UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Paulo Guilherme Kipper - 224126

IMPLEMENTAÇÃO E AMBIENTE DE VALIDAÇÃO EM LÓGICA PROGRAMÁVEL DE UM DECODIFICADOR LDPC

PORTO ALEGRE
MAIO DE 2021

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Paulo Guilherme Kipper - 224126

IMPLEMENTAÇÃO E AMBIENTE DE VALIDAÇÃO EM LÓGICA PROGRAMÁVEL DE UM DECODIFICADOR LDPC

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para Graduação em Engenharia Elétrica.

ORIENTADOR: Altamiro Amadeu Susin

PORTO ALEGRE
MAIO DE 2021

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Paulo Guilherme Kipper - 224126

IMPLEMENTAÇÃO E AMBIENTE DE VALIDAÇÃO EM LÓGICA PROGRAMÁVEL DE UM DECODIFICADOR LDPC

Este projeto foi julgado adequado para fazer jus aos créditos da Disciplina de “Projeto de Diplomação”, do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

ORIENTADOR: ______________________

Prof. Dr. Altamiro Amadeu Susin, UFRGS

Banca Examinadora:

____

Prof. Dr. Altamiro Amadeu Susin, UFRGS

____

Prof. Dr. Tiago Roberto Balen, UFRGS

____

Dr. Cezar Rodolfo Wedig Reinbrecht, TU Delft

DEDICATÓRIA

Dedico este trabalho a meus pais, Paulo e Marta Kipper, e a minha namorada, Carolina Beckenkamp.

“Máquinas frequentemente nos surpreendem.”
Alan Turing

AGRADECIMENTOS

Agradeço aos meus pais, pelo apoio e carinho, e à minha namorada, pelo suporte e dedicação constantes.

Sou muito grato ao professor orientador Altamiro Amadeu Susin, não só pela orientação no presente trabalho, mas também pela caminhada de muito aprendizado que tive no LaPSI.

Agradeço aos colegas do LaPSI pela amizade e troca de experiências.

RESUMO

A concepção de um circuito integrado envolve uma sequência algorítmica de passos a serem cumpridos para transformar uma ideia em "silício". De forma simplificada, um desses passos é a implementação de uma determinada lógica usando linguagens apropriadas para esta finalidade. Fundamentalmente é de suma importância efetivar testes e simulações nessa lógica, propiciando ao desenvolvedor menor risco financeiro, pois é uma oportunidade de encontrar defeitos e assim realizar novos e rápidos ciclos de projeto na lógica gerada. Com o intuito de realizar testes que demandariam excessivo tempo computacional de simulação na lógica em questão, é possível realizar a prototipação em lógica programável, em *Field Programmable Gate Array* (FPGA) e assim, fisicamente exercitar os circuitos digitais nela contida. Porém, para se realizar esta, é necessária a implementação não só do módulo de lógica em questão como também de uma infraestrutura adjacente para estimular o bloco e gerenciar os testes. Neste trabalho é proposta uma arquitetura para executar esses estímulos em um decodificador de correção de erros com estratégia LDPC. Para tal, é efetuada a implementação deste mesmo bloco, que fora anteriormente descrito pelo autor em HDL, juntamente com módulos de gerenciamento dos estímulos para exercitar e coletar os resultados.

Palavras-chaves: Lógica Programável. FPGA. Correção de erros. LDPC. Telecomunicação. Teste e verificação.

ABSTRACT

The conception of an integrated circuit involves an algorithmic sequence of steps to be followed to transform an idea into “silicon”. In a simplified way, one of these steps is the implementation of a certain logic, using languages appropriate for this task. Fundamentally, it is crucial to carry tests and simulations in this logic, providing the developer with less financial risk, as it is an opportunity to find defects and thus carry out new and fast design cycles in the generated logic. To carry out tests that would require excessive computational simulation time in the logic in question, it is possible to perform prototyping in programmable logic, in Field Programmable Gate Array (FPGA), and therefore, physically exercise the digital circuits contained therein. However, to perform, it is necessary to implement the logic module in question and adjacent infrastructure to stimulate the block and manage the tests. An architecture is proposed to execute these stimuli in an error correction decoder with the LDPC strategy in this work. To this end, the implementation of this same block is carried out, which was previously described by the author in HDL, together with modules for managing the stimuli to exercise and collect the results.

Keywords: Programmable Logic. FPGA. Error Correction Codes. LDPC. Telecommunication. Test and verification.

SUMÁRIO

LISTA DE FIGURAS 10

LISTA DE ABREVIAÇÕES 11

1. INTRODUÇÃO 12

    1.1. MOTIVAÇÃO 13

2. DESCRIÇÃO DO PROBLEMA 14

    2.1 DELIMITAÇÃO DO ESCOPO 15

3. ENCAMINHAMENTO DA SOLUÇÃO 17

4. FUNDAMENTAÇÃO TEÓRICA 20

5. METODOLOGIA 24

    5.1 MATERIAIS UTILIZADOS 26

    5.1.1 *SOFTWARE* 26

    5.1.2 *HARDWARE* 27

6. CONTRIBUIÇÕES 29

    6.1 APLICAÇÃO 29

7. DESCRIÇÃO DA SOLUÇÃO IMPLEMENTADA 30

    7.1 ENCAPSULAMENTO DA INTERFACE PCIe: XILLYBUS 31

    7.2 CRUZAMENTO DE DOMÍNIOS DE CLOCK ASSÍNCRONOS 32

    7.3 CONVERSÃO PONTO FLUTUANTE PARA PONTO FIXO 34

8. RESULTADOS E EXPERIMENTOS 36

    8.1 RELATÓRIOS DA IMPLEMENTAÇÃO EM FPGA 36

    8.2 COMPARAÇÃO DO MODELO EM *SOFTWARE* 42

9. CONCLUSÃO 49

10. REFERÊNCIAS 51

LISTA DE FIGURAS

Figura 1 - Algoritmo LDPC implementado no bloco sob teste...15
Figura 2 - Diagrama de blocos da arquitetura proposta 16
Figura 3 - Diagrama do *Software* da cadeia de simulação 18
Figura 4 - Diagrama de teste e verificação...23
Figura 5 - Diagrama do *Software* da cadeia de simulação 23
Figura 6 - Gráfico de resultados esperado para a decodificação LDPC-ADMM com diferentes parâmetros. 25
Figura 7 - kit FPGA modelo XC6VLX365T 27
Figura 8 - Kit FPGA inserido no computador utilizado 27
Figura 9 - Máquina finita de estados simplificada que descreve o controle do módulo sob teste 29
Figura 10 - detalhamento das instâncias e do CDC no sistema 32
Figura 11 - Estratégia de CDC para cruzamento domínio lento para rápido 33
Figura 12 - Estratégia de CDC para cruzamento domínio rápido para lento 33
Figura 13 - Interfaces do conversor de ponto flutuante para ponto fixo após as customizações 34
Figura 14 - Interfaces da block RAM em dispositivos da família Virtex 6 38
Figura 15 - Mapeamento dos pinos exteriores em questão com os transceptores GTX de dispositivos da família Virtex 6 40
Figura 16 - Relatório de posicionamento dos elementos dentro do FPGA 41
Figura 17 - Curvas características da taxa de erro de bit do decodificador em *Software* 43
Figura 18 - Curvas características da taxa de erro de quadros do decodificador em *Software* 44
Figura 19 - Curvas características da taxa de erro de bit do decodificador em *Hardware*.................45
Figura 20 - Curvas características da taxa de erro de quadro do decodificador em *Hardware*............46
Figura 21 - Diferenças da taxa de erro de bit do decodificador em *Software* e *Hardware*...................47
Figura 22 - Diferenças da taxa de erro de quadro do decodificador em *Software* e *Hardware*............47

LISTA DE ABREVIAÇÕES

IP: *Intellectual Property*
FPGA: *Field Programmable Gate Array*
LDPC: *Low Density Parity Check*
HDL: *Hardware Description Language*
IoT: *Internet of Things*
LP: *Linear Programming*
ADMM: *Alternating Directions Method of Multipliers*
LLR: *Log-Likelihood Ratio*
DUT: *Design Under Test*
USB: *Universal Serial Bus*
FIFO: *First In First Out*
PCIe: *Peripheral Component Interconnect Express*
RAM: *Random Access Memory*
SNR: *Signal to Noise Ratio*
BER: *Bit Error Rate*
FER: *Frame Error Rate*
SFP: *Small Form-Factor Pluggable Transceiver*
GPIO: *General Pin Input Output*
CPU: *Central Process Unit*
DMA: *Direct Memory Access*
CDC: Cross Domain Clock
AXI: *Advanced eXtensible Interface*
LUT: *LookUp Table*

1. INTRODUÇÃO

A comunicação entre dispositivos eletrônicos é de suma importância em grande parte das aplicações comerciais criadas na atualidade. Soluções como Internet das Coisas (IoT), telefonia e uso de satélites exigem cada vez mais confiabilidade, segurança, redução de custos e de potência e inúmeras outras restrições específicas para cada aplicação. Adicionalmente, o número de dispositivos enviando e recebendo informações por meio de ondas eletromagnéticas é diretamente proporcional ao nível de ruído causado por interferências. Na prática, a operação desses dispositivos em um meio cada vez mais ruidoso representa um problema de engenharia que abrange diversas áreas de conhecimento. Uma delas se atém à utilização de algoritmos para corrigir eventuais erros em uma mensagem que foi deteriorada na transmissão, propagação ou recepção. A ideia principal é inserir redundância na informação transmitida. Esse trecho adicional é verificado junto com a mensagem original permitindo corrigir um número limitado de erros e com isso adicionar robustez ao sistema. A desvantagem é que todo canal de comunicação obedece ao teorema fundamental da teoria da informação, o Limite de Shannon [1], e por isso acrescentar dados extras reduz a quantidade de informação útil transmitida. Um grande esforço é dedicado para otimizar a solução deste problema.

Enquanto classes específicas de codificadores e decodificadores que atingem considerável desempenho em correção de erros avançaram nas últimas duas décadas, especialmente os métodos de *Low Density Parity Check* (LDPC) e *Turbo Codes*. Seus algoritmos são baseados em iterações e podem ser paralelizados em múltiplas etapas internamente. A proposta de aplicar métodos de programação linear (LP) em decodificação LDPC foi inserida por Feldman et al. [3–5] e estabeleceu uma nova área de desenvolvimento chamada *LP Decoding* [2].

A capacidade desses métodos serem realizados com eletrônica digital, com alta eficiência, ocupando pouca área em silício e consequentemente pouca potência, expande a usabilidade das teorias desenvolvidas descritas cada vez mais, pois são peças elementares em comunicações sem fio. O presente trabalho descreve os testes de um módulo de correção de erros que utiliza as estratégias supracitadas.

No entanto, qualquer implementação em lógica programável deve ser extensivamente testada para exercitar todos os possíveis cenários e idealmente todas as combinações possíveis entre eles, garantindo assim, seu correto funcionamento. Particularmente, no âmbito da correção de erros em transmissão de mensagens, alcançar significância estatística para analisar erros é um desafio. Isso ocorre pois busca-se encontrar mensagens problemáticas sem possibilidade de correção, variando-se a qualidade do canal de comunicação. Quando se trata de um canal deteriorado, essas mensagens irrecuperáveis acontecem com frequência tornando-as facilmente detectáveis. Já quando se deseja analisar a capacidade do decodificador atuar com boas qualidades de sinal, é preciso uma quantidade enorme de mensagens, pois erros acontecem raramente nessas condições. Para tanto, métodos como a co-simulação se tornam uma solução viável, pois estes fornecem a capacidade de testar o *Hardware* desejado num ambiente controlado por um *Software*.

1.1 MOTIVAÇÃO

A concepção de um circuito integrado demanda o processo de testagem de toda a lógica contida no mesmo. Por sua vez, para completar o ciclo de *projeto* de cada um dos IPs descritos em linguagem de *Hardware,* internos ao circuito integrado, é necessária uma etapa de testes para garantir o correto funcionamento da lógica antes de partir para os estágios de *Layout* e fabricação. Desta forma, evita-se riscos e consequentemente custos adicionais neste tipo de desenvolvimento, pois é uma oportunidade de encontrar defeitos e assim realizar novos e rápidos ciclos de projeto na lógica gerada. Com o intuito de realizar testes que demandariam excessivo tempo computacional de simulação na lógica em questão, a prototipação em lógica programável possibilita fisicamente exercitar os circuitos digitais nela contida. Dessa forma, a motivação deste trabalho é criar uma arquitetura genérica que possibilita exercitar testes em IP não só da presente aplicação como também qualquer outro módulo que demande uma grande quantidade de informação nas entradas e saídas.

