Transponder x10Giga

Sumário

[1. Descrição Geral 2](#_Toc225857011)

[2. Módulos internos 4](#_Toc225857012)

[2.1. Serializador / Deserializador 4](#_Toc225857013)

[2.2. Alinhador 5](#_Toc225857014)

[2.3. Embaralhador / Desembaralhador 10](#_Toc225857015)

[2.4. Codificador/ Decodificador FEC 11](#_Toc225857016)

[2.5. Insersor / Extrator de Carga Útil 14](#_Toc225857017)

[2.6. Extrator de Cabeçalho 18](#_Toc225857018)

[2.7. Banco de Registradores 19](#_Toc225857019)

[3. Interfaces Externas do Framer OTN 21](#_Toc225857020)

[3.1. Interface OTN 21](#_Toc225857021)

[3.2. Interface Cliente 22](#_Toc225857022)

[3.3. Interface Software Embarcado 22](#_Toc225857023)

[4. Ambiente de Simulação 23](#_Toc225857024)

[4.1. Cenários de teste 1 (Genérico de rw em registradores) 24](#_Toc225857025)

[4.2. Cenários de teste 2 (inserção de erros em payload) 25](#_Toc225857026)

[5. Resultados de Síntese 27](#_Toc225857027)

[6. Árvore de diretórios do projeto X10 Giga (SVN) 28](#_Toc225857028)

[7. Problemas Conhecidos 30](#_Toc225857029)

[8. Pendências do Projeto 31](#_Toc225857030)

[9. Referências 32](#_Toc225857031)

[10. Anexo I 33](#_Toc225857032)

# Descrição Geral

O *circuito* em desenvolvimento pelo Grupo de Apoio ao Hardware (GAPH) em parceria com a TERACOM Telemática é o protótipo de um *transponder* capaz de receber/transmitir sinais SDH (*Synchronous Digital Hierarchy*) e Gigabit Ethernet sobre redes do tipo OTN (*Optical Transport Network*) com enlaces ópticos em longas distâncias.

O diagrama de blocos a baixo ilustra de forma sucinta como é organizado o *transponder*.



Figura - Visão Geral da Arquitetura do Projeto x10Giga

Os **módulos serializador/deserializador** são módulos existentes na placa de prototipação e são responsáveis por receber o sinal da fibra ótica serialmente e repassá-los ao FPGA de forma paralela. O sinal da fibra ótica é recebido a aproximadamente 10gbit/s, em uma freqüência de 669.327MHz (freqüência Inalcançável pelo FPGA), conforme padrão OTU2. O repasse do sinal ao FPGA é realizado em palavras de 64 bits, com a freqüência de 167.332MHz, assim podendo ser utilizado no FPGA. O processo inverso é aplicado no serializador, retornando à freqüência de transmissão.

O **módulo alinhador** de quadro é composto por sub-módulos que consistem em identificar a ocorrência da seqüência de alinhamento, identificar em qual bit a seqüência está começando, realizar o deslocamento de modo que a seqüência de alinhamento comece no início da palavra de 64 bits e alimentar uma máquina de estados que realiza o sincronismo. Para ser atendida a restrição de tempo de propagação combinacional do projeto de 5 ns, cada sub-módulo é separado por um registrador, formando assim um pipeline.

O **módulo embaralhador** consiste em embaralhar os dados a serem transmitidos utilizando a técnica de geração de dados pseudo-aleatórios, conhecida como um LFSR (Linear Feedback Shift Register). Com isto, tem-se uma proteção para que não ocorram longas seqüências de 0s ou de 1s, o que garante suficiente troca de estado dos bits para facilitar a regeneração do relógio e também evita uma possível repetição da seqüência de alinhamento. A operação de embaralhar é realizada em todo o multi-quadro OTN G.709, com exceção do campo FAS. A funcionalidade de desembaralhar é a mesma de embaralhar, visto que a operação de embaralhamento é reversa. Basta passar um dado previamente embaralhado pelo módulo embaralhador que ele sairá desembaralhado.

O **módulo Forward Error Correction (FEC)** utiliza o método de geração de código de correção de erros Reed-Solomon (RS) para produzir informação redundante, que é transmitida com o quadro OTN e utilizado na recepção. Esta informação é utilizada para localizar e corrigir erros que porventura venham a acontecer durante a transmissão [SIL07]. O ganho gerado pela utilização de correção de erros FEC pode ser de até 6.2 dB, ou seja, devido ao menor número de erros na comunicação é possível transmitir um sinal a certa taxa BER (Bit Error Rate) utilizando 6.2 dB a menos de potência. Tal ganho permite que seja utilizada uma extensão de fibra óptica de maior comprimento. Possibilita, também, a utilização de um maior número de canais em um sistema DWDM e, o mais importante, possibilita a evolução de enlaces ponto a ponto para redes ópticas ao diminuir a necessidade de regeneração do sinal no decorrer do caminho [PAI07].

O **módulo extrator de cabeçalho** é responsável pela comunicação com o software embarcado. Este módulo retira os diversos dados encontrados no cabeçalho do pacote OTN, interpretá-os e se necessário ativa os diversos alarmes do sistema. Este módulo também é responsável por atualizar algumas informações que são armazenadas em um banco de registradores, o qual poderá ser lido/escrito pelo software embarcado sendo executado no processador Microblaze.

Após o início da recepção de quadros válidos no **módulo de extração de carga útil**, este analisa os campos de Justification Control (JC) para determinar se há ou não justificativa no multi-quadro. Este módulo executa deslocamentos nas palavras de 64 bits recebidas para alinhá-las de acordo com a carga útil quando há justificativa, além de escrever na fila assíncrona os dados que serão entregues ao cliente 10GbE. Este módulo também é responsável por gerar um sinal de referência para o controle de *jitter* externo.

O **banco de registradores** é utilizados para armazenar os sinais extraídos dos cabeçalhos, assim como receber os sinais que deverão ser inseridos nos cabeçalhos enviados, conforme especificado pelo software embarcado sendo executado no processador Microblaze. O banco de registradores foi projetado de forma que possa ser lido e escrito tanto pelo Microblaze quanto pelos módulos extrator de cabeçalho e insersor de carga útil.

O **módulo insersor de carga útil** é responsável pelo controle da leitura dos dados a serem transmitidos, armazenados em uma fila de entrada de dados na interface do cliente. Este módulo realiza o controle da ocupação desta fila, reduzindo a freqüência de inserção de dados na fila. O módulo também é responsável pela geração do fluxo de dados a serem transmitidos, inserindo justificativas, codificando os dados, embaralhando-os e os transmitindo ao serializador.

# Módulos internos

Neste capítulo serão detalhados todos os módulos internos presentes na arquitetura descrita na .

## Serializador / Deserializador

Um módulo de interface com a fibra óptica, é responsável por paralelizar o fluxo de dados a 10,688 Gbps em 16 canais diferenciais (LVDS), cada um a uma taxa de 669,3125 Mbps. O FPGA é responsável por receber cada um destes fluxos, paralelizando-os em 4 bits (). Desta forma, a freqüência interna de trabalho é reduzida a 167,33 MHz, e os dados disponibilizados em uma largura de 64 bits.

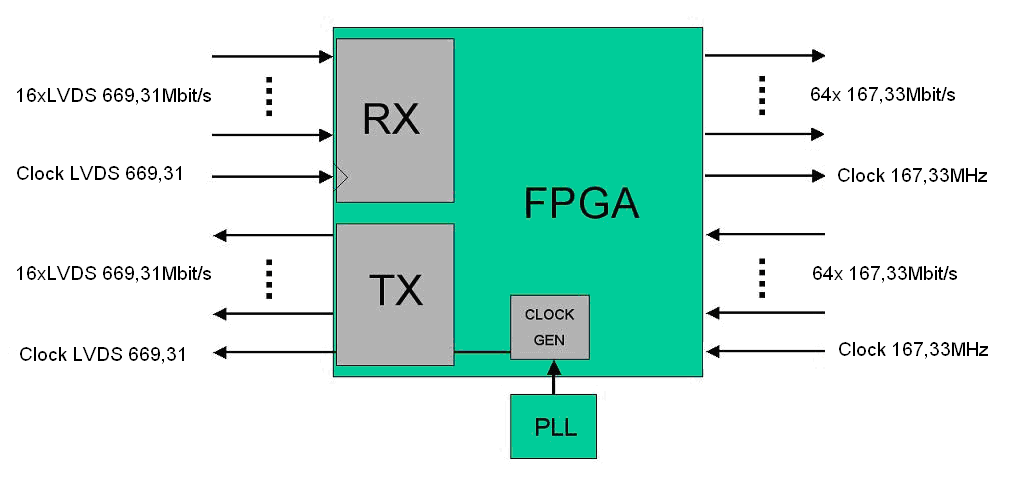


Figura – Interface com a fibra óptica.

Os módulos RX e TX, ilustrados na (), são detalhados na Figura 3. O desenvolvimento destes módulos é de responsabilidade da TERACOM, e são utilizados como blocos “caixa-preta” no presente projeto. Notar apenas que na figura abaixo a interface OTN está representada no lado direito da figura. O escopo do projeto é o bloco **OTN FRAMER.**

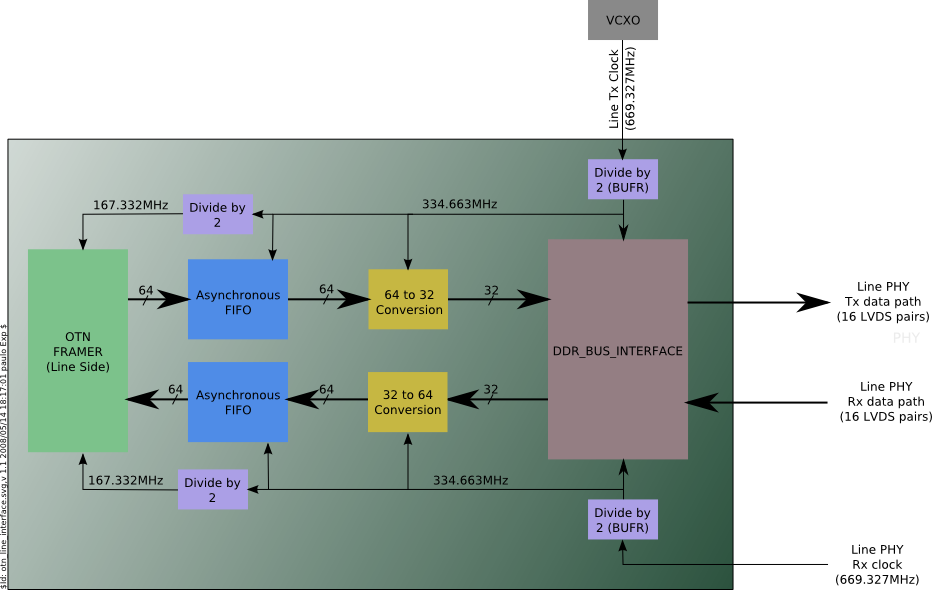


Figura – Diagrama de blocos dos módulos de recepção e transmissão de dados OTN.

