UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Tiago Cardoso Barbosa

Implementação em FPGA de uma Arquitetura Reed-Solomon para Uso em Comunicações Ópticas

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Área de Concentração: Automação e Sistemas Elétricos Industriais

Orientador: Robson Luiz Moreno

Agosto de 2010
Itajubá - MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Cristiane N. C. Carpinteiro- CRB_6/1702

B238i

Barbosa, Tiago Cardoso
Implementação em FPGA de uma arquitetura reed-solomon para uso em comunicações ópticas / por Tiago Cardoso Barbosa. -- Itajubá (MG) : [s.n.], 2010.

55 p.: il.

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

1. Reed-Solomon. 2. FPGA. 3. VHDL. I. Moreno, Robson Luiz, orient. II. Universidade Federal de Itajubá. III. Título.

Dedico a toda minha família e todos que acreditam no meu trabalho.

O rio atinge seus objetivos porque contorna seus obstáculos.

Lao Tsé

Agradecimentos

Ao professor orientador, Robson Luiz Moreno, pelo crédito, pela confiança, pela amizade e pela ajuda na realização desse trabalho.

A todo grupo de Microeletrônica, pela amizade, ajuda, compreensão, que muito contribuiu para meu crescimento profissional.

Ao INATEL pela compreensão e flexibilidade proporcionada para termino do trabalho.

Aos familiares e amigos pelo apoio incondicional.

Resumo

A crescente demanda por redes com alta velocidade de transmissão, impulsionada pelos serviços que exigem altas taxas de dados, requer uma infra-estrutura de redes de fibra óptica de grande eficiência. A norma ITU-T G.709 que rege as comunicações ópticas tem como sua principal inovação, em relação a padrões antigos, o uso do código corretor de erros Reed-Solomon. O código Reed-Solomon é o mais utilizado em sistemas digitais na atualidade, devido a sua alta eficácia.

Este trabalho visa implementar o codificador e decodificador Reed-Solomon em hardware de alta performance para rápido processamento, o que é exigido em comunicações de alta velocidade. O hardware utilizado é um Field-Programmable Gate Array (FPGA), que permite uma rápida prototipagem, comparado com circuito integrado dedicado de aplicação específica (ASIC). O codificador e decodificador Reed-Solomon são desenvolvidos utilizando a linguagem de descrição de hardware VHDL, que permite um alto nível de abstração no desenvolvimento de circuitos digitais.

No trabalho é apresentada toda a estrutura dos circuitos que compõe o sistema, e é proposta uma nova topologia para o decodificador, que reduz a utilização lógica. Todo circuito é sintetizado e implementado na FPGA Stratix II do fabricante Altera que disponibiliza também o software para síntese denominado Quartus II. Os resultados da implementação são mostrados e discutidos no trabalho.

Palavras-chave: Reed-Solomon, FPGA, VHDL

Abstract

The increasing demand for networks with high transmission speed, driven by services requiring high data rates, requires an infrastructure of fiber optic networks for high efficiency. The ITU-T G.709 standard which governs the optical communications has as its main innovation, compared to old patterns, the use of Reed-Solomon as its error-correcting code. Nowadays, the Reed-Solomon code is used in most digital systems due to its high efficiency.

This work aims to implement the Reed-Solomon encoder and decoder in high performance hardware for fast processing, which is required in high-speed communications. The hardware used is a Field-Programmable Gate Array (FPGA), which allows a fast prototyping, compared with Application-Specific Integrated Circuit (ASIC). The Reed-Solomon encoder and decoder are designed using hardware description language VHDL, which allows a high level of abstraction in the development of digital circuits.

This thesis presents the whole structure of the circuits that make up the system, and proposes a new topology for the decoder by reducing the use of logic elements. The entire circuit is synthesized and implemented in the Altera's Stratix II FPGA, which also provides the Quartus II software for synthesis. The implementation results are shown and discussed in the following work.

Keywords: Reed-Solomon, FPGA, VHDL

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

ARQ	Automatic Repeat-reQuest
ASIC	Application-Specific Integrated Circuit
BCH	Bose Chaudhuri Hocquenghem
BM	Berlekamp-Massey
CD	Compact Disc
CI	Circuito Integrado
CLB	Configurable Logic Block
CODEC	Codificador e Decodificador
DLL	Delay Locked Loops
DSP	Digital Signal Processor
DVD	Digital Video Disc
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Array
GF	Galois Field
IP	Intellectual Property
ITU-T	International Telecommunication Union
OTN	Optical Transport Network
PCI	Peripheral Component InterConnect
PLL	Phase Locked Loops
RAM	Random Access Memory
ROM	Read Only Memory
RS	Reed-Solomon
SONET	Synchronous Optical Networking
SDH	Synchronous Digital Hierarchy
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

1 Introdução

1.1 Considerações Gerais

A crescente demanda por serviços de alta qualidade que exigem altas taxas de transmissão como transferência de dados em tempo real, propõe o desenvolvimento redes de transmissão de dados de alta eficiência a custos competitivos. Um sistema de transmissão que oferece altas taxas na faixa de Gbps é a transmissão em redes de fibra óptica. A norma G.709 define o protocolo para transmissão em fibra óptica e sua principal inovação foi o campo denominado FEC (Forward Error Corretion) que transporta símbolos adicionais que permite que até uma certa quantidade de erros ocasionada por ruídos no canal seja corrigida.

A norma G.709 é padronizada pelo ITU (União Internacional de Telecomunicações) para transporte de dados em interfaces ópticas, define o código Reed-Solomon (RS) para o FEC. Este é o código corretor de erro mais utilizado em sistemas digitais atualmente. Sua utilização vai desde sistema de transmissão via satélite até armazenamento de dados como CD (Compact Disc) e DVD (Digital Video Disc). O código RS realiza a correção de até uma certa quantidades de símbolos ao custo da transmissão de dados adicionais, conhecidos como símbolos de paridade. A norma G.709 especifica o código RS(255,239) para correção. Esse código tem a capacidade de corrigir até 8 símbolos em cada pacote de 255 símbolos. Para isso o pacote é constituído de 239 símbolos úteis e 16 símbolos de paridade, onde cada símbolo é composto de 8 bits.

O sistema OTN (Optical Transport Network) exige um hardware de alta capacidade para operar em altas taxas de transmissão. O processamento pode ser implementado em ASIC (Application-Specific Integrated Circuit) ou FPGA (Field-Programmable Gate Array) dependendo da demanda de produção. As FPGA's têm a vantagem para o projetista de ser um circuito de rápida prototipagem.

1.2 Justificativas do Trabalho

A correção de erros em redes de fibra óptica propicia uma diminuição de custos nas implantações. Com o FEC as distâncias entre os repetidores e regeneradores da rede são aumentadas proporcional-

mente com capacidade de correção do código. Isso ocasiona uma diminuição do uso de componentes ao longo da rede e consequentemente, um custo menor de manutenção e implantação devido aos altos preços dos componentes ópticos.

O dispositivo FPGA é um hardware com alto poder de processamento, o que atende a necessidade para implementação do FEC. O custo do dispositivo FPGA é diretamente proporcional ao número de elementos lógicos disponíveis. Cada circuito processador de pacotes da rede existe a necessidade de vários codificadores e decodificadores trabalhando paralelamente para realizar o trabalho na faixa de Gpbs exigido na transmissão. Os codificadores tem uma estrutura simples, por isso necessitam de menos recursos para serem implementados. Já no caso do decodificador a situação não é a mesma, porque ele faz uso de algoritmos de alto grau de complexidade que exigem maior área na implementação. Uma topologia que diminuir a área necessária para implementação da decodificação se traduziria em menores custos do componente.

1.3 Objetivo

O objetivo desse trabalho é desenvolver um circuito em hardware, utilizando FPGA, de um codificador e decodificador RS(255, 239). O circuito codificador utiliza poucos elementos lógicos sendo assim será implementado na sua forma tradicional utilizando registradores de deslocamento realimentados. Uma nova arquitetura do circuito decodificador, que reduz a área utilizada da FPGA, será proposta nesse trabalho para aproveitar melhor a capacidade de processamento diminuindo o tempo de ociosidade dos blocos que os constituem. A nova arquitetura irá reduzir a utilização de elementos lógicos necessária na implementação do sistema de processamento de dados, possibilitando assim o uso de um dispositivo de menor custo no hardware.

Os circuitos codificador e decodificador serão modelados, simulados, implementados e validados, utilizando o KIT NIOS II do fabricante Altera que também disponibiliza as ferramentas para todas as fases do projeto.

2 Códigos Corretores de Erro

2.1 Conceitos Básicos

A crescente demanda por altas velocidades de transmissão e grandes capacidades de armazenamento é uma exigência crescente no mundo atual. Isso é em decorrência de uma maior qualidade prestada aos consumidores em serviços como transmissão de vídeo e voz em tempo real. Neste contexto nasceu a necessidade de redes de alta performance capaz de transmitir altas taxas de dados. A norma G.709 do padrão OTN, define as interfaces para o transporte de dados em redes ópticas. A vantagem do padrão OTN sobre os padrões mais antigos, como SONET (Synchronous Optical Networking) e SDH (Synchronous Digital Hierarchy), é melhorias na forma da correção de erros[1].

Um processo de transmissão ou armazenamento de dados está sujeito a erros provenientes de diversas fontes como ruído, interferência, deterioração do meio, entre outros. Uma forma de corrigir a informação seria um pedido de retransmissão dos dados, através do protocolo *ARQ* (Automatic Repeat-reQuest), mas isso nem sempre é possível ou conveniente, porque quando existe a retransmissão diminui a taxa efetiva transmissão. Os códigos corretores de erro, que também são conhecidos como FEC, permitem que até uma certa quantidade de erros seja corrigida sem a necessidade da retransmissão.

A história da correção de erros começou pelo trabalho do matemático e engenheiro Shannon em 1948. Inicialmente a teoria foi desenvolvida por matemáticos nas décadas de 50 e 60. Em 1960 Irving S. Reed e Gustave Solomon desenvolveram o código RS (*Reed-Solomon*), que é utilizado na maioria dos sistemas hoje em dia. Com a popularização dos computadores a teoria de correção de erro chamou mais a atenção dos engenheiros [2]. Hamming em 1974 introduziu o chamado *Código Hamming*, e no mesmo ano Golay inventou o *Código Golay* [3].

Os códigos corretores de erro têm o mesmo princípio básico, onde redundância é adicionada a mensagem original para que seja possível sua correção, caso ocorra algum erro no processo de transmissão ou armazenamento. Depois que os símbolos de redundância são adicionados na mensagem original que contém k símbolos, a seqüência obtida é chamada de mensagem codificada, ou palavra código, que conterá um total n símbolos. Consequentemente a quantidade dos símbolos de redundân-

cia será $n - k$.

Existem dois formatos de codificação: sistemática e não sistemática. A diferença entre os dois formatos é que no primeiro caso a mensagem codificada consiste nos k símbolos da mensagem original seguido dos símbolos de redundância, como mostrado na figura 1. No caso da codificação não sistemática não é possível identificar a mensagem original na forma codificada.

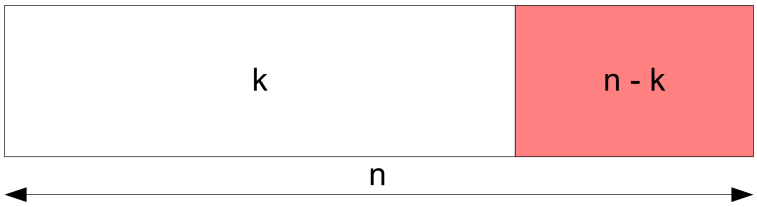


Figura 1: Codificação Sistemática

De acordo com a maneira pela qual a redundância é adicionada nas mensagens, os códigos corretores de erro são divididos em códigos de bloco e códigos convolucionais. Os códigos de bloco processam a informação pacote por pacote independentemente um do outro, por isso são denominados códigos sem memória [3]. Os principais códigos de bloco são o BCH (Bose Chaudhuri Hocquenghem), Hamming, Golay e o Reed-Solomon que é o código implementado neste trabalho. Diferentemente dos códigos de bloco a operação do código convolucional depende não somente do bloco que está sendo processado mas também do bloco anterior, um exemplo de código convolucional é o *Turbo Code* . Ambas as classes de código tem sua aplicação prática.

2.2 Códigos de Correção de Erro em Sistema de Comunicação Óptica

A correção de erros em sistemas de comunicações ópticas está sendo usada largamente hoje em dia. O uso do FEC aumenta capacidade de transmissão na fibra óptica. A comunicação óptica opera em altas taxas de transmissão, necessita de um código de alto desempenho e baixo custo. A implementação dos códigos convolucionais é difícil para altas taxas. Os códigos de bloco são capazes de corrigir múltiplos erros e introduz redundância constante, $n - k$ [4].

A norma de sistemas de comunicações ópticas ITU-T G.709 recomenda o uso do código Reed-Solomon RS(255,239), que é implementado nesse trabalho e discutido nos próximos capítulos.

2.2.1 Vantagens e Desvantagens dos Códigos Corretores de Erro

Em sistemas de fibra óptica de grande alcance é necessário a utilização de repetidores e uso de componentes de alta qualidade para aumentar a potência transmitida. Isso aumenta o custo da transmissão no sistema, devido ao alto custo dos componentes ópticos. O uso do FEC tem as seguintes vantagens [4]:

- Ganho significante de potência, que reduz a quantidade de repetidores no enlace.
- Diminui o custo final dos componentes ópticos.
- Corrige rajadas de erros em sistemas de fibras multimodo por interferência entre canais.

Desvantagens do uso dos códigos de correção de erro:

- Diminuição na taxa de dados efetiva, devido ao aumento de símbolos para serem transmitidos.
- Uso de *"hardware"* ou *"software"* para codificar e decodificar os dados.

Em sistemas com alta taxa de transmissão de dados, por volta de 10 Gbits/s a complexidade aumenta, o que requer um *"hardware"* com alto poder de processamento[4].

2.3 Campos de Galois

Nessa seção é introduzida uma parte da Álgebra Abstrata chamada teoria dos campos finitos, mais conhecida como campos de Galois (em homenagem ao matemático francês Évariste Galois). Estão descritos os conceitos básicos dos campos finitos para entendimento do código Reed-Solomon.

2.4 Conceitos Básicos: Grupóide, Semi-Grupo, Grupo e Campo

Grupóide é um espaço fechado onde a operação com dois de seus elementos resulta em um terceiro elemento pertencente ao conjunto.

Se a operação "." for uma composição interna de um conjunto e também tiver a propriedade associativa que satisfaz a equação 2.1, então a estrutura adquire uma nova propriedade e é designada por semi-grupo [5].

$$a.(b.c) = (a.b).c \tag{2.1}$$

Se existir um elemento identidade neutro e também existir um elemento inverso para cada elemento de um semi-grupo, então podemos denominar como um grupo. Se a operação "." for comutativa 2.2, temos um grupo albeliano ou comutativo.

$$a.b = b.a \tag{2.2}$$

O grupo tem uma ordem, ou cardinalidade, definida como sendo o número de elementos que o constitui. Cada elemento do grupo também tem uma ordem definida como sendo o menor número inteiro a que se tem que elevar o elemento para se ter igual a 1 [5].

$$B^m = 1 \tag{2.3}$$

Onde B é o elemento e m a ordem do mesmo. Por fim é definido campo com sendo um conjunto de elementos os quais é possível realizar as operações de adição, multiplicação, subtração e divisão, no qual o resultado sempre é um elemento do próprio campo. A adição e a multiplicação satisfaz as propriedades comutativa, associativa e distributiva [6].

2.5 Propriedades do Campo de Galois

A ordem do campo é determinada pela sua cardinalidade, ou seja, pelo número de elementos do campo. Por exemplo, um campo de ordem 2 representado por $GF(2)$, tem dois elementos, 0 e 1. Esse campo de dois elementos é chamado de campo binário.

Não existe um campo de Galois para um número qualquer de elementos. Os campos de Galois são formados por um número primo de elementos ou por sua extensão, onde a extensão é uma potência do número primo, onde p representa um número primo e m é um número inteiro positivo tal que $GF(q = p^m)$.

É definido como elemento primitivo, ou gerador, o elemento de ordem $(q - 1)$. As potências consecutivas do elemento primitivo gera todos os outros elementos do campo [5].

2.6 Polinômios Primitivos

É importante a compreensão dos polinômios primitivos, porque eles geram os elementos dos campos de Galois necessários para construção do código de correção de erro Reed-Solomon. Também se faz necessário a compreensão da classe de polinômios irredutíveis, pois uma das condições necessárias para o polinômio ser primitivo é ele ser irredutível.

Um polinômio $f(x)$ é dito irredutível se não for divisível por nenhum outro polinômio de grau inferior a ele e grau maior que 0 [5]. Um polinômio de grau 2 é irredutível se, somente se, ele não for divisível por polinômios de grau 1. Por exemplo o polinômio $X^2 + X + 1$ é irredutível, porque ele não é divisível por $X + 1$, nem por X [6].

Um polinômio $f(x)$ irredutível de grau m é primitivo se o menor número positivo inteiro n em que f(x) divide $X^n + 1$ é igual $a = 2m - 1$ como mostrado na equação 2.4. Polinômios primitivos são representados por $p(x)$ [4].

$$resto = \left(\frac{x^n + 1}{p(x)}\right) = 0 \tag{2.4}$$

As raízes de um polinômio primitivo são elementos primitivos [5], e como mencionado no item 2.5 o campo é construído a partir desses elementos.

2.7 Construção dos Campos de Galois 2^m

Os campos de Galois são construídos de acordo com um polinômio primitivo escolhido. Para exemplo pode-se considerar o polinômio primitivo $p(x) = X^4 + X + 1$. O grau do polinômio, definido por m, define o campo finito $GF(2^m)$. Assim $2^m = 2^4 = 16$ elementos no campo.

