UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GEFERSON LUIS HESS JÚNIOR

Implementação e caracterização de falhas em um decodificador LDPC

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Gabriel Luca Nazar

Porto Alegre
dezembro de 2016

AGRADECIMENTOS

Ao meus pais e irmão por todo apoio, carinho e investimento. Aos meus primos e tios por me aturarem durante esses anos.

Ao meu orientador Gabriel Luca Nazar pela oportunidade e incontável ajuda na elaboração e implementação desse trabalho. Aos meus colegas do laboratório LSE, especialmente ao Marcos, pela imensa ajuda com os testes de injeção de falhas.

A Júlia Reuwsaat por ter me apoiado durante toda essa etapa. Não fazes ideia do quão fundamental e importante foi desde que te conheci.

RESUMO

Os códigos LDPC (Low-Density Parity Check) são muito utilizados pela indústria e foram extensivamente estudados pela comunidade acadêmica. Inúmeros algoritmos, políticas de escalonamento e arquiteturas foram propostas para implementar esses códigos em FPGAs, mas sem preocupação com falhas que podem ocorrer na memória de configuração desses dispositivos. Esse trabalho apresenta um estudo sobre os códigos LDPC e alguns de seus algoritmos, como: *Sum-Product*, *Min-Sum*, λ*-min* e *Modified Min-Sum* (MMS).
Foi implementado um decodificador em hardware de códigos LDPC utilizando a política de escalonamento *Layered Belief Propagation* e o algoritmo MMS. Ambos foram escolhidos por se adequarem melhor as características do LDPC a ser implementando, como: utilização de uma matriz de paridade do tipo *Quasi-cyclic*, uma pequena ocupação de área em hardware e uma eficiência energética, em relação ao canal de comunicação, dentro dos padrões esperados pela literatura.
Por fim, o trabalho demostra os resultados do teste de injeção de falhas que foi realizado no módulo *Check-Node*. Esse módulo foi escolhido por ser o principal do LDPC, pois trabalha diretamente com todas as operações aritméticas descritas no algoritmo e ocupa a maior parte da área do decodificador. Os resultados demonstram a quantidade de bits sensíveis a erros, categorizados em diferentes tipos de erros, bem como o impacto desses na a eficiência energética, em relação ao canal de comunicação, do decodificador.

Palavras-chave: LDPC. Low-density parity-check. correção de erros. códigos de canal. sistemas embarcados. FPGA. injeção de falhas. Comunicação de dados.

Implementation and characterization of faults in an LDPC decoder

ABSTRACT

LDPC (Low-Density Parity Check) codes are widely used by the industry and were the subject to extensive studies by the academic community. Many algorithms, schedules and architectures have been proposed to implement these codes in FPGAs, but with no concern for faults that may occur in the configuration memory of these devices. This work presents a study about LDPC codes and some of its algorithms, like: Sum-Product, Min-Sum, λ-min and Modified Min-Sum (MMS).

A LDPC decoder was implemented in hardware using the Layered Belief Propagation schedule with the Modified Min-Sum algorithm. Both have been chosen because they adapt better to the necessary characteristics of the LDPC implemented, like: using a Quasi-cyclic parity-check matrix, a small hardware utilization and a Bit Error Rate that is consistent with the literature.

Lastly, this work show the results of the fault injection tests performed in the Check-Node. This is the main LDPC module, because it implements all the arithmetic operations described in the algorithm and occupies most of the decoder area. The results demonstrate the amount of bits that are sensitive to errors, categorized in different types, as well as the impact of these bits in the Bit Error Rate of the decoder.

Keywords: LDPC, Low-Density Parity-Check code, Forward Error Correction, Error detection and correction, Embedded Systems, Fault injection ,Data Communication, FPGA.

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuits
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
BUB	Bit Update Blocks
CN	Check-Node
FEC	Forward Error-Correction
FPGA	Field Programmable Gate Arrays
IDS	Informed Dynamic Scheduling
LBP	Layered Belief Propagation
LDPC	Low-Density Parity Check
LLR	Logarithmic-Likelihood Ratio
LUT	Lookup Table
MMS	Modified Min-Sum
PCUB	Parity Check Update Blocks
QC	Quasi-Cyclic
SEU	Single Event Upset
SNR	Signal to Noise Ratio
SRAM	Static Random Access Memory
VN	Variable Node

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e motivação

Dentre os diversos códigos de correção de erros para sistemas de comunicação de dados existentes, o *Low-Density Parity Check* (LDPC) é um dos mais utilizado e estudado, pois possui uma performance próxima ao limite de Shannon (CHUNG et al., 2001). Como o próprio nome sugere, são códigos com uma matriz de paridade que contém poucas entradas com valores diferentes de zero.

O LDPC foi proposto por Gallager em 1962 na sua tese de doutorado (GALLAGER, 1962), porém sua implementação em sistemas reais foi considerada muito complexa na época. Décadas depois, acabou sendo redescoberto (MACKAY; NEAL, 1996) e, desde então, vem sendo amplamente estudado e utilizado na indústria, pois é a codificação de canal sugerida na especificação de vários padrões, como: WiFi (IEEE, 2012), WiMAX (IEEE, 2004), CCSDS (CCSDS, 2009), ITU G.hn (OKSMAN; GALLI, 2009) e DVB-S2 (ETSI, 2014). Consiste em um *Forward Error-Correction* (FEC), ou seja, um código que permite que o receptor dos dados corrija uma certa quantidade de erros que possam ter ocorrido no meio de transmissão. Esses códigos utilizam uma matriz de paridade, que representa a conectividade entre os *Check Nodes* (CN) e *Variable Nodes* (VN), onde um CN representa uma linha e um VN uma coluna dessa matriz.

Os códigos LDPC são muito flexíveis e apresentam grande diversidade na escolha dos parâmetros que compõem o código, como: matriz de paridade, o número de elementos diferentes de zero em cada linha ou coluna (peso) e se o código é regular ou irregular. Um código LDPC é chamado de regular se todas as linhas e colunas da matriz possuírem o mesmo peso, entretanto não é necessário que o peso das linhas seja igual ao peso das colunas. Caso os valores dos pesos forem diferentes, o código é chamado irregular.

Field Programmable Gate Arrays (FPGAs) são dispositivos muito utilizados pela indústria, incluindo a local, para implementar diferentes funcionalidades utilizadas em sistemas de comunicação de dados. Apresentam vantagens úteis a essas funcionalidades, como reconfigurabilidade, alto paralelismo e perfomance. Porém, há uma grande lacuna na disponibilidade de implementações de LDPC em FPGAs, mesmo com o grande interesse na utilização desses dispositivos na implementação desses códigos ((HAILES et al., 2015)).

Em ambientes com alta incidência de radiação ionizante, como é o caso das apli-

cações aeroespaciais, há o risco de efeitos adversos serem causados em dispositivos semicondutores. Há dois tipos principais de falhas, conhecidas como permanentes e transientes, sendo que no presente trabalho estamos mais interessados nas transientes, pois essas são mais comuns em FPGAs. Um evento que gera mudança de estado lógico em um dispositivo é conhecido como *Single Event Upset* (SEU) (LABEL, 1996).

Os FPGAs possuem uma memória de configuração que é a responsável por configurar os elementos que os compõem, como LUTs (*Lookup Tables)*, flip-flops, blocos de memória e a interconexão desses elementos. Isso faz com que a ocorrência de SEUs seja muito maléfica à funcionalidade dos módulos implementados no FPGA, pois a memória de configuração de dispositivos baseados em SRAM (*Static Random Access Memory)* é muito sensível a esse tipo de evento (FULLER et al., 1999) e a inversão do valor de um bit dessa memória pode acarretar no funcionamento incorreto de todo o sistema. Porém, nem sempre que um bit dessa memória for invertido haverá uma falha no funcionamento do sistema (VIOLANTE et al., 2004), pois essa pode ocorrer em um módulo ou bit que não está envolvido no processamento daqueles dados.

As características de códigos LDPC tornam a implementação desses em FPGAs muito interessante, pois há diversos pontos a serem explorados e escolhidos pelo projetista do sistema. A utilização em aplicações comerciais que precisam de um alto desempenho e vazão é inviável nos processadores comuns atuais, sendo necessário o uso de hardware dedicado, para uma máxima utilização do paralelismo inerente aos códigos LDPC. Existem inúmeras propostas de implementação de LDPC para FPGAs (HAILES et al., 2015), cada qual variando nos parâmetros do código e/ou nas características da implementação, como: vazão de dados, eficiência energética na transmissão, requisitos de hardware, algoritmo utilizado e flexibilidade.

Entretanto, como citado anteriormente, existe um baixíssima disponibilidade e reusabilidade dessas implementações, pois os autores raramente disponibilizam o código publicamente. Esse é um dos grandes problemas para uma justa comparação de diferentes arquiteturas, como explicitado em (HAILES et al., 2015).

1.2 Objetivos

Não foram encontrados trabalhos preocupados em estudar os efeitos de falhas em decodificador LDPC implementados em FGPA, mesmo sabendo o quão frequente essas falhas podem acontecer nesses dispositivos. Há um trabalho (MAY; ALLES; WEHN,

2008) que estuda falhas transientes em implementações de LDPC em ASICs (*Application Specific Integrated Circuits*), porém esses dispositivos têm um modelo de falhas diferentes das FPGAs.

Esse trabalho apresenta a descrição de um hardware parametrizável de um decodificador LDPC, juntamente com um simulador do mesmo descrito em C, visando as características do padrão 802.11(IEEE, 2012), mas também funcional para outros padrões que seguem as mesmas características em sua matriz de paridade.

O simulador em C é funcionalmente equivalente a implementação em hardware, sendo de grande valia para o estudo de certas características da arquitetura que foi implementada, como: a definição do melhor algoritmo, melhor política de escalonamento, precisão do ponto-fixo, etc. O simulador foi bastante utilizado nesse trabalho e poderá ser muito útil para novas pesquisas na área, pois sua implementação é parametrizável e buscar ser de fácil uso.

Como mencionado, não há trabalhos preocupados com os efeitos das falhas em decodificadores LDPC implementados em FPGA e esse trabalho visa suprir essa lacuna. O estudo foi feito apenas em módulos do decodificador LDPC, pois esse é mais complexo que o codificador. Falhas foram injetadas em um módulo bastante crítico para o comportamento do decodificador e a caracterização desses resultados poderá ser utilizada em futuros trabalhos na proposta de um decodificador resiliente à falhas.

Além disso, o trabalho ainda apresenta um breve estudo sobre os principais algoritmos e políticas de escalonamento encontrados na literatura e suscetíveis a implementação em hardware. Há comparações dentre essas características e seus possíveis impactos nos resultados referentes ao hardware e ao funcionamento do decodificador LDPC implementando em FPGA.

1.3 Estrutura do trabalho

No capítulo 2 são apresentados conceitos básicos e necessários sobre o funcionamento de um sistema de comunicação de dados. No capítulo 3, são discutidas as características dos códigos LDPC e os principais algoritmos e políticas de escalonamento estudados para então, no capítulo 4, demonstrar e explicar a arquitetura implementada seguida de uma discussão sobre o algoritmo utilizado. Serão apresentados, também, resultados de desempenho e utilização de recursos do FPGA para a arquitetura implementada. No capítulo 5 será discutida a abordagem utilizada na injeção de falhas e serão apresentados os

resultados obtidos sobre a caracterização das mesmas. No capítulo 6 é feita a conclusão do trabalho.

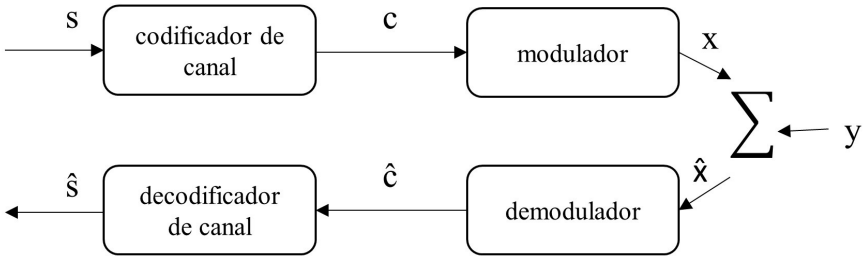
2 SISTEMA DE COMUNICAÇÃO DE DADOS

A Figura 2.1 exemplifica um sistema básico de comunicação de dados. Uma mensagem binária **s**, de K bits, é codificada pelo codificador de canal, obtendo uma nova mensagem **c**, de N bits ($N > K$). Temos que $M = N - K$ bits foram adicionados ao fim da mensagem original **s**, esses M bits são também chamados de bits de paridade. A razão do código (R) pode ser calculada da seguinte forma:

$$R = \frac{K}{N} = \frac{N - M}{N}; 0 < R < 1 \tag{2.1}$$

ou seja, para uma mensagem $\mathbf{s} = 5$ bits e uma razão $R = \frac{1}{2}$, a palavra codificada **c** terá 10 bits. Note que os bits de paridade não carregam nenhuma informação útil, eles são usados apenas pelo decodificador para tentar recuperar a mensagem original **s**.

Figura 2.1: Sistema básico de comunicação de dados



Fonte: o autor.

O modulador irá gerar um vetor de símbolos **x**. Tipicamente, o modulador utilizado é o BPSK (*Binary Phase-Shift Keying*) (HAILES et al., 2015), pois é uma modulação simples e robusta a erros. Cada símbolo terá apenas um bit, e a mensagem enviada terá os mesmos N bits da mensagem codificada **c**. O valor de cada bit de **x**, pode ser calculado como:

$$\begin{cases} x_j = \sqrt{E_s}, \text{quando } c_j = 0 \\ x_j = -\sqrt{E_s}, \text{quando } c_j = 1 \end{cases}$$

sendo E_s a energia por símbolo, e para fins de simplicidade seu valor igual a 1. Então, **x** será transmitida através de um canal, tipicamente um canal com *Additive White Gaussian Noise* (AWGN), e esse canal fará com que os bits de **x** sejam modificados através da adição de um ruído aleatório.

$$\hat{x}_j = x_j + \mathcal{N}(0, N_0) \tag{2.2}$$

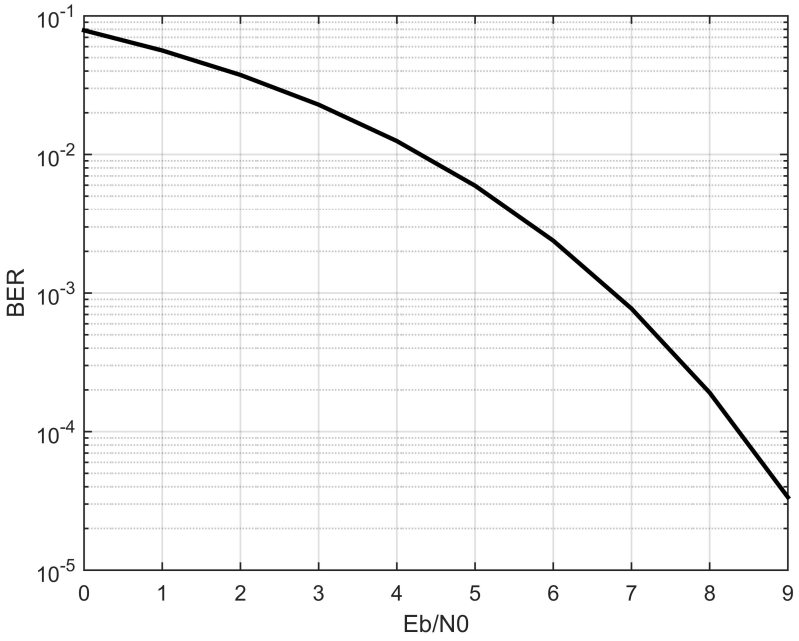
onde $\mathcal{N}$ é uma distribuição normal, com um valor de média igual a 0 e variância N_0, sendo que N_0 é a densidade espectral do ruído.