## Alinhador

O módulo alinhador de quadro tem por restrição de projeto trabalhar com um fluxo de dados de largura 64 *bits*, operando a uma freqüência de 167,33 MHz. Foi adotada como freqüência de operação 200 MHz, o que resulta em uma taxa de 12,8 Gbps. Esta maior taxa de operação é adotada como cenário de pior caso, permitindo o desenvolvimento de um projeto mais robusto, levando em conta que as principais considerações quanto ao alinhamento de multi-quadros são alta confiabilidade e resposta rápida [CHO90].

Para delimitar o início de cada multi-quadro OTN é utilizada uma seqüência de *bytes* definida pela palavra em hexadecimal “F6F6F6282828”. De acordo com o padrão OTN, pode-se utilizar um subconjunto destes seis *bytes* para realizar o alinhamento e o sincronismo inicial. Optou-se por utilizar um subconjunto de três *bytes*, sendo eles os seguintes: “F6XXF628XXXX”, sendo que “X” representa posições inicialmente desconsideradas na procura pela seqüência de alinhamento. Esta seqüência de *bytes* pode estar começando em qualquer *bit* dos 64 *bits* recebidos nos pinos de entrada, inclusive o caso de estar começando nos últimos *bits*. O que nos levará a ter os *bits* restantes no início da próxima palavra de 64 *bits* recebida pelos pinos de entrada (, caso 1) denominada palavra futura.

O início de cada multi-quadro é composto por esta seqüência, por isso então terá que se realizar a identificação bem como o deslocamento dos *bits* das palavras de 64 *bits* recebidas. O módulo alinhador de quadro é formado por sub-módulos que consistem em identificar a ocorrência da seqüência de alinhamento, identificar em qual bit a seqüência está começando, realizar o deslocamento de modo que a seqüência de alinhamento comece no início da palavra de 64 *bits* (, caso 2) e alimentar a máquina de estados que realiza o sincronismo.

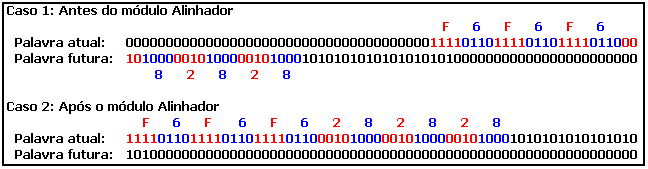


Figura – Exemplo de desalinhamento e posterior alinhamento.

Uma vez realizado o alinhamento, todas as palavras seguintes do multi-quadro terão que ser deslocadas o mesmo número de bits.

### A Arquitetura do Módulo Alinhador

O alinhador de quadro é dividido em sub-módulos. Cada sub-módulo foi separado por um registrador, formando então um *pipeline* (Figura 5). Esta divisão é necessária para que seja atendida a restrição de tempo de propagação combinacional do projeto, igual a 5 ns.

Primeiramente, os 64 *bits* da palavra atual (palavra armazenada no registrador do 1º estágio) e os 64 *bits* da palavra futura (palavra que ainda não entrou no primeiro estágio) são processados pelo sub-módulo **comparador parcial**, que é responsável por identificar a ocorrência da seqüência de alinhamento a partir da verificação de um subconjunto de três *bytes*. É necessário realizar esta verificação 64 vezes, pois a seqüência de alinhamento pode estar começando em qualquer *bit* da palavra atual. Uma vez identificada esta subseqüência de alinhamento, são passados 64 *bits* ao sub-módulo **gerador de endereço**. Havendo alinhamento um destes *bits* estará em nível lógico ‘1’ indicando onde o início da seqüência de alinhamento começou. De posse do índice onde este bit está localizado é gerado o endereço em que o mesmo se encontra e o sinal match indica a ocorrência da seqüência de alinhamento atuando como “ativador” do registrador de endereço. O módulo deslocador utiliza este endereço, que corresponde ao número de bits a serem deslocados para realizar o deslocamento correto entre a palavra atual e a palavra futura. Por fim, é realizada uma última verificação pelo comparador completo, que confere se os 6 bytes da seqüência de alinhamento estão corretos e alimenta a máquina de estados que realiza o controle do sincronismo de multi-quadros.



Figura – Arquitetura do módulo alinhador de quadro com pipeline.

Para a implementação dos módulos que compõem o alinhador de quadro fez-se uso de lógica combinacional tanto na sua arquitetura como na codificação, para ganho de desempenho tanto em área utilizada como em velocidade de processamento. Devido aos requisitos do projeto, esta não foi uma escolha, e sim, uma necessidade.

A partir de estudos realizados utilizando a opção FPGA EDITOR presente na ferramenta ISE da Xilinx, foi possível identificar como a codificação estava sendo mapeada fisicamente. Foi verificado que a ferramenta de síntese inferia multiplexadores quando eram realizadas operações lógicas com mais de quatro entradas, o que resultava em um acréscimo no caminho crítico. Assim, optou-se por utilizar portas lógicas de quatro entradas nos módulos desenvolvidos.

### O Comparador Parcial

O sub-módulo comparador parcial, é responsável pela identificação de um subconjunto de três dos seis *bytes* da seqüência de alinhamento, sendo estes “F6XXF628XXXX”. Como a seqüência de alinhamento pode estar começando em qualquer *bit* dos 64 *bits* da palavra recebida, são necessárias 64 instâncias deste sub-módulo. O sinal de saída, que indica a ocorrência da seqüência de alinhamento, é um vetor de 64 posições onde cada posição do vetor corresponde ao número de *bits* deslocados em que a seqüência foi encontrada. Cada *bit* deste vetor representa a saída de uma instância do comparador parcial.

A Figura 6 ilustra uma instância em que a verificação é feita exatamente nos primeiros *bits* da palavra atual. Neste caso, se a saída indicada por C[0] estiver em nível lógico ‘1’, significa que a seqüência de alinhamento já está na posição correta e não precisa ser deslocada. Já a Figura 7 ilustra uma instância em que a verificação está começando no último *bit* da palavra atual, o *bit* 63. Os demais *bits*, do 64 ao 94 são da palavra futura, sendo o *bit* 64 equivalente ao *bit* 0 da palavra futura e o *bit* 94 equivalente ao *bit* 30 da palavra futura. Neste caso, se a saída indicada por C[63] estiver em nível lógico ‘1’, significa que a seqüência de alinhamento está desalinhada em 63 *bits*.



Figura – Instância do Comparador Parcial sem deslocamento.



Figura – Instância do Comparador Parcial com deslocamento de 63 bits.

Portanto, no sinal de saída composto por 64 *bits*, o *bit* que estiver em nível lógico ‘1’ indica o *bit* de início da seqüência de alinhamento. A próxima etapa consiste em transformar o índice onde este *bit* se encontra em um endereço para realizar o deslocamento necessário.

### Gerador de Endereços

O sub-módulo gerador de endereço é responsável pela conversão do sinal *controle*, proveniente da saída do comparador parcial, para um endereço de 6 *bits*. Este endereço corresponde ao índice em que a saída do comparador parcial estava em nível lógico ‘1’ (). Também cabe ao gerador de endereço a tarefa de acionar o sinal *match* toda vez que o sinal controle tiver um de seus *bits* em nível lógico ‘1’.

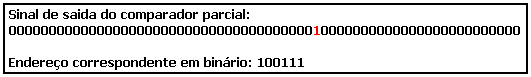


Figura – Exemplo de conversão realizada pelo Gerador de Endereço.

A arquitetura utilizada no sub-módulo gerador de endereço está representada na . Cada instância computa um *bit* do endereço através da operação lógica OU. As entradas das portas lógicas OU correspondem aos *bits* que, para sua identificação por um endereço binário, irão utilizar aquele *bit* de endereço.



Figura – Instancias do Gerador de Endereço.

O endereço gerado pelo módulo corresponde ao *bit* em que a seqüência de alinhamento está iniciando, que por sua vez indica também o número de *bits* que deverão ser deslocados pelo sub-módulo deslocador.

### Deslocador

O sub-módulo deslocador tem como objetivo realizar o deslocamento da seqüência de alinhamento de N posições, onde N varia de 0 a 63. O número de posições a serem deslocadas é passado pelo gerador de endereço. Usou-se um arranjo logarítmico de multiplexadores 2x1, pois uma LUT (*LookUp Table*) tem 4 entradas. Assim, otimizou-se a área e o tempo de processamento. A ilustra um deslocador de 8 *bits*, o qual possui três estágios (colunas de multiplexadores). Neste caso, cada estágio realiza o deslocamento de 1, 2 ou 4 *bits*. Para o deslocamento de 64 *bits* são necessários seis estágios. O princípio de funcionamento do deslocador de 64 *bits* é exatamente o mesmo do de 8 *bits*, porém o número de multiplexadores cresce exponencialmente, o que torna sua ilustração gráfica muito complicada.

O endereço recebido entra como controle dos multiplexadores. Tendo como exemplo a , teríamos um endereço de 3 *bits*, sendo que as entradas identificadas pelos números 1 até 8 podem ser entendidas como a palavra atual, e as entradas identificadas por N\_1 até N\_7 podem ser entendidas como a palavra futura. Caso se tivesse, por exemplo, o endereço 010, teríamos que deslocar 2 *bits* pois 010 em binário corresponde a 2 em decimal. De acordo com a figura, o *bit* menos significativo é o controle dos multiplexadores que estão ligados diretamente a saída, o *bit* mais significativo é o controle dos multiplexadores que estão ligados na entrada e o *bit* intermediário é o controle dos multiplexadores intermediários. Neste exemplo, colocando os multiplexadores intermediários em nível lógico ‘1’ se terá um deslocamento de 2 *bits*, o que está de acordo com a figura.



