TIAGO AMARO MARTINS

DESENVOLVIMENTO DE UM *FRAMEWORK* PCI *EXPRESS* PARA FPGA

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos,
Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em Eletrônica

ORIENTADOR: Prof. Dr. Alberto Tannús

São Carlos
2015

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

M379d

Martins, Tiago Amaro
Desenvolvimento de um framework PCI Express para FPGA / Tiago Amaro Martins; orientador Alberto Tannús. São Carlos, 2015.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2015.

1. FPGA. 2. PCI Express. 3. framework. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Tiago Amaro Martins

Título: "Desenvolvimento de um framework PCI Express para FPGA"

Trabalho de Conclusão de Curso defendido e aprovado
em 23/06/2015,

com NOTA 9,7 (NOVE, SETE), pela Comissão Julgadora:

Prof. Dr. Alberto Tannús - (Orientador - IFSC/USP)

Prof. Associado Carlos Dias Maciel - (SEL/EESC/USP)

Prof. Associado Evandro Luís Linhari Rodrigues - (SEL/EESC/USP)

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Homero Schiabel

Agradecimentos

Primeiramente agradeço de forma carinhosa aos meus pais, Mateus José Martins e Virginia da Conceição Amaro Martins por toda dedicação, apoio, compreensão e respeito, assim como por tudo que eles me ensinaram durante esses anos de minha vida.

Ao Prof. Dr. Alberto Tannús, pela orientação, confiança e pelas oportunidades que me foram dadas.

Ao Dr. Edson Luiz Géa Vidoto pela amizade, ajuda e suporte no desenvolvimento desse trabalho.

Aos meus amigos, em especial ao Bruno Fernando Mendonça Callegaro, João Alberto Baccarin Robles Tardelli e Mario Augusto Kushima, os quais estiveram presentes durante os desafios de graduação.

À Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP), à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelas bolsas concedidas.

Resumo

A demanda de produtos eletrônicos mais eficientes e mais rápidos é uma característica do cenário mundial. Essa demanda força a comunicação entre dispositivos com taxas cada vez mais altas. Essas velocidades não permitem que utilizemos apenas processadores para execução dessas funções, necessitando de circuitos dedicados, deixando apenas os protocolos de alto-nível para os processadores. Estes, hoje em dia, dispõem de poucas interfaces para o mundo externo como USB (*Universal Serial Bus*), *Ethernet* e PCI *Express* (*Peripheral Component Interconnect Express*), sendo essa última a mais rápida disponível. O desenvolvimento de um circuito de interface PCI *Express* apresenta grande dificuldade, porém com o advento das FPGAs (*Field-Programmable Gate Array*) é possível o projeto dessa interface de maneira mais fácil e rápida. Um sistema de alta velocidade com uma biblioteca de utilização em *software* e canais de fácil conexão em *hardware* possui uma gama grande de aplicações e um apelo por parte de desenvolvedores tanto de *hardware* quanto de *software*. Assim, o objetivo desse projeto foi a criação de um *framework* que viabilize a comunicação entre um computador de mesa, utilizando a interface PCI *Express*, e uma placa com FPGA, dando ao desenvolvedor dos dispositivos, um canal de comunicação rápido, com velocidades na ordem de 1 GB/s e ainda fácil de ser ligado a qualquer projeto. Esse *framework* foi desenvolvido conseguindo-se 0,9 GB/s para a leitura e 1,5 GB/s para a escrita, utilizando dutos de 128 *bits* para DMA (*Direct Memory Access*) e para I/O (Entrada e Saída) que encobrem o protocolo e os circuitos de comunicação. As seis chamadas criadas em *software* são capazes de realizar a comunicação diretamente da camada de usuário do sistema operacional, ocultando toda a complexidade das chamadas ao sistema.

Palavras-chave: FPGA, PCI *Express*, *framework*, DMA.

Abstract

The demand for more efficient and faster electronic products is a characteristic of the global scenario. This demand forces each time more a higher communication rate between devices. At these speeds, processors cannot handle all the communication by themselves, dedicated circuitry is necessary to have the processors handling only the high level protocols. Now a day, processors have only a few external world interfaces like USB (Universal Serial Bus), Ethernet and PCI Express (Peripheral Component Interconnect Express), with the latter being the fastest one. The development of a circuit with PCI Express interface is quite complicated, but the advent of FPGAs (Field-Programmable Gate Array) has made the creation of projects with this interface easier and faster. A high speed system with a software library and easy to use hardware channels has multiple applications and an appeal by the software and hardware developers. Nevertheless, the main goal of this project was to create a framework to make viable a communication between desktop computer and FPGA using PCI Express, giving device developers a fast and easy to use communication channel with speeds around 1 GB/s. This framework was developed and achieved read speeds of 0.9GB/s and write speeds of 1.5GB/s, using 128 bits DMA and I/O buses, masking the protocol and the communication circuitry. The six software calls developed can be used directly from the user space of the operational system, hiding all the complexity of system calls.

Keywords: FPGA, PCI *Express*, *framework*, DMA.

Índice de Figuras

Figura 1 - Estrutura de uma FPGA [18]. 19
Figura 2 - Esquemático de uma CLB simples [5]. 20
Figura 3 - Esquemático do encadeamento entre várias células lógicas [5]. 21
Figura 4 - Esquemático de interconexão especial para o *carry* [5]. 21
Figura 5 - Esquemático do encadeamento de células lógicas com células de entrada e saída [5]. 22
Figura 6 - Esquemático do bloco de memória com as células lógicas da FPGA [19]. 24
Figura 7 - Comparação entre os padrões PCI e PCI *Express* [21] (Adaptada). 25
Figura 8 - Exemplo de topologia PCI *Express* [7]. 26
Figura 9 - Diagrama das camadas PCI *Express* em uma comunicação entre dois dispositivos [23]. 27
Figura 10 - Diagrama da comunicação entre duas camadas físicas de dois dispositivos PCI *Express* [23] (Adaptado). 28
Figura 11 - Representação de *links* e *lanes* na comunicação PCI *Express* [20]. 28
Figura 12 - Conector PCI *Express* para conexão x1 [22]. 29
Figura 13 - Conector PCI *Express* x1 em uma placa de circuito [22]. 30
Figura 14 - Diagrama da comunicação entre duas camadas de vinculação de dados de dois dispositivos PCI *Express* [23]. 30
Figura 15 - Diagrama de empacotamento da camada de vinculação de dados [23]. 31
Figura 16 - Diagrama da comunicação entre duas camadas de transação de dois dispositivos PCI Express [23]. 32
Figura 17 - Diagrama do empacotamento da camada de transação [23]. 32
Figura 18 - Exemplo de TLP de requisição de escrita de memória com endereçamento de 32 *bits* [9]. 34
Figura 19 - Exemplo de TLP de requisição de leitura de memória com endereçamento de 32 *bits* [9]. 35
Figura 20 - Exemplo de TLP de complementação [9]. 35
Figura 21 - Diagrama do espaço de configuração PCI [27]. 37
Figura 22 - Diagrama geral das relações entre as camadas do PCI *Express* [24]. 38
Figura 23 - Exemplo de utilização de DMA (AMD DirectGMA) [28]. 39
Figura 24 - Kit de desenvolvimento Terasic TR4 [25]. 41
Figura 25 - Diagrama da ideia do *framework* desenvolvido. 42
Figura 26 - Diagrama de blocos básicos de uma comunicação PCI *Express*. 42
Figura 27 - Conexão PCI *Express* entre o Kit de Desenvolvimento TR4 e um computador *desktop* [26]. 43
Figura 28 - Diagrama de blocos dos módulos de *hardware*. 44
Figura 29 - Configurações do compilador de IP para PCI *Express* da Altera®. 45

Figura 30 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 3 DWs não alinhadas [13]. 47
Figura 31 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 3 DWs alinhadas [13]. 48
Figura 32 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 4 DWs não alinhadas [13]. 48
Figura 33 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 4 DWs alinhadas [13]. 49
Figura 34 - Forma de onda da interface Avalon® de 128 *bits* para recepção PCI *Express* com 3 DWs alinhadas [13]. 50
Figura 35 - Diagrama do módulo de DMA de transmissão. 51
Figura 36 - Diagrama do módulo de DMA de recepção. 53
Figura 37 - Diagrama de blocos dos módulos de *software*. 55
Figura 38 - Captura de tela com os valores de resultado da compilação do *framework*. 61
Figura 39 - Programa de teste do *framework* PCI *Express* para Windows com a imagem original enviada para a FPGA. 64
Figura 40 - Programa de teste do *framework* PCI *Express* para *Windows* com o resultado do processamento da imagem pela FPGA. 65

Índice de Tabelas

Tabela 1 - Desempenho teórico do PCI *Express*...24
Tabela 2 - Descrição do campo *Fmt* do cabeçalho de um TLP [8]..33
Tabela 3 - Resultados da síntese do *framework* com o programa Quartus®...........................62
Tabela 4 - Valores de velocidade para as transferências utilizando o *framework* PCI *Express* com 64KB de *buffer* no *driver*...66
Tabela 5 - Valores de velocidade para as transferências utilizando o *framework* PCI *Express* com 4KB de *buffer* no *driver*...67

Sumário

1 Introdução 17
1.1 Justificativa 17
1.2 Objetivo 17
1.3 Organização do Trabalho 18
2 Embasamento Teórico 19
2.1 *Field-Programmable Gate Array* 19
2.1.1 Blocos de Lógica Programável 20
2.1.2 Interconexões 20
2.1.3 Células de Entrada e Saída 22
2.1.4 Memória Interna 23
2.2 *Peripheral Component Interconnect Express* 24
2.2.1 Topologia 25
2.2.2 Camadas do protocolo 26
2.2.3 Compatibilidade 36
2.2.4 Espaço de Configuração 37
2.2.5 Panorama Geral 38
2.3 *Direct Memory Access* 39
3 Desenvolvimento 41
3.1 Visão Geral 42
3.2 *Hardware* 43
3.2.1 PCIe *Hard* IP 45
3.2.2 Configuração do IP 46
3.2.3 *Reset* de Sincronismo 46
3.2.4 Condicionamento da Transmissão 46
3.2.5 Condicionamento da Recepção 49
3.2.6 DMA de Transmissão 50
3.2.7 DMA de Recepção 52
3.2.8 Lógica do Usuário 54
3.3 *Software* 54
3.3.1 Aplicação de *Software* 55
3.3.2 Biblioteca 55
3.3.3 *Driver* 56
4 Resultados 61
4.1 *Hardware* 61
4.2 *Software* 63
4.2.1 *Driver* 63

4.2.2 Aplicações de teste 63
5 Conclusões 69
6 Trabalhos futuros 71
Referências Bibliográficas 73

1 Introdução

A reutilização de códigos de programação é uma prática muito empregada. Ferramentas e ambientes de programação promovem a simplificação do desenvolvimento de aplicações. O reaproveitamento de código chega a âmbitos nos quais uma aplicação pode ser desenvolvida utilizando apenas chamadas de um *framework*, como uma cola entre funcionalidades já existentes.

Um *framework* é um conceito da ciência de computação usado para descrever uma abstração que provê funcionalidades genéricas para usuários que venham a utilizar dessas funções para criar aplicações específicas [1]. O mesmo conceito pode ser aplicado para *hardware* ao se interpretar que um módulo pode ser genérico e prover funcionalidades que podem ser utilizadas para aplicações específicas.

Um módulo genérico capaz de executar tarefas complicadas através de chamadas simples, como um módulo de comunicação PCI *Express* (*Peripheral Component Interconnect Express*), pode ser um *framework* caso esse módulo possua funções genéricas que simplifiquem a comunicação e que possam ser utilizadas em várias aplicações específicas que necessitem de uma comunicação de alta velocidade.

1.1 Justificativa

Esse trabalho foi desenvolvido para servir como plataforma de comunicação para o espectrômetro de ressonância magnética do Centro de Imagens e Espectroscopia *in vivo* por Ressonância Magnética (CIERMag) devido ao requerimento de altas taxas de transferência dos dados adquiridos.

O espectrômetro do CIERMag já possui um *hardware* em *Field-Programmable Gate Array* (FPGA) e *software* rodando em computadores *desktop* que se comunicam através de *Ethernet Gigabit*, porém para uma expansão do número de receptores do espectrômetro a conexão atual não é capaz de suprir as velocidades de comunicação necessárias.

Dessa forma, a nova plataforma de comunicação será o *framework* PCI *Express* desse trabalho que além de proporcionar uma comunicação de alta velocidade, ainda proporciona facilidade para os desenvolvedores do resto do sistema que precisaram adaptar os sistemas existentes a nova comunicação.

1.2 Objetivo

Devido à necessidade atual de transferência de grandes quantidades de dados, há uma busca muito grande por conexões de alta velocidade. Dentre as várias tecnologias existentes,

o padrão de alta velocidade mais estabelecido no mercado é o PCI *Express*. Utilizando esse padrão com elementos de lógica digital reprogramáveis, é possível fazer o processamento de grandes volumes de informações de forma rápida.

O objetivo desse trabalho foi criar um *framework* de fácil utilização e alta velocidade, que seja composto por vários módulos, capaz de fazer a comunicação de um computador com uma FPGA utilizando PCI *Express*. Com esse bloco é possível mover grandes quantidades de dados sem que o usuário precise entender todo o processo de movimentação de informações entre as várias camadas do sistema tanto do *software* como do *hardware*.

1.3 Organização do Trabalho

O trabalho elaborado foi dividido em seis capítulos. O primeiro contém uma descrição, a justificativa e o objetivo desse trabalho. O segundo possui o embasamento teórico do projeto. No terceiro capítulo encontram-se os métodos e os detalhes do desenvolvimento desse trabalho. No quarto são apresentados os resultados obtidos. O quinto é composto pela conclusão referente ao trabalho realizado. Por fim, no sexto são apresentados os possíveis trabalhos futuros.

2 Embasamento Teórico

2.1 *Field-Programmable Gate Array*

Field-Programmable Gate Array (FPGA) é um circuito integrado cuja funcionalidade pode ser reconfigurada [2]. A essência desse circuito é conjunto de blocos de lógica programável (CLB, *Configurable Logic Blocks*), com interconexões reconfiguráveis e blocos de entrada e saída (I/O *Block*), como representado na Figura 1. Cada célula de lógica pode executar diversas operações booleanas e têm capacidade de armazenamento do resultado.

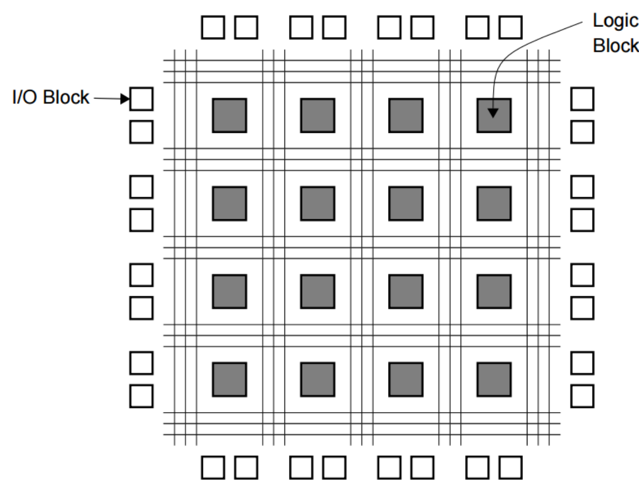

Figura 1 - Estrutura de uma FPGA [18].

Antes de existirem FPGAs, memórias programáveis somente de leitura (PROM, do inglês *Programmable Read-Only Memory*) eram utilizadas para criar funções de lógica combinacional programáveis, porém, por problemas de velocidade e consumo de energia outros dispositivos de lógica programável (PLD, *Programmable Logic Devices*) foram desenvolvidos com o objetivo de criar essa maleabilidade na utilização de circuitos digitais.

Dentre os vários tipos de PLDs existentes como PLA (*Programmable Logic Array*), PAL (*Programmable Array Logic*), GAL (*Generic Array Logic*), CPLD (*Complex Programmable Logic Devices*) [3] e FPGA, esse último diferencia-se dos demais por não possuir as portas lógicas, mas conter tabelas verdade que são configuradas com os resultados da lógica combinacional desejada. Além disso, a configuração das FPGAs é volátil, dessa forma é necessária uma memória externa capaz de armazena-la para quando a FPGA for desligada.

2.1.1 Blocos de Lógica Programável

A base de qualquer FPGA são as CLBs. Essas células de lógica contêm em sua essência uma pequena tabela verdade (LUT, *Look-Up Table*), um dispositivo de armazenamento (*flip-flop*) do tipo D [4] e um multiplexador, para desviar do *flip-flop* se desejado [5]. Cada família de FPGAs possui um tipo diferente de CLB, apesar de o princípio de funcionamento ser o mesmo para todas, as diferenças podem estar na LUT, ou em componentes extras como logica de vai um (*carry*) ou de registradores de deslocamento (*shifters*).

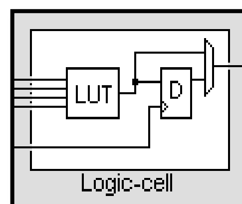

Figura 2 - Esquemático de uma CLB simples [5].

As LUTs são pequenas memórias de acesso randômico (RAM, *Random Access Memory*), que podem ser programadas com qualquer função lógica. Por serem memórias voláteis existe a necessidade de que toda a configuração seja guardada externamente a FPGA. Tradicionalmente as LUTs são de quatro entradas, como representado na Figura 2. Dessa forma é possível com uma única unidade, programar qualquer função lógica booleana de até quatro entradas. No caso da função lógica, a ser executada, conter um maior número de entradas do que a LUT básica, faz-se uma composição de LUTs, interligando uma CLB à outra.

2.1.2 Interconexões

Para executar a interligação entre as LUTs, como representado pela Figura 3, é necessário que existam caminhos programáveis. Esses caminhos basicamente são compostos por fios e multiplexadores e são colocados ao redor das células lógicas, formando dutos de comunicação. Esses dutos são fundamentais, pois é com eles que uma grande quantidade de células pode ser ligada em sequência a fim de constituir um sistema digital complexo.

