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

THULLYO DENNIER CASTRO REIS FERREIRA

Uma Arquitetura Reed-Solomon baseada em FPGA/Soft-core

Itajubá, junho de 2011.

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

Thullyo Dennier Castro Reis Ferreira

Uma Arquitetura Reed-Solomon baseado em FPGA/Soft-core

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: Microeletrônica

Orientador: Prof. Dr. Robson Luiz Moreno

Co-orientador: Prof. Dr. Luis Henrique de Carvalho Ferreira

Junho de 2011
Itajubá - MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

Dedico este trabalho à minha amada família:

Donizete, Valcirlene, Walgney, Thawber, Salvador Neto e Gabryella.

"Riquezas e glória vêm de diante de Ti, e Tu dominas sobre tudo, e na Tua mão há força e poder; e na Tua mão está o engrandecer e o dar força a tudo. (...)
Como a sombra são os nossos dias sobre a terra,
e sem Ti não há esperança."
(I Cr 29:11-16)

Agradecimentos

Agradeço, primeiramente, a Deus por me capacitar a concluir este trabalho com êxito. “Porque o SENHOR dá a sabedoria; da sua boca é que vem o conhecimento e o entendimento.” (Pv 2:6)

À minha família por ser meu suporte e auxílio em todo o tempo.

Agradeço a meus professores e orientadores, Robson Luiz Moreno e Luis Henrique Carvalho, pelo companheirismo, ajuda e esclarecimentos sem os quais a realização deste trabalho não seria possível.

Aos professores Tales Cleber Pimenta e Paulo César Crepaldi que, com valiosas sugestões, me auxiliaram durante a revisão do texto deste trabalho.

Aos colegas do Grupo de Microeletrônica pelo companheirismo, sugestões e contribuições que permitiram o aperfeiçoamento deste.

Ao CNPq, CAPES e FAPEMIG que, através do suporte financeiro, viabilizaram a realização desse trabalho.

Meus mais sinceros agradecimentos.

Resumo

O objetivo deste trabalho é avaliar e projetar uma arquitetura Reed-Solomon (codificador e decodificador), uma das mais utilizadas em sistemas digitais. Esta arquitetura será implementada utilizando processador *soft-core* e desenvolvida em *Field Programmable Gate Array – FPGA*. Esta arquitetura visa à economia da potência consumida pelo transmissor e à confiabilidade oferecida pelo uso de códigos corretores de erro tornando a comunicação mais robusta em ambientes ruidosos. Neste trabalho, os códigos apresentados proporcionam uma melhoria no desempenho do *Bit Error Rate – BER*, podendo citar como aplicações o uso de redes de sensores sem fio (monitoramento remoto) e aplicações médicas. Portanto, serão apresentados resultados que mostram o ganho de desempenho do BER obtidos nas implementações em *software* aumentando o alcance da comunicação ou reduzindo o número de nós sensores para cobrir uma área específica. Este trabalho propõe, ainda, a implementação utilizando *soft-core* PicoBlaze para redução da quantidade de componentes lógicos necessários, que reduz o uso da área do dispositivo lógico programável, FPGA.

Palavras chave: Codificação Reed-Solomon de canal; FPGA; soft-core; rede de sensores sem fio; aplicações médicas; bit error rate; redução de potência; PicoBlaze

Abstract

The main goal of this work was evaluate and design of a Reed-Solomon Architecture (encoder and decoder) because the Reed-Solomon coding is one the most used in digital systems. This architecture will be implemented using soft-core processor and developed Field Programmable Gate Array – FPGA. This architecture aims the reduction of power consumption and the reliability of the error correction codes that provide a robust communication link in noisy environments. The codes shown in this work improve the Bit Error Rate – BER performance that will be used in applications with Wireless Sensor Network – WSN to remote monitoring and medical applications. Therefore, the results show that it is possible to have BER gains in the system using software oriented implementations increasing the communication range or reducing the number of nodes to cover a specific area. This work proposes an implementation using the processor soft-core PicoBlaze to reduce the number of logic components and the used area of the FPGA.

Keywords: Reed-Solomon channel coding; FPGA; soft-core; wireless sensor network; medical application; bit error rate; power reduction; PicoBlaze

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BCH	Bose-Chaudhuri-Hocquenqhen
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
DoD	Department of Defense
DSP	Digital Signal Processor
EEG	Eletroencefalograma
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
HDL	Hardware DescriptionLanguage
IDE	Ambiente de Desenvolvimento Gráfico
IEEE	Institute of Electrical and Electronics Engineers
ISS	Simulador Gráfico do Conjunto de Instruções
LFSR	Linear Feedback Shift Register
LUT	Look-Up Tables
MFSK	MultipleFrequency Shift Keying
MIPS	Milhões de Instruções por Minuto
RAM	Random Access Memory
RF	Rádio Frequência
ROM	Read Only Memory
RS	Reed-Solomon
SNR	Signal-to-Noise Ratio
SoC	System-on-a-Chip
ULA	Unidade Lógica-Aritmética
VHDL	VHSIC Hardware DescriptionLanguage
VHSIC	Very High Speed Integrated Circuits
WSN	Wireless Sensor Network
XOR	OU - Exclusivo

Capítulo 1

Introdução

O uso de códigos corretores de erro é uma das mais poderosas técnicas disponíveis para melhorar a qualidade da comunicação digital e de sistemas de armazenamento, pois possibilita a correção de erros que podem na informação durante a transmissão ou armazenamento. A codificação é feita através da adição de símbolos de paridade à informação que é transmitida através de um canal de comunicação. Tal paridade permite ao decodificador detectar e/ou corrigir efeitos de ruído e interferência encontrados na transmissão da informação através do canal de comunicação. Existem vários códigos corretores de erros, tais como: Código de Hamming, Código Golay, Código Reed-Solomon, entre outros. A codificação de canal foi introduzida em 1948 por Claude E. Shannon.

O código Reed-Solomon é um dos mais importantes e conhecidos códigos não binários Bose-Chaudhuri-Hocquenqhen – BCH, classe parametrizada de códigos corretores de erros. O acrônimo BCH comprime as iniciais dos inventores da classe. Possui grande capacidade de correção de erros em rajada, que o torna um dos mais utilizados em ambientes onde é necessária uma comunicação robusta.

O crescente interesse e desenvolvimento na área da rede de sensores sem fio têm aberto caminho para aplicações dos mesmos em sistemas de monitoramento remoto, tais como: médico, ambiente (clima, poluição, etc.), segurança ou industrial [1]. As principais preocupações desses sistemas são confiabilidade, área e eficiência, porém, a mais importante é o consumo de potência. A eficiência energética e o gerenciamento de potência efetiva são cruciais na operação de rede de sensores e autonomia dos nós sensores.

O consumo de potência dos nós sensores sem fio é distribuído em diferentes partes do dispositivo, como micro controlador, unidade Digital Signal Processor – DSP, transmissor, etc, sendo o módulo Rádio Frequência – RF o maior consumidor de potência. Para economizar o consumo de potência do sistema, os dados são armazenados e o transmissor é configurado para operar à maior taxa de dados possível no modo rajada, minimizando o tempo em que a comunicação ocorre.

A codificação do canal e da fonte é utilizada para aumentar a eficiência e a confiabilidade da comunicação. A codificação de fonte remove as redundâncias inerentes à informação através da quantização e/ou compressão, reduzindo, assim, a quantidade de energia necessária para transmissão através do canal. A codificação do canal, por outro lado, introduz símbolos de paridade sistemáticos à informação. Então, transmitindo a informação a uma reduzida energia por bit, a comunicação atinge a mesma BER – *Bit Error Rate* (probabilidade de erro por bit), ou seja, aumenta-se a distância. A Figura 1.1 apresenta o fluxograma da codificação e transmissão *wireless* de uma informação, no caso, um *Eletro encefalograma – EEG* [2]. Faz-se, inicialmente, uma codificação de fonte da informação inicial, *d(x)*, obtendo-se a informação *m(x)*. Esta mensagem é codificada utilizando a codificação Reed-Solomon obtendo-se a palavra código, *c(x)*. Essa mensagem codificada é modulada e transmitida através de um canal onde podem ocorrer erros. A mensagem recebida, *r(x)*, é de modulada e, então, decodificada obtendo-se a informação inicial *d(x)*.

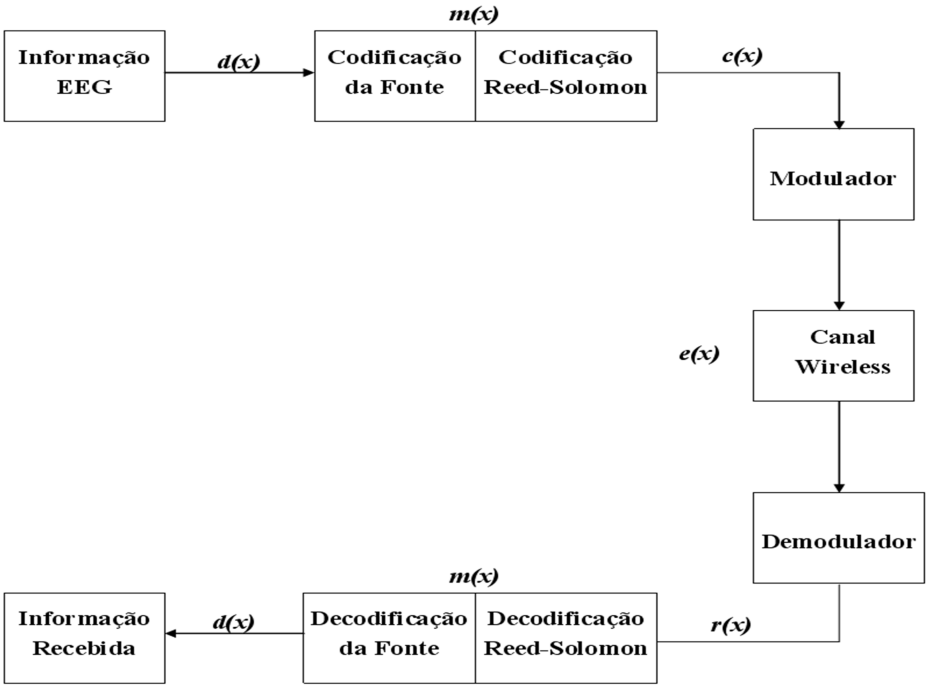

Figura 1.1 – Fluxo da Comunicação

1.1 – Motivações e Objetivos

A utilização de códigos corretores de erros em redes de sensores sem fio representa economia da potência. Com o *Foward Error Correction* – FEC (correção antecipada de erros), as distâncias entre transmissor e receptor são aumentadas proporcionalmente com a capacidade de correção do código.

Um processador soft-core é um micro processador ou micro controlador totalmente descrito em software, usualmente em Hardware Description Language – HDL (linguagem de descrição de hardware), que pode ser sintetizado em *hardware* programável, por exemplo, FPGA. Um processador *soft-core* direcionado para FPGA é flexível, pois seus parâmetros podem ser alterados a qualquer momento, apenas reprogramando o dispositivo.

A utilização de uma implementação em *software* representa uma significativa redução de área, maior desempenho e menor consumo de energia. Para a implementação desse trabalho, foi escolhido como processador *soft-core* o micro controlador PicoBlaze™ da Xilinx, pois requer pequena quantidade de lógica digital e possui código livre.

Um dispositivo FPGA é um hardware com alto poder de processamento e flexibilidade, que atende a necessidade para implementação do FEC. O custo do dispositivo FPGA é diretamente proporcional ao número de elementos lógicos disponíveis e, como visto acima, o PicoBlaze requer uma pequena quantidade permitindo assim a escolha de um dispositivo de baixo custo. A implementação de um decodificador Reed-Solomon requer um alto grau de complexidade, o que exige uma grande quantidade de elementos lógicos. Logo, a implementação de uma decodificação em área reduzida também reduz os custos [3].

Este trabalho foi desenvolvido considerando as diversas aplicações que podem beneficiar-se de uma implementação de correção de erros baseada em software, porém, implementada em hardware, tais como aplicações médicas. E também contribuir com o Grupo de Microeletrônica da Universidade Federal de Itajubá para a formação de massa crítica de profissionais especializados na área de confiabilidade e segurança na transmissão de dados.

Portanto, o objetivo dessa dissertação é avaliar as capacidades de correção de erro que podem ser implementadas utilizando o micro controlador PicoBlaze da Xilinx.

Verificar a economia do consumo de potência devido à redução da quantidade de componentes lógicos e o aumento da distância entre transmissor e receptor para um mesmo BER. O sistema foi projetado para atuar em aplicações médicas e monitoramento remoto. Os circuitos do codificador e decodificador são modelados, simulados, implementados e validados utilizando a FPGA Spartan-3E do fabricante Xilinx.

1.2 – Estrutura do Trabalho

No Capítulo 2 é feito uma explanação sobre códigos corretores de erro, incluindo as características do código Reed-Solomon, como a probabilidade de erro, o ganho da codificação e a capacidade de correção do código. Apresenta-se também a matemática utilizada no código Reed-Solomon, incluindo a teoria e os algoritmos utilizados para efetuar a codificação e a decodificação da informação. Para facilitar essa apresentação, é utilizado, como exemplo, o código RS(7,3).

Já o Capítulo 3 apresenta uma explanação sobre FPGA e linguagem de descrição de *hardware*, em especial VHDL, que é utilizada neste trabalho. São apresentadas também as principais características do PicoBlaze da Xilinx.

O Capítulo 4 descreve as etapas da implementação deste trabalho. Apresentam-se, ainda, as simulações obtidas utilizando a plataforma ISE 10.1 da Xilinx para verificar o correto funcionamento tanto da codificação quanto da decodificação. Há um estudo sobre o tempo de processamento e o ganho obtido utilizando a codificação Reed-Solomon, onde esse ganho é avaliado quanto ao aumento de distância entre transmissor e receptor em relação à distância obtida inicialmente (sem o uso de código corretor de erro) mantendo-se o mesmo valor de BER e potência do transmissor.

Capítulo 2

Códigos Corretores de Erros

2.1 – Introdução

A transmissão confiável de informações em sistemas de comunicação, em taxas cada vez maiores, tem representado um desafio constante para engenheiros e pesquisadores em telecomunicações. Os códigos corretores de erros [4], sem dúvida, têm contribuído de modo significativo para os avanços nesta área. Mais recentemente, problemas ligados ao armazenamento e recuperação de grandes volumes de dados, em memórias semicondutoras [5], também passaram a ser beneficiados pelas técnicas de códigos corretores de erros.

A confiabilidade da transmissão referida diz respeito apenas à sua imunidade à ação do ruído e de outras interferências, não considerando o problema da interceptação das mensagens por terceiros [6]. Uma forma de corrigir a informação seria retransmitir a mensagem, porém, isso nem sempre é possível ou conveniente, pois a retransmissão diminui a taxa efetiva de transmissão. Os códigos corretores de erros permitem que uma quantidade limitada de erros seja corrigida sem a necessidade de retransmissão [3], onde essa quantidade é diretamente proporcional à capacidade de correção do código.

O usuário do sistema de transmissão digital geralmente estabelece uma taxa de erro máxima aceitável, acima da qual os dados recebidos não são considerados utilizáveis pelo usuário. Essa taxa de erro aceitável depende da mensagem que transita pelo canal [7].

O teorema da codificação de C. E. Shannon para um canal ruidoso, “A Mathematical Theory of Communication”, foi estabelecida em 1948. A partir deste artigo, houve um desenvolvimento continuo e significativo da teoria dos códigos corretores de erros até hoje.

Em 1960, Irving S. Reed e Gustave Solomon desenvolveram o código Reed-Solomon [8], que é um dos mais utilizados atualmente. Com a popularização dos computadores, a teoria da correção de erros chamou a atenção dos engenheiros. Richard

W. Hamming introduziu, em 1974, o chamado Código Hamming e no mesmo ano, Marcel J. E. Golay apresentou o Código Golay [3].

Um código corretor de erros é, basicamente, uma forma organizada de acrescentar algum dado a cada informação que precise ser transmitida ou armazenada, de modo que permita, ao recuperar a informação, detectar e corrigir erros no processo de transmissão da informação [9].

Os erros que podem ocorrer durante a transmissão podem ser erros esporádicos e independentes, sendo chamados de **erros aleatórios**, ou podem ocorrer em surtos de vários erros, chamados de **erros em surto ou em rajada** [6].

A codificação pode ser sistemática, isto é, a mensagem codificada consiste nos *k* símbolos da mensagem original seguidos dos símbolos de redundância [3], conforme a Figura 2.1 ou não sistemática, isto é, a mensagem codificada de *n* símbolos não apresenta explicitamente os *k* símbolos da mensagem original [3,7].

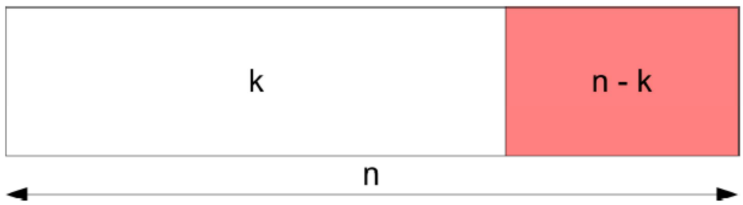

Figura 2.1 – Codificação Sistemática

Os códigos podem ser, ainda, lineares ou não-lineares. O primeiro tem seus dígitos redundantes calculados como combinações lineares dos dígitos de informação, enquanto o segundo faz uso de operações ou circuitos não-lineares, como portas E, OU, NÃO-OU, etc., no caso binário, para combinar os dígitos de informação [6].

A maneira como a redundância é anexada à informação determina se o código é de bloco ou convolucional. Os códigos que verificam a ocorrência ou não de erros bloco a bloco, independentemente um do outro, são chamados de *códigos de bloco* (sem memória) [10]. Os principais códigos de bloco são: BCH, Hamming, Golay e Reed-Solomon, que é implementado neste trabalho. Os códigos em que a operação depende não somente do bloco que está sendo processado, mas também de um bloco anterior são chamados de *códigos convolucionais* [3]. Ambos são competitivos em muitas situações práticas, onde a escolha final depende de fatores como o formato dos dados, o retardo na decodificação e a complexidade do sistema necessário para alcançar uma determinada taxa de erros [6].

2.2 – Algoritmo de Reed-Solomon

Os códigos Reed-Solomon são amplamente utilizados no armazenamento e transmissão de dados, devido sua baixa complexidade e alto poder de correção.