Comumente compara-se a taxa de erros na transmissão dos dados através de uma relação sinal-ruído, *Signal to Noise Ratio* (SNR), sendo essa dada por $\frac{E_s}{N_0}$. Podemos então calcular

$$\frac{E_b}{N_0} = \frac{E_s}{N_0} * \frac{1}{R} \tag{2.3}$$

onde E_b é a energia por bit da mensagem. $\frac{E_b}{N_0}$ é uma grandeza muito utilizada, pois com ela podemos medir a *Bit Error Rate* (BER) de diferentes sistemas de comunicação. A Figura 2.2 mostra a taxa de erro de bit para diferentes valores de $\frac{E_b}{N_0}$ dada uma modulação BPSK em um canal AWGN, mas sem um codificador de canal.

Figura 2.2: BER de um canal AWGN com modulação BPSK



Fonte: o autor

O ruído adicionado pelo canal irá alterar os valores dos bits de uma maneira imprevisível, fazendo com que a mensagem recebida pelo demodulador, $\hat{\mathbf{x}}$, seja diferente da mensagem enviada $\mathbf{x}$. Isso irá acarretar em uma mensagem $\hat{\mathbf{c}}$ que difere de $\mathbf{c}$, e que possivelmente irá gerar uma mensagem errônea $\hat{\mathbf{s}}$. Cabe ao decodificador de canal tentar recuperar a mensagem original $\mathbf{s}$, a partir de $\hat{\mathbf{c}}$.

O demodulador pode gerar $\hat{\mathbf{c}}$ como um vetor de bits, ou oferecendo um valor de confiança para aquele bit, onde o sinal informa o valor do bit e a magnitude expressa a confiança que se tem no sinal daquele bit. Essa confiança é expressa na forma de um *Logarithmic-Likelihood Ratio* (LLR) (YEO et al., 2001). Um valor de magnitude zero,

significa que há total incerteza sobre o valor daquele bit, ao passo que $\stackrel{+}{-}\infty$ demostra total confiança sobre o valor daquele bit. Dado um modulador BPSK, sobre um canal AWGN, o demodulador pode calcular o LLR de um bit como:

$$LLR_j = 4R\frac{E_b}{N_0}\hat{x}_j \tag{2.4}$$

3 CÓDIGOS LDPC

Nesse capítulo serão abordadas as principais características dos códigos LDPC.

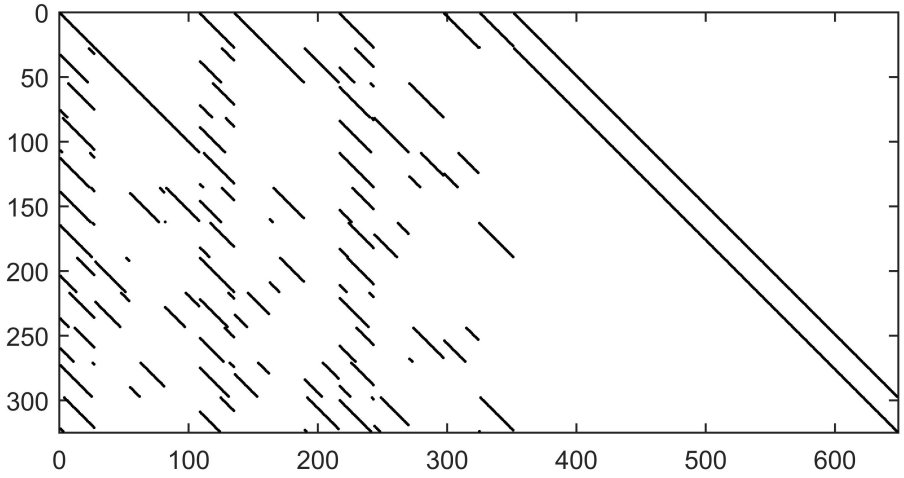
3.1 Matriz de paridade

A matriz de paridade (H) de um código LDPC deve ser esparsa (Figura 3.1), ou seja, conter poucas entradas de valores diferentes de zero. Um código LDPC é dito (d_v,d_c)-regular se **H** possui um número fixo d_v de valores não zero em cada coluna e um número fixo d_c de valores não zero em cada linha de sua matriz de paridade, onde o d_v e d_c representarão o grau de cada VN e de cada CN, respectivamente. É mais interessante que um CN possua um grau baixo, pois a informação que ele propagará no processo de decodificação terá mais valor. Além disso, é interessante que um VN tenha um grau maior, pois estará conectado a mais CNs e a tendência é que no iterar da decodificação o seu valor convirja para o correto.

Um código é dito irregular se o grau variar dentro do conjunto de nodos. Códigos LDPC irregulares estão entre os códigos de correção mais poderosos atualmente. Em (CHUNG et al., 2001) foi construído um código LDPC irregular que ficou distante apenas $0.0045dB$ do limite de Shannon de um canal AWGN.

A matriz H tem um total de N colunas, pois essas representam os bits da palavra codificada (**c**). Ela terá também $N(1-R)$ linhas representando as equações de paridade.

Figura 3.1: Matriz H do padrão 802.11 para N = 648 e R = 1/2



Fonte: o autor.

Por exemplo, dada a matriz Ĥ (JOHNSON, 2006), o conjunto de bits que compõem a *j*-ésima equação de paridade do código (B_j) pode ser definido como:

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

$$B_1 = \Big\{1, 2, 4\Big\}; \quad B_2 = \Big\{2, 3, 5\Big\}; \quad B_3 = \Big\{1, 5, 6\Big\}; \quad B_4 = \Big\{3, 4, 6\Big\}.$$

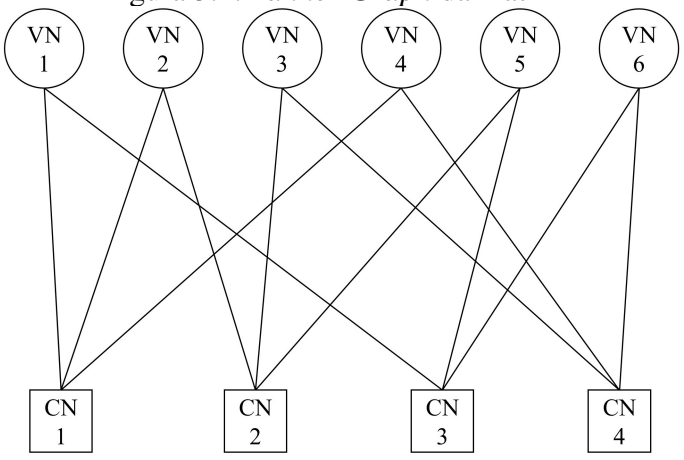
O conjunto A_i, que representa as equações de paridade que verificam o *i*-ésimo bit, pode ser definido como:

$$A_1 = \Big\{1, 3\Big\}; \quad A_2 = \Big\{1, 2\Big\}; \quad A_3 = \Big\{2, 4\Big\};$$
$$A_4 = \Big\{1, 4\Big\}; \quad A_5 = \Big\{2, 3\Big\}; \quad A_6 = \Big\{3, 4\Big\}.$$

3.1.1 Tanner Graph

É comum expressar a conectividade entre os CN e VN através de um grafo bipartido (TANNER, 1981). Isso significa que os nodos do grafo podem ser separados em dois conjuntos diferentes e as arestas conectam apenas nodos de conjuntos diferentes. Haverá N VNs representando as colunas de H e M CNs representando as linhas de H. A conectividade desses nodos dá-se pelas entradas diferentes de zero em H. Na figura 3.2 verifica-se o grafo correspondente de $\hat{\text{H}}$.

Figura 3.2: *Tanner Graph* da matriz $\hat{\text{H}}$



Fonte: o autor.

3.1.2 Códigos LDPC Quasi-Cyclic

A matriz de paridade de um código *quasi-cyclic* (QC) LDPC é uma matriz composta por matrizes circulantes e na Figura 3.3 vemos a matriz de paridade adotada em (IEEE, 2012), que segue esse formato. Note que a matriz da Figura 3.3 é a mesma matriz mostrada em 3.1, porém com as sub-matrizes permutadas expandidas.

Figura 3.3: Exemplo de matriz de paridade no formato QC-LDPC.

0	-	-	-	0	0	-	-	0	-	-	0	1	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	0	0	-	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	0	-	-	-	-	-	0	0	-	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	0	0	-
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
3	-	-	-	16	-	-	2	25	5	-	-	1	-	-	-	-	-	-	-	-	-	-	0

Fonte: retirado de (IEEE, 2012)

Cada posição com um número é uma sub-matriz identidade (ZxZ) rotacionada pelo valor mostrado. Temos que uma sub-matriz P_X é a matriz identidade com suas colunas rotacionadas ciclicamente X vezes para direita. Posições sem nenhum valor equivalem a uma matriz nula. Alguns exemplos estão ilustrados na Figura 3.4.

Figura 3.4: Exemplo de sub-matrizes permutadas para um Z = 8.

$$P_0 = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix}, P_1 = \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}, P_5 = \begin{bmatrix} 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \end{bmatrix}$$

Fonte: retirado de (IEEE, 2012)

Códigos QC-LDPC possuem vantagens na implementação em hardware da decodificação, pois é possível paralelizar parcialmente o processo, já que nenhuma linha está conectada a mais de uma coluna naquela sub-matriz, ou seja, o valor de confiança de um bit não é utilizado por mais de um CN. Isso permite que a computação de vários CNs seja feita em paralelo.

3.2 Codificação

O processo de codificação de uma mensagem, gerar **c** a partir de **s**, consiste em selecionar uma palavra pertencente a código entre todas as que compõem o código.

O processo de codificação dá-se pela equação 3.1:

$$\mathbf{c} = \mathbf{s}G \tag{3.1}$$

onde a matriz geradora (G) é construída a partir de H. Para códigos sistemáticos, a mensagem **c** será da forma

$$(s_1, ..., s_K, p_1, ..., p_{N-K}) \tag{3.2}$$

onde os elementos **s** são os bits das palavra original e o restante são os bits do vetor de paridade.

Para construir G é necessário transformar H, usualmente com Eliminação de Gauss-Jordan, para que fique na forma

$$H = \left[A, I_{N-K}\right] \tag{3.3}$$

onde I_{N-K} é a matriz identidade de tamanho $N - K$ e A é uma matriz de tamanho $(N - K)$xK. Então, calcula-se

$$G = \left[I_K, A^T\right] \tag{3.4}$$

É importante ressaltar que a operação de codificação 3.1 será muito lenta, pois G será uma matriz densa. Para diminuir a complexidade das operações, existem trabalhos que mostram como codificar uma mensagem a partir de H, como (RICHARDSON; URBANKE, 2001).

3.3 Decodificação

Essa seção aborda algumas características importantes no processo de decodificação do LDPC. Na seção 3.3.1 serão descritos os algoritmos mais relevantes para esse trabalho. Por fim, serão mostradas as políticas de escalonamento descritas em 3.3.2, que descrevem outro fator importante no processo que é a forma com as mensagens serão

trocadas entre os nodos.

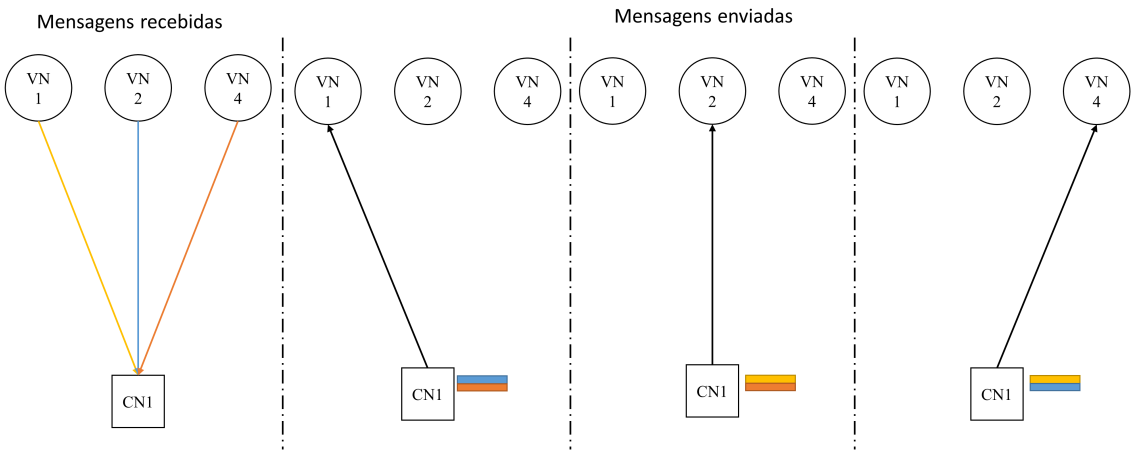
3.3.1 Algoritmos

Nessa seção serão descritos os algoritmos mais relevantes para esse trabalho. Será feita uma breve descrição de cada um e suas mudanças em comparação com os demais. O algoritmo escolhido para implementação em hardware será discutido no capítulo 4.

Note que em todos os algoritmos que serão propostos, um CN/VN sempre ignora, no cálculo da mensagem, o valor do nodo cuja mensagem será entregue. Por exemplo, no Tanner Graph da Figura 3.2, ao calcular a mensagem a ser enviada para o VN 1, o CN 1 não irá utilizar o valor recebido desse nas operações, logo irá considerar apenas os valores do VN 2 e VN 4. Essa prática é utilizada para garantir que os nodos recebam somente informação extrínseca, ou seja, independente da informação que ele próprio já possuía em iterações anteriores.

Um exemplo com todas as mensagens recebidas e enviadas por um CN estão exemplificadas na Figura 3.5. Os valores utilizados pelo CN no cálculo de uma determinada mensagem está mostrado nas cores ao lado do seu quadrado.

Figura 3.5: Exemplo de troca de mensagens entre os nodos.



Fonte: o autor.

3.3.1.1 Sum-Product Algorithm

Sum-Product (MACKAY, 1999) é o algoritmo mais utilizado e é a base para o desenvolvimento dos demais algoritmos que serão citados. Esse algoritmo é baseado na troca de mensagens entre os nodos, sendo essas mensagens LLRs dos valores de confiança para cada bit em cada iteração. É comumente chamado de *Belief-Propagation*.

