UNIVERSIDADE FEDERAL DO PARANÁ – UFPR
CURSO SUPERIOR DE ENGENHARIA ELÉTRICA

LEANDRO SANT'ANA BELLI

IMPLEMENTAÇÃO EM HARDWARE DE UM CÓDIGO DE HAMMING (72,64)

CURITIBA
2016

LEANDRO SANT'ANA BELLI

IMPLEMENTAÇÃO EM HARDWARE DE UM CÓDIGO DE HAMMING (72,64)

Trabalho de conclusão de curso apresentado como requisito parcial para graduação no curso de Engenharia Elétrica da Universidade Federal do Paraná.

Orientadora: Profª. Drª. Sibilla Batista da Luz França

CURITIBA

2016

RESUMO

Códigos corretores de erros são frequentemente utilizados para melhorar a qualidade da informação dos dados transmitidos. Os erros na transmissão de uma mensagem podem ocorrer de várias formas, estes erros podem ser detectados e corrigidos através de métodos matemáticos, aumentando a confiabilidade dos sistemas de comunicação, utilizando códigos detectores e/ou corretores de erros. Este projeto teve como objetivo realizar a implementação em hardware de um codificador e um decodificador genérico, utilizando códigos de Hamming, a fim de flexibilizar as sínteses e testes para diferentes tamanhos de códigos. A linguagem utilizada na implementação do algoritmo genérico proposto neste projeto foi a linguagem de programação VHDL. A partir das simulações realizadas no envio de informação por um canal ruidoso e comparando o resultado com as mesmas mensagens enviadas sem que estas estivessem codificadas/decodificadas, mostrou-se eficaz quando da implementação em hardware FPGA.

Palavras-chave: códigos corretores de erros, códigos de Hamming. FPGA. VHDL.

ABSTRACT

By looking at improving the quality of transmitted data, they can be used for error codes. Errors in the transmission of a message can be used in different ways, these errors can be detected and corrected through mathematical methods, increasing the reliability of communication systems, detectors and / or error correctors. This project aimed to perform a hardware implementation of an encoder and a generic decoder, using Hamming codes, an end of flexibility as synthases and tests for different code sizes. The language used in the implementation of the generic algorithm proposed in this project for a VHDL programming language. From the simulations performed not sending information through a channel and comparing results with the same messages sent without them being coded / decoded, it proved to be effective when implementing FPGA hardware.

Key Words: error correcting codes, Hamming codes, FPGA e VHDL.

LISTAS DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

A comunicação é uma necessidade primordial para a humanidade. Historiadores relatam que o homem sempre teve a necessidade de expressar o que sentiam, viviam e observavam. Há milênios as formas de comunicação eram rudimentares, ilustradas através de marcas em rochas, expressão corporal, dentre outros. Posteriormente meios físicos foram desenvolvidos como cartas/telegramas, banners, outdoors, a fim de facilitar a comunicação entre os indivíduos. Atualmente, enfrentamos uma evolução significativa no âmbito digital, através da expansão da internet, ultrapassando os limites físicos.

Muitos aparelhos e dispositivos que utilizamos no dia-a-dia, além de nos trazer conforto, facilidade e segurança, apresentam pontos em comum, principalmente a conectividade entre si. A comunicação se tornou um dos principais objetivos no mundo moderno, através dela podemos saber a localização de um indivíduo ou objeto, ouvir uma transmissão via rádio ou televisão, conhecer as condições climáticas de um determinado local, enviar uma mensagem de socorro, manipular um satélite ou um míssil.

As informações que são transmitidas por diversos meios eletro/eletrônicos (antenas, cabos, satélites, entre outros) podem sofrer deformações e interferências devido a diversos fatores externos e internos aos sistemas de transmissão. Os ruídos podem alterar o conteúdo das informações e parâmetros físicos das operações de recepção dos sinais transmitidos, eventualmente tornando a informação incorreta ou incompleta.

Os sistemas de comunicação que necessitam de exatidão quanto à informação recebida, utilizam códigos corretores de erros para que a informação, ao passar por meios ruidosos não se altere a ponto de que o receptor realize a leitura correta dos dados.

Para que a informação codificada e transmitida possa chegar no decodificador em condição de ser recuperada integralmente é necessário a utilização de códigos corretores de erros, os quais adicionam bits de redundância à sequência a ser enviada permitindo a decodificação, e consequentemente, a recepção da informação integra.

Sistemas de comunicação que utilizam códigos corretores de erros, como redes de comunicação HFC (Hibrido de Fibra e Cabo) e redes de comunicação via satélite, comumente utilizam códigos corretores de erros para que a informação não sofra com ruídos pelos canais de transmissão, devido ao fato que o conteúdo das mensagens transmitidas não possa ser retransmitido, ou mesmo pelo alto custo da retransmissão. Outro dispositivo de comunicação que é comumente afetado pelo meio ruidoso e provoca a perda da informação é o aparelho celular e/ou fixa via telefone sem fio na faixa de frequência de 2.4GHz. Estes equipamentos sofrem interferências, causando distorções nas informações como: voz com baixa qualidade, perda de palavras, entre outras perdas sensíveis aos usuários. Estes equipamentos utilizam algoritmos de correção de erros para melhorar o desempenho da comunicação.

A partir das dificuldades apresentadas quanto a transmissão de mensagens e a corrupção dos dados que estas carregam por causa dos meios ruidosos no qual transitam, este projeto tem como objetivo a implementação de um codificador e decodificador para um código corretor de erros Hamming genérico em uma FPGA, para que os possíveis erros ocorridos nas mensagens transmitidas sejam corrigidos e a mensagem entregue ao usuário final corretamente.

O capítulo 2 aborda os conceitos básicos de códigos detectores e corretores de erros, os principais tipos, funcionamento, código de Hamming, conceitos sobre circuitos lógicos programáveis, e linguagem VHDL. No capítulo 3 é apresentado a metodologia aplicada para implementar o algoritmo do codificador e decodificador para o código corretor de erros de Hamming utilizando a ferramenta MATLAB e a implementação do código corretor de erros de Hamming genérico através de linguagem VHDL. O capítulo 4, apresenta os resultados obtidos nas simulações via software (MATLAB) e simulação do funcionamento do hardware para os tipos de código corretores de erros Hamming na ferramenta Isim. Ao fim projeto no capítulo 5, são apresentados as conclusões dos códigos que foram implementados tanto em software quanto em hardware, as considerações do uso de códigos corretores de erros em sistemas de transmissão e as vantagens da implementação em hardware para diferentes tamanhos de código corretor de erros de Hamming.

1.1. JUSTIFICATIVA

A definição do tema do projeto está ligado com a quantidade de informação que hoje é transmitida e pode ser perceptível ao usuário final quando esta chega corrompida, com a evolução da tecnologia a quantidade de falhas que são perceptíveis pelo usuário final vem diminuindo, este decremento de falhas ocorre principalmente pelo uso dos códigos corretores de erros.

Este projeto de implementação em hardware de um codificador e decodificador para um código corretor de erros de Hamming genérico, busca mostrar a importância do código corretor de erros em sistemas de transmissão de informação para que a informação mesmo sofrendo corrupção dos dados durante a transmissão, chegue ao destino final com os mesmos dados do envio, estes códigos são responsáveis por detectar erros e corrigir.

1.2. OBJETIVOS

Através de um codificador e decodificador lógico programável (FPGA) o projeto proposto implementa um código corretor de erros Hamming, tornando o algoritmo de codificação e decodificação genérico para que o usuário possa determinar a quantidade de bits presente na informação a ser transmitida.

1.3. OBJETIVO GERAL

Implementar em software especifico e hardware o código corretor de erros de Hamming, a implementação em hardware disponibiliza uma flexibilidade quanto ao tamanho do código a ser codificado, tornando-o genérico para que o usuário insira a matriz geradora **G** e matriz verificadora de paridade **H** para realizar a codificação e a decodificação da sequência de dados sempre que necessitar.

1.4. OBJETIVOS ESPECÍFICOS

Através da linguagem de programação VHDL e do software ISE da Xilinx, o presente projeto propõe a criar um codificador e decodificador para um código corretor de erros Hamming genérico. A ferramenta Isim do software ISE e o software MATLAB, simulam os sinais nas portas de entrada do hardware, os sinais referente a saída são exibidos na ferramenta Isim.

O codificador adiciona os *bits* de redundância calculados através da matriz geradora **G** e o sinal de entrada, o sinal codificado é submetido a um canal ruidoso que irá variar a intensidade do erro na mensagem transmitida.

A mensagem que foi submetida ao canal ruidoso entra é lida pelo decodificador que através dos métodos matemáticos e a matriz verificadora de paridade, reconstrói a mensagem, se a mensagem sofrer até um erro em seus símbolos, a mesma é corrigida e o sinal final é apresentado na porta de saída do decodificador.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. SISTEMAS DE COMUNICAÇÃO

SHANNON (1928), em seu artigo cientifico intitulado, The Mathematical Theory of Communication, descreveu as formas para codificar e transmitir uma informação, Shannon também recebeu crédito por ser um dos percursores do computador digital, apresentando como projeto de mestrado em 1937 no MIT. Reconhecido como um dos maiores contribuidores no crescimento e difusão da era digital.

Claude Shannon foi responsável pela criação de três limites fundamentais do sistema de comunicação digital, estes três limites são o teorema da codificação da fonte, o teorema da codificação do canal e o teorema da capacidade do canal.

No sentido mais fundamental a comunicação envolve a transmissão da informação de um ponto a outro, como ilustrado pela Figura 1. O sistema de transmissão opera com o envio da mensagem proveniente da fonte de informação (como por exemplo: musica, voz, imagem, ou dados de computador), então a mensagem é modulada por alguma técnica especifica que depende do meio de transmissão que será adotado para a transmissão da mensagem (QAM, QPSK, PAM, entre outras técnicas de modulação). Após a transmissão da mensagem por um meio ruidoso a mensagem é demodulada afim de entregar ao destino a mensagem originalmente enviada.

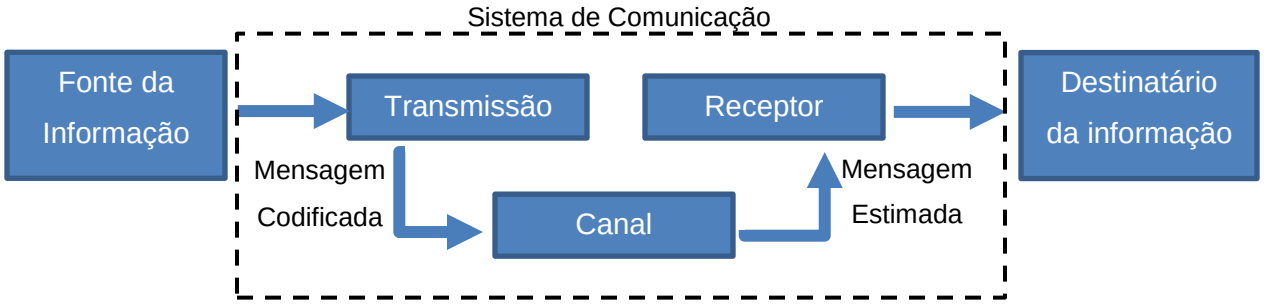

Figura 1 - Elementos do um sistema de comunicação.

Fonte: Haykin, 2001.

Segundo HAYKIN (2001), existem basicamente 2 tipos de comunicação, os *broadcasting* e *point-to-point communication*. *Broadcasting* é um tipo de comunicação que envolve apenas uma fonte de sinal, a qual transmite a inúmeros receptores, por exemplo, rádio e televisão. No modelo *point-to-point* o processo de comunicação ocorre entre um transmissor e um receptor dedicado, a mensagem enviada não é disponibilizada para demais receptores, por exemplo, a comunicação entre a Terra e um robô de exploração em uma superfície de outro planeta.

Um ponto importante que deve ser levado em consideração na transmissão de dados é a presença inevitável de ruído em um sistema de comunicação. O ruído consiste em sinais indesejados que tendem a perturbar a mensagem transmitida e corromper os seus símbolos. As fontes de ruído podem ser internas ou externas ao sistema.

Uma forma quantitativa de contabilizar o efeito do ruído é introduzir a relação sinal / ruído (SNR – S*ignal-to-Noise Ratio*) como um parâmetro do sistema, sua unidade é o decibel (dB). Por exemplo, pode-se definir o SNR na entrada do receptor como a razão da potência média do sinal pela a potência média do ruído, sendo ambas medidas no mesmo ponto.

Para LIN (1983) a transmissão e armazenamento digital da informação tem muito em comum. Ambos transferem dados de uma fonte de informação para o destinatário (ou usuário), o sistema de transmissão típico (ou armazenamento) pode ser representado por um diagrama de blocos conforme a Figura 2. A mensagem pode ser proveniente de uma pessoa ou uma máquina. A saída escolhida para a comunicação ao destino, pode ser uma forma de onda continua ou uma sequência discreta de símbolos.

O codificador transforma os dados da mensagem em uma sequência binária (*bits*) chamada de informação sequencial *u*, se a mensagem for continua é necessário uma conversão analógica para digital (A/D).

O bloco de codificação é o bloco responsável por realizar a transformação da mensagem ***u*****,** em *c*, através da adição de bits de redundância. Esta nova sequência é denominada palavra-código.

Para realizar a transmissão dos dados é necessário modular a sequência codificada, a fim de tornar viável o tráfego digital e submeter a menor quantidade de ruído possível, porém é nesta etapa que é maior a suceptividade do sistema. Por exemplo, na linha telefônica o ruído pode ser gerado por impulsos gerados por chaveamento, temperatura, cruzamento de linhas ou iluminação.