2. DESCRIÇÃO DO PROBLEMA

Para validar o funcionamento de uma implementação em eletrônica digital de um decodificador de correção de erros, pode-se procurar falhas que ocorrem raramente quando o mesmo é submetido a condições de sinal boas. Por exemplo, em uma certa relação sinal ruído, um determinado decodificador apresenta um erro dentro de um milhão de mensagens decodificadas. Para se ter significância estatística a fim de quantificar essas falhas, é necessário testar uma quantidade de mensagens muito maior do que um milhão. Isso inviabiliza a validação da lógica descrita utilizando ferramentas computacionais, já que cada mensagem decodificada leva um tempo grande para ser simulada, e milhões delas devem ser executadas. A solução para este problema é ter o decodificador implementado fisicamente em circuitos digitais e submeter o mesmo a testes de estresse alimentando suas entradas e conferindo as saídas.

Tendo em vista a relevância dos resultados obtidos utilizando a maneira supracitada de atacar o problema da decodificação LDPC, é proposto um trabalho de testagem da lógica de uma implementação de um algoritmo proposto por Zhang e Siegel [6], em lógica programável. Esta estratégia utiliza os conceitos de *LP Decoding* juntamente com um método matemático que se chama *Alternating Directions Method of Multipliers* (ADMM). O algoritmo é mostrado na Figura 1, no qual a variável “x” carrega os valores de entrada do processo, contendo os valores oriundos da demodulação, em forma de *Log-Likelihood Ratio* (LLR), que indicam, de forma sucinta, a probabilidade de um símbolo demodulado recebido pertencer a um símbolo da constelação utilizada, conforme indica a Equação 1 [7]:

$$\gamma i = \eta \, log\left(\frac{P(yi|xi=0)}{P(yi|xi=1)} \right)$$

Equação 1 - Descrição matemática da LLR.

Figura 1 - Algoritmo LDPC implementado no bloco sob teste.

Algoritmo 1 Algoritmo LDPC iterativo baseado em ADMM

1: **Inicialização:** Inicializar x_{ij} y_{ij} e z_{ij} para todos i e j
2: **Cheque de atualização:** Para cada nodo de checagem em j, atualizar z e y e reescrever como:

$$\mathbf{w} \leftarrow \mathbf{T}_j\mathbf{x} + \mathbf{y}_j \quad \mathbf{z}_j \leftarrow \Pi_{\mathcal{P}_{d_j}}(\mathbf{w}) \quad \mathbf{y}_j \leftarrow \mathbf{w} - \mathbf{z}_j.$$

A mensagem trasmitida para os nodos de atualização de variáveis é

$$L_{j\to i} \leftarrow (z_j)_i - (y_j)_i.$$

3: **Atualização de variável:** Para cada variável em i, as mesagens transmitidas aos vizinhos é a mesma, e pode ser reescrita como:

$$x_i \leftarrow \frac{1}{d_i}\left(\sum_{j'\in N_i} L_{j'\to i} - \frac{\gamma_i}{\rho}\right).$$

4: **Critério de finalização:** o algoritmo termina suas operações se as equações forem satisfeitas

$$\textstyle\sum_{j\in J} \|\mathbf{T}_j\mathbf{x} - \mathbf{z}_j\|_2 < \epsilon^{\text{pri}} \quad \text{e} \quad \sum_{j\in J} \|\mathbf{z}_j^k - \mathbf{z}_j^{k-1}\|_2 < \epsilon^{\text{dual}}$$

com $\epsilon^{\text{dual}} > 0$ e $\epsilon^{\text{pri}} > 0$

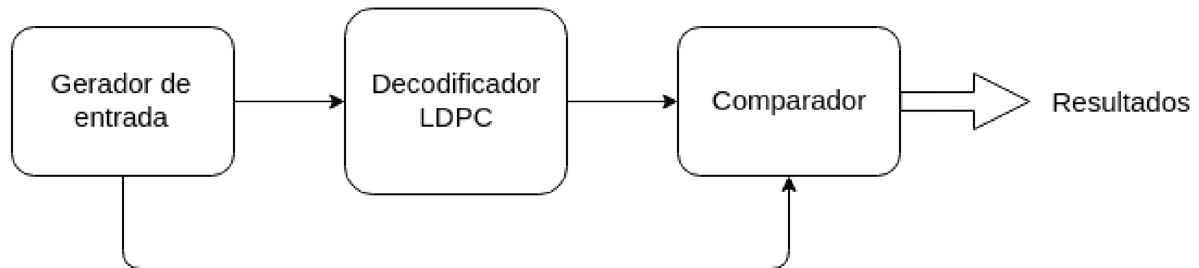
Adaptado de: X. Zhang e P. Siegel - Efficient Iterative LP Decoding of LDPC Codes with Alternating Direction Method of Multipliers.

2.1 DELIMITAÇÃO DO ESCOPO

O presente trabalho tem como objetivo principal a implementação e validação em lógica programável de um bloco de propriedade intelectual (IP) que realiza o algoritmo supracitado já descrito em linguagem de *Hardware* (HDL) e simulado previamente pelo aluno em atividade de estágio obrigatório. Para tal, uma infra estrutura de testes precisa envolver o módulo a ser testado (DUT). Este ambiente tem por finalidade simular uma situação mais próxima o possível do que o IP será submetido quando na fase final. Todos os módulos estarão contidos em um *Field Programmable Gate Array* (FPGA). Para excitar o DUT, é necessário emular o funcionamento de um demodulador, provendo as informações de LLR que servem como entrada. Dois blocos básicos de interface e de infraestrutura serão aplicados

nas interfaces do decodificador LDPC e, assim, validar os resultados experimentais. Concomitantemente, um conjunto de lógica será responsável por avaliar a entrada e saída de dados, contabilizando erros e acertos quantitativamente.

Figura 2 - Diagrama de blocos da arquitetura proposta.



Fonte: O autor.

3. ENCAMINHAMENTO DA SOLUÇÃO

A implementação de um decodificador LDPC em lógica programável demanda o correto dimensionamento das interfaces de dados. Por se tratar de um módulo de correção de dados em alta velocidade, grandes quantias de informação deverão ser injetadas no bloco sob teste. Outra informação relevante é que a entrada do decodificador obrigatoriamente recebe suas LLR's no nível digital de forma fracionária representada em ponto fixo. Este fato implica que cada porção de dados presente num pacote demandará uma largura de *bits* pré determinada a nível de projeto. Já que para a execução do algoritmo é necessário que um pacote inteiro de dados esteja presente na interface de entrada, o decodificador LDPC tem suas entradas em formato de quadros, e ainda mais, como supracitado, cada elemento é representado em *fixed point*. Embora o bloco de *Hardware* tenha sido especificado genericamente, sendo assim projetado para ter seus tamanhos parametrizáveis, um número comum para a aplicação primária a qual ele foi desenvolvido é 16 *bits* de largura na representação de ponto fixo e um tamanho de bloco de 32 elementos.

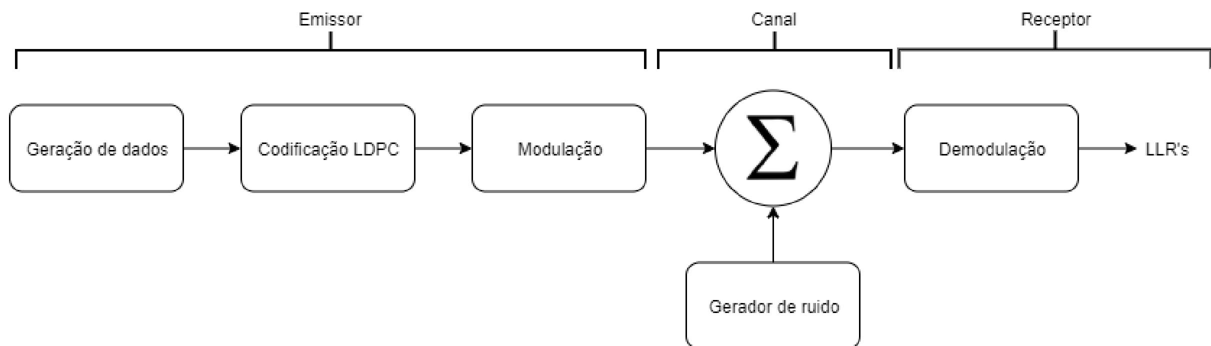
A saída do bloco de decodificação é de maneira geral mais simples, pois conta com uma saída booleana para cada elemento do pacote. Isso ocorre devido à natureza do sistema, que é justamente corrigir eventuais erros nas mensagens e executar uma decisão minimizando a chance de um erro estar presente na saída.

Outro fator a ser implementado é a geração de um sistema de *clock* e *reset* para o correto funcionamento dos circuitos lógicos. Atenta-se que a implementação do bloco lógico numa tecnologia distinta de FPGA (a ser determinada) implicará uma velocidade máxima estipulada pela ferramenta de síntese. Essa frequência é dada pelos caminhos críticos e tempos de *setup and hold* dos elementos lógicos específicos de cada família de FPGA, daí a dependência de velocidade. A fim de maximizar esta característica, o bloco foi desenvolvido atentando para regras de otimização, o que pode ser feito a nível lógico projetando caminhos críticos menores.

Os dados de entrada serão gerados por um sistema de simulação de canal, o qual tem 5 funções elementares, conforme a Figura 3. A cadeia de simulação como será referenciada neste trabalho, executada em *Software*, tem sua primeira tarefa constituída pela geração de dados dos pacotes. Esses dados binários são codificados também com estratégia LDPC de forma a casar com o decodificador a

ser validado. O fluxo é direcionado a um simulador de modulação genérico. A partir daí os dados entram no simulador de canal, que acrescenta ruído às componentes da modulação. Feito isso, é simulada a demodulação da informação oriunda do canal. A saída do demodulador é então formatada para servir de entrada para o decodificador.

Figura 3 - Diagrama do *Software* da cadeia de simulação.



Fonte: O autor

Após a execução da simulação da cadeia de geração, será preciso, obrigatoriamente, implementar a transferência dos dados que estão no domínio do sistema computacional, para o domínio físico da lógica programável. Essa interface é responsável pela limitação da operação de velocidade do sistema, devido à grande troca de dados.

Propõe-se 4 ideias preliminares para solucionar o problema de transferência de dados.

1. Transferir dados por USB do domínio do computador para o FPGA. Este método de transferência apresenta robustez e alta velocidade. O tempo de desenvolvimento de tal interface seria intermediário, já que existem soluções comerciais disponíveis de circuitos integrados que são responsáveis por decodificar o protocolo USB e fornecer dados em forma de FIFO em sua saída, simplificando o projeto.
2. Utilizar o protocolo PCIe: Concede enorme taxa de transferência, mas tem um longo tempo de desenvolvimento e um complexo sistema de infraestrutura a ser projetado.
3. Pré carregar uma grande quantidade de dados embutidos na lógica programável. Isso implicaria um tempo maior de síntese e por consequência

um número menor de ciclos de projeto. A vantagem dessa modalidade consiste na facilidade de desenvolvimento do bloco de entrada de dados. A principal desvantagem seria a limitação de dados a serem testados no sistema, necessitando de uma nova síntese para cada remessa de testes.

4. Através de um *Socket Ethernet*: enviar e receber dados utilizando *Ethernet* propicia uma quantidade suficiente de dados trocados.

Neste trabalho, a segunda opção, a do PCIe, será utilizada para efetuar a comunicação entre o *Software* e o FPGA.

Por fim, o último elemento da cadeia é responsável por contabilizar a quantidade de erros e acertos do bloco em questão, para propiciar a análise dos dados. Tecnicamente se tratariam de comparadores seguidos por dois contadores com uma grande largura de *bits*. Uma máquina de estados seria necessária a fim de gerenciar toda parte de controle do sistema. As figuras de mérito no âmbito da correção de erros são velocidade, latência, proporção de número de bits corretos e errôneos e proporção de número de pacotes corretos e errôneos. Esse último bloco também deve proporcionar comunicação com o domínio do computador, para que os dados possam ser colhidos, apresentados e analisados de forma útil.

4. FUNDAMENTAÇÃO TEÓRICA

Codificação de correção de erros é uma área de estudo que visa transmissão de dados com máxima confiança possível em um canal ruidoso, encontrando a combinação mais provável de dados no receptor que fora enviada por um emissor.

A construção matemática da teoria de decodificação especializada em correção de erros derivou inicialmente no trabalho de Richard Hamming [11]. Após a devida estruturação matemática, um nicho muito grande foi deflagrado com o objetivo de aperfeiçoar cada vez mais a técnica de decodificação, tornando-a mais eficiente e também mais confiável, possibilitando a extração de mensagens corretas em canais cada vez mais ruidosos. Um trabalho oriundo dessa fase e muito importante na área de correção de erros até hoje, trata da codificação Reed-Solomon [1], que utiliza álgebra para atacar o problema da correção de erros.