Figura – Deslocador de 8 bits.

Devido à restrição de *timing* deste projeto, dividiu-se o deslocador de 64 *bits* em três estágios de *pipeline* para diminuir o tempo de processamento, sendo que cada estágio do *pipeline* é formado por duas colunas de multiplexadores.

### Comparador Completo

O sub-módulo Comparador Completo é muito parecido com o Comparador Parcial. A diferença está na quantidade de *bytes* verificados, que no caso do comparador completo são verificados todos os 6 *bytes* da seqüência de alinhamento, sendo estes “F6F6F6282828”. Outra diferença está no número de instâncias. Como o comparador completo encontra-se após o módulo que realiza o deslocamento, ele só precisa verificar a ocorrência da seqüência de alinhamento no início da palavra atual. Essa verificação assegura que a seqüência de alinhamento está com todos os seus *bytes* corretos, uma vez que no módulo comparador parcial são verificados apenas 3 dos 6 *bytes*. Se fossem utilizadas 64 instâncias do comparador completo haveria além do acréscimo de área, um tempo de processamento muito elevado caso fossem verificados todos os 6 *bytes*. A ilustra o módulo Comparador Completo, o qual gera na saída o sinal *fullmatch,* este sinal alimenta a máquina de estados que controla o sincronismo dos multi-quadros sinalizando a ocorrência da seqüência de alinhamento.



Figura – Comparador Completo.

## Embaralhador / Desembaralhador

A operação de scrambling consiste em embaralhar a informação a ser transmitida utilizando um LFSR, que é uma técnica de geração de dados pseudo-aleatório. Com isto, tem-se uma proteção para que não ocorram longas seqüências de 0s ou de 1s, o que garante suficiente troca de estado dos bits para facilitar a regeneração do relógio e também evita uma possível repetição da seqüência de alinhamento. A operação de embaralhar é realizada em todo o multi-quadro OTN G.709, com exceção do campo FAS. A funcionalidade do embaralhador e do desembaralhador é idêntica, uma vez que a operação de embaralhamento é simétrica. Basta passar um dado previamente embaralhado pelo sub-módulo que ele sairá desembaralhado.

Esta arquitetura foi desenvolvida tendo em vista atender as restrições de desempenho impostas pelo projeto, ao invés de se computar as seqüências do polinômio utilizando LFSR em hardware. Uma arquitetura do tipo LFSR demanda um maior tempo de processamento do que uma abordagem mais simples em que todas as seqüências do polinômio embaralhador estão inicializadas em memórias BRAM (*Block RAM*). Dado que o polinômio gerador padrão do OTN é de grau 16, tem-se então  *bits*, o que equivale a 65535 *bits* pseudo-aleatórios. Para se alocar 65535 *bits* foram utilizadas quatro BRAM de 1024 x 16 *bits*. A mostra a arquitetura desenvolvida, onde o sinal entrada é a palavra de 64 *bits* proveniente do deslocador e será submetida à operação lógica “XOR” com a palavra de 64 *bits* proveniente das memórias BRAM, identificada na figura pelo sinal *s* também de 64 *bits*.



Figura – Arquitetura de embaralhamento utilizando memórias BRAM.

As seqüências do polinômio contidas nas memórias foram organizados de forma a otimizar o acesso, portanto cada palavra de 64 *bits* foi dividida entre as quatro BRAM, sendo então 16 *bits* por memória. Desta forma é possível ter os 64 *bits* com leituras em paralelo. Para realizar o endereçamento das memórias utilizou-se um contador. Este contador é parte do sub-módulo chamado gerador de endereço da memória. Este módulo tem também a tarefa de realizar a repetição da seqüência do polinômio quando necessária, pois como pôde ser visto anteriormente, o multi-quadro OTN tem 16320 *bytes*, ou seja, 130560 *bits*, o que não fecha com os 65535 *bits* gerados pelo polinômio. Portanto após 65535 *bits* a seqüência terá que ser repetida, completando quase duas seqüências em um multi-quadro. Ainda há um pequeno detalhe, por serem 65535 *bits* gerados pelo polinômio, fica faltando um último bit na última palavra de 64 *bits.* Esta ausência de um *bit* requer que seja executada uma operação de deslocamento de um *bit* entre a palavra em que este *bit* está faltando e a próxima palavra, que por sua vez é a primeira palavra a ser repetida. Para tornar esta operação possível foram utilizadas memórias com dupla porta, permitindo leituras de posições distintas de uma mesma memória em paralelo. Esta operação é feita em todos os multi-quadros.

## Codificador/ Decodificador FEC

O processo de codificação normalmente segue o mesmo princípio de registrador de deslocamento apresentado na figura abaixo (Figura 13). Entretanto, a arquitetura do decodificador pode variar bastante dependendo das necessidades de cada projeto. Diversas propostas de implementação foram estudadas para definir qual é a mais adequada para este trabalho, sendo avaliadas as vantagens e desvantagens de cada uma delas.



Figura – Codificador Reed-Solomon para RS(7,3).

No padrão OTN, um frame é dividido em 16 sub-frames de 255 bytes, e cada sub-frame corresponde a uma mensagem que deve ser codificada ou decodificada paralelamente. O módulo responsável por fornecer os dados para o decodificador deve ser capaz de realizar esta tarefa a uma freqüência de 200Mhz e 64 bits por ciclo de clock. Portanto, a replicação dos módulos, tanto do codificador quanto do decodificador, é obrigatória, pois cada módulo pode tratar um dado de 8 bits por ciclo.

### Codificador FEC

A Figura 14 mostra o esquemático do codificador desenvolvido neste trabalho. Por motivos de espaço não foi possível representar todos os registradores e multiplicadores utilizados no cálculo dos símbolos de paridade, mas como podemos ver, a estrutura é bastante similar com a arquitetura apresentada na Figura 13.



Figura - Arquitetura do codificador desenvolvido.

No caso específico da recomendação G.709, são necessários 16 registradores de 8 bits para armazenar os valores temporários dos símbolos de paridade e por conseqüência, 16 multiplicadores GF são utilizados para realizar o cálculo. É importante observar que as multiplicações realizadas nesta etapa sempre são por constantes, que equivalem aos coeficientes do polinômio gerador da recomendação G.709. Os coeficientes do polinômio gerador, são apresentados na equação abaixo.

|  |  |  |
| --- | --- | --- |
|  |  |  |

A forma de onda apresentada na Figura 15 mostra o início do funcionamento do codificador.

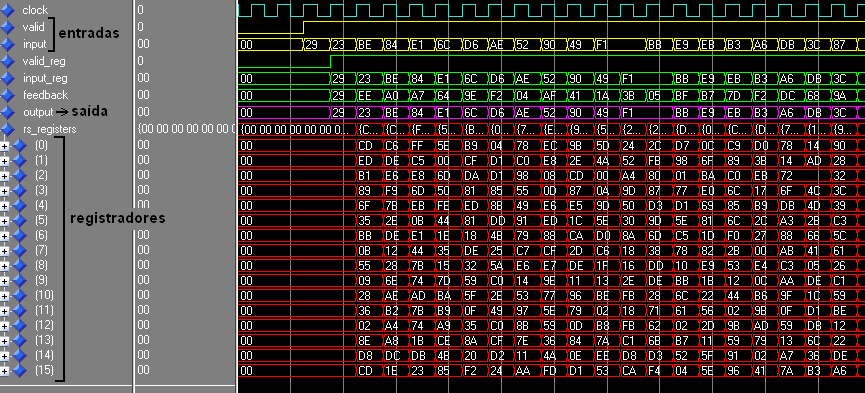


Figura – Início do processo de codificação.

O codificador possui duas entradas principais: *valid* e *input*. O sinal *valid*, quando em nível lógico ‘1’, indica que os dados de 8 bits presentes no sinal *input* são dados válidos e devem ser utilizados para o cálculo de paridade. As duas entradas são armazenadas nos registradores *valid\_reg* e *input\_reg*. Este armazenamento sincroniza as entradas aos registradores (*rs\_registers*) evitando-se um caminho combinacional entre a entrada *input* com os multiplicadores. O registrador *rs\_registers* é subdividido em 16 registradores de 8 bits que armazenam os valores temporários para o cálculo dos símbolos paridade. Cada registrador é alimentado pela soma de seu registrador vizinho com a multiplicação do sinal *feedback* pelo respectivo coeficiente do polinômio gerador. Durante o recebimento dos dados válidos, enquanto o sinal *valid* estiver em ‘1’, o codificador apenas repassa os dados da entrada diretamente para a saída do circuito através do sinal *output*.

Após 239 ciclos de clock com o sinal *valid* em ‘1’, os 16 bytes de paridade correspondentes ao pacote recebido encontram-se armazenados em *rs\_registers*. Nos 16 ciclos de clock seguintes estes dados são enviados para a saída como mostra a forma de onda apresentada na Figura 16.

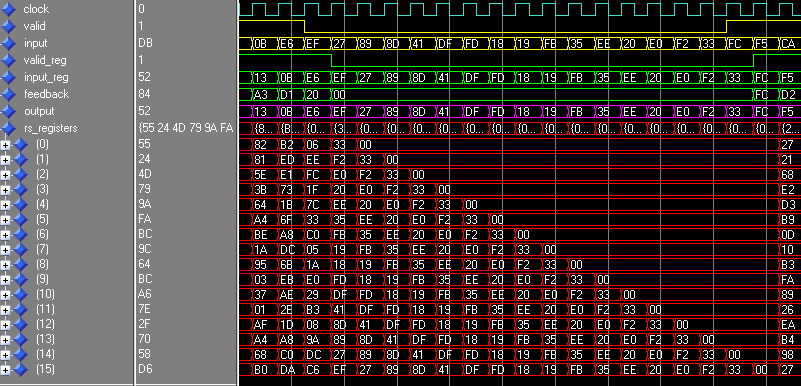


Figura – Envio dos símbolos de paridade.