Inicialmente, os valores das mensagens que cada VN envia aos seus CNs conectados é iniciado com os valores de LLR para cada bit de $\hat{\mathbf{c}}$, ou seja, os valores são inicializados com os valores recebidos pelo canal, de acordo com (2.4). O segundo passo é calcular, para cada CN, as mensagens a serem enviadas aos VNs conectados. Em seguida, cada VN irá calcular novas mensagens e enviar aos CNs. O algoritmo, dessa forma, prossegue iterativamente com os CNs e VNs trocando mensagens a respeito do valor (sinal) e confiança (magnitude) de cada bit. Há duas condições de término para o algoritmo: número máximo de iterações ou uma palavra pertencente ao código ser encontrada. Entretanto, o cálculo dessa palavra é muito custoso e dificilmente implementado, a saber:

$$Hz^T = 0 \tag{3.5}$$

onde z são os bits correspondentes ao sinal de cada LLR. Portanto, essa verificação dificilmente é implementada e o algoritmo, em geral, encerra a execução após atingir o número máximo de iterações.

As mensagens do CN *j* para o VN *i* são representadas por $E_{j,i}$ e as mensagens do VN *j* para o CN *i*, $M_{j,i}$. L_i é o vetor final de bits, onde é feita a *hard-decision* dos valores de confiança, sendo que o valor final será o correspondente ao sinal desse valor. O vetor r_i representa os valores de LLR para cada bit de $\hat{\mathbf{c}}$, calculados com os valores obtidos do canal. O pseudo código do algoritmo é mostrado no Algoritmo 1.

Algoritmo 1: Sum-Product

Input: r, maximo_iteracoes

Output: $\hat{s}$

begin

iteracoes = 0

for $j = 1 : N$ **do**

for $i \epsilon A_j$ **do**

$M_{j,i} = r_i$

end

end

while $iteracoes < maximo\_iteracoes$ **do**

// Operações dos CN

for $j = 1 : M$ **do**

for $i \epsilon B_j$ **do**

$$E_{j,i} = log\left(\frac{1+\prod_{i' \epsilon B_{j,i' \neq i}} tanh(M_{j,i'}/2)}{1-\prod_{i' \epsilon B_{j,i' \neq i}} tanh(M_{j,i'}/2)}\right)$$

end

end

// Operações dos VN

for $j = 1 : N$ **do**

for $i \epsilon A_j$ **do**

$M_{j,i} = \sum_{j' \epsilon A_{i,j' \neq j}} E_{j',i} + r_i$

end

end

$iteracoes = iteracoes + 1$

end

for $i = 1 : N$ **do**

$L_i = \sum_{j \epsilon A_i} E_{j,i} + r_i$

$$\hat{s}_i = \begin{cases} 1, L_i \leq 0 \\ 0, L_i > 0 \end{cases}$$

end

end

3.3.1.2 λ-Min

O algoritmo proposto por (GUILLOUD; BOUTILLON; DANGER, 2003) propõe uma nova maneira de calcular as mensagens trocadas entre os nodos. Consegue uma redução de até 75% na área necessária para guardar as mensagens entre as iterações e sem uma degradação significante na taxa de erros, menos de 1dB para um BER de 10^{-4}.

Um CN utilizará apenas os λ menores valores de magnitude para calcular as mensagens $E_{i,j}$, sendo $\lambda > 1$. Define-se então um conjunto $N\lambda = \{n\{0\}, ..., n\{\lambda-1\}\}$, onde seus elementos são os menores valores de magnitude dentre os VNs conectados aquele CN.

$$E_{j,i} = log\left(\frac{1+\prod_{i'\epsilon N\lambda_{j,i'\neq i}} tanh(M_{j,i'}/2)}{1-\prod_{i'\epsilon N\lambda_{j,i'\neq i}} tanh(M_{j,i'}/2)}\right) \tag{3.6}$$

Há dois possíveis casos: se o bit pertencer ao subconjunto $N\lambda$, uma mensagem será calculada desconsiderando-o, ou seja, computada sobre $\lambda - 1$ valores. Além disso, se o bit não pertencer ao subconjunto, será calculada a mensagem considerando os λ menores valores. Com isso, serão calculados apenas $\lambda + 1$ mensagens, uma para cada bit que pertence ao subconjunto e uma para os demais. Para o cálculo de futuras mensagens do processo de decodificação, será necessário armazenar somente as magnitudes das $\lambda+1$ mensagens e o valor dos sinais de cada VN conectado.

Os valores de menor magnitude são escolhidos, pois são os que tem mais influência sobre a magnitude da mensagem final calculada e com isso a informação associada a eles é mais relevante ao decodificador.

3.3.1.3 Min-Sum Algorithm

O algoritmo de *Min-Sum* (FOSSORIER; MIHALJEVIC; IMAI, 1999) parte do mesmo princípio do λ-Min: calcular menos mensagens e considerar os menores valores de confiança dos bits conectados a um CN.

A diferença existente é que o algoritmo *Min-Sum* não utiliza a função logarítmica apresentada no *Sum-Product* no cálculo de $E_{j,i}$, mas sim uma nova função de mínimo, a saber:

$$E_{j,i} \approx \prod_{i'\epsilon B_{j,i'\neq i}} sign(M_{j,i'}) * min_{i'\epsilon B_{j,i'\neq i}}(\left\|M_{j,i'}\right\|) \tag{3.7}$$

Duas mensagens serão calculadas: uma para o menor valor de magnitude de um

bit, que considerará o menor valor entre os demais VNs conectados e a outra mensagem, que irá considerar o menor valor entre todos e será a mensagem enviada a todos os VNs, menos o que possui esse valor. Esse é um ponto muito importante, pois podemos reduzir drasticamente a área ocupada pelo CN, já que só iremos guardar o valor de duas mensagens entre as iterações e não uma mensagem para cada VN conectado.

Esse algoritmo simplifica muito as operações aritméticas do CN, principalmente em implementações em hardware. O cálculo das mensagens é reduzido apenas a comparações entre os valores, ao contrário de custosas operações aritméticas.

3.3.1.4 Modified Min-Sum

O *Modified Min-Sum* (MMS) (KARKOOTI; RADOSAVLJEVIC; CAVALLARO, 2008) é uma versão modificada do do *Min-Sum* onde um fator de correção ($\beta > 0$) é adicionado ao cálculo dos valores das mensagens com intuito de diminuir a perda de informação decorrente da função simplificada que é utilizada no cálculo. Temos então que $E_{j,i}$ pode ser escrita como:

$$E_{j,i} \approx \prod_{i' \epsilon B_{j,i' \neq i}} sign(M_{j,i'}) * max(min_{i' \epsilon B_{j,i' \neq i}}(\left\| M_{j,i'} \right\|) - \beta, 0) \tag{3.8}$$

Esse algoritmo possui as mesmas vantagens do *Min-Sum* em relação ao número de mensagens salvas durante as iterações do algoritmo, porém possui uma eficiência energética no uso do canal melhor.

3.3.2 Políticas de escalonamento

As políticas de escalonamento na troca de mensagens ente os nodos é outro fator importante no desempenho do decodificador. A política adotada impacta diretamente nas características finais do decodificador e as principais serão descritas a seguir.

3.3.2.1 Flooding

Na política de *Flooding* (ZHANG; HUANG; CHENG, 2012), todos os CNs são ativados, seguidos da ativação de todos os VNs. É uma política com alto poder de paralelismo, já que vários nodos podem ser ativados em paralelo, acarretando em altas taxas de

envio de dados e também uma grande área ocupada em hardware. O algoritmo anterior utiliza essa política.

3.3.2.2 Informed Dynamic Scheduling

Ao utilizar *Informed Dynamic Scheduling* (IDS) (CASADO; GRIOT; WESEL, 2007), inspeciona-se todas as mensagens geradas em uma iteração, ativando o nodo que possui um maior resíduo entre as mensagens, ou seja, ativa o CN que na próxima iteração irá calcular mensagens com maior importância. Essa política tende a gerar um hardware com maior área, porém converge a um resultado satisfatório com um número muito menor de iterações.

3.3.2.3 Layered Belief Propagation

No *Layered Belief Propagation* (LBP) (CHANG et al., 2008), é explorado o formato das matrizes Quasi-Cyclic, pois instancia-se o número de CNs necessários para processar uma linha da matriz QC. Conforme descrito na seção 3.1.2, cada linha desta matriz é composta por sub-matrizes identidade ou matrizes nulas de tamanho Z.

Como essas sub-matrizes têm somente um valor diferente de zero por coluna, não há dependência de dados entre os CNs e esses podem ser processados em paralelo. Computa-se, então, todas as mensagens referentes a uma linha e, por fim, atualiza-se os valores dos VNs. No próximo passo, os próximos CNs irão computar suas mensagens com valores já atualizados dos VNs.

A diferença para o *Flooding* é que não é necessário calcular as mensagens de todos CNs, mas de uma parte deles, antes de atualizar os valores dos VNs. Isso faz com que o processo de decodificação convirja para um resultado correto com um menor número de iterações.

Nessa política também é eliminada a computação de mensagens dos VNs para os CNs, pois agora os valores de cada VN são diretamente atualizados nos CNs. No algoritmo 2, é demonstrada a implementação da política de LBP juntamente com o algoritmo *MMS*, sendo possível comparar as diferenças em relação ao Algoritmo 1 (*Sum-Product*).

Nota-se que todas as operações aritméticas são muito simples, sendo possível calcular todas as mensagens necessárias a partir de somas/substrações e comparações. Além disso, poucas mensagens são calculadas, pois como já foi dito em 3.3.1.4, apenas duas mensagens $E_{j,i}$ são computadas. Para o VN de menor valor, será calculada uma mensa-

gem que o exclui do cálculo. Para todos os outros VNs, as mensagens terão o mesmo valor, pois o valor mínimo da função será o menor valor entre os VNs conectados.

Algoritmo 2: LBP-MMS

Input: r, maximo_iteracoes

Output: $\hat{s}$

begin
  iteracoes = 0
  while $iteracoes < maximo\_iteracoes$ **do**
    for $j = 1 : M$ **do**
      for $i \epsilon B_j$ **do**
        // Mensagens dos VNs
        $M_{i,j} = r_i - E_{j,i}$
        // Mensagens dos CNs
        $E_{j,i} = \prod_{i' \epsilon B_{j,i' \neq i}} sign(M_{j,i'}) * max(min_{i' \epsilon B_{j,i' \neq i}}(\left\| M_{j,i'} \right\|) - \beta, 0)$
        // Atualiza valores dos VNs
        $ri = M_{i,j} + E_{j,i}$
      end
    end
    $iteracoes = iteracoes + 1$
  end
  for $i = 1 : N$ **do**
    $L_i = \sum_{j \epsilon A_i} E_{j,i} + r_i$
    $\hat{s}_i = \begin{cases} 1, L_i \leq 0 \\ 0, L_i > 0 \end{cases}$
  end
end

Todas as operações aritméticas são realizadas nos CNs, incluindo as mensagens dos VNs. Além disso, pode-se paralelizar o laço que percorre os CNs e assim aproveitar o paralelismo entre as linhas da matriz QC.

4 ARQUITETURA

Serão abordadas as dificuldades e benefícios de cada política de escalonamento e de cada algoritmo descrito no capítulo anterior. Além disso será demonstrada a arquitetura escolhida e implementada bem como seu funcionamento.

4.1 Estudo dos algoritmos e políticas de escalonamento

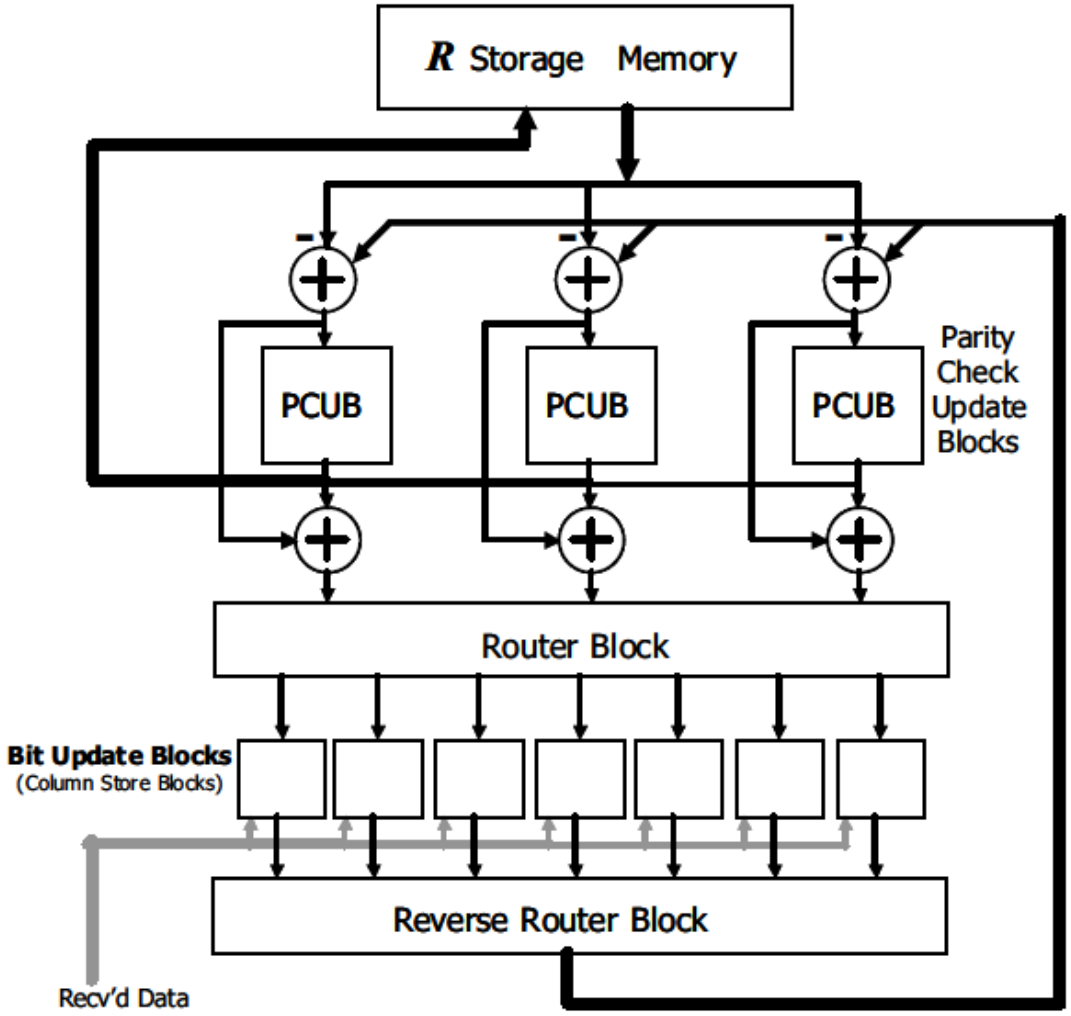
Nessa seção serão abordadas a política de escalonamento escolhida bem como o algoritmo. Será feita uma comparação entre os algoritmos já estudados, demonstrando suas características e limitações quando prevista uma arquitetura em hardware para sua implementação.