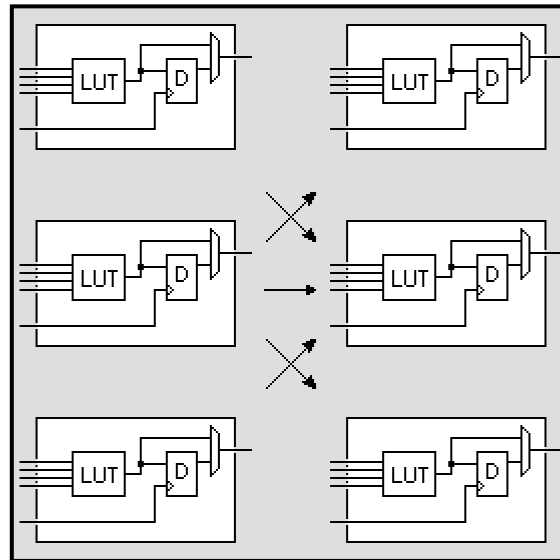

Figura 3 - Esquemático do encadeamento entre várias células lógicas [5].

Além dos caminhos que interligam uma célula a outra, as FPGAs possuem um caminho especial de ligação entre as células vizinhas, na qual cada célula possui uma conexão de saída e outra de entrada, de forma a proporcionar um acesso rápido para o sinal de *carry* que pode ser cascateado (Figura 4) para a realização de operações aritméticas. A utilização desses caminhos especiais possibilita o descongestionamento de outras linhas de interconexão proporcionando uma redução no número de CLB utilizadas e uma velocidade de operação maior.

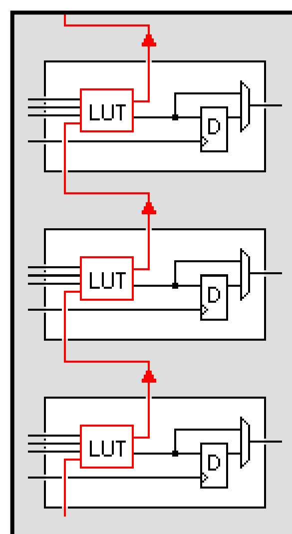

Figura 4 - Esquemático de interconexão especial para o *carry* [5].

Um dos grandes desafios na programação das FPGAs é a determinação das interconexões. O roteamento dessas conexões deve ser feito com muito cuidado de forma a

reduzir as distâncias que um sinal deve percorrer. Por essa razão os algoritmos de mapeamento e roteamento das FPGAs são bastante complexos e as ferramentas capazes de bem utilizar as interconexões são significativamente caras, cerca de milhares de reais por ano.

2.1.3 Células de Entrada e Saída

O terceiro componente básico de uma FPGA são as células de entrada e saída. Como todo circuito eletrônico há a necessidade de comunicação com o mundo exterior. Na FPGA os blocos de entrada e saída são responsáveis pela conexão da lógica com os pinos do encapsulamento dela assim como o condicionamento elétrico do sinal para que este possa entrar ou sair da FPGA (Figura 5).

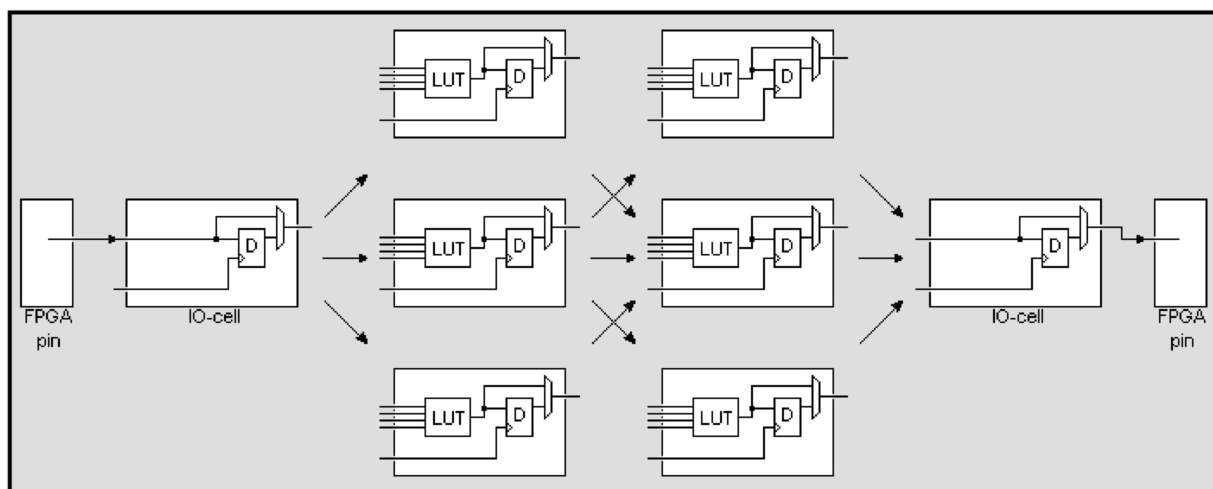

Figura 5 - Esquemático do encadeamento de células lógicas com células de entrada e saída [5].

As FPGAs possuem várias categorias de pinos, portanto apresenta também blocos de entrada e saída com diferentes finalidades. Os pinos da FPGA são divididos essencialmente em dois grupos: Pinos dedicados e pinos do usuário [6].

2.1.3.1 Pinos dedicados

Os pinos dedicados da FPGA são pinos cuja função atribuída a eles foi definida pelo fabricante e não pode ser alterada. Existem três subcategorias de pinos de entrada e saída:

- **Pinos de alimentação:** São pinos em que as tensões de operação da FPGA entram ou saem do circuito integrado. Podem ser divididos em dois tipos:
    - **Tensões do núcleo:** São pinos que devem ser alimentados com a tensão necessária para o funcionamento das LUTs e dos *flip-flops*.
    - **Tensões de Entrada e Saída:** São as tensões utilizadas pelas células de entrada e saída e devem ser compatíveis com os valores esperados pelos dispositivos externos que estão ligados a esses pinos.

- **Pinos de configuração:** São pinos utilizados para carregar os valores de configuração nas LUTs e nas interconexões para que a FPGA execute as funções desejadas.
- **Pinos de *clock* ou entradas dedicadas:** São pinos conectados em fios especiais dentro da FPGA, com o objetivo de carregar maior corrente com menor atraso possível. Muito utilizados para sinais de *clock* e com várias derivações.

2.1.3.2 Pinos do usuário

Os pinos do usuário, também chamados de pinos de I/O (do inglês *Input/Output*), são os pinos padrões da FGPA. Esse tipo de pino é configurável como entrada, saída ou bidirecional, nesse caso com *buffers tri-state*. Os pinos de I/O são conectados às células de entrada e de saída e são alimentados pelos pinos de força.

2.1.4 Memória Interna

A existência desses blocos de memória nas FPGAs é muito importante devido à necessidade de armazenamento de dados na maioria dos circuitos digitais. Existem duas formas principais de armazenamento em uma FPGA: *Blockram* e *Distributed RAM*.

Blockrams são blocos de memória localizados em certas posições da FPGA, normalmente são em pequeno número, na ordem de algumas centenas, mas são capazes de armazenar uma boa quantidade de dados. São memórias conectadas a estrutura de interconexão das células da FPGA para que dados possam ser armazenados e lidos desse bloco (Figura 6). Esses blocos de memória podem ser de uma ou múltiplas portas. Quanto maior o número de portas de uma memória, maior a possibilidade de acesso simultâneo da mesma. As memórias de múltiplas portas são muito utilizadas para transferir dados entre diferentes domínios de *clock*, uma vez que o dado pode ser escrito utilizando um *clock* e lido utilizando outro.

Distributed RAM são pequenos blocos de memória espalhados pela FPGA, eles utilizam as próprias células lógicas para armazenar os dados. A vantagem desse tipo de armazenamento é a sua maleabilidade uma vez que várias configurações de memória podem ser criadas com esses pequenos blocos. Porém essa configuração utiliza muitas células lógicas e não possui altas velocidades de leitura e escrita, uma vez que não são blocos especializados.

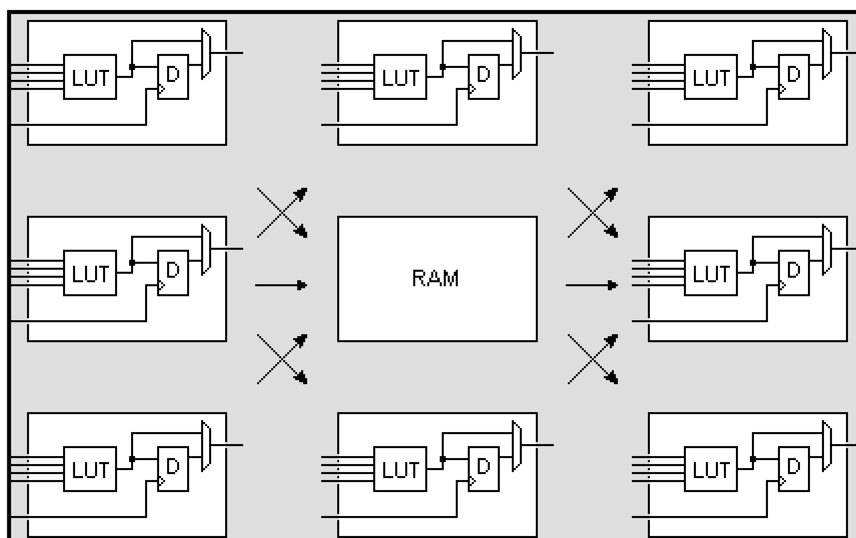

Figura 6 - Esquemático do bloco de memória com as células lógicas da FPGA [19].

2.2 *Peripheral Component Interconnect Express*

O PCI *Express* (*Peripheral Component Interconnect Express* - PCIe) é um padrão de comunicação serial de alta velocidade (Tabela 1), utilizado como duto de expansão para computadores. Esse padrão é o resultado de melhorias feitas em padrões antigos como o PCI (*Peripheral Component Interconnect*) e AGP (*Accelerated Graphics Port*). As principais modificações foram o aumento do *throughput*, diminuição do número de pinos de entrada e saída, melhor escalabilidade dos dispositivos, adição de um sistema de detecção de erros mais eficiente e adição da funcionalidade *hot-plug* nativamente [7].

Tabela 1 - Desempenho teórico do PCI *Express*

Versão do PCI Express	Codificação Utilizada	Taxa de transferência	Largura de Banda por *lane*	Largura de Banda (*link* x4)
1.0	8b/10b	2.5 GT/s	2 Gbit/s (250 MB/s)	8 Gbit/s (1 GB/s)
2.0	8b/10b	5 GT/s	4 Gbit/s (500 MB/s)	16 Gbit/s (2 GB/s)
3.0	128b/130b	8 GT/s	7,877 Gbit/s (984,6 MB/s)	31,508 Gbit/s (3,938 GB/s)
4.0	128b/130b	16 GT/s	15,754 Gbit/s (1969,2 MB/s)	63,015 Gbit/s (7,877 GB/s)

Apesar de grandes mudanças físicas e elétricas feitas entre os padrões antigos e o PCIe, este preserva uma compatibilidade de *software* com o PCI, tornando assim possível a utilização de dispositivos PCI *Express* em sistemas que não oferecem oficialmente suporte ao novo padrão, mas somente ao padrão antigo.

2.2.1 Topologia

A forma de operação do PCI *Express* é mais parecida com a operação de redes de computadores do que de um duto no qual trafegam dados de múltiplas fontes. O PCIe é baseado na topologia ponto-a-ponto (Figura 7), mas com conexões (*links*) seriais separados tornando possível que todos os dispositivos sejam ligados ao módulo raiz (*root complex* ou *host*).

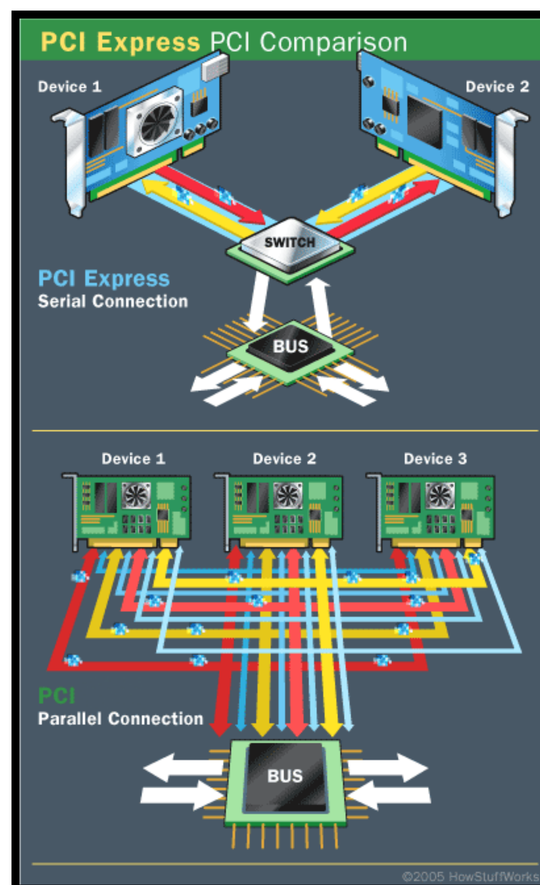

Figura 7 - Comparação entre os padrões PCI e PCI *Express* [21] (Adaptada).

O antigo PCI possui as linhas paralelas e compartilhadas entre todos os dispositivos o que faz com que o duto possa ser utilizado somente com um dispositivo mestre ao mesmo tempo. Com o padrão PCI *Express* é possível uma comunicação em *full-duplex* entre quaisquer duas extremidades (*endpoints*), sem nenhuma limitação devido ao acesso

concorrente de múltiplos *endpoints*. Ou seja, os dispositivos são capazes de transmitir e receber simultaneamente sem que exista conflito no duto utilizado.

Por possuir conexões separadas para cada um dos dispositivos, um módulo controlador do duto (*root complex*) necessita de roteadores de dados (*switches*) para que mais dispositivos possam ser conectados à mesma raiz.

Um exemplo comum de topologia PCI *Express* em um computador pode ser representado pela Figura 8. Tipicamente nos computadores, os módulos *root complex* e *switch* estão presentes diretamente na ponte norte da placa mãe. Como fisicamente as conexões não são compatíveis com os padrões mais antigos são necessárias pontes para fazer essa adaptação.

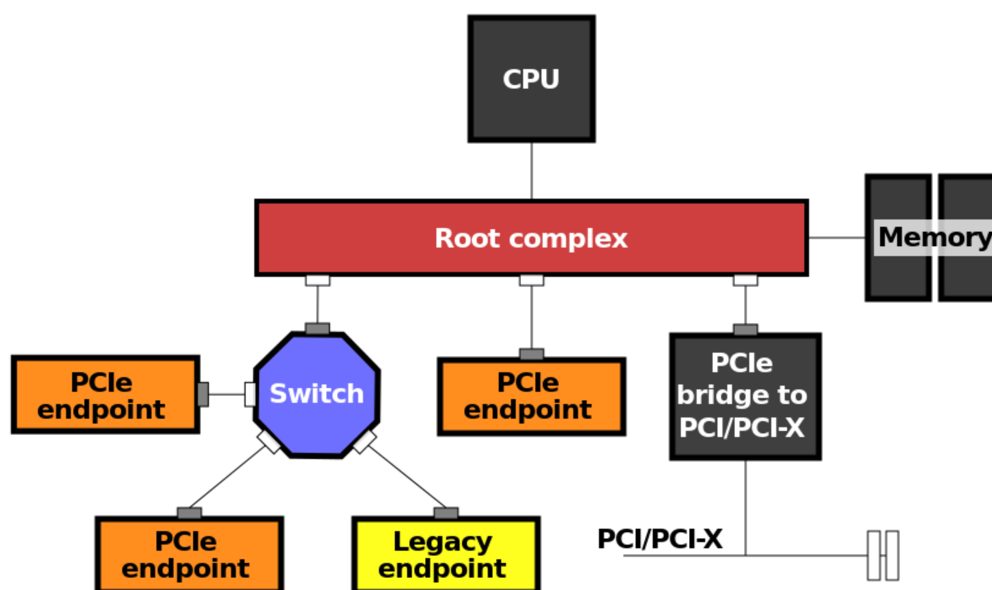

Figura 8 - Exemplo de topologia PCI *Express* [7].

2.2.2 Camadas do protocolo

O protocolo PCI *Express* é um protocolo de camadas, que pode ser dividido em camada de transação (*transaction layer*), camada de vinculação dos dados (*data link layer*) e camada física (*physical layer*). A comunicação entre dois dispositivos PCIe é dada pelo empacotamento e desempacotamento de dados por essas camadas como representado pela Figura 9. Os dispositivos PCI *Express* podem ser divididos em duas partes, o transmissor (TX) e o receptor (RX).

O transmissor é responsável pela formação dos pacotes obtidos pelo núcleo do dispositivo, que são armazenados em *buffers* até que a transmissão possa ocorrer. Os dados passam primeiramente pela camada de transação, depois pela camada de vinculação dos dados e por último pela camada física onde os dados são enviados pelo *link*.

O receptor é responsável pela decodificação dos pacotes que chegam pelo *link*. Os dados, por sua vez, seguem o fluxo inverso ao do transmissor. Ou seja, passam da camada física para a camada de vinculação dos dados e por último para a camada de transação onde o dado é extraído para ser processado pelo núcleo do dispositivo.

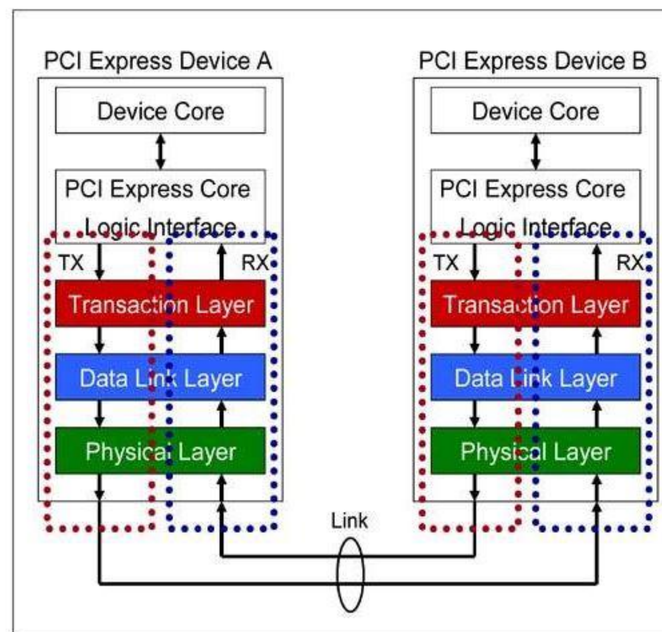

Figura 9 - Diagrama das camadas PCI *Express* em uma comunicação entre dois dispositivos [23].

2.2.2.1 Camada Física

Segundo a especificação do padrão PCIe, a camada física (costumeiramente chamada de PHY) é dividida em duas subcamadas, a elétrica e a lógica. Na subcamada elétrica são especificados alguns componentes como a existência de um serializador/deserializador (SerDes) e outros circuitos analógicos. Na subcamada lógica, entretanto, são especificados as formas de divisão dos dados em canais e o esquema de codificação desses dados, além dos dados adicionados para a definição da comunicação.