Contudo, a complexidade quadrática dos algoritmos propostos até então é um contraponto de se utilizar ferramentas algébricas para solucionar o problema de decodificação, pois esta imprime atrasos durante todas as operações aritméticas executadas no processo. Em seguida, o extenso trabalho de Gallager [12] originou as codificações de Low Density Parity Check (LDPC). Estas, por sua vez, visavam simplicidade nas operações para conseguir atender melhor desempenho, indo contra o uso de operações matemáticas algébricas complexas. Juntamente com a simplicidade, os algoritmos nessa subárea se tornaram iterativos, com complexidade linear, se opondo à quadrática previamente apresentada. Adicionalmente, uma característica que discerne essa nova geração de decodificação da anterior, é o uso de entradas *soft-decision*, as quais consistem em informação no domínio contínuo provenientes do bloco anterior, usualmente demodulador. Isso propicia uma maior maleabilidade do sistema em se adaptar a diferentes canais.

Com o avanço contínuo da área de decodificação, outras formas de modelar o problema foram sendo postas à prova. Dentre elas, a tentativa de relaxar a programação linear presente nas propostas supracitadas. Isto é, priorizar ainda mais desempenho abrindo mão de exatidão nas operações matemáticas. Feldman [3] propôs que mesmo não garantindo matematicamente a máxima probabilidade da mensagem ter sido decodificada corretamente, o desempenho desse método é

extremamente competitivo com os demais algoritmos. Além disso, as ideias subsequentes, como as de Siegel [6], provenientes deste trabalho, passaram a habilitar arquiteturas ainda mais simples e, em especial, paralelizáveis, tornando a implementação em *Hardware* mais atrativa.

Neste viés, Barman [9] introduz alternativas que implementam estruturas de *Message passing*, e técnicas de decomposição, usando lógica de *Alternating Direction Method of Multipliers* (ADMM).

ADMM é uma solução simples e poderosa otimizada que combina decomponibilidade e propriedades de convergência de um problema de otimização convexo [8]. Mesmo não se tratando de uma concepção recente, pois foi introduzido por Gabay, Mercier, Glowinski, e Marrocco na década de 70, o uso do ferramental ADMM se estende desde problemas estatísticos, *machine learning* e até computação gráfica modernos.

Um componente chave na solução de problemas utilizando ADMM é o método utilizado internamente para projetar um vetor de valores reais num conjunto fazível de soluções, chamado politopo fundamental, no contexto de programas lineares. Como citado em [9] e aprimorado em [8], este algoritmo é baseado no fato de que um politopo convexo inscrito definido por uma matriz de paridade pode ser expresso como uma superfície convexa de todos os vetores binários com a mesma ponderação. Dessa forma, para um determinado vetor, o algoritmo primeiramente categoriza ordenadamente as coordenadas de um vetor em ordem decrescente e depois as caracteriza como duas fatias cuja superfície contém a projeção [6]. Contudo, essa reordenação dos vetores implica um grande peso computacional, prejudicando majoritariamente a eficiência do método. Para tal, em [13] é apresentado uma nova forma de projeção estabelecendo a teoria de paridade do politopo no algoritmo de projeção, no qual algumas componentes da entrada são fixadas durante as iterações e o problema é constantemente reduzido. Em contraste com as atuais soluções, este demonstrou 37% menos operações aritméticas e, adicionalmente, não utilizou as operações de reordenamento.

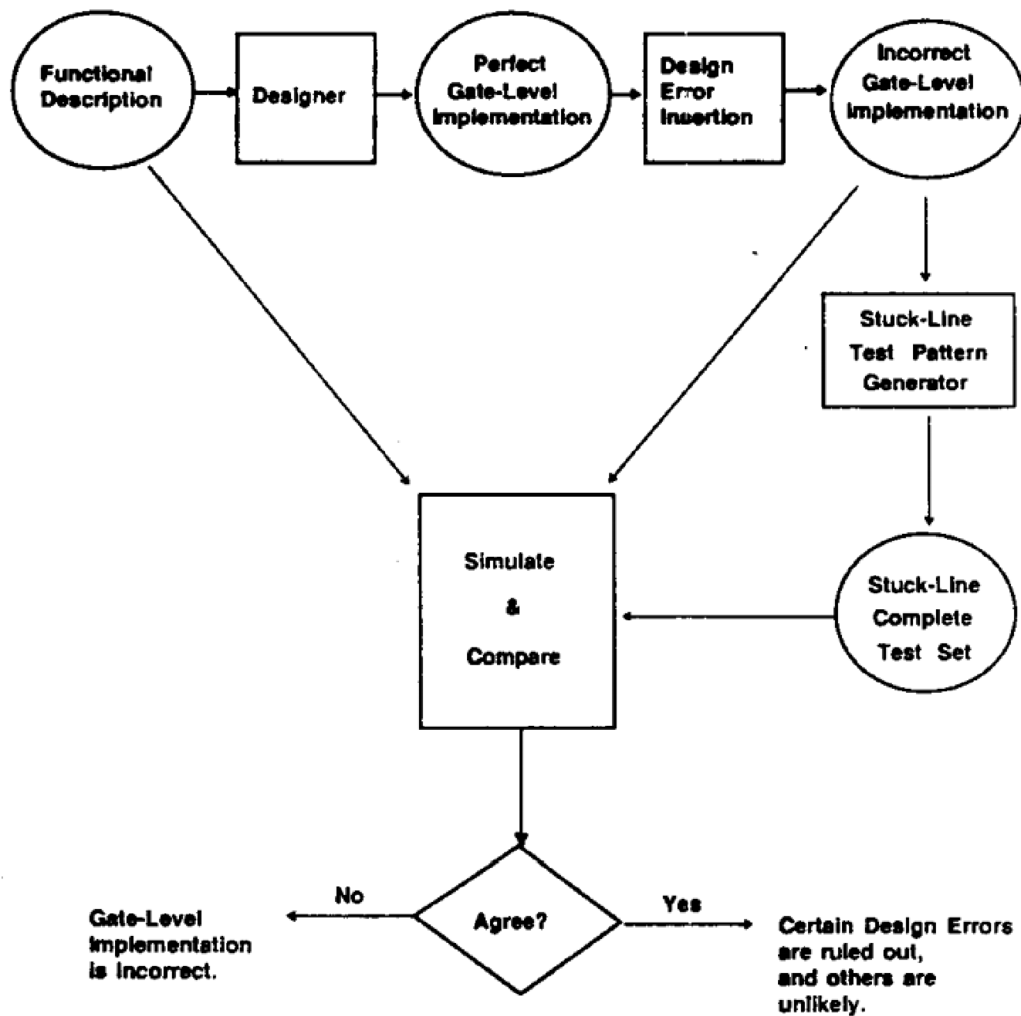
Culminando o conhecimento adquirido e assim utilizando todas as técnicas que se mostraram competitivas, se tem como resultado uma ferramenta de decodificação que funciona com *message passing* e aplica ADMM com uma projeção ultimamente otimizada para garantir a convergência do algoritmo iterativo.

Estas características até então apresentadas foram sintetizadas em lógica programável e, a fim de validar e medir o desempenho do módulo apresentado, serão implementadas em FPGA neste trabalho.

O bloco resultante dessa cadeia de implementação será visto como um acelerador de *Hardware*. Implementações semelhantes foram efetuadas por Samuel Bayliss, Christos-S. Bouganis, George A. Constantinides e Wayne Luk em [10], no qual uma lógica programável resolve um problema de programação linear *Simplex,* explorando o paralelismo parametrizável inerente associado a essa tecnologia. Como resultado do trabalho apresentado, o DUT soluciona o problema cuja entrada é um *Stream* de dados que são salvos em uma FIFO no próprio FPGA.

Na concepção de um circuito integrado, em todos os níveis de abstração, ferramentas de teste e verificação comparam o projeto em diferentes níveis para se ter certeza que o processo de síntese, os projetistas ou mesmo as ferramentas de otimização não introduziram erros, particularmente erros de lógica. Devido à atual complexidade dos projetos e também das ferramentas de síntese, isso se tornou extremamente importante [17]. No âmbito de teste e verificação de uma lógica descrita, a Figura 4 explicita um diagrama do procedimento básico de um projeto digital e sua verificação.

Figura 4 - Diagrama de teste e verificação



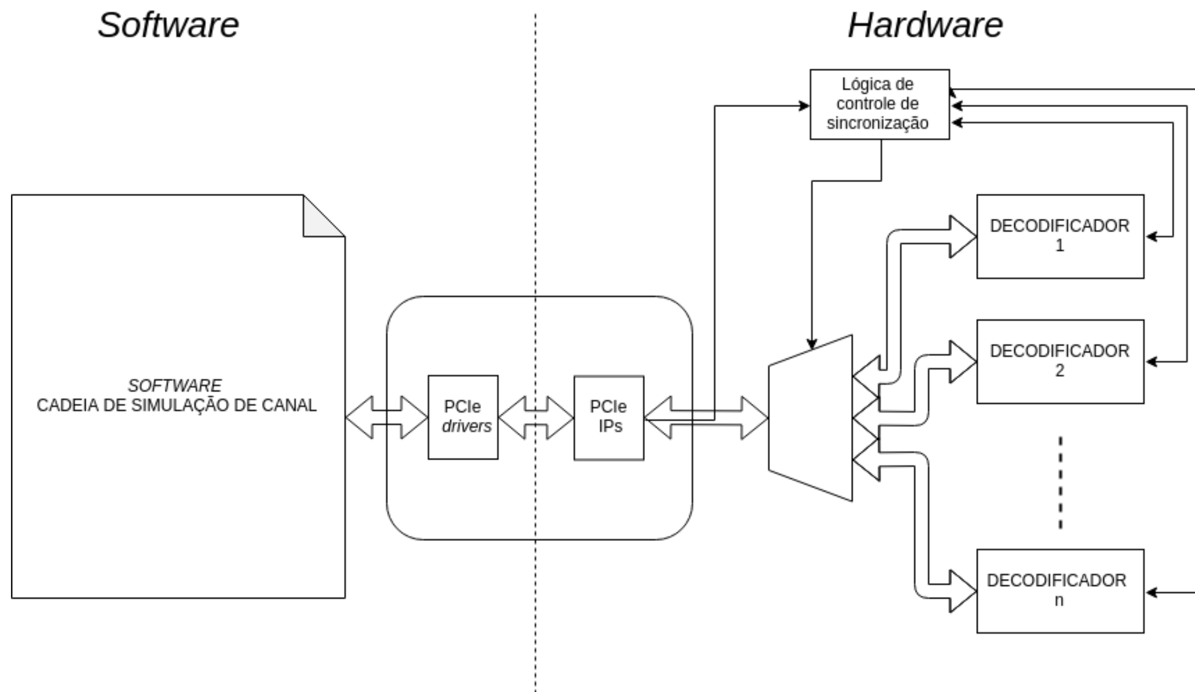
Fonte: ABADIR et al.: Logic Design Verification via Test Generation [16]

Na Figura 4 a primeira etapa é uma descrição funcional original do circuito sendo dada a um projetista. Este, idealmente, produz uma perfeita implementação do circuito. Porém o próximo passo trata de um possível erro sendo inserido na implementação. Na maior parte dos casos são justamente os projetistas que inserem erros de projeto acidentalmente. Na figura 4 este passo é separado para fins de visualização. Testes então são gerados para o possível incorreto circuito e simulados com a especificação funcional e o possível circuito incorreto. Se os resultados da simulação diferirem, isto mostra que realmente o erro de projeto foi inserido. Caso contrário, se eles concordarem, uma certa classe de erros, os erros estruturais de projeto podem não ocorrer, e que provavelmente outras classes de erros, como os de *timing,* também não ocorreram [16].

5. METODOLOGIA

A implementação do bloco de decodificação LDPC em FPGA e sua infra estrutura de testes terá a arquitetura como mostra o diagrama da Figura 5.

Figura 5 - Diagrama de blocos arquitetural do sistema.



