Universidade Estadual Paulista "Júlio de Mesquita Filho"
Câmpus de Ilha Solteira

Victor Afonso dos Reis

ESTUDO E DESENVOLVIMENTO DE UM MODELO DA CODIFICAÇÃO 64B66B EM AMBIENTE SIMULINK APLICADO À SISTEMAS DE ALTA VELOCIDADE

Ilha Solteira

2018

Victor Afonso dos Reis

ESTUDO E DESENVOLVIMENTO DE UM MODELO DA CODIFICAÇÃO 64B66B EM AMBIENTE SIMULINK APLICADO À SISTEMAS DE ALTA VELOCIDADE

Relatório Científico desenvolvido para a Fundação para o Desenvolvimento da Unesp (FUNDUNESP), da Universidade Estadual Paulista "Júlio de Mesquita Filho"(UNESP).

Orientador: PROF. DR. AILTON AKIRA SHINODA

Falcudade de Engenharia de Ilha Solteira

Ilha Solteira

2018

RESUMO

REIS, V. A. **ESTUDO E DESENVOLVIMENTO DE UM MODELO DA CODIFICAÇÃO 64B66B EM AMBIENTE SIMULINK APLICADO À SISTEMAS DE ALTA VELOCIDADE**. 2018. 63p. Relatório Científico (Graduação em Engenharia Elétrica) - Falcudade de Engenharia de Ilha Solteira, Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP), Ilha Solteira, 2018.

Em sistemas de alta velocidade, onde o maior número de dados deve ser transmitido em um curto espaço de tempo, a codificação 64b66b pode ser implementada no canal de transmissão.

Para uma sequência digital gerada e transmitida em alta velocidade, pode ocorrer uma série de problemas na transmissão do dado. Estes problemas são caracterizados por ruídos devido a radiações, interferências eletromagnéticas, ionizações indesejáveis e uma dessincronização entre o transmissor e receptor dada por uma longa sequência de zeros (0's) ou um (1's) no canal de transmissão. Esta longa sequência interfere nos circuitos adicionais presentes no canal de realizarem a sincronização, sendo necessário realizar um balanço nos bits (1's) e nos bits (0's) transmitidos. Neste projeto é realizado um estudo da codificação 64b66b através de um sistema que implementa um modelo da codificação, para transmissão somente de dados, no software Matlab^{TM} dentro do ambiente Simulink. Adicionalmente, dentro da codificação implementou-se um *Cyclic Redundancy Checking (CRC)* de 8 bits para a detecção de erros nos dados transmitidos.

Com a implementação dos sistemas é possível realizar um balanceamento dos dados e a detecção de erros no canal de transmissão. Com o balanceamento dos dados possibilita a recuperação do *clock*, sincronizando o dispositivo transmissor e receptor, ao mesmo tempo possibilita um canal de transmissão mais confiável por conta da detecção dos erros.

Palavras-chave: *High Speed Serial Link*. Codificação 64b/66b. Transmissões.Matlab^{TM}. Simulink.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em qualquer sistema de comunicação deve-se garantir uma alta confiabilidade dos dados transmitidos, de forma que no lado do receptor sejam recebidos os mesmos dados enviado pelo transmissor. Esta preocupação ocorre devido aos problemas apresentados nas transmissões como por exemplo: a atenuação do sinal, dessincronização entre o transmissor e o receptor e ruídos apresentados no canal de transmissão (MACHADO, 2018). Para amenizar os efeitos da dessincronização e dos ruídos, desenvolveu-se codificações de linha a fim de aumentar a confiabilidade da transmissão. Dessa forma, torna-se possível a detecção de erros e a implantação de circuitos que sincronizem os dispositivos comunicantes.

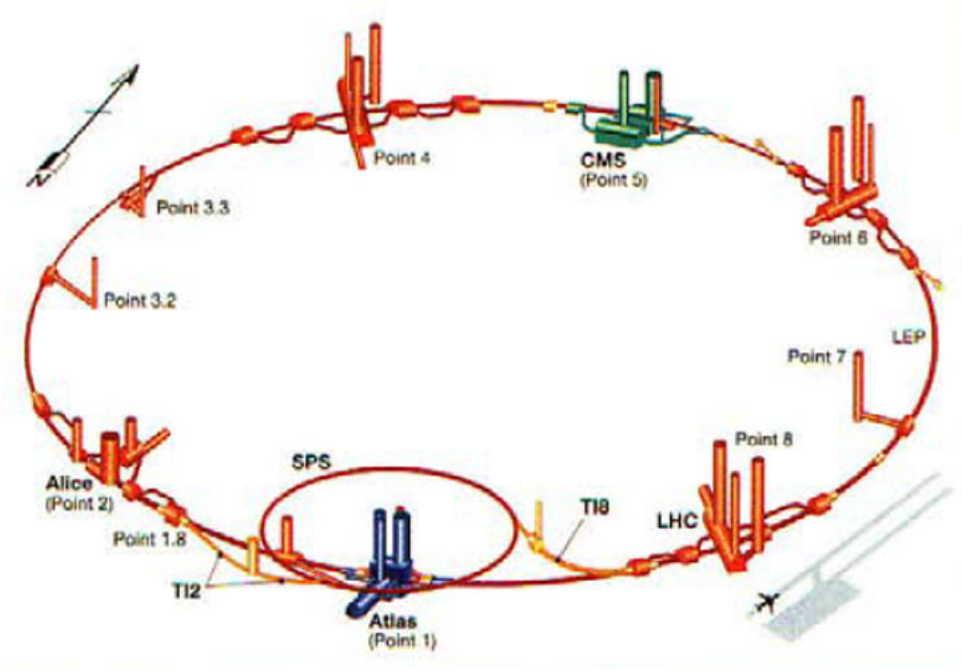
Uma codificação inteligente em um canal de transmissão, possibilita transmitir uma maior quantidade de dados por unidade de tempo (COMER, 2016). Estas codificações são extremamente úteis em sistemas para física de altas energias. Nestes sistemas, há a presença de um clock elevado e um interesse de uma alta confiabilidade no canal, implicando na necessidade de implementação de uma codificação no canal de transmissão. Esta codificação possibilita a introdução de mecanismos para identificar possíveis erros, com a inserção de circuitos corretores de erros, além de possibilitar que circuitos externos sincronizem os dispositivos comunicantes.

1.1 Motivação

Este trabalho é parte de uma colaboração com o laboratório "São Paulo Research and Analysis Center" (SPRACE) (SPRACE, 2018). O laboratório SPRACE possui vários ramos de pesquisa, sendo uma delas a instrumentação eletrônica para os sistemas do LHC. O LHC tem uma extensão de 27 km de circunferência, localizado na fronteira Franco-Suiça, tendo por objetivo descobrir a origem da massa das partículas elementares e outras dimensões do espaço (WIKIPÉDIA, 2018). O colisor é o maior equipamento já construído para pesquisa em física de altas energias do mundo, obtendo resultados expressivos como a descoberta do Bóson de Higgs. Este Bóson é uma partícula elementar prevista pelo modelo padrão de partículas, que ajuda a explicar a massa de outras partículas elementares (RANDALL, 2013).

No percurso do colisor há 4 detectores : ATLAS, CMS, Alice, e LHCb. O acelerador de partículas fornece velocidade e os detectores captam os produtos do impacto das partículas. Dessa forma, pode-se observar a existência de traços de partículas elementares que explicam teorias importantes sobre a física de altas energias (FERREIRA, 2009). Na Figura 1 é ilustrado um esquema do grande colisor de Hádrons que está localizado a 175 metros abaixo do solo.

Figura 1: Esquema do grande colisor de Hádrons (LHC).



Fonte: Elaborado pelo Autor

O grande colisor está em um processo de pesquisa para realizar uma grande atualização, a chamada "Grande Luminosidade". Desta forma, a instrumentação existente no colisor deve ser atualizada para suportar o novo volume de dados gerados. Os sistemas desenvolvidos devem serem capazes de transmitir e processar um grande volume de dados em uma faixa muito curta de tempo (BRÜNING, 2018).

O trabalho desenvolvido pelo laboratório SPRACE está diretamente ligado ao detector “Solenóide de Muon Compacto” (CMS) do LHC. Os detectores do LHC tem estruturas diferentes e cada um obtém dados de partículas específicas. A junção de todos os dados de todos os detectores forma uma imagem completa do experimento, ajudando a realizar novas descobertas.

O detector CMS tem 6 metros de diâmetro e 13 metros de comprimento captando ao longo do diâmetro as posições das partículas. Dessa forma, é possível traçar o caminho das partículas por produzirem dados de posições ao longo do diâmetro do detector. As partículas carregadas seguirão caminho em espiral no campo magnético de 4 Tesla (T) do detector possibilitando calcular os seus momentos, uma vez que momentos diferentes indicam partículas diferentes. Portanto, por meio desses dados é possível coletar evidências para comprovar teorias de novas partículas (TSESMELIS, 2018).

A colaboração entre o laboratório SPRACE com o CMS objetiva-se colaborar no desenvolvimento de sistemas na área da instrumentação eletrônica, o qual serão implementados dentro de um sistema completo de detecção. O novo sistema que está em

desenvolvimento, principalmente objetiva-se eliminar as restrições de latência que o atual possui (COLLABORATION, 2018).

1.2 TRABALHO DESENVOLVIDO

No ambiente do detector há uma alta taxa de radiação eletromagnética, por conta da alta velocidade dos átomos presentes no tubo. Dessa forma, em transmissões de alta velocidade de uma placa para outra podem acarretar a presença de ruídos no canal de transmissão. Os ruídos presentes no canal de transmissão danificam os dados originais, acarretando no armazenamento de dados incoerentes para um posterior estudo. Portanto, uma codificação presente no canal de transmissão possibilita a detecção de erros e a solução de problemas envolvidos na transmissão em altas velocidades, como por exemplo a dessincronização entre o transmissor e o receptor.

Este trabalho desenvolve o estudo da codificação 64b66b e suas características por meio do Simulink($MATLAB^{TM}$). Dessa forma, o sistema desenvolvido pode ser usado como codificação do canal de transmissão entre duas placas FPGA garantindo uma maior confiabilidade dos dados transmitidos.

O trabalho confeccionado foi dividido em 7 partes. No Capítulo 2 é apresentado uma noção geral sobre campos finitos, teoria essencial para o entendimento dos sistemas presentes na codificação. No Capítulo 3 é apresentado a teoria dos *Linear FeedBack Shift Registes (LFSR)*, componente essencial para os *Cyclic Redundancy Check (CRC)* e os *scramblers*. No Capítulo 4 é apresentado a teoria dos *scramblers*, sistemas essenciais para garantir o balanço entre os bits 1's e 0's na transmissão. No Capítulo 5 é apresentado a teoria dos CRC's, sendo os responsáveis pela detecção de erro no dado transmitido. A codificação 64b66b é descrita no Capítulo 6, bem como todos os componentes para realizar a transmissão do dado. No Capítulo 7 é apresentado a descrição do sistema desenvolvido com a codificação 64b66b. No Capítulo 8 é apresentado algumas simulações e conclusões sobre o funcionamento do sistema desenvolvido.

2 CAMPOS FINITOS (FINITE FIELDS)

Uma noção básica de um campo finito pode ser explicitada como um conjunto que possui um número finito de elementos. Neste, define-se 2 operações, normalmente adição e multiplicação, possuindo as seguintes propriedades para qualquer campo F:

- Para qualquer elemento a,b dentro de um conjunto F, as operações de adição e multiplicação são operações binárias em F.
- Para qualquer elemento dentro do conjunto F as propriedades associativas e distributivas devem serem mantidas.
- Para qualquer a,b em F, as seguintes relações são mantidas: $a+b=b+a$ e $a.b=b.a$.
- No conjunto F o elemento identidade aditivo $(a+0=a)$ e o multiplicativo $(a.1=a)$ deve existir.
- Para qualquer a,b e c em F a igualdade $a.(b+c)=(a.b)+(a.c)$ deve ser mantida.

Por fim pode-se definir campo com sendo um conjunto de elementos os quais é possível realizar as operações de adição, multiplicação, subtração e divisão, no qual o resultado sempre é um elemento do próprio campo. A adição e a multiplicação satisfazem as propriedades comutativa, associativa e distributiva (SILVA, 2011).

Um exemplo de campo finito é o campo de Galois, descrito na próxima seção.

2.1 Propriedades do Campo de Galois

Os campos Galois $GF(n)$ referem-se à um conjunto de n elementos, fechado com relação às operações de adição e multiplicação. Pode-se enunciar que nos campos finitos todo elemento possui o seu inverso aditivo e multiplicativo, com exceção para o elemento nulo (0).

Um campo de Galois possui $n=p^q$ elementos, porém não existe um campo de Galois para um número qualquer de elementos. Os campos de Galois são formados por um número primo de elementos ou por sua extensão, onde a extensão é uma potência do número primo, onde p representa um número primo e q é um número inteiro positivo. Os elementos são definidos por n, em que $GF(n)=0,1,.....,n-1$ sendo n a sua ordem.

Um campo descrito por $GF(2)$ possui o seguinte formato: $GF(2)=0,1$. Desta forma, um campo de ordem 2 possui somente dois elementos que são $0,1$, sendo denominado de campo binário. Portanto, um campo de Galois de ordem q é um conjunto de q elementos com duas operações binárias módulo-p.

Para um campo da forma $GF(2)$, pela fórmula $n = p^q$, os coeficientes são definidos como: $p = 2$ e $q = 1$. As operações binárias de adição e multiplicação executadas dentro deste campo são em módulo - 2 .

É definido como elemento primitivo, ou gerador, o elemento de ordem $(n - 1)$. As potências consecutivas do elemento primitivo gera todos os outros elementos do campo (KERL, 2004).

2.2 Polinômios Primitivos

Uma condição para um polinômio $f(x)$ de ordem m ser primitivo é ele ser irredutível. A irredutibilidade ocorre quando um polinômio $f(x)$ não for divisível por nenhum outro polinômio de grau inferior a ele e maior que 0. Um polinômio de grau 2 é irredutível se, somente se, ele não for divisível por polinômios de grau 1.

Por exemplo o polinômio $X^2 + X + 1$ é irredutível, porque ele não é divisível por $X + 1$, nem por X. Um polinômio irredutível (que não pode ser fatorado) $f(X)$ de ordem m é primitivo se,somente se, o menor número inteiro positivo n, para o qual $f(X)$ é um divisor de $X^n + 1$, seja igual a: n = 2m – 1 (FARRELL; MOREIRA, 2006).

2.3 Construção dos Campos de Galois

Os campos de Galois são construídos de acordo com um polinômio primitivo escolhido. Qualquer polinômio de grau m, define o campo finito GF(2^m). Desta forma, $2^m = 24 = 16$ elementos no campo. Tomando o polinômio primitivo $p(x) = X^4 + X + 1$.

As m raízes de um polinômio $p(x)$ de grau m, fornecem os seus elementos. As raízes do polinômio serão representadas por α, então faz-se $p(\alpha) = 0$ para encontrar as respectivas raízes. Sendo assim pode-se escrever para o polinômio $p(x) = X4 + X + 1$:.

$$p(\alpha) = 0$$

$$\alpha^4 + \alpha + 1 = 0$$

$$\alpha^4 = -\alpha - 1$$

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

$$\alpha^5 = \alpha.\alpha^4 = \alpha(\alpha + 1)$$

$$\alpha^5 = \alpha + \alpha^2$$

Repetindo esse processo podemos encontrar todos os m elementos do $GF(2^4)$ (KERL, 2004).

2.4 Operações nos Campos de Galois

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

Essas operações podem ser realizadas por portas lógicas, tabelas, flip-flops e registradores de deslocamento (*Shift Registers*) (KERL, 2004).

2.4.1 Adição e Subtração

Para um campo $GF(m)$ a soma de dois elementos i e j é dado pelo resto da divisão $\frac{(i+j)}{m}$. Essa operação é chamada de adição módulo m.

A adição no módulo 2 é definida pelo campo $GF(2) = 0, 1$ e a sua operação é demonstrada abaixo:

$$0 \bigoplus 0 = 0$$
$$1 \bigoplus 1 = 0$$
$$0 \bigoplus 1 = 1$$
$$1 \bigoplus 0 = 1$$

Assim para um campo com mais elementos 2^m a soma de quaisquer elementos tem que resultar em um outro elemento pertencente ao mesmo campo, pois o campo de Galois é um conjunto fechado. Para adição de um elemento representado na notação vetorial, é feita como uma adição elemento por elemento módulo 2. Observa-se que esta operação possui os mesmos resultados que uma porta lógica XOR, executada bit a bit nos vetores a serem adicionados:

$$\alpha_i \bigoplus \alpha_j = (a_{i0} \bigoplus a_{j0}) + (a_{i1} \bigoplus a_{j1})X + (a_{i2} \bigoplus a_{j2})X^2 + ... + (a_{i,m-1} \bigoplus a_{j,m-1})X^{m-1}$$

A subtração de elementos no campo de Galois é definida como sendo a mesma porta XOR, uma vez que na subtração de módulo - 2 $-\alpha = \alpha$ então $\alpha^n - \alpha^j = \alpha^n + \alpha^j$ (MORENO, 2010).