O próximo passo para construção do campo é encontrar as raízes de $p(x)$, e de acordo com o teorema fundamental da álgebra: **"um polinômio de grau m deve ter m raízes"**. As raízes do polinômio serão representadas por "α", então $p(\alpha) = 0$. Sendo assim pode-se escrever para o polinômio $p(x) = X^4 + X + 1$ as equações 2.5.

$$\begin{aligned} p(\alpha) &= 0 \\ \alpha^4 + \alpha + 1 &= 0 \\ \alpha^4 &= -\alpha - 1 \end{aligned} \tag{2.5}$$

Conforme definido na referência [4] para um campo $GF(q)$ as operações de soma e subtração são realizadas da mesma forma, simplifica-se para: $\alpha^4 = \alpha + 1$ Os elementos do campo $GF(2^m)$ podem ser expressos em potência de α, polinômios de degrau $(m-1)$ ou também como vetor binário. A raiz α^5 é expresso na forma polinomial pela equação 2.6.

$$\begin{aligned} \alpha^5 &= \alpha.\alpha^4 = \alpha(\alpha+1) \\ \alpha^5 &= \alpha + \alpha^2 \end{aligned} \tag{2.6}$$

Repetindo esse processo podemos encontrar todos os m elementos do $GF(2^4)$. A representação no formato vetorial binária é dada com o valor 1 onde existe o elemento α da potência representada e 0 caso contrário. O campo de $GF(2^4)$ do polinômio primitivo $p(x) = X^4 + X + 1$ está mostrado na tabela 1 com os respectivos elementos nas três representações [4].

Tabela 1: Construção do Campo de Galois

Representação Exponencial	Representação Polinomial	Representação vetorial binária
0	0	0000
α^0	1	1000
α^1	α	0100
α^2	α^2	0010
α^3	α^3	0001
α^4	$1 + \alpha$	1100
α^5	$\alpha + \alpha^2$	0110
α^6	$\alpha^2 + \alpha^3$	0011
α^7	$1 + \alpha + \alpha^3$	1101
α^8	$1 + \alpha^2$	1010
α^9	$\alpha + \alpha^3$	0101
α^{10}	$1 + \alpha + \alpha^2$	1110
α^{11}	$\alpha + \alpha^2 + \alpha^3$	0111
α^{12}	$1 + \alpha + \alpha^2 + \alpha^3$	1111
α^{13}	$1 + \alpha^2 + \alpha^3$	1011
α^{14}	$1 + \alpha^3$	1001

2.8 Operações nos Campos de Galois

Em um campo finito as operações entre quaisquer elementos resultam em outro elemento dentro do mesmo campo. Em um campo pode ser realizadas as operações de adição, subtração, multiplicação e divisão. As operações de adição e multiplicação satisfazem as propriedades comutativa, associativa e distributiva. O elemento identidade para adição é o 0 e para multiplicação é o 1[4][6].

O codificador e o decodificador RS necessitam das operações de adição e multiplicação e divisão que obedecem as regras dos campos finitos. Essas operações podem ser realizadas por portas lógicas, tabelas, flip-flops e registradores de deslocamento [4].

2.8.1 Adição e Subtração

Para um campo $GF(m)$ a soma de dois elementos i e j é dado pelo resto da divisão $(i + j)$ por m. Essa operação é chamada de adição módulo m. Para adição no módulo 2 é definida pelo campo G = 0, 1 está mostrada nas equações 2.7 [6].

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 1 \oplus 1 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \end{aligned} \tag{2.7}$$

Assim para um campo com mais elementos 2^m a soma de quaisquer elementos tem que resultar em um outro elemento pertencente ao mesmo campo, pois o campo de Galois é um conjunto fechado. Para adição de um elemento representado na notação vetorial, é feita como uma adição elemento por elemento módulo 2. O que é igual a uma operação lógica *OU EXCLUSIVO* bit a bit nos vetores a serem adicionados, como mostrado na equação 2.8.

$$\alpha^i \oplus \alpha^j = (a_{i0} \oplus a_{j0}) + (a_{i1} \oplus a_{j1})X + (a_{i2} \oplus a_{j2})X^2 + ... + (a_{i,m-1} \oplus a_{j,m-1})X^{m-1} \tag{2.8}$$

A subtração de elementos no campo de Galois é definida como sendo a operação OU Exclusivo porque $-\alpha = \alpha$ então $\alpha^n - \alpha^j = \alpha^n + \alpha^j$ [7].

2.8.2 Multiplicação e Divisão

A multiplicação entre dois elementos, i e j, de um campo de ordem prima, m é dado por $(ij)/m$. Para o campo binário $GF(2)$ tem-se o conjunto de equações 2.9 [6]:

$$\begin{aligned} 0 \otimes 0 &= 0 \\ 1 \otimes 1 &= 1 \\ 0 \otimes 1 &= 0 \\ 1 \otimes 0 &= 0 \end{aligned} \tag{2.9}$$

A operação de multiplicação entre dois elementos pertencentes a $GF(2^8)$ pode ser realizada na forma de tabela, registrador de deslocamento ou em uma forma genérica que utiliza portas lógicas *OU* e *E* [1]. A divisão de um elemento por outro é realizada multiplicando o elemento pelo inverso do outro que o divide como mostrado na equação 2.10. O inverso é obtido utilizando uma tabela que contém todos os elementos pertencentes ao campo e o seu respectivo inverso.

$$X = \frac{\alpha^i}{\alpha^j} = \alpha^i \times \alpha^{-j} \tag{2.10}$$

2.9 Códigos Cíclicos Lineares

Um código linear C é denominado código cíclico se todo deslocamento cíclico de uma palavra código gera uma outra palavra código pertencente a C. Se tivermos uma palavra $C = (c_0, c_1, c_2, ..., c_{n-1})$ e deslocarmos i vezes a palavra resultante será $C^{(i)} = (c_{n-i}, c_{n-i+1}, ..., c_{n-1}, c_0, c_1, ..., c_{n-i-1})$ [7]. Assim cada palavra codificada pode ser representada por um polinômio de grau igual ou menor a $n-1$. A palavra codificada pode ser expressa em formato polinomial como descrito na equação 2.11 e algebricamente existe uma relação entre o polinômio em sua forma original e o vetor deslocado dado pela equação 2.12.

$$C(x) = c_0 + c_1X + c_2X^2 + c_3x^3 + ... + c_{n-1}X^{n-1} \tag{2.11}$$

$$X^iC(x) = c_0X^i + c_1X^{i+1} + ... + c_{n-i-1}X^{n-1} + ... + c_{n-1}X^{n+i-1} \tag{2.12}$$

Manipulando a equação 2.12 para chegarmos a um resultado que obedeça a condição de um código cíclico, isto é, o polinômio não deve ter grau maior que $n-1$ procedemos como mostrado na

equação 2.13.

$$X^i C(x) = c_{n-i} + c_{n-i+1}X + ... + c_{n-1}Xi - 1 + c_0 X^i + ... + c_{n-i-1}X^{n-1} + c_{n-i}(X^n + 1) + c_{n-i+1}X(X^n + 1) + ... + c_{n-1}X^{i-1}(X^n + 1) \quad (2.13)$$

Se simplificarmos a equação 2.13 substituindo $c_{n-i} + c_{n-i+1}X + ... + c_{n-1}X^{i-1}$ por $q(x)$ teremos a equação dada na equação 2.14.

$$X^i C(x) = q(x)(X^n + 1) + c^{(i)}(x) \quad (2.14)$$

Então conclui-se que o vetor $C^{(i)}$ é dado pela divisão de $X^{(i)}C(x)$ por $(X^n + 1)$. Com isso temos a forma dada na equação 2.15.

$$C^{(i)}(x) = X^{(i)}C(x) mod(X^n + 1) \quad (2.15)$$

Onde mod é o módulo definido sobre polinômios, isto é, o grau do polinômio será no máximo igual a $(n - 1)$. Em um código linear cíclico $C(n, k)$ existe um único polinômio $g(x)$, mostrado na equação 2.16, de grau mínimo $(r < n)$ não zero chamado gerador polinomial que gera todas as palavras do código. As principais propriedades desse polinômio estão listadas a seguir [5]:

$$g(x) = g_0 + g_1 X + g_2 X^2 + g_3 x^3 + ... + X^r \quad (2.16)$$

- O polinômio gerador é mônico, o coeficiente do termo de maior grau é 1.
- O termo constante do polinômio deve ser 1 ($g_0 = 1$).
- Toda palavra codificada $C(x)$ é múltipla de $g(x)$ e ela pode ser expressa na forma de $c(x) = u(x)g(x)$, onde $u(x)$ é a palavra original.
- O polinômio deve ser fator de X^{n+1}.

Se multiplicarmos uma polinômio $u(x)$ de grau k por um polinômio gerador de grau r teremos como resultado um polinômio $c(x)$ 2.17 de grau $n = k + r$. Sendo assim o grau do polinômio gerador define a quantidade de símbolos de paridade contidos no código $c(n, k)$.

$$c(x) = (u_0 + u_1 X + u_2 X^2 + u_3 x^3 + ... + u_{k-1}X^{k-1})g(x) \quad (2.17)$$

2.9.1 Codificação Sistemática

Se multiplicarmos a mensagem original $u(x)$ pelo polinômio gerador $g(x)$, teremos um outro vetor $c(x)$, que em seus coeficientes não possuem nenhuma informação da mensagem $u(x)$. Isso é suficiente para termos uma mensagem codificada na forma não sistemática. Existe outro modo de codificação chamado de codificação sistemática onde os k últimos termos da mensagem $c(x)$ é a mensagem original $u(x)$ e os $(n-k)$ primeiros símbolos são os de paridade. Isso é conseguido fazendo as manipulações dadas com o polinômio $c(x)$. Primeiro desloca o polinômio $u(x)$ $n-k$ para direita conforme mostrado na equação 2.18.

$$X^{n-k}u(x) = u_0X^{n-k} + u_1X^{n-k+1} + u_2X^{n-k+2} + ... + u_{k-1}X^{n-1} \tag{2.18}$$

Depois divide-se $X^{n-k}u(x)$ por $g(x)$ para obter o polinômio restante $b(x)$ como mostrado na equação 2.19.

$$X^{n-k}u(x) = q(x)g(x) + b(x) \tag{2.19}$$

O grau de de $b(x)$ deve ser $n-k-1$ ou menor, porque o grau de $g(x)$ é $n-k$ [7]. Rearranjando a equação 2.19 obtemos a equação 2.20.

$$b(x) + X^{n-k}u(x) = q(x)g(x) \tag{2.20}$$

Como esse polinômio é múltiplo do polinômio gerador $g(x)$ entretanto ele é um código cíclico gerado por $g(x)$ definido como sendo $b(x) + X^{n-k}u(x)$. Assim a mensagem $c(x)$ codificada fica na forma de $c(x) = b_0 + b_1X + ... + b_{n-k-1} + u_0X^{n-k} + u_1X^{n-k-1} + ... + u_{k-1}Xn - 1$.

Os primeiros $n-k$ símbolos da mensagem codificada na forma sistemática são definidos como símbolos de paridade e os k símbolos restantes é a mensagem em sua forma original. A vantagem na codificação sistemática é que se receber uma palavra codificada sem erros, não precisamos de nenhum processamento para recuperar a mensagem original, apenas existe a necessidade da retirada dos símbolos de paridade. E quando a mensagem contém mais erros que a capacidade de correção, o código não tenta corrigi-la, porque uma correção errada pode ocasionar em mais erros na mensagem [7].

2.10 Código Reed-Solomon

O código de correção de erros Reed-Solomon é o mais utilizado em sistemas digitais de transmissão, seu uso vai desde sistemas de comunicação por satélite até sistemas de armazenamento de dados em massa como Compact Disc (CD) e Digital Video Disc (DVD). O código Reed-Solomon é um código de bloco cíclico da classe linear e não binário com símbolos de dentro de $GF(q^m)$. Ele é o mais poderoso código de bloco tendo capacidade de corrigir erros aleatórios e também erros em rajadas. Como o código RS é cíclico ele pode ser implementado usando registradores de deslocamento de alta velocidade, assim sendo indicado para comunicações de altas velocidades como sistemas de comunicação por fibra óptica [6][4].

Em 1960, Irving Reed e Gus Solomon publicou um artigo no Journal of Society for Industrial an Applied Mathematics, que continha um novo código corretor de erros que foi chamado de Reed-Solomon (RS), em homenagem aos respectivos pesquisadores. Esse código obteve uma ótima aceitabilidade devido a sua eficácia [8].

O Reed-Solomon é um código de bloco cíclico não linear e não binário, onde os símbolos são formados por sequências de m bits [8]. O código é citado como $RS(n, k)$, no qual n é o número total de símbolos de dados, k o número de símbolos que foram codificados e $n - k$ a quantidade de símbolos de paridade [9] [3] [8].

Uma das principais aplicações do código RS é na rede óptica regida pela norma G.709, o qual tem altas taxas de transmissão. Os k símbolos saem da fonte e passam pelo codificador $RS(n, k)$, onde ele introduz os $n - k$ símbolos de paridade, e são transmitidos. Do outro lado, na recepção, os dados são processados por um decodificador que corrige os eventuais erros e entrega ao destinatário os k símbolos transmitidos pela fonte. A figura 2 demonstra o esquema de codificação e decodificação [9].

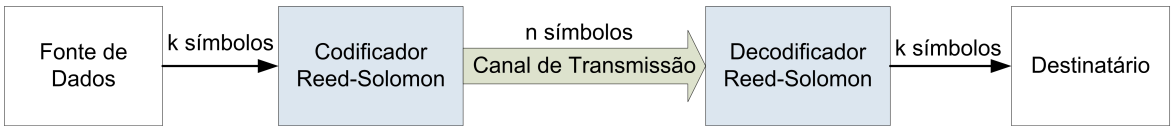


Figura 2: Esquema de Codificação RS

O código Reed-Solomon é capaz de corrigir t erros onde t é dado pela equação 2.21 [8] [6].

$$t = \frac{n-k}{2} \tag{2.21}$$

O código RS foi desenvolvido com base na álgebra abstrata e faz uso da teoria dos campos finitos, como discutido no anteriormente. Os campos de Galois são construídos de acordo com um polinômio primitivo. O polinômio primitivo especificado na norma G.709 para construção do campo

do RS(255,239) é o polinômio 2.22 [10].

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \tag{2.22}$$

2.11 Codificador

O código Reed-Solomon é codificado por um polinômio gerador de grau $n - k$, mostrado na equação 2.23, em que os elementos desse polinômio são pertencentes a $GF(q)$. Esse polinômio é formado pela multiplicação de $n - k$ polinômios de grau mínimo que tem suas raízes como potências consecutivas de α.

$$g(x) = \prod_{i=b}^{b+2t-1} (x + \alpha^i) \tag{2.23}$$

Se α é um elemento primitivo em $GF(q)$, o gerador polinomial $g(x)$ de um código Reed-Solomon com os símbolos pertencentes a $GF(q)$ tem como suas raízes $\alpha, \alpha^2, \alpha^3, ..., \alpha^{2t}$. Como as raízes de $g(x)$ são raízes de $X^{q-1} - 1$, $g(x)$ divide $X^{q-1} - 1$. Sendo assim o código é capaz de corrigir t símbolos ou menos em cada mensagem [7]. Um polinômio codificado $c(x)$ de grau $n - 1$ ou menor é pertencente a $GF(q)$ se somente se $c(x)$ for divisível por $g(x)$.

A variável b da equação 2.23 é um número aleatório, mas que deve ser escolhido com cuidado, pois pode aumentar muito a complexidade do sistema. A norma G.709 define o valor de b como sendo 0 [10]. Sendo assim a equação do polinômio gerador para esse caso está mostrado em 2.24.

$$\begin{aligned} g(x) = \prod_{i=0}^{15}(x + \alpha^i) \quad = \quad & X^{16} + \alpha^{120}X^{15} + \alpha^{104}X^{14} + \alpha^{107}X^{13} + \alpha^{109}X^{12} + \alpha^{102}X^{11} + \alpha^{161}X^{10} \\ & + \alpha^{76}X^9 + \alpha^3X^8 + \alpha^{91}X^7 + \alpha^{191}X^6 + \alpha^{147}X^5 + \alpha^{169}X^4 \\ & + \alpha^{182}X^3 + \alpha^{194}X^2 + \alpha^{255}X + \alpha^{120} \end{aligned} \tag{2.24}$$

O dado é codificado de forma sistemática, primeiro são enviados os dados provenientes da fonte e em seguida são enviados os símbolos de paridade, que são calculados de acordo com o gerador polinomial dado pela equação 2.24.

2.12 Decodificador

Depois dos dados passarem por um canal ruidoso na transmissão o vetor de dados recebidos $r(x)$ pode ser representado pela equação 2.25 que relaciona o vetor transmitido com os possíveis erros ocorridos no percurso.

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

Onde $c(x)$ é o dado transmitido e $e(x)$ representa o erro. De acordo com a matemática dos campos de Galois podemos encontrar a solução descrita na equação 2.26 para correção do erro.

$$r(x) = c(x) + e(x) + e(x) = c(x) \tag{2.26}$$

A operação de $e(x) + e(x) = 0$, pois, a adição nos campos finitos é uma operação lógica ou exclusivo. Então para encontrarmos e corrigirmos o erro no vetor transmitido basta encontrar o valor do erro e sua posição para adicionarmos no vetor recebido. O decodificador RS é capaz de fazer as operações necessárias[9].

A decodificação é realizada resumidamente em 5 passos: calculo da síndrome, calculo da equação chave, busca pela localização do erro, calculo valor do erro, correção da mensagem. O diagrama de bloco do decodificador está mostrado na figura 3 [11].

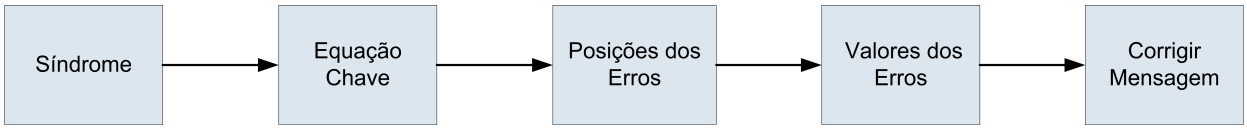


Figura 3: Decodificador Reed-Solomon

O calculo da síndrome verifica se houve o erro no meio de transmissão, o calculo da equação chave relaciona o resultado da síndrome com um polinômio que contém informações sobre a localização dos erros e outro que tem informações sobre os valores dos erros. Depois com posse desses polinômios o código faz uma busca nos mesmos para encontrar em qual símbolo está contido o erro e qual sua magnitude para que o decodificador possa corrigi-los [12] [9].