Os dados da camada física que são adicionados aos pacotes do PCI *Express* (PLP – *Physical Layer Packets*) são múltiplos de 4 *bytes* e contém informações de início e fim da transmissão. Esses pacotes são decodificados pela camada física do dispositivo que os recebeu, mesmo que o pacote não seja endereçado a ele, como representado pela Figura 10. Por serem decodificados a cada dois pontos os PLP não são propagados por um *switch* e dessa forma não possuem qualquer informação de roteamento dos dados.

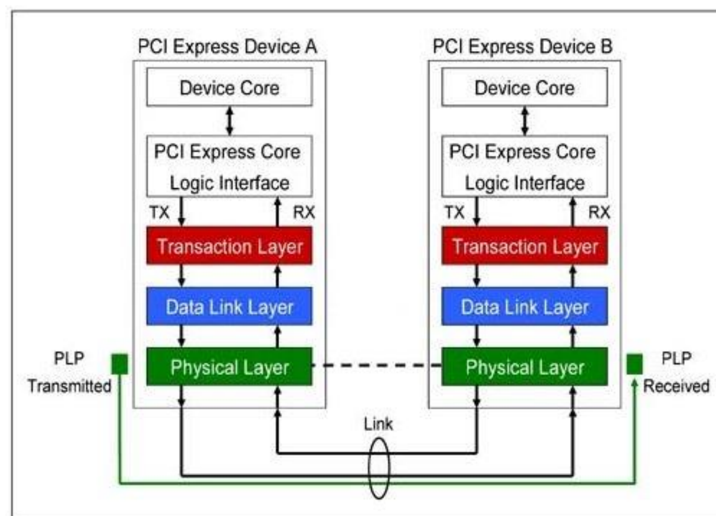

Figura 10 - Diagrama da comunicação entre duas camadas físicas de dois dispositivos PCI *Express* [23] (Adaptado).

2.2.2.1.1 *Lanes*

Cada *link* PCI *Express* possui um ou mais canais de comunicação chamados *lanes*, usualmente representado por *xN* onde *N* é o número de *lanes* por *link*. Cada *lane* contém dois pares de fios um par que envia e outro que recebe dados de forma balanceada. A taxa de transmissão de cada *lane* é um *bit* por ciclo, porém com a utilização de várias *lanes* em paralelo proporciona um número maior de *bits* por ciclo como representado pela Figura 11.

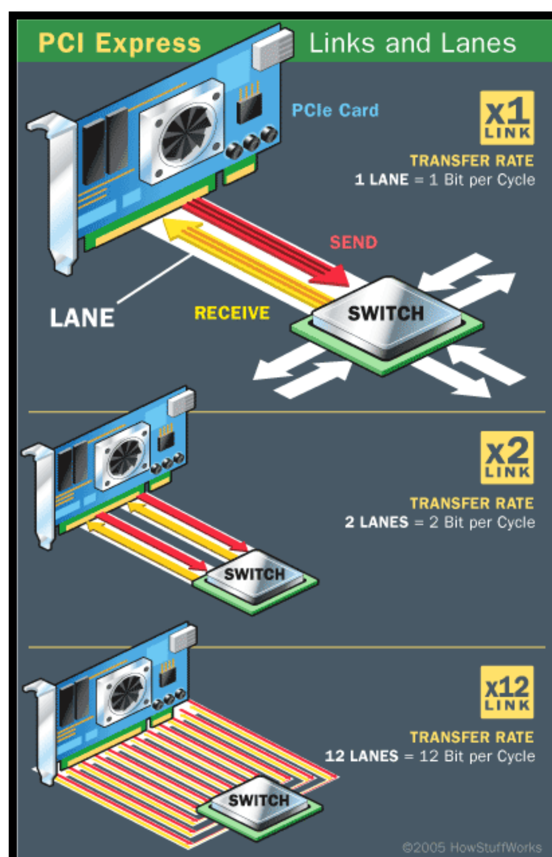

Figura 11 - Representação de *links* e *lanes* na comunicação PCI *Express* [20].

A transmissão dos dados em múltiplas *lanes* é feita de forma que cada *byte* sucessivo é transmitido em uma *lane* sucessiva. Apesar de adicionar uma complexidade maior no sincronismo dos dados recebidos, a latência é significativamente reduzida para pacotes grandes.

Devido as altas taxas de operação a codificação utilizada para cada *lane* é 8b/10b para as versões 1 e 2 do PCI *Express* e 128b/130b para a versão 3. Essa codificação garante balanceamento DC e recuperação do *clock*, permitindo ao dispositivo recebedor identificar as bordas de cada *bit*. Na codificação 8b/10b cada 8 *bits* são substituídos por 10 *bits* causando um *overhead* de 20% devido a codificação. Por esse motivo a versão 3 implementa a codificação 128b/130b para que o *overhead* seja reduzido para 1,5%.

2.2.2.1.2 Conector

Os conectores PCI *Express* variam de tamanho dependendo do número de *lanes* que são suportados pelo dispositivo. Além disso, os conectores também podem variar dependendo do tipo de aplicação, por exemplo: se for um conector interno (Figura 12), uma placa ligada diretamente a outra, ou um conector externo no qual as placas são ligadas por um cabo.

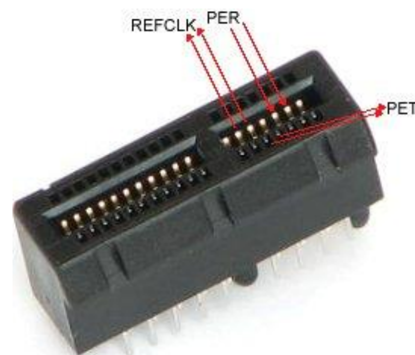

Figura 12 - Conector PCI *Express* para conexão x1 [22].

Todos os conectores internos PCIe possuem 22 pinos e um dente de separação física. Os pinos após a separação física também variam em função das *lanes* suportadas. Os sinais do PCIe são diferenciais e estão sempre em pares, os sinais de recepção (PER – PCI *Express Receive*) são colocados de um lado e os de transmissão (PET – PCI *Express Transmit*) de outro. Para um conector com mais do que uma *lane*, os sinais de dados são colocados em ordem crescente com um conjunto de pinos de terra como separação. O sinal de *clock* (REFCLK – *reference clock*) diferencial também é enviado pelo conector, mas não é duplicado se o *link* tiver mais do que uma *lane*.

Nos casos em que uma placa é conectada a outra, utiliza-se a própria placa como conector como representado pela Figura 13. É possível observar pela imagem os sinais diferenciais de transmissão de dados que são as duas trilhas saindo da FPGA e indo até a borda da placa como mostrado pela elipse vermelha.

Figura 13 - Conector PCI *Express* x1 em uma placa de circuito [22].

2.2.2.2 Camada de Vinculação dos Dados

A camada de agrupamento de dados, chamada em inglês de *Data Link Layer*, é a responsável pela verificação dos dados e pela retransmissão caso necessário. Os dados dessa camada são decodificados somente pela mesma camada no outro dispositivo, portanto os pacotes (DLLP – *Data Link Layer Packets*) trafegam como representado pela Figura 14.

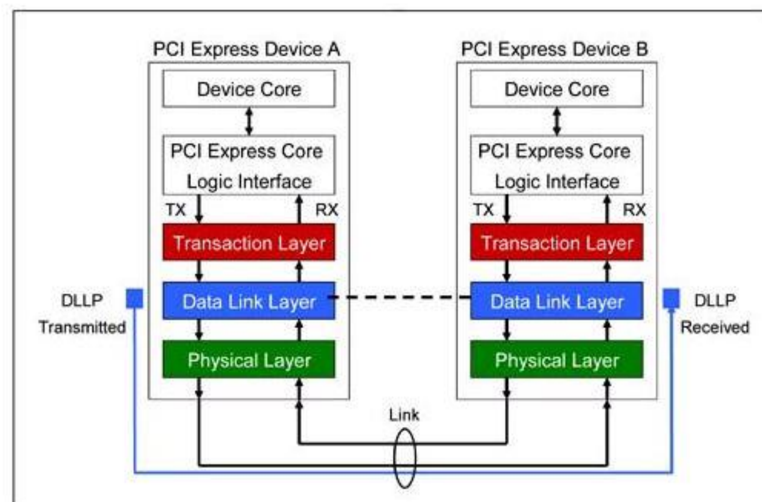

Figura 14 - Diagrama da comunicação entre duas camadas de vinculação de dados de dois dispositivos PCI *Express* [23].

Como os DLLP necessariamente precisam passar pela camada física, estes são empacotados pela camada física como representado pela Figura 15. Assim como os PLPs, os DLLPs são empacotados e desempacotados a cada dois pontos, ou seja, esses pacotes não passam por *switches*.

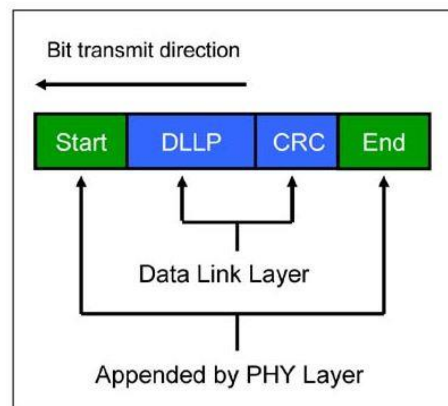

Figura 15 - Diagrama de empacotamento da camada de vinculação de dados [23].

Com um tamanho fixo de 8 *bytes* os DLLPs transportam as seguintes informações:

- Confirmações de recebimento (com ACK e NAK).
- Gestão de energia (PMx).
- Informações de fluxo de controle (FCx).

Além disso, é utilizado um CRC de 16 *bits* para o receptor verificar erros dos pacotes transmitidos. Com o encapsulamento da camada física mais 1 *byte* de início e 1 *byte* de fim são adicionados ao pacote total (Figura 15).

A existência das confirmações de recebimento, através dos sinais ACK (*acknowledged*) e NAK (*not acknowledged*) presentes no DLLP, garante o correto recebimento do pacote. Em raras situações onde há a necessidade de retransmissão do pacote, ou seja, através do CRC detecta-se um erro, o dispositivo transmissor é informado com um sinal de NAK para que o procedimento de transmissão possa recomeçar. Caso a resposta enviada seja ACK o *buffer* do transmissor é esvaziado para que outro pacote possa ser transmitido.

Para que o transmissor possa saber a quantidade de dados que o receptor ainda pode receber e armazenar no seu *buffer* os FCx DLLP são transmitidos periodicamente do receptor para o transmissor, informando através de um sistema de crédito, quanto espaço ainda há disponível no *buffer* do canal virtual do receptor. Dessa forma, a transmissão só ocorre quando o receptor tiver espaço para armazenar dados.

2.2.2.3 Camada de transação

A última camada é a de transação, ou *Transaction Layer* em inglês. Está e a camada responsável por cuidar de cada um dos dados que são recebidos ou serão transmitidos. Assim como anteriormente os dados gerados são lidos somente por uma camada de mesmo tipo. O percurso dos pacotes (*Transaction Layer Packets* - TLP) gerados por esse tipo de camada é representado pela Figura 16.

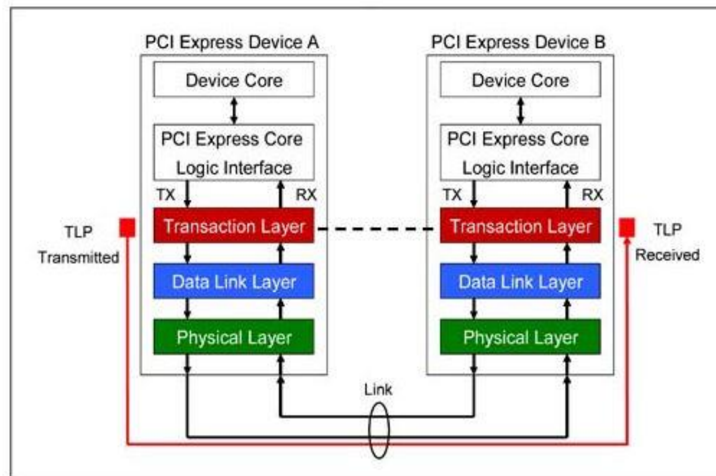

Figura 16 - Diagrama da comunicação entre duas camadas de transação de dois dispositivos PCI Express [23].

Por ser a camada que empacota e gerencia os dados, é também a camada mais interna do PCI *Express* como pode ser observado na Figura 17. O pacote TLP tem como componentes campos de cabeçalho (em inglês *header*), de dados, que é opcional, e um de verificação de erro de ponta a ponta (*End-to-End* CRC - ECRC) também opcional.

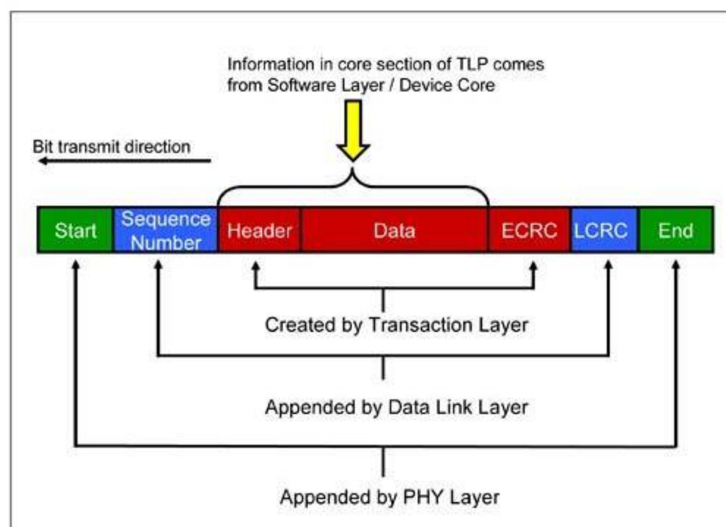

Figura 17 - Diagrama do empacotamento da camada de transação [23].

O cabeçalho do pacote define sua funcionalidade, a existência de dados, informações do transmissor e do receptor entre outras informações. Basicamente o TLP contém todas as informações, providas pelo sistema operacional e pelo *software* que deseja efetuar a comunicação, para que o pacote seja transmitido para o local correto.

Como os pacotes PCI *Express* podem conter vários *bytes* de tamanho o cabeçalho contém o comprimento desse pacote. A unidade básica escolhida para a medição do tamanho de um pacote foi de 32 *bits* ou um *Double Word* (DW). Dessa forma, qualquer dado a ser transmitido pelo PCIe deve ser múltiplo de 32 *bits*. O tamanho do cabeçalho do TLP é 3 ou 4 DWs dependendo se o endereçamento é de 32 ou 64 *bits*.

Um dos campos do cabeçalho é chamado de *Fmt* e tem como responsabilidade informar o tamanho do cabeçalho e se o pacote possui dado ou não. Como descrito na Tabela 2.

Tabela 2 - Descrição do campo *Fmt* do cabeçalho de um TLP [8].

Fmt	**Formato do TLP**
00	Cabeçalho de 3 DWs, sem dado
01	Cabeçalho de 4 DWs, sem dado
10	Cabeçalho de 3 DWs, com dado
11	Cabeçalho de 4 DWs, com dado

Seguindo o campo *Fmt* tem-se o campo *Type* o qual descreve o tipo de operação do TLP. Os valores de *Type* podem ser diferentes dependendo do seu significado, mas o mais importante é o valor $0x00$, que representa um pacote de acesso a memória, e o $0x0A$, que representa um pacote de complementação.

Com os campos *Fmt* e *Type* é possível definir diferentes tipos de pacotes dividindo-os por função, como por exemplo, pacotes de escrita de memória, de leitura de memória e de complementação.

2.2.2.3.1 Pacote de escrita de memória

As informações carregadas pelo cabeçalho variam dependendo do tipo de pacote que se deseja transmitir, porém existem campos como o de um formulário no qual as informações devem ser preenchidas para que ocorra o correto funcionamento do sistema. Um exemplo de um pacote de escrita de memória é representado pela Figura 18. Neste exemplo supõe-se a escrita do valor $0x12345678$ no endereço físico $0xFDAFF040$ [9].

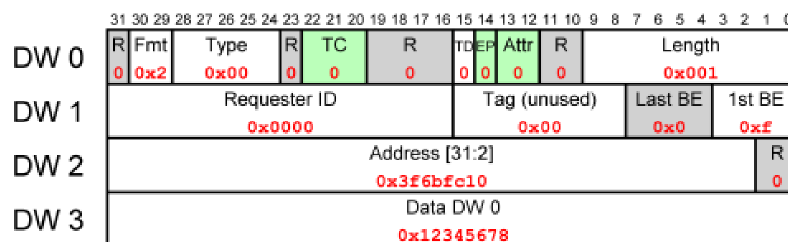

Figura 18 - Exemplo de TLP de requisição de escrita de memória com endereçamento de 32 *bits* [9].

Observando os valores presentes na Figura 18, os valores do pacote TLP que serão transmitidos serão: $0x40000001$, $0x0000000F$, $0xFDAFF040$, $0x12345678$.

A coloração de cada campo do cabeçalho tem um significado específico: A cor cinza representa campos reservado, representados pela letra “R”, devendo ser transmitidos com valor zero e são ignorados pelo receptor. Os campos marcados em verde podem possuir valor diferente de zero, porém são raramente utilizados. Todos os outros campos devem ser preenchidos e os valores para este exemplo estão em vermelho.

Outros campos de interesse são:

- O campo *Length* que representa a quantidade de dados do pacote em número de *Double Words*, no caso da Figura 18 é de somente 1 DW.
- O campo *Requester ID* é um identificador do dispositivo que enviou o pacote para a escrita e serve somente para retornar erros.

Para que uma granularidade maior possa ser obtida na comunicação PCIe, os campo *First BE* e *Last BE* funcionam como uma máscara para o primeiro e para o último DW, respectivamente, indicando quais *bytes* dos DWs são válidos. Sendo possível assim que os dados não estejam alinhados com um DW e que dados menores que um DW possam ser transmitidos.

Por último tem-se o campo *Address* que representa o endereço no qual o dado deve ser escrito. Esse campo pode ser de 30 ou 62 *bits* dependendo do tipo de endereçamento. Além disso, os últimos dois *bits* da linha do campo de endereços são sempre zeros uma vez que pelo menos 1 DW é transmitido e seu o endereçamento é sempre múltiplo de 32 *bits*. Note que ao multiplicar o valor $0x3F6BFC10$, como representado pela Figura 18, por 4 (2 *bits*) obtêm-se o endereço desejado ($0xFDAFF040$).