2.4.2 Multiplicação e Divisão

A multiplicação entre dois elementos, i e j, de um campo de ordem primária, m é dado por $\frac{(ij)}{m}$. Para o campo binário $GF(2)$ tem-se:

$$0 \bigoplus 0 = 0$$
$$1 \bigoplus 1 = 1$$
$$0 \bigoplus 1 = 0$$
$$1 \bigoplus 0 = 0$$

A operação de multiplicação entre dois elementos pertencentes a $GF(2^8)$ pode ser comparada às portas lógicas AND. A divisão de um elemento por outro é realizada multiplicando o elemento pelo inverso do outro que o divide. Portanto, pode-se representar a operação descrita com a equação abaixo:

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

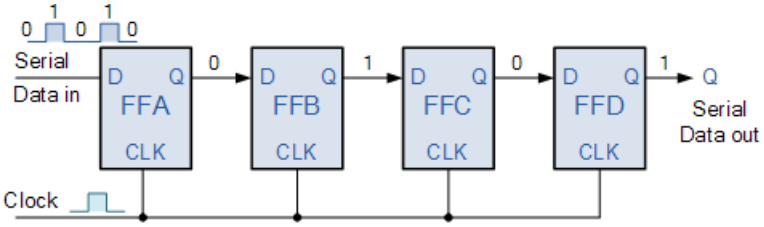
O inverso é obtido utilizando uma tabela que contém todos os elementos pertencentes ao campo e o seu respectivo inverso (MORENO, 2010).

3 LINEAR FEEDBACK SHIFT REGISTERS

Um *Linear Feedback Shift Register* (LFSR) são descritos com base na álgebra de campos finitos, possuindo 2 formas: por meio de polinômios e a outra por meio de matrizes. Para analisar o comportamento básico do sistema pode-se utilizar primeiramente uma abordagem bit a bit, o qual refere-se ao comportamento sistema explicitando os seus componentes e funcionalidade.

Um registrador de deslocamento (*shift register*) é um circuito digital que pode tanto armazenar dados, bem como movê-los. O armazenamento dos dados é feito por meio de *flip-flops*, como por exemplo o do tipo D. Quando um sinal de *clock* é enviado ao circuito, os *flip-flops* armazenam o valor de entrada a cada estágio. Determinada saída de uma célula está conectada na entrada da próxima, desta forma os bits são propagados de um lado para o outro até encontrar a saída do sistema na última célula (FLOYD, 2009). Um shift register é ilustrado na Figura 2.

Figura 2: Esquema de um registrador de deslocamento (*shift register*).

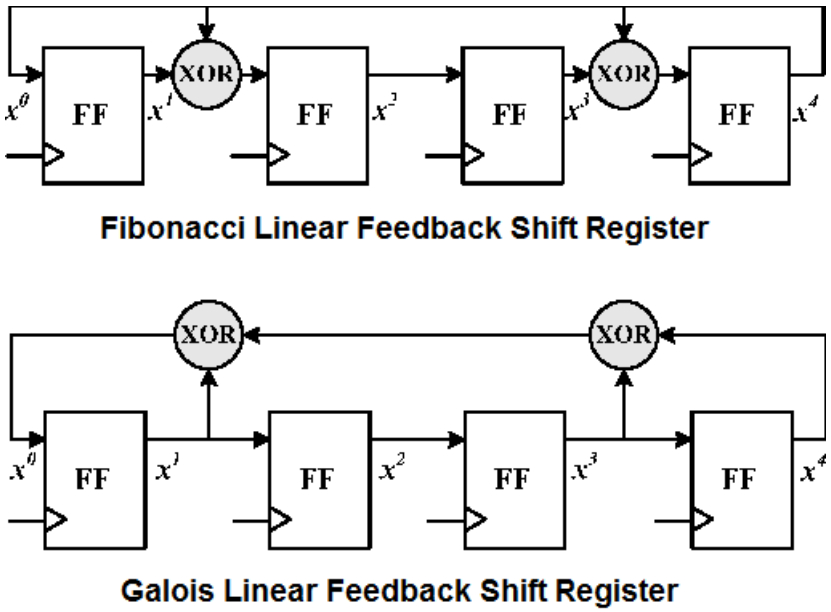


Fonte: Elaborado pelo Autor

Ao inserir uma realimentação no sistema *shift register*, altera-se a sua entrada e consequentemente os seus estados. A introdução da realimentação gera um sistema LFSR, porém este procedimento deve ser feito de acordo com algumas regras quando deseja-se obter uma determinada propriedade na saída.

Esta realimentação é implementada adicionando portas XOR, juntamente com *flip-flops* como ilustrado na Figura 3. Cada porta XOR inserida no sistema é denominado de *tap*, definindo um padrão nos dados de saída além de gerar um polinômio característico do LFSR.

Figura 3: Tipos de sistemas *Linear Feedback Shift Registers*.



Fonte: Elaborado pelo Autor

A cada porta XOR inserida no sistema, indica uma expressão matemática ou um polinômio. Cada LFSR possui um padrão na geração dos bits de saída, ou seja, os dados na saída não são puramente aleatórios possuindo um ciclo de repetição. Este padrão é determinado de acordo com o número de estágios e as ligações na realimentação, ou seja, cada esquema (polinômio) implementado possui um padrão no dado da saída. Portanto, a cada ciclo o dado começa a ser repetido e o comprimento deste ciclo é dado por $2^n - 1$, em que n é o número de *shift registers* usados no sistema. Porém, para se obter o máximo comprimento no circuito deve-se usar os polinômios primitivos, ou seja, inserir determinados *taps* em estágios específicos para produzirem o máximo comprimento.

A teoria de campos finitos é utilizada para definir quando um polinômio é ou não primitivo. No capítulo 2.2 é descrito como obter um polinômio primitivo, ou seja, um polinômio irredutível. Normalmente, estes polinômios são utilizados para construir circuitos geradores de números aleatórios, pelo fato de ocuparem um espaço menor quando comparado com os circuitos desenvolvidos com contadores.

Na tabela da Figura 4 é esboçado alguns polinômios primitivos para 4, 8, 16, e 32 estágios. Observa-se pela tabela que há vários polinômios, ou *taps*, para o mesmo número de *flip flops* porém só há um polinômio primitivo que produz o máximo ciclo no sistema. Estes polinômios são os mesmos para as configurações Fibonacci e Galois.

Figura 4: Tabela de polinômios com máximo comprimento para circuitos LFSR (*shift register*).

Size of LFSR	Possible feedback Polynomial	Maximum length feedback polynomial
4bit	X^4+X^2+1, X^4+X^3+1 etc.,	X^4+X^2+1
8bit	$X^8 + X^7 + 1$, $X^8 + X^5 + 1$, $X^8 + X^7 + X^6 + X^5 + 1$, $X^8 + X^6 + X^4 + X^3 + X^2 + X^1 + 1$,Etc.,	$X^8 + X^7 + X^6 + X^5 + 1$
16bit	$X^{16} + X^{15} + 1$, $X^{16} + X^{13} + X^{12} + X^9 + 1$, $X^{16} + X^{11} + X^{10} + X^7 + X^3 + X^1 + 1$, $X^{16} + X^{15} + X^{14} + X^{12} + X^7 + X^6 + + X^3 + X^2 + 1$, etc.,	$X^{16} + X^{14} + X^{13} + X^{11} + + 1$
32bit	$X^{32} + X^{31} + 1$, $X^{32} + X^{28} + X^{27} + X^9 + 1$, $X^{32} + X^{21} + X^{15} + X^{13} + X^{12} + X^{10} + X^8 + X^4 + 1$, $X^{32} + X^{31} + X^{27} + X^{24} + X^{19} + X^{18} + X^{17} + X^{14} + X^{13} + X^{11} + X^5 + X^4 + X^1$, etc.,	$X^{32} + X^{22} + X^2 + X^1 + 1$

Fonte: (ABINAYA; PRAKASAM, 2014)

Deve-se notar que há algumas regras para a escolha do polinômio primitivo que será implementado pelo circuito LFSR. Estas regras são descritas em (ABINAYA; PRAKASAM, 2014) e possuem as seguintes características:

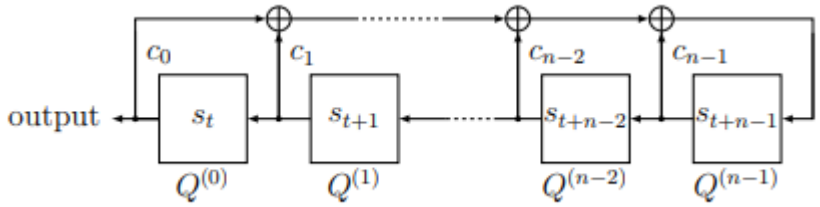
- O número 1 descrito nos polinômios da tabela da Figura 4 não correspondem à um *tap* e sim à entrada para o primeiro estágio do sistema.
- Os exponentes dos termos do polinômio correspondem aos estágios, e a sua contagem normalmente é feita da esquerda para a direita. Porém, nem todo sistema pode ser montado desta forma.
- Um LFSR só terá comprimento máximo se o número de *taps* for par. Somente 2 ou 4 *taps* pode ser suficiente para gerar longas sequências.
- O número do conjunto de *taps*, tomados ao todo, deve ser relativamente primo. Em outras palavras, o polinômio tomado deve ser primitivo.

- Depois que um polinômio primitivo for encontrado, outro segue automaticamente. Se os exponentes em um sistema LFSR com n estágios são da forma [n, A, B, C, 0], onde o 0 corresponde ao termo 1, então a sequência 'espelho' correspondente é [n, n-C, n-B, n-A, 0]. Assim, com uma sequência igual a [32, 7,3, 2 e 0] pode-se produzir a sua contraparte igual a [32, 30, 29, 25, 0]. Ambos dão uma sequência de comprimento máximo.

3.1 Fibonacci Linear Feedback Shift Registers

Um sistema LFSR Fibonacci é uma das formas de criar um LFSR a partir de um *shift register*. A realimentação de sistemas Fibonacci LFSR é caracterizada como externa, uma vez que no caminho da mesma encontra-se portas XOR's. Um esquema da realimentação em circuitos fibonacci é ilustrado na Figura 5

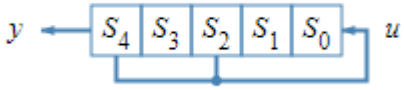
Figura 5: Esquema de um registrador de deslocamento (*shift register*).



Fonte: Elaborado pelo Autor

Desta forma, pode-se simplificar a representação de um fibonacci LFSR uma vez que é conhecido a existência de seus estágios e ligações XOR. Uma representação simplificada é ilustrada na Figura 6. Cada estágio é representado pelo símbolo $S_j[k]$ e a ligação entre o estágio e a realimentação, feito por meio de portas XOR's, é representado por b_j o qual pode ser 0 ou 1. O j especifica o estágio e o k o período que está sendo referido.

Figura 6: Esquema de um sistema (*Fibonacci Linear Feedback Shift Register*).



Fonte: Elaborado pelo Autor

A entrada do sistema representado na Figura 6 pode ser definido pela expressão matemática:

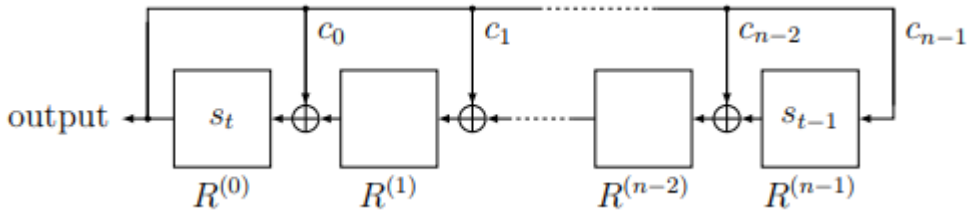
$$u[k] = \bigoplus_{j=0}^{N-1} b_j S_j[k]$$

O símbolo $\bigoplus$ significa uma operação XOR com todas as entradas no mesmo tempo. Desta forma, na Figura 6 as variáveis b_j são definidas como: $b_2 = b_4 = 1$ e $b_0 = b_3 = b_0 = b_1 = 0$. Portanto, a equação da entrada é definida como $u[k] = S_4[k] \bigoplus S_2[k] = u[k-5] \bigoplus u[k-3]$ e a saída é simplesmente a entrada atrasada o número de estágios no LFSR, ou seja, neste caso a saída é atrasada 5 períodos obtendo a equação $y[k] = u[k-5] = y[k-5] \bigoplus y[k-3]$.

3.2 Galois Linear Feedback Shift Registers

O outro tipo de sistema que pode ser construído realimentando um *shift register* é o Galois LFSR. A realimentação destes sistemas é caracterizada como interna, uma vez que as portas XOR's estão no caminho entre os *flips flops* ao invés de estarem na realimentação. Um esquema de circuitos Galois LFSR é ilustrado na Figura 7

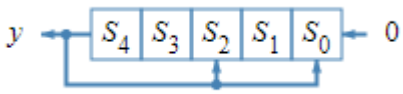
Figura 7: Esquema de um sistema (*Galois Linear Feedback Shift Register*).



Fonte: Elaborado pelo Autor

Desta forma, pode-se simplificar a representação de um Galois LFSR uma vez que é conhecido a existência de seus estágios e ligações XOR. Uma representação simplificada é ilustrada na Figura 8. Cada estágio é representado pelo símbolo $S_j[k]$ e a ligação entre o estágio e a realimentação, feito por meio de portas XOR's, é representado por a_j o qual pode ser 0 ou 1. O j especifica o estágio e o k em qual período está sendo referido.

Figura 8: Esquema simplificado de um sistema (*Galois Linear Feedback Shift Register*).



Fonte: Elaborado pelo Autor

Na representação de Galois de um LFSR, a entrada u é setada para 0 mas somente os estágios que possuem uma porta XOR ligada na saída podem alterar os bits transmitidos. Portanto, a saída $y[k]$ do LFSR representado na Figura 8 pode ser definida como:

$$S_j[k] = S_{j-1}[k-1] \bigoplus a_j y[k-1] \quad para \quad j \quad > \quad 0$$
$$S_0[k] = a_0 y[k-1]$$
$$y[k] = S_{N-1}[k] \quad N: \quad número \quad de \quad estágios$$

Desta forma, na Figura 8 as variáveis a_j são definidas como: $a_0 = a_2 = 1$ e $a_1 = b_3 = b_4 = 0$. Portanto, a equação dos estágios e da saída são definidas da forma:

$$S_0[k] = y[k-1]$$
$$S_1[k] = S_0[k-1] = y[k-2]$$
$$S_2[k] = S_1[k-1] \bigoplus y[k-1] = y[k-3] \bigoplus y[k-1]$$
$$S_3[k] = S_2[k-1] = y[k-4] \bigoplus y[k-2]$$
$$S_4[k] = S_3[k-1] = y[k-5] \bigoplus y[k-3]$$
$$y[k] = S_4[k] = y[k-5] \bigoplus y[k-3]$$

Na configuração de Galois os estágios que não possuem *taps* conectados, são deslocados uma posição para o próximo estágio. Os *taps*, por outro lado, realizam uma operação XOR com o bit de saída do estágio antes de serem armazenados na próximo *flip-flop*.O efeito disto é que quando o bit de saída é zero, todos os bits no registrador mudam para a direita inalterados, e o bit de entrada se torna zero. Quando o bit de saída é um, os bits nas posições da derivação são invertidos (se forem 0, eles se tornarão 1, e se forem 1, eles se tornarão 0). Desta forma, o registrador inteiro será deslocado para a direita e o bit de entrada torna-se 1.

Um Fibonacci LFSR, na presença de muitos taps, opera em velocidades mais baixas quando comparado com os sistemas Galois LFSR. Isto ocorre pelo fato das diversas portas XOR no caminho da realimentação ocasionarem um *delay* maior do que somente uma porta entre cada estágio descrito nos sistemas Galois LFSR. Porém, sistemas Fibonacci LFSR podem operar em velocidades altas como os Galois, se existir no máximo uma porta XOR no caminho da realimentação (DHINGRA, 2018).

4 SCRAMBLERS

Em sistemas de comunicação, um *scrambler* é um dispositivo que consegue embaralhar ou modificar uma mensagem no lado do emissor para tornar a mensagem ininteligível ou com determinadas propriedades para o receptor que não possui a capacidade de interpretar aquela mensagem. O embaralhamento é realizado pela adição ou reordenamento de componentes ao sinal original a fim de dificultar a extração do mesmo. Alguns *scramblers* modernos são, na verdade, dispositivos de criptografia, permanecendo o nome devido às semelhanças no uso, em oposição à operação interna (HASSAN, 2018).