Estes códigos são cíclicos não binários com símbolos compostos por sequências de *m* bits, onde *m* é qualquer inteiro positivo maior que 2 [11]. Os códigos Reed-Solomon são representados na forma RS (*n*, *k*), onde *n* é o número de símbolos da mensagem codificada e *k* é o número de símbolos da mensagem original. Usualmente,

$$(n, k) = (2^m - 1, 2^m - 1 - 2t) \tag{2.1}$$

onde *t* é a capacidade de correção do código (metade dos símbolos de paridade) e $n - k = 2t$ é o número de símbolos de paridade.

2.2.1 – Características

A capacidade de detecção e correção de erros de um código está diretamente ligada à sua distância mínima [6]. Os códigos Reed-Solomon – RS alcançaram a maior distância mínima possível para um código linear dada por [12]

$$d_{min} = n - k + 1 \tag{2.2}$$

Logo, o código RS pode corrigir qualquer combinação de t ou menos erros. Porém, é necessário um símbolo redundante para localizar o erro e outro símbolo redundante para encontrar o valor correto. Portanto, a quantidade de símbolos redundantes é $2t$.

Os códigos RS possuem ótimo desempenho contra ruídos em rajada, pois cada símbolo é formado por *m* bits, isto é, mesmo que apenas um bit seja afetado, caso seja possível, todo o símbolo será corrigido, ou seja, os *m* bits.

2.2.2 – Probabilidade de Erro dos Códigos RS

A probabilidade de erro-símbolo decodificado do código RS, P_E, em função da probabilidade erro-símbolo do canal, *p*, pode ser escrito conforme a fórmula abaixo [11] usando o princípio da distribuição de probabilidade de Bernoulli:

$$P_E \approx \frac{1}{2^m-1}\sum_{j=t+1}^{2^m-1} j \binom{2^m-1}{j} p^j (1-p)^{2^m-1-j} \tag{2.3}$$

onde *t* é a capacidade de correção do código e os símbolos são formados por *m* bits cada.

A probabilidade de erro-bit, P_B ou *BER*, pode ser limitada superiormente pela probabilidade de erro-símbolo, P_E, para cada tipo de modulação. Por exemplo, para a modulação Multiple Frequency Shift Keying – MFSK com $M = 2^m$, a relação entre P_B e P_E é apresentada abaixo [11]:

$$\frac{P_B}{P_E} = \frac{2^{(m-1)}}{(2^m)-1} \tag{2.4}$$

A Figura 2.2 apresenta P_B versus a probabilidade de erro-símbolo do canal *p*, plotado a partir da Equação (2.3) e da Equação (2.4) para vários valores de *t*.

Já a Figura 2.3 apresenta P_B versus E_b/N_0 (relação energia de bit por ruído) para tal sistema usando modulação MFSK sobre um canal Additive White Gaussian Noise – AWGN. Para códigos RS, a probabilidade de erro é exponencialmente decrescente em função do comprimento do bloco, *n*, e a complexidade de decodificação é proporcional a uma pequena potência do comprimento do bloco [11].

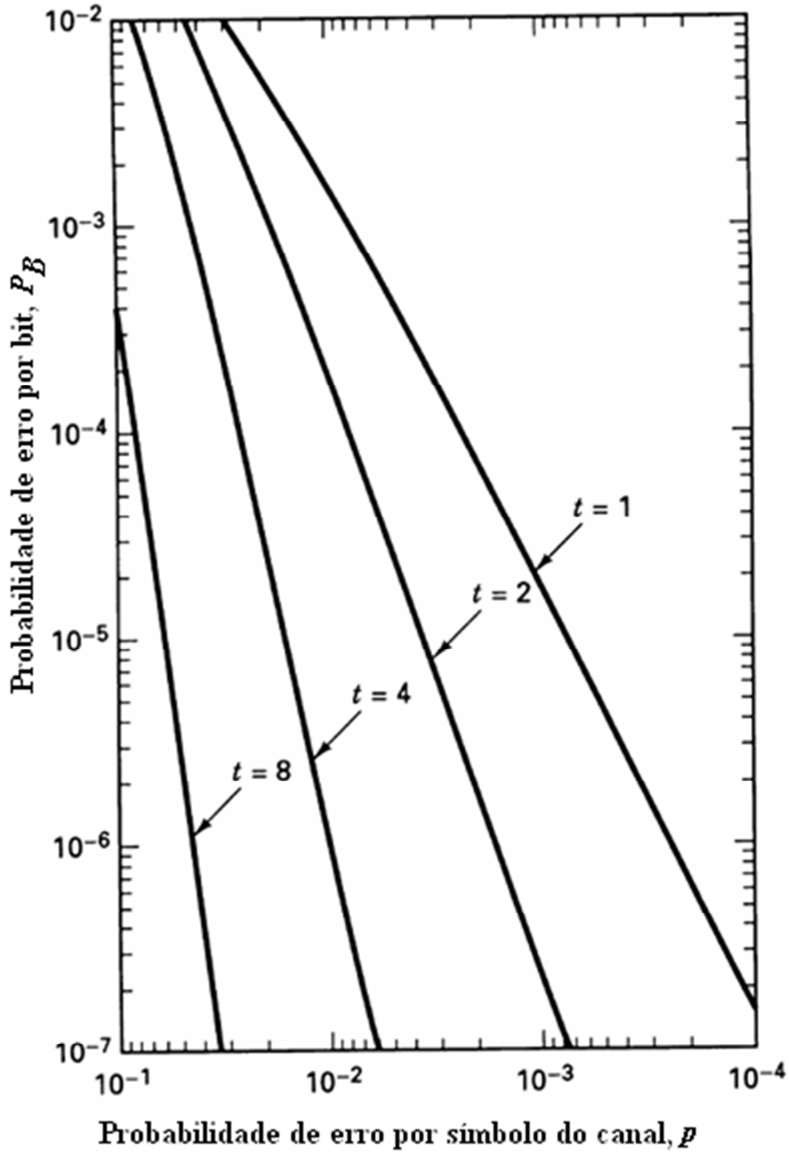

Figura 2.2 – P_B versus p para n = 31

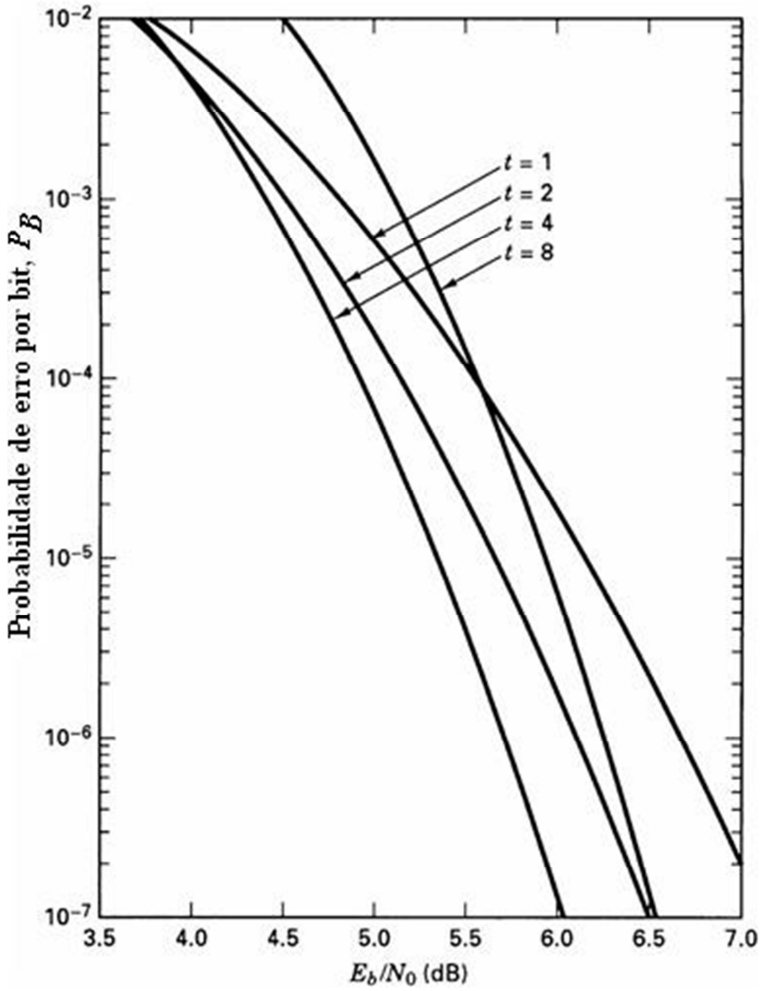

Figura 2.3 – BER versus E_b/N_0 para n = 31 e Modulação MFSK sobre canal AWGN

2.2.3 – Ganho da Codificação RS

O ganho da codificação RS geralmente é calculado definindo o BER (taxa de erro bit) desejado para um sinal não codificado e codificado na modulação Binary Phase Shift Keying – BPSK e, então, medindo a diferença entre o Signal-to-Noise Ratio – SNR (relação sinal e ruído).

Para verificar o ganho da codificação RS, o seguinte modelo é utilizado [2]:

$$P_{TX,U}[W] = \eta_U \frac{E_b}{N_0} N \left(\frac{4\pi}{\lambda}\right)^2 d^n \qquad (2.5)$$

onde $P_{TX,U}$ é a potência do sinal não codificado transmitido, η_U é a eficiência espectral (igual a 1 para BPSK), E_b/N_0 é a relação energia de bit por ruído, N é o ruído do sinal (produto do ruído termal pela largura da banda), λ é o comprimento de onda transmitida,

d é a distância entre o transmissor e o receptor e *n* é o coeficiente de perda (2 para espaço livre, 3 para ambiente fechado e 4 para ambiente fechado com muitos obstáculos) [2].

Então, colocando o caso utilizando a codificação e o caso sem a codificação em proporção, a seguinte relação é obtida.

$$\frac{P_{TX,RS}[W]}{P_{TX,U}[W]} = \frac{\eta_{max}\left(\frac{E_b}{N_0}\right)_{RS} N\left(\frac{4\pi}{\lambda}\right)^2 d^n}{\eta_{max}\left(\frac{E_b}{N_0}\right)_{U} N\left(\frac{4\pi}{\lambda}\right)^2 d^n} \tag{2.6}$$

onde o índice *RS* faz referência ao caso utilizando a codificação Reed-Solomon e o índice *U* faz referência ao caso sem a utilização da codificação Reed-Solomon.

Considerando mesmo canal e mesmo transmissor, a Equação (2.6) pode ser simplificada para:

$$\frac{\left(\frac{E_b}{N_0}\right)_U}{\left(\frac{E_b}{N_0}\right)_{RS}} = \left(\frac{d_{RS}}{d_U}\right)^n \tag{2.7}$$

onde d_{RS} é a distância alcançada utilizando a codificação Reed-Solomon e d_U é a distância alcançada sem a utilização da codificação Reed-Solmon.

2.3 – Campos Finitos

Esta seção tem como objetivo introduzir o conceito de campos finitos ou campos de Galois, homenagem ao francês Évariste Galois, os quais possuem grande importância nos códigos corretores de erros.

2.3.1 – Grupos

Grupo é uma estrutura algébrica definida como um conjunto de elementos que estão relacionados por operações específicas. Um grupo é um conjunto *G* com uma operação binária * possuindo as seguintes propriedades [13]:

A operação * é fechada: se *a*, *b* ϵ *G*, então

$$a * b \epsilon G \tag{2.8}$$

A operação * é associativa: se a, b, $c \epsilon G$, então

$$(a * b) * c = a * (b * c) \tag{2.9}$$

Existe elemento *identidade* $e \epsilon G$: para todo a ϵG, tem-se

$$a * e = e * a = e \tag{2.10}$$

Todo elemento tem *inverso*: se $a \epsilon G$, então existe $a' \epsilon G$ tal que

$$a * a' = a' * a = e \tag{2.11}$$

2.3.2 – Campos

Um campo finito é definido como um conjunto de elementos F para o qual adição, multiplicação, subtração e divisão (exceto divisão por zero) realizadas entre seus elementos resultam em outro elemento do mesmo conjunto [14].

2.3.3 – Campos de Galois

Para um número primo qualquer, q, existe um campo finito representado por GF(q) com q elementos. É possível estender o campo GF(q) para um campo com q^m elementos, chamado de campo estendido de GF(q) e representado por GF(q^m), onde m é um inteiro positivo diferente de zero. Percebe-se que o GF(q^m) tem como subconjunto de elementos o GF(q). Os símbolos do campo estendido GF(q^m) são utilizados na construção dos códigos RS [11].

O campo binário GF(2) é um sub-campo do campo estendido GF(2^m), assim como o campo dos números reais é um sub-campo do campo dos números complexos. Além dos números 0 e 1, existem elementos únicos no campo estendido que serão representados pelo novo símbolo α, onde cada elemento diferente de zero pode ser representado por uma potência de α. Um conjunto *infinito* de elementos, F, é formado iniciando-se com os elementos {0, 1, α}, e, então, gerando elementos multiplicando-se o último elemento por α, que produz:

$$F = \{0, 1, \alpha, \alpha^2, ..., \alpha^j, ...\} = \{0, \alpha^0, \alpha^1, \alpha^2, ..., \alpha^j, ...\} \tag{2.12}$$

Uma condição deve ser imposta sobre *F* para obter-se um conjunto de elementos *finitos* de *F*, isto é, só pode conter 2^m elementos e fechado nas operações de soma e multiplicação. Tal condição é caracterizada pelo polinômio irredutível:

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

$$\alpha^{(2^m-1)} = 1 = \alpha^0 \tag{2.13}$$

Usando essa restrição, qualquer elemento do campo que tenha uma potência igual ou maior que $2^m - 1$ pode ser reduzido a um elemento com uma potência menor, conforme Equação abaixo.

$$\alpha^{(2^m+n)} = \alpha^{(2^m-1)}\alpha^{(n+1)} = \alpha^{(n+1)} \tag{2.14}$$

Portanto, percebe-se que os elementos do campo finito GF(2^m) são:

$$\text{GF}(2^m) = \{ 0, \alpha^0, \alpha^1, \alpha^2, ..., \alpha^{(2^m-2)} \} \tag{2.15}$$

2.3.3.1 – Adição no Campo Estendido GF(2^m)

Cada um dos 2^m elementos do campo finito, GF(2^m), pode ser representado por um polinômio distinto de grau $m - 1$ ou menor. Cada elemento diferente de zero de GF(2^m) é representado por um polinômio, $\alpha_i(X)$, onde pelo menos um dos *m* coeficientes é diferente de zero. Para $i = 0,1,2,...,2^m - 2$,

$$\alpha^i = \alpha_i(X) = \alpha_{i,0} + \alpha_{i,1}X + \alpha_{i,2}X^2 + ... + \alpha_{i,m-1}X^{m-1} \tag{2.16}$$

Considerando o caso onde $m = 3$, o campo finito é representado por GF(2^3). A Figura 2.4 mostra o mapeamento dos sete elementos $\{\alpha^i\}$ e o elemento nulo, em termos de elementos de base $\{X^0, X^1, X^2\}$. Cada linha do mapeamento compreende uma sequência de valores binários representando os coeficientes $\alpha_{i,0}$, $\alpha_{i,1}$ e $\alpha_{i,2}$ da Equação (2.16). Adição de dois elementos do campo finito é, então, definida como a adição módulo-2 de cada coeficiente do polinômio,

$$\alpha^i + \alpha^j = \alpha_i(X)=(\alpha_{i,0} + \alpha_{j,0})+(\alpha_{i,1} + \alpha_{j,1})X+(\alpha_{i,2} + \alpha_{j,2})X^2+...+(\alpha_{i,m-1} + \alpha_{j,m-1})X^{m-1} \tag{2.17}$$

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

Figura 2.4 – Mapeamento dos Elementos de um campo GF(2^3) com f(x) = 1 + x + x^3.

2.3.4 – Polinômio Primitivo

Os *polinômios primitivos* são de interesse, pois definem os campos finitos GF(2^m) que são utilizados nos códigos RS. A seguinte condição é necessária e suficiente para garantir que o polinômio seja primitivo: um polinômio irredutível *f(X)* de grau *m* é dito primitivo se o menor inteiro positivo *n* para o qual *f(X)* divide $X^n + 1$ é $n = 2^m - 1$, sendo o quociente da divisão diferente de zero e o resto igual à zero. A Equação (2.18) representa essa condição:

$$resto = \left(\frac{X^n+1}{f(X)}\right) = 0 \tag{2.18}$$

A Tabela 2.1 contém uma lista de alguns polinômios primitivos utilizados na construção de campos finitos, de acordo com o valor de *m*[15].

Tabela 2.1 – Polinômios Primitivos

m	
3	$1 + X + X^3$
4	$1 + X + X^4$
5	$1 + X^2 + X^5$
6	$1 + X + X^6$
7	$1 + X^3 + X^7$
8	$1 + X^2 + X^3 + X^4 + X^8$
9	$1 + X^4 + X^9$
10	$1 + X^3 + X^{10}$
11	$1 + X^2 + X^{11}$
12	$1 + X + X^4 + X^6 + X^{12}$
13	$1 + X + X^3 + X^4 + X^{13}$
14	$1 + X + X^6 + X^{10} + X^{14}$
15	$1 + X + X^{15}$
16	$1 + X + X^3 + X^{12} + X^{16}$
17	$1 + X^3 + X^{17}$
18	$1 + X^7 + X^{18}$
19	$1 + X + X^2 + X^5 + X^{19}$
20	$1 + X^3 + X^{20}$
21	$1 + X^2 + X^{21}$
22	$1 + X + X^{22}$
23	$1 + X^5 + X^{23}$
24	$1 + X + X^2 + X^7 + X^{24}$

2.3.5 – O Campo Estendido GF(2^3)

Para o exemplo considerado, m = 3, tem-se oito elementos no campo definido por $f(X) = 1 + X + X^3$. É necessário, então, encontrar as raízes de $f(X)$, isto é, $f(X) = 0$. É importante lembrar-se que um polinômio de grau m tem precisamente m raízes. Então, as três raízes existem no campo estendido GF(2^3). Sendo α, um elemento do campo estendido, definida como raiz do polinômio $f(X)$. Então,

$$f(\alpha) = 0$$

$$1 + \alpha + \alpha^3 = 0 \tag{2.19}$$

$$\alpha^3 = -1 - \alpha$$

Sendo a operação soma igual à operação subtração, pode-se substituir os termos negativos por termos positivos [11]. Logo,

$$\alpha^3 = 1 + \alpha \tag{2.20}$$

Então, α^3 é definido como uma soma de elementos α de menor ordem, onde, a partir de α^3, é possível obter os demais elementos do campo através de multiplicações por α, assim representadas nas equações a seguir:

$$\alpha^4 = \alpha.\alpha^3 = \alpha.(1 + \alpha) = \alpha + \alpha^2$$

$$\alpha^5 = \alpha.\alpha^4 = \alpha.(\alpha + \alpha^2) = \alpha^2 + \alpha^3 = \alpha^2 + \alpha + 1 \quad (2.21)$$

$$\alpha^6 = \alpha.\alpha^5 = \alpha.(\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha = (1 + \alpha) + \alpha^2 + \alpha$$

$$\alpha^6 = 1 + \alpha^2$$

Para os elementos α de ordem menor que 3, têm-se algumas definições:

$$\alpha^0 = 1$$

$$\alpha^1 = X \quad (2.22)$$

$$\alpha^2 = X^2$$

É possível, então, construir o campo estendido GF(2^3) a partir das equações (2.21) e (2.22), conforme apresentado na Tabela 2.2 [15].