Como podemos observar na Figura 16, imediatamente após a transição de *valid* para ‘0’, o sinal *feedback* passa a ser “00” e a saída recebe o último byte de *rs\_registers*, que é deslocado a cada ciclo de clock, de maneira que todos os símbolos contidos nos registradores são enviados para a saída e os registradores do módulo são zerados.

### Decodificador FEC

O decodificador possui três blocos principais, são eles: a unidade de cálculo das síndromes, o solucionador da Equação Chave, que relaciona as síndromes aos polinômios localizador e avaliador de erros, o algoritmo de busca das localizações e dos valores de erros.

A unidade de cálculo da síndrome é responsável por verificar se as raízes do polinômio gerador são as mesmas raízes da mensagem recebida, vista como um polinômio. No caso do RS(255,239) a mensagem recebida é vista como um polinômio de grau 255, e se não tiver erros, as raízes deste polinômio são as mesmas do polinômio gerador. No caso de uma mensagem recebida com erros, as raízes do polinômio gerador, aplicadas como entrada da mensagem recebida, geram um conjunto não nulo de síndromes, que correspondem ao padrão de erro inserido na mensagem e podem ser relacionadas com o polinômio localizador de erros e com o polinômio avaliador de erros através da seguinte equação:

|  |  |  |
| --- | --- | --- |
|  |  | ( |

Onde S(*X*) representa o conjunto de síndromes, Λ(*X*) representa o polinômio localizador de erros e Ω(*X*), o polinômio avaliador de erros.

Esta equação é normalmente denominada pela literatura de Equação Chave, pois ela é a etapa principal e mais complexa etapa do processo de decodificação. Diversos métodos de implementação em hardware para resolvê-la foram propostas. O algoritmo de Berlekamp-Massey e o algoritmo Euclideano são os mais eficientes para resolver este problema [WIL99]. O algoritmo Euclideano é mais simples que o algoritmo de Berlekamp-Massey e possui estrutura mais regular e escalável, entretanto, necessita mais blocos lógicos para ser implementado devido ao emprego de divisão polinomial [WAI05]. O algoritmo original de Berlekamp-Massey não tem mais sido utilizado em implementações atuais devido a sua estrutura irregular e por possuir um caminho crítico mais longo [SAR01], o que não possibilita freqüências de clock muito elevadas. Algumas modificações no algoritmo original de Berlekamp-Massey permitem estruturas mais regulares e caminho crítico mais curto, porém utilizam mais multiplicadores de campo finito [PAR05], o que acaba consumindo mais blocos lógicos.

Em [MAS69] e [LIU84] são propostas arquiteturas para implementar o algoritmo original de Berlekamp-Massey. Essas implementações utilizam um módulo de divisão de campos finitos, composto por um módulo inversor e um multiplicador. Já nas implementações propostas em [SAR01], [PAR05], [CHA01] e [REE91], são utilizados algoritmos modificados do Berlekamp-Massey, sem o uso de inversores, porém utilizam 3*t* multiplicadores contra 2*t* do algoritmo original. As implementações propostas em [WIL99] e [LEE05] utilizam variações do algoritmo Euclideano para resolver a Equação Chave.

Na última etapa da decodificação, os polinômios obtidos através da solução da Equação Chave são utilizados para determinar as localizações e os valores dos erros através de dois algoritmos, denominados *Chien search* e *Forney*. A saída do último módulo possui o padrão de erro inserido na mensagem recebida, desde que esta mensagem não possua mais de *t* erros. Para se obter a mensagem decodificada é necessário armazenar a mensagem recebida até que o decodificador processe esta mensagem e gere o padrão de erro, para isso é necessário utilizar uma memória organizada em estrutura de fila. O tamanho desta memória depende do tempo de processamento dos componentes do decodificador.

A Figura 17 mostra o diagrama de blocos do decodificador desenvolvido. Detalhes da implementação de cada bloco estão disponíveis em [TEC\_ZEH08]:



Figura – Arquitetura do decodificador RS(255,239) desenvolvido.

## Insersor / Extrator de Carga Útil

A interface com o cliente é apresentada na . A O sistema desenvolvido compreende os módulos *Client Drop Interface* (extrator de carga útil) e *Client Add Interface* (insersor de carga útil).

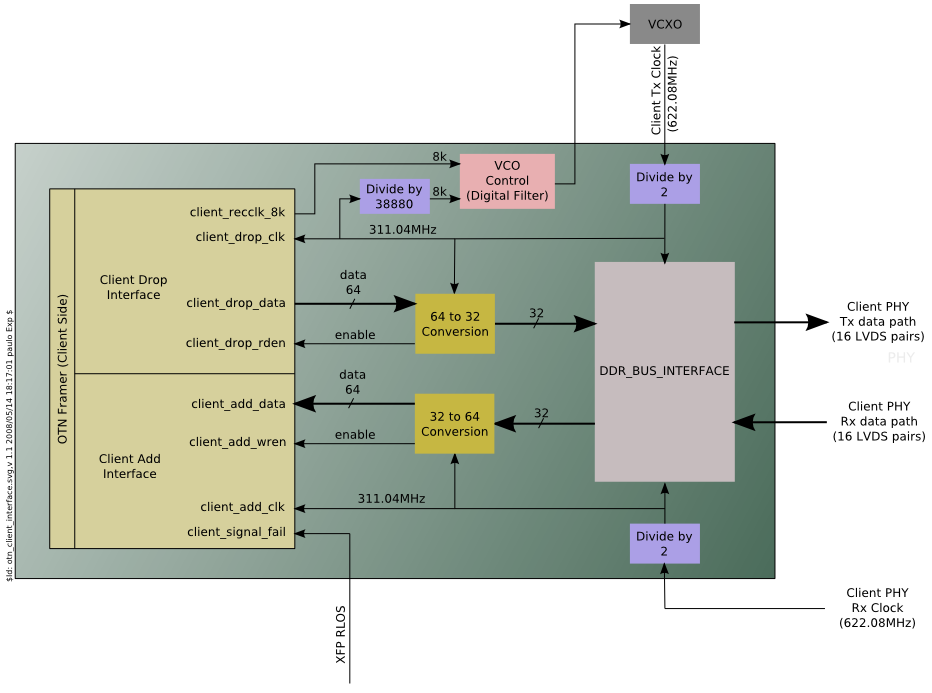


Figura – Diagrama de blocos da interface com o cliente [TER08].

### Insersor de Carga Útil

Uma fila interna ao módulo *insersor de carga útil* é carregada, ininterruptamente, com os dados do cliente a uma freqüência nominal de escrita igual a 161,13 MHz. O controle desta fila é responsável por evitar que a fila não estoure ou esvazie completamente quando houver variações nessa freqüência, devido ao *jitter* provindo do sinal do cliente. O módulo insersor de carga útil desenvolvido tem como principais funções:

* controle da leitura da fila com os dados do cliente a serem transmitidos;
* controle da ocupação da fila;
* inserção de justificativa;
* gerar o *stream* de dados para ser codificado, embaralhado e transmitido.

Considerando que cada quadro OTN contém 510 palavras de 64 bits, podemos dividi-lo como na abaixo.



Figura – Divisão do quadro OTN em áreas de palavras de 64 bits.

O *stream* de transmissão do quadro é ilustrado no diagrama de tempo da . Podemos observar que a carga útil é inserida nos 476 ciclos de *payload*, ou seja, nesse momento a fila está esvaziando, porém, nos outros 34 ciclos a fila enche rapidamente por não estar sendo lida, somente a escrita está habilitada para guardar os dados do cliente que chegam constantemente. Desse modo, o gráfico de ocupação da fila se comporta como uma serra, variando entre um máximo e um mínimo constantes, no caso nominal.



Figura – Diagrama de tempo da transmissão do quadro OTN.

O ponto médio entre a ocupação máxima e mínima deve ser a metade da fila para que tenha o máximo de folga antes de atingir os estados de fila cheia ou vazia. O tamanho da fila da interface com o cliente utilizado no projeto é de 512 palavras, logo, a ocupação deve excursionar em torno da metade da sua totalidade, no caso ideal. Portanto, quando o sistema é iniciado, o controle do insersor aguarda a fila encher até um determinado ponto, para quando entrar em regime de operação, os pontos de máximo e mínimo fiquem a mesma distância da metade da ocupação da fila. A latência causada pelo processo inicialização é de 1685,56 ns, aproximadamente, o que corresponde um pouco mais do que a metade de um quadro, ou seja, 291 palavras de 64 bits.

A inserção de justificativa, necessária para o controle de *jitter*, é feita em cada multi-quadro, sendo que nos três primeiros são colocados, nos campos de JC, os votos de controle de justificativa, definidos pela condição da ocupação extrapolar os limites de máximo e mínimo. Nominalmente os votos são nulos, se a fila estiver mais cheia, é inserido um voto para justificativa negativa, caso contrário, voto vai para justificativa positiva. No quarto quadro, a eleição é feita, e o voto majoritário define como serão preenchidos os bytes de PJO e NJO.

A estrutura do módulo insersor de carga útil desenvolvido nesse trabalho é representada no diagrama de blocos da , contendo a disposição de alguns dos sinais mostrados nas simulações.



Figura – Estrutura do módulo insersor de carga útil.

Nos 34 ciclos em que não é inserida carga útil no fluxo de dados de transmissão, 32 deles são enviadas palavras somente com zeros, que serão sobrescritas pelo resultado do decodificador Reed-Solomon [SIL07]. Nos outro dois ciclos são escolhidos os dados para formar as palavras que correspondem ao cabeçalho.

### Extrator de Carga Útil

O sinal transmitido pela linha OTN é recebido, deserializado, e entregue em palavras de 64 bits aos módulos da recepção no projeto X10GIGA que alinha, desembaralha, corrige possíveis erros e extrai a informação dos cabeçalhos recebidos, procedimentos executados pelos módulos apresentados na, antes de disponibilizar os quadros para o módulo de extração de carga útil. Uma máquina de estados, que controla se o sinal recebido é um quadro válido, gera um sinal que indica o início de quadros válidos, sinal start da .