2.12.1 Síndrome

O calculo da síndrome é o resultado da verificação de paridade, determina se o vetor recebido $r(x)$ contém possíveis erros. O vetor recebido tem uma representação polinomial dada pela equação 2.27.

$$r(x) = r_0 + r_1x + r_2x^2 + ... + r_{n-1}x^{n-1} \tag{2.27}$$

O polinômio recebido tem que ser múltiplo do gerador polinomial $g(x)$, porque toda palavra do código é múltipla do polinômio gerador, sendo assim, para que a mensagem não contenha erros a condição da equação 2.28 tem que ser verdadeira.

$$\frac{r(x)}{g(x)} = q(x) + 0 \tag{2.28}$$

O resto da divisão polinomial da equação 2.28 é chamado de síndrome S, e para um vetor sem erros ela tem valor 0. Um valor da síndrome diferente de zero indica a presença de erro. A síndrome também pode ser representada em forma polinomial com um grau de no máximo $(n - k - 1)$. Isso significa que ela pode conter $(n-k)$ símbolos. Sendo assim temos o polinômio da síndrome mostrado em 2.29 [6] [8].

$$s(x) = s_0 + s_1x + s_2x^2 + ... + s_{n-k-1}x^{n-k-1} \tag{2.29}$$

Como um vetor válido $r(x)$ é múltiplo de $g(x)$, as raízes de $g(x)$ devem ser raízes de $r(x)$. Sendo assim $r(x)$ tem que ser avaliado a cada raiz de $g(x)$ e o resultado deve ser zero para um vetor livre de erros. O vetor da síndrome pode ser encontrado usando a equação 2.30, onde o índice i representa as raízes de $g(x)$ [8].

$$s_i = r(x)|_{x=\alpha^i} = r(\alpha^i) \tag{2.30}$$

Relacionando 2.30 e 2.25 encontramos 2.31.

$$s_i = c(\alpha^i) + e(\alpha^i) = e(\alpha^i) \tag{2.31}$$

Se $e(x)$ conter v erros nas posições $x^{j_1}, x^{j_2}, x^{j_3}, ..., x^{j_v}$, o polinômio do erro se resume em 2.32.

$$e(x) = e_{j_1}x^{j_1} + e_{j_2}x^{j_2} + e_{j_3}x^{j_3} + ... + e_{j_3}x^{j_v} \tag{2.32}$$

Das equações 2.31 e 2.32 temos o conjunto de equações 2.33 que relacionam a síndrome, a localização dos erros e o valor dos erros.

$$
\begin{aligned}
S_1 &= e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + e_{j_3}\alpha^{j_3} + ... + e_{j_3}\alpha^{j_v} \\
S_2 &= e_{j_1}\alpha^{2j_1} + e_{j_2}\alpha^{2j_2} + e_{j_3}\alpha^{2j_3} + ... + e_{j_3}\alpha^{2j_v} \\
&\cdot \\
&\cdot \\
&\cdot \\
S_{2t} &= e_{j_1}\alpha^{2tj_1} + e_{j_2}\alpha^{2tj_2} + e_{j_3}\alpha^{2tj_3} + ... + e_{j_3}\alpha^{2tj_v}
\end{aligned} \tag{2.33}
$$

Para simplificar as equações substitui-se $\alpha_{j_i} = \beta_i$ e $e_{j_i} = \delta_i$. Com isso a equação 2.33 se dá na forma de 2.34.

$$
\begin{aligned}
S_1 &= \delta_1\beta_1 + \delta_2\beta_2 + \delta_3\beta_3 + ... + \delta_v\beta_v \\
S_2 &= \delta_1\beta_1^2 + \delta_2\beta_2^2 + \delta_3\beta_3^2 + ... + \delta_v\beta_v^2 \\
&\cdot \\
&\cdot \\
&\cdot \\
S_{2t} &= \delta_1\beta_1^{2t} + \delta_2\beta_2^{2t} + \delta_3\beta_3^{2t} + ... + \delta_v\beta_v^{2t}
\end{aligned} \tag{2.34}
$$

O polinômio localizador de erros é definido como sendo um polinômio de grau mínimo v, dependente da quantidade de erros. Assim o polinômio pode ser representado por 2.35 que pode ser escrito na forma de 2.36.

$$\Lambda(x) = (1 - \beta_1 x) + (1 - \beta_2 x) + (1 - \beta_3 x) + ... + (1 - \beta_v x) \tag{2.35}$$

$$\Lambda(x) = \lambda_0 + \lambda_1 x^1 + \lambda_2 x^2 + \lambda_3 x^3 + ... + \lambda_v x^v \tag{2.36}$$

Da equação 2.35 e do conjunto de equações 2.34 podemos obter as equações conhecidas como Identidades de Newton 2.37, que relaciona os coeficientes de $\Lambda(x)$ com os coeficientes de $S(x)$.

$$\begin{aligned} S_{v+1} + \lambda_1 s_v + \lambda_2 s_{v-1} + ... + \lambda_v s_1 &= 0 \\ S_{v+2} + \lambda_1 s_{v+1} + \lambda_2 s_v + ... + \lambda_v s_2 &= 0 \\ &\vdots \\ S_{2t} + \lambda_1 s_{2t-1} + \lambda_2 s_{2t-2} + ... + \lambda_v s_{2t-v} &= 0 \end{aligned} \tag{2.37}$$

Se a mensagem recebida contiver erros e esse erro for uma palavra do código esse erro é indetectável pelo decodificador [7].

2.12.2 Equação Chave

Existem diferentes algoritmos para resolver a equação chave, mas os dois mais usados são o algoritmo Euclidiano e o Berlekamp-Massey(BM). O algoritmo Euclidiano tem uma estrutura mais simples, porém utiliza mais lógica para sua implementação devido a um divisor polinomial. O algoritmo BM tem uma estrutura mais complexa, porém gasta menos lógica e levando em conta que a economia de área em uma FPGA significa menor custo, o algoritmo utilizado nesse trabalho foi o BM[9].

O algoritmo de Berlekamp-Massey realiza o calculo do polinômio localizador de erro de grau mínimo, $\Lambda(x)$, através de um método interativo [7] [6]. Esse método é composto $2t$ passos. O conjunto de equações denominado como identidades de Newton está mostrado em sua forma genérica na equação 2.38.

$$s_{\tau+1} + \lambda_1 s_\tau + ... + \lambda_{\tau-1} s_2 + \lambda_\tau s_1 = 0 \tag{2.38}$$

O primeiro passo do algoritmo BM é determinar um polinômio de grau mínimo $\Lambda^{(1)}_{BM}(x)$ que satisfaz a primeira identidade de Newton da equação 2.39.

$$s_1 + \lambda_1 = 0 \tag{2.39}$$

O próximo passo é verificar se o polinômio também satisfaz a segunda identidade descrita na equação 2.40.

$$s_2 + \lambda_1 s_1 = 0 \tag{2.40}$$

Se o polinômio $\Lambda_{BM}^{(1)}(x)$ satisfaz a segunda identidade, então $\Lambda_{BM}^{(2)}(x)$ é igual a $\Lambda_{BM}^{(1)}(x)$. Caso contrário um outro polinômio d_p deve ser calculado para satisfazer as duas primeiras identidades de Newton. Esse processo é aplicado subsequentemente até alcançar o polinômio $\Lambda_{BM}^{(2t)}(x)$. O polinômio obtido na μ-géssima interação é dado pela forma genérica mostrada na equação 2.41.

$$\Lambda_{BM}^{(\mu)}(x) = 1 + \lambda_1^{(\mu)} x + \lambda_2^{(\mu)} x^2 + ... + \lambda_{l\mu}^{(\mu)} x^{l\mu} \tag{2.41}$$

Para verificar a condição do polinômio $\Lambda(x)^{(u)} = \Lambda(x)^{\mu+1}$ o algoritmo BM faz um calculo chamado discrepância. A equação da discrepância, denotada por d_μ 2.42, for igual a zero o polinômio satisfaz a $(\mu+1)$-géssima identidade de Newton, então o polinômio $\Lambda_{BM}^{(\mu+1)}(x)$ será igual ao $\Lambda_{BM}^{(\mu)}(x)$.

$$d_{(\mu)} = s_{\mu+1} + \lambda_1^{(\mu)} s_\mu + \lambda_2^{(\mu)} s_{\mu-1} + ... + \lambda_{l_\mu}^{\mu} s_{\mu+1-l_\mu} \tag{2.42}$$

Se a discrepância for diferente de zero o polinômio não satisfaz a $(\mu+1)$-géssima identidade de Newton, então um outro polinômio deve ser calculado para satisfazer todas as identidades de Newton menor e igual a $(\mu+1)$. O calculo do termo de correção utiliza um passo anterior da interação μ denominado p tal que a discrepância d_p seja diferente de zero e $p - l_p$ tenha o maior valor, onde lp é o grau de $d_p(x)$. Então o polinômio para satisfazer a $(\mu+1)$-géssima identidade de Newton está mostrado em 2.43.

$$\Lambda_{BM}^{(u+1)}(x) = \Lambda^{(\mu)}(x) + d_\mu d_p^{-1} x^{\mu-p} \Lambda^{(p)}(x) \tag{2.43}$$

O algoritmo de Berlekamp-Massey pode ser implementado seguindo o fluxograma da figura 4 [9].

Para facilitar os cálculos pode-se utilizar uma tabela 2 padrão em várias referências [7] [6].

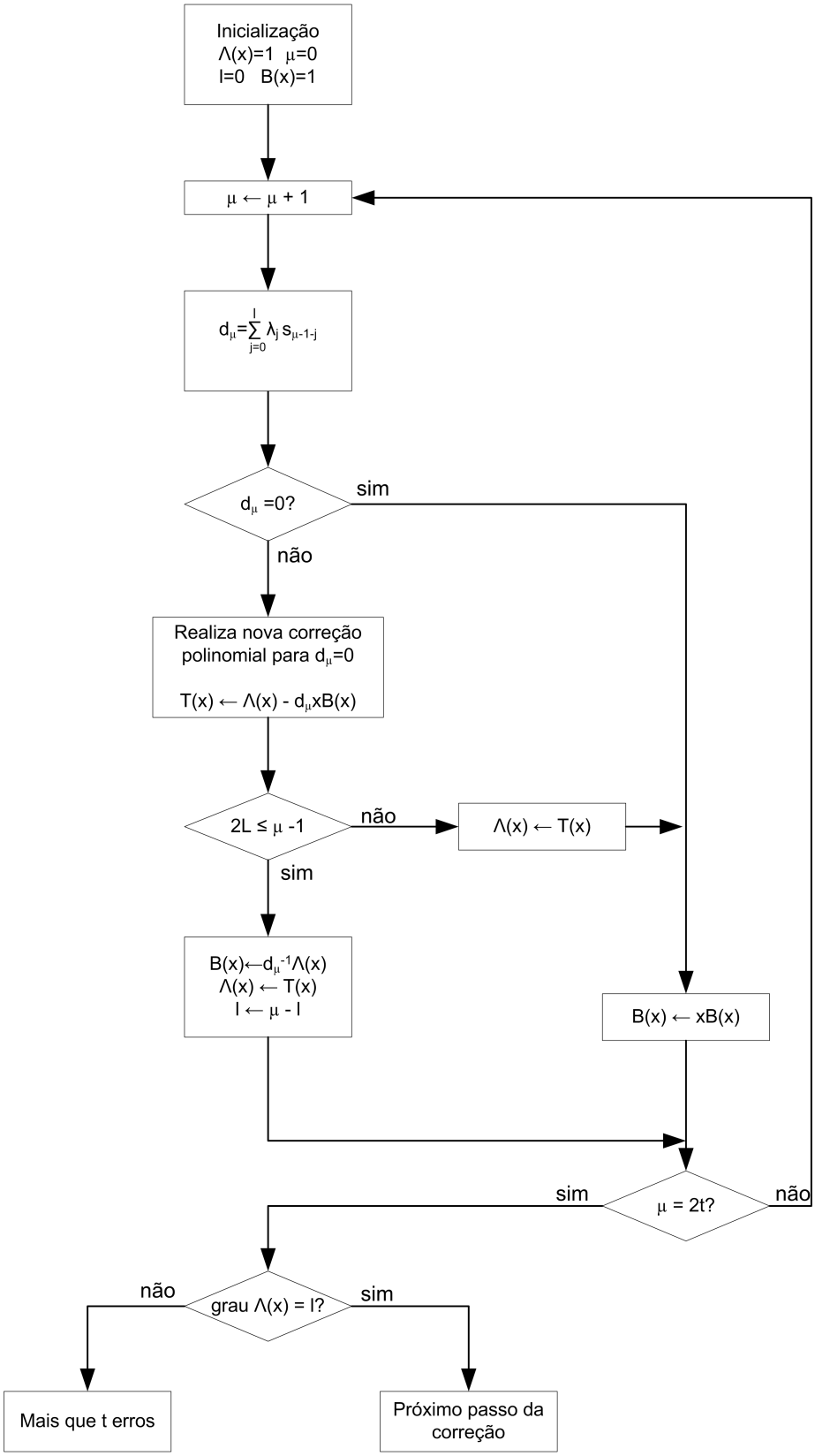


Figura 4: Fluxograma do Algoritmo Berlekamp-Massey

Tabela 2: Tabela para cálculo do algoritmo BM

μ	Λ_{BM}^{μ}	d_μ	l_μ	$\mu - l_\mu$
-1	1	1	0	-1
0	1	s_1	0	0
1				
.				
.				
.				
$2t$				

No último passo se o grau de $\Lambda_{BM}^{(2t)}(x)$ for maior que t significa que os erros contidos no vetor recebido é maior que t então o código o que excede a capacidade de correção[6].

Depois que o polinômio $\Lambda(x)$ é encontrado e seu grau for menor ou igual a t o próximo passo da correção é encontrar a localização dos símbolos errados na palavra codificada. Chien propôs um método eficiente de substituição para encontrar os símbolos errados. Os símbolos de maior ordem são testados em primeiro lugar, para verificar se r_{n-1-i} é um símbolo livre de erro, o algoritmo testa se α^i é uma posição de erro, isto é equivalente a testar se o inverso de α é raiz de $\Lambda(x)$. Se a equação 2.44 for satisfeita r_{n-1-i} é um símbolo errado, caso contrário r_{n-1-i} é um símbolo correto.

$$1 + \lambda_1\alpha + \lambda_2\alpha^2 + ... + \lambda_v\alpha^v = 0 \tag{2.44}$$

Para uma posição genérica r_{n-1-i} a equação 2.44 se resume na equação 2.45.

$$1 + \lambda_1\alpha^i + \lambda_2\alpha^{2i} + ... + \lambda_v\alpha^{vi} = 0 \tag{2.45}$$

No código RS se faz necessário encontrar também a magnitude do erro, devido a sua natureza não binária. Para tal função temos uma equação denominada equação chave 2.46, que relaciona a síndrome $S(x)$ com o polinômio localizador de erros $\Lambda(x)$ e o polinômio que contém informação sobre os valores dos erros definido como $\Omega(x) = 1 + \omega_1 x + \omega_2 x^2 + ... + \omega_{2t} x^{2t}$ [6][9].

$$\Lambda(x)s(x) = \Omega(x) mod x^{2t+1} \tag{2.46}$$

A equação chave 2.46 é denominada desta forma porque com apenas o polinômio da síndrome é suficiente para encontrar os polinômios que identificam o local do erro e sua magnitude. Neste ponto da decodificação temos os valores de $S(x)$ e $\Lambda(x)$, então podemos rearranjar a equação chave 2.46 da

maneira mostrada na equação 2.47 [9].

$$\Omega(x) = \Lambda(x)S(x) mod x^{2t+1} \tag{2.47}$$

O algoritmo de Forney é usado para encontrar a magnitude do erro. O polinômio $\Omega(x)$ é relacionado com o valor do erro pela expressão 2.48 [7].

$$e_{j_l} = \frac{\Omega(\beta_l^{-1})}{\Lambda'(\beta_l^{-1})} \tag{2.48}$$

Assim para um erro na posição α^i a equação 2.48 pode ser reescrita como mostrado na equação 2.49.

$$e = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})} \tag{2.49}$$

A dedução da equação 2.49 pode ser encontrada na referência [7].

Na expressão 2.48 Λ' é a derivada formal do polinômio $\Lambda(x)$ em um ponto x. A derivada formal de um polinômio pertencente a $GF(2^m)$ é apenas os termos das potências ímpares, por exemplo a derivada formal de 2.50 é 2.51 [4].

$$f(x) = f_0 + f_1 x + f_2 x^2 + ... + f_n x^n \tag{2.50}$$

$$f(x) = f_1 + f_2 x + ... + f_n x^{n-1} \tag{2.51}$$

2.12.3 Exemplo de Decodificação

Este exemplo dado na referência [7] ilustra o processo de decodificação. Os símbolos do cógigo são pertencentes a $GF(2^4)$ com o gerador polinomial dado pela equação 2.52.

$$\begin{aligned} g(x) &= (x+\alpha)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4)(x+\alpha^5)(x+\alpha^6) \\ &= \alpha^6 + \alpha^9 x + \alpha^6 x^2 + \alpha^4 x^3 + \alpha^{14} x^4 + \alpha^{10} x^5 + x^6 \end{aligned} \tag{2.52}$$

A mensagem recebida no decodificador é $r(x) = \alpha^7 x^3 + \alpha^3 x^6 + \alpha^4 x^1 2$. O primeiro passo no processo de decodificação é verificar a paridade do código, calculando a síndrome, assim temos o conjunto de equações 2.53.

$$\begin{aligned} S_1 &= r(\alpha) = \alpha^{10} + \alpha^9 + \alpha = \alpha^{12} \\ S_2 &= r(\alpha^2) = \alpha^{13} + \alpha^0 + \alpha^{13} = \alpha^0 \\ S_3 &= r(\alpha^3) = \alpha + \alpha^6 + \alpha^{10} = \alpha^{14} \\ S_4 &= r(\alpha^4) = \alpha^4 + \alpha^{12} + \alpha^7 = \alpha^{10} \\ S_5 &= r(\alpha^5) = \alpha^7 + \alpha^3 + \alpha^4 = 0 \\ S_6 &= r(\alpha^6) = \alpha^{10} + \alpha^9 + \alpha = \alpha^{12} \end{aligned} \tag{2.53}$$

Para encontrar o polinômio localizador de erros utilizando a tabela 2 do algoritmo BM, obtém a tabela 3.

