PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MÁRCIO ALMEIDA GAMA

NÚCLEOS IP CORRETORES DE ERROS PARA PROTEÇÃO DE MEMÓRIA EM SoC

Orientador: Prof. Dr. Fabian Luis Vargas

PORTO ALEGRE
2008

MÁRCIO ALMEIDA GAMA

NÚCLEOS IP CORRETORES DE ERROS PARA PROTEÇÃO DE MEMÓRIA EM SoC

Dissertação apresentada como requisito para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Engenharia Elétrica da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Dr. Fabian Luis Vargas

PORTO ALEGRE
2008

Dedicatória

Dedico este trabalho para a minha família, em especial para o meu filho Marcelo.

Agradecimentos

Em primeiro lugar a Deus por ter me concedido a vida, e permitir que hoje eu atinja mais um objetivo nesta caminhada e ao lado de pessoas tão especiais.

Ao orientador, Dr. Fabian Vargas, pela orientação, dedicação, atenção e apoio científico.

Aos membros da minha Comissão de Avaliação, Dr. Avelino Francisco Zorzo e Dr. Luis Fernando Alves Pereira.

Aos colegas do laboratório SISC, em especial a minha colega Cláudia que sempre me auxiliou e me motivou juntamente com o Marlon.

Aos funcionários do Laboratório LEP da PUCRS.

Ao colega Luis Vitório pela ajuda dada no início deste trabalho.

Aos colegas do SERPRO e da ECT.

Aos meus pais, Paulo e Vera, a minha irmã Bianca, meu vô Balbino, a minha vó Ivaema e o meu tio Ni.

A todos meus amigos que ajudaram nesta caminhada.

E especialmente para a minha esposa Cleonice por nunca permitir que eu desanimasse, por sempre confiar em mim, pelo apoio irrestrito, compreensão, consideração, amor, amizade, dedicação, paciência.

E por fim, ao meu querido filho Marcelo por esperar por mim para brincar. Agora o papai pode abrir a porta e brincar “muito, muito sim” porque não tem mais mestrado.

Muito Obrigado.

Resumo

O constante avanço no processo de fabricação de circuitos integrados tem reduzido drasticamente a geometria dos transistores e os níveis das tensões de alimentação. Em circuitos de alta densidade operando a baixa tensão, as células de memória são capazes de armazenar informação com menos capacitância, o que significa que menos carga ou corrente é necessária para armazenar os mesmos dados. Durante o período de armazenamento, os dados envolvidos estão suscetíveis a sofrerem influência de meio, tais como interferências eletromagnéticas, radiações ou até mesmo falhas do próprio *hardware* envolvido. A falha é caracterizada como uma inversão de um ou mais *bits* de um dado armazenado na memória. Conseqüentemente, os dados poderão apresentar falhas, que provocarão erros e comprometerão a utilização destes dados.

Uma forma de resolução destes problemas é a utilização de Códigos Corretores de Erros. Um Código Corretor de Erros é, em essência, um modo organizado de acrescentar algum dado adicional a cada informação que se queira armazenar e que permita, ao recuperarmos a mesma, detectar e corrigir os erros encontrados.

A maioria dos Códigos Corretores de Erro em uso são desenvolvidos para corrigirem erros aleatórios, isto é, erros que ocorrem de maneira independente da localização de outros erros. Contudo, em muitas situações, os erros podem aparecer em rajadas. De uma maneira geral, Códigos Corretores de Erros aleatórios não se constituem na forma mais adequada e eficiente para correção de erros em rajadas, e a recíproca também é verdadeira. Dos vários métodos propostos pela literatura, para corrigirmos simultaneamente estes dois tipos de erros, o mais efetivo é o Embaralhamento.

O Embaralhador é um algoritmo, um método que pode ser implementado tanto em *hardware* quanto em *software*. É essencialmente constituído por um reordenamento dos *bits* e é executado anteriormente ao armazenamento em memória (Embaralhador) e na leitura, os *bits* são novamente reordenados, ou seja, são colocados novamente em sua posição original (Desembaralhador). Isto provoca um aumento na taxa de detecção e correção destes erros, uma vez que se houver uma interferência concentrada (rajada de erros) em uma memória, por exemplo, durante o armazenamento, na operação de leitura, ao se fazer o desembaralhamento, os erros ficam expostos de forma distribuída, aparecendo como erros aleatórios ao decodificador.

Esta dissertação apresenta uma proposta que combina a utilização de Códigos de Detecção e Correção de erros amplamente referenciados na literatura (*Hamming*, *Hamming* Estendido, *Reed-Muller* e *Matrix*) associados à técnica de Embaralhamento aplicada a *Hardware*, com o objetivo de aumentar a capacidade de detecção e correção de erros em rajada (erros concentrados). A execução dos testes de injeção de falhas do tipo *bit-flip*, aplicadas às técnicas corretoras de erros utilizadas nesta dissertação, mostraram que com a associação da técnica de Embaralhamento as mesmas passaram a ser eficientes também para erros em rajadas.

Abstract

The constant technology process improvement has remarkably reduced the transistor geometry and power supply levels in the integrated circuits. In high-density circuits operating at low voltage, the memory cells are able to store information with less capacitance, which means that less charge or current is required to store the same data. During the storage period, the data involved are likely to suffer influence of media, such as electromagnetic interference, radiation or even failures of the hardware involved. The fault is characterized as a reversal of one or more bits of data stored in a memory. Consequently, the data might fail, leading to mistakes in the use of these data.

One way of solving these problems is the use of error correction codes. An error correction code is, in essence, an organized way to add something extra to every information that you want to store, allowing, the recovery of the same information, detecting and correcting any errors found.

Most error correction codes in use are designed to correct random errors, that is, errors that occur independently of the location of other errors. However, in many situations, errors can occur in bursts. Generally, random error correction codes are not efficient for correction of errors in burst, and the reciprocal is also true. From the various methods proposed in the literature for rectifying these two types of errors, the most effective is interleaving.

The interleaving is a method that can be implemented both in hardware and in software. This method is mainly made up of a reordering of the bits and runs earlier in the storage memory (interleaver) and in reading, the bits are reordered again, that is, they are placed back into its original position (deinterleaver). This causes an increase in the rate of detection and correction of these errors, because if there is a concentrated interference (burst errors) in a memory, for example, during storage, in the operation of reading, to getting the deinterleaving, errors are exposed in a distributed manner, appearing as random errors to the decoder.

This dissertation presents a proposal that combines the use of Error Detection And Correction Codes widely referenced in literature (Hamming, Extended Hamming, Reed-Muller and Matrix) associated with the technique of interleaving applied to hardware, aiming to increase the capacity of detection and correction of burst errors (Concentrated errors). The implementation of bit-flip testing failures, applied to the error correction techniques, showed that association these techniques have been effective also for burst errors.

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

ASIC – *Application Specific Integrated Circuit*

AWGN – *Additive White Gaussian Noise*

DVD – *Digital Versatile Disc*

CI – Circuito Integrado

CMOS – *Complementary Metal-Oxide-Semiconductor*

DED – *Double Error Detection*

EDAC – *Error Detection and Correction*

FIFO – *First In First Out*

FPGA – *Field Programmable Gate Array*

IP – *Intellectual Property*

MBUs – *Multiple Bits Upsets*

MOS – *Metal Oxide Semiconductor*

MOSFET – *Metal-Oxide Semiconductor Field Effect Transistor*

MTBF – *Mean Time Between Failure*

MTTF – *Mean Time to Failure*

MTTR – *Mean Time to Repair*

NMOS – *Negative Metal Oxide Semiconductor*

PCB – *Printed Circuit Board*

RAM – *Random Access Memory*

SECDED – *Single Error Correction and Double Error Detection*

SECSED – *Single Error Correction and Single Error Detection*

SEU – *Single Event Upset*

SoC – *Systems-on-Chip*

SOI – *Silicon on Insulator*

SRAM – *Static Random Access Memory*

TED – *Triple Error Detection*

VHDL – VHSIC *Hardware Description Language*

VHSIC – *Very High Speed Integrated Circuits*

VLSI – *Very Large Scale Integration*

Sumário

Parte I - Fundamentos Teóricos

1. Introdução

1.1. Motivação

Tolerância a falhas em dispositivos semicondutores tem sido um assunto significativo desde o momento em que as primeiras falhas foram observadas em aplicações espaciais, há vários anos atrás. Desde então, o interesse em estudar técnicas tolerantes a falhas, a fim de manter circuitos integrados operando em ambientes hostis aumentou, impulsionado por todas as aplicações possíveis de se utilizar circuitos tolerantes a radiação, como missões espaciais, satélites, experimentos físicos de alta energia e outros [32]. Sistemas aeroespaciais incluem uma grande variedade de circuitos analógicos e digitais, elementos que são potencialmente sensíveis à radiação e devem ser protegidos ou, pelo menos qualificados para as operações aeroespaciais.

De acordo com [45], mais de 70 % da área de um moderno SoC (*Systems-on-Chip*) é ocupado por memórias, e em 2014 poderão ocupar até 94% da área total de um SoC. Nos atuais microprocessadores, mais de 60 % da área do *chip* é ocupado por *caches*. O constante avanço no processo de fabricação de circuitos integrados tem reduzido drasticamente a geometria dos transistores e os níveis das tensões de alimentação. Em circuitos de alta densidade operando a baixa tensão, as células de memória são capazes de armazenar informação com menos capacitância, o que significa que menos carga (ou corrente) é necessária para armazenar os mesmos dados. Infelizmente, uma conseqüência direta é o aumento da vulnerabilidade do dispositivo à radiação, pois partículas energizadas que eram desconsideradas, agora podem produzir falhas [33]. A falha é caracterizada como uma inversão de um ou mais *bits* de um dado armazenado na memória. Quando uma única partícula energizada acerta o silício, perde sua energia, resultando um denso caminho ionizado na região. A ionização causa um pulso de corrente transiente. Esse efeito é chamado de *Single Event Upset* (SEU) ou falha transiente.

Embora *Single Event Upset* tenha sido de maior interesse até o momento, falhas transientes múltiplas, também chamadas de *Multiple Bits Upsets* (MBUs), começam a ser importantes por causa dos tamanhos nanométricos dos transistores nas tecnologias atuais. Este

fato se observa não somente em aplicações espaciais, mas começa a se observar também ao nível do mar. Quando um único íon de alta energia passa pelo silício, pode energizar duas ou mais células de memória adjacentes [34]. MBUs podem ser induzidas por ionização direta ou por influência nuclear dos outros átomos (ionização secundária). Experimentos em memórias sob altos fluxos de prótons e íons mostraram que a probabilidade de falhas múltiplas provocadas por um único íon vem aumentando ao longo dos anos [35] [36] [37].

Este processo de evolução dos circuitos integrados atingirá um ponto onde será inviável o funcionamento de circuitos integrados livres destes efeitos. Prevê-se que nêutrons produzidos pela atividade solar irão afetar drasticamente o funcionamento dos futuros CI's (Circuitos Integrados). Ao nível do mar, a energia dessas partículas não é forte o suficiente para afetar drasticamente o funcionamento dos atuais CI's. Mas à medida que evoluímos para tecnologias inferiores a 0.1µm e a tensões de alimentação inferiores a 1 V, as taxas de erros aleatórios induzidos por nêutrons cósmicos será inaceitável mesmo ao nível do mar [38]. A necessidade de proteger circuitos integrados contra esses problemas torna-se eminente [33].

Neste cenário, aplicações terrestres que são consideradas como críticas, tais como servidores de bancos, servidores de telecomunicações, equipamentos médicos ou mesmo a aviação comercial, forçam cada vez mais o desenvolvimento de técnicas tolerantes a falhas para garantir a confiabilidade.

Para isso, várias técnicas são usadas atualmente para proteger componentes de memória contra falhas transientes múltiplas (MBU). Elas são baseadas em processos específicos de tecnologia, como *Silicon on Insulator* (SOI); ou baseadas em projeto, tais como a substituição de cada célula convencional de memória por células robustas; ou técnicas de monitoração de corrente para detectar falha [39] [40] [41] [42] [43]; ou ainda técnicas que usam códigos de detecção e correção de erros (EDAC) [44]. Cada técnica tem suas vantagens e desvantagens, e há sempre um compromisso entre custo, área, desempenho, potência e confiabilidade.

1.2. Objetivos

O objetivo deste trabalho é o desenvolvimento de núcleos IP (*Intellectual Property*) através da descrição de códigos de detecção e correção de erros em VHDL para proteger dados armazenados em memórias RAM (*Random Access Memory*). Estes núcleos IP implementam códigos de detecção e correção de erros (*Error detection and correction* –

EDAC) associados a técnica de embaralhamento de *bits* (*interleaving*) com a intenção de dar um poder maior de detecção e correção de erros em rajadas (erros múltiplos em seqüência) nas informações armazenadas em memórias RAM. Os EDAC's desenvolvidos servem tanto para correção de erros simples quanto para erros em rajadas na memória. Os EDAC's serão validados através de simulação em ambiente ModelSim. A Figura 1.1 apresenta um modelo genérico de SoC que integra o Núcleo IP para correção de erros em memórias RAM.

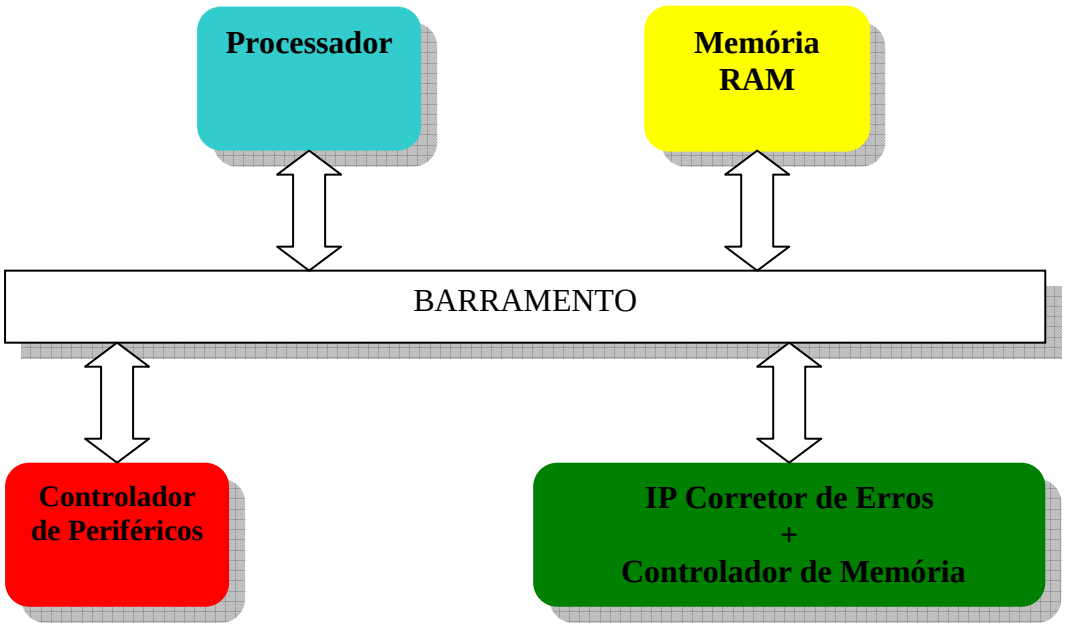


Figura 1.1 – Arquitetura de um SoC Genérico e o Núcleo IP Corretor de Erros proposto.

Todos os processos de escrita e leitura realizados na memória RAM pelo processador devem passar pelo Núcleo IP de correção de erros em memórias RAM + Controlador de Memória. A Figura 1.2 apresenta a estrutura do bloco que agrega o Núcleo IP para correção de erros em memórias RAM e o Controlador de Memória.

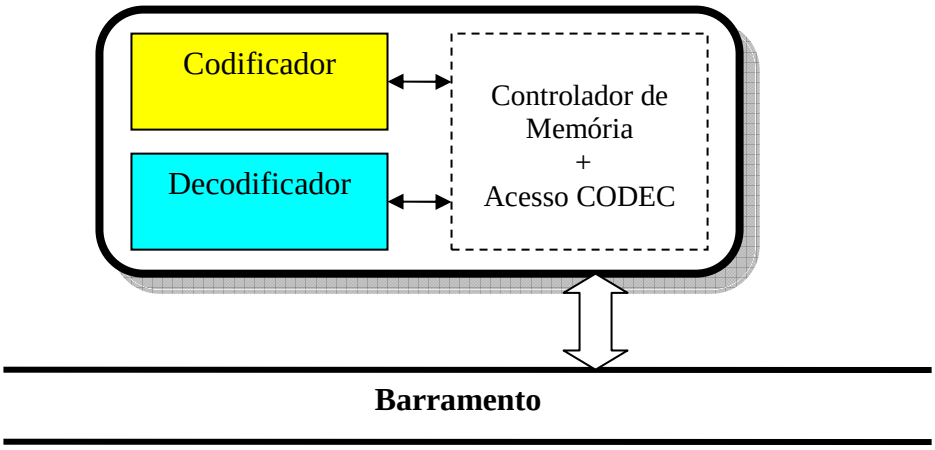


Figura 1.2 – Estrutura Núcleo IP para detecção e correção de erros + controlador de memória.

Basicamente, cada bloco da Figura 1.2 possui as seguintes funções:

- **Codificador**: é responsável por gerar as Palavras-Código das informações recebidas do Controlador de Memória;
- **Decodificador**: é responsável por gerar as informações a partir das Palavras-Código recebidas do Controlador de Memória;
- **Controlador de Memória**: é responsável pela troca de informações entre o processador, a memória, o codificador e decodificador.

O processo de acesso a memória realizado pelo processador é executado da seguinte forma:

Tabela 1.1 – Processo de leitura e escrita na memória.

Processo para Escrita na Memória	**Processo para Leitura na Memória**
1) O processador sinaliza que irá executar um processo de escrita na memória e informa o endereço de memória e a informação a ser armazenada;	1) O processador sinaliza que irá executar um processo de leitura na memória e informa o endereço de memória a ser lido;
2) O Controlador de Memória recebe a informação e repassa ao codificador que gera a palavra-Código e devolve para o controlador de Memória;	2) O Controlador de Memória recebe a posição de memória do processador e faz a leitura na posição (ou posições) correspondente(s) na memória RAM;
3) Essa Palavra-Código é armazenada na memória RAM pelo Controlador de Memória;	3) O Controlador de Memória recebe a Palavra-Código armazenada e repassa ao decodificador que extrai a informação e devolve para o Controlador de Memória;
4) O Controlador de Memória associa o endereço (ou endereços) da memória RAM com o endereço de memória enviado pelo processador. Isto é necessário porque o processador trabalha com 32 *bits* e as palavras-código geradas são superiores a 32 *bits*, necessitando um número maior de posições de memória para realizar o armazenamento.	4) O Controlador de Memória repassa a informação para o processador.

2. Conceitos de Tolerância a Falhas

2.1. Introdução

Tolerância a falhas é a habilidade de um circuito e/ou sistema continuar a execução correta das suas tarefas (sem degradação de desempenho), mesmo diante da ocorrência de falhas em seu *hardware* e/ou *software* [1] evitando assim prejuízos físicos e materiais.

A seguir, serão abordados os conceitos clássicos de tolerância a falhas. Serão apresentados alguns conceitos formais retirados da literatura [1] [2] [3] [4] [5]:

1) **Falha:** podem ocorrer tanto no âmbito de *hardware* quanto de *software*, sendo esta a causa do erro. Componentes envelhecidos e interferências externas são exemplos de fatores que podem levar o sistema à ocorrência de falhas. As falhas de *hardware* podem ser classificadas em permanentes, transientes e intermitentes:
    a) **Falhas Permanentes:** ocorrem no meio físico e são provocadas através de falhas no processo de fabricação e/ou pelo envelhecimento dos componentes do sistema; curtos circuitos, nós abertos e *stuck-at* são exemplos de falhas permanentes.
    b) **Falhas Transientes:** ocorrem durante a vida útil dos componentes e são provocadas por adversidades e/ou fenômenos ambientais aleatórios onde o sistema está implementado; variações de tensão de alimentação e interferências eletromagnéticas são exemplos de falhas transientes.
    c) **Falhas Intermitentes:** é caracterizada pela ocorrência temporária e cíclica do erro a partir de variações das condições externas e/ou ambientais do sistema; vibrações e variações da temperatura são exemplos de falhas intermitentes.
2) ***Stuck-at*:** tipo de falha permanente onde um nó do circuito está sempre no mesmo nível lógico seja ele zero (*stuck-at-zero*) ou um (*stuck-at-one*).
3) ***Bit flips***: tipo de falha transiente ocasionada por interferências externas, que resultam em uma mudança temporária no nível lógico em um determinado elemento de memória do

circuito. Esta mudança pode ocorrer em ambos os sentidos de zero para um ou de um para zero.

4) **Latência:** período de tempo medido desde a ocorrência da falha até a manifestação da mesma.
5) **Dependabilidade:** esse termo é uma tradução literal do termo inglês *dependability*, que indica a qualidade e a confiança depositada no serviço fornecido por um dado sistema. Confiabilidade e disponibilidade são dois dos principais atributos da dependabilidade.
6) **Confiabilidade:** capacidade de operar corretamente dentro de condições definidas durante certo período de funcionamento e estar operacional no início desse período.
7) **Disponibilidade:** é a probabilidade de o sistema estar operacional quando a utilização deste for necessária.

2.2. Falha, Erro ou Defeito

Estamos interessados que um determinado sistema atenda satisfatoriamente as suas especificações de projeto e as necessidades dos seus usuários. Um defeito (do inglês, *failure*) é definido como um desvio da especificação. Define-se que um sistema está em estado errôneo, ou em erro, se o processamento posterior a partir deste estado pode levá-lo a um defeito. Finalmente define-se falha ou falta (do inglês, *fault*) como a causa física ou algorítmica do erro[1].

Falhas são inevitáveis, pois componentes físicos envelhecem e/ou sofrem com interferências externas, sejam ambientais ou humanas, projetos de *software* e *hardware* são vítimas de sua alta complexidade e também da fragilidade humana em trabalhar com grande volume de detalhes ou ainda com a deficiência de especificações.

A Figura 2. 1 apresenta uma simplificação, sugerida por Dhiraj K. Pradhan [3], para os conceitos de falha, erro e defeito. Nela, as falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

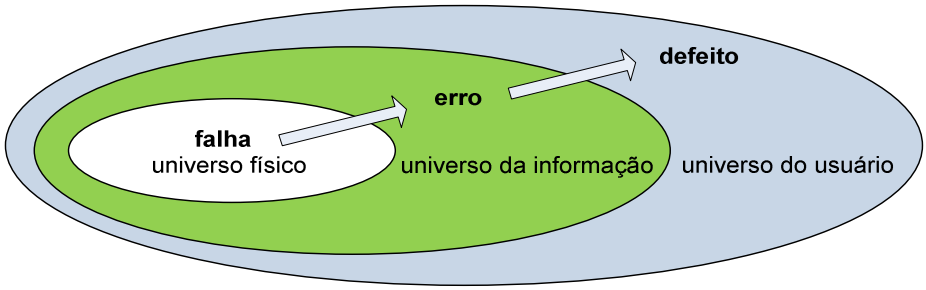


Figura 2. 1 - Modelo dos três universos [3].

Um exemplo para este modelo de três universos [3] seria um *chip* de memória, que apresenta uma falha do tipo *stuck-at-zero* em um de seus *bits* (falha no universo físico). Esta falha pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro no universo da informação). Como resultado deste erro, por exemplo, o sistema pode negar autorização de embarque para todos os passageiros de um vôo (defeito no universo do usuário). É interessante observar que uma falha não necessariamente leva a um erro (pois a porção da memória sob falha pode nunca ser usada) e um erro não necessariamente conduz a um defeito (no exemplo, a informação de vôo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura).

2.2.1. Latência

Define-se latência de falha como o período de tempo desde a ocorrência da falha até a manifestação do erro provocado por esta falha. Seguindo esta linha de raciocínio, define-se como latência de erro o período de tempo desde a ocorrência do erro até a manifestação do defeito. Baseando-se no modelo de três universos [3], o tempo total medido desde a ocorrência da falha até o aparecimento do defeito é a soma das latências de falha e de erro. A Figura 2. 2 apresenta o conceito de latência de falha e latência de erro.

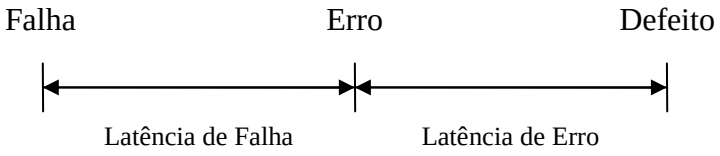


Figura 2. 2 - Conceito de latência de falha e latência de erro.

2.2.2. Classificação de Falhas

Existem diversas classificações para falhas na literatura [1][2][3], entretanto estas são geralmente classificadas em falhas físicas (aquelas de que padecem os componentes) e falhas humanas (que compreendem falhas de projeto e falhas de operação).

Grande parte das causas de falhas são atribuídas a problemas de especificação, problemas de implementação, componentes defeituosos, imperfeições de manufatura, fadiga dos componentes físicos, além de distúrbios externos como radiação, interferência eletromagnética, variações ambientais (temperatura, pressão, umidade) e também problemas de operação.

Para definir uma falha, além do agente causador consideram-se também os seguintes itens:

a) **Natureza:** falha de *hardware*, falha de *software*, de projeto, de operação;
b) **Duração ou persistência:** permanente ou temporária (intermitente ou transitória);
c) **Extensão:** global ou local a um determinado módulo do circuito;
d) **Valor:** determinado ou indeterminado no tempo.

Existe uma crescente ocorrência de falhas provocadas por interação humana maliciosa, ou seja, por ações que visam propositalmente provocar danos aos sistemas. Essas falhas não são tratadas por técnicas de tolerância a falhas, mas sim por técnicas de segurança computacional (do inglês, *security*). Entretanto deve-se considerar que um sistema tolerante a falhas deve também ser seguro a intrusões e ações maliciosas.

Falhas de *software* e também de projeto de *hardware* são consideradas atualmente o mais grave problema em computação crítica. Sistemas críticos são, tradicionalmente, construídos de forma a suportar e tolerar falhas físicas. Tendo em vista isto, é compreensível que falhas não tratadas e não previstas no projeto sejam as que mais danos causem aos sistemas, pois possuem um grande potencial de comprometer a sua confiabilidade e disponibilidade. Um exame de estatísticas disponíveis em [6] confirma essas considerações conforme mostrado na Tabela 2.1.

Tabela 2.1 - Causas usuais em sistemas computacionais [6].

Sistemas Tradicionais			
Não tolerantes a falhas		**Tolerantes a falhas**	
MTTF: 6 a 12 semanas Indisponibilidade após defeito: 1 a 4 horas		**MTTF: 21 anos (tendem)**	
Defeitos:		**Defeitos:**	
hardware	50%	*software*	65%
software	25%	operações	10%
comunicação/ambiente	15%	*hardware*	8%
operações	10%	ambiente	7%

2.3. Defeitos e Modelos de Falhas

Os testes em circuitos ou sistemas eletrônicos são realizados com o intuito de detectar falhas eventualmente presentes nestes dispositivos. Conseqüentemente, para a realização destes testes é necessário o emprego de modelos de falhas baseados em falhas reais definidas a partir de mecanismos físicos e *layouts* reais. Segundo Paul H. Bardell [7], um modelo de falha especifica a série de defeitos físicos que podem ser detectados através de um procedimento de teste. Um bom modelo de falha, segundo Charles E. Stroud [8], deve ser

computacionalmente eficiente em relação ao dispositivo de simulação e refletir fielmente o comportamento dos defeitos que podem ocorrer durante o processo de projeto e manufatura, bem como o comportamento das falhas que podem ocorrer durante a operação do sistema. Estes modelos são utilizados na emulação de falhas e defeitos durante a etapa de simulação do projeto.

Neste contexto, nos últimos anos surgiram diversos modelos de falhas baseados nos principais defeitos físicos dos circuitos e sistemas eletrônicos, alguns destes serão apresentados nos itens a seguir.

a) Modelo de falha *Gate-Level Stuck-at*

Este modelo de falha define que as portas de entrada e/ou saída do circuito podem estar fixadas em nível lógico ‘0’ (*stuck-at-zero*) ou fixadas em nível lógico ‘1’ (*stuck-at-one*). Salienta-se que as falhas *stuck-at* são emuladas como se as portas de entradas e/ou saídas estivessem desconectadas e fixadas ao nível lógico ‘0’ (*stuck-at-zero*) ou ao nível lógico ‘1’ (*stuck-at-one*) [6] A Figura 2.3 apresenta as formas de notação e emulações utilizadas para falhas *stuck-at*.

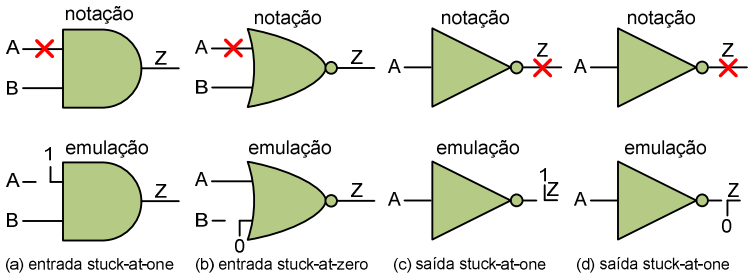


Figura 2.3 - Notação e emulação do modelo de falha *Stuck-at* [8].

b) Modelo de Falha *Transistor-Level Stuck*

Este modelo reflete o comportamento exato das falhas de transistores em circuitos *CMOS* (*Complementary Metal-Oxide-Semiconductor*) e define que qualquer transistor pode estar em *stuck-on* (*s-on*) ou em *stuck-off* (*s-off*) [6].

Salienta-se que as falhas *stuck-on* (também denominado *stuck-short*) podem ser emuladas através de um curto circuito entre o *source* e o *drain* do transistor e as falhas *stuck-off* (também denominado *stuck-open*) desconectando-se o transistor do circuito.

Alternativamente, falhas *stuck-on* podem ser emuladas desconectando o pino de *gate* de um determinado *MOSFET* (*Metal-Oxide Semiconductor Field Effect Transistor*) do circuito e conectando-o ao nível lógico ‘1’ para transistores *NMOS* (*Negative Metal Oxide Semiconductor*) ou ao nível lógico ‘0’ para transistores *PMOS* (*Positive Metal Oxide*

Semiconductor). O raciocínio inverso desta lógica pode ser realizado para emular falhas do tipo *stuck-off*. A Figura 2.4 apresenta um exemplo de modelamento de falha *Transistor-Level Stuck* em uma porta lógica NOR.

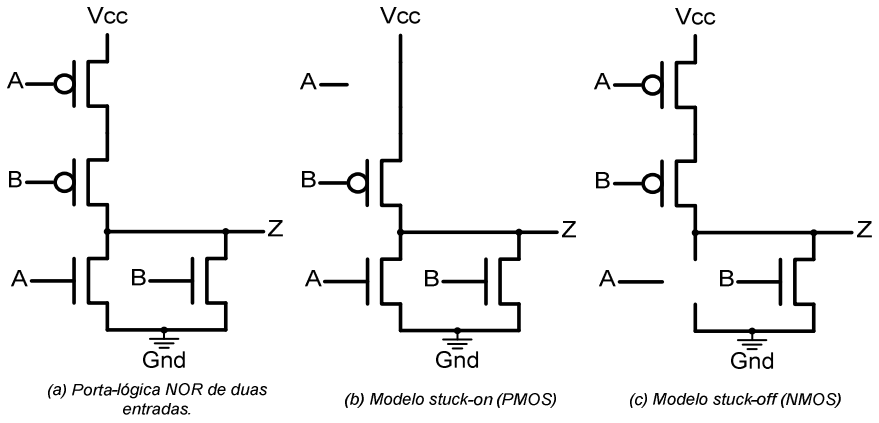


Figura 2.4 - Modelo de falha *Transistor-Level Stuck* [8].

c) Modelo de Falha *Bridging*

Este modelo inclui outro importante conjunto de falhas, tais como rompimentos e/ou curtos entre trilhas de um determinado circuito.

Basicamente, a presença deste tipo de falha é resultante da deposição excessiva (*over-etching)* e/ou reduzida (*under-etching)* de material condutor durante o processo de fabricação das trilhas de circuitos VLSI (*Very Large Scale Integration*) ou ainda em PCB (*Printed Circuit Board*) [9].

Observa-se que embora falhas *transistor-level* e *bridging* reflitam com maior fidelidade o comportamento das falhas presentes em circuitos, sua emulação e avaliação em simuladores são computacionalmente mais complexas em relação às tradicionais falhas *stuck-at* [6].

d) Modelo de Falha *Delay*

Este modelo representa outra importante classe de falha. Ao contrário dos demais modelos de falha aqui apresentados, os circuitos que apresentam falhas de *delay* executam suas operações corretamente do ponto de vista lógico combinacional, entretanto, estas operações lógicas não são executadas ao longo do circuito na freqüência de operação nominal requerida pelo projeto inicial, ocasionando assim um erro de *timing* proveniente dos diferentes tempos de propagação entre os sinais internos do circuito [9].

Este tipo de falha pode originar-se em um caráter permanente ou transiente. Como exemplo de falha de *delay* em caráter permanente, pode-se citar um *over* e/ou *under-etching* durante o processo de fabricação de circuitos *MOSFET's* com canais muito mais estreitos e/ou

longos do que os pretendidos no projeto inicial. Entretanto, existe também a possibilidade de circuitos *MOSFET's* fabricados sem a presença de *over* e/ou *under-etching* apresentarem falhas de *delay* em caráter transiente.

Assim, o teste de *delay* concentra-se em encontrar e expor toda e qualquer falha que possa existir no dispositivo. O objetivo básico deste tipo de teste é verificar o tempo de propagação dos sinais nos caminhos entre *flip-flops*, entre entradas primárias e *flip-flops* e finalmente entre *flip-flops* e saídas primárias, ou seja, verificar através da lógica combinacional se durante a operação na freqüência requerida, algum caminho interno do dispositivo apresenta erro de *timing*.

Tipicamente, o teste de *delay* consiste na aplicação seqüencial de vetores tal que o caminho através da lógica combinacional é carregado com o primeiro vetor enquanto o segundo vetor gera a transição através dos caminhos internos do circuito para detecção da falha [6].

2.4. Medidas Relacionadas ao Tempo Médio de Funcionamento

As medidas para avaliação de dependabilidade mais usadas na prática são: taxa de defeitos, MTTF (do inglês, *mean time to failure*), MTTR (do inglês, *mean time to repair*), MTBF (do inglês, *mean time between failure*). Estas medidas estão por sua vez relacionadas a outro parâmetro importante chamado confiabilidade [3]. A Tabela 2.2 apresenta uma definição informal dessas medidas.

Tabela 2.2 - Medidas de dependabilidade [3].

Medida	Significado
Taxa de defeitos (*failure rate, hazard function, harzard rate*)	Número esperado de defeitos em um dado período de tempo; é assumido um valor constante durante o tempo de vida útil do componente.
MTTF (*mean time to failure*)	Tempo esperado até a primeira ocorrência de defeito.
MTTR (*mean time to repair*)	Tempo médio para reparo do sistema.
MTBF (*mean time between failure*)	Tempo médio entre os defeitos do sistema.

Estas medidas de dependabilidade são determinadas estatisticamente pelos fabricantes, através da observância do comportamento dos componentes e dispositivos fabricados e deveriam ser fornecidas, tanto para os componentes e dispositivos eletrônicos, quanto para os sistemas de computação mais complexos.

A taxa de defeitos de um componente e/ou dispositivo é mensurada em defeitos por unidade de tempo e é diretamente proporcional ao tempo de vida do componente e/ou dispositivo. Uma representação usual para a taxa de defeitos de componentes de *hardware* é dada pela curva da banheira. A Figura 2.5 apresenta esta curva onde podemos distinguir três fases:

a) Mortalidade infantil: componentes fracos e mal fabricados;
b) Vida útil: taxa de defeitos constantes;
c) Envelhecimento: taxa de defeitos crescentes.

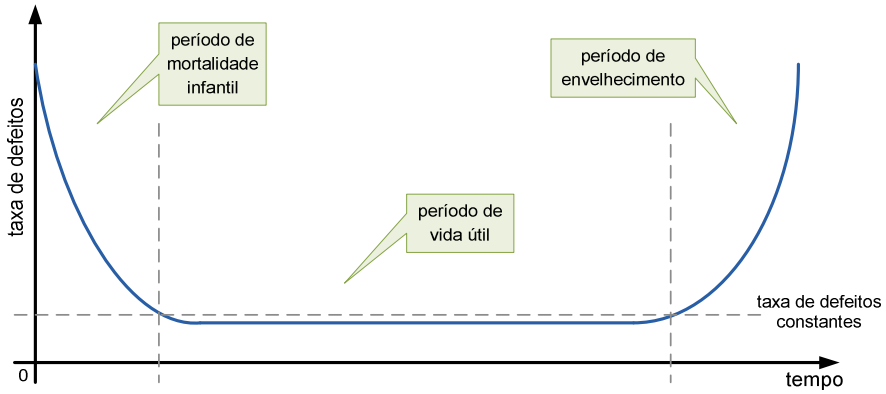


Figura 2.5 - Curva da banheira [3].

Os componentes de *hardware* só apresentam taxa de defeitos constante durante um período de tempo chamado de vida útil, que segue uma fase com taxa de defeitos decrescente chamada de mortalidade infantil. Para acelerar a fase de mortalidade infantil, os fabricantes recorrem a técnicas de *burn-in*, onde é efetuada a remoção de componentes fracos sendo estes substituídos por componentes que já sobreviveram à fase de mortalidade infantil através do processo de aceleração de operação.

É questionável se a curva da banheira pode ser aplicada também para componentes de *software*. Entretanto, pode ser observado que os componentes de *software* também apresentam uma fase de mortalidade infantil ou taxa de erros acentuados no início da sua fase de testes, que decresce rapidamente até a sua entrada em operação. A partir desse momento, o *software* apresenta um taxa de erros constante até que, eventualmente, precise sofrer alguma alteração ou sua plataforma de *hardware* torne-se obsoleta. Nesse momento, a taxa de erros volta a crescer [10]. Intencionalmente mencionamos, quando referido a *software*, taxa de erros ao contrário de defeitos, já que erro é o termo usualmente empregado quando se trata de programas incorretos.

3. Códigos Corretores de Erros

3.1. Introdução

Os códigos corretores de erros aparecem no nosso cotidiano de várias formas, estão presentes, por exemplo, quando fazemos uso de informações digitalizadas, tais como assistir a um programa de televisão, falar pelo telefone, ouvir um CD (*Compact Disc*) de música, assistir a um filme pelo DVD (*Digital Versatile Disc*) ou simplesmente navegar pela Internet. Um código corretor de erros é, essencialmente, uma maneira organizada de acrescentar algum dado adicional a cada informação que precise ser transmitida ou armazenada, de modo que permita, ao recuperar a informação, detectar e corrigir os erros cometidos no processo de armazenamento ou transmissão da informação [11]. Em 1965 o *Mariner* 4 enviou ao planeta Terra as primeiras imagens (não muito boas em qualidade) do planeta Marte, o que foi um grande avanço. Anos depois (1969-1972) os *Mariners* 6, 7 e 8 repetiram o experimento, melhorando a qualidade das imagens [11]. A razão principal dessa melhoria foi que, para a transmissão destas últimas imagens, tinha sido utilizado um potente código corretor, capaz de corrigir até 5 *bits* errôneos de cada seqüência de 32 *bits*.

Existem basicamente dois tipos de códigos corretores de erros utilizados, os Códigos de Bloco e os Códigos Convolucionais.

3.1.1. Códigos de Bloco

Nesta seção, serão apresentados os conceitos básicos dos códigos de Bloco e em especial de uma subclasse - os códigos de bloco lineares [12]. Os códigos de Bloco, basicamente, processam a informação bloco por bloco, tratando cada bloco de *bits* de informação de forma independente em relação a outro bloco. Diz-se que um código é linear se duas palavras-código quaisquer do código puderem ser somadas em aritmética módulo 2 para

produzir uma terceira palavra-código. A codificação da informação é realizada em duas etapas:

1. A seqüência de informação é dividida em blocos de informação, com *k* dígitos sucessivos de informação;
2. O codificador transforma, então, cada bloco de informação em um bloco maior com *n* (*n*>*k*) dígitos binários (palavra-código), de acordo com uma determinada regra de codificação. Este novo bloco é denominado Palavra-Código.

Cada bloco de informação contém *k* dígitos binários. Portanto, existem 2^k blocos de informação distintos. Conseqüentemente, existem 2^k palavras-código correspondentes. Este conjunto de 2^k palavras-código é denominado Código de Bloco.

Um Código de Bloco Linear é formado pelo conjunto das 2^k palavras-código que constituem um subespaço do espaço vetorial V_n de todas as palavras-código.

Seja **C** um codificador que divide a seqüência de informação em blocos de 3 dígitos e transforma cada mensagem em uma palavra-código de 6 dígitos, conforme demonstrado na Tabela 3.1.

Tabela 3.1 - Exemplo de código de bloco (6,3).

Informação	Palavra-Código
000	000 000
001	001 101
010	010 011
011	011 110
100	100 110
101	101 011
110	110 101
111	111 000

Com *k*=3, existem $2^3 = 8$ informações distintas possíveis. Para cada bloco de informação o codificador gerou uma palavra-código distinta, com 6 dígitos. Assim, a partir de cada uma das palavras-código geradas é possível obter-se a informação original.

3.1.2. Matriz Geradora

Todas as palavra-código (vetores) de um subespaço vetorial **S** de V_n podem ser obtidas através de uma combinação linear de um conjunto de palavra-código linearmente independentes. Assim, um código de bloco linear com 2^k vetores código (palavras-código) pode ser representado por um conjunto de *k* vetores código linearmente independentes. Estes vetores linearmente independentes podem ser organizados como linhas em uma matriz *k* x *n*:

$$G = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} & v_{13} & \cdots & v_{1n} \\ v_{21} & v_{22} & v_{23} & \cdots & v_{2n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ v_{k1} & v_{k2} & v_{k3} & \cdots & v_{kn} \end{bmatrix} \tag{3.1}$$

Esta matriz é denominada Matriz Geradora do código, pois combinações lineares entre suas linhas geram um código de bloco linear. A palavra-código correspondente a um bloco de informação **m= (**m_1, m_2,..., m_k**)** pode ser obtida por:

$$\begin{aligned} u &= mG \\ u &= \begin{pmatrix} m_1, & m_2, & \cdots, & m_k \end{pmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix} \\ u &= m_1 V_1 + m_2 V_2 + \cdots + m_k V_k \end{aligned} \tag{3.2}$$

Conforme já observado, a palavra-código correspondente ao bloco de informação **(**m_1, m_2,..., m_k**)** é uma combinação linear das linhas da matriz geradora **G**. Este código linear é conhecido como um **Código de Bloco (*n, k*)**, no qual um bloco de informação com *k* dígitos é codificado em uma palavra-código com *n* dígitos para ser armazenado numa memória. A razão R= *k* / *n* é denominada **Taxa do Código** e representa a quantidade de redundância contida na palavra-código.

O código apresentado na Tabela 3.1, no início desta seção, é um código (6,3) e sua matriz geradora está representada abaixo.

$$G = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{matrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{matrix} \tag{3.3}$$

A palavra-código correspondente a informação **m**=101 é dada por:

$$u = \begin{pmatrix}1 & 0 & 1\end{pmatrix}\begin{bmatrix}V_1\\V_2\\V_3\end{bmatrix} = 1\cdot V_1 + 0\cdot V_2 + 1\cdot V_3$$
$$u = 1\cdot(100110) + 0.(010011) + 1.(001101) \qquad (3.4)$$
$$u = 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1$$

Deve-se notar que, devido ao fato de o código de bloco linear ser completamente especificado por sua matriz geradora, o codificador não precisa armazenar todas as palavras-código em sua memória. Assim, para o conjunto de 2^k palavras-código apenas k linhas da matriz geradora são armazenadas. Outro ponto importante é que, como a palavra-código atual depende apenas do bloco de informação atual, não há memória envolvida no processo de codificação. Portanto, o codificador pode ser implementado através de uma lógica combinacional.

3.1.3. Códigos Sistemáticos

É possível definir-se um processo de codificação em que os k dígitos iniciais de cada palavra-código sejam exatamente os mesmos k dígitos de cada bloco de informação, enquanto os n-k dígitos restantes serão os dígitos de redundância. Um código deste tipo é chamado de **Código Sistemático** e sua estrutura está representada na Figura 3.1.

Palavra-Código	
Bloco de informação	**Dígitos de Redundância**
k	***n-k***

Figura 3.1 - Estrutura da palavra-código em um código sistemático.

Um código linear sistemático (n, k) pode ser descrito por uma matriz geradora, com a seguinte estrutura:

$$G = \begin{bmatrix} 1000...0 & p_{11} & p_{12} & \cdots & p_{1,n-k} \\ 0100...0 & p_{21} & p_{22} & \cdots & p_{2,n-k} \\ 0010...0 & p_{31} & p_{32} & \cdots & p_{3,n-k} \\ \vdots & \vdots & \vdots & & \vdots \\ 0000...1 & p_{k1} & p_{k2} & \cdots & p_{k,n-k} \end{bmatrix} \qquad (3.5)$$

onde p_{ij}=0 ou 1.