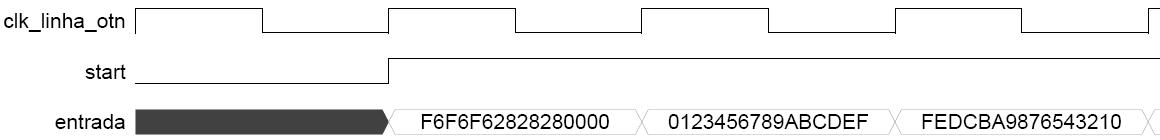


Figura – Geração do sinal que indica o início de quadros válidos ao Extrator de Carga Útil.

Após o início da recepção de quadros válidos no módulo de extração de carga útil, este deve ser capaz de:

* Analisar os campos de *Justification Control* (JC) e determinar se há ou não justificativa no multi-quadro;
* Executar deslocamentos nas palavras de 64 bits recebidas para alinhá-las de acordo com a carga útil quando há justificativa;
* Escrever na fila assíncrona os dados que serão entregues ao cliente 10GbE;
* Gerar sinal de referência para o controle de *jitter* externo.

A apresenta a arquitetura do módulo de extração de carga útil desenvolvido. Nesta Figura, o item 1 apresenta o módulo de votação. Este módulo verifica os bits de JC (*justification control*), presentes nos três primeiros quadros de um multi-quadro e disponibiliza o resultado para o módulo de controle de justificativa, item 2 da .



Figura – Extrator de Carga Útil 10GbE de quadros OTN. (1) Módulo de Votação de *Justification Control* (JC); (2) Módulo que faz o controle de justificativa; (3) Módulo de deslocamento; (4) Registrador de entrada do circuito; (5) Concatenação do sinal *resto*, devido a justificativas; (6) Concatenação do sinal *atual*, devido a justificativas; (7) Controle para atraso do sinal de saída do Deslocador, devido a justificativas; (8) Fila Assíncrona; (9) Gerador de pulsos de referência para controle de variação na taxa do cliente 10GbE.

O módulo Controle de Justificativa, item 2 da , é responsável pelo controle de toda a manipulação dos dados de entrada do Extrator de Carga Útil, até a escrita na fila assíncrona, item 8 da . De acordo com os resultados disponibilizados pelo módulo de votação após o terceiro quadro de cada multi-quadro o controle de justificativa decide que atitude deve ser tomada tomando em conta os procedimentos executados até então, em função de um contador de justificativas. Devido ao fato da justificativa ser feita de 8 em 8 bits e trabalharmos com palavras de 64 bits, se faz necessário o deslocamento das palavras de 64 bits para que estas contenham a justificativa executada. Para este processo de deslocamento é utilizado o módulo Deslocador, que tem como entradas a palavra “atual” e o “resto” da entrada anterior do Extrator de Carga Útil, item (3) da . Este módulo tem a capacidade de deslocar de 8 a 56 bits, de acordo com o valor do sinal *deslocamento*.

O sinal *enable\_resto* controla o fluxo de dados que serão utilizados como carga útil, item 4 da . Quando não é necessário executar deslocamento nas palavras de entrada, os dados que saem do registrador R1 são os dados que chegam ao sinal *resto*,a partir do multiplexador controlado pelo sinal *mux\_resto*. Logo após, ao passarem pelo módulo Deslocador, item 3 da , sem deslocamentos, os dados chegam à entrada da fila assíncrona, sinal *DI* do item 8 da , através do multiplexador que é controlado pelo sinal *mux\_fifo*. Por fim, os dados são escritos na fila de acordo com o sinal *write\_enable*. Este processo é exemplificado na , onde é possível observar que as duas primeiras palavras de entrada, cabeçalhos, são ignoradas e apenas a carga útil é escrita na fila.



Figura – Extração de carga útil. Caso: contador de justificativas zerado e/ou justificativa não é necessária.

O contador de justificativas é incrementado sempre que há uma justificativa negativa, ou seja, um byte a mais no multi-quadro, e decrementado sempre que há justificativa positiva, ou seja, um byte a menos no multi-quadro. Caso o contador esteja em zero e ocorra uma justificativa positiva, então, o contador é ajustado para o valor sete. Analogamente, quando ocorre uma justificativa negativa e o contador está em sete, este é ajustado para o valor zero.

## Extrator de Cabeçalho

Como visto anteriormente, o cabeçalho representa a área do quadro onde estão definidos os *bytes* de controle da comunicação OTN. As informações contidas no cabeçalho estão divididas em quatro quadros OTN, sendo 16 *bytes* por quadro, totalizando 64 *bytes* por multi-quadro OTN (Figura 25). Nestes *bytes* existem campos relacionados a informações de identificação da origem e do destino, identificação do operador, identificação do tipo de dado a ser ou sendo transportado bem como identificação de falhas e alarmes variados. Estas informações precisam ser interpretadas e processadas, o que será feito pelo software embarcado presente na plataforma de prototipação externa ao contexto deste trabalho a qual este projeto será integrado.

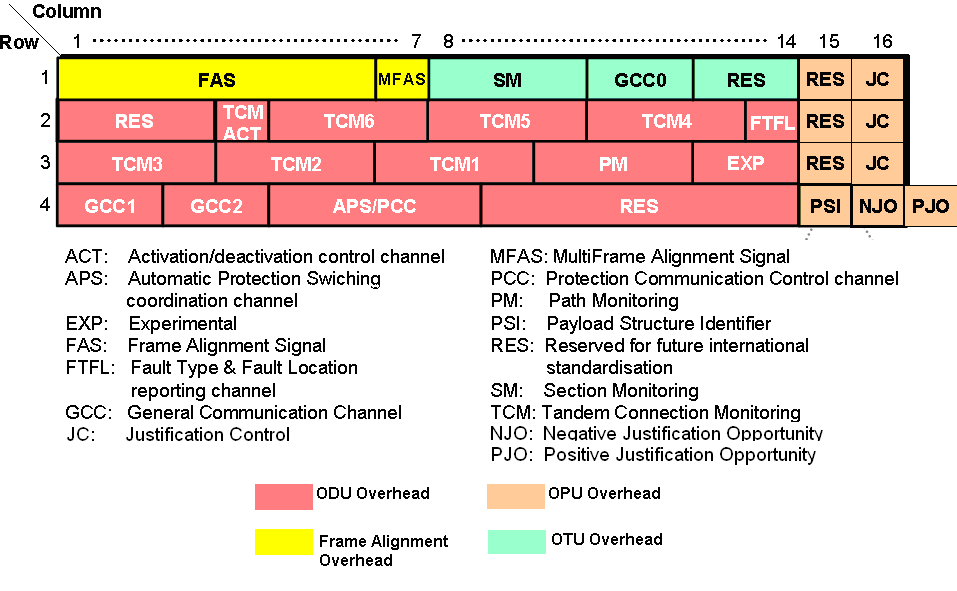


Figura – Bytes que compõem o cabeçalho e seus nomes [VIS07].

A extração dos cabeçalhos consiste em identificar em quais palavras do quadro cada tipo de cabeçalho se encontra e armazená-los em memórias BRAMs. Os cabeçalhos serão armazenados em três memórias lógicas distintas, referente aos três diferentes tipos de cabeçalhos (ODU, OTU e OPU). Nestas memórias estarão armazenados os dados referentes aos últimos 256 multi-quadros.

A arquitetura responsável pela extração dos cabeçalhos é composta por sete BRAMs de 2048x1 *bytes*. Estas memórias possuem duas portas de 8 *bits* disponíveis para escrita e leitura dos dados. Os cabeçalhos foram organizados nas memórias de forma a otimizar a escrita em paralelo dos dados, tornando possível a extração dos cabeçalhos contidos nas palavras de entrada em no máximo três ciclos de relógios. A Figura 26 ilustra as sete memórias com seus cabeçalhos associados.



Figura – Organização dos cabeçalhos nas memórias.

## Banco de Registradores

O banco de registradores é composto por diferentes tipos de registradores. Os registradores do tipo *Read/Write* (R/W) são responsáveis, em sua maioria, por configurações com as quais os tipos de alarme do OTN serão sinalizados para o software embarcado. Os registradores do tipo *Read and Clear* (RC) em sua maioria são registradores de status, indicando a ocorrência de erros de transmissão, alinhamento, paridade entre outros. Estes registradores ao serem atualizados, serão somente apagados quando o software embarcado realizar uma operação de leitura.

Os registradores do tipo *Read, Latch and Clear* (RLC) são utilizados em contadores de uso geral. A principal diferença deste tipo de registrador para os registradores do tipo RC é que existe uma *Latch* entre o banco de registradores e a interface com o software embarcado. Este latch adicional evita leituras incorretas, nos registradores onde ocorram muitas operações de escrita via o hardware do tranponder simultâneas a leituras do software embarcado. A

Tabela 1 exibe o nome dos registradores e seus correspondentes endereços no banco de registradores do transponder OTN X10.