Tabela 3: Tabela do Algoritmo BM

μ	Λ^{μ}_{BM}	d_μ	l_μ	$\mu - l_\mu$
-1	1	1	0	-1
0	1	α^{12}	0	0
1	$1 + \alpha^1 2x$	α^7	1	0 (considere $= \rho = -1$)
2	$1 + \alpha^3 x$	1	1	1 (considere $= \rho = 0$)
3	$1 + \alpha^3 x + \alpha^3 x^2$	α^7	2	1 (considere $= \rho = 0$)
4	$1 + \alpha^4 x + \alpha^{12} x^2$	α^{10}	2	2 (considere $= \rho = 2$)
5	$1 + \alpha^7 x + \alpha^4 x^2 + \alpha^6 x^3$ 0		3	2 (considere $= \rho = 3$)
6	$1 + \alpha^7 x + \alpha^4 x^2 + \alpha^6 x^3$			

O algoritmo Chien realiza uma busca em todos os elementos do campo $\alpha^0, \alpha^1, ..., \alpha^{14}$, quais são raízes de $\Lambda(x)$. Realizando a busca tem-se que α^3, α^9 e α^{12}, são raízes de $\Lambda(x)$, assim os erros estão contidos nas posições de x^3, x^6 e x^{12}.

Para encontrar o polinômio avaliador de erros utiliza-se a equação chave 2.47. Assim tem $\Omega(x) = \alpha^{12} + \alpha x$. Assim podemos encontrar os valores dos erros dados pela equação 2.49. Para as posições x^3, x^6 e x^{12} obtem os valores dos erros dado pelo conjundo de equação 2.54.

$$
\begin{aligned}
e_3 &= \frac{\Omega(\alpha^{-3})}{\Lambda'(\alpha^{-3})} = \frac{\alpha^{12} + \alpha\alpha^{-3}}{\alpha^3(1+\alpha^6\alpha^{-3})(1+\alpha^{12}\alpha^{-3})} = \frac{\alpha}{\alpha^9} = \alpha^7 \\
e_6 &= \frac{\Omega(\alpha^{-6})}{\Lambda'(\alpha^{-6})} = \frac{\alpha^{12} + \alpha\alpha^{-6}}{\alpha^6(1+\alpha^3\alpha^{-6})(1+\alpha^{12}\alpha^{-6})} = \frac{\alpha^3}{1} = \alpha^3 \\
e_{12} &= \frac{\Omega(\alpha^{-12})}{\Lambda'(\alpha^{-12})} = \frac{\alpha^{12} + \alpha\alpha^{-12}}{\alpha^{12}(1+\alpha^3\alpha^{-12})(1+\alpha^6\alpha^{-12})} = \frac{\alpha^6}{\alpha^2} = \alpha^4
\end{aligned} \tag{2.54}
$$

Assim o padrão de erro é dado pela expressão $e(x) = \alpha^7 x^3 + \alpha^3 x^6 + \alpha^4 x^1 2$. Da equação 2.25 temos que a mensagem corrigida dada por $r(x) = 0$.

3 Implementação

3.1 FPGA

FPGA (Field-Programmable Gate Array) é um dispositivo lógico programável que possui uma arquitetura baseada em blocos lógicos configuráveis, chamados de CLBs (Configuration Logical Blocks). Os CLB's são formados por look-up tables, multiplexadores e flip-flops que implementam funções lógicas. As FPGA's também são compostas por estruturas chamadas de blocos de entrada e saída (IOB Blocks), os quais são responsáveis pela interface entre as entradas e saídas provenientes das combinações de CLBs. A estrutura de um CLB está mostrado na figura 5.

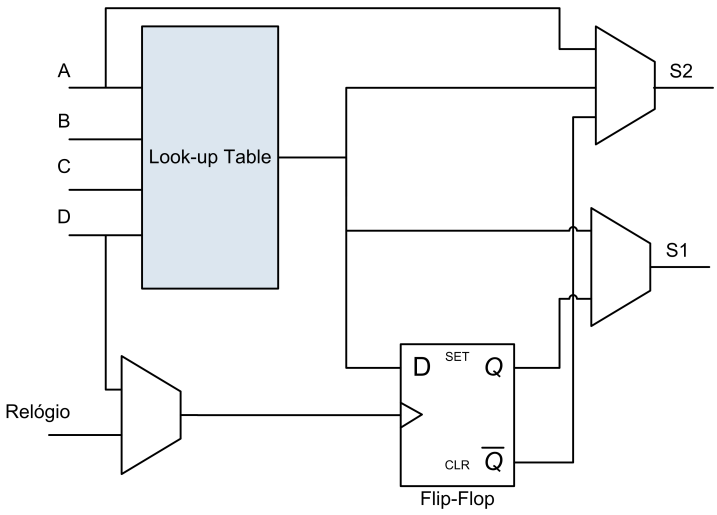


Figura 5: Bloco Lógico Configurável

As FPGA's modernas além dos blocos lógicos configuráveis, também tem em sua estrutura blocos de memória RAM (Random Access Memory) de alta profundidade, blocos DSP (Digital Signal Processor) que realizam operações aritméticas em centenas de megahertz, PLL (Phase-Locked Loop) e DLL (Delay-Locked Loop) que gerenciam os sinais de relógio e blocos com aplicação específica como interface PCI Express (Peripheral Component Interconnect Express) implementado em circuito dedicado. Como exemplo está mostrado na figura 6, retirada da referência [13], a estrutura de uma FPGA da família Stratix do fabricante Altera.

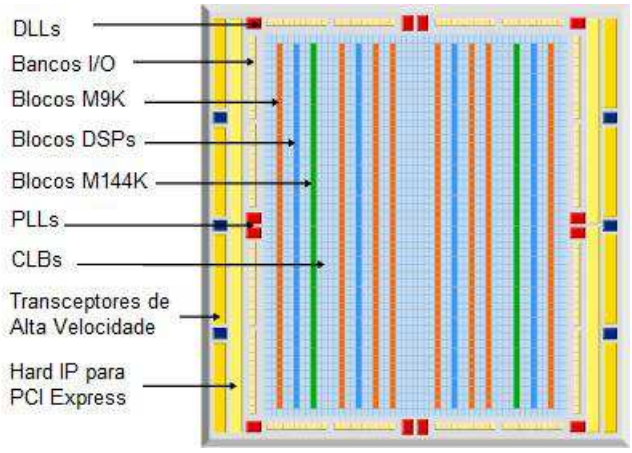


Figura 6: Arquitetura da FPGA Stratix

O mercado de FPGA é ocupado em 80% por dois principais fabricantes que são Xilinx e Altera. As FPGA's também são utilizadas para prototipagem de ASIC's devido a possibilidade de inúmeras regravações. Elas também são usadas na produção de equipamentos em baixa escala distribuídos em diversas áreas como equipamentos médicos, setor automotivo, comunicações com e sem fio, área militar entre outras. As principais vantagens do uso de FPGA são o alto desempenho, redução do tempo de projeto, possibilidade de reconfiguração do dispositivo depois de implantado e possibilidade de utilização de linguagens com alto nível de abstração para configuração do comportamento do circuito. A principal desvantagem em relação aos ASIC's é o maior custo por unidade, dependendo da demanda.

Um fluxo de projeto de FPGA consiste nos seguintes passos:

- Especificação - Levantar todos os requisitos e características do sistema como: atraso máximo, freqüência de operação, consumo de potência, custo, etc.
- Modelamento - O circuito é projetado em uma linguagem de descrição de hardware ou utilizando esquemáticos.
- Simulação - Momento em que se avalia o comportamento do circuito e valida o modelo produzido até aquele momento. O processo contínuo evita a propagação de erros em etapas posteriores.
- Síntese - O compilador transforma a linguagem em elementos lógicos primitivos.
- Mapeamento - O circuito é implementado dentro da tecnologia disponível na FPGA.
- Roteamento - É definido o lugar e roteamento de cada bloco da FPGA.
- Implementação - O arquivo de configuração é descarregado na FPGA.

- Testes - O circuito é analisado utilizando um analisador lógico ou ferramentas disponibilizadas pelo fabricante.

3.2 Introdução a Linguagem VHDL

O circuito do CODEC (Codificador e Decodificador) Reed-Solomon foi projetado e implementado utilizando a linguagem de descrição de hardware VHDL (VHSIC Hardware Description Language), onde VHSIC (Very High Speed Integrated Circuits). Esta seção tem como objetivo uma breve introdução a essa linguagem.

O VHDL é uma linguagem que permite ao projetista de circuitos digitais um alto nível de abstração para descrever o comportamento do circuito a ser implementado. Na linguagem não existe a necessidade de especificar explicitamente as ligações entre os componentes físicos dentro do dispositivo de hardware [14].

Historicamente o VHDL nasceu da necessidade de uma ferramenta padrão para simplificar os manuais que descreviam o comportamento dos ASICs. Os circuitos integrados aumentavam sua complexidade a medida que os dispositivos que os constituíam foram tomando dimensões cada vez menores, como consequência os CI's foram se tornando cada vez mais densos e os esquemáticos cada vez maiores, sendo assim o VHDL tornou-se uma ferramenta poderosa e rápida para a descrição dos circuitos. Hoje em dia a linguagem é utilizada para documentação, descrição, síntese e teste de circuitos digitais.

Nessa linguagem, como descreve hardware, todos os comandos são executados concorrentemente, não importa a ordem de descrição do código. A linguagem permite delimitar regiões onde o código é executado seqüencialmente, tal região é denominada processo. Dentro de um processo o código é executado seqüencialmente, mas todos os processos são executados de forma concorrente [14].

Dentre as particularidades da linguagem é que nenhuma distinção é feita entre comandos empregando letras minúsculas e maiúsculas e os comentários se iniciam após dois hífens e termina no final da linha.

Um código em VHDL é composto de três partes principais: biblioteca, entidade e arquitetura. As bibliotecas agregam comandos que são usados frequentemente na descrição. As bibliotecas work, que armazena o projeto compilado, e a std (standart), que contém os comandos padrões do VHDL, estão implícitas em todo projeto, assim não existe a necessidade de declará-las. A entidade define a interface entre o circuito e o ambiente exterior. A arquitetura contém todo o comportamento do circuito, a relação entre as entradas e saídas. Na figura 7 tem-se exemplo de código com o circuito

correspondente.

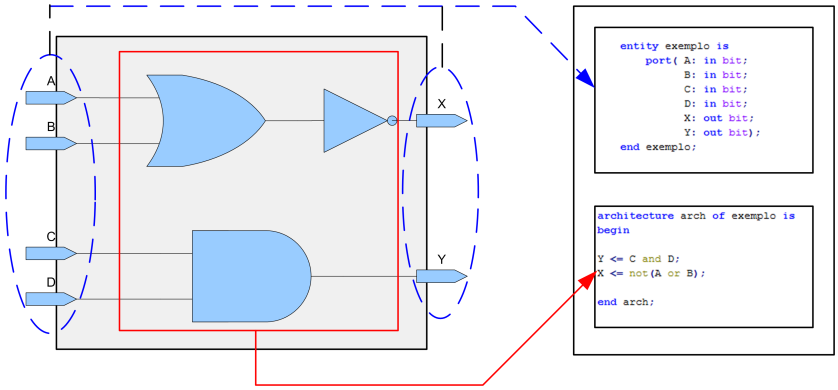


Figura 7: Exemplo de Código em VHDL

Na figura 7 temos um circuito composto por 4 portas de entrada denominadas, A, B, C e D e duas portas de saída denominadas X e Y. Tais portas são declaradas na primeira parte do código, na entidade. A relação entre as portas de entrada e saída, isto é, o comportamento do circuito é descrito na segunda parte denominada arquitetura. No caso desta figura a arquitetura é formada por 3 portas sendo elas uma porta lógica OU, outra inversora e uma porta E.

No VHDL existe a possibilidade de 4 modos de portas possíveis:

- IN - as portas operam exclusivamente como entrada
- OUT - as portas operam exclusivamente como saída
- BUFFER - a porta opera unicamente no modo saída, diferenciando do modo OUT porque o valor apresentado pode ser referenciado internamente pela arquitetura. Uma porta no modo OUT não pode, por exemplo, controlar um sinal interno
- INOUT - caracteriza uma porta bidirecional

Em VHDL também temos os itens denominados objetos, e são eles:

- Sinal - sinais internos
- Constante - carrega um valor estático
- Variável - semelhante a um sinal. Ela só pode ser usada em regiões de código seqüencial. Seu valor é atualizado instantaneamente, ao contrário do sinal que seu valor só é alterado depois que o código executa o processo

- Arquivo - a linguagem também nos permite a manipulação de arquivos de dados no formato, .mif, .hex, etc.

Os objetos diferentemente das portas são declarados em uma região específica do código depois da linha *Architecture* e antes de *begin* da arquitetura.

Os tipos de dados contidos na biblioteca padrão estão descritos na tabela 4. Além da utilização dos tipos pré-definidos a linguagem VHDL permite a criação de novos tipos. A criação de novos tipos é muito utilizada em descrição de máquinas de estado e memórias.

Tabela 4: Tipos Predefinidos em VHDL

Tipo	Valor
Bit	$1, 0$
Boolean	verdadeiro, falso
Character	Caracteres ASCII
Integer	$-2^{31} - 1$ à $2^{31} - 1$
Natural	0 à $2^{31} - 1$
Positive	1 à $2^{31} - 1$
Real	$-3,65 \times 10^{47}$ à $3,65 \times 10^{47}$
Time	ps, ns, us, ms, sec, min, hr
Bit_vector	vetores de 0 e 1
String	conjunto de caracter

Dentre as principais vantagens da utilização do VHDL se destacam:

- Facilidade de atualização dos projetos
- Verificação do comportamento do sistema digital, através de simulação
- Redução do tempo e custo do projeto
- Eliminação de erros de baixo nível do projeto

3.3 Codificador Reed-Solomon

O codificador foi projetado de acordo com o polinômio gerador estabelecido pela norma G.709 mostrado na equação 2.24. O projeto em hardware utiliza registradores de deslocamento com realimentação para cálculo e inserção dos símbolos de paridade. O circuito também contém somadores e

multiplicadores, que realizam operações de acordo com as regras dos campos de Galois. Os multiplicadores utilizados foram os multiplicadores de Mastrovito, que através de combinações entre portas OU e E realizam a operação. A dedução do multiplicador está descrito detalhadamente na referência [15]. O codificador está mostrado na figura 8.

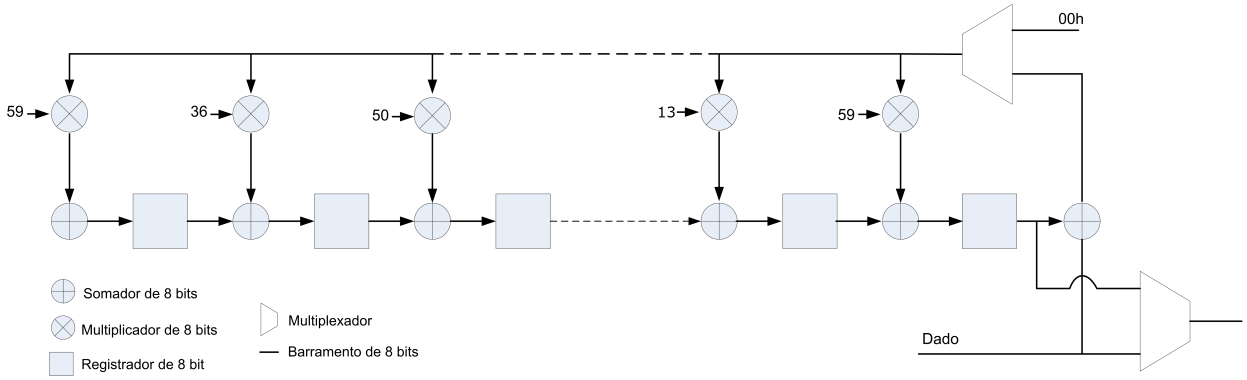


Figura 8: Circuito do Codificador Reed-Solomon

Os coeficientes que multiplicam o dado, são os coeficientes do polinômio gerador $g(x)$ em sua forma decimal. A equação 2.24 com coeficientes decimais está mostrado na 3.1.

$$\begin{aligned} g(x) = \prod_{i=0}^{15}(x+\alpha^i) \quad &= \quad X^{16} + 59X^{15} + 13X^{14} + 104X^{13} + 189X^{12} + 68X^{11} + 209X^{10} \\ &\quad +30X^9 + 8X^8 + 163X^7 + 65X^6 + 41X^5 + 229X^4 + 98X^3 + 50X^2 \\ &\quad +36X + 59 \end{aligned} \tag{3.1}$$

Primeiramente os $k = 239$ símbolos da mensagem a ser codificada são enviados e ao mesmo tempo alimentam o circuito que realiza as operações. Depois que os k símbolos são enviados, o conteúdo dos registradores de deslocamento, que são os $n = 16$ símbolos de paridade, são enviados em sequência.