Fonte: O autor

Primeiramente, utilizando as ferramentas computacionais já mencionadas, serão efetivadas sínteses lógicas com o projeto de decodificação já desenvolvido previamente. Essas sínteses consistem em modelar a camada de descrição de *Hardware* efetuada pelo usuário e mapeá-la em operações de portas lógicas passíveis de implementação em FPGA, se tendo assim, os módulos chamados de decodificação apresentados na área de *Hardware* da Figura 5. Nessa etapa serão obtidos indicadores como número de elementos lógicos que o módulo unitário demanda. Outro número importante a ser levado em conta nesta etapa é a utilização de blocos de memória RAM, pois é sabido que o algoritmo tem ampla utilização desse recurso. Com esses resultados, e sabendo a capacidade do FPGA a ser utilizado, será possível determinar quantas instâncias do decodificador serão embutidas na lógica programável. Adicionalmente, deverá ser considerada a utilização de recursos da infraestrutura de comunicação com o computador. Essa infraestrutura também é ordenada em hierarquias de desenvolvimento devido à

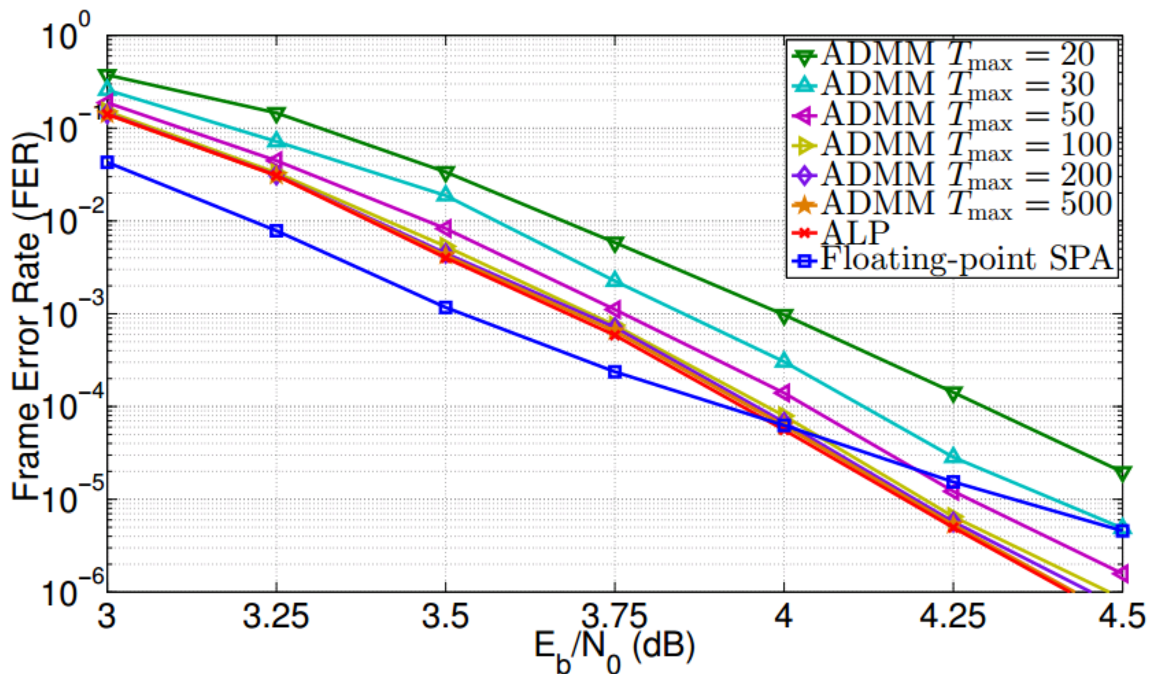
complexidade de sincronização. Neste âmbito, é necessária a implementação de uma máquina de estados para supervisionar os blocos de decodificação enquanto gerencia os dados que chegam pelo método de comunicação proposto.

Opta-se por utilizar o protocolo de comunicação PCIe devido à sua alta capacidade de transmissão de dados. Ferramentas disponíveis que facilitam a usabilidade de *drivers* como *Xillybus* são utilizadas para efetivar o casamento entre o *Software* e o *Hardware* proposto. Esse agregado de soluções envolve tanto rotinas em *Software* como encapsulamento dos complexos IPs de comunicação PCIe. Isso se mostra necessário na remessa de trabalho referente à interface entre os dois domínios.

Serão elaboradas baterias de testes imputando conjuntos de LLRs para cada unidade de decodificador instanciada. A lógica de controle de sincronização garantirá que nem um decodificador fique ocioso durante os testes, fazendo com que a cadeia de simulação possa calcular novas LLR imediatamente depois de acabadas as iterações de um quadro de dados.

Estruturalmente, serão testadas diferentes remessas de dados, cada qual com um padrão de erro menor do anterior, para que se possa elaborar gráficos como o da Figura 6, no qual o bloco de decodificação proposto foi ensaiado em *Software* com uma taxa progressiva de deterioração de canal. As diferentes curvas também mostram a variação de parâmetros internos ao decodificador como número máximo de iterações permitidas explicitadas como Tmax, que varia entre 20 e 500. Além disso, outras duas estratégias foram inseridas no gráfico para fins de comparação. Para quantificar a qualidade de sinal nos testes propostos, usualmente é utilizada a métrica de relação entre energia por bit e potência espectral do ruído. Ela é representada por Eb/N0 e é amplamente utilizada na avaliação de performance de modulações e codificações digitais, pois trata-se de uma medida de *Signal to Noise Ratio* (SNR) normalizada por bit. O uso de gráficos com visualização logarítmica no eixo vertical é mandatório, pois a taxa de erro cai drasticamente conforme a qualidade do canal aumenta.

Figura 6 - Gráfico de resultados esperado para a decodificação LDPC-ADMM com diferentes parâmetros.



Adaptado de: Zhang & Siegel, 2013

No eixo vertical, conceitos como *Bit Error Rate* (BER) e *Frame Error Rate* (FER) serão amplamente utilizados nos testes desse trabalho, pois são a métrica final para quantificar a performance de decodificadores de correção de erros.

5.1 MATERIAIS UTILIZADOS

5.1.1 *SOFTWARE*

Como supracitado, para realização das sínteses lógicas se utilizará os *Softwares* fornecidos pelo fabricante do FPGA, que será Xilinx. O *Software* a ser utilizado é o ISE. O ISE Design Suite é um pacote de *Software* também produzido pela Xilinx para síntese e análise de projetos HDL, com recursos adicionais para sistema em desenvolvimento de chip e síntese de alto nível.

O conglomerado de soluções do *Software* Xillybus será utilizado e tem licença aberta para fins educacionais. O sistema operacional suportado pelo Xillybus se estende para Windows e Ubuntu.

Adicionalmente, no domínio do *Software* será utilizada a cadeia de simulação de canal que consiste em um gerador de *Bitstream*, modulador, simulador de canal, gerador de ruído e demodulador.

5.1.2 *HARDWARE*

A parte física do projeto será composta por um computador e um FPGA. O segundo está contido num kit de desenvolvimento que conta não só com o circuito integrado do FPGA em si, mas também com infra-estrutura de alimentação e também múltiplos periféricos inclusos numa placa. Na placa é possível verificar a presença de conectores de alta velocidade do tipo *Small Form-Factor Pluggable Transceiver* (SFP), amplamente utilizados em conexões de rede, mas que não são utilizados neste presente trabalho. O kit de desenvolvimento também conta com um barramento de pinos com 67 *general pin input output* (GPIO) com pares diferenciais que possibilita a conexão do FPGA com dispositivos externos como conversores analógicos digitais, câmeras, telas, etc...

Dentre essas características adicionais, a plataforma dispõe do barramento PCIe de segunda geração com um canal de comunicação (PCIe Gen2 8x). O FPGA contido no kit é o XC6VLX365T, da família Virtex 6, que contém uma quantia de lógica programável grande o suficiente para conter toda a implementação deste presente trabalho.

Figura 7 - kit FPGA modelo XC6VLX365T.

Fonte: O Autor.

Figura 8 - Kit FPGA inserido no computador utilizado.

Fonte: O Autor.

6. CONTRIBUIÇÕES

É de suma importância possibilitar que a cobertura dos padrões de testes tenham abrangência suficiente para revelar todas as possíveis falhas de descrição e concepção de um bloco IP. Testes como o do presente trabalho, visam exercitar ao máximo a lógica em questão, conferindo os padrões de cada ciclo de testes com as normativas esperadas garantindo que a saída de um determinado bloco IP se comporte como planejado, não apresentando falhas nas situações propostas.

6.1 APLICAÇÃO

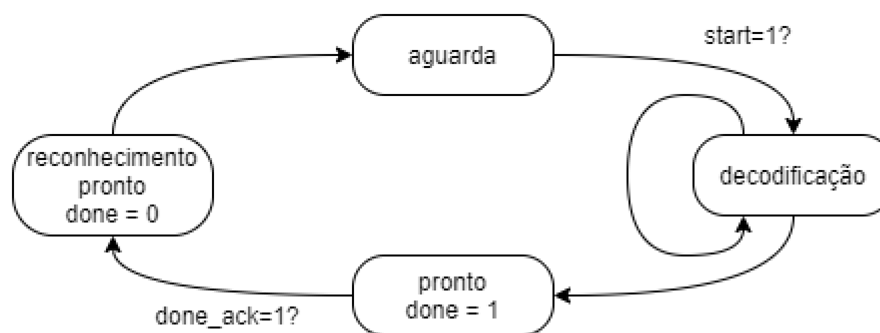
A aplicação desta plataforma de testes não se dá apenas ao bloco IP de decodificação em questão. Mudanças podem ser estabelecidas nos geradores de padrões e os módulos que efetuam a conferência das saídas, fazendo com que qualquer arquitetura que necessite alta velocidade de barramentos possa ser testada e validada com facilidade. Outra vantagem é que esse bloco também pode, por meio de pequenas mudanças, ser utilizado como coletor de dados em aplicações específicas, como interface digital para a leitura e armazenamento de dados de um conversor analógico digital, por exemplo.

7. DESCRIÇÃO DA SOLUÇÃO IMPLEMENTADA

Para efetuar os testes propostos do módulo LDPC, que é a unidade sob teste, deve-se alimentar a sua interface de dados e também de controle e monitorar a saída. Esta exige a conexão da entrada em forma de LLRs supracitada. A saída do bloco é composta por um barramento de bits de largura genérica, obedecendo o número de LLRs que são conectadas na entrada.

Também são necessários alguns bits de controle que garantem o correto funcionamento das máquinas de estado internas ao bloco sob teste. De forma simplificada, o controle do módulo sob teste é dado conforme indica a Figura 9 que contém a descrição de uma máquina de estados finita.

Figura 9 - Máquina de estados simplificada que descreve o controle do módulo sob teste.



Fonte: O autor.

No estado de aguardo o bloco ignora as entradas de dados e as saídas não são válidas. Neste momento o CPU deve, por meio de toda infra estrutura de PCIe, escrever os registradores que armazenam temporariamente os valores das LLRs. Estas devem ser feitas por uma interface semelhante a de memória comum. Acrescendo-se o sinal de endereço, intercambia-se para qual registrador o sinal de dados deve ser redirecionado. Pelo fato dessa escrita ser feita de forma sequencial, cada registrador seguido do próximo, esta etapa toma um determinado número de ciclos de relógio do domínio do PCIe, que é proporcional ao número de LLRs. Quando o CPU termina de escrever todos os índices, é acionado o sinal de *start*. A partir daí, o LDPC entra no estado de decodificação, no qual o mesmo passa a ler os dados de entrada e fazer todas as operações necessárias para efetivar os cálculos matemáticos de acordo com sua arquitetura interna. Esta etapa também demora o

número de ciclos a ser avaliado nesse trabalho. Enquanto isso, os sinais de entrada devem continuar estáveis já que optou-se por não se registrar os dados imputados para poupar recursos de lógica programável. Uma vez que o decodificador termina suas iterações, a máquina principal de controle é direcionada para o estado de prontidão, e isso é informado para o CPU através do sinal de *done*. Concomitantemente as iterações do módulo sob testes, as saídas do mesmo são chaveadas de forma que devem ser desconsideradas, pois são resultantes das operações internas e por consequência, ainda não devem ser levadas em consideração. Novamente isso foi adotado para poupar recursos, já que não registrar a saída de dados economiza o mesmo número de registradores quanto a quantidade genérica de LLRs admitidas. Uma vez que o sinal *done* é comandado, as saídas estipuladas são válidas e serão consideradas estáveis até a confirmação de leitura por parte do CPU. Este procedimento é utilizado para desacoplar as temporizações entre CPU e o módulo sob teste, organizando e sequenciando as escritas e leituras de dados para executar o correto funcionamento de todo o teste. O sistema descrito na lógica programável é direcionado então para o estado no qual é reconhecido que o CPU efetivou suas leituras e está pronto para seguir para próxima carga de LLRs. Neste estado, os sinais de controle são reiniciados de forma a entrar no estado de prontidão garantidamente sem nenhuma pendência. Outro sinal de suma importância no controle de todo o sistema é a indicação de dados válidos que são oriundos do decodificador LDPC. O sinal *valid*, como é chamado ao longo do código, indica ao final de todas iterações do algoritmo, se os dados presentes na saída satisfizeram as condições de término de acordo como descrito no capítulo explicativo do módulo sob teste. Os sinais de controle são intercomunicados entre as duas unidades da mesma forma com a qual os dados são transferidos, tendo como única diferença, o *offset* para um banco de registrador específico, no qual cada *bit* é um sinal de controle. O mesmo é repetido para registradores no sentido CPU para teste e vice-versa.

7.1 ENCAPSULAMENTO DA INTERFACE PCIe: XILLYBUS

Com o único intuito de facilitar a utilização da comunicação por PCIe, fez-se uso das ferramentas de *Hardware* e *Software* da Xillybus. Esse conjunto de códigos visa fornecer simplicidade e eficiência em transações do CPU para a lógica

programável. Transações essas que podem ser via DMA ou simples escrita e leitura de registradores.

No domínio do Software, *drivers* de Linux e Microsoft Windows são disponibilizados para que as escritas no FPGA sejam efetivadas através de simples escritas e leituras a arquivos disponibilizados no sistema operacional, sem precisar envolver uma API para tal tarefa [18].