Tabela 2.2 – Campo Estendido GF(2^3)

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

O mapeamento dos elementos do campo pode ser demonstrado através de um registrador de deslocamento com realimentação linear ou Linear Feedback Shift

Register – LFSR, como apresentado na Figura 2.5 [15]. O circuito gera (com $m = 3$) os $2^m - 1$ elementos diferente de zero do campo apresentado na Figura 2.4 definidos nas equações (2.16) e (2.17). Observa-se no circuito da Figura 2.5 que as conexões de realimentação correspondem aos coeficientes do polinômio $f(X) = 1 + X + X^3$. Iniciando com o estado não nulo 001 (α^0), todas as demais potências de α aparecem ciclicamente na ordem apresentada na Tabela 2.2.

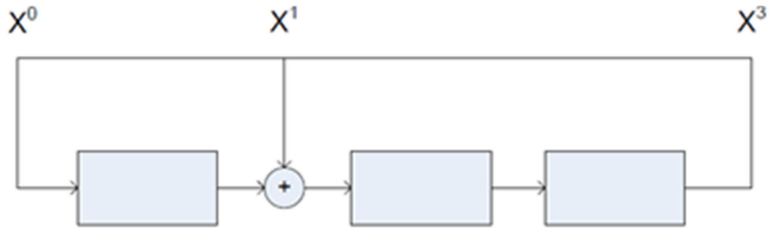

Figura 2.5 – Circuito LFSR

Duas operações aritméticas; adição e multiplicação, podem ser definidas para o registrador de deslocamento. A adição é apresentada na Tabela 2.3, obtida através da operação OU - Exclusivo – XOR entre os elementos na forma binária (adição módulo 2). A multiplicação é apresentada na Tabela 2.4, obtida através da soma dos expoentes de α módulo $(2^m - 1)$, neste caso módulo 7.

Tabela 2.3 – Adição sobre o campo GF(2^3)

	$\boldsymbol{\alpha^0}$	$\boldsymbol{\alpha^1}$	$\boldsymbol{\alpha^2}$	$\boldsymbol{\alpha^3}$	$\boldsymbol{\alpha^4}$	$\boldsymbol{\alpha^5}$	$\boldsymbol{\alpha^6}$
$\boldsymbol{\alpha^0}$	0	α^3	α^6	α^1	α^5	α^4	α^2
$\boldsymbol{\alpha^1}$	α^3	0	α^4	α^0	α^2	α^6	α^5
$\boldsymbol{\alpha^2}$	α^6	α^4	0	α^5	α^1	α^3	α^0
$\boldsymbol{\alpha^3}$	α^1	α^0	α^5	0	α^6	α^2	α^4
$\boldsymbol{\alpha^4}$	α^5	α^2	α^1	α^6	0	α^0	α^3
$\boldsymbol{\alpha^5}$	α^4	α^6	α^3	α^2	α^0	0	α^1
$\boldsymbol{\alpha^6}$	α^2	α^5	α^0	α^4	α^3	α^1	0

Tabela 2.4 – Multiplicação sobre o campo GF(2^3)

	$\boldsymbol{\alpha^0}$	$\boldsymbol{\alpha^1}$	$\boldsymbol{\alpha^2}$	$\boldsymbol{\alpha^3}$	$\boldsymbol{\alpha^4}$	$\boldsymbol{\alpha^5}$	$\boldsymbol{\alpha^6}$
$\boldsymbol{\alpha^0}$	α^0	α^1	α^2	α^3	α^4	α^5	α^6
$\boldsymbol{\alpha^1}$	α^1	α^2	α^3	α^4	α^5	α^6	α^0
$\boldsymbol{\alpha^2}$	α^2	α^3	α^4	α^5	α^6	α^0	α^1
$\boldsymbol{\alpha^3}$	α^3	α^4	α^5	α^6	α^0	α^1	α^2
$\boldsymbol{\alpha^4}$	α^4	α^5	α^6	α^0	α^1	α^2	α^3
$\boldsymbol{\alpha^5}$	α^5	α^6	α^0	α^1	α^2	α^3	α^4
$\boldsymbol{\alpha^6}$	α^6	α^0	α^1	α^2	α^3	α^4	α^5

2.4 – Codificação Reed-Solomon

O processo de codificação RS consiste na geração de símbolos redundantes ou símbolos de paridade a partir de uma informação inicial. Conforme citado anteriormente, a quantidade de símbolos de paridade é igual a 2*t*, onde *t* é a quantidade de símbolos que o código pode corrigir. Então, a codificação RS utiliza o polinômio gerador representado por:

$$g(X) = g_0 + g_1X + g_1X^2 + \ldots + g_{2t-1}X^{2t-1} + X^{2t} \tag{2.23}$$

O grau do polinômio gerador é igual ao número de símbolos de paridade (2*t*), ou seja, devem existir exatamente 2*t* potências de α que sejam raízes deste polinômio. As raízes de *g(X)* são representadas como α, α^2, ..., α^{2t}, não necessariamente iniciando com a raiz α, mas podendo iniciar com qualquer potência de α. Considerando o exemplo RS(7,3), que possui quatro símbolos de paridade, o seu polinômio gerador é:

$$\begin{aligned} g(X) &= (X-\alpha)(X-\alpha^2)(X-\alpha^3)(X-\alpha^4) \\ g(X) &= (X^2-(\alpha+\alpha^2)X+\alpha^3)(X^2-(\alpha^3+\alpha^4)X+\alpha^7) \\ g(X) &= (X^2-\alpha^4X+\alpha^3)(X^2-\alpha^6X+\alpha^0) \\ g(X) &= X^4-(\alpha^4+\alpha^6)X^3+(\alpha^3+\alpha^{10}+\alpha^0)X^2-(\alpha^4+\alpha^9)X+\alpha^3 \\ g(X) &= X^4-\alpha^3X^3+\alpha^0X^2-\alpha^1X+\alpha^3 \end{aligned} \tag{2.24}$$

Colocando na ordem do grau menor para o grau maior e trocando os sinais negativos por positivos, visto que no campo binário + 1 = – 1, *g(X)* pode ser expresso assim:

$$g(X) = \alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4 \quad (2.25)$$

2.4.1 – Codificação de Forma Sistemática

Uma vez que códigos RS são códigos cíclicos, a codificação de forma sistemática é análoga ao procedimento de codificação binária [11]. Desloca-se o polinômio da mensagem, **m**(X), para a direita da palavra código e adiciona-se à esquerda o polinômio paridade, **p**(X). Portanto, multiplica-se **m**(X) por X^{n-k} manipulando, assim, o polinômio da mensagem algebricamente para deslocá-lo. Então, divide-se X^{n-k}**m**(X) pelo polinômio gerador **g**(X), descrito na Equação (2.26):

$$X^{n-k}\boldsymbol{m}(X) = \boldsymbol{q}(X)\boldsymbol{g}(X) + \boldsymbol{p}(X) \quad (2.26)$$

onde **q**(X) e **p**(X) são os polinômios cociente e resto, respectivamente. Como no caso binário, o resto é a paridade. A Equação (2.26) também pode ser expressa na seguinte forma:

$$\boldsymbol{p}(X) = X^{n-k}\boldsymbol{m}(X)\ modulo\ \boldsymbol{g}(X) \quad (2.27)$$

O polinômio da palavra código resultante, **U**(X), pode ser escrito na seguinte forma:

$$\boldsymbol{U}(X) = \boldsymbol{p}(X) + X^{n-k}\boldsymbol{m}(X) \quad (2.28)$$

Para demonstrar as Equações (2.27) e (2.28), toma-se, como exemplo, a seguinte mensagem com três símbolos:

$$\alpha^1\alpha^3\alpha^5 = 010\ 011\ 111$$

Para codificar a mensagem com o código RS(7,3), cujo polinômio gerador é dado na Equação (2.25), multiplica-se, primeiramente, o polinômio da mensagem m(X) = $\alpha^1 + \alpha^3 X + \alpha^5 X^2$ por $X^{n-k} = X^4$, obtendo-se a mensagem deslocada para a direita $\alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$. Então, divide-se a mensagem deslocada pelo polinômio gerador apresentado na Equação (2.25). As operações necessárias nessa divisão polinomial

(adição e multiplicação) devem seguir as regras das Tabelas 2.3 e 2.4. O polinômio da paridade resultante é:

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

Então, da Equação (2.28), tem-se que o polinômio da palavra código é:

$$\boldsymbol{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$$

Assim, a partir da mensagem original, 010 011 111 (2 3 7), obteve-se a palavra código 001 100 110 101 010 011 111 (1 4 6 5 2 3 7), onde é composta por quatro símbolos de paridade obtidos na codificação e os três símbolos da mensagem original.

Na prática, a codificação RS é realizada através do registrador de deslocamento – LFSR.

2.4.2 – Codificação Sistemática com Registrador de Deslocamento

Para codificar de forma sistemática a sequência de três símbolos é feita a implementação do circuito do registrador de deslocamento, conforme mostrado na Figura 2.6.

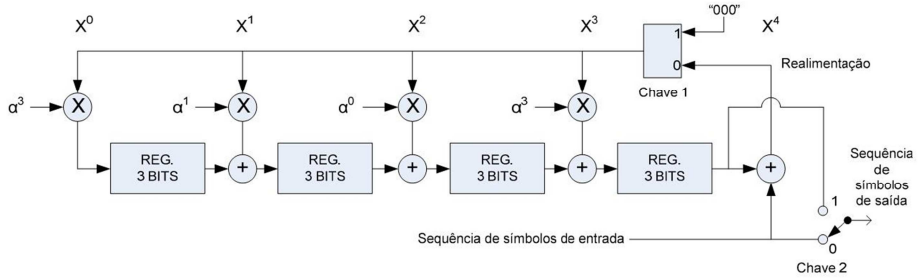

Figura 2.6–Codificador RS(7,3)

Percebe-se que os termos que multiplicam a entrada realimentada do registrador de deslocamento correspondem aos coeficientes do polinômio gerador conforme a Equação (2.25). Cada registrador armazena um valor de 3 bits, sendo qualquer um dos oito valores (7 potências de α e o elemento nulo).

A operação não binária implementada pelo circuito descrito na Figura 2.6 forma as palavras códigos de forma sistemática. Os passos podem assim ser descritos:

1) A chave 1 permanece na posição 0 durante os primeiros *k* ciclos de clock para permitir o deslocamento dos símbolos da mensagem nos (*n* – *k*) estágios do registrador de deslocamento.

2) A chave 2 fica na posição 0 durante os primeiros *k* ciclos de clock para permitir a transferência simultânea dos símbolos da mensagem diretamente para a saída do registrador.

3) Após a transferência dos k símbolos da mensagem para a saída do registrador, a chave 1 e a chave 2 é movida para a posição 1.

4) Os (*n* – *k*) ciclos de clocks restantes limpam os símbolos de paridade contidos no registrador de deslocamento, movendo-os para a saída do registrador.

5) O número total de ciclos de clock é igual a *n*, e o conteúdo da saída do registrador é o polinômio da palavra código $\mathbf{p}(X) + X^{n-k}\mathbf{m}(X)$, onde $\mathbf{p}(X)$ representa os símbolos de paridade e $\mathbf{m}(X)$ representa os símbolos da mensagem na forma polinomial.

Utilizando a mesma sequência de símbolos escolhida anteriormente, $\alpha^1\ \alpha^3\ \alpha^5$, representa-se na Tabela 2.5 o processo de geração dos símbolos de paridade utilizado pelo circuito da Figura 2.6.

Tabela 2.5 – Conteúdo dos Registradores do Circuito da Figura 2.6

Entrada da Fila			Ciclo de Clock	Conteúdo dos Registradores				Realimentação
α^1	α^3	α^5	0	0	0	0	0	α^5
	α^1	α^3	1	α^1	α^6	α^5	α^1	α^0
		α^1	2	α^3	0	α^2	α^2	α^4
		-	3	α^0	α^2	α^4	α^6	-

Após o terceiro ciclo de clock, o conteúdo do registrador são os quatro símbolos de paridade, $\alpha^0\ \alpha^2\ \alpha^4\ \alpha^6$, conforme observado na Tabela 2.5. A chave 1 e a

chave 2 são alternadas para a posição 1, e os símbolos de paridade contidos no registrador são deslocados para a saída. Portanto, a palavra código, **U**(X), escrita na forma polinomial, pode ser expressada seguinte forma:

$$\boldsymbol{U}(X) = \sum_{n=0}^{6} u_n X^n$$

$$\boldsymbol{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \qquad (2.29)$$

$$\boldsymbol{U}(X) = (001) + (100)X + (110)X^2 + (101)X^3 + (010)X^4 + (011)X^5 + (111)X^6$$

As raízes do polinômio gerador, **g**(X), também devem ser raízes da palavra código gerada por **g**(X), pois uma palavra código é válida na seguinte forma:

$$\boldsymbol{U}(X) = \boldsymbol{m}(X)\boldsymbol{g}(X) \qquad (2.30)$$

Portanto, uma palavra código arbitrária, quando avaliada para qualquer raiz de **g**(X) deve resultar em zero. Em outras palavras [11],

$$\boldsymbol{U}(\alpha) = \boldsymbol{U}(\alpha^2) = \boldsymbol{U}(\alpha^3) = \boldsymbol{U}(\alpha^4) = 0$$

Então,

$$\boldsymbol{U}(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^9 + \alpha^5 + \alpha^8 + \alpha^{11}$$

$$\boldsymbol{U}(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^2 + \alpha^5 + \alpha^1 + \alpha^4$$

$$\boldsymbol{U}(\alpha) = \alpha^1 + \alpha^0 + \alpha^6 + \alpha^4$$

$$\boldsymbol{U}(\alpha) = \alpha^3 + \alpha^3 = 0$$

$$\boldsymbol{U}(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^8 + \alpha^{12} + \alpha^9 + \alpha^{13} + \alpha^{17}$$

$$\boldsymbol{U}(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^1 + \alpha^5 + \alpha^2 + \alpha^6 + \alpha^3$$

$$\boldsymbol{U}(\alpha^2) = \alpha^5 + \alpha^6 + \alpha^0 + \alpha^3$$

$$\boldsymbol{U}(\alpha^2) = \alpha^1 + \alpha^1 = 0$$

$$\boldsymbol{U}(\alpha^3) = \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^{15} + \alpha^{13} + \alpha^{18} + \alpha^{23}$$

$$\boldsymbol{U}(\alpha^3) = \alpha^0 + \alpha^5 + \alpha^3 + \alpha^1 + \alpha^6 + \alpha^4 + \alpha^2$$

$$\boldsymbol{U}(\alpha^3) = \alpha^4 + \alpha^0 + \alpha^3 + \alpha^2$$

$$\boldsymbol{U}(\alpha^3) = \alpha^5 + \alpha^5 = 0$$

$$\boldsymbol{U}(\alpha^4) = \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{18} + \alpha^{17} + \alpha^{23} + \alpha^{29}$$

$$\boldsymbol{U}(\alpha^4) = \alpha^0 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1$$

$$\boldsymbol{U}(\alpha^4) = \alpha^2 + \alpha^0 + \alpha^5 + \alpha^1$$

$$\boldsymbol{U}(\alpha^4) = \alpha^6 + \alpha^6 = 0$$

Logo, a palavra código avaliada para qualquer raiz de **g**(X) deve resultar em zero.

2.5 – Decodificação Reed-Solomon

Anteriormente, uma mensagem codificada na forma sistemática utilizando o código RS(7,3) resultou em polinômio descrito pela Equação (2.29). Agora, assumindo que durante a transmissão houve corrupção da sequência de tal forma que dois símbolos tenham sido recebidos com erro, sendo esse o número máximo da capacidade de correção do código RS(7,3). Para os sete símbolos da palavra código, o padrão de erro, **e**(X), pode ser descrito na forma polinomial:

$$\boldsymbol{e}(X) = \sum_{n=0}^{6} e_n X^n \tag{2.31}$$

Suponha, então, que a mensagem corrompida seja:

$$\boldsymbol{e}(X) = 0 + 0X + 0X^2 + \alpha^2 X^3 + \alpha^5 X^4 + 0X^5 + 0X^6 \tag{2.32}$$

$$\boldsymbol{e}(X) = (000) + (000)X + (000)X^2 + (100)X^3 + (111)X^4 + (000)X^5 + (000)X^6$$

Em outras palavras, um símbolo de paridade foi corrompido com um erro de 1 bit (α^2) e outro símbolo corrompido com um erro de 3 bits (α5). O polinômio da palavra código corrompida recebida, r(X), é, então, representado pela soma do polinômio da palavra código transmitido e do polinômio do padrão de erro

$$\boldsymbol{r}(X) = \boldsymbol{U}(X) + \boldsymbol{e}(X) \tag{2.33}$$

Utilizando-se a Equação (2.29) e a Equação (2.32) obtém-se a mensagem recebida após a corrupção dos dados, conforme representada abaixo:

$$\mathbf{r}(X) = (001) + (100)X + (110)X^2 + (001)X^3 + (101)X^4 + (011)X^5 + (111)X^6$$

$$\mathbf{r}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \underline{\alpha^0 X^3} + \underline{\alpha^6 X^4} + \alpha^3 X^5 + \alpha^5 X^6 \quad (2.34)$$

Então, para esse exemplo, têm-se quatro incógnitas, duas localizações de erro e dois valores de erro. Nota-se uma importante diferença entre a decodificação não binária e a binária, pois, na decodificação binária, é necessário apenas encontrar a localização dos erros [11].

