, Oct. 2001. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=953498

[30] H. M. Shao, T. K. Truong, L. J. Deutsch, J. H. Yuen, and I. S. Reed, "A VLSI design of a pipeline Reed-Solomon decoder." *IEEE transactions on computers. Institute of Electrical and Electronics Engineers*, vol. C-34, no. 5, pp. 393–403, May 1985. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/11539661

[31] W. Stein and D. Joyner, "SAGE," *ACM SIGSAM Bulletin*, vol. 39, no. 2, p. 61, Jun. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1101884.1101889

[32] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A method for solving key equation for decoding goppa codes," *Information and Control*, vol. 27, no. 1, pp. 87–99, Jan. 1975. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S001999587590090X

[33] R. M. Sweeney, C. Spagnol, and E. Popovici, "Comparative study of software vs. hardware implementations of shortened Reed-Solomon code for Wireless Body Area Networks," *2010 27th International Conference on Microelectronics Proceedings*, no. Miel, pp. 223–226, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5490471

[34] *ReedSolomon IP CoreProgram*, Open Source, System LSI CO., LTD, 2011. [Online]. Available: http://opencores.org/download,reed_solomon_codec_generator

[35] *LogiCORE IP Reed-Solomon Decoder v7.1*, DS252, Xilinx, Inc., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/rs_decoder_ds252.pdf

[36] *LogiCORE IP Reed-Solomon Encoder v7.1*, DS251, Xilinx, Inc., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/rs_encoder_ds251.pdf

[37] *ML505/ML506/ML507 Evaluation Platform User Guide*, UG347, Xilinx, Inc., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[38] *LogiCORE IP Reed-Solomon Decoder v9.0*, PG107, Xilinx, Inc., 2013. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/rs_decoder/v9_0/pg107-rs-decoder.pdf

[39] *LogiCORE IP Reed-Solomon Encoder v9.0*, PG025, Xilinx, Inc., 2013. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/rs_encoder/v9_0/pg025_rs_encoder.pdf

[40] H. Yang, Y. Zhong, and L. Yang, "An FPGA prototype of a forward error correction (FEC) decoder for ATSC digital TV," *IEEE Transactions on Consumer Electronics*, vol. 45, no. 2, pp. 387–395, May 1999. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=793424

[41] J. You and S. Wu, "Design and realization of reed-solomon codec based on FPGA technique," *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, no. x, pp. 2086–2089, Aug. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6025902

[42] X. Youzhi, "Implementation of Berlekamp-Massey algorithm without inversion," *IEE Proceedings - Part I: Communications, Speech & Vision*, vol. 138, no. 3, pp. 138–140, 1991. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=87857