Nas comunicações digitais, em muitos casos, um *scrambler* é usado para manipular um fluxo de dados antes de transmitir. As manipulações são invertidas por um *descrambler* no lado de recepção. Estas manipulações tem o objetivo de embaralhar o dado a ser transmitido, porém pode não haver criptografia neste processo. A intenção nesse caso não é tornar a mensagem ininteligível, mas dar aos dados transmitidos propriedades de engenharia úteis (HASSAN, 2018). Estas propriedades são úteis para a codificação 64b/66b.

Estas propriedades podem serem resumidas em dua principais (HASSAN, 2018):

- O embaralhamento em sistemas de comunicação digital facilita a atuação dos circuitos de recuperação de *clock*, controle de ganho e outros circuitos adaptativos do receptor pela eliminação de sequências consistindo apenas de $0's$ ou $1's$.
- Um circuito *scrambler* torna o espectro de potência do sinal mais disperso para atender ao requisitos de densidade espectral de potência. Este requisito trata da potência concentrada em uma faixa de frequência estreita, uma vez que esta característica pode interferir canais devido à modulação cruzada e à intermodulação causada pela não linearidade do receptor.

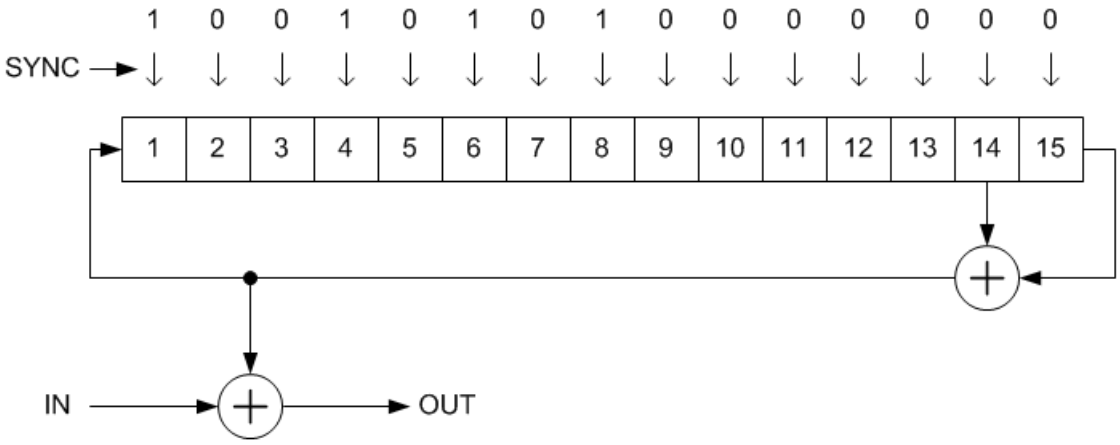
Os *scramblers* usualmente são definidos com base em LFSR por conta de suas boas características estatísticas, bem como pela facilidade de implementação em hardware. Os *scramblers* podem serem separados em dois tipos: *Scramblers* Aditivos (Synchronous) e Multiplicativos (Self-Synchronizing).

4.1 *Scramblers* Aditivos (Synchronous)

Scramblers Aditivos transformam o fluxo de dados de entrada, aplicando uma sequência binária pseudoaleatória (PRBS) por adição módulo-2. Em alguns casos, um PRBS pré-calculado é armazenado em uma memória ROM, sendo usado quando necessário.

Entretanto, frequentemente é gerado por um LFSR devidamente implementado no sistema. Um esquema de um *scrambler* aditivo é ilustrado na Figura 9.

Figura 9: Esquema de um *Scramblers* Aditivo.



Fonte: Elaborado pelo Autor

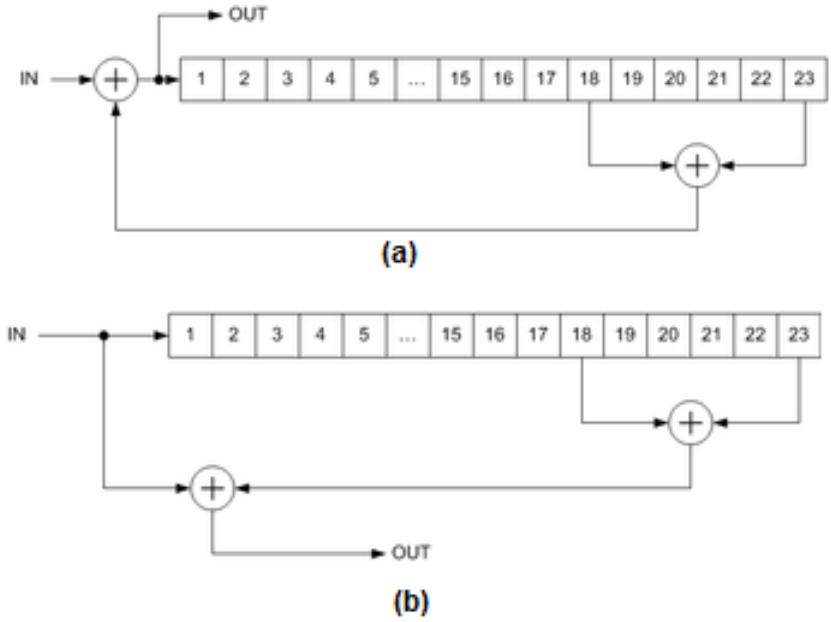
Uma palavra de sincronização é usada para assegurar uma operação síncrona do LFSR. Esta palavra é um padrão inserido no fluxo de dados em intervalos de tempos iguais, como por exemplo logo após o tempo para processar o dado introduzido no circuito. Um receptor procura algumas palavras de sincronização em dados adjacentes e, portanto, determina o local em que seu LFSR deve ser recarregado com um estado inicial predefinido (HASSAN, 2018).

O *descrambler* aditivo é apenas o mesmo dispositivo que o *scrambler* aditivo. O *scrambler* / *descrambler* aditivo são definidos pelo polinômio de seu LFSR e seu estado inicial.

4.2 *Scramblers* Multiplicativos (Self-Synchronizing)

Os *scramblers* multiplicativos são chamados assim por executarem uma multiplicação do sinal de entrada pela função de transferência do *scrambler* no espaço Z. Eles são sistemas lineares invariantes no tempo. Ao contrário dos *scramblers* aditivos, os *scramblers* multiplicativos não precisam da palavra sincronização, por isso eles também são chamados de auto-sincronizadores. Um exemplo da topologia dos *scramblers* multiplicativos está representado na Figura 10. Na letra (a) é representado um *scrambler* e na letra (b) um *descrambler* (HASSAN, 2018).

Figura 10: Esquema de um *Scramblers* Multiplicativo.



Fonte: Elaborado pelo Autor

Scrambler multiplicativo é definido similarmente por um polinômio, que também é uma função de transferência do *descrambler*. A saída codificada, $S(x)$, é gerada no transmissor dividindo os dados $M(x)$ por um polinômio gerador $G(x)$:

$$S(x) = \frac{M(x)}{G(x)}$$

A operação de divisão é realizada bit a bit e cada etapa da divisão resulta em um novo bit embaralhado. O receptor reordena o sinal embaralhado multiplicando pelo mesmo polinômio gerador:

$$M(x) = S(x) * G(x)$$

A implementação da divisão e multiplicação polinomial é feita usando *Linear feedback shift registers*, além de toda a teoria de campos finitos e suas operações em módulo 2.

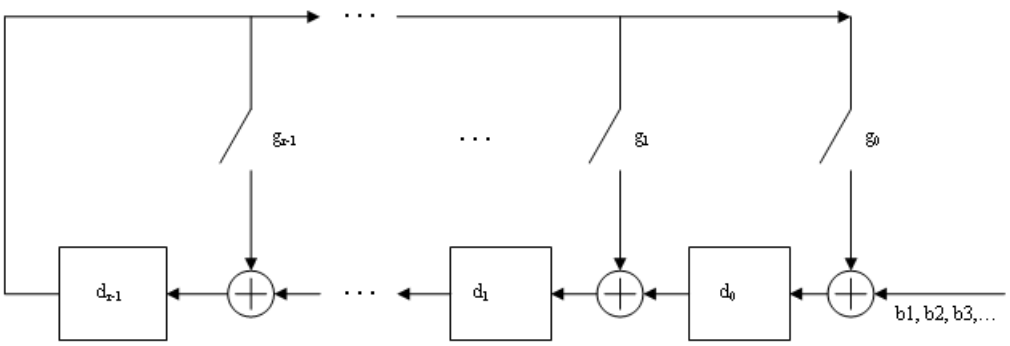
Os embaralhadores multiplicativos levam à multiplicação de erros durante a descodificação. Um erro de bit único na entrada do descodificador resultará em w erros na sua saída. Este aumento no número de erros depende do número de *taps* existente no

sistema. Caso existir duas *taps*, um único bit de erro inserido no sistema resultará em 3 bits errôneos na saída do *descrambler* (HASSAN, 2018).

5 CYCLIC REDUNDANCY CHECK (CRC)

Um CRC é um algorítimo somador para detectar erros durante a transmissão dos dados. Dado um bloco de com k bits, o CRC produz r bits que são adicionados ao bloco original e sendo transmitidos pelo meio de comunicação. O adendo r é uma constante e normalmente está entre 8 e 32 bits, para implementações reais. Portanto a mensagem final possui $n = k + r$ bits, gerados pela soma dos k bits do bloco mais o adendo gerado pelo CRC. Um exemplo de um CRC genérico é ilustrado na Figura 11.

Figura 11: Exemplo genérico de um sistema CRC.



Fonte: Elaborado pelo Autor

Cada mensagem final possui uma distância mínima de Hamming para auxiliar na detecção de erros. A distância de Hamming significa que a partir daquele número de bits errôneos no dado transmitido o sistema não consegue detectar. Para o mesmo circuito CRC pode-se obter diferentes distâncias de Hamming, dependendo do número k de bits do dado a ser transmitido (KOOPMAN; CHAKRAVARTY, 2004).

Um CRC é um exemplo de um código polinomial, bem como um exemplo de um código cíclico. A ideia em um código polinomial é representar cada bloco de códigos $w = w_{n-1}w_{n-2}w_{n-2}w_0$ como um polinômio de grau $(n-1)$. Portanto, tem-se:

$$w(x) = \sum_{i=0}^{n-1} w_i x^i$$

O propósito no sistema CRC é garantir que todo polinômio $w(x)$ seja múltiplo de um polinômio gerador, $g(x)$. Toda aritmética no CRC será feita por meio dos campo finitos com operações em módulo-2. As regras normais de adição polinomial, divisão e multiplicação são aplicáveis, exceto quando todos os coeficientes são 0 ou 1 (KOOPMAN; CHAKRAVARTY, 2004).

5.1 Codificação do Dado

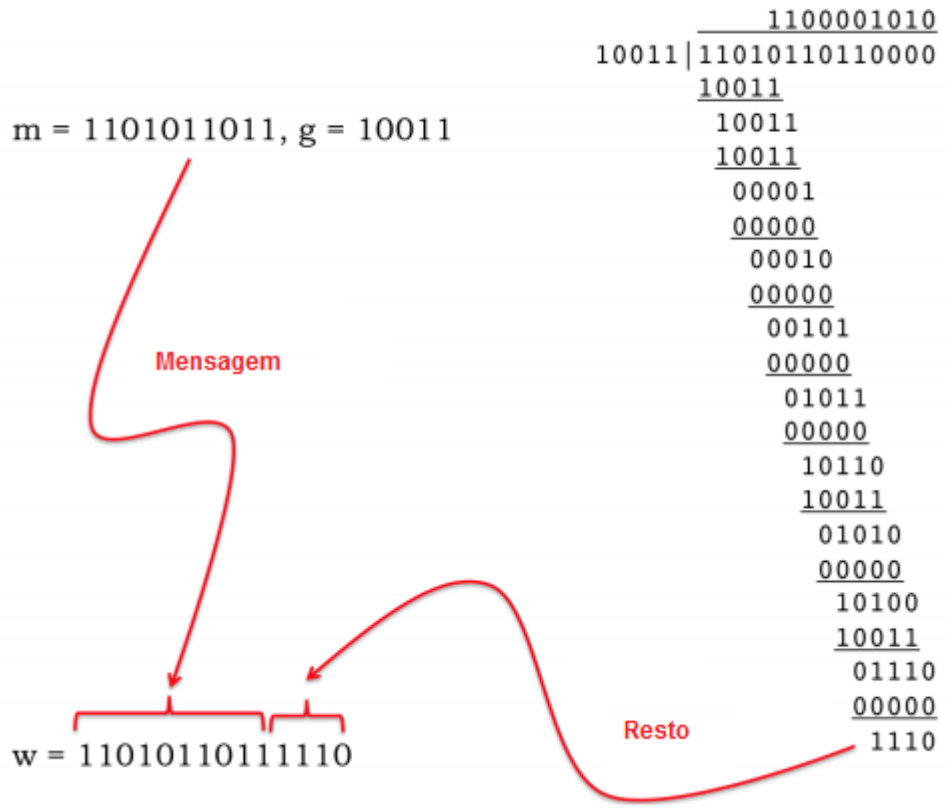
Para que $w(x)$ seja múltiplo de $g(x)$ deve-se obter $w(x)$ por meio do dado de entrada $m(x)$ e $g(x)$, de modo que $g(x)$ divida $w(x)$. Deve-se multiplicar $m(x)$ por x^{n-k}. Desta forma o dado de entrada é deslocado (nk) bits, então pode-se adicionar os bits produzidos pelo CRC. Após deslocar o dado de entrada em $n-k$ bits, nos espaços vagos são inseridos bits $0's$. Portanto, o dado final com os bits $0's$ inseridos nos espaços vagos é representado por: $x^{n-k}m(x)$ (KOOPMAN; CHAKRAVARTY, 2004).

Posteriormente, divide-se o dado deslocado $x^{n-k}m(x)$, por $g(x)$. Se o restante da divisão polinomial é 0, portanto os bits adicionados no espaço vagos estão corretos. Caso contrário, temos um resto, R. Ao subtrair este resto R do polinômio $x^{n-k}m(x)$, obtêm-se um novo polinomial que será um múltiplo de $g(x)$. Como a subtração é feita no campo finito $GF(2)$, pode-se substituir a subtração pela soma em $GF(2)$, obtendo-se:

$$w(x) = x^{n-k}m(x) + R\frac{x^{n-k}m(x)}{g(x)}$$

Um exemplo desta operação está ilustrada na Figura 12. A mensagem é definida como $m(x) = 1101011011$ e o polinômio gerador do CRC é definido como $g(x) = 10011$. Deve-se salientar que foi usado a divisão longa no campo finito $GF(2)$.

Figura 12: Procedimento da codificação para o sistema CRC.



Fonte: Elaborado pelo Autor

O $R(\frac{x^{n-k}m(x)}{g(x)})$ refere-se para o resto da divisão de $x^{n-k}m(x)$ por $g(x)$. Portanto, $w(x)$ é a mensagem final com $n = k+r$ bits que será transmitida pelo canal de comunicação (KOOPMAN; CHAKRAVARTY, 2004).

5.2 Decodificação do Dado

A etapa de decodificação é idêntica à etapa de codificação, além do decodificador possuir o mesmo polinômio gerador $g(x)$ do codificador. Nesta etapa, separa-se o dado $x^{n-k}m(x)$ do resto do CRC, adicionando $0's$ no lugar do resto. Posteriormente, verifica-se o resto calculado pela divisão de $x^{n-k}m(x)$ recebido por $g(x)$, comparando com o resto recebido no decodificador. Uma incompatibilidade entre os dois restos garante que ocorreu um erro (KOOPMAN; CHAKRAVARTY, 2004).

5.3 Seleção de Polinômios Geradores

Normalmente nas transmissões ocorre padrões de erro para os dados recebidos, desta forma pode-se formar algumas ideias na escolha do polinômio. Para desenvolver

propriedades adequadas para $g(x)$, deve-se obter o polinômio $r(x)$ que é a soma do dado enviado $w(x)$ e um erro polinomial $e(x)$ (KOOPMAN; CHAKRAVARTY, 2004).