4.1.1 Política

A política adotada é a de *Layered Decoding*. Existem diversas implementações de LDPC em FPGA que utilizam essa política, sendo todas fortemente baseadas na arquitetura descrita por (HOCEVAR, 2004) e mostrada na Figura 4.1. Essa arquitetura consiste de uma memória utilizada para guardar os valores dos bits da mensagem a ser decodificada, um permutador que será utilizado para encaminhar os bits corretos aos CNs, os Z CNs que implementam o cálculo da troca de mensagens e um controlador que será utilizado para coordenar as iterações do algoritmo.

No inicio, todos os valores dos bits são salvos nos *Bit Update Blocks* (BUB). Durante o processo de decodificação esses valores serão lidos e encaminhados para os *Parity Check Update Blocks* (PCUB) que irão implementar, com auxílio dos *Routers* e da *Storage Memory* (R), as equações descritas no algoritmo 2. A memória R contém os valores das mensagens trocadas durante o processo de decodificação e seu valor inicial é zero. O fim do processamento de uma linha, os valores são salvos novamente na memória e o processamento continua para uma nova linha. O processamento se repete até que todas as linhas sejam processadas e todas iterações completadas, onde uma iteração é considerada completa ao processar a última linha da matriz.

Figura 4.1: Modelo de arquitetura utilizando *Layered Decoding*



Fonte: retirado de (HOCEVAR, 2004)

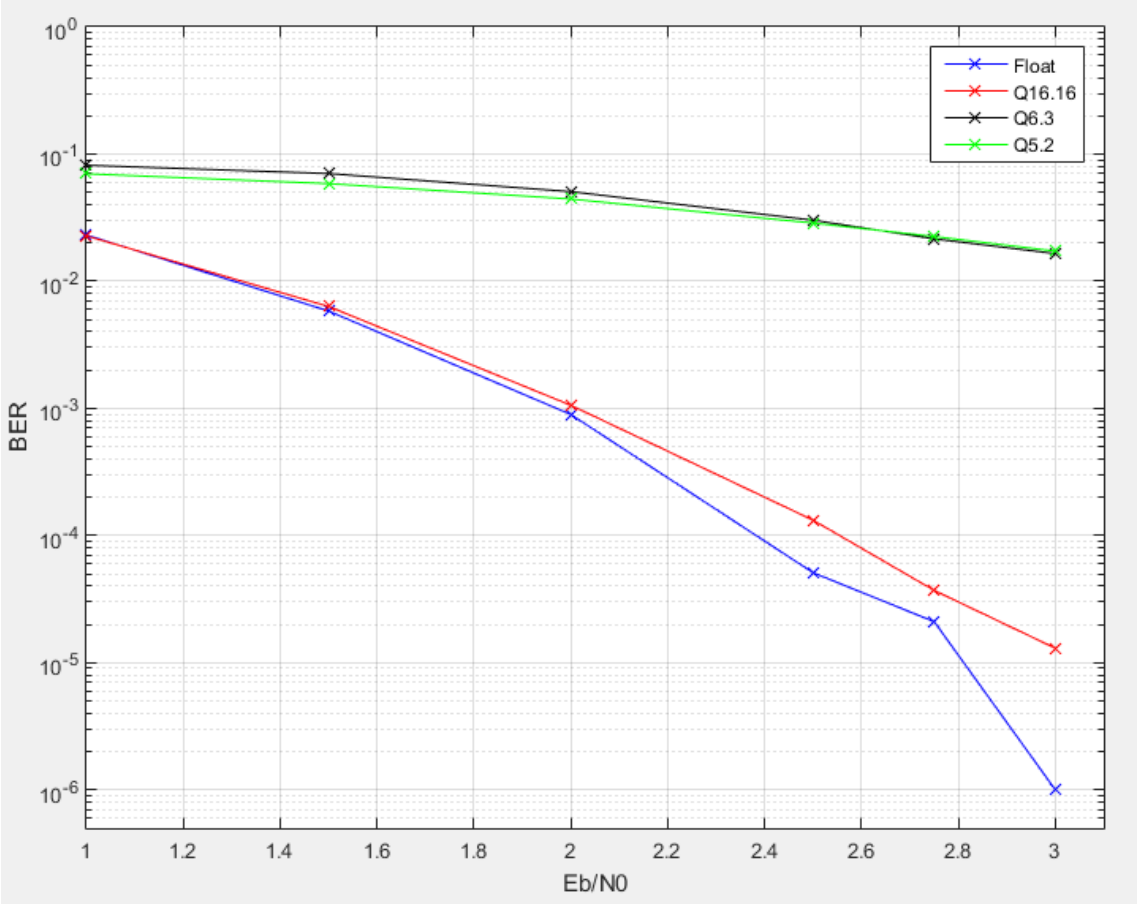
4.1.2 Algoritmo

Com o intuito de implementar uma arquitetura que melhor se enquadrasse nos requisitos necessários, foi implementado uma simulador em ponto-fixo dos algoritmos citados. Todos os resultados são obtidos com base na matriz QC do padrão IEEE 802.11, para um $N = 648$, $R = 0.5$ e $Z = 27$ e cinco iterações do algoritmo de decodificação. A notação adotada para representar o ponto-fixo será a Qx.y, onde x é o total de bits da parte inteira, y o total de bits da parte fracionária e sempre haverá um bit de sinal.

Os melhores resultados foram obtidos com o *Sum-Product*. Isso já era esperado, pois nesse algoritmo não há nenhuma simplificação na hora de calcular os valores das mensagens passadas entre os nodos. Entretanto, como pode ser visto na na Figura 4.2, houve uma grave degradação da performance energética ao utilizarmos os valores de LLR

com no máximo 8 bits, sendo esse o valor escolhido por ser bastante adotado na literatura, como em (KARKOOTI; RADOSAVLJEVIC; CAVALLARO, 2008), (KUO; WILLSON, 2008), (BEUSCHEL; PFLEIDERER, 2008), (CHEN; DAI; YAN, 2006) e outros citados em (HAILES et al., 2015).

Figura 4.2: BER do *Sum Product* para diferentes tamanhos de ponto-fixo



Fonte: o autor

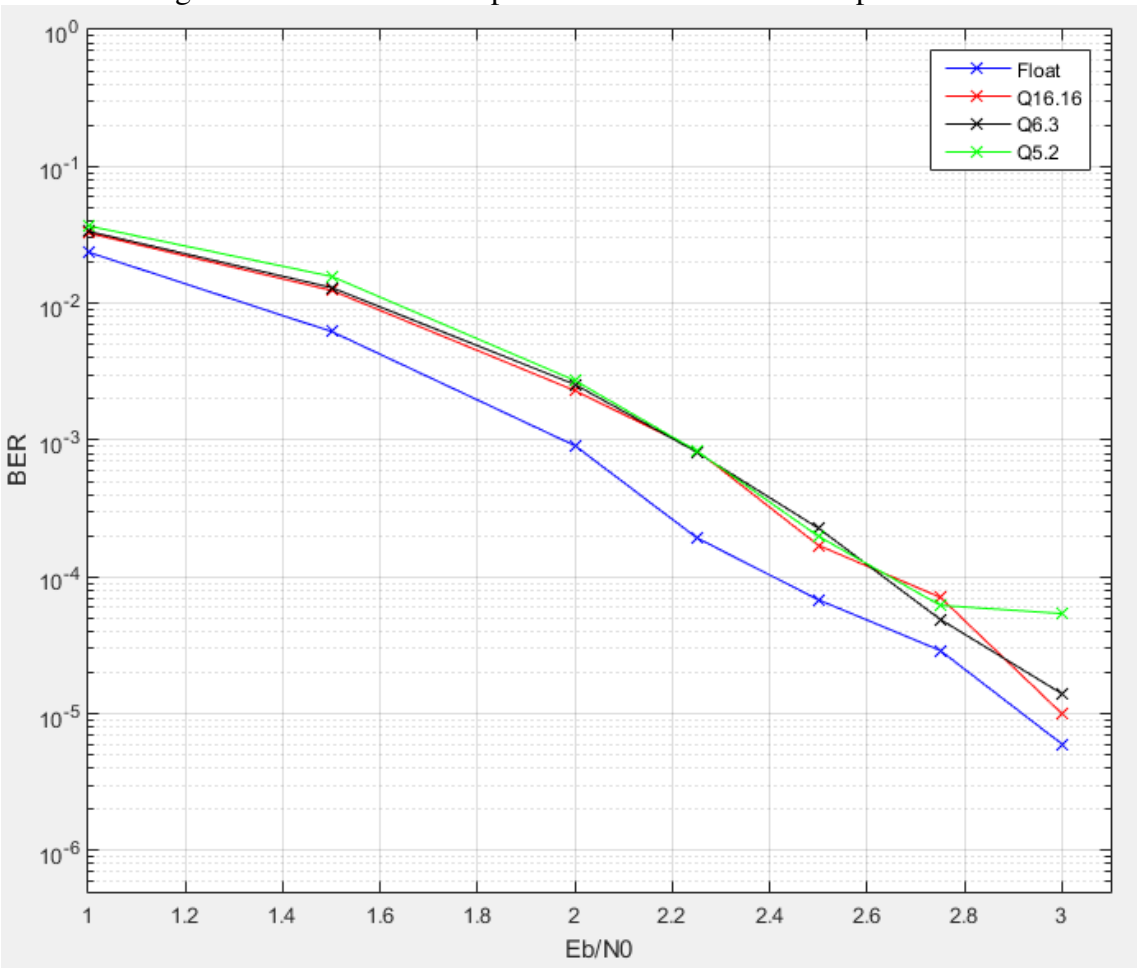
Após estudo com o simulador, conclui-se que a degradação se deve ao fato de que, ao decorrer das iterações, há perda de informação entre as mensagens trocadas entre os nodos, devido aos valores máximos representáveis em uma dada representação em ponto-fixo. Note que as mensagens $E_{j,i}$ que utilizam a função $log(x)$ e $tanh(x)$, como a equação 3.6 e a mostrada no Algoritmo 1, precisam de muitas casas decimais para representar seus valores, pois esses podem chegar na casa de 10^{-7}. Com isso, precisaríamos de muitos bits para representar esses valores e o hardware ficaria com uma área muito grande. Além disso, outro fator contra a utilização desse algoritmo é que precisaríamos tabelar os valores do $log(x)$ para todas as entradas possíveis, pois seria muito custoso calculá-los durante o processo de decodificação.

Um hardware com uma área muito grande pode tornar o processo de injeção de

falhas muito lento, portanto optou-se em buscar uma solução que minimizasse o número de bits na representação em ponto-fixo, mas sem muita perda na performance energética do decodificador.

Foi implementando, então, o algoritmo MMS. Na Figura 4.3 observa-se que a eficiência energética não varia muito ao diminuirmos a quantidade de bits. O fato do cálculo das mensagens ser simplificado é outro fator interessante ao utilizá-lo, pois tudo é feito com somadores e comparadores, sendo desnecessário tabelar os diversos valores de $log(x)$. O valor de $\beta = 0.75$ foi determinado através de várias simulações e foi escolhido por que representava os melhores resultados. Nota-se que há um *error-floor* para um valor de BER = 10^{-4} devido a representação em ponto-fixo. Esse valor é o citado em (HAILES et al., 2015) como sendo o valor alvo para uma avaliação justa sobre a eficiência energética do decodificador, por isso considera-se um valor aceitável.

Figura 4.3: BER do MMS para diferentes tamanhos de ponto-fixo

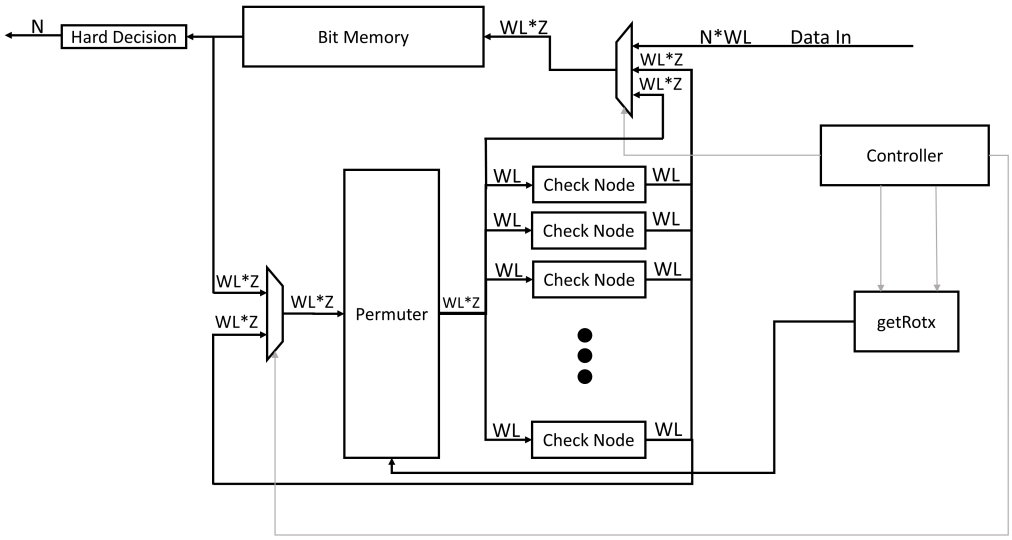


Fonte: o autor

4.2 Arquitetura implementada

A arquitetura implementada utiliza a política de *Layered Decoding* junto com o algoritmo MMS. Na Figura 4.4 temos uma representação simplificada da arquitetura, lembrando que Z é o tamanho de uma sub-matriz da matriz de paridade e WL é o tamanho da palavra em ponto-fixo mais o sinal. Cada módulo será mais bem explicado nas subseções seguintes.

Figura 4.4: Representação simplificada da arquitetura implementada



Fonte: o autor

Um detalhe que difere essa arquitetura da implementada por (HOCEVAR, 2004) (Figura 4.4) é a utilização de apenas um permutador. Não foi utilizado o *Reverse Router Block*, com isso há uma diminuição no número de ciclos de relógio para decodificar uma entrada e uma diminuição da área ocupada, pois só iremos permutar a saída dos CNs ao processar a última linha da matriz.

Com isso, temos que os valores de saída da *BitMemory* são permutados e enviados aos respectivos *Check Nodes*. Ao fim do processamento, esses valores saíram dos CNs também permutados e assim serão armazenados na *BitMemory*. Ao processar a última linha da matriz, os *Check Nodes* irão utilizar o permutador para rotacionar as saídas a posição original, assim ao recomeçar da primeira linha o funcionamento é mantido. Foi necessário alterar a matriz de paridade tabelada em *getRotX* para que essa retorne a diferença do valor de rotação dessa linha/coluna em relação a linha/coluna anterior.

4.2.1 *Check Nodes*

A estrutura de um CN que implementa o algoritmo 2 é demonstrada na Figura 4.5. Sua estrutura interna é divida dois estágios, sendo o primeiro referente ao processamento dos valores de entrada e o segundo é o cálculo da mensagem e atualização do valores de saída.