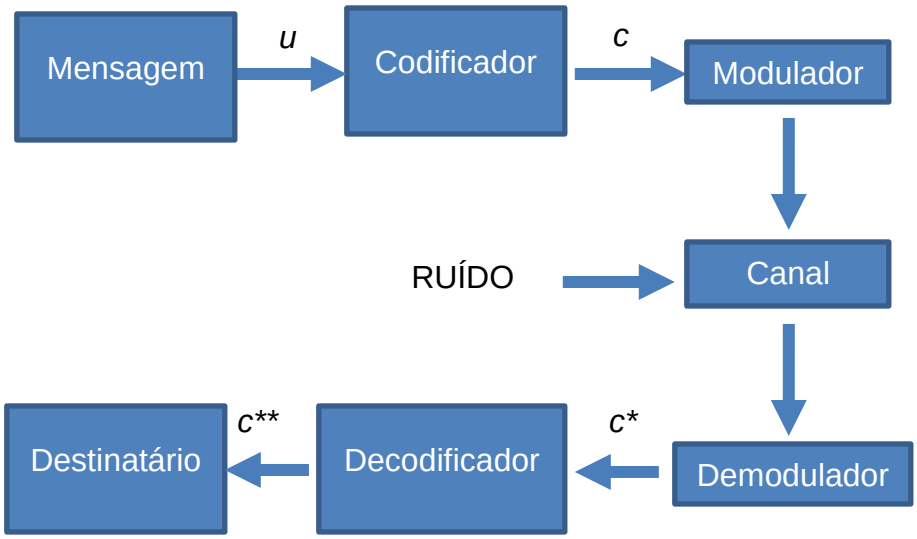

Figura 2 - Diagrama de blocos de transmissão de dados.

Fonte: Adaptado de Lin, 1983.

Os blocos de modulador e demodulador apresentados na Figura 2, são responsáveis pela conversão do sinal analógico em digital e vice e versa para que seja possível a transmissão da informação.

O decodificador é responsável por transformar a sequência do receptor *c** em uma sequência binária *c***, conhecida por ser a saída dos dados esperado (a nomenclatura *c*** diferencia a mensagem decodificada recebida no destino da mensagem original enviada *c*, pois não pode afirmar que as mensagens enviadas e recebidas são iguais). O bloco de decodificação busca através de métodos matemáticos corrigirem possíveis erros na mensagem transmitida, entretanto, se mais de um erro ocorrer na mensagem transmitida *c*,* o decodificador não consegue corrigir os símbolos e entrega na saída à mensagem *c*** corrompida.

Para os sistemas que apresentam até um erro na mensagem decodificada se comparada a mensagem original, a reprodução da mensagem será fiel a saída da fonte devido a eficiência do código corretor de erros.

2.2. CÓDIGOS DETECTORES/CORRETORES DE ERROS

2.2.1. CÓDIGOS DETECTORES DE ERROS

Segundo MENEGHESSO (2012), os códigos detectores de erros utilizam bits de paridade para definir se a mensagem transmitida possui erros. A adição de bits de paridade à mensagem original aumenta a proteção contra erros nos meios de transmissão segundo PEDRONI (2010).

Os códigos de paridade simples conhecidos como SPC (single parity check), são os códigos de detecção de erros mais simples. Um bit extra é adicionado à palavra de código original, fazendo com que a nova palavra de código exiba sempre um número par (ou ímpar) de ‘1’s. Por consequência, o código pode detectar um erro, embora não possa corrigi-lo. O exemplo mais comumente utilizado é denominado PS/2 *(personal system version* 2), este foi desenvolvido pela IBM em 1987 para a comunicação serial entre os PCs e seus periféricos (teclado e mouse). Neste método são inseridos 3 *bits*, denominados: *bit* de *start*, *bit* de paridade e *bit* de *stop*. Sendo o *start* sempre ‘0’, o *stop* sempre ‘1’ e o de paridade será ‘1’ quando o número de ‘1s’ no vetor de dados for par.

2.2.2. CÓDIGOS CORRETORES DE ERROS

Assim como nos códigos detectores de erros, os códigos corretores adicionam bits de paridade a mensagem e dependendo do algoritmo implementado para a correção, a mensagem pode ser corrigida em 1 ou mais bits.

Para o autor PEDRONI (2010) “No caso dos códigos detectores/corretores de erros, bits extras, denominados *bits* de redundância ou bits de paridade, são sempre introduzidos, portanto, o número de *bits* na saída do codificador (*n*) é sempre maior do que em sua entrada (*k*)”.

Os códigos corretores de erros são utilizados em diversas áreas da eletrônica e trazem os benefícios de aumentar a confiabilidade dos dados, ampliando a capacidade dos sistemas quanto ao processamento destes dados.

2.2.3. CÓDIGOS DE BLOCOS

Os códigos de blocos buscam dividir a mensagem original em mensagens menores, agrupadas em blocos com tamanho de k bits. Uma mensagem também chamada de sequência de dados, é representada por $u = (u_1, u_2, \dots, u_k)$.

Existe um total de 2^k diferentes possiblidades de mensagens. Este formato de 2^k palavras-código com tamanho *n* é chamado *c*(*n*,*k*) códigos de blocos.

2.2.4. CÓDIGOS DE HAMMING

Os códigos de Hamming são códigos de blocos lineares que foram desenvolvidos por Richard Hamming, baseados na adição de bits de paridade para diminuir a influência de ruídos nas transmissões digitais. São capazes de detectar dois bits e corrigir um bits de acordo com o tamanho do código a ser implementado.

O parâmetro fundamental de um código corretor de erros é distância mínima ($d_{mín}$). A distância mínima ($d_{mín}$) entre duas palavras binárias é o número de posições nas quais elas diferem.

A quantidade de palavras-código está relacionado com a distância mínima de Hamming, que para uma certa mensagem codificada a mesma possa ser decodificada corrigindo um erro se este houver.

Segundo PEDRONI (2010) os códigos corretores de erros são mais complexos de implementar em relação aos detectores. Como a distância mínima nos códigos de Hamming, é $d_{mín}$= 3, eles são capazes de corrigir apenas um erro e detectar até 2 erros. Os parâmetros dos códigos Hamming são os seguintes (para qualquer inteiro m>1):

- Número de *bits* de paridade: m;
- Número de *bits* na entrada (*bits* de informação): $k = 2^m - m - 1$;

- Número de *bits* na saída (comprimento das *codewords*): $n = 2^m - 1$;
- Taxa do código: R=k/n;
- Número de *codewords*: $M = 2^m$;
- Distância mínima de Hamming: $d_{mín} = 3$ (sempre);
- Capacidade de correção de erros: [($d_{mín}$-1)] = 1 erro;

Alguns exemplos são apresentados a seguir:

Para m=3:k=4, n=7, r=4/7, M=16 | Código Hamming (7,4);

Para m=4:k=11, n=15, r=11/15, M=2048 | Código Hamming (15,11);

Para m=5:k=26, n=31, r=26/31, M=67.108.864 | Código Hamming (31,26);

A implementação do código corretor de erros de Hamming pode ocorrer em duas categorias: códigos não sistemáticos (os *bits* de paridade estão misturados com os *bits* de dados) ou código sistemático (os *bits* de paridade estão separados dos *bits* de dados). Um exemplo do último é mostrado na tabela da Figura 33 (a) com o código corretor de erros Hamming (7,4), sendo que as mensagens, *u*, aparecem à esquerda e as palavras codificadas, *c*, aparecem à direita.

Como o código da Figura 33 tem que adicionar *m*=3 bits de paridade a cada palavra de entrada de *k* bits, são necessárias três equações de verificação de paridade, são elas:

$$c_5 = c_1 \oplus c_2 \oplus c_3 \quad (1)$$

$$c_6 = c_2 \oplus c_3 \oplus c_4 \quad (2)$$

$$c_7 = c_1 \oplus c_2 \oplus c_4 \quad (3)$$

Uma representação equivalente para essas equações é usando uma matriz chamada de geradora (**G**) e contém *k*=4 linhas (uma para cada equação de verificação de paridade) e *n*=7 colunas. A Figura 3 (c) ilustra a matriz geradora **G** para o código descrito pelas equações anteriores. Como o código é sistemático, **G** pode ser dividida em duas partes: a parte da esquerda, denominada I_k, corresponde aos bits originais da mensagem, através de uma matriz identidade de tamanho *k.* A parte da direita, denominada $\boldsymbol{Q^T}$, corresponde aos bits de dados de tamanho k, portanto, **G** pode ser escrita como $\boldsymbol{G} = [I_k\ Q^T]$.

Entrada	Saída	
$u_1u_2u_3u_4$	$c_1c_2c_3c_4$	$c_5c_6c_7$
0000	0000	000
0001	0001	011
0010	0010	110
0011	0011	101
0100	0100	111
0101	0101	100
0110	0110	001
0111	0111	010
1000	1000	101
1001	1001	110
1010	1010	011
1011	1011	000
1100	1100	010
1101	1101	001
1110	1110	100
1111	1111	111

(a)

$$H = \overbrace{\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}}^{Q} \overbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{I_m}_{m \times n}$$

(b)

$$G = \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{I_k} \overbrace{\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}}^{Q^T}_{k \times n}$$

(c)

Figura 3 – Sistema de codificação C(7,4).

Fonte: Pedroni.

Para determinar se ocorreu erro na transmissão, o decodificador usa uma equação de verificação de paridade, na qual executa operação XOR entre os bits da mensagem recebida e a matriz de verificação de paridade. O resultado desta operação retorna um valor, que pode acusar que não houve erro na transmissão quando a operação retorna zero.

Dependendo do algoritmo implementado e do tamanho do código utilizado, o resultado da operação pode retornar a posição do bit que sofreu erro ou retornar um valor informando que há mais de um erro na transmissão.

2.2.5. OUTROS CÓDIGOS DETECTORES E CORRETORES DE ERROS

Há os códigos de Reed-Muller, estes códigos tiveram uma das maiores aplicações ao enviar dados de fotografia por satélites para a Terra. O objetivo da Mariner 9, em 1971, era sobrevoar o planeta Marte e fotografar sua superfície com uma resolução de 700x832 pixels em escala de cinza, enviando para a Terra através da implementação de códigos corretores. Utilizou-se código de blocos dividindo em blocos

de 6 bits cada, estes blocos foram codificados por um vetor com 26 bits redundantes, totalizando um bloco de 32 bits. Essa implementação ficou conhecida como código de Reed-Muller de ordem 1, cujos vetores tinham tamanho de 64 bits gerados pela interpolação de dois planos de 32 bits.

De acordo com THOMPSON (1983), Desenvolvido por Marcel J. E. Golay em 1949, o código corretor de erros de Golay, é um código linear aplicado em comunicação digital. São divididos em código binário estendido de Golay e código binário perfeito de Golay. O primeiro utiliza codificação de 12 bits de dados em uma palavra de 24 bits, o qual qualquer erro de 3 bits pode ser corrigido e de 7 bits possa ser detectado. O segundo tem uma palavra de 23 bits obtido a partir do anterior, excluindo um bit de paridade, estes possuem parâmetros 23,12,7 correspondendo ao comprimento das palavras codificadas, dimensão do código e a distância mínima de Hamming, respectivamente.

Segundo TAÍSA e OSCAR (2013), os códigos BCH, iniciais dos seus autores (Bose, Chaudhuri e Hocquenghem) são código cíclicos e definidos através de um conjunto de raízes resultantes de seus polinômios geradores, tornando códigos corretores de erros múltiplos, sendo aplicados em comunicação espacial. Para aplicações em sequencias de dados maiores os métodos para obtenção das raízes começam a se tornar inviáveis, no entanto para aplicações menores a técnica de decodificação algébrica faz o uso de algoritmo de Berlekamp-Massey, tornando iterativo para obtenção de seus respectivos polinômios.

Os códigos corretores de Reed Solom (RS), foram criados por Irving S. Reed e Gustave Solomon em 1960, nos laboratório Lincoln do MIT, classificados como linear e cíclicos do tipo RS (n,k), no qual consistem em adicionar informação redundante no sinal para que o receptor possa detectar e corrigir erros ocasionados na transmissão, a palavra código é composta pela mensagem a ser transmitida (n) + bits de paridade (k), sendo sua capacidade de correção de erros relacionada diretamente com a metade do valor dos bits de paridade.

De acordo com GOMES (2003), os códigos Low Density Parity-Check Codes (LDPC) ou códigos definidos por matrizes de paridade esparsas, desenvolvido em 1962 por Robert Gallager, buscava através de matrizes o teste de paridade possuindo

distancias mínimas elevadas. No final da década de 80, Makay e Neal provaram que este código conseguia atingir uma probabilidade de erro próxima ao limite de Shannon.

Esta igualdade será verdade apenas se houver um número ímpar de ‘1s’ em c. Por consequência, qualquer número ímpar de erros será detectado, ao passo que qualquer número par de erros (incluindo zero) fará com que a palavra de código recebida esteja correta.

Outro código de identificação de erros é conhecido como CRC (*Cyclic Redundancy Check*), consiste basicamente em incluir, no final de uma mensagem, uma palavra binária, normalmente com 16 ou 32 bits, denominada valor CRC, a qual é resultado dos cálculos efetuado sobre todo o bloco de dados a ser transmitido. O receptor é responsável por executar os mesmos cálculos nos dados recebidos e então compara o valor CRC resultante com o recebido, este código é usado nos protocolos descritos na Ethernet IEEE 802.3, o qual pode obter mais de 12 kbits de dados seguidos por um CRC de 32 bits.

O cálculo dos *bits* de redundância CRC dependem da quantidade de *bits* de dados a ser transmitido sendo conhecida os polinômios geradores para 8, 16 e 32 *bits*. Para tal, deve ser dividido a cadeias de dados, pelo polinômio gerador g(x), dessa operação resultam q(x) – quociente, r(x) – resto (este resto é o valor CRC).