A matriz identidade é uma matriz quadrada na qual todos os elementos da diagonal principal (elementos de índice a_{ii}, ou seja, a_{11}, a_{22}, etc.) são iguais a 1. Considerando-se que $\mathbf{I}_k$ seja a matriz identidade de ordem k e que **P** seja a matriz $k \times (n\text{-}k)$ de p_{ij}, a matriz geradora de um código sistemático pode ser representada por:

$$\mathbf{G = [I_k P]} \tag{3.6}$$

A partir das Equações (3.2) e (3.6), a palavra-código correspondente a um bloco de informação $\mathbf{m} = (m_1, m_2, ..., m_k)$ é dada por:

$$\begin{aligned} u &= (u_1, \quad u_2, \quad u_3, \quad \cdots, \quad u_n) \\ u &= (m_1, \quad m_2, \quad \cdots, \quad m_k).G \\ u &= (m_1, \quad m_2, \quad \cdots, \quad m_k).\begin{bmatrix} 1000...0 & p_{11} & p_{12} & \cdots & p_{1,n-k} \\ 0100...0 & p_{21} & p_{22} & \cdots & p_{2,n-k} \\ 0010...0 & p_{31} & p_{32} & \cdots & p_{3,n-k} \\ \vdots & \vdots & \vdots & & \vdots \\ 0000...1 & p_{k1} & p_{k2} & \cdots & p_{k,n-k} \end{bmatrix} \end{aligned} \tag{3.7}$$

Pode-se observar, pela multiplicação das matrizes, que:

$$u_i = m_i \quad \text{para} \quad i = 1, \quad 2, \quad \cdots, \quad k \tag{3.8}$$

$$u_{k+j} = p_{1j}m_1 + p_{2j}m_2 + ... + p_{kj}m_k \ \text{ para } \ j = 1, \quad 2, \quad \cdots, \quad n-k \tag{3.9}$$

Pelas equações (3.8) e (3.9) verifica-se facilmente que os primeiros k dígitos da palavra-código são exatamente os dígitos do bloco de informação, enquanto os últimos $n\text{-}k$ dígitos são funções lineares dos dígitos de informação. Estes $n\text{-}k$ últimos dígitos da palavra-código são conhecidos como Dígitos de Verificação de Paridade do código.

Considerando-se mais uma vez o código apresentado na Tabela 3.1, juntamente com sua matriz geradora, a palavra-código gerada para o bloco de informação (m_1, m_2, m_3) será:

$$
\begin{aligned}
u &= \begin{pmatrix} u_1, & u_2, & u_3, & \cdots, & u_6 \end{pmatrix} \\
u &= \begin{pmatrix} m_1 & m_2 & m_3 \end{pmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \\
u &= \begin{pmatrix} m_1, & m_2, & m_3, & m_1 + m_3, & m_1 + m_2, & m_2 + m_3 \end{pmatrix}
\end{aligned} \tag{3.10}
$$

Então, $u_1 = m_1$, $u_2 = m_2$, $u_3 = m_3$. Os n-k dígitos restantes (u_4, u_5, u_6) são funções lineares dos k dígitos de informação. Assim, para um código de bloco linear na forma sistemática, a complexidade de codificação fica ainda mais reduzida, pois o codificador tem que armazenar apenas a matriz $P_{k\times(n-k)}$ de p_{ij}, ao invés de armazenar todas as linhas da matriz geradora **G**.

3.1.4. Matriz de Verificação de Paridade

Para uma dada matriz $G_{k\times n}$ existe uma correspondente matriz $\mathbf{H}_{(n-k)\times n}$, tal que o espaço linha da matriz **G** é ortogonal a **H**, isto é, o produto interno de um vetor do espaço linha de **G** e uma linha de **H** é zero[27]. Então, um código de bloco linear gerado pela matriz **G** pode ser descrito de forma alternativa: **u** é uma palavra-código gerada por **G** se e somente se $\mathbf{uH^T}$=0. Isto significa que os vetores que pertencem ao espaço linha de **G** também pertencem ao espaço nulo de **H**. A matriz **H** é denominada **Matriz de Verificação de Paridade**. Se a matriz geradora de um código binário sistemático estiver na forma apresentada na equação (3.6), sua matriz de verificação de paridade será:

$$
H = \begin{bmatrix} p_{11} & p_{21} & \cdots & p_{k1} & 1000...0 \\ p_{12} & p_{22} & \cdots & p_{k2} & 0100...0 \\ \vdots & \vdots & & \vdots & \vdots \\ p_{1,n-k} & p_{2,n-k} & \cdots & p_{k,n-k} & 0000...1 \end{bmatrix} = \begin{bmatrix} P^T I_{n-k} \end{bmatrix} \tag{3.11}
$$

onde $\mathbf{P^T}$ é a matriz **P** transposta.

Conforme descrito anteriormente, o espaço linha da matriz **G** é o espaço nulo de **H**, e vice-versa. Então, um código de bloco linear pode ser unicamente especificado tanto por sua matriz **G**, quanto pela **H**.

3.1.5. Capacidade de Correção de Um Código Linear

Para as considerações referentes à capacidade de correção de um código linear são necessárias algumas definições, apresentadas a seguir:

- O **Peso de *Hamming*** de uma palavra-código **v**, ω(**v**)**,** é definido como o número de elementos de **v** diferentes de zero. Se **v**= (1001011001), por exemplo, ω(v) =5;
- A **Distância de *Hamming*** entre dois vetores **u** e **v**, *d*(**u, v**), é definida como o número de *bits* nos quais eles diferem. Se **u**= (1001011001) e **v**= (1**10**0**0**10**10**), por exemplo, *d*(**u, v**) =5;
- A **Distância Mínima** de um código de bloco linear, $d_{\min}$, é definida como a menor distância de *Hamming* entre todos os possíveis pares de palavras-código.

Pela regra da adição *módulo*-2, verifica-se que:

$$d(\mathbf{u},\mathbf{v})=\omega(\mathbf{u}+\mathbf{v}) \tag{3.12}$$

Ou seja, a distância entre dois vetores **u** e **v** é igual ao peso de sua soma vetorial, **u** + **v**.

Seja **v** = (v_1, v_2,..., v_n) o vetor código (palavra-código) armazenado numa memória, que utiliza um código corretor de erros aleatórios. Devido à influência de interferências presentes no ambiente, o vetor ***r*** = (r_1, r_2,..., r_n) pode ser qualquer uma das 2^n combinações. A diferença entre **r** e **v** é chamado **Padrão de Erros** ou **Vetor de Erros** causado pelos distúrbios que afetam os dados armazenados na memória e é dado por:

$$\begin{aligned} e&=(e_1,e_2,\ldots e_n)\\ e&=r+v\\ e&=(r_1,r_2,\ldots,r_n)+(v_1,v_2,\ldots,v_n)\\ e&=(r_1+v_1,r_2+v_2,\ldots,r_n+v_n) \end{aligned} \tag{3.13}$$

Quando $e_i = r_i + v_i = 1$, há um erro na posição *i* do vetor lido **r**.

A função do decodificador é a de identificar o vetor código **v** a partir do vetor recebido **r**. A decodificação de máxima verossimilhança (também denominada decodificação de máxima probabilidade, ou *maximum-likelihood decoding*) irá identificar **v** como sendo o vetor código mais próximo do vetor recebido **r** com relação à distância de *Hamming*, isto é, aquele para o qual $d(\mathbf{v}, \mathbf{r})$ é mínimo [13].

Pode-se demonstrar que um código com distância mínima $d_{\min}$ pode corrigir $\left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$ erros [46]. Assim, a capacidade de correção deste código fica definida por:

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{3.14}$$

Retomando-se o código exemplo da Tabela 3.1, verifica-se que sua distância mínima é 3 e, portanto, sua capacidade de correção é de 1 erro. Para um vetor armazenado igual ao padrão (000000) pode-se observar um caso de decodificação incorreta considerando-se a ocorrência de dois erros, um na 3ª posição e outro na 6ª posição. O vetor lido será, então, igual ao padrão (001001). A distância de *Hamming* entre o vetor lido e o vetor código (001101), por exemplo, é igual a 1, enquanto que a distância entre o vetor recebido e o vetor originalmente armazenado é 2. Assim, o decodificador identifica incorretamente o vetor (001101) como sendo a palavra-código armazenada.

3.1.6. Síndrome

Seja **u** um vetor código armazenado numa memória. O decodificador obtém um vetor corrompido **r**, que é uma soma vetorial do vetor originalmente armazenado **u** com o vetor de erros **e**. O objetivo do decodificador é recuperar o vetor **u** a partir de **r**.

$$\mathbf{r} = \mathbf{u} + \mathbf{e} \tag{3.15}$$

A **Síndrome** do vetor decodificado **r** é um vetor com $(n\text{-}k)$ componentes, utilizada no processo de detecção e correção de erros e definida pela seguinte equação:

$$\mathbf{s} = \mathbf{r}\mathbf{H}^{\mathbf{T}} \tag{3.16}$$

Pode-se observar que se o vetor decodificado for uma palavra-código ou, em outras palavras, se ele pertencer ao espaço nulo de **H**, a síndrome será igual a zero. Portanto, se o decodificador calcular a síndrome e o resultado for diferente de zero ele já terá detectado a presença de erros no vetor decodificado, conforme demonstrado no exemplo a seguir.

A matriz geradora do código apresentado na Tabela 3.1 é:

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.17}$$

De acordo com a equação (3.11), a correspondente matriz de verificação de paridade é:

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{3.18}$$

A informação (111000) é uma palavra-código, conforme observado na Tabela 3.1. Então, sua síndrome será nula:

$$s = (111000)\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (0 \quad 0 \quad 0) \tag{3.19}$$

A informação (111001), no entanto, não é uma palavra-código (não pertence ao espaço linha de **G** ou ao espaço nulo de **H**). A síndrome deste vetor não será nula:

$$s = (111001)\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (0 \quad 0 \quad 1) \tag{3.20}$$

No caso de códigos sistemáticos, a síndrome equivale à soma *módulo-2* dos dígitos de paridade lidos com os dígitos de paridade recalculados a partir dos dígitos de informação lidos [14]. Isto pode ser facilmente observado, notando-se que a matriz é formada pela matriz **P** (parte da matriz geradora) e pela matriz identidade $\mathbf{I}_{n-k}$.

3.1.7. Exemplos de Códigos de Bloco

O problema de encontrar um código com uma dada capacidade de correção se reduz ao de encontrar um código com uma dada distância mínima. Infelizmente, porém, não existem regras genéricas para sua construção. A seguir, serão apresentados alguns exemplos de códigos de bloco binários lineares bastante simples e bem conhecidos [15].

O **Código de Repetição** é a maneira mais simples de se implementar detecção e correção de erros. Trata-se de um código do tipo (n, 1), no qual cada dígito de informação é repetido n vezes. Pode-se observar que a distância mínima de um código de bloco de repetição é n.

O **Código de Paridade Simples**, muito utilizado nos antigos sistemas de "cartão perfurado" de computador e em alguns sistemas de comunicação serial como o padrão RS-232C, por exemplo, é formado por apenas um dígito de paridade obtido através da soma *módulo-2* dos $k = n - 1$ dígitos de informação.

Os **Códigos de *Hamming*** são uma classe de códigos para correção de erros simples inventados por *R. W. Hamming* em 1950. Foram utilizados em sistemas de telefonia de longa distância. Sua distância mínima é 3 e, portanto, a capacidade de correção é de 1 erro. Esta família de códigos é obtida a partir de inteiros $c \geq 2$, com os seguintes parâmetros:

• Comprimento da Palavra-Código $n = 2^c - 1$

• Dígitos de Informação $k = 2^c - c - 1$

• Número de Dígitos de Paridade $c = n - k$

A matriz de verificação de paridade de um código de *Hamming* **C** (n, k) pode ser construída colocando-se todas as palavras-código de c elementos nas colunas de uma matriz c x n. Para o código de *Hamming* **C** (7,4), por exemplo, a matriz de verificação de paridade é:

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad (3.21)$$

3.2. Códigos de *Hamming*

São códigos binários utilizados para detecção e correção de erros e constituem uma família de códigos de bloco lineares com $n = 2^c - 1$, $k = 2^c - c - 1$, $c = n - k$ e $d_{min} = 3$, para qualquer inteiro $c \geq 2$, onde ***c*** é o número de *bits* de paridade, ***n*** é o número total de *bits* da palavra código resultante da codificação (*Codeword*), *k* é o número de *bits* da informação a ser codificada e d_{min} é a distância mínima de *hamming*[27].

Figura 3.2 - Representação Básica do Código de *Hamming.*

Estes códigos foram inventados por *Richard Wesley Hamming* (1915-1998) que publicou no *The Bell System Technical Journal* em abril de 1950 o artigo *Error Detecting and Error Correcting Codes* descrevendo o funcionamento e características destes códigos[16]. De [16][17][18] extraimos algumas características:

- Para determinar o número de *bits* de dados (ou *bits* de informação) , *k,* e o número de *bits* de paridade (*c*), utiliza-se a seguinte relação:

$$2^c \geq k + c + 1 \qquad (3.22)$$

- A taxa de código (***r = k/n***) é a relação entre a quantidade de *bits* de informação (***k***) a ser codificada e a palavra código resultante da codificação (***n***). É uma relação adimensional e representa a eficiência do código.
- A relação entre a quantidade de *bits* de paridade (***c***) e a quantidade de *bits* da mensagem (***k***) resulta no *overhead* do código.
- Cada código de *Hamming* é representado na forma (***n,k***). Ex.: *Hamming* (7,4)

A tabela 3.2 mostra a relação entre os *bits* de dados (*k*), os *bits* de paridade (*c*), o tamanho da palavra código (*n*), a taxa de código (*r*) e o *overhead* de alguns códigos de *Hamming*.

Tabela 3.2 - Relação entre k, c, n, r e *overhead* de alguns códigos de *hamming*.

Hamming ***(n,k)***	**k**	**c**	**n**	**r**	***Overhead*** **(%)**
(7,4)	4	3	7	0,571	75
(12,8)	8	4	12	0,667	50
(21,16)	16	5	21	0,762	31,25
(38,32)	32	6	38	0,842	18,75

Hamming introduziu o uso de vários *bits* de paridade gerados a partir da informação a ser codificada, para detectar e corrigir erro num único *bit*. Um *bit* de paridade diz se o número de 1s de um grupo de *bits* é par ou impar. Ele é acrescentado para que o número de 1s, em um grupo de *bits*, seja par ou impar[19].

Para explicar o processo de codificação será utilizada uma informação com o tamanho de 4 *bits* (k =4). Da tabela 3.2, com k =4, temos n =7 e c =3. Desta forma, temos 3 *bits* de paridade (P_1, P_2 e P_3) e 4 *bits* de dados (D_1, D_2, D_3 e D_4), totalizando 7 *bits* de *codeword* e cada *bit* ocupa uma posição numerada de 1 até 7. Os *bits* de paridade estão localizados nas posições cujos números correspondem as potências de 2 (1, 2, 4) e cada *bit* de paridade provê uma verificação em outros determinados *bits* no código total [19].

Cada posição da Palavra-Código possui uma representação binária (posição 1 = 001_2). O número da posição em binário do *bit* de paridade P_1 possui valor 1 no digito mais a direita (1 = $00\mathbf{1}_2$). Esse *bit* de paridade verifica todas as posições, incluindo ele mesmo, que têm 1s no *bit* mais a direita. Portanto, o *bit* de paridade P_1 verifica as posições de *bit* 1, 3, 5 e 7. O número da posição em binário do *bit* de paridade P_2 possui valor 1 no digito do meio (2 = $0\mathbf{1}0_2$) e verifica as posições 2, 3, 6 e 7. O *bit* de paridade P_3 ocupa a posição 4 e em binário possui valor 1 no digito da esquerda (4 = $\mathbf{1}00_2$) e verifica as posições 4, 5, 6 e 7.

$$k = D_1D_2D_3D_4 = 1001_2 \; c = P_1P_2P_3$$

$P_1 = D_1 \oplus D_2 \oplus D_4 = 1 \oplus 0 \oplus 1 = \mathbf{0}$

$P_2 = D_1 \oplus D_3 \oplus D_4 = 1 \oplus 0 \oplus 1 = \mathbf{0}$

$P_3 = D_2 \oplus D_3 \oplus D_4 = 0 \oplus 0 \oplus 1 = \mathbf{1}$

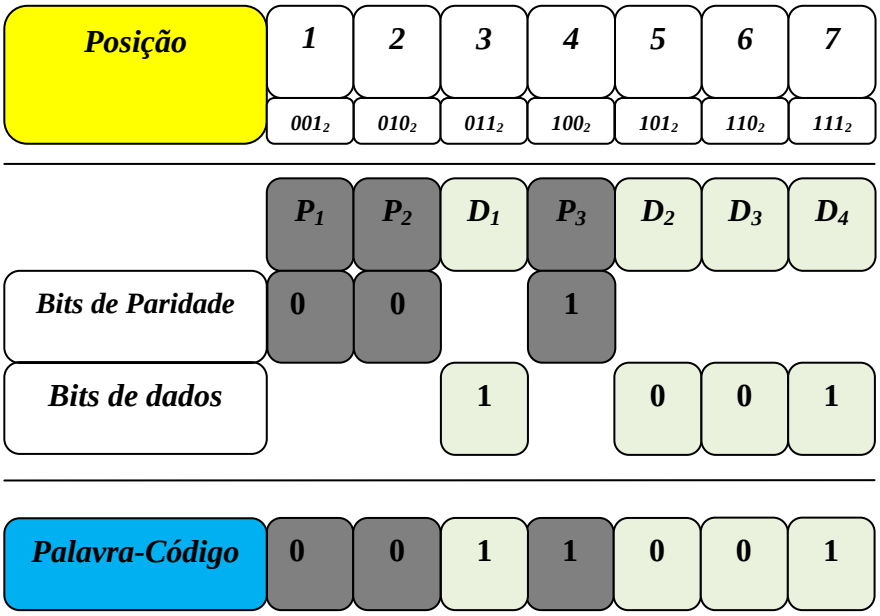


Figura 3.3 - Codificação *Hamming* (7,4) Paridade Par.

A detecção e correção são proporcionadas para todos os *bits* da palavra código, inclusive os *bits* de paridade. Na decodificação é realizada a verificação dos *bits* de paridade da palavra código recebida. Para k =4, há 3 *bits* de paridade (P_1, P_2 e P_3) que formam uma palavra binária $P_3P_2P_1$, onde P_1 é o *bit* menos significativo. Essa palavra é chamada de síndrome. Se não ocorrer nenhum erro, a síndrome é zero. Se ocorrer um simples erro, a síndrome é diferente de zero e o valor indica a posição do *bit* errado e para corrigi-lo, basta inverter o seu valor (0 → 1 ou 1 → 0). Se ocorrerem dois erros, a síndrome é diferente de zero, entretanto, a posição indicada do *bit* errado é incorreta. Os códigos de *hamming* com d_{min}=3 detectam e corrigem um erro (conhecido como SECSED - *Single Error Correction and Single Error Detection*). Ou podem ser usados para detectar erros duplos (DED - *Double Error Detection*) [18].

3.3. Código de *Hamming* + Paridade

Adicionando um *bit* extra de paridade no código de *Hamming*, é possível aumentar a distância mínima para 4 (d_{min} = 4). Este código de *Hamming* com um *bit* de paridade extra é chamado de código de *Hamming* + Paridade ou código de *Hamming* Estendido. Isto dá ao código a capacidade de detectar e corrigir um erro e, ao mesmo tempo, detectar um duplo erro (conhecido como SECDED - *Single Error Correction and Double Error Detection*). Ou podem ser usados para detectar três erros (TED - *Triple Error Detection*).

A diferença no processo de codificação entre o *Hamming* convencional e *Hamming* + Paridade (*Hamming* Estendido) é a adição de um *bit* de paridade no final do processo de codificação.

$$k = D_1D_2D_3D_4 = 1001_2 \quad c = P_1P_2P_3$$

$P_1 = D_1 \oplus D_2 \oplus D_4 = 1 \oplus 0 \oplus 1 = \mathbf{0}$ $P_2 = D_1 \oplus D_3 \oplus D_4 = 1 \oplus 0 \oplus 1 = \mathbf{0}$ $P_3 = D_2 \oplus D_3 \oplus D_4 = 0 \oplus 0 \oplus 1 = \mathbf{1}$

$P = D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus P_1 \oplus P_2 \oplus P_3 = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = \mathbf{1}$

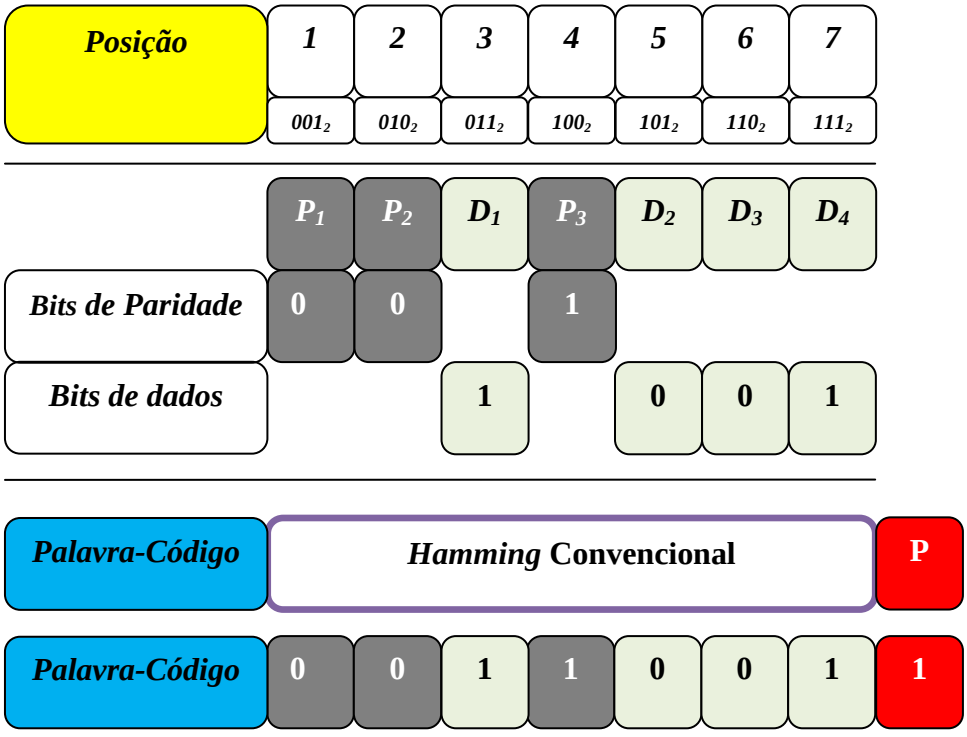


Figura 3.4 - Codificação *Hamming* + Paridade (8,4) Paridade Par.

No processo de decodificação, a diferença entre os códigos é a verificação da paridade P. No código de *Hamming* + paridade podem haver quatro situações que são interpretadas da seguinte maneira (tabela 3.3):

Tabela 3.3 - Interpretação da Síndrome e do *bit* de Paridade do código de *Hamming* + Paridade.

Sindrome	Paridade	Interpretação
Zero	P = 0	*Não Ocorreu nenhum erro*
Diferente de Zero	P = 1	*Ocorreu um erro que pode ser corrigido*
Diferente de Zero	P = 0	*Ocorreu erro duplo mas não pode ser corrigido*
Zero	P = 1	*Um erro ocorreu no bit de paridade P*

A tabela 3.4 mostra a relação entre os *bits* de dados (*k*), o tamanho da Palavra-Código (*n*) e o *overhead* de alguns códigos de *Hamming* + Paridade.

Tabela 3.4 - Relação entre k, n e *overhead* de alguns códigos de *hamming* + Paridade.

Hamming + Paridade (n,k)	**k**	**n**	***Overhead* (%)**
(8,4)	4	**8**	100
(13,8)	8	**13**	62,50
(22,16)	16	**22**	37,50
(39,32)	32	**39**	21,87

3.4. *Matrix Code*

Este código foi desenvolvido em 2006 por *Costas Argyrides, Hamid R. Zarandi e Dhiraj K. Pradhan* [20] [21] [22] [23] para proteger memórias SRAM de Múltiplas falhas transientes. O método proposto combina o código de *Hamming* Estendido e o código da paridade para assegurar a confiabilidade da memória na presença de múltiplas falhas transientes.

O esquema proposto para detecção/correção é chamado código Matriz (*Matrix Code*) porque os *bits* a serem protegidos e a redundância necessária para protegê-los estão dispostos num formato de matriz. Neste caso, os X *bits* a serem protegidos são divididos em k_1 linhas de tamanho k_2. A matriz (k_1, k_2) é formada por k_1 e k_2 que representam, respectivamente, os números de linhas e o número de colunas da matriz. Para cada uma das k_1 linhas, são adicionadas redundâncias para detecção e correção de um erro e detecção de erro duplo (código de *Hamming* Estendido). Também são adicionados k_2 *bits* de paridade vertical nas colunas. Para um entendimento completo será utilizado um exemplo de codificação utilizando uma palavra de 16 *bits*.

Nesta situação, uma palavra de 16 *bits* é dividida em uma matriz 4 x 4 como mostrado na Figura 3.5, onde $k_1 = k_2 = 4$. Os códigos de *Hamming* + Paridade são aplicados para cada linha (neste exemplo são quatro linhas). Para 4 *bits* de dados são adicionados ao final de cada linha, quatro *bits* de verificação de *Hamming* + Paridade. Nas colunas, para cada quatro *bits*, é adicionado um *bit* de paridade, totalizando neste exemplo quatro *bits* de paridade (P1 - P4).

X_1	X_2	X_3	X_4	C_1	C_2	C_3	C_4
X_5	X_6	X_7	X_8	C_5	C_6	C_7	C_8
X_9	X_{10}	X_{11}	X_{12}	C_9	C_{10}	C_{11}	C_{12}
X_{13}	X_{14}	X_{15}	X_{16}	C_{13}	C_{14}	C_{15}	C_{16}
P_1	P_2	P_3	P_4				

Figura 3.5 - Exemplo do Código Matriz.

Por exemplo, na Figura 3.5, as posições C_1, C_2, C_3 e C_4 são os *bits* de *Hamming* + Paridade que servem para verificar os *bits* da primeira linha. Os *bits* de paridade são calculados usando as equações (3.1 – 3.4):

$$\mathbf{C_1=X_1 \oplus X_2 \oplus X_4} \quad (3.1)$$

$$\mathbf{C_2=X_1 \oplus X_3 \oplus X_4} \quad (3.2)$$

$$\mathbf{C_3=X_2 \oplus X_3 \oplus X_4} \quad (3.3)$$

$$\mathbf{C_4= X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus C_1 \oplus C_2 \oplus C_3} \quad (3.4)$$

onde Xi são os *bits* da palavra de dados. Os *bits* paridade vertical (P1-P4) são adicionados conforme Figura 3.6. As equações (3.5 – 3.8) são usadas para a geração dos *bits* de paridade.

$$\mathbf{P_1=X_1 \oplus X_5 \oplus X_9 \oplus X_{13}} \quad (3.5)$$

$$\mathbf{P_2=X_2 \oplus X_6 \oplus X_{10} \oplus X_{14}} \quad (3.6)$$

$$\mathbf{P_3=X_3 \oplus X_7 \oplus X_{11} \oplus X_{15}} \quad (3.7)$$

$$\mathbf{P_4=X_4 \oplus X_8 \oplus X_{12} \oplus X_{16}} \quad (3.8)$$

Desta forma, X_1 - X_{16} são os *bits* de dados, C_1 - C_{16} são os *bits* de *hamming* e P_1 - P_4 são os *bits* de paridade vertical.

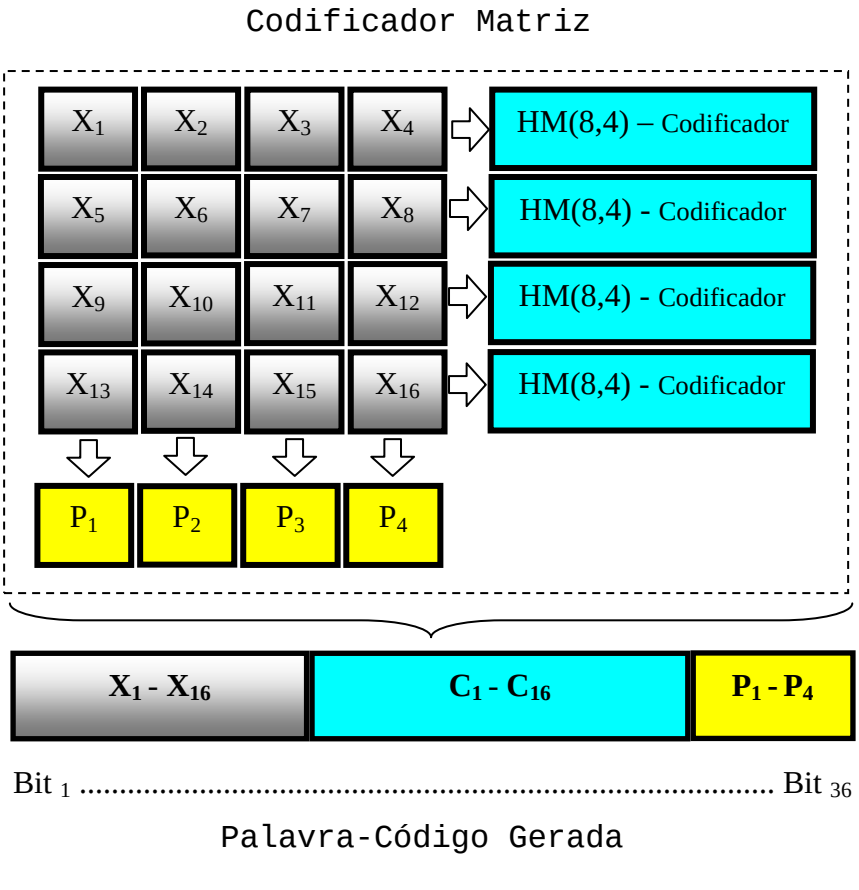
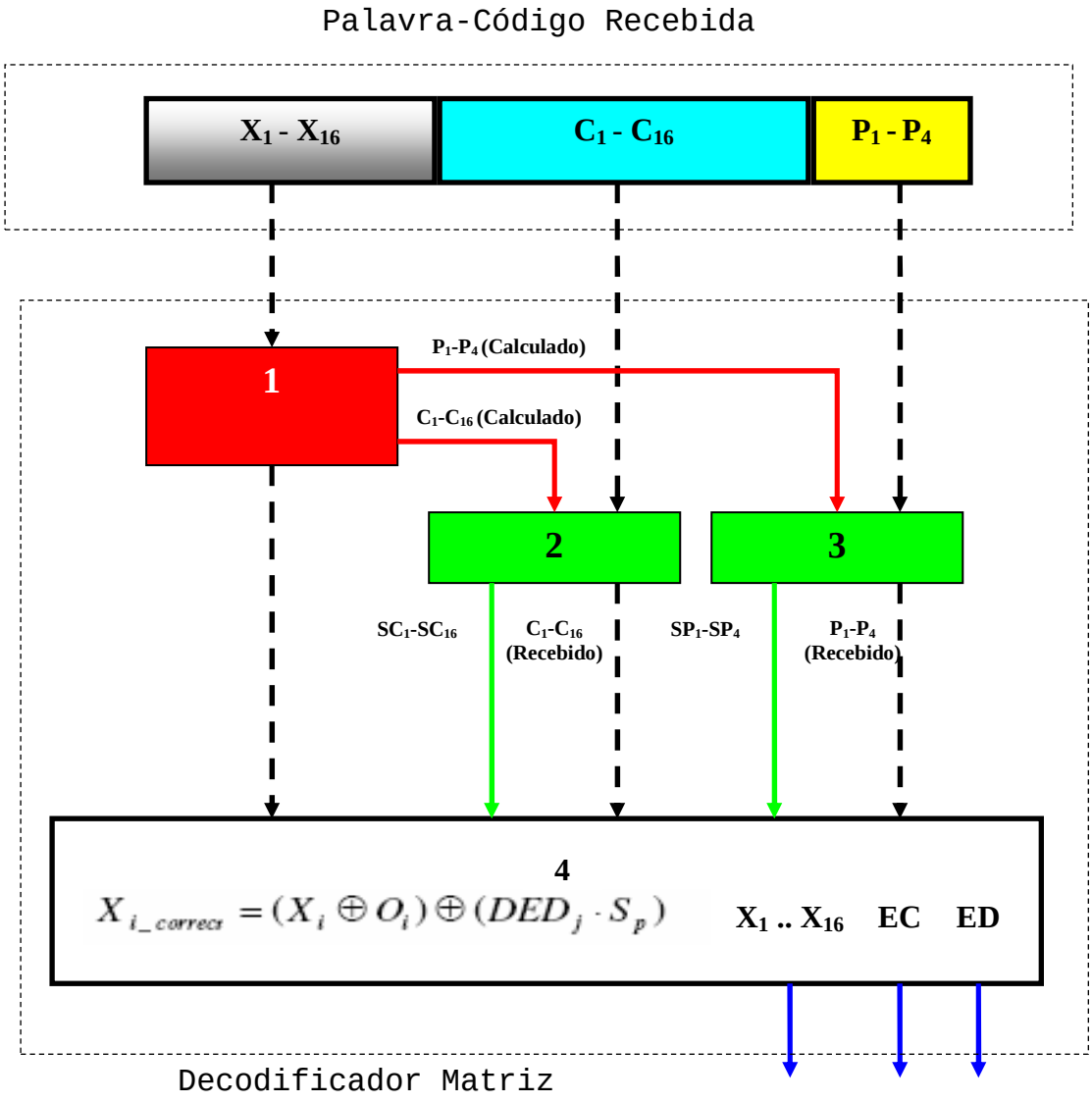


Figura 3.6 - Codificador Matriz.



Bloco 1: *Calcula os bits P1-P4 e os bits C1-C16 da Palavra-Código recebida*

Bloco 2: *Calcula a síndrome dos bits de Hamming + Paridade(SC1 – SC16)*

Bloco 3: *Calcula a síndrome da paridade vertical (SP1-SP4)*

Bloco 4: *Corrigi os bits recebidos utilizando a equação 3.9 e gera os sinais EC e ED e os bits X_1 - X_{16}.*

Figura 3.7 - Decodificador Matriz.

O processo de decodificação é realizado conforme a Figura 3.7. O processo de decodificação é realizado da seguinte forma:

- Após receber a Palavra-Código, o bloco 1 do decodificador calcula os *bits* de *Hamming* + Paridade (**C_1-C_{16}**) e a paridade vertical (**P_1-P_4**).
- No bloco 2 é gerada a síndrome (**SC_1-SC_{16}**) entre os *bits* de *Hamming* + Paridade calculados pelo bloco 1 com os *bits* de *hamming* + paridade recebidos.
- No bloco 3 é gerada a síndrome (**SP_1-SP_4**) entre os *bits* de paridade vertical calculados pelo bloco 1 com os *bits* de paridade vertical recebidos.
- O bloco 4 é responsável por gerar os sinais necessários para o cálculo da equação 3.9. Na saída temos os dados decodificados (X_1-X_{16}) e os pinos EC e ED que indicam, respectivamente, Erro Corrigido e Erro Detectado.

$$X_{i\_correct} = (X_i \oplus O_i) \oplus (DED_j . S_p) \qquad (3.9)$$

Algoritmo de Verificação do Código Matriz (X: informação)

{

Leitura dos *bits* armazenados na memória – **X**

Geração dos *bits* de verificação usando os *bits* armazenados(X) - **(C_1' – C_{12}')**

Geração dos *bits* de síndrome dos *bits* de verificação - **(SC_1' – SC_{12}')**

Geração dos *bits* de paridade usando os *bits* armazenados(X) - **(P_1' – P_4')**

Geração dos *bits* de síndrome dos *bits* de paridade - **(SP_1' – SP_4')**

Correção dos *bits* errados armazenados utilizando a equação 3.9

Palavra corrigida na saída do decodificador

}

Figura 3.8 - Procedimento da detecção de erro/correção no código Matriz.

Neste exemplo, o código Matriz pode corrigir dois erros por linha desde que haja somente um erro em cada coluna. Ou corrigir até 4 erros (um erro por linha) desde que haja somente um erro por coluna.

3.5. Código *Reed-Muller*

3.5.1. Introdução

Os códigos *Reed-Muller* são das famílias de código mais antigas, inventados em 1954 por *D. E. Muller* e *I.S. Reed*. Em 1971, a nave espacial *Mariner 9* utilizou um código *Reed-Muller* para transmitir fotografias em preto e branco de Marte. Os códigos *Reed-Muller* são relativamente fáceis de decodificar, e os códigos de 1ª ordem são muito eficientes [24] [25] [26].

3.5.2. Definição de Termos e Operações

O vetor espaço utilizado consiste em uma seqüência de *bits* de tamanho 2^m, onde *m* é um inteiro positivo, de números em $F_2 = \{0,1\}$. As Palavras-Código de um código *Reed - Muller* formam um subespaço de um espaço vetorial. Os vetores podem ser manipulados por três operações:

Adição:

Para os vetores $x = (x_1, x_2, ..., x_n)$ e $y = (y_1, y_2, ..., y_n)$ a adição é definida como:

$$\mathbf{x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)}$$

onde cada x_i ou y_i é 1 ou 0, e

$$\mathbf{1 + 1 = 0 \qquad 0 + 1 = 1 \qquad 1 + 0 = 1 \qquad 0 + 0 = 0}$$

A adição de um escalar $a \in F_2$ com o vetor x, é definido por:

$$\mathbf{a + x = (a + x_1, a + x_2, ..., a + x_n)}$$

O complemento do vetor x é o vetor igual a 1 + x.

Multiplicação:

A multiplicação é definida pela fórmula:

$$\mathbf{x * y = (x_1 * y_1, x_2 * y_2, \ldots, x_n * y_n)}$$

onde cada x_i ou y_i é 1 ou 0, e

$$\mathbf{1 * 1 = 1 \qquad 0 * 1 = 0 \qquad 1 * 0 = 0 \qquad 0 * 0 = 0}$$

A multiplicação de uma constante $a \in F_2$ com o vetor x, é definido por:

$$\mathbf{a * x = (a * x_1, a * x_2, ..., a * x_n).}$$

Produto Escalar:

O produto escalar de x e y é definido por

$$\mathbf{x} \cdot \mathbf{y} = (x_1 * y_1 + x_2 * y_2 + \ldots + x_n * y_n)$$

Essas três operações requerem vetores de mesmo tamanho.

Os vetores podem ser associados com polinômios Booleanos. Um polinômio Booleano é uma combinação linear de um monômio Booleano com coeficientes em F_2. Um monômio Booleano **p** das variáveis $x_1, \ldots, x_m$,é uma expressão na forma:

$$\mathbf{p} = \chi_1^{r_1} \chi_2^{r_2} \ldots \chi_m^{r_m} \text{ onde } r_i \in \{0, 1, 2, \ldots\} \text{ e } 1 \leq i \leq m.$$

A forma reduzida p' de p é obtida aplicando as regras:

$$\mathbf{x}_i\mathbf{x}_y = \mathbf{x}_y\mathbf{x}_i \text{ e } \mathbf{x}_i^2 = \mathbf{x}_i$$

até os fatores serem distintos. O grau de **p** é o grau ordinário de **p′**, que é o número de variáveis em **p′**. Um polinômio booleano está na forma reduzida se cada monômio está na forma reduzida. O grau do polinômio booleano **q** é o grau ordinário desta forma reduzida **q′**. Um exemplo de um polinômio booleano na forma reduzida com grau três é:

$$q = \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_1\mathbf{x}_2 + \mathbf{x}_2\mathbf{x}_3\mathbf{x}_4$$

Pode-se agora descrever como associar um monômio booleano em *m* variáveis para um vetor com 2^m entradas. O monômio de grau zero é 1 e o monômio de grau um é $x_1, x_2, \ldots, x_m$. . Primeiro nós definimos os vetores associados com esses monômios. O vetor associado com o monômio 1 é simplesmente um vetor de tamanho 2^m, onde todos os elementos do vetor são 1. Desta forma, num espaço de tamanho 2^3, o vetor associado com 1 é (11111111).

O vetor associado com o monômio x_1 é $2^m - 1$ uns, seguidos por $2^m - 1$ zeros. O vetor associado com o monômio x_2 é $2^m - 2$ uns seguidos de $2^m - 2$ zeros. Em geral, o vetor associado com o monômio x_i segue o padrão $2^m - 1$ uns seguidos de $2^m - 1$ zeros, repetindo – se até 2^m valores. Por exemplo, num espaço de tamanho 2^4, o vetor associado com x_4 é (1010101010101010).

Para formar o vetor a partir de um monômio $= \chi_1^{r_1} \chi_2^{r_2} \ldots$, primeiro coloca-se na forma reduzida. Depois, multiplicam-se os vetores associados com cada monômio x_i na forma reduzida. Por exemplo, num espaço com $m = 3$, o vetor associado com o monômio $x_1x_2x_3$ pode ser encontrado multiplicando (11110000) ∗ (11001100) ∗ (10101010) que resulta em (10000000).

Para formar um vetor a partir de um polinômio, basta reduzir o polinômio em monômios, e descobrir os vetores associados a cada uma dos monômios. Em seguida,

adicionam-se todos os vetores associados a cada uma destes monômios para formar o vetor associado com o polinômio. Isto dá-nos uma bijeção entre os polinômios reduzidos e os vetores.

3.5.3. Código *Reed-Muller* e Matriz de Codificação

Um código *Reed-Muller* de ordem *r* é o conjunto de todas as seqüências binárias (vetores) de comprimento $n = 2^m$ associados ao polinômio booleano $p(x_1, x_{2,\ldots}, x_m)$ de grau r. O código *Reed- Muller* de grau zero - $\Re(0, m)$ é constituído pela seqüência binária associada com as constantes polinomiais 0 e 1, isto é,

$$\Re(0, m) = \{0,1\} = \text{Rep}\,(2^m)$$

Assim, $\Re(0, m)$ é apenas uma repetição de zeros e uns, de comprimento 2^m. No outro extremo, o código *Reed-Muller* $\Re(m, m)$ consiste de todas as seqüências binárias de comprimento 2^m. Para definir a matriz de codificação $\Re(r, m)$, a primeira linha da matriz de codificação é 1, ou seja, o comprimento do vetor 2^m com todos os elementos iguais a 1. Se r é igual a 0, então, há somente uma linha na matriz de codificação. Se r é igual a 1, então se acrescentam *m* linhas correspondentes aos vetores $x_1, x_{2,\ldots}, x_m$ da matriz de codificação $\Re(r, m)$.

Para formar uma matriz de codificação $\Re(r, m)$ onde r é maior que 1, adiciona-se $\binom{m}{r}$ linhas na matriz de codificação $\Re(r\text{-}1, m)$. Essas linhas adicionadas consistem em todas as possibilidades possíveis de monômios na forma reduzida de grau r que poder ser obtidos usando as linhas $x_1, x_{2,\ldots}, x_m$. Por exemplo, quando m=3, têm-se:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \Re(1,3)$$

As linhas x_1x_2 = 11000000, x_1x_3 = 10100000, e x_2x_3 = 10001000 são adicionadas para formar:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ x_1x_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_2x_3 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \Re(2,3)$$

Finalmente, as linhas $x_1x_2x_3$ são adicionadas para formar:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ x_1x_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1x_3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_2x_3 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ x_1x_2x_3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Re(3,3)$$

Outro exemplo de uma matriz de codificação *Reed-Muller* é

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_3 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_4 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ x_1x_2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1x_3 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1x_4 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_2x_3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_2x_4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_3x_4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \Re(2,4)$$

Para codificar uma informação usando *Reed-Muller* é simples. O código a ser usado é $\Re$ (r, m). A sua dimensão é:

$$k = 1 + \binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{r}$$

Em outras palavras, a matriz codificação tem *k* linhas. As informações são armazenadas em blocos de comprimento *k*. Desta forma, m = (m_1, $m_{2,...}$, m_k) deve ser um bloco, e a informação codificada **M**c é:

$$M_c = \sum_{i=1}^{k} m_i R_i$$

Onde R*i* é a linha da matriz de codificação ℜ (*r*, *m*).

Por exemplo, usando ℜ (1,3) para codificar a informação m = (0110) temos:

0*(11111111)+1*(11110000)+1*(11001100)+0*(10101010) = (**00111100**).

3.5.4. Decodificação do Código *Reed-Muller*

O processo de decodificação das informações é mais complexo que o processo de codificação. A teoria em torno da codificação e decodificação é baseada na distância entre vetores. A distância entre quaisquer dois vetores é o número de lugares nos dois vetores que são diferentes entre si. A distância entre quaisquer duas palavras-código em ℜ (*r*, *m*) é calculada por $2^{m-r.}$ A base de codificação de *Reed-Muller* é o pressuposto de que a mais próxima palavra-código seja a informação original codificada. Assim, para que *e* erros sejam corrigidos na informação lida, a distância entre quaisquer duas palavras-código em ℜ (*r*, *m*) deve ser superior a 2*e*.