Tabela - Nome e endereço dos registradores presentes no banco de registradores do transponder.

|  |  |  |
| --- | --- | --- |
| **Registrador** | **Endereço** | **Tipo do Registrador** |
| OTU\_STS\_PTR\_ADDR | 00 | RC |
| OTU\_STS\_ADDR | 01 | RC |
| OTU\_CFG\_PTR\_ADDR | 02 | RC |
| OTU\_CFG\_ADDR | 03 | RW |
| SM\_BEI\_CNT\_ADDR | 04 | RLC |
| SM\_BIP\_CNT\_ADDR | 05 | RLC |
| OTU\_OOF\_CNT\_ADDR | 06 | RLC |
| OTU\_FRAME\_CNT\_ADDR | 07 | RLC |
| TTI\_ACCESS\_CTRL\_ADDR | 08 | RW |
| TTI\_DATA\_ADDR | 09 | RW |
| FTFL\_ACCESS\_CTRL\_ADDR | 0A | RW |
| FTFL\_DATA\_ADDR | 0B | RW |
| ODU\_PM\_STS\_ADDR | 0C | RC |
| ODU\_PM\_CFG\_ADDR | 0D | RW |
| PM\_BEI\_CNT\_ADDR | 0E | RLC |
| PM\_BIP\_CNT\_ADDR | 0F | RLC |
| OPU\_RX\_PT\_ADDR | 10 | RW |
| OPU\_EXP\_PT\_ADDR | 11 | RW |
| OPU\_TX\_PT\_ADDR | 12 | RW |
| TCM\_STS\_PTR\_ADDR | 13 | RC |
| TCM\_CFG\_PTR\_ADDR | 14 | RW |
| ODU\_TCM\_STS\_ADDR | 15 | RC |
| ODU\_TCM\_CFG\_ADDR | 16 | RW |
| TCM\_BEI\_CNT\_ADDR | 17 | RLC |
| TCM\_BIP\_CNT\_ADDR | 18 | RLC |
| OPU\_CFG\_ADDR | 19 | RW |
| OPU\_STS\_ADDR | 1A | RC |
| RX\_PJUST\_CNT\_ADDR | 1B | RLC |
| RX\_NJUST\_CNT\_ADDR | 1C | RLC |
| TX\_PJUST\_CNT\_ADDR | 1D | RLC |
| TX\_NJUST\_CNT\_ADDR | 1E | RLC |
| FEC\_STS\_PTR\_ADDR | 1F | RW |
| FEC\_CORRERR\_CNT\_HI\_ADDR | 20 | RC |
| FEC\_CORRERR\_CNT\_LO\_ADDR | 21 | RLC |
| FEC\_UNCORR\_CNT\_ADDR | 22 | RLC |
| OPU\_TSE\_CNT\_ADDR | 23 | RLC |

# Interfaces Externas do Framer OTN

A entidade de mais alto nível do transponder se conecta a três interfaces distintas (Figura 27): (1) interface com a linha de transmissão ótica; (2) interface com o cliente; (3) interface com o software embarcado. Este módulo pode ser encontrado dentro do diretório de descrições VHDL (/HDL/Global/transponder\_x10.vhd).



Figura - Descrição das entradas/saídas na entidade *top-level* do projeto.

## Interface OTN

A interface “Linha OTN” conecta-se com a linha de transmissão ótica da rede OTN, e possui os seguintes sinais de entrada/saída:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Pino** | **Direção** | **Bits** | **Descrição** |
| Interface da Linha OTN | clock\_in | in | 1 | Clock da linha OTN (311.04MHz) |
| reset\_placa | in | 1 | Reset da placa |
| clk0\_out | out | 1 | Clock 1x clock\_in |
| clk180\_out | out | 1 | Clock 180 graus defasado de clock\_in |
| clk2x\_out | out | 1 | Clock 2x clock\_in |
| rst\_out | out | 1 | DCM da linha OTN trancado |
| valid\_out | out | 1 | Indicador de dado válido |
| assinatura | out | 64 | Assinaturas |
| entrada | in | 64 | Dados de entrada da linha OTN |
| saida | out | 64 | Dados de saída da linha OTN |

## Interface Cliente

A interface “Cliente” é responsável pela troca de informações com o lado do cliente, e possui os seguintes sinais de entrada/saída:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Pino** | **Direção** | **Bits** | **Descrição** |
| Interface com o Software Embarcado | bus\_addr | in | 8 | Endereços |
| bus\_wdata | in | 16 | Dados de escrita |
| bus\_rdata | out | 16 | Dados de leitura |
| bus\_ncs | in | 1 | Chip select (negado) |
| bus\_nwr | in | 1 | Write enable (negado) |
| bus\_nrw | in | 1 | Read enable (negado) |

## Interface Software Embarcado

A interface “Software Embarcado” é responsável pela troca de informações com o processador Microbalze, e possui os seguintes sinais de entrada/saída:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Pino** | **Direção** | **Bits** | **Descrição** |
| Interface com o Cliente | client\_clk\_i | in | 1 | Clock do cliente |
| client\_rst\_i | in | 1 | Reset do cliente |
| client\_clk\_o | out | 1 | Clock 1x client\_clk\_i |
| client\_rst\_o | out | 1 | DCM do cliente trancado |
| client\_drop\_data\_o | out | 64 | Dados de saida do cliente |
| client\_add\_data\_i | in | 64 | Dados de entrada do cliente |
| client\_drop\_ipperthold\_o | out | 1 | Limiar para liberação de leitura |
| client\_drop\_rden\_i | in | 1 | Read enable do cliente |
| client\_add\_wren\_i | in | 1 | Write enable do cliente |

# Ambiente de Simulação

O ambiente de simulação do módulo permite que sejam testadas e avaliadas todas as interfaces externas e o correto funcionamento do módulo. O ambiente de simulação está descrito na **Erro! Fonte de referência não encontrada.** e sua utilização será detalhada nesta Seção.



Figura - Ambiente de simulação do transponder X10.

A entidade de mais alto nível hierárquico no ambiente de simulação é o *top\_tb*, *testbench* que instancia o hardware do transponder X10 e o simulador do software embarcado que será executado pelo microprocessador Microblaze. Este testbench é responsável por realizar a realimentação dos sinais na interface cliente através de um *loopback* e fornecer e salvar os frames na entrada/saída da interface da linha OTN.

Os frames simulados são gerados a partir de uma ferramenta automática, chamada de “geraframe” e estão disponíveis no arquivo “*entrada.x10*”. Este gerador de frames pode ser configurado, gerando frames controlados para testes específicos. A configuração do gerador de frames é realizada através de um arquivo de configuração, neste arquivo poderão ser modificados os campos do cabeçalho do frame, além da inserção de erros, embaralhamento e desalinhamento. O Anexo I possui um exemplo de arquivo de configuração, bem como a descrição de cada um dos seus campos. Os frames transmitidos pelo transponder são armazenados no arquivo “*saída.x10*” para avaliação pós-simulação.

O simulador de software embarcado permite realizar operações de leitura e escrita no banco de registradores. Este simulador pode ser configurado através de um arquivo de instruções (*instructions.x10*), que define qual operação será executada, sobre qual registrador do banco de registradores e em que momento esta operação será realizada. O arquivo de instruções deve conter uma instrução por linha em ordem cronológica e cada parâmetro deve estar separado por vírgulas respeitando a seguinte sintaxe:

**<tipo de acesso>,<frame>.<décimo do frame>,<dado>,<registrador alvo>**

* **Tipo de Acesso:** Tipo de acesso ao registrador alvo, sendo 0 para escrever e 1 para ler valor.
* **Frame:** Deve ser um valor em hexadecimal de 5 dígitos seguido de um ponto e por sua vez seguido de um valor em decimal de um dígito. Refere-se ao ponto que se quer fazer a leitura/escrita.
* **Décimo do Frame:** Valor em decimal, de 0 a 9, que corresponde à parte do frame (divido em 10 partes) que será acessada.
* **Dado:** Dado deve ser um valor em hexadecimal de 4 dígitos. Se o tipo de acesso for de escrita neste campo deve ser colocado o dado para ser escrito, mas se o tipo de acesso for de leitura este campo deve conter o valor esperado. Se o dado esperado for desconhecido pode-se usar XXXX para não abortar a simulação.
* **Registrador Alvo:** Nome do registrador que se quer acessar.

A seguir seguem exemplos de instruções presentes no arquivo “*instructions.x10*”:

Escrita do valor 0x0440 no registrador TTI\_ACCESS\_CTRL\_ADDR(0x08) no Mframe 0x00010.5

1,00010.5,0440,TTI\_ACCESS\_CTRL\_ADDR

Leitura do registrador TTI\_DATA\_ADDR(0x09) no Mframe 0x00008.2, valor esperado 0x0000

0,00008.2,0000,TTI\_DATA\_ADDR

Leitura do registrador PM\_BIP\_CNT\_ADDR(0x0F) no Mframe 0x0002A.7, valor esperado 0x0440

0,0002A.7,0440,PM\_BIP\_CNT\_ADDR

Leitura do registrador OTU\_FRAME\_CNT\_ADDR(0x07) no Mframe 0x00FAF.0, valor esperado desconhecido

0,00FAF.0,XXXX,OTU\_FRAME\_CNT\_ADDR

Os cenários de simulação são arquivos inseridos no ambiente de simulação, capazes de estimular as interfaces da linha óptica OTN e do banco de registradores. Fazem parte do cenário de simulação os arquivos que contém frames simulados (*entrada.x10*) e instruções de leitura/escrita no banco de registradores (*instructions.x10*). O objetivo geral dos cenários de simulação é avaliar o hardware do transponder, interface de escrita/leitura no banco de registradores e a correta verificação de alarmes gerados pelo frame OTN. Ao final da execução do ambiente de simulação um relatório de testes é criado (Figura 29), neste relatório estão descritos todos os testes realizados. A lista de testes realizados contém o tempo no qual o teste foi executado, o status do teste (OK/FAIL), o valor lido do registrador e o valor esperado no registrador.

O ambiente de simulação é executado na ferramenta Modelsim da Mentor Graphics, através do comando “*do compila.do*”, a seguir deve-se informar a quantidade de tempo a ser executada através do comando “*run <tempo>*”. No contexto da avaliação do transponder, foram desenvolvidos dois cenários de simulação, um que avalia a correção de erros nos frames recebidos na interface óptica OTN, e outro que realiza uma série de avaliações dos alarmes presentes no padrão OTN através de leituras e escritas no banco de registradores.

## Cenários de teste 1 (Genérico de rw em registradores)

Os alarmes do padrão OTN são extraídos pelo hardware do transponder e escritos no banco de registradores, para serem avaliados pelo microprocessador Microblaze. O cenário de teste 1 utiliza o testbench para realizar operações de leitura/escrita necessárias na validação destes alarmes, validando também operações de leitura/escrita no banco de registradores.