2.2.2.3.2 Pacote de leitura de memória

Diferentemente do pacote de escrita de memória, a leitura é um pouco mais complexa. Toda leitura envolve dois agentes ou duas operações. Primeiramente o dispositivo que deseja

ler deve enviar um pacote requisitando a leitura, então o dispositivo a ser lido deve responder com um pacote com os dados correspondente a leitura requerida.

Para ilustrar o funcionamento, supõe-se como exemplo a leitura de 1 DW do endereço $0xFDAFF040$ de um dispositivo, inicialmente um pacote de leitura de memória deve ser enviado. A Figura 19 ilustra como seria o TLP dessa requisição.

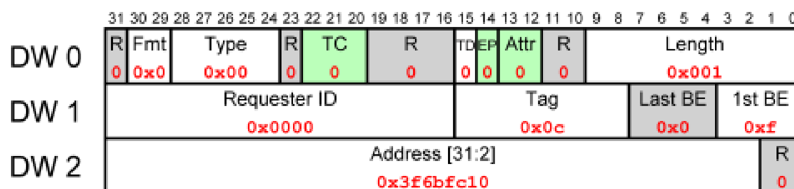

Figura 19 - Exemplo de TLP de requisição de leitura de memória com endereçamento de 32 *bits* [9].

Assim como anteriormente os campos são preenchidos com os valores referentes a operação desejada. Porém nesse caso o campo *Length* representa a quantidade de dados que devem ser lidos. O campo *Requester ID* torna-se fundamental pois é para esse identificador que os dados devem ser retornados. Assim como o campo *Tag*, que não apresneta importância na escrita, mas para a leitura é fundamental.

O campo *Tag* não tem uma função por si só, mas torna-se importante devido a possibilidade de um dispositivo enviar várias requisições de leitura para outro dispositivo. Nesse caso o campo *Tag* serve para identificar qual a resposta para cada requisição, pois este é copiado no pacote de resposta para garantir a correta resposta.

2.2.2.3.3 Pacote de complementação

Uma vez que a requisição de leitura foi recebida, o dispositivo deve então responder a essa requisição com outro pacote. O pacote de complementação (do inglês *completion packet*) é responsável por essa operação de retornar os dados da leitura ao dispositivo correto.

A resposta da requisição do exemplo anterior poderia ser um pacote de complementação como representado pela Figura 20.

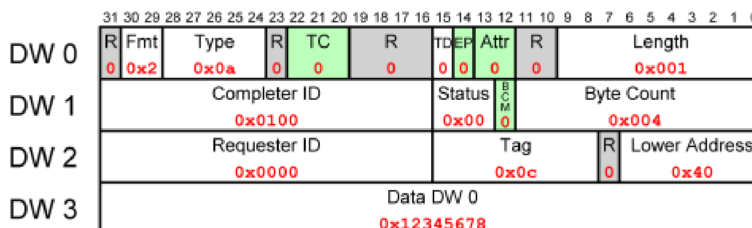

Figura 20 - Exemplo de TLP de complementação [9].

Como anteriormente os campos *Fmt* e *Type* indicam qual o tipo do pacote e o campo *Length* a quantidade de dados sendo enviados. Apesar do dispositivo requerente saber

quantos dados devem ser lidos, há um limite no tamanho dos TLP e portanto, pode ser necessário mais do que um pacote para a transferência de todos os dados.

Além da quantidade de dados sendo enviada, outra informação pertinente é o número de *bytes* que faltam ser transmitidos. Para isso existe o campo chamado *Byte Count* que proporciona a possibilidade da transmissão desalinhada a 32 *bits* bem como a transmissão de menos de um DW.

Outra informação necessária, devido a essa possibilidade de múltiplos TLP de complementação, é o endereço do qual os dados foram lidos. Essa informação é obtida pelos últimos 7 *bits* do endereço de leitura que são transmitidos no campo *Lower Address*. Note que os pacotes não precisam conter o dado em ordem, pois as informações de como ordená-los estão presentes no cabeçalho dos pacotes.

O campo *Status*, por sua vez, indica problemas durante a obtenção dos dados, ou seja, se o valor for zero significa que os dados foram obtidos sem problemas. E o campo *BCM* é utilizado somente por pontes PCI-X caso contrário deve permanecer em zero. Por último têm-se os campos *Requester ID* e *Completer ID* que são identificações do requerente e do requerido respectivamente.

2.2.3 Compatibilidade

Outros tipos de pacotes são especificados pelo padrão PCI *Express*, porém não são utilizados tanto quanto os já apresentados, e por serem específicos a certos dispositivos ou por estarem presentes por questão de compatibilidade. Por exemplo, os pacotes de entrada e saída (I/O) existem, mas têm seu uso desencorajado, uma vez que somente aceitam endereços de 32 *bits* e forçam a espera de uma resposta para cada requisição.

As interrupções também possuem seu modo de compatibilidade. Os dois tipos de interrupções do PCIe são INTx e MSI. O tipo INTx é suportado apenas por compatibilidade com dispositivos PCI. Por isso existem pacotes no protocolo PCI *Express* que ligam e desligam os pinos do *root complex* causando a interrupção no processador.

Devido a problemas desse tipo de interrupção, como retenção da interrupção e compartilhamento de interrupções, o tipo MSI foi criado. Com esse novo tipo as interrupções passaram a ser ações de escrita a endereços pré-estabelecidos, facilitando a identificação do dispositivo que está causando a interrupção e removendo a necessidade de limpeza do pino de interrupção.

2.2.4 Espaço de Configuração

Os dispositivos PCI possuem um conjunto especifico de registradores chamados de espaço de configuração (do inglês *configuration space*). Esses registradores servem para que o sistema operacional, ou um *firmware* do sistema, possa identificar e mapear o dispositivo em memória. Dessa forma, é possível a introdução de um dispositivo sem que o sistema precise ser configurado por pinos (*plug and play*).

O tamanho desse banco de registradores é de 256 *bytes* e pode ser representado pelo diagrama da Figura 21. Este diagrama representa o espaço de configuração para um dispositivo final (*endpoint*), ou seja, um dispositivo que não roteia as informações para outros dispositivos.

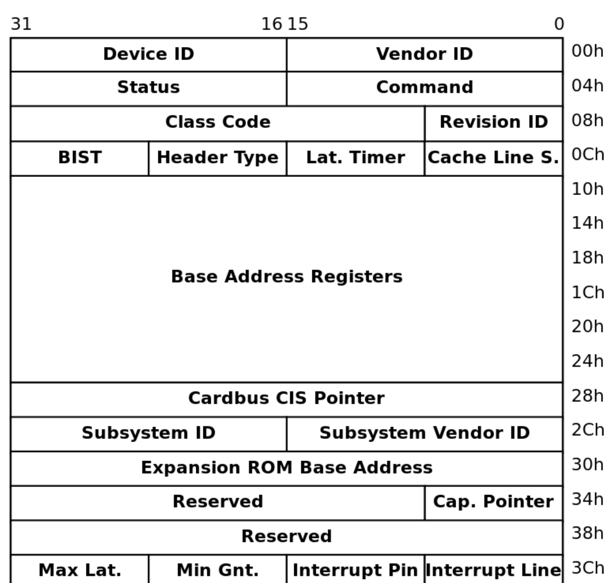

Figura 21 - Diagrama do espaço de configuração PCI [27].

Dentre os diversos campos que esse espaço possui os que valem a pena serem mencionados são os campos *Device ID*, *Vendor ID*, *Status*, *Command* e os *Base Address Registers* (*BARs*). Os registros *Device ID* e *Vendor ID* são registros gravados pelo fabricante do dispositivo e servem para a identificação do dispositivo pelo *software*.

O registro *Status* serve para sinalizar as funções suportadas bem como condições de erro. E o registro *Command* serve para ativar e desativar cada uma das funções do dispositivo.

Os mais interessantes de todos os registros são os *Base Address Registers* esses registros servem para armazenar o endereço no qual o dispositivo está mapeado em memória. Um dispositivo PCI pode ter até 6 BARs de 32 *bits* cada. Com isso o dispositivo é capaz de

ser mapeado e responder para 6 conjuntos de endereços diferentes ou pode combinar 2 BARs para um endereço de 64 *bits*.

O valor do BAR é comparado com o endereço do pacote para verificar qual o dispositivo ou qual função do dispositivo deve ser ativada. Normalmente o valor atribuído a esses registradores é dado por funções do núcleo do sistema no momento de configuração do dispositivo.

2.2.5 Panorama Geral

Para que seja possível ilustrar todo o funcionamento do PCI *Express* com o seu protocolo a Figura 22 representa as camadas e o fluxo de informações entre elas, tanto para a transmissão quanto para a recepção.

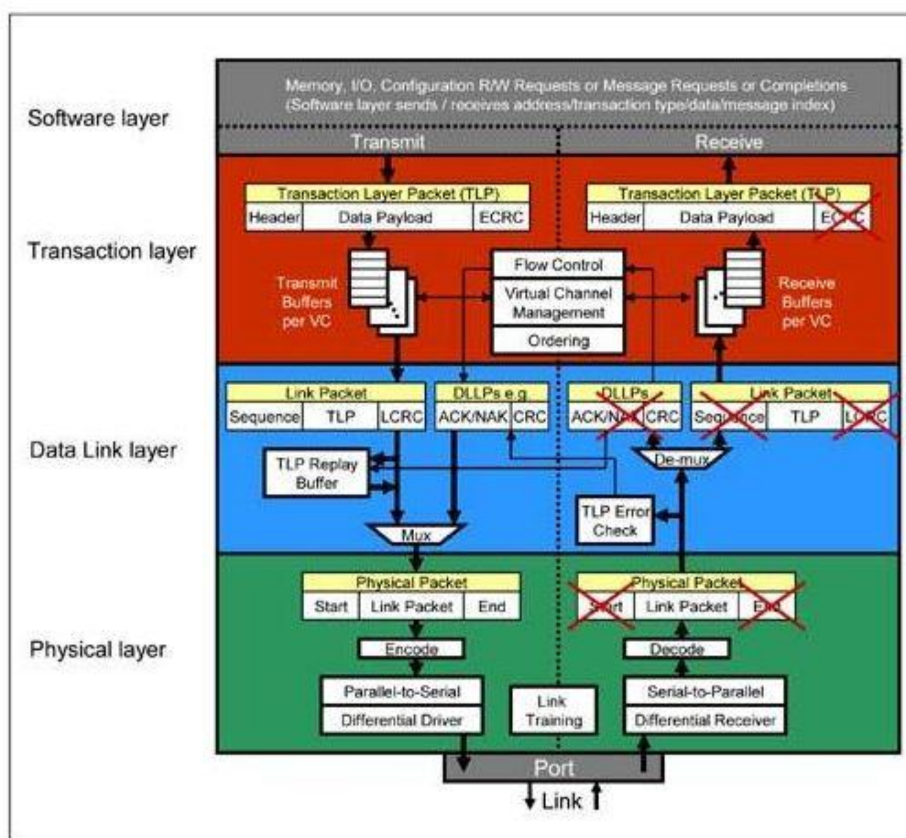

Figura 22 - Diagrama geral das relações entre as camadas do PCI *Express* [24].

A parte cinza superior representa a parte de interna do dispositivo ou a parte interna do *root complex* do computador. É nessa camada que se localiza o espaço de configuração, bem como os dispositivos que fornecem os dados para o PCIe, seja ele o processador ou um controlador de acesso a memória.

Os elementos físicos da comunicação, ou seja, as linhas de dados, onde os sinais elétricos trafegam, são representadas pelas setas de *Link*, em que os dados já foram divididos e encapsulados e prontos para a transmissão.

Portanto, todo o fluxo interno do PCI *Express* pode ser resumido nas camadas verde, azul e vermelha representadas pela Figura 22.

2.3 *Direct Memory Access*

Direct Memory Access (DMA) é um método muito utilizado nos sistemas de computadores para acesso a memória principal por subsistemas sem a necessidade do envolvimento da unidade de processamento central (*Central Processing Unit* – CPU).

A vantagem da utilização desse método é que a CPU fica livre para executar outras instruções enquanto a memória está sendo escrita ou lida por outra unidade. A CPU precisa somente programar o dispositivo que fará o acesso e aguardar uma notificação de fim da operação.

O DMA é muito utilizado quando as taxas de transferências são muito altas ou necessitariam de muito tempo da CPU. Diversos dispositivos utilizam esse tipo de transferência como, por exemplo, controladores de discos e placas gráficas, que são capazes de mover dados da e para a memória principal. Ou até mesmo internamente em um *chip* de processador de múltiplos núcleos no qual os dados podem ser movidos do *cache* de um processador para o *cache* do outro sem que os processadores precisem parar para isso.

Um exemplo atual de utilização de controladores DMA através do duto PCI *Express* é o *DirectGMA* da AMD. Dispositivos que suportem esse tipo de transferência são capazes de transferir dados para e da placa de vídeo, como representado pela Figura 23.

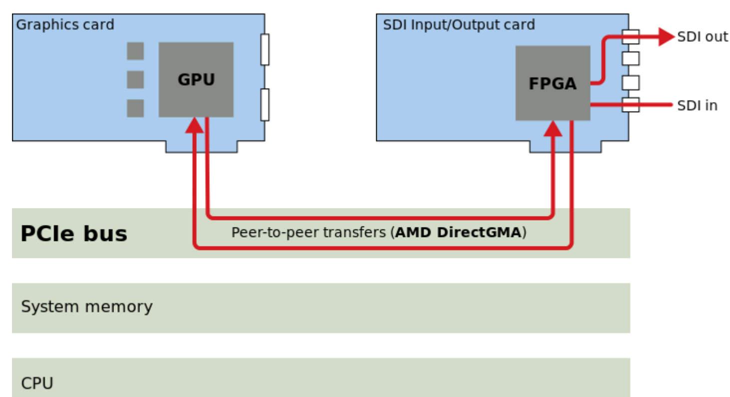

Figura 23 - Exemplo de utilização de DMA (AMD DirectGMA) [28].

3 Desenvolvimento

O desenvolvimento e a validação deste trabalho foram feitos no kit de desenvolvimento TR4® da Terasic® mostrado pela Figura 24. Este kit contém uma FPGA Stratix® IV EP4SGX230KF40C2 da Altera® que possui dois conectores de cabos PCI *Express* com *link* x4.

Figura 24 - Kit de desenvolvimento Terasic TR4 [25].

A elaboração dos códigos em HDL (*Hardware Description Language*) foi feita utilizando o *software* Altera® Quartus® II v14.0 [10], que inclui todas as ferramentas necessárias para a análise, síntese e programação do código nas FPGAs da família Stratix®. Para os códigos em C e C++ utilizou-se o compilador GNU G++ [11] para a compilação no Linux e o ambiente Microsoft Visual Studio [12] para o Windows.

O primeiro passo do desenvolvimento do *framework* foi identificar os módulos básicos para que a comunicação pudesse ocorrer desde a aplicação de alto nível rodando no sistema operacional até o ponto mais baixo do *hardware* e vice-versa (Figura 25).

Em seguida analisou-se como deveriam ser construídos os módulos de *hardware* na FPGA. Baseando-se nas especificações do padrão PCI *Express*. Por último o *software* foi analisado, resultando em dois blocos de atuação: o *driver* que é executado junto ao núcleo do sistema operacional e a aplicação de alto nível que utiliza o *framework* para realizar a validação do sistema.

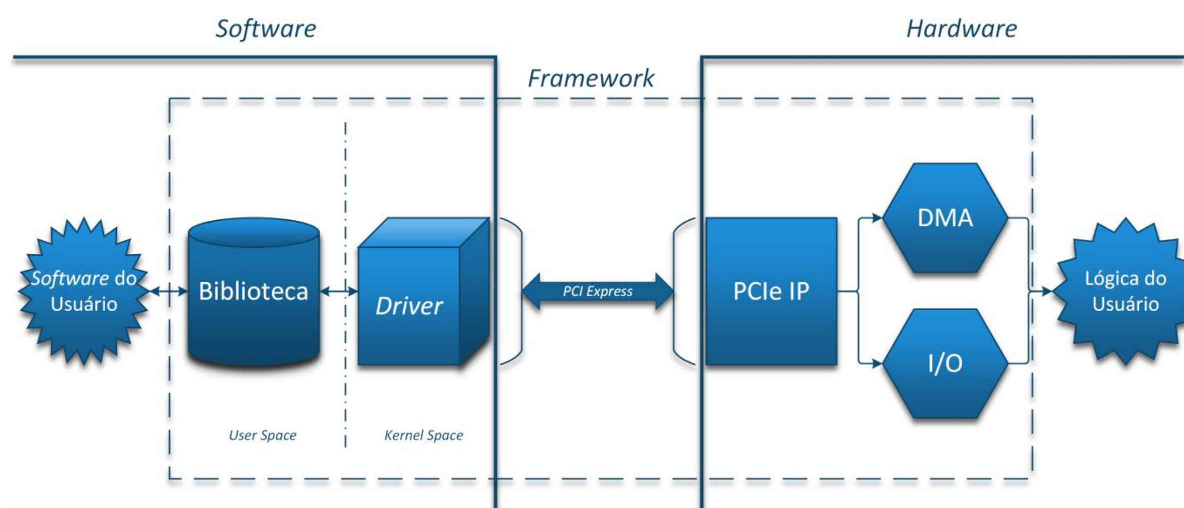

Figura 25 - Diagrama da ideia do *framework* desenvolvido.

3.1 Visão Geral

Ao identificar os principais elementos do fluxo de informação através do PCI *Express*, chegou-se nos elementos representados na Figura 26. Para cada um dos blocos representados nessa figura foram pensadas as funcionalidades e subdivisões para que fosse possível a execução como planejado.

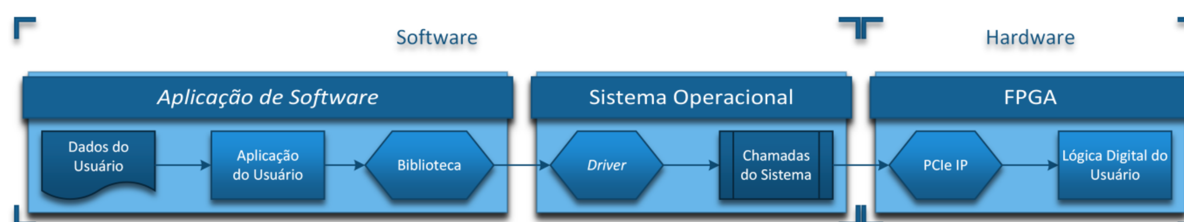

Figura 26 - Diagrama de blocos básicos de uma comunicação PCI *Express*.

