Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Engenharia – Faculdade de Informática

Curso de Engenharia de Computação

Implementação de uma Arquitetura Reed-Solomon para uso em Redes OTN 10.7 Gbps

Volume Final do Trabalho de Conclusão

Autores:
Alzemiro Henrique Lucas da Silva
Taciano Ares Rodolfo

Orientador:
Prof. Dr. Fernando Gehm Moraes

Porto Alegre, Dezembro de 2007.

Índice

Índice de Figuras

Índice de Tabelas

Lista de Siglas

APS/PCC	Automatic Protection Switching / Protection Communication Channel
ATM	Asynchronous Transfer Mode
CD	Compact Disc
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DVD	Digital Video Disc
ECC	Error Correction Coding
FAS	Frame Alignment Signal
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
FTFL	Fault Type and Fault Location
GAPH	Grupo de Apoio ao Projeto de Hardware
GCC	General Communication Channel
GF	Galois Field
GFP	Generic Framing Procedure
HD	Hard Disk
ITU-T	International Telecommunications Union – Telecommunication
JC	Justification Control
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
MFAS	Multi Frame Alignment Signal
NJO	Negative Justification Opportunity
ODU	Optical channel Data Unit
OPU	Optical channel Payload Unit
OTN	Optical Transport Network
OTU	Optical channel Transport Unit
PJO	Positive Justification Opportunity
PM	Path Monitoring
PSI	Payload Structure Identifier
RS	Reed-Solomon
SDH	Synchronous Digital Hierarchy
SM	Session Monitoring
SONET	Synchronous Optical Networking
TCM	Tandem Connection Monitoring
WAN	Wide Area Network

1 Introdução

Observa-se, no cenário mundial, uma crescente demanda por largura de banda e qualidade de serviço, principalmente em redes de longo alcance (WANs). Esta demanda é evidenciada, cada vez mais, pelo uso de técnicas de transmissão de voz e vídeo em tempo real, o que proporciona economia e aumento de rendimento para diversas instituições.

A crescente necessidade de banda incentivou a criação da recomendação G.709 OTN (Optical Transport Network), pela ITU-T (International Telecommunications Union – Telecomunications), que define interfaces de rede para enlaces de fibra óptica de até 40Gbps [1]. A recomendação G.709 foi construída a partir da experiência e dos benefícios dos protocolos SONET e SDH, para proporcionar a base para as redes de fibra óptica da próxima geração [2]. Porém, alguns elementos foram adicionados para aumentar o seu desempenho e reduzir custos. O elemento mais importante e mais justificativo para a criação desta nova recomendação é a inclusão de um mecanismo de controle de erros mais robusto, o que aumenta o desempenho e possibilita enlaces mais longos de fibra óptica.

Codificação de detecção de erros é um mecanismo onde o receptor é capaz apenas de detectar a presença de erros em uma mensagem recebida. Coletivamente, a codificação de detecção de erros e a codificação de correção de erros, são chamadas de codificação de controle de erros. A codificação de controle de erros tem um papel significativo na evolução dos sistemas de telecomunicações e de armazenamento digital.

A recomendação G.709 define um frame dividido em três campos: *overhead*, *payload* e FEC. O campo *overhead* provê funções de monitoramento, gerenciamento e proteção de roteamento em canais ópticos, o campo *payload* pode transportar dados úteis provenientes de diversos protocolos, e o campo FEC possui informações redundantes codificadas para realizar a correção de erros no momento da recepção da mensagem. A codificação de canal adotada pela recomendação G.709 é o Reed-Solomon (RS), que possui excelentes propriedades de correção de erros e é usado na maioria dos sistemas que aplicam correção de erro na atualidade.

O algoritmo RS utiliza o conceito de campos finitos, também conhecidos como campos de Galois, e seus algoritmos funcionam basicamente através de cálculos sobre polinômios, uma vez que as mensagens são vistas como polinômios pertencentes a um campo finito pré-determinado. Códigos Reed-Solomon pertencem a uma classe de códigos corretores de erro, chamada de códigos cíclicos não binários [3], pois sua construção é orientada a símbolos de m bits, onde m pode ser qualquer inteiro positivo maior que 2, normalmente 8 para aplicações de comunicação de dados, onde cada símbolo representa um byte.

O algoritmo RS serve tanto para a correção de alguns bits com erro em lugares aleatórios, quanto para a correção de erros em rajada, desde que a quantidade de símbolos com erro não ultrapasse a capacidade de correção do código. Desta forma, é possível melhorar significativamente a relação sinal ruído de um canal, inclusive aumentando a distância de um enlace sem que seja necessária regeneração [4].

Códigos RS têm sido utilizados não somente em sistemas de transmissão de dados, mas também em comunicação wireless, sistemas de armazenamento de grandes volumes de dados, tais como Hard Disk (HD), Compact Disc (CD) e Digital Video Disc (DVD), e também em TV digital e sistemas espaciais.

1.1 Motivação

O domínio da tecnologia de transmissão de dados em alta velocidade utilizando fibra óptica, com tecnologia nacional, pode trazer ao Brasil a possibilidade de aumento de suas redes de comunicação e também um forte potencial para a exportação de equipamentos eletrônicos. Utilizando os conhecimentos adquiridos ao longo de nosso curso, acreditamos que seja possível desenvolver módulos de hardware que possibilitem a produção nacional de equipamentos de alta tecnologia.

Este trabalho é motivado pela parceria entre a TERACOM Telemática Ltda, empresa brasileira que desenvolve equipamentos de telecomunicações, e o Grupo de Apoio ao Projeto de Hardware (GAPH), da Faculdade de Informática desta Universidade, do qual os autores fazem parte, para o desenvolvimento de um transponder OTN 10.7Gbps, o que deve aumentar ainda mais a competitividade desta empresa em âmbito nacional e internacional.

1.2 Objetivos

Este trabalho tem como principal objetivo desenvolver uma arquitetura de codificador e decodificador Reed-Solomon, capaz de fazer detecção e correção de erro em um sistema de transmissão de dados em alta velocidade, sendo completamente compatível com a recomendação G.709, para interoperabilidade com equipamentos de diferentes fabricantes. Os módulos desenvolvidos devem ser capazes de operar em conjunção com o *framer*, trabalho que está sendo desenvolvido paralelamente, responsável por fazer o alinhamento e a montagem dos frames OTN.

Outro importante objetivo do presente trabalho é colocar em prática alguns dos conhecimentos adquiridos ao longo do curso, tais como estruturas algébricas para a construção dos códigos, redes de comunicação dados, arquiteturas de sistemas computacionais, desenvolvimento em linguagem de descrição de hardware e prototipação em FPGAs.

1.3 Contribuições

Este trabalho propõe o desenvolvimento de uma arquitetura de Reed-Solomon capaz de realizar a detecção e correção de erros de acordo com o padrão OTN. A arquitetura desenvolvida é baseada nas vantagens e desvantagens das implementações propostas em diversos artigos sobre o assunto.

As principais contribuições deste trabalho se dão no estudo das diversas arquiteturas existentes que implementam o algoritmo RS em hardware e no desenvolvimento de uma arquitetura que satisfaz os requisitos de desempenho de um sistema de transmissão de dados em 10.7 Gbps, aplicando paralelismo e utilizando uma alta freqüência de operação.

1.4 Organização do Documento

Este trabalho é dividido em 8 capítulos. Além da presente introdução, há 2 capítulos teóricos: um sobre o padrão OTN e o outro sobre o algoritmo de Reed-Solomon, 3 capítulos sobre o projeto desenvolvido, um capítulo de conclusão e, por último, as referências bibliográficas utilizadas.

No capítulo 2 é feito um levantamento teórico do padrão OTN, pois a arquitetura alvo de Reed-Solomon que é desenvolvida ao longo deste trabalho deve ser compatível com este padrão. Neste capítulo é apresentada a estrutura do frame OTN, um resumo sobre as funções dos sub-campos que compõem o frame, bem como uma breve explicação de como é realizada a correção de erros neste padrão. O capítulo 3 aborda a teoria de campos finitos utilizada na construção dos códigos Reed-Solomon, bem como os algoritmos de codificação e decodificação, explicados de maneira teórica.

No capítulo 4 é feito um pequeno levantamento bibliográfico sobre as implementações existentes de Reed-Solomon em hardware que puderam ser encontradas na literatura, onde são discutidas as vantagens e desvantagens de cada uma destas implementações. Neste capítulo também apresentamos os módulos desenvolvidos através de esquemáticos e formas de onda, justificando também a escolha da arquitetura que foi desenvolvida. O capítulo 5 trata da validação dos módulos tanto em simulação quanto em hardware, utilizando uma FPGA como arquitetura alvo. No capítulo 5 também são apresentados os resultados obtidos com o presente trabalho. O capítulo 6 apresenta alguns dados sobre a integração com o *framer*, módulo que faz a interface com a rede de fibra óptica. Para finalizar, no capítulo 7 discutiremos as conclusões obtidas pelo grupo ao término do trabalho, bem como os trabalhos futuros que poderão ser realizados.

2 Padrão OTN

O padrão OTN inclui uma série de recomendações criadas a partir da necessidade de aumentar o desempenho e padronizar o funcionamento das redes de comunicação de dados em fibra óptica. A ITU-T define uma rede OTN como um conjunto de elementos de redes ópticas, conectados através de enlaces de fibra óptica, capazes de prover funcionalidades de transporte, multiplexação, roteamento, gerenciamento, supervisão e sobrevivência de canais ópticos, transportando sinais de diversos protocolos.

Este trabalho tem como foco a recomendação G.709 do padrão OTN, que define as interfaces para o transporte de dados em redes ópticas, sem especificar o comportamento elétrico e mecânico do padrão. Esta recomendação define principalmente a estrutura do frame que trafega na rede, as taxas de transmissão suportadas pelo padrão e a definição da codificação de canal, que é o foco do presente trabalho.

A principal vantagem do padrão OTN em relação aos seus antecessores, SONET e SDH, reside justamente na melhoria das técnicas de correção de erro, que permitem enlaces mais longos, redução na potência de transmissão e uma melhora significativa na taxa de erros de um canal. Outras vantagens que tornam este padrão mais evoluído em relação aos anteriores são: a transparência no transporte de diferentes protocolos e algumas melhorias realizadas para facilitar o gerenciamento de canais óticos. A única desvantagem que pode ser citada é a sua maior complexidade de implementação [4].

2.1 Taxas de Transmissão

A recomendação G.709 oferece atualmente três diferentes taxas de transmissão: OTU-1, OTU-2 e OTU-3. Essas velocidades são derivadas das taxas já existentes nos padrões SONET e SDH. A Tabela 1 lista as taxas de transmissão de acordo com a interface G.709 e a interface correspondente em SONET e SDH [5].

Tabela 1 – Taxas de Transmissão Previstas Pela Recomendação G.709.

Interface G.709	Taxa de Transmissão Total	Interface Correspondente em SONET	Interface Correspondente em SDH	Taxa de Transmissão do payload
OTU-1	2.666 Gbps	OC-48	STM-16	2.488 Gbps
OTU-2	10.709 Gbps	OC-192	STM-64	9.953 Gbps
OTU-3	43.018 Gbps	OC-768	STM-256	39.813 Gbps

2.2 Estrutura do Frame G.709 OTN

O frame definido pela recomendação G.709 é constituído de três partes: overhead, payload e FEC. Dois destes campos, overhead e payload, já faziam parte dos protocolos SONET e SDH. A novidade do padrão OTN é a inclusão de um campo exclusivo para FEC, pois seus antecessores utilizavam apenas alguns bytes do overhead para realizar esta técnica, o que limitava a capacidade de correção de erro.

A Figura 1 apresenta a estrutura detalhada do frame G.709 OTN, destacando seus campos e ilustrando sua organização em linhas e colunas.

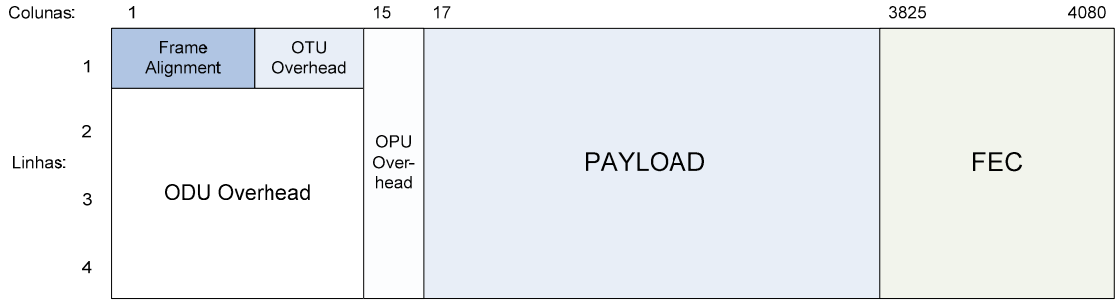


Figura 1 – Estrutura Frame G.709 OTN.

O frame G.709 OTN, também chamado de multi-frame, é composto de 4 frames de 4080 bytes. Cada frame contém 16 bytes de overhead, 3808 bytes de payload e 256 bytes de FEC. Os dados são transmitidos serialmente pelo canal, iniciando-se pelo primeiro frame, seguido pelo envio do segundo frame e assim sucessivamente. O tamanho total do multi-frame, 4x4080, permanece inalterado para qualquer uma das taxas de transmissão vistas anteriormente, diferentemente dos padrões SONET e SDH.

2.3 Overhead

As sessões a seguir apresentarão uma breve explicação sobre os campos que compõem o overhead de um frame G.709 OTN.

2.3.1 Alinhamento

Como os dados são transmitidos serialmente em alta velocidade através da fibra óptica, é necessário que o receptor seja capaz de identificar corretamente o início e o fim de um frame [2]. A capacidade de detectar o início de frame no padrão OTN é feita através do campo 'Frame Alignment', primeiro campo do frame, conforme a Figura 1.

A área reservada para alinhamento possui 3 campos. Um sinal de alinhamento de 6 bytes (FAS), um byte para indicar alinhamento de múltiplos frames (MFAS) e um campo reservado. Dado que o byte MFAS é incrementado a cada multi-frame, este campo permite realizar o alinhamento de até 256 multi-frames consecutivos, permitindo atribuir diferentes definições a

alguns campos do overhead de acordo com seu valor. O sinal de alinhamento (FAS) é reconhecido pelo valor 0xF6F6F6282828, e é a única parte do pacote que é enviada sem a técnica de embaralhamento (*scrambling*). O propósito da técnica de embaralhamento é impedir longos pulsos de 0's ou 1's, permitindo a regeneração do clock e períodos mais longos de transmissão sem alinhamento.

O receptor precisa achar o início do frame antes de começar a processar os dados de controle, e ainda deve ter a capacidade de identificar ausência de sinal, ausência de frame e perda de frame.

2.3.2 OTU Overhead

OTU significa unidade de transporte do canal óptico, do inglês 'Optical channel Transport Unit'. Este campo provê funções de supervisão entre dois pontos conectados por um único canal óptico, que possuem em suas extremidades elementos chamados de 3R (Regeneração, Reformatação e Retemporização), responsáveis pela regeneração do sinal. Está localizado no primeiro frame, entre as colunas 8 e 14.

O OTU overhead consiste em um campo de três bytes para monitoramento de sessão (SM), dois bytes para o campo GCC0 (General Communication Channel 0), utilizado para troca de mensagens entre duas terminações OTU, e um campo de dois bytes reservado para uso futuro. O propósito do campo GCC ainda não está completamente definido, porém provavelmente será usado para gerenciamento de rede ou para sinalização completa de protocolos como o G-MPLS (Generic Multi Protocol Label Switching).

2.3.3 ODU Overhead

O campo ODU overhead está localizado entre as colunas 1 e 14, nos frames 2, 3 e 4 de um multi-frame. A informação contida neste campo provê funções de gerenciamento de rede e supervisão fim-a-fim de canais ópticos. A Figura 2 ilustra os campos que constituem o ODU overhead.

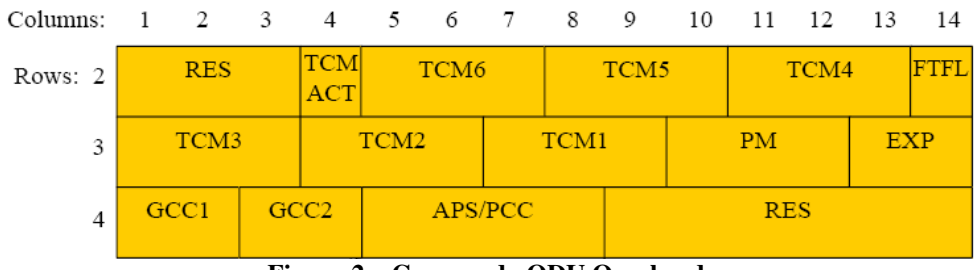


Figura 2 – Campos do ODU Overhead.

A maioria dos campos contidos no ODU overhead traz uma funcionalidade chamada TCM (Tandem Connection Monitoring). Existem seis campos TCM definidos no ODU overhead para uso

em funções de gerência de redes, possibilitando que administradores de rede possam monitorar taxas de erro em diversos enlaces de uma rede de canais ópticos chaveados. A partir da monitoração destes campos é possível identificar os pontos da rede em que o sinal encontra-se mais fraco, possibilitando verificar e localizar falhas. O campo APS/PCC (automatic protection switching and protection communication channel) utiliza as informações de monitoramento para proteger a rede de roteamento em caminhos onde o sinal encontra-se degradado. O campo FTFL (Fault Type and Fault Location) também está relacionado com funções de monitoramento de falhas na rede.

O ODU overhead contém um campo de três bytes para monitoramento de caminho (PM). Este campo tem sua estrutura muito semelhante ao campo SM do OTU overhead, porém sua função é monitorar a conexão de um caminho contendo diversos chaveamentos de canais ópticos, enquanto o SM monitora apenas uma conexão entre dois pontos interligados por apenas uma fibra óptica.

Os demais campos executam funções menos relevantes, os campos RES são reservados para futura padronização, o campo EXP é um campo experimental e não possui padronização e os campos GCC executam praticamente a mesma função que exercem no OTU overhead.

2.3.4 OPU Overhead

O OPU overhead é adicionado ao payload do frame e contém informações responsáveis por adaptar os dados de diferentes protocolos para serem transportados através do frame OTN. Possui um campo chamado PSI (Payload Structure Identifier), que identifica o conteúdo do payload. Atualmente a recomendação G.709 prevê o transporte de protocolos como o SONET/SDH, ATM, GFP entre outros.

Este campo possui três bytes para controle de justificação, chamados de JC (Justification Control), utilizados para compensar deslocamentos no payload, o que pode acontecer quando o frame transporta sinais de protocolos assíncronos. A partir destes três bytes, é feita uma votação para indicar justificação positiva (PJO), ou justificação negativa (NJO), neste caso o último campo do OPU overhead conterá dados do payload.

2.4 FEC

À medida que as taxas de transmissão aumentam e ultrapassam a barreira dos 10Gbps, a degradação do sinal, causada pelo meio de transmissão, causa um impacto significativo na taxa de erro de um canal. Para amenizar este problema e garantir a correta transmissão dos dados, mesmo em taxas elevadas, são utilizados mecanismos de correção de erros.

Algoritmos de correção de erro utilizam informações redundantes codificadas para fazer a verificação e correção de erros no momento da recepção, podendo aumentar significativamente a eficácia na transmissão de dados. Por este motivo, a inclusão de um campo específico para correção de erro no frame OTN é vista como a principal justificativa para o desenvolvimento deste padrão.

A recomendação G.709 define que o mecanismo utilizado para a correção de erros no padrão OTN é o Reed-Solomon RS(255,239). Isto significa que para cada 239 bytes de dados, são adicionados 16 bytes para correção de erro. Este código tem a capacidade de corrigir oito erros para cada 255 bytes. Dessa maneira, em cada frame podem ser corrigidos até 128 erros. Utilizando-se a técnica de entrelaçamento para o cálculo dos bytes de paridade de um frame, é possível recuperar 128 erros consecutivos.

A técnica de entrelaçamento separa um frame em 16 sub-frames de 255 bytes, sendo o primeiro byte do frame pertencente ao primeiro sub-frame, o segundo byte ao segundo sub-frame e assim sucessivamente até o 16° byte, o 17° byte é visto como o segundo byte do primeiro sub-frame. Para cada sub-frame calculam-se 16 bytes de FEC, garantindo a cobertura contra erros em rajada. A Tabela 2 ilustra graficamente a divisão dos frames em sub-frames para o cálculo do FEC.

Tabela 2 – Divisão do frame em sub-frames.

	1	**2**	**...**	**239**	**240**	**...**	**255**
1	1	17	PAYLOAD	3809	3825	FEC	4065
2	2	18		3810	3826		4066
3	3	19		3811	3827		4067
4	4	20		3812	3828		4068
5	5	21		3813	3829		4069
6	OVERHEAD	22		3814	3830		4070
7		23		3815	3831		4071
8		24		3816	3832		4072
9		25		3817	3833		4073
10		26		3818	3834		4074
11		27		3819	3835		4075
12	12	28		3820	3836		4076
13	13	29		3821	3837		4077
14	14	30		3822	3838		4078
15	15	31		3823	3839		4079
16	16	32		3824	3840		4080

É importante observar que cada sub-frame corresponde a uma linha da tabela e possui um cálculo independente de FEC. Os dados são transmitidos na mesma ordem em que estão numerados na tabela, ou seja, coluna por coluna.

3 Algoritmo de Reed-Solomon

Em 1960, Irving Reed e Gus Solomon publicaram o artigo "*Polymonial Codes Over Certain Finite Fields*", no "*Journal of the Society for Industrial and Applied Mathematics*" [6]. Este artigo apresentou uma nova classe de códigos corretores de erro que hoje são chamados de códigos Reed-Solomon. Estes códigos tornaram-se muito úteis devido a sua baixa complexidade e alto poder de correção, e hoje são utilizados em diversas aplicações de armazenamento e transmissão de dados.

Os códigos RS são representados na forma RS(n,k), onde *n* é o número total de símbolos em uma seqüência e *k* é número de símbolos úteis de uma seqüência. Outra variável importante em uma seqüência é a capacidade de correção do código, representada pela letra *t*. O total de símbolos de paridade é dado por 2*t*. A equação (1) apresenta algumas equivalências na construção de códigos RS.

$$\begin{aligned} n &= 2^m - 1 \\ k &= 2^m - 1 - 2t \\ n - k &= 2t \end{aligned} \tag{1}$$

A norma G.709 utiliza símbolos de 8 bits (m=8), o que resulta em n=255. O código utilizado é o RS(255,239) (k=239). Portanto, existem 16 símbolos de paridade para cada 239 bytes de dados, e a capacidade de correção é de 8 símbolos em uma seqüência de 255 (t=8).

3.1 Campos Finitos

O processo de codificação e decodificação de códigos não binários utiliza matemática de campos finitos, também conhecidos como Campos de Galois (GF), em homenagem ao matemático francês Évariste Galois, que postulou a teoria de campos finitos. Para cada número primo, p, existe um campo finito denominado GF(p) que contém p elementos. É possível estender GF(p) em um campo contendo p^m elementos, chamando de campo estendido de GF(p), denominado GF(p^m). A construção de códigos RS utiliza somente campos estendidos, uma vez que se trata de um código não binário [3].