2.3. CIRCUITOS LÓGICOS PROGRAMÁVEIS

PEDRONI (2010), afirma que os CIs lógicos podem ser divididos em três classes quando refere-se a programabilidade, os CIs não programáveis são circuitos com estrutura interna fixa e nenhuma capacidade de manuseio de software, os CIs programáveis são circuitos com capacidade de manuseio em softwares, ainda que sua estrutura física seja fixa, as tarefas que eles executam podem ser programadas, já os CIs com hardware programável são circuitos com estrutura física programável. Em outras palavras o hardware pode ser modificado, são os casos dos chips CPLD/FPGA.

Pedroni descreve que: “Os PDLs (P*rogrammable Logic Devices*) foram introduzidos em meados da década de 1970. A ideia era construir circuitos combinacionais lógicos que fossem programáveis. Todavia, ao contrário dos microprocessadores, que podem executar um programa, mas possuem *hardware* fixo, a programabilidade dos PLDs se dá no nível do *hardware*. Em outras palavras, um PLD é um *chip* de uso geral cujo *hardware* pode ser configurado para atender às especificações particulares. Os primeiros foram denominados PAL (*Programmable Array Logic*) ou PLA (*Programmable Logic Array*) dependendo do esquema de programação. Eles empregavam somente portas lógicas convencionais (nenhum *Flip-Flop*), portanto visavam apenas a implementação de circuitos combinacionais.” “Para ampliar sua abrangência, em seguida foram lançados PLDs com um *flip-flop* em cada saída do circuito, permitindo assim que funções sequenciais simples também pudessem ser implementadas”.

No início de 1980, circuitos lógicos foram adicionados a cada saída dos PLDs, a nova célula de saída, denominada de macro célula, continha *flip-flop*, portas lógicas e multiplexadores. Essa nova estrutura de PLD foi denominada GAL (*Generic Array Logic*). Nesta mesma década vários dispositivos foram fabricados no mesmo *chip*, tal abordagem ficou conhecida como CPLD (*Complex* PLD). Alguns anos depois foram lançadas as FPGAs (*Field Programmable Gate Arrays*).

Os dispositivos lógicos programáveis dos tipos são não voláteis, enquanto que FPGA são voláteis. Ambos possuem características muito atraentes, como elevado número de portas lógicas e *flip-flop*, larga variedade de I/Os (entradas e saídas) e

tensões de alimentação, grande quantidade de pinos de I/O para usuários, alta velocidade, custo decrescente e tempo curto para lançamento no mercado e capacidade de modificação rápida de produtos desenvolvidos com tais dispositivos.

2.3.1. FPGAS (FIELD PROGRAMMABLE GATE ARRAYS)

Segundo CARDOSO e FERNANDES (2007), o dispositivo FPGA “é um circuito integrado que contém um grande número (na ordem de milhares) de unidades lógicas idênticas.”. Assim é possível configurar e interconectar as unidades lógicas do dispositivo de modo que o circuito opere com a quantidade de hardware definido pelo código criado na linguagem de programação VHDL.

Em meados de 1980, A fabricante Xilinx lançou as FPGA’s com tecnologia superior aos CPLD’s, com arquitetura própria para o processamento avançado de dados. O diagrama da Figura 44, ilustra uma arquitetura geral de uma FPGA, trata-se de uma matriz de blocos ao invés de uma pilha de blocos (conforme os CPLDs), portanto o seu tamanho dos blocos é reduzido, porém são mais sofisticados.

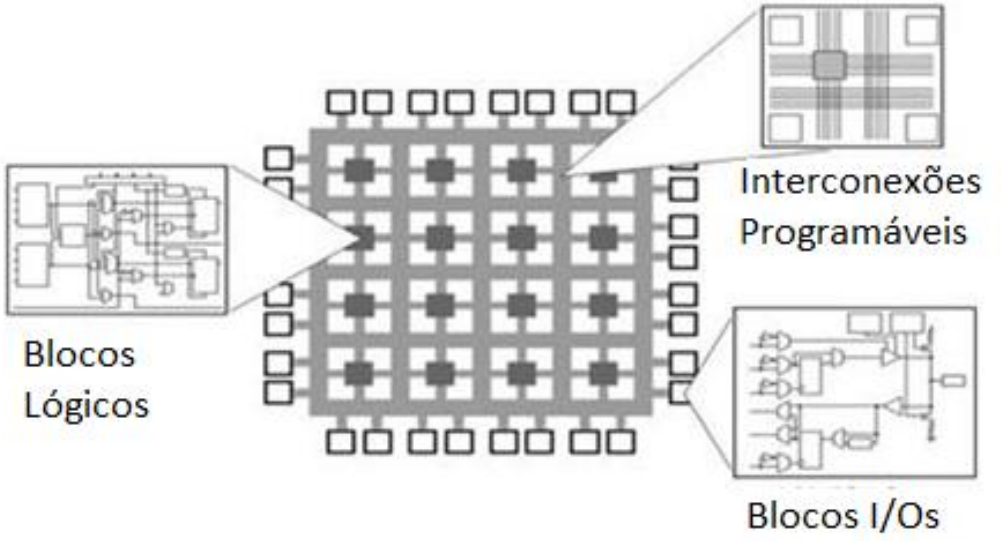

Figura 4 - Arquitetura simplificada de FPGA.

Fonte: http://www.ni.com/cms/images/devzone/tut/figure3-%20fpga_400x212.jpg

Para definir a quantidade de hardware e as ligações das trilhas na FPGA é utilizada a linguagem de programação VHDL, a partir do script desenvolvido é obtido o quantitativo de hardware utilizado e as operações que os hardware precisa executar. As ferramentas para a manipulação dos códigos VHDL, permitem aferir os resultados a partir de testes e simulações, estas ferramentas podem montar os circuitos utilizados e o seu roteamento para a criação de chips menores e com o hardware necessário para o funcionamento do algoritmo criado em VHDL.

3. DESENVOLVIMENTO

O projeto para desenvolver a implementação em hardware de um codificador e decodificador de um código corretor de erros genérico baseado nos critérios de Hamming, partiu da evolução do projeto que comparou a implementação de um código corretor de erros Hamming (72,64) em software (Matlab) e hardware (FPGA). A partir dos resultados obtidos nos códigos de corretores de erros Hamming (72,64) para um código estático foi realizado o presente projeto para tornar o algoritmo implementado em um algoritmo genérico dado as informações necessárias para a codificação e decodificação.

Os cálculos necessários para obtenção da codificação e decodificação seguem os cálculos descritos por Hamming.

A matriz geradora *G* é usada para gerar as palavras código, enquanto H é usada para decodifica-las. O procedimento de codificação/decodificação é ilustrado Figura 5. O codificador recebe a mensagem, *u*, e a converte em uma palavra-código, *c*, usando **G**. A mensagem é submetida a transmissão por um canal ruidoso e o decodificador recebe *c**, que é uma versão possivelmente corrompida de *c*. O decodificador calcula a síndrome *s*, usando a matriz verificadora de paridade **H** (isto é, $s = c^{*}.H^{T}$).

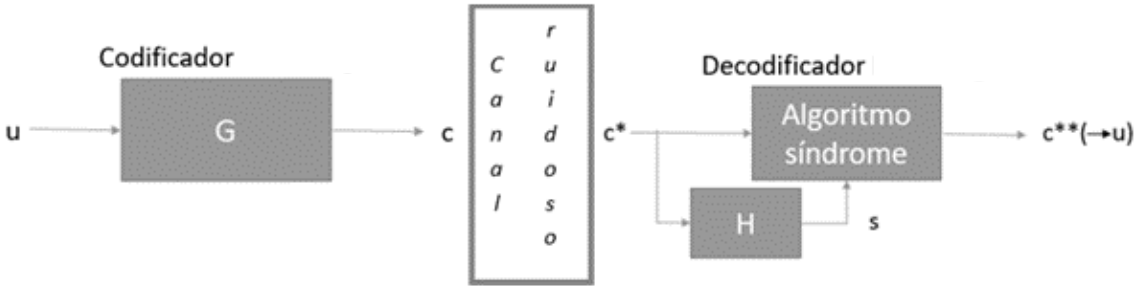

Figura 5 - Procedimento codificação/decodificação.

Fonte: Pedroni (página 141).

Como exemplo, considere que a mensagem *u* = “0111”, após a codificação resultando em *c* = “0111010”, seja transmitida, e suponha que o canal corrompe o quarto bit de *c* (a partir da esquerda), de modo que *c**=”0110010” é recebida. A

síndrome será *s*=”011”, a qual coincide com o conteúdo da quarta coluna de H (Figura 3 (b)), indicando que o quarto bit de *c** deve ser corrigido, resultando, assim *c***=”0111010”. Tomando os quatro primeiros bits de *c***, resulta u=”0111”.

3.1 IMPLEMENTAÇÃO DO CODIFICADOR

O desenvolvimento do codificador implementado em hardware utilizando a linguagem VHDL, é realizado através do software ISE da fabricante Xilinx, com base na placa de desenvolvimento Spartan 3E. A linguagem VHDL permite que o mesmo código possa ser implementado em demais kit FPGA’s de desenvolvimento desde que este possua os requisitos mínimos de hardware utilizados pelo codificador. O algoritmo implementado possui os cálculos necessários para que qualquer tamanho de mensagem possa ser utilizado no codificador, desde que suas matrizes codificadora **G** e verificadora de paridade **H** sejam informadas. A validação do codificador genérico é comprovada utilizando 4 tamanhos de códigos diferentes, são eles: (72,64), (39,32), (15,11) e (7,4).

O codificador Hamming apresentado na Figura 6, é responsável por adicionar os bits de paridade a mensagem original. A adição dos bits de paridade ocorre por meio da multiplicação vetorial da mensagem original pela matriz geradora **G**.

A Figura 6 apresenta o diagrama do codificador do código corretor de erros Hamming, a saída *c* é o resultado da multiplicação entre o vetor de entrada *u* e a matriz geradora **G**, a equação apresentada no codificador segue o modelo apresentado por PEDRONI (2010).

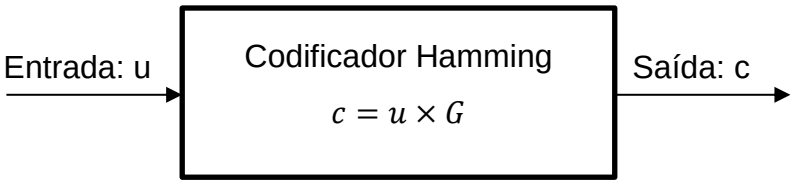

Figura 6 – Diagrama Codificador

Fonte: O autor.

Para que o projeto possa ser genérico as das matrizes codificadora e verificadora de paridade são inseridas em um pacote (package) tornando o código organizado e de fácil interpretação. O arquivo *Codificador.vhd* é o programa principal para o processo de codificação da mensagem *u*, neste arquivo há as variáveis utilizadas e os cálculos matemáticos necessários para a codificação da mensagem de entrada *u*. A saída da palavra-código ou mensagem codificada *c* é disponibilizada na porta de saída do codificador. Os pacotes listados: *Hamming72.vhd*, *Hamming38.vhd*, *Hamming15.vhd* e *Hamming7.vhd*, conforme a Figura 7, possuem as matrizes **G** e **H** para cada código testado.

Ao adicionar um novo tamanho de código e/ou utilizar outro tamanho de código que já possui as matrizes codificadora e verificadora de paridade é necessário alterar 2 parâmetros no arquivo Codificador.vhd. A Figura 8 apresenta parte do código que possui as variáveis *TAM* e *M* que devem ser alterados com os valores do tamanho total do código e com a quantidade de bits de paridade. O pacote com as matrizes deve ser declarado no código principal.

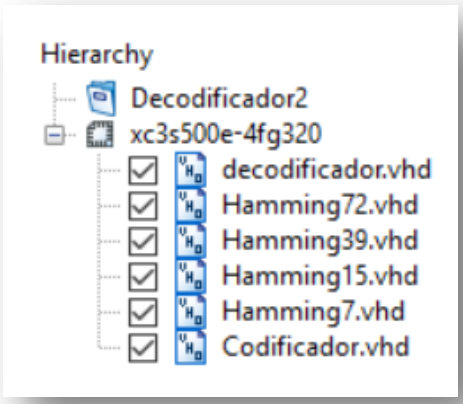

Figura 7 - Estrutura VHDL

Fonte: O autor.

Após a alteração destes parâmetros é necessário realizar a síntese e implementação no software ISE para que o hardware opere com as quantidades corretas de bits nas portas de entrada e saída do codificador.

```
library IEEE;
use ieee.std_logic_1164.all;
use work.Hamming72.all;
entity Codificador is
Generic(
      M : INTEGER := 8;
      TAM : INTEGER := 72
      );
    Port ( u : in  STD_LOGIC_VECTOR(0 to TAM-M-1);
           c : out  STD_LOGIC_VECTOR(0 to TAM-1);
           clk: in STD_LOGIC
           );
end Codificador;
```

Figura 8 - Declaração VHDL – Codificador

Fonte: O autor.

O processo de codificação consiste em realizar a multiplicação da mensagem de entrada, *u*, pela matriz geradora ***G***. As operações para a codificação da mensagem de entrada *u* ocorrem em um processo que depende exclusivamente da mensagem de entrada. Este processo está em paralelo com o processo do clock que é responsável por fazer a atualização da porta de saída do codificador.

```
architecture Behavioral of Codificador is
shared variable S : STD_LOGIC_VECTOR(0 to TAM-1);
begin

process (u)
begin
for i in 0 to TAM-M-1 loop
   S(i) := u(i);
end loop;
for i in TAM-M to TAM-1 loop
   S(i) := u(0) AND Decoder(0)(i);
end loop;
for i in TAM-M to TAM-1 loop
   for j in 1 to TAM-M-1 loop
   S(i) := S(i) XOR (u(j) AND Decoder(j)(i));
   end loop;
end loop;

end process;
process(clk)
begin
c<=S;
end process;
end Behavioral;
```

Figura 9 - Processo VHDL – Codificador.