A figura 9 mostra os registros na descrição do código em VHDL. Os registros foram inferidos como uma matriz denominada no código como *reg*, onde cada linha dessa matriz é composta de 8 posições que armazenam os símbolos de cada multiplicador mostrado na figura 8.

```
reg(0)  <= alpha_120(s_aux);
reg(1)  <= reg(0)  xor alpha_225(s_aux);
reg(2)  <= reg(1)  xor alpha_194(s_aux);
reg(3)  <= reg(2)  xor alpha_182(s_aux);
reg(4)  <= reg(3)  xor alpha_169(s_aux);
reg(5)  <= reg(4)  xor alpha_147(s_aux);
reg(6)  <= reg(5)  xor alpha_191(s_aux);
reg(7)  <= reg(6)  xor alpha_091(s_aux);
reg(8)  <= reg(7)  xor alpha_003(s_aux);
reg(9)  <= reg(8)  xor alpha_076(s_aux);
reg(10) <= reg(9)  xor alpha_161(s_aux);
reg(11) <= reg(10) xor alpha_102(s_aux);
reg(12) <= reg(11) xor alpha_109(s_aux);
reg(13) <= reg(12) xor alpha_107(s_aux);
reg(14) <= reg(13) xor alpha_104(s_aux);
reg(15) <= reg(14) xor alpha_120(s_aux);
```

Figura 9: Descrição dos Registros do Codificador RS

Os registradores na figura 9 recebem a soma do valor armazenado no registro anterior com a multiplicação do sinal *s_aux*. Esse sinal é a soma do dado de entrada com o registrador 15. Os multiplicadores foram implementados em uma biblioteca com várias funções, onde cada função corresponde a multiplicação de um símbolo por uma constante, por exemplo, a função $alpha\_120$ multiplica um símbolo por α^{120}.

O codificador foi implementado com interface baseada nos IP's do fabricante Altera, denominada Avalon, essa interface está descrita com mais detalhes na referência [16]. O bloco do codificador foi implementado com os pinos de entrada e saída mostrados na figura 10 e na tabela 5 os sinais são descritos detalhadamente.

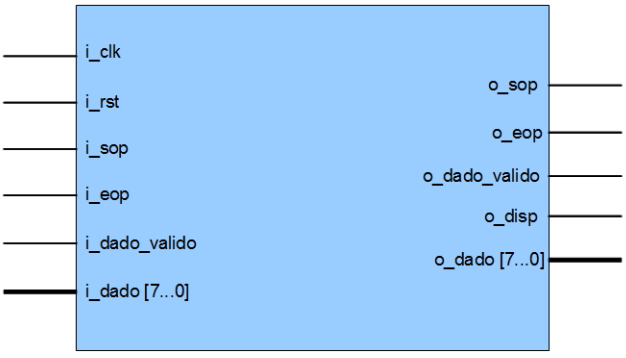


Figura 10: Bloco do Codificador Reed-Solomon

Na figura 11 mostra a simulação, no software ModelSim, do início da codificação de uma palavra. Todos os dados na simulação estão sendo mostrados no formato hexadecimal. Depois que o pino *i_rst*

Tabela 5: Descrição Funcional dos Pinos do Codificador Reed-Solomon

Pino	Tamanho (bits)	Direção	Descrição
i_clk	1	Entrada	Pino para entrada do sinal de relógio do sistema. O codificador funcionará na frequência deste sinal.
i_rst	1	Entrada	Inicializa e reiniciaza o sistema. Esse pino foi projetado para funcionar de forma assíncrona, não depende do sinal de relógio.
i_sop	1	Entrada	Pino para indicar o primeiro símbolo da palavra a ser codificada. Ela deve ser manter em nível lógico alto quando o primeiro símbolo estiver presente na porta i_dado.
i_eop	1	Entrada	Indica o final da palavra a ser codificada. Deve se manter em nível lógico alto quando o último símbolo da palavra estiver na porta i_dado.
i_dado_valido	1	Entrada	Indica que um dado válido está presente na porta i_dado.
i_dado	8	Entrada	Entrada dos dados para serem codificados.
o_sop	1	Saída	Indica o primeiro símbolo da palavra codificada.
o_eop	1	Saída	Indica o último símbolo da palavra codificada.
o_dado_valido	1	Saída	Quando em nível lógico alto os dados presentes na porta o_dado são válidos.
o_disp	1	Saída	Indica que o codificador está apto a receber dados a para serem codificados. Quando o codificador está ocupado, isto é, transmitindo os símbolos de paridade este pino fica em nível lógico baixo.
o_dado	8	Saída	Saída dos dados codificados.

vai para nível lógico baixo os dados começam a entrar no codificador a cada ciclo de relógio, que está representado pelo sinal *i_clk*. O sinal *i_dado_valido* indica que os dados contidos na porta *i_dado* são válidos. O pino *i_sop* indica o primeiro dado da palavra a ser codificada, que nesse caso é o 01.

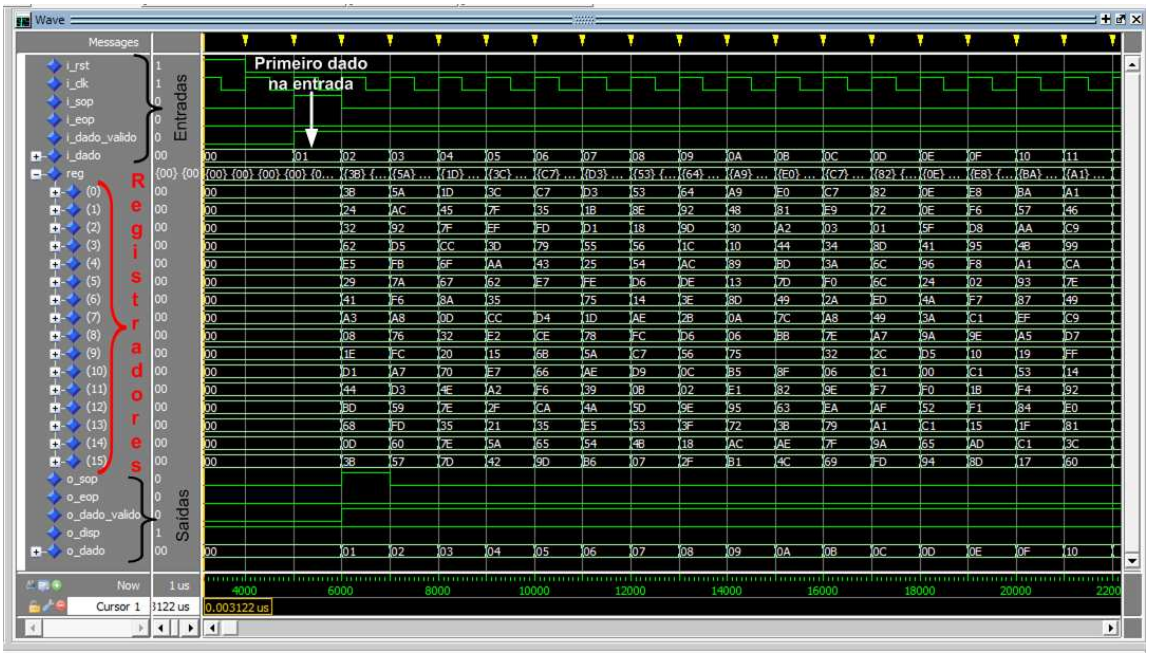


Figura 11: Simulação 1 do Codificador Reed-Solomon

Também está mostrada na figura 11 os valores dos registradores internos, que estão sendo alimentados pelos dados presentes na entrada. Depois de um ciclo de relógio o dado da entrada é repetido na saída e o pino *o_sop* indica o primeiro dado da mensagem está presente na saída *o_dado*. Na figura 12 temos o final do ciclo de codificação onde os valores os símbolos de paridade são inseridos na mensagem.

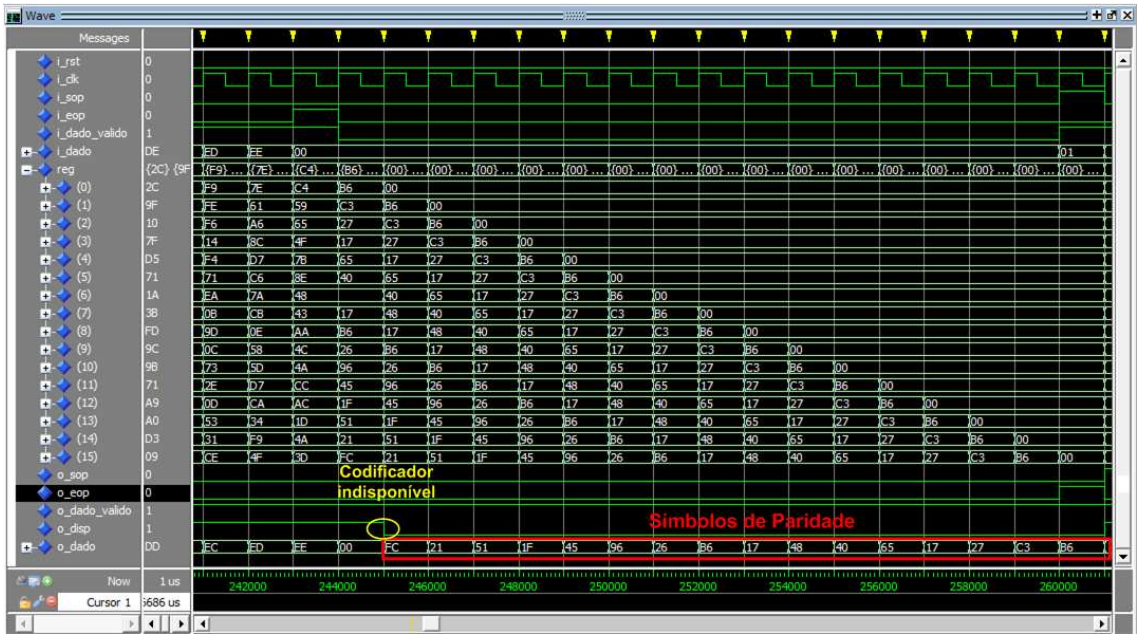


Figura 12: Simulação 2 do Codificador Reed-Solomon

Quando o último dado é entregue ao codificador, que nesse caso é o 00, indicado pelo sinal *i_eop*, os valores contidos nos registradores são enviados em seguida, esses símbolos são os de paridade. Após o último símbolo de paridade $B6$, sinalizado pelo pino *o_eop*, o codificador volta a estar disponível para o próximo processamento, pois seus registradores internos estão todos reinicializados com o valor 00. Existe um sinal nas figuras 11 e 12 nomeado de *o_disp*, que indica quando o codificador está apto a receber dados. O codificador não está apto a receber dados quando ele está inserindo os símbolos de paridade, por isso o sinal *o_disp* vai para nível lógico baixo, para indicar essa situação de indisponibilidade do codificador.

3.4 Decodificador Reed-Solomon

O decodificador implementado é composto por 5 principais módulos: memória, síndrome, algorítmo de Berlekamp-Massey, Chien e Forney. Em um decodificador convencional mostrado na figura 13 o próximo vetor teria que esperar até a atual palavra ser decodificada para começar seu processamento. Isso ocasiona a necessidade do uso do dobro de decodificadores para a norma G.709, enquanto um está ocupado decodificando deve existir outro que já esteja apto a receber os dados. O decodificador proposto nesse trabalho visa melhorar a performance em relação a área gasta no sistema total.

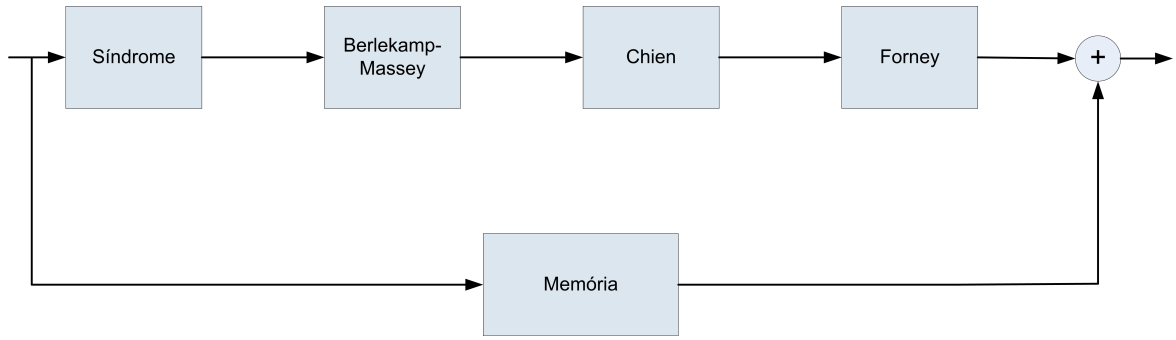


Figura 13: Decodificador Reed-Solomon Convencional

O decodificador proposto está mostrado na figura 14. Ele aumenta a eficiência, porque uma palavra código não precisa esperar a outra ser processada. O custo desse sistema mais eficiente é o aumento de um bloco demultiplexador um multiplexador e o uso de duas síndromes e duas memórias. A princípio a área gasta é maior no convencional, mas no sistema como um todo o uso do número de decodificadores diminui pela metade. Nas próximas seções serão discutidas a implementação de todos os blocos separadamente.

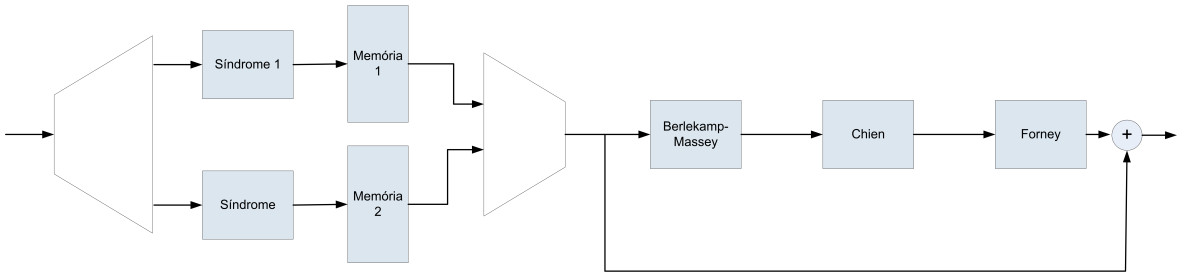


Figura 14: Decodificador Reed-Solomon Proposto

3.4.1 Síndrome

O bloco da síndrome deve verificar se as raízes do polinômio gerador são também raízes da palavra código. Se a palavra código satisfazer todas as raízes o vetor recebido é considerado livre de erros. Caso contrário o valor da síndrome e passado para o próximo módulo.

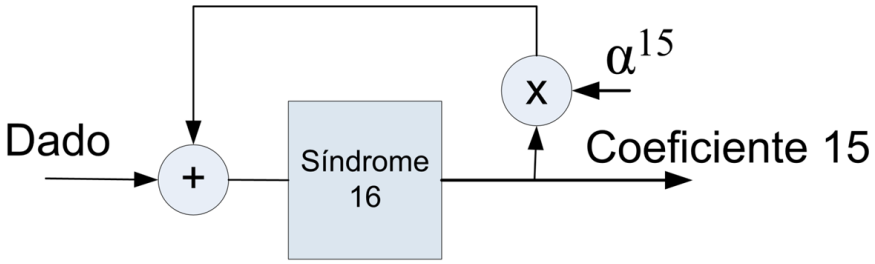
O circuito que realiza o cálculo da síndrome está mostrado na figura 16.

Cada um dos registradores verifica se uma raiz do polinômio gerador da equação 2.24 é raiz do vetor recebido. As 15 raízes do polinômio gerador são: $\alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, ..., \alpha^{13}, \alpha^{14}, \alpha^{15}$. Os valores dos registradores são multiplicados por α^i e somados com valores do vetor recebido a cada símbolo. Como exemplo o registrador Síndrome 16 da figura 16 realiza a função da equação 3.2 nos passos da equação 3.3.

$$S_{16} = r_{254}(\alpha^{15})^{254} + r_{253}(\alpha^{15})^{253} + r_{252}(\alpha^{15})^{252} + ... + r_0 \tag{3.2}$$

$$S_{16} = ((((r_{254}\alpha^{15} + r_{253})\alpha^{15}) + r_{252})\alpha^{15})... \tag{3.3}$$

Esse registrador foi implementado em código como mostrado na figura 15. A cada pulso do relógio o registrador *s_sin(16)* recebe o seu valor anterior multiplicado por α^{15}, mais o símbolo atual na entrada representado por *r_dado*. A multiplicação é realizada utilizando a função denominada de *alpha_15*. Sendo assim o valor do calculo da síndrome está pronto quando o último símbolo do vetor r_0 é registrado.




```
s_sin(16) <= alpha_015(s_sin(16)) xor r_dado;
```