No primeiro estágio, os valores de $M_{i,j}$ são calculados pelo subtrator que recebe o LLR do bit r_j e o valor $E_{j,i}$ que estão salvos nas memórias internas de cada CN. Cada memória possui Z posições que representam as linhas da matriz de paridade, ou seja, há uma posição para armazenar as informações entre as iterações de cada linha de H. Seus valores iniciais são zerados.

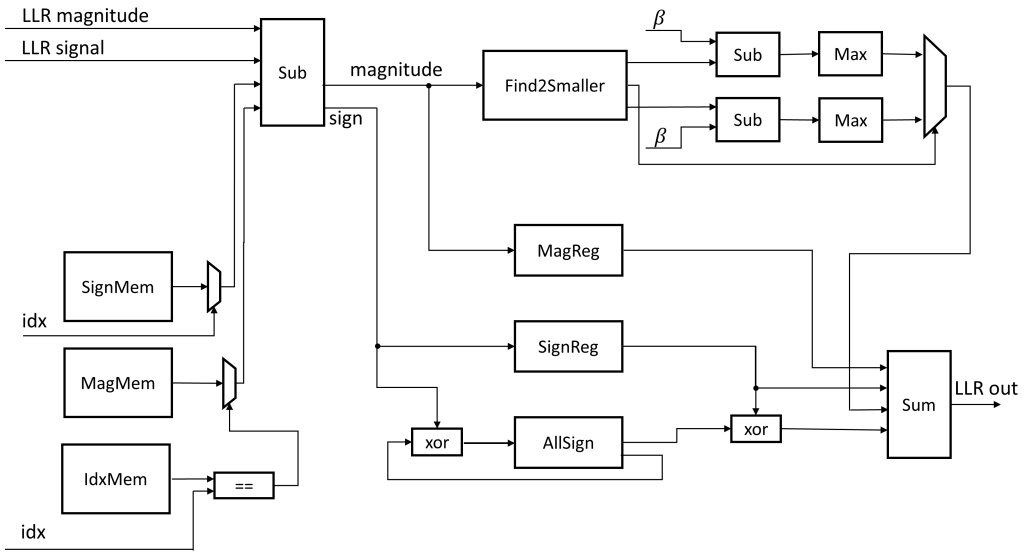
A *MagMem* guarda o módulo das duas mensagens de menor valor calculadas. A *SignMem* armazena os sinais de todos os valores. A *IdxMem* armazena o índice (idx) do LLR que possui menor valor. Com essas informações e as comparações demonstradas na Figura 4.5, é possível recuperar os valores corretos de cada mensagem $E_{j,i}$ calculadas na iteração anterior, pois com o índice conseguimos recuperar a magnitude correta referente ao índice daquele LLR de entrada e também o seu sinal. O índice faz referência a qual valor de confiança estamos recebendo naquele momento.

O valor saída do subtrator ($M_{i,j}$), em módulo, é enviado para o *Find2Smaller*, que irá calcular os dois menores valores entre todos os calculados, e para um registrador que armazena todos os os valores de $M_{i,j}$ calculados e será usado no segundo estágio para calcular os valores de saída do CN ($E_{j,i}$). Pelo mesmo motivo, todos os sinais dos valores $M_{i,j}$ e um registrador (*AllSign*) que armazena o $xor(sign(M_{i,j}))$ são salvos para utilização no próximo estágio.

No segundo estágio é preciso implementar as duas últimas equações do algoritmo, ou seja, $E_{j,i}$ e o r_i atualizado. As magnitudes dos dois valores de saída do *Find2Smaller* são subtraídas da constante β e passam por uma função Max, que efetua a comparação com 0. A saída de cada função Max representa os dois valores calculados para a mensagem $E_{j,i}$.

Para o cálculo de r_i precisamos selecionar o valor correto dentre os dois calculados, para isso utilizamos um índice que também é uma saída do *Find2Smaller* e representa o índice do menor valor. Por fim, somamos o valor correto com os salvos no estágio anterior. Note que *AllSign* representa o somatório dos sinais de todos os valores, logo ao efetuar o xor com o valor do bit salvo no estágio anterior temos o sinal correto.

Figura 4.5: Arquitetura básica de um *Check Node*



Fonte: o autor

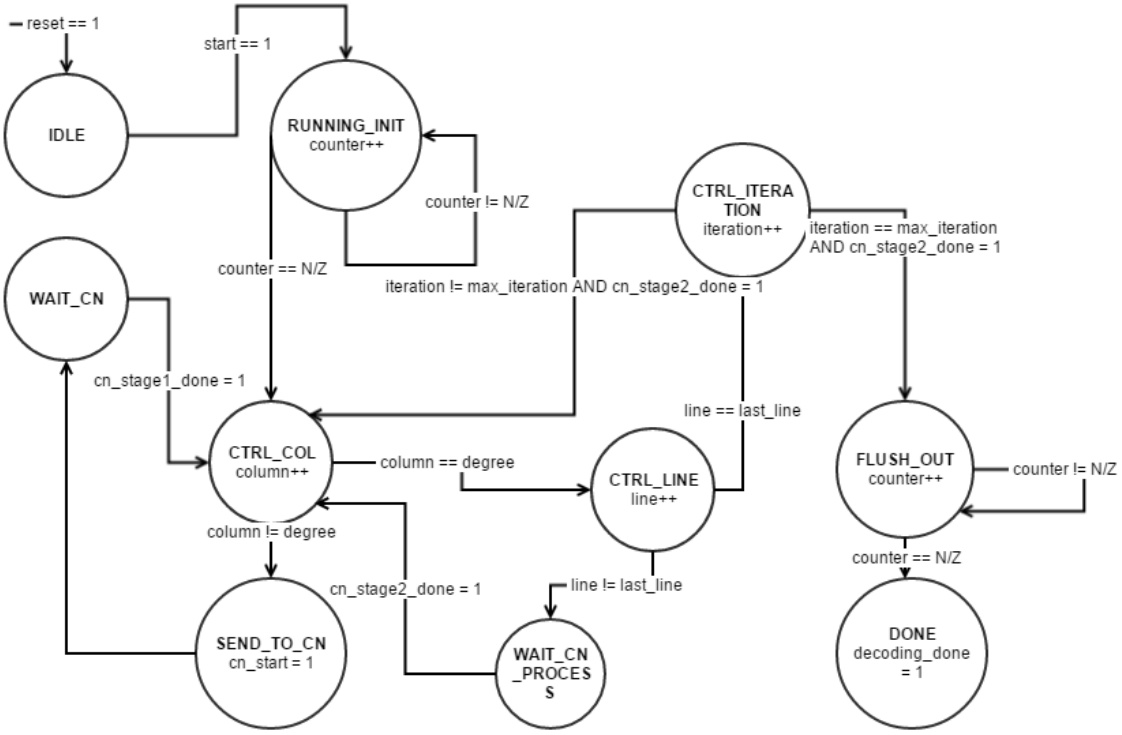
4.2.2 *Controller*

O *Controller* é responsável por controlar o fluxo de dados entre os demais módulos do hardware. No inicio da execução, gerencia que os dados recebidos sejam salvos na *BitMemory* e durante a decodificação controla as informações do algoritmo, como: linha e coluna atual da matriz de paridade, número de iterações, etc. Nas Figura 4.6 está representada a máquina de estados implementada no *Controller*

Os dados de entrada do decodificador são recebidos (RUNNING_INIT) e guardados na *BitMemory*. Na sequência, o processamento das colunas de uma linha da matriz de paridade é realizado (CTRL_COL). O valor *degree* se refere a quantas colunas válidas há naquela linha, ou seja, processa todas colunas de uma linha da matriz.

Então, o controle das linhas é realizado (CTRL_LINE). Se o algoritmo processou a última linha da matriz e o segundo estágio terminou, incrementamos o número de iterações (CTRL_ITERATION) e voltamos para o controle das colunas ou para o término do algoritmo. Se não processou a última linha e o segundo estágio terminou, volta para o controle das colunas e repete o processamento.

Figura 4.6: Máquina de estados implementada no *Controller*



Fonte: o autor

4.2.3 *BitMemory*

Utilizada para salvar os valores de confiança de cada bit de entrada. Possui $N(1-R)/Z$ posições, que representam os valores dos bits de cada sub-bloco da matriz de paridade QC do código (3.3). Ou seja, cada posição é composta pelos valores de confiança dos Z bits de um sub-bloco.

Essa abordagem facilita o funcionamento do decodificador, pois em um ciclo de relógio recuperamos todos os valores que devem ser enviados aos CNs, precisando apenas rotacioná-los no *Permuter*.

4.2.4 *Permuter*

O *Permuter* deve rotacionar os Z valores de saída da *BitMemory* pelo valor de saída do bloco *GetRotX* para que cada *Check Node* receba os valores corretos. Possui uma grande ocupação na área final do hardware, pois todas as possíveis possibilidades são descritas.

Uma outra abordagem seria rotacionar cada valor em um ciclo de relógio, porém

o processo de decodificação ficaria muito demorado.

4.2.5 *GetRotX*

É uma tabela que retorna quantas posições o *Permuter* deve rotacionar os valores que são encaminhados aos *Check Nodes*, para uma dada combinação de linha e coluna. Caso a combinação linha/coluna seja referente a inválida (sub-bloco vazio na matriz de paridade) retorna o valor igual a Z. Note que o valor Z nunca será um valor de rotação útil, pois rotacionar Z valores Z vezes é o mesmo que não rotacioná-los e quando isso é necessário utiliza-se 0. Portanto, Z é o valor escolhido para representar um sub-bloco inválido.

4.2.6 Resultados

Na Tabela 4.1 verifica-se o resultado da síntese do decodificador na FPGA Xilinx Virtex 5, plataforma XUPV5-LX110T, para diferentes valores ponto-fixo. Confirma-se a criticalidade desse fator na ocupação em área do hardware, bem como na performance do decodificador. O número de ciclos se mantêm o mesmo, mas há uma decréscimo na frequência de operação e consequentemente a vazão total do decodificador.

Tabela 4.1: Recursos necessários na FPGA Xilinx Virtex 5

Componentes do LDPC	Quantidade necessária Q5.2			Quantidade necessária Q6.3		
	LUT	FF	BRAM	LUT	FF	BRAM
Controller	146	62	0	152	55	0
getRotX	204	0	0	150	0	0
Check Nodes	12019	10255	0	13525	11712	0
Permuter	1388	216	0	2715	270	0
BitMemory	0	0	4	0	0	7
Decodificador completo	13757	10533	4	16542	12037	7
Frequência de operação	107.720MHz			90.919MHz		
Vazão	13.19Mbps			11.13Mbps		
Ciclos para uma iteração	5290					

Fonte: o autor

Entre todas as implementações de LDPC em FPGA citadas em (HAILES et al., 2015), a que mais se assemelha com as características da implementada nesse trabalho é a implementada em (KARKOOTI; RADOSAVLJEVIC; CAVALLARO, 2008). Ambas implementam uma arquitetura semelhante, baseada no algoritmo MMS, utilizando LBP

Tabela 4.2: Comparação entre decodificadores implementados

Componentes do LDPC	Karkooti Mensagens de 8 bits			o Autor : Q5.2 Mensagens de 8 bits		
	LUT	FF	BRAM	LUT	FF	BRAM
Decodificador completo	19265	13823	87	13757	10533	4
Frequência de operação	160.00MHz			107.720MHz		
Vazão	20Mbs			13.19Mbs		
Ciclos para uma iteração	-			5290		
Número de iterações	-			5		
FPGA utilizada	Virtex 4 - xc4vfx60			Virtex 5 - XUPV5-LX110T		

Fonte: o autor

e com um ponto-fixo de 8 bits. Entretanto, há diferenças na FPGA utilizada, matriz de paridade e funcionamento da arquitetura. Além disso, alguns informações não são apresentadas com clareza, como o número de iterações e o total de ciclos por iteração. O comparativo entre as implementações consta na Tabela 4.2

5 INJEÇÃO DE FALHAS

Nesse capítulo será discutida a abordagem utilizada nos testes de injeção de falhas, bem como os resultados buscados e obtidos.

5.1 Abordagem

A injeção de falhas foi realizada utilizando um injetor de falhas em FPGA especialmente desenvolvido para sistemas de comunicação de dados (LEIPNITZ; HESS; NAZAR, 2016), onde as falhas são injetadas na memória de configuração do FPGA na forma de *single-bit upsets*. Como o processo de injeção pode ser muito lento, optou-se por avaliar a criticalidade dos bits referentes apenas a um *Check Node*. O *Check-Node* é principal módulo do decodificador, pois realiza todas operações aritméticas e faz processamento útil em grande parte do processo de decodificação. Além disso, o total de bits da memória de configuração ocupados é muito menor se comparado ao decodificador LDPC inteiro, diminuindo consideravelmente o tempo necessário para injeção de falhas.

Buscou-se resultados que demostram a criticalidade dos bits para combinações diferentes de entradas. Essas variando em sua relação sinal ruído (Eb/N0), com a finalidade de abranger da melhor maneira possível as situações de funcionamento do decodificador. Para cinco valores diferentes de Eb/N0, foram injetadas falhas para vinte entradas (blocos) do decodificador LDPC, sendo que cada entrada do decodificador acarreta em 440 entradas e 525 saídas do *Check-Node*. Ou seja, foram simuladas 100 entradas do decodificador LDPC para um total de 44 mil entradas e 52.500 saídas no *Check-Node*, para cada falha injetada.

5.2 Resultados

Os erros foram classificados em diferentes tipos para melhor visualização do seu impacto no funcionamento do *Check-Node*. A saber:

- Erros de controle: erros em sinais que controlam o comportamento do *Check-Node*;
- Erros de sinal: alteram o sinal do valor de saída do *Check-Node*;
- Erros de magnitude: altera a magnitude do valor de saída do *Check-Node*, que podem ser:

  - Acréscimo no valor da magnitude
  - Decréscimo no valor da magnitude
- Timeout : o *Check-Node* não gera nenhuma saída;

A cada nova saída gerada no *Check-Node*, os valores de saída (*LLR-OUT* na Figura 4.5) são comparados com o resultado correto. Além disso, são comparados os valores de alguns sinais internos que são utilizados para controle de algumas funções do *Check-Node*.

Foram injetadas falhas num total 272033 bits da memória de configuração, sendo que 11,2% desses geraram algum tipo de erro na saída do *Check-Node*. Na Tabela 5.1 estão os resultados do teste de injeção de falhas para os diferentes tipos de erros e valores de Eb/N0. É importante ressaltar que um bit da memória de configuração pode gerar mais de um tipo diferente de erro, pois o comportamento do *Check-Node* depende dos valores de entrada.