Fonte: O autor.

A arquitetura da implementação do codificador apresenta a lógica operacional utilizada para o funcionamento correto do componente, a Figura 9 ilustra a arquitetura e a lógica utilizada para definir a implementação em um código corretor de erros genérico. Na Figura 9 há dois processos envolvidos, um que varia com a mensagem de entrada *u*, e outro que varia com o clock do processador.

O processo que depende do clock atualiza o valor da saída do codificador, enquanto o processo que depende da entrada *u* realiza as demais operações logicas para que a mensagem original acrescente os bits de paridade.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package Hamming72 is
constant TAM : INTEGER := 72;
constant M : INTEGER :=8;
type linha is array(0 to M-1) of STD_LOGIC;
type matriz72x8 is array(0 to TAM-1) of linha;
type linha2 is array(0 to TAM-1) of STD_LOGIC;
type matrizG is array(0 to TAM-M-1) of linha2;
constant Hamming : matriz72x8:=(
('1','1','0','0','0','0','0','1'),
('1','0','1','0','0','0','0','1'),
('0','1','1','0','0','0','0','1'),
                ...
('0','0','0','0','0','0','0','1'));
constant Decoder : matrizG;
shared variable L: integer := TAM;
shared variable C: integer := TAM;
for i=0; i < L; i++){
        for (j=0; j < C; j++){
                if(i==j){
                        Decoder(i)(j) = 1;
                } else {
                        Decoder(i)(j) = 0;
                }
        }
}
constant Decoder : matrizG:=(
...

end Hamming72;
```

Figura 10 - Pacote VHDL – Hamming72.

Fonte: O autor.

O pacote com as matrizes geradora **G** e verificadora de paridade **H** estão ilustradas na Figura 10, o tamanho do pacote e seu conteúdo variam de acordo com o tamanho do código que será utilizado.

3.2 IMPLEMENTAÇÃO DO DECODIFICADOR

O componente responsável por realizar a decodificação (decodificador.vhd) tem suas inicializações parecidas com o codificador, apenas com alterações com o tamanho dos dados nas portas de entrada e saída. O decodificador possui uma porta de entrada nomeada *c_rec* para a mensagem codificada com tamanho genérico *n*, e uma porta de saída nomeada *c_decod* com tamanho igual a *k*, conforme ilustrado na Figura 11.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.Hamming72.all;
entity decodificador is
Generic(
        M : INTEGER := 8;
        TAM : INTEGER := 72
        );
Port ( c_rec : in  STD_LOGIC_VECTOR(0 to TAM-1);
        c_decod : out STD_LOGIC_VECTOR(0 to TAM-M-1);
        clk : IN  STD_LOGIC
        );
end decodificador;
```

Figura 11 - Declaração VHDL – Decodificador.

Fonte: O autor.

O decodificador Hamming genérico apresentado na Figura 12 recebe uma mensagem possivelmente corrompida (c*_c) e calcula a síndrome através do cálculo da mensagem recebida com a matriz decodificadora que está no arquivo de configuração.

A arquitetura do decodificador faz o cálculo da síndrome de acordo com a equação, $s = c^{*}.H^{T}$, a partir dos valores da síndrome é possível corrigir um erro na informação se esta mensagem possuir apenas 1 erro.

Figura 12 - Diagrama decodificador Hamming genérico.

Fonte: Própria.

A quantidade de bits presentes na síndrome varia de acordo com a quantidade de bits de paridade que o código implementado possui, o valor da síndrome corresponde ao valor da coluna da matriz **H** da posição do bit que foi corrompido ao passar pelo canal ruidoso. Sendo assim o valor da síndrome é comparado com o valor da coluna da matriz decodificadora, e através dos índices da coluna é identificado qual o bit que sofreu alterações. A correção da mensagem recebida *c** é realizada e a mensagem corrigida *c*** é disponibilizada na porta de saída do decodificador.

O código do decodificador é apresentado na Figura 1313, para aferir o funcionamento do decodificador foi inserido mensagens codificadas com um erro no seu conteúdo, simulando assim uma transmissão no qual a mensagem foi corrompida, a saída do decodificador foi comparada com a mensagem original para aferir sua eficácia quanto a correção.

```
architecture Behavioral of decodificador is
Signal WW: STD_LOGIC_VECTOR(0 TO TAM-M-1);
Signal HTlinha : STD_LOGIC_VECTOR(0 to TAM-1);
shared variable S : STD_LOGIC_VECTOR(0 to M-1);
shared variable Aux : STD_LOGIC_VECTOR(0 to M-1);
shared variable Entrada: STD_LOGIC_VECTOR(0 to TAM-1):=c_rec;
begin
process (c_rec)
variable V00 : STD_LOGIC :='1';
begin
for i in 0 to TAM-1 loop
        Entrada(i) := c_rec(i);
end loop;
for i in 0 to M-1 loop
        S(i) := c_rec(0) AND Hamming(0)(i);
end loop;
for j in 0 to M-1 loop
        for i in 1 to TAM-1 loop
        S(j) := S(j) XOR (c_rec(i) AND Hamming(i)(j));
        end loop;
end loop;
for i in 0 to TAM-1 loop
        for j in 0 to M-1 loop
        Aux(j):= Hamming(i)(j);
        end loop;
        if (M>7) then
                Aux(M-1):=(Aux(M-1) XOR V00);
                if (S = Aux) then
                        Entrada(i) := ((c_rec(i) XOR V00));
                end if;
        else
                if (Aux = S) then
                Entrada(i) := (c_rec(i) XOR V00);
                end if;
        end if;
        HTlinha <= Entrada;
end loop;
end process;
WW <= HTlinha(0 to TAM-M-1);
process(clk)
begin
if(rising_edge(clk)) then
  c_decod<=WW;
end if;
end process;
end Behavioral;
```

Figura 13 - Processo VHDL – Decodificador.

Fonte: O autor.

4. RESULTADOS

Os resultados obtidos com o software Matlab para o código implementado foram comparados aos da referência do autor Pedroni, utilizando os mesmos parâmetros e mensagens.

O código implementado em software (Matlab) poderia ser criado de forma mais direta, visto que o software realiza operações matemáticas de forma direta (sem uso de loop's for) porém o código foi criado de forma complexa para que pudesse ser aproveitado para a implementação do mesmo código em linguagem VHDL.

A fim de validar o codificador e o decodificador com códigos corretores de erros criados, que operaram com códigos de tamanhos diferentes em FPGA, faz-se necessário a utilização da ferramenta Isim que gera estímulos nas portas dos componentes criados na FPGA, estes estímulos podem ser manipulado através de códigos computacionais.

Os estímulos gerados na ferramenta Isim de simulação criam sinais para as portas de entrada, as respostas aos estímulos são observados nas portas de saída, estes sinais podem ser lidos e visualizados no próprio Isim.

Este capítulo apresenta os testes realizados nos componentes codificador e decodificador desenvolvidos em linguagem VHDL, para determinar se os componentes funcionam conforme o esperado, para isto alguns experimentos isolados foram executados.

4.1 CODIFICADOR

A implementação do codificador em software Matlab resultou em uma função que codifica uma mensagem por vez, dentro da função está a matriz geradora **G** e a mensagem a ser codificada é enviada como argumento da função.

Além do código que faz a codificação da mensagem a ser transmitida, foi criado um script que realiza testes com todas as mensagens possíveis para um codificador Hamming (7,4), este código possui uma quantidade menor de dados (7 bits) e há referência do autor Pedroni para aferir a autenticidade dos resultados.

```
Script para teste do codificador Hamming(7,4)
Teste utilizando como matriz geradora:
G=[
1 0 0 0 1 0 1
0 1 0 0 1 1 1
0 0 1 0 1 1 0
0 0 0 1 0 1 1 ]
##Codificador Hamming(7,4)##
Possiveis mensagens:
u=0000
u=0001
u=0010
u=0011
u=0100
u=0110
u=0111
u=1000
u=1001
u=1010
u=1011
u=1100
u=1101
u=1110
u=1111
Mensagens codificadas:
c=0000000
c=0001011
c=0010110
c=0011101
c=0100111
c=0101100
c=0110001
c=0111010
c=1000101
c=1001110
c=1010011
c=1011000
c=1100010
c=1101001
c=1110100
c=1111111
```

Figura 14 - Resultado script teste Hamming (7,4).

Fonte: O autor.

A Figura 14 apresenta os resultados obtidos do script que testa todas as mensagens possíveis para um C (7,4), dada as entradas *u* (mensagens possíveis), o codificador adiciona os bits paridade e retorna as mensagens codificadas *c*.

O correto funcionamento do algoritmo implementado em software pode ser comprovado ao compararmos os resultados da Figura 14 com o que foi apresentado pelo autor Pedroni na Figura 3 do presente projeto. As respostas obtidas através do algoritmo implementado em Matlab são as mesmas apresentadas por Pedroni apresentadas na Figura 3.

O código utilizado para a implementação no Matlab foi utilizado como base para a criação do código em VHDL a ser implementado na FPGA.

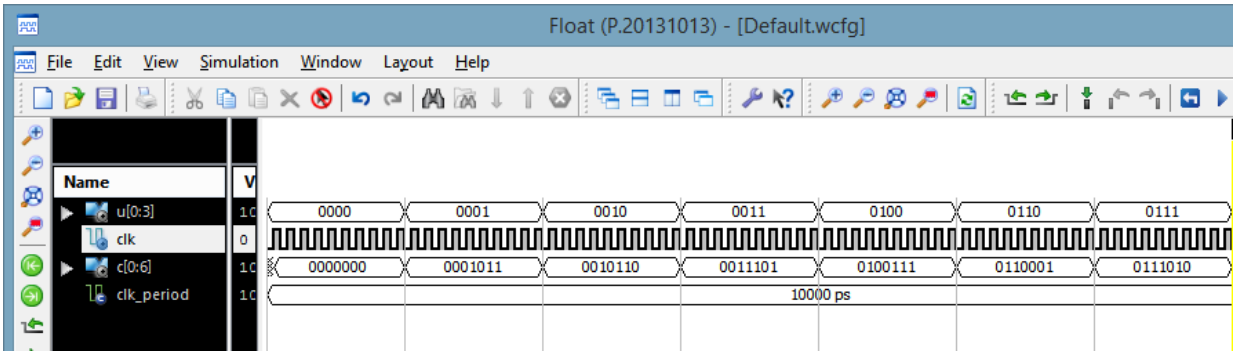

Figura 15 – Resultados codificação em FPGA.

Fonte: O autor.

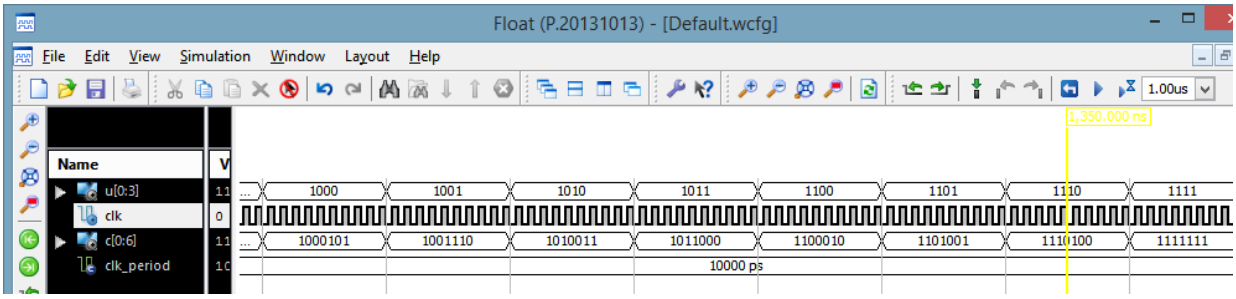

Figura 16 – Resultados codificação em FPGA 2.

Fonte: O autor.

Os resultados da implementação do código em FPGA para um codificador Hamming (7,4) são apresentados nas Figuras 15 e 16. Estas figuras apresentam todas as possibilidades de codificação possíveis para um código com 4 bits de entrada, este é o mesmo exemplo apresentado por Pedroni ilustrado na Figura 3.

O código que foi implementado para a FPGA possui dimensões e cálculos genéricos, que dependem exclusivamente do tamanho do código que será utilizado e das matrizes geradora e verificadora de paridade. A partir destas informações a FPGA consegue realizar as operações matemáticas e codificar qualquer tamanho de mensagem (k bits), desde que informe no código o tamanho de bits de entrada e as matrizes geradora e verificadora de paridade.

Além do teste com o código Hamming (7,4) o algoritmo implementado também foi testado para tamanhos diferentes de código (k bits). Para realizar a verificação do funcionamento do algoritmo para o codificador, foram criados sinais de entrada *u* com tamanhos variados, para cada tamanho utilizado foi informado o tamanho do código e suas matrizes **G** e **H,** os tamanhos utilizados são: 11 *bits*, 32 *bits*, 64 *bits*.

O resultado da codificação destas mensagens é observado na ferramenta Isim e estão apresentados nas figuras abaixo.

Os resultados obtidos para o código Hamming C(72,64) estão apresentados na Figura 17 e Figura 18.

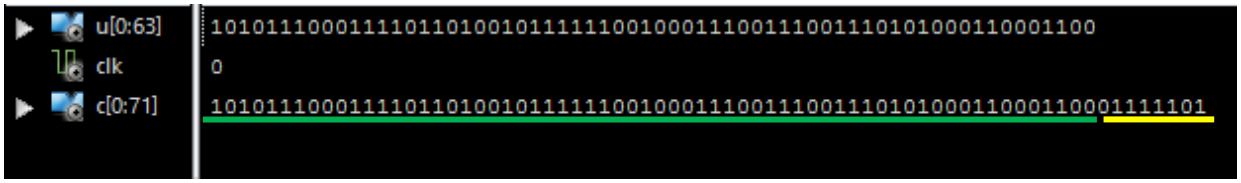

Figura 17 - Resultado codificação testbech, teste1.