O método de decodificação usado não é muito eficiente, mas é simples de se implementar. Ele verifica cada linha da matriz de codificação, e usa a lógica majoritária para determinar se a linha foi utilizada na formação da mensagem codificação. Assim, é possível determinar o erro na mensagem codificada e qual foi a mensagem original. Este método de decodificação é dado pelo seguinte algoritmo: Aplicam-se os passos 1 e 2 abaixo para cada linha da matriz, começando de baixo para cima. Os passos são os seguintes:

Etapa 1. Escolha uma linha da matriz de codificação ℜ (*r*, *m*). Encontre 2^{m-r} vetores (este processo é descrito abaixo) para cada linha e, em seguida, determine o produto escalar de cada uma dessas linhas com as mensagens codificadas.

Etapa 2. Aplique a lógica majoritária nos valores resultantes do produto escalar.

Etapa 3. Após fazer os passos 1 e 2 para cada linha, exceto para a linha superior, multiplique cada coeficiente calculado na etapa 2 pela sua correspondente linha e adicione ao vetor resultante para formar **My.** Adicione este resultado a informação codificada lida. Se o vetor

resultante tem mais uns do que zeros e, a linha superior é 1, caso contrário é 0. Adicionando a linha superior, multiplicada pelos seus coeficientes, dá a informação original codificada. Assim, podemos identificar os erros. O vetor formado pela seqüência dos coeficientes a partir da primeira linha da matriz de codificação e termina na linha inferior corresponde a informação original.

Para descobrir os vetores característicos de qualquer linha da matriz, pegamos o monômio *r* associado com a cada linha da matriz de codificação. Em seguida, considere que E seja o conjunto de todos os x_i que não estão no monômio r, mas estão na matriz codificação. Os vetores característicos são os vetores correspondentes aos monômios em x_i e $\overline{x_i}$, de tal forma que exatamente um de x_i ou $\overline{x_i}$, ou seja, em cada monomial para todos os x_i em E. Por exemplo, a última linha da matriz codificação $\Re$ (*2,4*) está associada a x_3x_4, de modo que os vetores característicos correspondem às seguintes combinações de x_1, x_2; $\overline{x_1}$ e $\overline{x_2}$; x_1 x_2; x_1 $\overline{x_2}$; $\overline{x_1}$ x_2; $\overline{x_1}$ $\overline{x_2}$. Estes vetores característicos têm a propriedade de que o produto escalar é zero com todas as linhas de $\Re$ (*r*, *m*), exceto na linha que corresponde ao vetor característico.

3.5.5. Exemplo de codificação e decodificação

Se a informação original é **m** = (0110) usando $\Re$ (*1,3*), então a informação codificada é Mc = (00111100). A distância em $\Re$ (*1,3*) é de $2^{3-1} = 4$, então este código pode corrigir 1 erro. A informação codificada com erro é **Me** = (10111100): Os vetores característicos da última linha x_3 = (10101010) são x_1 x_2; x_1 $\overline{x_2}$; $\overline{x_1}$ x_2 e $\overline{x_1}$ $\overline{x_2}$. O vetor associado a x_1 é (11110000), de modo que $\overline{x_1}$ = (00001111). O vetor associado com x_2 é (11001100), de modo que $\overline{x_2}$ = (00110011). Por isso, temos x_1 x_2 = (11000000), x_1 $\overline{x_2}$ = (00110000), $\overline{x_1}$ x_2 = (00001100), e $\overline{x_1}$ $\overline{x_2}$ = (00000011). Determinando os produtos vetoriais destes vetores com **Me**, obtem-se:

(11000000) . (10111100) = 1, (00110000) . (10111100) = 0,
(00001100) . (10111100) = 0, (00000011) . (10111100) = 0.

Podemos concluir que o coeficiente de x_3 é 0. Fazendo o mesmo para a linha x_2 da matriz, x_2 é (11001100), obtemos os vetores característicos x_1 x_3; x_1 $\overline{x_3}$; $\overline{x_1}$ x_3 e $\overline{x_1}$ $\overline{x_3}$. Estes vetores são (10100000), (01010000), (00001010), e (00000101), respectivamente. Determinando os produtos vetoriais destes vetores com **Me**, obtemos:

(10100000) . (10111100) = 0, (01010000) . (10111100) = 1,
(00001010) . (10111100) = 1, (00000101) . (10111100) = 1.

Portanto, podemos concluir que o coeficiente de x_2 é 1. Fazendo o mesmo para a linha x_1 da matriz, x_1 é (11110000), obtemos:

(10001000) . (10111100) = 0, (00100010) . (10111100) = 1,
(01000100) . (10111100) = 1, (00010001) . (10111100) = 1.

Podemos concluir que o coeficiente de x_1 é também 1. Se acrescentarmos 0∗ (10101010) e 1∗ (11001100) e 1∗ (11110000), temos $\mathbf{M_y}$, que é igual a (00111100). Em seguida, vemos que a soma de $\mathbf{M_y}$ com **Me** é igual a

(00111100) + (10111100) = (10000000)

Esta informação possui mais 0's do que 1's, de modo que o primeiro coeficiente da linha da matriz de codificação é zero. Assim, podemos colocar os coeficientes das quatro linhas da matriz, 0, 1, 1, e 0, e vemos que a informação original era (0110). Podemos ver também que o erro estava na primeira posição da informação sem erro Mc = (00111100).

4. Técnica de Embaralhamento

A maioria dos códigos corretores de erro em uso são desenvolvidos para corrigirem erros aleatórios, isto é, erros que ocorrem de maneira independente da localização de outros erros. São apropriados para canais AWGN (*Additive White Gaussian Noise*). O termo AWGN é utilizado em modelos matemáticos para caracterizar aqueles canais onde o tipo de ruído responsável por degradar a comunicação, é um ruído branco adicionado ao sinal.

Contudo, em muitos canais ou sistemas, erros podem aparecer em rajadas. Por exemplo, num *compact disc*, um arranhão na mídia pode causar erros em vários *bits* consecutivos. Numa mídia magnética como um *Hard disk* ou uma fita, uma marca na superfície magnética pode introduzir vários erros [28]. Em memórias SRAM, partículas energizadas que atingem o silício podem causar vários erros nas informações armazenadas.

De uma maneira geral, códigos corretores de erros aleatórios não são muito eficientes para corrigir erros em rajadas e códigos corretores de erros em rajadas não são muito eficientes para corrigir erros aleatórios. Infelizmente, em muitos sistemas, podem ocorrer os dois tipos de erro: aleatório e em rajadas. Dos vários métodos propostos para corrigir simultaneamente os dois tipos de erros, o mais efetivo é o embaralhamento [29].

Esta técnica rearranja a ordem nas quais as palavras-código são armazenadas. Sem esta técnica, as mensagens m_1, m_2,... são codificadas e as respectivas palavras-códigos c_1, c_2,... são armazenadas em seqüência. Agora, imagine que as primeiras s palavras-códigos são selecionadas, e que o primeiro *bit* de cada s palavra-código são armazenados, depois o segundo *bit*, depois o terceiro *bit*, e assim por diante. Este rearranjo é chamado de embaralhamento de grau s[30]. Em resumo, esta técnica é essencialmente constituída por um reordenamento dos *bits* antes do armazenamento (Embaralhador) e na leitura os *bits* são novamente reordenados, ou seja, são colocados de volta para a posição original (desembaralhador).

O Embaralhador intercala os *bits* de tal modo que, se na informação armazenada, houver uma interferência concentrada (rajada de erros), na leitura, ao se fazer o desembaralhamento, os erros ficam distribuídos e aparecem como erros aleatórios ao decodificador. Nesta situação, o decodificador tem a sua capacidade de correção de erros

significativamente aumentada quando os erros ocorrem espaçados uns dos outros porque esta quantidade, muitas vezes, não excede a capacidade de correção de erros do decodificador.

Na Figura 4.1 temos três palavras-códigos *(C_1, C_2 e C_3)*, cada uma possuindo 3 *bits* e o resultado da aplicação do embaralhamento de grau três e da não aplicação do embaralhamento no barramento de dados de uma memória.

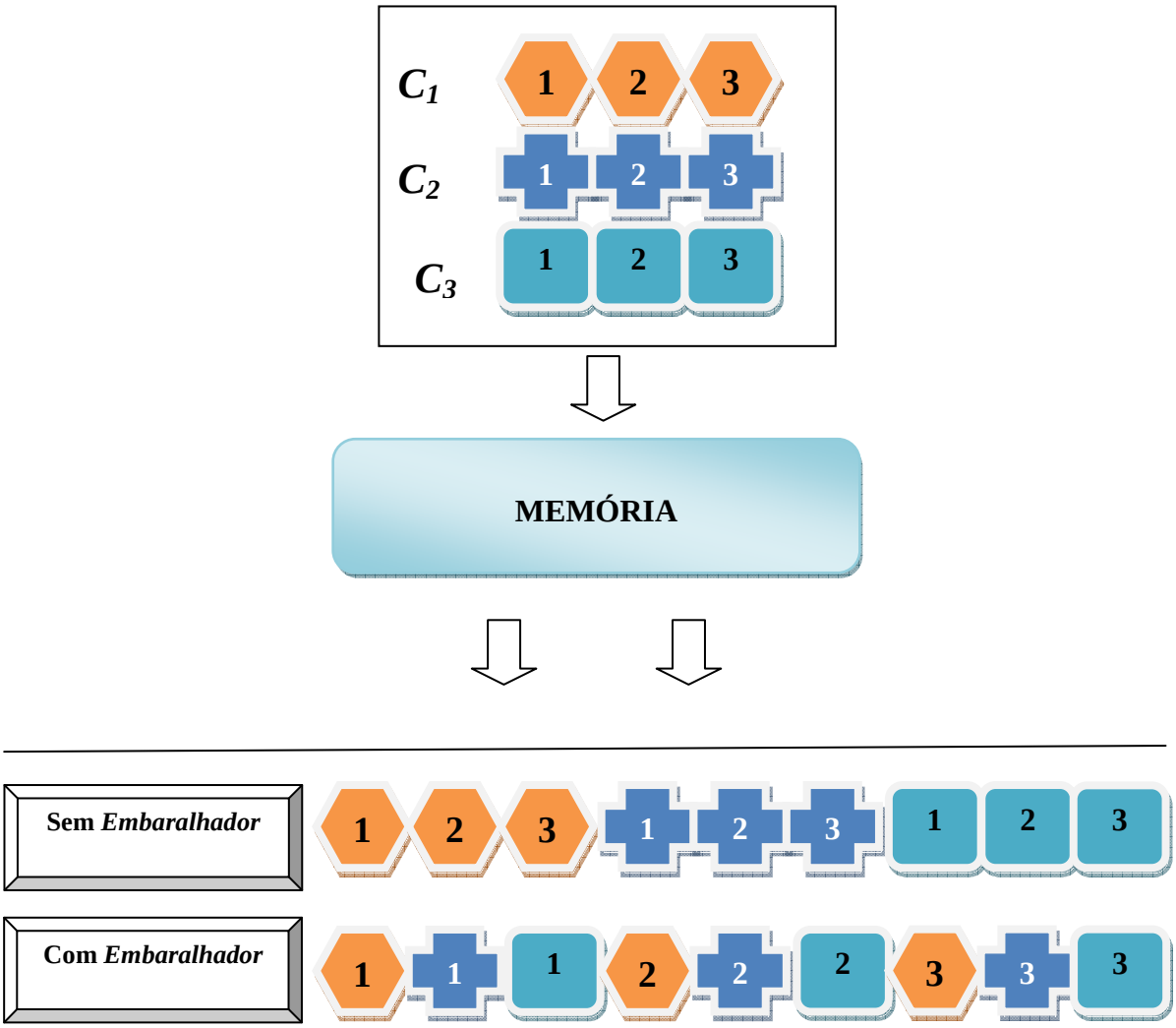


Figura 4.1 - Aplicação da técnica de embaralhamento.

Há dois tipos principais de embaralhadores: O Embaralhador de bloco e o Embaralhador convolucional. Como os nomes sugerem o Embaralhador de bloco e o Embaralhador convolucional são mais comumente usados para condicionar os dados de codificadores de blocos e codificadores convolucionais, respectivamente[31].

A técnica de embaralhamento de Blocos pode ser utilizada, então, para redistribuir os erros de maneira mais uniforme e aleatória entre os blocos codificados e aumentar a eficiência

no processo de decodificação. Uma forma de se obter o entrelaçamento é armazenar os blocos codificados como linhas em uma matriz. Quando esta matriz estiver completa uma leitura em colunas é realizada, redistribuindo os dados antes de escrevê-los na memória. Na leitura, o processo inverso é realizado também através da utilização da matriz. Desta vez, porém, erros agrupados em rajadas serão redistribuídos entre os blocos antes da decodificação

Pode-se observar que a seqüência de bits superior é inserida na matriz, preenchendo-se cada uma das linhas. Na etapa seguinte, os bits são extraídos a partir das colunas, gerando a seqüência inferior que será armazenada na memória. Repetindo-se este processo na leitura, a seqüência original será recuperada, enquanto os erros agrupados serão redistribuídos.

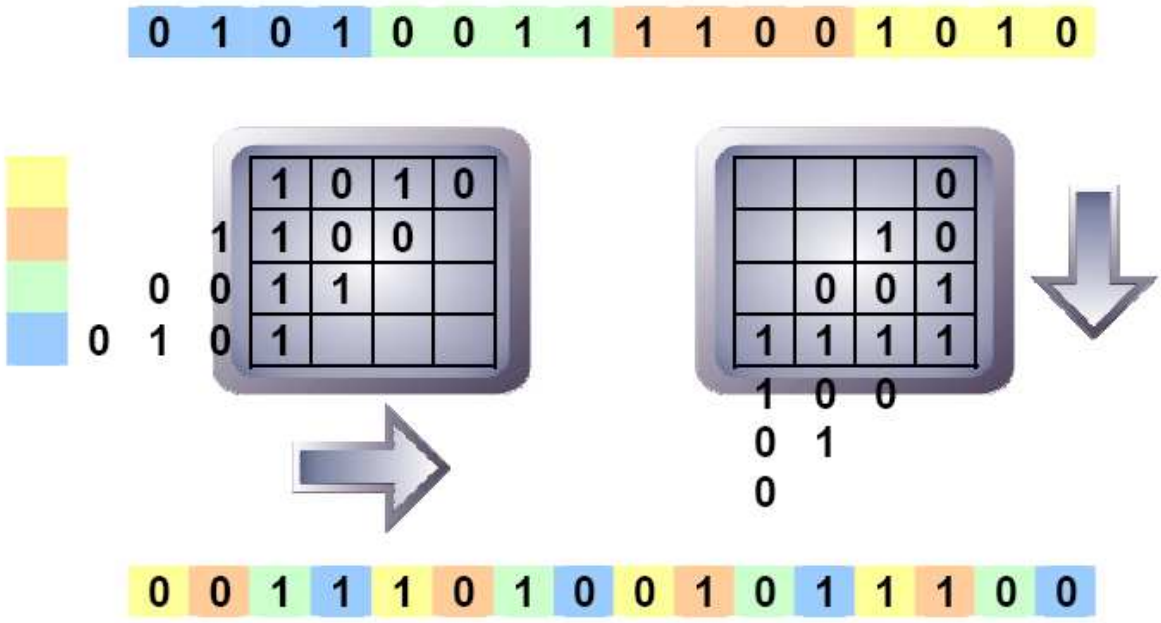


Figura 4.2 - Mecanismo de entrelaçamento de blocos.

O embaralhador convolucional é considerado o tipo de intercalação ideal [46] para funcionar em associação com a codificação e a decodificação de códigos convolucionais. O embaralhador convolucional é feito atrasando-se *bits* da seqüência de dados a serem armazenados em relação aos demais *bits*, afastando-os dos seus bits adjacentes durante a etapa de armazenamento na memória. O tipo de embaralhador convolucional de *bits* mais comumente utilizado é composto por um número *M* de memórias tipo *shift register* utilizadas como FIFOs (*First In First Out*) e dispostas em paralelo, conforme a ilustração da Figura 4.3. Cada uma destas FIFOs tem uma quantidade *T* de posições para armazenamento de *bits* a mais que a FIFO anterior. É usual adotar *T* =1 como também adotar que a primeira

"memória" não tenha nenhuma posição de armazenamento de *bits* (um condutor simples, portanto). As Entradas das FIFOs são "alimentadas" com *bits*, da seqüência de dados a ser armazenada, através de uma chave rotatória cíclica, que funciona em sincronismo com uma segunda chave, de mesmas características, que "recolhe" os *bits* na saída de cada uma das filas.

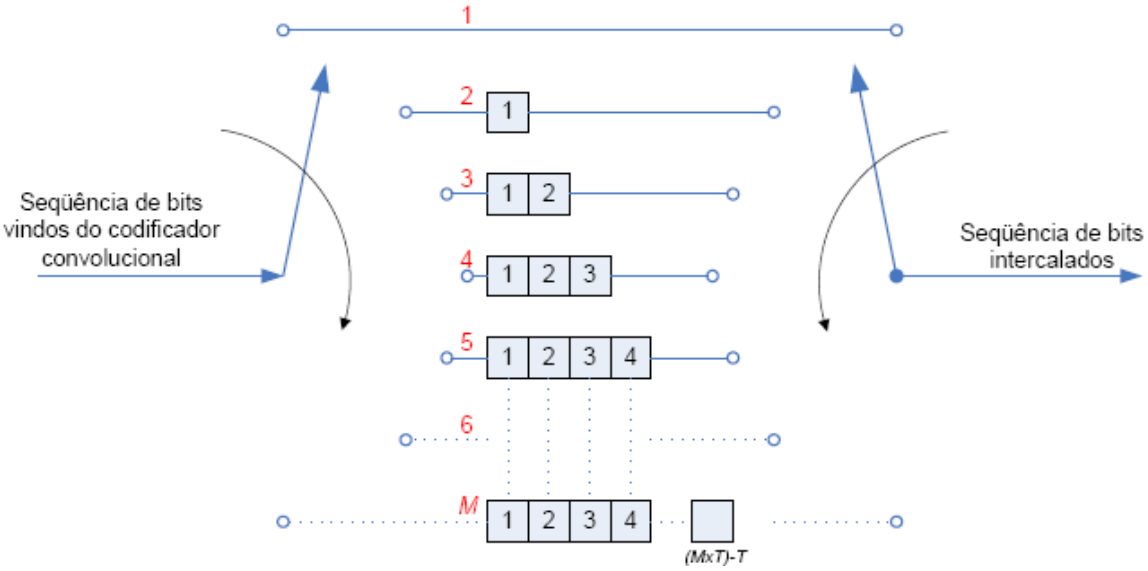


Figura 4.3 - Mecanismo de entrelaçamento convolucional.

Parte II - Metodologia

5. Desenvolvimento

5.1. Códigos e Ferramentas Utilizadas

A tabela 5.1 apresenta os códigos de detecção e correção de erros desenvolvidos em VHDL para a proteção das informações em memórias RAM.

Tabela 5.1 - EDAC's implementados em VHDL.

EDAC	**Número de Informações (*Bits*)**		
Hamming	4	8	16
***Hamming* + Paridade**	4	8	16
Reed- Muller	4	16	
Matrix Code	4	16	

Foram utilizadas as ferramentas de sintetização e simulação da ALTERA (QUARTUS II 7.1) [47] e XILINX (XILINX ISE 7.1i) [48]. Todos os códigos foram desenvolvidos em VHDL padrão, não utilizando nenhuma função específica da XILINX ou ALTERA, visando tornar o código VHDL portável para qualquer um dos fabricantes, assim como para uma eventual prototipação em ASIC.

5.2. Arquitetura de 32 *Bits* em Blocos

O trabalho foi desenvolvido baseado em uma arquitetura de 32 *bits*. O presente trabalho também pode ser aplicado a outras arquiteturas (por exemplo: 64 *bits* e 128 *bits*) bastando algumas alterações de *hardware* (número de blocos, dimensão do embaralhador, barramento de entrada e saída de dados, etc).

Os EDAC's utilizados são de 4, 8 e 16 *bits*. A informação a ser protegida é de 32 *bits*. Para utilizar EDAC's de tamanho inferior a 32 *bits* e também para aplicar a técnica de embaralhamento, a informação de 32 *bits* é segmentada em blocos. O número de blocos

depende do EDAC utilizado. A partir dessa segmentação em blocos, essa estrutura resultante será chamada de código combinado.

$$N^{o}_{\text{Blocos}} = \frac{Bits\ \ a\ \ serem\ \ protegidos}{N^{o}\ de\ \ bits\ \ de\ \ Entrada\ \ do\ \ EDAC\ \ Utilizado} \qquad (5.1)$$

O número de blocos deve ser um número inteiro maior ou igual a dois para permitir a utilização da técnica de embaralhamento ($N^{o}_{Blocos} \geq 2$). A técnica de embaralhamento é aplicada dentro de cada palavra de 32 *bits* e não entre palavras de 32 *bits*. Assume-se também que os códigos apresentados na tabela 5.2 serão segmentados em "n" blocos. Neste caso, a palavra de informação, que é de 32 bits e que está segmentada, por exemplo, em 8 blocos de 4 bits, o grau de segmentação é 8, calculado conforme equação 5.1.

Tabela 5.2 - Nº de blocos em função dos *bits* de entrada do EDAC utilizado.

Nº de *Bits* de entrada do EDAC utilizado	Nº de Blocos
4	**8**
8	**4**
16	**2**

Por exemplo, se for utilizado um código de *Hamming* de 8 bits, haverá, conforme a equação 5.1, quatro blocos. Desta forma, serão utilizados quatro CODEC's (Codificadores e Decodificadores) de *Hamming* de 8 *bits*. Nas Figuras 5.1, 5.2 e 5.3 temos a arquitetura da segmentação (e o processo inverso) da palavra de 32 *bits* em 2, 4 e 8 blocos, respectivamente.

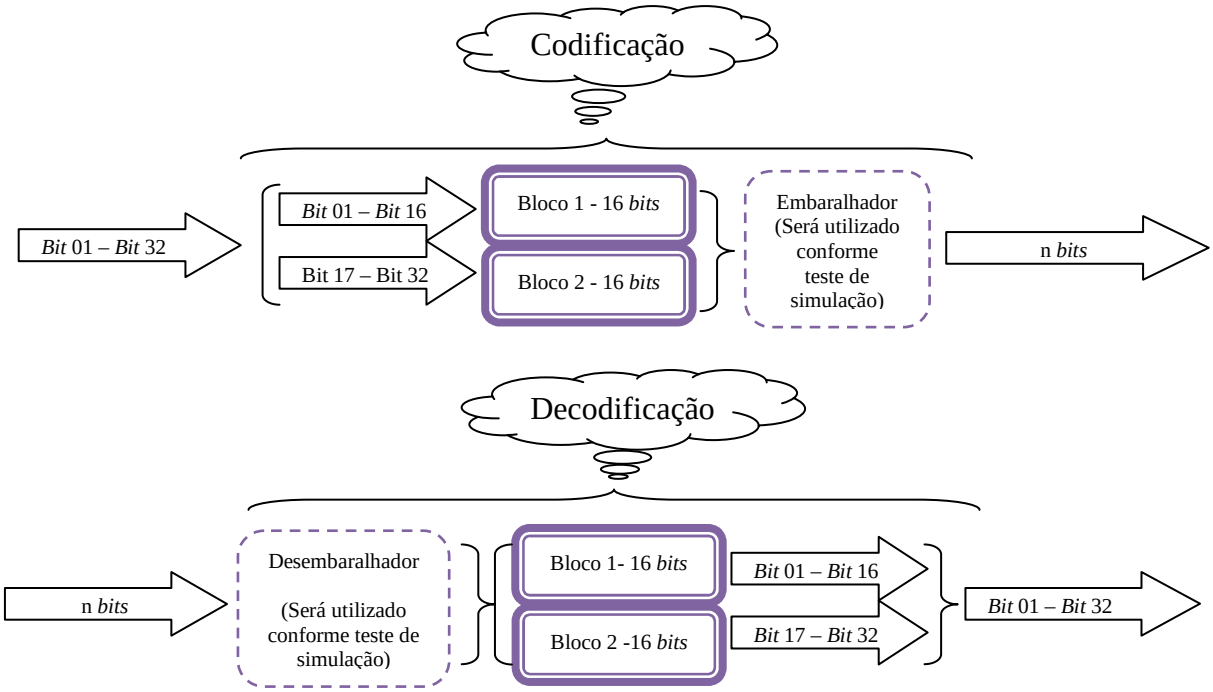


Figura 5.1 - Arquitetura de segmentação utilizando blocos de 16 *bits*.

Figura 5.2 - Arquitetura de segmentação utilizando blocos de 8 *bits*.

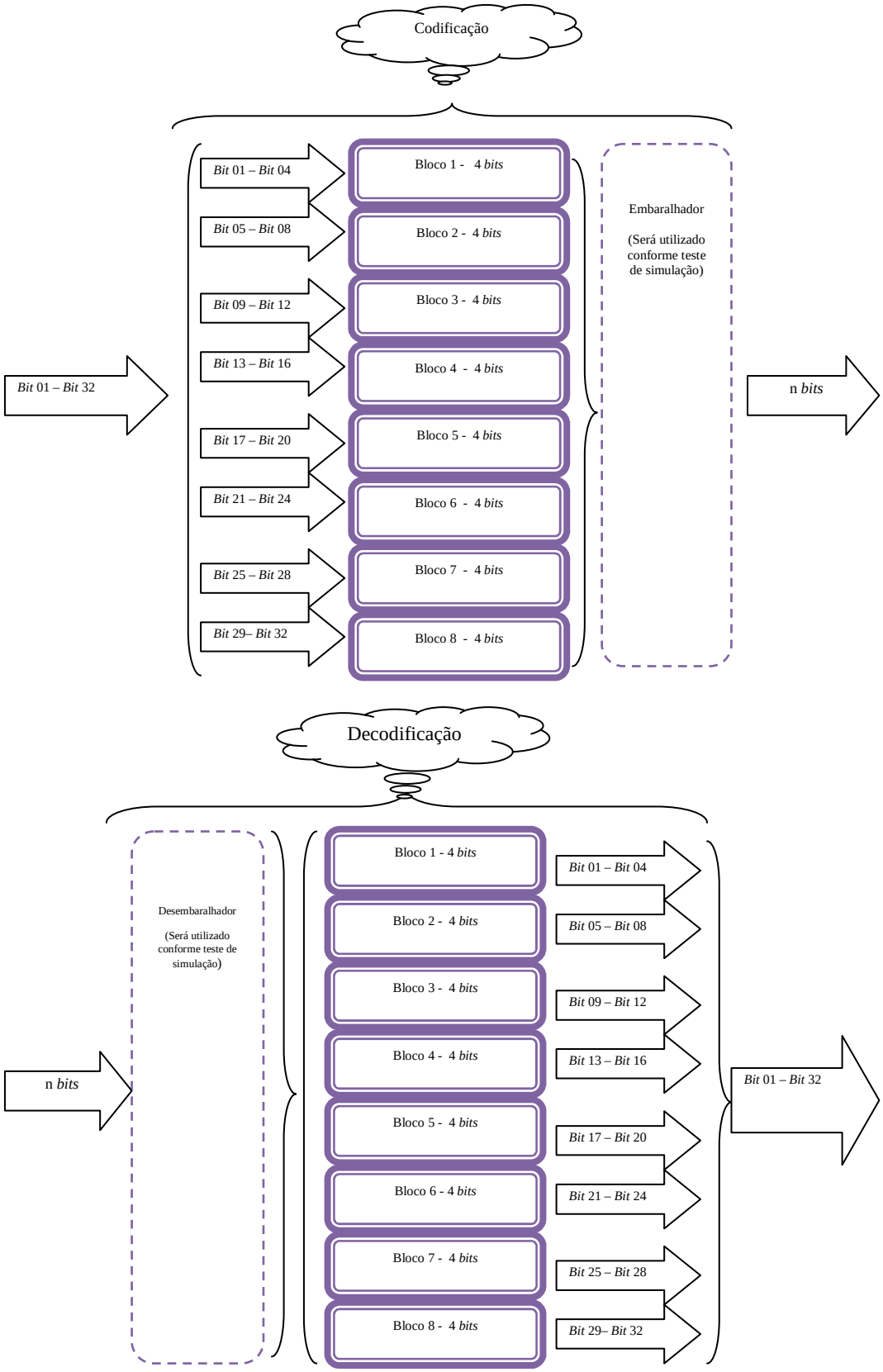


Figura 5.3 - Arquitetura de segmentação utilizando blocos de 4 *bits*.

5.3. Testes Realizados para cada código

5.3.1. *Hamming*

Foram desenvolvidos três CODECs (codificadores e decodificadores) de *Hamming*: *Hamming* (7,4), *Hamming* (12,8) e *Hamming* (21,16). A tabela 5.3 mostra algumas características destes códigos tais como: *bits* de entrada (k), *bits* de saída(n), *overhead* e grau de embaralhamento (somente no combinado com embaralhador) para cada código básico e também para a forma combinada (com e sem *embaralhador*).

Tabela 5.3 - Informações dos códigos de *Hamming*.

	Básico			Combinado Sem Embaralhador			Combinado Com Embaralhador			
Código *Hamming*	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	Grau de Embaralhamento
HM(7,4)	4	7	75	32	56	75	32	56	75	8
HM (12,8)	8	12	50	32	48	50	32	48	50	4
HM (21,16)	16	21	31,25	32	42	31,25	32	42	31,25	2

Na tabela 5.3 podemos observar que a aplicação da técnica de embaralhamento não altera o tamanho final do código combinado. A Figura 5.4 mostra a representação em bloco do CODEC do código de *Hamming* (*n*,*k*), os pinos de entrada e saída e a descrição de cada pino.

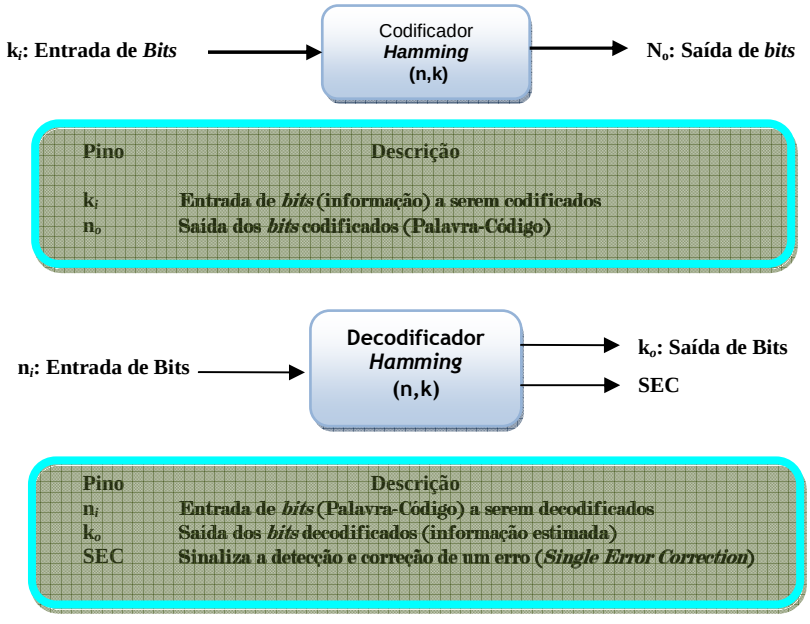


Figura 5.4 - Representação em bloco do código de *Hamming*.

5.3.1.1. *Hamming* (7,4)

A Figura 5.5 mostra a arquitetura do codificador do código de *Hamming* Combinado (HM (7,4)) e o bloco equivalente.

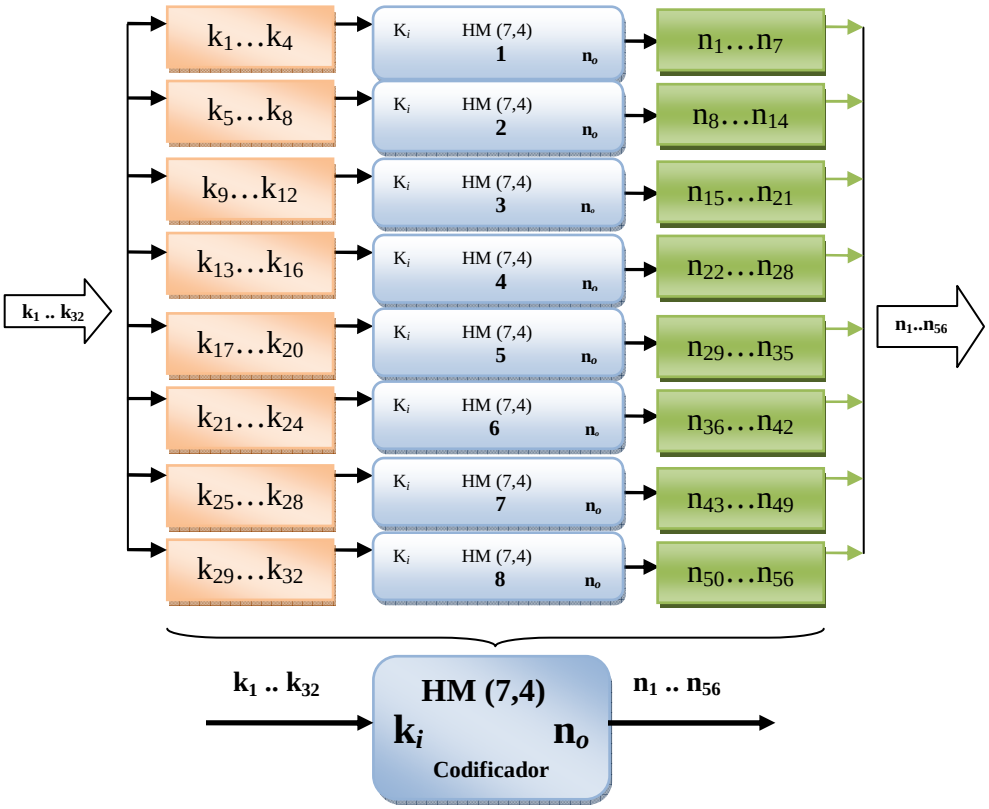


Figura 5.5 - Arquitetura do codificador do código Combinado de *Hamming* HM (7,4) e o bloco equivalente.

O codificador do código combinado de *Hamming* (7,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 8 seqüências de *bits*, cada uma com 4 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_4$, $k_5 \ldots k_8$, … e $k_{29} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de *hamming*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{56}$).

A Figura 5.6 mostra a arquitetura do decodificador do código Combinado de *Hamming* HM (7,4) e o bloco equivalente.

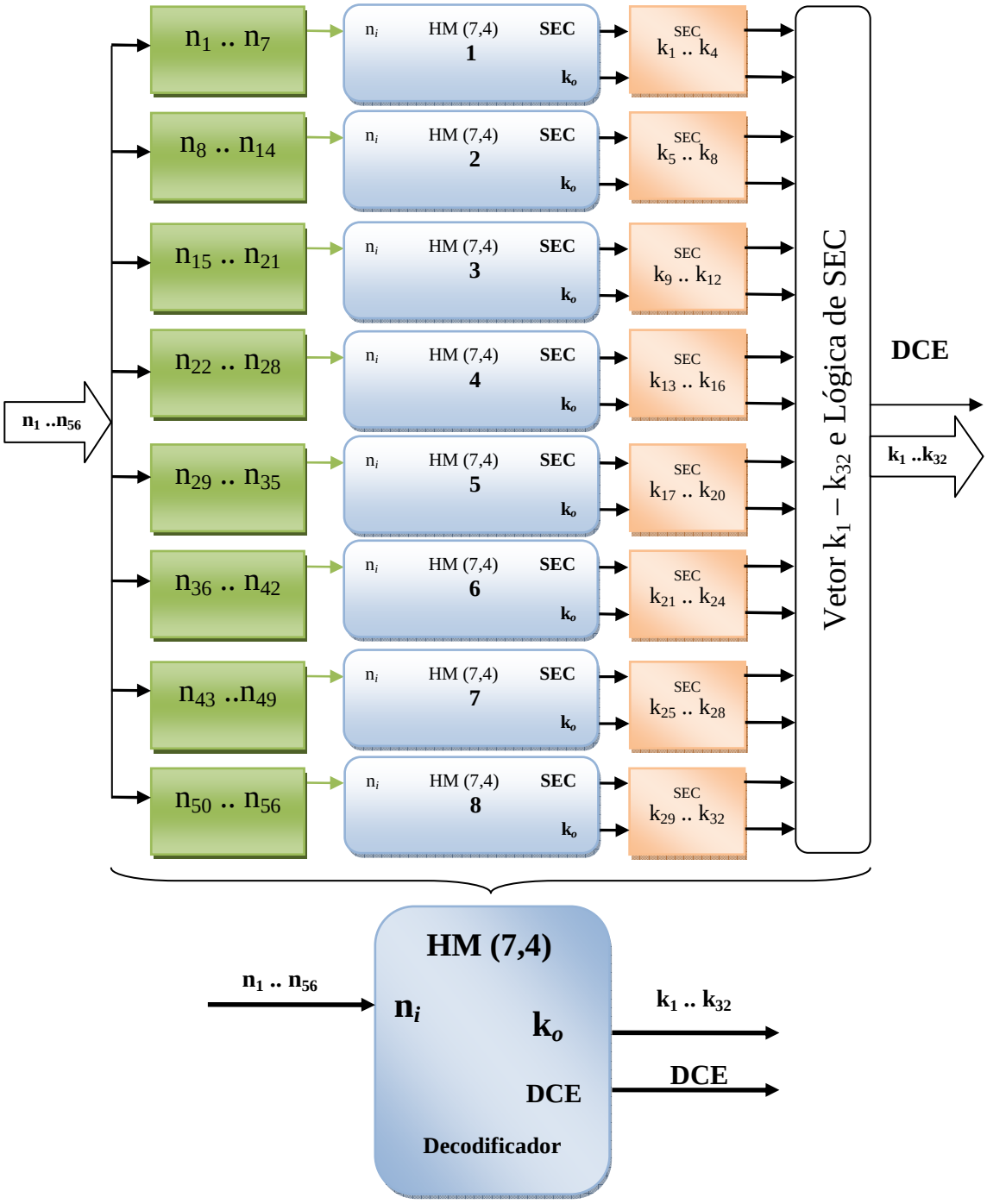


Figura 5.6 - Arquitetura do decodificador do código Combinado de *Hamming* HM (7,4) e o bloco equivalente.

O decodificador HM (7,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada (n_1.. n_{56}) é dividido em 8 seqüências de *bits*, cada uma com 7 *bits*;
- Cada seqüência de *bits* (n_1…n_7, n_8…n_{14}, … e n_{50}…n_{56}) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de *hamming*, segunda seqüência para o segundo decodificador e assim por diante;

- O resultado de cada decodificador (junto com o sinal SEC) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro (DCE).

5.3.1.2. *Hamming* (12,8)

A Figura 5.7 mostra a arquitetura do codificador do código Combinado de *Hamming* HM (12,8) e o bloco equivalente.

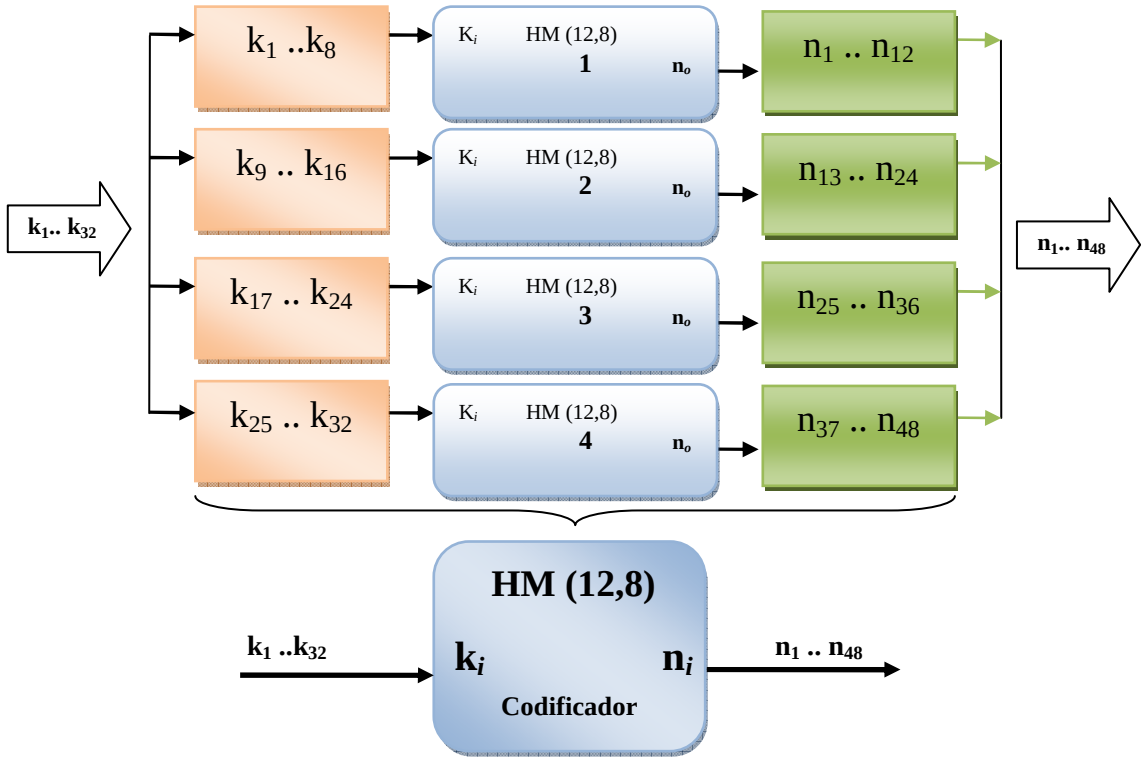


Figura 5.7 - Arquitetura do codificador do código Combinado de *Hamming* HM (12,8) e o bloco equivalente.

O codificador do código combinado de *Hamming* (12,8) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 4 seqüências de *bits*, cada uma com 8 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_8$, $k_9 \ldots k_{16}$, $k_{17} \ldots k_{24}$ e $k_{25} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de *hamming*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{48}$).

A Figura 5.8 mostra a arquitetura do decodificador do código Combinado de *Hamming* HM (12,8) e o bloco equivalente.

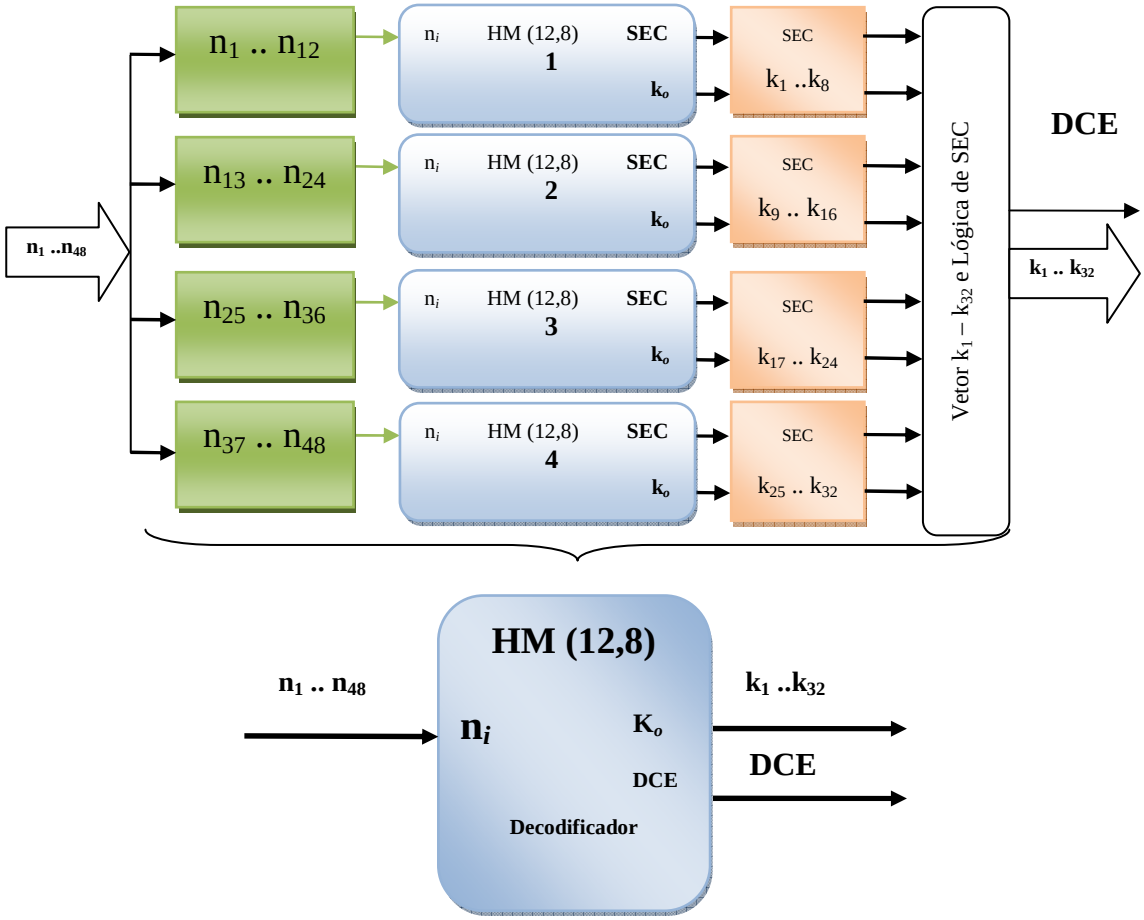


Figura 5.8 - Arquitetura do decodificador do código Combinado de *Hamming* HM (12,8) e o bloco equivalente.