O bloco de dados do usuário representa as informações que deverão ser transmitidas ou recebidas pelo *framework*. Tais informações são passadas a uma aplicação de alto nível, que é executada pelo sistema operacional em modo de usuário, que deverá então iniciar a comunicação com o *hardware*.

Para que a comunicação seja estabelecida, a aplicação, criada pelo usuário, deve fazer uma requisição ao sistema operacional de acesso ao dispositivo. Essa requisição deve ser feita por meio da biblioteca do *framework* que já possui funções capazes de facilitar o acesso às funções do sistema operacional.

A biblioteca, por sua vez, realiza chamadas ao *driver* do dispositivo que encontra-se alocado pelo sistema operacional. O *driver* utilizando chamadas internas do sistema

operacional faz a comunicação com o *hardware* através do duto PCI *Express* existente na placa mãe do computador como representado pela Figura 27.

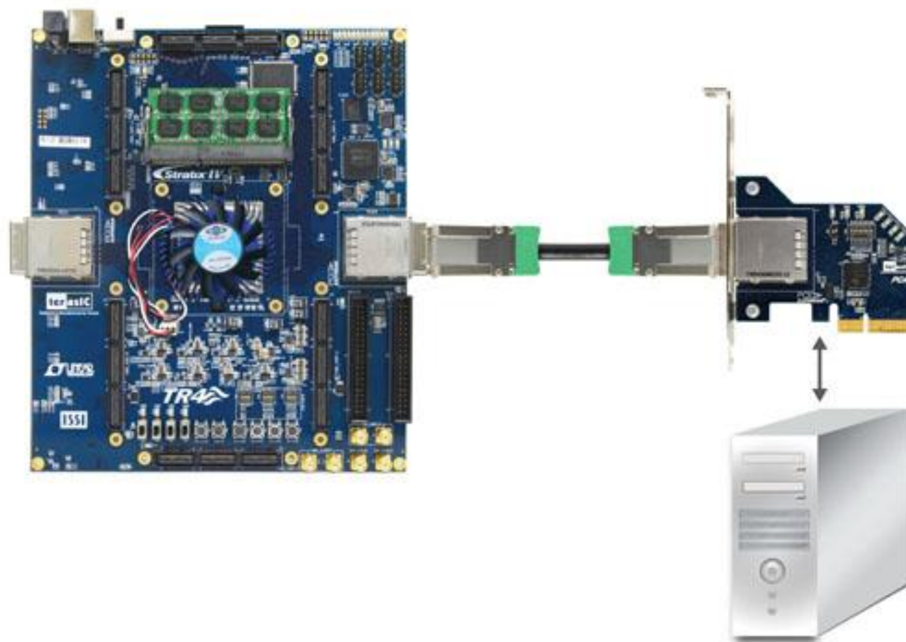

Figura 27 - Conexão PCI *Express* entre o Kit de Desenvolvimento TR4 e um computador *desktop* [26].

O *hardware* recebe os dados através dos pinos do PCI *Express* e realiza todas as operações de tratamento dos pacotes para a decodificação das informações. Por último, a lógica digital, programada pelo usuário na FPGA, recebe as informações do *framework* de forma simples como um duto de dados e de endereços e realiza as operações necessárias com essas informações.

3.2 *Hardware*

Sabendo-se que a funcionalidade básica do *hardware* deve ser a comunicação PCI *Express*, foram pesquisadas as formas possíveis de se realizar isso. Duas formas diferentes foram encontradas utilizando cada uma, um tipo de propriedade intelectual (*Intelectual Property* – IP):

- **Por *Soft* IP:** Criação de uma propriedade intelectual, utilizando as LUTs da FPGA. Essa forma oferece grande flexibilidade e controle total dos módulos existentes no *hardware*. Porém possui a desvantagem de não possuir um alto desempenho, uma vez que a frequência de operação das LUTs é mais baixa do que a de portas lógicas em circuitos integrados.
- **Por *Hard* IP:** Utilização de uma propriedade intelectual já existente na FPGA. Essa forma oferece uma menor flexibilidade, porém um desempenho superior

ao *soft* IP, uma vez que os componentes desse IP são componentes físicos produzidos junto ao *chip* da FPGA. São, portanto, capazes de operar em frequências mais elevadas e dessa forma conseguem movimentar os dados de forma mais rápida.

Priorizando a velocidade de operação, optou-se pela utilização do *hard* IP já existente na FPGA. Para utilizar esse módulo foi necessária a instanciação desse componente utilizando o compilador de IP de PCI *Express* da Altera® [13]. Os módulos criados por esse compilador servem para a ativação do dispositivo físico existente e não consomem os blocos de lógica da FPGA.

Com a utilização do *hard* IP foi possível projetar todo o sistema de *hardware* para que os pacotes pudessem ser recebidos e enviados pelo *hard* IP e os dados pudessem chegar à lógica digital do usuário para serem processados, como representado pelo diagrama da Figura 28.

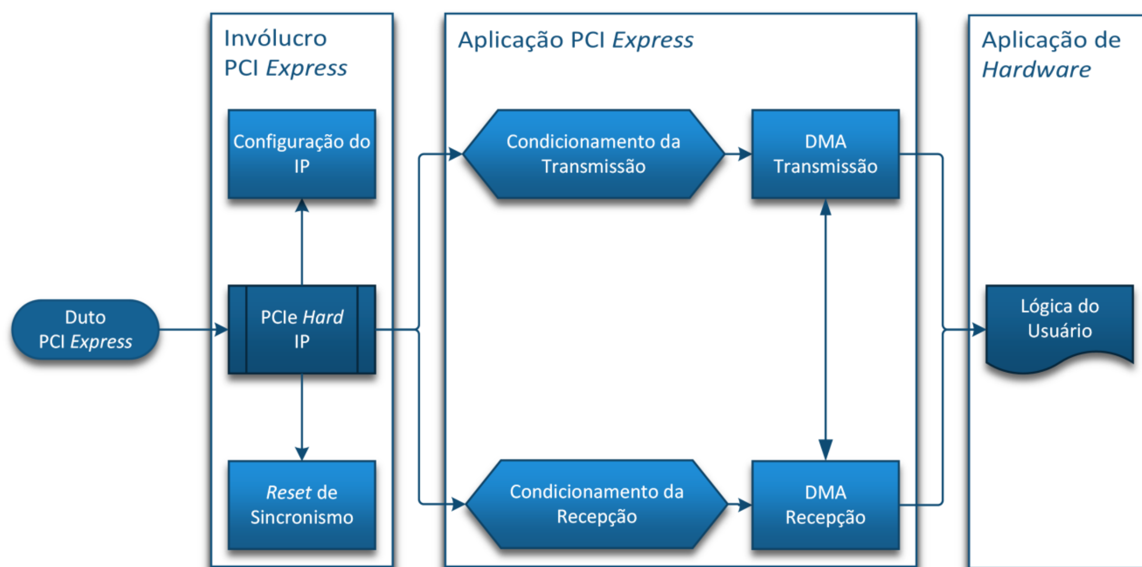

Figura 28 - Diagrama de blocos dos módulos de *hardware*.

O fluxo de dados do *hardware* foi dividido em três blocos principais, o invólucro PCI *Express*, a aplicação PCI *Express* e a aplicação de *hardware*. O *framework* em si, consiste dos dois primeiros blocos.

O invólucro foi criado, pois o PCIe *Hard* IP necessita de alguns blocos de configuração e de *reset* para a correta operação. A aplicação PCI *Express* é responsável por entregar à aplicação de *hardware*, que contém a lógica do usuário, o dado já sem o encapsulamento e pronto para ser utilizado.

3.2.1 PCIe *Hard* IP

Seguindo o fluxo de dados do *hardware*, o módulo que recebe dados pelo duto PCI *Express* é o PCIe *Hard* IP, que por ser um *hard* IP necessita ser ativado. Para ser ativado o IP necessita de algumas configurações básicas que são enviadas para a FPGA durante a fase de programação. A utilização do compilador de IP para PCI *Express* é representada pela Figura 29, na qual os parâmetros físicos de configuração do módulo são estabelecidos.

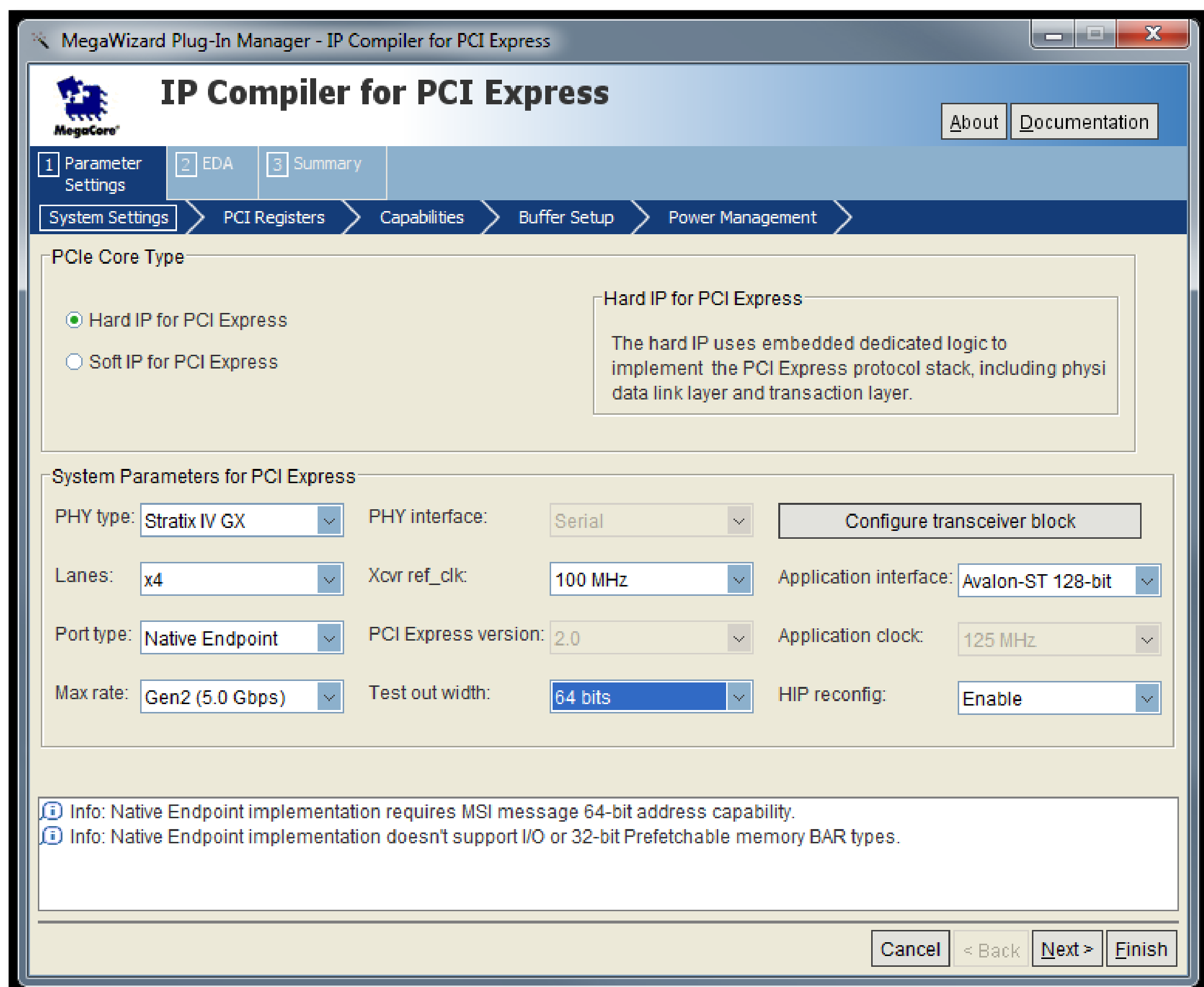

Figura 29 - Configurações do compilador de IP para PCI *Express* da Altera®.

Utilizando o compilador, alguns módulos HDL são criados proporcionando uma forma de conectar esses blocos ao resto dos blocos desenvolvidos em HDL. O módulo topo criado por esse compilador engloba todos os outros módulos e é referenciado nesse trabalho como o PCIe *Hard* IP.

3.2.2 Configuração do IP

A utilização do PCIe *Hard* IP é a mais básica possível, ou seja, foram utilizadas somente as funções de transmissão e recepção do IP como um *endpoint*. Por isso, o bloco de configuração é um bloco simples que ativa a configuração básica e desabilita as outras funções existentes no IP.

As funções de reconfiguração dinâmica e de *root point* foram desabilitadas por não fazerem parte do escopo desse projeto. Porém podem ser habilitadas para um projeto futuro, pois esse bloco de configuração pode ser expandido para suportar os outros modos de operação.

Juntamente com o módulo de configuração do IP, outro módulo necessário é uma PLL (do inglês *Phase-Locked Loop*). Esse componente é necessário para o correto funcionamento do PCIe, uma vez que as frequências de trabalho da FPGA e do PCI *Express* são diferentes.

Essa PLL gera os sinais de *clock* para o PCIe *Hard* IP e para o módulo de configuração para que ambos tenham o mesma base de *clock*, prevenindo qualquer problema de desvio de fase entre os *clocks* dos módulos.

3.2.3 *Reset* de Sincronismo

Como a frequência de operação do PCI *Express* é dada pelo *clock* recebido externamente, existe um módulo específico dedicado ao sincronismo de todo o sistema para a liberação do *reset*.

O sincronismo durante o *reset* envolve os módulos do usuário, bem como o *reset* de componentes específicos de *hardware* da FPGA, dessa forma, o *reset* deve ter uma duração mínima necessária para que todos os componentes possam partir simultaneamente. Além disso, algumas situações de erro internas ao PCIe *Hard* IP devem forçar o reinício do sistema com um *reset* do sistema.

3.2.4 Condicionamento da Transmissão

A comunicação com o PCIe *Hard* IP, é feita através de uma interface proprietária chamada Avalon® utilizada para a interconexão de componentes nas FPGAs da Altera®. Por isso, os dados a serem transmitidos e recebidos pelo IP devem ser condicionados. O condicionamento é necessário para o encapsulamento do pacote PCI *Express* nessa interface.

Para o envio de um pacote para o PCIe *Hard* IP, além do pacote, informações adicionais são passadas ao IP para sinalização dessa operação. Tais informações são: início

do pacote a ser transmitido, fim do pacote transmitido, transmissão de um pacote semivazio e um sinal de erro da operação.

O tráfego de dados por essa interface pode ser feito em blocos de 128 *bits* ou 64 *bits*. Foi escolhido o de 128 *bits* para poder trabalhar a uma frequência de *clock* menos elevada. Portanto, para que um pacote PCIe possa ser enviado ao IP várias transmissões podem ser necessárias. Para isso o módulo de condicionamento da transmissão armazena o pacote PCIe e gera os sinais de início e fim de transmissão.

No caso do envio de um pacote com 64 *bits* existe um sinal responsável pela notificação de que o pacote passando pela interface está semivazio. Existe, também, um sinal que indica uma condição de erro, sendo o pacote descartado pelo IP.

O armazenamento dos pacotes a serem transmitidos é feito por um módulo de FIFO (*First In First Out*). Essa FIFO garante o armazenamento temporário dos dados para que o PCI *Hard* IP possa recebe-los com sucesso. A utilização desse módulo serve como sincronismo entre os módulos e garante o correto funcionamento do circuito.

A transmissão de um pacote PCI *Express* com 3 DWs de cabeçalho seguidas das DWs de dados de forma não alinhada, ou seja, o primeiro dado não é transmitido nos primeiros 32 *bits* da interface, mas logo após o cabeçalho como representado pela Figura 30.

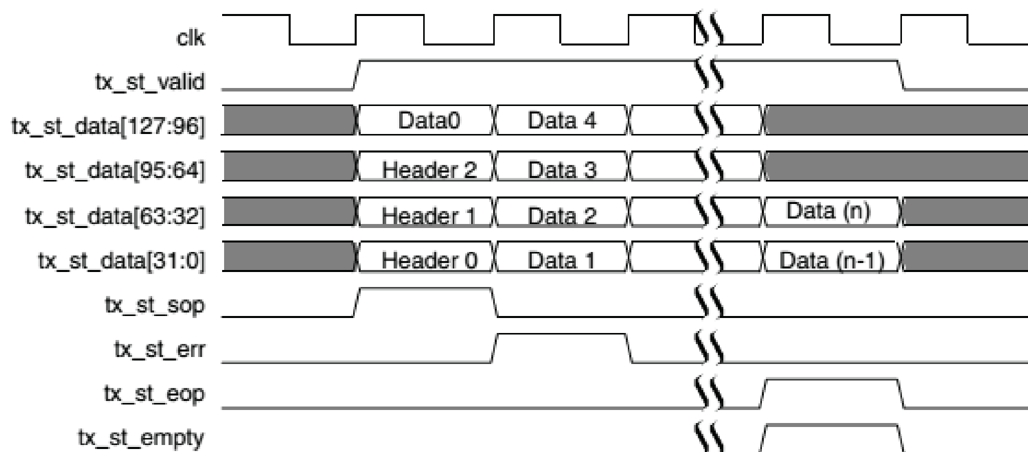

Figura 30 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 3 DWs não alinhadas [13].

A situação similar com os dados alinhados é representada pela Figura 31. Nesse caso parte da primeira transferência é desconsiderada, ou seja, os *bits* de 127 a 96 não são considerados ao se iniciar a transferência de um pacote.

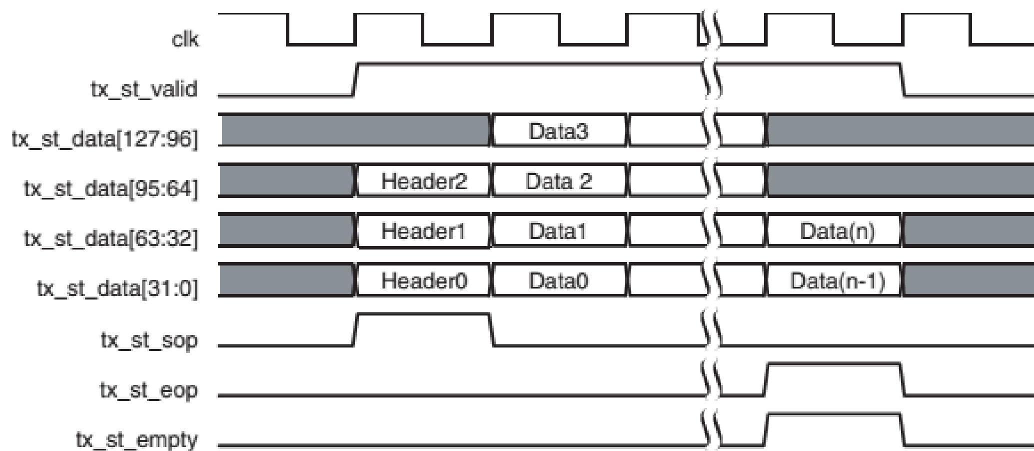

Figura 31 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 3 DWs alinhadas [13].