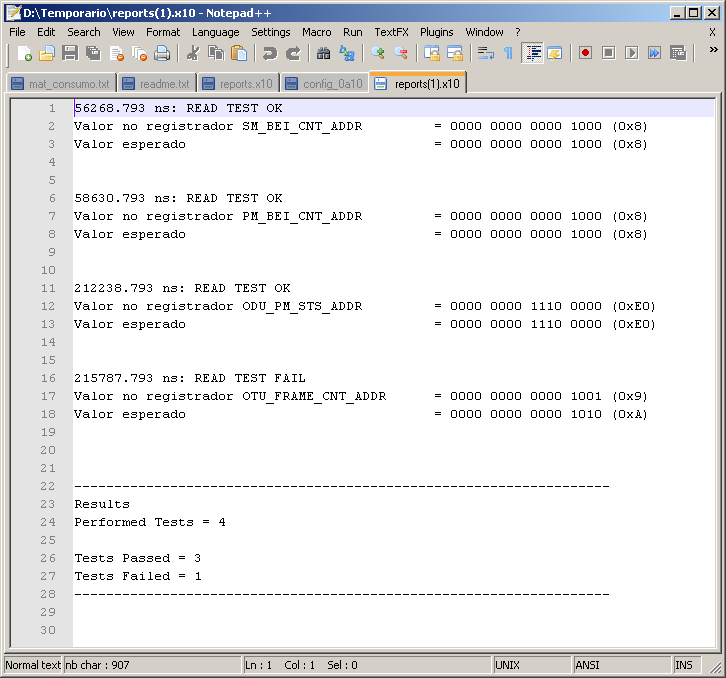


Figura - Exemplo de relatório produzido pelo ambiente de simulação do transponder X10.

A montagem deste cenário de teste é iniciada a partir da montagem do arquivo de configuração da ferramenta de geração automática de frames (**geraframes**). Um exemplo de um arquivo de configuração para a geração de frames da simulação encontra-se no Anexo I, nesta configuração são descritos quais alarmes serão ativados ou não, desta forma, foram gerados diversos frames cada qual derivado de um arquivo de configuração que irá ativar um ou mais alarmes. Estes diversos frames foram concatenados em um único arquivo de entrada de frames e foi alimentado no ambiente de simulação.

Após a criação do arquivo de entrada de frames é necessário configurar o arquivo de instruções que realizarão escritas ou leituras no banco de registradores. Estas operações serão realizadas sobre os frames gerados anteriormente, através do simulador do software embarcado. A execução do ambiente de simulação é realizada através do programa Modelsim.

Para os frames de exemplo gerados pelo arquivo presente no Anexo I, pode-se realizar as seguintes instruções de teste:

#VERIFICA OS REGISTRADORES SE NAO SOMARAM ERRADO

0,00003.6,XXXX,SM\_BEI\_CNT\_ADDR

0,00003.7,0008,SM\_BEI\_CNT\_ADDR

0,00003.8,XXXX,PM\_BEI\_CNT\_ADDR

0,00003.9,0008,PM\_BEI\_CNT\_ADDR

#VERIFICA O ERRO DE BIP, BEI E BIAE (BITS 7,6,5 = E0)

0,00010.9,00E0,ODU\_PM\_STS\_ADDR

As primeiras 4 instruções são para verificar os contadores SM\_/PM\_BEI, duas leituras são necessárias, visto que os registradores são do tipo RLC. E a última instrução verifica os erros de BIP, BEI e BIAE inseridos no frame de teste.

## Cenários de teste 2 (inserção de erros em payload)

O próximo cenário de teste avalia a correta recepção dos frames do protocolo OTN. Neste teste são inseridos erros na carga útil do frame, fazendo com que a correção de erros seja realizada pelo módulo de FEC. A montagem dos frames que contém erros é similar ao cenário de testes 1, porém no arquivo de configuração da ferramenta “*geraframes*” não se tem a preocupação com os campos de alarme do padrão OTN e sim quantos e qual tipo de erros serão inseridos no frame. Os erros são inseridos através dos campos “NUMERRORS” e “TYPERRORS”, sendo que no primeiro se determina quantos erros serão inseridos em um frame e o segundo campo informa qual o tipo de erro a ser inserido.

#VERIFICA OS REGISTRADORES DE ERROS DE FEC

0,0005.0,0006, FEC\_CORRERR\_CNT\_HI

0,0005.0,0006, FEC\_CORRERR\_CNT\_LO

0,0005.0,0006, FEC\_UNCORR\_CNT

Para saber quantos erros ocorrem durante a simulação é necessário realizar operações de leitura em dois registradores distintos, um que possui o número de erros detectados por linha do frame e outro que possui a quantidade de erros não corrigidos. Ao final execução da simulação um relatório é gerado, descrevendo a leitura dos dois registradores citados. Neste relatório, de mesmo formato da Figura 29, serão exibidas as informações de leitura dos registradores: esperada e realmente lidas.

# Resultados de Síntese

De maneira a verificar o correto funcionamento do hardware desenvolvido, todos os módulos do projeto foram sintetizados e prototipados em FPGA (Field Programmable Gate Array). Assim, pode-se verificar dois elementos importantes ao projeto: (i) ocupação de área (slices, memória, etc.) e (ii) se as restrições de timing puderam ser alcançadas. Para a etapa de síntese utilizou-se os FPGAs Virtex 4 (FX100) e Virtex 5 (LX220t), os relatórios de área são apresentados nas tabelas 2 e 4, e de timing nas tabelas 1 e 3, sendo a primeira tabela de cada relatório referente ao projeto em Virtex 4 e a segunda em Virtex 5. A etapa de prototipação foi realizada somente em Virtex 4 visto a disponibilidade de placas, com capacidade suficiente para receber o protótipo de projeto.

Tabela – Relatório de timing para o FPGA virtex 5 (LX220t), com dados retirados dos logs do XST.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Slice Logic Distribution** | | | | |
| Speed Grade: -2 | | | | |
| Minimum period: **3.413ns** (Maximum Frequency: **292.985**MHz) | | | | |
| Minimum input arrival time before clock: 3.152ns | | | | |
| Maximum output required time after clock: 2.830ns | | | | |
| **Slice Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slice Registers | 23,893 | 138,240 | 17% |
| Number used as Flip Flops | 23,838 |  |  |
| Number used as Latches | 55 |  |  |
| Number of Slice LUTs | **43,149** | **138,240** | **31%** |
| Number used as logic | 42,497 | 138,240 | 30% |
| Number using O6 output only | 42,088 |  |  |
| Number using O5 output only | 127 |  |  |
| Number using O5 and O6 | 282 |  |  |
| Number used as Memory | 88 | 36,480 | 1% |
| Number used as Dual Port RAM | 80 |  |  |
| Number using O6 output only | 80 |  |  |
| Number used as Single Port RAM | 8 |  |  |
| Number using O5 and O6 | 8 |  |  |
| Number used as exclusive route-thru | 564 |  |  |
| Number of route-thrus | 706 | 276,480 | 1% |
| Number using O6 output only | 687 |  |  |
| Number using O5 output only | 17 |  |  |
| Number using O5 and O6 | 2 |  |  |
| **Slice Logic Distribution** |  |  |  |
| Number of occupied Slices | **15,431** | **34,560** | **44%** |
| Number of LUT Flip Flop pairs used | 49,850 |  |  |
| Number with an unused Flip Flop | 25,957 | 49,850 | 52% |
| Number with an unused LUT | 6,701 | 49,850 | 13% |
| Number of fully used LUT-FF pairs | 17,192 | 49,850 | 34% |
| Number of unique control sets | 1,984 |  |  |
| Number of slice register sites lost         to control set restrictions | 2,767 | 138,240 | 2% |

# Árvore de diretórios do projeto X10 Giga (SVN)

Nesta Seção está descrita a árvore de diretórios do projeto X10GIGA no servidor de SVN.

* **Constraints 🡪** Diretório que contém as constraints do projeto. No arquivo *xc4vfx100-ff1152-11.ucf*
* **Documentação 🡪** Contém toda a documentação do projeto nas pastas a seguir:
  + **Apresentações 🡪** Apresentações e slides em PowerPoint do projeto, do padrão, etc;
  + **Datasheet 🡪** Contém o datasheet do projeto, com as especificações detalhadas de cada módulo e sistema;
  + **FPGAs 🡪** Manuais e UserGuides dos FPGAS Virtex4 e Virtex5;
  + **Interface-SE 🡪** Está o andamento dos registradores para interface, assim como a documentação dos mesmos;
  + **Normas 🡪** As recomendações pela ITU-T sobre a OTN, incluindo um Tutorial de OTN;
  + **Posters 🡪** Posters sobre o SONET, SDH, OTN, 10GbE, etc;
  + **TCCs 🡪** Encontram-se aqui os Trabalhos de Conclusão pertinentes ao projeto, sobre o Framer, O Reed-Solomon e a Interface Cliente;
* **HDL 🡪** Diretório raiz de todos os arquivos fonte (.vhd) do projeto em si. Na raiz possui também o top de todo x10:
  + **Client 🡪** A interface cliente, as filas assíncronas e o deslocador, encontram-se aqui:
    - **ExtratorCargaUtil 🡪** Modulo Extrator de Carga Úitil, controle de justificativa e pulso de referência;
    - **InsersorCargaUtil 🡪** Modulo que insere a Carga útil, o cabeçalho e outros dados de controle;
  + **Global 🡪** Todos arquivos fontes que são importantes para todo projeto, como o package com constantes, funções e designs prontos;
  + **Interface-SE 🡪** Contém o banco de registradores, módulo de Bip8, extrator de cabeçalho e o interface SE;
  + **Package 🡪** Aqui estão os packages de partes específicas do projeto, como Package do extrator, package do estrator, etc;
  + **Receive 🡪** A implementação interna do Receive encontra-se aqui, seu top está na raiz;
    - **Decoder 🡪** Contém o decoder para o Reed Solomon;
    - **Framer 🡪** Módulos dos frames:
      * **Alinhador 🡪** Possui os comparadores e deslocadores para fazer o alinhamento;
      * **Scrambler 🡪** Possui o código do embaralhador utilizado no receive;
  + **Send 🡪** Estão os códigos do enconder do RS para envio e o top da trasmissão;
* **Simulação 🡪** Todos scripts de simulação e dados de entrada encontram-se aqui, incluindo um script para compilação universal:
  + **Client 🡪** Simulações da parte de Cliente:
    - **ExtratorCargaUtil 🡪** Simulações do Extrator de carga útil, script de simulação:
      * **JustExaustivo 🡪** Arquivos de entrada do teste exaustivo;
      * **JustNegative 🡪** Arquivos de entrada do teste de justificativa negativa;
      * **JustNominal 🡪** Arquivos de entrada do teste de justificativa nominal;
      * **JustNothing 🡪** Arquivos de entrada do teste sem justificativa;
      * **JustPositive 🡪** Arquivos de entrada do teste de justificativa positiva;
    - **InsersorCargaUtil 🡪** Simulação do insersor de carga útil, scripts:
      * **Inputs 🡪** Arquivos de entrada;
  + **Geral 🡪** Simulação geral;
  + **Interface-SE 🡪** Encontram-se aqui as simulações da interface com o Software Embarcado:
    - **Cenario1**
    - **Cenario2**
    - **Cenario3**
    - **Cenario4**
    - **Cenario5**
    - **Cenario6**
    - **Cenario8**
    - **Cenario9**
    - **Cenario12**
    - **Cenario13**
    - **Cenario-Generico 🡪** Cenário customizável, contém arquivos de simulação e as instruções para o testbench;
  + **Testbench 🡪** Aqui estão os arquivos fontes dos TBs acima:
    - **Cenario-Generico 🡪** Testbench genérico que pode ser customizável a partir de um arquivo de configurações;
    - **Testbenches-Client 🡪** Fonde (.vhd) dos testbenches do cliente;
      * **ExtratorCargaUtil**
      * **InsersorCargaUtil**
* **Síntese 🡪** Scripts de sínteses.