Se $r(x) = w(x) + e(x)$ não é um múltiplo de $g(x)$, então o receptor certamente detectará o erro. O dado $w(x)$ é construído como um múltiplo de $g(x)$, se $e(x)$ não é um múltiplo de $g(x)$, o receptor detectará o erro. Por outro lado, se $r(x)$ e, portanto, $e(x)$, é um múltiplo de $g(x)$, então o erro não é detectado. Deve-se garantir que esta situação não ocorra para erros que ocorrem com frequência na transmissão (KOOPMAN; CHAKRAVARTY, 2004) . Portanto, pode-se seguir a seguinte instruções:

- Para padrões de erro únicos, $e(x) = x^i$ para alguns i. Isso significa que devemos assegurar que $g(x)$ tenha pelo menos dois termos.
- Para detectar todos erros duplos (ou pares) acontecidos na transmissão, pode-se representar este padrão de erro como $x^i + x^j = x^i(1 + x^{j-i})$, para alguns i e $j > i$. Se $g(x)$ não for múltiplo deste termo, então o CRC pode detectar todos os erros duplos.
- Para detectar todos os números ímpares de erros, deve-se ter um $g(x)$ que possua um número par de termos e que $(1 + x)$ seja um fator de $g(x)$. A divisão de qualquer polinômio no campo finito $GF(2)$ da forma $(1 + x)h(x)$ é avaliado como 0 quando setamos x para 1. Ao expandir $(1 + x)h(x)$ deve possuir um número par de termos. Portanto, se $(1 + x)$ for um fator de $g(x)$, o CRC será capaz de detectar todos os padrões de número ímpar de erros. No entanto, a inversa não é verdadeira: um CRC pode detectar um número ímpar de erros mesmo quando o seu $g(x)$ não é um múltiplo de $(1 + x)$. Porém, todos os CRC's usados na prática possuem $(1 + x)$ como um fator de $g(x)$ porque é a maneira mais simples de atingir esse objetivo.
- Para detectar rajada de erros, primeiramente define-se um o padrão do erro de rajada com comprimento b como uma sequência de bits $1\varepsilon_b - 2\varepsilon_{b-3}...\varepsilon_1 1$. O número de bits é b, o primeiro e o último bits são ambos 1, e os bits ε_i no meio podem ser 0 ou 1. O comprimento mínimo da rajada é 2, correspondente ao padrão "11".

  Para o CRC detectar todos esses padrões de erros, primeiro define-se o padrão de erros de rajada como $e(x) = x^s(1 * x^{b-1} + \sum_{i=1}^{b-2} \varepsilon_i x^i + 1)$. Este polinômio possui tamanho b e começa com s bits à esquerda no final do pacote. Se escolher-se $g(x)$ para ser um polinômio de grau b, e se $g(x)$ não tem x como um fator, então qualquer padrão de erro com comprimento $\leq b$ pode ser detectado. Isto deve-se ao fato de $g(x)$ não dividir um polinômio de grau menor que o seu. Além disso, existe um padrão de erro de comprimento $b + 1$ que corresponde quando o padrão de erro de rajada iguala-se aos coeficientes do próprio $g(x)$. Quando essa condição é atingida não pode-se detectar erros no dado recebido. Caso contrário, todos os outros padrões de erro de comprimento $b + 1$ serão detectados pelo CRC.

5.4 Implementação em Hardware

O fluxo de dados de entrada no CRC é geralmente bastante longo, sendo normalmente mais de 1 bit. Portanto não é possível executar uma divisão simples. A computação deve ser executada passo a passo, desta forma um *shift register* é utilizado.

Um *shift register* possui um comprimento fixo, sendo possível o deslocamento de bits no seu interior. Portanto, é possível remover o bit na borda direita ou esquerda e deslocar outro bit na posição liberada. O CRC usa um *shift register* que desloca dados da posição menos significativa (LSB) para a mais significativa (MSB). A posição do bit menos significativo é de livre escolha.

O processo de cálculo do CRC usando um *shift register* é o seguinte:

- Inicializar todos os *shift register* com o bit 0.
- Desloca-se o primeiro bit do dado de entrada dentro do sistema. Quando o bit MSB que sair do sistema for um '1', realiza-se uma operação XOR com o valor de todos os *shift registers* (contanto com o bit MSB que foi deslocado) com o polinômio do gerador. O resultado é inserido nos registradores e continua a operação.
- Se todos os bits de entrada forem inseridos, os *shift registers* do CRC contém o valor de CRC que será adicionado ao dado.

Para implementações em software o sistema pode ser implementado descrevendo em código o LFSR do CRC desenvolvido, usando a técnica bit a bit. Porém, na literatura há várias implementações em software mais que calculam o resultado do CRC de forma mais rápida do que realizar bit a bit.

6 A CODIFICAÇÃO 64B/66B

Este tipo de codificação garante transições suficientes para realizar a recuperação de *clock* no lado do receptor, além de preservar a probabilidade de detectar somente um ou múltiplos erros nos bits durante a transmissão. Os dois bits de sincronização adicionados no código permitem alinhar o fluxo dos blocos de bits no receptor.

6.1 Convenções de Notação

A codificação codifica um byte de dados ou um byte de caracteres de controle em um bloco. Os blocos que contém caracteres de controle também contém um campo com o tipo do bloco. Os octetos de dados são rotulados de D0 até D7. Caracteres de controle, tirando os O, S, e T, são rotulados de C0 até C7. Os caracteres de controle para definição de ordem são rotulados como O0 e O4 desde que seja válido o primeiro octeto do XGMII. Os caracteres de controle para o início são rotulados como S0 e S4 para a mesma condição do octeto do XGMII. Os caracteres de controle para o fim são rotulados como T0 e T7.

Duas transferências consecutivas de XMGII fornece oito caracteres que são codificados em um bloco da transmissão de 66-bits. O subíndice dos rótulos acima indica a posição dos caracteres na transferência dos 8 caracteres do XMGII.

O conteúdo do campo do tipo de bloco, os octetos de dados e os caracteres de controle são exibidos em valores hexadecimais. O bit menos significativo (LSB) do valor hexadecimal representa o primeiro a ser transmitido. Por exemplo, enviando o campo do tipo de bloco Ox1E na verdade em binário oque é enviado é “01111000”. Os bits de um bloco transmitido ou recebido são rotulados como TxB<65:0> e RxB<65:0>, respectivamente, em que TxB<0> e RxB<0> representam o primeiro bit transmitido. O valor do cabeçalho de sincronização é mostrado como valor binário. Os valores binários são mostrados com o primeiro bit transmitido (LSB) na esquerda.

6.2 Estrutura do Bloco

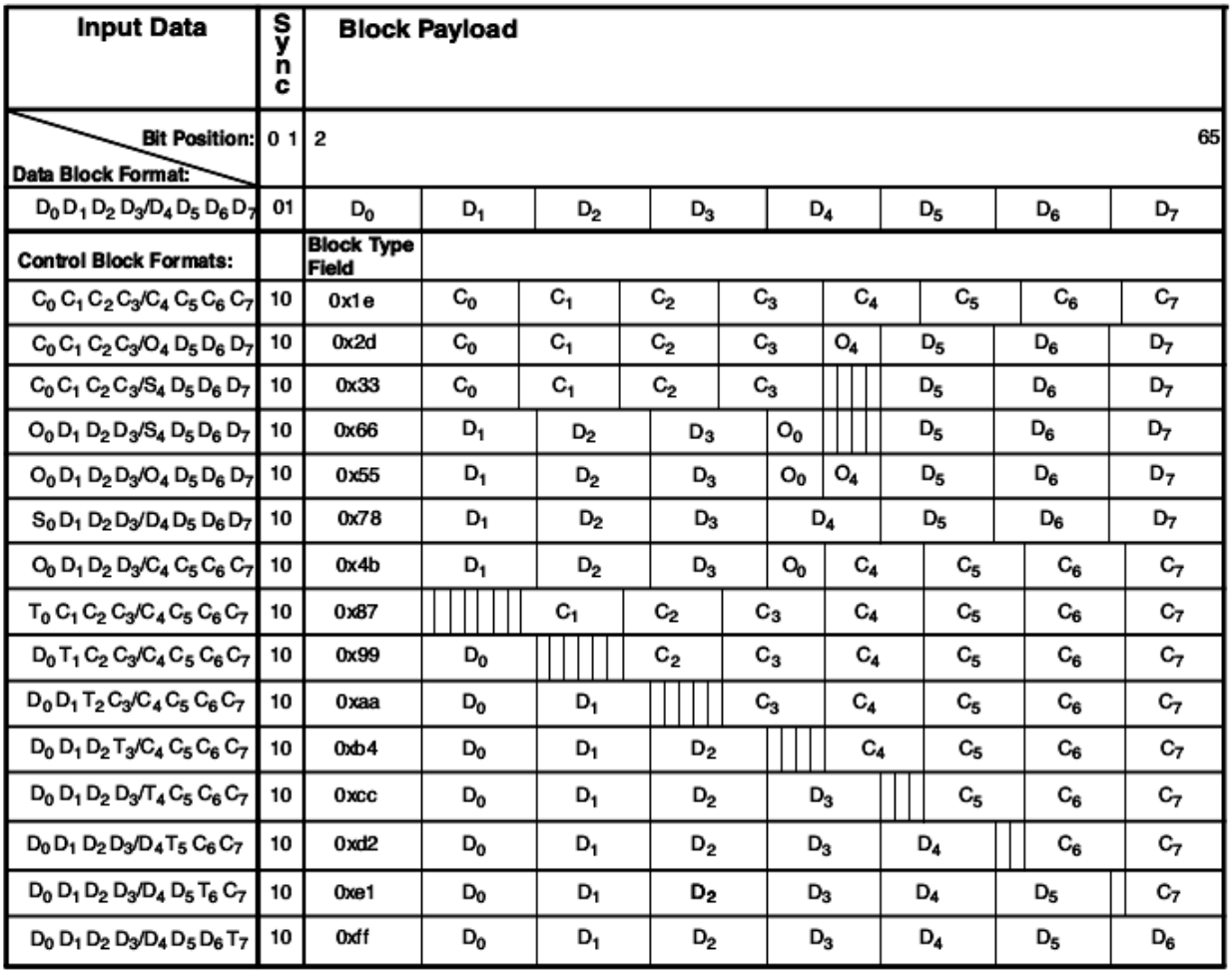
Os blocos consistem em 66 bits. Os primeiros dois bits de um bloco são os cabeçalhos de sincronização. Os blocos podem ser dado, controle ou os dois ao mesmo tempo. Os bits de sincronismo são definidos como “01” para blocos de dados e “10” para blocos de controle. Portanto, sempre há uma transição entre os dois primeiros bits do bloco para obter uma sincronização do bloco. O restante do bloco contém o dado útil. Somente o dado útil passa pelo *scrambler*, diferentemente dos bits de sincronismo uma vez que são somente adicionados ao dado de saída do *scrambler* sem passar pelo mesmo.

Blocos de dados contém 8 bytes diferentemente dos blocos de controle os quais

começam com um bloco de tipo de 8 bits, indicando o formato do restante do bloco. Para blocos de controle que possuem caracteres de começo e de término no meio dos 64 bits, sendo definidos pelo *control tipe field*. Outros caracteres de controles são codificados em 7 bits ou 4 bits representando o código de controle "O".

O formato dos blocos é como ilustrado na Figura 13. Na coluna *Input Data* mostra de forma abreviada o formato dos 8 bytes usados para criar o bloco de 66 bits. Os campos com retângulos finos representam um único bit, sendo preenchidos com o bit "0" e ignorados pelo receptor.

Figura 13: Formato dos blocos da codificação 64b/66b



Fonte: Elaborado pelo Autor

Os bits e as posições dos campos são mostrados com o bit menos significativo na esquerda. Por exemplo, o *block type field* "0x1E" é enviado como 01111000 representando os bits de 2 até 9 do bloco de 66 bits. O bit menos significativo de cada campo está numerado na menor posição do bloco.

Todos os outros valores não utilizados para o *control tipe field* são reservados. Estes

valores foram escolhidos para possuírem uma distância de Hamming de 4 bits. O único valor, em hexadecimal, não utilizado o qual mantém esta regra é 0x00.

6.3 Códigos de Controle

O mesmo conjunto de caracteres de controle é suportado pelo XMGII e pelo 10GBASE-R PCS. Os valores correspondentes para os caracteres de controle são denominados como códigos de controle. O XMGII codifica um caracter para um octeto (dado de 8bits). O 10GBASE-R PCS codifica implicitamente os caracteres de controle de *start* e de *terminate* pelo *control tipe field*. O 10BASE-R PCS codifica o código de controle de ordenamento (*ordered_set control codes*) usando uma combinação de *block type field* e um código de ordenamento de 4 bits (*4-bit O code*) para cada código *ordered_set*. O 10GBASE-R PCS codifica todos os outros caracteres de controle para um código de 7 bits (*7-bits C code*). Os caracteres de controle e o seu mapeamento para o 10GBASE-R PCS está descrito na tabela da Figura 14. Qualquer ou valor que não está presente na tabela da Figura 14 não deve ser transmitido e deverá ser tratado como erro.

Figura 14: Tabela dos códigos de controle da codificação 64b/66b.

Control Character	Notation	XGMII Control Code	10GBASE-R Control Code	10GBASE-R O Code	8B/10B Code[a]
idle	/I/	0x07	0x00		K28.0 or K28.3 or K28.5
start	/S/	0xfb	Encoded by block type field		K27.7
terminate	/T/	0xfd	Encoded by block type field		K29.7
error	/E/	0xfe	0x1e		K30.7
Sequence ordered_set	/Q/	0x9c	Encoded by block type field plus O code	0x0	K28.4
reserved0	/R/[b]	0x1c	0x2d		K28.0
reserved1		0x3c	0x33		K28.1
reserved2	/A/	0x7c	0x4b		K28.3
reserved3	/K/	0xbc	0x55		K28.5
reserved4		0xdc	0x66		K28.6
reserved5		0xf7	0x78		K23.7
Signal ordered_set[c]	/Fsig/	0x5c	Encoded by block type field plus O code	0xF	K28.2

Fonte: Elaborado pelo Autor

A coluna do 8b/10b code está representada de forma informativa. A utilização dos códigos de controle da codificação 8b/10b está descrito na secção 48 da especificação do IEEE 802.3ae (ISO/IEC/IEEE. . . , 2014). Os códigos /A/, /K/ e /R/ são usados nas interfaces XAUI para sinais de espera.

6.3.1 Idle (/I/)

Este comando de controle refere-se quando se deseja inserir uma espera no sistema, bem como para adaptar o sistema aos ciclos de clock. Os caracteres de controle (/I/) são transmitidos quando um sinal de espera é recebido do XGMII. Inserção ou retirada de caracteres /I/ devem ocorrer em grupos de 4 bits. Ao adicionar estes caracteres, deve-se seguir o controle de espera ou os ordered_sets. Os caracteres de controle /I/ não devem serem inseridos enquanto está recebendo algum dado. Quando

6.3.2 Start (/S/)

Este caracter de controle é comumente usado somente para os blocos onde um início pode acontecer. O *start control character* (/S/) indica o início de um pacote. Este delimitador é válido somente no primeiro bloco dos 64 bits do XGMII (TXD<0:7> e RXD <0:7>). A recepção de um *start control character* em qualquer outro octeto do TxD indica-se um erro na transmissão. Os valores do *block type field* implicitamente codifica um /S/ como quinto ou o primeiro bloco de 8 bits do dado de 64 bits.

6.3.3 Teminate (/T/)

O *terminate control character* (/T/) indica o final do pacote. Como os pacotes possuem comprimentos variados, o caracter de controle (/T/) pode ocorrer em qualquer octeto da interface XGMII. A localização do comando é codificada implicitamente pelo *block type field*. Um término de pacote é válido quando um bloco contendo um controle /T/ é seguido por um bloco que não contém um controle /T/.

6.3.4 Ordered_set (/O/)

O caracter controle *ordered_set* pode indicar dois tipos de comando: uma sequência que o caracter de controle possui ou um sinal de ordenamento. Necessitando denominar a sequência dos caracteres de controle para o *ordered_sets*, deverá ser usado o controle /Q/ descrito na tabela da Figura 14. O caracter /O/ somente é válido no primeiro octeto do XGMII, caso é recebido em qualquer outro bloco de 8 bits indica um erro. O próprio *block type field* codifica implicitamente um comando /O/ no primeiro ou no quinto bloco de 8 bits do dado de 64 bits, desta forma qualquer outra posição que o comando /O/ estiver significa um erro. O *block type field* já codifica implicitamente o controle /O/ no primeiro ou no quinto bloco de 8 bits. O 4-bit O codifica o caracter /O/ específico para o *ordered_set*.

A sequência dos *ordered_sets* pode ser deletada pelo PCS para se adaptar entre os ciclos de *clock*. Esta operação só pode ser realizada quando duas sequências consecutivas do comando forem recebidas e somente uma destas é deletada. Para a compensação de *clock*, exclusivamente comandos *idle* podem serem inseridos. Sinais *ordered_sets* não podem serem excluídos para compensação do *clock*.

6.3.5 Error (/E/)

O comando /E/ é enviado sempre que o mesmo ou um bloco inválido é recebido. O comando de erro permite as camadas físicas como o XGXS e o PCS propagar sinais de erros. Um esclarecimento maior sobre os sinais pode ser obtido na cláusula 49.2.13.2.3 da especificação do IEEE 802.3ae (ISO/IEC/IEEE. . . , 2014).