Figura 15: Registrador 16 da Síndrome

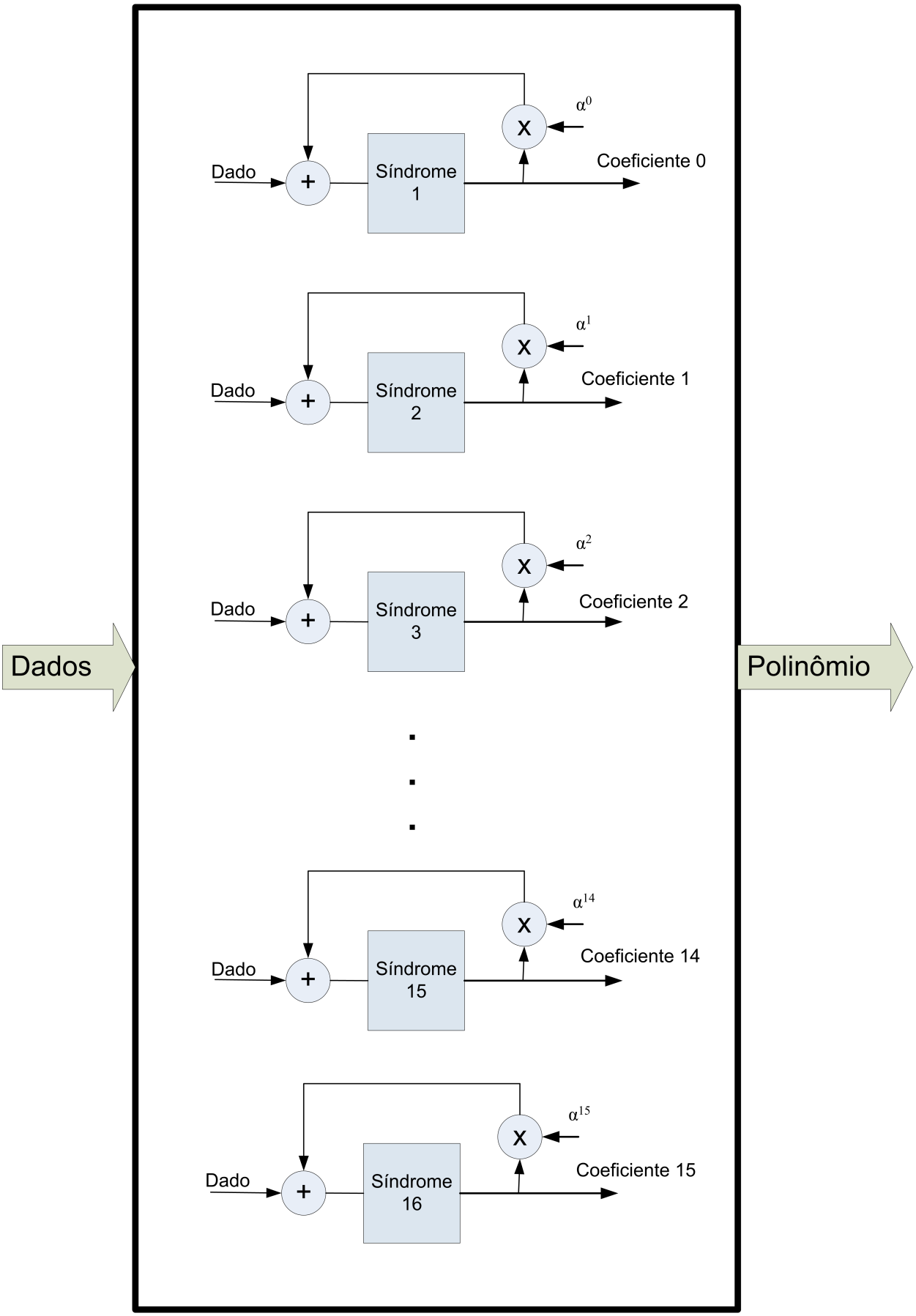


Figura 16: Circuito da Síndrome

A simulação da figura 17 mostra um vetor recebido livre de erros. As entradas do bloco são as mesmas do módulo principal, que é o decodificador. Os símbolos são inseridos na entrada *i_dado*, e eles são válidos quando o sinal *i_dado_valido* está em nível lógico alto. As entradas *i_sop* e *i_eop* indicam o início e final da mensagem respectivamente. O vetor é livre de erros porque todos os coeficientes da síndrome têm o valor 00. Assim todas as raízes do polinômio gerador $g(x)$ também são raízes do polinômio recebido $r(x)$. Também pode ser observado nas figuras 17 e 18 os valores dos registradores internos do módulo da síndrome.

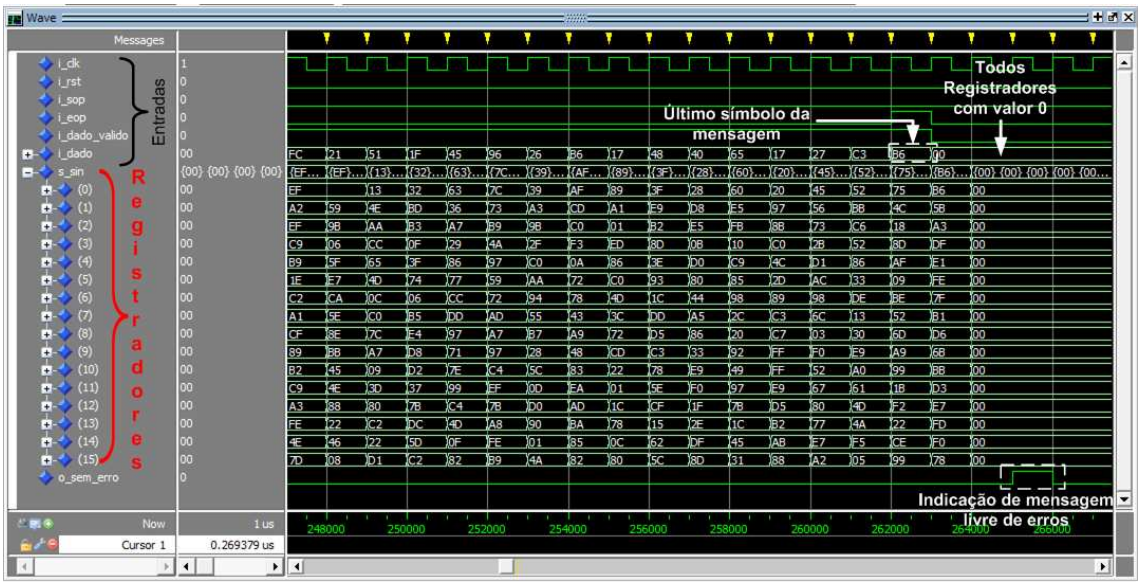


Figura 17: Simulação 1 da Síndrome

Quando o vetor é recebido sem erros o bloco da síndrome informa aos blocos sequentes, através do sinal *o_sem_erro*, para transmitir o pacote sem a necessidade de processamento. A figura 18 mostra outro cenário onde o vetor recebido contém erros.

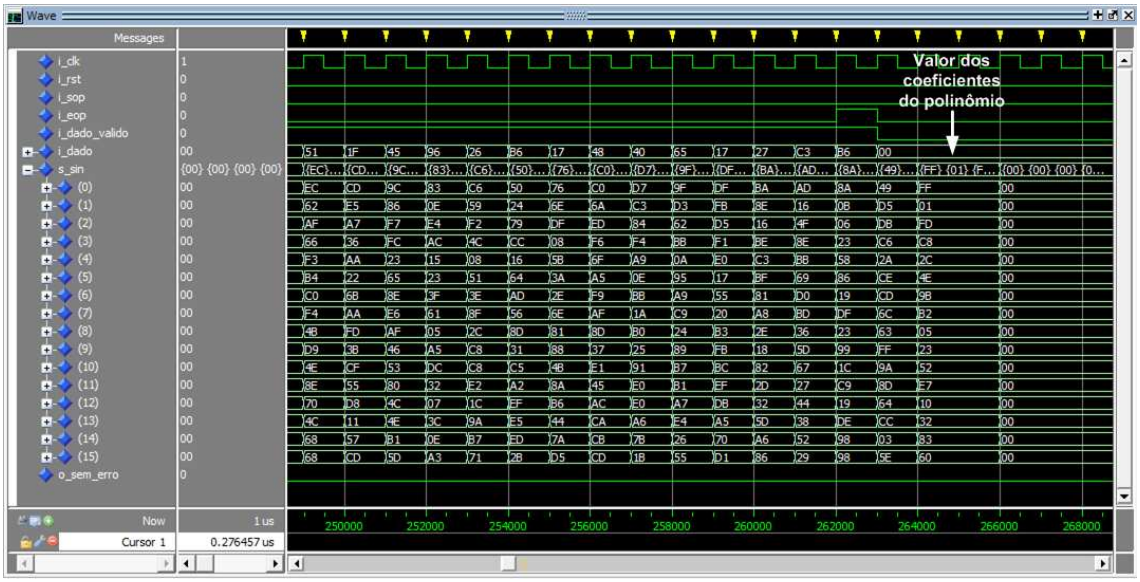


Figura 18: Simulação 2 da Síndrome

Nesse caso cada registrador contém um coeficiente do polinômio da síndrome, assim o bloco deve entregar para o módulo Berlekamp-Massey o valor armazenado, para que o mesmo possa encontrar os polinômios avaliador e localizador de erros.

3.4.2 Berlekamp-Massey

O módulo Berlekamp-Massey (BM) é responsável por encontrar o polinômio localizador de erros e o polinômio que contém a relação com os valores dos erros. Para implementação do módulo foi desenvolvido uma máquina de estados baseada no algoritmo do fluxograma da figura 4. A máquina de estados está mostrada na figura 19.

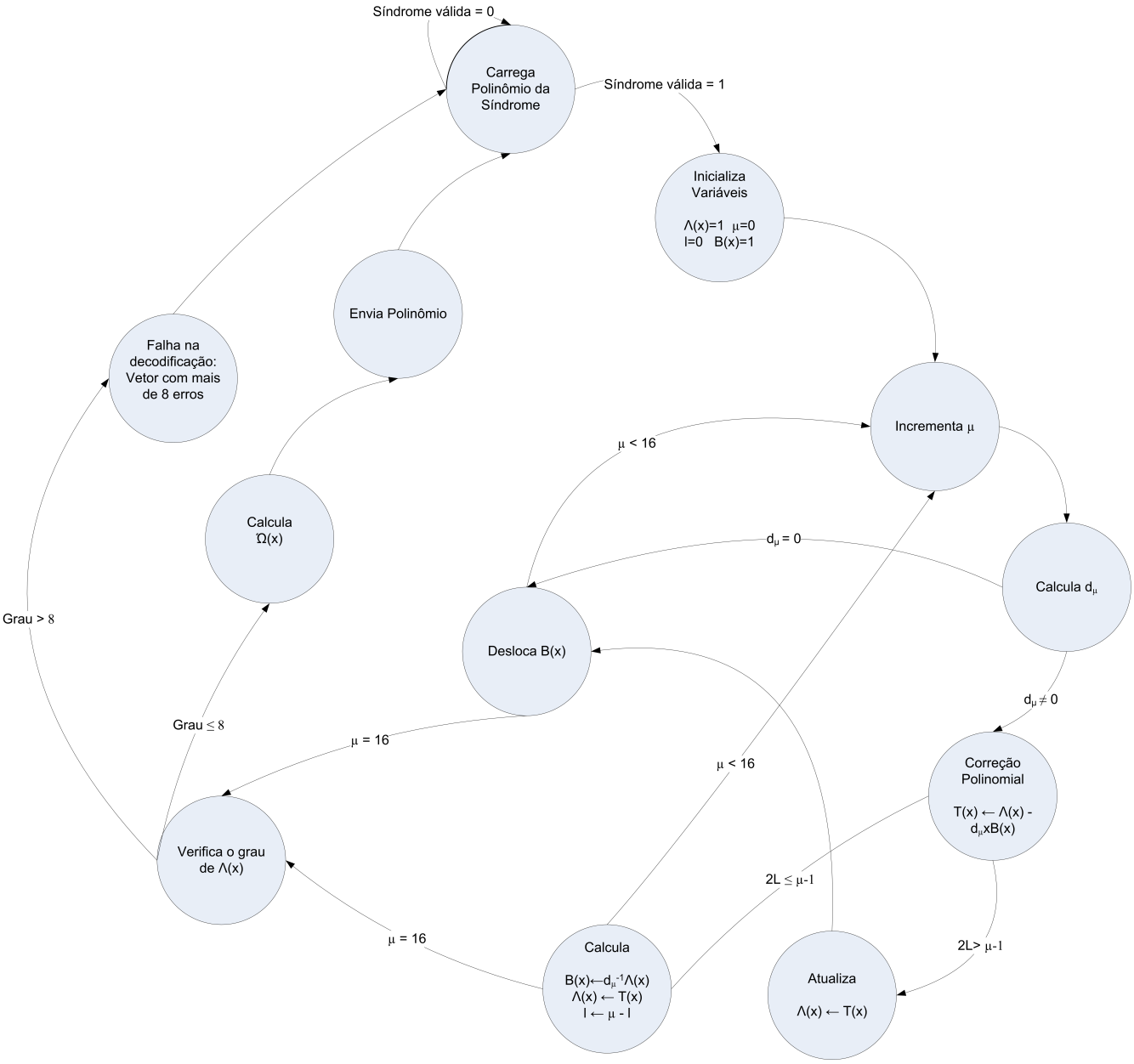


Figura 19: Diagrama da Máquina de Estados Berlekamp-Massey

Quando o bloco da síndrome termina o cálculo para um determinado vetor de símbolos, se o vetor contiver erros, a síndrome é passada para o módulo BM. Nesse ponto a máquina de estados BM carrega o valor da síndrome e inicia a operação. No segundo estado é inicializada todas as variáveis que irão ser necessárias no cálculo da equação chave. Seguindo o fluxo da máquina a variável μ é incrementada. Para divisão foi implementado uma ROM (Read Only Memory), denominada ROM inversa que contém todos os elementos do campo e seu respectivo inverso. Para implementar a ROM inversa foi utilizado blocos de RAM disponíveis na arquitetura da FPGA.

A figura 20 mostra a parte final da simulação do bloco Berlekamp-Massey, onde o sinal *Polinomio Valido* indica que os polinômios estão calculados e prontos para serem utilizados pelo bloco Chien. O registrador denominado como *Sindrome* contém os valores da síndrome que foi calculada pelo

seu respectivo bloco. O registrador *Lambda* contém os valores dos coeficientes do polinômio $\Lambda(x)$. Também é mostrado na figura 20 o estado *calcula_omega* onde o polinômio $\Omega(x)$ é obtido utilizando a equação 2.47. O sinal *Grau do Polinomio* é o grau do polinômio $\Lambda(x)$, ele indica a quantidade de erros contido no vetor recebido. Se esse sinal for maior que 8 o decodificador não é capaz de corrigir os erros, sendo assim o bloco deve avisar os blocos posteriores para enviar os dados sem corrigi-los.

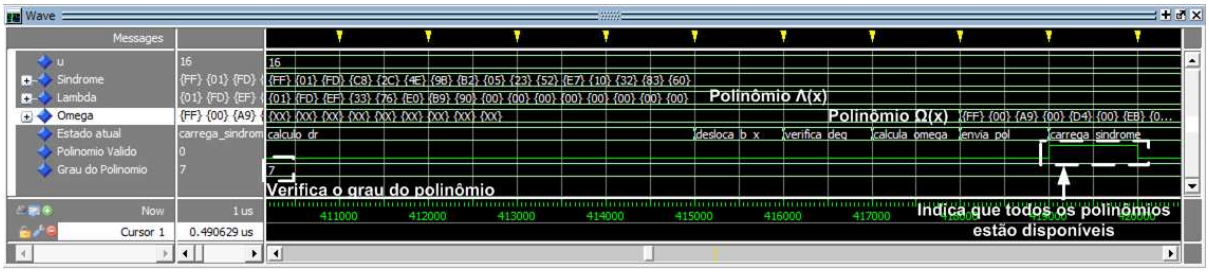


Figura 20: Simulação do Módulo Berlekamp-Massey

Quando o algoritmo alcança o passo $\mu = 16$, mostrado na figura 20 como *u*, o módulo volta para seu estado inicial esperando o próximo valor da síndrome de outros vetores com erros para reiniciar a máquina de estados.

3.4.3 Chien

Módulo Chien realiza um busca pelas raízes do polinômio localizador de erros e pelo polinômio que contém informações sobre os valores dos erros. O algoritmo avalia todos os 255 elementos não nulos pertencentes ao campo de Galois.

Para encontrar o local onde se encontra os erros no vetor de símbolos recebidos, o algoritmo Chien deve testar todas as raízes não nulas do campo de Galois $GF(2^8)$. Quando o algoritmo o resultado da equação é $\Lambda(\alpha^i) = 0$ o circuito encontrou o local de um erro no lugar $(n - i)$. O circuito que realiza a função de encontrar o lugar dos erros está mostrado na figura 21.