O processo de decodificação RS possui diversas etapas, tais como: o cálculo da síndrome, o cálculo dos coeficientes do polinômio localizador de erro, o cálculo das localizações de erro dos valores dos erros. Desta maneira representa-se a arquitetura simplificada de um decodificador Reed-Solomon de acordo com a Figura 2.7 [15]:

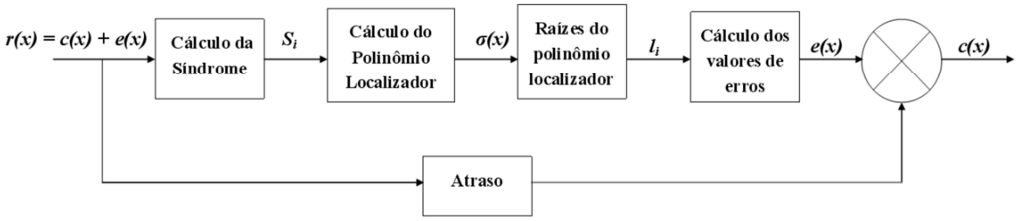

Figura 2.7 – Diagrama de Blocos do Decodificador RS

Existem vários algoritmos que permitem encontrar o polinômio localizador de erros, dentre eles:

- Algoritmo Sugiyama;
- Algoritmo Berlekamp-Massey;
- Algoritmo Fitzpatrick;
- Algoritmo Peterson-Gorenstein-Zierler.

Dentre esses, foi escolhido o algoritmo de Peterson-Gorenstein-Zierler para a decodificação Reed-Solomon apresentada neste trabalho, pois apresenta simples implementação quando a distância mínima é pequena, justificando, assim, o uso de uma implementação utilizando o PicoBlaze.

2.5.1 – Cálculo da Síndrome

A *síndrome* é o resultado de uma verificação de paridade realizada em **r** para determinar se **r** é um membro válido do conjunto de palavras do código [11]. Se, de fato, **r** é membro, a síndrome **S** é igual à zero. Qualquer valor diferente de zero em **S**

indica a presença de erros. Similar ao caso binário, a síndrome **S** é feita de $n - k$ símbolos, $\{S_i\}$ (i = 1, ..., $n - k$). Assim, para o código RS(7,3), existem quatro símbolos no vetor da síndrome.

Olhando para a Equação (2.30), percebe-se que todo polinômio válido de uma palavra código **U**(X) é múltiplo do polinômio gerador **g**(X). Portanto, as raízes de **g**(X) também devem ser raízes de **U**(X). Sendo **r**(X) = **U**(X) + **e**(X), então, **r**(X) avaliado para cada raiz de **g**(X) deve resultar em zero quando a palavra código é válida. Qualquer erro levará a um valor diferente de zero. O cálculo da síndrome pode ser descrito da seguinte forma:

$$S_i = \boldsymbol{r}(X)|_{X=\boldsymbol{\alpha}^i} = \boldsymbol{r}(\alpha^i) \quad i = 1, \dots, n-k \tag{2.35}$$

Caso **r**(X) contenha uma palavra código válida, todo símbolo da síndrome será zero. Para este exemplo, os quatro símbolos da síndrome são assim descritos:

$$S_1 = \boldsymbol{r}(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^{10} + \alpha^8 + \alpha^{11}$$

$$S_1 = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^3 + \alpha^1 + \alpha^4 \tag{2.36}$$

$$S_1 = \alpha^3$$

$$S_2 = \boldsymbol{r}(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^8 + \alpha^6 + \alpha^{14} + \alpha^{13} + \alpha^{17}$$

$$S_2 = \alpha^0 + \alpha^4 + \alpha^1 + \alpha^6 + \alpha^0 + \alpha^6 + \alpha^3 \tag{2.37}$$

$$S_2 = \alpha^5$$

$$S_3 = \boldsymbol{r}(\alpha^3) = \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^9 + \alpha^{18} + \alpha^{18} + \alpha^{23}$$

$$S_3 = \alpha^0 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^4 + \alpha^2 \tag{2.38}$$

$$S_3 = \alpha^6$$

$$S_4 = \boldsymbol{r}(\alpha^4) = \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{12} + \alpha^{22} + \alpha^{23} + \alpha^{29}$$

$$S_4 = \alpha^0 + \alpha^6 + \alpha^5 + \alpha^5 + \alpha^1 + \alpha^2 + \alpha^1 \tag{2.39}$$

$$S_4 = 0$$

Observando-se as equações (2.36), (2.37), (2.38) e (2.39), nota-se que a palavra código recebida contém erros (o qual foi inserido), pois $\mathbf{S} \neq 0$.

2.5.2 – Localização do Erro

Supondo que há ν erros na palavra código nas localizações$X^{j_1}, X^{j_2}, \dots, X^{j_\nu}$. Então, o polinômio de erro **e**(X) apresentado nas equações (2.31) e (2.32) pode assim ser escrito:

$$\boldsymbol{e}(X) = e_{j_1}X^{j_1} + e_{j_2}X^{j_2} + \dots + e_{j_\nu}X^{j_\nu} \tag{2.40}$$

Para corrigir uma palavra código corrompida, cada valor de erro e_{j_l} e sua localização X^{j_l}, onde $l = 1, 2, \dots, \nu$, precisa ser determinado. Define-se como localizador do erro $\beta_l = \alpha^{j_l}$. Então, obtêm-se os $n - k = 2t$ símbolos da síndrome substituindo α^i no polinômio de erro **e**(X) para $i = 1, 2, \dots, 2t$ como mostra a Equação (2.41). É importante observar que as síndromes podem ser tanto avaliadas através da mensagem recebida **r**(X), quanto através do padrão de erro **e**(X), pois, como mostra a Equação (2.33), o polinômio **r**(X) é a soma de **U**(X) e **e**(X), sendo a síndrome de **U**(X) sempre zero [15].

$$\begin{aligned} S_1 &= \boldsymbol{e}(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \dots + e_{j_\nu}\beta_\nu \\ S_2 &= \boldsymbol{e}(\alpha^2) = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \dots + e_{j_\nu}\beta_\nu^2 \\ &\vdots \\ S_{2t} &= \boldsymbol{e}(\alpha^{2t}) = e_{j_1}\beta_1^{2t} + e_{j_2}\beta_2^{2t} + \dots + e_{j_\nu}\beta_\nu^{2t} \end{aligned} \tag{2.41}$$

Existem *2t* incógnitas (*t* valores de erro e *t* localizações) e *2t* equações. Entretanto, estas equações não podem ser resolvidas de forma usual, pois elas são não lineares (algumas incógnitas possuem expoente). Qualquer técnica que resolva esse sistema de equações é conhecida como algoritmo de decodificação Reed-Solomon [11], sendo escolhido o algoritmo de Peterson-Gorenstein-Zierler.

Uma vez que o vetor de síndromes diferente de zero foi calculado, significa que a sequência recebida contém erros. Então, é necessário descobrir a localização dos erros. Um polinômio localizador de erros, **σ**(X), é definido como:

$$\begin{aligned} \boldsymbol{\sigma}(X) &= (1 + \beta_1 X)(1 + \beta_2 X) \dots (1 + \beta_\nu X) \\ \boldsymbol{\sigma}(X) &= 1 + \sigma_1 X + \sigma_2 X^2 + \dots + \sigma_\nu X^\nu \end{aligned} \tag{2.42}$$

As raízes de **σ**(X) são $1/\beta_1$, $1/\beta_2$, ..., $1/\beta_\nu$. Logo, o inverso das raízes de **σ**(X) indicam as localizações de erro do polinômio padrão de erro **e**(X). Então, utilizando a técnica de modelagem auto regressiva [16], forma-se a matriz de síndromes, onde as ν primeiras síndromes são utilizadas para prever a próxima síndrome. Isto é,

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_\nu \\ S_2 & S_3 & \cdots & S_{\nu+1} \\ \vdots & \vdots & \cdots & \vdots \\ S_\nu & S_{\nu+1} & \cdots & S_{2\nu-1} \end{bmatrix} \begin{bmatrix} \sigma_\nu \\ \sigma_{\nu-1} \\ \vdots \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu} \end{bmatrix} \tag{2.43}$$

$$M_\mu = \begin{bmatrix} S_1 & S_2 & \cdots & S_\mu \\ S_2 & S_3 & \cdots & S_{\mu+1} \\ \vdots & \vdots & \cdots & \vdots \\ S_\mu & S_{\mu+1} & \cdots & S_{2\mu-1} \end{bmatrix}$$

Sendo $\mu = t,\ \mu = t-1$, ..., verifica se M_μ é singular ou não, parando no primeiro valor de μ onde M_μ é não-singular. Então, faz-se $\nu = \mu$ e resolve-se (2.43) [17].

Para o código RS(7,3), onde a capacidade máxima de correção é de dois erros, a matriz é 2 x 2 e é apresentada a seguir:

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} = \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \therefore \begin{vmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{vmatrix} = \alpha^2 + \alpha^3 = \alpha^5$$

Para evitar funcionamento incorreto do algoritmo, verifica-se, também, a singularidade da seguinte matriz:

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_\nu & S_{\nu+1} \\ S_2 & S_3 & \cdots & S_{\nu+1} & S_{\nu+2} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ S_\nu & S_{\nu+1} & \cdots & S_{2\nu-1} & S_{2\nu} \\ S_{t+1} & S_{t+2} & \cdots & S_{t+\nu} & S_{t+\nu+1} \end{bmatrix}$$

Caso a matriz acima seja não singular, pode-se anunciar a falha e parar a decodificação [18].

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \tag{2.44}$$

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} \tag{2.45}$$

Utilizando a matriz inversa da matriz 2 x 2 é possível obter-se os valores dos coeficientes do polinômio localizador de erro, σ_1 e σ_2. A inversão de matrizes em campos finitos leva em consideração as mesmas propriedades utilizadas nos conjuntos

numéricos conhecidos, porém, sempre considerando as operações de adição e multiplicação restritas ao campo finito [15]. Logo,

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

$$\begin{bmatrix}\alpha^1 & \alpha^0\\ \alpha^0 & \alpha^5\end{bmatrix}\begin{bmatrix}\alpha^6\\ 0\end{bmatrix}=\begin{bmatrix}\alpha^0\\ \alpha^6\end{bmatrix}=\begin{bmatrix}\sigma_2\\ \sigma_1\end{bmatrix} \quad (2.46)$$

A partir dos coeficientes encontrados na Equação (2.46) e da Equação (2.42), é possível representar o polinômio localizador de erro da seguinte forma:

$$\boldsymbol{\sigma}(X)=\alpha^0+\alpha^6X+\alpha^0X^2 \quad (2.47)$$

As raízes do polinômio **σ**(X) são as inversas das localizações de erro. Uma vez encontradas, as localizações de erro serão conhecidas. Estas raízes podem ser encontradas através de um teste exaustivo do polinômio **σ**(X) para todos os elementos do campo, como mostrado na Equação (2.48), ou seja, qualquer elemento do campo que resultar **σ**(X) = 0 é uma raiz e corresponde a localização de erro.

$$\boldsymbol{\sigma}(\alpha^0)=\alpha^0+\alpha^6+\alpha^0=\alpha^6\neq 0$$

$$\boldsymbol{\sigma}(\alpha^1)=\alpha^0+\alpha^7+\alpha^2=\alpha^2\neq 0$$

$$\boldsymbol{\sigma}(\alpha^2)=\alpha^0+\alpha^8+\alpha^4=\alpha^6\neq 0$$

$$\boldsymbol{\sigma}(\alpha^3)=\alpha^0+\alpha^9+\alpha^6=0 => ERRO \quad (2.48)$$

$$\boldsymbol{\sigma}(\alpha^4)=\alpha^0+\alpha^{10}+\alpha^8=0 => ERRO$$

$$\boldsymbol{\sigma}(\alpha^5)=\alpha^0+\alpha^{11}+\alpha^{10}=\alpha^2\neq 0$$

$$\boldsymbol{\sigma}(\alpha^6)=\alpha^0+\alpha^{12}+\alpha^{12}=\alpha^0\neq 0$$

Percebe-se que os elementos α^3 e α^4 são raízes do polinômio **σ**(X). Logo, existem erros nas localizações $\beta_1 = 1/\alpha^4 = \alpha^3$ e $\beta_2 = 1/\alpha^3 = \alpha^4$, pois as raízes de **σ**(X) equivalem ao inverso das localizações de erro.

2.5.3 – Valores de Erro

Um erro foi denotado como e_{j_l}, onde o índice j se refere à localização do erro e o índice l identifica a ordem do erro. Sendo cada valor de erro associado a uma localização em particular, a notação pode ser simplificada para e_l. Para determinar os

valores de erro e_1 e e_2 associados às posições encontradas, $\beta_1 = \alpha^3$ e $\beta_2 = \alpha^4$, qualquer uma das quatro equações de síndromes pode ser utilizada. A partir da Equação (2.41), usando S_1 e S_2, tem-se:

$$S_1 = \boldsymbol{e}(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2$$

$$S_2 = \boldsymbol{e}(\alpha^2) = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 \tag{2.49}$$

Então, na forma matricial, tem-se:

$$\begin{bmatrix}\beta_1 & \beta_2 \\ \beta_1^2 & \beta_2^2\end{bmatrix}\begin{bmatrix}e_1 \\ e_2\end{bmatrix} = \begin{bmatrix}S_1 \\ S_2\end{bmatrix}$$

$$\begin{bmatrix}\alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1\end{bmatrix}\begin{bmatrix}e_1 \\ e_2\end{bmatrix} = \begin{bmatrix}\alpha^3 \\ \alpha^5\end{bmatrix} \tag{2.50}$$

Então, utilizando a mesma técnica utilizada na Equação (2.46), tem-se:

$$\begin{bmatrix}\alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1\end{bmatrix}^{-1}\begin{bmatrix}S_1 \\ S_2\end{bmatrix} = \begin{bmatrix}e_1 \\ e_2\end{bmatrix}$$

$$\begin{bmatrix}\alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4\end{bmatrix}\begin{bmatrix}\alpha^3 \\ \alpha^5\end{bmatrix} = \begin{bmatrix}\alpha^2 \\ \alpha^5\end{bmatrix} = \begin{bmatrix}e_1 \\ e_2\end{bmatrix} \tag{2.51}$$

2.5.4 – Correção do Polinômio Recebido

Com as localizações e os valores de erro encontrados, é possível estimar o polinômio de erro **ê**(X), conforme a Equação abaixo:

$$\hat{\mathrm{e}}(X) = e_1 X^{j_1} + e_2 X^{j_2}$$

$$\hat{\mathrm{e}}(X) = \alpha^2 X^3 + \alpha^5 X^4 \tag{2.52}$$

O algoritmo apresentado deve decodificar corretamente a mensagem recebida, uma vez que ela foi corrompida em dois símbolos, que é exatamente a capacidade de correção do código RS(7,3). Para confirmar se a decodificação foi executada corretamente, é necessário estimar a sequência transmitida, **Û**(X), e verificar se representa a verdadeira mensagem transmitida, **U**(X) [15]. Então,

$$\hat{\mathrm{U}}(X) = \boldsymbol{r}(X) + \hat{\mathrm{e}}(X) \tag{2.53}$$

Utilizando a Equação (2.34) e a Equação (2.52), encontra-se o seguinte polinômio:

$$\hat{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^0 X^3 + \alpha^6 X^4 + \alpha^3 X^5 + \alpha^5 X^6 + \alpha^2 X^3 + \alpha^5 X^4$$

$$\hat{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + (\alpha^0 + \alpha^2) X^3 + (\alpha^6 + \alpha^5) X^4 + \alpha^3 X^5 + \alpha^5 X^6$$

$$\hat{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \quad (2.54)$$

$$\hat{U}(X) = (001) + (100)X + (110)X^2 + (101)X^3 + (010)X^4 + (011)X^5 + (111)X^6$$

Sendo os símbolos da mensagem os k = 3 símbolos mais a direita, a mensagem decodificada é:

$$\underbrace{010}_{\alpha^1}\underbrace{011}_{\alpha^3}\underbrace{111}_{\alpha^5}$$

que é exatamente a mensagem teste que foi escolhida anteriormente para exemplo.

Capítulo 3

Desenvolvimento de Hardware

3.1 – FPGA

FPGA é um dispositivo lógico que contém um *array* bidimensional de células lógicas genéricas, formado por blocos de entrada e saída, e conectado por fios de interconexão, onde todos esses itens são configuráveis. A computação reconfigurável é uma solução intermediária na resolução de problemas complexos, possibilitando combinar a velocidade do *hardware* com a flexibilidade do *software* [19].

O termo “programável” em FPGA indica uma habilidade de programar uma função no chip mesmo depois que a fabricação de silício esteja completa, isto é, são dispositivos de silício pré-fabricados que podem ser programados eletricamente em quase todo tipo de circuito ou sistema digital. Esta tecnologia oferece algumas vantagens quando comparadas com *Application Specific Integrated Circuit* – ASIC, como custo do primeiro dispositivo e tempo de fabricação mais baixos [20].

A estrutura conceitual de um FPGA é apresentada na Figura 3.1. Uma célula lógica pode ser configurada para executar uma simples função, e um *switch* programável pode ser personalizado para prover interconexões entre as células lógicas. Um projeto personalizado pode ser implementado especificando a função de cada célula lógica e seletivamente fazendo a conexão de cada *switch* programável. Uma vez que o projeto e síntese estejam completos, é gerado um arquivo binário com as informações necessárias especificando a função de cada unidade lógica e fechando as chaves da matriz de interconexões. Usando um simples cabo adaptador, faz-se o *download* da configuração desejada para o dispositivo FPGA, obtendo-se um circuito personalizado [21].

Os recursos adicionais como flip-flops, multiplexadores, lógica de transporte (*carry*) dedicado, e portas lógicas, podem ser utilizados em conjunto com os *Look-Up Tables* – LUT para implementar diversas funções booleanas, multiplicadores e somadores, contadores, conversores, e memórias com praticamente qualquer comprimento de palavra, fornecendo flexibilidade ao desenvolvimento [19].