Da mesma forma as transmissões de pacotes com 4 DWs de cabeçalho e com dados não alinhados são representadas pela Figura 32. Nesse caso os *bits* de 31 a 0 da segunda transmissão é que são desconsiderados.

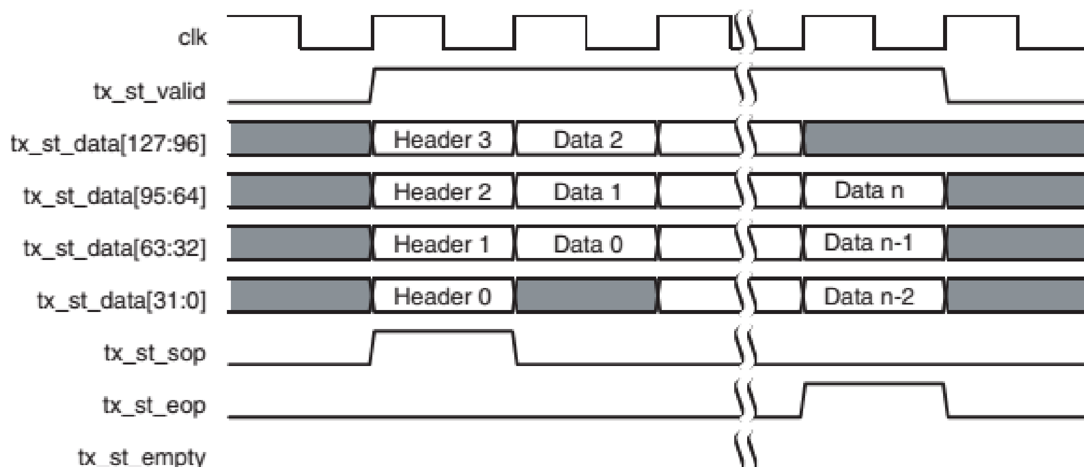

Figura 32 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 4 DWs não alinhadas [13].

O último caso de transmissão em que o pacote possui 4 DWs de cabeçalho e os dados são alinhados é representado pela Figura 33.

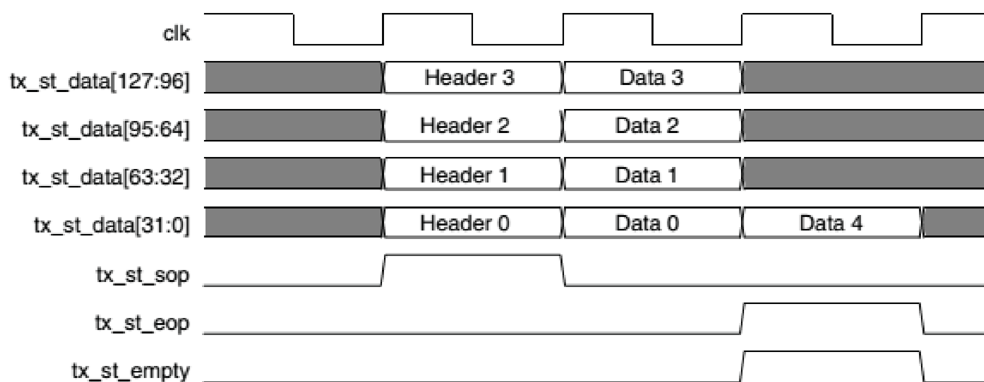

Figura 33 - Forma de onda da interface Avalon® de 128 *bits* para transmissão PCI *Express* com 4 DWs alinhadas [13].

Nota-se também que a última transmissão de um pacote também possui suas peculiaridades, quando o último pacote possui 64 *bits* ou menos de dados válidos, o sinal de semivazio (*tx_st_empty*) é ativo.

Os dados recebidos por essa interface são ordenados de forma que os primeiros dados são colocados nas DWs menos significativas e os últimos nas mais significativas. Porém, os primeiros dados no protocolo PCI *Express* vêm das DWs mais significativas. Portanto os dados devem ser invertidos em ordem para que possam ser passados adiante.

O módulo de condicionamento da transmissão tem como função separar o cabeçalho e alinhar os dados para que o protocolo possa ser tratado por outro módulo.

3.2.5 Condicionamento da Recepção

Assim como a transmissão, o módulo de condicionamento da recepção também é capaz de lidar com os quatro tipos de transmissão pela interface Avalon®. Além disso, algumas informações a mais são passadas pelo PCI *Hard* IP como: *byte enable* e o BAR do PCI *Express*.

As informações de *byte enable* são opcionais e servem somente para os dados transmitidos e não para o cabeçalho. Cada *bit* do *byte enable* representa um *byte* dos 128 *bits* de dados, o *bit* menos significativo representa o *byte* de dados menos significativo e assim por diante até o *bit* mais significativo que representa o *byte* mais significativo. A utilização dessas informações proporciona uma facilidade na interpretação dos dados.

O BAR do PCI *Express* serve como uma informação de acesso a mais que também é passada aos módulos seguintes para que cada BAR possa representar uma função diferente a ser acessada. O BAR é passado juntamente com o cabeçalho do pacote como representado pela forma de onda da Figura 34.

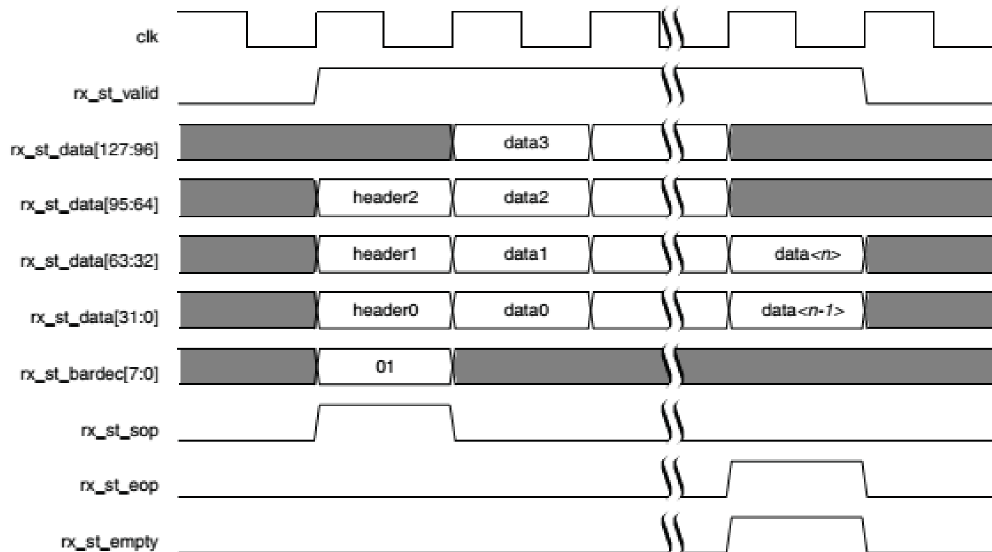

Figura 34 - Forma de onda da interface Avalon® de 128 *bits* para recepção PCI *Express* com 3 DWs alinhadas [13].

O módulo de condicionamento da recepção também possui uma FIFO para o correto sincronismo dos dados. Dessa forma a comunicação tanto para transmissão quanto para a recepção são isoladas por FIFOs para uma melhor eficiência do sistema.

3.2.6 DMA de Transmissão

Para que seja possível fazer transferências em alta velocidade para a memória do computador optou-se pela criação de um módulo capaz de realizar operações de DMA entre a FGPA e o computador. O diagrama geral do funcionamento do módulo de DMA de transmissão é representado pela Figura 35.

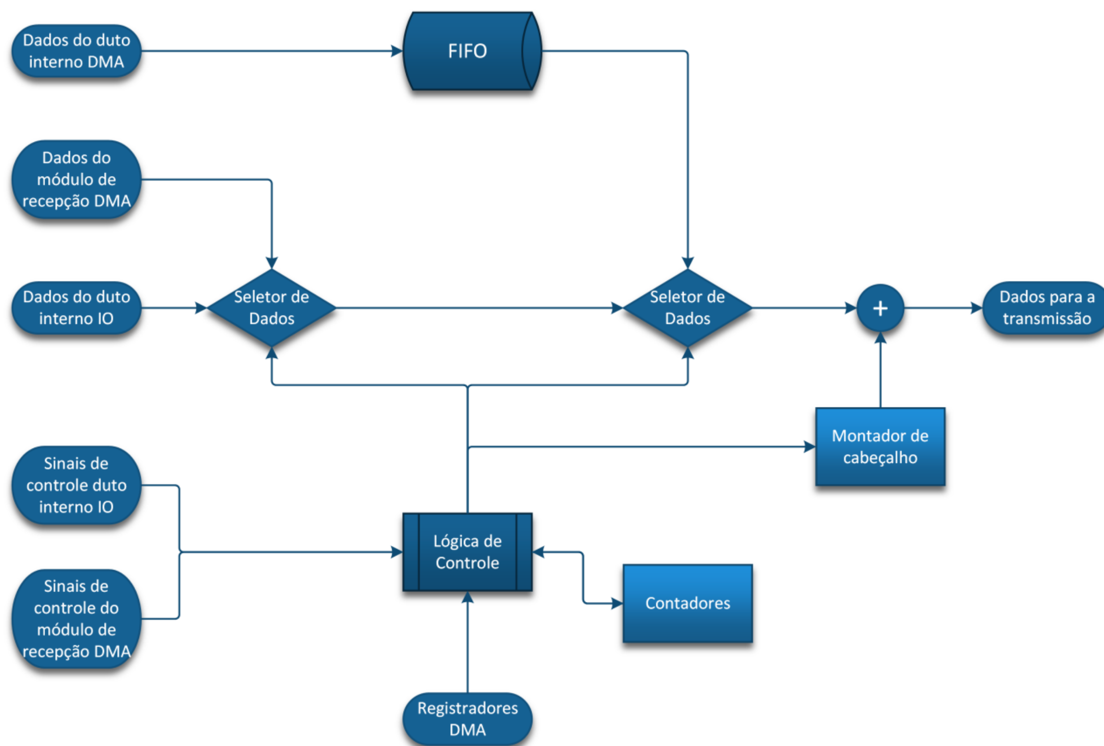

Figura 35 - Diagrama do módulo de DMA de transmissão.

O módulo de DMA de transmissão é responsável pela criação dos pacotes TLP a serem transmitidos para que as informações possam ser enviadas para os endereços corretos de destino. Uma vez que o pacote TLP esteja finalizado ele passa pelo condicionamento e depois é enviado ao *hard* IP para sua transmissão.

Esse módulo cria dois tipos de TLPs, os de escrita de memória e os de complementação para a resposta de uma leitura realizada pelo computador. Como esse módulo é capaz de criar outros tipos de pacotes PCI *Express*, foi adicionada mais uma funcionalidade de transferência de dados direta sem DMA.

O duto criado para a transferência direta sem DMA é chamado de duto I/O, isso porque sua característica é de transmissão de poucos dados de forma direta. Assim como representado pela Figura 35 os dados desse duto podem ser diretamente ligados à saída quando selecionados.

A transferência através do método DMA necessita o armazenamento das informações de destino e tamanho dos pacotes em registradores. Como os registradores da DMA, os quais são responsáveis pela programação da comunicação, estão localizados no módulo de recepção, o módulo de transmissão possui uma comunicação direta com o módulo de recepção para o compartilhamento dessas informações.

A lógica de controle recebe os registradores DMA e, com base nos sinais de controle provenientes do módulo de recepção DMA e do duto de I/O, seleciona qual dos dutos deve ter os dados transferidos. A prioridade para a transferência é dos dados de I/O, depois dos dados do módulo de recepção e por último a transferência do duto DMA.

Os dados do módulo de recepção são os dados dos pacotes de complementação, portanto os dados vêm prontos do módulo de recepção e são transmitidos com o cabeçalho correspondente de forma a responder a requisição recebida pelo módulo de recepção.

A transferência dos dados do duto DMA, por sua vez, é feita utilizando uma FIFO para armazenar os dados até que todas as informações necessárias tenham sido recebidas. As informações recebidas são utilizadas pelo montador de cabeçalho, que cria o cabeçalho do pacote de escrita em que os dados serão encapsulados.

Para manter um controle do fluxo de dados, bem como o estado de cada uma das transmissões, a lógica de controle necessita de contadores que fazem parte da seleção das operações do módulo, bem como de parte do cabeçalho.

3.2.7 DMA de Recepção

Após os pacotes recebidos serem condicionados e prontos para a interpretação, o módulo de DMA de recepção recebe esses pacotes e decide o destino dessas informações. Primordialmente esse módulo é responsável por receber os dados de uma transmissão por DMA, devendo decidir de onde e para onde essa informação deve ir, bem como se uma resposta é necessária ou não. O diagrama geral de funcionamento do módulo de DMA de recepção é representado pela Figura 36

A esse módulo foi atribuído o armazenamento das informações que um controlador DMA deve ter para realizar suas operações de acesso à memória. Para isso os seguintes registradores foram criados: Um registrador de controle, um de endereço de destino, um de endereço da fonte e um contador do número *bytes* a serem recebidos.

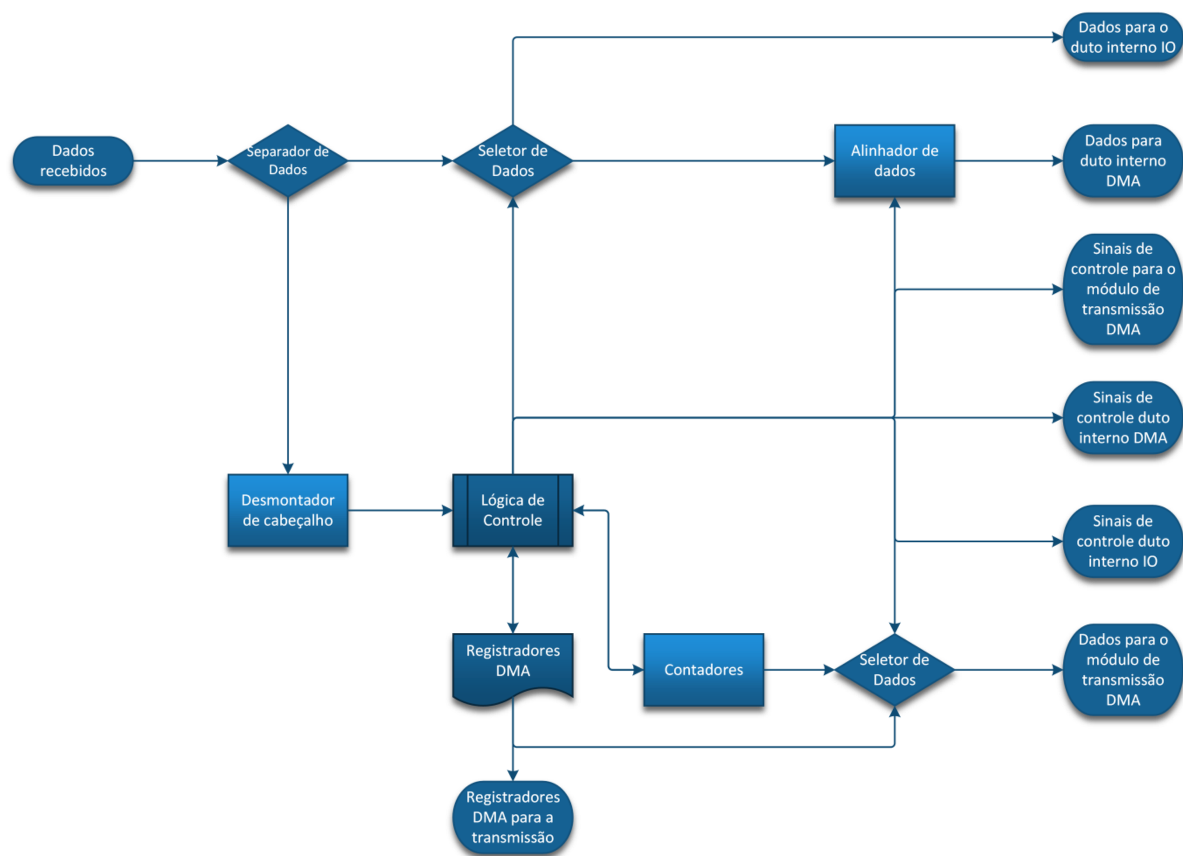

Figura 36 - Diagrama do módulo de DMA de recepção.

As informações recebidas são separadas em dados e cabeçalho para que este possa ser analisado. A lógica de controle seleciona o destino das informações recebidas, além de atribuir os valores correspondentes nos registradores DMA.

Com os registradores DMA carregados com os valores corretos, os dados são enviados para a lógica do usuário através do duto DMA com o endereço e os sinais de controle correspondente à operação em curso. Porém como os dados podem ser recebidos de forma não alinhada ao início do pacote, esses dados passam por uma estrutura de alinhamento.

Para os casos em que há a necessidade de um pacote de complementação, como uma resposta a requisição corrente, os dados correspondentes são selecionados e enviados ao módulo de transmissão que cuidará que o pacote seja corretamente montado.

Os casos em que a operação deve ser feita com o duto de I/O, o BAR presente no pacote recebido é diferente do BAR das operações DMA. Com isso a lógica de controle decodifica as informações e atribui os valores diretamente no duto de I/O, enviando também os sinais de controle do duto. A decodificação escolhida foi BAR 0 para operações com DMA e BAR 1 para operações com o duto de I/O.

Toda decodificação e validação dos dados utilizando os BARs, os endereços e os registros de *byte enable* são feitos pela lógica de controle. Além disso, a lógica de controle ainda tem acesso aos contadores de dados de entrada para manter o fluxo de informações até que a operação corrente seja finalizada.

3.2.8 Lógica do Usuário

O bloco de lógica do usuário corresponde ao módulo criado pelo usuário do *framework* para a comunicação com o PCI *Express*. Essa interação pode ser feita de duas maneiras diferentes: Através de um duto para pequenas quantidades de dados com comunicação direta. Ou através de um duto para maior quantidade de dados com comunicação através de um fluxo de dados utilizando DMA.