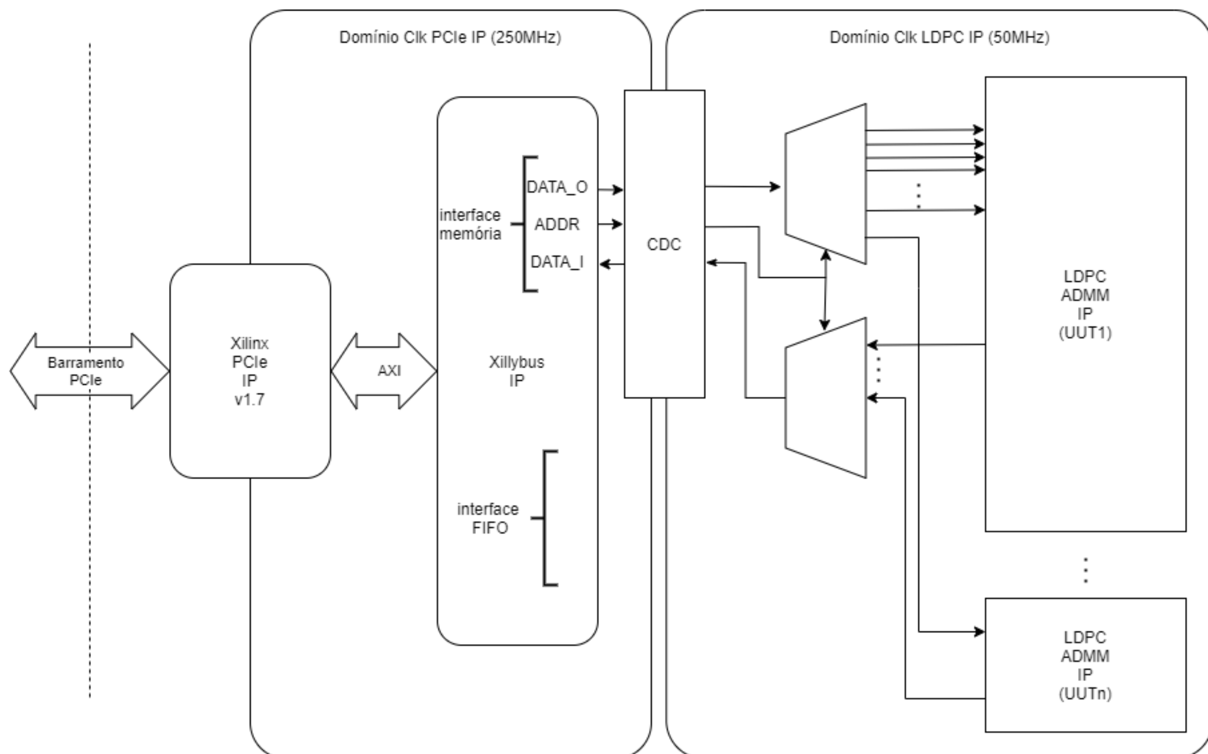
Já no domínio da lógica programável, os dados são disponibilizados em forma de FIFO para transações para DMA ou em forma de memória, contando com sinais como endereço, saída e entrada de dados e também habilitação de escrita e leitura.

Todo esse conjunto de ferramentas é disponibilizado gratuitamente ao usuário final e conta com uma boa documentação e uma comunidade responsiva que torna essa uma solução interessante para não só simples, como também complexos projetos. Optou-se por utilizar a interface de memória para garantir a escalabilidade do projeto, já que adicionar endereços de registradores é extremamente simples, bastando conectar esses registradores adicionados a um novo bloco de decodificação para se contar com mais um acelerador.

7.2 CRUZAMENTO DE DOMÍNIOS DE CLOCK ASSÍNCRONOS

Por definição de prioridades de projeto, a concepção do decodificador não foi otimizada para operação em altas frequências. Os caminhos críticos são longos devido a grandes comparações e extensas cadeias de lógica combinacional. Tendo isso em vista, a escolha da frequência para operação no FPGA utilizado foi de 50MHz. Contudo, os blocos de PCIe e os blocos do Xillybus trabalham numa frequência superior, que é 250MHz, para dar conta do fluxo de dados que o PCIe proporciona. Isso acarreta problemas de cruzamento de domínio de *clock* ou *cross domain clock* (CDC). A quantidade de dados não é necessariamente um problema neste caso devido ao obedecimento dos sinais de controle acima, sendo as escritas e leituras de forma desacoplada e as rajadas ou *bursts* efetuados nas leituras de registradores pré disponibilizados. A Figura 10 demonstra a localidade de cada instância e também qual domínio os blocos se encontram.

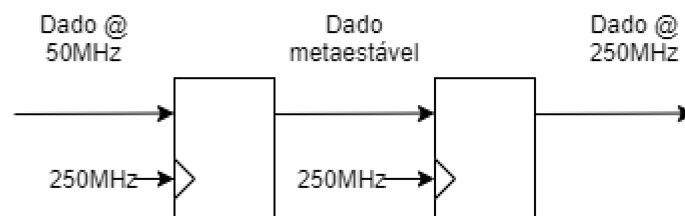
Figura 10 - Detalhamento das instâncias e do CDC no sistema.



Fonte: O autor.

Como pode ser observado, todos os blocos responsáveis pela comunicação PCIe estão instanciados no domínio de 250MHz. Já os módulos de teste encontram-se na seção cuja frequência é substancialmente mais baixa para se adequar ao período necessário do LDPC. Um módulo de CDC se faz necessário entre os domínios para efetivar a tradução de um domínio para o outro. Problemas de CDC são comuns no âmbito de desenvolvimento de *Hardware* para lógica programável entre múltiplos circuitos síncronos desacoplados. A solução para cruzamentos de domínios mais lentos para rápidos é efetuada conforme esquematizado na figura 11.

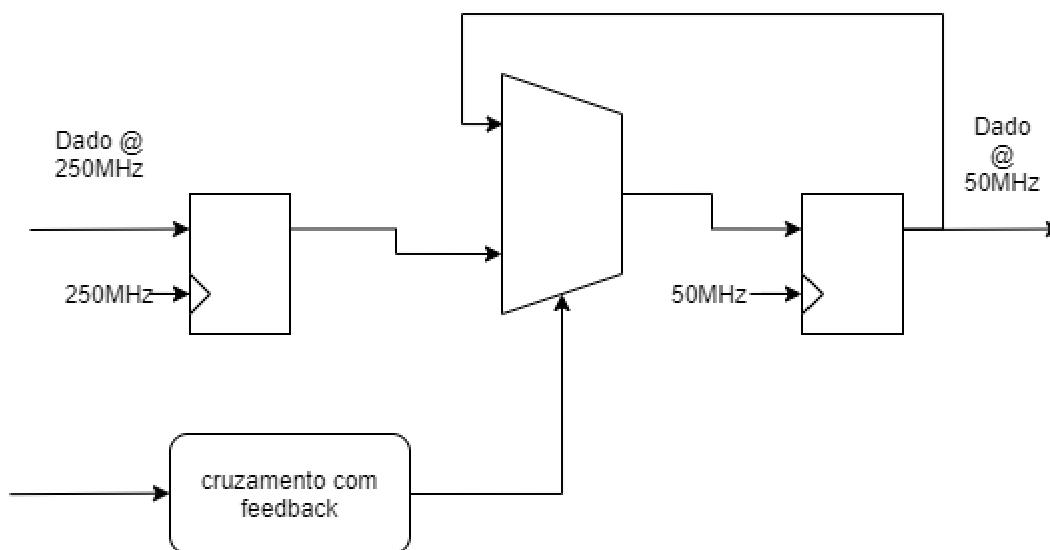
Figura 11 - Estratégia de CDC para cruzamento domínio lento para rápido.



Fonte: O autor.

Já o cruzamento do domínio rápido para o lento tem uma complexidade elevada em relação ao caso anterior. A solução é dada pelo cruzamento do sinal de controle que, no domínio lento, indica a disponibilidade de sinais de dados válidos. O sinal de controle só é válido quando o seu retorno anterior é conferido. O passo de *feedback* mostrado na parte inferior da Figura 12 utiliza a estratégia descrita no parágrafo anterior, a qual expõe o cruzamento do domínio lento para o rápido. Só então, os dados são registrados no domínio mais lento. Vale ressaltar que essa operação só é válida para sinais que são cambiados esporadicamente, que é o caso nesse escopo. A arquitetura consta na figura 12.

Figura 12 - Estratégia de CDC para cruzamento do domínio rápido para lento.



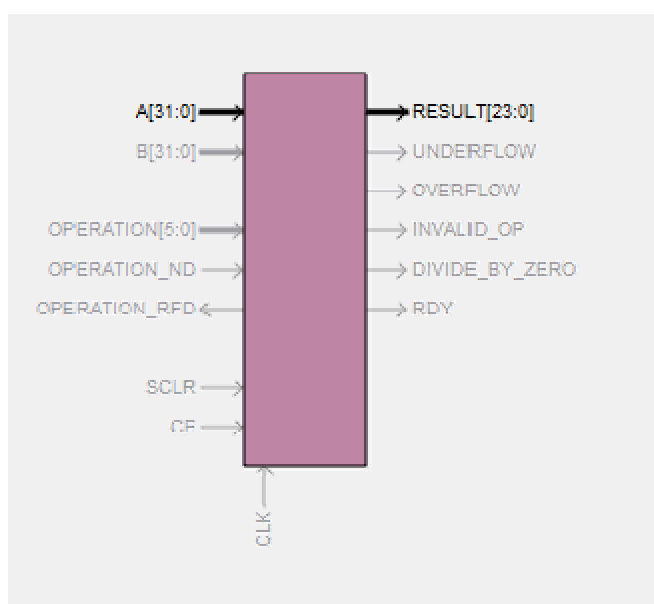
Fonte: O autor.

7.3 CONVERSÃO PONTO FLUTUANTE PARA PONTO FIXO

A cadeia de simulação de canal realizada no CPU dispõe de variáveis representadas em ponto flutuante, algumas até com dupla precisão. As LLRs são manipuladas em *Software* com operações em ponto flutuante de 32 bits. Como determinado no capítulo de encaminhamento da solução. O decodificador foi concebido por definição de projeto para lidar com a representação em ponto fixo. Isso acarreta uma diferença que deve ser considerada na implementação.

Tentativas para executar a conversão de ponto flutuante para ponto fixo em uma rotina de *Software* , para transmitir os dados pelo PCIe já na representação correta, foram elaboradas, mas que apresentaram problemas dificilmente resolvíveis. Optou-se então, para resolver esse problema, pela utilização de um bloco IP da Xilinx LogiCORE IP Floating-Point Operator v5.0. Uma das capacidades desse módulo é realizar a conversão de ponto flutuante para ponto fixo com precisões customizáveis, tornando-o ideal para solucionar o problema de diferença de representação.

Figura 13 - Interfaces do conversor de ponto flutuante para ponto fixo após as customizações.



Fonte: O autor.

Como se pode perceber na figura 13, o sinal de relógio não está presente na interface do módulo. Isso é possível pois a conversão de ponto flutuante para ponto fixo pode ser resolvida totalmente com lógica combinacional. Vários outros sinais estão inativos no estado atual devido à generalidade deste IP. Basicamente todas as operações aritméticas em ponto flutuante podem ser realizadas utilizando esse mesmo módulo, mas com customizações diferentes. Versões mais novas deste dispositivo também contam com um barramento de conexão de processadores e outros periféricos *Advanced eXtensible Interface* (AXI) da Arm.

8. RESULTADOS E EXPERIMENTOS

8.1 RELATÓRIOS DA IMPLEMENTAÇÃO EM FPGA

A implementação de somente um decodificador LDPC e de toda infraestrutura de testes ocupou os recursos do FPGA conforme constam no quadro 1.

Quadro 1 - Relatório de utilização do projeto referente ao FPGA XC6VLX365T.