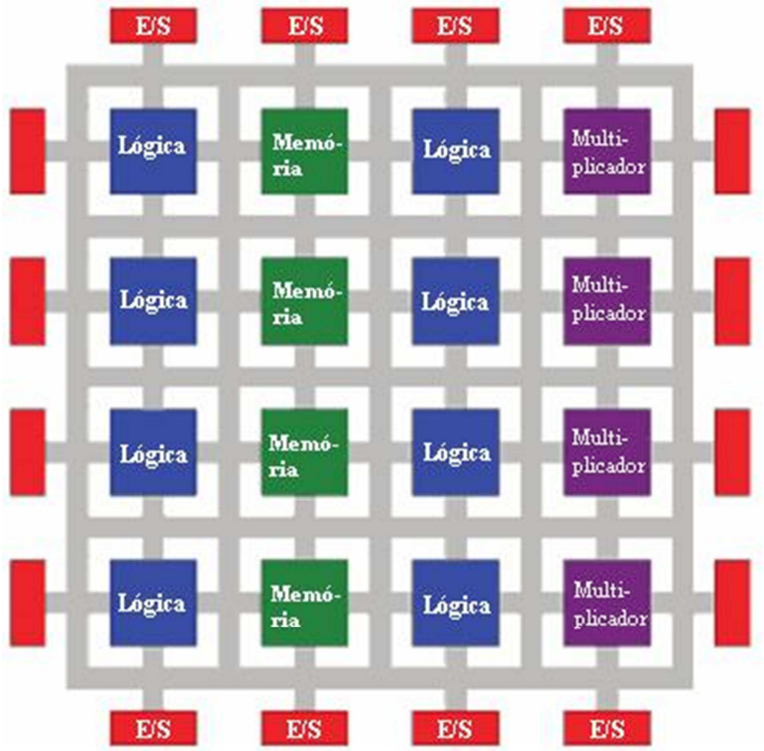

Figura 3.1 – Estrutura Básica da FPGA

3.2 – VHDL

VHSIC Hardware Description Language – VHDL (linguagem para descrição de *hardware* de circuitos integrados de altíssima velocidade) é uma linguagem para descrição de hardware, onde descreve o comportamento de um circuito ou sistema eletrônico. Tal linguagem foi uma iniciativa do Departamento de Defesa Norte Americano – *DoD* nos anos 80, sendo sua primeira versão o VHDL 87, depois atualizada para VHDL 93. Foi a primeira linguagem para descrição de *hardware* padronizada pelo *Institute of Electrical and Electronics Engineers* – IEEE através do padrão IEEE 1076.

Uma das motivações fundamentais para o uso dessa linguagem é o fato de ser uma linguagem padrão independente de fornecedor, tecnologia ou processo, portanto, portável e reutilizável. Uma vez que o código tenha sido escrito, ele pode ser utilizado para implementar o circuito em um dispositivo programável (como FPGA) ou submetido para fabricação de um chip ASIC.

Contrariamente aos habituais programas de computadores, que são sequenciais, suas declarações são inerentemente *concorrentes* [22]. A linguagem permite delimitar regiões onde o código é executado sequencialmente, sendo denominada tal região como *processo*. Porém, todos os processos são executados concorrentemente.

O VHDL é uma linguagem que permite ao projetista um alto nível de abstração para descrever o circuito a ser implementado. Portanto, foi escolhida como linguagem para descrição de hardware neste projeto devido ao conhecimento e experiência já existentes e sua ampla utilização.

3.3 – Micro controlador PicoBlaze

Existem literalmente dúzias de micro controladores com arquiteturas e conjunto de instruções de 8-bits. As modernas FPGAs podem eficientemente implementar praticamente qualquer micro controlador de 8-bits, e *soft-cores* disponíveis suportam conjuntos de instruções mais comuns, como o PIC, 8051, AVR, 6502, 8080 e micro controladores Z80. O micro controlador Xilinx PicoBlaze foi especificamente projetado e otimizado para a série de FPGAs Virtex e Spartan, e CPLDs CoolRunner-II [23].

A solução PicoBlaze consome consideravelmente menos recursos que comparáveis micro controladores com arquiteturas de 8-bits. O código-fonte em VHDL é livre para utilização e alteração para ser utilizado nas FPGAs da Xilinx. Uma vez que o código-fonte em VHDL é oferecido, o micro controlador PicoBlaze não se torna obsoleto, pois o micro controlador pode se adaptar a futuras gerações de FPGAs Xilinx, explorando futuras reduções de custo e futuras funcionalidades [23].

O Assembler PicoBlaze é oferecido como um simples arquivo executável em DOS. O Assembler compila o programa em menos de 3 segundos e gera o VHDL, Verilog e um arquivo M (pelo Sistema de Geração Xilinx) para definir o programa dentro de um bloco de memória. Outras ferramentas de desenvolvimento incluem ambientes de desenvolvimento integrados gráficos (IDE), simulador gráfico do conjunto de instruções (ISS), código-fonte em VHDL e modelos de simulações [23].

O PicoBlaze executa de 44 a 100 milhões de instruções por segundo (MIPS) dependendo da família da FPGA e *speed grade* – muito mais rápido que dispositivos micro controladores disponíveis no mercado [23].

PicoBlaze ocupa 192 células lógicas, ou seja, apenas 5% de uma Spartan-3 XC3S200. Uma vez que o núcleo consome apenas uma pequena fração dos recursos da FPGA ou CPLD, muitos engenheiros podem utilizar múltiplos dispositivos PicoBlaze para resolver maiores tarefas ou simplesmente manter tarefas isoladas [23].

O núcleo do micro controlador PicoBlaze é totalmente incorporado dentro da FPGA ou CPLD sem requerer nenhum recurso externo. A funcionalidade básica é facilmente estendida ou ampliada conectando lógica adicional às portas de entrada e saída do micro controlador [23]. A Figura 3.2 apresenta o diagrama de blocos do PicoBlaze.

As principais características do PicoBlaze são:

- 16 registradores de dados de uso geral de 1 byte cada;
- 1024 instruções programáveis armazenadas on-chip, automaticamente carregadas durante a configuração da FPGA;
- Unidade Lógica-Aritmética– ULA de 8 bits com indicadores de flag de CARRY e ZERO;
- Memória – RAM interna de 64 bytes;
- 256 portas de entrada e 256 portas de saída para fácil expansão;
- Pilha de localização automática CALL/RETURN (31 níveis);
- Desempenho previsível, sempre dois ciclos de clock por instrução, até 200 MHz ou 100 MIPS na Virtex-4 e 88 MHz ou 44 MIPS na Spartan-3;
- Rápida resposta a interrupções (cinco ciclos de clock no pior caso);

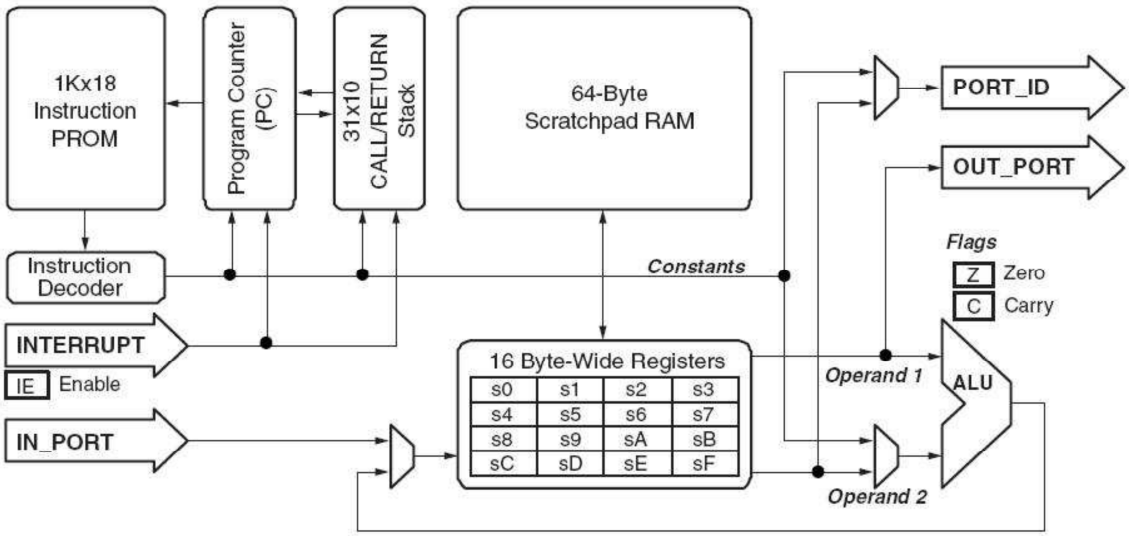

Figura 3.2 – Diagrama de bloco – PicoBlaze

O desempenho do PicoBlaze é afetado principalmente pela família de FPGAs onde o mesmo encontra-se implementado. A Tabela 3.1 apresenta as diferenças de desempenho e recursos para três diferentes versões do PicoBlaze.

Tabela 3.1 – Comparação de desempenho e recursos do PicoBlaze

Característica	PicoBlaze para Spartan3 Virtex-II/Pro e Virtex-4	PicoBlaze para VirtexE e SpartanII/E	PicoBlaze para CoolRunnerII
Memória de código	1024	256	256
Largura da instrução	18 bits	16 bits	16 bits
Registradores de 8 bits	16	16	8
Profundidade da pilha	31	15	4
Ocupação	96 slices na Spartan 3	76 Slices na Spartan II E	212 Macrocélulas na XC2C256
Performance	44 MIPS (Spartan-3) 76 MIPS (Virtex-II) 100 MIPS (Virtex-II Pro) 100 MIPS (Virtex-4 LX, SX) 102 MIPS (Virtex-4 FX)	37 MIPS (Spartan-IIE)	21 MIPS

O simples fato de qualquer lógica poder ser conectada ao módulo dentro da FPGA significa que qualquer recurso adicional pode ser adicionado provendo máxima flexibilidade. A Figura 3.3 mostra as interfaces de comunicação do Picoblaze e a Tabela 3.2 explica essas interfaces.

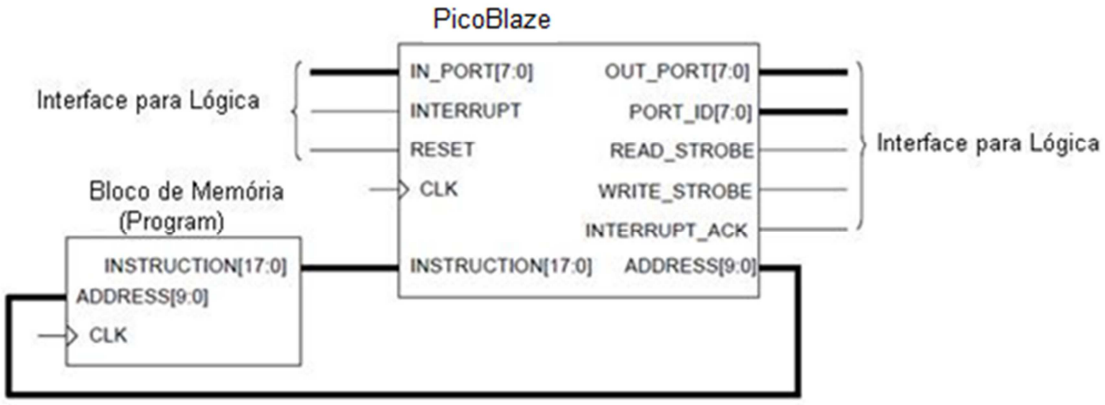

Figura 3.3 – Conexões das interfaces do PicoBlaze

Tabela 3.2 – Interfaces de comunicação – PicoBlaze

Sinal	Direção	Descrição
IN_PORT [7:0]	Entrada	**Porta de entrada de dados:** Captura os dados durante as instruções de entrada. A sincronização ocorre na borda de subida do ciclo de clock.
INTERRUPT	Entrada	**Interrupção:** Se o "*flag*" INTERRUPT_ENABLE estiver em nível lógico alto no código-fonte, será gerado um evento de interrupção que permanecerá em pelo menos dois ciclos de clock. Caso o "*flag*" INTERRUPT_ENABLE estiver em nível lógico baixo, a entrada será ignorada.
RESET	Entrada	**Reset:** Para reiniciar o micro controlador PicoBlaze e para gerar um evento RESET, deve-se habilitar essa porta com nível lógico alto por pelo menos um ciclo de clock.
CLK	Entrada	**Clock:** A frequência pode variar entre DC e a frequência máxima de funcionamento relatado no *software* de desenvolvimento da Xilinx. A sincronização dos elementos do PicoBlaze é feita na borda de subida do *clock*.
INSTRUCTION [17:0]	Entrada	**Instrução:** Código de 18 bits representando a instrução armazenada no bloco de memória do programa, sendo a instrução e os possíveis registradores ou operandos.
OUT_PORT [7:0]	Saída	**Porta de saída de dados:** A saída de dados nesta porta utiliza dois ciclos de clock durante uma instrução OUTPUT. A captura de dados de saída dentro da FPGA ocorre na borda de subida do *clock* quando o WRITE_STROBE está em nível lógico alto.
PORT_ID[7:0]	Saída	**Endereço de porta:** os endereços das portas de entrada e saída (E/S) aparecem nesta porta durante dois ciclos de *clock* durante uma instrução de entrada ou saída.
READ_STROBE	Saída	**Readstrobe:** Quando estiver em nível lógico alto, esse sinal indica que os dados de entrada na porta IN_PORT[7:0] serão capturados segundo seus registradores durante as instruções de entrada.
WRITE_ STROBE	Saída	**Write strobe:** Quando estiver em nível lógico alto, valida a saída de dados sobre a porta OUT_PORT[7:0] durante uma instrução de saída.
INTERRUPT_ ACK	Saída	**Reconhecimento de interrupção:** Quando em nível lógico alto, reconhece que um evento de interrupção ocorreu. Esse sinal é reconhecido durante o segundo ciclo de *clock* do evento INTERRUPT.
ADRESS[9:0]	Saída	**Endereço da instrução:** Endereço da instrução no bloco de memória do programa.

O PicoBlaze pode utilizar seus próprios recursos de memória embarcada no FPGA para a execução do código. Como pode ser observado na Figura 3.3, ele possui um bloco definido como *Program*, responsável por armazenar o código da aplicação em questão.

O outro bloco da Figura 3.3, o PicoBlaze, contém a unidade lógica e aritmética – ALU, os registradores, a memória de dados (*scratchpad RAM*), e outros. Neste bloco são executadas as instruções armazenadas no bloco *Program*, que irá interagir com o resto do sistema através da interface de comunicação [24].

Capítulo 4

Implementação e Resultados

4.1 – Visão Geral

A Figura 4.1 mostra os passos para a implementação tanto do codificador quanto do decodificador Reed-Solomon implementado neste projeto. Primeiramente, é criado um arquivo .PSM contendo o código do codificador ou decodificador e, em seguida, utilizando-se o *software* da pBlazeIDE da Mediatronix, é gerado um arquivo .VHD. Este arquivo corresponde ao bloco *program* do PicoBlaze apresentado na seção anterior, que será compilado junto à descrição do bloco do PicoBlaze na plataforma ISE da Xilinx. Então, é gerado o arquivo BITSTREAM, o arquivo de configuração que tem por finalidade programar o *hardware*.

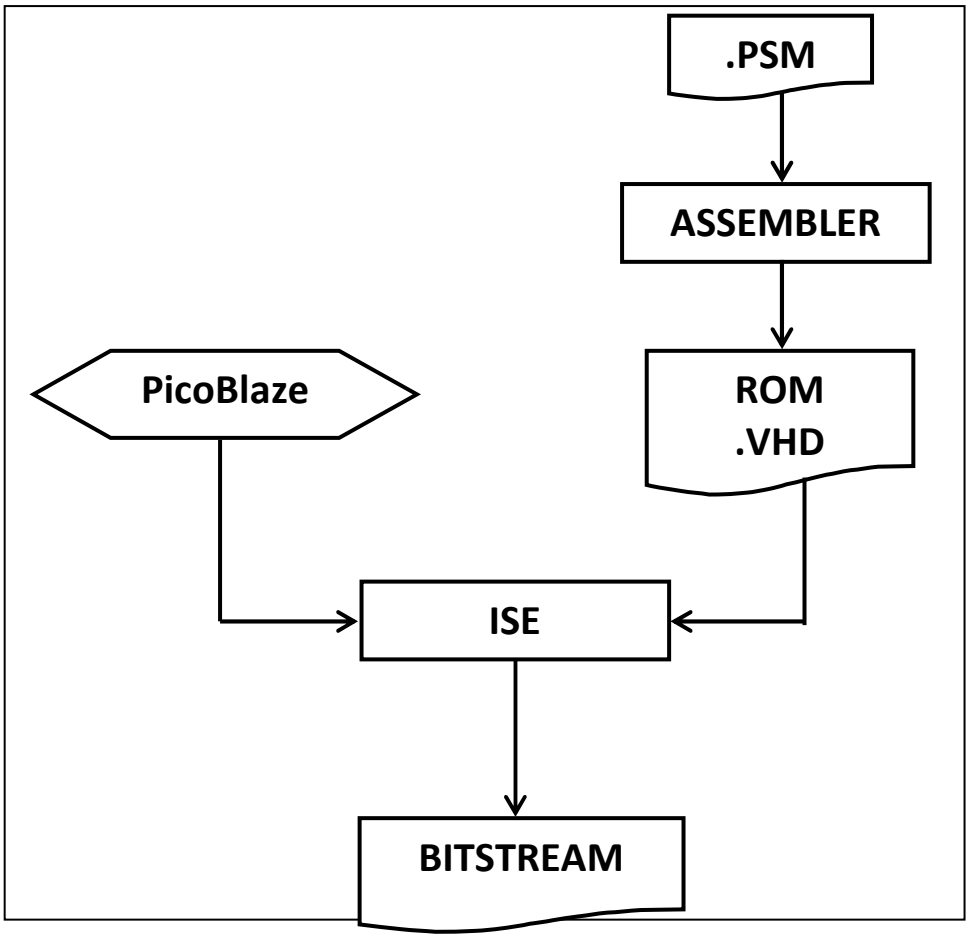

Figura 4.1 – Etapas da Implementação

As interfaces do Codificador/Decodificador no PicoBlaze foram codificadas em VHDL, utilizando-se o ambiente de desenvolvimento ISE 10.1 da Xilinx.

Neste trabalho foram feitas duas implementações de códigos Reed-Solomon, o RS(7,5) e o RS(15,11). Sendo a Taxa de Código igual a razão entre a quantidade de símbolos da mensagem original e a quantidade de símbolos da palavra-código (k/n), foram levantadas as taxas de alguns códigos e, então, apresentada na Tabela 4.1. Neste trabalho implementaram-se códigos Reed-Solomon que tivessem taxas de código próximas e que coubessem no *scratchpad RAM* do PicoBlaze (64 bytes), logo, o RS(31,23) não foi escolhido, pois mesmo possuindo taxa de código próxima, mas não pode ser implementado utilizando somente o *scratchpad RAM* do PicoBlaze.