O campo binário GF(2) é um subcampo do campo estendido GF(2^m). Além de 0's e 1's os campos estendidos possuem uma série de elementos representados por potências de α, onde α^n é apenas uma representação abstrata de um dos elementos do campo. Um conjunto infinito de elementos pode ser formado iniciando-se com os elementos $\{0,1,\alpha,\alpha^2,...\}$, onde o próximo elemento da seqüência é gerado através da multiplicação do último elemento obtido por α. Entretanto, para obter um conjunto **finito** de elementos devemos impor as seguintes condições: um campo finito

deve conter apenas 2^m elementos[1] e deve ser fechado nas operações de soma e multiplicação, isto é, uma soma ou produto de um elemento do campo por qualquer outro elemento do campo, deve resultar sempre em um elemento que também pertença ao campo. Desse modo, qualquer elemento de um campo que tiver potência maior que 2^m-1, pode ser reduzido a um elemento com potência menor que 2^m-1, conforme a equação (2).

$$\alpha^{(2^m+n)} = \alpha^{(2^m-1)}\alpha^{n+1} = \alpha^{n+1} \tag{2}$$

Portanto, podemos dizer genericamente que um campo definido por GF(2^m) possui os seguintes elementos:

$$GF(2^m) = \{0, \alpha^0, \alpha^1\alpha^2, ..., \alpha^{2^m-2}\} \tag{3}$$

Cada um dos 2^m elementos de um campo GF(2^m), pode ser representado por um polinômio distinto com grau menor ou igual a *m*-1. Os coeficientes deste polinômio podem ser 0 ou 1, e o grau do polinômio corresponde ao maior expoente com coeficiente igual a 1.

Considerando *m*=3, temos um campo denotado por GF(2^3). Este campo possui 8 elementos, sendo um elemento nulo e 7 elementos não nulos. A Figura 3 mostra o desenvolvimento deste campo em função do polinômio $f(x)=x^3+x+1$. O método de obtenção desta seqüência será apresentado ao longo desta seção.

	X^2	X^1	X^0
0	0	0	0
α^0	0	0	1
α^1	0	1	0
α^2	1	0	0
α^3	0	1	1
α^4	1	1	0
α^5	1	1	1
α^6	1	0	1
α^7	0	0	1

Figura 3 – Elementos de um campo desenvolvido em GF(2^3).

Como podemos ver, o elemento denotado por α^7 é igual ao elemento α^0, isto comprova a equação (3), que mostra que um expoente máximo de α em um campo é 2^m-2, no caso de *m*=3, o campo deve terminar em α^6.

Para gerar um campo finito é necessário utilizar uma classe especial de polinômios, denominada de polinômios primitivos. Existe uma série de regras matemáticas que definem se um

[1] Não confundir 2^m elementos do campo com $n=2^m-1$ (quantidade de símbolos em uma sequência).

polinômio é primitivo ou não, porém já existem polinômios primitivos pré-determinados de acordo com o valor de *m*, e neste trabalho, apenas sua aplicação prática é relevante, portanto a seguinte definição se faz satisfatória: "Um polinômio primitivo é aquele que gera a seqüência completa de elementos de um campo finito sem repetições". Polinômios não primitivos não geram toda a seqüência de elementos pertencentes a um campo, repetindo seqüências incompletas antes de se chegar ao número de elementos do campo.

A Tabela 3 apresenta uma série de polinômios primitivos utilizados na construção de campos finitos, de acordo com o valor de *m* [3].

Tabela 3 – Polinômios Primitivos.

m		*m*	
3	$1+X+X^3$	13	$1+X+X^3+X^4+X^{13}$
4	$1+X+X^4$	14	$1+X^2+X^3+X^4+X^8$
5	$1+X^2+X^5$	15	$1+X+X^{15}$
6	$1+X+X^6$	16	$1+X+X^3+X^{12}+X^{16}$
7	$1+X^3+X^7$	17	$1+X^3+X^{17}$
8	$1+X^2+X^3+X^4+X^8$	18	$1+X^7+X^{18}$
9	$1+X^4+X^9$	19	$1+X+X^2+X^5+X^{19}$
10	$1+X^3+X^{10}$	20	$1+X^3+X^{21}$
11	$1+X^2+X^{11}$	21	$1+X^2+X^{22}$
12	$1+X+X^4+X^6+X^{12}$	22	$1+X+X^{23}$

O mapeamento dos elementos de um campo finito é feito através do seu polinômio primitivo. Para ilustrar de maneira simples a formação de um campo finito a partir de seu polinômio primitivo, consideramos o campo apresentado na Figura 3, onde o polinômio primitivo é $f(x)=x^3+x+1$ e $m=3$. As raízes deste polinômio podem ser encontradas achando-se os valores de x que satisfazem a igualdade $f(x)=0$. Os elementos binários básicos, 0 e 1, não são raízes deste polinômio, pois $f(0)=1$ e $f(1)=1$, e de acordo com a teoria da álgebra um polinômio de grau m deve ter m raízes. Entretanto, estamos trabalhando com o campo estendido $GF(2^3)$, que possui por definição 8 elementos, dos quais 3 devem ser raízes. Neste caso, tomando α^1 como uma das raízes é possível obter a seguinte igualdade:

$$\begin{aligned} & f(\alpha) = 0 \\ & 1+\alpha+\alpha^3 = 0 \\ & \alpha^3 = -1-\alpha \end{aligned} \qquad (4)$$

Em campos binários, a soma de dois elementos pode ser obtida através da operação XOR. Portando, a seguinte igualdade é válida: -1=+1. Logo, a igualdade obtida na equação (4) pode ser escrita da seguinte forma:

$$\alpha^3 = 1+\alpha \qquad (5)$$

Desta forma α^3 é definido como uma soma de elementos α de menor ordem, e a partir de α^3 é possível obter os demais elementos do campo através de multiplicações por α, como mostram as equações a seguir:

$$\begin{aligned}
\alpha^4 &= \alpha \cdot \alpha^3 = \alpha \cdot (1+\alpha) = \alpha + \alpha^2 \\
\alpha^5 &= \alpha \cdot \alpha^4 = \alpha \cdot (\alpha + \alpha^2) = \alpha^2 + \alpha^3 = \alpha^2 + \alpha + 1 \\
\alpha^6 &= \alpha \cdot \alpha^5 = \alpha \cdot (\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha = 1 + \alpha + \alpha^2 + \alpha = 1 + \alpha^2
\end{aligned} \tag{6}$$

A equação (7) apresenta algumas definições feitas para os elementos α de ordem menor que 3.

$$\begin{aligned}
\alpha^0 &= 1 \\
\alpha^1 &= x \\
\alpha^2 &= x^2
\end{aligned} \tag{7}$$

A partir das definições apresentadas na equação (7) e das igualdades encontradas nas equações (5) e (6), é possível obter o campo completo, como mostra a Tabela 4.

Tabela 4 – Construção do campo GF(2^3).

Potência de α	Polinômio	Valor	Decimal
0	0	000	0
α^0	1	001	1
α^1	X	010	2
α^2	X^2	100	4
α^3	$X+1$	011	3
α^4	X^2+X	110	6
α^5	X^2+X+1	111	7
α^6	X^2+1	101	5

O mapeamento dos elementos de um campo finito pode ser desenvolvido através de um registrador de deslocamento com realimentação linear, mais conhecido como LFSR (Linear Feedback Shift Register). O circuito apresentado na Figura 4 gera os 7 elementos não nulos do campo apresentado na Figura 3 e desenvolvido da equação (4) até a equação (7). É importante observar que as conexões com realimentação correspondem aos coeficientes do polinômio gerador $f(x)=x^3+x+1$. Iniciando-se com o estado não nulo 001, que corresponde a α^0, todos as demais potências de α são geradas, na mesma ordem em que aparecem na Tabela 4.

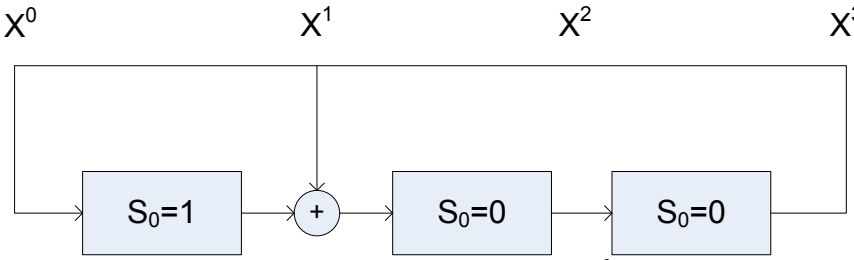


Figura 4 – Circuito LFSR gerador do campo GF(2^3), S_0 ≡ estado inicial.

Uma importante propriedade dos campos finitos é a definição das operações de adição e multiplicação, pois um campo finito deve ser fechado para estas operações. A Tabela 5 apresenta os resultados para as operações de adição e multiplicação dos elementos do campo GF(2^3), somente para elementos não nulos. A soma pode ser obtida através da operação XOR quando os elementos encontram-se na representação binária, já o produto é obtido através da soma dos expoentes de α módulo (2^m-1), neste caso módulo 7.

Tabela 5 – Adição e multiplicação sobre o campo GF(2^3).

Adição							
	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	0	α^3	α^6	α^1	α^5	α^4	α^2
α^1	α^3	0	α^4	α^0	α^2	α^6	α^5
α^2	α^6	α^4	0	α^5	α^1	α^3	α^0
α^3	α^1	α^0	α^5	0	α^6	α^2	α^4
α^4	α^5	α^2	α^1	α^6	0	α^0	α^3
α^5	α^4	α^6	α^3	α^2	α^0	0	α^1
α^6	α^2	α^5	α^0	α^4	α^3	α^1	0

Multiplicação							
	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^1	α^1	α^2	α^3	α^4	α^5	α^6	α^0
α^2	α^2	α^3	α^4	α^5	α^6	α^0	α^1
α^3	α^3	α^4	α^5	α^6	α^0	α^1	α^2
α^4	α^4	α^5	α^6	α^0	α^1	α^2	α^3
α^5	α^5	α^6	α^0	α^1	α^2	α^3	α^4
α^6	α^6	α^0	α^1	α^2	α^3	α^4	α^5

Tomando como exemplo a soma $\alpha^3(011)+\alpha^5(111)$, o resultado corresponde a operação 011 XOR 111, que resulta em 100, correspondente a α^2, conforme a Tabela 4.

3.2 Codificação Reed-Solomon

O processo de codificação Reed-Solomon consiste na geração dos símbolos de paridade de uma seqüência. Como foi citado anteriormente, o número de símbolos de paridade de uma seqüência corresponde a 2*t*, onde *t* é o número de símbolos que o código é capaz de corrigir. A codificação RS leva em conta o polinômio gerador do código, que genericamente possui a seguinte forma:

$$g(X) = g_0 + g_1 X + g_2 X^2 + \ldots + g_{2t-1} X^{2t-1} + X^{2t} \tag{8}$$

O grau do polinômio gerador é igual ao número de símbolos de paridade que o código possui, no caso 2*t*, o que significa que devem existir exatamente 2*t* potências de α que sejam raízes deste polinômio. Normalmente são utilizadas as potências α, α^2,..., α^{2t}, porém não é necessário iniciar com a raiz α, podendo-se iniciar com qualquer potência de α. A equação (9) apresenta a

obtenção de um polinômio gerador para o código RS(7,3), que possui 4 símbolos de paridade, logo seu polinômio gerador terá 4 raízes.

$$\begin{aligned} g(X) &= (X-\alpha)(X-\alpha^2)(X-\alpha^3)(X-\alpha^4) \\ g(X) &= (X^2-(\alpha+\alpha^2)X+\alpha^3)(X^2-(\alpha^3+\alpha^4)X+\alpha^7) \\ g(X) &= (X^2-\alpha^4X+\alpha^3)(X^2-\alpha^6X+\alpha^0) \\ g(X) &= X^4-(\alpha^4+\alpha^6)X^3+(\alpha^3+\alpha^{10}+\alpha^0)X^2-(\alpha^4+\alpha^9)X+\alpha^3 \\ g(X) &= X^4-\alpha^3X^3+\alpha^0X^2-\alpha^1X+\alpha^3 \end{aligned} \tag{9}$$

Seguindo a representação do grau menor até o maior, e trocando os sinais negativos por positivos, pois como vimos na sessão anterior $+\alpha^n=-\alpha^n$ na teoria de campos finitos, o polinômio $g(X)$ obtido na equação (9) pode ser representado conforme a equação abaixo:

$$g(X) = \alpha^3 + \alpha^1X + \alpha^0X^2 + \alpha^3X^3 + X^4 \tag{10}$$

No caso da recomendação G.709 OTN, existem 16 símbolos de paridade, o que leva a um polinômio gerador de grau 16. A equação (11) mostra como é obtido o polinômio gerador para o RS(255,239) do padrão OTN [1].

$$g(X) = \prod_{n=0}^{15}(X-\alpha^n) \tag{11}$$

Para ilustrar o mecanismo de geração dos símbolos de paridade, apresentaremos um exemplo utilizando o código RS(7,3), onde temos 3 símbolos de mensagem e 4 de paridade, levando em conta o envio da mensagem $\alpha^1\ \alpha^3\ \alpha^5$. Esta mensagem exemplo corresponde ao envio do símbolo $111(\alpha^5)$, seguido dos símbolos $011(\alpha^3)$ e $001(\alpha^1)$. Primeiramente representamos a mensagem na forma polinomial, $m(X)$, como segue:

$$m(X) = \alpha^1 + \alpha^3X + \alpha^5X^2 \tag{12}$$

Para obter os símbolos de paridade é necessário encontrar o resto da divisão polinomial de *m(X)* por *g(X)*. Para isso, devemos multiplicar $m(X)$ por X^{2t}, deslocando-a $2t$=4 estágios para direita, o que resulta no seguinte polinômio: $\alpha^1X^4+\alpha^3X^5+\ \alpha^5X^6$. O próximo passo é dividir a mensagem deslocada para a direita pelo polinômio gerador apresentado na equação (10), o que deve resultar no seguinte polinômio:

$$p(X) = \alpha^0 + \alpha^2X + \alpha^4X^2 + \alpha^6X^3 \tag{13}$$

O processo de divisão polinomial em campos finitos deve levar em conta as operações de adição e multiplicação contidas na Tabela 5, e é uma operação dispendiosa de se executar na forma algébrica, sem o auxílio de software ou de hardware. Na prática, a codificação RS é realizada

através de registrador de deslocamento (LFSR), que consiste numa operação bastante simples. Por este motivo, o processo de divisão não será demonstrado neste trabalho.

A seqüência completa de um código é obtida através da soma do polinômio de mensagem deslocado para a direita, $\alpha^1X^4+\alpha^3X^5+\alpha^5X^6$, e do polinômio de paridade, apresentado na equação (13), o que resulta no seguinte polinômio:

$$U(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \quad (14)$$

Onde os símbolos são enviados na ordem do maior grau do polinômio até o menor.

3.2.1 Codificação Reed-Solomon Através de Registrador de Deslocamento

A utilização de circuitos eletrônicos para realizar a codificação Reed-Solomon, levando em conta o polinômio gerador apresentado na equação (10), necessita a implementação de um registrador de deslocamento com realimentação (LFSR), conforme mostra a Figura 5.

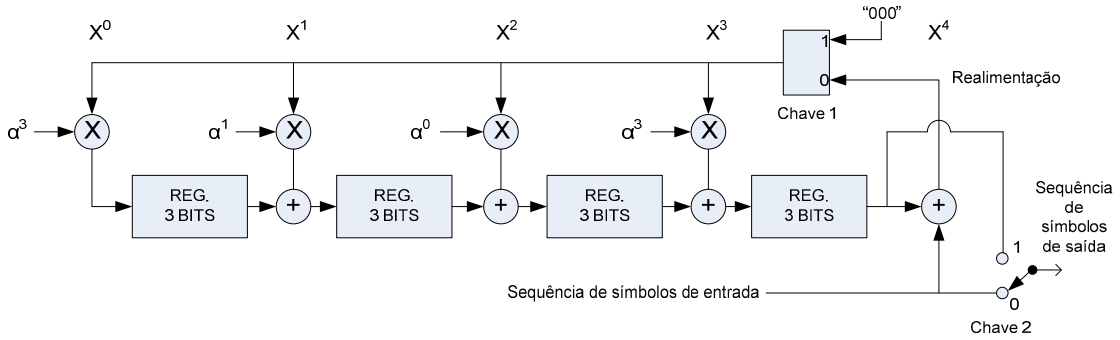


Figura 5 – Codificador Reed-Solomon para RS(7,3).

Observa-se que os termos que multiplicam a entrada realimentada do LFSR correspondem aos coeficientes do polinômio gerador conforme a equação (10). Cada registrador deste LFSR armazena um valor de 3 bits, que pode representar uma das 7 potências de α ou o valor 0.

O processo de geração dos símbolos de paridade pelo circuito da Figura 5 pode ser descrito da seguinte maneira:

1) Durante os primeiros *k* ciclos de clock, onde *k* representa o número de símbolos de dados úteis de uma seqüência, a chave 1 permanece na posição 0, permitindo a realimentação nos *n-k* estágios do registrador de deslocamento, onde *n-k* indica o número de símbolos de paridade.

2) A chave 2 fica na posição 0, conectando a entrada diretamente à saída, permitindo o *by-pass* dos símbolos de mensagem.

3) Após a transferência da mensagem de *k* símbolos, as chaves passam para a posição 1, ligando o LFSR com a saída do circuito e zerando a realimentação.

4) Nos ciclos de clock restantes (*n-k*), os símbolos de paridade contidos nos registradores são deslocados para a saída e os valores dos registradores são zerados.

A tabela abaixo mostra o processo de geração dos símbolos de paridade a partir do LFSR, detalhando o conteúdo dos registradores durante o processo de entrada dos 3 símbolos de mensagem utilizados anteriormente: $\alpha^1\ \alpha^3\ \alpha^5$.

Tabela 6 – Conteúdo dos registradores do circuito LFSR durante a codificação.

Input Queue			Ciclo de Clock	Conteúdo dos Registradores				Realimentação
α^1	α^3	α^5	0	0	0	0	0	α^5
	α^1	α^3	1	α^1	α^6	α^5	α^1	α^0
		α^1	2	α^3	0	α^2	α^2	α^4
		-	3	α^0	α^2	α^4	α^6	-

Após o terceiro ciclo de clock, os registradores possuem os quatro símbolos de paridade α^0, α^2, α^4 e α^6, como mostra a última linha da tabela. Após o cálculo, com as chaves 1 e 2 na posição 1, os símbolos de paridade são deslocados para a saída ao mesmo tempo que os registradores são zerados para receber uma nova mensagem. Desse modo, a seqüência completa na forma polinomial pode ser expressa da seguinte maneira:

$$\begin{aligned} U(X) &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \\ &= (001) + (100)X + (110)X^2 + (101)X^3 + (010)X^4 + (011)X^5 + (111)X^6 \end{aligned} \tag{15}$$

As raízes da seqüência gerada devem ser as mesmas raízes do polinômio gerador, de maneira que qualquer uma das raízes de $g(X)$ aplicadas como entrada no polinômio $U(X)$ deve resultar zero. A partir desta análise é possível verificar se existem erros em uma mensagem recebida.

3.3 Decodificação Reed-Solomon

Neste capítulo foi desenvolvido um exemplo de codificação Reed-Solomon utilizando um código RS(7,3), gerando a seqüência descrita pela equação (14). Para exemplificar o processo de decodificação assumiremos que durante a transmissão, a seqüência tenha sido corrompida de tal maneira que dois símbolos tenham sido recebidos com erro. Lembrando que um código RS(7,3) pode corrigir no máximo dois erros.

Para uma seqüência de sete símbolos, o padrão de erro pode ser representado pela seguinte equação:

$$e(X) = \sum_{n=0}^{6} e_n X^n \tag{16}$$

Supondo que para este exemplo o padrão de erro da mensagem corrompida seja:

$$
\begin{aligned}
e(X) &= 0 + 0X + 0X^2 + \alpha^2 X^3 + \alpha^5 X^4 + 0X^5 + 0X^6 \\
&= (000) + (000)X + (000)X^2 + (100)X^3 + (111)X^4 + (000)X^5 + (000)X^6
\end{aligned} \tag{17}
$$

Podemos dizer que um símbolo de paridade foi corrompido em 1 bit (α^2), e um símbolo da mensagem foi corrompido em 3 bits (α^5). A seqüência recebida, $r(X)$, pode ser representada como a soma da seqüência transmitida e do padrão de erro, como mostra a equação (18).

$$
r(X) = U(X) + e(X) \tag{18}
$$

A partir da equação (18), do polinômio transmitido $U(X)$, equação (14), e do polinômio de erro $e(X)$, equação (17), é possível obter a mensagem recebida após corrupção dos dados, como mostra a equação (19).

$$
\begin{aligned}
r(X) &= (001) + (100)X + (110)X^2 + (001)X^3 + (101)X^4 + (011)X^5 + (111)X^6 \\
r(X) &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \underline{\underline{\alpha^0 X^3}} + \underline{\underline{\alpha^6 X^4}} + \alpha^3 X^5 + \alpha^5 X^6
\end{aligned} \tag{19}
$$

Neste exemplo existem quatro incógnitas: duas localizações de erro e dois valores de erro. Esta é uma diferença importante de códigos não binários em relação a códigos binários. Em códigos binários basta achar a localização do erro, enquanto em códigos não binários é necessário descobrir também o valor do erro.

O processo de decodificação Reed-Solomon possui diversas etapas, tais como: o cálculo da síndrome, o cálculo dos coeficientes do polinômio localizador de erro, o cálculo das localizações de erro e dos valores dos erros. Como existe um tempo de processamento em cada uma das etapas da decodificação, é necessário também um bloco de atraso para sincronizar a geração dos valores de erro com a mensagem recebida. Desta maneira podemos representar a arquitetura simplificada de um decodificador Reed-Solomon da seguinte maneira:

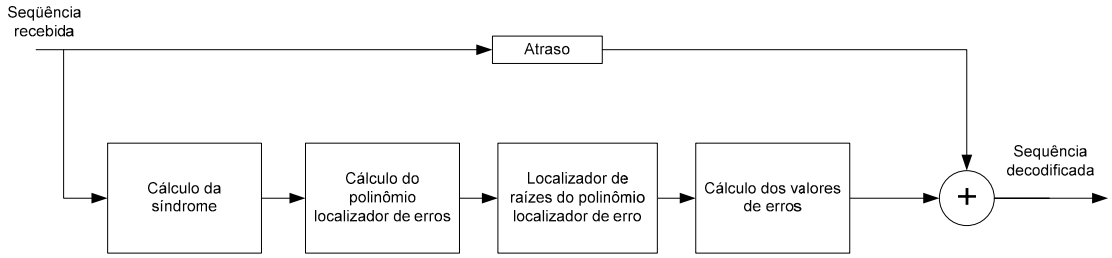


Figura 6 – Diagrama de blocos do decodificador Reed-Solomon.

3.3.1 Cálculo da Síndrome

Uma síndrome é o resultado da verificação das raízes do polinômio gerador como entrada do polinômio $r(x)$, para determinar se a mensagem recebida pertence ao conjunto de seqüências válidas de um código. Se uma seqüência pertence ao conjunto de seqüências válidas, o valor da

síndrome deve ser 0. Qualquer valor diferente de zero indica a presença de erros. Uma síndrome é composta de *n-k* símbolos. Logo o código RS(7,3) possui síndromes de 4 símbolos.