Fonte: O autor.

A Figura 17 apresenta a codificação para uma mensagem com 64 bits de dados na entrada *u.* Esta figura apresenta parte da janela da ferramenta Isim e as linhas que apresentam os dados são respectivamente: *u* entrada de dados, *clk* clock do sistema e *c* a mensagem codificada.

Na Figura 17 na linha referente a mensagem codificada sublinhado na cor verde esta evidenciado os bits referente aos dados originais da mensagem *u, o sublinhado* na cor amarelo estão os bits adicionados (bits de paridade).

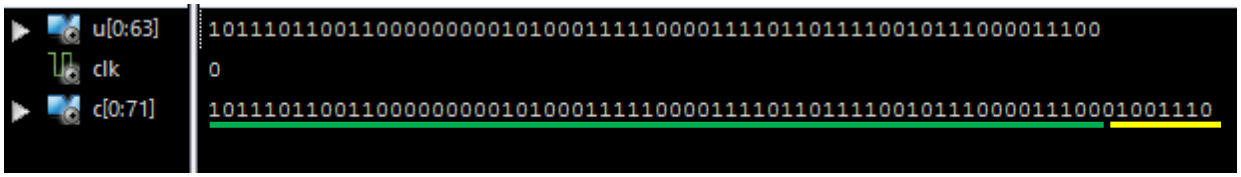

Figura 18 - Resultado codificação testbech, teste2.

Fonte: O autor.

A Figura 18 apresenta a mesma tela da ferramenta Isim, porém há outra mensagem *u* sendo codificada, seguindo as mesmas características de imagem da Figura 16, o sublinhado na cor verde estão os bits referente aos dados originais e o sublinhado na cor amarela estão os bits adicionados (bits de paridade).

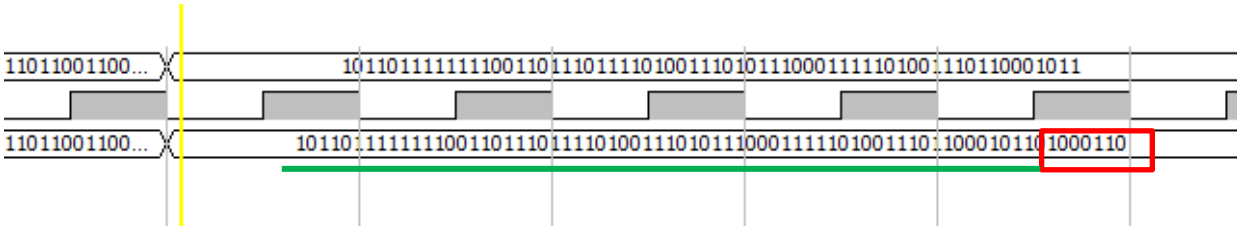

Figura 19 - Resultado codificação testbech, teste2.

Fonte: O autor.

A Figura 19 ilustra os sinais presentes no componente codificador, a primeira linha com dados se refere a mensagem de entrada *u,* a segunda se refere ao clock do sistema e a última linha de dados é a mensagem codificada *c.* Este teste é o mesmo realizado na Figura 17 porém com outra visualização. Os bits de paridade adicionados na mensagem codificados estão evidenciados pelo retângulo em vermelho.

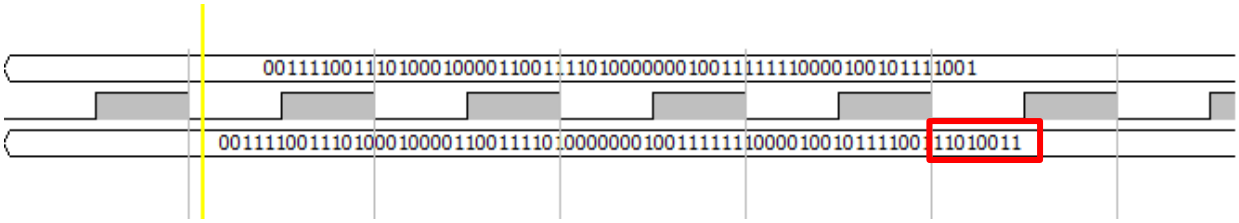

Figura 20 - Resultado codificação testbech, teste 3.

Fonte: O autor.

Outro teste com uma mensagem diferente das demais apresentadas anteriormente porém com o mesmo tamanho c(72,64), é apresentado na Figura 20, os bits de paridade estão evidenciados dentro do retângulo em vermelho.

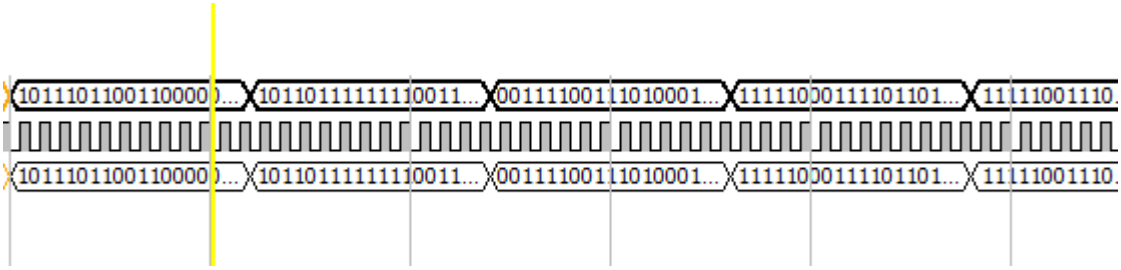

Figura 21 - Resultado codificação testbech, teste3.

Fonte: O autor.

Para múltiplas entradas de mensagens *u* no codificador há múltiplas saídas, e os dados não se misturam, a Figura 21 apresenta os sinais para múltiplas entradas que foram simuladas e as múltiplas saídas, devido ao tamanho da mensagem (72,64) não é possível ler todo o conteúdo de cada mensagem de entrada e codificada.

Após a conclusão do algoritmo do codificador para diferentes mensagens na entrada do componente, foi gerado um teste que faz a leitura de múltiplas mensagens em um arquivo de texto, as codifica e salva em outro arquivo de texto.

Todas as mensagens que foram enviadas para a entrada do Codificador, são codificadas e ficam armazenadas no arquivo de texto, possibilitando consultar os resultados posteriormente, caso seja necessário uma análise das entradas e saídas do codificador.

4.2 DECODIFICADOR

A implementação do decodificador em software Matlab resultou em uma função que decodifica uma mensagem por vez, dentro da função está a matriz verificadora de paridade **H** e a mensagem que será decodificada é enviada como argumento da função.

Assim como no codificador, o decodificador foi testado com todas as possibilidades de mensagens codificadas possíveis para um decodificador Hamming (7,4). Estes testes estão apresentados na Figura 22.

O decodificador Hamming realiza o cálculo da síndrome para determinar se houve falha ou não na mensagem recebida, de acordo com o resultado da síndrome o

algoritmo corrigi a mensagem (quando necessário) e armazena estes resultados em um arquivo texto que pode ser utilizado depois para comparar as mensagens de entrada e saída no decodificador.

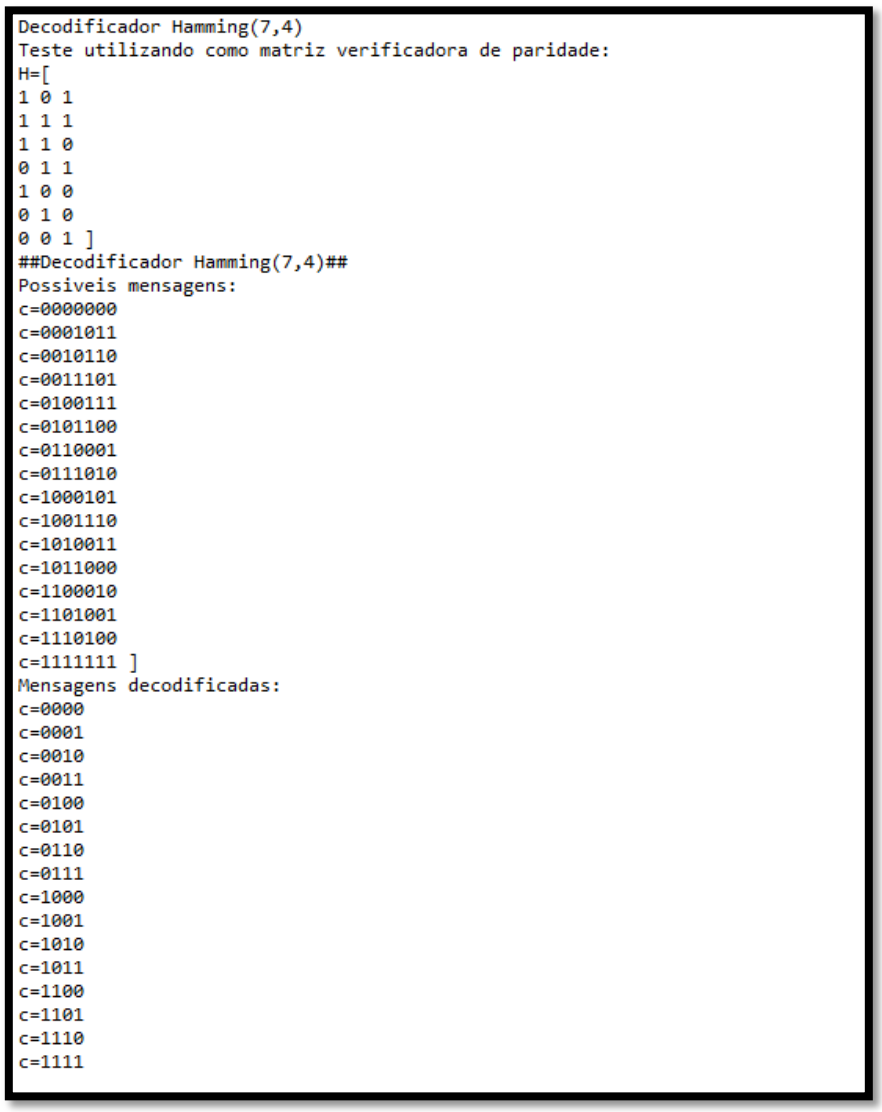

Figura 22 - Resultado script teste decodificador Hamming(7,4).

Fonte: O autor.

Além do teste que verifica todas as possibilidades possíveis de mensagens codificadas sem erros (mensagem codificada é igual a mensagem original), o algoritmo implementado em Matlab foi testado de forma que uma mensagem codificada foi corrompida e recuperada pelo algoritmo.

Para um teste isolado do decodificador é necessário o uso de sinais, ou mensagens codificados que foram corrompidas, se a mensagem chega correta na entrada no decodificador genérico este só remove os bits de paridade e envia para sua porta de saída.

Para corromper a mensagem codificada foi utilizado o software Matlab que lê o arquivo de texto que possuía as mensagens codificadas armazena em memória, altera os dados conforme foi projetado e salva em outro arquivo de texto, como mostra o diagrama da Figura 23.

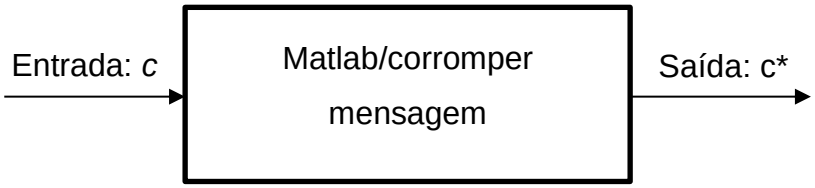

Figura 23 - Diagrama Matlab para corromper a mensagem codificada.

Fonte: O autor.

A corrupção dos dados foi em um bit do código de blocos e foi alternando o erro binário de modo que o cálculo da síndrome detectava o erro e corrigia a mensagem original, este teste está ilustrado na Figura 24.

Para aferir o funcionamento do decodificador considerando a falha em um único bit da mensagem, múltiplas alterações em uma mesma mensagem codificada foi realizado e os resultados estão apresentados na Figura 24. Para a entrada *u=0011* que codificada pelo algoritmo resulta em *c=0011101* o teste corrompeu os bits desta mensagem, um a um, e o decodificador decodificou todas as possibilidades (7 modificações na mensagem *c(7,4)*) corretamente, comprovando a correção de um erro na mensagem.

Figura 24 - Resultado script teste decodificador Hamming(7,4) – teste 2.

Fonte: O autor.

Os erros apresentados foram causados através do software Matlab, mudando os bits de forma ordenada, da esquerda para a direita na mensagem *c.* A codificação da mensagem original *u* é resultado do algoritmo implementado em software apresentado anteriormente.

A implementação do algoritmo em VHDL para a FPGA foi testado de acordo com os testes realizados no software, utilizando as mesmas entradas codificadas. Estes testes podem ser confrontados com os resultados originais apresentados na Figura 3 por Pedroni, os resultados originais apresentam o valor da codificação da mensagem, contudo a decodificação deve resultar no retorno da mensagem codificada *c* em uma mensagem de entrada *u.*

Os resultados obtidos para a decodificação das possíveis mensagens corrompidas apresentados na Figura 24 utilizando a FPGA são apresentados na Figura 25. O teste utilizou os mesmos critérios do teste apresentado na Figura 24 porém utilizou como ferramenta de decodificação das mensagens o hardware FPGA.