Tabela 4.1 – Comparação entre Códigos Reed-Solomon

	RS(7,3)	**RS(7,5)**	**RS(15,7)**	**RS(15,9)**	**RS(15,11)**	**RS(15,13)**	**RS(31,23)**
Qtde Elementos	8	8	16	16	16	16	32
Memória Mínima	23	23	47	47	47	47	95
Taxa	0,429	0,714	0,467	0,600	0,733	0,867	0,742

4.2 – Arquivo .PSM

Primeiramente, seguindo a descrição apresentada no Capítulo 2, tanto para o codificador quanto para o decodificador, é escrito um código em linguagem Assembly contendo as instruções que implementam o circuito desejado. Assim, para implementar o codificador RS(15,11), foi escrito e simulado um código em Assembly utilizado o *software* pBlazeIDE da Mediatronix que implementa os passos para executar a codificação RS(15,11), conforme apresentado no Capítulo 2.

A Figura 4.2 apresenta um exemplo de um código de um divisor de 8 bits escrito e simulado no *software* pBlazeIDE. O Apêndice A apresenta, ainda, o exemplo do codificador RS(7,3) em linguagem Assembly.

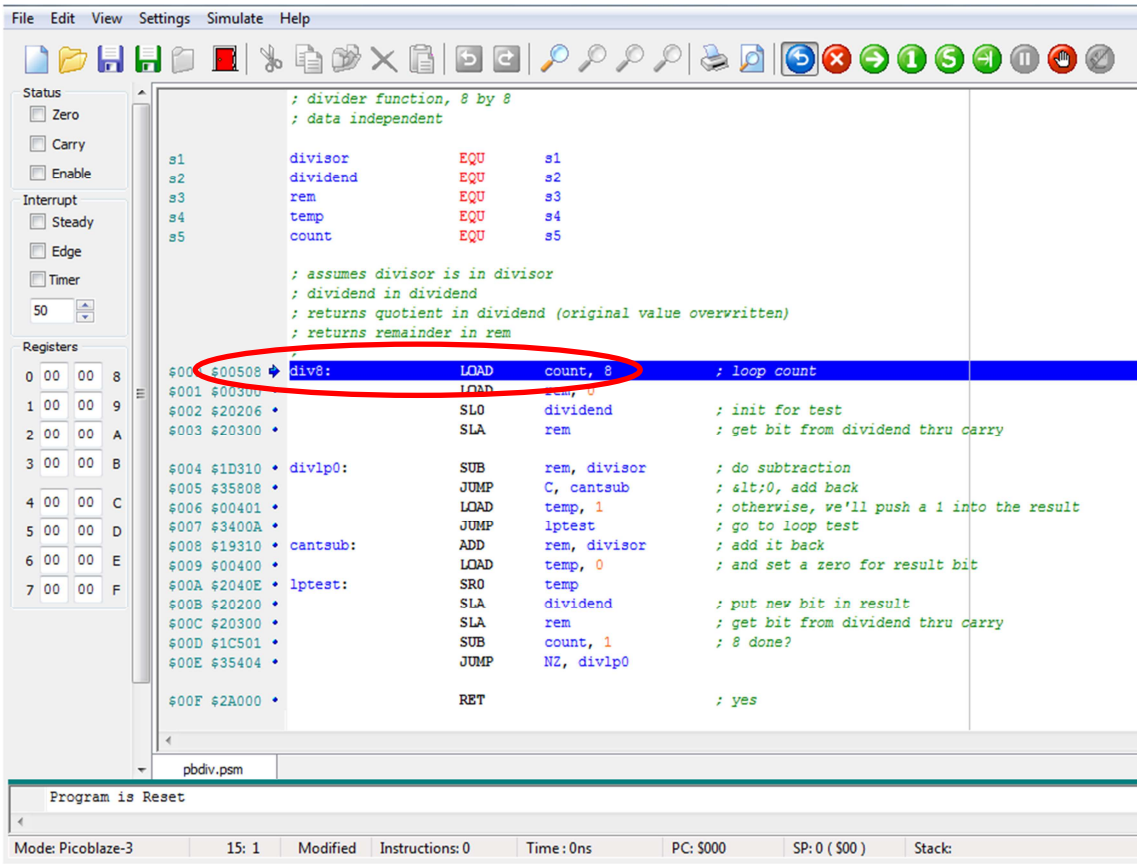

Figura 4.2 – Interface do Software pBlazeIDE

4.3 – Arquivo .VHD

Uma vez que o código em Assembly esteja finalizado, é gerado, a partir de um arquivo ROM template, um arquivo .VHD que é o bloco com as instruções a serem seguidas pelo PicoBlaze. O comando mencionado anteriormente é na forma: VHDL “template.vhd”, “target.vhd”, “ROM”. Utilizando esse comando e o arquivo *template* disponibilizado, gerou-se um arquivo .VHD que contém as instruções, conforme fragmento apresentado na Figura 4.3.

Observando-se a Figura 4.2, nota-se que o primeiro comando é “LOAD count, 8” e, no lado esquerdo, “00508”. O número hexadecimal “00508” representa o comando “LOAD”: registrador “5” com o número “8” [25]. Então, na Figura 4.3, percebe-se que o primeiro valor da direita para esquerda é o valor mencionado.

```
--
begin
--
  --Instantiate the Xilinx primitive for a block RAM
  ram_1024_x_18: RAMB16_S18
  --synthesis translate_off
  --INIT values repeated to define contents for functional simulation
  generic map ( INIT_00 => X"A0005404C50103000200040E04009310400A04015808D3100300020603000508",
                INIT_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_10 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_11 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_12 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_13 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_14 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_15 => X"0000000000000000000000000000000000000000000000000000000000000000",
                INIT_16 => X"0000000000000000000000000000000000000000000000000000000000000000",
```

Figura 4.3 – Fragmento do Arquivo .VHD

4.4 – Projeto

Foi criado um projeto utilizando o *software* ISE 10.1 da Xilinx contendo o código VHDL do PicoBlaze e o arquivo gerado na seção anterior, o qual contém as instruções a serem seguidas pelo PicoBlaze.

Para verificar o correto funcionamento, foram criados arquivos denominados Test Bench, isto é, arquivos que testam o projeto através de sinais ou estímulos, monitorando suas respostas. Com tais arquivos, foram geradas simulações para verificar o funcionamento do codificador e decodificador, tanto para RS(7,5) quanto para o RS(15,11). As simulações são apresentadas através das Figuras 4.4, 4.5, 4.6 e 4.7.

Na Figura 4.4 a mensagem a ser codificada é carregada nas portas “msg_0” a “msg_4” e a mensagem codificada é encontrada, após o processo de codificação, nas portas “cod_0” a “cod_6”. A Figura 4.5 apresenta a operação inversa, a mensagem corrompida a ser decodificada nas portas “cod_0” a “cod_6” e a mensagem corrigida nas portas “msg_0” a “msg_4”. As Figuras 4.6 e 4.7 seguem o mesmo padrão.

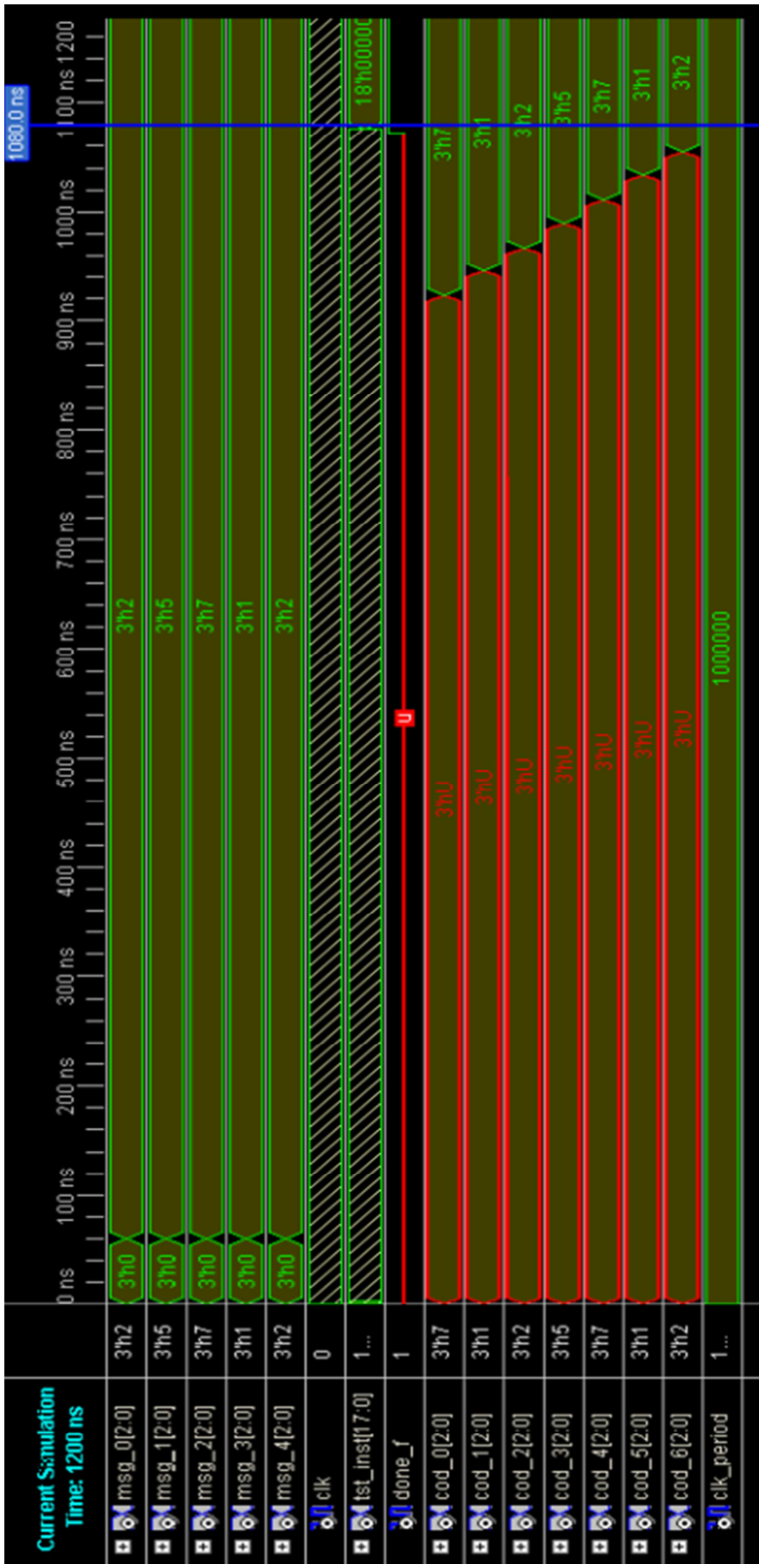

Figura 4.4 – Simulação da Codificação RS(7,5)

Considerando o teste apresentado na Figura 4.4, percebe-se que a mensagem original é, em hexadecimal, “2 5 7 1 2”. Então, como apresentado no Capítulo 2, foi inserida a redundância à mensagem deslocada. Logo, a mensagem codificada em hexadecimal é “7 1 2 5 7 1 2”.

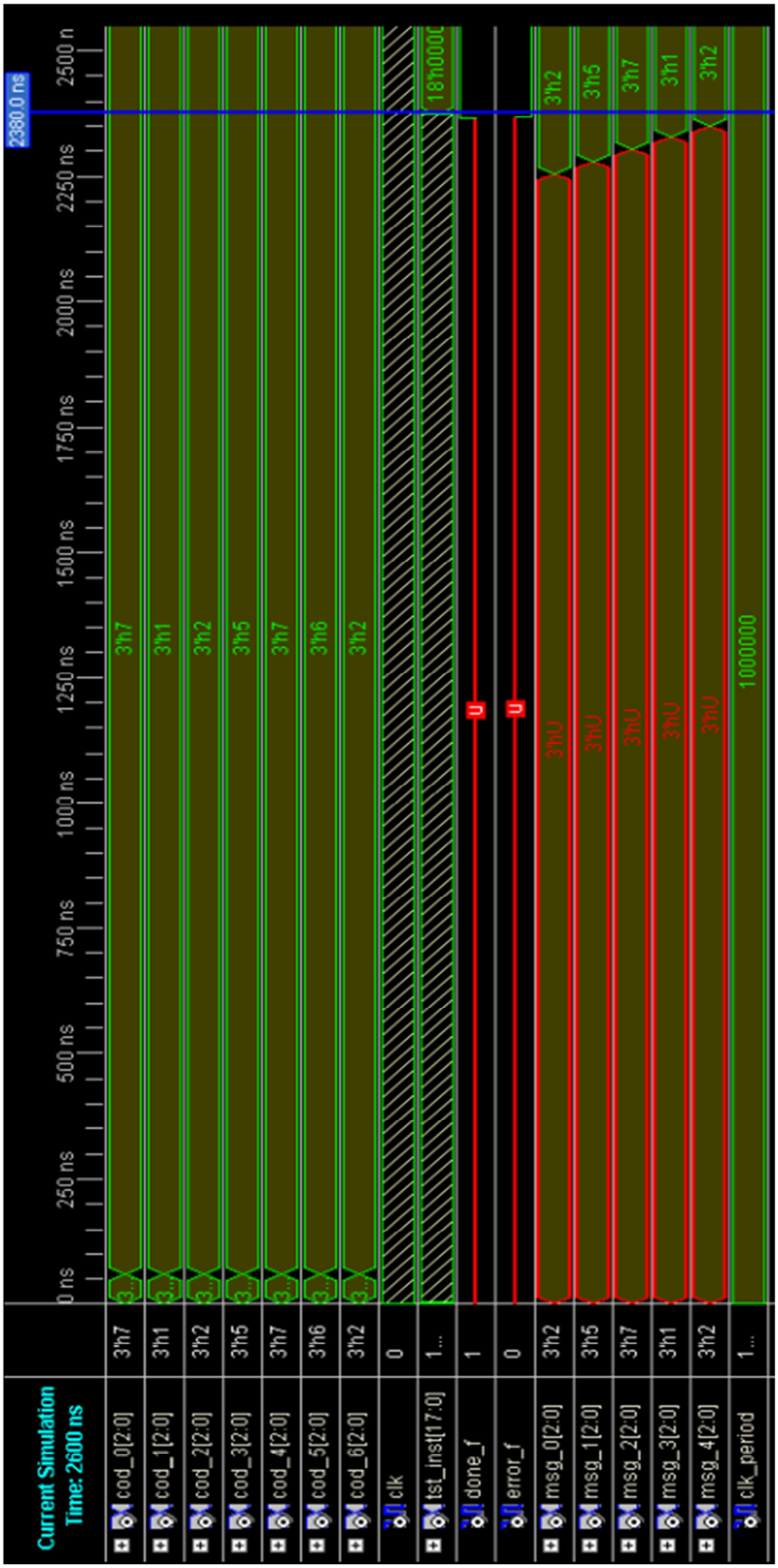

Figura 4.5 – Simulação da Decodificação RS(7,5)

Para testar o decodificador, considerando a mensagem codificada apresentada na Figura 4.4, foi inserido um erro ao penúltimo símbolo. Logo, a

mensagem a ser decodificada em hexadecimal é “7 1 2 5 7 **6** 2”. Observando a Figura 4.5, percebe-se que, após o processo de decodificação, a mensagem decodificada é “2 5 7 1 2”, igual à mensagem original apresentada na Figura 4.4.

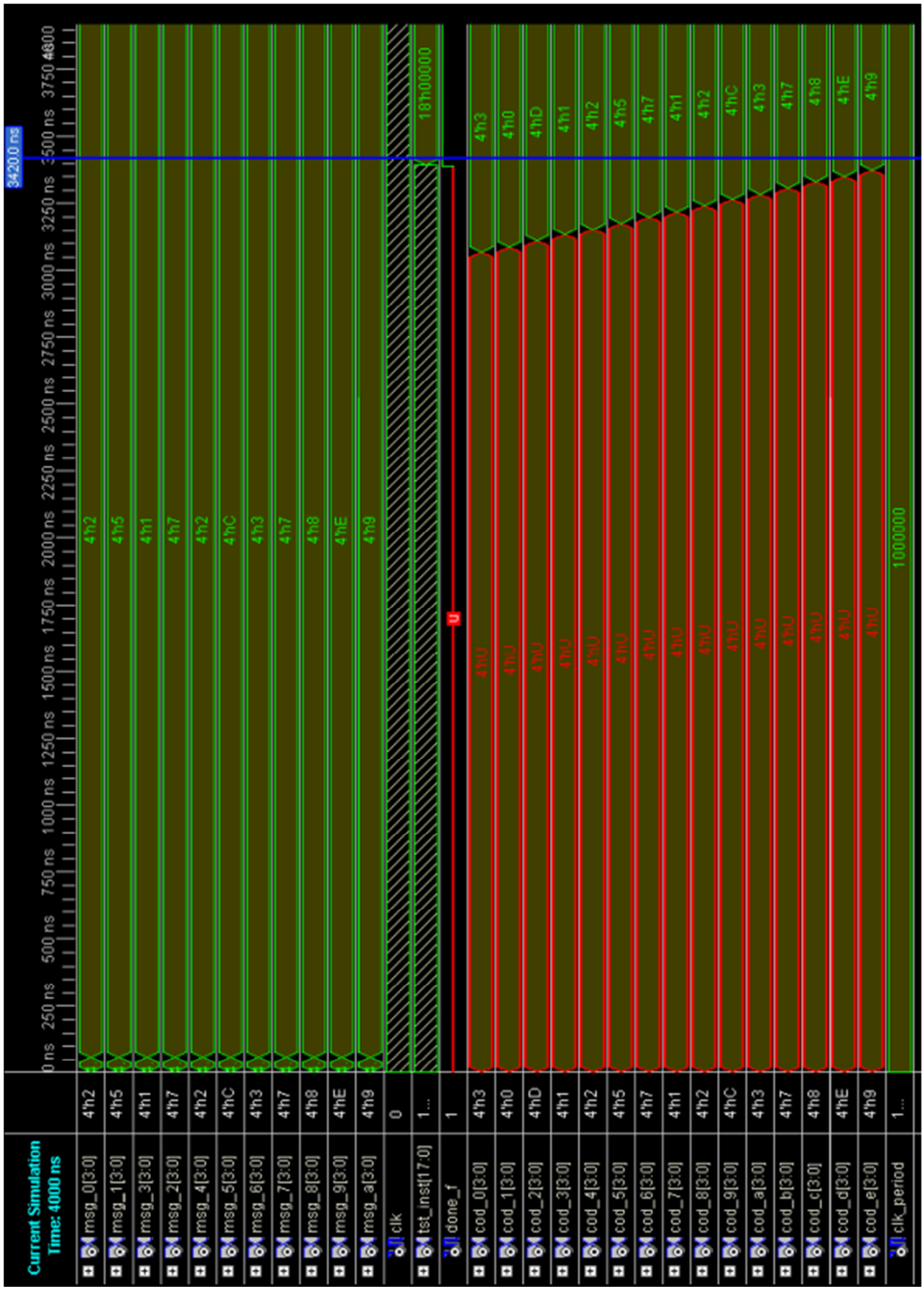

Figura 4.6 – Simulação da Codificação RS(15,11)