Entretanto, essa é uma avaliação muito simplista do comportamento do *Check-Node* em regime de falhas, pois um bit da memória de configuração que gerou apenas um erro na saída terá o mesmo peso que um bit da memória que gerou 2000 erros, por exemplo. Foi realizado, então, um novo estudo considerando a totalidade de saídas que causaram falhas para cada tipo, ou seja, a probabilidade de ocorrer um erro do tipo 'X', assumindo que um *Check-Node* possui falha, é o total de saídas que geraram erro desse tipo dividido pelo total de saídas dos bits da memória de configuração que geraram falhas. Os resultados podem ser vistos na Tabela 5.2.

Comparando ambas as tabelas nota-se uma grande diferença. Como dito anteriormente, isso se deve ao fato de que o número de saídas do *Check-Node* que geraram erros pode variar bastante para cada bit. Outro fator que influencia nos resultados é o fato do *Check-Node* depender bastante dos valores de entrada, logo a saída não depende apenas da falha injetada e sim da combinação de ambos. Além disso, todo o contexto de

Tabela 5.1: Porcentagem de bits da memória de configuração que geraram saídas com erro

	Eb/N0				
	2	2.25	2.5	2.75	3
Troca de sinal	72,65%	69,95%	68,5%	64,15%	67,8%
Acréscimo na magnitude	77,54%	77,34%	76,57%	76,43%	77,59%
Decréscimo na magnitude	80,92%	80,9%	80,66%	80,42%	80,75%
Erros de controle	20,38%	20,15%	20%	20%	19,8%
Timeout	5,4%	5,4%	5,4%	5,4%	5,4%

o autor.

Tabela 5.2: Probabilidade de ocorrer um erro, para cada tipo, dado que um *Check-Node* contém uma falha

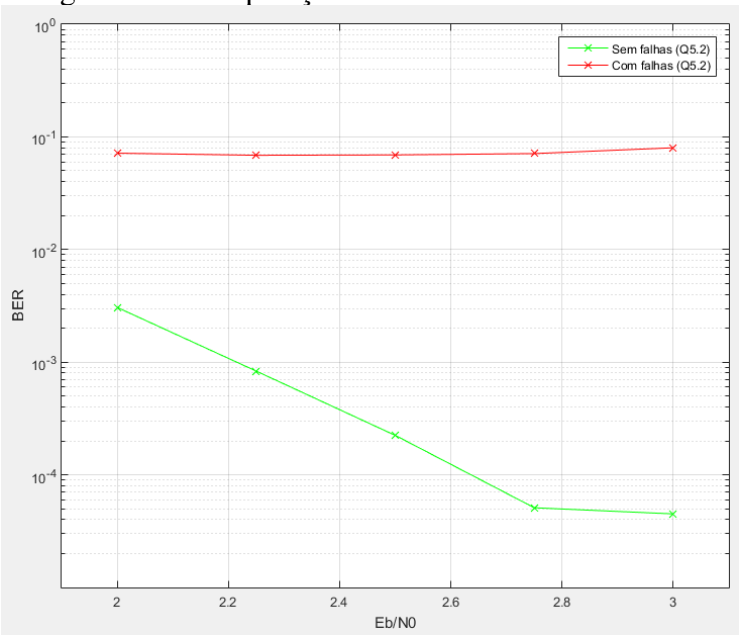
	Eb/N0				
	2	2.25	2.5	2.75	3
Troca de sinal	3,5%	3,35%	3,27%	3,25%	3,2%
Acréscimo na magnitude	8,1%	7,5%	7%	6,6%	6,4%
Decréscimo na magnitude	7,95%	8,1%	8%	8%	8,1%
Erros de controle	4,9%	4,84%	4,82%	4,8%	4,7%
Timeout	5,4%	5,4%	5,4%	5,4%	5,4%

Fonte: o autor.

mensagens que já foram trocadas durante o processo de iteração irá influenciar nas saídas.

O próximo passo é avaliar o impacto desses erros no *Bit Error Rate* do decodificador. Simulamos, então, que um *Check-Node*, do total de 27, está defeituoso e aplicamos à sua saída a probabilidade de ocorrer um erro, baseada nos resultados da Tabela 5.2. Ignoramos erros de Timeout e controle, pois nesses casos é muito difícil prever a saída do *Check-Node*. Por exemplo, ao processar a última linha da matriz de paridade, o *Check-Node* utiliza o permutador para rotacionar os valores e salvá-los na *BitMemory* de maneira correta. Caso o sinal utilizado para esse controle tenha seu valor alterado nesse momento, as saídas não serão permutadas e a memória terá seus valores todos corrompidos, fazendo com que os valores de entrada do *Check-Node* gerados para o teste de injeção não tenham mais validade, pois foram gerados assumindo o comportamento sem falhas do *Check-Node*. As diferenças entre o valor esperado e o valor com falhas simuladas podem ser vistos na Figura 5.1.

Figura 5.1: Comparação entre BER com e sem falhas



Fonte: o autor

O *Check-Node* é um módulo muito crítico no funcionamento do decodificador, pois todas operações aritméticas relacionadas as trocas de mensagens são feitas nele e, como esperado, falhas nesse módulo são muito prejudiciais ao BER do decodificador. Mesmo com o aumento do Eb/N0 o BER continua elevado. Isso pode ser devido ao fato de que nesses valores maiores há mais certeza sobre o valor de um bit, portanto um erro que cause a inversão do sinal do valor, por exemplo, tem uma grande influência negativa em todas as mensagens calculadas com seu valor. Outro agravante é a política de escalonamento utilizada, pois no LBP os valores incorretos calculados se propagam mais rapidamente, pois os valores de confiança são atualizados ao final de cada linha.

Em (MAY; ALLES; WEHN, 2008), foram injetadas falhas em um LDPC implementado em ASIC. Uma comparação mais justa entre os trabalhos é complicada de ser feita, pois a arquitetura utilizada no outro trabalho é diferente, assim como os testes aplicados aos módulos, uma vez que FPGAs e ASICs têm comportamentos bastante diferentes quando sujeitos a falhas.

6 CONCLUSÕES

Nesse trabalho foi apresentado um estudo sobre diversos algoritmos e políticas de escalonamento que podem ser utilizadas nas implementações de códigos LDPC em FPGAs. Foi implementada uma arquitetura que utiliza a política de LBP juntamente com o algoritmo MMS. Além disso, um simulador em C foi implementado, com intuito de auxiliar nos estudos sobre o código e na implementação da arquitetura em hardware.

Os bits sensíveis do *Check-Node* são muito suscetíveis a falhas que afetam o valor da mensagem calculada no módulo, dados que podem ser conferidos na Tabela 5.1. Entretanto, na Tabela 5.2 observa-se que poucas saídas são afetadas ao compararmos com o conjunto de saídas correto. Poderia-se inferir, então, que por esses bits da memória de configuração não afetarem muitas saídas o BER do decodificador não seria muito prejudicado. Entretanto, não é o que foi observado na simulação feita utilizando o simulador escrito em C. O decodificador LDPC aparenta ser muito sensível a modificações nas mensagens trocadas, mesmo que com uma probabilidade de ocorrência pequena. Por exemplo, um bit que deveria possuir um valor de confiança alto para o sinal com valor '1', caso tenha esse sinal trocado, fará com que todas as equações de paridade que utilizam o valor de confiança desse bit calculem mensagens errôneas, ajudando a propagar o valor errado.

Os resultados obtidos foram os esperados, tanto na arquitetura implementada quanto ao *Bit Error Rate* do decodificador LDPC em regime de falhas. Os resultados da ocupação em hardware foram semelhantes com uma arquitetura já proposta na literatura. A simulação realizada, assumindo que um *Check-Node* contém falhas, demonstrou que esse módulo é muito crítico ao funcionamento do decodificador e em alguns casos sendo comparados aos resultados do BER obtidos em (MAY; ALLES; WEHN, 2008). Nota-se que há resultados diferentes, pois falhas em FPGAs são persistentes na memória de configuração, e acabam tem um impacto bem maior que um SEU em um ASIC, mesmo considerando falhas em apenas um *Check-Node*.

Esse trabalho pode ser a base inicial para um estudo mais aprofundado sobre o efeito de falhas em decodificadores LDPC implementados em FPGA. A simulação do *Bit Error Rate* feita nesse trabalho é uma abordagem bem pessimista do decodificador LDPC funcionando em regime de falhas e provavelmente não é o que será visto na prática. Portanto, é interessante novos estudos sobre o *Bit Error Rate* do decodificador, considerando um comportamento diferente no regime de falhas. Além disso, um estudo nos demais módulos seria necessário para propor um decodificador resiliente à falhas. Essa sempre foi

uma preocupação desse trabalho, logo todas as implementações, hardware e simulador, podem ser utilizadas para esse fim.

REFERÊNCIAS

BEUSCHEL, C.; PFLEIDERER, H.-J. Fpga implementation of a flexible decoder for long ldpc codes. In: IEEE. **2008 International Conference on Field Programmable Logic and Applications**. [S.l.], 2008. p. 185–190.

CASADO, A. I. V.; GRIOT, M.; WESEL, R. D. Informed dynamic scheduling for belief-propagation decoding of LDPC codes. In: IEEE. **Communications, 2007. ICC'07. IEEE International Conference on**. [S.l.], 2007. p. 932–937.

CCSDS. CCSDS 131.0-B-2 Recommendation for Space Data System Standards; TM Synchronization and Channel Coding. In: . [S.l.: s.n.], 2009.

CHANG, Y.-M. et al. Lower-complexity layered belief-propagation decoding of ldpc codes. In: IEEE. **Communications, 2008. ICC'08. IEEE International Conference on**. [S.l.], 2008. p. 1155–1160.

CHEN, N.; DAI, Y.; YAN, Z. Partly parallel overlapped sum-product decoder architectures for quasi-cyclic ldpc codes. In: IEEE. **2006 IEEE Workshop on Signal Processing Systems Design and Implementation**. [S.l.], 2006. p. 220–225.

CHUNG, S.-Y. et al. On the design of low-density parity-check codes within 0.0045 db of the shannon limit. **IEEE Communications letters**, IEEE, v. 5, n. 2, p. 58–60, 2001.

ETSI. **ETSI EN 302 307 v1.4.1 Digital Video Broadcasting (DVB); Second generation**. 2014. Available from Internet: <https://www.dvb.org/standards/dvb-s2>.

FOSSORIER, M. P.; MIHALJEVIC, M.; IMAI, H. Reduced complexity iterative decoding of low-density parity check codes based on belief propagation. **IEEE Transactions on communications**, IEEE, v. 47, n. 5, p. 673–680, 1999.

FULLER, E. et al. Radiation test results of the virt x fpga and zbt sram for space based reconfigurable computing. Citeseer, 1999.

GALLAGER, R. G. Low-density parity-check codes. **Information Theory, IRE Transactions on**, IEEE, v. 8, n. 1, p. 21–28, 1962.

GUILLOUD, F.; BOUTILLON, E.; DANGER, J.-L. λ-min decoding algorithm of regular and irregular ldpc codes. In: CITESEER. **Proceedings of the 3rd international symposium on turbo codes and related topics**. [S.l.], 2003. p. 451–454.

HAILES, P. et al. A survey of fpga-based ldpc decoders. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 2, p. 1098–1122, 2015.

HOCEVAR, D. E. A reduced complexity decoder architecture via layered decoding of ldpc codes. In: IEEE. **Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on**. [S.l.], 2004. p. 107–112.

IEEE. IEEE 802.16-2004 Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed Broadband Wireless Access Systems. In: . [S.l.: s.n.], 2004.

IEEE. 802.11-2012-ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. **Retrived from http://standards. ieee. org/about/get/802/802.11. html**, 2012.

JOHNSON, S. Introduction to ldpc codes. In: . [S.l.: s.n.], 2006.

KARKOOTI, M.; RADOSAVLJEVIC, P.; CAVALLARO, J. R. Configurable ldpc decoder architectures for regular and irregular codes. **Journal of Signal Processing Systems**, Springer, v. 53, n. 1-2, p. 73–88, 2008.

KUO, T.-C.; WILLSON, A. N. A flexible decoder ic for wimax qc-ldpc codes. In: IEEE. **2008 IEEE Custom Integrated Circuits Conference**. [S.l.], 2008. p. 527–530.

LABEL, K. A. **Single Event Effect Criticality Analysis**. 1996. Available from Internet: <http://radhome.gsfc.nasa.gov/radhome/papers/seecai.htm>. Accessed in: 10 de maio de 2016.

LEIPNITZ, M. T.; HESS, G. L.; NAZAR, G. L. A fault injection platform for fpga-based communication systems. In: IEEE. **2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS)**. [S.l.], 2016. p. 59–62.

MACKAY, D. J. Good error-correcting codes based on very sparse matrices. **IEEE transactions on Information Theory**, IEEE, v. 45, n. 2, p. 399–431, 1999.

MACKAY, D. J.; NEAL, R. M. Near Shannon limit performance of low density parity check codes. In: . [S.l.]: [Stevenage, etc., Institution of Electrical Engineers], 1996. v. 32, n. 18, p. 1645–1646.

MAY, M.; ALLES, M.; WEHN, N. A case study in reliability-aware design: a resilient ldpc code decoder. In: ACM. **Proceedings of the conference on Design, automation and test in Europe**. [S.l.], 2008. p. 456–461.

OKSMAN, V.; GALLI, S. G. hn: The new ITU-T home networking standard. In: . [S.l.]: IEEE, 2009. v. 47, n. 10, p. 138–145.

RICHARDSON, T. J.; URBANKE, R. L. Efficient encoding of low-density parity-check codes. In: . [S.l.]: IEEE, 2001. v. 47, n. 2, p. 638–656.

TANNER, R. M. A recursive approach to low complexity codes. In: . [S.l.]: IEEE, 1981. v. 27, n. 5, p. 533–547.

VIOLANTE, M. et al. Simulation-based analysis of SEU effects in SRAM-based FPGAs. In: . [S.l.]: IEEE, 2004. v. 51, n. 6, p. 3354–3359.

YEO, E. et al. High throughput low-density parity-check decoder architectures. In: IEEE. **Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE**. [S.l.], 2001. v. 5, p. 3019–3024.

ZHANG, L.; HUANG, J.; CHENG, L. Reliability-based high-efficient dynamic schedules for belief propagation decoding of ldpc codes. In: IEEE. **Signal Processing (ICSP), 2012 IEEE 11th International Conference on**. [S.l.], 2012. v. 2, p. 1388–1392.

APÊNDICE A — TRABALHO DE GRADUAÇÃO I

Injeção e caracterização de falhas em um decodificador LDPC

Geferson Luis Hess Júnior[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

glhjunior@inf.ufrgs.br

Abstract. *LDPC codes are widely used by the industry and were subject to extensive studies by the academic community. Many algorithms and architectures have been proposed to implement these codes in FPGAs, but with no concern for failures that may occur in the configuration memory of these devices. This paper presents a study about LDPC codes and some of its algorithms and architectures, in order to implement an LDPC decoder appropriate for this scenario. Then, inject faults and characterize their effects to propose fault tolerance methods for our implementation.*

Resumo. *Os códigos LDPC são muito utilizados pela indústria e foram extensivamente estudados pela comunidade acadêmica. Inúmeros algoritmos e arquiteturas foram propostas para implementar esses códigos em FPGAs, mas sem preocupação com falhas que podem ocorrer na memória de configuração desses dispositivos. Este trabalho apresenta um estudo sobre os códigos LDPC e alguns de seus algoritmos e arquiteturas, a fim de implementar um decodificador LDPC apropiado para este cenário e, então, injetar falhas e caracterizar seus efeitos, para propor métodos de tolerância a falhas para o mesmo.*

1. Introdução

Dentre os diversos códigos de correção de erros para sistemas de comunicação de dados existentes, o *Low-Density Parity Check* (LDPC) é um dos mais utilizado e estudado, pois possui uma performance próxima ao limite de Shannon [Chung et al. 2001]. Como o próprio nome sugere, são códigos com uma matriz de paridade que contém poucas entradas com valores diferentes de zero.