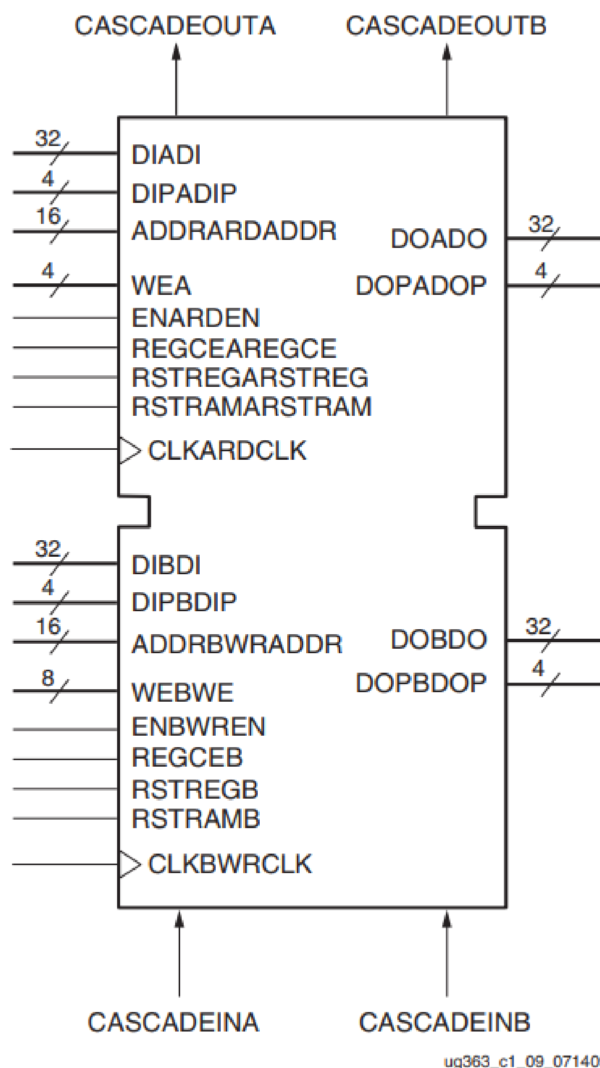
Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	7,909	455,040	1%
Number used as Flip Flops	7,200		
Number used as Latches	2		
Number used as AND/OR logics	707		
Number of Slice LUTs	14,501	227,520	6%
Number used as logic	12,949	227,520	5%
Number using O6 output only	8,419		
Number using O5 output only	187		
Number using O5 and O6	4,343		
Number used as Memory	1,298	66,080	1%
Number used as Dual Port RAM	244		
Number using O6 output only	148		
Number using O5 output only	1		
Number using O5 and O6	95		
Number used as Shift Register	1,054		
Number using O6 output only	1,054		
Number used exclusively as route-thrus	254		
Number with same-slice register load	202		
Number with same-slice carry load	52		
Number of occupied Slices	4,326	56,880	7%

Number of LUT Flip Flop pairs used	15,294		
Number with an unused Flip Flop	8,189	15,294	53%
Number with an unused LUT	793	15,294	5%
Number of fully used LUT-FF pairs	6,312	15,294	41%
Number of unique control sets	414		
Number of slice register sites lost to control set restrictions	1,181	455,040	1%
Number of bonded IOBs	17	600	2%
Number of LOCed IOBs	11	17	64%
Number of bonded IPADs	4		
Number of bonded OPADs	2		
Number of RAMB36E1/FIFO36E1s	12	416	2%
Number using RAMB36E1 only	12		
Number using FIFO36E1 only	0		
Number of RAMB18E1/FIFO18E1s	5	832	1%
Number using RAMB18E1 only	5		
Number of GTXE1s	1	20	5%

Fonte: O autor.

Podemos avaliar a pequena impressão utilizada do decodificador no dispositivo. Destaca-se que somente 6% das *Look-Up Tables* (LUTs) foram empregadas no dispositivo e também o grande uso de memórias RAM internas ao FPGA. Ao total, 12 unidades de RAMB36E1 foram utilizadas. Nos circuitos integrados da família Virtex 6, as *block RAMs* são inferidas quando um grande vetor de dados é utilizado na descrição lógica respeitando escritas e leituras não concomitantes. Outra possibilidade dessas memórias serem utilizadas no projeto é instanciando seu bloco primitivo que tem a interface como na figura 14. A preferência para utilização de memórias RAM no decodificador foi proposital devido a definições de projeto.

Figura 14 - Interfaces da *block RAM* em dispositivos da família Virtex 6.



Fonte: Adaptado de [15] Virtex-6 FPGA Memory Resources UG363 - Xilinx

Observa-se também no Quadro 1, a utilização de somente um transceptor GTX, que é responsável pela comunicação PCIe de segunda geração. Apesar das 8 linhas disponíveis no kit de desenvolvimento, para simplificar o roteamento e facilitar adequação das restrições de temporização dos caminhos do sistema, optou-se por utilizar apenas uma via PCIe. Dada a grande capacidade de transferência de uma única linha PCIe, e o tempo de decodificação ser muito superior ao das transferências de dados, isso não foi um fator limitante nos experimentos. Por não se tratar de um kit de desenvolvimento oficial e ser uma placa com pouca documentação e informação disponível, foi necessário elaborar uma customização do bloco padrão do PCIe, encapsulado pelos módulos da Xillybus, que é voltado para funcionar na placa ML605, da Xilinx. A diferença está nos pinos diferenciais

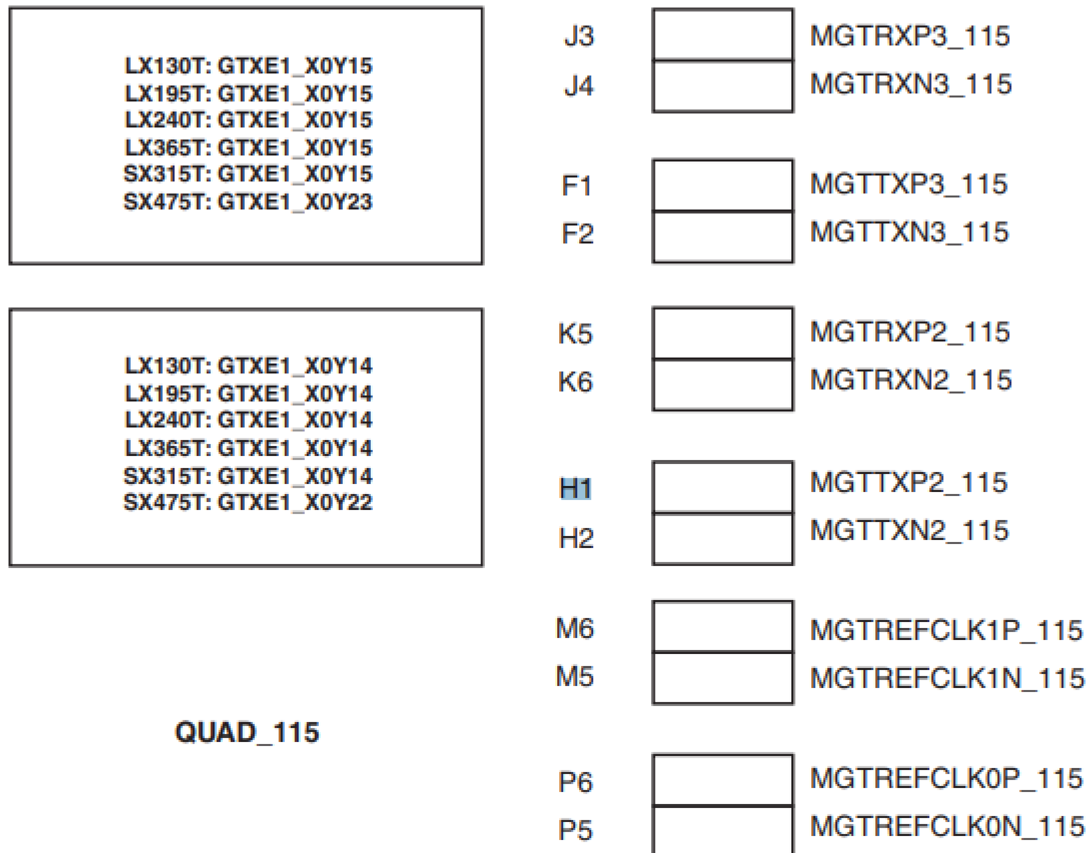
conectados às trilhas de PCIe de dados e relógio. É somente informado que os pinos do pcie têm as seguintes conexões.

Quadro 2 - Documentação de conexões dos pinos PCIe no kit de desenvolvimento utilizado.

Função	Pino FPGA	Relógio a ser mapeado	Transceptor
PCIE_RX0	J3	p: J3 MGTRXP3_115 n:J4 MGTRXN3_115	GTXE1_X0Y15
PCIE_RX1	K5	p: K5 MGTRXP2_115 n:K6 MGTRXN2_115	GTXE1_X0Y14
PCIE_TX0	F1	p: F1 MGTTXP3_115 n:F2 MGTTXN3_115	GTXE1_X0Y15
PCIE_TX1	H1	p: H1 MGTTXP2_115 n:H2 MGTTXN2_115	GTXE1_X0Y14
PCIE_CLKP	P6	MGTREFCLK0P_115	
PCIE_CLKN	P5	MGTREFCLK0N_115	

Fonte: O autor.

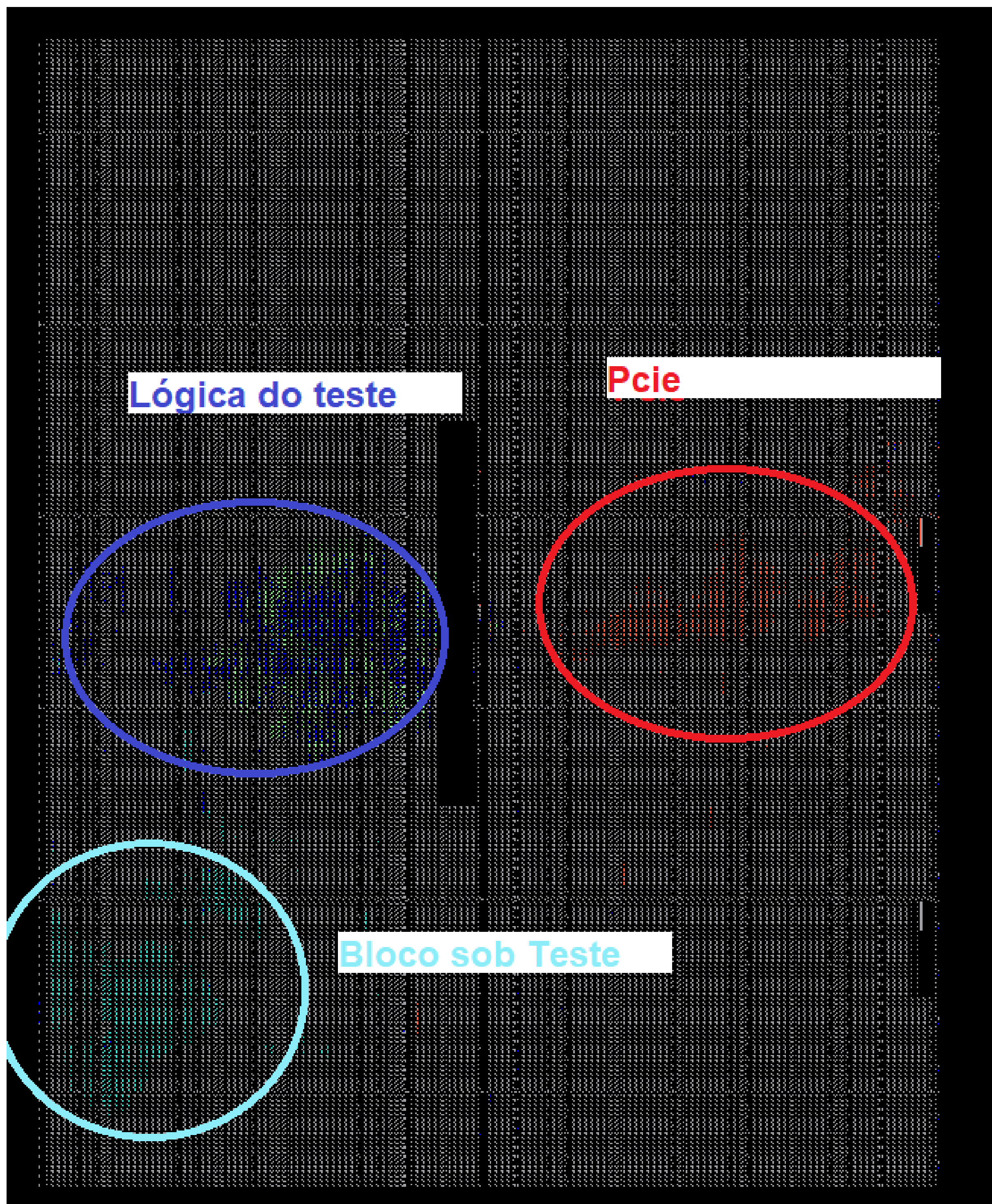
Figura 15 - Mapeamento dos pinos exteriores em questão com os transceptores GTX de dispostivos da família Virtex 6.



Fonte: Adaptado de [14] Virtex-6 FPGA GTX Transceivers UG366 - Xilinx

Conforme consta na documentação do kit de desenvolvimento, as restrições foram escritas para que o projeto seja sintetizado utilizando os transceptores corretos juntamente com os respectivos domínios de relógio. Todos os outros elementos do projeto não precisaram ser restringidos fisicamente, de forma que as ferramentas de síntese, posicionamento e roteamento possam decidir a melhor posição dos elementos de lógica internos ao FPGA. A Figura 16 mostra o resultado do posicionamento dos blocos internamente ao circuito integrado.

Figura 16 - Relatório de posicionamento dos elementos dentro do FPGA.