O decodificador HM (12,8) funciona da seguinte maneira:

- O vetor de *bits* de entrada (n_1...n_{48}) é dividido em 4 seqüências de *bits*, cada uma com 12 *bits*;
- Cada seqüência de *bits* (n_1...n_{12}, n_{13}...n_{24}, n_{25}...n_{36} e n_{37}...n_{48}) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de *hamming*, segunda seqüência para o segundo decodificador e assim por diante;
- O resultado de cada decodificador (junto com o sinal SEC) é entregue ao bloco responsável por compor o vetor de saída (k_1...k_{32}) e também para sinalizar se houve detecção e correção de erro (DCE).

5.3.1.3. *Hamming* (21,16)

A Figura 5.9 mostra a arquitetura do codificador do código Combinado de *Hamming* HM (21,16) e o bloco equivalente.

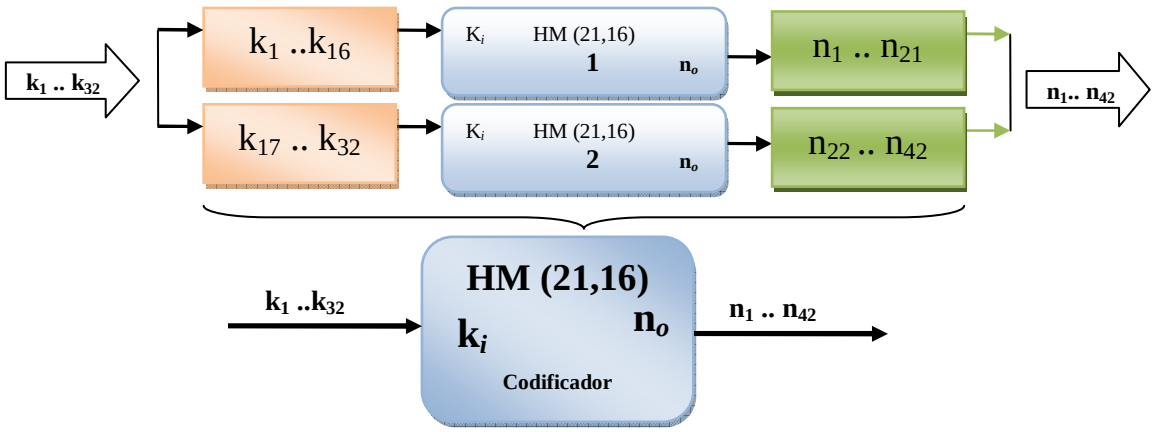


Figura 5.9 - Arquitetura do codificador do código Combinado de *Hamming* HM (21,16) e o bloco equivalente.

O codificador do código combinado de *Hamming* (21,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 2 seqüências , cada uma com 16 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_{16}$, $k_{17} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de *hamming* e segunda seqüência para o segundo codificador ;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{42}$).

A Figura 5.10 mostra a arquitetura do decodificador do código Combinado de *Hamming* (HM 21,16) e o bloco equivalente.

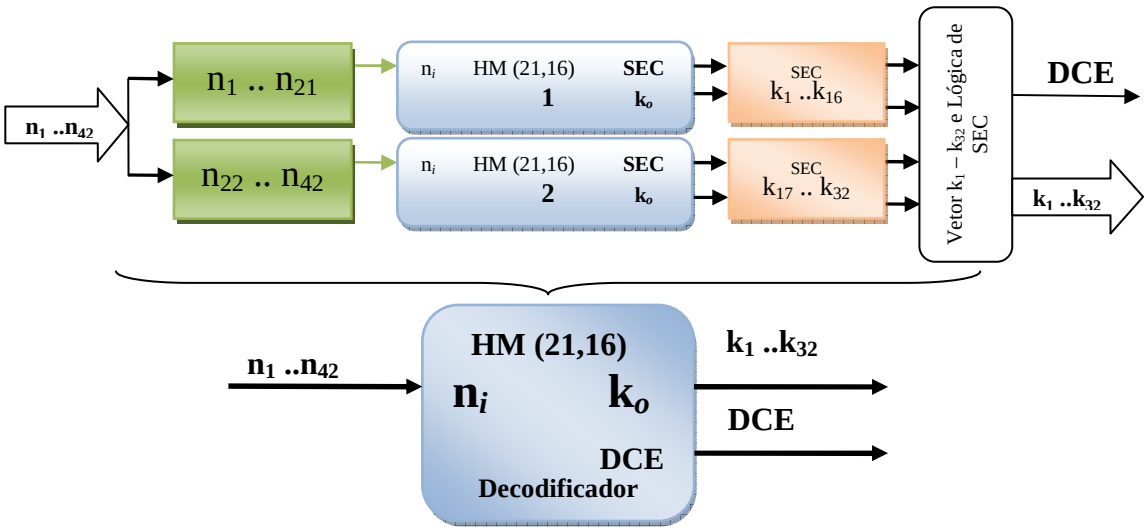


Figura 5.10 - Arquitetura do decodificador do código Combinado de *Hamming* HM (21,16) e o bloco equivalente.

O decodificador do HM (21,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{42}$) é dividido em 2 seqüências , cada uma com 21 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_{21}$ e $n_{22} \ldots n_{42}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de *hamming* e segunda seqüência para o segundo decodificador;
- O resultado de cada decodificador (junto com o sinal SEC) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro (DCE).

5.3.2. *Hamming* + Paridade

Foram desenvolvidos três CODECs (codificadores e decodificadores) de *Ex-Hamming*: *Ex-Hamming* (8,4), *Ex-Hamming* (13,8) e *Ex-Hamming* (22,16). A tabela 5.4 mostra algumas características destes códigos tais como: *bits* de entrada (k), *bits* de saída(n) e *overhead* para cada código básico e também para a forma combinada (com e sem *embaralhador*).

Tabela 5.4 - Informações dos códigos de *Ex-Hamming*.

	Básico			Combinado Sem Embaralhador			Combinado Com Embaralhador			
Código *Ex-Hamming*	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	Grau de Embaralhamento
EHM(8,4)	4	8	100	32	64	100	32	64	100	8
EHM (13,8)	8	13	62,50	32	52	62,50	32	52	62,50	4
EHM (22,16)	16	22	37,50	32	44	37,50	32	44	37,50	2

Na tabela 5.4 podemos observar que a aplicação da técnica de embaralhamento não altera o tamanho final do código combinado. A Figura 5.11 mostra a representação em bloco do CODEC do código de *Ex-Hamming* (*n*,*k*), os pinos de entrada e saída e a descrição dos mesmos.

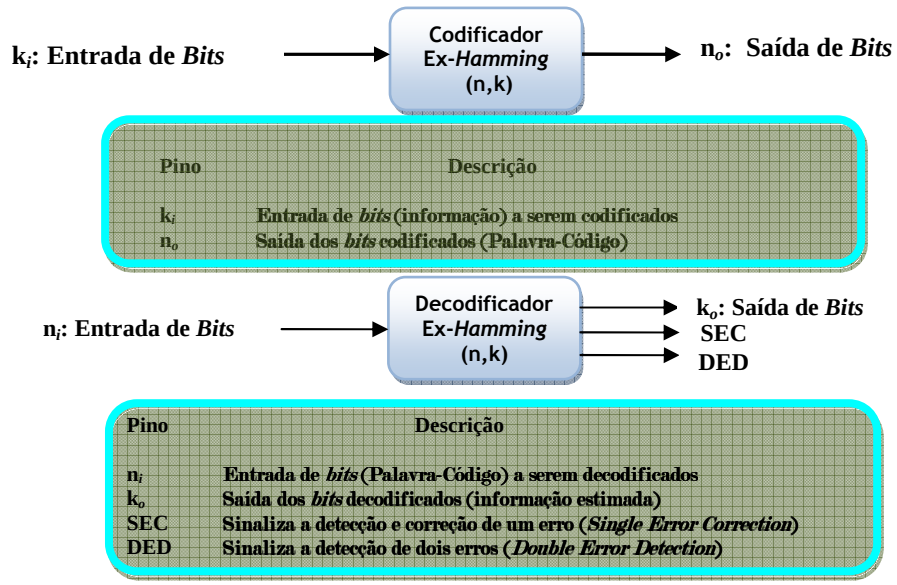


Figura 5.11 - Representação em bloco do código de *Ex-Hamming*.

5.3.2.1. *Ex-Hamming* (8,4)

A Figura 5.12 mostra a arquitetura do codificador do código Combinado de *Ex-Hamming* EHM (8,4) e o bloco equivalente.

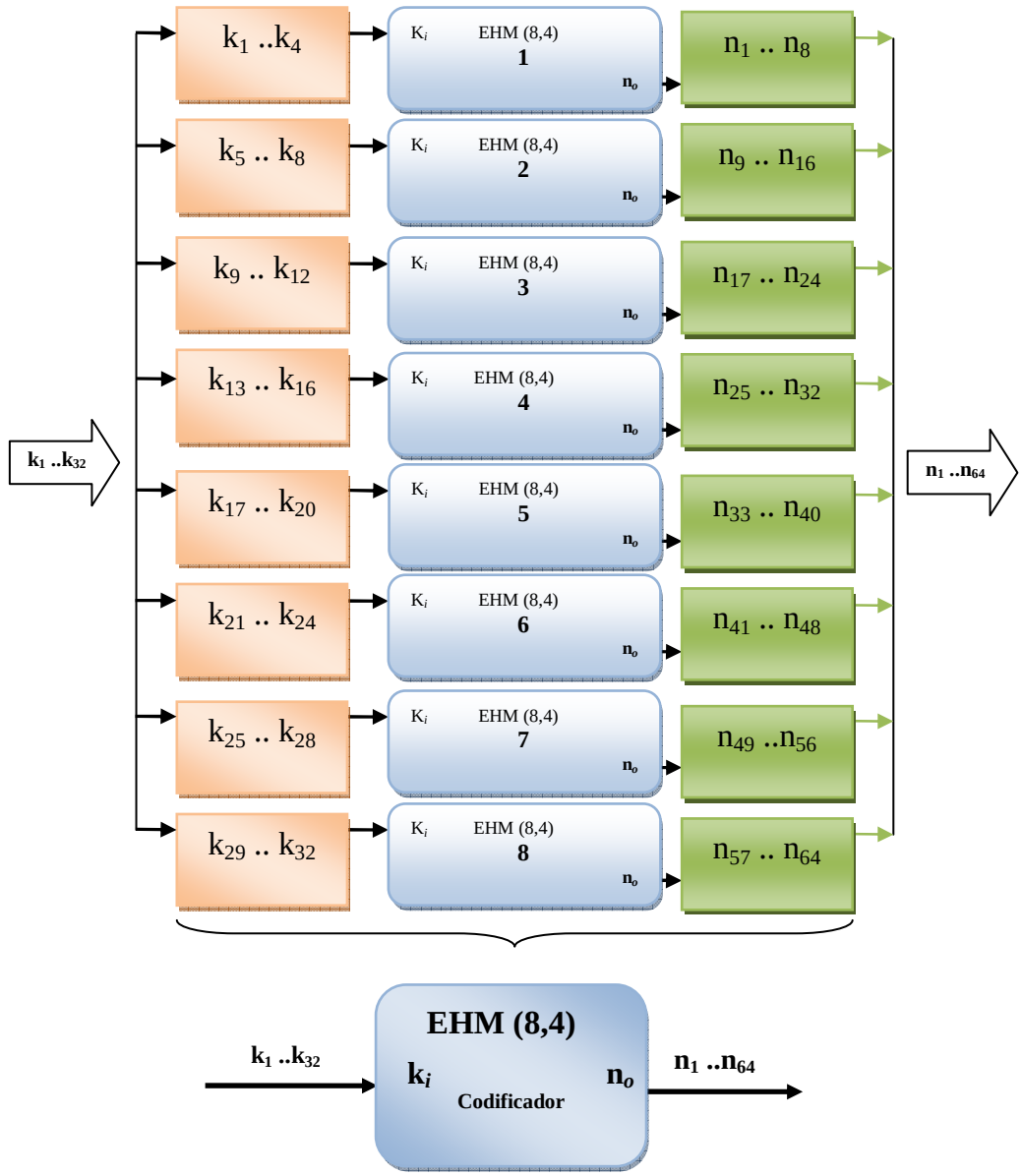


Figura 5.12 - Arquitetura do codificador do código Combinado *Ex-Hamming* EHM (8,4) e o bloco equivalente.

O codificador do código combinado *Ex-Hamming* (8,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 8 seqüências de *bits*, cada uma com 4 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_4$, $k_5 \ldots k_8$, …e $k_{29} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de *Ex-hamming*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{64}$).

A Figura 5.13 mostra a arquitetura do decodificador do código Combinado de *Ex-Hamming* EHM (8,4) e o bloco equivalente.

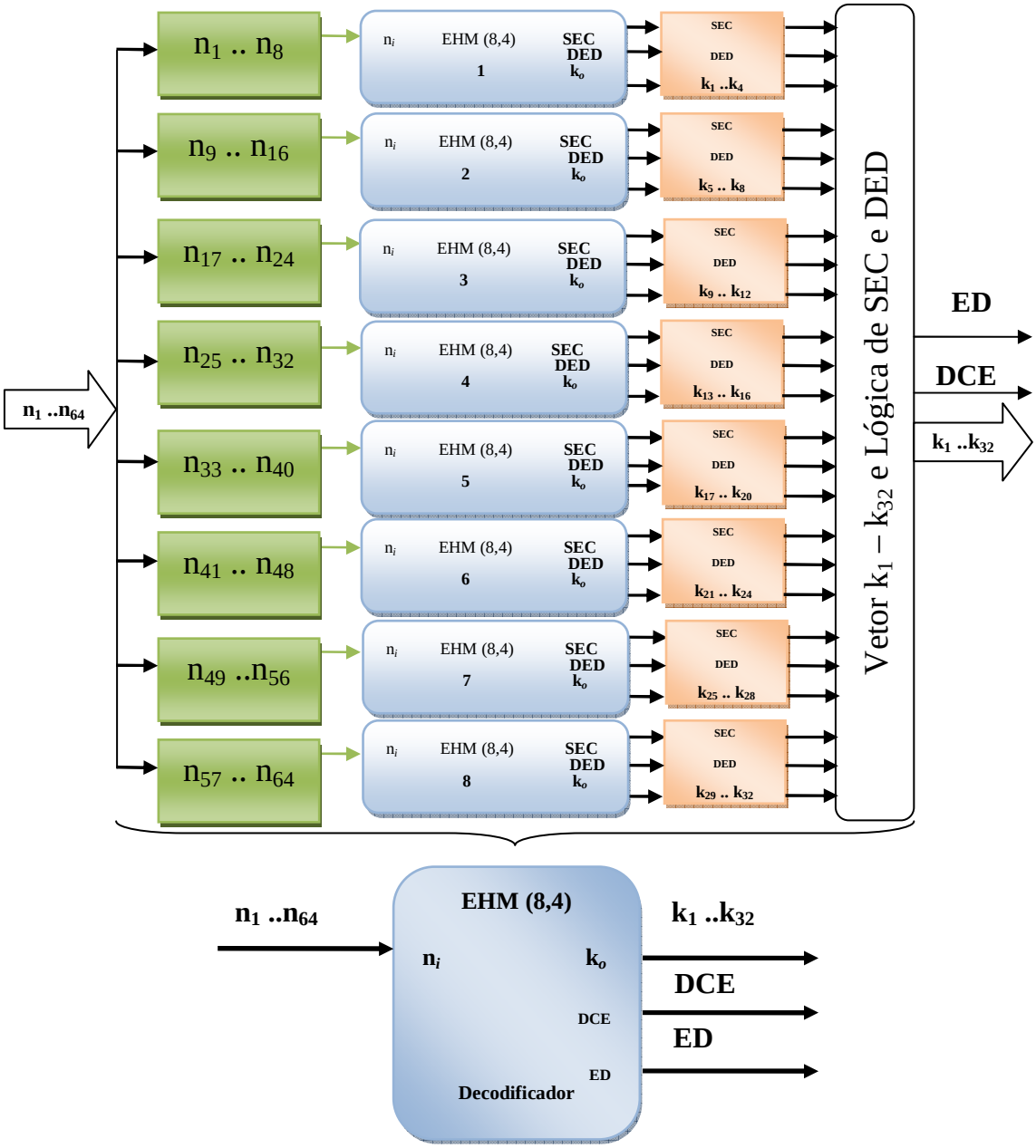


Figura 5.13 - Arquitetura do decodificador do código Combinado Ex-*Hamming* (EHM (8,4)) e o bloco equivalente.

O decodificador EHM (8,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{64}$) é dividido em 8 seqüências de *bits*, cada uma com 8 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_8$, $n_9 \ldots n_{16}$, …e $n_{57} \ldots n_{64}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de *Ex-hamming*, segunda seqüência para o segundo decodificador e assim por diante;

- O resultado de cada decodificador (junto com o sinal SEC e DED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro (DCE) ou Detecção de erro (ED).

5.3.2.2. Ex *Hamming* (13,8)

A Figura 5.14 mostra a arquitetura do codificador do código Combinado de *ex-Hamming* EHM (13,8) e o bloco equivalente.

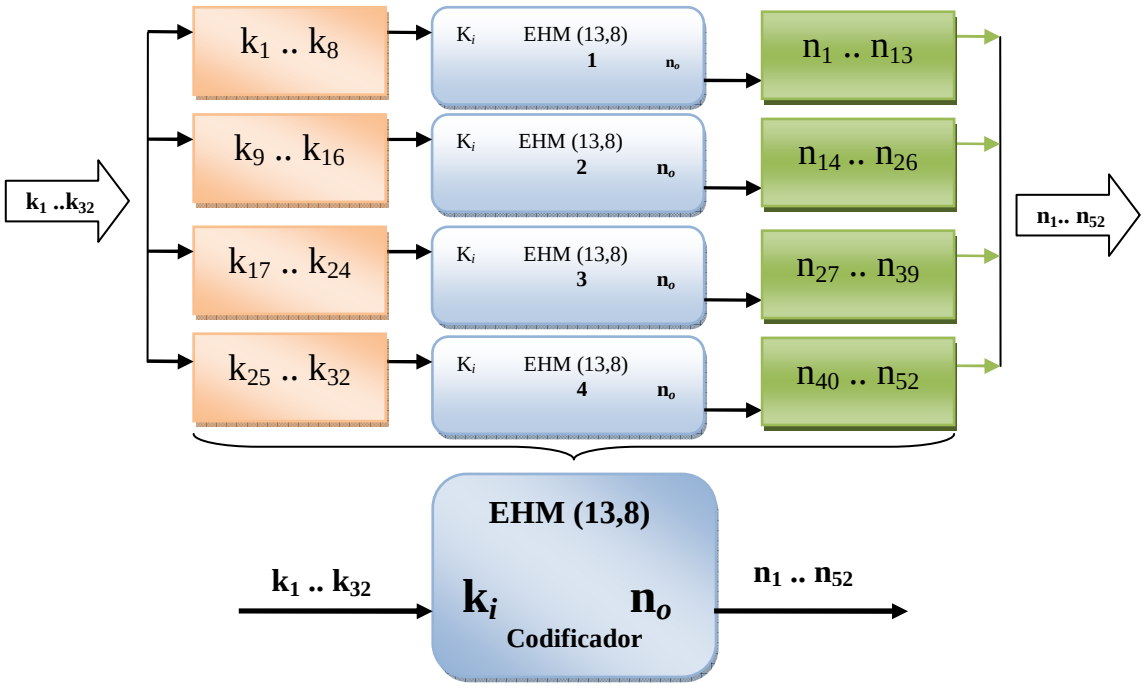


Figura 5.14- Arquitetura do codificador do código Combinado *Ex-Hamming* EHM (13,8) e o bloco equivalente.

O codificador do código combinado de Ex-*Hamming* (13,8) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 4 seqüências de *bits*, cada uma com 8 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_8$, $k_9 \ldots k_{16}$, $k_{17} \ldots k_{24}$ e $k_{25} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de Ex-*hamming*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{52}$).

A Figura 5.15 mostra a arquitetura do decodificador do código Combinado *ex-Hamming* EHM (13,8) e o bloco equivalente.

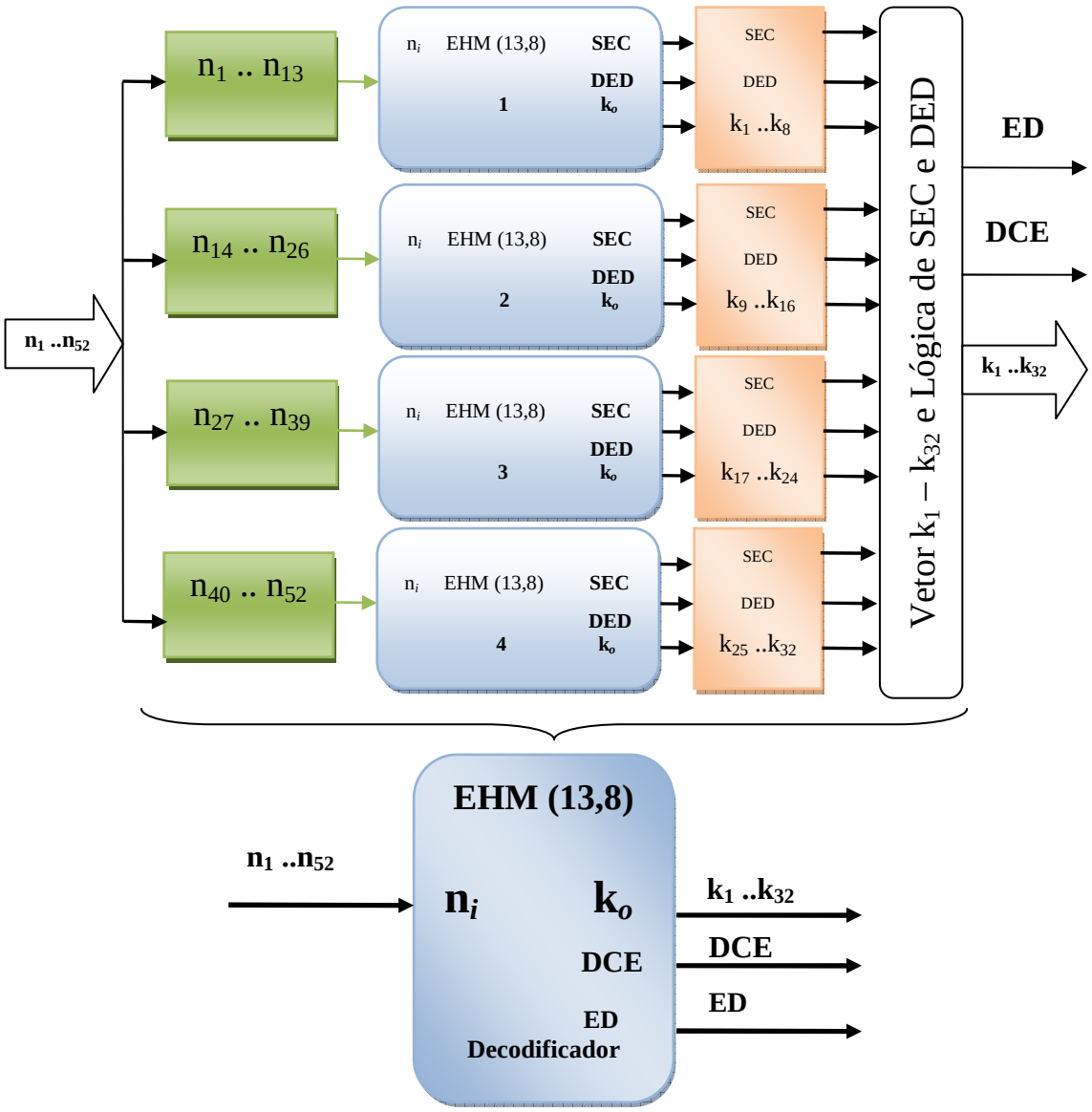


Figura 5.15 - Arquitetura do decodificador do código Combinado EHM (13,8) e o bloco equivalente.

O decodificador EHM (13,8) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{52}$) é dividido em 4 seqüências de *bits*, cada uma com 13 *bits*;
- Cada seqüência de bits ($n_1 \ldots n_{13}$, $n_{14} \ldots n_{26}$, $n_{27} \ldots n_{39}$ e $n_{40} \ldots n_{52}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de *Ex-hamming*, segunda seqüência para o segundo decodificador e assim por diante;
- O resultado de cada decodificador (junto com o sinal SEC e DED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro (DCE) ou detecção de erro (ED).

5.3.2.3. Ex *Hamming* (22,16)

A Figura 5.16 mostra a arquitetura do codificador do código Combinado Ex-*Hamming* EHM (22,16) e o bloco equivalente.

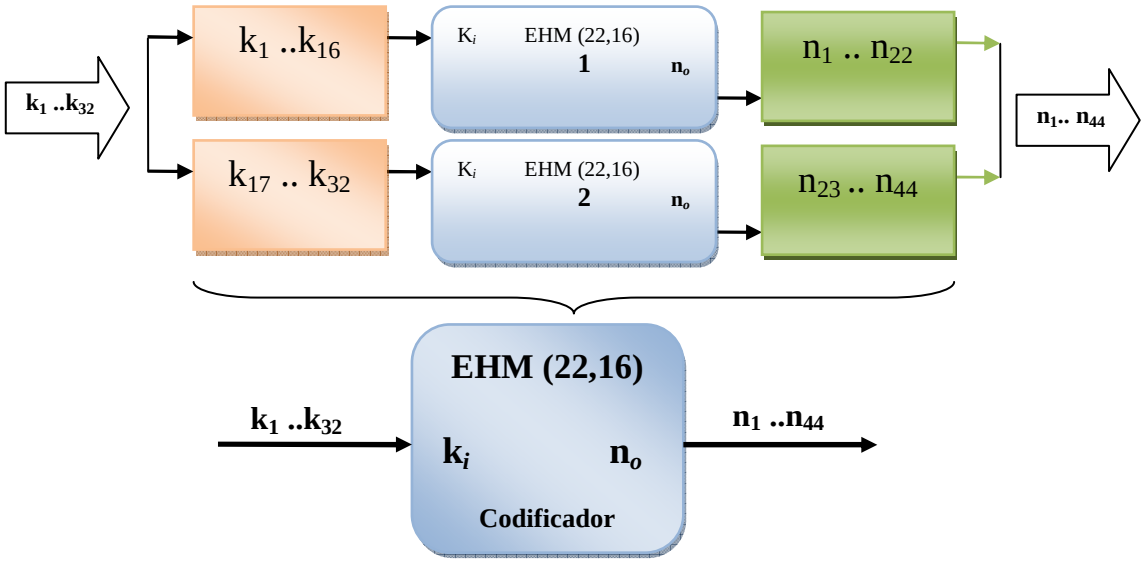


Figura 5.16- Arquitetura do codificador do código Combinado Ex-*Hamming* EHM (22,16) e o bloco equivalente.

O codificador do código combinado de Ex-*Hamming* (22,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 2 seqüências de *bits*, cada uma com 16 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_{16}$ e $K_{17} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador de Ex-*Hamming* e segunda seqüência para o segundo codificador ;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{44}$).

A Figura 5.17 mostra a arquitetura do decodificador do código Combinado *Ex-Hamming* EHM (22,16) e o bloco equivalente.

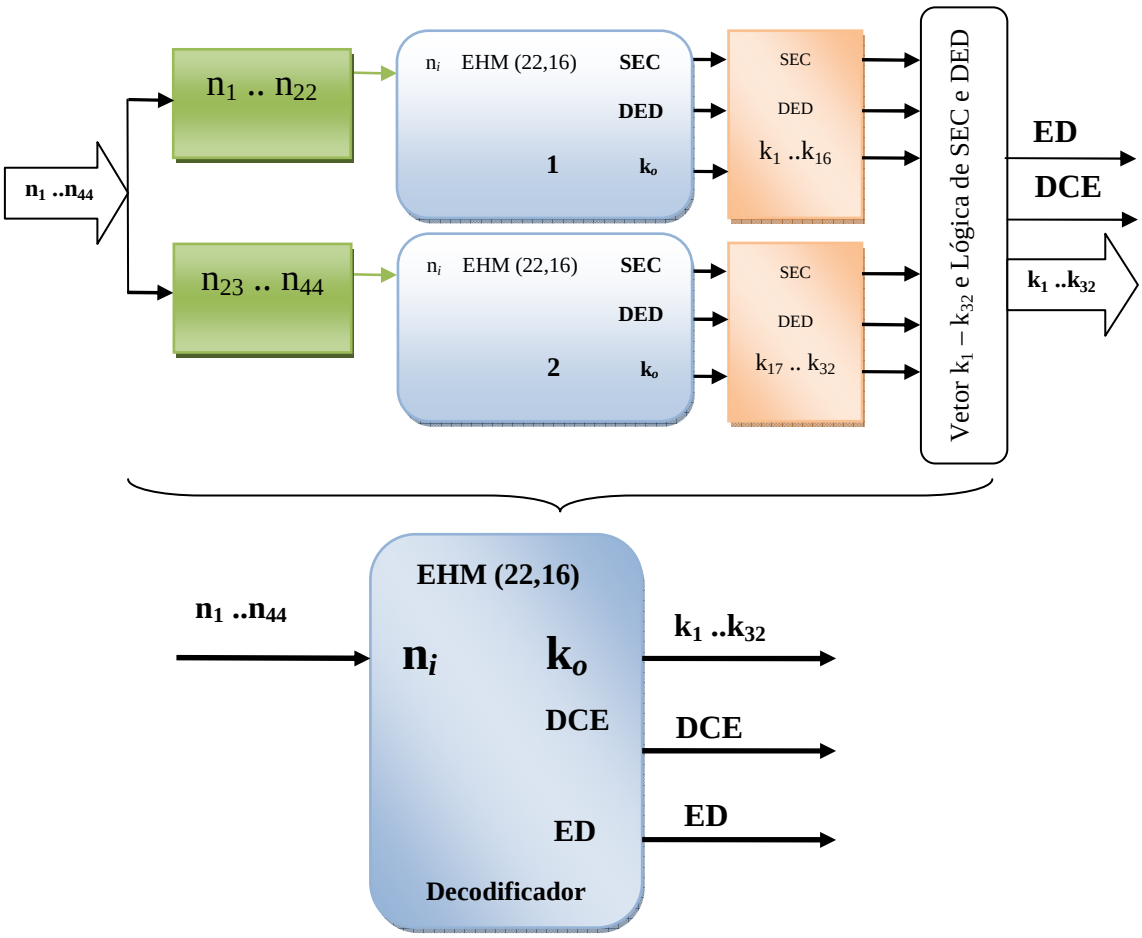


Figura 5.17 - Arquitetura do decodificador do código Combinado *Ex-Hamming* EHM (22,16) e o bloco equivalente.

O decodificador EHM (22,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{44}$) é dividido em 2 seqüências de *bits*, cada uma com 22 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_{22}$ e $n_{23} \ldots n_{44}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador de Ex-*hamming* e segunda seqüência para o segundo decodificador;
- O resultado de cada decodificador (junto com o sinal SEC e DED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro ou detecção de erro.

5.3.3. *Matrix Code*

Foram desenvolvidos dois CODECs (codificadores e decodificadores) do *Matrix*: *Matrix* (12,4) e *Matrix* (36,16). A tabela 5.5 mostra algumas características destes códigos tais como: *bits* de entrada (*k*), *bits* de saída(*n*) e *overhead* para cada código básico e também para a forma combinada (com e sem embaralhador).

Tabela 5.5 - Informações dos códigos *Matrix*.

	Básico			Combinado Sem Embaralhador			Combinado Com Embaralhador			
Código *Matrix*	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	*k*	*n*	*Overhead* (%)	Grau de Embaralhamento
MC(12,4)	4	12	200	32	96	200	32	96	200	8
MC (36,16)	16	36	125	32	72	125	32	72	125	2

Na tabela 5.5 podemos observar que a aplicação da técnica de embaralhamento não altera o tamanho final do código combinado. A Figura 5.18 mostra a representação em bloco do CODEC do código *Matrix* (*n*,*k*), os pinos de entrada e saída e a descrição de cada pino.

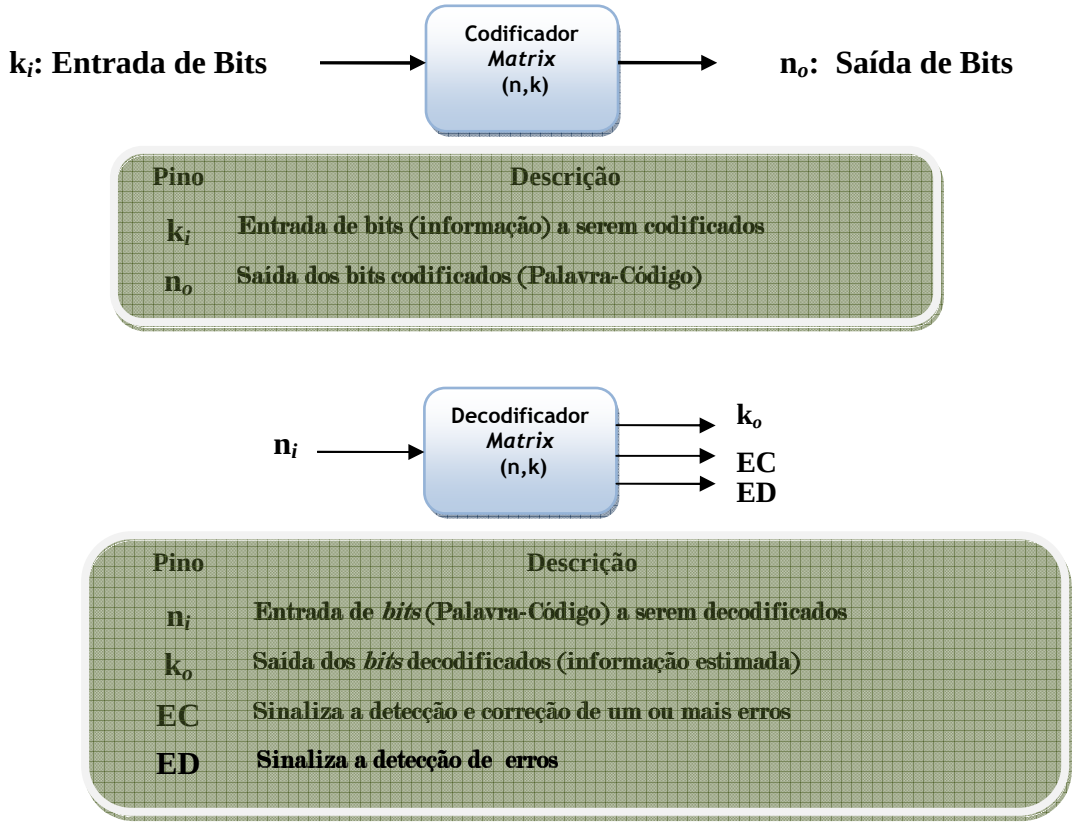


Figura 5.18 - Representação em bloco do código Matriz.

5.3.3.1. *Matrix Code* (12,4)

A Figura 5.19 mostra a arquitetura do codificador Combinado *Matrix* MC (12,4) e o bloco equivalente.

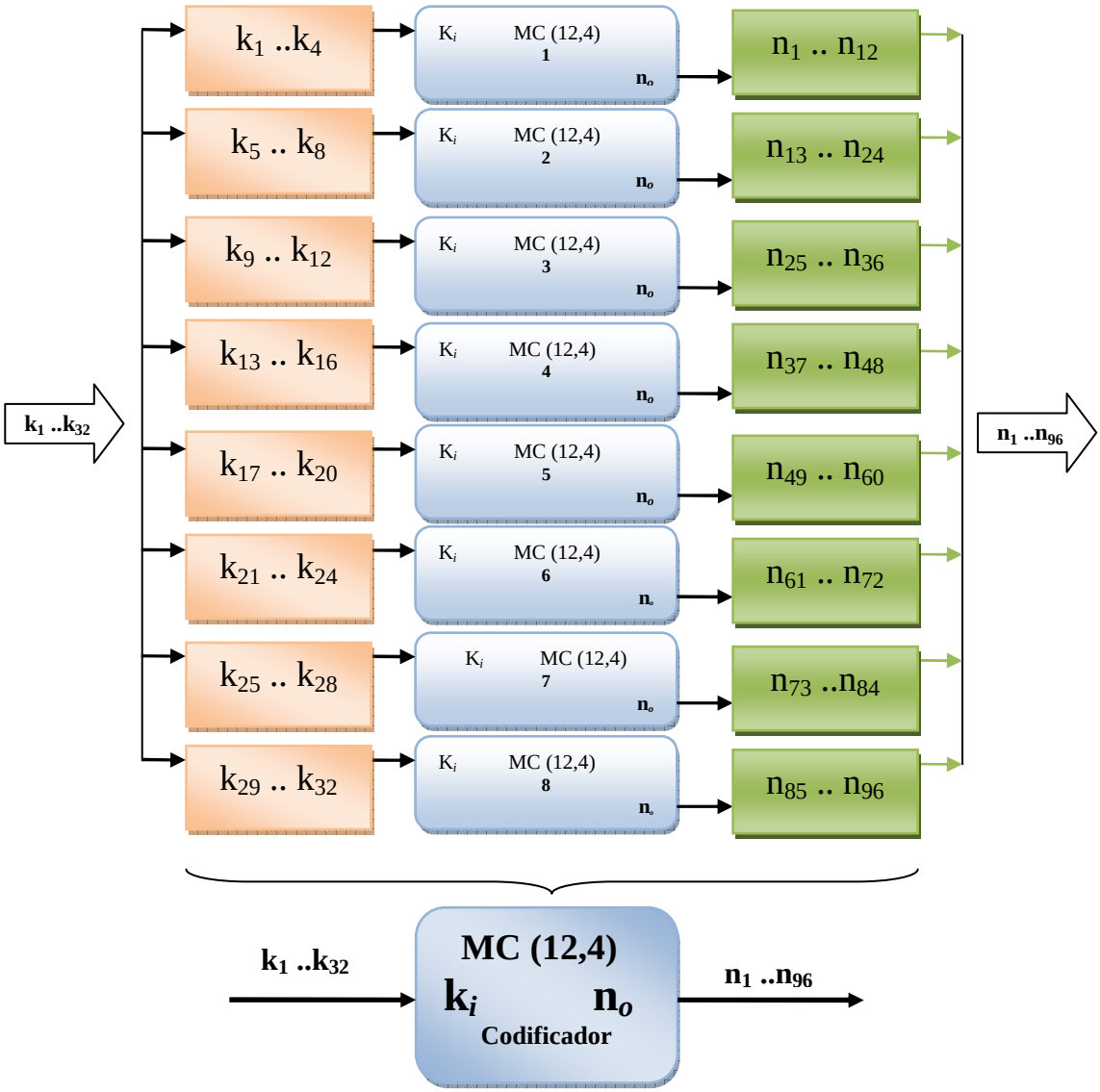


Figura 5.19 - Arquitetura do codificador do código Combinado *Matrix* MC (12,4) e o bloco equivalente.

O codificador do código combinado Matriz (12,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 8 seqüências de *bits*, cada uma com 4 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_4$, $k_5 \ldots k_8$, …e $k_{29} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador *Matrix*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{96}$).

A Figura 5.20 mostra a arquitetura do decodificador do código Combinado *Matrix* MC (12,4) e o bloco equivalente.

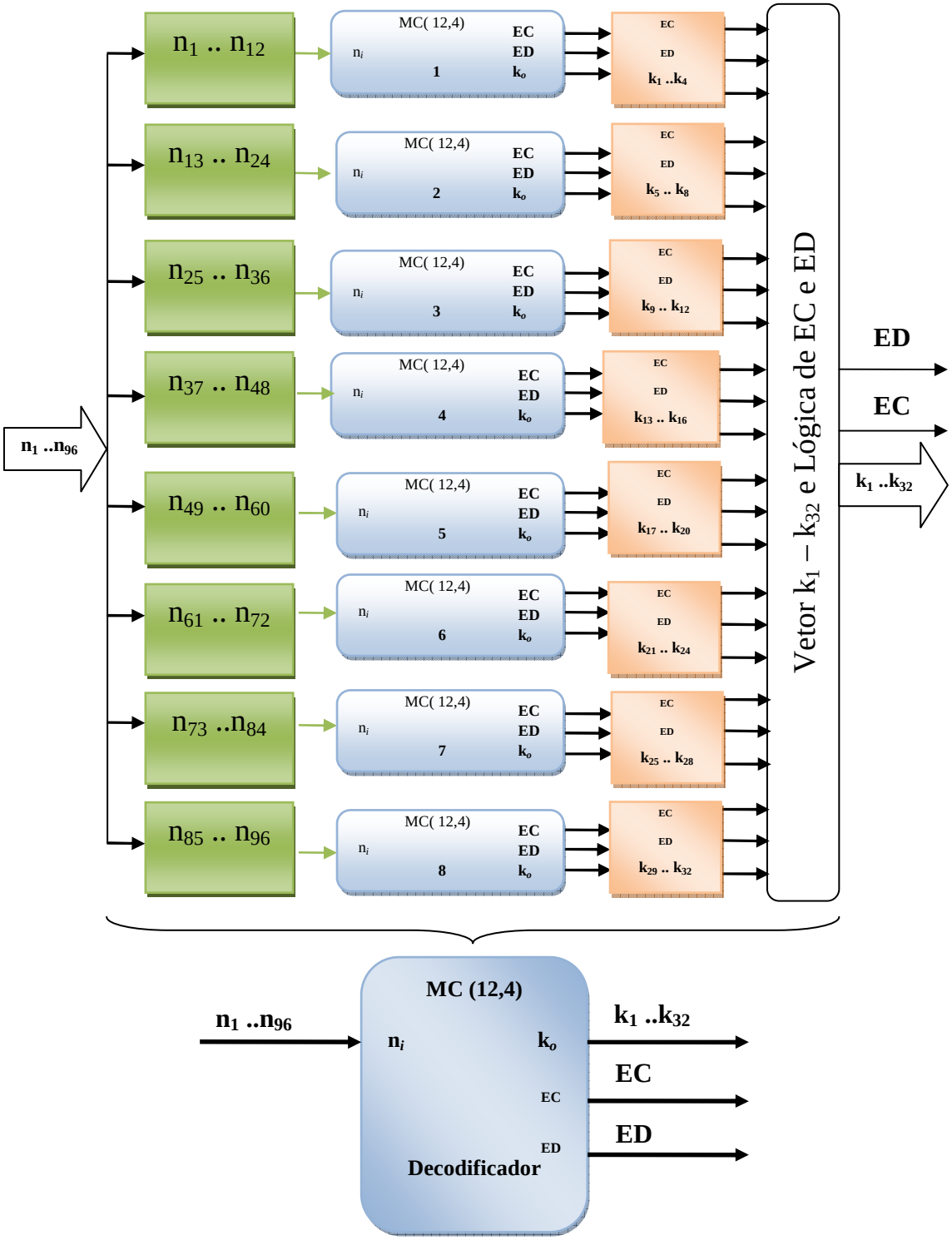


Figura 5.20 - Arquitetura do decodificador do código Combinado *Matrix* MC (12,4) e o bloco equivalente.

O decodificador MC (12,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{96}$) é dividido em 8 seqüências de *bits*, cada uma com 12 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_{12}$, $n_{13} \ldots n_{24}$, ... e $n_{85} \ldots n_{96}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador *Matrix*, segunda seqüência para o segundo decodificador e assim por diante;
- O resultado de cada decodificador (junto com o sinal EC e ED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro(EC) ou Detecção de erro(ED).

5.3.3.2. *Matrix Code* (36,16)

A Figura 5.21 mostra a arquitetura do codificador do código Combinado *Matrix* (MC (36,16)) e o bloco equivalente.

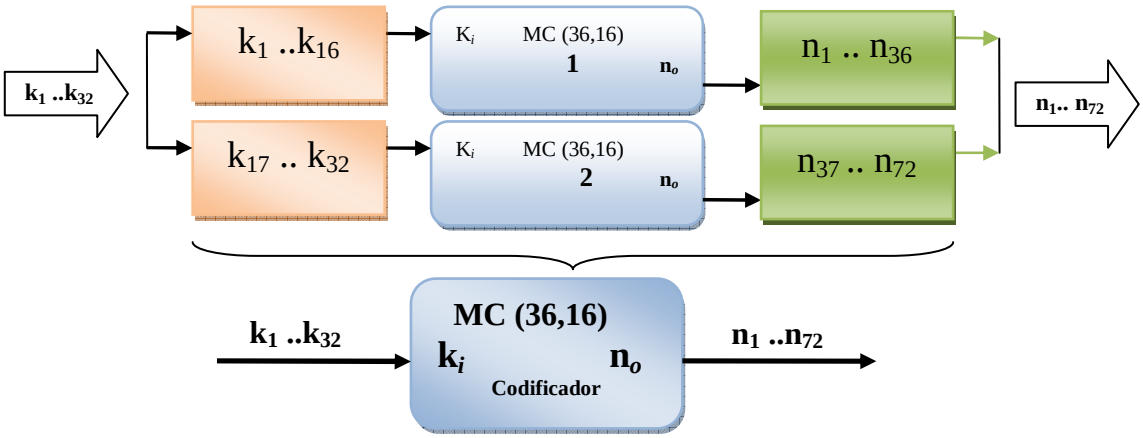


Figura 5.21- Arquitetura do codificador do código Combinado *Matrix* MC (36,16) e o bloco equivalente.