A equação (20) mostra que qualquer seqüência válida é múltipla do polinômio gerador (equação (10)), e por este motivo as raízes de $g(X)$ devem também ser raízes de $U(X)$.

$$U(X) = m(X) \cdot g(X) \tag{20}$$

Como $r(X)=U(X)+e(X)$, o polinômio $r(X)$ avaliado para cada raiz do polinômio $g(X)$, só vai retornar zeros se a seqüência recebida não conter erros. O cálculo da síndrome pode ser descrito da seguinte forma:

$$S_i = r(X)\big|_{X=\alpha^i} = r(\alpha^i) \qquad i = 1,...,n-k \tag{21}$$

Como exemplo de cálculo da síndrome, aplicaremos a equação (21) no polinômio $r(X)$ descrito na equação (19), que possui dois símbolos corrompidos. As equações (22), (23), (24) e (25) apresentam o cálculo dos quatro símbolos de síndrome deste exemplo. Os expoentes de α são obtidos através da substituição de X por α^i na equação $r(X)$, logo após são reduzidos utilizando-se a operação 'mod 7'. O resultado final é obtido através da aplicação da função XOR em todos os símbolos que compõem a soma.

$$\begin{aligned} S_1 &= r(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^9 + \alpha^8 + \alpha^{11} \\ S_1 &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^4 \\ S_1 &= \alpha^3 \end{aligned} \tag{22}$$

$$\begin{aligned} S_2 &= r(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^8 + \alpha^6 + \alpha^{14} + \alpha^{13} + \alpha^{17} \\ S_2 &= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^6 + \alpha^0 + \alpha^6 + \alpha^3 \\ S_2 &= \alpha^5 \end{aligned} \tag{23}$$

$$\begin{aligned} S_3 &= r(\alpha^3) = \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^9 + \alpha^{18} + \alpha^{18} + \alpha^{23} \\ S_3 &= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^4 + \alpha^2 \\ S_3 &= \alpha^6 \end{aligned} \tag{24}$$

$$\begin{aligned} S_4 &= r(\alpha^4) = \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{12} + \alpha^{22} + \alpha^{23} + \alpha^{29} \\ S_4 &= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^5 + \alpha^1 + \alpha^2 + \alpha^1 \\ S_4 &= 0 \end{aligned} \tag{25}$$

Este exemplo confirma que existem erros em $r(X)$, pois três símbolos da síndrome são diferentes de zero.

3.3.2 Localização dos Erros

Supondo que existem v erros em uma seqüência, nas localizações X^{j1}, X^{j2},..., X^{jv}, o polinômio de erro $e(X)$ apresentado na equação (16) pode ser representado da seguinte maneira:

$$e(X) = e_{j1}X^{j1} + e_{j2}X^{j2} + ... + e_{jv}X^{jv} \tag{26}$$

Para corrigir uma seqüência corrompida é necessário encontrar os valores dos erros, denotados por e_{jl}, e as localizações dos erros, denotadas por X_{jl}, onde l=1,2,...,v. Primeiramente definimos uma variável correspondente a uma localização de erro como $\beta_l = \alpha^{jl}$. Em seguida, obtemos os 2t símbolos da síndrome substituindo α^i no polinômio de erro $e(X)$, para i=1,2,...,2t, como mostra a equação (27). É importante observar que as síndromes podem ser tanto avaliadas através da mensagem recebida $r(X)$, quanto através do padrão de erro $e(X)$, pois como mostra a equação (18) o polinômio $r(X)$ é uma soma de $U(X)$ e $e(X)$, e a síndrome de $U(X)$ sempre será zero.

$$\begin{aligned} S_1 &= e(\alpha) = e_{j1}\beta_1 + e_{j2}\beta_2 + ... + e_{jv}\beta_v \\ S_2 &= e(\alpha^2) = e_{j1}\beta_1^2 + e_{j2}\beta_2^2 + ... + e_{jv}\beta_v^2 \\ &\cdots \\ S_{2t} &= e(\alpha^{2t}) = e_{j1}\beta_1^{2t} + e_{j2}\beta_2^{2t} + ... + e_{jv}\beta_v^{2t} \end{aligned} \tag{27}$$

Existem 2t incógnitas: t valores de erros e t localizações. Também existem 2t equações. Entretanto, as 2t equações não podem ser resolvidas de modo usual pois são não-lienares (algumas incógnitas possuem expoentes). As técnicas que resolvem este sistema de equações são conhecidas como algoritmos de decodificação Reed-Solomon.

Uma vez que um vetor de síndromes diferente de zero foi calculado, podemos afirmar que a seqüência recebida possui erros. Portanto, é necessário descobrir a localização dos erros. Um polinômio localizador de erros, $\sigma(X)$, pode ser definido da seguinte maneira:

$$\begin{aligned} \sigma(X) &= (1+\beta_1 X)(1+\beta_2 X)...(1+\beta_v X) \\ \sigma(X) &= 1 + \sigma_1 X + \sigma_2 X^2 + ... + \sigma_v X^v \end{aligned} \tag{28}$$

As raízes de $\sigma(X)$ são $1/\beta_1$, $1/\beta_2$,...,$1/\beta_v$. Logo, o inverso das raízes de $\sigma(X)$ indicam as localizações de erro de do padrão de erro $e(X)$. Para determinar os coeficientes σ_1, σ_2,..., σ_v, é necessário utilizar uma técnica denominada modelagem auto-regressiva, que utiliza uma matriz de síndromes, onde as t primeiras síndromes são utilizadas para prever a próxima síndrome, como mostra a equação abaixo:

$$\begin{bmatrix} S_1 & S_2 & \dots & S_t \\ S_2 & S_3 & \dots & S_{t+1} \\ \dots & \dots & \dots & \dots \\ S_t & S_{t+1} & \dots & S_{2t-1} \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \dots \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \dots \\ -S_{2t} \end{bmatrix} \tag{29}$$

No caso de um código RS(7,3) a capacidade máxima de correção é de 2 erros, portanto devemos encontrar σ_1 e σ_2, o que leva a construção de uma matriz 2x2, apresentada na equação abaixo:

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} S_2 \\ S_4 \end{bmatrix} \tag{30}$$

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} \tag{31}$$

Um dos possíveis métodos para encontrar os valores de σ_1 e σ_2, é através da inversa da matriz 2x2, como mostra a equação (32). A inversão de matrizes em campos finitos leva em consideração as mesmas propriedades utilizadas nos conjuntos numéricos conhecidos, porém sempre levando em conta as operações de adição e multiplicação restritas ao campo finito.

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix}^{-1} \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} = \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix}$$

$$\begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix} \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha^0 \\ \alpha^6 \end{bmatrix} \tag{32}$$

A partir dos coeficientes σ_1 e σ_2 encontrados na equação (32) e do polinômio $\sigma(X)$ apresentado na equação (28), podemos representar o polinômio localizador de erro da seguinte forma:

$$\sigma(X) = \alpha^0 + \alpha^6 X + \alpha^0 X^2 \tag{33}$$

As raízes do polinômio $\sigma(X)$ são as inversas das localizações de erro. Uma vez que estas raízes são encontradas, as localizações de erro serão conhecidas. As raízes de $\sigma(X)$ podem ser um ou mais elementos do campo. Estas raízes podem ser encontradas através de um teste exaustivo do polinômio $\sigma(X)$ para todos os elementos do campo, como mostram as equações abaixo. Qualquer elemento X que resultar em $\sigma(X)=0$ é uma raiz e corresponde a uma localização de erro.

$$
\begin{aligned}
\sigma(\alpha^0) &= \alpha^0 + \alpha^6 + \alpha^0 = \alpha^6 \neq 0 \\
\sigma(\alpha^1) &= \alpha^0 + \alpha^7 + \alpha^2 = \alpha^2 \neq 0 \\
\sigma(\alpha^2) &= \alpha^0 + \alpha^8 + \alpha^4 = \alpha^6 \neq 0 \\
\sigma(\alpha^3) &= \alpha^0 + \alpha^9 + \alpha^6 = 0 \Rightarrow ERRO \\
\sigma(\alpha^4) &= \alpha^0 + \alpha^{10} + \alpha^8 = 0 \Rightarrow ERRO \\
\sigma(\alpha^5) &= \alpha^0 + \alpha^{11} + \alpha^{10} = \alpha^2 \neq 0 \\
\sigma(\alpha^6) &= \alpha^0 + \alpha^{12} + \alpha^{12} = \alpha^0 \neq 0
\end{aligned}
\tag{34}
$$

Podemos ver que os elementos α^3 e α^4 são raízes do polinômio $\sigma(X)$. Isto significa que existem erros nas localizações $\beta_1=1/\alpha^4=\alpha^3$ e $\beta_2=1/\alpha^3=\alpha^4$, pois as raízes de $\sigma(X)$ equivalem ao inverso das localizações de erro.

3.3.3 Valores dos Erros

Um erro foi definido como e_{jl}, onde o índice j se refere a localização e o índice l identifica a ordem do erro. Como cada valor de erro é vinculado a uma localização particular, a notação pode ser simplificada para e_l. Para determinar os valores de erro associados as localizações encontradas na sessão anterior, $\beta_1=\alpha^3$ e $\beta_2=\alpha^4$, qualquer uma das quatro equações de síndromes podem ser utilizadas. A partir da equação (27) vamos usar S_1 e S_2.

$$
\begin{aligned}
S_1 &= e(\alpha) = e_1\beta_1 + e_2\beta_2 \\
S_2 &= e(\alpha^2) = e_1\beta_1^2 + e_2\beta_2^2
\end{aligned}
\tag{35}
$$

Estas equações podem ser escritas na forma polinomial da seguinte maneira:

$$
\begin{aligned}
\begin{bmatrix} \beta_1 & \beta_2 \\ \beta_1^2 & \beta_2^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} &= \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} \\
\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} &= \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix}
\end{aligned}
\tag{36}
$$

Para resolver a equação (36) e encontrar os valores de erro, e_1 e e_2, podemos utilizar a mesma técnica apresentada na equação (32), como mostram as equações a seguir:

$$
\begin{aligned}
\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1 \end{bmatrix}^{-1} \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} &= \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} \\
\begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix}^{-1} \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} &= \begin{bmatrix} \alpha^2 \\ \alpha^5 \end{bmatrix}
\end{aligned}
\tag{37}
$$

Com as localizações e os valores de erro encontrados é possível estimar o polinômio de erro $\hat{e}(X)$ conforme a equação abaixo:

$$\begin{aligned} \hat{e}(X) &= e_1 X^{j1} + e_2 X^{j2} \\ \hat{e}(X) &= \alpha^2 X^3 + \alpha^5 X^4 \end{aligned} \tag{38}$$

O algoritmo apresentado deve decodificar corretamente a mensagem recebida, uma vez que ela foi corrompida em dois símbolos, que é exatamente a capacidade de correção do código. Para confirmar se a decodificação foi executada corretamente é necessário estimar a seqüência transmitida, $\hat{U}(X)$, e verificar se ela representa a verdadeira mensagem transmitida, $U(X)$. Esta verificação é feita da seguinte maneira:

$$\hat{U}(X) = r(X) + \hat{e}(X) \tag{39}$$

Extraindo $r(X)$ da equação (19) e $\hat{e}(X)$ da equação (38), encontramos o seguinte polinômio $\hat{U}(X)$:

$$\begin{aligned} \hat{U}(X) &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + (\alpha^0 + \alpha^2) X^3 + (\alpha^6 + \alpha^5) X^4 + \alpha^3 X^5 + \alpha^5 X^6 \\ \hat{U}(X) &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \end{aligned} \tag{40}$$

A partir das equações (14) e (40) podemos constatar que $\hat{U}(X)=U(X)$, o que indica que a mensagem foi decodificada corretamente, após a corrupção de dois símbolos.

3.4 Conclusões

Neste capítulo estudamos os algoritmos de codificação e decodificação Reed-Solomon através de um exemplo simples, levando em conta o código RS(7,3). O propósito deste trabalho é aplicar estes algoritmos para a implementação do código RS(255,239), que possui símbolos de 8 bits, o que acelera as etapas de codificação e decodificação e possui maior poder de correção de erros. Entretanto, quanto maior o símbolo, maior é a complexidade de implementação.

4 Arquitetura desenvolvida

No capítulo anterior estudamos o algoritmo de Reed-Solomon com ênfase na matemática de campos finitos e na demonstração matemática das etapas de codificação e decodificação. Porém, na prática, o algoritmo de Reed-Solomon é implementado de uma maneira um pouco diferente, pois não estamos mais no domínio de campos finitos e sim no domínio binário. Por estar trabalhando no nível lógico e utilizando a matemática de campos finitos foi necessário desenvolver um multiplicador especial. O projeto do multiplicador utilizado neste trabalho é apresentado na seção 4.1.

O processo de codificação normalmente segue o mesmo princípio de registrador de deslocamento apresentado na Figura 5. Entretanto, a arquitetura do decodificador pode variar bastante dependendo das necessidades de cada projeto. Diversas propostas de implementação foram estudadas para definir qual é a mais adequada para este trabalho, sendo avaliadas as vantagens e desvantagens de cada uma delas.

No padrão OTN, um frame é dividido em 16 sub-frames de 255 bytes, e cada sub-frame corresponde a uma mensagem que deve ser codificada ou decodificada paralelamente, como foi descrito no capítulo 2. O módulo responsável por fornecer os dados para o decodificador deve ser capaz de realizar esta tarefa a uma freqüência de 200Mhz e 64 bits por ciclo de clock. Portanto, a replicação dos módulos, tanto do codificador quanto do decodificador, é obrigatória, pois cada módulo pode tratar um dado de 8 bits por ciclo.

4.1 Multiplicador Genérico Para GF(2^m)

Para realizar o produto de dois símbolos arbitrários do campo de Galois(2^m), onde m representa o tamanho em bits destes símbolos, pode-se fazer o uso de um dos diversos circuitos multiplicadores existentes. O tipo mais comum de multiplicador utiliza LFSR, ou registrador de deslocamento com realimentação linear [7]. Este tipo mostra-se bastante simples e econômico, porém efetua o produto sequencialmente em m ciclos de relógio o que conduz a baixo desempenho. Sendo o desempenho um dos requisitos deste trabalho, pois a taxa de injeção de dados no codificador e decodificador é muito alta, se decidiu utilizar um multiplicador que execute sua computação espacialmente ao invés de sequencialmente.

Inicialmente utilizou-se multiplicadores implementados em tabela. Porém, esta alternativa se mostrou dispendiosa, pois requer uma ocupação de área demasiada grande. Sendo a área também um dos principais requisitos deste trabalho, recorreu-se ao uso do multiplicador genérico, organizado de maneira celular, descrito em [7]. A única vantagem do multiplicador implementado em tabela é a velocidade de processamento, porém enquanto este utiliza 128 blocos lógicos para

uma multiplicação por constante, um multiplicador celular utiliza apenas 10. Para uma multiplicação genérica utiliza-se 360 blocos lógicos para um multiplicador implementado em tabela, enquanto que um celular utiliza 60.

Este multiplicador é composto de uma matriz de m^2 células combinacionais idênticas conectadas entre si em cadeia. Na Figura 7 observa-se a célula e o multiplicador juntamente com suas linhas, ou registradores, de entrada e saída.

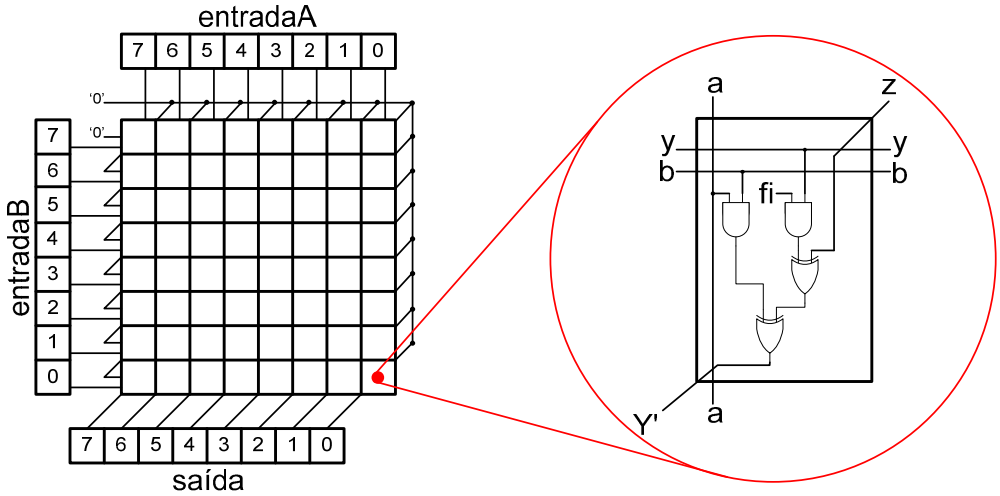


Figura 7 – Multiplicador Genérico para GF(2^8).

A ***entradaA*** e a ***entradaB*** são conectadas, respectivamente, no sinal ***a*** das células da primeira linha e no sinal ***b*** das células da primeira coluna do multiplicador. Por exemplo, o sétimo bit da ***entradaA*** é propagado para todas as células da primeira coluna, o sexto bit para todas da segunda coluna, e assim por diante. Ocorre da mesma forma com a ***entradaB***. O sétimo bit é propagado para todas as células da primeira linha, o sexto bit pra todas da segunda linha, etc. O sinal ***y*** da primeira célula permanece sempre conectado em '0', assim como os sinais ***z*** de todas as células da primeira linha e da ultima coluna. Existe ainda, na célula, um sinal de entrada denominado ***fi,*** que permanece constante, utilizado para determinar polinômio primitivo do Campo de Galois que será usado no multiplicador.

A primeira linha reproduz o polinômio ***entradaB(7)·entradaA(x)***. Os coeficientes gerados em ***y'*** são propagados para ***z***, uma coluna à esquerda e uma linha abaixo, deslocando o resultado do produto um bit à esquerda, produzindo o polinômio ***entradaB(7)·x·entradaA(x).*** Esta operação é análoga a um ciclo de deslocamento no LFSR correspondente. Na primeira coluna, as saídas ***y'*** de cada célula são conectados nas entradas ***y*** da célula imediatamente abaixo representando o sinal de *feedback* do LFSR. Isto reduz o termo x^m usando $x^m = fi(0) + fi(1)\cdot x + \ldots + fi(m-1)\cdot x^{m-1}$. O polinômio ***entradaB(6)·entradaA(x)*** é adicionado na segunda linha sendo então produzido outro polinômio: ***[entradaB(7)·x·entradaA(7) + entradaB(6)·entradaA(x)] mod fi(x)***. Isto é feito linha após linha até que o produto ***saída(x) = entradaA(x)·entradaB(x) mod fi(x)*** seja propagado para os sinais ***y'*** de todas as células da última linha.

4.2 Codificador

A Figura 8 mostra o esquemático do codificador desenvolvido neste trabalho. Por motivos de espaço não foi possível representar todos os registradores e multiplicadores utilizados no cálculo dos símbolos de paridade, mas como podemos ver, a estrutura é bastante similar com a arquitetura apresentada na Figura 5.

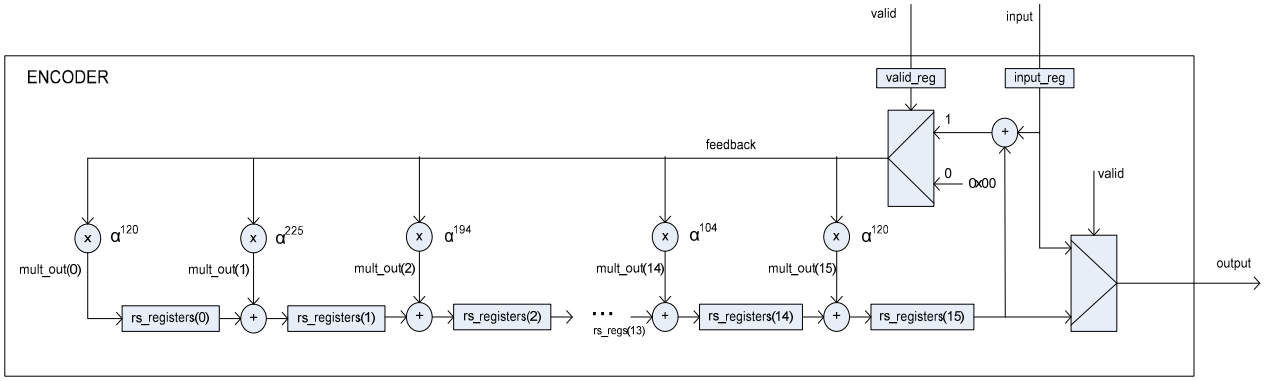


Figura 8 – Arquitetura do codificador desenvolvido.

No caso específico da recomendação G.709, são necessários 16 registradores de 8 bits para armazenar os valores temporários dos símbolos de paridade e por conseqüência, 16 multiplicadores são utilizados para realizar o cálculo. É importante observar que as multiplicações realizadas nesta etapa sempre são por constantes, que equivalem aos coeficientes do polinômio gerador da recomendação G.709. Os coeficientes do polinômio gerador, derivados da equação (11), são apresentados na equação (41).

$$\begin{aligned} g(X) &= x^{16} + \alpha^{120}x^{15} + \alpha^{104}x^{14} + \alpha^{107}x^{13} + \alpha^{109}x^{12} + \alpha^{102}x^{11} \\ &+ \alpha^{161}x^{10} + \alpha^{76}x^{9} + \alpha^{3}x^{8} + \alpha^{91}x^{7} + \alpha^{191}x^{6} + \alpha^{147}x^{5} \\ &+ \alpha^{169}x^{4} + \alpha^{182}x^{3} + \alpha^{194}x^{2} + \alpha^{225}x + \alpha^{120} \end{aligned} \tag{41}$$

A forma de onda apresentada na Figura 9 mostra o início do funcionamento do codificador.

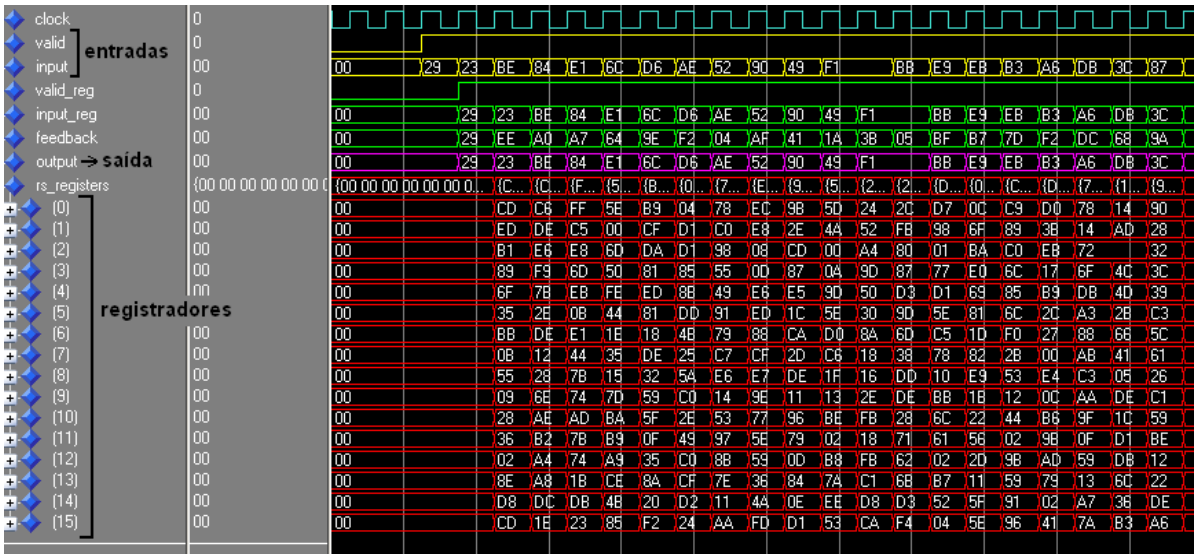


Figura 9 – Início do processo de codificação.

O codificador possui duas entradas principais: *valid* e *input*. O sinal *valid*, quando em nível lógico ‘1’, indica que os dados de 8 bits presentes no sinal *input* são dados válidos e devem ser utilizados para o cálculo de paridade. As duas entradas são armazenadas nos registradores *valid_reg* e *input_reg*. Este armazenamento sincroniza as entradas aos registradores (*rs_registers*) evitando-se um caminho combinacional entre a entrada *input* com os multiplicadores. O registrador *rs_registers* é subdividido em 16 registradores de 8 bits que armazenam os valores temporários para o cálculo dos símbolos paridade. Cada registrador é alimentado pela soma de seu registrador vizinho com a multiplicação do sinal *feedback* pelo respectivo coeficiente do polinômio gerador. Durante o recebimento dos dados válidos, enquanto o sinal *valid* estiver em ‘1’, o codificador apenas repassa os dados da entrada diretamente para a saída do circuito através do sinal *output*.