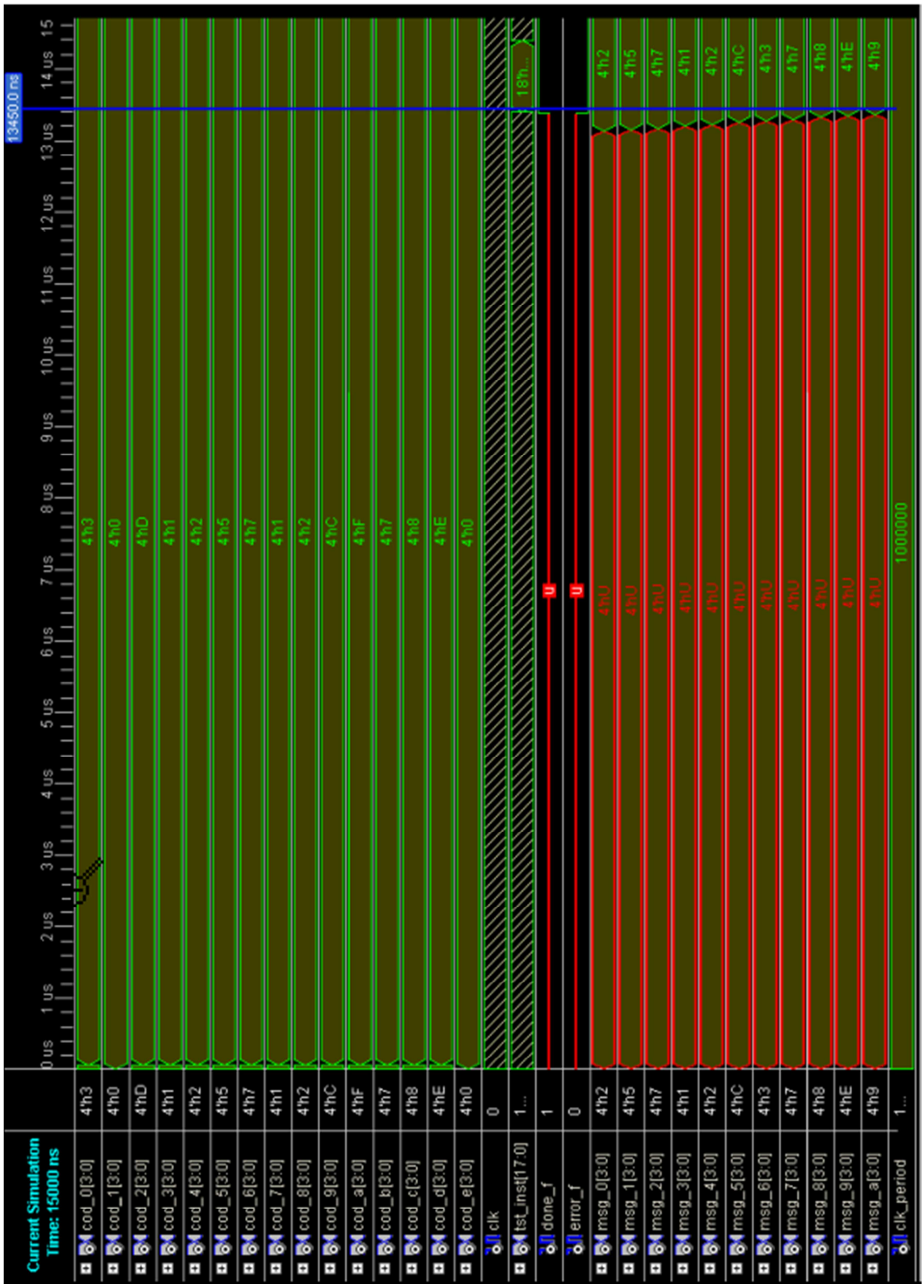

Figura 4.7 – Simulação da Decodificação RS(15,11)

Considerando o teste apresentado na Figura 4.6, percebe-se que a mensagem original é, em hexadecimal, “2 5 7 1 2 C 3 7 8 E 9”. Então, como apresentado no Capítulo 2, foi inserida a redundância à mensagem deslocada. Logo, a mensagem codificada em hexadecimal é “3 0 D 1 2 5 7 1 2 C 3 7 8 E 9”.

Para testar o decodificador, considerando a mensagem codificada apresentada na Figura 4.6, foi inserido um erro ao décimo primeiro e décimo quinto símbolo. Logo, a mensagem a ser decodificada em hexadecimal é “3 0 D 1 2 5 7 1 2 C **F** 7 8 E **0**”. Observando a Figura 4.7, percebe-se que, após o processo de decodificação, a mensagem decodificada é “2 5 7 1 2 C 3 7 8 E 9”, igual à mensagem original apresentada na Figura 4.6.

Então, uma vez observado o correto funcionamento, foi criado o arquivo BitStream utilizando o *software* ISE 10.1 da Xilinx, conforme apresentado na Figura 4.1.

4.5 – Ciclos de *Clock*

Foi utilizado um período, *T*, igual a 1*ns* (*f = 1 GHz*) para executar as simulações apresentadas nas Figuras 4.4, 4.5, 4.6 e 4.7. A Tabela 4.2 apresenta as quantidades aproximadas do número de ciclos de clock.

Tabela 4.2 – Quantidade Aproximada de Ciclos de Clock

	RS(7,5)	**RS(15,11)**
Codificação	1080	3420
Decodificação	2380	13450
TOTAL	**3460**	**16870**

A quantidade de ciclos de *clock* para executar a codificação RS(15,11) é três vezes a quantidade para executar a codificação RS(7,5). Por outro lado, a quantidade exigida na decodificação RS(15,11) é aproximadamente seis vezes a quantidade exigida no caso RS(7,5).

Considerando a implementação em um FPGA da família Spartan3 da Xilinx, o desempenho previsível acontece quando o PicoBlaze é submetido a uma frequência igual a *88 MHz (T = 11,36ns).*

4.6 – Cálculos do Ganho da Codificação RS

Primeiramente, utilizando a Equação (2.3) e a Equação (2.4) apresentadas no Capítulo 2, é possível obter-se um gráfico BER versus E_b/N_0 utilizando o *software* MatLab 6.5 da MathWorks para o sinal não codificado e as duas codificações apresentadas anteriormente, RS(7,5) e RS(15,11), apresentado na Figura 4.8.

Assim, como mostrada no Capítulo 2, define-se um BER para verificar o valor SNR associado para as três situações, sinal não codificado e sinal codificado tanto com RS(7,5) quanto com RS(15,11). Para realização dos cálculos nessa seção, são considerados dois valores de BER, 10^{-3} e 10^{-7}. O primeiro valor foi escolhido como referência [2], para codificação de canal de uma rede de sensores sem fio (Wireless Sensor Network – WSN) de aplicações médicas considerando monitoramento remoto de pacientes. Observa-se que 10^{-3} é um valor alto, ou seja, representa uma alta probabilidade de erro para o canal. Já o segundo valor foi escolhido como referência [26], uma modulação BPSK para dispositivos médicos implantáveis. Portanto, utilizando esses dois valores, verificou-se o ganho que cada codificação implementada traz ao sinal não codificado.

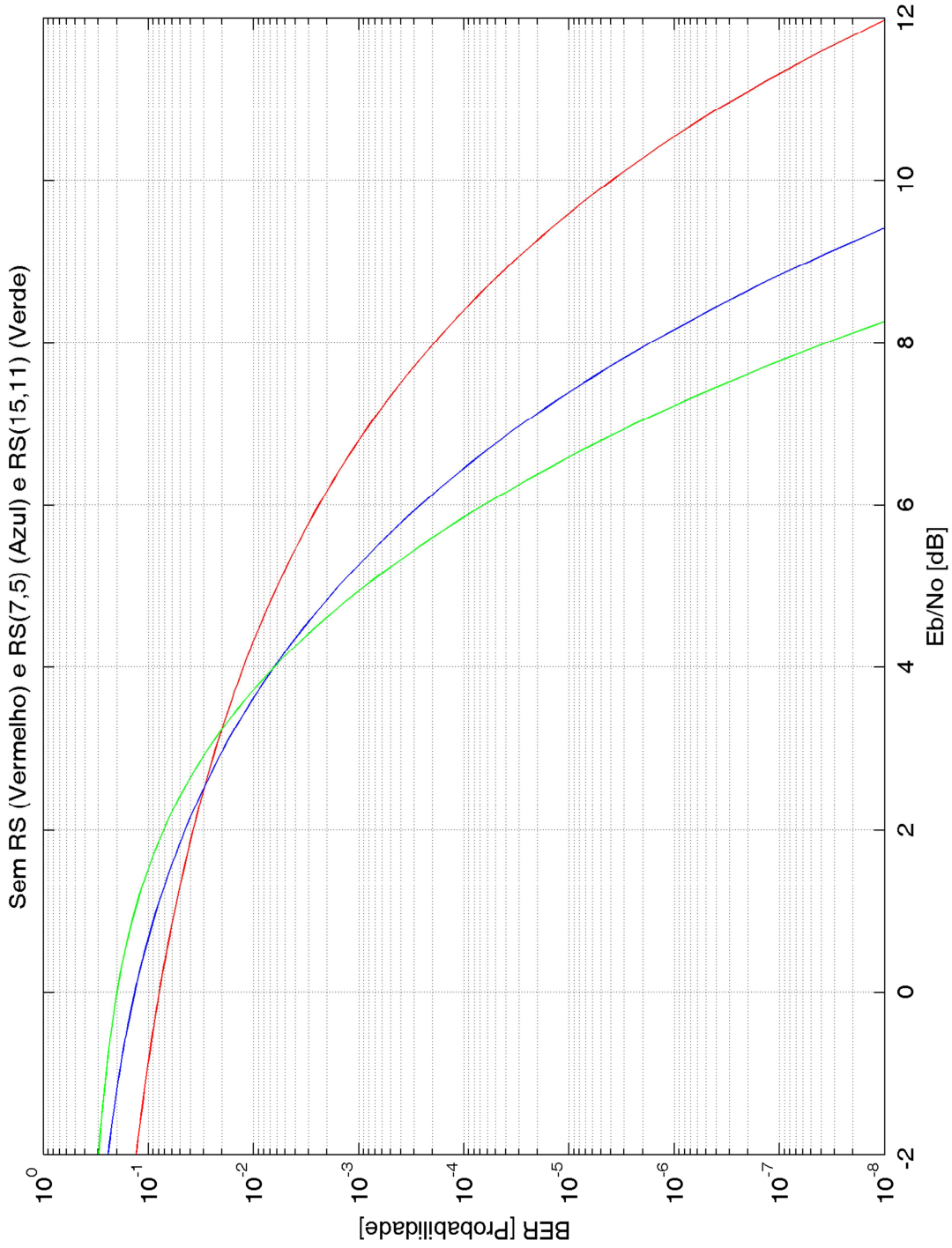

Figura 4.8 – BER versus E_b/N_0 para Sinal Não Codificado e Codificado com RS(7,5) e RS(15,11)

Então, para BER igual a 10^{-3}, do gráfico apresentado na Figura 4.8, são obtidos os valores de E_b/N_0 para o sinal não codificado, codificado utilizando o RS(7,5) e codificado utilizando o RS(15,11), respectivamente.

- E_b/N_0 (vermelho) = 6,8125 [dB]
- E_b/N_0 (azul) = 5,2641 [dB]
- E_b/N_0 (verde) = 4,9375 [dB]

Assim, através da Equação (2.7), pode-se obter a seguinte relação:

$$d_{RS} = d_U \sqrt[n]{\frac{(E_b/N_0)_U}{(E_b/N_0)_{RS}}} \tag{4.1}$$

Considerando, agora, os três ambientes mencionados na seção 2.2.3 (livre, indoor e indoor com muitos obstáculos), é possível obter uma relação entre o valor da distância utilizando a codificação e da distância sem a codificação.

Para o primeiro ambiente:

$$d_{RS}(azul) = d_U \sqrt[2]{\frac{6{,}8125[dB]}{5{,}2641[dB]}} = d_U \sqrt[2]{6{,}8125 - 5{,}2641} = d_U \sqrt[2]{1{,}5484[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[2]{1{,}4284} = 1{,}1951 d_U$$

$$d_{RS}(verde) = d_U \sqrt[2]{\frac{6{,}8125[dB]}{4{,}9375[dB]}} = d_U \sqrt[2]{6{,}8125 - 4{,}9375} = d_U \sqrt[2]{1{,}8750[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[2]{1{,}5399} = 1{,}2409 d_U$$

Para o segundo ambiente:

$$d_{RS}(azul) = d_U \sqrt[3]{\frac{6{,}8125[dB]}{5{,}2641[dB]}} = d_U \sqrt[3]{6{,}8125 - 5{,}2641} = d_U \sqrt[3]{1{,}5484[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[3]{1{,}4284} = 1{,}1262 d_U$$

$$d_{RS}(verde) = d_U \sqrt[3]{\frac{6{,}8125[dB]}{4{,}9375[dB]}} = d_U \sqrt[3]{6{,}8125 - 4{,}9375} = d_U \sqrt[3]{1{,}8750[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[3]{1{,}5399} = 1{,}1548 d_U$$

Para o terceiro ambiente:

$$d_{RS}(azul) = d_U \sqrt[4]{\frac{6{,}8125[dB]}{5{,}2641[dB]}} = d_U \sqrt[4]{6{,}8125 - 5{,}2641} = d_U \sqrt[4]{1{,}5484[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[4]{1{,}4284} = 1{,}0932 d_U$$

$$d_{RS}(verde) = d_U \sqrt[4]{\frac{6{,}8125[dB]}{4{,}9375[dB]}} = d_U \sqrt[4]{6{,}8125 - 4{,}9375} = d_U \sqrt[4]{1{,}8750[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[4]{1{,}5399} = 1{,}1140 d_U$$

Para BER igual a 10^{-7}, através da Figura 4.8, são obtidos os valores de E_b/N_0 para o sinal não codificado, codificado utilizando o RS(7,5) e codificado utilizando o RS(15,11), respectivamente.

- E_b/N_0 (vermelho) = 11,3246 [dB]
- E_b/N_0 (azul) = 8,8327 [dB]
- E_b/N_0 (verde) = 7,7802 [dB]

Considerando os mesmos três ambientes mencionados (livre, indoor e indoor com muitos obstáculos) e a relação apresentada na Equação (4.1), obtêm-se as seguintes relações:

Para o primeiro ambiente:

$$d_{RS}(azul) = d_U \sqrt[2]{\frac{11{,}3246[dB]}{8{,}8327[dB]}} = d_U \sqrt[2]{11{,}3246 - 8{,}8327} = d_U \sqrt[2]{2{,}4919[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[2]{1{,}7750} = 1{,}3323 d_U$$

$$d_{RS}(verde) = d_U \sqrt[2]{\frac{11{,}3246[dB]}{7{,}7802[dB]}} = d_U \sqrt[2]{11{,}3246 - 7{,}7802} = d_U \sqrt[2]{3{,}5444[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[2]{2{,}2617} = 1{,}5039 d_U$$

Para o segundo ambiente:

$$d_{RS}(azul) = d_U \sqrt[3]{\frac{11{,}3246[dB]}{8{,}8327[dB]}} = d_U \sqrt[3]{11{,}3246 - 8{,}8327} = d_U \sqrt[3]{2{,}4919[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[3]{1{,}7750} = 1{,}2108 d_U$$

$$d_{RS}(verde) = d_U \sqrt[3]{\frac{11{,}3246[dB]}{7{,}7802[dB]}} = d_U \sqrt[3]{11{,}3246 - 7{,}7802} = d_U \sqrt[3]{3{,}5444[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[3]{2{,}2617} = 1{,}3126 d_U$$

Para o terceiro ambiente:

$$d_{RS}(azul) = d_U \sqrt[4]{\frac{11{,}3246[dB]}{8{,}8327[dB]}} = d_U \sqrt[4]{11{,}3246 - 8{,}8327} = d_U \sqrt[4]{2{,}4919[dB]}$$

$$d_{RS}(azul) = d_U \sqrt[4]{1{,}7750} = 1{,}1542 d_U$$

$$d_{RS}(verde) = d_U \sqrt[4]{\frac{11{,}3246[dB]}{7{,}7802[dB]}} = d_U \sqrt[4]{11{,}3246 - 7{,}7802} = d_U \sqrt[4]{3{,}5444[dB]}$$

$$d_{RS}(verde) = d_U \sqrt[4]{2{,}2617} = 1{,}2263 d_U$$

Utilizando as relações acima entre a distância com codificação e a distância sem codificação foi obtida a Tabela 4.3, sendo que as porcentagens obtidas são referentes à distância sem a codificação.

Tabela 4.3 – Ganho da Codificação para Vários Modelos de BER e Ambiente

BER	10^{-3}			10^{-7}		
Coeficiente de Perda	*n = 2*	*n = 3*	*n = 4*	*n = 2*	*n = 3*	*n = 4*
RS(7,5)	19,51 %	12,62 %	9,32 %	33,23 %	21,08 %	15,42 %
RS(15,11)	24,09 %	15,48 %	11,40 %	50,39 %	31,26 %	22,63 %

Observa-se na Tabela 4.3 um aumento de desempenho significativo utilizando a codificação Reed-Solomon, pois a distância entre transmissor e receptor é aumentada em quase 10 % no pior caso.

Para comparação entre as duas codificações, foram considerados os dois valores de BER apresentados e o ambiente indoor. Para o primeiro valor, 10^{-3}, observa-se que a diferença entre as duas codificações é de 2,86 % da distância não codificada. Para o segundo valor, 10^{-7}, a diferença entre as duas codificações é maior, sendo de 10,18 % da distância não codificada. Considerando os cenários apresentados acima, percebe-se que a distância entre transmissor e receptor é da ordem de grandeza 10^0 metros. Então, um ganho de 2,86 % utilizando RS(15,11) ao invés de utilizar RS(7,5) pode não representar um ganho significativo, uma vez que, observando a Tabela 4.2, nota-se que a quantidade de ciclos de clock para executar a codificação e decodificação RS(15,11) é aproximadamente cinco vezes a quantidade para executar a RS(7,5).