6.4 Ordered_sets

Os sinais de comando *ordered_sets* são usado para ampliar a capacidade de enviar sinais de controle e estado pelo link, como falha remota e estado da falha remota local. *Ordered_sets* são caracteres de controle seguidos de 3 caracteres de dados e sempre começam no primeiro bloco de 8 bits do XGMII. O *10 Gigabit Ethernet* usa um tipo de *ordered_set* descrito na cláusula 46.3.4 do padrão IEEE 802.3ae (ISO/IEC/IEEE. . . , 2014). A sequência de caracteres de controle *ordered_set* é denotado /Q/. Um código adicional de controle, o sinal *ordered_sets*, está reservado e começa com outro código de controle. O campo de 4 bits (O) codifica os códigos de controle. Veja mais na tabela da Figura 14.

6.5 Blocos válidos e inválidos

Um bloco é inválido quando:

- Os bits de sincronismo do bloco de 66bits forem "00"ou "11"
- O *block type field* conter um valor reservado descrito na tabela da Figura 14.
- Qualquer caracter de controle não possuir alguns dos valores descritos na tabela da Figura 14.
- Qualquer caracter de ordenamento (*ordered_sets*) /O/ que não esteja na tabela da Figura 14.
- O bloco de 64 bits não possuir algum dos formatos descritos na tabela da Figura 13.

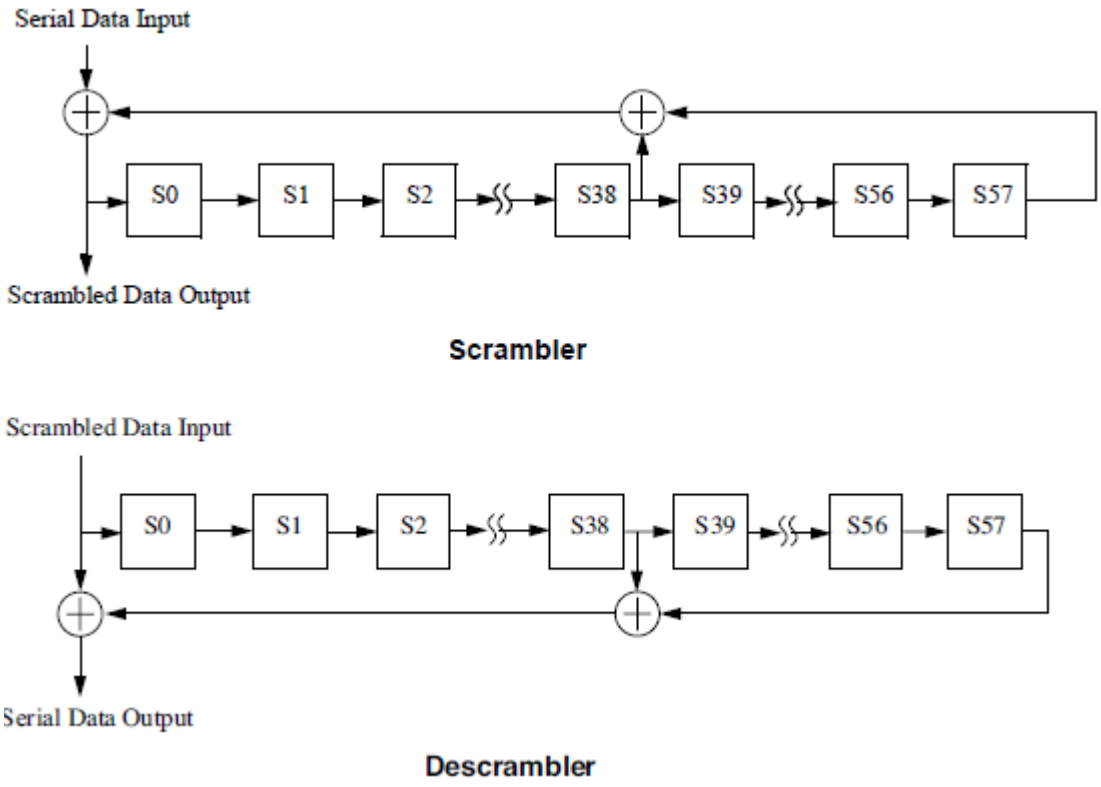
6.6 Scrambler

Na maioria dos sistemas de comunicações o propósito de um circuito scrambler é balancear o máximo possível as transições no dado a ser transmitido, evitando níveis lógicos repetidos por muito tempo. Desta forma, o circuito scrambler possibilita uma recuperação de *clock* do lado do receptor, além de fornecer um espectro de potência mais disperso diminuindo as interferências de rádio e o *crosstalk*, uma vez que a potência não está concentrada em uma única frequência.

O princípio básico de funcionamento de um circuito scrambler são os *linear feedback shift register* (LFSR). Um *shif register* de comprimento "n"consiste em n-flipflops interconectados, possuindo o estado binário desta célula de memória com índice (i) transferida para uma célula com índice (i +1) quando um sinal de *clock* é aplicado em todas estas células. Cada flip-flop é visto como um estágio do registro e a informação binária do último estágio é sempre acessível fisicamente.

Para a implementação da codificão 64b/66b foi utilizado o *scrambler* e o *descrambler* definidos nas cláusulas 49.2.6 e 49.2.10 do padrão IEEE 802.3ae (ISO/IEC/IEEE. . . , 2014).O sistema do *scrambler* e do *descrambler* é ilustrado na Figura 15.

Figura 15: Esquema do *scrambler* e do *descrambler* descrito no padrão IEEE 802.3ae.



Fonte: Elaborado pelo Autor

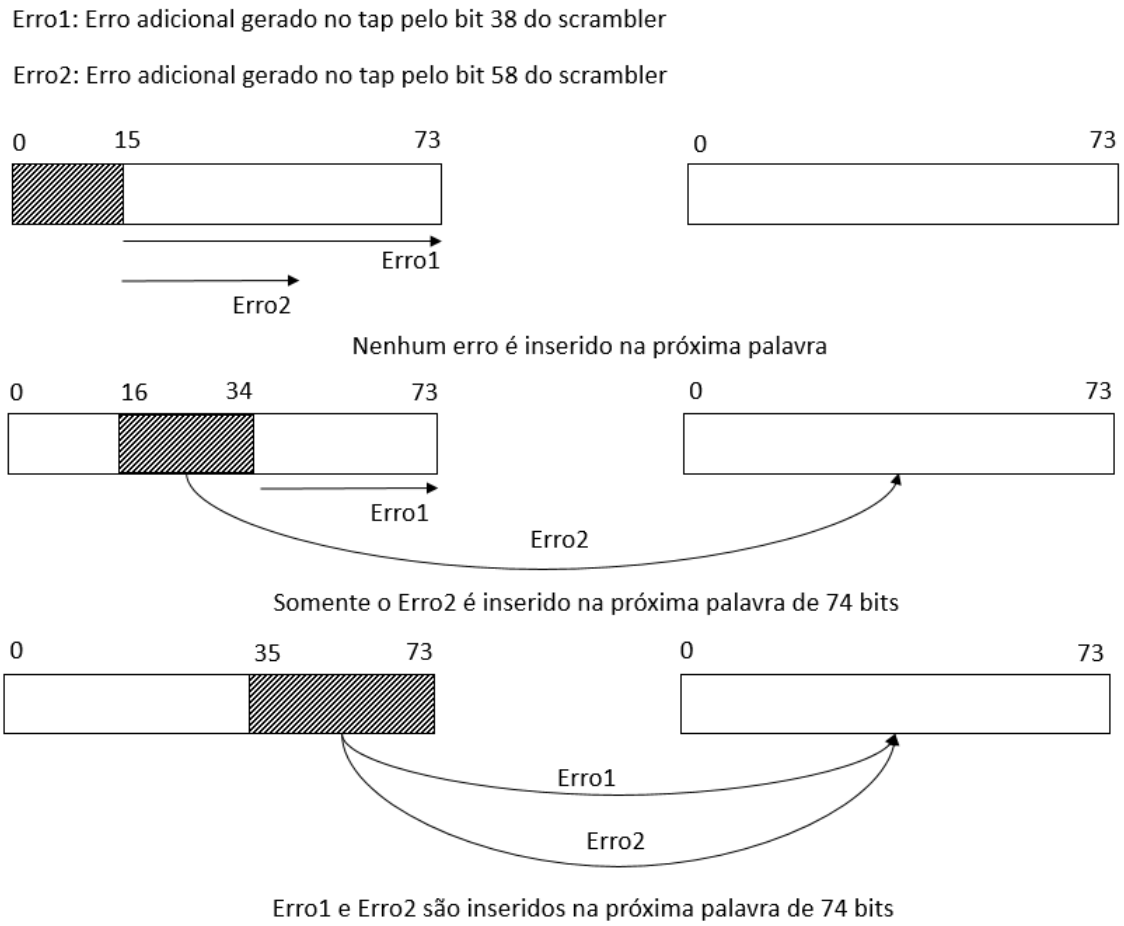
Os 64 bits são inseridos no sistema, já os 2 bits de sincronismo são anexados com a saída do *scrambler* sem passar pelo mesmo. Para o *descrambler* o procedimento é o mesmo que no *scrambler*. Nos dois sistemas é usado o seguinte polinômio:

$$G(x) = x^{58} + x^{39} + 1$$

Este polinômio é primitivo gerando uma sequência de $2^{58} - 1 = 288230376151711743$ números, sendo muito difícil de a sequência ser descoberta por alguém.

Em qualquer transmissão de dados é possível ocorrer ruídos, desta forma o *descrambler* da Figura 15 ao receber os dados errôneos, além de multiplicar erros como descrito no seção 4.2, pode carregar estes para um novo dado de entrada. Na Figura 16 é descrito as possibilidades de carregar o erro ocorrido no dado atual para um novo dado.

Figura 16: Possibilidades de carregamento de erros no Descrambler.



Fonte: Elaborado pelo Autor

Observa-se que desta forma um único bit errôneo pode gerar erros nos novos dados que possivelmente estão corretos.

6.7 CRC-8 bits

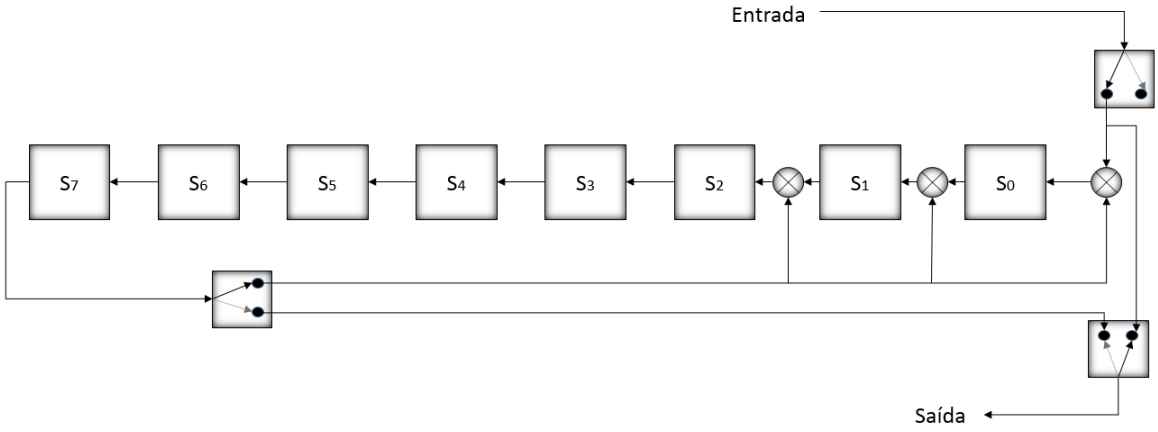
Como visto, o erro de um bit pode se propagar para um erro de três bits depois de passar pelo *descrambler* além de poder ser carregado para um novo dado de entrada. Por isso, é necessário detectá-los com um sistema adequado. A teoria para o entendimento do funcionamento do CRC implementado no sistema e a escolha do polinômio adequado pode ser vista na seção Capítulo 5. A tabela com os polinômios ideais para cada distância de hamming pode ser encontrada em (KOOPMAN, 2018b).

Portanto, implementou-se um CRC de 8 bits com o polinômio $0x83 = [10000011] = x^8 + x^2 + x + 1$ e com distância de Hamming igual a 4, como descrito na tabela presente em (KOOPMAN, 2018a). Fatorando este polinômio obtêm-se $(x+1)(x^7+x^6+x^5+x^4+x^3+x^2+1)$ que ao ser comparado com a teoria da seção 5.3 pode-se obter algumas características do

sistema. Portanto, pela teoria descrita observa-se que o polinômio pode detectar qualquer tipo de erro. Os erros de rajada podem serem detectados pois o polinômio não possui um fator x presente, a não ser na condição descrita na seção 5.3. Já os erros ímpares podem serem detectados pois o polinômio possui um fator $(x + 1)$ ao ser expandido. Os erros duplos ou pares não podem serem detectados uma vez que o polinômio é múltiplo do fator $x^i(1 + x^{j-i})$. Os erros simples também são detectáveis pois o polinômio possui mais que dois termos.

O CRC que implementa o polinômio descrito está ilustrado na Figura 17. Observa-se que primeiramente o dado é inserido, permanecendo a chave na posição mais escura. Posteriormente a chave move-se para a posição mais clara e o resto é obtido dos registradores.

Figura 17: Esquemático do CRC-8 Bits implementado na codificação 64b/66b.



Fonte: Elaborado pelo Autor

Desta maneira, o CRC pode detectar erros de até 3 bits em com comprimento até 119 bits, possuindo um polinômio primitivo e capaz de detectar erros de número ímpar de bits (KOOPMAN; CHAKRAVARTY, 2004).

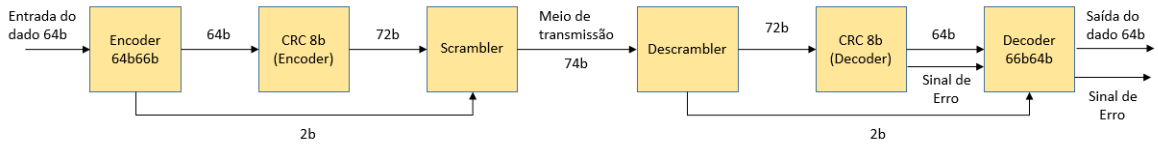
7 SISTEMA IMPLEMENTADO NO MATLAB™ EM AMBIENTE SIMULINK

O sistema foi implementado no Matlab™ usando os recursos *Embedded* Matlab™ *Function Block* (EMFB) do Simunlink. Nesta ferramenta pode-se descrever o *encoder* e o *decoder* da codificação 64b66b. Pela grande variedade de recursos que esta ferramenta possui é possível criar mecanismos para obter as características da codificação. Estas podem serem obtidas inserindo erros no dado transmitido do *encoder* para o *decoder* analisando o número de erros obtido pelo número de dados transmitidos. Estes erros inseridos devem ser aleatórios para a obtenção de uma característica que se aproxime da realidade.

A codificação 64b66b foi originalmente descrita para mapear os dados de 8 bits do protocolo XMGII, como descrito no capítulo Capítulo 6. Este protocolo é responsável pela comunicação entre duas sistemas de hardwares diferentes no padrão ethernet 10GBASE-R IEEE 802.3: *Media Access Control*(MAC) e o *Physical Layer*(PHY). Portanto, a codificação possui uma parte de códigos para controle porém o objetivo do trabalho não necessita desta funcionalidade. O propósito do sistema é verificar se a codificação possui propriedades interessantes para ser implementada em um chip *Field Programmable Gate Array* (FPGA). Desta forma, o único interesse é na transmissão de dados puros sendo desnecessário a descrição dos controles para o teste das propriedades da codificação.

A codificação não possui uma característica robusta para detectar erros. Isto deve-se pelo fato do padrão ethernet ,para o qual foi designada, possuir um CRC-32 que gera bits redundantes, sendo estes adicionados no pacote da transmissão. Portanto, para este sistema inseriu-se um CRC 8 bit descrito na seção seção 6.7. Um esquema do sistema implementado é representado na Figura 18.

Figura 18: Esquema do sistema implementado da codificação 64b66b.