A comunicação direta tem como objetivo a troca de comandos direta. Por isso para grandes quantidades de dados há uma grande sobrecarga fazendo com que as taxas de transferência caiam.

Em oposição tem-se o duto DMA capaz de transferir grandes quantidades de dados, sem que o processador do computador gaste ciclos com essa operação. Dessa forma os dados fluem entre a FPGA e a memória do computador como água em um cano. Para a lógica do usuário esses dados são acessados como dois dutos com endereços de leitura e escrita distintos.

3.3 *Software*

Para o controle e comando de toda a estrutura de *hardware* criada, foi necessária a criação de um *software* capaz de fazer a comunicação entre a aplicação do usuário e o duto PCI *Express* presente na placa mãe do computador.

Como *softwares* são executados por um sistema operacional, foi necessário o entendimento e criação de módulos capazes de interagir com o sistema operacional para controlar o *hardware*. Devido à comunicação de alguns módulos com o sistema operacional, é necessária a utilização de comandos específicos tornando assim esses módulos dependentes da plataforma escolhida.

A escolha da plataforma foi feita com base na existência de documentação e facilidade de interação com o sistema, portanto o desenvolvimento desse trabalho foi voltado para a plataforma Linux, devido a sua grande comunidade e extensa documentação.

A partir da plataforma escolhida, estudou-se como fazer a comunicação de uma aplicação rodando no sistema operacional com as chamadas de PCI *Express* do sistema operacional. O diagrama geral dos módulos necessários para essa comunicação é representado pela Figura 37.

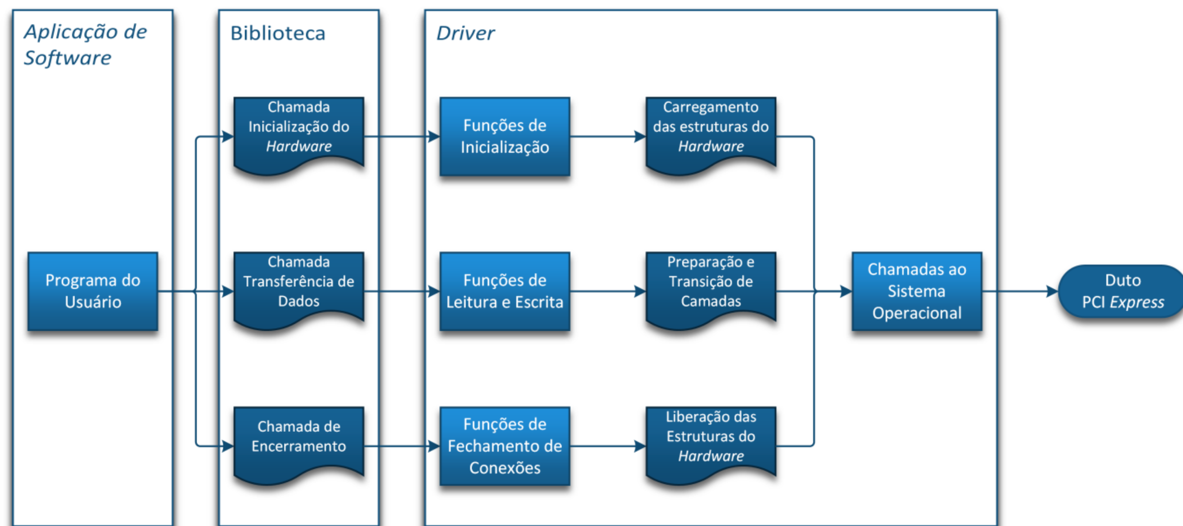

Figura 37 - Diagrama de blocos dos módulos de *software*.

O fluxo todo foi dividido em três blocos devido a suas características de programação e forma de execução. Essas divisões foram: Aplicação de *Software*, Biblioteca e *Driver*. A aplicação de *software* consiste do programa que irá utilizar do *framework* para fazer a comunicação PCI *Express*. A biblioteca é o *software* que possibilita ao programa do usuário acessar as funções do *framework*. E por último o *driver* que é o responsável por utilizar o sistema operacional para a comunicação com o *hardware*.

3.3.1 Aplicação de *Software*

A aplicação de *software* consiste do programa executado pelo sistema operacional que faz as chamadas de acesso ao *hardware*. Esse *software* pode ser compilado a partir de qualquer linguagem desde que seja possível fazer chamadas a funções escritas na linguagem C.

3.3.2 Biblioteca

Para que chamadas simples sejam feitas por essa aplicação é necessária a utilização de uma biblioteca. O objetivo dessa biblioteca é simplificar a utilização do *driver*. Ao invés de chamar diversas funções para a execução de uma tarefa, a biblioteca simplifica para um número menor de chamadas. Além disso, aumenta a reutilização de código, fazendo com que

não seja preciso que todos os programas, que desejam fazer uma comunicação com o *hardware*, tenham o mesmo conjunto de funções, basta nesses casos chamar a biblioteca.

As funções básicas da biblioteca são: abrir e fechar uma comunicação, escrever e ler do PCI *Express* e realizar operações de DMA pelo PCI *Express*. Portanto um conjunto de oito funções é definido na biblioteca. Esse conjunto pode ser dividido em três etapas de comunicação. A inicialização, a transferência de dados e a finalização.

A utilização da biblioteca é análoga a operações em arquivos. Primeiro deve ser feita a inicialização. Para a inicialização foi criada uma função de abertura de dispositivo. Essa função é responsável por inicializar o *driver* tornando possível o recebimento de comandos pelo mesmo.

Em seguida é possível realizar quatro tipos de operações diferentes com o dispositivo já inicializado. Essa é a etapa de transferência de dados, portanto existem quatro funções na biblioteca capazes de realizar essas operações.

A primeira função de transferência de dados é a função de leitura. A utilização dessa função implica na leitura de dados do dispositivo, porém sem utilizar a função de DMA. Essa função, com o BAR correto tem acesso direto ao duto de I/O do dispositivo.

A segunda função de transferência de dados é a operação reversa, ou seja, a função de escrita. Assim como anteriormente essa função não utiliza os recursos de DMA, portanto é um acesso direto. Para escrever no duto I/O do dispositivo deve-se utilizar essa função com o BAR correspondente.

Com as duas funções de acesso direto, o que resta é o acesso através de DMA. As duas funções seguintes são responsáveis para a comunicação DMA. A terceira função de transferência de dados é a função capaz de configurar o dispositivo e o sistema operacional para a operação de DMA.

Para execução da operação de DMA existe a quarta função de transferência de dados que libera o dispositivo para a transferência. Essa operação pode ser tanto a leitura como a escrita de dados, pois a direção da operação é atribuída na configuração da DMA.

Por último, a função de fechamento do dispositivo, encerra as comunicações e libera o *driver* do sistema operacional.

3.3.3 *Driver*

Apesar das funções presentes no *driver* serem utilizadas pela biblioteca, nenhuma chamada é feita diretamente ao *driver*, deve ser feita uma chamada ao sistema operacional para que este chame as funções presentes no *driver* quando necessário.

O *driver* por sua vez é o *software* capaz de comunicar com o *hardware* utilizando o sistema operacional. Como esse bloco é muito dependente de plataforma, e devido à documentação disponível, custo e facilidade de desenvolvimento, o foco foi o desenvolvimento de um *driver* para Linux.

Sua execução se dá em modo privilegiado (*kernel mode*), garantindo acesso a regiões de memória e a parâmetros do sistema, normalmente não disponíveis as aplicações. Por isso, a estrutura de construção e de acesso é padronizada e para o desenvolvimento do *driver* devem ser seguidas certas recomendações baseadas no tipo de *driver*.

O tipo de *driver* desenvolvido foi o *driver* baseado em caracteres, que é o tipo mais comum e o mais simples [14]. Porém não significa perda de desempenho ou impossibilidade de transferência de grandes quantidades de dados.

Todo *driver* deve ter um jeito de avisar ao sistema operacional de sua disponibilidade. Portanto, ao carregar ou descarregar um *driver* no Linux as seguintes funções podem ser executadas:

- **Função *init*:** Tem como objetivo carregar as tabelas de identificação e estruturas do dispositivo, como *mutexes*, assim como registrar o *driver* no sistema operacional.
- **Função *exit*:** Opostamente à função *init*, essa função remove o registro do *driver* do sistema operacional, tornando o dispositivo inacessível.

As funções apresentadas são executadas utilizando as macros *module_init* e *module_exit*, definidas pelo sistema operacional. Do mesmo modo a tabela de identificação do dispositivo PCI também é registrada no sistema operacional por meio da macro *MODULE_DEVICE_TABLE*, passando como parâmetros o tipo do dispositivo (*pci*) e a tabela a ser registrada.

Para um *driver* baseado em caracteres algumas funções básicas são necessárias como as funções seguintes:

- **Função *open*:** Utilizada quando a aplicação do usuário abre o dispositivo causando a inicialização do *driver*. É a função responsável por salvar o ponteiro correspondente a estrutura do *driver*.
- **Função *release*:** É o par da função *open*. A função *release* é chamada quando a aplicação do usuário fecha o dispositivo e portanto encerra o *driver*. No caso desse trabalho a função simplesmente retorna zero, uma vez que a função *open* não cria nada que necessite ser destruído.

- **Função *read*:** A função *read* é chamada quando o usuário realiza a leitura do dispositivo. A leitura do dispositivo, bem como, passagem dos dados do ambiente *kernel* para o ambiente do usuário é feita por essa função.
- **Função *write*:** Quando a aplicação do usuário escreve no dispositivo o sistema operacional chama a função *write* do *driver*. Essa função, assim como na *read*, passa os dados do ambiente do usuário para o ambiente do *kernel* e faz a escrita do dispositivo.
- **Função *llseak*:** Para fazer a procura no dispositivo pode-se utilizar a função *llseak*, que é chamada quando a aplicação realiza a operação de busca. Para o dispositivo desse trabalho a busca só faz sentido se significar um incremento (*offset*) no endereço utilizado.
- **Função *ioctl*:** Essa função é executada quando as chamadas *ioctl* do sistema são executadas. Com essa função é possível passar comandos para o *driver*, portanto essa função é utilizada pela biblioteca para executar as operações de DMA, tanto configuração quanto execução.

Esse conjunto de funções é registrado no sistema operacional através de uma estrutura chamada *file_operations* na inicialização do dispositivo.

Além dessas funções, por se tratar de um *driver* para um dispositivo PCI algumas funções a mais devem ser registradas no sistema operacional através da estrutura *pci_driver* passada na inicialização do dispositivo. Os atributos da estrutura *pci_driver* são:

- **Propriedade *name*:** Conjunto de caracteres para representar o dispositivo. Esse nome é mostrado em $/proc/devices$ e ao se acessar o dispositivo em $/dev/\{\text{name}\}\{número\}$ como por exemplo $/dev/ciermag_pcie0$.
- **Tabela *id_table*:** É a tabela com informações do dispositivo como identificadores de fabricantes e identificadores de produtos. Essa tabela é utilizada para o acesso ao *hardware* correto, verificando essas identificações.
- **Função *probe*:** É a função chamada quando o núcleo PCI do sistema operacional tem uma estrutura *pci_driver* e deseja controla-la. Nessa função são feitas as inicializações do dispositivo no sistema, como alocação de memória para o dispositivo e habilitação do dispositivo no sistema operacional.
- **Função *remove*:** A função *remove* é chamada quando o núcleo PCI do sistema operacional deseja tirar da memória a estrutura *pci_driver*. Portanto, nessa

função é feita a desativação do dispositivo no sistema operacional e remoção da memória alocada pela função *probe*.

A estrutura *pci_driver* é então registrada pela função *init* do *driver* para que o sistema operacional saiba onde achar cada uma das funções a serem chamadas.

O *driver* além de ser muito dependente da plataforma utilizada é também dependente da versão utilizada. Cada versão traz algumas diferenças nos nomes das funções presentes e na forma como as tarefas devem ser codificadas. Principalmente no sistema Linux em que o código sofre contribuições por parte uma grande quantidade de pessoas, é natural que ocorra uma mudança nos parâmetros e nos nomes das rotinas do núcleo do sistema operacional com o passar do tempo. Como o *driver* utiliza essas rotinas para gerenciar os dispositivos, uma mudança no núcleo, implica em uma mudança no *driver* também.

4 Resultados

O principal resultado desse trabalho foi à criação e execução da comunicação através do PCI *Express* entre um computador e a FPGA. Dentre as diversas HDL existentes, o *framework* foi criado utilizando VHDL para o *hardware*, a qual já vinha sendo utilizada no CIERMag, e a linguagem C para a biblioteca e o *driver* e C++ para o *software* de teste.

4.1 *Hardware*

O desenvolvimento do código para a FPGA Stratix® IV EP4SGX230KF40C2 da Altera®, resultou em um total de 7247 linhas de código em VHDL, das quais os maiores blocos foram os blocos de DMA, o de transmissão com 1018 linhas de código e o de recepção com 978.

A ferramenta utilizada para a compilação e gravação do *hardware* na FPGA foi o Altera® Quartus® II v14.0.2 [10]. Os resultados apresentados após a síntese do *framework* estão representados na Figura 38.

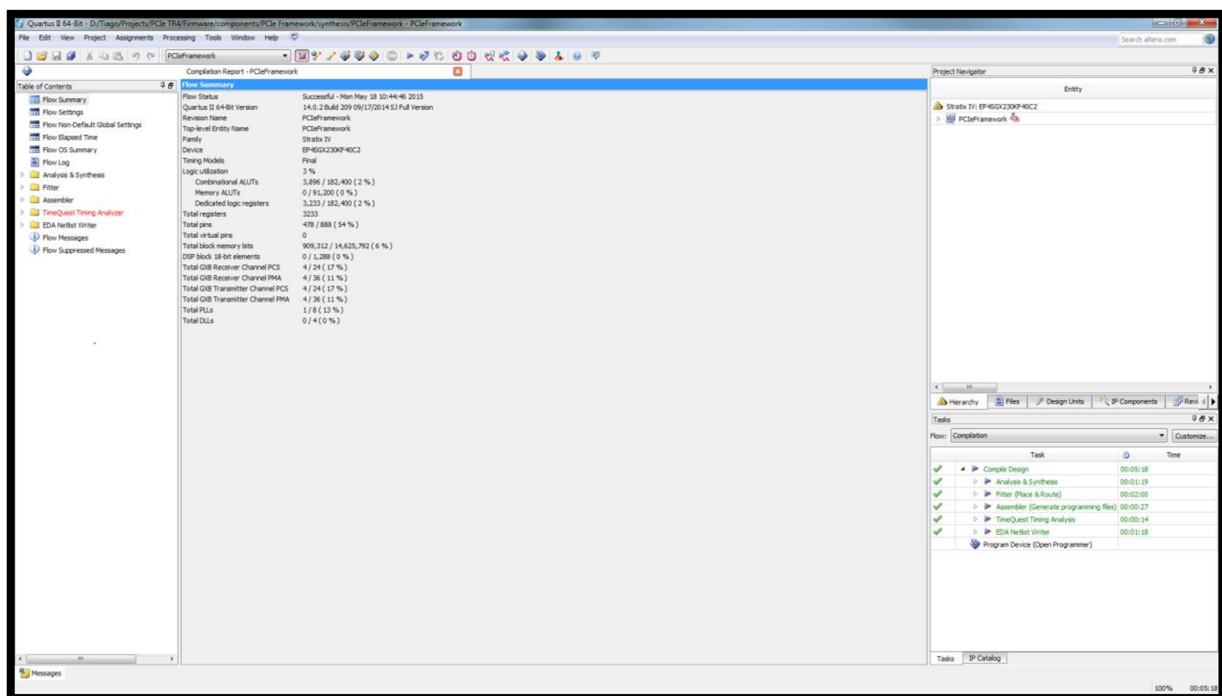

Figura 38 - Captura de tela com os valores de resultado da compilação do *framework*.

Os valores apresentados pelo programa Quartus® podem ser observados mais facilmente na Tabela 3. Esses resultados demonstram de uma forma geral como são aproveitados os recursos da FPGA. Tentou-se utilizar a menor quantidade de recursos possíveis, sem perder funcionalidade ou desempenho do sistema, para que o desenvolvedor final tenha disponível mais recursos.

O circuito foi projetado sem que a ferramenta reportasse erros, avisos de temporização ou de problemas de lógica. Todos os *warnings* da lógica projetada foram resolvidos, porém a ferramenta ainda indica 20 *warnings* que são referentes a módulos internos do PCIe *Hard* IP.

Tabela 3 - Resultados da síntese do *framework* com o programa Quartus®.

	Quantidade Utilizada	Quantidade Disponível	Porcentagem Utilizada
Lógica total	-	100%	3%
ALUTs combinacionais	3896	182.400	2%
ALUTs de memória	0	91.200	0%
Lógica dedicada a registro	3233	182.400	2%
Total de blocos de memória (*bits*)	909.312	14.625.792	6%
Total de blocos de DSP de 18-*bits*	0	1.288	0%
Total de receptores GXB canal PCS	4	24	17%
Total de receptores GXB canal PMA	4	36	11%
Total de transmissores GXB canal PCS	4	24	17%
Total de transmissores GXB canal PMA	4	36	11%
Total de PLLs	1	8	13%
Total de DLLs	0	4	0%

Do total de lógica apresentado, o PCIe *Hard* IP não é computado pois ele não consome lógica por já existir um *hardware* dedicado. Analisando a Tabela 3 nota-se que o consumo de lógica é pequeno, 3% de toda lógica disponível na FPGA sobrando assim 97% de lógica para a lógica do usuário. Considerando os blocos de memória utilizada, apesar do uso de 6% para as FIFOs, a quantidade ainda é baixa e não implica em problema nenhum para a utilização do *framework*.

Percentualmente os recursos mais consumidos são os receptores e transmissores GXB. O consumo foi de 4 para cada pois foram utilizadas 4 *lanes* PCI *Express* e esses circuitos são responsáveis pela transmissão de dados em alta velocidade.

A partir dos resultados da ferramenta sobre cada um dos módulos que compõem o *framework*, percebe-se que o módulo de DMA de transmissão utiliza 678 ALUTs combinacionais e 801 de blocos lógica dedicada a registros, apesar de ser um módulo grande e complexo, o consumo de lógica foi aproximadamente 17% e o consumo de registradores

25%. Isso pode ser explicado pelo fato de que esse módulo lida com vários estados e parâmetros de operação.

O módulo de DMA de recepção é um pouco menor com 446 ALUTs combinacionais e 407 blocos de lógica dedicada a registro o que representa 11% e 13% respectivamente dos recursos consumidos. Esse módulo, diferentemente da transmissão, lida com os vários estados e não necessita armazenar tantas informações, uma vez que sua tarefa é desmontar as informações recebidas e não montá-las como a transmissão.