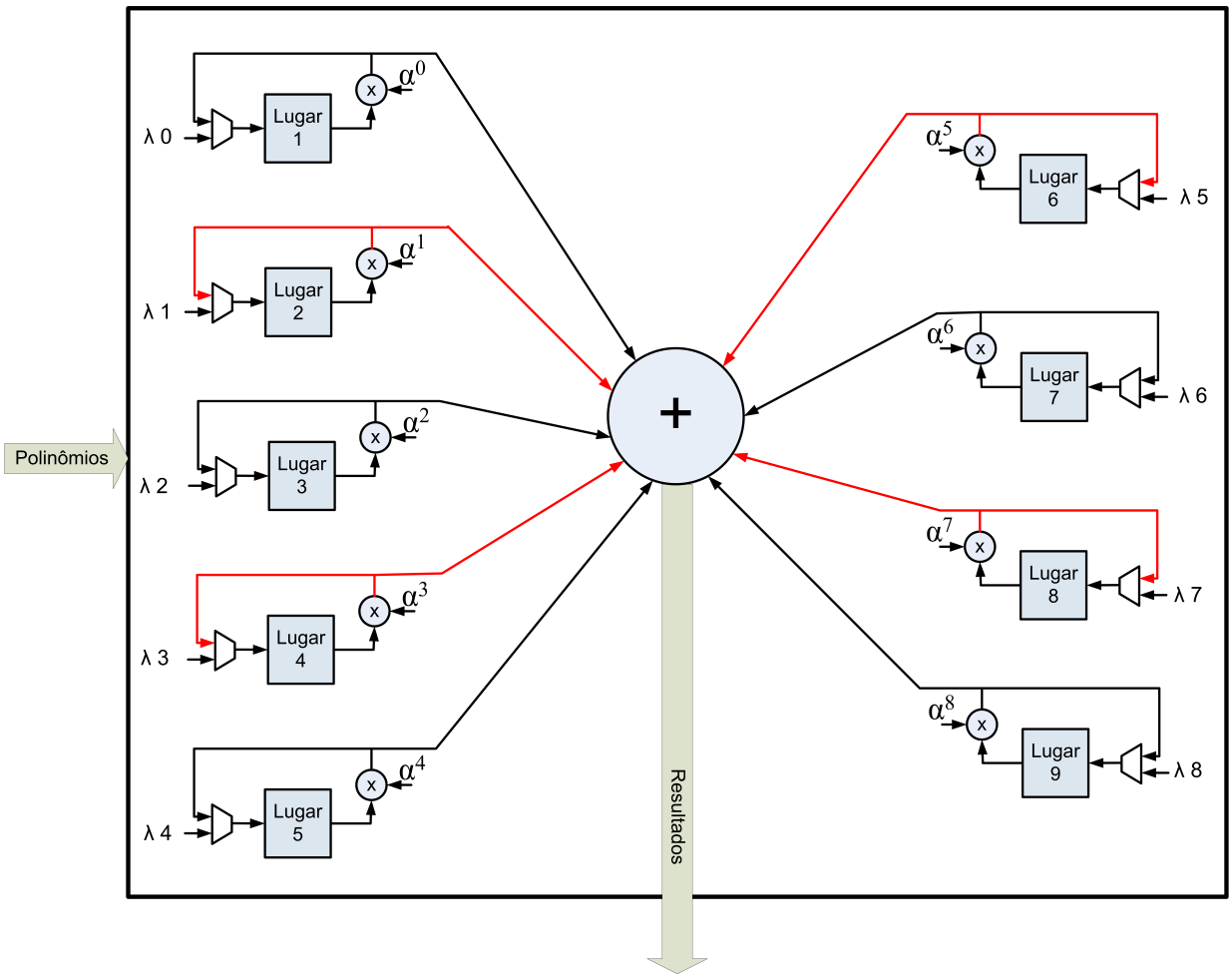


Figura 21: Módulo Localizador de Erro

Os registradores recebem inicialmente os valores dos coeficientes de $\Lambda(x)$. No próximo ciclo de relógio é calculado $\Lambda(\alpha^1)$, e no pulso seguinte $\Lambda(\alpha^2)$ até o $\Lambda(\alpha^{255})$. O circuito é baseado no conjunto de equação 3.5.

$$\begin{aligned}
\Lambda(x) &= \lambda X^0 + \lambda X^1 + \lambda X^2 + \lambda X^3 + \lambda X^4 + \lambda X^5 + \lambda X^6 + \lambda X^7 + \lambda X^8 \\
\Lambda(\alpha^1) &= \lambda\alpha^{1*0} + \lambda\alpha^{1*1} + \lambda\alpha^{1*2} + \lambda\alpha^{1*3} + \lambda\alpha^{1*4} + \lambda\alpha^{1*5} + \lambda\alpha^{1*6} + \lambda\alpha^{1*7} + \lambda\alpha^{1*8} \\
\Lambda(\alpha^2) &= \lambda\alpha^{2*0} + \lambda\alpha^{2*1} + \lambda\alpha^{2*2} + \lambda\alpha^{2*3} + \lambda\alpha^{2*4} + \lambda\alpha^{2*5} + \lambda\alpha^{2*6} + \lambda\alpha^{2*7} + \lambda\alpha^{2*8} \quad (3.4) \\
&\vdots \\
\Lambda(\alpha^{255}) &= \lambda\alpha^{255*0} + \lambda\alpha^{255*1} + \lambda\alpha^{255*2} + \lambda\alpha^{255*3} + \lambda\alpha^{255*4} + \lambda\alpha^{255*5} + \lambda\alpha^{255*6} \\
&+ \lambda\alpha^{255*7} + \lambda\alpha^{255*8}
\end{aligned}$$

Este circuito também nos fornece o resultado da derivada formal de $\Lambda(\alpha^i)$, calculada levando em conta os valores dos coeficientes ímpares do polinômio destacado na figura 21 em linhas verme-

lhas. Está mostrado na figura 22 o código utilizado para síntese do resultado do polinômio $\Lambda(x)$ e a derivada do mesmo. O sinal *resultado_lambda* armazena o somatório de todos os registradores denominados de *Lugar* mostrado na figura 21, e esse somatório é o resultado do polinômio $\Lambda(x)$ para um determinado símbolo. O cálculo da derivada de $\Lambda(x)$ é armazenado no sinal *resultado_deriv_lambda*, onde ele realiza o somatório dos registradores 2, 4, 6 e 8. A simulação do módulo está mostrado na figura 24. Na simulação está mostrado o sinal *i*, que indica a potência de α. O sinal *Lambda* armazena os valores dos registradores denominados *Lugar* na figura 21.

```
resultado_lambda <= lugar(1) xor lugar(2) xor lugar(3) xor lugar(4) xor lugar(5)
                    xor lugar(6) xor lugar(7) xor lugar(8) xor lugar(9) ;


resultado_deriv_lambda <= lugar(2) xor lugar(4) xor lugar(6) xor lugar(8);
```

Figura 22: Código VHDL da Implementação da Resolução de $\Lambda(x)$ e sua Derivada

Na simulação da figura 24 os sinais *resultado de Lambda* e *Derivada de Lambda* é o resultado da resolução do polinômio $\Lambda(x)$ e sua derivada, mostrado na figura 22. Pode-se notar que no vetor recebido existem 7 erros. Esses erros estão contidos nas posições 255, 254, 253, 252, 251, 250, 249, porque de $i = 1$ até $i = 7$ o valor do sinal *Resultado de Lambda* que está representando o valor de $\Lambda(\alpha^i)$ é igual a 0. Os valores dos sinais *resultado de Lambda* e *Derivada de Lambda* irão ser usados posteriormente no módulo Forney.

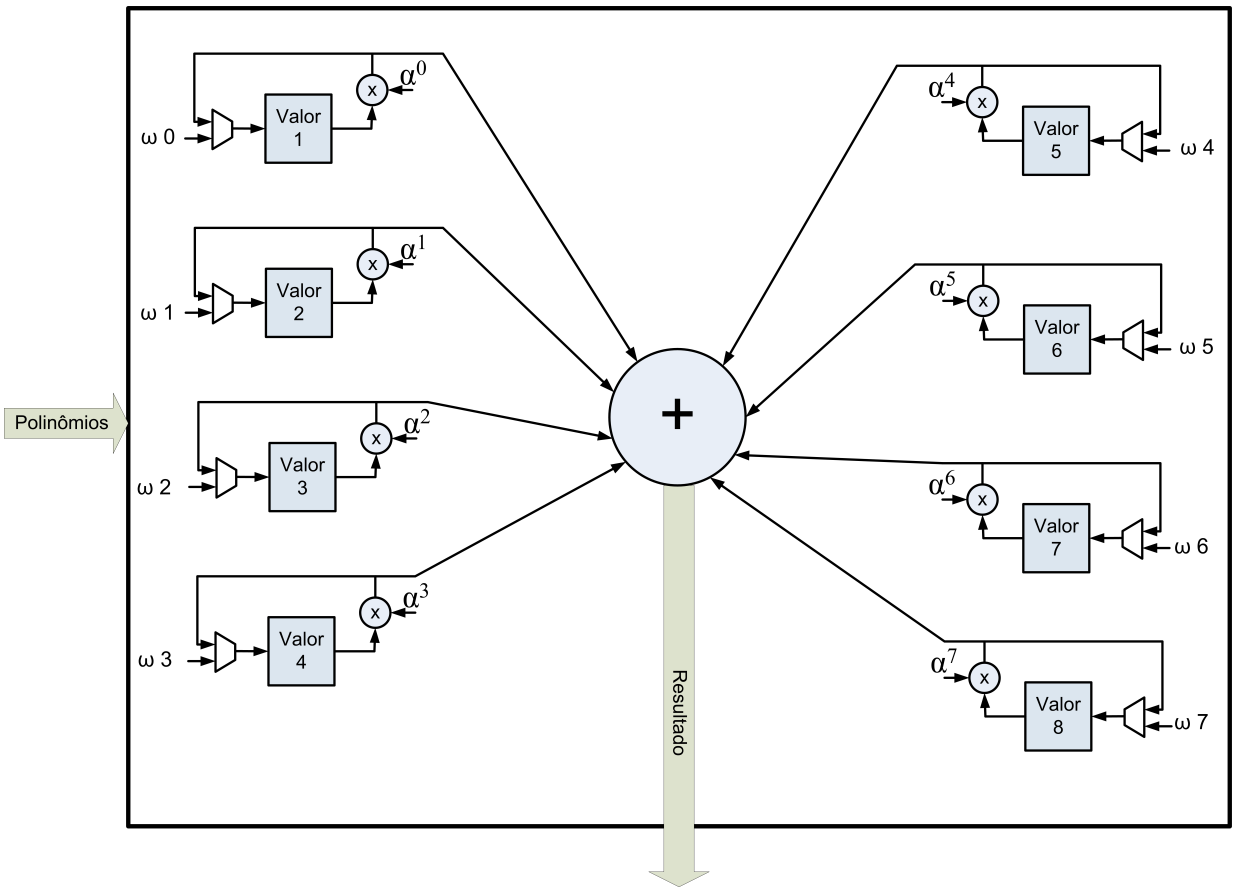


Figura 23: Módulo Avaliador de Erro

O circuito que calcula o resultado do polinômio avaliador de erro mostrado na figura 23 é semelhante ao localizador de erros. No primeiro momento é carregado os coeficientes de $\Omega(x)$, depois é calculado $\Omega(\alpha^1)$, $\Omega(\alpha^2)$, até o $\Omega(\alpha^{255})$. Os dois módulos trabalham paralelamente porque quando localizador de erros encontrar um erro, $\Lambda(\alpha^i) = 0$, o resultado $\Omega(\alpha^i)$ de irá ser usado no bloco Forney para encontrar o valor do erro que irá ser adicionado ao símbolo do vetor recebido para sua correção. Na figura 25 mostra a simulação do circuito que faz a busca no polinômio $\Omega(x)$.

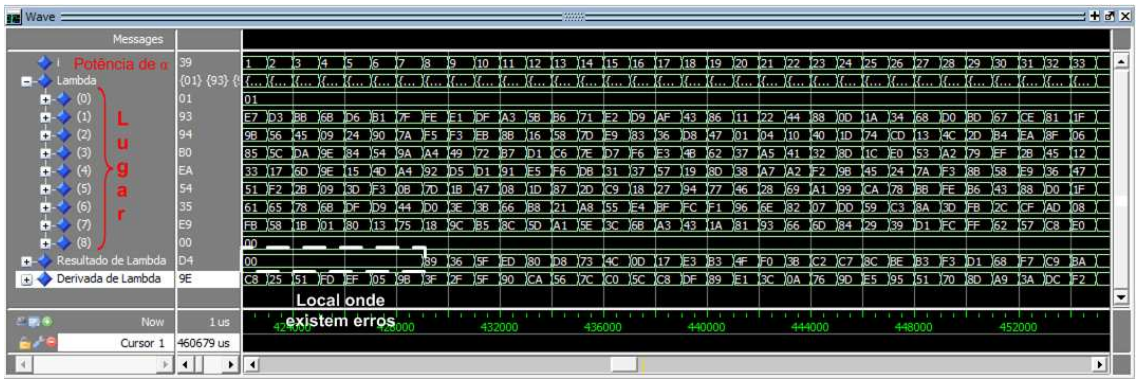


Figura 24: Simulação 1 do Módulo Chien Search

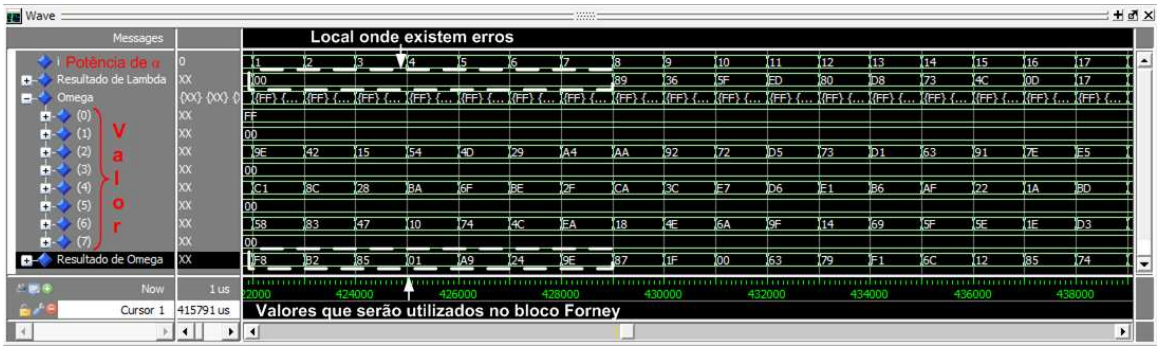


Figura 25: Simulação 2 do Módulo Chien Search

O registro denominado como Omega contém o resultado dos registradores denominados *Lugar* da figura 23. O sinal *Resultado de Omega* é o resultado de $\Omega(\alpha^i)$. Quando o sinal *Resultado de Lambda* for igual a 00 o sinal *Resultado de Omega* será utilizado no bloco Forney para cálculo do valor dos erros.

3.4.4 Forney

O módulo Forney realiza o último passo da decodificação. Ele é responsável por interpretar os valores fornecidos pelo bloco Chien e corrigir os erros. Os valor do erro é determinado pela equação 2.49.

O circuito que realiza essa função está mostrado na figura 26.

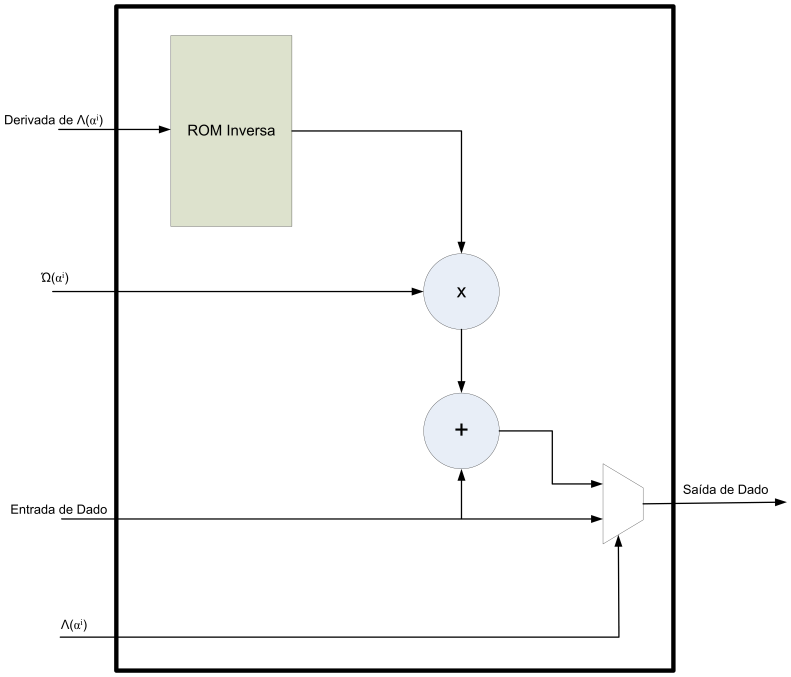


Figura 26: Módulo Forney

Quando a entrada $\Lambda(\alpha^i)$ tem o valor 00 significa que naquele lugar existe um erro para ser corrigido. O algoritmo Forney deve encontrar o valor do erro e adicionar ao dado recebido naquela posição. O bloco utiliza uma memória, denominada ROM inversa, para calcular o inverso da derivada de $\Lambda(\alpha^i)$. O inverso do elemento é calculado para utilizar a operação de multiplicação que é mais simples do que a divisão.

Caso o valor da entrada Resultado de $\Lambda(\alpha^i)$ seja diferente de 00 o algoritmo deve apenas repetir o dado sem nenhuma alteração, pois, naquele lugar não existe nenhum erro. A figura 27 demonstra a parte de maior importância no módulo Forney. O sinal *resultado_mult* armazena a multiplicação da saída da ROM inversa, *q_rom*, com o resultado do polinômio $\Omega(x)$, *omega*. Na sequência do código da figura 27 obtem o multiplexador que funciona da seguinte maneira: quando $\Lambda(x)$, representado pelo sinal *lambda*, for igual a 0, a saída *dado_corrigido* recebe o dado de entrada, *i_dado*, mais o sinal *resultado_mult*. Caso contrário o símbolo não contém erro e a saída *dado_corrigido* recebe a entrada *i_dado*.

```
resultado_mult <= gf_mult_gen(q_rom, omega);

  if lambda = X"00" then
      dado_corrigido <= i_dado xor resultado_mult;
  else
      dado_corrigido <= i_dado;
  end if;
```

Figura 27: Código da Implementação do Multiplexador do Módulo Forney

A simulação do bloco Forney está mostrado na figura 28.

Figura 28: Simulação do Módulo Forney

Essa simulação mostra as entradas dos resultados dos polinômios e a entrada de dados. Quando a entrada *Lambda* é igual a 00 o algoritmo soma o sinal *Valor do Erro* ao dado a ser corrigido, nesse caso o valor dos erros é FF, ou seja, todos os bits dos 7 primeiros símbolos estão invertidos. No módulo Forney também foi implementado uma saída para controle dos erros das mensagens. Esse sinal é denominado como *Numero de Erros*, ele contabiliza a quantidade de erros contidos em cada

mensagem. Ele atualiza seu valor no momento que o erro é encontrado, e reinicializa quando começa o processamento de uma mensagem posterior. Por fim a saída *Dado Corrigido* contém os valores dos símbolos da mensagem corrigida.

3.4.5 Interface do Decodificador

A interface utilizada no bloco do decodificador também foi baseado na interface Avalon da Altera. A interface está mostrada na figura 29.

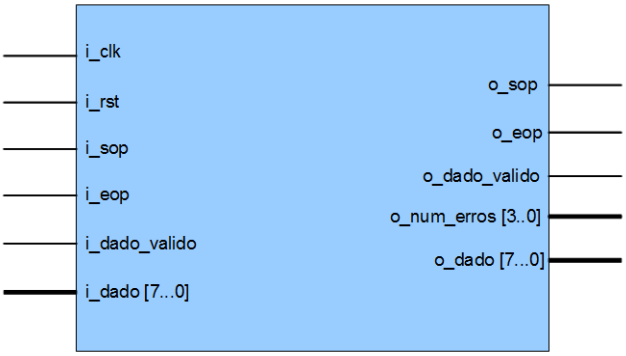


Figura 29: Bloco Decodificador Reed-Solomon

Os sinais da interface de entrada e saída da figura 29 estão descritos na tabela 6

Os sinais da tabela 6 são basicamente iguais ao da interface do decodificador, a única mudança é o acréscimo do sinal o_num_erros, que indica a quantidade de símbolos errados que estão presentes do vetor que está sendo processado. O sinal é incrementado toda vez que um erro é encontrado, ou seja, todo tempo em que $\Lambda(\alpha^i) = 00$.