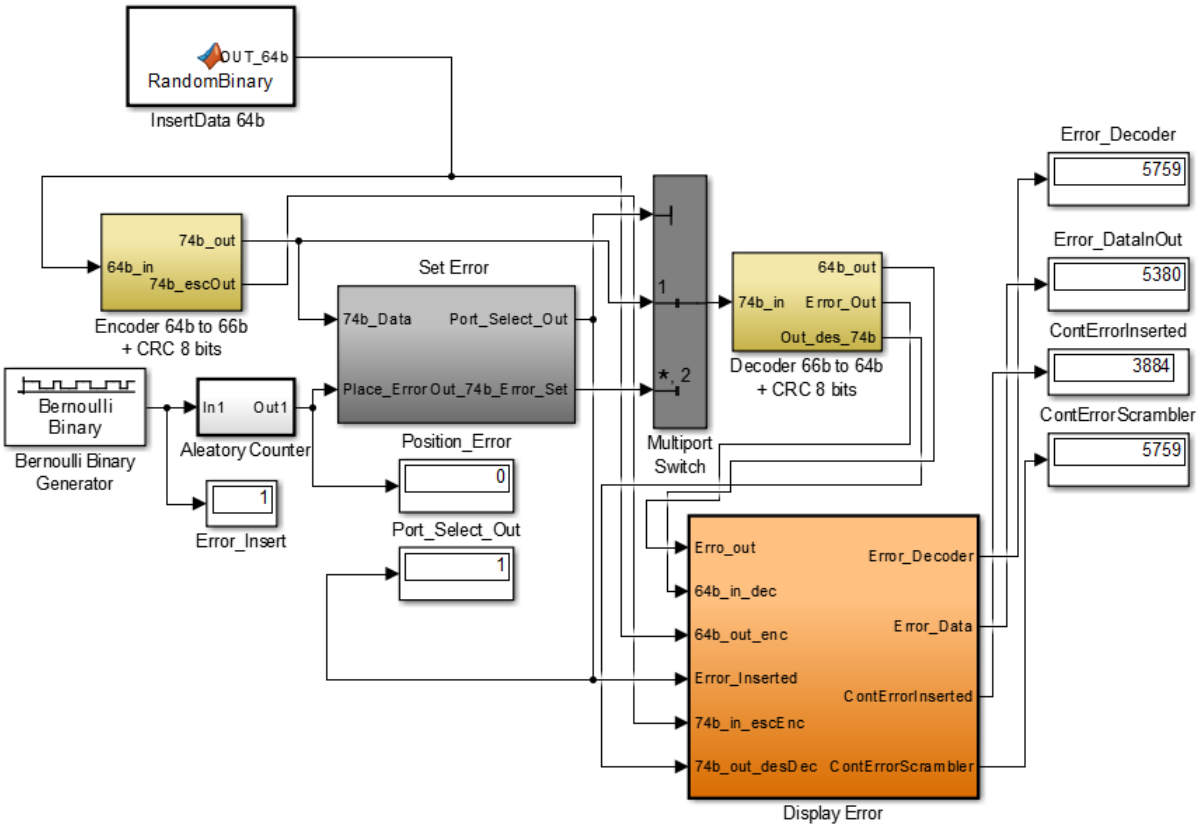
Fonte: Elaborado pelo Autor

Na figura Figura 19 é ilustrado o sistema desenvolvido usando a ferramenta EMFB do Simulink. Neste sistema observa-se o bloco que possui a função *RandomBinary* que realiza a inserção de dados binários randômicos de 64 bits. Os subsistemas *Bernoulli Binary Generato* e *Aleatory Counter* (*ALC*) são responsáveis por gerarem o erro na transmissão. Este

erro é inserido após o codificador gerar um dado de 74 bits.

No subsistema *Set Error* é aplicado o erro no dado transmitido e no subsistema Multiport Switch (MPS) é selecionado se há ou não erro na transmissão. No subsistema *Decoder 66b to 64b + CRC 8 bits* os dados são decodificados para 64 bits novamente. No subsistema *Display Error* o número de erros do sistema é obtido de quatro formas. Primeiramente analisa-se o sinal de saída do *decoder Error_out*, caso estiver com valor lógico alto aciona um contador. A segunda obtêm-se o número de erros pela comparação entre os dados de entrada do *encoder* com os dados de saída do *decoder*, caso forem diferentes aciona um contador registrando o erro. A terceira é comparando o dado de 74 bits que entra no *scrambler* com o dado de 74 bits de saída do *descrambler*, o resultado é exibido na saída *ContErrorScrambler* do bloco *Display Error*. A quarta forma é a contagem de erros inseridos no canal, sendo exibido na saída *ContErrorInserted* do bloco *Display Error*

Figura 19: Sistema da codificação 64b66b implementado no Matlab™(SIMULINK).



Fonte: Elaborado pelo Autor

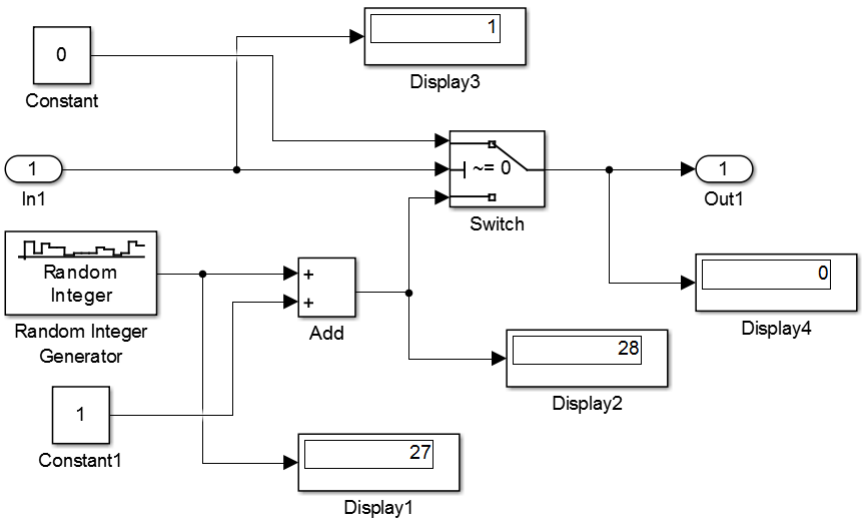
O subsistema BBG gera números binários de 1 bit (0's ou 1's) aleatoriamente de acordo com uma porcentagem pré-definida. Pode-se dizer que este subsistema é como

se fosse uma moeda viciada, de acordo com um número total de eventos define-se a porcentagem do número de vezes que cada lado da moeda vai sair quando for jogada. O erro é gerado quando o subsistema gerar o bit 0, dessa forma dentro do subsistema pode-se definir a probabilidade de ocorrer erro no canal.

Na Figura 20 observa-se a estrutura dentro do subsistema ALC que recebe o bit do BBG. Desta forma, caso o ALC receba um bit '0' do BBG é enviado um número aleatório ao subsistema *Set Error* representando a posição que o erro vai ser inserido no vetor de 74 bits. Caso o ALC receba um bit '0' do BBG então envia-se um número 0 para o *Set Error*, não gerando nenhum erro na transmissão. O subsistema *Set Error* seleciona a porta 1 do MPS, se não for inserido erro, ou porta 2 quando não há erros na transmissão. Pela Figura 20 observa-se a entrada In1 a qual recebe o bit do subsistema BBG selecionando, através do bloco Switch, entre a constante de valor inteiro "1" ou o subsistema *Random Integer generator* (RIG) adicionado com uma constante de valor inteiro "1" como saída.

O RIG está configurado para gerar números de "0" até "73", dessa forma somado com a constante de valor inteiro "1" são gerados números de "1" até "73" aleatoriamente. O subsistema Switch seleciona a constante de valor 1 se na entrada In1 possuir o bit 1, caso possuir o bit 0 é selecionado o RIG adicionado com a constante 1.

Figura 20: Estrutura Interna do bloco Aleatory Counter.

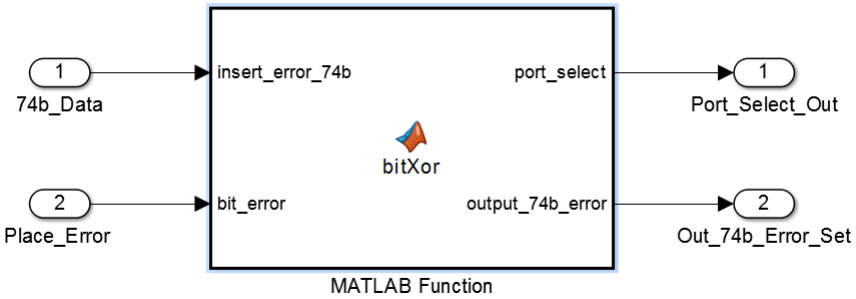


Fonte: Elaborado pelo Autor

No MPS a porta sem número é a que seleciona as portas 1 ou 2, dessa forma se na porta 1 é inserido o valor inteiro "1" logo é selecionado a porta 2 e assim por diante. Pela Figura 21, no MPS o erro é inserido quando na porta 1 estiver presente o

valor inteiro 2. Dessa forma, pela lógica presente no subsistema ALC pode-se inserir um erro aleatoriamente no canal de transmissão. Na Figura 21 apresenta-se a estrutura do subsistema Error Set que insere erros no dado de saída. No bloco está presente uma função que recebe uma posição para inserir o erro no dado transmitido. Caso esta posição seja diferente de 0, o erro é inserido no dado transmitido e a porta do MPS é selecionada para 2. Caso o bit recebido seja 0, é selecionado a posição 1 no MPS e nenhum erro é inserido.

Figura 21: Estrutura interna do subsistema *Set Error*.



Fonte: Elaborado pelo Autor

Na Figura 22 é apresentado a programação do subsistema *Matlab*^{TM} *Function* da Figura 21 em que usa-se o comando *bitxor* do Matlab^{TM} para realizar a operação XOR. Neste comando realiza uma operação XOR entre o dado transmitido de 74 bits e o um vetor de zeros de 74 bits setado o bit 1 na posição de erro, definida no bloco ALC.

Figura 22: Programação interna do subsistema *Matlab*^{TM} *Function*.

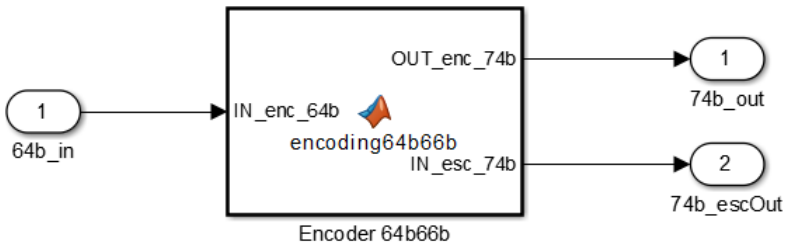
```
function [port_select,output_74b_error] = bitXor(insert_error_74b,bit_error)
%#codegen
xorOp = randi([0 0],1,74);

if strcmp(dec2bin(bit_error,1),'0')
    port_select = 1;
else
    port_select = 2;
    xorOp(bit_error)= 1;
end

output_74b_error = bitxor(insert_error_74b,xorOp);
end
```

Fonte: Elaborado pelo Autor

Na Figura 23 é apresentado a estrutura interna do subsistema *Encoder* 64b to 66b. Nesta estrutura está presente um *Function Block* em que está descrito a codificação do *encoder* 64 bits para 66 bits. Como está presente o CRC 8 bits nesta estrutura, o dado

de saída possui 74 bits. A programação completa do *Function Block Encoder 64b66b* está descrita no Apêndice A.

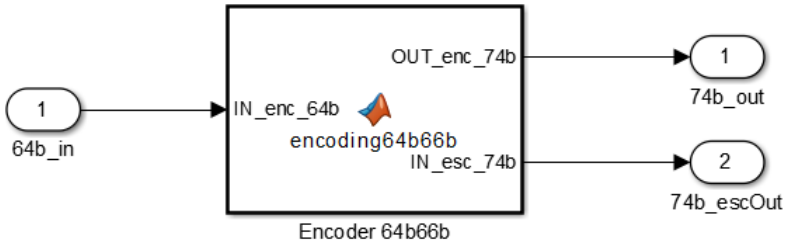
Figura 23: Estrutura interna do Encoder 64b to 66b.



Fonte: Elaborado pelo Autor

Na Figura 24 é apresentado a estrutura interna do subsistema *Decoder* 64b to 66b. Nesta estrutura está presente um *Function Block* em que está descrito a codificação do *Decoder* 66 bits para 64 bits. Como possui um CRC 8 bits implementado o dado recebido é de 74 bits. A programação completa do *Function Block Decoder* 66b64b está descrita no Apêndice B.

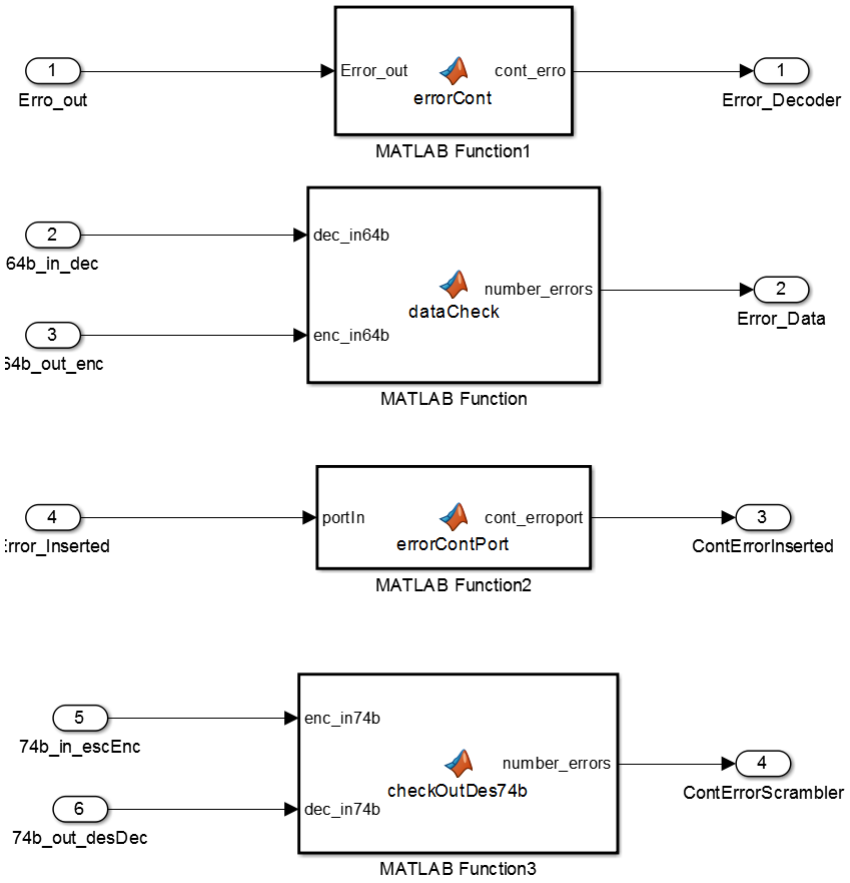
Figura 24: Estrutura interna do *Decoder* 66b to 64b.



Fonte: Elaborado pelo Autor

Na Figura 25 é apresentado a estrutura interna do subsistema *Display Error*. A estrutura possui 6 entradas sendo a primeira o sinal de saída *Error_Out* do *Decoder*, indicando um sinal de erro do dado recebido. O par de entrada *64b_in_dec* e *64b_out_enc* representam o dado de 64b de saída do *decoder* e entrada do *encoder*, respectivamente. O outro par de entrada *74b_in_escEnc* e *74b_out_desDec*, representam o dado de 74 bits que entra no *escrambler* e o que sai do *descrambler*, respectivamente. A entrada *Error_Inserted* é a porta que foi selecionado no MPS, desta forma quando é recebido o valor 2 um contador é acionado na função *errorContPort* sendo possível obter o número de erros do sistema.

Figura 25: Estrutura interna do subsistema *Display Error*.



Fonte: Elaborado pelo Autor

Todos os códigos das funções implementadas no sistema estão descritos nos apêndices no final do trabalho.