Após 239 ciclos de clock com o sinal *valid* em ‘1’, os 16 bytes de paridade correspondentes ao pacote recebido encontram-se armazenados em *rs_registers*. Nos 16 ciclos de clock seguintes estes dados são enviados para a saída como mostra a forma de onda apresentada na Figura 10.

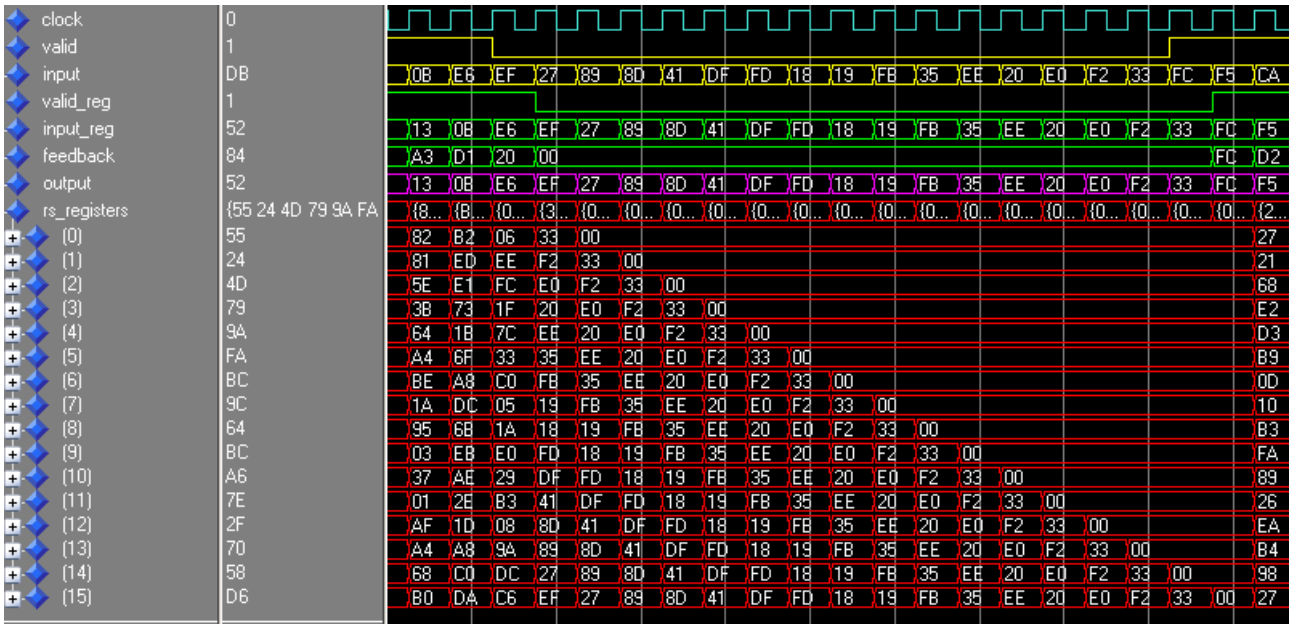


Figura 10 – Envio dos símbolos de paridade.

Como podemos observar na Figura 10, imediatamente após a transição de *valid* para ‘0’, o sinal *feedback* passa a ser “00” e a saída recebe o último byte de *rs_registers*, que é deslocado a cada ciclo de clock, de maneira que todos os símbolos contidos nos registradores são enviados para a saída e os registradores do módulo são zerados.

4.3 Decodificador

Como vimos no capítulo anterior, o processo de decodificação Reed-Solomon é muito mais complexo que a codificação, sendo realizado em diversas etapas. Por este motivo, o circuito

responsável pela decodificação possui diversos sub-módulos, e diversas implementações possíveis de realizar. Uma arquitetura proposta em [8] possui basicamente as mesmas características da arquitetura alvo deste trabalho, sendo utilizada como base para a comparação de outras arquiteturas.

Basicamente o decodificador possui três blocos principais, são eles: a unidade de cálculo das síndromes, o solucionador da Equação Chave, que relaciona as síndromes aos polinômios localizador e avaliador de erros, o algoritmo de busca das localizações e dos valores de erros.

A unidade de cálculo da síndrome é responsável por verificar se as raízes do polinômio gerador são as mesmas raízes da mensagem recebida, vista como um polinômio. No caso do RS(255,239) a mensagem recebida é vista como um polinômio de grau 255, e se não tiver erros, as raízes deste polinômio são as mesmas do polinômio gerador. No caso de uma mensagem recebida com erros, as raízes do polinômio gerador, aplicadas como entrada da mensagem recebida, geram um conjunto não nulo de síndromes, que correspondem ao padrão de erro inserido na mensagem e podem ser relacionadas com o polinômio localizador de erros e com o polinômio avaliador de erros através da seguinte equação:

$$\Lambda(X)S(X) = \Omega(X) \bmod x^{2t} \tag{42}$$

Onde S(X) representa o conjunto de síndromes, $\Lambda(X)$ representa o polinômio localizador de erros e $\Omega(X)$, o polinômio avaliador de erros.

A equação (42) é normalmente denominada pela literatura de Equação Chave, pois ela é a etapa principal e mais complexa etapa do processo de decodificação. Diversos métodos de implementação em hardware para resolvê-la foram propostas. O algoritmo de Berlekamp-Massey e o algoritmo Euclideano são os mais eficientes para resolver este problema [9]. O algoritmo Euclideano é mais simples que o algoritmo de Berlekamp-Massey e possui estrutura mais regular e escalável, entretanto, necessita mais blocos lógicos para ser implementado devido ao emprego de divisão polinomial [8]. O algoritmo original de Berlekamp-Massey não tem mais sido utilizado em implementações atuais devido a sua estrutura irregular e por possuir um caminho crítico mais longo [10], o que não possibilita freqüências de clock muito elevadas. Algumas modificações no algoritmo original de Berlekamp-Massey permitem estruturas mais regulares e caminho crítico mais curto, porém utilizam mais multiplicadores de campo finito [11], o que acaba consumindo mais blocos lógicos.

Em [12] e [13] são propostas arquiteturas para implementar o algoritmo original de Berlekamp-Massey. Essas implementações utilizam um módulo de divisão de campos finitos, composto por um módulo inversor e um multiplicador. Já nas implementações propostas em [10], [11], [14] e [15], são utilizados algoritmos modificados do Berlekamp-Massey, sem o uso de inversores, porém utilizam $3t$ multiplicadores contra $2t$ do algoritmo original. As implementações propostas em [9] e [16] utilizam variações do algoritmo Euclideano para resolver a Equação Chave.

Na última etapa da decodificação, os polinômios obtidos através da solução da Equação Chave são utilizados para determinar as localizações e os valores dos erros através de dois algoritmos, denominados *Chien search* e *Forney*. A saída do último módulo possui o padrão de erro inserido na mensagem recebida, desde que esta mensagem não possua mais de *t* erros. Para se obter a mensagem decodificada é necessário armazenar a mensagem recebida até que o decodificador processe esta mensagem e gere o padrão de erro, para isso é necessário utilizar uma memória organizada em estrutura de fila. O tamanho desta memória depende do tempo de processamento dos componentes do decodificador.

A Figura 11 mostra o diagrama de blocos do decodificador desenvolvido.

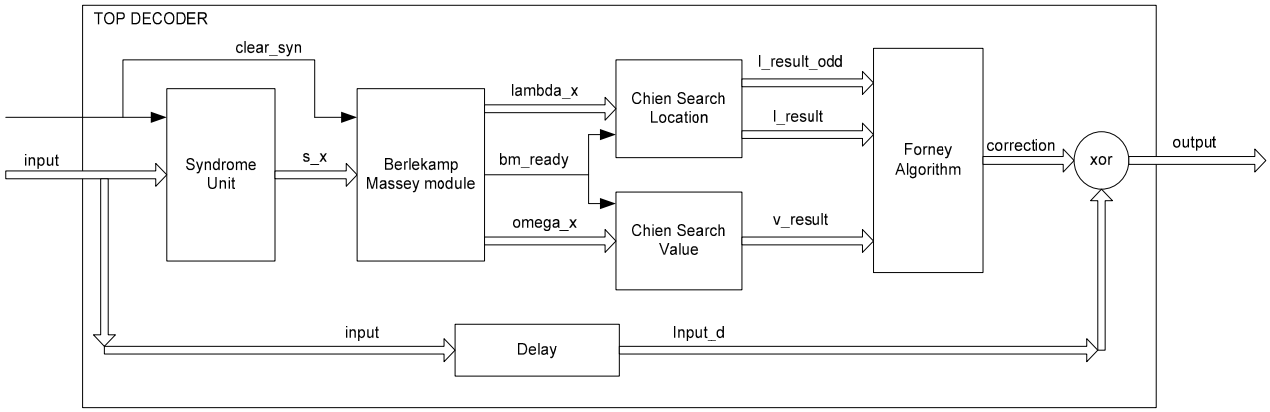


Figura 11 – Arquitetura do decodificador RS(255,239) desenvolvido.

As seções a seguir detalharão as estruturas internas de cada bloco desenvolvido.

4.3.1 Unidade de Cálculo da Síndrome

Por definição um vetor de síndromes para o RS(255,239) é calculado da seguinte maneira:

$$S(X) = \sum_{i=0}^{15} S_i \cdot x^i$$
$$S_i = \sum_{j=0}^{254} r_j \cdot \alpha^{j \cdot i} \qquad (43)$$

Neste trabalho, o hardware utilizado para calcular o vetor de síndromes possui 16 registradores que armazenam os valores intermediários do cálculo. Cada registrador é realimentado pela soma da entrada do circuito com a multiplicação de seu próprio valor por uma das raízes do polinômio gerador. Dessa maneira, o cálculo da síndrome acontece de forma recursiva segundo a equação abaixo:

$$r(X) = r_{254}x^{254} + r_{253}x^{253} + ... + r_1x^1 + r_0$$
$$S_i = (...((r_{254}\alpha^i + r_{253})\alpha^i + r_{252})\alpha^i + ...r_1)\alpha^i + r_0 \qquad (44)$$

Onde $r(X)$ representa a mensagem recebida, r_i representa um dos símbolos da mensagem e S_i, um dos valores da síndrome. O recebimento da mensagem acontece do coeficiente de grau maior até o coeficiente de grau menor. Assim, a síndrome leva n ciclos de clock para ser calculada, neste caso 255, pois o vetor de síndromes só estará pronto após o recebimento de r_0. É importante observar que o cálculo da síndrome através da equação (43) é equivalente ao cálculo pela equação (44), sendo que a última é mais natural para implementação em hardware, pois possui estrutura regular e disponibiliza o vetor de síndromes sem latência adicional ao final do recebimento de uma mensagem.

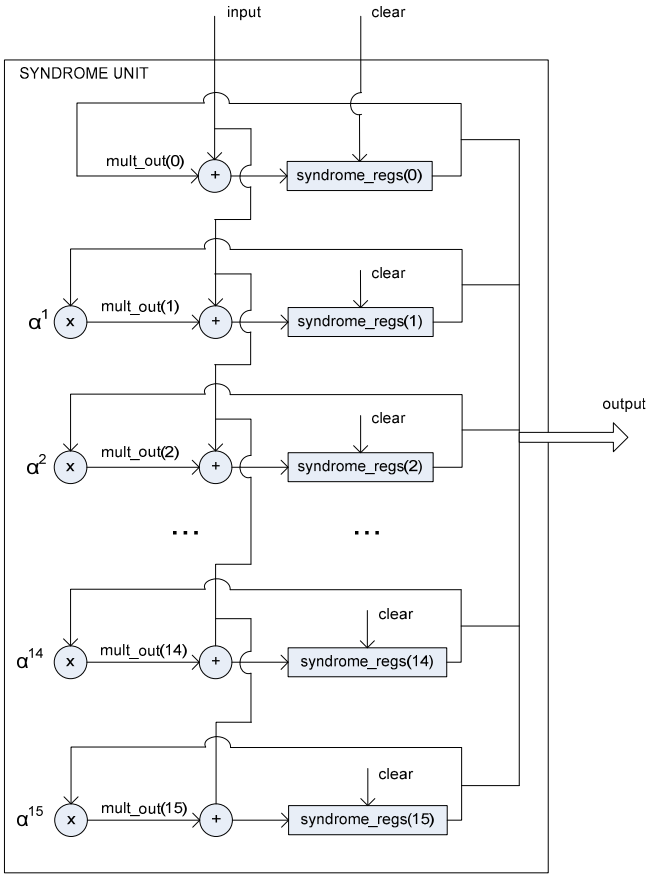


Figura 12 – Arquitetura do bloco desenvolvido para o cálculo da síndrome.

Novamente por motivos de espaço não foi possível representar todos os registradores do módulo, porém, através da Figura 12 é possível observar que a sua estrutura é regular. O módulo possui basicamente duas entradas, uma entrada de 8 bits que recebe serialmente os símbolos da mensagem recebida, e um sinal denominado *clear*, que deve ficar um ciclo de clock ativo para limpar os registradores do módulo e indicar o início de uma nova mensagem. A saída do módulo é o vetor de síndromes, que possui 16 sinais de 8 bits, sendo que cada um destes sinais representa uma síndrome. O sinal *clear* deve ser gerado pelo *framer* com um intervalo de no mínimo 255 ciclos de clock entre dois pulsos, caso contrário o decodificador não funcionará corretamente.

O decodificador Reed-Solomon é organizado em forma de um *pipeline* e a unidade de cálculo da síndrome é seu primeiro estágio. Portanto, é necessário armazenar as síndromes

calculadas a cada início de mensagem, caso contrário seu valor será perdido. O vetor de síndromes é armazenado pelo módulo seguinte, Berlekamp-Massey, que utiliza este vetor para realizar sua operação. A forma de onda apresentada na figura a seguir ilustra o funcionamento do módulo de cálculo da síndrome.

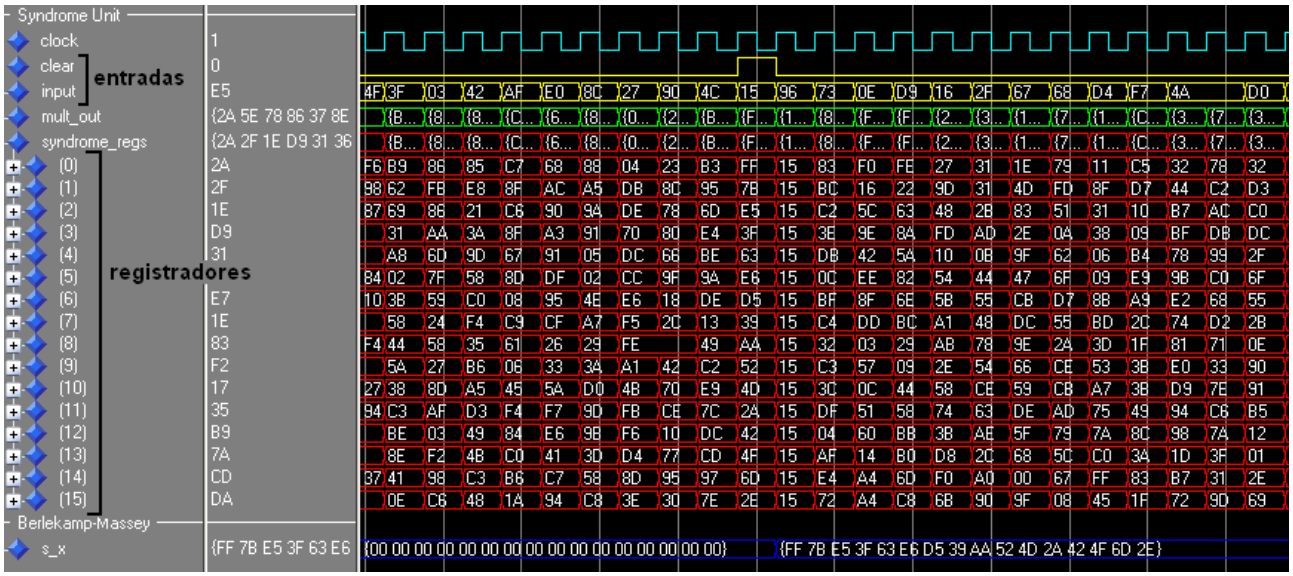


Figura 13 – Funcionamento do módulo de cálculo da síndrome.

Esta figura mostra o término do cálculo do vetor de paridade de uma mensagem e o início do cálculo para outra mensagem. O sinal *input* fica ativo durante o envio do primeiro byte da segunda mensagem. Neste instante, os valores presentes nos registradores do módulo, representados contêm a síndrome correta da primeira mensagem recebida, sendo os demais valores, apenas resultados intermediários. No ciclo de clock seguinte, todos os registradores recebem o valor de *input* para iniciar o cálculo da síndrome para a próxima mensagem. A partir desta forma de onda é possível concluir que a primeira mensagem recebida possui erros, pois o vetor de síndromes calculado é diferente de zero.

Para facilitar o entendimento da estrutura de *pipeline*, também inserimos na Figura 13 o sinal interno do módulo Berlekamp-Massey responsável pelo armazenamento da síndrome. Podemos ver que este vetor recebe os valores presentes nos registradores das síndromes no momento em que o sinal *clear* está ativo. Como podemos ver, os valores do vetor de síndromes armazenados pelo módulo Berlekamp-Massey são todos diferentes de zero, o que indica a presença de erros na mensagem recebida.

4.3.2 Módulo Berlekamp-Massey

Dada à necessidade de replicação do hardware, escolheu-se utilizar o algoritmo de Berlekamp-Massey para realizar a solução da equação chave, pois este é o que consome menos área e é o mais utilizado na literatura estudada. A única desvantagem que deverá ser analisada é a

questão da freqüência de operação que se obterá com este circuito, pois seu caminho crítico é mais longo se comparado a implementações baseadas no algoritmo Euclideano.

4.3.2.1 O Algoritmo de Berlekamp-Massey

O algoritmo de Berlekamp-Massey é um método iterativo que funciona a partir da identidade de Newton, que relaciona a síndrome com os coeficientes do polinômio localizador de erros, apresentada na equação a seguir:

$$S_j = \sum_{i=1}^{v} \Lambda_i S_{j-i} \quad j = v+1, v+2, ..., 2t$$

Esta equação descreve os coeficientes do polinômio localizador de erros de menor grau possível, dado um conjunto de síndromes. Para que esta fórmula funcione, é preciso encontrar os coeficientes Λ_j que gerem a seqüência já conhecida de síndromes $S_1,S_1,...S_{2t}$ [1]. O grau do polinômio localizador de erros deve ser equivalente ao número de erros, e seus coeficientes devem ser condizentes com as síndromes observadas.

Através do algoritmo de Berlekamp-Massey, encontram-se os coeficientes do polinômio localizador de erros que produzem a seqüência completa de síndromes $\{S_1,S_2,...,S_{2t}\}$ por sucessivas modificações em um polinômio existente, se necessário, produzindo seqüências cada vez mais longas. Inicia-se com um polinômio que pode produzir apenas S_1 e verifica-se se este polinômio é capaz de produzir {S1,S2}, caso seja capaz, não é necessário modificá-lo. Se o polinômio que produz S_1, não pode produzir uma seqüência mais longa, é necessário determinar os novos coeficientes que produzirão esta seqüência. Procedendo-se desta maneira, determina-se um polinômio que é capaz de produzir a seqüência $\{S_1,S_2,...,S_{k-1}\}$ e determina-se se este polinômio também é capaz de produzir $\{S_1,S_2,...,S_k\}$. A cada estágio, as modificações são feitas no polinômio localizador de erros de maneira que ele tenha o menor grau possível, de maneira que ao final do algoritmo teremos o polinômio de menor grau capaz se gerar a seqüência completa de síndromes.

Para construir o polinômio localizador de erros a partir de cálculos sucessivos, é necessário definir uma notação para representar $\Lambda(X)$ em diferentes estágios do algoritmo. Definindo L_k como o tamanho do polinômio produzido no estágio K, podemos ter a seguinte expressão:

$$\Lambda^k(X) = 1 + \Lambda_1^k x + ... + \Lambda_{L_k}^k x^{L_k} \tag{45}$$

Este polinômio é denominado polinômio de conexão no estágio k, e indica o polinômio capaz de gerar a seqüência $\{S_1,S_2,...,S_k\}$.

Supondo um polinômio de conexão no estágio k-1, denotado por $\Lambda^{k-1}(X)$, capaz de produzir a seqüência $\{S_1,S_2,...,S_{k-1}\}$. Verifica-se se este polinômio também é capaz de produzir S_k da seguinte maneira:

$$\hat{S}_k = \sum_{i=1}^{L_k} \Lambda_i^{k-1} \cdot S_{k-i} \tag{46}$$

Caso $\hat{S}_k$ seja igual à S_k, não é necessário atualizar o polinômio $\Lambda^{k-1}(X)$, que passa a ser $\Lambda^k(X)$, e $L_k=L_{k-1}$. Caso contrário, existe uma discrepância associada com $\Lambda^{k-1}(X)$, que é calculada através da equação a seguir:

$$d_k = S_k - \hat{S}_k = S_k + \sum_{i=1}^{L_{k-1}} \Lambda_i^{k-1} \cdot S_{k-i} = \sum_{i=0}^{L_{k-1}} \Lambda_i^{k-1} \cdot S_{k-i} \tag{47}$$

Neste caso atualiza-se o polinômio de conexão usando a seguinte fórmula:

$$\Lambda^k(X) = \Lambda^{k-1}(X) + A \cdot x^l \Lambda^{m-1}(X) \tag{48}$$

Onde A é um elemento do campo, que será definido a seguir, e $\Lambda^{m-1}(X)$ é o último polinômio de conexão associado a uma discrepância não nula, antes de ser alterado.

Utilizando o novo polinômio de conexão, é possível calcular uma nova discrepância da seguinte maneira:

$$\begin{aligned} d_k^{'} &= \sum_{i=0}^{L_k} \Lambda_i^k \cdot S_{k-i} \\ &= \sum_{i=0}^{L_{k-1}} \Lambda_i^{k-1} \cdot S_{k-i} + A \cdot \sum_{i=0}^{L_{m-1}} \Lambda_i^{m-1} \cdot S_{k-i-l} \end{aligned} \tag{49}$$

Sendo $l=k-m$, ou seja, a diferença entre o estágio atual e último estágio que o polinômio de conexão foi modificado, e relacionado a equação (49) com a definição de discrepância apresentada na equação (47), temos a seguinte expressão:

$$A \cdot \sum_{i=0}^{L_{m-1}} \Lambda_i^{m-1} \cdot S_{m-i} = A \cdot d_m \tag{50}$$

Se $A=-d_m^{-1}.d_k$, resumindo a equação (49), obtemos:

$$d_k^{'} = d_k - d_m^{-1} d_k d_m = 0 \tag{51}$$

Logo, o novo polinômio de conexão, calculado a partir de (48), produz a seqüência $\{S_1,S_2,...,S_k\}$ sem discrepância.

A seguir apresentaremos o algoritmo completo de Berlekamp-Massey, em pseudocódigo, conforme descrito em [17].

```
Entradas: S1,S1,...,SN
Inicialização:
L=0 (Grau atual do polinômio de conexão)
c(x)=1 (Polinômio de conexão atual)
p(x)=1 (Polinômio de conexão antes da última alteração)
l=1 (k−m, quantidade de ciclos sem alteração de c(x))
dm=1 (Ultima discrepância antes da alteração)
d=0 (discrepância atual calculada)
for k=1 to N
    d=Sk+ Σ_{i=1}^{L} ci Sk−1  (calcula a discrepância)
    if(d=0) (não altera o polinômio c(x))
        l=l+1
    elsif(2L>=K) then (não muda o grau do polinômio c(x))
        c(x)=C(x)−d.dm^-1.x^l.p(x) (atualiza c(x))
        l=l+1;
    else
        t(x)=c(x)
        c(x)=C(x)−d.dm^-1.x^l.p(x)
        L=k−L
        p(x)=t(x)
        dm=d
        l=1
    end
end
```

4.3.2.2 Implementação do Berlekamp-Massey

Inicialmente implementamos o algoritmo original de Berlekamp-Massey, pois sua área é menor que a implementação do algoritmo modificado. O capítulo a seguir vai analisar os requisitos de área e freqüência para este módulo. A estrutura do módulo Berlekamp-Massey implementado é apresentado na Figura 14.