# Problemas Conhecidos

Problema no banco de registradores 🡪 erro de leitura por parte SW no registrador OTU\_STS.

# Pendências do Projeto

Nesta Seção estão descritas as pendências referentes à entrega parcial do transponder X10.

1. PN11 e PRBS: Estes dois padrões de dados referem-se a registradores de deslocamento (LFSR) que são utilizados para detecção de alarmes, dentre eles o alarme de AIS (Alarm Indication Signal). Os dados destes padrões encontram-se preenchidos em memórias (visto que o LFSR funciona serialmente, e o tamanho da palavra do projeto é de 64 bits) bem como a parte de detecção e alinhamento para ser feita a comparação com dados recebidos. Falta ser feita a análise do número de erros de bit recebidos por estes padrões.
2. Registrador com a contagem de erros aferidos pelo módulo FEC: Os registradores relativos ao FEC encontrados no banco de registradores são: FEC\_STS\_PTR, FEC\_CORRERR\_CNT\_HI, FEC\_CORRERR\_CNT\_LO e FEC\_UNCORR\_CNT. Eles são basicamente utilizados para a contagem do número de erros corrigidos em cada uma das 16 linhas de um frame, assim como o número de quantas vezes não foi possível corrigir totalmente os erros de uma linha.
3. O registrador FEC\_STS\_PTR é utilizado para saber sobre qual das linhas do frame são referenciados os valores dos contadores dos registradores FEC\_CORRERR\_CNT\_HI, FEC\_CORRERR\_CNT\_LO e FEC\_UNCORR\_CNT.
4. Os registradores FEC\_CORRERR\_CNT\_HI, FEC\_CORRERR\_CNT\_LO formam em conjunto um contador que reporta o número de bits corrigidos pelo módulo de FEC.
5. O registrador FEC\_UNCORR\_CNT possui o valor de quantas vezes não foi possível corrigir os erros de uma linha, ou seja, quando houve mais de 8 erros, limite máximo de erros corrigidos pelo módulo de FEC.
6. Incomming Alignment Error (IAE) 🡪 O grupo não sabe como implementar este sistema, novas reuniões deverão esclarecer o problema.
7. Fault Type and Fault Location (FTFL): Acabar a implementação da máquina de estados que controla as modificações e a leitura do banco de registradores no módulo Interface SE. A parte referente à implementação da máquina de estados no módulo Insersor de Cabeçalhos que controla a chegada dos dados do módulo Interface SE, assim como a montagem da informação completa e controles necessários para multiplexação dos dados para posterior inserção no quadro OTN já está pronta.
8. Quando ocorre algum dos erros de OTU\_TSF ou OTU\_SSF, não poderão ser ativados os contadores de erros.
9. Além das pendências citadas acima, a geração de padrões do PN11 é uma frente em aberto, visto que a G.709-2003 (página 54, cap.16.4.1) afirma que estes padrões não são obrigatórios.

# Referências

[CHA01] Chang, H.; Shung, C.; Lee, C. “A Reed-Solomon Product-Code (RS-PC) Decoder Chip for DVD Aplications”, IEEE Journal of Solid State Circuits, Vol. 36, No 2. Fevereiro de 2001, pp. 229-238.

[CHO90] Choi, D. “Frame Alignment in a Digital Carrier - A Tutorial”. IEEE [Communications Magazine](http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=35), Vol. 28, No 2. Fev 1990.

[LEE05] Lee, H. “A High-Speed Low-Complexity Reed-Solomon Decoder for Optical Communications”. IEEE Transactions on Circuits and Systems – II: Express Briefs, Vol. 52, No 8. Agosto de 2005, pp. 461-465.

[LIU84] Liu, K. “Architecture for VLSI Design of Reed-Solomon Decoders”. *IEEE Transactions on Computers*, Vol. C-33, No 2. Fevereiro de 1984, pp. 178-189.

[MAS69] Massey, J. “Shift-register syntesis and BCH decoding.”. *IEEE Transactions on Information Theory*, Vol. IT-15, No 1. Janeiro de 1969, pp. 122-127.

[PAI07] Paiva, R. e Marczak, S. “Desenvolvimento de Módulos de Hardware para Recepção e Transmissão de Quadros OTN”. Trabalho de Conclusão de Curso de Graduação em Engenharia de Computação, Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS, Dez 2007.

[PAR05] Park, T. “Design of the (248,216) Reed-Solomon Decoder with Erasure Correction for Blu-ray Disc”. *IEEE Transactions on Consumer Eletronics*, Vol. 51, No 3. Agosto de 2005, pp. 872-878.

[REE91] Reed, I.; Shih, M.; Truong, T. “VLSI design of inverse-free Berlekamp-Massey”. IEEE Computer and Digital Techniques, Vol. 138, No 5. Setembro de 1991, pp. 295-298.

[SAR01] Sarwate, D.; Shanbhag, N. “High-Speed Architecture for Reed-Solomon Decoders”. *IEEE Transactions on Very Large ScaleIintegration Systems*. Vol. 9, No 5. Outubro de 2001, pp. 641-655.

[SIL07] Silva, A. e Rodolfo, T. “Implementação de uma arquitetura Reed-Solomon para uso em Redes OTN 10.7 Gbps”. Trabalho de Conclusão de Curso de Graduação em Engenharia de Computação, Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS, Dez 2007.

[TER08] TERACOM, “X10GIGA Functional Description”. Capturado em: http://www.inf.pucrs.br/~gaph/x10giga/TERACOM/x10Giga\_functional.html, Nov 2008.

[VIS07] Vissers, M. “Optical Transport Network & Optical Transport Module“. Capturado em: <http://ties.itu.ch/ftp/public/itu-t/tsg15opticaltransport/OTN/g709-intro-v2.ppt>, Ago 2007.

[WAI05] Wai, K.; Yang S. “Field Programmable Gate Array Implementation of Reed-Solomon Code, RS(255,239)”. Capturado em: www.ce.rit.edu/~sjyeec/papers/nyworkshop-rs-codec.pdf, Outubro de 2005.

[WIL99] Wilhelm, W. “A New Scalable VLSI Architecture for Reed-Solomon Decoders”. *IEEE Journal of Solid State Circuit*, Vol. 34, No 2. Março de 1999, pp. 388-396.

# Anexo I

Nesta Seção apresenta-se um exemplo de arquivo de configuração para geração de frames. A seguir temos um exemplo de arquivo de configuração para o cenário de teste 1, que realiza uma verificação dos alarmes do padrão OTN no banco de registradores do transponder.

FILE\_OUT = frames\_0a10

BASE\_OUT = HEXBIN

GEN\_MEM\_MS = NO

PRINT\_AFTER = NO

RAMB\_OUT = NO

ASSINATURA = YES

TEST\_SIGNAL = NULL

FEC = YES

NUMERRORS = 0

TIPERRORS = RAJADA

ALIGNMENT = 0

SCRAMBLING = YES

FAS = YES

MFAS = YES

SM\_TTI = 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77…

SM\_BIP8 = 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22…

SM\_ALARMS = 83 03 13 73 A3 D3 B3 B3 B3 B3 B3 83 83 83 83 83 83 83 83 83 83 83 83 83…

GCC0 = 4444 4444 4444 4444 4444 4444 4444 4444 4444 4444 4444 4444 4444 4444…

RES1 = 5555 5555 5555 5555 5555 5555 5555 5555 5555 5555 5555 5555 5555 5555…

RES2 = 666666 666666 666666 666666 666666 666666 666666 666666 666666 666666…

TCM\_ACT = 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77…

TCM6\_TTI = 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88…

TCM6\_BIP8 = 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99…

TCM6\_ALARMS = AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA…

TCM5\_TTI = BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB…

TCM5\_BIP8 = CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC…

TCM5\_ALARMS = DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD…

TCM4\_TTI = EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE…

TCM4\_BIP8 = FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF…

TCM4\_ALARMS = 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A…

FTFL = 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B…

TCM3\_TTI = 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C 2C…

TCM3\_BIP8 = 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D 3D…

TCM3\_ALARMS = 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E…

TCM2\_TTI = 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F…

TCM2\_BIP8 = A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6 A6…

TCM2\_ALARMS = B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7…

TCM1\_TTI = C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8 C8…

TCM1\_BIP8 = D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9 D9…

TCM1\_ALARMS = E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0 E0…

PM\_TTI = 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77…

PM\_BIP8 = 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A…

PM\_ALARMS = 83 01 11 71 A1 D1 B1 B1 B1 B1 B1 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7 B7…

EXP = 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C 4C4C…

GCC1 = 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D 5D5D…

GCC2 = 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E 6E6E…

APS\_PCC = 7F7F7F7F 7F7F7F7F 7F7F7F7F 7F7F7F7F 7F7F7F7F 7F7F7F7F 7F7F7F7F…

RES3 = 808080808080 808080808080 808080808080 808080808080 808080808080…

PSI = 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55…

JUSTIF\_GEN = NO