O codificador do código combinado *Matrix* (36,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 2 seqüências de *bits*, cada uma com 16 *bits*;
- Cada seqüência de bits ($k_1 \ldots k_{16}$, $k_{17} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador Matriz e segunda seqüência para o segundo codificador ;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{72}$).

A Figura 5.21 mostra a arquitetura do decodificador do código Combinado *Matrix* MC (36,16) e o bloco equivalente.

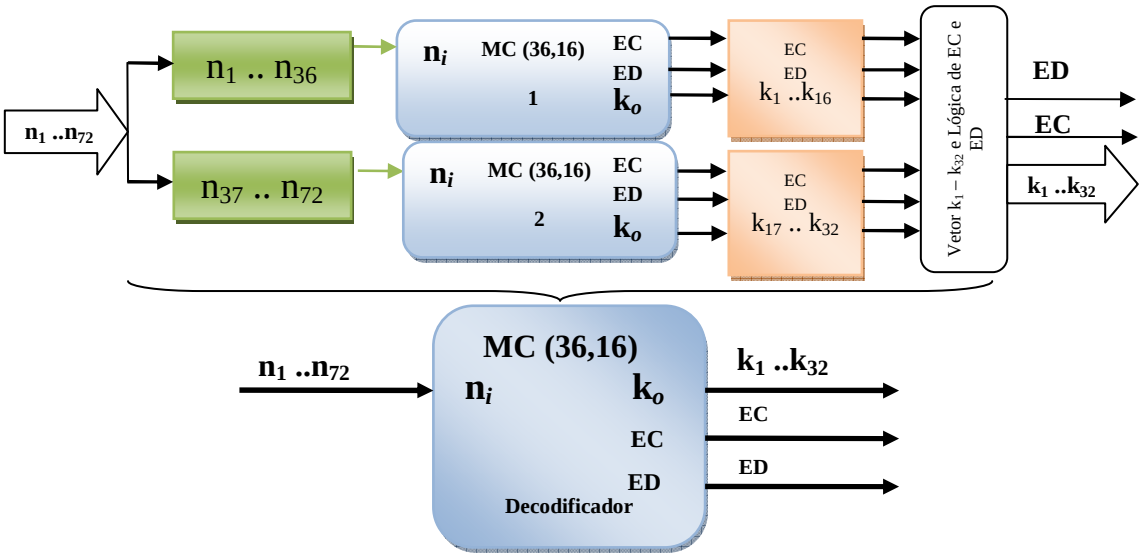


Figura 5.22 - Arquitetura do decodificador do código Combinado *Matrix* MC(36,16) e o bloco equivalente.

O decodificador MC (36,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{72}$) é dividido em 2 seqüências de *bits*, cada uma com 36 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_{36}$ e $n_{37} \ldots n_{72}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador *Matrix* e segunda seqüência para o segundo decodificador;
- O resultado de cada decodificador (junto com o sinal EC e ED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro ou detecção de erro.

5.3.4. *Reed-Muller*

Foram desenvolvidos dois CODECs (codificadores e decodificadores) do *Reed-Muller* : *Reed-Muller* (8,4) e *Reed-Muller* (32,16). A tabela 5.6 mostra algumas características destes códigos tais como: *bits* de entrada (k), *bits* de saída(n) e *overhead* para cada código básico e também para a forma combinada (com e sem embaralhador).

Tabela 5.6 - Informações dos códigos *Reed- Muller*

	Básico			Combinado Sem Embaralhador			Combinado Com Embaralhador			
Código *Reed-Muller*	*k*	*n*	*Overhead (%)*	*k*	*n*	*Overhead (%)*	*k*	*n*	*Overhead (%)*	Grau de Embaralhamento
RM(8,4)	4	8	100	32	64	100	32	64	100	8
RM (32,16)	16	32	100	32	64	100	32	64	100	4

Na tabela 5.6 podemos observar que a aplicação da técnica de embaralhamento não altera o tamanho final do código combinado. A Figura 5.23 mostra a representação em bloco do CODEC do código *Reed-Muller* (n,k), os pinos de entrada e saída e a descrição de cada pino.

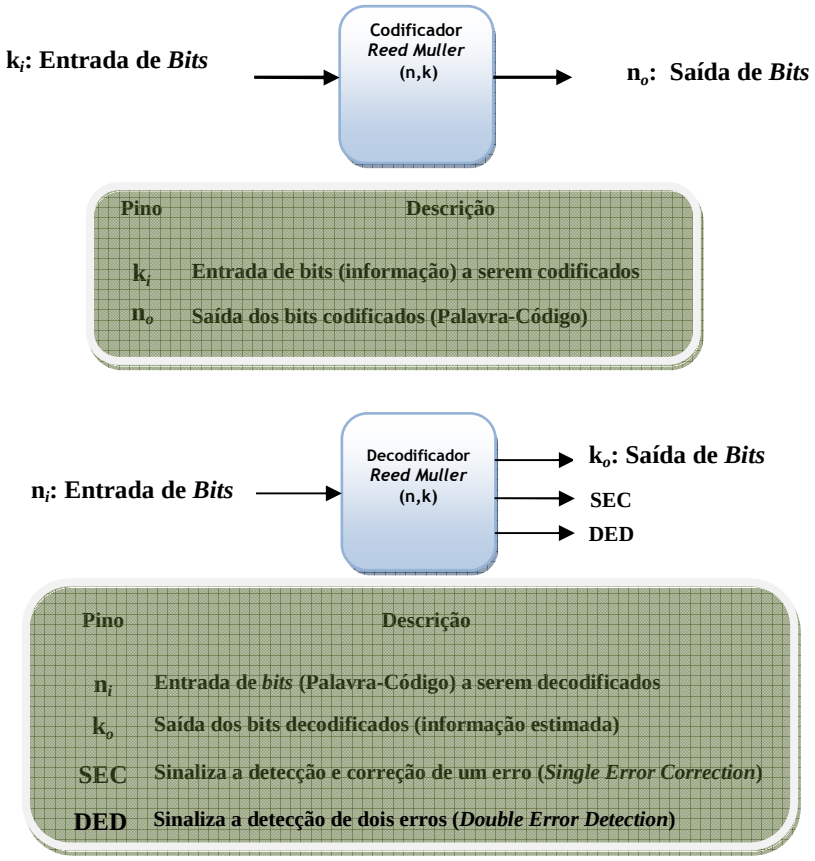


Figura 5.23 - Representação em bloco do código *Reed-Muller*.

5.3.4.1. *Reed-Muller* (8,4)

A Figura 5.24 mostra a arquitetura do codificador do código Combinado *Reed-Muller* RM (8,4) e o bloco equivalente.

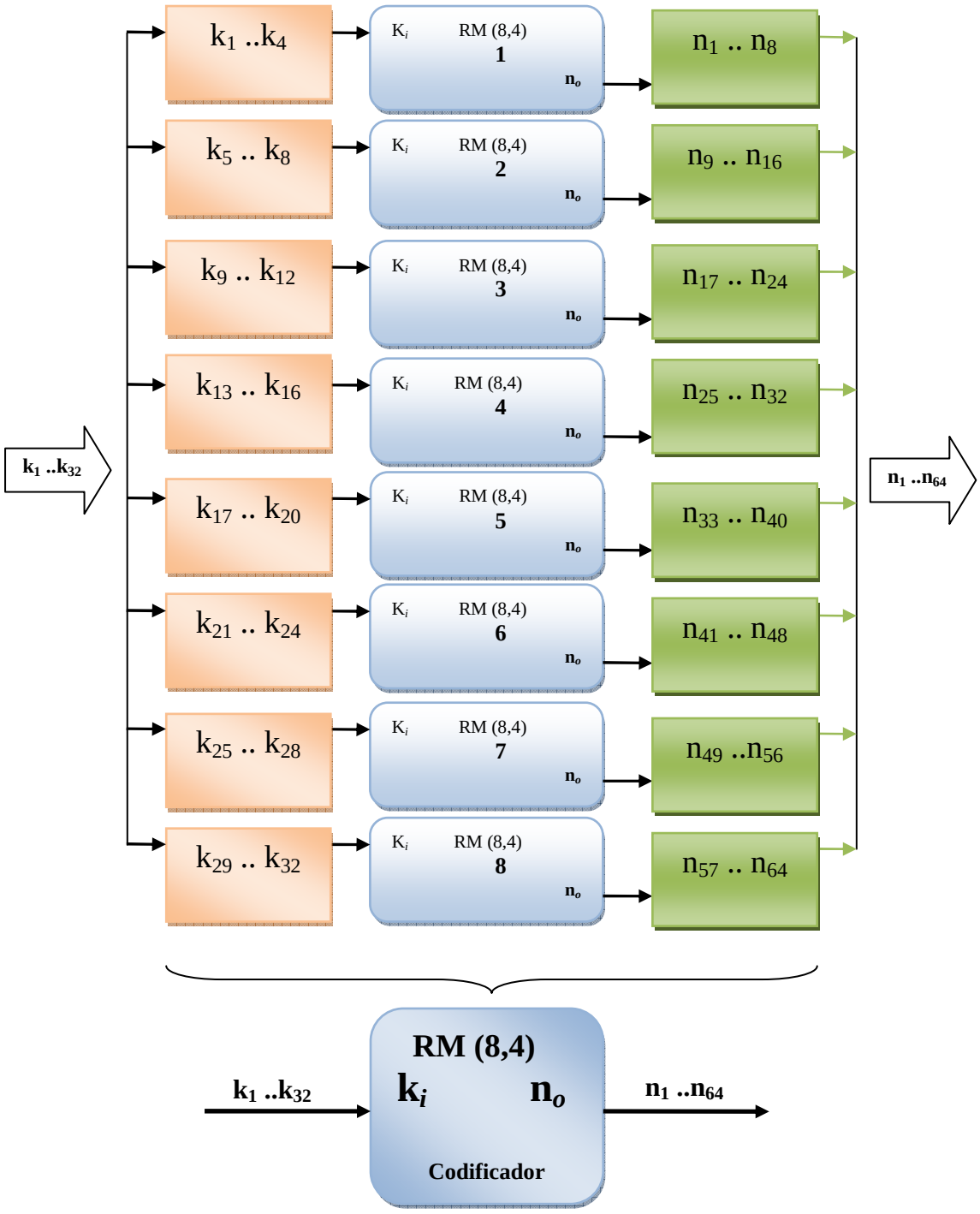


Figura 5.24 - Arquitetura do codificador do código Combinado *Reed-Muller* RM (8,4) e o bloco equivalente.

O codificador do código combinado *Reed-Muller* (8,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 8 seqüências de *bits*, cada uma com 4 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_4$, $k_5 \ldots k_8$, …e $k_{29} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador *Reed-Muller*, segunda seqüência para o segundo codificador e assim por diante;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{64}$).

A Figura 5.25 mostra a arquitetura do decodificador do código Combinado *Reed-Muller* RM (8,4) e o bloco equivalente.

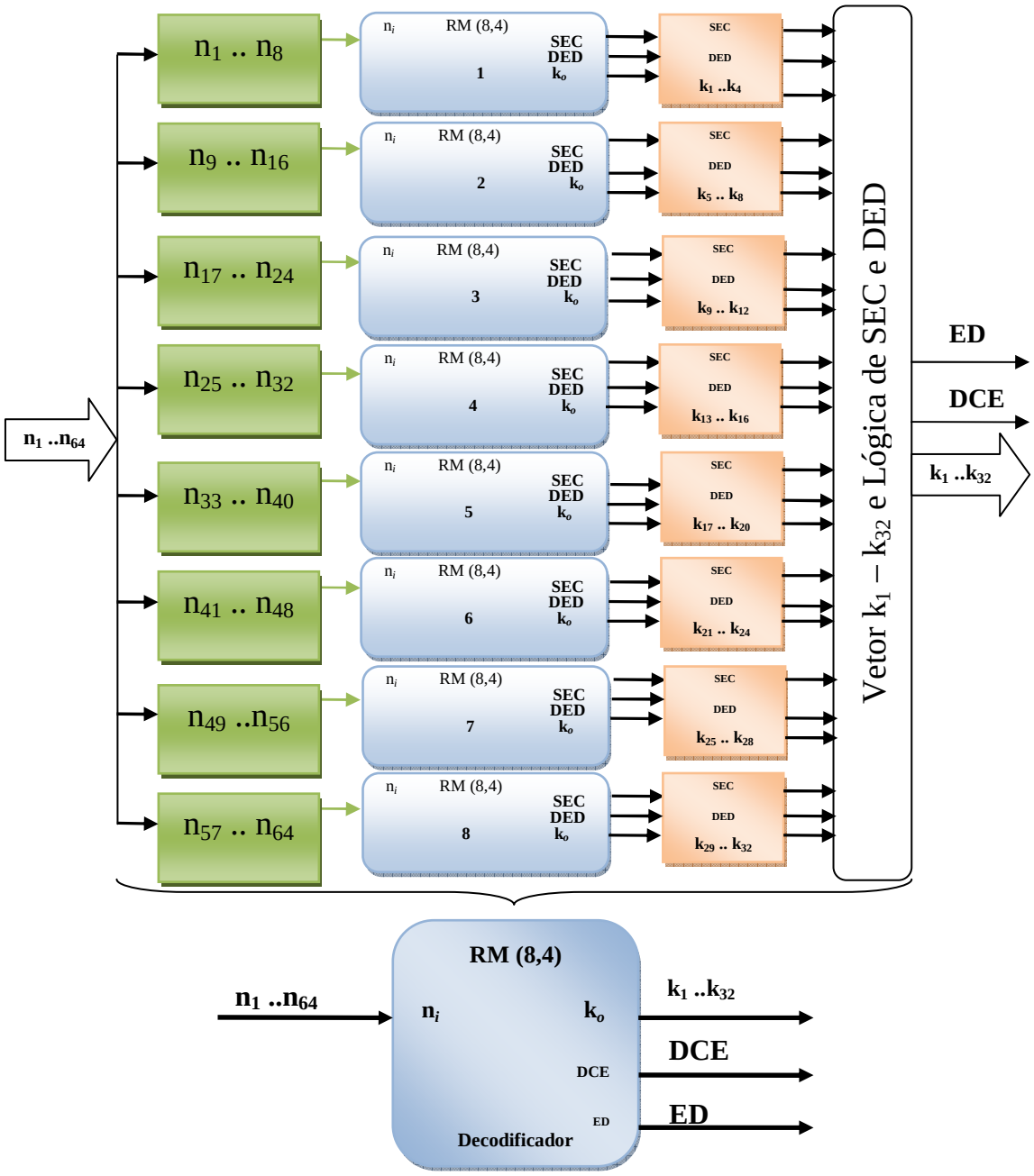


Figura 5.25 - Arquitetura do decodificador do código Combinado *Reed-Muller* RM (8,4) e o bloco equivalente.

O decodificador RM (8,4) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{64}$) é dividido em 8 seqüências de *bits*, cada uma com 8 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_8$, $n_9 \ldots n_{16}$,... e $n_{57} \ldots n_{64}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador *Reed-Muller*, segunda seqüência para o segundo decodificador e assim por diante;
- O resultado de cada decodificador (junto com o sinal DCE e ED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro (DCE) ou Detecção de erro (ED).

5.3.4.2. Reed-Muller (32,16)

Para codificar uma palavra de 16 de *bits* utiliza-se um RM(32,16) que tem a capacidade de corrigir até 3 erros em rajada e detectar 4 erros com um *overhead* de 100%. Utilizando 4 codificadores de RM(8,4) de forma combinada e com a técnica de embaralhamento aplicada, é possível corrigir até 4 erros e detectar até 8 erros com um overhead de 100%. Para obter um melhor rendimento na correção e detecção de erros, o RM (32,16) será substituído por 4 codificadores de RM(8,4) (Figura 5.26). O decodificador é mostrado na Figura 5.27.

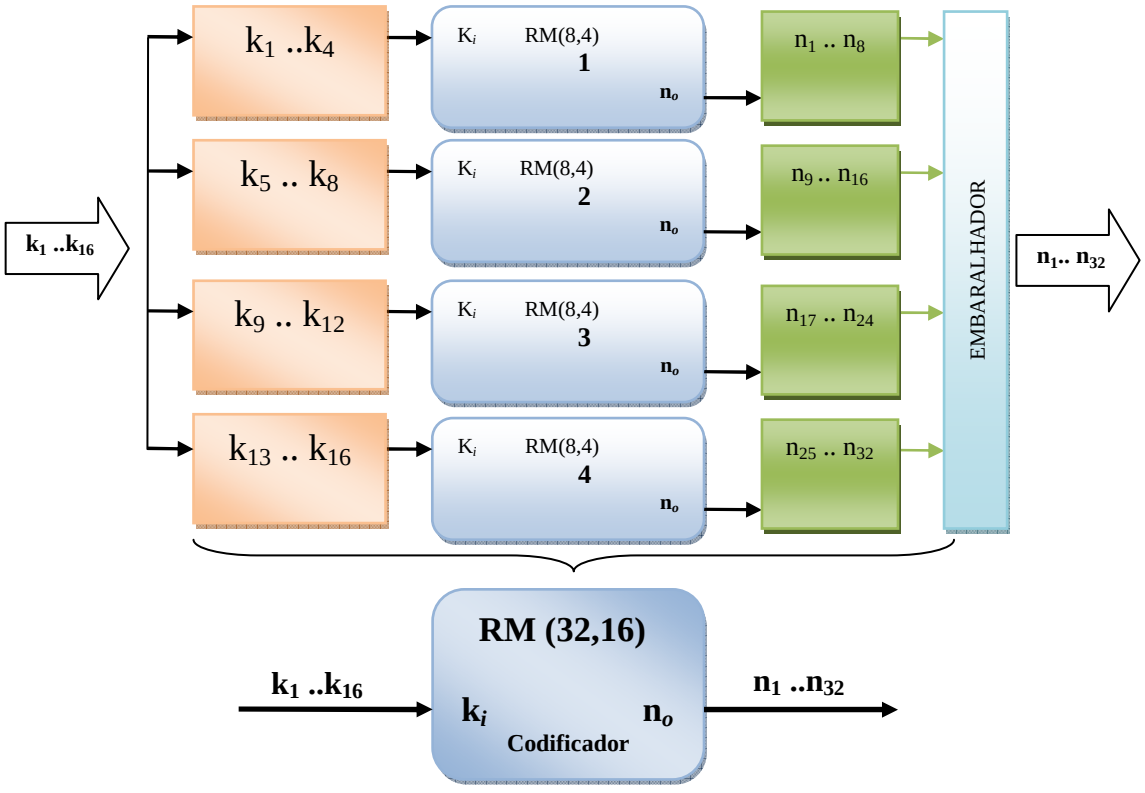


Figura 5.26 – Codificador RM(32,16) utilizando por 4 codificadores de RM(8,4).

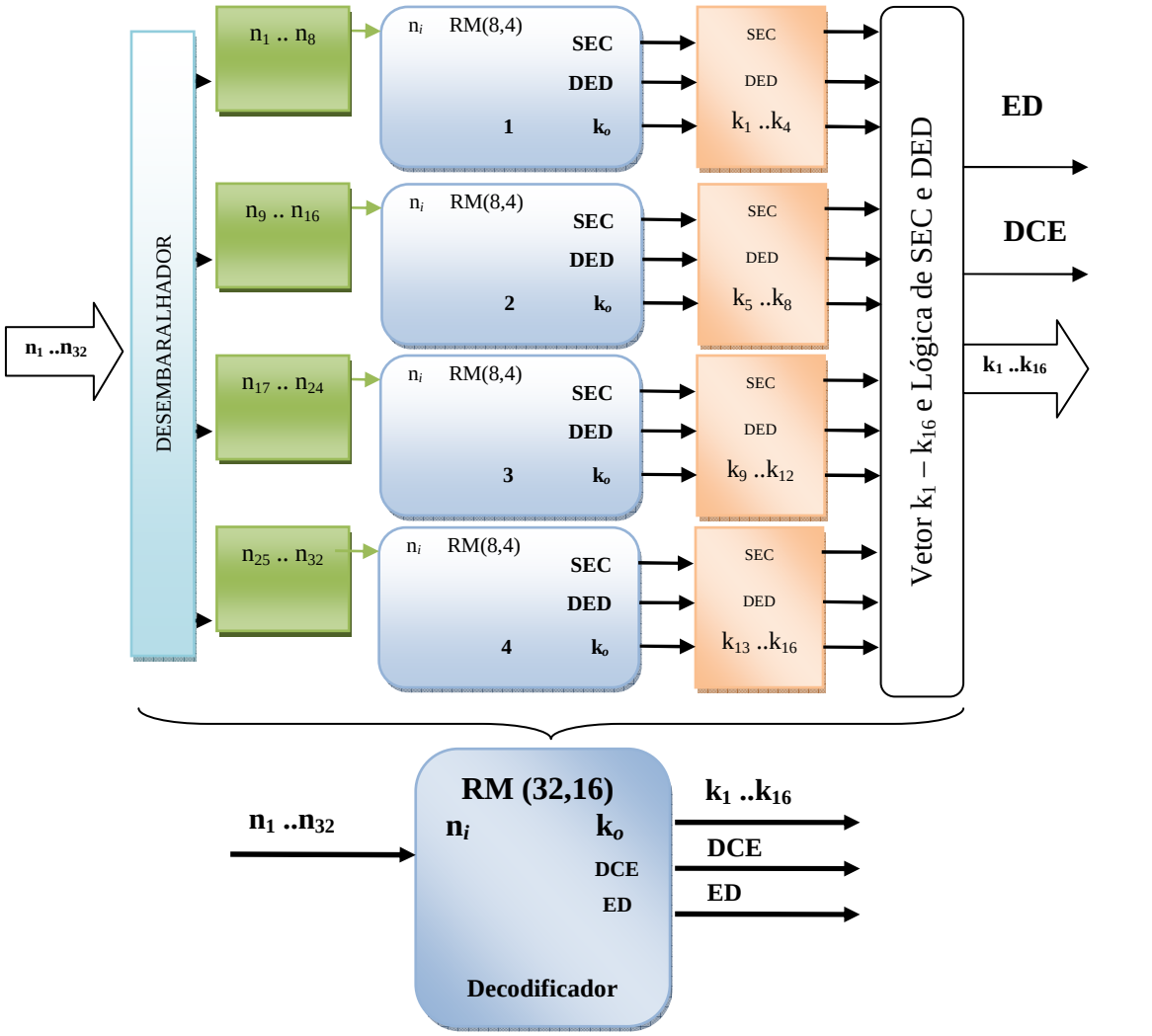


Figura 5.27 – Decodificador RM(32,16) utilizando 4 decodificadores de RM(8,4).

A Figura 5.28 mostra a arquitetura do codificador do código Combinado *Reed-Muller* RM(32,16) e o bloco equivalente. A Figura 5.29 mostra a arquitetura do decodificador do código Combinado *Reed-Muller* RM(32,16) e o bloco equivalente.

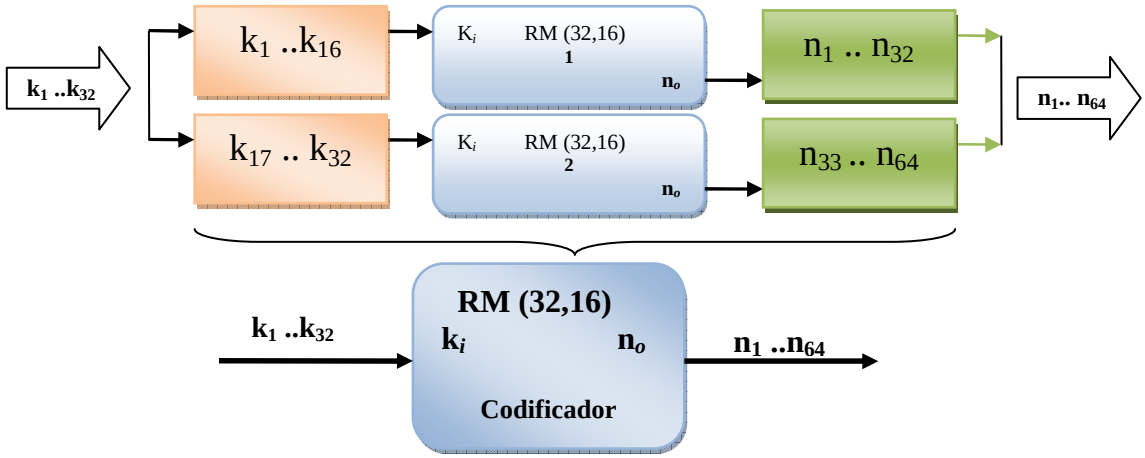


Figura 5.28- Arquitetura do codificador do código Combinado *Reed-Muller* RM (32,16) e o bloco equivalente.

O codificador do código combinado *Reed-Muller* (32,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($k_1 \ldots k_{32}$) é dividido em 2 seqüências de *bits*, cada uma com 16 *bits*;
- Cada seqüência de *bits* ($k_1 \ldots k_{16}$, $k_{17} \ldots k_{32}$) é entregue para os codificadores da seguinte forma: primeira seqüência para o primeiro codificador *Reed-Muller* e segunda seqüência para o segundo codificador ;
- O resultado de cada codificador compõe o vetor de saída ($n_1 \ldots n_{64}$).

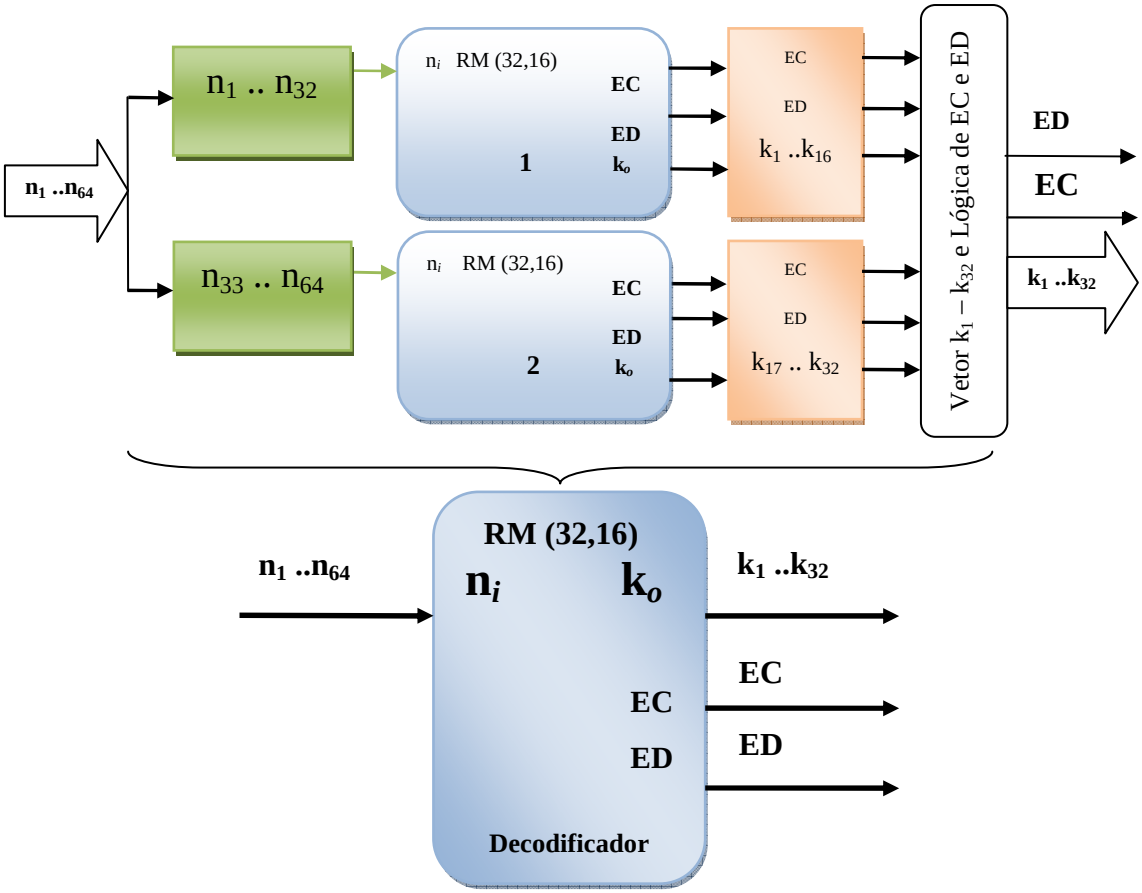


Figura 5.29 - Arquitetura do decodificador do código Combinado *Reed-Muller* RM (32,16) e o bloco equivalente.

O decodificador do RM (32,16) funciona da seguinte maneira:

- O vetor de *bits* de entrada ($n_1 \ldots n_{64}$) é dividido em 2 seqüências de *bits*, cada uma com 32 *bits*;
- Cada seqüência de *bits* ($n_1 \ldots n_{32}$ e $n_{33} \ldots n_{64}$) é entregue para os decodificadores da seguinte forma: primeira seqüência para o primeiro decodificador *Reed-Muller* e segunda seqüência para o segundo decodificador;
- O resultado de cada decodificador (junto com o sinal EC e ED) é entregue ao bloco responsável por compor o vetor de saída ($k_1 \ldots k_{32}$) e também para sinalizar se houve detecção e correção de erro ou detecção de erro.

5.4. Plataforma de Testes para Validação dos Códigos

A Figura 5.30 apresenta a plataforma de testes utilizada para validar os códigos combinados. Toda a estrutura foi desenvolvida em VHDL padrão e foram utilizadas as ferramentas de sintetização e simulação da Altera (QUARTUS II 7.1) e XILINX (XILINX ISE 7.1i).

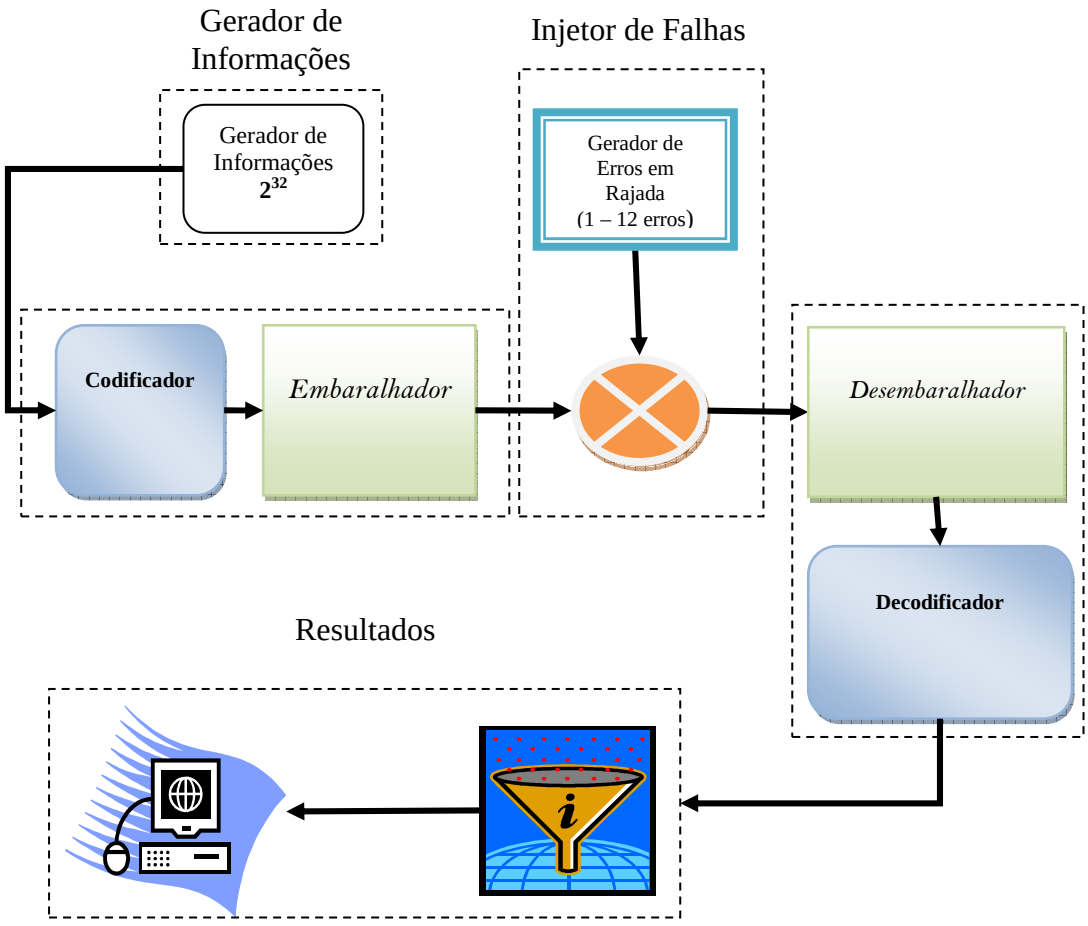


Figura 5.30 - Plataforma de testes utilizada para validação dos códigos combinados.

A estrutura de testes possui basicamente cinco blocos com as seguintes funções:

- **Gerador de Informações**: é responsável por gerar as informações a serem codificadas pelo codificador. Esse bloco possui a capacidade de gerar 2^{32} informações diferentes, ou seja, todas as combinações possíveis de 32 bits.

- **Codificador**: tem a finalidade de codificar a informação recebida do gerador de informações. A tabela 5.7 apresenta os códigos combinados desenvolvidos e utilizados na plataforma de testes com o respectivo tamanho do vetor de entrada (informação) e de saída (Palavra-Código).

Tabela 5.7 – Códigos combinados utilizados na plataforma de teste.

Código Combinado		**Vetor de Entrada Informação (*bits*)**	**Vetor de Saída Palavra-Código (*bits*)**
Hamming	**(7,4)**	**32**	**56**
	(12,8)	**32**	**48**
	(21,16)	**32**	**42**
***Hamming* Estendido (*Ex-Hamming*)**	**(8,4)**	**32**	**64**
	(13,8)	**32**	**52**
	(22,16)	**32**	**44**
Matrix Code	**(12,4)**	**32**	**96**
	(36,16)	**32**	**72**
Reed-Muller	**(8,4)**	**32**	**64**
	(32,16)	**32**	**64**

- **Injetor de Falhas**: este bloco injeta nas Palavras-Código recebidas do codificador todas as combinações possíveis de erros em rajada, variando de 1 até 12 erros. Este bloco consiste em executar uma operação XOR entre a Palavra-Código recebida e a máscara que representa as falhas injetadas.

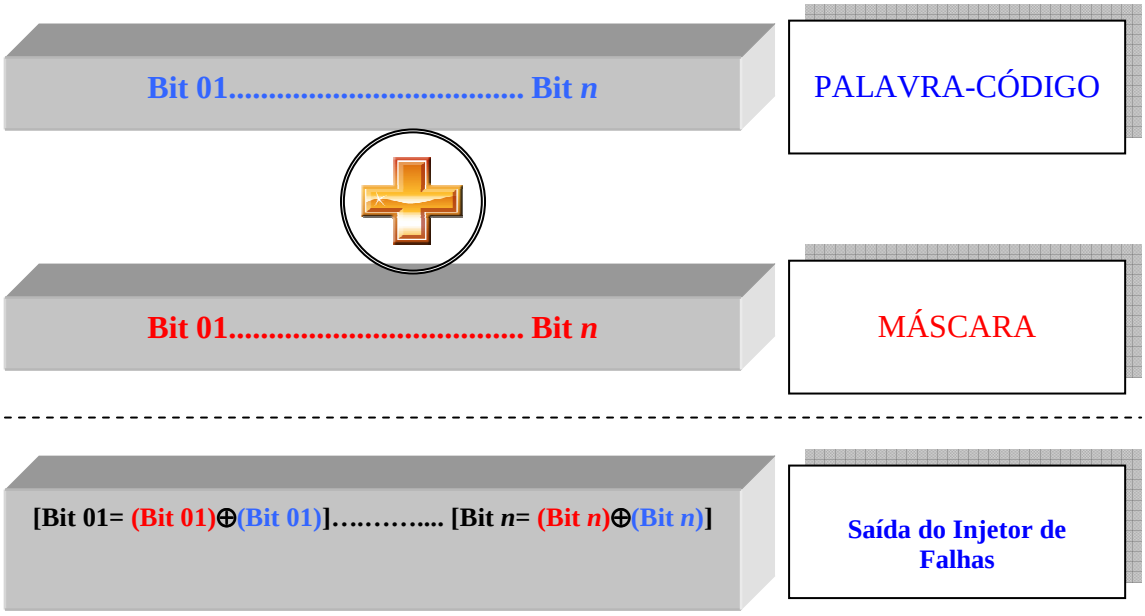


Figura 5.31 – Procedimento de injeção de falhas.

Para o cálculo das combinações é utilizada a equação 5.2. A Figura 5.31 apresenta um exemplo da aplicação da equação 5.2.

$$N°_{Combinações} = N°_{Bits\ da\ Palavra\text{-}Código} - N°_{Erros\ em\ Rajada} + 1 \qquad (5.2)$$

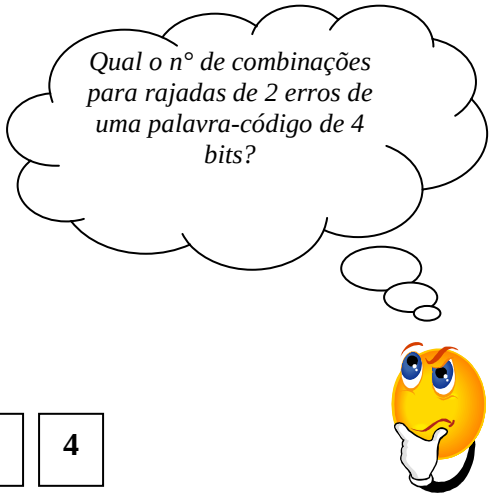
Resposta:

Utilizando a equação 5.2 temos:

$N°_{Combinações} = N°_{Bits\ da\ Palavra\text{-}Código} - N°_{Erros\ em\ Rajada} + 1$

$N°_{Combinações} = 4 - 2 + 1 = 3$

O número de combinações é 3. A ilustração Abaixo apresenta as combinações possíveis.



Palavra-Código	1	2	3	4
1ª Combinação	**1**	**2**	3	4
2ª Combinação	1	**2**	**3**	4
3ª Combinação	1	2	**3**	**4**

Figura 5.32 – Exemplo de cálculo das combinações de erro em rajadas.

A tabela 5.8 apresenta as combinações de erros em rajada para cada código combinado utilizando a equação 5.2

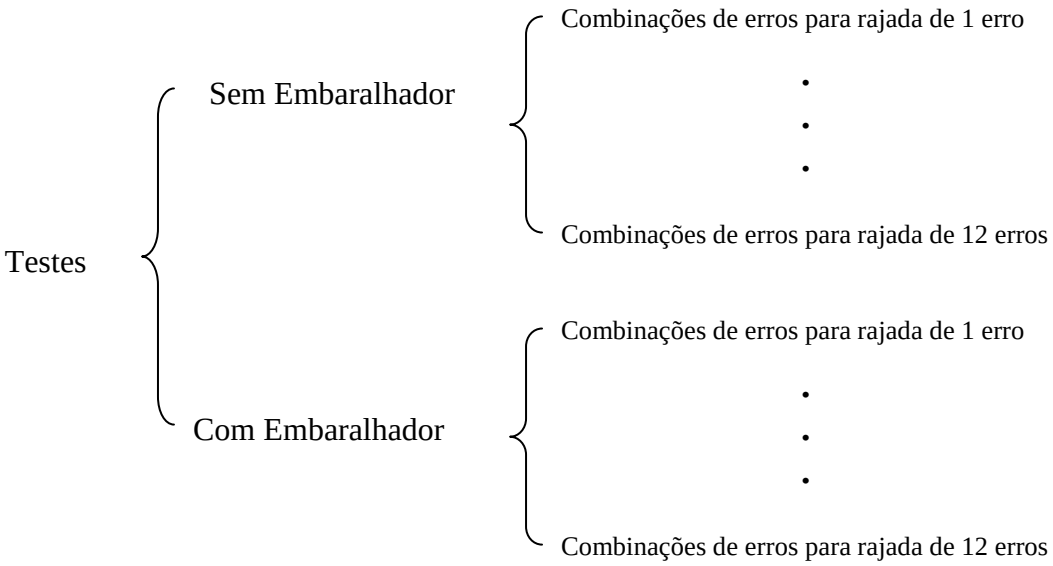
Tabela 5.8 – combinações de erros em rajada para cada código combinado utilizando a equação 5.2.

Código Combinado		Palavra-Código (*bits*)	Erros em Rajadas											
			1	2	3	4	5	6	7	8	9	10	11	12
			N° de combinações de erros em rajadas											
Hamming	**(7,4)**	**56**	56	55	54	53	52	51	50	49	48	47	46	45
	(12,8)	**48**	48	47	46	45	44	43	42	41	40	39	38	37
	(21,16)	**42**	42	41	40	39	38	37	36	35	34	33	32	31
Hamming **Estendido** (***Ex-Hamming***)	**(8,4)**	**64**	64	63	62	61	60	59	58	57	56	55	54	53
	(13,8)	**52**	52	51	50	49	48	47	46	45	44	43	42	41
	(22,16)	**44**	44	43	42	41	40	39	38	37	36	35	34	33
Matrix Code	**(12,4)**	**96**	96	95	94	93	92	91	90	89	88	87	86	85
	(36,16)	**72**	72	71	70	69	68	67	66	65	64	63	62	61
Reed-Muller	**(8,4)**	**64**	64	63	62	61	60	59	58	57	56	55	54	53
	(32,16)	**64**	64	63	62	61	60	59	58	57	56	55	54	53

- **Decodificador**: responsável pela decodificação da palavra-código e pela geração dos sinais de detecção e correção de erro.

- **Resultados**: contabiliza as informações recebidas do decodificador e classifica as informações como corrigidas, detectadas e não detectadas.

Para cada código combinado foram executados os seguintes testes:



Desta forma, foram executados 13.128 testes de injeção de falha.

Parte III - Resultados e Conclusões

6. Resultados

6.1. *Hamming*

6.1.1. *Hamming* (7,4)

Nas Tabelas 6.1 e 6.2 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* (7,4) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.1 - Resultados dos testes realizados para HM (7,4) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	12,73	87,27
3	85,19	14,81	0,00	100,00
4	69,81	30,19	0,00	100,00
5	100,00	0,00	0,00	100,00
6	100,00	0,00	0,00	100,00
7	70,00	30,00	0,00	100,00
8	100,00	0,00	0,00	100,00
9	100,00	0,00	0,00	100,00
10	85,11	14,89	0,00	100,00
11	84,78	15,22	0,00	100,00
12	100,00	0,00	0,00	100,00

Observamos na Tabela 6.1 que na rajada de um erro o código combinado de *hamming* (7,4) conseguiu corrigir todas as combinações de erros (100% de correção). Para rajada de dois erros o código conseguiu corrigir de maneira correta somente 12,73 % das combinações de erros e as demais foram corrigidas erradas e também para as demais rajadas, as correções

foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foi possível realizar as correções dos *bits* errados.

Tabela 6.2 - Resultados dos testes realizados para HM (7,4) com embaralhador.

Rajada **(*Bits*)**	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	100,00	0,00
3	100,00	0,00	100,00	0,00
4	100,00	0,00	100,00	0,00
5	100,00	0,00	100,00	0,00
6	100,00	0,00	100,00	0,00
7	100,00	0,00	100,00	0,00
8	100,00	0,00	100,00	0,00
9	100,00	0,00	0,00	100,00
10	100,00	0,00	0,00	100,00
11	100,00	0,00	0,00	100,00
12	100,00	0,00	0,00	100,00

Observamos na Tabela 6.2 que com o uso da técnica de embaralhamento o código combinado de *hamming* (7,4) conseguiu corrigir todas as combinações de erros de até oito *bits* (100% de correção). Para as demais rajadas superiores (de nove até doze *bits*), as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código.

A Figura 6.1 apresenta os resultados do código combinado de *hamming* (7,4) obtidos das Tabelas 6.1 e 6.2. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou consideravelmente. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até oito *bits* errados.