8 RESULTADOS E CONCLUSÕES FINAIS

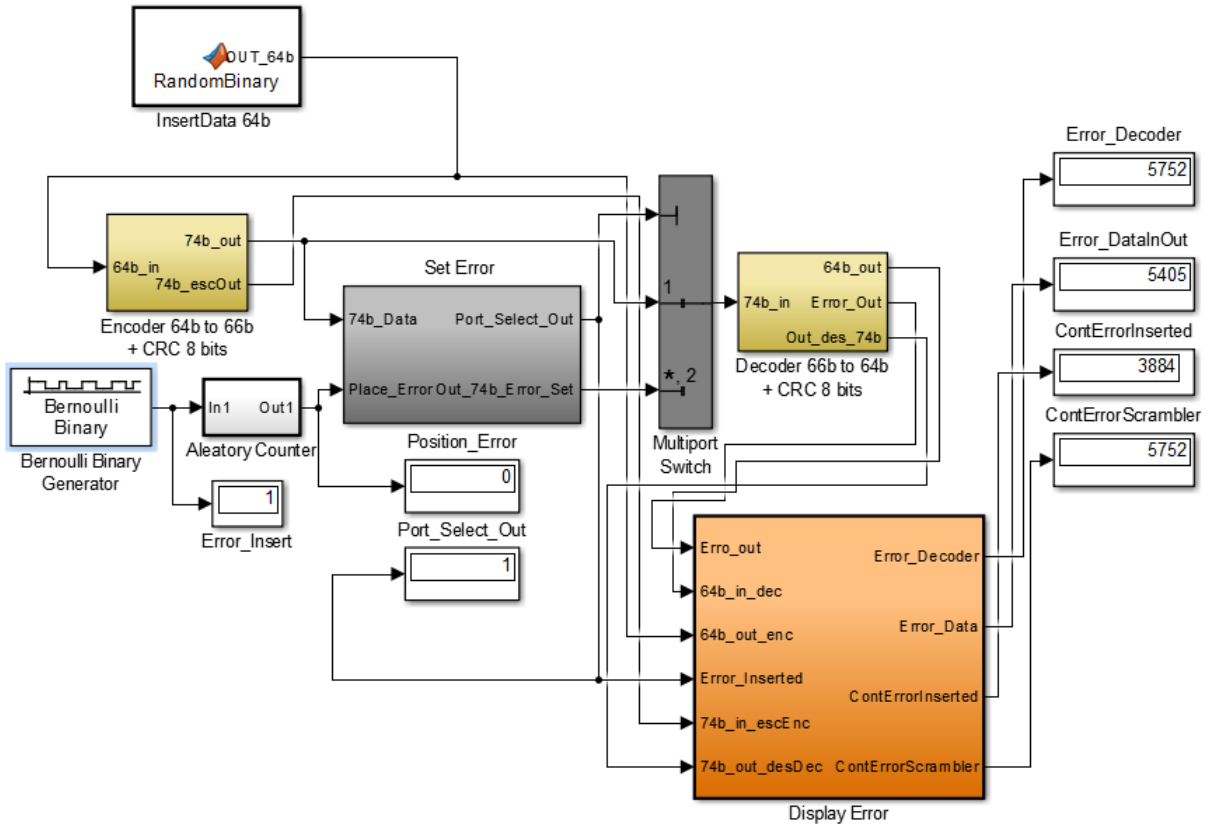
O acelerador de partículas LHC busca encontrar respostas sobre os fundamentos da matéria, mais especificamente as partículas físicas elementares. O estudo envolve uma enorme quantidade de dados e consequentemente uma enorme quantidade de colisões de partículas. Por conta dessa alta taxa de colisões, os sistemas eletrônicos desenvolvidos para o colisor trabalham em uma velocidade muito alta de processamento e estão imersos em um ambiente de uma taxa de radiação elevada.

A transmissão de dados entre os dispositivos eletrônicos do LHC possui diversos problemas. Estes problemas referem-se à alta velocidade de processamento e a alta radiação que os dispositivos estão expostos. A alta velocidade de processamento gera diversos problemas nas comunicações digitais, como por exemplo a dessincronização entre os dispositivos emissor e receptor. O ambiente com alta radiação induz ruídos nos dados transmitidos, sendo necessário um mecanismo de detecção de erros. O sistema implementado da codificação 64b66b possibilita a detecção de erros nos dados transmitidos e o sincronismo entre os dispositivos comunicantes. A detecção de erros na codificação 64b66b não é robusta, sendo necessário implementar um CRC. Dependendo da característica deste sistema pode-se detectar todos os erros inseridos na transmissão. O sincronismo é garantido por meio do *scrambler*, uma vez que esse sistema fornece um balanço no número de bits $1's$ e $0's$ sendo possível circuitos recuperadores de *clock* atuarem.

O estudo das características da codificação 64b66b foi obtido testando o sistema descrito no programa Matlab^{TM}, dentro do ambiente do Simulink. O programa desenvolvido da codificação recebe os 64 bits na entrada do sistema passando-o pelo CRC, posteriormente pelo *scrambler* e no final adiciona os 2 bits de sincronismo. Posteriormente o dado codificado de 74 bits passa pelo canal de transmissão que pode ser adicionado erros ou não. O sistema da decodificação do dado de 74 bits possui o caminho inverso. Primeiramente o dado sem os bits de sincronismo passa pelo *descrambler* desembaralhando o dado. Este dado desembaralhado é inserido no CRC para verificar se ocorreram erros. Posteriormente, faz-se uma verificação dos registradores do CRC (se estão todos em nível lógico 0) e os bits de sincronismo (se são $'01'$). Caso contrário é fornecido um sinal de erro no *decoder*.

Para uma simulação de teste, introduziu-se no BBG uma probabilidade de erro de 40% e obteve-se o número de erro introduzidos, detectados e também os gerados pelo *scrambler*. Esta simulação está descrita na Figura 26 e no total foram transmitidos 10000 pacotes de 74 bits.

Figura 26: Simulação do sistema implementado da codificação 64b66b.



Fonte: Elaborado pelo Autor

Para uma probabilidade de erro no canal de 40%, obteve-se 3884 erros inseridos presentes no *display* $ContErrorInserted$ e 5752 erros gerados ao total obtidos pelo *display* $ContErrorScrambler$. Pode-se observar o comportamento do *scrambler* na presença de erros, uma vez que o número de erros presentes na transmissão é maior do que o número de erros inseridos no canal. Isto deve-se ao fato de o *descrambler* carregar um erro para o novo dado de entrada em alguns casos. Estas possibilidades foram descritas na seção 6.6.

Pelo *display* $Error\_Decoder$ obteve-se 5752 erros detectados, sendo o mesmo número de erros gerados após o *descrambler* atuar no dado transmitido. Desta forma, pode-se confirmar a teoria desenvolvida no Capítulo 5 uma vez que o CRC detectou todos os erros únicos gerados no sistema. Portanto, o sistema desenvolvido para detectar erros é bastante confiável e robusto, capaz de fornecer dados totalmente confiáveis ao final da decodificação pois sinaliza os que estão errados para erros únicos.

Pelo *display* $Error\_dataInOut$ obteve-se 5405 erros detectados entre o dado de entrada e o de saída de 64 bits, sendo menor que o total de erros detectados no *decoder*. Isto acontece pois o erro pode ocorrer dentro da faixa do dado de 64 bits, dos 2 bits de sincronismo ou do resultado de 8 bits do CRC. Desta forma, o número de erros entre o

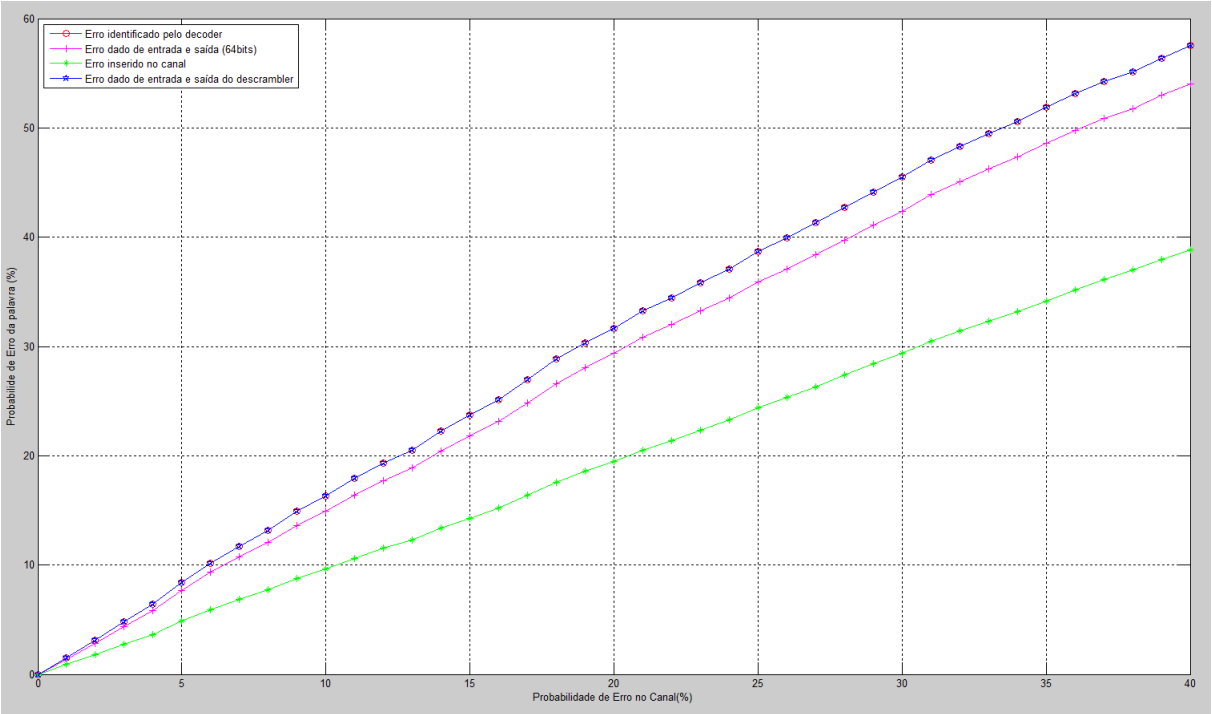
dado de entrada e de saída do sistema da codificação é explicado pelo fato de algumas vezes o erro ocorrer na faixa do dado do CRC ou dos bits de sincronismo.

Em uma transmissão de dados em alta velocidade é impossível não haver erros na transmissão, mesmo com a implementação de uma codificação muito robusta. Nota-se que o sistema cumpre as características da codificação, pois ao codificar obtêm-se dados para serem transmitidos com a quantidade de bits 0's e bits 1's balanceada garantidos pelo sistema *scrambler*. No lado do *decoder* é possível detectar todos os erros na transmissão por meio do CRC implementado.

Para observar a proporção de erros em relação ao número de dados transmitidos no sistema, configurou-se no ambiente Simulink o número máximo de simulações para 1000 com passos de 0.1 totalizando 10000 simulações. Variou-se a probabilidade de erro no canal no bloco *Bernoulli Binary Generator* de 0 até 40 por cento, coletando o número de erros da comparação entre o dado de entrada e saída do sistema ($Error\_DataInOut$), de entrada e saída do *scrambler* ($ContErrorScrambler$), do sinal de erro do *decoder* ($Error\_Decoder$) e o número de erros inseridos no sistema ($ContErrorInserted$). Na Tabela 1, observa-se a porcentagem de erro obtido na transmissão de acordo com a variação da probabilidade de erro no canal.

Na Figura 27 é apresentado um gráfico com os dados da Tabela 1. Observa-se que o comportamento da codificação em relação aos erros acometidos em um bit do dado transmitido, é aproximadamente linear para todas as probabilidades de erro no canal até 40%. Pelos dados observados, o *decoder* é capaz de detectar todos os erros gerados no canal. Observa-se, que a diferença entre os erros do dado de entrada e saída do *scrambler* e o erro inserido no canal é no máximo de aproximadamente 20%, de acordo com a simulação de até 40% de probabilidade de erro no canal. Em casos de baixa probabilidade de erro no canal de transmissão, a diferença está em torno de 10%. Dessa forma, observa-se que a codificação é adequada para a transmissão uma vez que mesmo gerando mais erros do que os inseridos é possível identificá-los em sua totalidade.

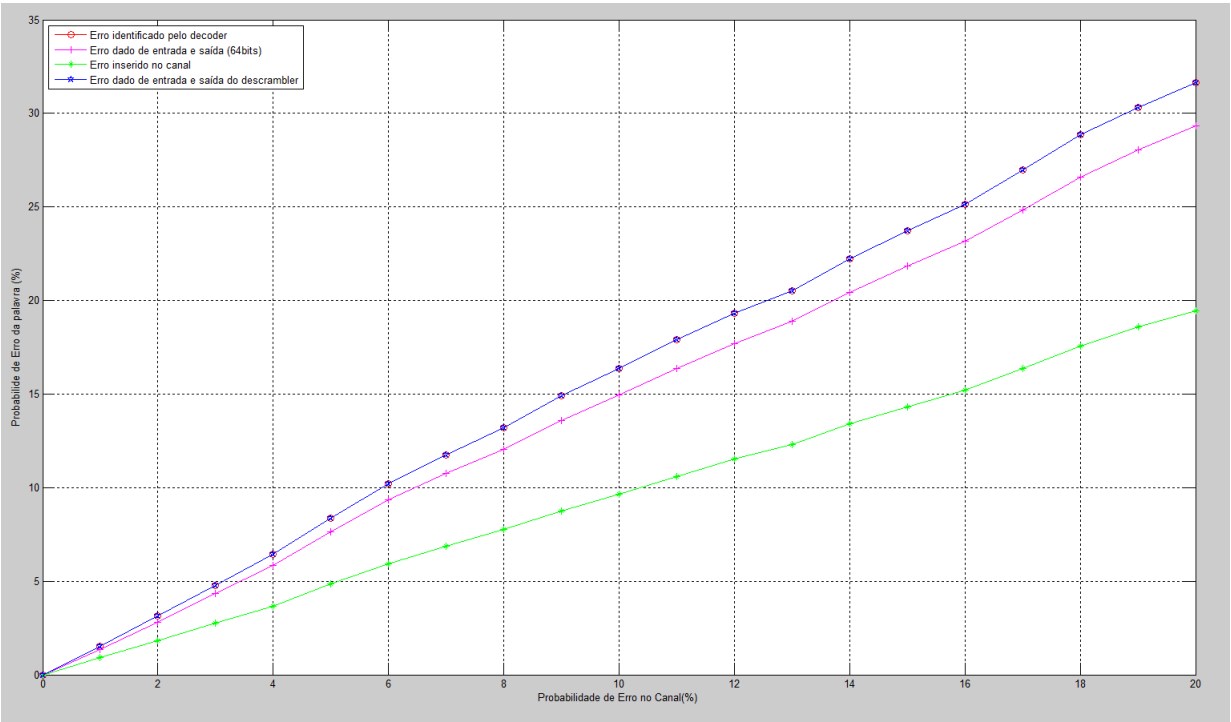
Figura 27: Gráfico do comportamento da codificação 64b66b em relação aos erros.



Fonte: Elaborado pelo Autor

Para transmissões em alta velocidade, normalmente trabalha-se com uma probabilidade de erro de aproximadamente 5% no canal de transmissão. Na Figura 28 é apresentado o gráfico com os dados da Tabela 1 traçados com uma probabilidade de erro no canal de transmissão de até 20%. Neste caso, observa-se um comportamento aproximadamente linear, sendo obtido uma porcentagem de erro no dado de entrada e saída do *scrambler* de 8.38% para uma probabilidade de erro no canal de 5% , de acordo com a Tabela 1. Para uma probabilidade de erro no canal de 5%, o *decoder* identificou todos os erros gerados, dessa forma a porcentagem de erro na transmissão pode ser reduzida a zero.

Figura 28: Comportamento do sistema da codificação 64b66b com a probabilidade de erros de até 20%.



Fonte: Elaborado pelo Autor

Tanto o sistema do *encoder* e *decoder*, foram intensamente testados e analisados por meio das entradas e saídas produzidas, observando a capacidade de detecção de erros e a taxa de erros adicionais inserido no sistema por meio do *scrambler*. Observou-se que o sistema descrito no software Matlab^{TM} no ambiente do Simulink é capaz de codificar e decodificar dados seguindo a descrição da codificação 64b66b, além de ser capaz de identificar todos os erros simples gerados no canal de transmissão por meio do CRC implementado.

As características obtidas da codificação, observou-se que o sistema implementado gera um *overhead* na palavra codificada total de 15,625%, contra os 3,125% descritos pela codificação original. Este aumento no *overhead* da palavra codificada deve-se à introdução do CRC 8 bits introduzido para detecção de erros, uma vez que a descrição da codificação 64b66b não possui uma detecção de erros robusta. Porém, um *overhead* de 15,625% é menor do que 25% descritos pela codificação 8b10b.

Portanto, conclui-se que uma implementação em um sistema real pode definir qual a melhor codificação a ser utilizada para um ambiente de transmissões de dados puros em alta velocidade. Este tipo de comparação é necessária uma vez que em uma implementação real em um FPGA deve-se considerar outros fatores, como por exemplo o *delay* na codificação do dado.