O LDPC foi proposto por Gallager em 1962 na sua tese de doutorado [Gallager 1962], porém sua implementação em sistemas reais foi considerada muito complexa na época. Décadas depois, acabou sendo redescoberto [MacKay and Neal 1996] e, desde então, vem sendo amplamente estudado e utilizado na indústria, pois é a codificação de canal sugerida na especificação de vários padrões, como: WiFi [IEEE 2012], WiMAX [IEEE 2004], CCSDS [CCSDS 2009], ITU G.hn [Oksman and Galli 2009] e DVB-S2 [ETSI]. Consiste em um *Forward Error-Correction* (FEC) que utiliza uma matriz de paridade, que representa a conectividade entre os *Check Nodes* (CN) e *Virtual Nodes* (VN). Um CN representa uma linha e um VN uma coluna dessa matriz. A matriz de paridade e a função de cada nodo será melhor detalhada na seção 2.

Os códigos LDPC são muito flexíveis e apresentam grande diversidade na escolha dos parâmetros que compõem o código, como: matriz de paridade, o número de elementos diferentes de zero em cada linha ou coluna (peso) e se o código é regular ou irregular. Um

código LDPC é chamado de regular se todas as linhas e colunas da matriz possuírem o mesmo peso, entretanto não é necessário que o peso das linhas seja igual ao peso das colunas. Caso os valores dos pesos forem diferentes, o código é chamado irregular.

Field Programmable Gate Arrays (FPGAs) são dispositivos muito utilizados pela indústria, incluindo a local, para implementar diferentes funcionalidades utilizadas em sistemas de comunicação de dados. Apresentam vantagens úteis a essas funcionalidades, como reconfigurabilidade, alto paralelismo e alta perfomance.

Em ambientes com alta incidência de radiação ionizante, como é o caso das aplicações aeroespaciais, há o risco de efeitos adversos serem causados em dispositivos semicondutores. Há dois principais tipos de falhas, permanentes e transientes, sendo que estamos mais interessados nas transientes, pois essas são mais comuns em FPGAs. Um evento que gera mudança de estado lógico em um dispositivo é conhecido como *Single Event Upset* (SEU) [LaBel 1996].

Os FPGAs possuem uma memória de configuração que é a responsável por configurar os elementos que os compõem, como LUTs (*Lookup Tables)*, flip-flops, blocos de memória e a interconexão desses elementos. Isso faz com que a ocorrência de SEUs sejam muito maléficas à funcionalidade dos módulos implementados no FPGA, pois a memória de configuração de dispositivos baseados em SRAM (*Static Random Access Memory)* são muito sensíveis a esse tipo de evento [E. Fuller and Salazar 1999] e a inversão do valor de um bit dessa memória pode acarretar no funcionamento incorreto de todo o sistema. Note que nem sempre que um bit dessa memória for invertido haverá uma falha no funcionamento do sistema [Violante et al. 2004], pois essa pode ocorrer em um módulo que não está envolvido no processamento daqueles dados.

As características do código LDPC tornam a implementação desse em FPGAs muito interessante, pois há diversos pontos a serem explorados e escolhidos pelo projetista do sistema. Existem inúmeras propostas de implementação de LDPC para FPGAs [Hailes et al. 2015], cada qual variando nos parâmetros do código e/ou nas características da implementação, como: vazão de dados, eficiência energética na transmissão, requisitos de hardware, algoritmo utilizado e flexibilidade. Além disso, a implementação do LDPC em FPGAs traz vantagens devido a simplicidade nas operações aritméticas e ao alto poder de paralelismo dos algoritmos existentes.

O trabalho está estruturado da seguinte forma: a seção 2 apresenta uma base teórica pra a compreensão do funcionamento do decodificador LDPC; a seção 3 contém uma breve análise dos trabalhos relacionados; a seção 4 apresenta um cronograma desejado para realização das tarefas e na seção 5, as considerações finais sobre o trabalho.

2. Fundamentação teórica

Nessa seção, será apresentada a base teórica para o entendimento do correto funcionamento do código LDPC. Na seção 2.1 é discutido um sistema básico de comunicação de dados, a fim de embasar todo o processo de troca de mensagens do sistema; Na seção 2.2 serão abordadas características relevantes aos processos referentes ao codificador e decodificador LDPC.

2.1. Sistema de comunicação de dados

A Figura 1 exemplifica um sistema básico de comunicação de dados. Uma mensagem binária **s**, de K bits, é codificada pelo codificador de canal, obtendo uma nova mensagem **c**, de N bits ($N > K$). Temos que $M = N - K$ bits foram adicionados ao fim da mensagem original **s**, esses M bits são também chamados de bits de paridade. A razão do código (R) pode ser calculada da seguinte forma:

$$R = \frac{K}{N} = \frac{N - M}{N}; 0 < R < 1$$

ou seja, para uma mensagem $\mathbf{s} = 5$ bits e uma razão $R = \frac{1}{2}$, a palavra codificada **c** terá 10 bits. Note que os bits de paridade não carregam nenhuma informação útil, eles são usados apenas pelo decodificador para tentar recuperar a mensagem original **s**.

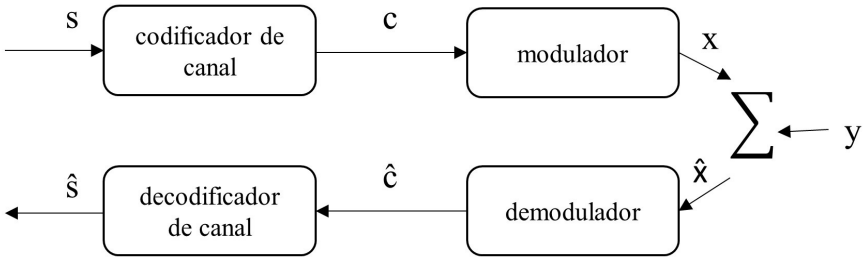


Figura 1. Sistema básico de comunicação de dados

O modulador irá gerar um vetor de símbolos **x**. Tipicamente, o modulador utilizado é o BPSK (*Binary Phase-Shift Keying*) [Hailes et al. 2015], pois é uma modulação simples e robusta a erros. Cada símbolo terá apenas um bit, e a mensagem enviada terá os mesmos N bits da mensagem codificada **c**. O valor de cada bit de **x**, pode ser calculado como:

$$\begin{cases} x_j = \sqrt{E_s}, \text{quando } c_j = 0 \\ x_j = -\sqrt{E_s}, \text{quando } c_j = 1 \end{cases}$$

sendo E_s a energia por símbolo, e para fins de simplicidade seu valor igual a 1. Então, **x** será transmitida através de um canal, tipicamente um *Additive White Gaussian Noise* (AWGN), e esse canal fará com que os bits de **x** sejam modificados através da adição de um ruído **y**.

$$y_j = x_j + \mathcal{N}(0, N_0)$$

onde $\mathcal{N}$ é uma distribuição normal, com um valor de média igual a 0 e variância N_0, sendo que N_0 é a densidade espectral do ruído.

Comumente compara-se a taxa de erros na transmissão dos dados através de uma relação sinal-ruído, *Signal to Noise Ratio* (SNR), sendo essa dada por $\frac{E_s}{N_0}$. Podemos então calcular

$$\frac{E_b}{N_0} = \frac{E_s}{N_0} * \frac{1}{R}$$

onde E_b é a energia por bit da mensagem. $\frac{E_b}{N_0}$ é uma grandeza muito utilizada, pois com ela podemos medir a *Bit Error Rate* (BER) de diferentes sistemas de comunicação. A

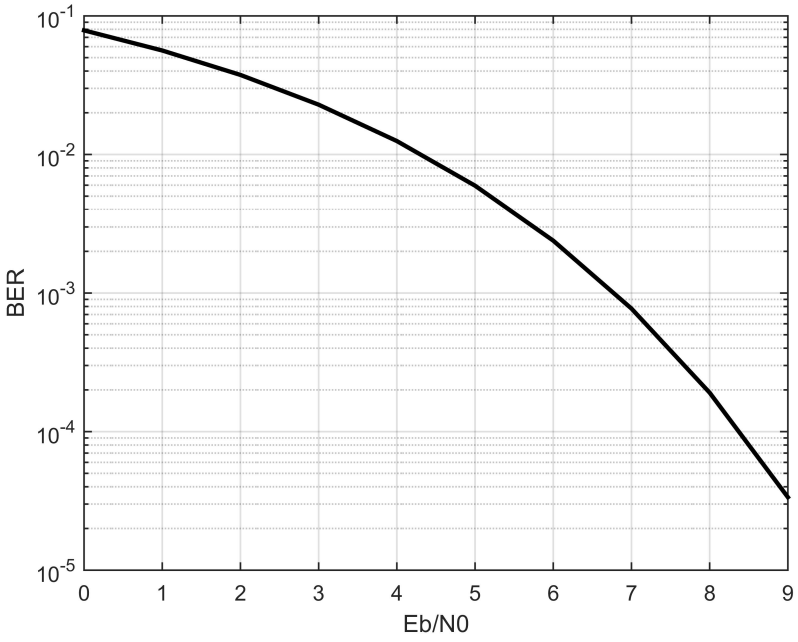


Figura 2. BER de um canal AWGN com modulação BPSK

Figura 2 mostra a taxa de erro de bit para diferentes valores de $\frac{E_b}{N_0}$ dada uma modulação BPSK em um canal AWGN, mas sem um codificador de canal.

O ruído adicionado pelo canal irá alterar os valores dos bits de uma maneira imprevisível, fazendo com que a mensagem recebida pelo demodulador, $\hat{\mathbf{x}}$, seja diferente da mensagem enviada $\mathbf{x}$. Isso irá acarretar em uma mensagem $\hat{\mathbf{c}}$ que difere de $\mathbf{c}$, e que possivelmente irá gerar uma mensagem errônea $\hat{\mathbf{s}}$. Cabe ao decodificador de canal tentar recuperar a mensagem original $\mathbf{s}$, a partir de $\hat{\mathbf{c}}$.

O demodulador pode gerar $\hat{\mathbf{c}}$ como um vetor de bits, ou oferecendo um valor de confiança para aquele bit, onde o sinal informa o valor do bit e a magnitude expressa a confiança que se tem no sinal daquele bit. Essa confiança é expressa na forma de um *Logarithmic-Likelihood Ratio* (LLR) [Yeo et al. 2001].Um valor de magnitude zero, significa que há total incerteza sobre o valor daquele bit, ao passo que $\stackrel{+}{-}\infty$ demostra total confiança sobre o valor daquele bit. Dado um modulador BPSK, sobre um canal AWGN, o demodulador pode calcular o LLR de um bit como:

$$LLR_j = 4R\frac{E_b}{N_0}\hat{x}_j$$

2.2. Códigos LDPC

Nessa seção, serão apresentados detalhes sobre o LDPC. Na seção 2.2.1 será discutido algumas características da matriz de paridade do código; Na seção 2.2.2 a codificação será brevemente comentada, pois não é o enfoque desse trabalho e, por fim, na seção 2.2.3 será apresentando o algoritmo mais utilizado, assim como algumas características relevantes do processo.

2.2.1. Matriz de paridade (H)

A matriz de paridade de um código LDPC deve ser esparsa (Figura 3), ou seja, conter poucas entradas de valores diferentes de zero. A matriz H tem um total de N colunas, pois essas representam os bits da palavra codificada (**c**).Terá, também, $N * R$ linhas representando as equações de paridade.

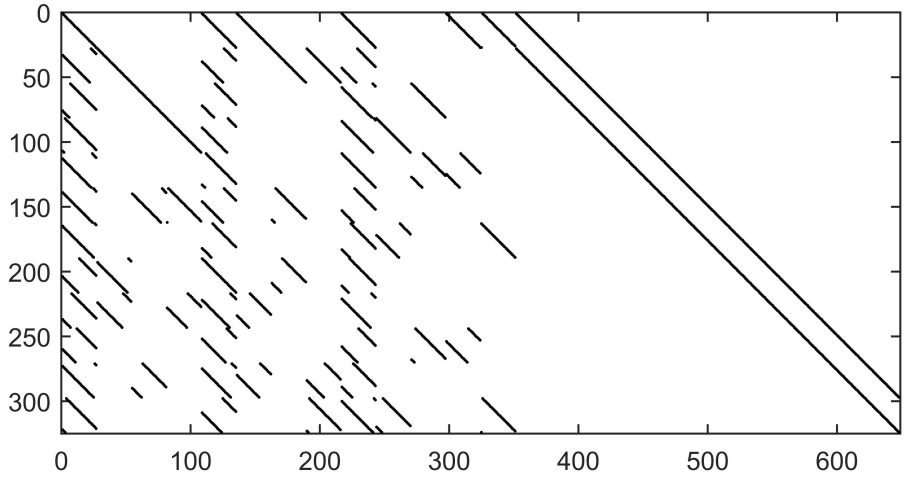


Figura 3. Matriz H do padrão 802.11 para N = 648 e R = 1/2

Por exemplo, dada a matriz $\hat{H}$ [Johnson 2006], podemos definir o conjunto de bits que compõem a *j*-ésima equação de paridade do código (B_j), como:

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

$$B_1 = \{1, 2, 4\}; \quad B_2 = \{2, 3, 5\}; \quad B_3 = \{1, 5, 6\}; \quad B_4 = \{3, 4, 6\}.$$

Podemos definir o conjunto A_i, que representa as equações de paridade que verificam o *i*-ésimo bit:

$$A_1 = \{1, 3\}; \quad A_2 = \{1, 2\}; \quad A_3 = \{2, 4\};$$
$$A_4 = \{1, 4\}; \quad A_5 = \{2, 3\}; \quad A_6 = \{3, 4\}.$$

É comum expressar a conectividade entre os CN e VN através de um grafo [Tanner 1981]. Nesse, haverá N VNs representando as colunas de H e M CNs representando as linhas de H. A conectividade desses nodos dá-se pelas entradas diferentes de zero em H. Na figura 4 verifica-se o grafo correspondente de $\hat{H}$.