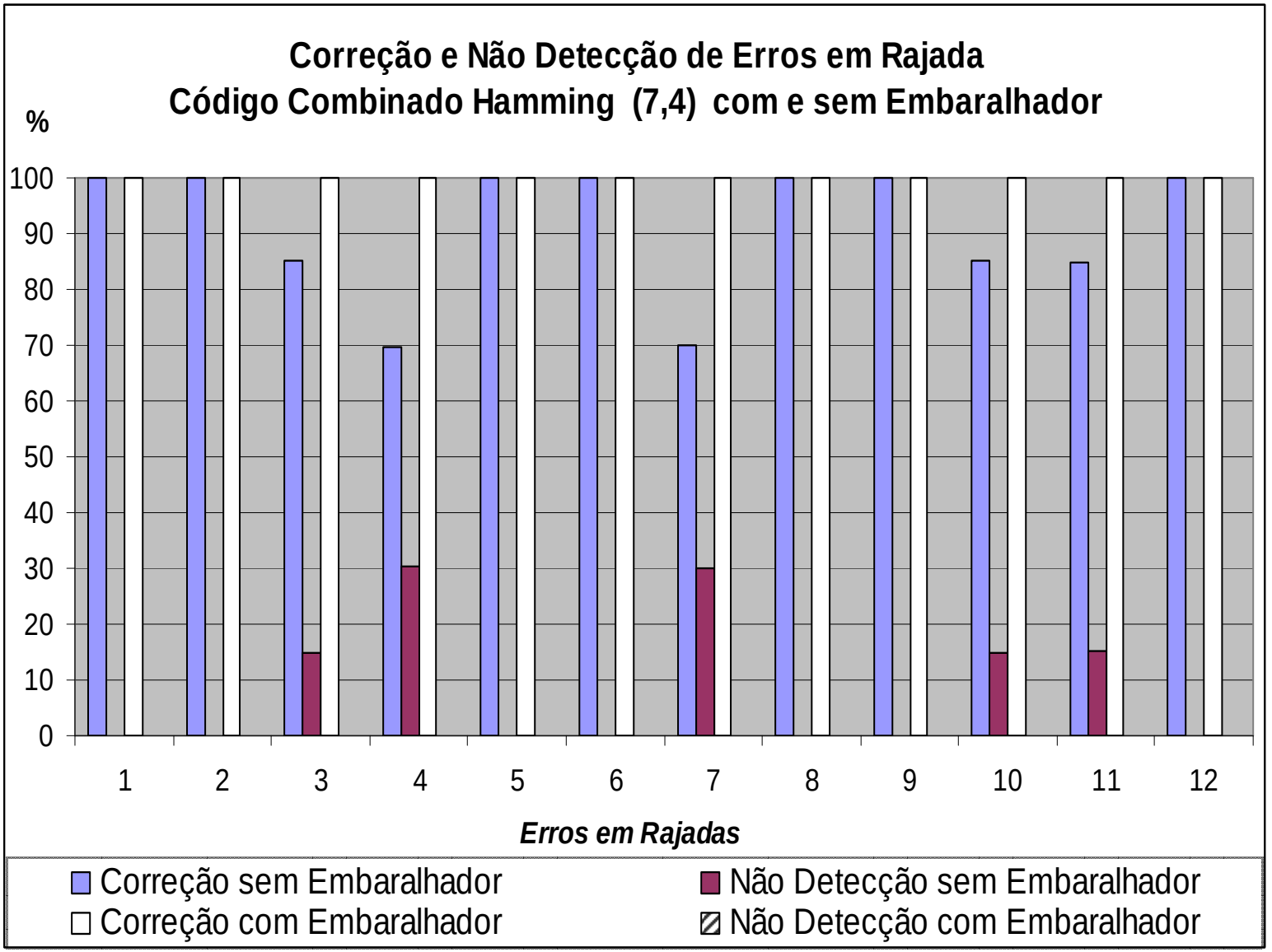


Figura 6.1 - Resultados de correção e não detecção com e sem o uso do embaralhador (HM (7,4)).

6.1.2. *Hamming* (12,8)

Nas Tabelas 6.3 e 6.4 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* (12,8) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.3 - Resultados dos testes realizados para HM (12,8) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	6,38	93,62
3	91,30	8,70	0,00	100,00
4	64,44	35,56	0,00	100,00
5	100,00	0,00	0,00	100,00
6	100,00	0,00	0,00	100,00
7	90,48	9,52	0,00	100,00
8	80,49	19,51	0,00	100,00
9	100,00	0,00	0,00	100,00
10	100,00	0,00	0,00	100,00
11	89,47	10,53	0,00	100,00
12	100,00	0,00	0,00	100,00

.

Observamos na Tabela 6.3 que na rajada de um erro o código combinado de *hamming* (12,8) conseguiu corrigir todas as combinações de erros (100% de correção). Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foi possível a correção do erro.

Tabela 6.4 - Resultados dos testes realizados para HM (12,8) com embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	100,00	0,00
3	100,00	0,00	100,00	0,00
4	100,00	0,00	100,00	0,00
5	100,00	0,00	0,00	100,00
6	100,00	0,00	0,00	100,00
7	100,00	0,00	0,00	100,00
8	100,00	0,00	0,00	100,00
9	100,00	0,00	0,00	100,00
10	100,00	0,00	0,00	100,00
11	100,00	0,00	0,00	100,00
12	97,30	2,30	0,00	100,00

Observamos na Tabela 6.4 que com o uso da técnica de embaralhamento o código combinado de *hamming* (12,8) conseguiu corrigir todas as combinações de erros de até quatro *bits* (100% de correção correta). Para as demais rajadas superiores (de cinco até onze *bits*), as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Na rajada de 12 *bits*, houve uma não detecção de 2,30% porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foi possível a correção do erro.

A Figura 6.2 apresenta os resultados do código combinado de *hamming* (12,8) obtidos das Tabelas 6.3 e 6.4. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou consideravelmente. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até quatro *bits* errados.

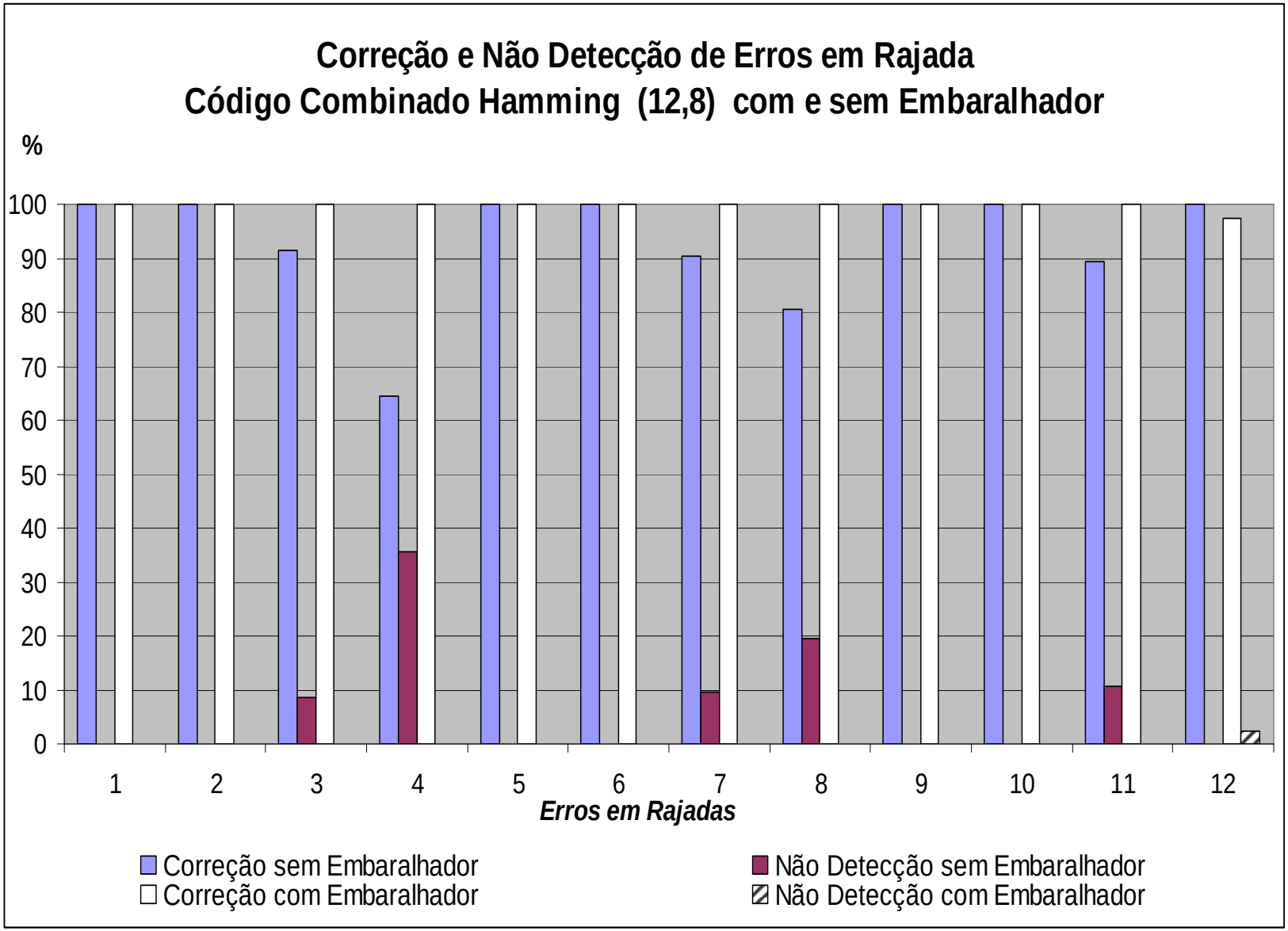


Figura 6.2 - Resultados da correção e não detecção com e sem o uso do embaralhador (HM (12,8)).

6.1.3. *Hamming* (21,16)

Nas Tabelas 6.5 e 6.6 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* (21,16) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.5- Resultados dos testes realizados para HM (21,16) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	2,44	97,56
3	95,00	5,00	0,00	100,00
4	53,85	46,15	0,00	100,00
5	100,00	0,00	0,00	100,00
6	100,00	0,00	0,00	100,00
7	91,67	8,33	0,00	100,00
8	60,00	40,00	0,00	100,00
9	100,00	0,00	0,00	100,00
10	100,00	0,00	0,00	100,00
11	87,25	12,75	0,00	100,00
12	67,74	32,26	0,00	100,00

Observamos na Tabela 6.5 que na rajada de um erro o código combinado de *hamming* (21,16) conseguiu corrigir todas as combinações de erros (100% de correção certa). Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foi possível a correção do erro.

Tabela 6.6 - Resultados dos testes realizados para HM (21,16) com embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Não Detecção (%)**	**Correção**	
			Certa (%)	**Errada (%)**
1	100,00	0,00	100,00	0,00
2	100,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00
4	100,00	0,00	0,00	100,00
5	100,00	0,00	0,00	100,00
6	97,30	2,70	0,00	100,00
7	100,00	0,00	0,00	100,00
8	74,29	25,71	0,00	100,00
9	100,00	0,00	0,00	100,00
10	100,00	0,00	0,00	100,00
11	100,00	0,00	0,00	100,00
12	100,00	0,00	0,00	100,00

Observamos na Tabela 6.6 que com o uso da técnica de embaralhamento o código combinado de *hamming* (21,16) conseguiu corrigir todas as combinações de erros de até dois *bits* (100% de correção correta). Para as demais rajadas superiores, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Na rajada de seis e oito *bits*, algumas combinações de erros não foram detectadas porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foi possível a correção do erro.

A Figura 6.3 apresenta os resultados do código combinado de *hamming* (21,16) obtidos das Tabelas 6.5 e 6.6. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até dois *bits* errados.

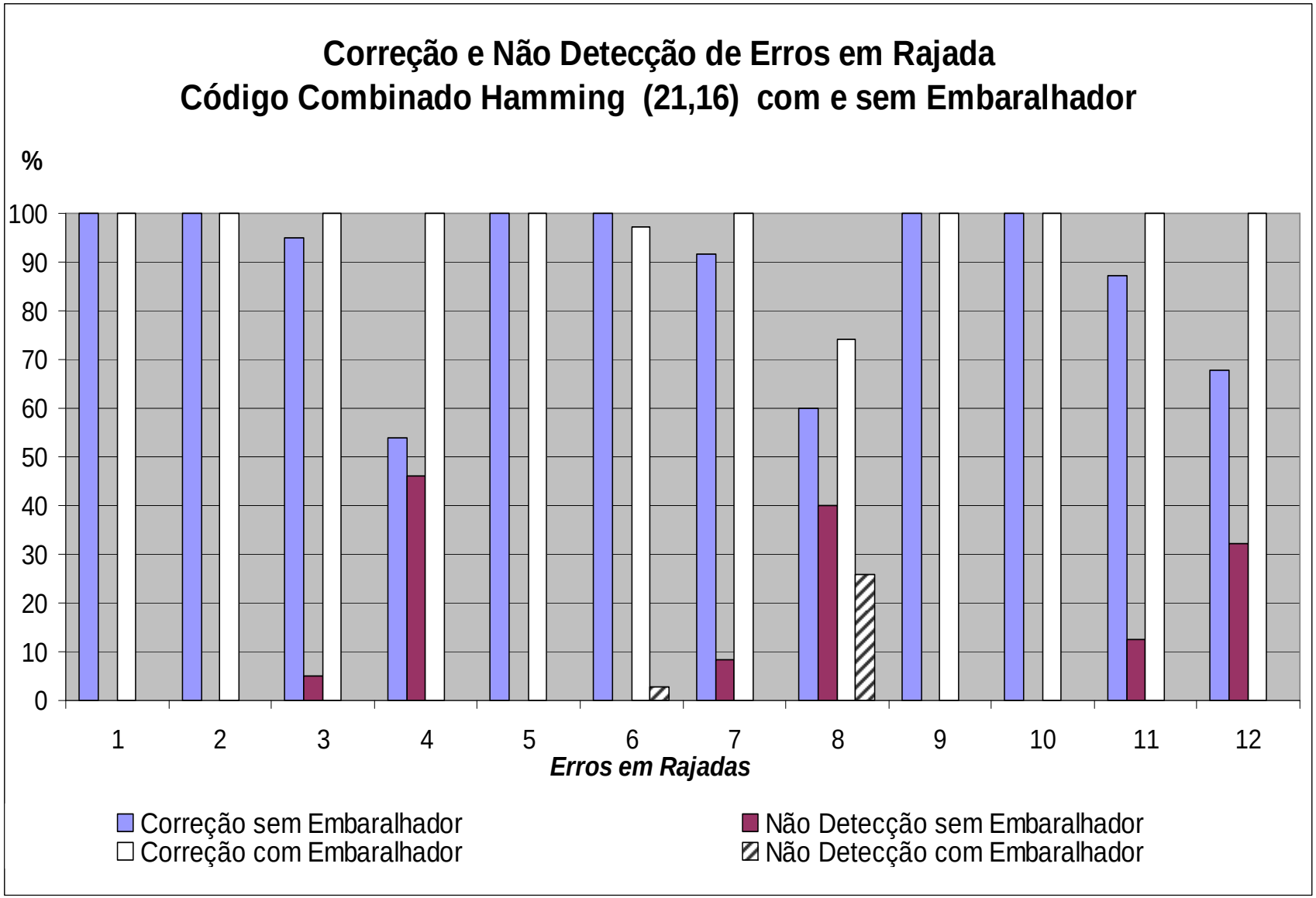


Figura 6.3 - Resultados de correção e não detecção com e sem o uso do interleaver (HM (21,16)).

6.2. *Hamming* + Paridade

6.2.1. *Hamming* + Paridade (8,4)

Nas Tabelas 6.7 e 6.8 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* + Paridade (8,4) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.7 - Resultados dos testes realizados para EHM (8,4) sem embaralhador.

Rajada (*Bits*)	**Detecção (%)**	**Correção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	0,00	100,00	0,00	100,00	0,00
2	88,89	11,11	0,00	100,00	0,00
3	22,58	77,42	0,00	0,00	100,00
4	50,82	22,95	26,23	0,00	100,00
5	46,67	53,33	0,00	0,00	100,00
6	64,41	35,59	0,00	0,00	100,00
7	72,41	27,59	0,00	0,00	100,00
8	36,84	49,12	14,04	0,00	100,00
9	75,00	25,00	0,00	0,00	100,00
10	50,91	49,09	0,00	0,00	100,00
11	74,07	25,93	0,00	0,00	100,00
12	50,94	49,06	0,00	0,00	100,00

Observamos na Tabela 6.7 que na rajada de um erro o código combinado de *hamming* + Paridade (8,4) conseguiu corrigir todas as combinações de erros (100% de correção). Para dois erros conseguiu corrigir 11,11 % das combinações de erros e detectar as combinações restantes. Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não houve a possibilidade de serem corrigidas. Nos casos onde houve a detecção de erro, esses resultados são válidos.

Tabela 6.8 - Resultados dos testes realizados para EHM (8,4) com embaralhador.

Rajada (*Bits*)	Detecção (%)	Correção (%)	Não Detecção (%)	Correção	
				Certa (%)	Errada (%)
1	0,00	100,00	0,00	100,00	0,00
2	0,00	100,00	0,00	100,00	0,00
3	0,00	100,00	0,00	100,00	0,00
4	0,00	100,00	0,00	100,00	0,00
5	0,00	100,00	0,00	100,00	0,00
6	0,00	100,00	0,00	100,00	0,00
7	0,00	100,00	0,00	100,00	0,00
8	0,00	100,00	0,00	100,00	0,00
9	100,00	0,00	0,00	0,00	0,00
10	100,00	0,00	0,00	0,00	0,00
11	100,00	0,00	0,00	0,00	0,00
12	100,00	0,00	0,00	0,00	0,00

Observamos na Tabela 6.8 que com o uso da técnica de embaralhamento o código combinado de *hamming* + Paridade (8,4) conseguiu corrigir todas as combinações de erros de até oito *bits* (100% de correção certa). Para as demais rajadas de erros superiores (de nove até doze *bits*) o código conseguiu detectar os erros.

A Figura 6.4 apresenta os resultados do código combinado de *hamming* + Paridade (8,4) obtidos das Tabelas 6.7 e 6.8. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até oito *bits* errados e detectar todas as combinações entre nove e doze *bits*.

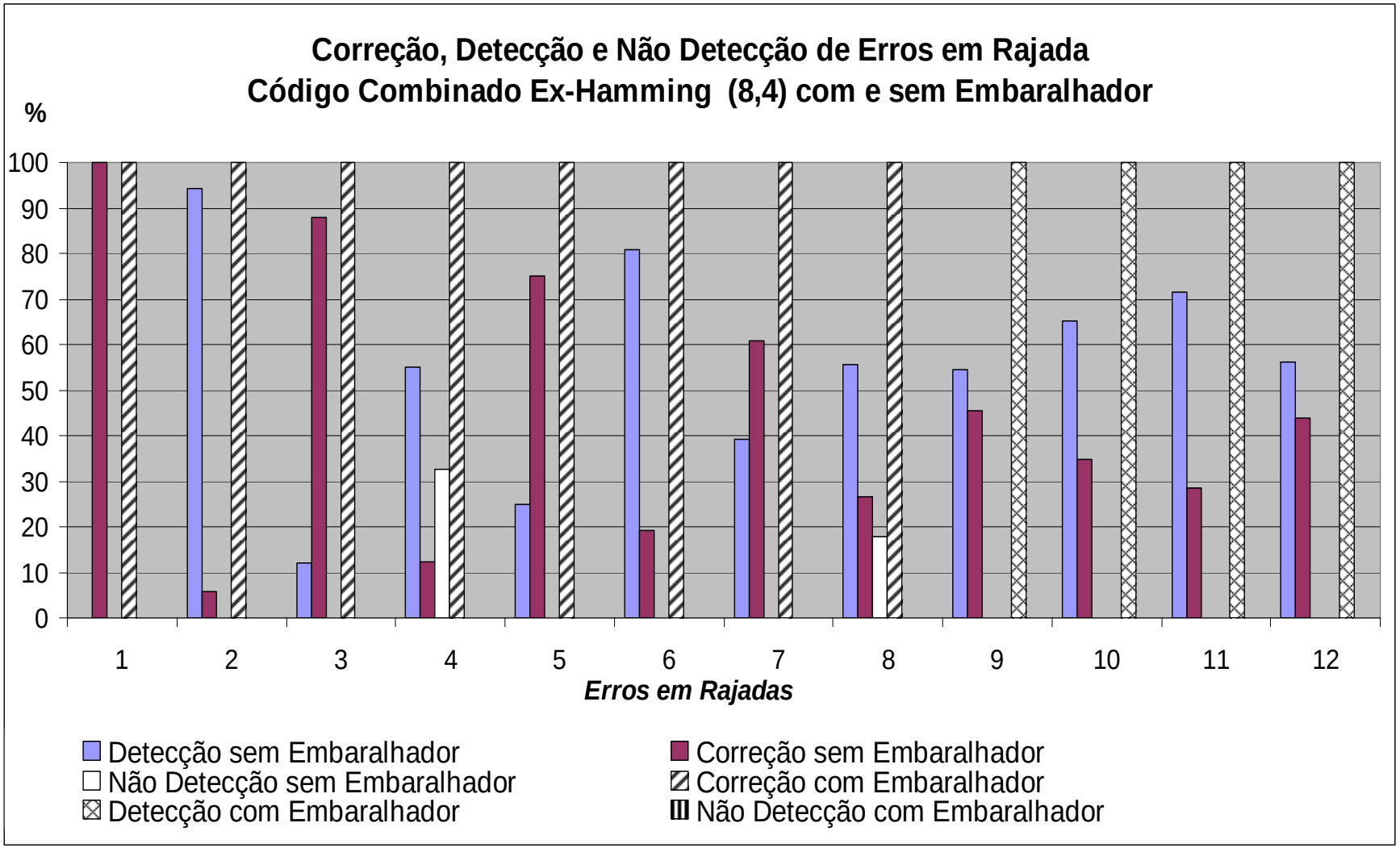


Figura 6.4 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (EHM (8,4)).

6.2.2. *Hamming* + Paridade (13,8)

Nas Tabelas 6.9 e 6.10 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* + Paridade (13,8) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.9 - Resultados dos testes realizados para EHM (13,8) sem embaralhador.

Rajada (*Bits*)	**Detecção (%)**	**Correção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	0,00	100,00	0,00	100,00	0,00
2	94,12	5,88	0,00	100,00	0,00
3	12,00	88,00	0,00	0,00	100,00
4	55,10	12,25	32,65	0,00	100,00
5	25,00	75,00	0,00	0,00	100,00
6	80,85	19,15	0,00	0,00	100,00
7	39,13	60,87	0,00	0,00	100,00
8	55,56	26,67	17,77	0,00	100,00
9	54,55	45,45	0,00	0,00	100,00
10	65,12	34,88	0,00	0,00	100,00
11	71,43	28,57	0,00	0,00	100,00
12	56,10	43,90	0,00	0,00	100,00

Observamos na Tabela 6.9 que na rajada de um erro o código combinado de *hamming* + Paridade (13,8) conseguiu corrigir todas as combinações de erros (100% de correção). Para dois erros conseguiu corrigir 5,88 % das combinações de erros e detectar as combinações restantes. Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foram corrigidas. Nos casos onde houve a detecção de erro, esses resultados são válidos.

Tabela 6.10 - Resultados dos testes realizados para EHM (13,8) com embaralhador.

Rajada (*Bits*)	**Detecção (%)**	**Correção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	0,00	100,00	0,00	100,00	0,00
2	0,00	100,00	0,00	100,00	0,00
3	0,00	100,00	0,00	100,00	0,00
4	0,00	100,00	0,00	100,00	0,00
5	100,00	0,00	0,00	0,00	0,00
6	100,00	0,00	0,00	0,00	0,00
7	100,00	0,00	0,00	0,00	0,00
8	100,00	0,00	0,00	0,00	0,00
9	100,00	0,00	0,00	0,00	0,00
10	100,00	0,00	0,00	0,00	0,00
11	100,00	0,00	0,00	0,00	0,00
12	0,00	100,00	0,00	0,00	100,00

Observamos na Tabela 6.10 que com o uso da técnica de embaralhamento o código combinado de *hamming* + Paridade (13,8) conseguiu corrigir todas as combinações de erros de até quatro *bits* (100% de correção certa). Para as demais rajadas de erros superiores (de cinco até onze *bits*) o código conseguiu detectar os erros. Para a rajada de doze *bits*, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código.

A Figura 6.5 apresenta os resultados do código combinado de *hamming* + Paridade (13,8) obtidos das Tabelas 6.9 e 6.10. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até quatro *bits* errados e detectar todas as combinações entre cinco e onze *bits*.

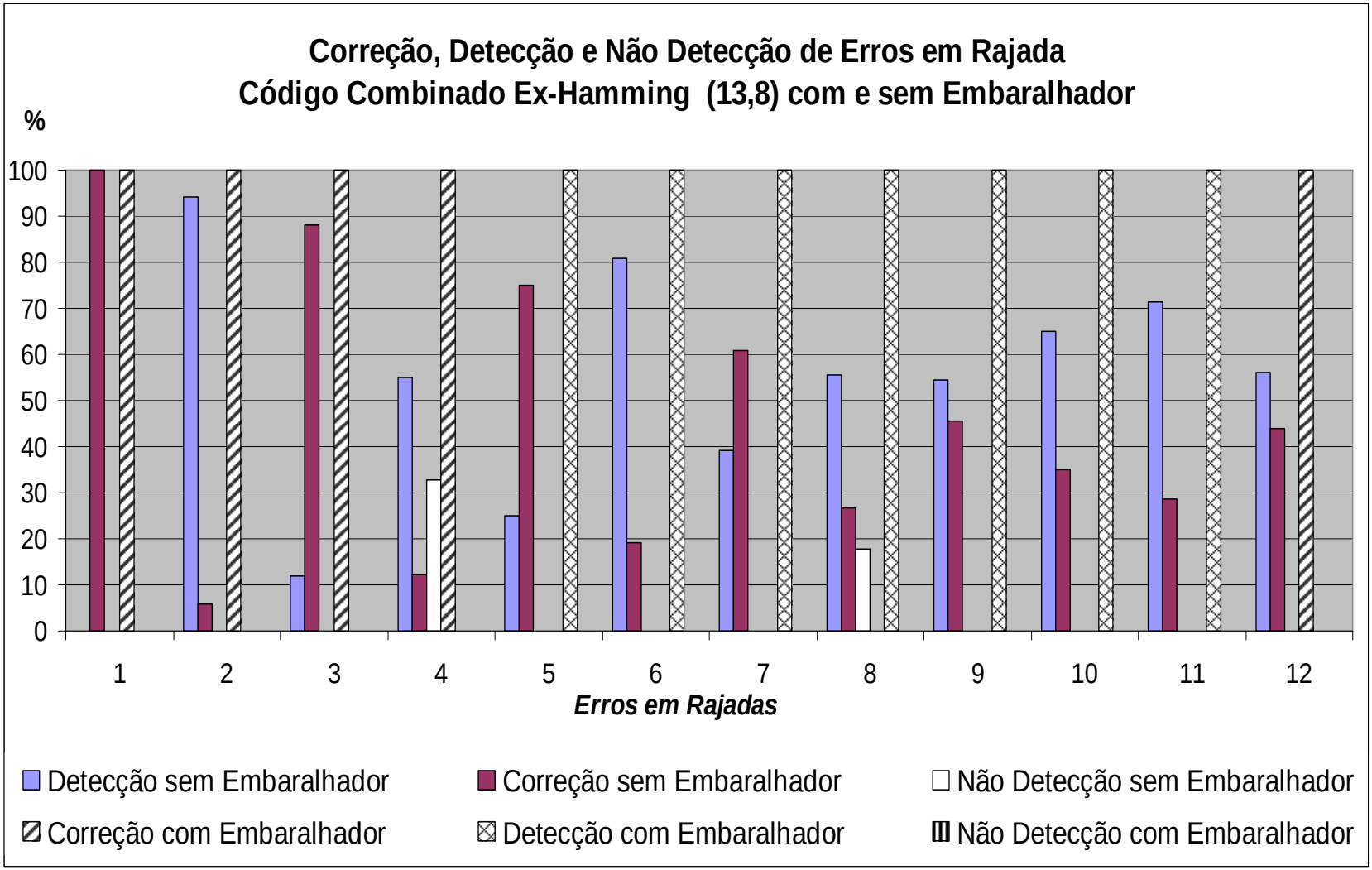


Figura 6.5 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (EHM (13,8)).

6.2.3. *Hamming* + Paridade (22,16)

Nas Tabelas 6.11 e 6.12 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado de *hamming* + Paridade (22,16) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.11 - Resultados dos testes realizados para EHM (22,16) sem embaralhador.

Rajada (*Bits*)	**Detecção (%)**	**Correção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	0,00	100,00	0,00	100,00	0,00
2	97,67	2,33	0,00	100,00	0,00
3	4,76	95,24	0,00	0,00	100,00
4	51,22	0,00	48,78	0,00	0,00
5	10,00	90,00	0,00	0,00	100,00
6	92,31	7,69	0,00	0,00	100,00
7	15,79	84,21	0,00	0,00	100,00
8	51,35	10,81	37,84	0,00	100,00
9	22,22	77,78	0,00	0,00	100,00
10	85,71	14,29	0,00	0,00	100,00
11	29,41	70,59	0,00	0,00	100,00
12	51,52	18,18	30,30	0,00	100,00

Observamos na Tabela 6.11 que na rajada de um erro o código combinado de *hamming* + Paridade (22,16) conseguiu corrigir todas as combinações de erros (100% de correção). Para rajada de dois erros o código conseguiu corrigir 2,33 % das combinações de erros e detectar as combinações restantes. Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não foram corrigidas. Nos casos onde houve a detecção de erro, esses resultados são válidos.

Tabela 6.12 - Resultados dos testes realizados para EHM (22,16) com embaralhador.

Rajada (*Bits*)	Detecção (%)	Correção (%)	Não Detecção (%)	Correção	
				Certa (%)	Errada (%)
1	0,00	100,00	0,00	100,00	0,00
2	0,00	100,00	0,00	100,00	0,00
3	100,00	0,00	0,00	0,00	0,00
4	100,00	0,00	0,00	0,00	0,00
5	100,00	0,00	0,00	0,00	0,00
6	0,00	100,00	0,00	0,00	100,00
7	52,63	47,37	0,00	0,00	100,00
8	75,68	0,00	24,32	0,00	0,00
9	50,00	50,00	0,00	0,00	100,00
10	0,00	100,00	0,00	0,00	100,00
11	100,00	0,00	0,00	0,00	0,00
12	100,00	0,00	0,00	0,00	0,00

Observamos na Tabela 6.12 que com o uso da técnica de embaralhamento o código combinado de *hamming* + Paridade (22,16) conseguiu corrigir todas as combinações de erros de até dois *bits* (100% de correção certa). Para as demais rajadas de erros superiores (de três até cinco *bits*) o código conseguiu detectar os erros. Para as rajadas superiores (de seis até dez *bits*), as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Nas rajadas de onze e doze *bits* as detecções são válidas.

A Figura 6.6 apresenta os resultados do código combinado de *hamming* + Paridade (22,16) obtidos das Tabelas 6.11 e 6.12. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até dois *bits* errados e detectar todas as combinações entre três e cinco *bits*.

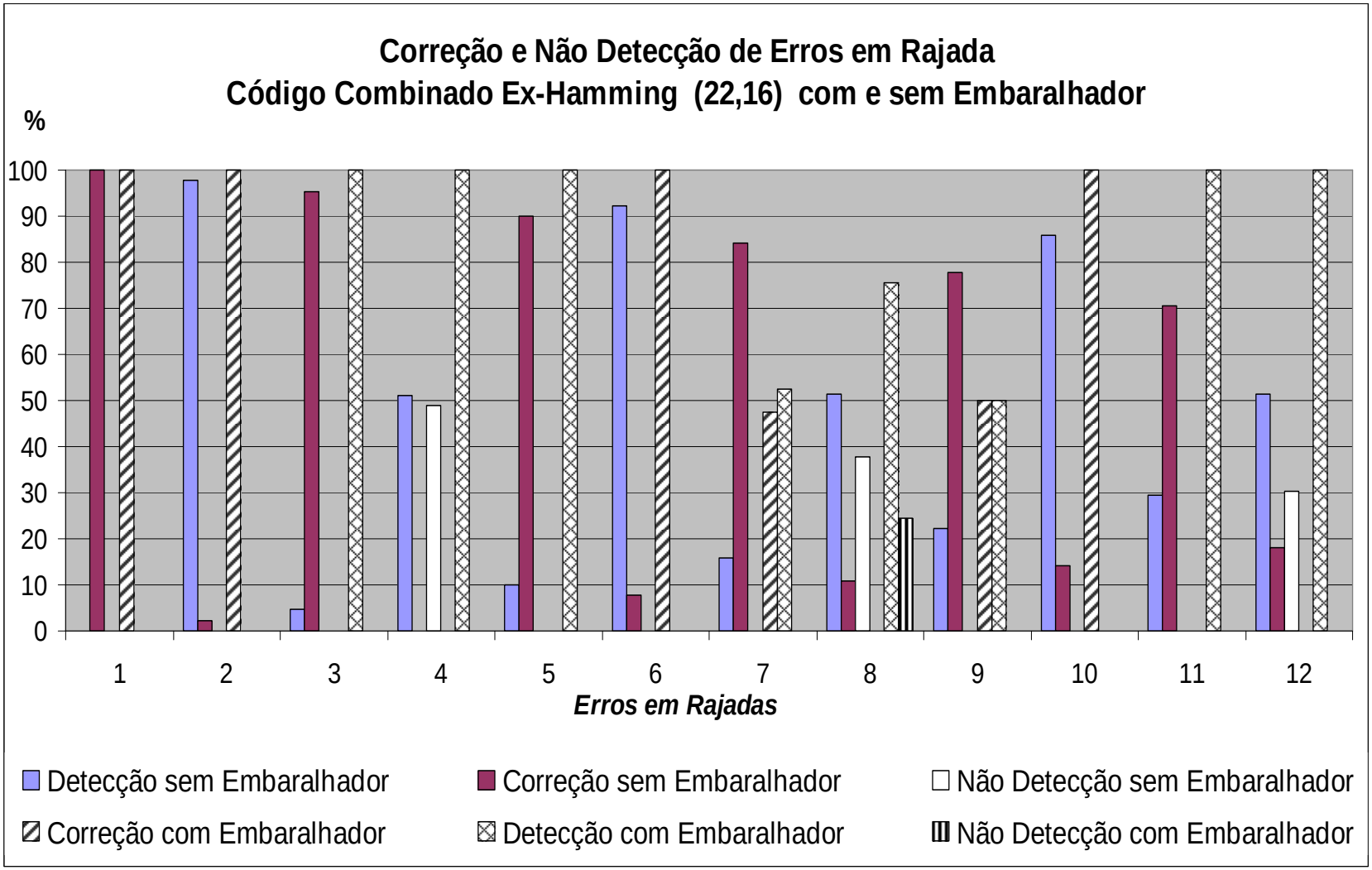


Figura 6.6 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (EHM (22,16)).

6.3. *Matrix Code*

6.3.1. *Matrix Code* (12,4)

Nas Tabelas 6.13 e 6.14 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado *Matrix Code* (12,4) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.13 - Resultados dos testes realizados para MC (12,4) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	65,59	34,41
4	92,55	7,45	0,00	43,68	56,32
5	66,30	7,61	26,09	34,43	65,57
6	90,11	8,79	1,10	52,44	47,56
7	83,33	7,78	8,89	29,33	70,67
8	92,13	0,00	7,87	17,07	82,93
9	92,05	7,95	0,00	17,28	82,72
10	91,95	0,00	8,05	8,50	92,50
11	100,00	0,00	0,00	8,14	91,86
12	82,35	0,00	17,65	0,00	100,00

Observamos na Tabela 6.13 que nas rajadas de até dois erros o código combinado *Matrix Code* (12,4) conseguiu corrigir todas as combinações de erros (100% de correção). Para as rajadas superiores, os resultados de correção não são confiáveis. Parte da correção foi realizada de maneira correta e outra parte das correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não poderam ser corrigidas. Os casos onde houve detecção de erro, esses resultados de detecção são válidos.

Tabela 6.14 - Resultados dos testes realizados para MC (12,4) com embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00	0,00
4	100,00	0,00	0,00	100,00	0,00
5	100,00	0,00	0,00	100,00	0,00
6	100,00	0,00	0,00	100,00	0,00
7	100,00	0,00	0,00	100,00	0,00
8	100,00	0,00	0,00	100,00	0,00
9	100,00	0,00	0,00	100,00	0,00
10	100,00	0,00	0,00	100,00	0,00
11	100,00	0,00	0,00	100,00	0,00
12	100,00	0,00	0,00	100,00	0,00

Observamos na Tabela 6.14 que com o uso da técnica de embaralhamento o código combinado *Matrix Code* (12,4) conseguiu corrigir todas as combinações de erros de até doze *bits* (100% de correção certa).

A Figura 6.7 apresenta os resultados do código combinado *Matrix Code* (12,4) obtidos das Tabelas 6.13 e 6.14. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir todas as combinações de erros em rajada de até dois *bits*. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até doze *bits*.

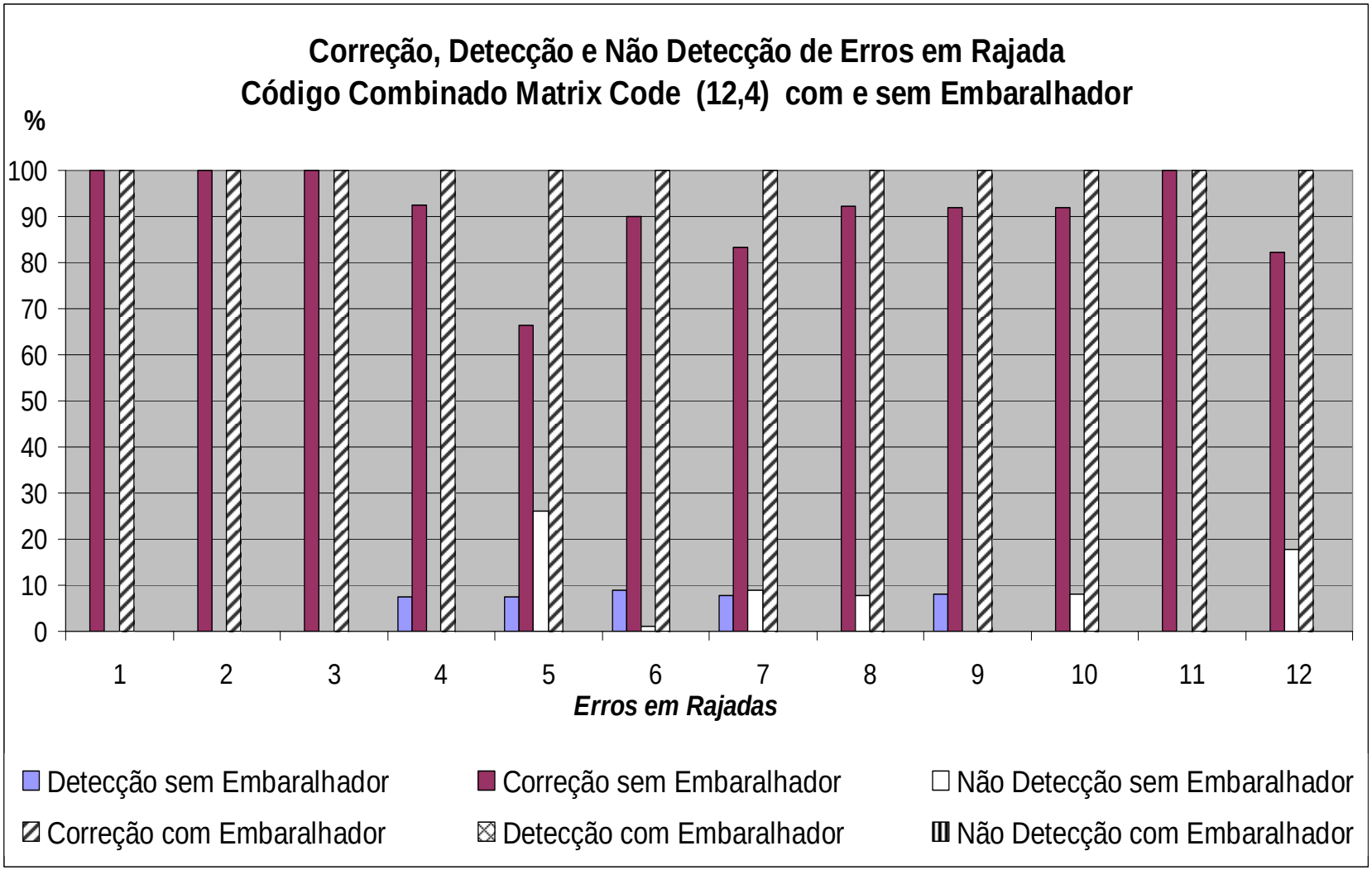


Figura 6.7 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (MC (12,4)).

6.3.2. *Matrix Code* (36,16)

Nas Tabelas 6.15 e 6.16 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado *Matrix Code* (36,16) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.15- Resultados dos testes realizados para MC (36,16) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	85,71	14,29
4	100,00	0,00	0,00	75,36	24,64
5	100,00	0,00	0,00	23,88	76,12
6	100,00	0,00	0,00	23,53	76,47
7	100,00	0,00	0,00	15,56	84,44
8	98,44	1,56	0,00	15,15	84,85
9	98,39	1,61	0,00	8,62	91,38
10	98,36	1,64	0,00	6,56	93,44
11	92,06	7,94	0,00	6,35	93,65
12	69,23	30,77	0,00	1,67	98,33

Observamos na Tabela 6.15 que nas rajadas de até dois erros o código combinado *Matrix Code* (36,16) conseguiu corrigir todas as combinações de erros (100% de correção). Para as rajadas superiores, os resultados de correção não são confiáveis. Parte da correção foi realizada de maneira correta e outra parte das correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Os casos onde houve detecção de erro, esses resultados de detecção são válidos.

Tabela 6.16 - Resultados dos testes realizados para MC (36,16) com embaralhador.

Rajada (*Bits*)	Correção (%)	Detecção (%)	Não Detecção (%)	Correção	
				Certa (%)	Errada (%)
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00	0,00
4	100,00	0,00	0,00	100,00	0,00
5	100,00	0,00	0,00	85,29	14,71
6	100,00	0,00	0,00	79,10	20,90
7	100,00	0,00	0,00	74,24	25,76
8	100,00	0,00	0,00	69,23	30,77
9	100,00	0,00	0,00	18,75	81,25
10	100,00	0,00	0,00	17,46	82,54
11	100,00	0,00	0,00	13,11	86,89
12	100,00	0,00	0,00	9,68	90,32

Observamos na Tabela 6.16 que com o uso da técnica de embaralhamento o código combinado *Matrix Code* (36,16) conseguiu corrigir todas as combinações de erros de até quatro *bits* (100% de correção certa). Para as rajadas superiores, os resultados de correção não são confiáveis. Parte da correção foi realizada de maneira correta e outra parte das correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código.

A Figura 6.8 apresenta os resultados do código combinado *Matrix Code* (36,16) obtidos das Tabelas 6.15 e 6.16. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir todas as combinações de erros em rajada de até dois *bits*. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até quatro *bits*.

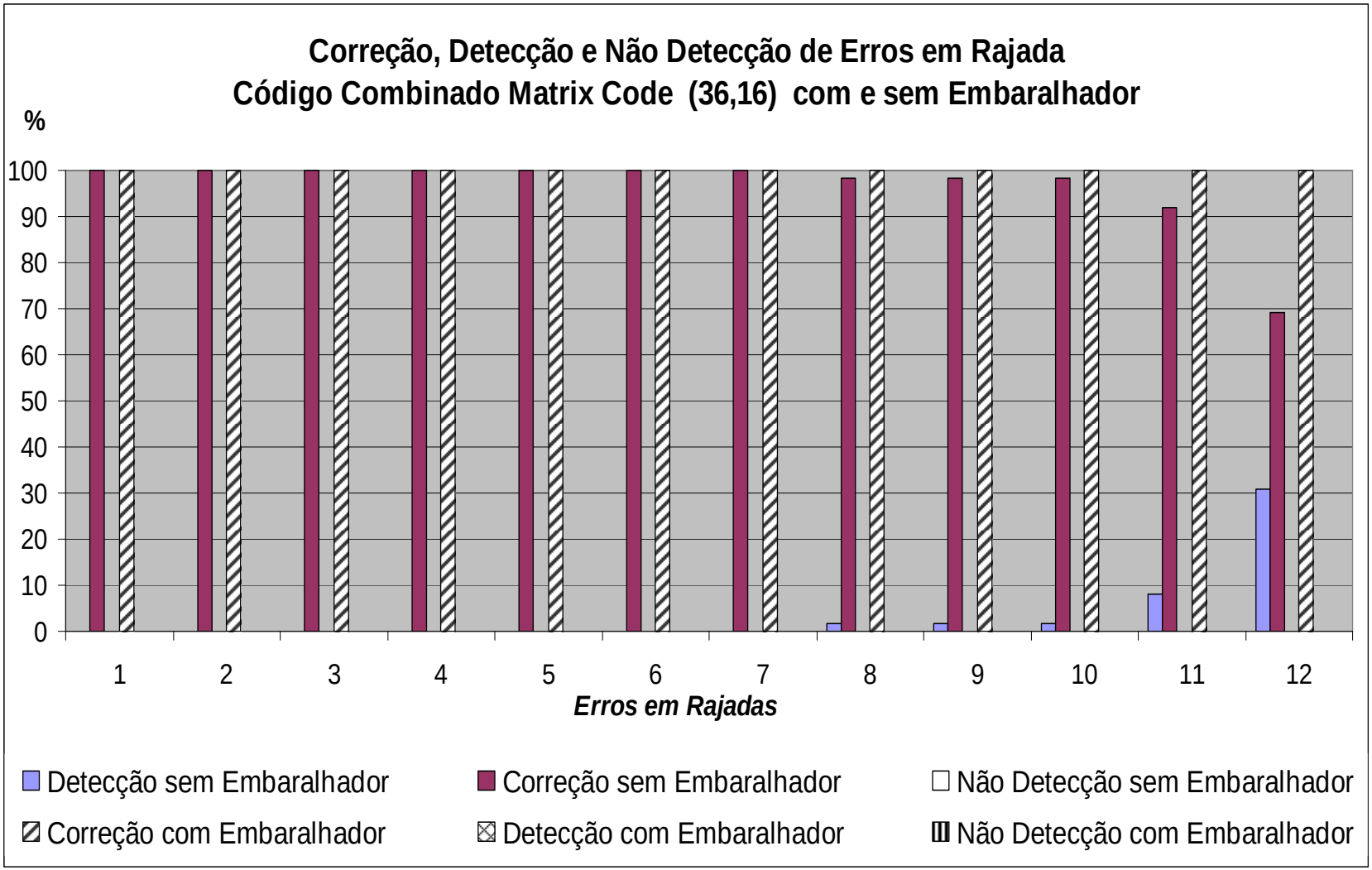


Figura 6.8 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (MC (36,16)).