4.7 – Resultados

Considerando a implementação do codificador e decodificador utilizando FPGA da família Spartan-3E XC3S500E do fabricante Xilinx, tem-se um período, *T*, igual a *8,8ns*, conforme mencionado anteriormente. Então, utilizando a Tabela 4.2, determinou-se a *latência* do processo de decodificação dos dois códigos, RS(7,5) e RS(15,11). Esse valor está apresentado na Tabela 4.4. Para a construção da Tabela 4.4, foi considerado, ainda, a *Taxa de Código* apresentada na Tabela 4.1 e o ganho da codificação considerando um ambiente indoor apresentado na Tabela 4.3.

Tabela 4.4 – Comparação entre RS(7,5) e RS(15,11)

	Latência	**Taxa de Código**	**Ganho @ BER = 10^{-3}**	**Ganho @ BER = 10^{-7}**
RS(7,5)	*21µs*	*0,714*	*12,62 %*	*21,08 %*
RS(15,11)	*118,3µs*	*0,733*	*15,48 %*	*31,26 %*

As aplicações utilizando rede de sensores sem fio baseiam-se na norma *IEEE 802.15.4*. Conforme visto em [27], o limite inferior do erro acontece no BER de 10^{-3} para *802.15.4* devido ao efeito Doppler.

Para aplicações com valores de BER elevados a codificação mais simples, RS(7,5), pode ser utilizada pois a diferença de distância não é significativa e a latência é menor. Entretanto, para níveis mais baixos, torna-se interessante o uso da codificação RS(15,11).

Capítulo5

Conclusões e Trabalhos Futuros

Durante o desenvolvimento deste trabalho foi possível conhecer os códigos corretores de erros utilizados na transmissão de informações. Foi possível testar e validar tais conhecimentos através da implementação em software dos codificadores e decodificadores Reed-Solomon RS(7,5) e RS(15,11) que podem ser utilizados em redes de sensores sem fio. A implementação neste trabalho utiliza o micro controlador de código livre, o PicoBlaze, para reduzir a área total utilizada na FPGA que reduz o consumo de potência. Os circuitos foram simulados e implementados na FPGA da família Spartan-3E do fabricante Xilinx.

As ferramentas utilizadas pertencem ao fabricante Xilinx, porém, o código Reed-Solomon implementado neste trabalho pode ser utilizado em qualquer outro micro controlador de qualquer fabricante, pois o algoritmo foi escrito em linguagem Assembly.

O desenvolvimento da codificação Reed-Solomon apresentada neste trabalho apresentou um redução da taxa de erro possibilitando significativo aumento da distância entre transmissor e receptor. A redução da taxa de erro mostrada neste trabalho pode ser utilizada tanto para aplicações que não exigem altos níveis de BER (aplicações médicas, por exemplo) quanto para aplicações que exigem níveis mais altos (aplicações em monitoramento remoto, por exemplo). Esse aumento do alcance da comunicação permite reduzir o número de nós sensores necessário para cobrir uma região específica de interesse.

A maior dificuldade encontrada na realização deste trabalho foi a execução da matemática abstrata dos Campos de Galois. Algumas das operações sobre o campo $GF(2^m)$ exigiam várias instruções para a execução de uma única operação matemática. Como trabalho futuro sugere-se a implementação de um micro controlador que efetue operações matemáticas sobre os campos $GF(2^m)$ em instruções únicas, diminuindo, assim, a quantidade de instruções e a quantidade de ciclos de clock necessárias para a execução das mesmas. Além disso, como trabalho futuro, percebe-se também que seria possível obter redução de área criando um micro controlador soft-core personalizado à quantidade de bits por símbolo.

O desenvolvimento apresentado neste trabalho representa vantagens comparadas às de trabalhos conhecidos, conforme mostrado, e um ponto de partida para futuros desenvolvimentos nessa área pelo Grupo de Microeletrônica da Universidade Federal de Itajubá.

Referências Bibliográficas

[1] F. RÖMER, K. E MATTERN, "**THE DESIGN SPACE OF WIRELESS SENSOR NETWORKS**," IEEE WIRELESS COMMUNICATIONS, VOL. 11, NO. 6, PP. 54–61, DECEMBER 2004.

[2] C. E. P. E. E. G. L. MCSWEENEY, R. E SPAGNOL, "**IMPLEMENTATION OF SOURCE AND CHANNEL CODING FOR POWER REDUCTION IN MEDICAL APPLICATION WIRELESS SENSOR NETWORK**," IN SENSOR TECHNOLOGIES AND APPLICATIONS, 2009. SENSORCOMM '09. THIRD INTERNATIONAL CONFERENCE ON, JUNE 2009, PP. 271 –276.

[3] T. BARBOSA, "**IMPLEMENTAÇÃO EM FPGA DE UMA ARQUITETURA REED-SOLOMON PARA USO EM COMUNICAÇÕES ÓTICAS**," MASTER'S THESIS, UNIVERSIDADE FEDERAL DE ITAJUBÁ, 2010.

[4] D. J. LIN, SHU E COSTELLO, **ERROR CONTROL CODING** (2ND EDITION), 2ND ED. PRENTICE HALL, JUN. 2004.

[5] M. Y. CHEN, C. L. E HSIAO, "**ERROR-CORRECTING CODES FOR SEMICONDUCTOR MEMORY APPLICATIONS: A STATE-OF-THE-ART REVIEW**," IBM JOURNAL OF RESEARCH AND DEVELOPMENT, VOL. 28, NO. 2, PP. 124 –134, MARCH 1984.

[6] M. S. ALENCAR, **TELEFONIA CELULAR DIGITAL**, 2ND ED. EDITORA ÉRICA, 2007.

[7] M. C. F. DE CASTRO, F. C. C. E DE CASTRO, "**COMUNICAÇÃO DIGITAL**," MASTER'S THESIS, PUCRS, 2001.

[8] I. S. REED AND G. SOLOMON, "**POLYNOMIAL CODES OVER CERTAIN FINITE FIELDS**," JOURNAL OF THE SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS, VOL. 8, PP. 300–304, 1960.

[9] M. J. SOUZA. **CÓDIGOS CORRETORES DE ERROS: UMA INTRODUÇÃO**. [ONLINE]. AVAILABLE: HTTP://WWW.IME.UFG.BR/

[10] R. H. MORELOS ZARAGOZA, **THE ART OF ERROR CORRECTING CODING**. WILEY, SEP. 2006.

[11] B. SKLAR, **DIGITAL COMMUNICATIONS: FUNDAMENTALS AND APPLICATIONS**, 2ND ED. PRENTICE HALL, 2001.

[12] R. G. GALLAGER, **INFORMATION THEORY AND RELIABLE COMMUNICATION**. WILEY, JAN. 1968.

[13] W. SCHÜTZER. (2005) **APRENDENDO ÁLGEBRA COM O CUBO MÁGICO**. V SEMANA DA MATEMÁTICA DA UFU. [ONLINE]. AVAILABLE: HTTP://WWW.DM.UFSCAR.BR/

[14] J. L. REBELATTO, "**CODIFICAÇÃO DE REDE BASEADA EM CÓDIGOS CORRETORES DE ERROS CLÁSSICOS**," MASTER'S THESIS, UFSC, 2010.

[15] T. A. RODOLFO, A. H. L. E SILVA, "**IMPLEMENTAÇÃO DE UMA ARQUITETURA REED-SOLOMON PARA USO EM REDES OTN 10.7 GBPS**," MASTER'S THESIS, PUCRS, 2007.

[16] R. E. BLAHUT, **THEORY AND PRACTICE OF ERROR CONTROL CODES**. ADDISON- WESLEY PUB. CO., 1983.

[17] V. HUFFMAN, CARY W. E PLESS, **FUNDAMENTALS OF ERROR-CORRECTING CODES**. CAMBRIDGE UNIVERSITY PRESS, AUG. 2003.

[18] A. DUR, "**AVOIDING DECODER MALFUNCTION IN THE PETERSON-GORENSTEIN-ZIERLER DECODER**," INFORMATION THEORY, IEEE TRANSACTIONS ON, VOL. 39, NO. 2, PP. 640 –643, MAR 1993.

[19] L. CASILLO, "**PROJETO E IMPLEMENTAÇÃO EM FPGA DE UM PROCESSADOR COM CONJUNTO DE INSTRUÇÃO RECONFIGURÁVEL UTILIZANDO VHDL**," MASTER'S THESIS, UFRN, 2005.

[20] R. E. R. J. KUON, I E TESSIER, "**FPGA ARCHITECTURE: SURVEY AND CHALLENGES**," FOUNDATIONS AND TRENDS IN ELCTRONIC DESIGN AUTOMATION, VOL. 2, PP. 135–253, 2008.

[21] P. P. CHU, **FPGA PROTOTYPING BY VHDL EXAMPLES: XILINX SPARTAN-3 VERSION**.

[22] V. A. PEDRONI, **CIRCUIT DESIGN WITH VHDL**. THE MIT PRESS, AUG. 2004.

[23] **PICOBLAZE 8-BIT MICROCONTROLLER REFERENCE DESIGN FOR FPGAS AND CPLDS**, XILINX, 2005.

[24] L. FRANTZ, "**GERADOR DE TRÁFEGO PARA REDES-EM-CHIP BASEADO NO PICOBLAZE**," MASTER'S THESIS, UNIVERSIDADE DO VALE DO ITAJAÍ, 2008.

[25] K. CHAPMAN, **KCPSM3 8-BIT MICROCONTROLLER FOR SPARTAN-3, VIRTEX-II AND VIRTEX-IIPRO**, XILINX, 2003.

[26] Y. HU AND M. SAWAN, "**A FULLY INTEGRATED LOW-POWER BPSK DEMODULATOR FOR IMPLANTABLE MEDICAL DEVICES**," CIRCUITS AND SYSTEMS I: REGULAR PAPERS, IEEE TRANSACTIONS ON, VOL. 52, NO. 12, PP. 2552 – 2562, DEC. 2005.

[27] B. SHEN, "**APPLICATION OF ERROR CORRECTION CODES IN WIRELESS SENSOR NETWORKS**," MASTER'S THESIS, UNIVERSITY OF MAINE, 2007.

Apêndice A

- **Codificador Reed-Solomon RS(7,3) em Assembly**

```
CONSTANT UART_status_port, 00          ;UART status input
CONSTANT tx_half_full, 01            ; Transmitter    half full - bit0
CONSTANT tx_full, 02                 ;  FIFO          full - bit1
CONSTANT rx_half_full, 04            ; Receiver       half full - bit2
CONSTANT rx_full, 08                 ;  FIFO          full - bit3
CONSTANT rx_data_present, 10          ;           data present - bit4
;
CONSTANT UART_read_port, 01          ;UART Rx data input
;
CONSTANT UART_write_port, 01         ;UART Tx data output
;
CONSTANT MEM_status_port, 00                    ;Status output Mememory
                                  ;
CONSTANT alpha_to_0, 01                ;Memory 00 - alpha_to[0]
CONSTANT alpha_to_1, 02                ;Memory 01
CONSTANT alpha_to_2, 03                ;Memory 02
CONSTANT alpha_to_3, 04                ;Memory 03
CONSTANT alpha_to_4, 05                ;Memory 04
CONSTANT alpha_to_5, 06                ;Memory 05
CONSTANT alpha_to_6, 07                ;Memory 06
CONSTANT alpha_to_7, 08                ;Memory 07 - alpha_to[7]
CONSTANT index_of_0, 09                ;Memory 08 - index_of[0]
CONSTANT index_of_1, 0A                ;Memory 09
CONSTANT index_of_2, 0B                ;Memory 0A
CONSTANT index_of_3, 0C                ;Memory 0B
CONSTANT index_of_4, 0D                ;Memory 0C
CONSTANT index_of_5, 0E                ;Memory 0D
CONSTANT index_of_6, 0F                ;Memory 0E
CONSTANT index_of_7, 10                ;Memory 0F - index_of[7]
CONSTANT gen_poly_0, 11                ;Memory 10 - generator_polynomial[0]
CONSTANT gen_poly_1, 12                ;Memory 11
CONSTANT gen_poly_2, 13                ;Memory 12
CONSTANT gen_poly_3, 14                ;Memory 13
CONSTANT gen_poly_4, 15                ;Memory 14 - generator_polynomial[4]
CONSTANT parity00_0, 16                ;Memory 15 - parity[0]
CONSTANT parity00_1, 17                ;Memory 16 - parity[1]
CONSTANT parity00_2, 18                ;Memory 17 - parity[2]
CONSTANT parity00_3, 19                ;Memory 18 - parity[3]
CONSTANT message0_0, 1A                ;Memory 19 - message[0]
CONSTANT message0_1, 1B                ;Memory 1A - message[1]
CONSTANT message0_2, 1C                ;Memory 1B - message[2]
                                  ;
                                  ;
                                  ;
                                  ;
CONSTANT n_minus_k, 04                          ;constant n - k = 7 - 3 = 4
CONSTANT n_min_k_1, 03                          ;constant n - k - 1 = 7 - 3 - 1 = 3
CONSTANT n_rs00000, 07                          ;constant n = 7
```

```
CONSTANT t_rs00000, 02                    ;constant t = 2
CONSTANT k_minus_1, 02                    ;constant k - 1 = 3 - 1 = 2
CONSTANT n_min_k_2, 06                    ;constant n - k + 2 = 7 - 3 + 2 = 6
;
;Special Register usage
;
NAMEREG sF, UART_data          ;used to pass data to and from the UART
;
NAMEREG sE, syn_error          ;used to pass location of data in scratch pad memory
;
NAMEREG sD, count                         ;used to count
NAMEREG sC, comp                          ;used to load the value to compare in for
NAMEREG s1, u                             ; s1 = u
NAMEREG s2, q                             ; s2 = q
NAMEREG s3, i                             ; s3 = i
NAMEREG s4, j                             ; s4 = j
;UART character strings will be stored in scratch pad memory ending in carriage return.
;A string can be up to 16 characters with the start location defined by this constant.
;
CONSTANT string_start, 20
;
;
;
;
                    cold_start: LOAD s0, 00                    ;
                                LOAD syn_error, 00
;
; Encode RS
;
                                LOAD i, 80
                                OUTPUT i, (s0)
                                LOAD i, 81
                                OUTPUT i, (s0)
                                LOAD i, 00
                          for_1: COMPARE i, n_minus_k
                                JUMP NC, end_for_1
                                LOAD s5, parity00_0
                                ADD s5, i
                                OUTPUT s0, (s5)
                                OUTPUT s0, (s5)
                                ADD i, 01
                                JUMP for_1
                      end_for_1: LOAD i, k_minus_1
                          for_2: COMPARE i, FF
                                JUMP Z, end_for_2
                                LOAD s5, message0_0
                                ADD s5, i
                                INPUT s6, (s5)
                                INPUT s6, (s5)
                                LOAD s5, parity00_0
                                ADD s5, n_min_k_1
                                INPUT s7, (s5)
                                INPUT s7, (s5)
                                XOR s6, s7
                                LOAD s5, index_of_0
                                ADD s5, s6
                                INPUT s7, (s5)
                                INPUT s7, (s5)
                                COMPARE s7, FF
                                JUMP Z, else_if_1
```

```
                LOAD j, n_min_k_1
for_3:          COMPARE j, 00
                JUMP Z, end_for_3
                LOAD s5, gen_poly_0
                ADD s5, j
                INPUT s6, (s5)
                INPUT s6, (s5)
                COMPARE s6, FF
                JUMP Z, else_if_2
                LOAD sA, 00
                LOAD sB, 00
                ADD sB, s6
                ADD sB, s7
                ADDCY sA, 00
                CALL modulo_nn
                LOAD s5, alpha_to_0
                ADD s5, sB
                INPUT s8, (s5)
                INPUT s8, (s5)
                LOAD s5, parity00_0
                ADD s5, j
                SUB s5, 01
                INPUT s6, (s5)
                INPUT s6, (s5)
                XOR s6, s8
                ADD s5, 01
                OUTPUT s6, (s5)
                OUTPUT s6, (s5)
                JUMP end_if_2
else_if_2:      LOAD s5, parity00_0
                ADD s5, j
                SUB s5, 01
                INPUT s6, (s5)
                INPUT s6, (s5)
                ADD s5, 01
                OUTPUT s6, (s5)
                OUTPUT s6, (s5)
end_if_2:       SUB j, 01
                JUMP for_3
end_for_3:      LOAD s5, gen_poly_0
                INPUT s6, (s5)
                INPUT s6, (s5)
                LOAD sA, 00
                LOAD sB, 00
                ADD sB, s6
                ADD sB, s7
                ADDCY sA, 00
                CALL modulo_nn
                LOAD s5, alpha_to_0
                ADD s5, sB
                INPUT s8, (s5)
                INPUT s8, (s5)
                LOAD s5, parity00_0
                OUTPUT s8, (s5)
                OUTPUT s8, (s5)
                JUMP end_if_1
else_if_1:      LOAD j, n_min_k_1
for_4:          COMPARE j, 00
                JUMP Z, end_for_4
                LOAD s5, parity00_0
```

```
                ADD s5, j
                SUB s5, 01
                INPUT s6, (s5)
                INPUT s6, (s5)
                ADD s5, 01
                OUTPUT s6, (s5)
                OUTPUT s6, (s5)
                SUB j, 01
                JUMP for_4
end_for_4:      LOAD s5, parity00_0
                OUTPUT s0, (s5)
                OUTPUT s0, (s5)
end_if_1:       SUB i, 01
                JUMP for_2
end_for_2:      LOAD i, 82
                OUTPUT i, (s0)
                OUTPUT i, (s0)
                OUTPUT i, (s0)
                LOAD i, 00
teste:          COMPARE i, n_rs00000
                JUMP Z, end_for_2
                LOAD s5, parity00_0
                ADD s5, i
                INPUT s6, (s5)
                INPUT s6, (s5)
                ADD i, 01
                JUMP teste
                ;
;
                ;
                ;
                ;
                ; Rotina multiplica i * j
mult_i_j:       LOAD sA, 00
                LOAD sB, 00
                LOAD s9, j
mult_i_j_begin: COMPARE s9, 00
                JUMP Z, end_mult_i_j
                ADD sB, i
                ADDCY sA, 00
                SUB s9, 01
                JUMP mult_i_j_begin
end_mult_i_j:   RETURN
                ;
                ;
                ; Rotina Modulo nn
modulo_nn:      COMPARE sA, 00
                JUMP Z, menor_FF
                SUB sB, n_rs00000
                SUBCY sA, 00
                JUMP modulo_nn
menor_FF:       COMPARE sB, n_rs00000
comp_nn:        JUMP C, menor_nn
                SUB sB, n_rs00000
                JUMP menor_FF
menor_nn:       RETURN
```