2.2.2. Codificação

Para realizarmos a codificação de uma mensagem, isto é, gerar **c** a partir de **s**, é necessário a matriz geradora (G). Seguindo alguns passos, podemos derivar G a partir de H. Para isso,

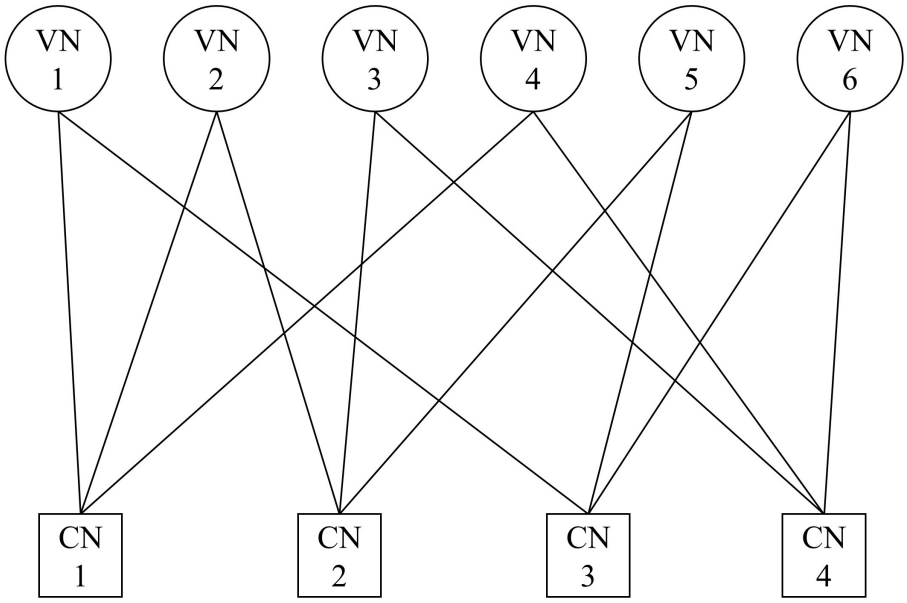


Figura 4. *Tanner Graph* da matriz $\hat{\mathbf{H}}$

é necessário transformar H para que essa fique da forma

$$H = \left[A, I_{N-K}\right]$$

onde I_{N-K} é a matriz identidade de tamanho $N - K$ e A é uma matriz identidade de tamanho $(N - K)K$. Então, calcula-se

$$G = \left[I_K, A^T\right]$$

A matriz geradora não será uma matriz esparsa, logo a operação de codificação

$$\mathbf{c} = \mathbf{s}G$$

será muito lenta. Para diminuir a complexidade das operações, existem trabalhos que mostram como codificar uma mensagem a partir de H, como [Richardson and Urbanke 2001].

2.2.3. Decodificação

Entre os possíveis tipos de algoritmos de decodificação, pode-se optar pela facilmente paralelizável política de *Flooding* [Zhang et al. 2012], onde todos os CNs são ativados simultaneamente, seguidos da ativação simultânea de todos os VNs. Na *Layered Belief Propagation* [Chang et al. 2008], apenas um conjunto de CNs é ativado simultaneamente, com isso um CN pode implementar as regras para diversos CNs, logo perdendo em poder de processamento, mas ocupando uma menor área em hardware. Por fim, a *Informed Dynamic Scheduling* [Vila Casado et al. 2007], inspeciona todas as mensagens geradas em uma iteração, ativando o que possui um maior resíduo entre as mensagens, ou seja, ativa o CN que na próxima iteração irá calcular mensagens com maior importância. Essa política tende a gerar um hardware de maior área, porém converge a um resultado satisfatório com um número muito menor de iterações.

Os algoritmos mais relevantes para esse trabalho serão discutidos em 3, ao passo que nessa seção será apresentando o algoritmo mais utilizado, conhecido como *Belief Propagation* (BF) ou *Sum-Product Decoding* (SP) [Johnson 2006]. Esse utiliza a política de flooding, e é baseado na troca de mensagens entre os nodos, sendo essas mensagens LLRs dos valores de confiança para cada bit em cada iteração.

Inicialmente, os valores das mensagens que cada VN envia aos seus CNs conectados é iniciado com os valores de LLR para cada bit de $\hat{\mathbf{c}}$. O segundo passo é calcular, para cada CN, as mensagens a serem enviadas aos VNs conectados. Por fim, cada VN irá calcular novas mensagens e enviar aos CNs. O algoritmo tem duas condições de término: número máximo de iterações ou uma palavra do código ser encontrada. Entretanto, o cálculo dessa palavra é muito custoso e dificilmente implementado, a saber:

$$Hz^T = 0$$

onde z são os bits correspondentes ao sinal de cada LLR. Portanto, essa verificação dificilmente é implementada e o algoritmo encerra a execução após atingir o número máximo de iterações.

As mensagens do CN *j* para o VN *i* são representadas por $E_{j,i}$ e as mensagens do VN *j* para o CN *i*, $M_{j,i}$. L_i é o vetor final de bits, onde é feita a *hard-decision* dos valores de confiança, sendo que o valor final será o correspondente ao sinal desse valor. O vetor r_i representa os valores de LLR para cada bit de $\hat{\mathbf{c}}$.

function Sum−Product(r,maximo_iteracoes)
  iteracoes = 0
  for j = 1:N **do**
    for i ϵ A_j **do**
      $M_{j,i} = r_i$
    end for
  end for
  repeat
    for j = 1:M **do**
      for i ϵ B_j **do**
        $E_{j,i} = log\left(\frac{1+\prod_{i'\epsilon B_{j,i'\neq i}} tanh(M_{j,i'}/2)}{1-\prod_{i'\epsilon B_{j,i'\neq i}} tanh(M_{j,i'}/2)}\right)$
      end for
    end for
    for j = 1:N **do**
      for i ϵ A_j **do**
        $M_{j,i} = \sum_{j'\epsilon A_{i,j'\neq j}} E_{j',i} + r_i$
      end for
    end for
    iteracoes = iteracoes + 1
  until iteracoes $<$ maximo_iteracoes
  for i = 1:N **do**

$L_i = \sum_{j \epsilon A_i} E_{j,i} + r_i$

$$z_i = \begin{cases} 1, L_i \leq 0 \\ 0, L_i > 0 \end{cases}$$

end for
end function

3. Trabalhos relacionados

Nesta seção são apresentados os trabalhos relacionados referentes a esse trabalho. Na seção 3.1 é discutido um algoritmo diferente do visto acima e na seção 3.2 é apresentado um algoritmo que visa diminuir a área ocupado pelo decodificador. Por fim, na Seção 3.3 é discutido uma arquitetura proposta para um decodificador resiliente a falhas.

3.1. Layered Decoding

É proposto um algoritmo que diminui pela metade o número de iterações necessárias e a área ocupada do decodificador [Hocevar 2004]. O algoritmo tem por base a construção padronizada de alguns códigos, como o do padrão 802.11, onde a matriz de paridade desse código é composta por matrizes identidades permutadas. Com isso, para cada subconjunto de CN que compõem essa submatriz identidade, há garantia de que cada CN irá alterar o valor de apenas um VN.

Visando a diminuição do número de iterações, o algoritmo sugere que cada subconjunto de CN, ao ser processado, atualize os valores dos VNs, propagando com maior velocidade as informações calculadas. Não é mais necessário existência dos VNs, onde há o ganho em área, pois agora todos os cálculos são feitos pelos CNs.

Cada CN do subconjunto, que podem ser processados em paralelo, irá ler o valor do VN conectado a ele, realizar os mesmos cálculos do algoritmo BF, e então atualizar o VN com um novo valor. Quando o próximo subconjunto começar o processamento, eventualmente, algum CN irá ler o valor do VN que foi recentemente atualizado.

3.2. λ-Min

Propõe um novo algoritmo para o cálculo das mensagens trocadas entre os nodos [Guilloud et al. 2003]. Consegue uma redução de até 75% na área necessária para guardar as mensagens entre as iterações e sem uma degradação significante na taxa de erros, menos de 1dB para um BER de 10^4.

Ao invés de calcular uma mensagem para cada VN conectado, um CN irá definir um conjunto com os λ menores valores, em magnitude, conectados a ele. Com isso, irá calcular $\lambda + 1$ mensagens, sendo λ para esses menores valores e a outra mensagem será enviada para os demais VNs conectados.

Os valores de menor magnitude são escolhidos, pois são os que há menor confiança do valor do bit, logo a informação associada a eles é mais relevante para o codificador, pois valores de alta confiança dificilmente irão acarretar em uma possível mudança no valor de um bit. Como são calculadas apenas $\lambda + 1$ mensagens por CN, é necessário apenas guardar o valor dessas mensagens $\lambda + 1$, o que leva a diminuição da área ocupada.

3.3. LDPC Resiliente

Esse trabalho apresenta um decodificador LDPC resiliente à falhas [May et al. 2008]. Nele é proposta uma arquitetura implementada com base no algoritmo apresentando na seção 2.2.3 juntamente com a seção 3.2. Apresenta técnicas que melhoram a performance do decodificador em meios com ocorrência de SEUs. Em sua maioria, triplicação e votação do sinal do bit e, caso valor de magnitude seja modificado, o valor é mudado para zero. Com isso, não associamos nenhuma informação a esse bit e o erro tende a ser amenizado. O trabalho mostra que existe um degradação considerável da performance do decodificador nesses ambientes, sendo assim de extrema importância o estudo de técnicas de tolerância a falhas nesses casos. O trabalho, porém, é focado em *Application Specific Integrated Circuits* (ASICs), que têm um modelo de falhas bastante diferentes dos FPGAs a serem utilizados neste trabalho.

4. Cronograma

As tarefas necessárias para a segunda etapa estão descritas a seguir. O tempo estimado para cada uma estão na Tabela 1. É importante ressaltar que as tarefas 1 e 2 já estão sendo desenvolvidas e estão em estágio de conclusão.

1. Conclusão da implementação dos simulador C++
2. Conclusão da descrição dos módulos VHDL
3. Validação do simulador
4. Validação do VHDL
5. Injeção de falhas
6. Caracterização das falhas
7. Proposta de técnicas de tolerância a falhas para o decodificador implementando
8. Escrita e apresentação do TG2

Tabela 1. Cronograma de atividades

Tarefa	Jun	Jul	Ago	Set	Out	Nov	Dez
1	X						
2	X						
3		X					
4		X	X				
5			X	X			
6				X	X		
7					X	X	
8						X	X

5. Considerações finais

Esse trabalho mostra a importância da utilização dos códigos LDPC e de suas implementações em FPGAs. Apenas encontramos o trabalho [May et al. 2008] que se preocupa em estudar as falhas transientes em implementações de LDPC, porém focado em ASICs. Como já foi dito, existe uma variedade muito grande de algoritmos e arquiteturas, o que leva a uma diversidade muito grande de implementações, tornando necessário mais estudos sobre esses tipos de falhas em códigos LDPC.

Esse trabalho propõe a descrição do hardware parametrizável de um decodificador LDPC, juntamente com um simulador do mesmo descrito em C, visando as características do padrão 802.11[IEEE 2012]. Com esse hardware descrito, injetar e caracterizar as falhas afim de propor técnicas de tolerância a falhas eficientes e de baixo custo. Para isso, existem diversas plataformas para injeção de falhas em FPGAs, sendo que recentemente foi proposta uma plataforma específica para sistemas de comunicação de dados [Leipnitz et al. 2016] que será usada.

Referências

CCSDS (2009). CCSDS 131.0-B-2 Recommendation for Space Data System Standards; TM Synchronization and Channel Coding.

Chang, Y.-M., Casado, A. I. V., Chang, M.-C. F., and Wesel, R. D. (2008). Lower-complexity layered belief-propagation decoding of ldpc codes. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 1155–1160.

Chung, S.-Y., Forney, G. D., Richardson, T. J., and Urbanke, R. (2001). On the design of low-density parity-check codes within 0.0045 db of the shannon limit. volume 5, pages 58–60. IEEE.

E. Fuller, M. Caffrey, P. B. C. C. N. K. and Salazar, A. (1999). Radiation Test Results of the Virtex FPGA and ZBT SRAM for Space Based Reconfigurable Computing. Proceeding of the Military and Aerospace Programmable Logic Devices International Conference.

ETSI. ETSI EN 302 307 v1.4.1 Digital Video Broadcasting (DVB); Second generation. `https://www.dvb.org/standards/dvb-s2`.

Gallager, R. G. (1962). Low-density parity-check codes. *Information Theory, IRE Transactions on*, 8(1):21–28.

Guilloud, F., Boutillon, E., and Danger, J.-L. (2003). λ-min decoding algorithm of regular and irregular ldpc codes. In *Proceedings of the 3rd international symposium on turbo codes and related topics*, pages 451–454.

Hailes, P., Xu, L., Maunder, R., Al-Hashimi, B., and Hanzo, L. (2015). A survey of FPGA-based LDPC decoders. IEEE.

Hocevar, D. E. (2004). A reduced complexity decoder architecture via layered decoding of ldpc codes. In *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 107–112.

IEEE (2004). IEEE 802.16-2004 Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed Broadband Wireless Access Systems.

IEEE (2012). 802.11-2012-ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *Retrived from http://standards. ieee. org/about/get/802/802.11. html*.

Johnson, S. (2006). Introduction to ldpc codes.

LaBel, K. A. (1996). Single Event Effect Criticality Analysis. `http://radhome.gsfc.nasa.gov/radhome/papers/seecai.htm`. Acessado em: 10/05/2016.

Leipnitz, M. T., Geferson, L., and Nazar, G. L. (2016). A fault injection platform for fpga-based communication systems. In *2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 59–62.

MacKay, D. J. and Neal, R. M. (1996). Near Shannon limit performance of low density parity check codes. volume 32, pages 1645–1646. [Stevenage, etc., Institution of Electrical Engineers].

May, M., Alles, M., and Wehn, N. (2008). A case study in reliability-aware design: a resilient ldpc code decoder. In *Proceedings of the conference on Design, automation and test in Europe*, pages 456–461.

Oksman, V. and Galli, S. (2009). G. hn: The new ITU-T home networking standard. volume 47, pages 138–145. IEEE.

Richardson, T. J. and Urbanke, R. L. (2001). Efficient encoding of low-density parity-check codes. volume 47, pages 638–656. IEEE.

Tanner, R. M. (1981). A recursive approach to low complexity codes. volume 27, pages 533–547. IEEE.

Vila Casado, A. I., Griot, M., and Wesel, R. D. (2007). Informed dynamic scheduling for belief-propagation decoding of LDPC codes. In *Communications, 2007. ICC'07. IEEE International Conference on*, pages 932–937.

Violante, M., Sterpone, L., Ceschia, M., Bortolato, D., Bernardi, P., Reorda, M. S., and Paccagnella, A. (2004). Simulation-based analysis of SEU effects in SRAM-based FPGAs. volume 51, pages 3354–3359. IEEE.

Yeo, E., Pakzad, P., Nikolic, B., and Anantharam, V. (2001). High throughput low-density parity-check decoder architectures. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, volume 5, pages 3019–3024.

Zhang, L., Huang, J., and Cheng, L. (2012). Reliability-based high-efficient dynamic schedules for belief propagation decoding of ldpc codes. In *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, volume 2, pages 1388–1392.