6.4. *Reed-Muller*

6.4.1. *Reed-Muller* (8,4)

Nas Tabelas 6.17 e 6.18 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado *Reed-Muller* (8,4) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.17 - Resultados dos testes realizados para RM (8,4) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	100,00	0,00	0,00	100,00	0,00
2	11,11	88,89	0,00	100,00	0,00
3	77,42	22,58	0,00	14,58	85,42
4	23,36	37,30	39,34	50,00	50,00
5	76,67	23,33	0,00	0,00	100,00
6	35,59	64,41	0,00	0,00	100,00
7	51,72	48,28	0,00	0,00	100,00
8	49,12	24,56	26,32	0,00	100,00
9	50,00	50,00	0,00	0,00	100,00
10	49,09	50,91	0,00	0,00	100,00
11	51,85	48,15	0,00	0,00	100,00
12	49,05	24,53	26,42	0,00	100,00

Observamos na Tabela 6.17 que na rajada de um erro o código combinado *Reed-Muller* (8,4) conseguiu corrigir todas as combinações de erros (100% de correção). Para dois erros conseguiu corrigir 11,11 % das combinações de erros e detectar as combinações restantes. Para as demais rajadas, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Também, em muitos casos, os erros não foram detectados porque as falhas injetadas geraram uma nova Palavra-Código válida e desta forma não poderam ser corrigidas. Os casos onde houve detecção de erro, esses resultados de detecção são válidos.

Tabela 6.18 - Resultados dos testes realizados para RM (8,4) com embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção(%)**	**Correção**	
				Certa(%)	**Errada(%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00	0,00
4	100,00	0,00	0,00	100,00	0,00
5	100,00	0,00	0,00	100,00	0,00
6	100,00	0,00	0,00	100,00	0,00
7	100,00	0,00	0,00	100,00	0,00
8	100,00	0,00	0,00	100,00	0,00
9	0,00	100,00	0,00	0,00	0,00
10	0,00	100,00	0,00	0,00	0,00
11	0,00	100,00	0,00	0,00	0,00
12	0,00	100,00	0,00	0,00	0,00

Observamos na Tabela 6.18 que com o uso da técnica de embaralhamento o código combinado *Reed-Muller* (8,4) conseguiu corrigir todas as combinações de erros de até oito *bits* (100% de correção certa). Para as demais rajadas de erros superiores (de nove até doze *bits*) o código conseguiu detectar os erros.

A Figura 6.9 apresenta os resultados do código combinado *Reed-Muller* (8,4) obtidos das Tabelas 6.17 e 6.18. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir um erro. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até oito *bits* errados e detectar todas as combinações entre nove e doze *bits*.

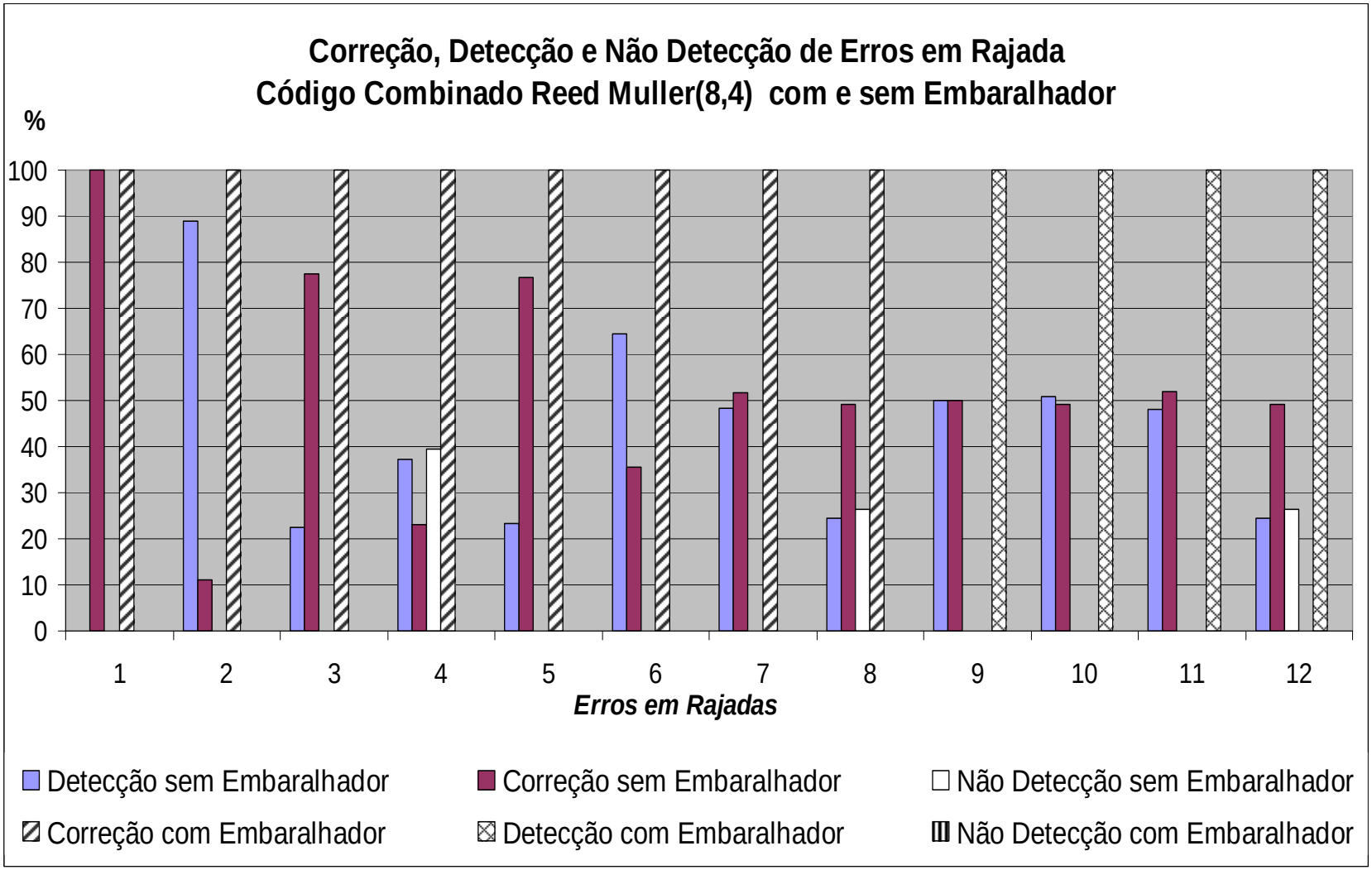


Figura 6.9 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (RM (8,4)).

6.4.2. *Reed-Muller* (32,16)

Nas Tabelas 6.19 e 6.20 são apresentados, respectivamente, os resultados dos testes da plataforma de validação do código combinado *Reed-Muller* (32,16) sem o uso do embaralhador e com o uso do embaralhador. Para cada teste de erro em rajada aplicado, as tabelas mostram as informações de correção, detecção e não detecção de erros. Analisando a informação da entrada do codificador com a informação da saída do decodificador, após a injeção de falhas, temos os resultados de correção certa e correção errada para cada teste de erro em rajada executado.

Tabela 6.19- Resultados dos testes realizados para RM (32,16) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção (%)**	**Correção**	
				Certa (%)	**Errada (%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00	0,00
4	100,00	0,00	0,00	100,00	0,00
5	6,67	93,33	0,00	100,00	0,00
6	5,08	94,92	0,00	100,00	0,00
7	3,45	96,55	0,00	100,00	0,00
8	1,75	98,25	0,00	100,00	0,00
9	0,00	100,00	0,00	0,00	0,00
10	0,00	100,00	0,00	0,00	0,00
11	0,00	100,00	0,00	0,00	0,00
12	23,08	76,92	0,00	0,00	100,00

Observamos na Tabela 6.19 que o código combinado *Reed-Muller* (32,16) conseguiu corrigir todas as combinações de erros de até quatro *bits* (100% de correção). Para as rajadas entre cinco *bits* e oito *bits* o código corrigiu uma pequena parte das combinações de erros e as demais foram detectadas. Para a rajada de doze *bits*, as correções foram realizadas de forma errada porque a magnitude dos erros extrapolou a capacidade de correção do código. Onde houve detecção de erro, esses resultados de detecção são válidos.

Tabela 6.20 - Resultados dos testes realizados para RM (32,16) sem embaralhador.

Rajada (*Bits*)	**Correção (%)**	**Detecção (%)**	**Não Detecção(%)**	**Correção**	
				Certa(%)	**Errada(%)**
1	100,00	0,00	0,00	100,00	0,00
2	100,00	0,00	0,00	100,00	0,00
3	100,00	0,00	0,00	100,00	0,00
4	100,00	0,00	0,00	100,00	0,00
5	100,00	0,00	0,00	100,00	0,00
6	100,00	0,00	0,00	100,00	0,00
7	100,00	0,00	0,00	100,00	0,00
8	100,00	0,00	0,00	100,00	0,00
9	0,00	100,00	0,00	0,00	0,00
10	0,00	100,00	0,00	0,00	0,00
11	0,00	100,00	0,00	0,00	0,00
12	0,00	100,00	0,00	0,00	0,00

Observamos na Tabela 6.20 que com o uso da técnica de embaralhamento o código combinado *Reed-Muller* (32,16) conseguiu corrigir todas as combinações de erros de até oito *bits* (100% de correção certa). Para as demais rajadas de erros superiores (de nove até doze *bits*) o código conseguiu detectar todos os erros.

A Figura 6.10 apresenta os resultados do código combinado *Reed-Muller* (32,16) obtidos das Tabelas 6.19 e 6.20. Podemos observar que com o uso do embaralhador, a capacidade de correção do código para erros em rajadas melhorou bastante. Sem o embaralhador, conseguíamos de maneira confiável corrigir todas as combinações de até quatro erros. Com a associação do embaralhador, é possível corrigir todas as combinações de erro em rajada de até oito *bits* errados e detectar todas as combinações entre nove e doze *bits*.

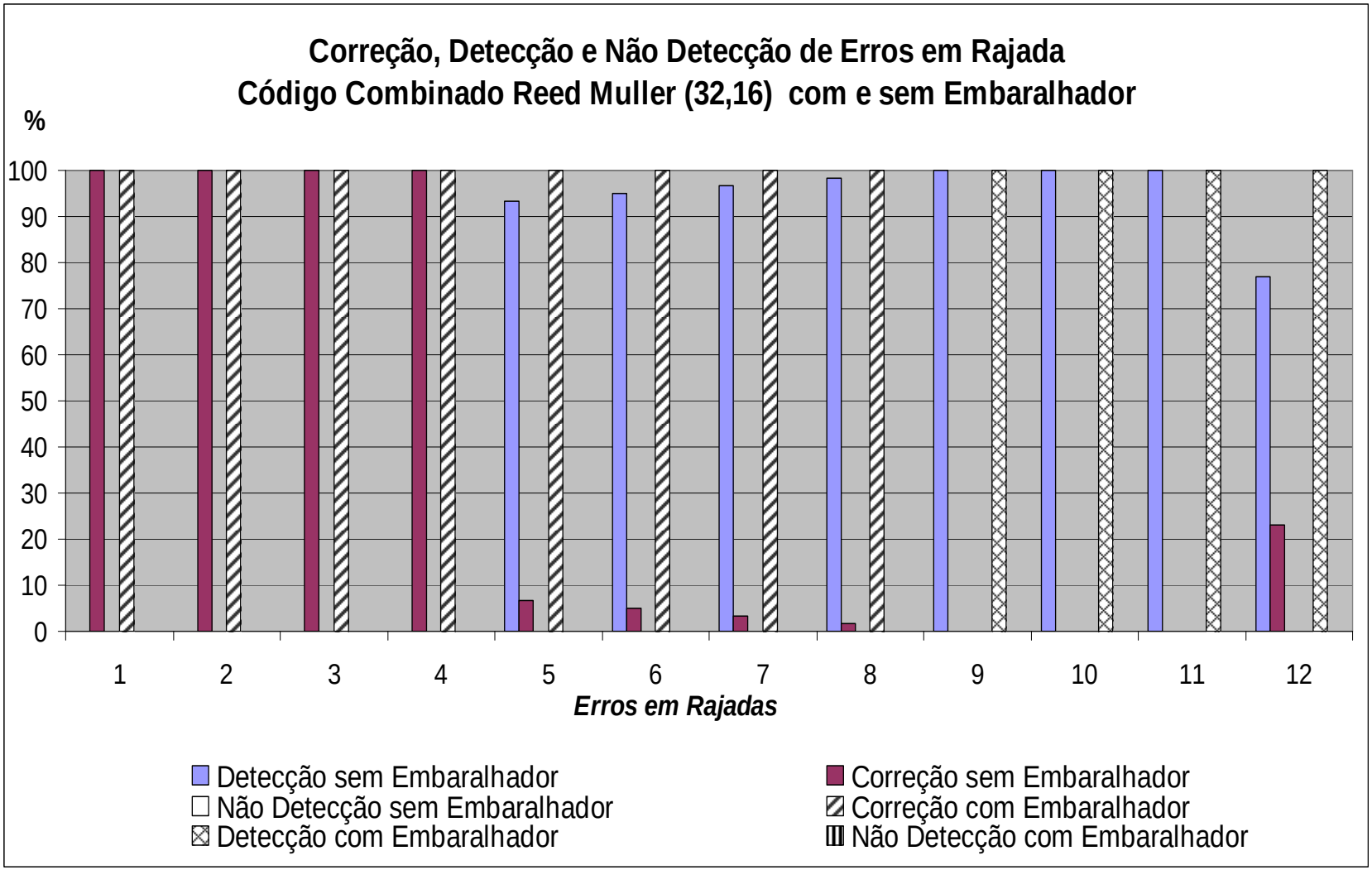


Figura 6.10 - Resultados de correção, detecção e não detecção com e sem o uso do embaralhador (RM(32,16)).

7. Caracterização dos Núcleos IP's

Os códigos corretores de erros foram sintetizados com a ferramenta ISE da XILINX. Foi utilizado o dispositivo xc2vp30 da família Xilinx Virtex II Pro. As tabelas apresentam os resultados de área ocupada no FPGA após a síntese e o período mínimo (ou freqüência máxima) de cada codificador e decodificador.

Tabela 7.1 – Resultado de Área e *timming* para os códigos de *Hamming*.

	***Hamming* (7,4)**				***Hamming* (12,8)**				***Hamming* (21,16)**			
	Codificador		Decodificador		Codificador		Decodificador		Codificador		Decodificador	
	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado
Número de Slices	2	14	5	37	3	14	13	53	7	16	27	53
Número de Slice Flip Flops	0	0	0	0	0	0	0	0	0	0	0	0
Número de 4 input LUTs	3	24	8	64	6	24	23	93	13	27	47	94
Número de pinos - IOBs	11	88	12	89	20	80	21	81	37	74	38	75
Entrada - IBUF	4	32	7	56	8	32	12	48	16	32	21	42
Saída - OBUF	7	56	5	33	12	48	9	33	21	42	17	33
Atraso (ns)	5,0830	5,0830	6,0340	7,6380	5,9020	5,9210	7,8810	7,8810	6,6250	6,6550	9,1590	9,8190
Frequência(MHz)	196,7342	196,7342	165,7275	130,9243	169,4341	168,8904	126,8875	126,8875	150,9434	150,2630	109,1822	101,8434

Tabela 7.2 – Resultado de Área e *timming* para os códigos de *Hamming* + Paridade.

	***Ex- Hamming* (8,4)**				***Ex- Hamming* (13,8)**				***Ex- Hamming* (22,16)**			
	Codificador		Decodificador		Codificador		Decodificador		Codificador		Decodificador	
	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado
Número de Slices	2	18	12	100	4	16	21	84	10	19	32	66
Número de Slice Flip Flops	0	0	3	24	0	0	3	12	0	0	3	6
Número de 4 input LUTs	4	32	16	137	7	28	31	127	17	34	51	105
Número de pinos - IOBs	12	96	15	99	21	84	24	87	38	76	41	79
Entrada - IBUF	4	32	8	64	8	32	13	52	16	32	22	44
Saída - OBUF	8	64	7	35	13	52	11	35	22	44	19	35
Atraso (ns)	5,1020	5,1020	6,1180	6,1180	5,8690	5,8690	7,8670	7,8670	6,6740	6,6740	8,9130	8,9130
Frequência(MHz)	196,0016	196,0016	163,4521	163,4521	170,3868	170,3868	127,1133	127,1133	149,8352	149,8352	112,1957	112,1957

Tabela 7.3 – Resultado de Área e *timming* para os códigos Matriz.

	***Matrix* (12,4)**				***Matrix* (36,16)**			
	Codificador		Decodificador		Codificador		Decodificador	
	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado
Número de Slices	2	18	17	140	12	23	54	109
Número de Slice Flip Flops	0	0	0	2	0	0	3	6
Número de 4 input LUTs	4	32	31	246	20	40	95	192
Número de pinos - IOBs	16	128	18	130	52	104	54	106
Entrada - IBUF	4	32	12	96	16	32	36	72
Saída - OBUF	12	96	6	34	36	72	18	34
Atraso (ns)	5,1190	5,1190	8,8760	8,8760	5,1190	5,1190	8,0420	8,0420
Frequência(MHz)	195,3507	195,3507	112,6634	112,6634	195,3507	195,3507	124,3472	124,3472

Tabela 7.4 – Resultado de Área e *timming* para os códigos *Reed-Muller*.

	***Reed-Muller* (8,4)**				***Reed-Muller* (32,16)**			
	Codificador		Decodificador		Codificador		Decodificador	
	Básico	Combinado	Básico	Combinado	Básico	Combinado	Básico	Combinado
Número de Slices	4	32	35	281	16	32	141	261
Número de Slice Flip Flops	0	0	0	0	0	0	0	0
Número de 4 input LUTs	7	56	62	495	28	56	248	493
Número de pinos - IOBs	12	96	14	98	48	96	50	98
Entrada - IBUF	4	32	8	64	16	32	32	64
Saída - OBUF	8	64	6	34	32	64	18	34
Atraso (ns)	5,1780	5,1780	14,5570	14,5570	5,1780	5,1780	16,1170	15,9280
Frequência(MHz)	193,1248	193,1248	68,6955	68,6955	193,1248	193,1248	62,0463	62,7825

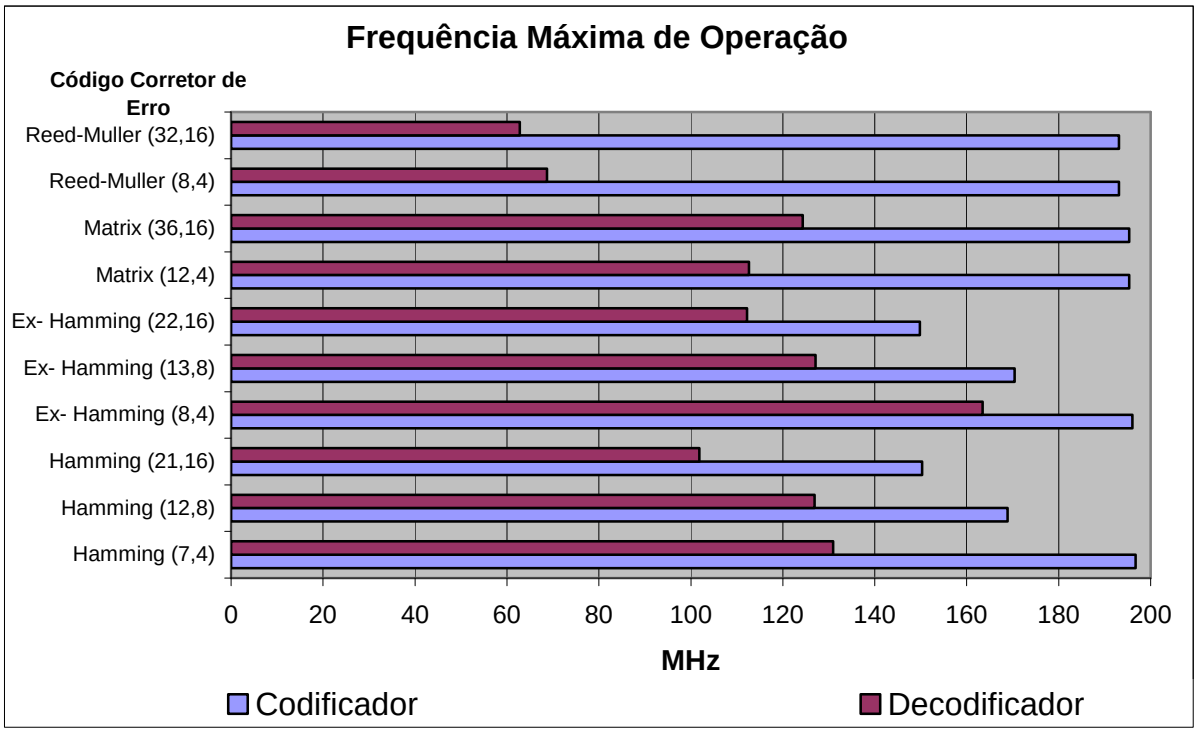


Figura 7.1 – Freqüência Máxima de operação de cada Código Corretor de Erro.

As tabelas 7.5, 7.6 e 7.7 apresentam, respectivamente, os desempenhos dos codificadores, decodificadores e ocupação de *hardware* de cada codificador/decodificador. Os resultados da tabela 7.5 e 7.6 são apresentados de forma decrescente, ou seja, os primeiros são mais velozes, trabalham em freqüências mais elevadas enquanto que os últimos são mais lentos, operando em freqüências menores. A tabela 7.7 apresenta de forma crescente a ocupação em *hardware* de cada código (Codificador/Decodificador).

Tabela 7.5 – Desempenho Codificadores.

Melhores Desempenhos (Codificador) (MHz)	
Hamming (7,4)	196,7342
Hamming + Paridade (8,4)	196,0016
Matrix (12,4)	195,3507
Matrix (36,16)	195,3507
Reed-Muller (8,4)	193,1248
Reed-Muller (32,16)	193,1248
Hamming + Paridade (13,8)	170,3868
Hamming (12,8)	168,8904
Hamming (21,16)	150,263
Hamming + Paridade (22,16)	149,8352

Tabela 7.6 – Desempenho Decodificadores.

Melhores Desempenhos (Decodificador) (MHz)	
Hamming + Paridade (8,4)	163,4521
Hamming (7,4)	130,9243
Hamming + Paridade (13,8)	127,1133
Hamming (12,8)	126,8875
Matrix (36,16)	124,3472
Matrix (12,4)	112,6634
Hamming + Paridade (22,16)	112,1957
Hamming (21,16)	101,8434
Reed-Muller (8,4)	68,6955
Reed-Muller (32,16)	62,7825

Tabela 7.7 – Ocupação de *hardware*.

Ocupação de Hardware
Hamming (7,4)
Hamming (12,8)
Hamming (21,16)
Hamming + Paridade (22,16)
Hamming + Paridade (13,8)
Hamming + Paridade (8,4)
Matrix (36,16)
Matrix (12,4)
Reed-Muller (8,4)
Reed-Muller (32,16)

Na tabela 7.5 se observa que a maioria dos códigos combinados a partir de 4 *bits* são mais rápidos porque envolvem um número menor de operações lógicas. A mesma linha de raciocínio pode ser adotada para se avaliar os decodificadores. O codificador mais rápido é o *Hamming* (7,4) que consegue trabalhar na freqüência máxima de 196,7342 MHz. O mais lento é o *Hamming* + Paridade (22,16) que trabalha na freqüência máxima de 149,8352 MHz. O decodificador mais rápido é o *Hamming* + Paridade (8,4) que consegue trabalhar na freqüência máxima de 163,4521 MHz e o mais lento é o *Reed-Muller* (32,16) que opera na freqüência máxima de 62,7825 MHz.Comparando as tabelas 7.5 e 7.6 temos que para cada Codec, o decodificador é mais lento que o codificador porque o processo de decodificação é mais complexo.

Tabela 7.8 – Número Máximo de Erros Corrigidos de cada Código.

Melhores Corretores com Embaralhamento	
Matrix (12,4)	12 erros
Hamming (7,4) *Hamming* + Paridade (8,4) *Reed-Muller* (8,4) *Reed-Muller* (32,16)	8 erros
Hamming (12,8) *Hamming* + Paridade (13,8) *Matrix* (36,16)	4 erros
Hamming (21,16) *Hamming* + Paridade (22,16)	2 erros

Tabela 7.9 – *Overhead* de Cada Código.

Overhead	
Hamming (21,16)	31,25%
Hamming + Paridade (22,16)	37,50%
Hamming (12,8)	50,00%
Hamming + Paridade (13,8)	62,50%
Hamming (7,4)	75,00%
Hamming + Paridade (8,4)	100,00%
Reed-Muller (8,4)	100,00%
Reed-Muller (32,16)	100,00%
Matrix (36,16)	125,00%
Matrix (12,4)	200,00%

A tabela 7.8 mostra o número máximo de erros corrigidos de cada Código com 100% de correção certa.. O código *Matrix*(12,4) apresentou o melhor resultado de erros corrigidos. Os códigos *Hamming* (21,16) e *Hamming* + Paridade (22,16) apresentaram os piores desempenhos.

A tabela 7.9 apresenta o *overhead* de cada código de maneira crescente. O código de *hamming*(21,16) apresenta o menor *overhead* (31,25%). O código *Matrix*(12,4) apresenta o maior *overhead*.

8. Conclusões

O constante avanço no processo de fabricação de circuitos integrados tem reduzido drasticamente a geometria dos transistores e os níveis das tensões de alimentação. Este processo de evolução torna os circuitos integrados, em especial as memórias, que atualmente chegam a representar até 70% de um sistema embarcado, muito vulneráveis às interferências tais como radiação, interferências eletromagnéticas, etc. que provocam não só apenas erros isolados, mas também vários erros concentrados que afetam vários bits armazenados nas memórias. A necessidade de proteger circuitos integrados contra esses problemas tornaram-se eminentes.

Desta forma, foi apresentada nesta dissertação uma proposta que combina a utilização de códigos corretores de erros associados a técnica de Embaralhamento objetivando um aumento na capacidade de detecção e correção de erros em rajada (erros concentrados). De um modo geral, para cumprir estes objetivos, foi necessário realizar as etapas abaixo:

1. Pesquisar, implementar e avaliar o comportamento em VHDL, de quatro tipos de códigos corretores de erros (*Hamming*, *Hamming* Estendido, *Reed-Muller* e *Matrix*), e suas respectivas variações de *bits* de entrada, totalizando 10 diferentes versões de códigos corretores de erros a serem validados;
2. Pesquisar, desenvolver e validar em VHDL uma plataforma de testes composta por Codificador, Decodificador, Embaralhador, Gerador de Informações e Injetor de Falhas;
3. Projetar e desenvolver as rotinas de injeção de falhas em VHDL.

Os resultados obtidos indicam que a combinação do uso do Embaralhador com códigos corretores de erros aumentam a taxa de detecção e correção de, pois o Embaralhador reordena os *bits* de tal modo que se houver uma interferência concentrada (rajada de erros) na memória, ao se fazer o desembaralhamento, os erros ficam distribuídos e aparecem como erros aleatórios ao decodificador. Este embaralhamento dos *bits* provoca um aumento da taxa de correção de erros e permite a utilização de códigos corretores de erros com baixa capacidade de detecção e correção ao invés de se usar outros códigos corretores de erros com

maior capacidade de detecção e correção que provocaria um aumento considerável de ocupação de memória (*overhead*), degradação de desempenho, maior área de hardware e consumo de energia. A execução dos testes de injeção de falhas do tipo *bit-flip* em rajadas, aplicadas às quatro técnicas corretoras de erros utilizadas nesta dissertação (*Hamming*, *Hamming* Estendido, *Reed-Muller* e *Matrix*), mostraram que apesar das mesmas serem dedicadas e mais eficientes para erros aleatórios, com o a associação da técnica de embaralhamento sobre as mesmas, estas técnicas passaram a ser eficientes também para erros em rajadas.

Mas para definir qual o melhor código a ser utilizado, é necessário levar em conta algumas considerações tais como *overhead*, capacidade de correção, freqüência máxima de operação e ocupação de *hardware*. Desta forma, a escolha do código poderá ficar condicionada a essas considerações. Poderá haver situações em que seja necessário utilizar um código com baixo *overhead* e para isto será necessário utilizar um código com baixa capacidade de correção. O exemplo é o *Hamming* (21,16) que apresenta o menor *overhead*, mas que também possui o pior desempenho em relação aos outros códigos estudados quando se considera capacidade de correção. Então, para se utilizar um código, será necessário fazer uma avaliação das características de cada um e escolher o que melhor atende as necessidades da aplicação que queremos implementar porque cada código tem as suas peculiaridades e dependendo da aplicação ele poderá ser melhor ou pior em relação aos outros códigos.

8.1. Trabalhos Futuros

Como sugestão de trabalhos futuros, apresentamos atividades que inicialmente constavam do planejamento inicial, mas que devido ao tempo consumido nas etapas de implementação computacional e execução dos testes, não foram incluídas neste trabalho. Estas atividades estão diretamente relacionadas à revisão bibliográfica apresentada e são elas:

a) Estudar novos códigos corretores de erros visando a melhoria na capacidade de detecção e correção e diminuição de *overhead*;
b) Realização de testes de injeção de falhas com erros aleatórios;
c) Ampliação dos testes de injeção de falhas que simulam erros em rajadas;
d) Automatização dos testes de injeção de falhas e
e) Executar testes de injeção de falhas fora do ambiente de simulação.

9. Referências Bibliográficas

[1] **Laprie, J. C.** *Dependable Computing and Fault-Tolerance: Concepts and Terminology.* Ann Arbor. New York : IEEE, 1985. Proceedings. pp. 2-11.

[2] **Anderson, T. e Lee, P.A.** *Fault Tolerance - Principles and Practice.* Englewood Cliffs : Prentice-Hall, 1981.

[3] **Pradhan, Dhiraj K.** *Fault-Tolerant Computer System Design.* Englewood Cliffs : Prentice Hall, 1996.

[4] **Iyer, R. K. e Kalbarczyk, Z.** *Hardware and Software Error Detection. [Online] 2002.* http://www.crhc.uiuc.edu/~kalbar/MotorolaCourse/HW&SW_ErrorDetection.pdf.

[5] **Bolzani, Letícia Maria Veiras.** *Explorando uma Solução Híbrida: Hardware + Software para a Detecção de Falhas em Systems-on-Chip (SoCs).* Faculdade de Engenharia, Programa de Pós-Graduação em Engenharia Elétrica. Porto Alegre : Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, 2005. Dissertação de Mestrado.

[6] **Laprie, J. C.** *Dependability: From concepts to limits.* Johannesburg, South Africa : s.n., 1998. Proceedings of the IFIP International Workshop on Dependable Computing and its Applications. DCIA 98. pp. 108-126.

[7] **Bardell, P. H.** *Built in Test for VLSI: Pseudorandom Techniques.* New York : John Wiley & Sons, 1987.

[8] **Stroud, E. C.** *A Designer's Guide to Built-In Self-Test.* Boston : Kluwer Academic Publishers, 2002. pp. 15-27.

[9] **Cortner, J. M.** *Digital Test Engineering.* United States of America : Wiley-Interscience, 1987. pp. 1-27.

[10] **Benfica, Juliano D'Ornelas.** *Plataforma para Desenvolvimento de SoC (System-on-Chip) Robusto à Interferência Eletromagnética.* Programa de Pós-Graduação em Engenharia Elétrica, Faculdade de Engenharia. Porto Alegre : Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, 2007. Dissertação de Mestrado.

[11] **Hefez, Abramo e Villela, Maria Lúcia**. *Códigos corretores de erros*. Rio de Janeiro: Impa,2002.

[12] **Shu Lin**, *"An Introduction to Error-Correcting Codes"*. USA ,Prentice-Hall, 1970.

[13] **Vera Pless**, *"Introduction to the Theory of Error-Correcting Codes"*. USA,John Wiley & Sons, 1982.

[14] **Bernard Sklar**, *Digital Communications, Prentice Hall*, 2 Ed.. New Jersey, 2001.

[15] **L. H. Charles Lee**, *"Error-Control Block Codes for Communications Engineers"*, Artech House, USA, 2000.

[16] **Hamming, R. W.**. *Error Detecting and Error Correcting Codes*. The Bell System Technical Journal, Vol. XXVI (1950), 147-160.

[17] **Haykin, Simon**. *Sistemas de Comunicação : analógicos e digitais. 4. ed***.** Bookman, Porto Alegre, 2004.

[18] **Shooman, M. L**., *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. John Wiley, New York, 2002.

[19] **Floyd, Thomas L**., *Sistemas Digitais - Fundamentos e Aplicações*. Bookman, Porto Alegre, 2007.

[20] **Costas Argyrides,** H.R. Zarandi, D.K. Pradhan, "*Multiple Upsets Tolerance in SRAM Memory*," Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), New Orleans, USA, 27-30 May, 2007.

[21] **Costas Argyrides, H.R. Zarandi, Dhiraj Pradhan**, *"Efficient Method to Tolerate Multiple Bit Upsets in SRAM Memory"*. Proceedings of European Test Symposium (ETS). May 2007.

[22] **Costas Argyrides, D.K. Pradhan**, *"Highly Reliable Power Aware Memory Design"* Proceedings of IEEE International On-Line Testing Symposium 2007 (IOLTS), Hersonisos of Heraklion, Crete, Greece, July 20-24, 2007.

[23] **Costas Argyrides, H.R. Zarandi, D.K. Pradhan**, "*Matrix Codes: Multiple Bit Upsets Tolerant Method for SRAM Memories*" Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems 2007, Rome, Italy, 26-28 Sept 2007.

[24] **Cooke, Ben**, "*Reed-Muller Error Correcting Codes*", MIT Undergraduate Journal of Mathematics, vol. 1, MIT Department of Mathematics, 1999.

[25] **Raaphorst, S.**. "*Reed-Muller Codes*". Carleton University, 2003.

[26] **S. Reed,** "*A Class of Multiple-Error-Correcting Codes and the Decoding Scheme*," IEEE Trans. Inf. Theory, vol. IT-4, pp. 38-49, 1954.

[27] **Proakis, John G.,** *Communications Systems Engineering, 2 Ed* . Prentice Hall, New Jersey, 2002.

[28] **T. K. Moon**, *Error Correcting Coding: Mathematical Methods and Algorithms.* Wiley, New York, 2005.

[29] **B. P. Lathi**, *Modern Digital and Analog Communication Systems, 3. ed*., Oxford University Press, New York, 1998.

[30] **K.G. Hoffman, et al**, *Coding Theory, The Essentials*. Marcel Dekker, New York, 1991.

[31] **Hanna S. A.** *Convolutional interleaving for digital radio communications***.** ICUPC'93**.** 1993, IEEE, vol.: 1, p. 443-447.

[32] **Kastensmidt, F. G. et al**. *Designing and Testing Fault-Tolerant Techniques for SRAM-based FPGAs*. In: ACM Computer Frontiers Conference (1 : 2004 : Ischia). Proceedings. New York : ACM, 2004.

[33] **Johnston, A**. "*Scaling and Technology Issues for Soft Error Rates*", In: roceedings of 4th Annual Research Conference on Reliability, Stanford University, Oct 2000.

[34] **Reed, R.** "*Heavy Ion and Proton Induced Single Event Multiple Upsets*", In: Proceedings. of IEEE Nuclear and Space Radiation Effects Conference (NSREC), July 1997.

[35] **Wrobel, F., Palau, J., Calvet, M., Bersillon, O., and Duarte, H.** "*Simulation of Nucleon- Induced Nuclear Reactions in a Simplified SRAM Structure: Scaling Effects on SEU and MBU Cross Sections*", In: Proceedings of IEEE Transactions on Nuclear Science, December,2001.

[36] **Johansson, K., Ohlsson, M., Olsson, N., Blomgren, J., and Renberg, P**. "*Neutron Induced Single-Word Multiple-bit Upset in SRAM*", In: Proceedings of IEEE Transactions on Nuclear Science, December 1999.

[37] **Buchner, S., Campbell, A., Meehan, T., Clark, K., McMorrow, D., Dyer, C., Anderson, C.,Comber, C., and Kuboyama, S.** "*Investigation of Single-Ion Multiple-Bit Upsets in Memories on Board a Space Experiment*", In: Proceedings of IEEE Transactions on Nuclear Science, June 2000.

[38] **Kastensmidt, F. G.** "*Fault-tolerance techniques for SRAM-based FPGAs"*. Netherlands: Springer, c2006 xv, 183 p.

[39] **Vargas F.L.; Nicolaidis, M.**, "*SEU-Tolerant SRAM Design Based on Current Monitoring*," Proc. IEEE Int'l Symp. Fault-Tolerant Computing, *pp. 106-115, 1994.*

[40] **Vargas, F. ; Nicolaidis, M. .** *SEU-Tolerant SRAM Design Based on Current Monitoring*. In: 24th FTCS - International Symposium on Fault-Tolerant Computing, 1994, Austin - TX, USA. Proceedings of the 24th FTCS - International Symposium on Fault-Tolerant Computing. Los Alamitos - CA, USA : IEEE Computer Society, 1994. v. único. p. 106-115.

[41] **Vargas, F. ; Nicolaidis, M. ; Courtois, B.** . *Quiescent Current Monitoring to Improve the Reliability of Electronic Systems in Space Environments. In: International Conference on Computer Design* - ICCD'93, 1993, Cambridge - MA, USA. Proceedings of the International Conference on Computer Design - ICCD'93. Los Alamitos - CA, USA : IEEE Computer Society, 1993. v. único. p. 596-600.

[42] **Vargas, F. ; Nicolaidis, M.** . *Current Testing for ICs Used in Space Applications. In: Workshop on Current Testing,* 1992, Barcelona, Spain, 1992.

[43] **Vargas, F. ; Nicolaidis, M. ; Courtois, B.** . *Design of Reliable Current Sensors for Radiation Exposed Environments*. In: 30th Annual International Nuclear and Space Radiation Effects Conference - NSREC'93, 1993, Snowbird - Utah, USA, 1993

[44] **Houghton, A. D**. "*The Engineer's Error Coding Handbook*" Chapman & Hall, London. Single-Word Multiple-bit Upset in SRAM", In: Proceedings of IEEE Transactions on Nuclear Science, December 1997.

[45] **Keitel-Schulz, D**. "*Memory Issues in SoC*", MPSoC 2004, Munich, Germany, última visita em 2 de julho de 2008. http://tima.imag.fr/mpsoc/2004/slides/dks.pdf.http://www.mpsoc-forum.org/2004/slides/dks.pdf

[46] **Proakis, John G**.. *Digital Communications*, 4 Ed. McGraw-Hill, New York, 2002.

[47] http://www.altera.com**.** Última visita em 2 de julho de 2008.

[48] http://www.xilinx.com. Última visita em 2 de julho de 2008.