Fonte: O Autor

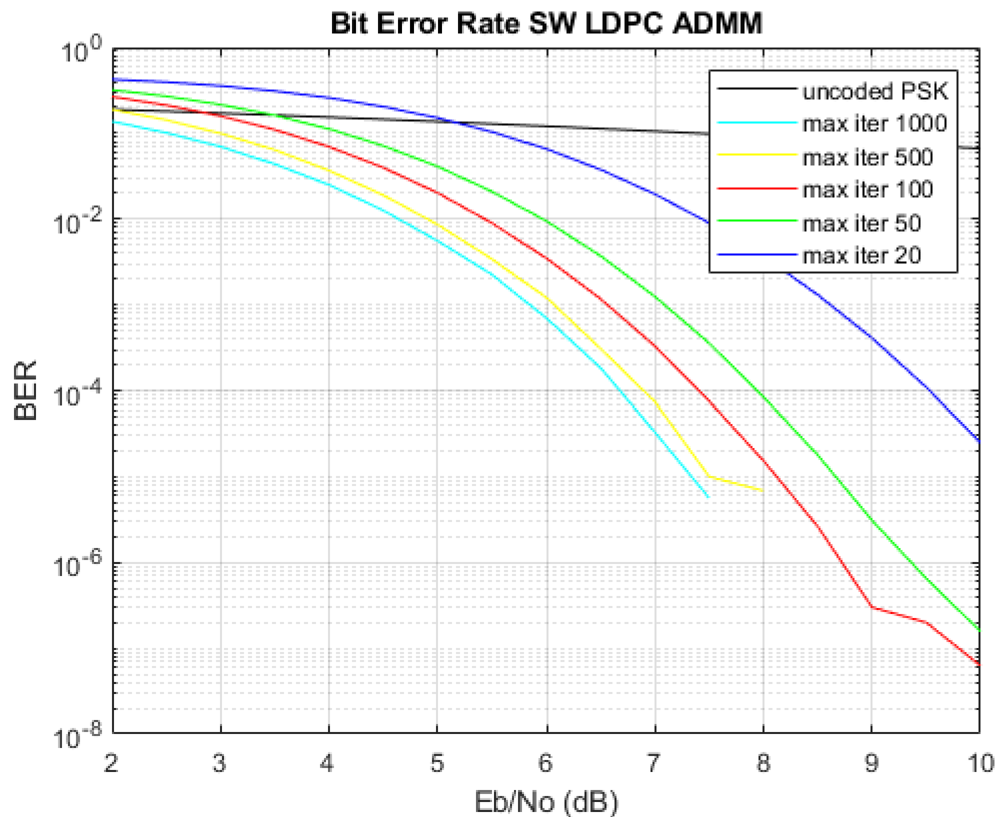
Na figura 16, podemos observar os elementos em vermelho que fazem parte da lógica de PCIe, os azuis escuros da infraestrutura do teste, a lógica de cruzamento de relógio em verde e os azuis claros o decodificador LDPC em

questão. Nota-se que os blocos foram posicionados de forma esparsa. É sabido que quanto maior são os caminhos entre os blocos, mais difícil é alcançar as metas de temporização. Isso pode acontecer devido a balanceamento de consumo de corrente no circuito integrado, já que se todo o projeto estivesse restrito a uma pequena parte do FPGA, o calor emitido por todos os transistores formaria uma área pontual de alta temperatura. Com os blocos devidamente separados, a emissão de calor é equalizada, evitando pontos quentes no circuito integrado e assim aumentando a vida útil do mesmo.

8.2 COMPARAÇÃO DO MODELO EM *SOFTWARE*

Primeiramente, para validar a lógica descrita do decodificador LDPC, foram executados os modelos de decodificador realizados em *Software* de forma a se utilizar como base para os próximos testes. Além de variar a deterioração do sinal no eixo horizontal de Eb/N0 2 dB até 10 dB com passo de 0.5 dB, foi explorado o número máximo de iterações permitidas, pois é um fator limitante na latência média de decodificação. Este também é um parâmetro genérico presente na implementação em lógica programável, já que trata-se somente de um contador interno ao decodificador.

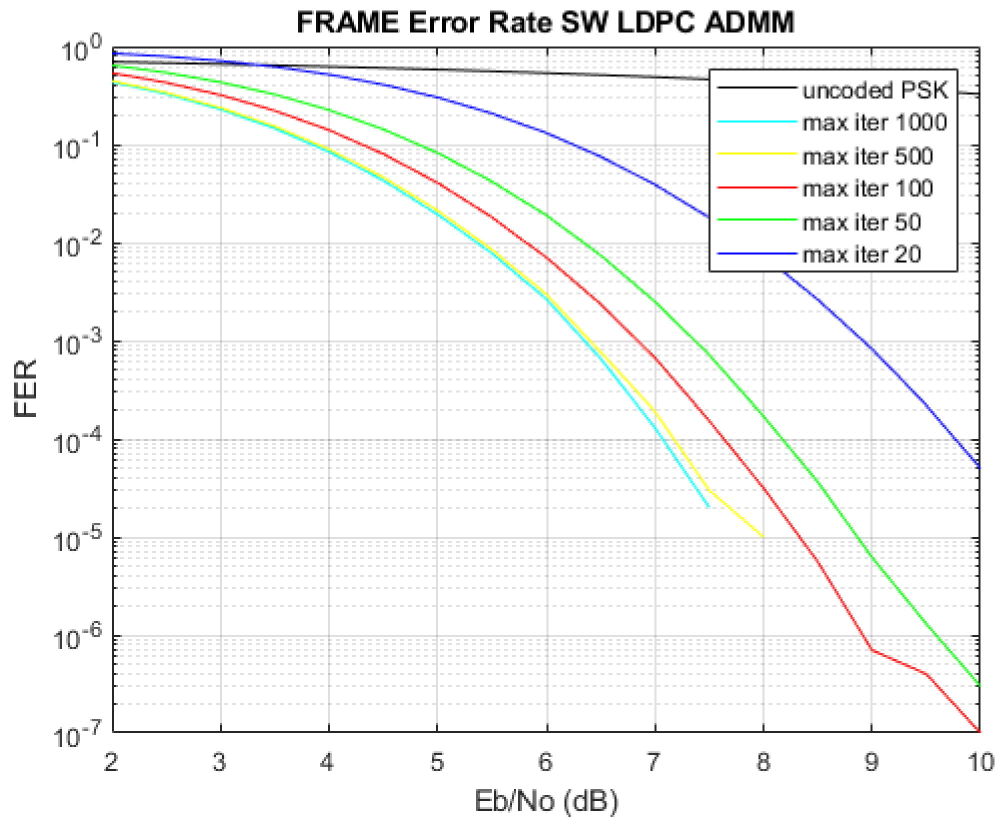
Figura 17 - Curvas características da taxa de erro de *bit* do decodificador em *Software*.



Fonte: O autor.

Observa-se que entre 20 a 100 como número máximo de iterações, foram exercitados 10 milhões de quadros transmitidos, cada quadro contendo 32 bits gerados para cada passo de deterioração do canal. Já para 500 e 1000 como máximo de iterações, limitou-se o número de quadros testados para 1 milhão, devido ao elevado tempo de simulação computacional. Isso acarreta no corte prematuro do gráfico para estes parâmetros devido a perda de potência estatística para Eb/N0 melhor que 7.5. Além de colher a informação da quantidade de erros na mensagem, é possível registrar também quantos quadros foram decodificados erroneamente. É natural que a taxa de quadros com erro seja maior do que a taxa de erros em bit, pois basta somente um bit errôneo para que o pacote inteiro seja considerado problemático. Na Figura 18 é possível conferir a taxa de erros de quadro conforme a variação de qualidade de sinal e número máximo permitido de iterações no decodificador LDPC.

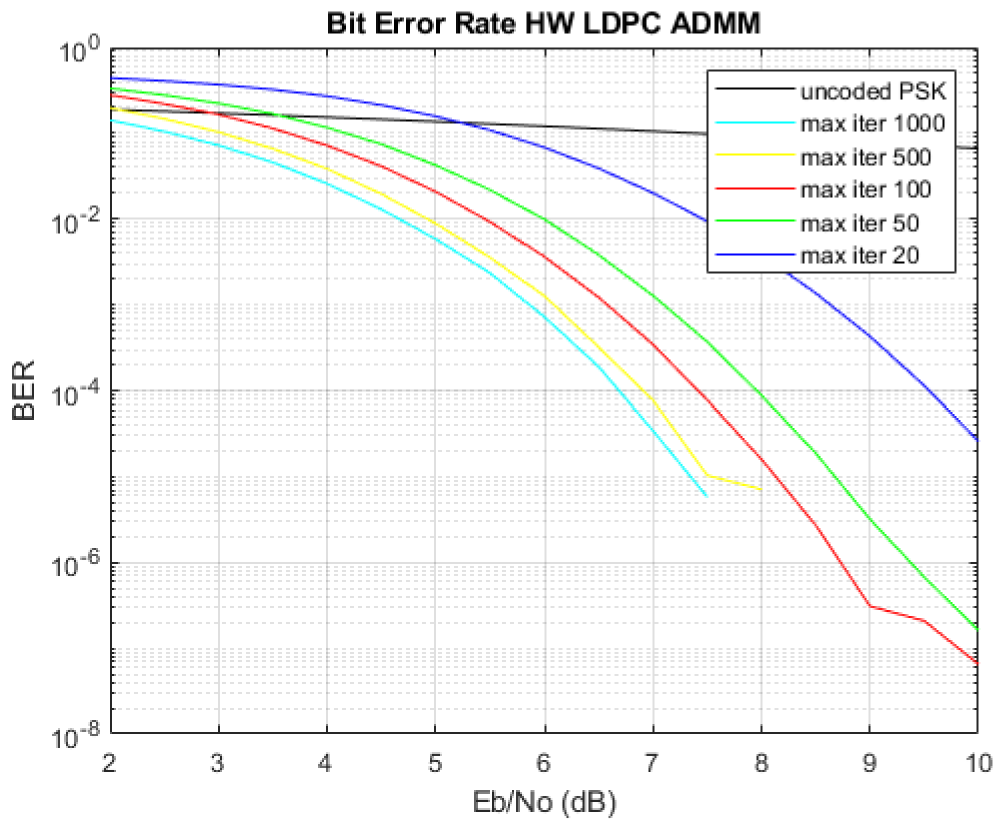
Figura 18- Curvas características da taxa de erro de quadros do decodificador em *Software*.



Fonte: O autor.

Com todos os blocos instanciados e devidamente conectados, e fazendo com que a cadeia de *Software* desviasse as LLRs para o destino correto através de escritas aos registradores físicos da FPGA, deu-se início aos testes de simulação de canal utilizando o decodificador LDPC implementado fisicamente. Obteve-se os seguintes gráficos da resposta de taxa de erros de *bit* do decodificador em *Hardware*, variando-se a qualidade do sinal e o número máximo de iterações.

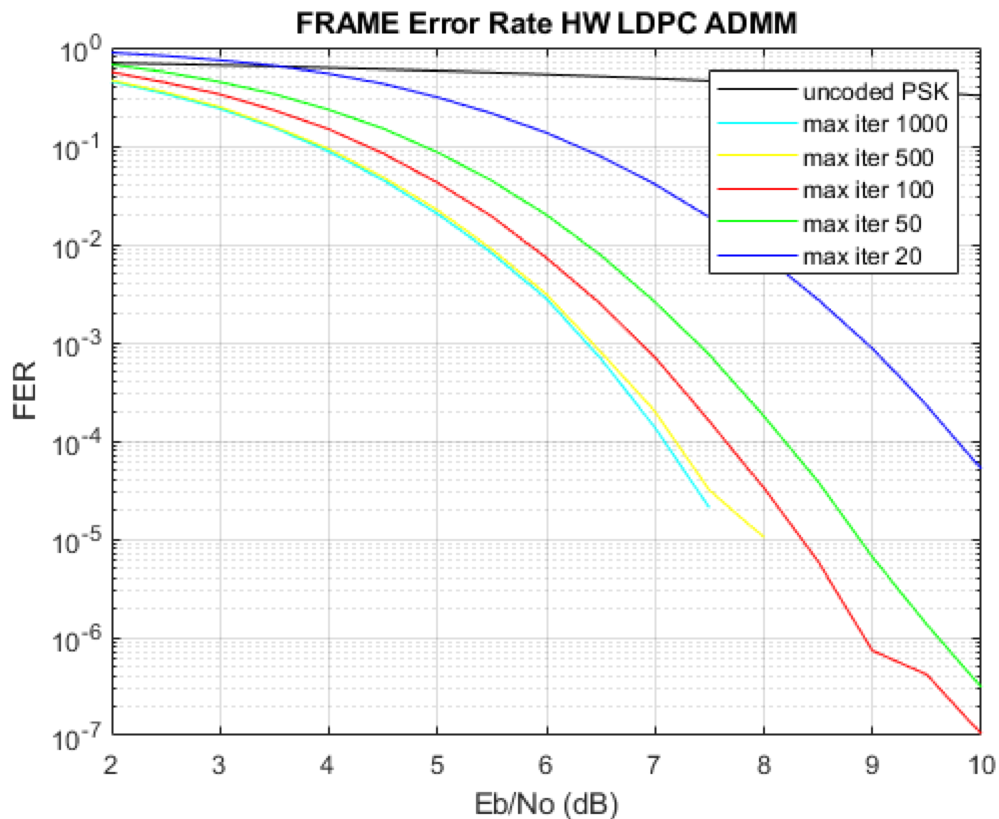
Figura 19 - Curvas características da taxa de erro de *bit* do decodificador em *Hardware*.