3.5 Implementação em Hardware

O circuito codificador e decodificador foi prototipado em hardware utilizando a placa de desenvolvimento NIOS II do fabricante Altera. A placa tem como seu principal componente a FPGA Stratix II EP2S60F672C3. Dentre as principais características se destacam:

- 672 pinos de entrada e saída
- 48352 Look-up tables
- 36 circuitos DSP's
- 6 PLLs

Tabela 6: Descrição Funcional dos Pinos do Decodificador Reed-Solomon

Pino	Tamanho (bits)	Direção	Descrição
i_clk	1	Entrada	Pino para entrada do sinal de relógio do sistema. O decodificador funcionará na frequência deste sinal.
i_rst	1	Entrada	Inicializa e reiniciaza o sistema. Esse pino foi projetado para funcionar de forma assíncrona, não depende do sinal de relógio.
i_sop	1	Entrada	Pino para indicar o primeiro símbolo da palavra a ser decodificada. Ela deve ser manter em nível lógico alto quando o primeiro símbolo estiver presente na porta i_dado.
i_eop	1	Entrada	Indica o último símbolo a ser decodificado. Deve se manter em nível lógico alto quando o último símbolo da palavra estiver na porta i_dado.
i_dado_valido	1	Entrada	Indica que um dado válido está presente na porta i_dado.
i_dado	8	Entrada	Entrada dos dados para serem decodificados.
o_sop	1	Saída	Indica o primeiro símbolo do vetor decodificado.
o_eop	1	Saída	Indica o último símbolo da palavra decodificada.
o_dado_valido	1	Saída	Quando em nível lógico alto os dados presentes na porta o_dado são válidos.
o_num_erros	1	Saída	Indica o número de erros contidos no vetor decodificado. Ela é atualizada assim que o erro é encontrado
o_dado	8	Saída	Saída dos dados codificados.

- 2.5 mega bits de memória
- 16 linhas de distribuição de clock

Para síntese do circuito foi utilizado o software Quartus II fornecido pelo mesmo fabricante da placa. Como resultado da síntese obteve os resultados mostrados do codificador e decodificador na tabela 7 e 8 respectivamente.

Tabela 7: Resultado da Síntese do codificador RS

Look-up Table	Registros	Memória (bits)
100	152	0

Tabela 8: Resultado da Síntese do Decodificador RS

Módulo	Look-up Table	Registros	Memória (bits)
Demultiplex	1	23	0
Síndrome 1	235	271	0
Síndrome 2	235	271	0
Multiplex	4	133	0
Berkamp-Massey	2966	885	2048
Chein Search	242	305	0
Forney	62	40	2048
Memória 1	0	0	2048
Memória 2	0	0	2048
Total	3510	1657	8192

O circuito do codificador utilizou poucos elementos lógicos para sua implementação devido a simplicidade do módulo. Referente ao modelo de decodificador proposto, adicionando blocos que utilizam pouca lógica, Demultiplex, Multiplex, Síndrome 2 e Memória 2, tem-se um aproveitamento melhor dos blocos que utilizam mais lógica como o Berlekamp-Massey.

Para validação em hardware foram criados dois blocos adicionais, um gerador de dados e um insersor de erros. O gerador de dados fornece para a entrada de dados do codificador valores de um contador na faixa de 0 à 238. Ele também fornece todos os sinais necessários na interface do codificador. O insersor de erros fica localizado entre o codificador e o decodificador. Ele insere a quantidade máxima de erros que o decodificador pode corrigir, que são 8 símbolos, em lugares aleatórios do vetor codificado. O sistema para validação está mostrado na figura 30.

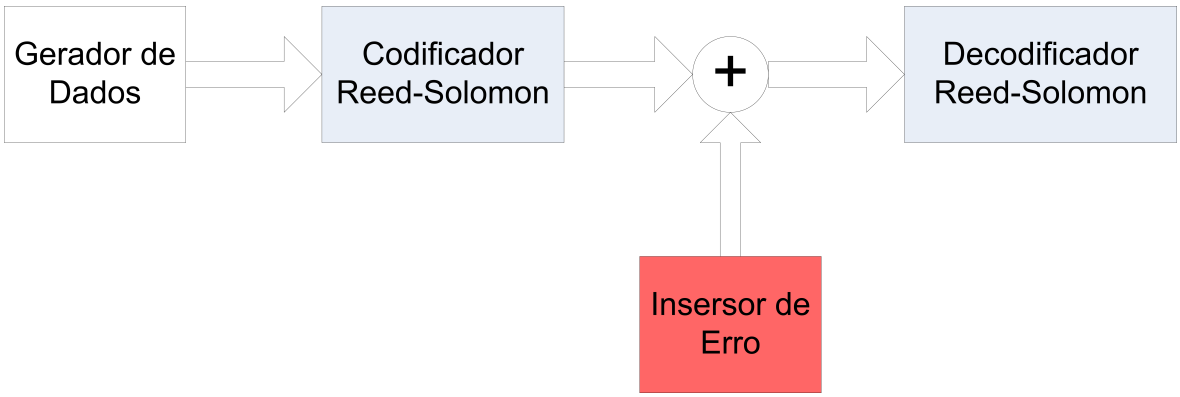


Figura 30: Sistema de Validação do Hardware

Os sinais foram analisados utilizando a ferramenta SignalTap disponibilizado pelo software Quartus II. O SignalTap permite analisar os sinais com o dispositivo em funcionamento. Isso simplifica o fluxo de projeto de modo que não é necessário o emprego de um analisador lógico. Para o codificador foi capturado os dois principais momentos, o início de um pacote entrando no codificador e o momento onde o codificador insere os símbolos de paridade. A figura 31 mostra o início do vetor de dados que está sendo recebido e retransmitido pelo codificador, já a figura 32 mostra o final do mesmo pacote, onde está sendo inseridos os símbolos de paridade.

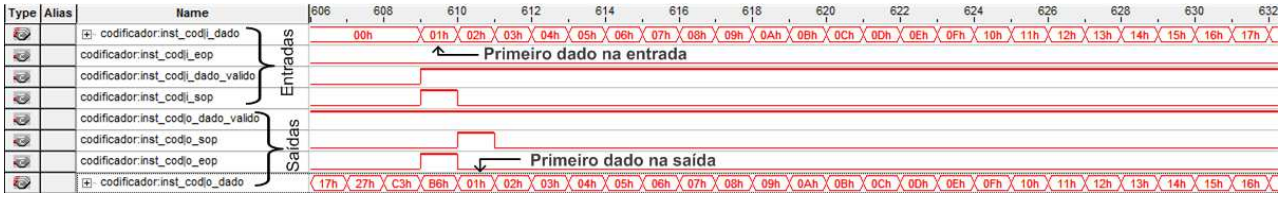


Figura 31: Vetor 1 de Dados do Codificador RS

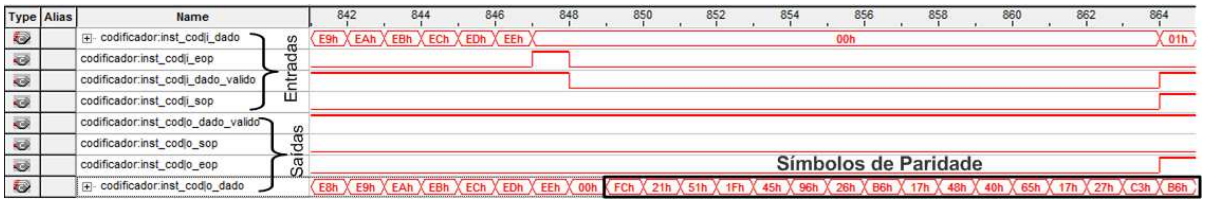


Figura 32: Vetor 2 de Dados do Codificador RS

O primeiro símbolo do vetor que será codificado é o 01 discriminado na entrada *i_dado* junto com o sinal *i_sop* e a entrada *i_dado_valido* em nível lógico alto. Logo após um ciclo de relógio o mesmo símbolo 01 é disposto na saída *o_dado* do codificador e é sinalizado como o primeiro símbolo da mensagem pelo sinal *o_sop* e também pela saída *o_dado_valido* que indica que o mesmo é um dado válido. Depois os símbolos da entrada são repetidos na saída na ordem correta de transmissão.

Na figura 32 destacam-se os símbolos de paridade sendo inseridos na mensagem logo após o último símbolo recebido, o símbolo 00. Os símbolos de paridade são os mesmos obtidos na simulação mostrada na figura 12, eles começam em FC e terminam em $B6$. O sinal *o_epo* sinaliza o final da mensagem codificada que é o momento em que terminam os símbolos de paridade. Após o final da mensagem o codificador está apto para receber a próxima mensagem.

Nas figuras 33 e 34 mostram os dados adquiridos nas entradas e saídas do decodificador.

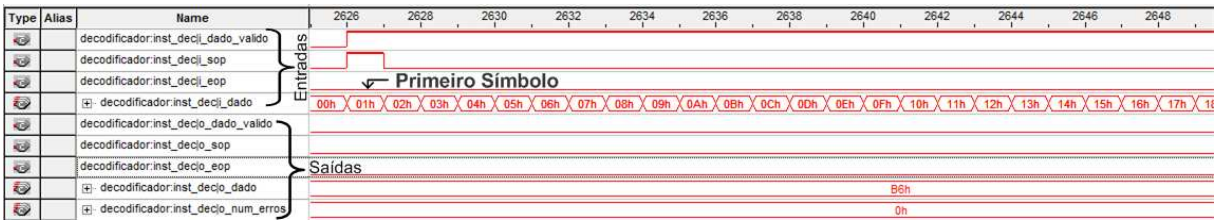


Figura 33: Vetor 1 de Dados do Decodificador RS

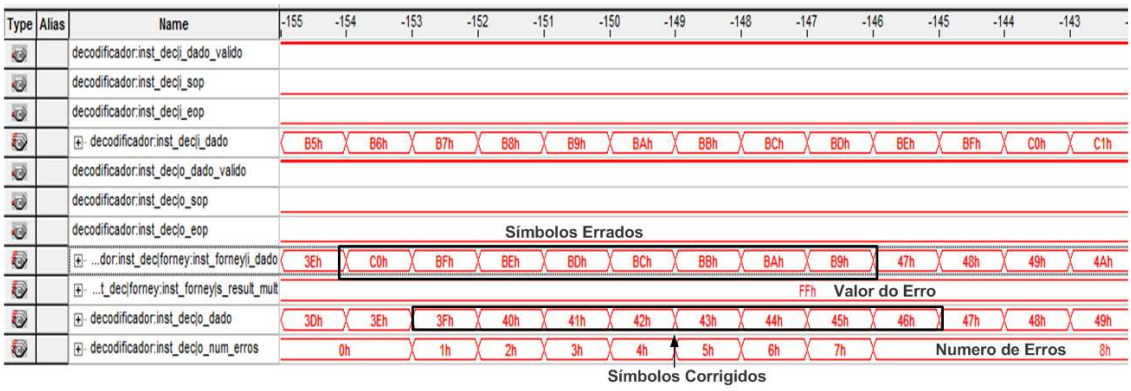


Figura 34: Vetor 2 de Dados do Decodificador RS

Na figura 33 tem-se o início do vetor recebido. Pode-se notar que a mensagem capturada é a mesma capturada no codificador mostrada nas figuras 31 e 32, onde também o primeiro símbolo da mensagem está sinalizado pela entrada i_sop. Os símbolos não são entreges instantaneamente na saída do decodificador, pois, o decodificador tem uma latênca para processar os dados.

Na figura 34 tem-se o momento de maior importância no funcionamento do decodificador, onde ele corrige os símbolos errados. Tem-se os símbolos errados na entrada do decodificador e o valor dos mesmos, que no caso da figura 34 é FF, o que significa que todos os bits dos símbolos estão invertidos. O decodificador realiza a soma do valor dos erros com o dado errado na entrada e obtem os símbolos corretos da mensagem codificada, essa operação está mostrado na figura 35. Também na figura 34 pode-se destacar que a cada momento que um símbolo é corrigido a saída *o_num_erros* informa a quantidade de erros presente na mensagem.

Figura 35: Correção dos Símbolos Errados no Decodificador

Na figura 35 a soma dos dados errados com o valor do erro obtem-se os dados corrigidos. A sequência dos dados obedece a sequência gerada na fonte de dados, que é um contador, com isso pode-se concluir que o decodificador está funcionando de forma correta.

4 Conclusões e Trabalhos Futuros

Esta dissertação apresentou a implementação em hardware do codificador e decodificador Reed-Solomon RS(255,239) utilizado em redes de fibra óptica. O trabalho propôs uma nova arquitetura para o decodificador que reduz a área utilizada no total do sistema na FPGA. A redução foi alcançada pela reutilização dos blocos que utilizam maior área, diminuindo o tempo de ociosidade do sistema. Os circuitos foram simulados e implementados de forma satisfatória na FPGA Stratix II.

O desenvolvimento dessa tecnologia possibilita uma redução de custos na implementação do processador de dados para as redes ópticas, porque para implementação do circuito RS seria necessário a compra de Propriedades Intelectuais que podem chegar a preços altos de até U$ 5.000,00.

As ferramentas que foram utilizadas são pertencentes ao fabricante Altera, porque para implementação foi utilizando o kit distribuído pela própria empresa. Porém nada impede que o codificador e decodificador sejam implementados em outra tecnologia disponibilizadas por outros fabricantes. A razão dessa independência tecnológica deve-se ao fato de todo o circuito ser implementado em VHDL padronizado, sem particularidades de fabricante. Futuramente o circuito pode também ser implementado em ASIC, o que levaria a uma alternativa mais eficiente e, dependendo da demanda, com maior viabilidade econômica.

Uma sugestão de otimização que pode ser realizada no decodificador é no módulo Berlekamp-Massey. O bloco implementado foi desenvolvido em alto nível de abstração, o que não permitiu otimizações no circuito. Um módulo mais eficiente, que pode ser encontrado na referência [1], pode ser incorporado nesse trabalho para melhora na utilização da área, visto que o objetivo da dissertação foi a melhora na arquitetura não nos blocos individuais. Nesse bloco é necessário a realização de alguns ajustes para melhora da performance em relação a frequência de utilização. Existem caminho longos de lógica combinacional onde deve ser iseridos alguns estágios registrados.

Para montar o processador de quadros OTN deve ser arranjado no sistema os codificadores e decodificadores de forma paralela para que o deserializador e serializador processe todos os pacotes do quadro que acontece de forma paralela devido as altas taxas de dados provenientes da fibra óptica. O sistema com todos os componentes nescessários para realização da correção de erros no quandro OTN é uma sugestão de continuação do trabalho.

Referências

[1] Rodolfo A. H. L., Silva T. A. Implementação de uma arquitetura reed-solomon para uso em redes otn 10.7 gbps. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Dezembro 2007.

[2] Souza M. A., Câmara A. O. Códigos corretores de erros lineares. *FAMAT em Revista*, Maio 2006.

[3] Morelos-Zaragoza R. H. *The Art of Error Correcting Coding*. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8S, Inglaterra, 2 edition, 2006.

[4] Ingale M. A. Error correcting codes in optical communication systems. Master's thesis, Chalmers University, Gothenburg, Suécia, Janeiro 2003.

[5] Ferreira A. Aplicação da teoria dos campos de galois na codificação de canal. Master's thesis, Instituto Superior de Engenharia de Lisboa, 1999.

[6] Farrell J. C., Moreira P. G. *Essentials of Error-Control Conding*. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8S, Inglaterra, 2006.

[7] Costello S., Lin D. J. *Error Control Coding*. Pearson Prentice Hall, Upper Saddle River, NJ, 2 edition, 2004.

[8] Sklar B. Reed-solomon codes.

[9] Yang K. C. C., Wai S. J. Field programmable gate array implementation of reed-solomon code, rs(255,239).

[10] Itu-t g.709/y.1331, interfaces for optical transport network (otn). 2001.

[11] Kuo I. Lan M., Song S. A low complexity design of reed solomon code algorithm for advanced raid system. *Consumer Electronics, IEEE Transactions*, 2007.

[12] Vot B. Mallet V. K. Bhargava T., Le-Ngoct M. T. A gate-array-based programmable reed-solomon codec: Struture-implementation-applications. *Military Comunications Conference, A New Era*, 1990.

[13] Altera. About stratix series high-end fpgas. 2010.

[14] d'Amore R. *VHDL Descrição e Síntese de Circuitos Digitais*. Livros Técnicos e Científicos Editora S. A., Rio de Janeiro, RJ, 2005.

[15] Koç A., Halbutogullari Ç. K. Mastrovito multiplier for general irreducible polynomials. *IEEE Transactions on Computers, vol 49*, Maio 2000.

[16] Altera. Avalon interface specifications. 2009.

APÊNDICE A – Artigos Publicados

Foram publicados dois artigos relacionados com a dissertação. O primeiro no 16^o congresso internacional Iberchip Workshop (IWS), realizado na cidade de Foz do Iguaçu no dia 23 de Fevereiro de 2010. O segundo artigo foi apresentado no 6^o International Conference on Wireless Communications, Networking and Mobile Computing, na cidade Chengdu, China em 25 de Setembro de 2010. Os artigos foram intitulados respectivamente como:

A Reed-Solomon CODEC for OTN G.709 Standard with Reduced Decoder FPGA Area

Tiago Barbosa, Robson Moreno and Tales Pimenta

FPGA Implementation of a Reed-Solomon CODEC for OTN G.709 Standard with Reduced Decoder Area

Tiago Barbosa, Robson Moreno and Tales Pimenta