ANEXO A1 – Código VHDL de um Núcleo IP Corretor de Erro Hamming

CODIFICADOR

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity HMe_7_4 is
port(
N: out std_logic_vector(1 to 7);                    -- codeword (n= 7 bits)
K: in std_logic_vector(1 to 4)          -- vetor mensagem (K= 4 bits)
);
end  HMe_7_4;

architecture ENCODER of HMe_7_4 is
signal P:std_logic_vector(1 to 3); -- Paridade
begin

P(1)<=K(1) XOR K(2) XOR K(4) ; -- Calcula  bit de paridade P1
P(2)<=K(1) XOR K(3) XOR K(4) ; -- Calcula  bit de paridade P2
P(3)<=K(2) XOR K(3) XOR K(4) ; -- Calcula  bit de paridade P3
N(1)<=P(1); -- recebe P1(paridade Par)
N(2)<=P(2);          -- recebe P2(paridade Par)
N(3)<=K(1);          -- recebe dado K1
N(4)<=P(3);          -- recebe P3(paridade Par)
N(5)<=K(2);          -- recebe dado K2
N(6)<=K(3);          -- recebe dado K3
N(7)<=K(4);          -- recebe dado K4

 end ENCODER;
```

DECODIFICADOR

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity HMd_7_4 is
port(
N: in std_logic_vector(1 to 7);-- codeword (n= 7 bits)
K: out std_logic_vector(1 to 4);-- vetor mensagem (K= 4 bits)
SEC: out std_logic :='0'-- indica se houve detecção e correção de um erro (ativo alto)
);
end  HMd_7_4;

architecture DECODER of HMd_7_4 is
signal CP:std_logic_vector(1 to 3); -- Paridade
signal ND :std_logic_vector(1 to 7);

begin
CP(1)<=N(1) XOR N(3) XOR N(5) XOR N(7); -- Verifica bit de paridade P1
CP(2)<=N(2) XOR N(3) XOR N(6) XOR N(7); -- Verifica bit de paridade P2
CP(3)<=N(4) XOR N(5) XOR N(6) XOR N(7); -- Verifica bit de paridade P3
SEC<=CP(1) OR CP(2) or CP(3);                -- Verifica se houve SEC (ativo alto)
ND<=N;

process(CP,N)-- calcula a posição do bit  onde ocorreu erro e corrigi
begin
if CP="100" then                -- bit 1
          K(1)<=ND(3);
          K(2)<=ND(5);
          K(3)<=ND(6);
          K(4)<=ND(7);
elsif CP="010" then             -- bit 2
          K(1)<=ND(3);
          K(2)<=ND(5);
          K(3)<=ND(6);
          K(4)<=ND(7);
elsif CP="110" then             -- bit 3
          K(1)<=not ND(3);
          K(2)<=ND(5);
          K(3)<=ND(6);
          K(4)<=ND(7);
elsif CP="001" then             -- bit 4
          K(1)<=ND(3);
          K(2)<=ND(5);
          K(3)<=ND(6);
          K(4)<=ND(7);
elsif CP="101" then             -- bit 5
          K(1)<=ND(3);
```

```
            K(2)<=not ND(5);
            K(3)<=ND(6);
            K(4)<=ND(7);
elsif CP="011" then             -- bit 6
            K(1)<=ND(3);
            K(2)<=ND(5);
            K(3)<=not ND(6);
            K(4)<=ND(7);
elsif CP="111" then             -- bit 7
            K(1)<=ND(3);
            K(2)<=ND(5);
            K(3)<=ND(6);
            K(4)<=not ND(7);
else
            K(1)<=ND(3);
            K(2)<=ND(5);
            K(3)<=ND(6);
            K(4)<=ND(7);
end if;
end process;

end DECODER;
```

ANEXO A2 – Código VHDL de um Núcleo IP Corretor de Erro Hamming + Paridade

CODIFICADOR

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ExHME_8_4 is
port(
N: out std_logic_vector(1 to 8);                    -- codeword (n= 7 bits)
K: in std_logic_vector(1 to 4)                      -- vetor mensagem (K= 4 bits)
);
end  ExHME_8_4;

architecture ENCODER of ExHME_8_4 is
signal P:std_logic_vector(1 to 4); -- Paridade
begin

P(1)<=K(1) XOR K(2) XOR K(4) ; -- Calcula  bit de paridade P1
P(2)<=K(1) XOR K(3) XOR K(4) ; -- Calcula  bit de paridade P2
P(3)<=K(2) XOR K(3) XOR K(4) ; -- Calcula  bit de paridade P3
P(4)<=K(1) XOR K(2) XOR K(3) XOR K(4) XOR P(1)XOR P(2)XOR P(3); -- Calcula  bit de paridade P3

N(1)<=P(1); -- recebe P1(paridade Par)
N(2)<=P(2);          -- recebe P2(paridade Par)
N(3)<=K(1);          -- recebe dado K1
N(4)<=P(3);          -- recebe P3(paridade Par)
N(5)<=K(2);          -- recebe dado K2
N(6)<=K(3);          -- recebe dado K3
N(7)<=K(4);          -- recebe dado K4
N(8)<=P(4);          -- recebe P4(paridade Par)

end ENCODER;
```

DECODIFICADOR

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder_8_4 is
port(
N: in std_logic_vector(1 to 8);-- codeword (n= 7 bits)
K: out std_logic_vector(1 to 4);-- vetor mensagem (K= 4 bits)
SEC: out std_logic :='0';-- indica se houve detecção e correção de um erro (ativo alto)
DED: out std_logic :='0';-- indica se houve detecção de dois erros (ativo alto)
ParityError: out std_logic :='0'-- indica se não houve erro (ativo baixo)

);
end  decoder_8_4;

architecture DECODER of decoder_8_4 is
signal CP:std_logic_vector(1 to 3); -- Paridade
signal ND :std_logic_vector(1 to 8);
signal P: std_logic;
begin

CP(1)<=N(7) XOR N(5) XOR N(3) XOR N(1); -- Verifica bit de paridade P1
CP(2)<=N(7) XOR N(6) XOR N(3) XOR N(2); -- Verifica bit de paridade P2
CP(3)<=N(7) XOR N(6) XOR N(5) XOR N(4); -- Verifica bit de paridade P3
P<=N(1) XOR N(2) XOR N(3) XOR N(4) xor N(5) XOR N(6) XOR N(7) XOR N(8); -- verifica all parity

ND<=N;

process(CP,N)
begin
if CP="100" then
        K(1)<=ND(3);
        K(2)<=ND(5);
        K(3)<=ND(6);
        K(4)<=ND(7);
elsif CP="010" then
        K(1)<=ND(3);
        K(2)<=ND(5);
        K(3)<=ND(6);
        K(4)<=ND(7);
elsif CP="110" then
        K(1)<=not ND(3);
```

```
		K(2)<=ND(5);
		K(3)<=ND(6);
		K(4)<=ND(7);
elsif CP="001" then
		K(1)<=ND(3);
		K(2)<=ND(5);
		K(3)<=ND(6);
		K(4)<=ND(7);
elsif CP="101" then
		K(1)<=ND(3);
		K(2)<=not ND(5);
		K(3)<=ND(6);
		K(4)<=ND(7);
elsif CP="011" then
		K(1)<=ND(3);
		K(2)<=ND(5);
		K(3)<=not ND(6);
		K(4)<=ND(7);
elsif CP="111" then
		K(1)<=ND(3);
		K(2)<=ND(5);
		K(3)<=ND(6);
		K(4)<=not ND(7);
else
		K(1)<=ND(3);
		K(2)<=ND(5);
		K(3)<=ND(6);
		K(4)<=ND(7);
end if;
end process;

SECDEC: process(CP,N,P)
begin

if CP="000" and P='0' then  --Não ocorreu nenhum Erro
		SEC<='0';
		DED<='0';
		ParityError<='0';

elsif CP/="000" and P='1' then --SEU ocorreu mas pode ser corrigido (SEC)
		SEC<='1';
		DED<='0';
		ParityError<='0';

elsif CP/="000" and P='0' then --Double error detected mas não pode ser corrigido
		SEC<='0';
		DED<='1';
		ParityError<='0';

elsif CP="000" and P='1' then --erro de paridade (P8)
		SEC<='0';
		DED<='0';
		ParityError<='1';
end if;
end process;

end DECODER;
```

ANEXO A3 – Código VHDL de um Núcleo IP Corretor de Erro *Matrix*

CODIFICADOR

```
---------------------------------------------------------------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY cod_all IS
	port
	(
		XI :  IN  STD_LOGIC_VECTOR(0 to 3);
		N :  OUT  STD_LOGIC_VECTOR(0 to 11)
	);
END cod_all;

ARCHITECTURE bdf_type OF cod_all IS

component encoder
	PORT(x : IN STD_LOGIC_VECTOR(0 to 3);
		 CP : OUT STD_LOGIC_VECTOR(0 to 2);
		 P : OUT STD_LOGIC_VECTOR(0 to 3)
	);
end component;

component codall
	PORT(Ov : IN STD_LOGIC;
		 CP : IN STD_LOGIC_VECTOR(0 to 2);
		 P : IN STD_LOGIC_VECTOR(0 to 3);
		 x : IN STD_LOGIC_VECTOR(0 to 3);
		 N : OUT STD_LOGIC_VECTOR(0 to 11)
	);
end component;

component enover1
	PORT(CP : IN STD_LOGIC_VECTOR(0 to 2);
		 x : IN STD_LOGIC_VECTOR(0 to 3);
		 Ov : OUT STD_LOGIC
	);
end component;

signal	SYNTHESIZED_WIRE_0 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(0 to 2);
signal	SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(0 to 3);


BEGIN


b2v_inst : encoder
PORT MAP(x => XI,
		 CP => SYNTHESIZED_WIRE_4,
		 P => SYNTHESIZED_WIRE_2);

b2v_inst2 : codall
PORT MAP(Ov => SYNTHESIZED_WIRE_0,
		 CP => SYNTHESIZED_WIRE_4,
		 P => SYNTHESIZED_WIRE_2,
		 x => XI,
		 N => N);

b2v_inst6 : enover1
PORT MAP(CP => SYNTHESIZED_WIRE_4,
		 x => XI,
		 Ov => SYNTHESIZED_WIRE_0);

END;
---------------------------------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------------------------------
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity codall is
port(
```

```
x: in std_logic_vector(0 to 3);
CP: in std_logic_vector(0 to 2);
Ov: in std_logic;
P : in std_logic_vector(0 to 3);
N: out std_logic_vector(0 to 11)
);
end  codall;


architecture cod of codall is
begin

N(0 to 3)<=x;
N(4 to 6)<=CP;
N(7)<=Ov;
N(8 to 11)<=P;

end cod;
```

```
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Encoder is
port(
x: in std_logic_vector(0 to 3);
P : out std_logic_vector(0 to 3);
CP: out std_logic_vector(0 to 2)
);
end  Encoder;

architecture ENCODER of Encoder is
begin
P(0) <=X(0) ;
P(1) <=X(1) ;
P(2) <=X(2) ;
P(3) <=X(3) ;

CP(0)<= X(0) XOR X(1) XOR X(3);
CP(1)<= X(0) XOR X(2) XOR X(3);
CP(2)<= X(1) XOR X(2) XOR X(3);

end ENCODER;
```

```
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EnOver1 is
port(
x: in std_logic_vector(0 to 3);
CP: in std_logic_vector(0 to 2);
Ov: out std_logic
);
end  EnOver1;

architecture EnOver1 of EnOver1 is
begin

ov<=   CP(0) XOR CP(1) XOR CP(2)   XOR X(0) XOR X(1) XOR X(2)  XOR X(3) ;

end EnOver1;
```

DECODIFICADOR

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY decodificador IS
```

```
	port
	(
		XI :  IN  STD_LOGIC_VECTOR(0 to 11);
		EC :  OUT  STD_LOGIC;
		ED :  OUT  STD_LOGIC;
		Xo :  OUT  STD_LOGIC_VECTOR(0 to 3)
	);
END decodificador;

ARCHITECTURE bdf_type OF decodificador IS

component dec_out
	PORT(SEC : IN STD_LOGIC;
		DED : IN STD_LOGIC;
		P1 : IN STD_LOGIC;
		P2 : IN STD_LOGIC;
		P3 : IN STD_LOGIC;
		P4 : IN STD_LOGIC;
		Ni : IN STD_LOGIC_VECTOR(0 to 3);
		EC : OUT STD_LOGIC;
		ED : OUT STD_LOGIC;
		No : OUT STD_LOGIC_VECTOR(0 to 3)
	);
end component;

component decodall
	PORT(Ni : IN STD_LOGIC_VECTOR(0 to 11);
		Ov : OUT STD_LOGIC;
		CP : OUT STD_LOGIC_VECTOR(0 to 2);
		P : OUT STD_LOGIC_VECTOR(0 to 3);
		x : OUT STD_LOGIC_VECTOR(0 to 3)
	);
end component;

component deover1
	PORT(so : IN STD_LOGIC;
		CP : IN STD_LOGIC_VECTOR(0 to 2);
		x : IN STD_LOGIC_VECTOR(0 to 3);
		sov : OUT STD_LOGIC
	);
end component;

component dec
	PORT(d : IN STD_LOGIC;
		cx : IN STD_LOGIC_VECTOR(0 to 3);
		sp : IN STD_LOGIC_VECTOR(0 to 3);
		sx : IN STD_LOGIC_VECTOR(0 to 3);
		f : OUT STD_LOGIC_VECTOR(0 to 3)
	);
end component;

component dec_h
	PORT(s : IN STD_LOGIC;
		scp : IN STD_LOGIC_VECTOR(0 to 2);
		sx : OUT STD_LOGIC_VECTOR(0 to 3)
	);
end component;

component dec_o
	PORT(Sov : IN STD_LOGIC;
		SCP : IN STD_LOGIC_VECTOR(0 to 2);
		SP : IN STD_LOGIC_VECTOR(1 to 4);
		SEC : OUT STD_LOGIC;
		DED : OUT STD_LOGIC;
		P1 : OUT STD_LOGIC;
		P2 : OUT STD_LOGIC;
		P3 : OUT STD_LOGIC;
		P4 : OUT STD_LOGIC
	);
end component;

component decoder
	PORT(CP : IN STD_LOGIC_VECTOR(0 to 2);
		P : IN STD_LOGIC_VECTOR(0 to 3);
		x : IN STD_LOGIC_VECTOR(0 to 3);
		SCP : OUT STD_LOGIC_VECTOR(0 to 2);
```

```
                SP : OUT STD_LOGIC_VECTOR(0 to 3)
        );
end component;

component synd_d
        PORT(S : IN STD_LOGIC;
                SCP : IN STD_LOGIC_VECTOR(0 to 2);
                d : OUT STD_LOGIC
        );
end component;

signal  SYNTHESIZED_WIRE_0 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_1 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_2 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_3 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_4 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_5 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_6 :  STD_LOGIC_VECTOR(0 to 3);
signal  SYNTHESIZED_WIRE_7 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_24 :  STD_LOGIC_VECTOR(0 to 2);
signal  SYNTHESIZED_WIRE_25 :  STD_LOGIC_VECTOR(0 to 3);
signal  SYNTHESIZED_WIRE_10 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_26 :  STD_LOGIC_VECTOR(0 to 3);
signal  SYNTHESIZED_WIRE_13 :  STD_LOGIC_VECTOR(0 to 3);
signal  SYNTHESIZED_WIRE_27 :  STD_LOGIC;
signal  SYNTHESIZED_WIRE_28 :  STD_LOGIC_VECTOR(0 to 2);
signal  SYNTHESIZED_WIRE_20 :  STD_LOGIC_VECTOR(0 to 3);


BEGIN


b2v_inst : dec_out
PORT MAP(SEC => SYNTHESIZED_WIRE_0,
                DED => SYNTHESIZED_WIRE_1,
                P1 => SYNTHESIZED_WIRE_2,
                P2 => SYNTHESIZED_WIRE_3,
                P3 => SYNTHESIZED_WIRE_4,
                P4 => SYNTHESIZED_WIRE_5,
                Ni => SYNTHESIZED_WIRE_6,
                EC => EC,
                ED => ED,
                No => Xo);

b2v_inst1 : decodall
PORT MAP(Ni => XI,
                Ov => SYNTHESIZED_WIRE_7,
                CP => SYNTHESIZED_WIRE_24,
                P => SYNTHESIZED_WIRE_20,
                x => SYNTHESIZED_WIRE_25);

b2v_inst2 : deover1
PORT MAP(so => SYNTHESIZED_WIRE_7,
                CP => SYNTHESIZED_WIRE_24,
                x => SYNTHESIZED_WIRE_25,
                sov => SYNTHESIZED_WIRE_27);

b2v_inst3 : dec
PORT MAP(d => SYNTHESIZED_WIRE_10,
                cx => SYNTHESIZED_WIRE_25,
                sp => SYNTHESIZED_WIRE_26,
                sx => SYNTHESIZED_WIRE_13,
                f => SYNTHESIZED_WIRE_6);

b2v_inst4 : dec_h
PORT MAP(s => SYNTHESIZED_WIRE_27,
                scp => SYNTHESIZED_WIRE_28,
                sx => SYNTHESIZED_WIRE_13);

b2v_inst6 : dec_o
PORT MAP(Sov => SYNTHESIZED_WIRE_27,
                SCP => SYNTHESIZED_WIRE_28,
                SP => SYNTHESIZED_WIRE_26,
                SEC => SYNTHESIZED_WIRE_0,
                DED => SYNTHESIZED_WIRE_1,
                P1 => SYNTHESIZED_WIRE_2,
```

```
                        P2 => SYNTHESIZED_WIRE_3,
                        P3 => SYNTHESIZED_WIRE_4,
                        P4 => SYNTHESIZED_WIRE_5);

b2v_inst7 : decoder
PORT MAP(CP => SYNTHESIZED_WIRE_24,
                        P => SYNTHESIZED_WIRE_20,
                        x => SYNTHESIZED_WIRE_25,
                        SCP => SYNTHESIZED_WIRE_28,
                        SP => SYNTHESIZED_WIRE_26);

b2v_inst8 : synd_d
PORT MAP(S => SYNTHESIZED_WIRE_27,
                        SCP => SYNTHESIZED_WIRE_28,
                        d => SYNTHESIZED_WIRE_10);

END;
--------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity dec is
          port (
          sx : in std_logic_vector (0 to 3);
          cx : in std_logic_vector (0 to 3);
          d : in std_logic;
          sp : in std_logic_vector (0 to 3);
          f: out std_logic_vector (0 to 3)
                    );
end dec;

architecture dec of dec is

begin
f(0) <=((sx(0) XOR cx(0))  XOR (d AND sp(0)));
f(1) <=(sx(1)  XOR cx(1))  XOR (d AND sp(1));
f(2) <=(sx(2)  XOR cx(2))  XOR (d AND sp(2));
f(3) <=(sx(3)  XOR cX(3))  XOR (d AND sp(3));

end dec;

library ieee;
use ieee.std_logic_1164.all;

entity dec_h is
port (    s:  in std_logic;
         scp:in std_logic_vector (0 to 2);
        -- x:  in std_logic_vector (0 to 31);
         sx: out std_logic_vector (0 to 3)
);
end dec_h;

architecture dec_h of dec_h is
begin

sx(0) <= '1' when (s='1' AND scp(0)='1' AND scp(1)='1' and scp(2)='0') else '0';
sx(1) <= '1' when (s='1' AND scp(0)='1' AND scp(1)='0' and scp(2)='1') else '0';
sx(2) <= '1' when (s='1' AND scp(0)='0' AND scp(1)='1' and scp(2)='1') else '0';
sx(3) <= '1' when (s='1' AND scp(0)='1' AND scp(1)='1' and scp(2)='1') else '0';

end dec_h;
--------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity dec_o is
port(
Sov: in std_logic;
SCP: in std_logic_vector(0 to 2);
SP: in std_logic_vector(1 to 4);
SEC: out std_logic;
DED: out std_logic;
P1,P2,P3,P4: out std_logic

);
end  dec_o;
```

```
architecture dec_o of dec_o is

begin

P1<=SP(1);
P2<=SP(2);
P3<=SP(3);
P4<=SP(4);

u: process(Sov,SCP,SP)
begin

if SCP/="000"  and sov='1' and (sp="0001" or sp="0010" or sp="0100" or sp="1000") then--erro x 1e
              SEC<='1';
              DED<='0';

elsif SCP/="000"  and sov='0' and (sp="0011" or sp="0110" or sp="1100" or sp="1001") then--erro x 2e
              SEC<='0';
              DED<='1';
elsif SCP/="000"  and sov='1' and (sp="0111" or sp="1011" or sp="1101" or sp="1110") then--erro x 2e
              SEC<='0';
              DED<='1';

elsif  SCP/="000"  and sp="0000" then-- erro scp
              SEC<='1';
              DED<='0';
elsif  SCP="000"  and sp="0000" and sov='1' then-- erro ov
              SEC<='1';
              DED<='0';
elsif  SCP="000"  and sp/="0000" and sov='0' then-- erro cp
              SEC<='1';
              DED<='0';
elsif  SCP="000"  and sp/="0000" and sov='1' then-- erro cp
              SEC<='1';
              DED<='0';

elsif  SCP/="000"  and sp/="0000" and sov='0' then-- 2 erros no x
              SEC<='1';
              DED<='0';
else
   SEC<='0';
              DED<='0';
end if;

end process u;
end dec_o;
----------------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity dec_out is
port(
Ni: in std_logic_vector(0 to 3);

SEC: IN std_logic;
DED: IN std_logic;
P1,P2,P3,P4: IN std_logic;
EC: OUT std_logic;
ED: OUT std_logic;
No: out std_logic_vector(0 to 3)
);
end  dec_out;

architecture dec_out of dec_out is
signal P: std_logic;

begin

No<=Ni;
P<=P1 or P2 or P3 or P4;

process(SEC,DED,P,DED)
begin
              if SEC='0' AND DED='0' then
```

```
                                EC<='0';
                                ED<='0';
        elsif SEC='1' OR DED='1' then
                        IF SEC='1' AND DED='0' THEN
                                EC<='1';
                                ED<='0';
                        ELSe
                                IF P1='1' AND P2= '1' AND P3='0' AND P4='0' THEN
                                        EC<='1';
                                        ED<='0';
                                ELSIF P1='1' AND P2= '0' AND P3='1' AND P4='0' THEN
                                        EC<='1';
                                        ED<='0';
                                ELSIF P1='1' AND P2= '0' AND P3='0' AND P4='1' THEN
                                        EC<='1';
                                        ED<='0';
                                ELSIF P1='0' AND P2= '1' AND P3='1' AND P4='0' THEN
                                        EC<='1';
                                        ED<='0';
                                ELSIF P1='0' AND P2= '1' AND P3='0' AND P4='1' THEN
                                        EC<='1';
                                        ED<='0';
                                ELSIF P1='0' AND P2= '0' AND P3='1' AND P4='1' THEN
                                        EC<='1';
                                        ED<='0';
                                else
                                        EC<='0';
                                        ED<='1';
                                END IF;
                        END IF;
        end if;

end process;

end dec_out;
-------------------------------------------------------------------------------------------------------------------------------
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decodall is
port(
x: out std_logic_vector(0 to 3);
CP: out std_logic_vector(0 to 2);
Ov: out std_logic;
P : out std_logic_vector(0 to 3);
Ni: in std_logic_vector(0 to 11)
);
end  decodall;

architecture decod of decodall is
begin

x<=Ni(0 to 3);
CP<=Ni(4 to 6);
Ov<=Ni(7);
P<=Ni(8 to 11);

end decod;
-------------------------------------------------------------------------------------------------------------------------------
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DECODER is
        port (x: in std_logic_vector(0 to 3);
            P: in std_logic_vector(0 to 3);
           CP: in std_logic_vector(0 to 2);
          SCP: out std_logic_vector(0 to 2);
           SP: out std_logic_vector(0 to 3) );
end DECODER;

architecture DECODER of DECODER is
begin
```

```
SCP(0)<= X(0) XOR X(1) XOR X(3) XOR CP(0);
SCP(1)<= X(0) XOR X(2) XOR X(3) XOR CP(1);
SCP(2)<= X(1) XOR X(2) XOR X(3) XOR CP(2);

SP(0) <=X(0) xor P(0);
SP(1) <=X(1) xor P(1);
SP(2) <=X(2) xor P(2);
SP(3) <=X(3) xor P(3);

end DECODER;
--------------------------------------------------------------------------------------------------------------------------------
use work.all;
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DeOver1 is
port(
x: in std_logic_vector(0 to 3);
CP: in std_logic_vector(0 to 2);
so: in std_logic;
sov: out std_logic
);
end  DeOver1;

architecture DeOver1 of DeOver1 is
begin

sov<= so XOR   CP(0) XOR CP(1) XOR CP(2) XOR X(0) XOR X(1) XOR X(2) XOR X(3) ;

end DeOver1;
--------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SYND_d is
        port (
        S : in std_logic;
        SCP : in std_logic_vector (0 to 2);
        d: out std_logic
        );
end SYND_d;

architecture SYND_d of SYND_d is
begin

D<=(NOT S) AND (SCP(0) OR SCP(1) OR SCP(2) );

end SYND_d;
--------------------------------------------------------------------------------------------------------------------------------
```

ANEXO A4 – Código VHDL de um Núcleo IP Corretor de Erro *Reed-Muller*

CODIFICADOR

```
---------------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity RMe_8_4 is
port(
N: out std_logic_vector(1 to 8);                    -- codeword (n= 8 bits)
K: in std_logic_vector(1 to 4)                      -- vetor mensagem (K= 4 bits)
);
end  RMe_8_4;


architecture ENCODER of RMe_8_4 is

begin

N(1)<=K(1) xor K(2) xor K(3) xor K(4);
N(2)<=K(2) xor K(3) xor K(4);
N(3)<=K(1) xor K(3) xor K(4);
N(4)<=K(3) xor K(4);
N(5)<=K(1) xor K(2) xor K(4);
N(6)<=K(2) xor K(4);
N(7)<=K(1) xor K(4);
N(8)<=K(4);

end ENCODER;
---------------------------------------------------------------------------------------------------------------------------------------
```

DECODIFICADOR

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY decodificador IS
        port
        (
                N :  IN  STD_LOGIC_VECTOR(7 downto 0);
                DED :  OUT  STD_LOGIC;
                SEC :  OUT  STD_LOGIC;
                K :  OUT  STD_LOGIC_VECTOR(3 downto 0)
        );
END decodificador;

ARCHITECTURE bdf_type OF decodificador IS

component decin
        PORT(N : IN STD_LOGIC_VECTOR(7 downto 0);
                X1 : OUT STD_LOGIC_VECTOR(3 downto 0);
                X2 : OUT STD_LOGIC_VECTOR(3 downto 0);
                X3 : OUT STD_LOGIC_VECTOR(3 downto 0)
        );
end component;

component lmd_x3
        PORT(X3 : IN STD_LOGIC_VECTOR(3 downto 0);
                X_3 : OUT STD_LOGIC;
                X_3i : OUT STD_LOGIC
        );
end component;

component lmd_x2
        PORT(X2 : IN STD_LOGIC_VECTOR(3 downto 0);
                X_2 : OUT STD_LOGIC;
                X_2i : OUT STD_LOGIC
        );
end component;

component lmd_x1
        PORT(X1 : IN STD_LOGIC_VECTOR(3 downto 0);
                X_1 : OUT STD_LOGIC;
                X_1i : OUT STD_LOGIC
        );
end component;
```

```
component sum
	PORT(X_1 : IN STD_LOGIC;
		X_2 : IN STD_LOGIC;
		X_3 : IN STD_LOGIC;
		N : IN STD_LOGIC_VECTOR(7 downto 0);
		Ms0 : OUT STD_LOGIC;
		Ms1 : OUT STD_LOGIC;
		Ms2 : OUT STD_LOGIC;
		Ms3 : OUT STD_LOGIC;
		Ms4 : OUT STD_LOGIC;
		Ms5 : OUT STD_LOGIC;
		Ms6 : OUT STD_LOGIC;
		Ms7 : OUT STD_LOGIC;
		K : OUT STD_LOGIC_VECTOR(2 downto 0);
		Msi : OUT STD_LOGIC_VECTOR(7 downto 0)
	);
end component;

component somador
	PORT(P0 : IN STD_LOGIC;
		P1 : IN STD_LOGIC;
		P2 : IN STD_LOGIC;
		P3 : IN STD_LOGIC;
		P4 : IN STD_LOGIC;
		P5 : IN STD_LOGIC;
		P6 : IN STD_LOGIC;
		P7 : IN STD_LOGIC;
		S : OUT STD_LOGIC
	);
end component;

component converte
	PORT(si3 : IN STD_LOGIC;
		X3i : IN STD_LOGIC;
		X2i : IN STD_LOGIC;
		X1i : IN STD_LOGIC;
		MSi : IN STD_LOGIC_VECTOR(7 downto 0);
		Si : IN STD_LOGIC_VECTOR(2 downto 0);
		DED : OUT STD_LOGIC;
		SEC : OUT STD_LOGIC;
		K : OUT STD_LOGIC_VECTOR(3 downto 0)
	);
end component;

signal	SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(3 downto 0);
signal	SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(3 downto 0);
signal	SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(3 downto 0);
signal	SYNTHESIZED_WIRE_3 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_4 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_5 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_6 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_7 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_8 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_9 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_10 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_11 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_12 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_13 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_14 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_15 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_16 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_17 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_18 :  STD_LOGIC_VECTOR(7 downto 0);
signal	SYNTHESIZED_WIRE_19 :  STD_LOGIC_VECTOR(2 downto 0);

BEGIN

b2v_inst : decin
PORT MAP(N => N,
		X1 => SYNTHESIZED_WIRE_2,
		X2 => SYNTHESIZED_WIRE_1,
		X3 => SYNTHESIZED_WIRE_0);

b2v_inst1 : lmd_x3
PORT MAP(X3 => SYNTHESIZED_WIRE_0,
		X_3 => SYNTHESIZED_WIRE_5,
```

```
                        X_3i => SYNTHESIZED_WIRE_15);

b2v_inst2 : lmd_x2
PORT MAP(X2 => SYNTHESIZED_WIRE_1,
                        X_2 => SYNTHESIZED_WIRE_4,
                        X_2i => SYNTHESIZED_WIRE_16);

b2v_inst3 : lmd_x1
PORT MAP(X1 => SYNTHESIZED_WIRE_2,
                        X_1 => SYNTHESIZED_WIRE_3,
                        X_1i => SYNTHESIZED_WIRE_17);

b2v_inst4 : sum
PORT MAP(X_1 => SYNTHESIZED_WIRE_3,
                        X_2 => SYNTHESIZED_WIRE_4,
                        X_3 => SYNTHESIZED_WIRE_5,
                        N => N,
                        Ms0 => SYNTHESIZED_WIRE_6,
                        Ms1 => SYNTHESIZED_WIRE_7,
                        Ms2 => SYNTHESIZED_WIRE_8,
                        Ms3 => SYNTHESIZED_WIRE_9,
                        Ms4 => SYNTHESIZED_WIRE_10,
                        Ms5 => SYNTHESIZED_WIRE_11,
                        Ms6 => SYNTHESIZED_WIRE_12,
                        Ms7 => SYNTHESIZED_WIRE_13,
                        K => SYNTHESIZED_WIRE_19,
                        Msi => SYNTHESIZED_WIRE_18);

b2v_inst5 : somador
PORT MAP(P0 => SYNTHESIZED_WIRE_6,
                        P1 => SYNTHESIZED_WIRE_7,
                        P2 => SYNTHESIZED_WIRE_8,
                        P3 => SYNTHESIZED_WIRE_9,
                        P4 => SYNTHESIZED_WIRE_10,
                        P5 => SYNTHESIZED_WIRE_11,
                        P6 => SYNTHESIZED_WIRE_12,
                        P7 => SYNTHESIZED_WIRE_13,
                        S => SYNTHESIZED_WIRE_14);

b2v_inst6 : converte
PORT MAP(si3 => SYNTHESIZED_WIRE_14,
                        X3i => SYNTHESIZED_WIRE_15,
                        X2i => SYNTHESIZED_WIRE_16,
                        X1i => SYNTHESIZED_WIRE_17,
                        MSi => SYNTHESIZED_WIRE_18,
                        Si => SYNTHESIZED_WIRE_19,
                        DED => DED,
                        SEC => SEC,
                        K => K);

END;
--------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity converte is
port(
MSi: in std_logic_vector(7 downto 0);
Si: in std_logic_vector(2 downto 0);                    -- codeword (n= 8 bits)
si3: in std_logic;
K: out std_logic_vector(3 downto 0);
DED:out std_logic;
SEC:out std_logic;
X3i:in std_logic;
X2i:in  std_logic;
X1i:in  std_logic
--s0,s1,s2,s3: out std_logic                            -- vetor mensagem (K= 4 bits)
);
end  converte;

architecture conv of converte is

begin
--ded<=X3i or X2i or X1i;

k(0)<=Si(0);
```

```
k(1)<=Si(1);
k(2)<=Si(2);
k(3)<=Si3;
-------------------------------------------------
secded: process(MSi,X3i,X2i,X1i,si3,si)
begin
                        if Msi="00000000" and si3='0' then
                                SEC<='0';
                                DED<=X3i or X2i or X1i;
                        elsif Si3='0'and (MSi="10000000" or MSi="01000000" or MSi="00100000" or MSi="00010000" or
MSi="00001000" or MSi="00000100" or MSi="00000010" or MSi="00000001") then
                                SEC<='1';
                                DED<='0';
                        elsif Msi="11111111" and Si3='1' then
                                SEC<='0';
                                DED<=X3i or X2i or X1i;
                        elsif Si3='1' and (MSi="01111111" or MSi="10111111" or MSi="11011111" or MSi="11101111" or
MSi="11110111" or MSi="11111011" or MSi="11111101" or MSi="11111110") then
                                SEC<='1';
                                DED<='0';
                        else
                                SEC<='0';
                                ded<=X3i or X2i or X1i;
                        end if;

end process secded;
end conv;
-------------------------------------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
entity decin is
 port(
N: in std_logic_vector(7 downto 0);                             -- codeword (n= 8 bits)
X3:out std_logic_vector(3 downto 0);
X2:out std_logic_vector(3 downto 0);
X1:out std_logic_vector(3 downto 0)

);
end  decin;
architecture entrada of decin is
begin

X3(0)<=N(7)xor N(6);
X3(1)<=N(2)xor N(3);
X3(2)<=N(4)xor N(5);
X3(3)<=N(0)xor N(1);

X2(0)<=N(7)xor N(5);
X2(1)<=N(3)xor N(1);
X2(2)<=N(6)xor N(4);
X2(3)<=N(2)xor N(0);

X1(0)<=N(7)xor N(3);
X1(1)<=N(6)xor N(2);
X1(2)<=N(5)xor N(1);
X1(3)<=N(4)xor N(0);
end entrada;
-------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
entity LMD_X1 is
 port(
X1:  in std_logic_vector(3 downto 0);
X_1:  out std_logic;                    -- se for '1' numero de '1'> num. de '0', se for '0' '0'>'1'
X_1i: out std_logic                     -- se for '1' => numero de '0' igual ao numero de '1'
);
end  LMD_X1;

architecture LMDX1 of LMD_X1 is
BEGIN

LMJ_X1: process(X1) -- calculo do coeficiente X1
begin
if X1="0000" then
```

```
                X_1<='0';
                X_1i<='0';
elsif X1="0001" then
                X_1<='0';
                X_1i<='0';
elsif X1="0010" then
                X_1<='0';
                X_1i<='0';
elsif X1="0011" then
                X_1<='0';
                X_1i<='1';
elsif X1="0100" then
                X_1<='0';
                X_1i<='0';
elsif X1="0101" then
                X_1<='0';
                X_1i<='1';
elsif X1="0110" then
                X_1<='0';
                X_1i<='1';
elsif X1="0111" then
                X_1<='1';
                X_1i<='0';
elsif X1="1000" then
                X_1<='0';
                X_1i<='0';
elsif X1="1001" then
                X_1<='0';
                X_1i<='1';
elsif X1="1010" then
                X_1<='0';
                X_1i<='1';
elsif X1="1011" then
                X_1<='1';
                X_1i<='0';
elsif X1="1100" then
                X_1<='0';
                X_1i<='1';
elsif X1="1101" then
                X_1<='1';
                X_1i<='0';
elsif X1="1110" then
                X_1<='1';
                X_1i<='0';
elsif X1="1111" then
                X_1<='1';
                X_1i<='0';
end if;
end process LMJ_X1;

END LMDX1;
-------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
entity LMD_X2 is
 port(
X2:  in std_logic_vector(3 downto 0);
X_2:  out std_logic                          -- se for '1' numero de '1'> num. de '0', se for '0' '0'>'1'
X_2i: out std_logic                          -- se for '1' => numero de '0' igual ao numero de '1'
);
end  LMD_X2;

architecture LMDX2 of LMD_X2 is
BEGIN

LMJ_X2: process(X2) --  calculo do coeficiente X2
begin
if X2="0000" then
                X_2<='0';
                X_2i<='0';
elsif X2="0001" then
                X_2<='0';
                X_2i<='0';
elsif X2="0010" then
                X_2<='0';
```

```
                X_2i<='0';
elsif X2="0011" then
                X_2<='0';
                X_2i<='1';
elsif X2="0100" then
                X_2<='0';
                X_2i<='0';
elsif X2="0101" then
                X_2<='0';
                X_2i<='1';
elsif X2="0110" then
                X_2<='0';
                X_2i<='1';
elsif X2="0111" then
                X_2<='1';
                X_2i<='0';
elsif X2="1000" then
                X_2<='0';
                X_2i<='0';
elsif X2="1001" then
                X_2<='0';
                X_2i<='1';
elsif X2="1010" then
                X_2<='0';
                X_2i<='1';
elsif X2="1011" then
                X_2<='1';
                X_2i<='0';
elsif X2="1100" then
                X_2<='0';
                X_2i<='1';
elsif X2="1101" then
                X_2<='1';
                X_2i<='0';
elsif X2="1110" then
                X_2<='1';
                X_2i<='0';
elsif X2="1111" then
                X_2<='1';
                X_2i<='0';
end if;
end process LMJ_X2;
end LMDX2;
-------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
entity LMD_X3 is
 port(
X3:  in std_logic_vector(3 downto 0);
X_3:  out std_logic                          -- se for '1' numero de '1'> num. de '0', se for '0' '0'>'1'
X_3i: out std_logic                          -- se for '1' => numero de '0' igual ao numero de '1'
);
end  LMD_X3;

architecture LMDX3 of LMD_X3 is

begin

LMJ_X3: process(X3) -- calculo do coeficiente X3
begin
if X3="0000" then
                X_3<='0';
                X_3i<='0';
elsif X3="0001" then
                X_3<='0';
                X_3i<='0';
elsif X3="0010" then
                X_3<='0';
                X_3i<='0';
elsif X3="0011" then
                X_3<='0';
                X_3i<='1';
elsif X3="0100" then
                X_3<='0';
                X_3i<='0';
```

```
elsif X3="0101" then
		X_3<='0';
		X_3i<='1';
elsif X3="0110" then
		X_3<='0';
		X_3i<='1';
elsif X3="0111" then
		X_3<='1';
		X_3i<='0';
elsif X3="1000" then
		X_3<='0';
		X_3i<='0';
elsif X3="1001" then
		X_3<='0';
		X_3i<='1';
elsif X3="1010" then
		X_3<='0';
		X_3i<='1';
elsif X3="1011" then
		X_3<='1';
		X_3i<='0';
elsif X3="1100" then
		X_3<='0';
		X_3i<='1';
elsif X3="1101" then
		X_3<='1';
		X_3i<='0';
elsif X3="1110" then
		X_3<='1';
		X_3i<='0';
elsif X3="1111" then
		X_3<='1';
		X_3i<='0';
end if;
end process LMJ_X3;

end LMDX3;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY somador IS
		port
		(
			P0 :  IN  STD_LOGIC;
			P1 :  IN  STD_LOGIC;
			P2 :  IN  STD_LOGIC;
			P3 :  IN  STD_LOGIC;
			P4 :  IN  STD_LOGIC;
			P5 :  IN  STD_LOGIC;
			P6 :  IN  STD_LOGIC;
			P7 :  IN  STD_LOGIC;
			S :  OUT  STD_LOGIC
		);
END somador;

ARCHITECTURE bdf_type OF somador IS

component somador4bits
		PORT(c0 : IN STD_LOGIC;
			a0 : IN STD_LOGIC;
			a1 : IN STD_LOGIC;
			a2 : IN STD_LOGIC;
			a3 : IN STD_LOGIC;
			b0 : IN STD_LOGIC;
			b1 : IN STD_LOGIC;
			b2 : IN STD_LOGIC;
			b3 : IN STD_LOGIC;
			s0 : OUT STD_LOGIC;
			s1 : OUT STD_LOGIC;
			s2 : OUT STD_LOGIC;
			s3 : OUT STD_LOGIC;
			c4 : OUT STD_LOGIC
		);
```

```
end component;

component comparador
	PORT(S0 : IN STD_LOGIC;
		s1 : IN STD_LOGIC;
		s2 : IN STD_LOGIC;
		s3 : IN STD_LOGIC;
		Saida : OUT STD_LOGIC
	);
end component;

signal	SYNTHESIZED_WIRE_68 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_8 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_12 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_13 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_14 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_15 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_16 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_17 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_18 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_19 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_20 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_24 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_25 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_26 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_27 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_28 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_32 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_33 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_34 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_35 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_36 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_40 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_41 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_42 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_43 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_44 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_48 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_49 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_50 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_51 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_52 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_56 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_57 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_58 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_59 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_60 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_64 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_65 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_66 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_67 :  STD_LOGIC;

BEGIN
SYNTHESIZED_WIRE_68 <= '0';

b2v_inst : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_68,
		a0 => P0,
		a1 => SYNTHESIZED_WIRE_68,
		a2 => SYNTHESIZED_WIRE_68,
		a3 => SYNTHESIZED_WIRE_68,
		b0 => SYNTHESIZED_WIRE_68,
		b1 => SYNTHESIZED_WIRE_68,
		b2 => SYNTHESIZED_WIRE_68,
		b3 => SYNTHESIZED_WIRE_68,
		s0 => SYNTHESIZED_WIRE_24,
		s1 => SYNTHESIZED_WIRE_25,
		s2 => SYNTHESIZED_WIRE_26,
		s3 => SYNTHESIZED_WIRE_27,
		c4 => SYNTHESIZED_WIRE_20);

b2v_inst10 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_8,
		a0 => P7,
		a1 => SYNTHESIZED_WIRE_68,
		a2 => SYNTHESIZED_WIRE_68,
```

```
                a3 => SYNTHESIZED_WIRE_68,
                b0 => SYNTHESIZED_WIRE_12,
                b1 => SYNTHESIZED_WIRE_13,
                b2 => SYNTHESIZED_WIRE_14,
                b3 => SYNTHESIZED_WIRE_15,
                s0 => SYNTHESIZED_WIRE_16,
                s1 => SYNTHESIZED_WIRE_17,
                s2 => SYNTHESIZED_WIRE_18,
                s3 => SYNTHESIZED_WIRE_19);

b2v_inst2 : comparador
PORT MAP(S0 => SYNTHESIZED_WIRE_16,
                s1 => SYNTHESIZED_WIRE_17,
                s2 => SYNTHESIZED_WIRE_18,
                s3 => SYNTHESIZED_WIRE_19,
                Saida => S);

b2v_inst4 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_20,
                a0 => P1,
                a1 => SYNTHESIZED_WIRE_68,
                a2 => SYNTHESIZED_WIRE_68,
                a3 => SYNTHESIZED_WIRE_68,
                b0 => SYNTHESIZED_WIRE_24,
                b1 => SYNTHESIZED_WIRE_25,
                b2 => SYNTHESIZED_WIRE_26,
                b3 => SYNTHESIZED_WIRE_27,
                s0 => SYNTHESIZED_WIRE_32,
                s1 => SYNTHESIZED_WIRE_33,
                s2 => SYNTHESIZED_WIRE_34,
                s3 => SYNTHESIZED_WIRE_35,
                c4 => SYNTHESIZED_WIRE_28);

b2v_inst5 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_28,
                a0 => P2,
                a1 => SYNTHESIZED_WIRE_68,
                a2 => SYNTHESIZED_WIRE_68,
                a3 => SYNTHESIZED_WIRE_68,
                b0 => SYNTHESIZED_WIRE_32,
                b1 => SYNTHESIZED_WIRE_33,
                b2 => SYNTHESIZED_WIRE_34,
                b3 => SYNTHESIZED_WIRE_35,
                s0 => SYNTHESIZED_WIRE_40,
                s1 => SYNTHESIZED_WIRE_41,
                s2 => SYNTHESIZED_WIRE_42,
                s3 => SYNTHESIZED_WIRE_43,
                c4 => SYNTHESIZED_WIRE_36);

b2v_inst6 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_36,
                a0 => P3,
                a1 => SYNTHESIZED_WIRE_68,
                a2 => SYNTHESIZED_WIRE_68,
                a3 => SYNTHESIZED_WIRE_68,
                b0 => SYNTHESIZED_WIRE_40,
                b1 => SYNTHESIZED_WIRE_41,
                b2 => SYNTHESIZED_WIRE_42,
                b3 => SYNTHESIZED_WIRE_43,
                s0 => SYNTHESIZED_WIRE_48,
                s1 => SYNTHESIZED_WIRE_49,
                s2 => SYNTHESIZED_WIRE_50,
                s3 => SYNTHESIZED_WIRE_51,
                c4 => SYNTHESIZED_WIRE_44);

b2v_inst7 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_44,
                a0 => P4,
                a1 => SYNTHESIZED_WIRE_68,
                a2 => SYNTHESIZED_WIRE_68,
                a3 => SYNTHESIZED_WIRE_68,
                b0 => SYNTHESIZED_WIRE_48,
                b1 => SYNTHESIZED_WIRE_49,
                b2 => SYNTHESIZED_WIRE_50,
                b3 => SYNTHESIZED_WIRE_51,
                s0 => SYNTHESIZED_WIRE_56,
```

```
                        s1 => SYNTHESIZED_WIRE_57,
                        s2 => SYNTHESIZED_WIRE_58,
                        s3 => SYNTHESIZED_WIRE_59,
                        c4 => SYNTHESIZED_WIRE_52);

b2v_inst8 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_52,
                        a0 => P5,
                        a1 => SYNTHESIZED_WIRE_68,
                        a2 => SYNTHESIZED_WIRE_68,
                        a3 => SYNTHESIZED_WIRE_68,
                        b0 => SYNTHESIZED_WIRE_56,
                        b1 => SYNTHESIZED_WIRE_57,
                        b2 => SYNTHESIZED_WIRE_58,
                        b3 => SYNTHESIZED_WIRE_59,
                        s0 => SYNTHESIZED_WIRE_64,
                        s1 => SYNTHESIZED_WIRE_65,
                        s2 => SYNTHESIZED_WIRE_66,
                        s3 => SYNTHESIZED_WIRE_67,
                        c4 => SYNTHESIZED_WIRE_60);

b2v_inst9 : somador4bits
PORT MAP(c0 => SYNTHESIZED_WIRE_60,
                        a0 => P6,
                        a1 => SYNTHESIZED_WIRE_68,
                        a2 => SYNTHESIZED_WIRE_68,
                        a3 => SYNTHESIZED_WIRE_68,
                        b0 => SYNTHESIZED_WIRE_64,
                        b1 => SYNTHESIZED_WIRE_65,
                        b2 => SYNTHESIZED_WIRE_66,
                        b3 => SYNTHESIZED_WIRE_67,
                        s0 => SYNTHESIZED_WIRE_12,
                        s1 => SYNTHESIZED_WIRE_13,
                        s2 => SYNTHESIZED_WIRE_14,
                        s3 => SYNTHESIZED_WIRE_15,
                        c4 => SYNTHESIZED_WIRE_8);

END;
-------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
use IEEE.std_logic_arith.all;

entity comparador is
port(

S0,s1,s2,s3: in std_logic;
Saida: out std_logic
);
end  comparador;

architecture comp of comparador is
signal si: std_logic_vector(3 downto 0);
begin

si(0)<=s0;
si(1)<=s1;
si(2)<=s2;
si(3)<=s3;

Logica_comparacao: process(S0,S1,S2,S3)
begin
        if Si >= "0100" then
                Saida<='1';
        else
                Saida<='0';
        end if;
end process logica_comparacao;
end comp;
-------------------------------------------------------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY somador4bits IS
PORT (
```

```
c0 :                    IN STD_LOGIC;
a0,a1,a2,a3, b0,b1,b2,b3 :              IN STD_LOGIC;
s0,s1,s2,s3 :                   OUT STD_LOGIC;
c4 :                    OUT STD_LOGIC);
END somador4bits;
ARCHITECTURE comportamento OF somador4bits IS
SIGNAL c : STD_LOGIC_VECTOR (3 DOWNTO 1);
BEGIN
s0 <= (a0 XOR b0 XOR c0);
c(1) <= (a0 AND b0) OR (a0 AND c0) OR (b0 AND c0);
s1 <= a1 XOR b1 XOR c(1);
c(2) <= (a1 AND b1) OR (a1 AND c(1)) OR (b1 AND c(1));
s2 <= a2 XOR b2 XOR c(2);
c(3) <= (a2 AND b2) OR (a2 AND c(2)) OR (b2 AND c(2));
s3 <= a3 XOR b3 XOR c(3);
c4 <= (a3 AND b3) OR (a3 AND c(3)) OR (b3 AND c(3));
END comportamento;
--------------------------------------------------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
entity sum is
 port(
N: in std_logic_vector(7 downto 0);                     -- codeword (n= 8 bits)
X_1, X_2, X_3: in std_logic;
K: out std_logic_vector(2 downto 0);
Msi: out std_logic_vector(7 downto 0);
Ms0,Ms1,Ms2,Ms3,Ms4,Ms5,Ms6,Ms7:out std_logic                  -- My xor N
);
end  sum;

architecture soma of sum is
signal My:std_logic_vector(7 downto 0);
signal Msun:std_logic_vector(7 downto 0);-- My xor N

begin

My(7)<=X_3 xor X_2 xor X_1;
My(6)<=X_2 xor X_1;
My(5)<=X_3 xor X_1;
My(4)<=X_1;
My(3)<=X_3 xor X_2;
My(2)<=X_2 ;
My(1)<=X_3 ;
My(0)<='0';

Msun<= My xor N;

Ms0<=Msun(0);
Ms1<=Msun(1);
Ms2<=Msun(2);
Ms3<=Msun(3);
Ms4<=Msun(4);
Ms5<=Msun(5);
Ms6<=Msun(6);
Ms7<=Msun(7);

Msi(0)<=Msun(0);
Msi(1)<=Msun(1);
Msi(2)<=Msun(2);
Msi(3)<=Msun(3);
Msi(4)<=Msun(4);
Msi(5)<=Msun(5);
Msi(6)<=Msun(6);
Msi(7)<=Msun(7);


k(0)<=x_3;
k(1)<=x_2;
k(2)<=x_1;

end soma;
```