Fonte: O autor.

A mesma ferramenta foi utilizada para colher os dados de taxa de erros em bit e em quadros, portando é disposta, na figura 20, a informação de taxa de erros por quadro utilizando o decodificador em FPGA.

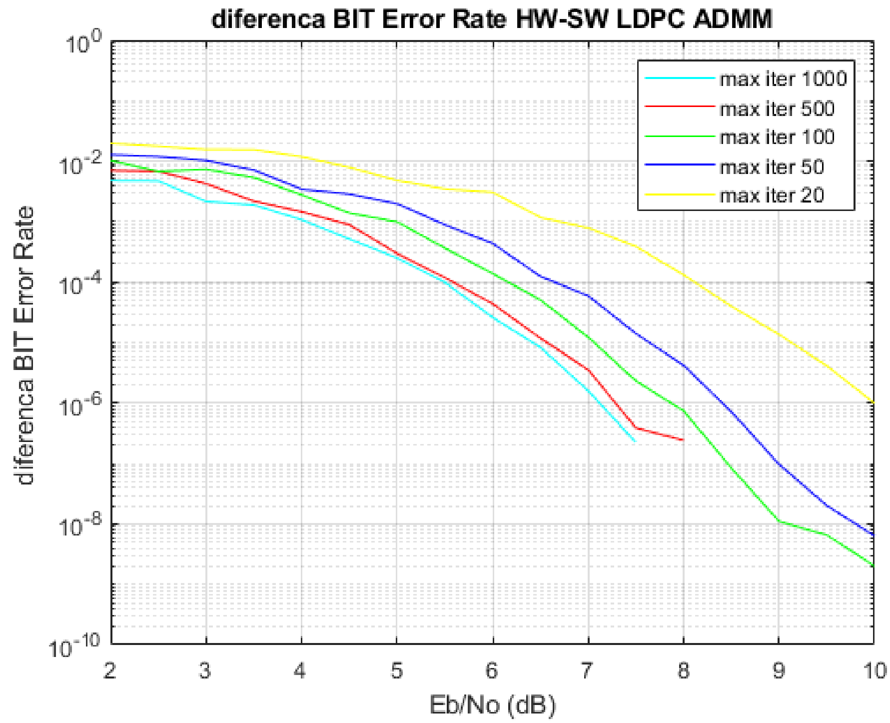
Figura 20 - Curvas características da taxa de erro de quadro do decodificador em *Hardware*.



Fonte: O autor.

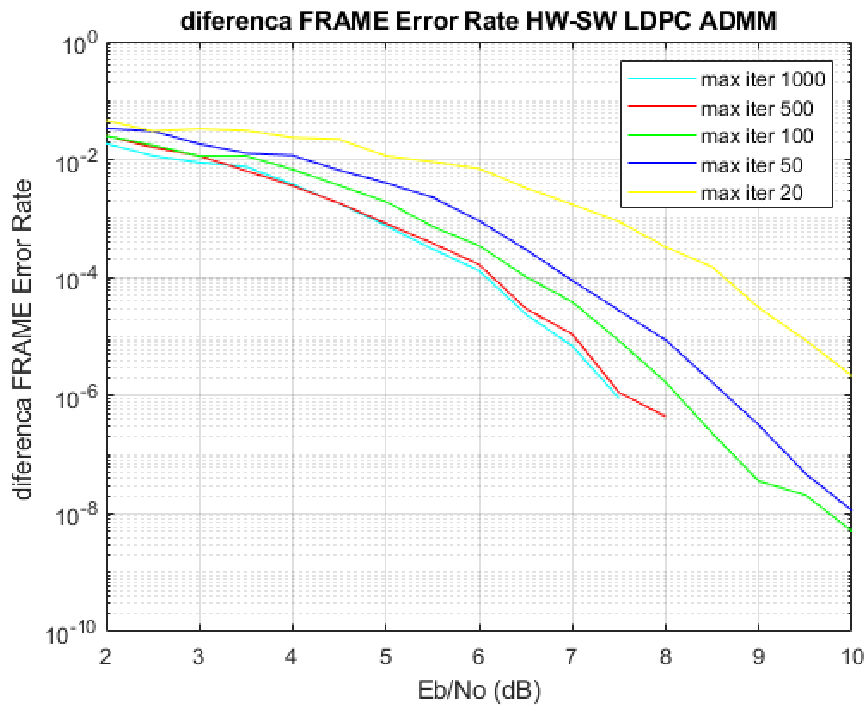
É possível observar que as curvas de taxa de erros do decodificador modelo em *Software* têm grande semelhança com o elemento sob teste em *Hardware*. Isso indica que ambas implementações apresentam um comportamento similar e que o teste apresenta funcionamento coerente. Porém, ao se investigar mais a fundo os dados gerados pela cadeia de simulação de canal, é possível detectar algumas não conformidades. Nas figuras 21 e 22 foram subtraídas a taxa de erro de cada ponto do eixo horizontal para melhor visualização e entendimento do comportamento dessa anomalia.

Figura 21 - Diferenças da taxa de erro de *bit* do decodificador em *Software* e *Hardware*.



Fonte: O autor.

Figura 22 - Diferenças da taxa de erro de quadro do decodificador em *Software* e *Hardware*.



Fonte: O autor.

Percebe-se que a taxa de erros de bit e de quadro do decodificador em *Hardware* ficou superior à do modelo em *Software* em todos os pontos de qualidade do sinal. Particularmente, as diferenças são mais expressivas quando a relação de energia de bit com nível de ruído é baixa, temos um BER cerca de 2% maior no decodificador em *Hardware* quando Eb/N0 equivale a 2dB. Essa diferença é mitigada quanto menor a densidade de ruído.

Essa anomalia pode ser justificada pela representação numérica utilizada no decodificador. Particularmente, a diferença de precisão entre ponto flutuante e ponto fixo do *Software* e *Hardware* respectivamente, causa a discrepância nas taxas de erro observadas.

Conforme definições de projeto, o decodificador em *Hardware* utiliza representação numérica em ponto fixo com um total de 20 bits fracionários e mais 4 bits para representar a parte inteira, acarretando uma resolução máxima de $E_{ponto\ fixo} = 2^{-20} = 0,95\ x\ 10^{-6}$ que é muito maior do que o mínimo valor representável em ponto flutuante de 32 bits, utilizado no *Software* . Essa diferença interage no cálculo iterativo do decodificador acarretando uma perda de desempenho que deve ser levada em conta no dimensionamento do *Hardware*. Mesmo assim, com 20 bits de ponto fracionário e 24 de precisão total, essas diferenças encontradas são aceitáveis para o decodificador em questão, que visa uma ocupação de área e potência reduzida. Certamente seria necessária uma quantidade de lógica muito maior caso fosse desejada a utilização da representação de ponto flutuante nessa arquitetura de decodificação.

9. CONCLUSÃO

Os testes evidenciaram o correto funcionamento da lógica descrita em FPGA do IP sob teste. As comparações entre as saídas do modelo em *Software* e as respostas do bloco em *Hardware* demonstraram coesão dentro da margem esperada para diferenças devido a desigualdade da representação numérica, que com ponto fixo de 24 *bits* de precisão versus ponto flutuante, não passaram de 2% no pior caso.

Para elaborar o teste de um módulo em lógica programável, utilizando as estratégias de co-simulação, um grande esforço se mostra necessário. A começar pelas interfaces do IP sob teste, que se corretamente dimensionadas para esse tipo de aplicação, tornariam o processo de teste mais simples. A exemplo disso são as interfaces que tiveram que ser implementadas neste trabalho. Uma simples memória FIFO poderia ser instanciada entre os blocos de PCIe e o decodificador se caso o último estivesse apto a receber um *stream* de dados. Todavia, por definição de projeto, a implementação se deu com registradores na entrada que demandam um quadro de LLRs inteiro para aí começar a decodificação. A estratégia implementada visa a conexão com um bloco de demodulação como antecessor na cadeia de dados. Como resultado disso, toda a parte inicial do ambiente de teste deve emular um demodulador.

Além de obedecer as interfaces, as diferenças de representação numérica devem ser respeitadas. Blocos de lógica programável que executam as conversões se mostraram de grande valia.

Os cruzamentos de domínio de *clock* são origem para muitos problemas de projeto por falha na sincronização de dados. Atenção teve que ser dada no correto cruzamento dos sinais e coordenar todas as etapas da decodificação tendo a parte de controle num domínio e o decodificador em outro.

Múltiplas instâncias do bloco sob teste podem ser instanciadas utilizando a arquitetura em *Hardware* proposta, conforme consta nos capítulos de metodologia. Porém, para que todos os blocos sejam excitados simultaneamente, a etapa de *Software* também deve acompanhar esta arquitetura e gerar as LLRs de forma independente para cada decodificador. A cadeia em de simulação de canal permite apenas a geração de um quadro de LLRs quando a análise de outro encontra-se pronta. Isso impossibilitou que vários decodificadores pudessem ser utilizados

concomitantemente. A solução disso é facilmente um trabalho futuro na qual a implementação de uma arquitetura não bloqueante em *Software* é efetivada. Isso só não foi executado neste trabalho pois foge do escopo em questão que visa os testes do decodificador, e não aumento de performance no teste. Caso se deseje o último, não só a implementação não bloqueante poderia ser implementada, como também buscar maiores frequências de operação do DUT.

Outro trabalho futuro que pode otimizar a resposta do decodificador em relação a área em silício que o mesmo ocupa, é o estudo matemático do quanto a precisão da representação numérica em ponto fixo influencia nas curvas de qualidade do sinal versus taxa de erros. Uma implementação prática pode utilizar essa mesma arquitetura de teste para averiguar os resultados.

O uso de ferramentas que encapsulam os blocos de PCIe fornecidos pela Xilinx como a utilizada neste trabalho, facilitam a etapa de implementação e reduzem o tempo de desenvolvimento, sem necessariamente, desenvolverem perda de performance na capacidade de transação de dados como também latência.

10. REFERÊNCIAS

[1] R. Irving and G. Solomon, “Polynomial codes over certain finite fields,” J. of the Society for Industrial and Applied Mathematics, vol. 8, no. 2, pp. 300–304, 1960

[2] HELMLING, Michael - Advances in Mathematical Programming-Based Error-Correction Decoding

[3] J. Feldman, D. R. Karger, and M. Wainwright. “Linear programming-based decoding of turbo-like codes and its relation to iterative approaches”. In: Proceedings of the 40th Annual Allerton Conference on Communication, Control and Computing. Monticello, IL,2002, pp. 467–477.

[4] J. Feldman. “Decoding error-correcting codes via linear programming”. PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 2003.

[5] J. Feldman, M. J. Wainwright, and D. R. Karger. “Using linear programming to decode binary linear codes”. IEEE Transactions on Information Theory 51.3 (Mar. 2005), pp. 954–972. doi: 10.1109/TIT.2004.842696. url: www.eecs.berkeley.edu/~wainwrig/Papers/FelWaiKar05.pdf

[6] X. Zhang e P. Siegel - Efficient Iterative LP Decoding of LDPC Codes with Alternating Direction Method of Multipliers

[7] Mitchell Wasson - Hardware-Based Linear Program Decoding with the Alternating Direction Method of Multipliers

[8] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” Foundations and Trends in Machine Learning, vol. 3, no. 1, pp. 1–122, 2011

[9] S. Barman, X. Liu, S. Draper, and B. Recht, “Decomposition methods for large scale LP decoding,” Proc. 46th Allerton Conf. Commun., Control, Computing, Monticello, IL, Sep. 2011, pp. 253–260.

[10] Samuel Bayliss, Christos-S. Bouganis, George A. Constantinides, Wayne Luk - “An FPGA Implementation of the Simplex Algorithm” - 2006 IEEE International Conference on Field Programmable Technology

[11] R. W. Hamming, “Error detecting and error correcting codes,” Bell Syst. Tech. J., vol. 29, no. 2, pp. 147–160, 1950.

[12] R. G. Gallager, “Low-density parity-check codes,” IRE Trans. Inf. Theory, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[13] F. Gensheimer, t. Dietz, K. Kraft, S. Ruzika, N. Wehn - “Reduced-Complexity Projection Algorithm for ADMM-based LP Decoding”

[14] Virtex-6 FPGA GTX Transceivers UG366 - Xilinx

[15] Virtex-6 FPGA Memory Resources UG363 - Xilinx

[16] Abadir, M. S., Ferguson, J., & Kirkland, T. E. (1988). Logic design verification via test generation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 7(1), 138–148.

[17] Sy-Yen Kuo. Locating Logic Design Errors via Test Generation and Don’t-Care Propagation

[18] Xillybus Ltd - Version 2.20 - Getting started with the FPGA demo bundle for Xilinx