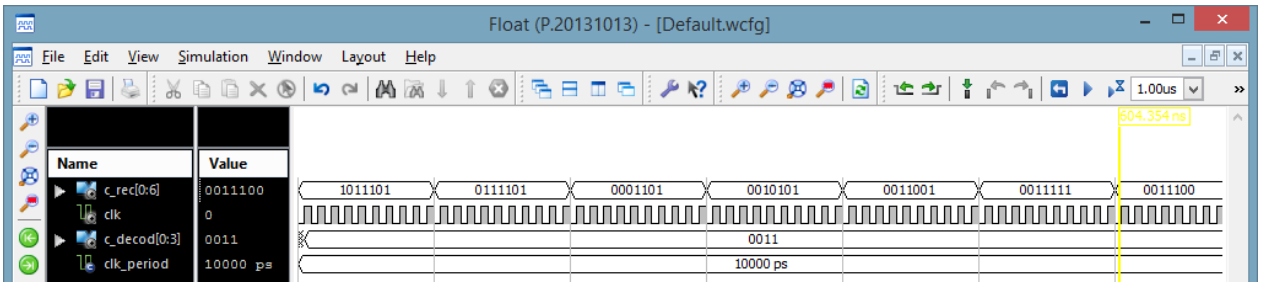

Figura 25 - Resultados decodificação em FPGA para um código (7,4).

Fonte: O autor.

O teste realizado na ferramenta Isim que simula os sinais para a FPGA apresentado na Figura 25 ilustra que as mensagens corrompidas enviadas para a FPGA ao serem decodificadas são corrigidas.

A Figura 25 apresenta 4 linhas de sinais, a primeira linha apresenta as mensagens corrompidas que são enviadas para FPGA (c_rec), a segunda linha é o clock da FPGA, a terceira linha apresentado na Figura 25 é a decodificação das mensagens *c_rec,* as respostas para as mensagens corrompidas retornam na mesma mensagens visto que as entradas remetem a mesma mensagem porém com variações no bit corrompido. A quarta linha é o período do clock da FPGA, através deste é possível verificar que em um ciclo do clock da FPGA as mensagens são decodificadas, a decodificação só ocorre em um ciclo de clock devido a operação em paralelo do hardware.

Afim de aferir o algoritmo para diferentes tamanhos de código corretor de erros, outros tamanhos de mensagem foram enviadas, os resultados da correção das mensagens são apresentados a seguir.

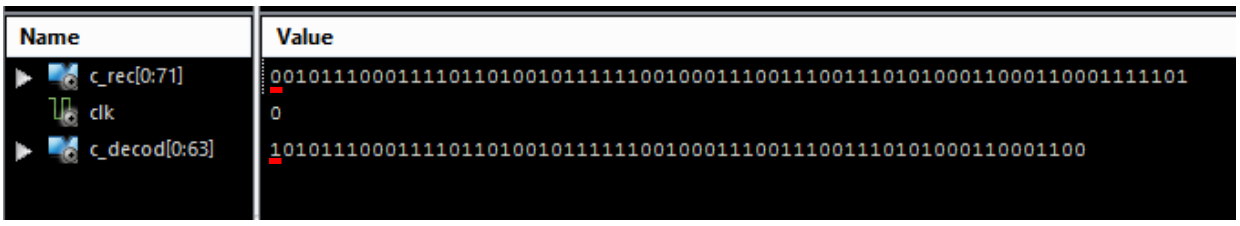

Figura 26 - Resultado decodificação testbech, teste1.

Fonte: O autor.

As Figuras 26 e 27 apresentam os sinais de entrada e saída do decodificador, na cor vermelha está o erro na mensagem original que entra no decodificador, e na saída do decodificador em vermelho está a correção que a mensagem sofreu *c_decod*.

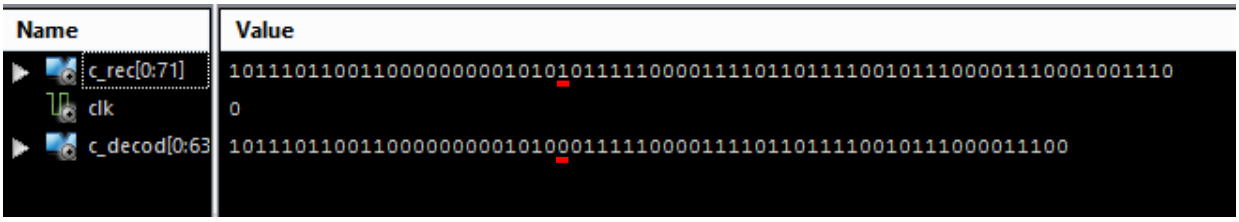

Figura 27 – Resultado decodificação testbech, teste2.

Fonte: O autor.

O teste 2 apresentado no decodificador, envia uma mensagem codificada com um erro (evidenciado na cor vermelha), assim como na Figura 26 e a devido aos bits de paridade que são inseridos a mensagem é corrigida e entregue na porta de saída do decodificador.

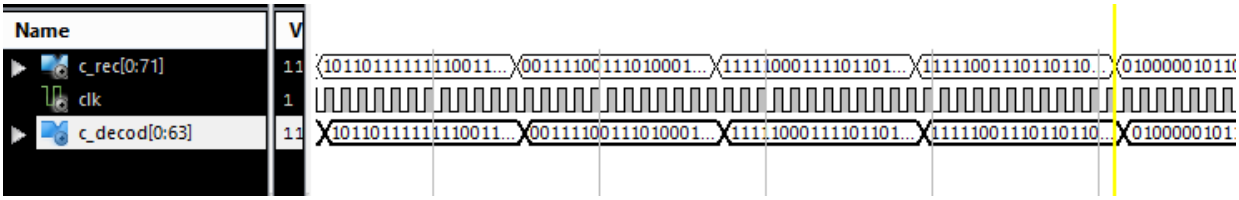

Figura 28 – Resultado decodificação testbech, teste3.

Fonte: O autor.

Assim como no decodificador para o arquivo que foi simulado com múltiplas entradas a Figura 28 apresenta os sinais das portas de entrada e saída que correspondem a *c** e *c*** respectivamente.

4.3 TESTES COM TAMANHOS DE CÓDIGO GENÉRICOS

Para aferir o funcionamento do código quanto aos parâmetros que tornam a implementação em hardware genérica foram realizados testes com outros tamanhos de código de Hamming.

A Figura 29 apresenta resultados para a codificação de um código C (39,32), a Figura 29 apresenta a decodificação da mensagem considerando que há um erro na mensagem recebida no decodificador *c*.*

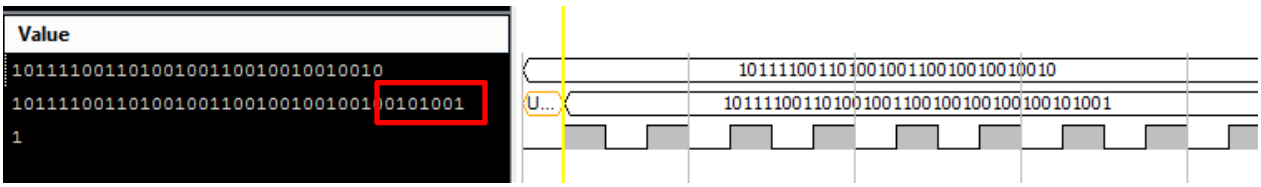

Figura 29 – Resultado codificador C (39,32)

Fonte: O autor.

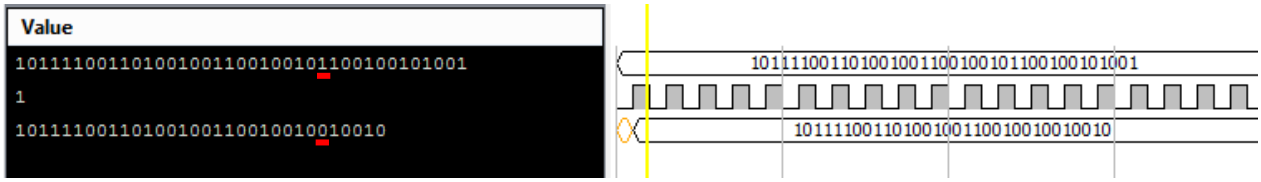

Figura 30 – Resultado decodificador C(39,32)

Fonte: O autor.

Os destaques nas Figuras 29 e 30, apresentam os bits de paridades que foram adicionados e a decodificação da mensagem considerando um bit errado, para um código corretor de erros c (39,32).

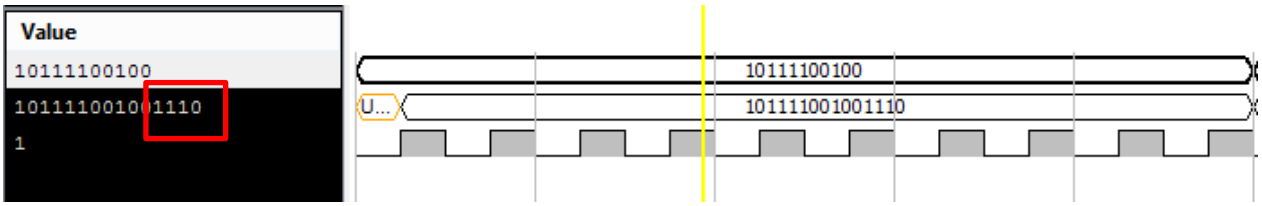

Figura 31 – Resultado codificador C (15,11).

Fonte: O autor.

Outros testes com tamanho diferente é realizado, utilizando o tamanho C (15,11), o resultado para a codificação e decodificação são apresentados nas Figuras 31 e 32.

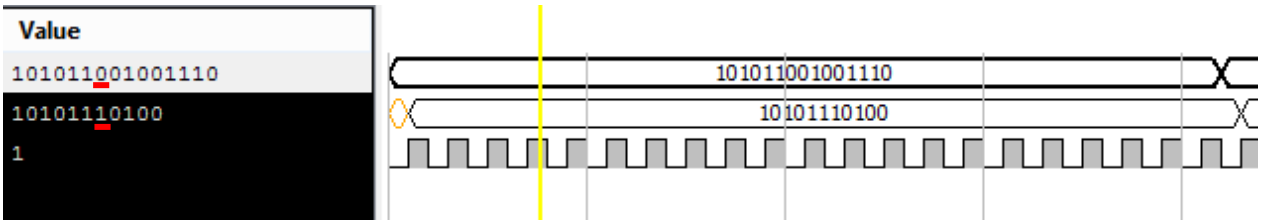

Figura 32 – Resultado decodificador C (15,11)

Fonte: O autor.

Assim como no exemplo do código corretor de erros Hamming (39,32) apresentado anteriormente, as Figuras 31 e 32 apresentam o mesmo teste porém o tamanho do código corretor de erros em questão é o Hamming (15,11).

4.4 TESTE COM SIMULAÇÃO DE MODULAÇÃO E RUÍDO BRANCO

A simulação utilizando técnicas de modulação permitem determinar a relação sinal ruído que o sinal sofre ao passar por um canal com ruído branco limitado a um determinado nível de interferência.

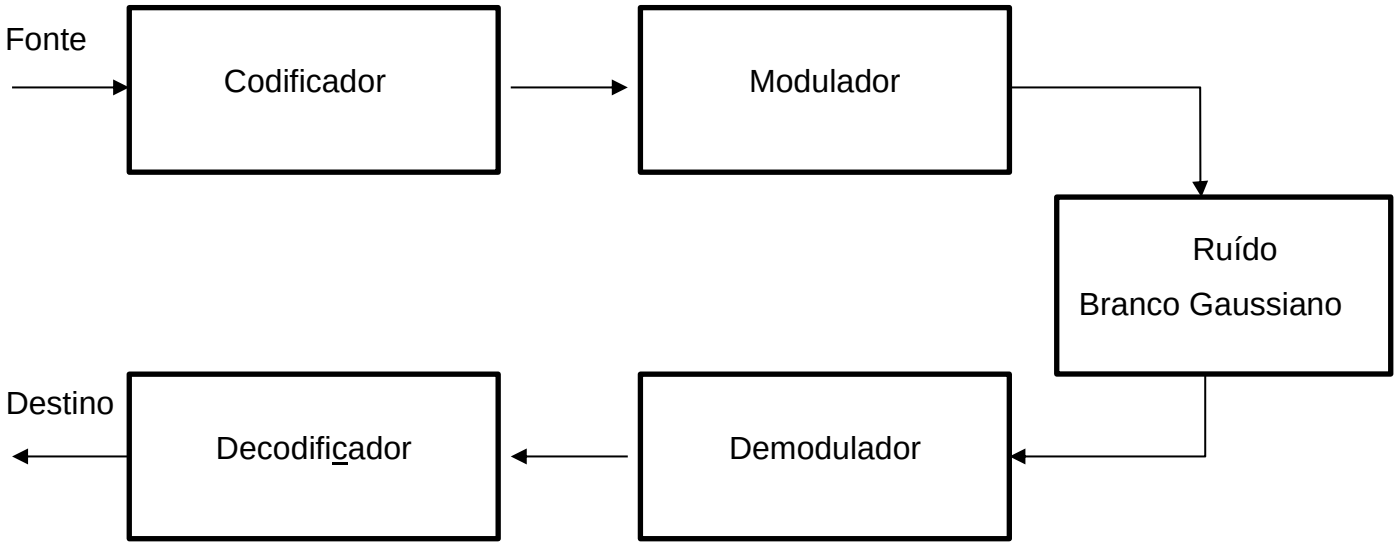

Figura 33 - Diagrama sistema de transmissão.

Fonte: O autor.

O fluxo de transmissão de informação ocorre conforme apresentado no diagrama da Figura 33. Inicialmente a mensagem passa ao codificador, depois passa para o modulador que converte o sinal digital em um sinal analógico e que varia de acordo com o tipo de modulação utilizado.

O ruído adicionado representa a interferência que a mensagem sofre pelos meios de transmissão (ar, cabos, entre outros), seguindo o diagrama, o sinal é demodulado e depois decodificado.

Para o sistema simulado utilizou-se a modulação do tipo PAM (Modulação por Amplitude de Pulso), o canal com ruído é simulado com a função AWGN do Matlab.