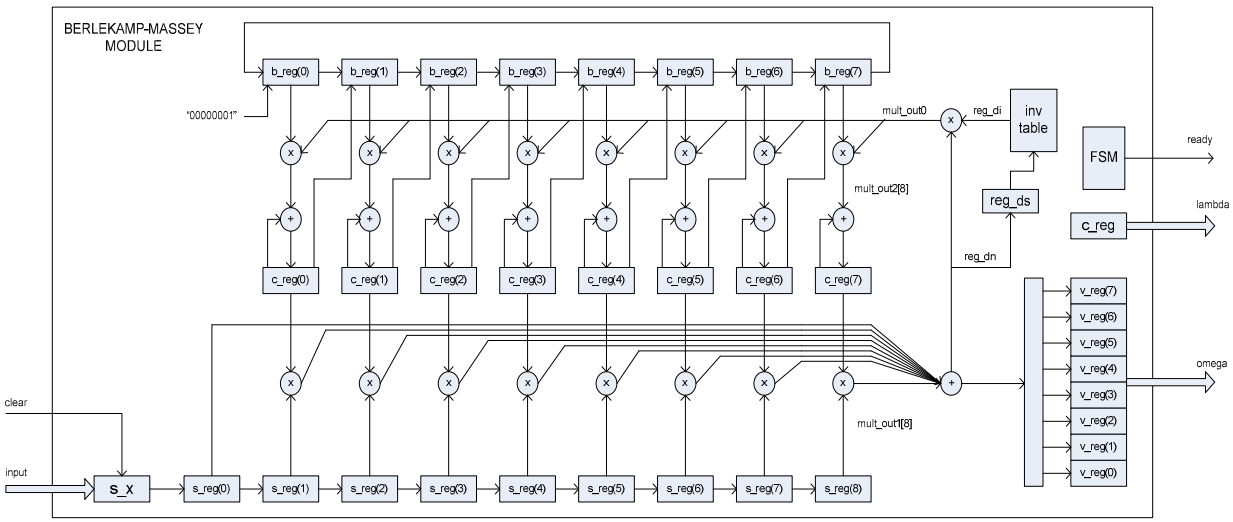


Figura 14 – Implementação do módulo Berlekamp-Massey.

Este bloco representa o segundo estágio do *pipeline* de decodificação e implementa o algoritmo apresentado na seção anterior. A forma de onda a seguir ilustra o funcionamento da parte inferior do módulo desenvolvido.

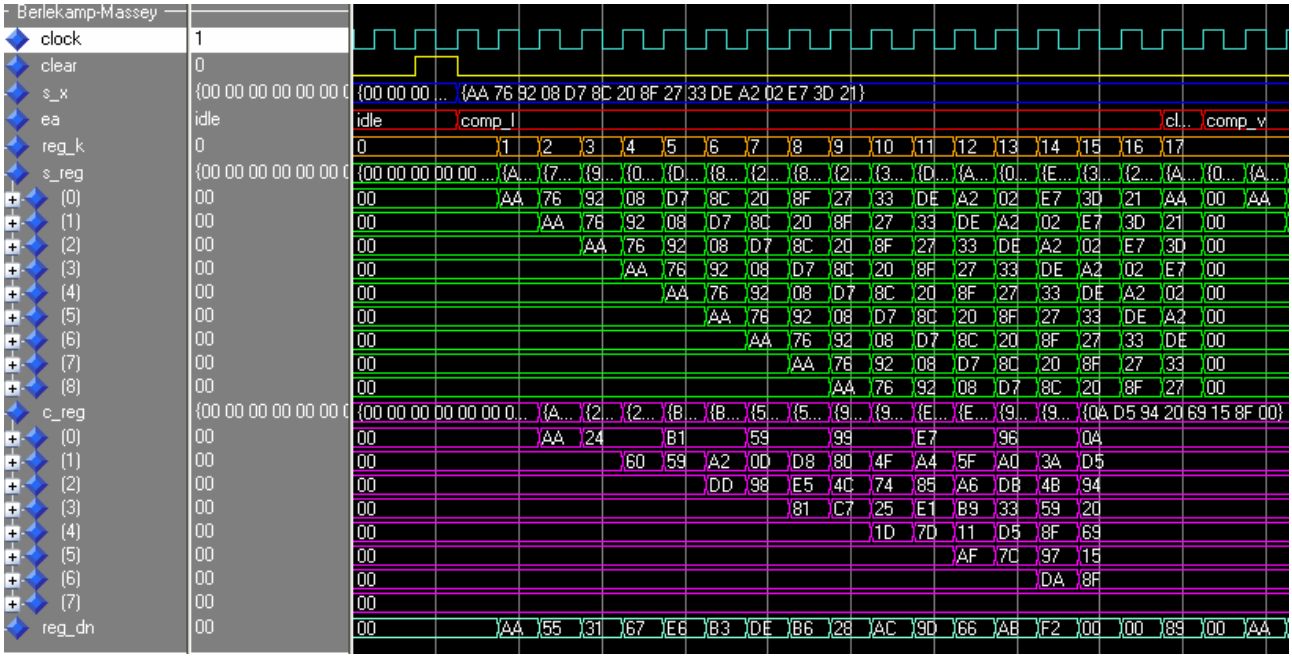


Figura 15 – Funcionamento da parte inferior do módulo Berlekamp-Massey.

O registrador *s_x* é responsável por armazenar todo o vetor de síndromes no momento em que o cálculo estiver concluído, no caso quando *clear* for igual a '1'. O registrador *s_reg*, é subdividido em 9 registradores, sendo que no primeiro ciclo de clock após o recebimento da síndrome, o primeiro registrador recebe o primeiro valor do vetor de síndromes. Nos demais ciclos, este registrador recebe os valores de síndrome de acordo com o estágio do algoritmo, denotado pelo contador *reg_k*. Os demais registradores de *s_reg* recebem sempre o valor de seu antecessor. Esta organização é feita para implementar o cálculo da discrepância, apresentado na equação (47). O registrador *c_reg*, é subdividido em 8 registradores, que representam os coeficientes de x^1 à x^8 do polinômio $\Lambda(X)$, pois o coeficiente de x^0 será sempre 1. Por este motivo, o primeiro registrador de deslocamento das síndromes não é multiplicado por nenhum dos coeficientes de *c_reg*, mas seu valor é lavado em conta no somatório para o cálculo da discrepância, como podemos observar na Figura 14. A discrepância também pode ser observada na Figura 15 através do sinal *reg_dn*. Podemos observar que quando *reg_k* é igual a 15, o resultado da discrepância é zero, e a partir deste momento o polinômio $\Lambda(X)$ não sofre mais alterações. Isto significa que com 7 coeficientes não nulos de $\Lambda(X)$, é possível prever todo vetor de síndromes, e neste caso podemos concluir que as síndromes recebidas correspondem a uma mensagem com 7 erros.

Para continuar a ilustrar o funcionamento do módulo Berlekamp-Massey, apresentaremos uma forma de onda que mostra o funcionamento da parte superior do módulo, incluindo a atualização do polinômio temporário e o cálculo de divisão.

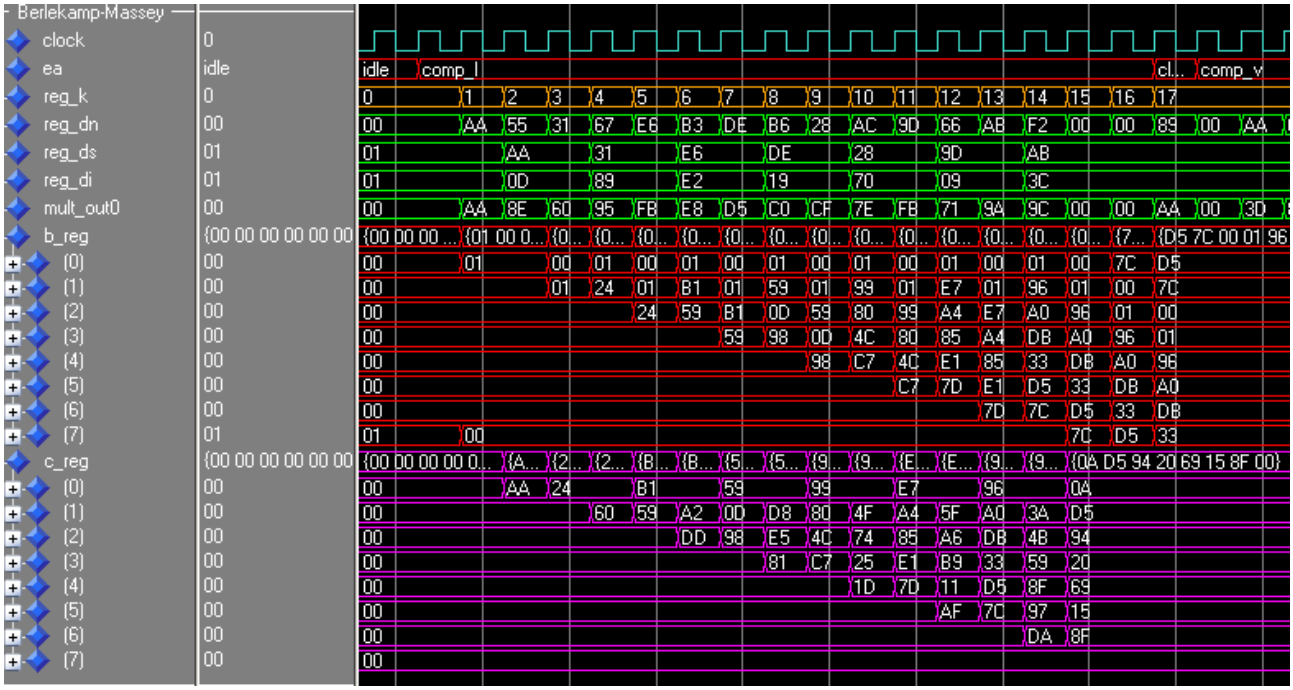


Figura 16 – Funcionamento da parte superior do módulo Berlekamp-Massey.

O sinal *reg_dn*, como vimos anteriormente, é o resultado do cálculo da discrepância. A cada dois ciclos de clock, caso a discrepância seja diferente de zero, o valor de *reg_dn* é armazenado no registrador *reg_ds*. Toda vez que um valor diferente de discrepância é armazenado em *reg_ds*, o grau do polinômio $\Lambda(X)$ é alterado. O sinal *mult_out0* contém o resultado da divisão de *reg_ds* por *reg_dn*, que é implementada através da inversão do valor presente em *reg_ds*, multiplicado por *reg_dn*. O resultado da inversão de *reg_ds* pode ser observado no sinal *reg_di*. Esta operação é feita para determinar a constante A, conforme as equações (50) e (51).

O registrador *b_reg* é subdividido em 8 registradores de 1 byte, sendo atualizado através da rotação de seus valores toda vez que o grau do polinômio $\Lambda(X)$, representado por *c_reg*, não seja alterado. Caso contrário, o registrador *b_reg* recebe os valores de *c_reg*, incluindo a constante 01 que é omitida nesta implementação para economia de recursos. O último registrador de *b_reg* é inicializado em 01 e os demais em zero. Isto acontece porque após o primeiro ciclo, com a rotação dos valores, apenas o primeiro registrador tem o valor 01, e a primeira atualização do registrador *c_reg* deve acontecer em seu primeiro registador. Todos os valores do registrador *c_reg* são inicializados com zero. Este registrador é atualizado através da multiplicação de *b_reg* por *mult_out0*, que representa a divisão de *reg_dn* por *reg_ds*, somado de seu próprio valor, implementando a equação descrita na equação (48).

Quando o contador *reg_k* atingir o valor 16, o cálculo do polinômio localizador de erros estará completo. Uma vez que tenhamos o polinômio localizador de erros, o polinômio avaliador de erros, responsável por indicar os valores dos erros, pode ser calculado facilmente através da equação (42). O mesmo hardware utilizado para calcular $\Lambda(X)$ pode ser reconfigurado para calcular

$\Omega(X)$, para isso utiliza-se uma pequena máquina de estados de controle. A Figura 17 apresenta o diagrama de estados de controle do módulo.

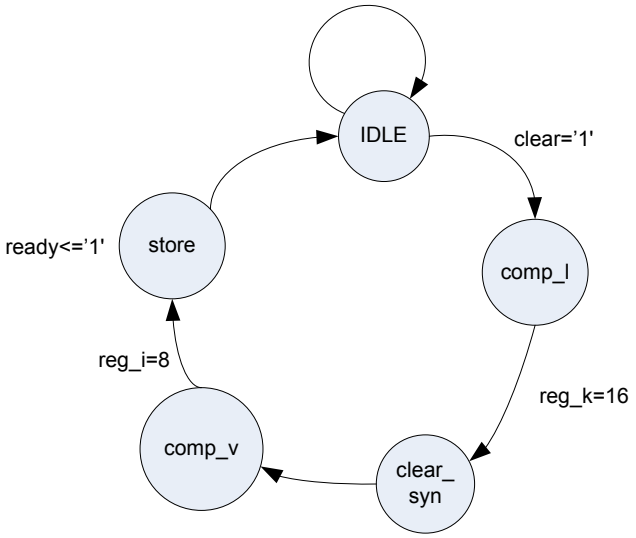


Figura 17 – Máquina de controle do módulo Berlekamp-Massey.

Ao receber um sinal de *clear*, a máquina passa para o estado *comp_l*. Neste estado é realizado o cálculo do polinômio localizador de erros em 16 ciclos de clock. Após este tempo, o estado passa a ser *clear_syn*, onde os registradores de deslocamento da síndromes são zerados para o cálculo do polinômio avaliador de erros. Após o estado *clear_syn*, a máquina passa diretamente para o estado *comp_v*, onde é feito o cálculo do polinômio avaliador de erros, em 8 ciclos. Após este período, a máquina fica um ciclo no estado *store*, onde indica para os módulos seguintes que os polinômios estão calculados.

A figura a seguir mostra como é feito o calculo do polinômio avaliador de erros.

Figura 18 – Cálculo do polinômio avaliador de erros.

Como podemos ver na Figura 18, tendo *c_reg* calculado, basta deslocar as síndromes novamente que a cada ciclo de clock teremos em *reg_dn* um dos coeficientes do polinômio avaliador de erros. Ao final do cálculo dos 8 coeficientes do polinômio gerador, o módulo Berlekamp-Massey gera um sinal de *ready*, alertando aos módulos seguintes a finalização do cálculo dos polinômios.

4.3.3 Chien Search e Forney

Os módulos *Chien search location*, *Chien search value* e *Forney Algorithm* pertencem ao último estágio do processo de decodificação Reed-Solomon e operam paralelamente. Neste estágio, inicia-se o processo de correção de erro.

Uma vez que o polinômio $\Lambda(X)$ está calculado, o módulo *Chien search location* é utilizado para achar suas raízes. O algoritmo *Chien search* avalia um determinado polinômio para todos os 255 elementos não nulos do campo GF(2^8). No caso do polinômio localizador de erro, se $\Lambda(\alpha^{-i})=0$ significa que existe um erro no *i*-ésimo byte recebido. A Figura 19 apresenta a arquitetura do módulo *chien search location*.

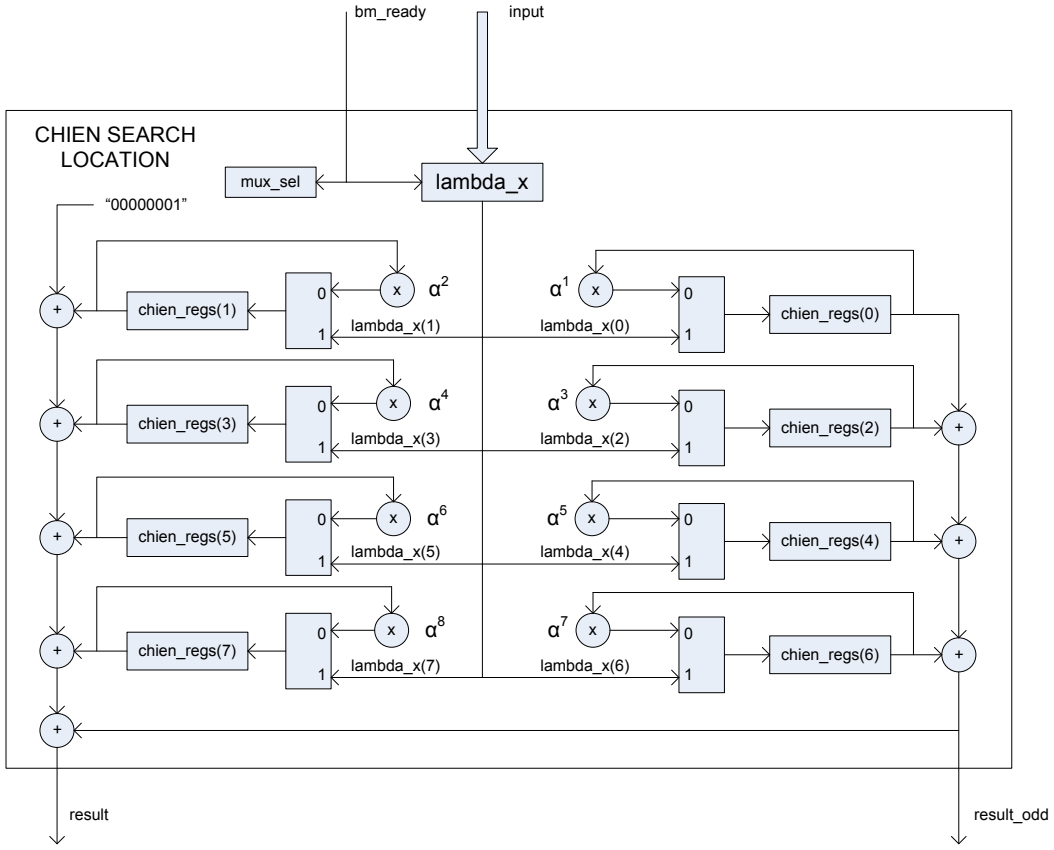


Figura 19 – Arquitetura do módulo Chien Search Location.

Podemos observar que a estrutura deste módulo é bastante regular. Basicamente temos 8 células contendo um registrador, um multiplicador e um multiplexador. Cada célula calcula um

valor intermediário para um dos coeficientes do polinômio $\Lambda(X)$, esses valores são todos somados, inclusive levando em conta a constante 01, que corresponde sempre ao coeficiente de x^0, e o resultado é $\Lambda(\alpha^{-i})$. Cada registrador funciona como um acumulador. Inicialmente os registradores recebem os coeficientes do polinômio localizador de erros. No ciclo seguinte os registradores passam a ser realimentados pelo seu próprio valor multiplicado por α^i, onde *i* representa o grau do coeficiente do polinômio que cada registrador está calculando. Através desta operação é possível calcular $\Lambda(\alpha^0)$, $\Lambda(\alpha^1)$, $\Lambda(\alpha^2)$ e assim sucessivamente, graças a matemática de campos finitos. Tomando como exemplo o oitavo coeficiente do polinômio localizador de erros, denotado por Λ_8, teremos após um ciclo de clock, no registrador *chien_reg(7)* o resultado de $\Lambda_8 \bullet \alpha^8$, que equivale a $\Lambda_8 \bullet x^8$ com $x=\alpha^1$. No ciclo de clock seguinte o conteúdo deste registrador passa a ser $\Lambda_8 \bullet \alpha^8 \bullet \alpha^8$, que é igual a $\Lambda_8 \bullet \alpha^{2 \bullet 8}$, ou seja, $\Lambda_8 \bullet x^8$ quando $x=\alpha^2$. Este é o método utilizado pelo algoritmo *chien search* para avaliar um dado polinômio para todos elementos de um campo finito. Podemos ver também na Figura 19, além da saída *result*, uma outra saída denominada *result_odd*. O valor visualizado nesta saída corresponde apenas à soma dos resultados dos coeficientes de grau impar, e é utilizado para ajudar a encontrar o valor do erro, uma vez que *result* seja zero.

A título de informação apresentaremos também, na Figura 20, a arquitetura do módulo *chien search value*.

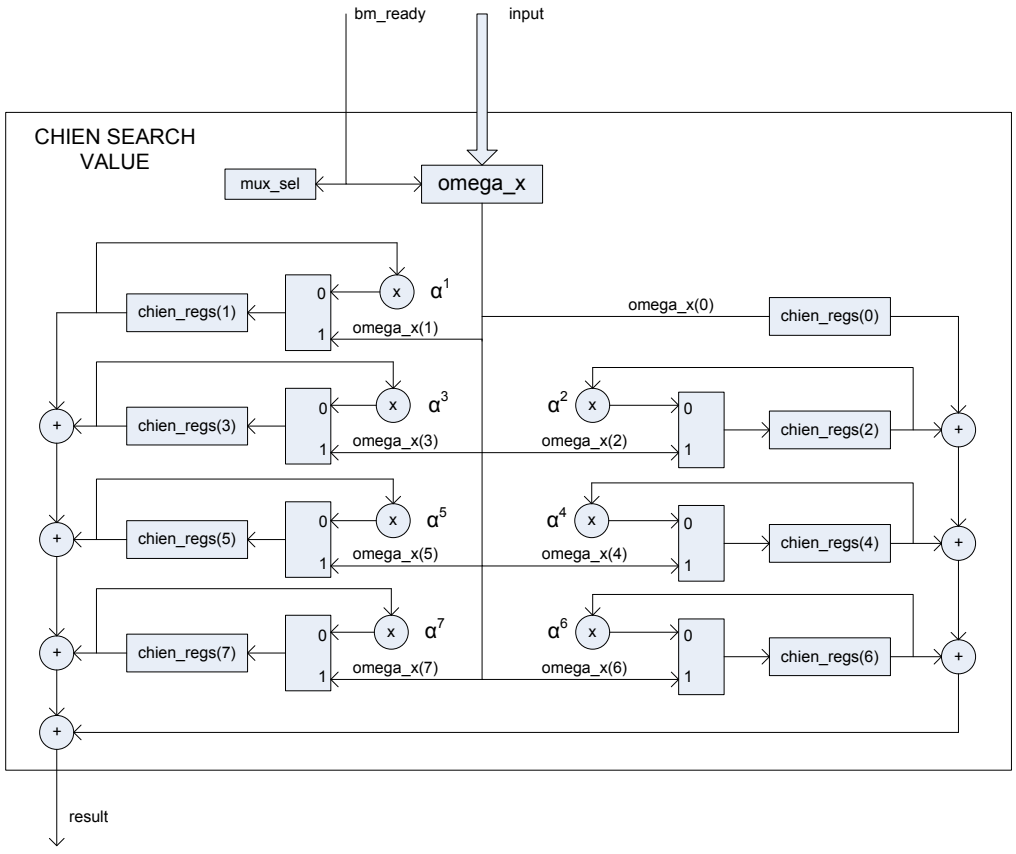


Figura 20 – Arquitetura do módulo Chien Search Value.