Tabela 1: Porcentagem de erros obtidos com a variação da probabilidade de erro no canal do sistema da codificação 64b66b

Probabilidade de erro no Canal(%)	**Erro detectado pelo decoder(%)**	**Erro no dado de entrada e saída de 64 bits(%)**	**Erros Inseridos no canal de transmis-são(%)**	**Erro no dado de entrada e saída do scrambler(%)**
0	0	0	0	0
1	1,52	1,34	0,9	1,52
2	3,14	2,82	1,81	3,14
3	4,77	4,34	2,74	4,77
4	6,44	5,85	3,66	6,44
5	8,38	7,65	4,87	8,38
6	10,19	9,33	5,91	10,19
7	11,73	10,74	6,87	11,73
8	13,19	12,04	7,76	13,19
9	14,90	13,60	8,75	14,9
10	16,35	14,95	9,63	16,35
11	17,92	16,38	10,59	17,92
12	19,32	17,70	11,52	19,32
13	20,53	18,87	12,31	20,53
14	22,23	20,44	13,40	22,23
15	23,73	21,85	14,30	23,73
16	25,12	23,16	15,20	25,12
17	26,98	24,85	16,37	26,98
18	28,85	26,59	17,57	28,85
19	30,32	28,04	18,58	30,32
20	31,66	29,34	19,46	31,66
21	33,23	30,82	20,52	33,23
22	34,45	32,03	21,41	34,45
23	35,81	33,26	22,35	35,81
24	37,05	34,42	23,29	37,05
25	38,66	35,89	24,40	38,66
26	39,93	37,06	25,31	39,93
27	41,33	38,40	26,31	41,33
28	42,73	39,74	27,36	42,73
29	44,14	41,09	28,40	44,14
30	45,50	42,39	29,36	45,50
31	47,07	43,89	30,45	47,07
32	48,29	45,10	31,40	48,29
33	49,45	46,25	32,30	49,45
34	50,59	47,36	33,17	50,59
35	51,88	48,56	34,16	51,88
36	53,14	49,78	35,18	53,14
37	54,21	50,86	36,16	54,21
38	55,14	51,78	36,99	55,14
39	56,36	52,97	37,93	56,36
40	57,52	54,05	38,84	57,52

Fonte: Elaborada pelo autor.

REFERÊNCIAS

ABINAYA, N. S.; PRAKASAM, P. Performance analysis of maximum length lfsr and bbs method for cryptographic application. In: **2014 International Conference on Electronics and Communication Systems (ICECS)**. [S.l.: s.n.], 2014. p. 1–5.

BRÜNING, L. R. **HIGH LUMINOSITY LARGE HADRON COLLIDER A DESCRIPTION FOR THE EUROPEAN STRATEGY PREPARATORY GROUP**. 2018. Disponível em: <https://cds.cern.ch/record/1471000/files/CERN-ATS-2012-236.pdf>. Acesso em: 20 jan. 2018.

COLLABORATION, C. **Techninal Proposal for the Phase-II Upgrade of the Compact Muon Solenoid**. 2018. Disponível em: <http://www.desy.de/~garutti/LECTURES/ParticleDetectorSS12/JournalClub/lhc-CMS.pdf>. Acesso em: 20 jan. 2018.

COMER, D. E. **Redes de Computadores e Internet**. São Paulo: Bookman, 2016. 557 p.

DHINGRA, S. **Comparison of LFSR and CA for BIST**. Auburn: Dept. of Electrical and Computer Engineering, 2018. 47 p. Disponível em: <http://www.eng.auburn.edu/~agrawvd/COURSE/E7250_05/REPORTS_TERM/Dhingra_LFSR.pdf>. Acesso em: 18 jan. 2018.

FARRELL, J. C.; MOREIRA, P. G. **Essentials of Error-Control Conding**. Upper Saddle River: John Wiley & Sons, 2006.

FERREIRA, B. C. **DETECÇÃO DE RAIOS CÓSMICOS COM CALORIMETRIA DE ALTAS ENERGIAS**. 2009. 118 p. Dissertação (Mestrado em Engenharia Elétrica) — UNIVERSIDADE FEDERAL do Rio de Janeiro, Rio de Janeiro, 2009.

FLOYD, T. L. **Digital Fundamentals**. New Delhi: Pearson, 2009. 497 p.

HASSAN, G. M. **Scramble Image Based on LFBSR (Linear Feedback Shift Registers)**. Baghdad: Dept. of Electrical and Computer Engineering, 2018. 14 p. Disponível em: <https://www.iasj.net/iasj?func=fulltext&aId=58271>. Acesso em: 20 jan. 2018.

ISO/IEC/IEEE International Standard for Ethernet. **ISO/IEC/IEEE 8802-3:2014(E)**, p. 1–3754, April 2014.

KERL, J. **Computation in finite fields**. [s.n.], 2004. Disponível em: <http://johnkerl.org/doc/ffcomp.pdf>. Acesso em: 08 janeiro 2018.

KOOPMAN, P. **Best CRC Polynomials**. Carnegie Mellon University, 2018. Disponível em: <http://users.ece.cmu.edu/~koopman/crc/index.html>. Acesso em: 20 jan. 2018.

_____. **Best CRCs**. [S.l.], 2018. Disponível em: <http://users.ece.cmu.edu/~koopman/crc/index.html>. Acesso em: 24 jan 2018.

KOOPMAN, P.; CHAKRAVARTY, T. **Cyclic Redundancy Code CRC Polynomial Selection For Embedded Networks**. Florences: International Conference on Dependable Systems and Network, 2004. 11 p. Disponível em: <http://users.ece.cmu.edu/~koopman/roses/dsn04/koopman04_crc_poly_embedded.pdf>. Acesso em: 20 jan. 2018.

MACHADO, R. **Problemas de Transmissão**. Santa Maria: UFSM, 2018. 11 p. Disponível em: <http://coral.ufsm.br/gpscom/professores/Renato%20Machado/comunicacaodedados.html>. Acesso em: 20 jan. 2018.

MORENO, R. L. **Implementação em FPGA de uma Arquitetura Reed-Solomon para Uso em Comunicações Ópticas**. 2010. 68 p. Dissertação (Mestrado em Automação e Sistemas Elétricos Industriais) — UNIVERSIDADE FEDERAL DE ITAJUBÁ, UNIFEI, Itajubá - MG, 2010.

RANDALL, L. **BATENDO À PORTA DO CÉU: O BÓSON DE HIGGS E COMO A FÍSICA MODERNA ILUMINA O UNIVERSO**. São Paulo: Companhia das Letras, 2013. 576 p.

SILVA, D. C. C. e. **Estudo da Codificação de Rede e Análise do seu Desempenho com Fonte de Tráfego HTTP**. 2011. 98 p. Dissertação (Mestrado em Telecomunicações) — Instituto Nacional de Telecomunicações, INATEL, Santa Rita do Sapucaí - MG, 2011.

SPRACE. **Sprace research group**. São Paulo: [s.n.], 2018. Disponível em: <http://sprace.org.br>. Acesso em: 20 jan. 2018.

TSESMELIS, E. **THE COMPACT MUON SOLENOID (CMS) EXPERIMENT: THE LHC FOR HIGH ENERGY AND LUMINOSITY**. 2018. Disponível em: <http://www.desy.de/~garutti/LECTURES/ParticleDetectorSS12/JournalClub/lhc-CMS.pdf>. Acesso em: 20 jan. 2018.

WIKIPÉDIA. **GRANDE COLISOR DE HÁDRONS**. 2018. Disponível em: <https://pt.wikipedia.org/wiki/Grande_Colisor_de_Hádrons>. Acesso em: 20 jan. 2018.

Apêndices

APÊNDICE A – ENCODER 64B66B (74 BITS COM O CRC)

```
1  function [OUT_enc_74b,IN_esc_74b] = encoding64b66b(IN\_enc\_64b)
2
3  % -------"CODIFICADOR 64B/66B - VICTOR AFONSO DOS REIS - FEIS ...
       UNESP"-------
4
5  % -----Entradas e saídas---------------------
6  %IN_64b:[X0 X1 X2 ... X62 X63 X64]
7  %OUT_enc_74b: [Y1 Y2 Y3 ... Y72 Y73 Y74]
8  %OUT_2b: [Y0 Y1]
9
10 % Variáveis
11 persistent reg_scrambler;
12 OUT_enc_2b = zeros(1,2);
13 IN_esc_74b = zeros(1,74);
14 %% TRATAMENTO DA ENTRADA
15 %CRC
16 valCRC = 0;
17 msgIn = [zeros(1,8) IN_enc_64b]; % Adiciona 8 zeros no início ...
       do dado
18 regCRCenc = ones(1,8);
19
20 %% Inicialização das saídas
21 enc\_2b = dec2bin(1,2); % Seta para '01' pois só sera ...
       transmitido dado
22 inLength = length(enc_2b);
23 for i = 1:inLength
24 if strcmp(enc_2b(i),'0')
25 OUT_enc_2b(i) = 0;
26 else
27 OUT_enc_2b(i) = 1;
28 end
29 end
30
31 %CRC
32 OUT_line_72b = randi([0 0],1,72);
33
34 %% CRC - Polinômio G(x) = x^8 + x^2 + x + 1
```

```
35 lenCRC = length(msgIn);
36 for j = lenCRC:-1:1
37 valCRC = regCRCenc(8);
38 regCRCenc = circshift(regCRCenc,[0 1]);
39 regCRCenc(3) = bitxor(regCRCenc(3),valCRC);
40 regCRCenc(2) = bitxor(regCRCenc(2),valCRC);
41 regCRCenc(1) = bitxor(valCRC,msgIn(j));
42 end
43 finalXOR = ones(1,8);
44 regxor = bitxor(regCRCenc,finalXOR);
45
46 OUT_line_72b = [IN_enc_64b regxor];
47 IN_esc_74b = [OUT_enc_2b OUT_line_72b];
48 %% Scrambler - Polinômio G(x) = x^58 + x^39 + 1
49 if isempty(reg_scrambler)
50 reg_scrambler = ones(1,58);
51 end
52 val = zeros(1,72);
53 v = randi([0 0],1,1);
54 for j = 1:72
55 v = bitxor(reg_scrambler(58),reg_scrambler(39));
56 v = bitxor(v, OUT_line_72b(j));
57 reg_scrambler = circshift(reg_scrambler, [0 1]);
58 reg_scrambler(1) = v;
59 val(j) = v;
60 end
61
62 OUT_enc_74b = [OUT_enc_2b val];
63 end
```

APÊNDICE B – DECODER 64B66B (74 BITS COM O CRC)

```
1  function [OUT_dec_64b,error_sig,OUT_des_74b] = ...
       decoding66b64b(IN_des_74b)
2
3  % ------"DECOFICADOR 64B/66B - VICTOR AFONSO DOS REIS - FEIS ...
       UNESP"--------
4
5  %-----Entradas e saídas---------------------
6  %IN_des_74b:[X1 X2 X3 ... X72 X73 X74]
7  %OUT_dec_64b: [Y1 Y2 Y3 ... Y62 Y63 Y64]
8  %error_sig: [Y1]
9  %OUT_des_74b: [Y1 Y2 Y3 ... Y72 Y73 Y74] Presente só para ...
       verificação de
10 %erros
11
12 %% Variáveis
13 persistent reg_descrambler;
14 OUT_dec_64b = randi([0 0],1,64);
15 OUT_des_74b = randi([0 0],1,74);
16 %CRC
17 OUT_crcDec_2b = randi([0 0],1,2);
18 regCRCdec= ones(1,8);
19 valCRC2 = 0;
20 CRC_word = randi([0 0],1,8);
21
22 %% Tratamento da Entrada
23
24 IN_des_72b = zeros(1,72);
25 for i = 1:74
26 if (i < 3)
27 OUT_crcDec_2b(i) = IN_des_74b(i);
28 else
29 IN_des_72b(i-2) = IN_des_74b(i);
30 end
31 end
32
33 %% Scrambler - Polinômio G(x) = x^58 + x^39 + 1
34 if isempty(reg_descrambler)
```

```
35 reg_descrambler = ones(1,58); % Estado inicial dos registradores
36 end
37 out = zeros(1,72);
38 v2 = zeros(1,1);
39 for j = 1:72
40 v2 = bitxor(reg_descrambler(58),reg_descrambler(39));
41 v2 = bitxor(v2, IN_des_72b(j));
42 reg_descrambler = circshift(reg_descrambler, [0 1]);
43 reg_descrambler(1) = IN_des_72b(j);
44 out(j) = v2;
45 end
46
47 OUT_des_72b = out;
48 OUT_des_74b = [OUT_crcDec_2b out];
49
50 %% CRC - Polinômio G(x) = x^8 + x^2 + x + 1
51
52 for i = 1:72
53 if (i < 65)
54 OUT_dec_64b(i) = OUT_des_72b(i);
55 elseif (i > 64)
56 CRC_word(i-64) = OUT_des_72b(i);
57 end
58 end
59
60 CRC_detect = [CRC_word OUT_dec_64b];
61
62
63 lenCRC = length(CRC_detect);
64 for j = lenCRC:-1:1
65 valCRC2 = regCRCdec(8);
66 regCRCdec = circshift(regCRCdec,[0 1]);
67 regCRCdec(3) = bitxor(regCRCdec(3),valCRC2);
68 regCRCdec(2) = bitxor(regCRCdec(2),valCRC2);
69 regCRCdec(1) = bitxor(valCRC2,CRC_detect(j));
70 end
71 finalXOR2 = ones(1,8);
72 regCRCdec = bitxor(regCRCdec,finalXOR2);
73
74 error_crc = 0;
75 lenreg = length(regCRCdec);
76 for i =1:lenreg
```

```
77 if (regCRCdec(i) == 1)
78 error_crc = 1;
79 end
80 end
81
82 if ((OUT_crcDec_2b(1) == 0) && (OUT_crcDec_2b(2) == 1) && ...
       (error_crc == 0))
83 error_sig = randi([0 0],1,1);
84 else
85 error_sig = randi([1 1],1,1);
86 end
87
88 end
```

APÊNDICE C – FUNÇÃO DE INSERÇÃO DE ERRO NO CANAL (BITXOR)

```
1  function [port_select,output_74b_error] = ...
       bitXor(insert_error_74b,bit_error)
2  %% Inicialização do vetor que irá armazenar a posição do erro
3  xorOp = randi([0 0],1,74);
4
5  %% Inserção do erro ou não no vetor
6  if strcmp(dec2bin(bit_error,1),'0')
7  port_select = 1;
8  else
9  port_select = 2;
10 xorOp(bit_error)= 1;
11 end
12
13 %% Seta saída
14 output_74b_error = bitxor(insert_error_74b,xorOp);
15 end
```

APÊNDICE D – FUNÇÃO PARA CONTAR OS SINAIS DE ERRO DO DECODER (ERRORCONT)

```
1  function cont_erro = errorCont(Error_out)
2  %% Variáveis
3  persistent error_countDec;
4  %% Inicialização do contador
5  if isempty(error_countDec)
6  error_countDec = 0;
7  end
8  %% Detecção do Erro
9  if (Error_out ≠ 0)
10 error_countDec = error_countDec + 1;
11 end
12 %% Incremento no contador
13 cont_erro = error_countDec;
14 end
```

APÊNDICE E – FUNÇÃO PARA DETECTAR E CONTAR ERRO NOS DADOS DE ENTRADA E SAÍDA DE 64 BITS (DATACHECK)

```
1  function number_errors= dataCheck(dec_in64b,enc_in64b )
2  %% Variáveis
3  persistent error_data
4
5  %% Inicialização do contador de erros
6  if isempty(error_data)
7  error_data = 0;
8  end
9
10 %% Detecção do Erro
11 error_cont = 0;
12 for i = 1:64
13 if (dec_in64b(i) ≠ enc_in64b(i))
14 error_cont = error_cont + 1;
15 end
16 end
17
18 %% Incremento do contador
19 if (error_cont ≠ 0)
20 error_data = error_data + 1;
21 end
22
23 %% Fornecendo o número de erros na saída
24 number_errors = error_data;
25 end
```

APÊNDICE F – FUNÇÃO PARA VERIFICAR E CONTAR OS ERROS INSERIDOS NO CANAL DE TRANSMISSÃO (ERRORCONTPORT)

```
1  function cont_erroport = errorContPort(portIn)
2  %% Variáveis
3  persistent error_count;
4
5  %% Inicialiazação do Contador
6  if isempty(error_count)
7  error_count= 0;
8  end
9  %% Verificação do Erro
10 if (portIn == 2)
11 error_count = error_count + 1;
12 end
13
14 %% Externalizando o número de erros
15 cont_erroport = error_count;
16 end
```

APÊNDICE G – FUNÇÃO PARA VERIFICAR E CONTAR OS ERROS ENTRE O DADO DE ENTRADA DO SCRAMBLER E SAÍDA DO DESCRAMBLER (CHECKOUTDES74B)

```
function number_errors = checkOutDes74b(enc_in74b,dec_in74b)
%% Variáveis
persistent error_data

%% Inicialização do contador de erro.
if isempty(error_data)
error_data = 0;
end

%% Detecção do Erro
error_cont = 0;
for i = 1:74
if (dec_in74b(i) ≠ enc_in74b(i))
error_cont = error_cont + 1;
end
end

%% Incremento no contador
if (error_cont ≠ 0)
error_data = error_data + 1;
end

%% Fornece o número de erros para a saída
number_errors = error_data;
end
```

APÊNDICE H – GERADOR DE DADOS DE 64 BITS PARA A ENTRADA DO SISTEMA (RANDOMBINARY)

```
1 function OUT_64b = RandomBinary
2 %#codegen
3
4 OUT_64b = randi([0 1],1,64);
5
6 end
```