As mensagens codificadas estão armazenadas no arquivo em disco. Através do software Matlab as mensagens codificadas são moduladas com o comando

“pammod(x,2)” onde o parâmetro x é a mensagem codificada lida do disco e o parâmetro com valor igual a 2 a quantidade de caracteres diferentes na mensagem. Logo após a modulação da mensagem, é aplicado o ruído com a função AWGN do Matlab, e em seguida é feita a demodulação. Os dados demodulados são armazenados em disco, em um arquivo de texto.

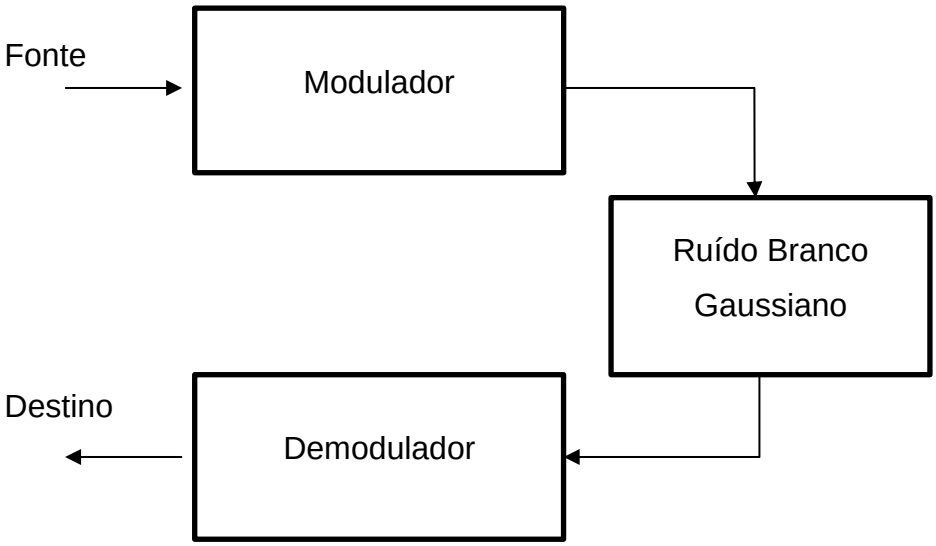

Figura 34 – Sistema de transmissão sem codificador/decodificador.

Fonte: O autor.

Outra simulação foi realizada para comprovar a eficiência do código corretor de erros. Para esta segunda simulação foi adotado o sistema apresentado na Figura 34, onde não há codificador e decodificador. A mensagem é diretamente modulada, inserido o ruído e demodulada.

Através dos resultados obtidos com os procedimentos descritos é possível determinar o gráfico BERxSNR que expressa a quantidade de bits errados em uma mensagem dada um valor especifico de SNR em dB, aplicado a função de ruído.

O gráfico apresentado na Figura 35 mostra estes dois resultados. A curva na cor vermelha refere-se ao sistema de transmissão de informação com um código corretor de erros implementado em hardware para um código (72,64). Os componentes responsáveis pela codificação e decodificação do sistema de transmissão estão

implementados em hardware FPGA e entregam os resultados das mensagens em arquivos de texto.

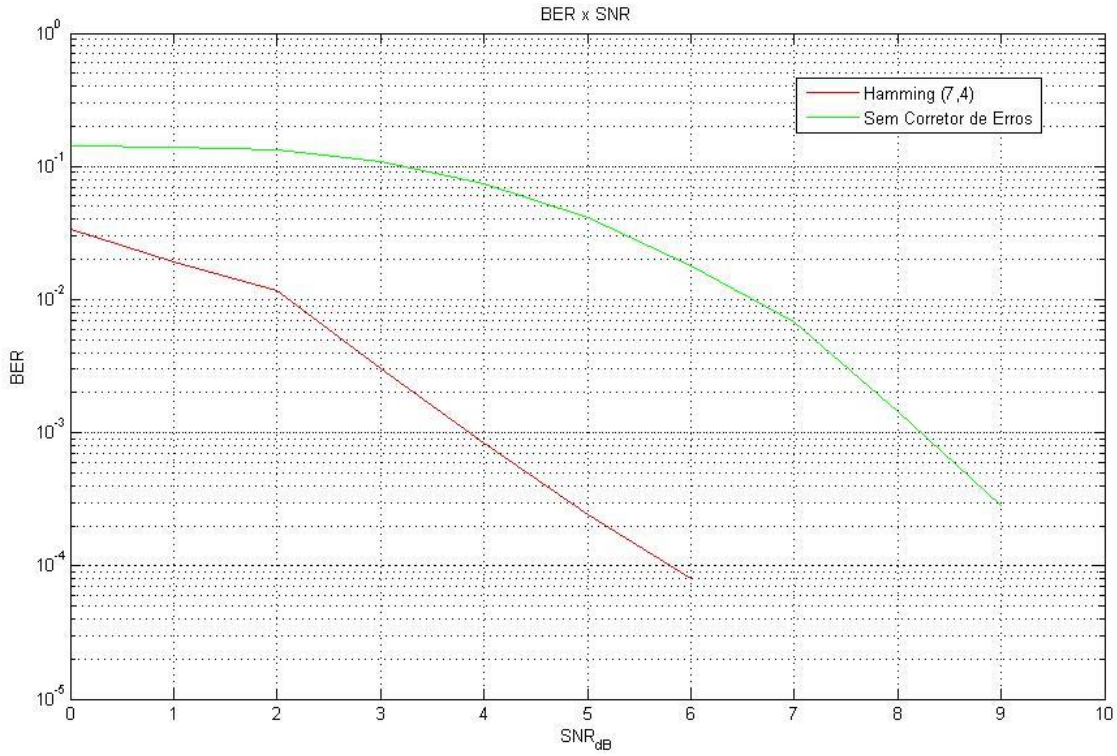

Figura 35 – Relação BER x SNR.

Fonte: O autor.

A curva na cor verde apresentado no gráfico da Figura 35, mostra um sinal que não foi codificado. As mensagens originais foram moduladas e submetidas a um canal ruidoso igual ao utilizado pelo sistema com codificação Hamming. Após o ruído ser aplicado durante a transmissão das mensagens, a mensagem é demodulada e entregue ao destino.

Após comparar as mensagens obtidas no destino com as mensagens originais enviadas, é possível determinar a quantidade de símbolos da mensagem que foram alterados devido ao ruído. Os valores de BER são definidos através do cálculo, $BER = \frac{Quantidade\ simbolos\ errados}{Quantidade\ total\ de\ simbolos\ enviados}$.

Para esta simulação foram enviadas 5000 mensagens, de 4 bits cada. A Figura 35 apresenta os resultados de BER para 10 diferentes níveis de ruído aplicado ao sistema, o ruído varia de 0 a 9 dB em um canal de ruído branco gaussiano.

4.5 ANALISE CONSUMO DE HARDWARE

O consumo de hardware por cada código corretor de erros de Hamming genérico é diferente. Os algoritmos implementados para a codificação e decodificação das mensagens são os mesmos para todo e qualquer tamanho de código corretor de erros, visto que a implementação foi genérica.

Contudo a quantidade de informação contida na matriz geradora **G** e na matriz verificadora de paridade **H** é diferente, pois a quantidade de informação destas matrizes dependem do tamanho do código que está sendo implementado, sendo assim a quantidade de hardware utilizada aumenta à medida que o tamanho do código aumentar.

As Figuras 36, 37 e 38 apresentam o consumo de hardware da FPGA considerando 3 tamanhos de códigos diferentes, Hamming (11,15), Hamming (39,32) e Hamming (72,64) respectivamente.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of 4 input LUTs	8	33,280	1%
Number of occupied Slices	4	16,640	1%
Number of Slices containing only related logic	4	4	100%
Number of Slices containing unrelated logic	0	4	0%
Total Number of 4 input LUTs	8	33,280	1%
Number of bonded IOBs	26	519	5%
Average Fanout of Non-Clock Nets	2.47		

Figura 36 – Consumo de Hardware Hamming (15,11).

Fonte: O autor.

Ao validar as informações do consumo de hardware, é possível verificar que não há uma linearidade de consumo versus tamanho do código implementado, visto que o maior consumo de hardware é devido ao tamanho das matrizes *G* e *H,* e da informação que estas carregam.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of 4 input LUTs	35	33,280	1%	
Number of occupied Slices	20	16,640	1%	
Number of Slices containing only related logic	20	20	100%	
Number of Slices containing unrelated logic	0	20	0%	
Total Number of 4 input LUTs	35	33,280	1%	
Number of bonded IOBs	71	519	13%	
Average Fanout of Non-Clock Nets	2.48			

Figura 37 - Consumo de Hardware Hamming (39,32).

Fonte: O autor.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of 4 input LUTs	82	33,280	1%	
Number of occupied Slices	44	16,640	1%	
Number of Slices containing only related logic	44	44	100%	
Number of Slices containing unrelated logic	0	44	0%	
Total Number of 4 input LUTs	82	33,280	1%	
Number of bonded IOBs	136	519	26%	
Average Fanout of Non-Clock Nets	2.44			

Figura 38 – Consumo de Hardware Hamming (72,64).

Fonte: O autor.

O algoritmo tanto para a codificação quanto para a decodificação, repete o ciclo de cálculos de acordo com o tamanho das matrizes *G e H*. Logo quanto menor as matrizes menor quantidade de hardware irá ocupar e vice e versa.

5. CONCLUSÃO

Neste trabalho buscou-se implementar em hardware um codificador e decodificador para códigos corretores de erros de Hamming de modo que estes operem com tamanhos genéricos de acordo com a necessidade do usuário.

O código desenvolvido por Hamming é sem dúvidas um dos códigos mais aplicados nos sistemas de transmissão devido a sua eficiência. A partir dos parâmetros apresentados por Hamming elaborou-se os algoritmos em VHDL capazes de codificar e decodificar mensagens corrigindo possíveis erros. Provendo uma implementação versátil para execuções de simulações e análises dos códigos corretores de erros em diferentes plataformas que aceitem a linguagem de programação.

As simulações feitas com as mensagens codificadas e decodificadas foram aferidas com a utilização de tamanhos diferentes de códigos corretores de erro aplicados c(72,64), c(39/32), c(15/11). Nota-se que os resultados no MATLAB para o codificador e decodificador são os mesmos do que foi implementados em hardware para o código corretor de erros Hamming (72,64) como já era esperado.

Do ponto de vista operacional e lógica, a partir da linguagem de programação VHDL foi possível a criação de um algoritmo limpo e eficiente, o algoritmo genérico implementado é capaz de codificar e decodificar qualquer tamanho de mensagem desde que o usuário insira as matrizes geradora e verificadora de paridade, há a necessidade de alterar os parâmetros de configuração para que os algoritmos possam operar corretamente.

Nos resultados obtidos no projeto é possível observar a eficiência do código corretor de erros Hamming, que torna a transmissão de dados por canais que possuem ruídos mais integras na entrega ao destinatário. O gráfico apresenta a análise da relação sinal ruído, SNR, pela relação de bits de erros, BER, nos fornecem visualmente informações que tornam as análises mais ricas e objetivas, onde as mensagens que foram transmitidas com codificação e decodificação possuem uma quantidade de erros menor em relação ao sinal enviado sem codificação e decodificação.

Para trabalhos futuros dentro desta mesma área de atuação sugere-se:

- Criar um circuito RF para o envio e recebimento de mensagens utilizando a placa de desenvolvimento FPGA;
- Melhorias no código com intuito de diminuir a quantidade de hardware utilizada pelo FPGA;

6. REFERENCIAS BIBLIOGRAFIAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. ***NBR* 6023: Informação e Documentação: Referências: Elaboração.** Rio de Janeiro, 2002.

BRANTE, G. G. O.; MUNIZ, D. N.; GODOY, W. J. ***Information Set Based Soft-Decoding Algorithm for Block Codes***, 2010.

GODOY, Walter Jr. **Introdução aos Códigos Controladores de Erro, Parte I – Códigos de Bloco.** São Paulo, 2008.

GOMES, M. A. C. G., **Códigos binários definidos por matrizes de teste de paridade esparsas algoritmos de descodificação**. *Disponível em*: <http://home.deec.uc.pt/~marco/Files/MsC_LDPC_Marco_Gomes.pdf>.*Acesso em: 30/11/2016.*

GORTAN, A.; JASINSKI, R. P.;GODOY, W. J.; PEDRONI, V. A., ***Hardware-Friendly Implementation of Soft Information Set Decoders***, 2010.

GORTAN, A.; JASINSKI, R. P.; GODOY, W. J..; PEDRONI V. A., ***Achieving Near-MLD Performance with Soft Information-Set Decoders Implemented in FPGAs***, 2010.

HAYKIN, Simon. ***Communication Systems***. New York: John Wiley & Sons, Inc., 2001.

LIN, Shu e COSTELLO, Daniel J. Jr. **Error Control Coding** – Fundamentals and Applications. New Jersey : Prentice-Hall, 1983.

PEDRONI, Volnei A. **Eletrônica Digital Moderna e VHDL**. Rio de Janeiro: Elsevier, 2010.

SHANNON, C.E., **A Mathematical Theory of Communication**. Disponível em:<http://worrydream.com/refs/Shannon%20-%20A%20Mathematical%20Theory%20of%20Communication.pdf> .Acesso em: 30/11/2016.

THOMPSON, THOMAS M - ***From Error Correcting Codes Through Sphere Packings to Simple Groups***.

WIKIPEDIA , **Turbo código**. Disponível em: <https://es.wikipedia.org/wiki/Turbo_c%C3%B3digo >. Acesso em: 30/11/2016.