Como podemos observar, a estrutura deste módulo é praticamente igual à estrutura do *Chien Search Location*. A primeira diferença é que neste módulo não temos a soma da constante 01 para gerar o resultado. A segunda está no fato de que o primeiro registrador não possui realimentação, pois o primeiro coeficiente no polinômio avaliador de erros corresponde a x^0, que é variável, ao contrario do polinômio localizador de erros. O restante do processo de cálculo dos resultados de $\Omega(\alpha^{-i})$, onde $\Omega(X)$ denomina o polinômio avaliador de erros, ocorre da mesma maneira que o cálculo de $\Lambda(\alpha^{-i})$, conforme descrito anteriormente. É importante observar que o valor calculado por este módulo, visualizado no sinal *result*, não representa o valor do erro no instante em que $\Lambda(\alpha^{-i})=0$. O valor do erro deve ser encontrado através da seguinte expressão:

$$e_i = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})} \tag{52}$$

Pela teoria de campos finitos a seguinte igualdade é válida:

$$\Lambda'(X) = \frac{\Lambda_{odd}(x)}{x} \tag{53}$$

Logo, para achar o valor do erro utiliza-se a equação a seguir:

$$e_i = \frac{\Omega(\alpha^{-i}) \cdot \alpha^{-i}}{\Lambda_{odd}(\alpha^{-i})} \tag{54}$$

Esta equação é implementada pelo módulo *Forney Algorithm*, que recebe os resultados dos módulos *Chien search value* e *Chien search location* e gera o valor de correção para ser adicionado ao pacote recebido, neste caso corrigindo os erros da mensagem recebida. A Figura 21 mostra a estrutura do módulo que implementa esta equação.

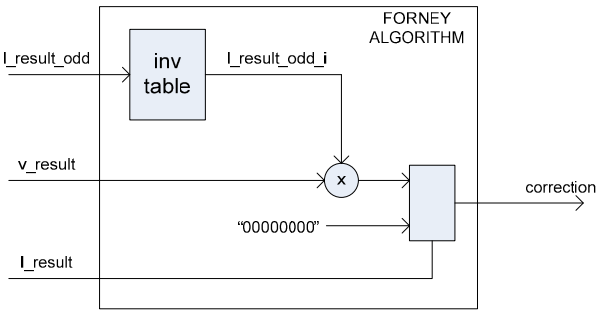


Figura 21 – Estrutura do módulo Forney Algorithm.

Este é basicamente um circuito combinacional que detecta quando o sinal *l_result* é igual a zero e coloca na saída o valor do erro calculado. O valor de erro é calculado pela divisão do sinal *v_result* pelo sinal *l_result_odd*. Esta divisão também é implementada através de um circuito inversor e um multiplicador, a exemplo do cálculo feito no módulo Berlekamp-Massey. Caso o sinal *l_result* seja diferente de zero, significa que não há erros na posição atual da mensagem

recebida, logo a saída do módulo deve ser zero, pois esse valor será sempre somado ao valor atual da mensagem, como pode ser visto na Figura 11.

Na Figura 22 será apresentada uma forma de onda que ilustra o funcionamento dos módulos *Chien search location*, *Chien search value* e *Forney Algorithm*.

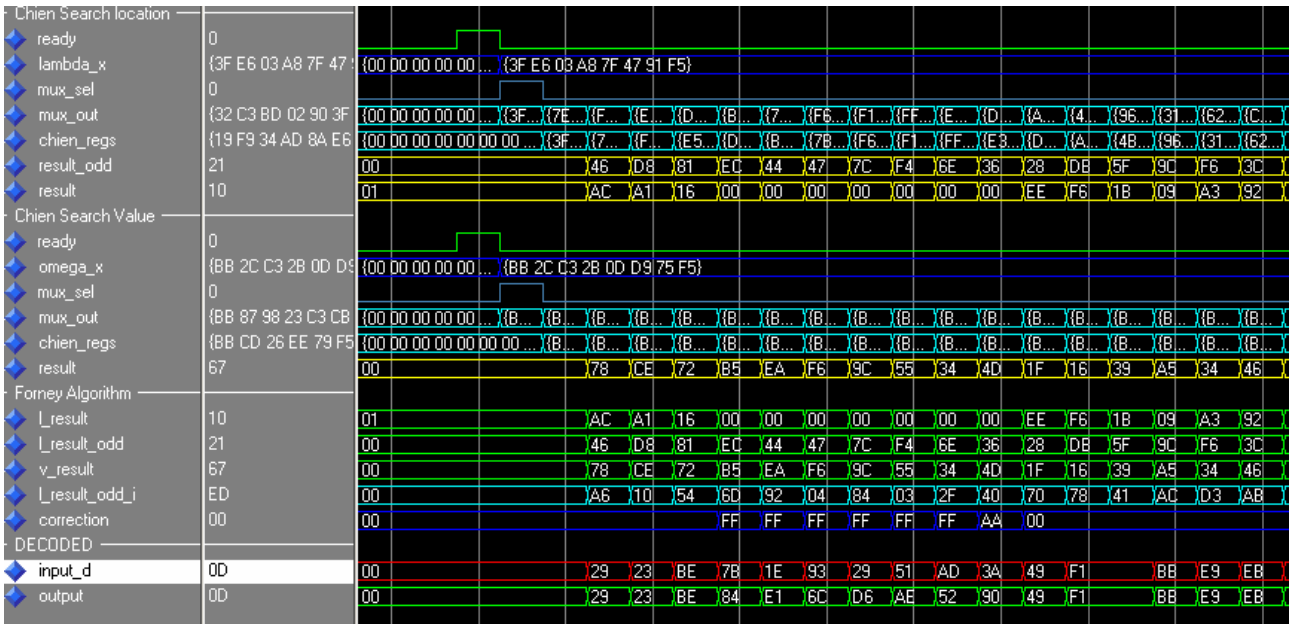


Figura 22 – Funcionamento do último estágio de decodificação.

Ao receber um sinal de *ready* do módulo Berlekamp-Massey, o polinômio $\Lambda(X)$ é armazenado pelo módulo *Chien Search Location* através do sinal *lambda*_x, e o polinômio $\Omega(X)$ é armazenado pelo *Chien Search Value*, através do sinal *omega_x*. Após o armazenamento dos polinômios, o sinal *mux_sel* fica ativo por um ciclo de clock, de maneira que os coeficientes dos polinômios entram nos registradores dos módulos para iniciar o cálculo dos valores de erro. No ciclo seguinte inicia-se o processo de correção de erros. Neste exemplo, os erros foram inseridos no início da mensagem para facilitar a visualização do processo de correção. Podemos observar que a partir do terceiro valor de *l_result*, após o início do cálculo, temos 7 resultados iguais à zero, o que indica que temos uma rajada se 7 erros na mensagem recebida. Os valores dos erros, calculados pelo módulo *Forney Algorithm*, podem ser observados no sinal *correction*.

Para realizar a correção de erros corretamente, a mensagem recebida deve ser armazenada até que os valores de erro comecem a ser calculados. O sinal *input_d* possui a mensagem recebida já com o atraso inserido pelos estágios anteriores do decodificador. Através da soma dos valores de erros encontrados com a mensagem atrasada, se obtém a mensagem decodificada, que é apresentada no sinal *output* e é a saída do decodificador.

4.3.4 Visão Geral do Decodificador

Nas seções anteriores apresentamos detalhadamente o funcionamento de cada um dos módulos que compõem o decodificador. Entretanto, alguns aspectos não são observados quando estudamos os módulos separadamente. Para dar uma idéia mais geral do funcionamento do decodificador, apresentaremos uma forma de onda que ilustra alguns destes aspectos.

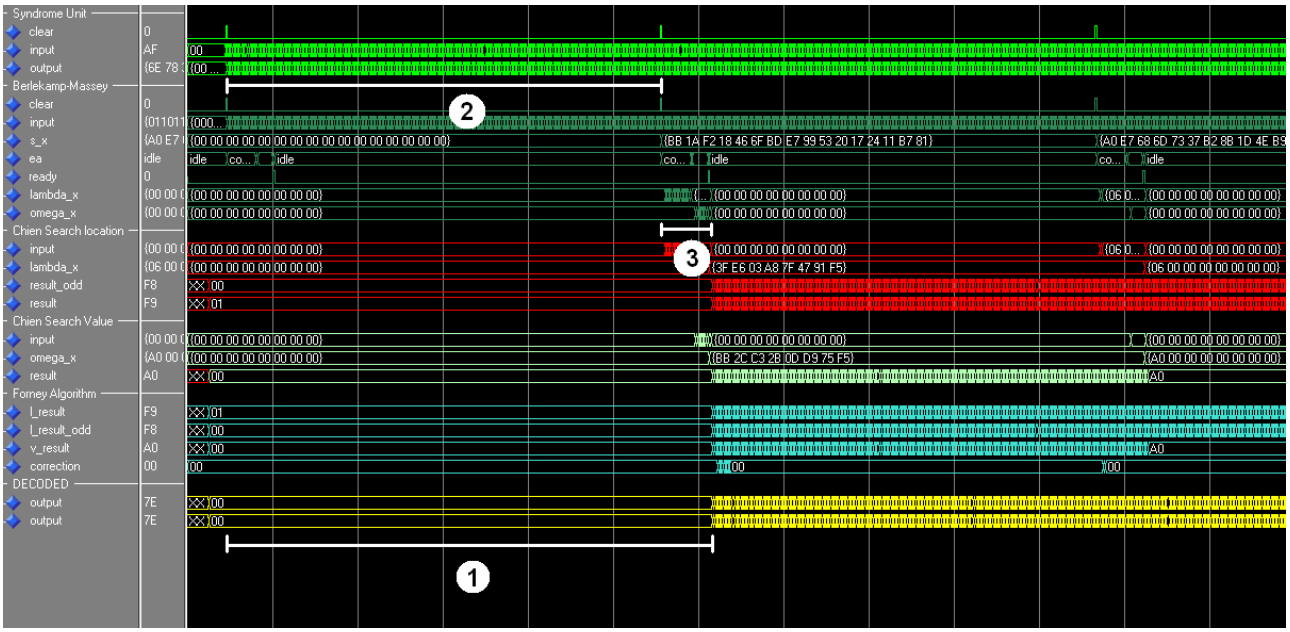


Figura 23 – Visão geral do funcionamento do decodificador.

Podemos observar na Figura 23 as três latências principais do decodificador Reed-Solomon desenvolvido. A latência número 1 corresponde ao tempo total de processamento do decodificador. São computados, ao todo, 286 ciclos de clock da entrada do primeiro byte de uma mensagem até a saída deste byte já decodificado. A latência número 2 corresponde ao tempo de cálculo da síndrome. Este tempo é de 255 ciclos de clock, que correspondem ao recebimento de uma mensagem completa. A terceira latência representada na Figura 23 é relativa ao módulo Berlekamp-Massey, e esta corresponde a 28 ciclos de clock.

Contando a latência de 255 ciclos adicionada pelo cálculo da síndrome, mais os 28 ciclos correspondentes à resolução da equação chave pelo módulo Berlekamp-Massey, e levando em conta o tempo que os módulos seguintes têm para armazenar os polinômios e adicioná-los em seus registradores antes de iniciar o cálculo, chega-se no total de 286 ciclos de latência.

É importante observar que os dados são recebidos sempre em rajada pelo decodificador, não há intervalo entre os pacotes de 255 bytes. Dessa maneira, o módulo que calcula a síndrome está sempre em funcionamento. Como podemos observar, o único módulo que fica ocioso durante o processo de decodificação é o Berlekamp-Massey, pois ele recebe um vetor de síndromes a cada 255 ciclos, e resolve a equação chave em apenas 28 ciclos. Esta situação abre precedentes para

diferentes implementações deste algoritmo, que podem aproveitar esta sobra de tempo para se obter alguma vantagem, em troca de um pequeno aumento na latência total do decodificador.

5 Validação do Codificador e Decodificador

Neste capítulo será reportada a validação do codificador e decodificador Reed-Solomon apresentados neste trabalho. A primeira parte desta validação tem como objetivo verificar a correta funcionalidade dos módulos através da simulação dos mesmos. A segunda parte compreende esta mesma simulação acrescida dos tempos de atraso gerados pela ferramenta de síntese. Isso torna a verificação mais precisa e mais perto da realidade. A terceira e última parte diz respeito à validação em hardware dos referidos módulos.

5.1 Validação Funcional

Para efetuar a simulação dos módulos utilizou-se a ferramenta ModelSim SE Plus de versão 6.1f da empresa Mentor Graphics Corp. Foi criado um *testbench* para implementar uma estrutura capaz de gerar os estímulos necessários para ambos os módulos. Para verificar a funcionalidade dos módulos, recorreu-se à visualização dos seus sinais nas formas de ondas, ou *waveforms*, geradas pelo software de simulação. Na Figura 24 pode-se observar de forma sucinta a estrutura criada para a simulação.

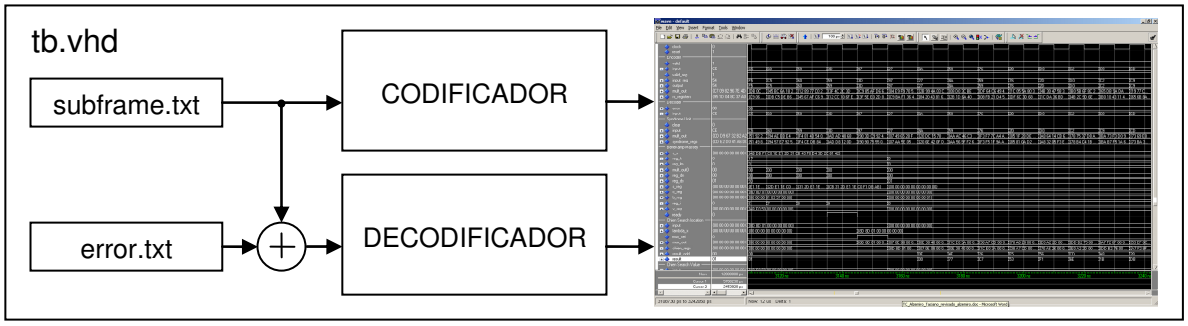


Figura 24 – Estrutura de simulação do codificador e decodificador.

Esta estrutura gera os sinais de *reset, clock, valid, clear_syn* além do sinal *input.* Para enviar os dados aos módulos, recorreu-se ao uso de dois arquivos texto: “subframe.txt”, que contém diversos sub-frames do padrão G.709 OTN, e “error.txt” que é utilizado para permitir a inserção de erros em quaisquer posições dos sub-frames. Tais arquivos possuem o mesmo formato: dois caracteres em hexadecimal por linha. Ou seja, é lido e enviado aos dois módulos um byte por ciclo de relógio. Na Figura 25 apresenta-se o formato dos dois arquivos texto. Esses arquivos foram gerados através de um software, desenvolvido no contexto deste trabalho, denominado fecGEN. Este software foi implementado pelos autores e realiza o cálculo da paridade a partir da geração aleatória de sub-frames. Os resultados obtidos nesta ferramenta foram validados a partir da comparação com ferramentas disponibilizadas na internet, tais como o *schifra* [18] e a biblioteca de FEC desenvolvida por Phil Karn [19].

Através do seu sinal *input*, são enviados ao codificador os 239 bytes referentes a um sub-frame sem o campo de FEC. Após a finalização do envio destes bytes, verifica-se a validade do novo FEC gerado. Isto é feito comparando-o com o FEC gerado pelo software fecGEN. Na Figura 26 verifica-se a validade do FEC gerado pelo codificador.

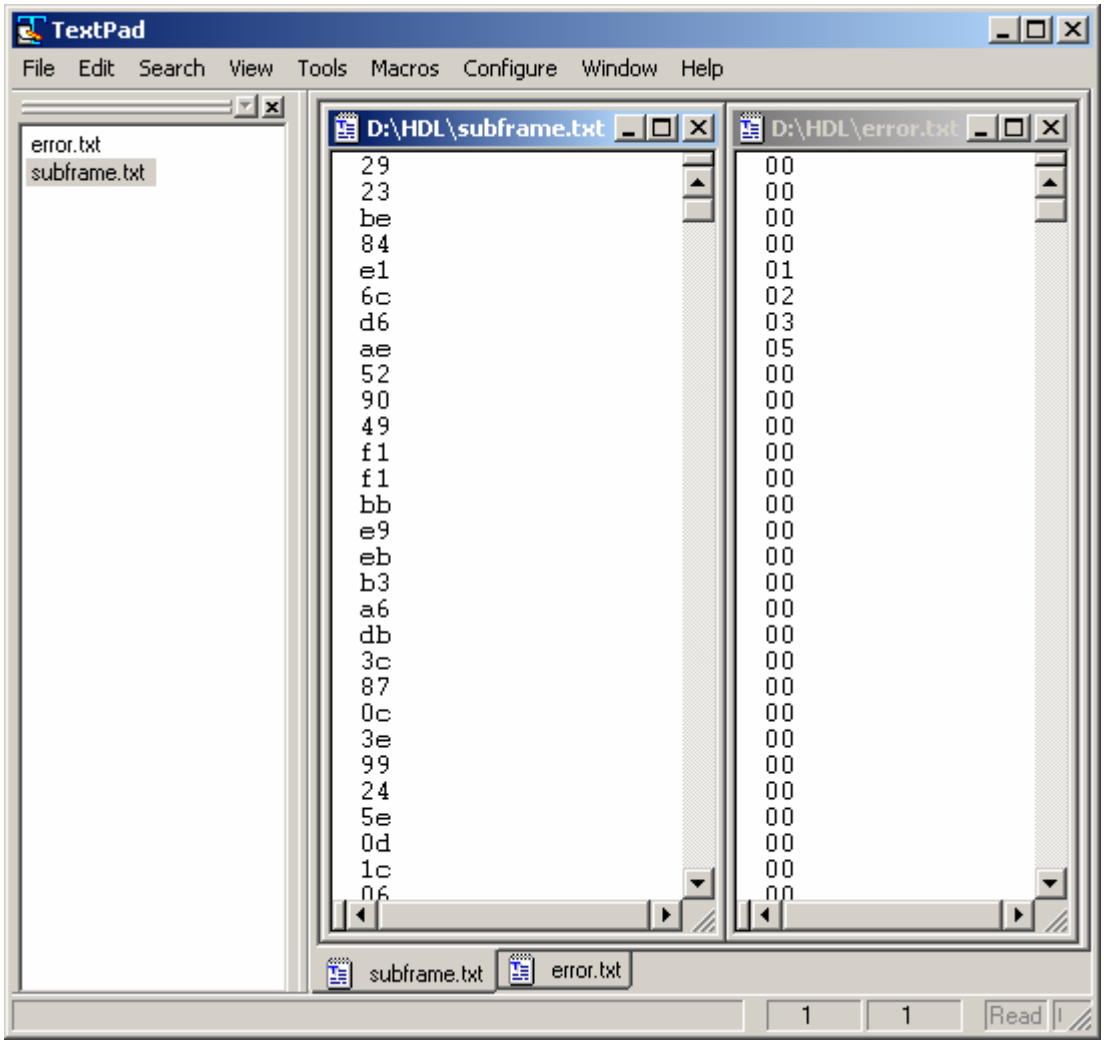


Figura 25 – Arquivos texto subframe.txt e error.txt.

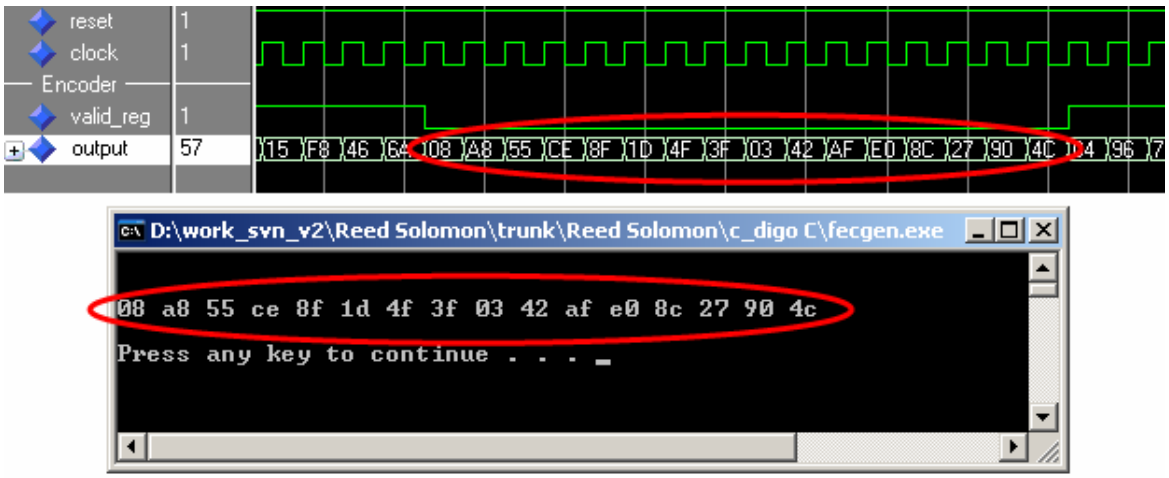


Figura 26 – Verificação do FEC gerado pelo codificador.

Ao decodificador são enviados 255 bytes de um sub-frame completo. No sinal *input* do decodificador é implementada uma porta lógica OU exclusivo que recebe os bytes dos dois arquivos texto possibilitando a inclusão de erros. A validação ocorre verificando se tais erros são corrigidos no sinal de saída do decodificador. Na Figura 27 encontramos a forma de onda demonstrando a correção dos erros adicionados em um sub-frame.

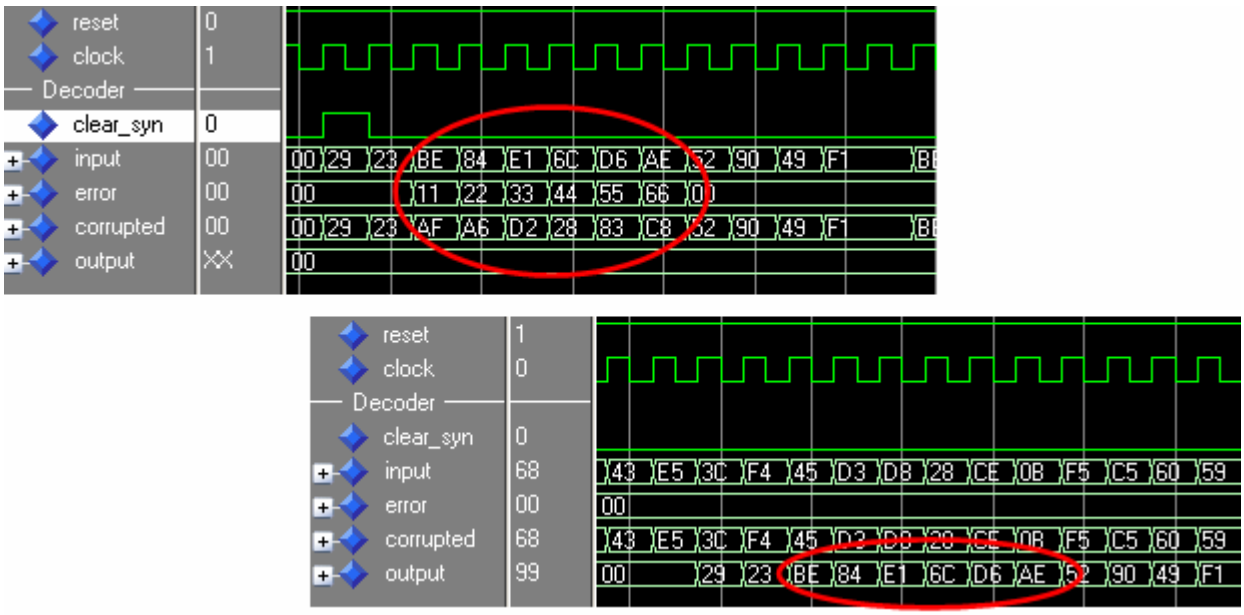


Figura 27 – Erros do sub-frame corrigidos no decodificador.

Os módulos não foram testados exaustivamente, porém, foram feitos diversos testes para verificar o funcionamento dos mesmos em seus limites. Além de erros adicionados em posições aleatórias, foram inseridos erros em rajada no inicio e no fim dos sub-frames, onde a possibilidade de falhas é mais elevada. Também realizou-se testes com apenas um erro por sub-frame em locais distintos e testes com mais de um erro, incluindo a presença de 8 erros, que é a capacidade máxima de correção da arquitetura desenvolvida. Após a finalização dos módulos, nenhuma falha foi verificada nos testes realizados.

5.2 Validação Funcional com Tempos de Atraso

A ferramenta ISE de versão 9.1i da Xilinx Inc. permite, após a síntese física, criar os dois arquivos necessários para a simulação com tempos de atraso. Um dos arquivos, de nome "timesim.vhd", contém o mapeamento dos blocos lógicos, que são utilizados para a implementação do codificador e decodificador no FPGA, incluindo informações de posicionamento e roteamento. O outro arquivo, de nome "timesim.sdf", contém os valores dos tempos de atraso de todos os blocos lógicos utilizados no "timesim.vhd". Estes dois arquivos são utilizados em uma nova simulação na ferramenta Modelsim, permitindo uma verificação mais acurada dos módulos. Toda a estrutura de teste e modo de validação permanecem os mesmos da simulação anterior. Na Figura 28 podemos

observar, a título de exemplo, uma simulação do codificador com todos os atrasos inseridos pelas portas e pelos fios utilizados na síntese realizada.

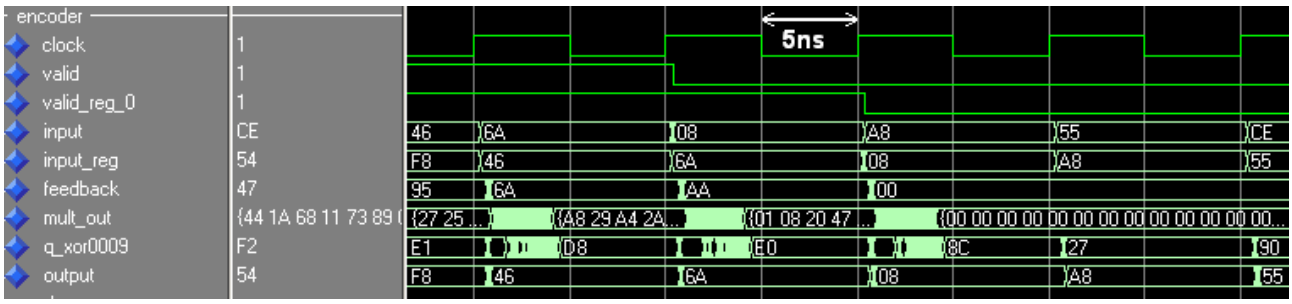


Figura 28 – Simulação com atraso do codificador RS no dispositivo XCV4FX100-10.

Nesta simulação utilizamos um clock de 100MHz para ilustrar o tempo de cálculo dos valores intermediários de paridade. Podemos ver a transição do sinal de entrada *valid* e respectivamente, um ciclo de clock após, a transição do sinal interno *valid_reg*, indicando o fim do cálculo de paridade. A partir desta transição não há mais atrasos por cálculos e a paridade calculada é enviada para a saída. Como podemos ver, existem linhas verticais na figura que indicam intervalos de 5 nanossegundos, sendo que estas coincidem com as transições do clock.

O processamento de todos os multiplicadores presentes na arquitetura pode ser observado no sinal *mult_out*. É possível observar que existe um pequeno intervalo de tempo entre a subida do sinal de clock e o início do cálculo realizado pelos multiplicadores. Este tempo corresponde à propagação do sinal de entrada até a saída do registrador *input_reg*, acrescido ao tempo de estabilização do sinal feedback mais o atraso dos fios. Podemos ver também que a saída de todos os multiplicadores estão estabilizadas antes da descida do clock, sendo que o tempo de cálculo, após a estabilização do sinal feedback, é de aproximadamente 3,3 nanossegundos. Para ilustrar o caminho crítico deste módulo, apresentamos também na Figura 28 o sinal *q_xor0009*, que é uma das saídas das portas XOR existentes no codificador, sendo este o último resultado que deve estar estável antes de uma nova borda de clock para garantir o funcionamento do circuito. Como podemos ver, este sinal se estabiliza em menos de 5 nanossegundos após a subida do clock, garantindo o funcionamento do circuito a 100MHz com certa folga.

5.3 Validação em Hardware

Após a validação em nível de simulação, é realizada a validação em hardware dos módulos codificador e decodificador Reed-Solomon. Para tanto, utilizou-se a plataforma de prototipação DN8000kPCI [20] da empresa The Dini Group Inc.

Esta plataforma possui um FPGA Xilinx Virtex4 XCV4FX100-10 e um FPGA Xilinx Spartan2 XC2S200 que é utilizado somente para o controle e interconexão dos diversos dispositivos existentes. O controlador PCI Quicklogic 5064 é compartilhado entre os dois FPGA. Existem, portanto, dois modos de transferência de dados entre o barramento PCI e a Virtex4. O primeiro, e

que não será utilizado aqui, é o acesso direto fazendo uso de DMA e interrupção. O segundo é o acesso por intermédio da Spartan2 utilizando uma estrutura criada pela The Dini Group denominada *MainBus* [21].

5.3.1 MainBus

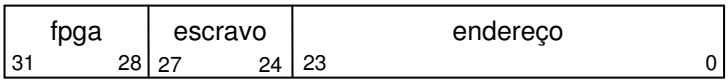
A estrutura *MainBus* emprega o paradigma *mestre-escravo* e consiste basicamente de um software denominado AETest, onde existem duas funções básicas de transferência de dados, e de dois módulos de hardware. O primeiro módulo, "MB_target.v", é responsável por receber as transições requisitadas através das duas funções, *mb_write* e *mb_read*, escrita e leitura respectivamente. Estas requisições são solicitadas ao segundo módulo, "slave.v", que por sua vez realiza a comunicação com o hardware sob teste.

5.3.1.1 mb_write e mb_read

As funções de transferência de dados, *mb_write* e *mb_read*, são bastante simples de se utilizar no software AETest. Abaixo se encontra a sintaxe das duas funções citadas.

```
    mb_write (uint addr, uint data)
int  mb_read (uint addr)
```

Na função *mb_write* o parâmetro *data*, de 32 bits, é o dado que se deseja enviar ao FPGA. Nas duas funções o parâmetro *addr*, também de 32 bits, possui o formato ilustrado abaixo.



O primeiro campo, denominado *fpga*, é utilizado para selecionar o FPGA entre todos os disponíveis. No caso da plataforma DN8000k10PCI que possui somente um FPGA este campo deve ser sempre igual a 2. O segundo campo é destinado à seleção do módulo *escravo* "slave.v" desejado. Este campo possui um tamanho de quatro bits e, portanto é possível utilizar até 16 *escravos*. O último campo, *endereço*, é utilizado para endereçar alguma estrutura interna do módulo *escravo*, como um banco de registradores, por exemplo.

5.3.1.2 MB_target.v e Slave.v

Como comentado anteriormente, o módulo "MB_target" é responsável por receber as transações de escrita e leitura do *mestre*, no caso o software AETest, e solicitá-las ao módulo *escravo* "slave.v". A Figura 29 demonstra a interface de comunicação do "MB_target.v".

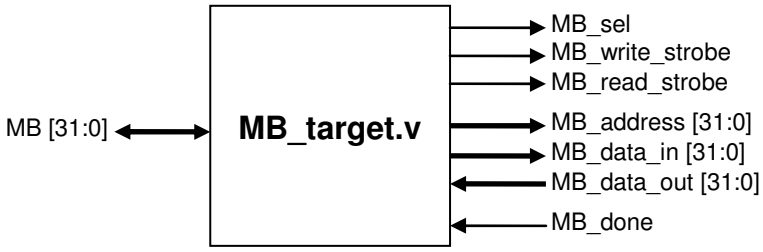


Figura 29 – Interface de comunicação do módulo MB_target.

Através do sinal *MB*, que é conectado diretamente no FPGA Spartan2, são recebidos os dados provenientes do barramento PCI. O restante dos sinais são responsáveis pelo protocolo de comunicação entre o módulo "MB_target.v" e o módulo "slave.v".

O sinal *MB_sel* inicia a requisição. Os sinais *MB_write_strobe* e *MB_read_strobe* indicam se a requisição é de escrita ou leitura, respectivamente. O endereço a ser acessado no "slave.v" é solicitado através de *MB_address*. Para a transferência dos dados das requisições são utilizadas os sinais *MB_data_in* e *MB_data_out*. Após o término de cada requisição, o *escravo* deve indicá-lo através de *MB_done*. A Figura 30 demonstra a interface de comunicação do módulo "slave.v".

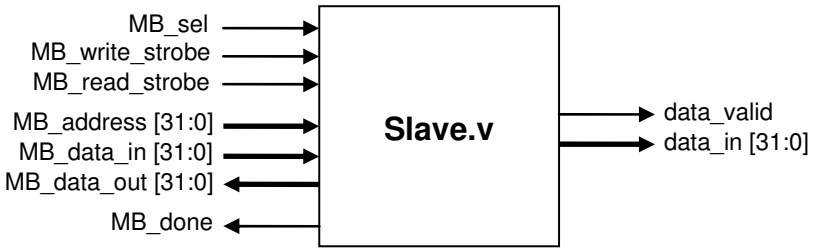


Figura 30 – Interface de comunicação do módulo slave.

No "slave.v", além dos sinais do protocolo de comunicação, existem ainda os sinais *data_valid* e *data_in.* Através destes sinais são entregues os dados ao hardware sob teste. O sinal *data_valid* indica novo dado de 32 bits no sinal *data_in.*

5.3.1.3 Software AEtest

O software AETest deve ser alterado para realizar a transferência de dados entre o computador e o FPGA. A seguir, apresentamos algumas linhas de código que demonstram a utilização das funções *mb_write* e *mb_read*.

```
scanf("%X",&dado_escrito);
mb_write(2|5|10, dado_escrito);
```

Neste exemplo, solicita-se primeiro o dado a ser escrito. Após, a função *mb_write* o escreve no endereço 10 do *escravo* 5, que por sua vez encontra-se no FPGA 2.

```
dado_lido=mb_read(2|5|10);
printf("Dado lido: %X do FPGA 2.", dado_lido);
```

Já neste exemplo, a função *mb_read* retorna o mesmo dado que foi escrito anteriormente pela função *mb_write*, ou seja, o dado do endereço 10, do escravo 5, do FPGA 2.

Abaixo se encontra a função implementada para o presente trabalho. Tal função é utilizada no software AETest e tem por objetivo ler dados de um arquivo texto e enviá-los ao FPGA.

```
if (!(p = fopen("packet.txt","r")))
{
      printf("Erro! Arquivo inexistente!\n");
}
else
{
      quad_byte=0;
      num_byte=0;

      while (!feof(p))
      {
            fscanf(p,"%X",&byte);
            quad_byte+=byte;
            if(num_byte<3)
            {
                  quad_byte<<=8;
                  num_byte++;
            }
            else
            {
                  //send quad_byte mb_write
                  mb_write(MB_FPGA_C | 0x00000000 | 1, quad_byte);
                  num_byte=0;
                  quad_byte=0;
            }
      }
      fclose (p);
}
```

Os códigos-fonte do software AETest podem ser compilados em diversos sistemas operacionais, incluindo Linux, Solaris, MS-Dos e as diversas versões do Microsoft Windows. Neste trabalho utilizou-se o sistema operacional Microsoft Windows XP e o compilador do VisualStudio 2005.

A DiniGroup disponibiliza *scripts* para facilitar a tarefa de compilação. Para utilizá-los é necessário criar um diretório de trabalho onde se copiará todos os arquivos do código-fonte do AETest. Após, é necessário definir no arquivo "Makefile" o sistema operacional que será utilizado. O trecho que deverá ser alterado é representado abaixo:

```
# set target operating system, define ONE of the following
DESTOS = WIN_WDM
#DESTOS = WINNT
#DESTOS = WIN98
#DESTOS = DOS_PHARLAP
#DESTOS = DOS_DJGPP
#DESTOS = LINUX
#DESTOS = SOLARIS
```

Feito isso, executa-se o terminal Visual Studio 2005 Command Prompt, apresentado na Figura 31. Deve-se alterar o diretório presente para o diretório de trabalho criado (por exemplo,

C:\Aetest\aetest). Após, deve-se digitar o comando “nmake”. O executável “aetest_wdm.exe” será criado.

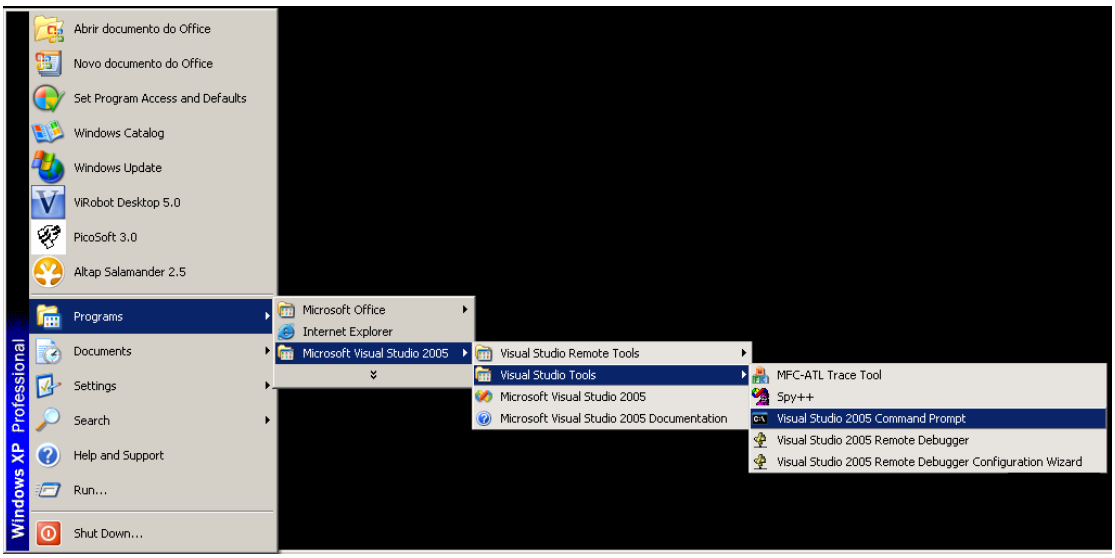


Figura 31 – Executando o terminal Visual Studio 2005 Command Prompt.

5.3.2 Estrutura de Teste

Para a validação em hardware, foi necessária a criação de uma estrutura de teste capaz de receber os bytes dos sub-frames provenientes do barramento PCI e enviá-los ao codificador e decodificador. Os módulos foram testados e validados separadamente na plataforma de prototipação. A estrutura criada pode ser observada de forma simplificada na Figura 32.

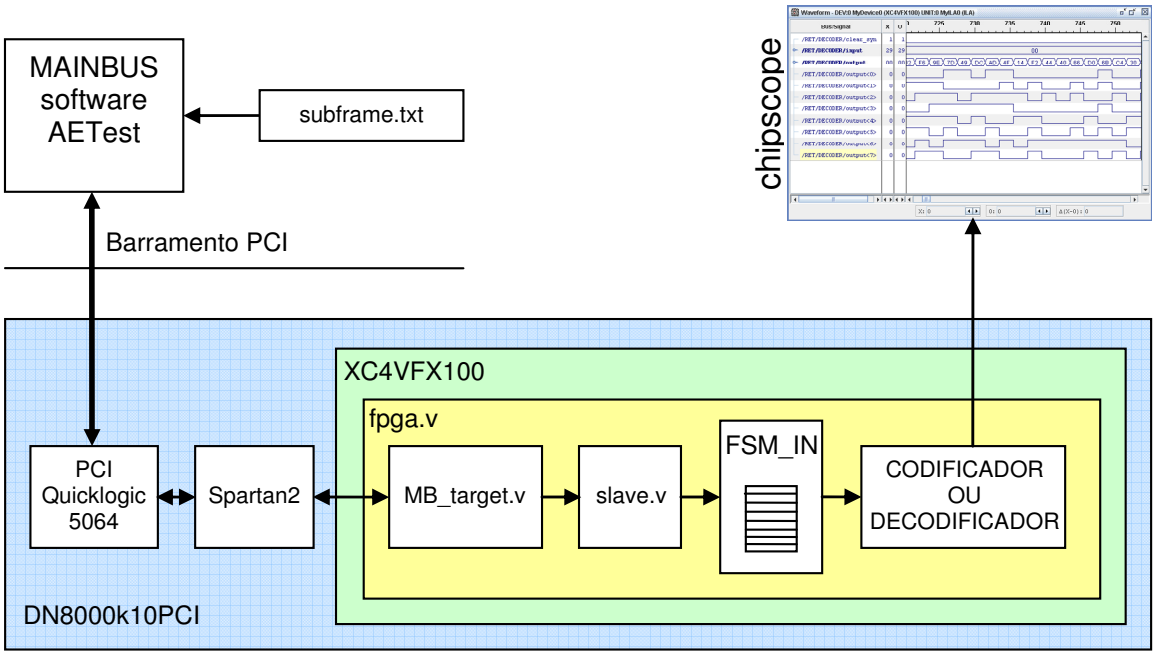


Figura 32 – Estrutura de teste para validação em hardware.

Através do software AETest é possível enviar ao FPGA os bytes do sub-frame existente no arquivo texto "subframe.txt". O dado enviado pelo *mb_write* possui o tamanho de 32 bits e, portanto, são enviados quatro bytes do sub-frame por vez. Os dados provenientes da função *mb_write* são recebidos pelo "MB_target.v" e disponibilizados ao módulo FSM_IN através do "slave.v". Na Figura 33 podemos observar a interface de comunicação do módulo FSM_IN.

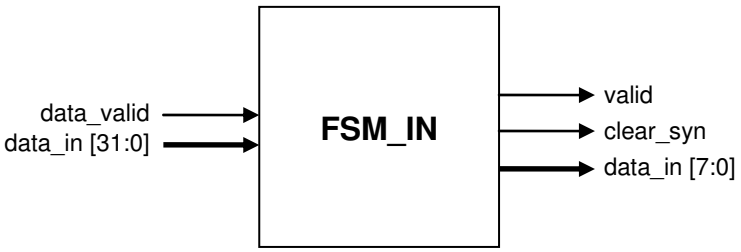


Figura 33 – Interface de comunicação do módulo FSM_IN.

O módulo FSM_IN é uma máquina de estados finita que recebe os bytes de um sub-frame e armazena-os em uma memória temporária antes de serem enviados ao módulo sob teste. Os dados são enviados através do sinal *data_in.* Os sinais *valid* e *clear_syn* são utilizados, dependendo do módulo que está sendo testado, respectivamente codificador ou decodificador. Na Figura 34 encontra-se a máquina de estados finita do módulo FSM_IN.

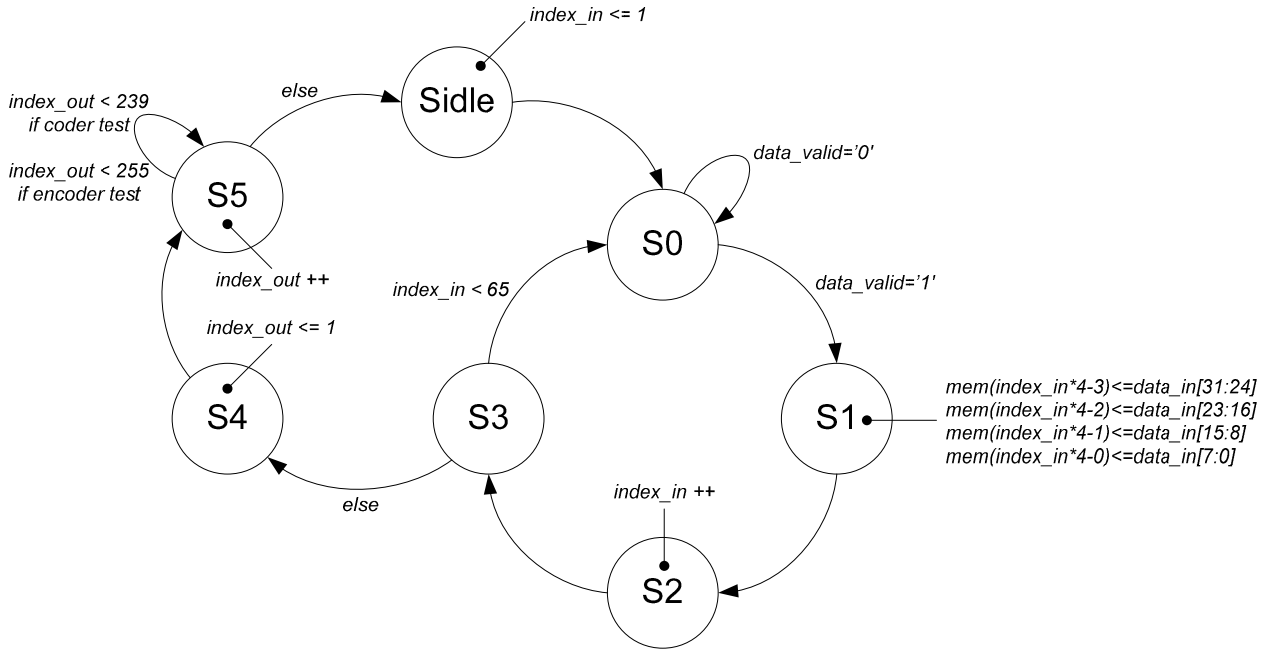


Figura 34 – Máquina de estados finita do módulo FSM_IN.

A verificação ocorre da mesma forma como é procedido na simulação. Para a visualização dos sinais dos módulos, utilizou-se a ferramenta ChipScope Pro Analyzer de versão 8.1.02i da Xilinx Inc.

No caso do codificador são enviados 239 bytes do sub-frame sem o campo de FEC. No ChipScope são visualizados os sinais de entrada, *input* e *valid*, e o sinal de saída *output*. Para verificar o correto funcionamento do codificador observa-se a validade do FEC gerado em seu sinal de saída, como podemos perceber na Figura 35.

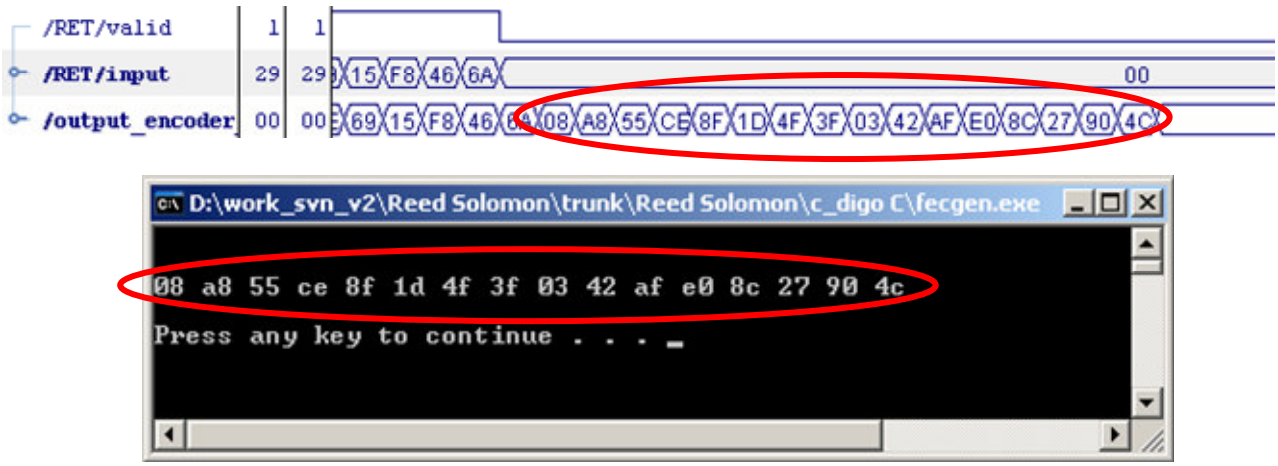


Figura 35 – FEC gerado pelo codificador.

No decodificador são enviados 255 bytes do sub-frame completo. No ChipScope são visualizados os sinais de entrada, *input* e *clear_syn*, e o sinal de saída *output*. Para verificar o seu funcionamento observa-se em seu sinal de saída se os erros, que são adicionados diretamente no "subframe.txt", foram corrigidos corretamente. A Figura 36 ilustra a correção dos seis erros, representados pelo caractere "00", inseridos no sub-frame.