Apêndice A

A.1. CÓDIGO MATLAB CODIFICADOR

%%%% Codificador Hamming(72,64) para entrada de dados do tipo em blocos%%%%

```
function y=encoder_Hammin72(c)

%%Matriz Geradorada
%G=[Ik | QT]; 64x72
Q=[
1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1
1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1
0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1
1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
];
QT=Q';
Ik= eye(64);
G1=[Ik QT];
G = (G1);

for a = 1:72
    for j = 1:64
       v(j)=c(j)*G(j,a);
    end
w(a)=xor(v(1),v(2));
    for j = 3:64
       w(a)=xor(w(a),v(j));
    end
end

y=double(w);
```

A.2. CÓDIGO MATLAB DECODIFICADOR

```
%%%% Decodificador Hamming(72,64) para entrada de dados do tipo em blocos%%%%
function [decodificado]=decoder_Hamming72(w)
%matriz decodificadora
% H = [ Q | Ik]



H = [
1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0
1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0
0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1
1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
];
w=logical(w);


HT = H';
Ht = logical(HT);
% %mensagem codificada


for a = 1:8
for j = 1:72
    v(j)=w(j)*HT(j,a);

end
v;
s(a)=xor(v(1),v(2));
    for j = 3:72
       s(a)=xor(s(a),v(j));
    end
end
s;
    if s(1)==true && s(2)==true && s(3)==false && s(4)==false && s(5)==false
&& s(6)==false && s(7)==false && s(8)==false
      if w(1)==true
          w(1)=false;
      elseif w(1)==false
          w(1)=true;
```

```
      end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==false && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(2)==true
          w(2)=false;
      elseif w(2)==false
          w(2)=true;
           end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==false && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(3)==true
            w(3)=false;
      elseif w(3)==false
          w(3)=true;
           end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==false && s(5)==false &&
s(6)==false && s(7)==false && s(8)==true
        if w(4)==true
          w(4)=false;
      elseif w(4)==false
          w(4)=true;
        end
    end
     if s(1)==true && s(2)==false && s(3)==false && s(4)==true && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(5)==true
          w(5)=false;
      elseif w(5)==false
          w(5)=true;
        end
     end
     if s(1)==false && s(2)==true && s(3)==false && s(4)==true && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(6)==true
          w(6)=false;
      elseif w(6)==false
          w(6)=true;
        end
     end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==true && s(5)==false &&
s(6)==false && s(7)==false && s(8)==true
        if w(7)==true
          w(7)=false;
      elseif w(7)==false
          w(7)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==true && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(8)==true
          w(8)=false;
      elseif w(8)==false
          w(8)=true;
```

```
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==true && s(5)==false &&
s(6)==false && s(7)==false && s(8)==true
        if w(9)==true
          w(9)=false;
      elseif w(9)==false
          w(9)=true;
        end
    end
     if s(1)==false && s(2)==true && s(3)==true && s(4)==true && s(5)==false
&& s(6)==false && s(7)==false && s(8)==true
        if w(10)==true
          w(10)=false;
      elseif w(10)==false
          w(10)=true;
        end
     end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==true && s(5)==false &&
s(6)==false && s(7)==false && s(8)==true
        if w(11)==true
          w(11)=false;
      elseif w(11)==false
          w(11)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==false && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(12)==true
          w(12)=false;
      elseif w(12)==false
          w(12)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==false && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(13)==true
          w(13)=false;
      elseif w(13)==false
          w(13)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==false && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(14)==true
          w(14)=false;
      elseif w(14)==false
          w(14)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==false && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(15)==true
          w(15)=false;
      elseif w(15)==false
          w(15)=true;
```

```
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==false && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(16)==true
          w(16)=false;
      elseif w(16)==false
          w(16)=true;
        end
    end
     if s(1)==false && s(2)==true && s(3)==true && s(4)==false && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(17)==true
          w(17)=false;
      elseif w(17)==false
          w(17)=true;
        end
     end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==false && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(18)==true
          w(18)=false;
      elseif w(18)==false
          w(18)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==false && s(4)==true && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(19)==true
          w(19)=false;
      elseif w(19)==false
          w(19)=true;
        end
    end
     if s(1)==true && s(2)==false && s(3)==false && s(4)==true && s(5)==true
&& s(6)==false && s(7)==false && s(8)==true
        if w(20)==true
          w(20)=false;
      elseif w(20)==false
          w(20)=true;
        end
     end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(21)==true
          w(21)=false;
      elseif w(21)==false
          w(21)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(22)==true
          w(22)=false;
      elseif w(22)==false
          w(22)=true;
```

```
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(23)==true
          w(23)=false;
      elseif w(23)==false
          w(23)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(24)==true
          w(24)=false;
      elseif w(24)==false
          w(24)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(25)==true
          w(25)=false;
      elseif w(25)==false
          w(25)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==true && s(5)==true &&
s(6)==false && s(7)==false && s(8)==true
        if w(26)==true
          w(26)=false;
      elseif w(26)==false
          w(26)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(27)==true
          w(27)=false;
      elseif w(27)==false
          w(27)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(28)==true
          w(28)=false;
      elseif w(28)==false
          w(28)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(29)==true
          w(29)=false;
      elseif w(29)==false
          w(29)=true;
```

```
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(30)==true
          w(30)=false;
      elseif w(30)==false
          w(30)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(31)==true
          w(31)=false;
      elseif w(31)==false
          w(31)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==false && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(32)==true
          w(32)=false;
      elseif w(32)==false
          w(32)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==false && s(5)==false &&
s(6)==true && s(7)==false && s(8)==true
        if w(33)==true
          w(33)=false;
      elseif w(33)==false
          w(33)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==false && s(4)==true && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(34)==true
          w(34)=false;
      elseif w(34)==false
          w(34)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==true && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(35)==true
          w(35)=false;
      elseif w(35)==false
          w(35)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==true && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(36)==true
          w(36)=false;
      elseif w(36)==false
          w(36)=true;
```

```
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==true && s(5)==false &&
s(6)==true && s(7)==false && s(8)==true
        if w(37)==true
          w(37)=false;
      elseif w(37)==false
          w(37)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==true && s(5)==false
&& s(6)==true && s(7)==false && s(8)==true
        if w(38)==true
          w(38)=false;
      elseif w(38)==false
          w(38)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==true && s(5)==false &&
s(6)==true && s(7)==false && s(8)==true
        if w(39)==true
          w(39)=false;
      elseif w(39)==false
          w(39)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==true && s(5)==false &&
s(6)==true && s(7)==false && s(8)==true
        if w(40)==true
          w(40)=false;
      elseif w(40)==false
          w(40)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==true && s(5)==false &&
s(6)==true && s(7)==false && s(8)==true
        if w(41)==true
          w(41)=false;
      elseif w(41)==false
          w(41)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==false && s(4)==false && s(5)==true
&& s(6)==true && s(7)==false && s(8)==true
        if w(42)==true
          w(42)=false;
      elseif w(42)==false
          w(42)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==false && s(5)==true
&& s(6)==true && s(7)==false && s(8)==true
        if w(43)==true
          w(43)=false;
      elseif w(43)==false
          w(43)=true;
```

```
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==false && s(5)==true
&& s(6)==true && s(7)==false && s(8)==true
        if w(44)==true
          w(44)=false;
      elseif w(44)==false
          w(44)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==false && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(45)==true
          w(45)=false;
      elseif w(45)==false
          w(45)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==false && s(5)==true
&& s(6)==true && s(7)==false && s(8)==true
        if w(46)==true
          w(46)=false;
      elseif w(46)==false
          w(46)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==false && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(47)==true
          w(47)=false;
      elseif w(47)==false
          w(47)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==false && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(48)==true
          w(48)=false;
      elseif w(48)==false
          w(48)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==false && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(49)==true
          w(49)=false;
      elseif w(49)==false
          w(49)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==false && s(4)==true && s(5)==true
&& s(6)==true && s(7)==false && s(8)==true
        if w(50)==true
          w(50)=false;
      elseif w(50)==false
          w(50)=true;
```

```
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(51)==true
          w(51)=false;
      elseif w(51)==false
          w(51)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(52)==true
          w(52)=false;
      elseif w(52)==false
          w(52)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(53)==true
          w(53)=false;
      elseif w(53)==false
          w(53)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(54)==true
          w(54)=false;
      elseif w(54)==false
          w(54)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(55)==true
          w(55)=false;
      elseif w(55)==false
          w(55)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(56)==true
          w(56)=false;
      elseif w(56)==false
          w(56)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==true && s(5)==true &&
s(6)==true && s(7)==false && s(8)==true
        if w(57)==true
          w(57)=false;
      elseif w(57)==false
          w(57)=true;
```

```
        end
    end
    if s(1)==true && s(2)==false && s(3)==false && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(58)==true
          w(58)=false;
      elseif w(58)==false
          w(58)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==false && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(59)==true
          w(59)=false;
      elseif w(59)==false
          w(59)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==false && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(60)==true
          w(60)=false;
      elseif w(60)==false
          w(60)=true;
        end
    end
    if s(1)==false && s(2)==false && s(3)==true && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(61)==true
          w(61)=false;
      elseif w(61)==false
          w(61)=true;
        end
    end
    if s(1)==true && s(2)==false && s(3)==true && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(62)==true
          w(62)=false;
      elseif w(62)==false
          w(62)=true;
        end
    end
    if s(1)==false && s(2)==true && s(3)==true && s(4)==false && s(5)==false
&& s(6)==false && s(7)==true && s(8)==true
        if w(63)==true
          w(63)=false;
      elseif w(63)==false
          w(63)=true;
        end
    end
    if s(1)==true && s(2)==true && s(3)==true && s(4)==false && s(5)==false &&
s(6)==false && s(7)==true && s(8)==true
        if w(64)==true
          w(64)=false;
      elseif w(64)==false
          w(64)=true;
```

```
    end
end
%Recompoe a mensagem decodificada com os valores alterados quando há
for e = 1:64
    msg_decoded(e) = w(e);
end
decodificado = msg_decoded;
```

A.3. CÓDIGO VHDL CODIFICADOR

```
----------------------------------------------------------------------------
----
-- Company:
-- Engineer: Leandro S. Belli
--
-- Create Date:    16:45:15 09/16/2016
-- Design Name:
-- Module Name:    Codificador - Behavioral
-- Project Name:
-- Target Devices: Spartan 3E
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
----
library IEEE;
use ieee.std_logic_1164.all;
use work.Hamming72.all;
entity Codificador is
Generic(
            M : INTEGER := 8;
            TAM : INTEGER := 72
            );
    Port ( u : in  STD_LOGIC_VECTOR(0 to TAM-M-1);
           c : out  STD_LOGIC_VECTOR(0 to TAM-1);
                    clk: in STD_LOGIC
                    );
end Codificador;

architecture Behavioral of Codificador is
shared variable S : STD_LOGIC_VECTOR(0 to TAM-1);
begin

process (u)
begin
for i in 0 to TAM-M-1 loop
      S(i) := u(i);
end loop;
for i in TAM-M to TAM-1 loop
      S(i) := u(0) AND Decoder(0)(i);
end loop;
for i in TAM-M to TAM-1 loop
      for j in 1 to TAM-M-1 loop
      S(i) := S(i) XOR (u(j) AND Decoder(j)(i));
      end loop;
end loop;
```

```
end process;
process(clk)
begin
c<=S;
end process;
end Behavioral;
```

A.4. CÓDIGO VHDL DECODIFICADOR

```
----------------------------------------------------------------------------
----
-- Company: UFPR
-- Engineer: Leandro S. Belli
--
-- Create Date:    16:23:26 09/25/2016
-- DeSign Name: Decodificador Hamming 72,64
-- Module Name:    decodificador - Behavioral
-- Project Name: FPGA Decoder
-- Target DeviceS: Spartan
-- Tool verSionS: V3.0 - Decoder working - solve one error.
-- DeScription: Decoder uSing Hamming 72,64 uSing the Sindrome and FPGA
--
-- DependencieS:
--
-- ReviSion:
-- ReviSion 0.01 - File Created
-- Additional CommentS:
--
----------------------------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.Hamming72.all;
entity decodificador is
      Generic(
            M : INTEGER := 8;
            TAM : INTEGER := 72
            );
      Port ( c_rec : in  STD_LOGIC_VECTOR(0 to TAM-1);
           c_decod : out STD_LOGIC_VECTOR(0 to TAM-M-1);
                    clk : IN  STD_LOGIC
                    );
end decodificador;
architecture Behavioral of decodificador is
Signal WW: STD_LOGIC_VECTOR(0 TO TAM-M-1);
Signal HTlinha : STD_LOGIC_VECTOR(0 to TAM-1);
shared variable S : STD_LOGIC_VECTOR(0 to M-1);
shared variable Aux : STD_LOGIC_VECTOR(0 to M-1);
shared variable Entrada: STD_LOGIC_VECTOR(0 to TAM-1):=c_rec;
begin
process (c_rec)
variable V00 : STD_LOGIC :='1';
begin
for i in 0 to TAM-1 loop
      Entrada(i) := c_rec(i);
end loop;
for i in 0 to M-1 loop
      S(i) := c_rec(0) AND Hamming(0)(i);
end loop;
for j in 0 to M-1 loop
      for i in 1 to TAM-1 loop
      S(j) := S(j) XOR (c_rec(i) AND Hamming(i)(j));
```

```
      end loop;
end loop;
for i in 0 to TAM-1 loop
      for j in 0 to M-1 loop
      Aux(j):= Hamming(i)(j);
      end loop;
      if (M>7) then
            Aux(M-1):=(Aux(M-1) XOR V00);
            if (S = Aux) then
                  Entrada(i) := ((c_rec(i) XOR V00));
            end if;
      else
            if (Aux = S) then
            Entrada(i) := (c_rec(i) XOR V00);
            end if;
      end if;
      HTlinha <= Entrada;
end loop;
end process;
WW <= HTlinha(0 to TAM-M-1);
process(clk)
begin
if(rising_edge(clk)) then
  c_decod<=WW;
end if;
end process;
end Behavioral;
```