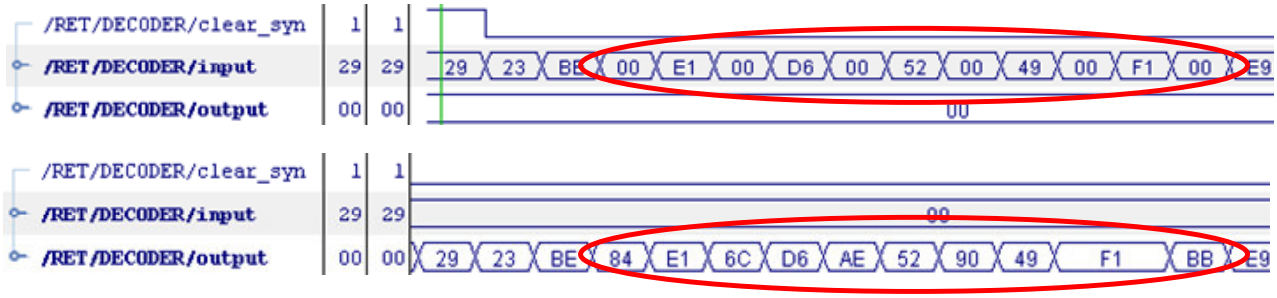


Figura 36 – Erros do sub-frame corrigidos pelo decodificador.

5.3.3 Resultados

Os módulos codificador e decodificador foram sintetizados, lógica e fisicamente, no software ISE versão 9.1i da Xilinx, fazendo uso da ferramenta de síntese XST. Um dos principais requisitos deste projeto é o desempenho. Ambos os módulos devem funcionar a uma freqüência de 100 Mhz. Outro importante requisito é a ocupação de área. O par de módulos codificador/decodificador ocupa 2847 LUTs, ou 3%, de 84352 LUT's disponíveis na Virtex4 XCV4FX100. Este número é similar ao total de LUT's observadas na literatura [8].

O codificador funcionou corretamente a 100 MHz. Já o decodificador apresentou um funcionamento falho a esta freqüência. Decidiu-se então reduzir a freqüência para 50 MHz. Nesta freqüência o decodificador funcionou corretamente, denotando um problema *timing* neste módulo.

Para localizar o problema, sintetizou-se cada sub-módulo separadamente para verificar a freqüência máxima de operação sugerida pela ferramenta de síntese. Na Tabela 7 são apresentadas as freqüências máximas sugeridas para cada módulo. O sub-módulo *forney* não possui freqüência máxima por ter uma lógica totalmente combinacional.

Tabela 7 - Freqüência máxima de operação de todos os módulos e sub-módulos.

Módulo	Freqüência Máxima de Operação em MHz
Codificador	276,67
Decodificador	83,87
- syndrome_unit	374,79
- berlekamp_massey	83,87
- chien_value	340,66
- chien_location	307,11
- forney	-
- delay	429,52

Podemos verificar que o gargalo do módulo decodificador encontra-se no sub-módulo Berlekamp-Massey, certamente por apresentar o pior caminho crítico entre todos os sub-módulos. Na Figura 37 verifica-se que o seu caminho crítico possui três multiplicadores genéricos. Este multiplicador, descrito na seção 4.1, possui oito linhas, sendo que cada uma dessas linhas possuem oito células que trabalham paralelamente. Uma linha intermediária depende do cálculo realizado pela linha superior. Portanto o multiplicador genérico apresenta um caminho combinacional de oito LUT's. Assim sendo, o sub-módulo Berlekamp-Massey herda o longo caminho combinacional de três multiplicadores, ou seja, 24 LUT's acrescido de duas operações XOR que consomem apenas uma LUT, totalizando 26 LUT's. Isto explica o motivo do seu baixo desempenho.

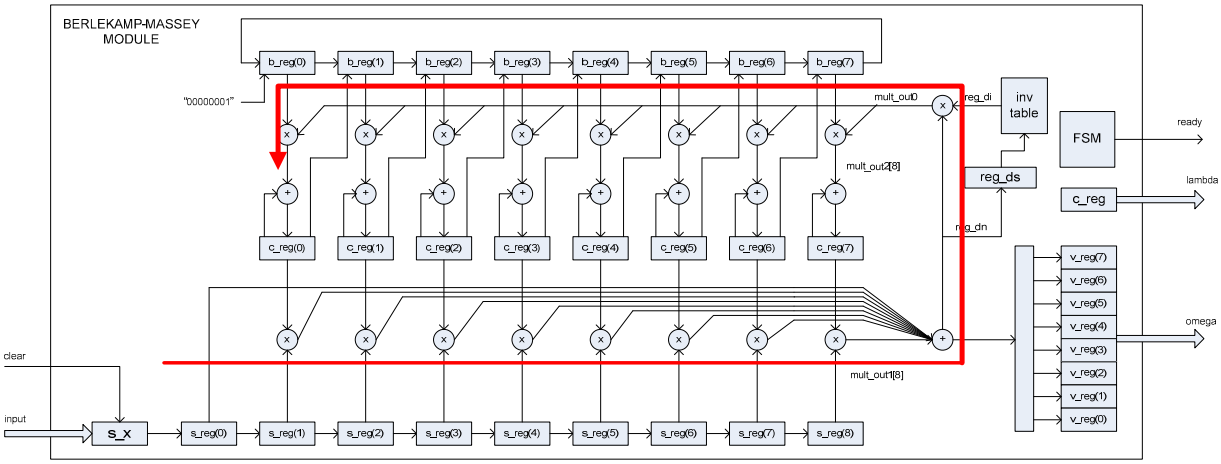


Figura 37 – Caminho crítico do sub-módulo *berlekamp_massey*.

Como a freqüência mínima de 100MHz não foi atingida pelo sub-módulo Berlekamp-Massey, implementamos ao final da redação deste documento, a partir de modificações na

arquitetura original, o algoritmo de Berlekamp-Massey sem inversão. Esta implementação diminui o caminho critico de três para dois multiplicadores, porém, apesar de não utilizar inversão, possui uma área muito maior quando comparada com a arquitetura original, pois esta implementação necessita de 10 multiplicadores a mais para ser implementada, além de alguns registradores e portas XOR adicionais. Para se ter uma idéia, um inversor é composto de 128 LUTs, enquanto um multiplicador, com duas entradas variáveis, consome 59 LUTS. Este acréscimo de área garante o funcionamento do módulo em uma freqüência de 100MHz, o que satisfaz o requisito de desempenho do projeto, porém prejudica profundamente o requisito de área. Outras tentativas foram feitas para tentar adequar a arquitetura original do Berlekamp-Massey para operar a 100MHz, utilizando-se do tempo que o módulo fica ocioso durante o processo de decodificação. Porém, essas tentativas não foram bem sucedidas.

Na Tabela 8 visualiza-se o total de área ocupada no FPGA, expresso em LUT's e Flip-Flop's, pelos módulos codificador e decodificador.

Tabela 8 – Ocupação de área no FPGA Virtex4 XCV4FX100.

Módulo	Lógica	Usados	Disponíveis	Utilização
Codificador	FlipFlop	132	84352	0,15%
	LUT	214		0,25%
Decodificador BM original	FlipFlop	883		1,04%
	LUT	2633		3,12%
Decodificador BM modificado	FlipFlop	908		1,08%
	LUT	3192		3,78%
Total BM original	FlipFlop	1015		1,20%
	LUT	2847		3,37%
Total BM modificadol	FlipFlop	1040		1,23%
	LUT	3406		4,04%

Como podemos ver na Tabela 8, a ocupação de um decodificador utilizando o algoritmo original de Berlekamp-Massey é de 3,12% em nossa arquitetura alvo, totalizando 3,37% incluindo o codificador. Utilizando o algoritmo modificado de Berlekamp-Massey, o decodificador passa a ocupar 3,78%, o que totaliza 4,04% com o decodificador. É importante saber que para cada LUT de um FPGA existe um Flip-Flop, portanto, desde que o número de Flip-Flops não ultrapasse o número de LUTs, a sua quantidade não influenciará na área total ocupada pelo projeto.

6 Integração

Como comentado na seção 1.2, um dos objetivos da atual parceria entre a TERACOM Telemática Ltda e o Grupo de Apoio ao Projeto de Hardware (GAPH) é a integração dos módulos codificador e decodificador Reed-Solomon com os módulos do *framer*, responsável por alinhar e montar os frames do padrão OTN. Na Figura 38 pode-se observar a integração de todos os módulos do projeto entre a TERACOM e GAPH.

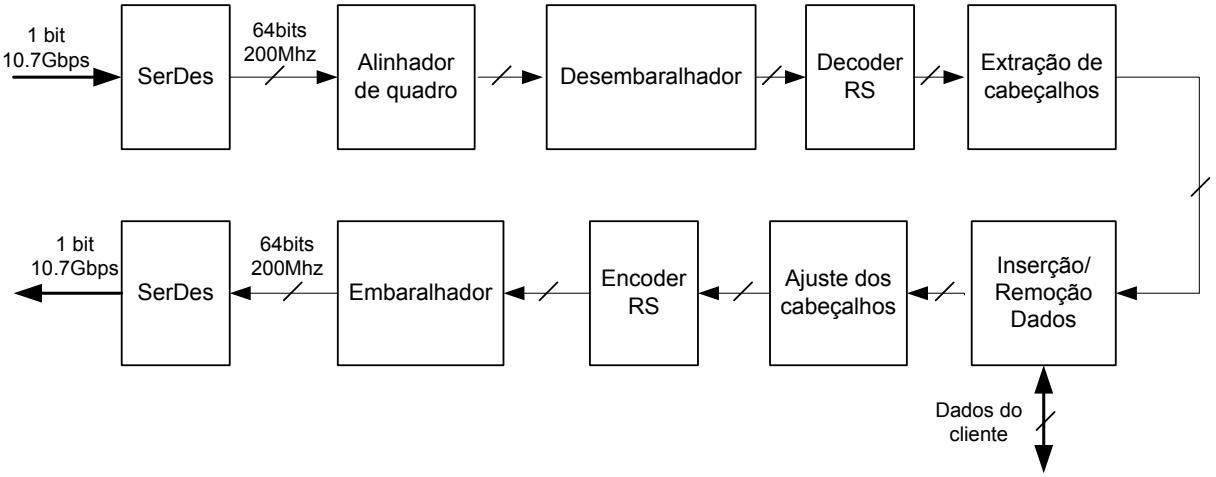


Figura 38 – Integração dos módulos do *transponder*.

O objetivo do módulo *SerDes* é receber o fluxo de dados provenientes da fibra óptica a 10.7 Gbps e disponibilizá-los em palavras de 64 bits a uma freqüência de 200 MHz. Este módulo também realiza a operação inversa, ou seja, transmite à fibra óptica os dados provenientes do módulo *Embaralhador*. O módulo *Alinhador de quadro* é responsável por identificar a ocorrência da seqüência de alinhamento e realizar o deslocamento, de modo que esta seqüência de alinhamento comece no início da palavra de 64 bits. Os módulos *Embaralhador* e *Desembaralhador* são responsáveis por embaralhar e desembaralhar os dados, de forma que não ocorram longas seqüências de 0s ou de 1s, o que garante uma suficiente troca de estado dos bits facilitando a recuperação do relógio e evitando uma possível repetição da seqüência de alinhamento. O módulo *Extração dos cabeçalhos* é utilizado para extrair o conteúdo dos cabeçalhos possibilitando a identificação de falhas, alarmes, tipo de dado transportado, origem e destino contidos nos frames OTN G.709. O módulo *Inserção/Remoção Dados* é utilizado para a análise dos frames e para a inserção de dados quando se faz necessário. O módulo *Ajuste dos cabeçalhos* é responsável pela montagem dos cabeçalhos dos frames a serem transmitidos.

Os módulos Decodes RS e Encoder RS são, na realidade, bancos de 16 decodificadores e 16 codificadores, pois como foi citado ao longo do trabalho, um frame possui o total de 16 sub-frames com cálculos de FEC independentes.

Inicialmente seriam utilizados bancos de 8 decodificadores e 8 codificadores, pois os dados são disponibilizados em palavras de 64 bits referentes a metade de uma coluna do frame, conforme a Tabela 2. Portanto, para realizar a computação da outra metade da coluna, foi necessária a implementação de uma estrutura que permitisse o compartilhamento dos 8 decodificadores e dos 8 codificadores. Esta alternativa foi logo descartada, pois esta estrutura requer um consumo de área praticamente igual à utilização de 16 decodificadores e 16 codificadores. Isso acontece porque as únicas estruturas que seriam compartilhadas seriam os multiplicadores e as portas XOR, e a adição de novos registradores para chavear o contexto entre dois cálculos independentes, requer a adição de muitos multiplexadores para controlá-los, o que não torna vantajoso compartilhar as demais estruturas.

Na Figura 39 encontra-se a arquitetura empregada nos módulos Decoder RS e Encoder RS. Como comentado anteriormente, os dados são disponibilizados em palavras de 64 bits por ciclo da frequência de 200 MHz. Como existem 16 cálculos de FEC independente, a frequência de operação do codificador e decodificador foi reduzida para 100 MHz. Isto significa que duas palavras de 64 bits, ou uma coluna do frame, são entregues aos módulos por ciclo da frequência de 100 MHz. Dessa forma, é possível calcular o FEC dos 16 sub-frames paralelamente e independentemente.

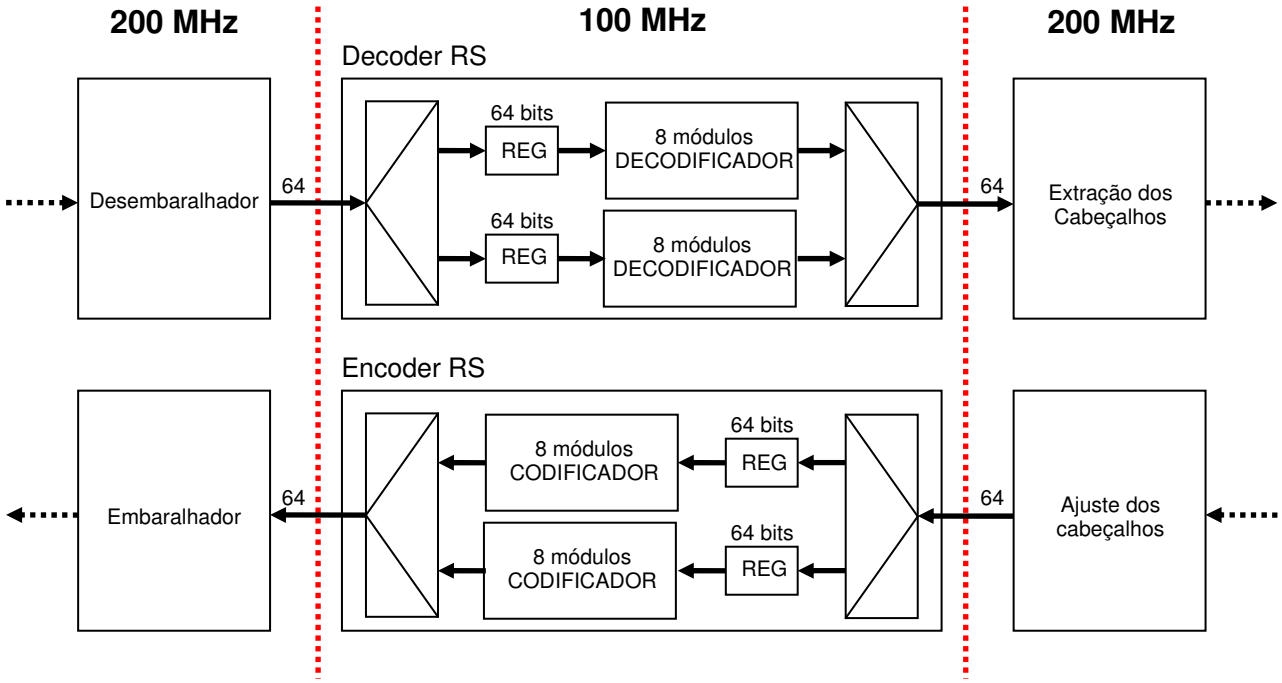


Figura 39 – Módulos Decoder RS e Encoder RS.

Ao final da redação deste documento a integração encontra-se em fase de validação funcional.

7 Conclusões e Trabalhos Futuros

Este trabalho abordou inicialmente o padrão OTN para dar uma base teórica sobre o protocolo onde o trabalho seria desenvolvido. A partir daí, estudou-se o algoritmo de Reed-Solomon para se ter uma idéia da matemática envolvida por trás dos processos de codificação e decodificação. Apresentamos a seguir, o projeto de todos os módulos desenvolvidos, bem como uma explanação básica de seu funcionamento. Também descrevemos os métodos utilizados para teste, bem como os resultados obtidos a partir dos módulos desenvolvidos.

Ao final deste trabalho pode-se dizer que praticamente todos os nossos objetivos foram atingidos. O objetivo principal, que seria o desenvolvimento de uma arquitetura de codificador e decodificador Reed-Solomon, de acordo com o padrão OTN, foi plenamente atingido. O objetivo secundário, que corresponde à inserção dos módulos desenvolvidos no *transponder*, não foi concluído, mas já se encontra em estágio avançado de desenvolvimento. Podemos dizer que já temos toda a estrutura de integração montada e em fase de simulação funcional, porém ela ainda não foi plenamente testada, pois para esta tarefa seria necessário uma ferramenta de geração de frames OTN completos, inclusive com os dados já embaralhados, que também está em desenvolvimento. Para um teste completo do *transponder,* também seria necessário um domínio de transmissão de dados em fibra óptica a partir do FPGA, que ainda não possuímos. Porém, os módulos desenvolvidos foram amplamente testados e o seu funcionamento foi completamente validado em FPGA.

Conclui-se que a maior dificuldade encontrada neste trabalho foi adequar os módulos desenvolvidos aos requisitos do projeto, pois estes são bastante restritivos. Em primeiro lugar, a taxa de injeção de dados não permitiu a utilização de multiplicadores iterativos, que consomem uma área extremamente pequena e não têm muitas restrições quanto à freqüência máxima de operação, visto que realizam poucas operações lógicas entre os registradores. O uso de multiplicadores completamente combinacionais satisfez os requisitos de desempenho, porém prejudicou muito a freqüência de operação e na área do projeto. Nesse ponto do trabalho já temos módulos que satisfazem à restrição quanto à freqüência de operação, porém, dado a necessidade de replicação dos módulos de codificação e decodificação, a área deste projeto chega a ocupar 64% do dispositivo alvo, que é praticamente o estado da arte em FPGAs de alta capacidade. Dessa maneira, acredita-se que algumas otimizações ainda podem ser feitas, ficando isto como trabalho futuro.

Acredita-se que este trabalho contribuiu bastante para a formação dos autores, pois através dele foi possível solidificar vários conceitos a respeito de desenvolvimento de hardware. O conceito de caminhos críticos foi amplamente explorado e compreendido. Também se pode por em prática o uso de simulação com atraso de portas e de roteamento, o que facilita muito a etapa de prototipação, na qual também adquiriu-se muita experiência. Através deste trabalho, também foi possível colocar

em prática o conceito de *pipelines*, além de aprender novos conceitos matemáticos e entender um pouco mais sobre redes de computadores.

Como trabalhos futuros, além de tentar realizar algumas otimizações no projeto, ocorrerá a validação completa da integração do Reed-Solomon com o *transponder* e o estudo das técnicas de transmissão e recepção de dados em fibra-óptica usando FPGAs. Acreditamos que através destas atividades será possível concluir o projeto.

8 Referências Bibliográficas

[1] International Telecommunication Union, *ITU-T G.709*. Março, 2003.

[2] Agilent Tecnologies. "An overview of ITU-T G.709". Capturado em: http://cp.literature.agilent.com/litweb/pdf/5988-3655EN.pdf, Setembro de 2001, Estados Unidos.

[3] Sklar, B. "Reed-Solomon Codes (Digital Communications: Fundamentals and Applications)". Upper Saddle River, NJ: Prentice-Hall, 2001, 1070p.

[4] Walker, T. "Optical Transport Network (OTN) Tutorial". Capturado em: www.itu.int/ITU-T/studygroups/com15/otn/OTNtutorial.pdf. Janeiro de 2007, Estados Unidos.

[5] Barlow, G. "A G.709 Optical Transport Network Tutorial", Innocor Ltd. Capturado em: http://www.innocor.com/pdf_files/g709_tutorial.pdf. Outubro de 2003, Almonte, ON, Canadá.

[6] Reed, I.; Solomon G. "Polymonial Codes Over Certain Finite Fields". *SIAM Journal of Applied Math*, vol. 8, Junho de 1960, pp. 300-304.

[7] Laws, B.; Rushforth, K. "A Cellular-Array Multiplier for GF(2^m). *IEEE Transactions on Computers*, Vol. C-20, No 12. Dezembro de 1971, pp. 1573- 1578.

[8] Wai, K.; Yang S. "Field Programmable Gate Array Implementation of Reed-Solomon Code, RS(255,239)". Capturado em: www.ce.rit.edu/~sjyeec/papers/nyworkshop-rs-codec.pdf, Outubro de 2005.

[9] Wilhelm, W. "A New Scalable VLSI Architecture for Reed-Solomon Decoders". *IEEE Journal of Solid State Circuit*, Vol. 34, No 2. Março de 1999, pp. 388-396.

[10] Sarwate, D.; Shanbhag, N. "High-Speed Architecture for Reed-Solomon Decoders". *IEEE Transactions on Very Large ScaleIintegration Systems*. Vol. 9, No 5. Outubro de 2001, pp. 641-655.

[11] Park, T. "Design of the (248,216) Reed-Solomon Decoder with Erasure Correction for Blu-ray Disc". *IEEE Transactions on Consumer Eletronics*, Vol. 51, No 3. Agosto de 2005, pp. 872-878.

[12] Massey, J. "Shift-register syntesis and BCH decoding.". *IEEE Transactions on Information Theory*, Vol. IT-15, No 1. Janeiro de 1969, pp. 122-127.

[13] Liu, K. "Architecture for VLSI Design of Reed-Solomon Decoders". *IEEE Transactions on Computers*, Vol. C-33, No 2. Fevereiro de 1984, pp. 178-189.

[14] Chang, H.; Shung, C.; Lee, C. "A Reed-Solomon Product-Code (RS-PC) Decoder Chip for DVD Aplications", IEEE Journal of Solid State Circuits, Vol. 36, No 2. Fevereiro de 2001, pp. 229-238.

[15] Reed, I.; Shih, M.; Truong, T. "VLSI design of inverse-free Berlekamp-Massey". IEEE Computer and Digital Techniques, Vol. 138, No 5. Setembro de 1991, pp. 295-298.

[16] Lee, H. "A High-Speed Low-Complexity Reed-Solomon Decoder for Optical Communications". IEEE Transactions on Circuits and Systems – II: Express Briefs, Vol. 52, No 8. Agosto de 2005, pp. 461-465.

[17] Moon, T. "Error Correction Coding: Mathematical Methods and Algorithms". Hoboken, NJ: Wiley, 2005, 794 p.

[18] Arash Partow Copyright, “Schifra”. Capturado em: http://www.schifra.com, Dezembro de 2007.

[19] Phil Karn, “FEC library”. Capturado em: http://www.ka9q.net/code/fec, Dezembro de 2007.

[20] The Dini Group INC. “DN8000K10PCI User Manual Version 3.0”. Capturado em: http://www.dinigroup.com/product/data/DN8000k10pci/files/dn8k10pci_manual.pdf, Novembro de 2007.

[21] The Dini Group INC. “Mainbus Specification”. Capturado em: http://www.dinigroup.com/product/common/mainbus_spec.pdf, Novembro de 2007.

[22] Almeida, G. “Códigos Corretores de Erros em Hardware para Sistemas de Telecomando e Telemetria em Aplicações Espaciais”, Dissertação de mestrado, Programa de Pós-Graduação em Ciências da Computação, PUCRS, 2007, 96 p.