4.2 *Software*

Para o teste e verificação do *hardware* foi necessário o desenvolvimento do *driver*, da biblioteca e também de aplicações de teste. Como o objetivo das aplicações é a validação do sistema o tema das aplicações foi escolhido para ser simples. Dessa forma, escolheu-se o processamento de imagens através da inversão das cores de uma imagem.

4.2.1 *Driver*

O *driver,* desenvolvido em C, possui 1707 linhas de código no total. Sua compilação foi feita utilizando o compilador GNU G++ [11]. Por seguir as recomendações para *drivers* de caracteres a criação do *driver* foi um pouco mais trabalhosa e portanto consumiu mais tempo de desenvolvimento do que as aplicações de teste.

Desenvolvimento de *drivers* é diferente da criação de aplicações em alto nível, pois se deve tomar cuidado com proteção de regiões contra concorrência, alocação e limpeza de regiões de memória, bem como da transferência de dados entre o modo de usuário e o modo de *kernel* do sistema operacional.

Os testes realizados foram feitos no *Ubuntu 14.10 x64* [15] e para teste do *driver* foi desenvolvido um *software* para linha de comando do Linux capaz de realizar transferências de dados através do PCI *Express*. Além do *software* de teste alguns *scripts* de carga e descarga do *driver* do sistema operacional também foram criados com o objetivo de facilitar essas operações, uma vez que é necessária atualização frequente do *driver* no sistema durante a etapa de desenvolvimento.

4.2.2 Aplicações de teste

Para teste do *hardware* desenvolveu-se aplicações para o sistema operacional *Windows 7 x64* [16] e para o *Ubuntu 14.10 x64* [15] com a versão 3.19 do *kernel*.

4.2.2.1 Windows

Para o teste no *Windows* utilizou-se o *driver* para PCI *Express* da Terasic que foi desenvolvido com base no WinDriver™ da empresa Jungo Connectivity Ltd. [17] para fazer a comunicação PCI *Express*. Apesar de o *driver* disponibilizado ser limitado, foi possível adaptar o *hardware* para que funcionasse com o *driver* existente.

Para testar o funcionamento, foi feita uma aplicação de teste em modo gráfico (utilizando C++) para a inversão de cores de uma imagem. Para isso uma aplicação de inversão de *bits* de uma memória foi gravada na FPGA utilizando o *framework* desenvolvido.

O funcionamento dessa aplicação de teste é o seguinte: primeiramente carrega-se uma imagem no programa pressionando o botão *Select Image*. Uma janela abrirá permitindo que o arquivo com a imagem desejada possa ser carregado. Em seguida envia-se a imagem para a FPGA ao pressionar o botão *Download Image*. Com a imagem carregada na memória da FPGA envia-se o comando de inversão de cores, através do botão *Process Image*. Por ultimo traz-se a imagem de volta com o botão *Upload Image*.

A imagem da Figura 39 representa o programa de teste desenvolvido para *Windows* com a imagem original carregada na memória e pronta para ser enviada para a FPGA, através do PCI Express, para que a lógica interna faça o processamento dessa imagem.

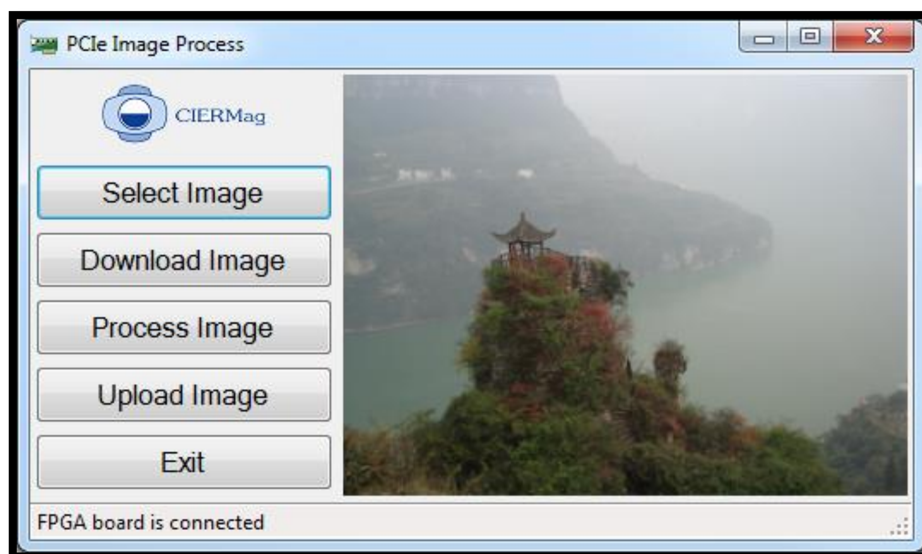

Figura 39 - Programa de teste do *framework* PCI *Express* para Windows com a imagem original enviada para a FPGA.

O método de envio da imagem é através de DMA. Portanto conectado diretamente ao duto DMA existe uma memória capaz de receber os dados enviados e consequentemente armazenar a imagem em questão.

Para processar a imagem, é enviado um comando que inicia o processamento da imagem na FPGA. Com esse comando todo o conteúdo da memória da FPGA tem seus *bits* invertidos. Diferentemente do caso anterior, esse comando é enviado através do duto de I/O por ser uma quantidade de dados muito pequena, um *byte* no caso.

Para obtenção de volta da imagem utiliza-se também a transferência através de DMA. A imagem é então carregada na memória do programa que atualiza a imagem mostrada na caixa correspondente.

O resultado desse procedimento é a imagem com as cores invertidas como pode ser observado na Figura 40.

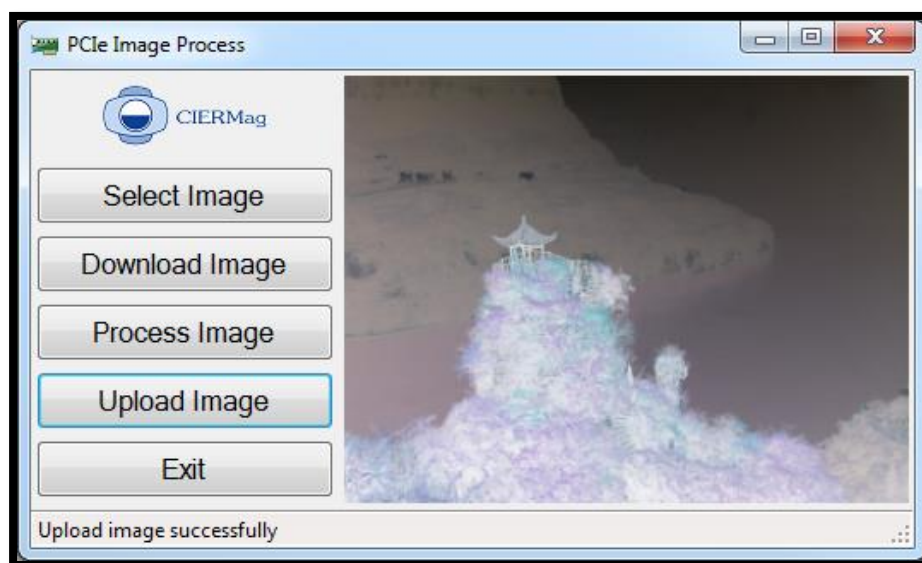

Figura 40 - Programa de teste do *framework* PCI *Express* para *Windows* com o resultado do processamento da imagem pela FPGA.

4.2.2.2 Linux

Aplicação de teste para Linux foi um pouco mais simples. Apesar de utilizar o mesmo *hardware* de inversão de cores de uma imagem, o objetivo era o envio de dados genéricos para a validação do *driver* e da biblioteca.

Aproveitando a funcionalidade dessa aplicação em enviar dados genéricos, foram transferidas várias quantidades diferentes de dados e mediram-se os tempos das transferências para que fosse possível a elaboração da Tabela 4 com o resumo das velocidades de transferências obtidas com o *framework* utilizando 4 *lanes* PCI *Express* versão 2.

Tabela 4 - Valores de velocidade para as transferências utilizando o *framework* PCI *Express* com 64KB de *buffer* no *driver*.

Tamanho dos dados [B]	Velocidade de leitura [MB/s]	Velocidade de escrita [MB/s]	*Throughput* de leitura [Mbits/s]	*Throughput* de escrita [Mbits/s]
262144	889,0	1556,5	7112	12452
131072	889,0	1556,5	7112	12452
65536	889,0	1555,3	7112	12443
32768	889,1	1524,9	7113	12200
16384	889,3	1463,9	7114	11711
8192	889,7	1354,5	7117	10836
4096	890,4	1187,9	7123	9503
2048	892,0	917,6	7136	7341
1024	895,1	666,7	7161	5333
512	901,4	423,8	7211	3391
256	914,3	242,4	7314	1939
128	941,2	139,1	7529	1113

Como podem ser observadas as velocidades conseguidas superam outras interfaces mais comuns de comunicação com um computador. O que torna atrativo a utilização do PCI *Express* para comunicação com unidades que necessitem de transferências em alta velocidade.

Além disso, alguns fatores são interessantes de se notar como, por exemplo: a velocidade de leitura ser mais constante e com valor máximo menor do que a de escrita. Isso se deve ao fato que os pacotes enviados tinham tamanho máximo de 128 *bytes* para a leitura, que é o valor padrão para esse tipo de operação. A escrita por sua vez foi feita em pacotes de 4 KB fazendo com que o *overhead* seja menor.

A fim de testar as características do *driver*, outros valores de velocidade foram obtidos utilizando o *buffer* de dados do *driver* com 4 KB ao invés de 64KB (Tabela 5).

Tabela 5 - Valores de velocidade para as transferências utilizando o *framework* PCI *Express* com 4KB de *buffer* no *driver*.

Tamanho dos dados [B]	**Velocidade de leitura [MB/s]**	**Velocidade de escrita [MB/s]**	***Throughput* de leitura [Mbits/s]**	***Throughput* de escrita [Mbits/s]**
262144	890,4	1180,3	7123	9442
131072	890,4	1180,6	7123	9445
65536	890,4	1181,3	7123	9450
32768	890,4	1180,7	7123	9446
16384	890,4	1180,4	7123	9443
8192	890,4	1186,6	7123	9492
4096	890,4	1187,9	7123	9503
2048	892,0	944,6	7136	7557
1024	895,1	659,8	7161	5278
512	901,4	418,3	7211	3346
256	914,3	240,6	7314	1925
128	941,2	141,6	7529	1133

Nota-se claramente a diminuição da velocidade máxima da escrita, uma vez que o número maior de alocações de memória pelo *driver* consome mais tempo. Portanto, o fator limitador nesse caso é a alocação de memória e não o duto em si.

5 Conclusões

A partir da realização desse projeto foi possível obter conhecimento sobre um leque extenso de assuntos como: FPGAs; o padrão PCI *Express*; a utilização de um *hardware* programável para a comunicação em alta velocidade; o funcionamento de um sistema operacional; o funcionamento de um *driver*; a utilização do método DMA; programação em VHDL; programação em C para *drivers* e a programação em C++.

Como o objetivo principal foi a criação de um *framework* que facilitasse a utilização do PCI *Express* com FPGA, o trabalho realizado conseguiu cumprir essa especificação com a transmissão e recepção de dados pelo PCI *Express* realizada com o kit de desenvolvimento TR4 da Terasic.

A interface criada é simples de utilizar através de funções em C e possibilitam a qualquer desenvolvedor utilizar desses comandos em qualquer linguagem de programação, proporcionando assim uma grande flexibilidade, uma vez que qualquer aplicação pode ser desenvolvida para acesso a FPGA.

Na FPGA basta a instanciação do componente e sua utilização como um duto de dados simples. Pela análise dos resultados da síntese, o pequeno consumo de lógica do *framework* possibilita a utilização mesmo com aplicações que utilizem grande parte da lógica da FPGA. Uma das preocupações durante o desenvolvimento da lógica de *hardware* foi evitar desperdício de lógica, para que o sistema utilizasse pouca quantidade de lógica e para que não ocorressem problemas de temporização durante a síntese.

As velocidades de transferências de dados são superiores a outros tipos de interfaces de comunicações, como *ethernet* e USB, portanto as aplicações com esse tipo de *framework* vão além do que se pode alcançar com os tipos mais convencionais de interface nos computadores atuais.

Portanto, o objetivo primordial deste trabalho foi cumprido, ou seja, tendo-se esse sistema funcional como observado pela validação no sistema Linux.

As dificuldades encontradas foram superadas através de pesquisas constantes aos manuais e regulamentações das interfaces e dos dispositivos utilizados. Contudo as maiores dificuldades foram aprender o funcionamento exato das estruturas internas do PCI *Express*, bem como, do funcionamento de um *driver*.

6 Trabalhos futuros

Alguns trabalhos futuros ainda podem ser realizados nesse assunto, como o desenvolvimento de um *driver* para *Windows*, e bibliotecas em *python*, que é a linguagem já utilizada no *software* desenvolvido no CIERMag, possibilitando a utilização em alto nível do *framework*. Podem ser feitas melhorias no sistema para que as velocidades de transferência aproximem-se das velocidades teóricas e a otimização de todo código. Com esse sistema pronto, os trabalhos futuros poderão basear-se na utilização desse *framework* como uma ferramenta de trabalho.

Para isso alguns trabalhos como desenvolvimento de sistemas de aquisição de alta velocidade, ou de comunicação utilizando fibra ótica ou até mesmo a execução de algoritmos paralelizados em *hardware* podem aproveitar do *framework* desenvolvido para enviar os dados ao computador em alta velocidade.

Referências Bibliográficas

[1] R. E. Johnson, “Documenting Frameworks using Pattern,” 1992. [Online]. Available: ftp://ssel.vub.ac.be/education/SoftArch/normal/projects/HotDraw/documenting.pdf. [Acesso em 29 Junho 2015].

[2] Xilinx Inc., “Field Programmable Gate Array (FPGA),” Xilinx Inc., 2015. [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. [Acesso em 29 Junho 2015].

[3] Xilinx Inc., “What is Programmable Logic?,” Xilinx Inc., 2015. [Online]. Available: http://www.xilinx.com/company/about/programmable.html. [Acesso em 29 Junho 2015].

[4] C. R. Nave, “The D Flip-Flop,” HyperPhysics, 2012. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/dflipflop.html. [Acesso em 14 Abril 2015].

[5] J. P. Nicolle, “fpga4fun.com - How FPGAs work,” fpga4fun.com & KNJN LLC, 14 Outubro 2013. [Online]. Available: http://www.fpga4fun.com/FPGAinfo2.html. [Acesso em 14 Abril 2015].

[6] J. P. Nicolle, “fpga4fun.com - FPGA pins,” fpga4fun.com & KNJN LLC, 20 Maio 2013. [Online]. Available: http://www.fpga4fun.com/FPGAinfo4.html. [Acesso em 14 Abril 2015].

[7] J. Winkles, “PCI Express® Basics,” PCI-SIG, 2006. [Online]. Available: http://kavi.pcisig.com/developers/main/training_materials/get_document?doc_id=689c1035c602d4f6dc03f2b928c3b2e182462a17. [Acesso em 29 Junho 2015].

[8] J. P. Nicolle, “fpga4fun.com - PCI Express - The transaction layer,” fpga4fun.com & KNJN LLC, 15 Fevereiro 2012. [Online]. Available: http://www.fpga4fun.com/PCI-Express4.html. [Acesso em 27 Abril 2015].

[9] Xillybus Ltd., “Down to the TLP: How PCI express devices talk (Part I),” Xillybus Ltd., [Online]. Available: http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1. [Acesso em 20 Abril 2015].

[10] Altera Corporation, “Quartus II - Overview,” Altera Corporation, 2015. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-ii/overview.html. [Acesso em 05 Maio 2015].

[11] GCC team, “GCC, the GNU Compiler Collection,” Free Software Foundation, Inc, 27 Abril 2015. [Online]. Available: https://gcc.gnu.org/. [Acesso em 05 Maio 2015].

[12] Microsoft, “Visual Studio - Microsoft Developer Tools,” Microsoft, 2015. [Online]. Available: https://www.visualstudio.com/. [Acesso em 05 Maio 2015].

[13] Altera Corporation, “IP Compiler for PCI Express User Guide,” Agosto 2014. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_pci_express.pdf. [Acesso em 14 Abril 2015].

[14] J. Corbet, A. Rubini e G. Kroah-Hartman, Linux Device Drivers, Sebastopol, California:

O'Reilly Media, 2005.

[15] Canonical Ltd., “Ubuntu for desktops,” Canonical Ltd., 2015. [Online]. Available: http://www.ubuntu.com/desktop. [Acesso em 19 Maio 2015].

[16] Microsoft, “Windows - Microsoft Windows,” Microsoft, 2015. [Online]. Available: http://windows.microsoft.com/en-us/windows/home. [Acesso em 19 Maio 2015].

[17] Jungo Connectivity Ltd., “WinDriver – PCI/USB Device Driver Development Tool,” Jungo Connectivity Ltd., 2015. [Online]. Available: http://www.jungo.com/st/products/windriver/. [Acesso em 19 Maio 2015].

[18] S. Brown e J. Rose, “Architecture of FPGAs and CPLDs: A Tutorial,” 1996. [Online]. Available: http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf. [Acesso em 14 Abril 2015].

[19] J. P. Nicolle, “fpga4fun.com - Internal RAM operation,” fpga4fun.com & KNJN LLC, 19 Maio 2013. [Online]. Available: http://www.fpga4fun.com/FPGAinfo3.html. [Acesso em 14 Abril 2015].

[20] T. V. Wilson, “PCIe Lanes - How PCI Express Works,” HowStuffWorks, [Online]. Available: http://computer.howstuffworks.com/pci-express1.htm. [Acesso em 20 Abril 2015].

[21] T. V. Wilson, “PCIe Lanes - How PCI Express Works,” HowStuffWorks, [Online]. Available: http://computer.howstuffworks.com/pci-express2.htm. [Acesso em 20 Abril 2015].

[22] J. P. Nicolle, “fpga4fun.com - PCI Express - Connector,” fpga4fun.com & KNJN LLC, 15 Fevereiro 2012. [Online]. Available: http://www.fpga4fun.com/PCI-Express1.html. [Acesso em 20 Abril 2015].

[23] H. S. Arora, “PCIE Protocol Layers,” ICVerification.com, 2013. [Online]. Available: http://www.icverification.com/BusProtocols/PCIE4.php. [Acesso em 28 Abril 2015].

[24] H. S. Arora, “Function of Each PCIE Layer,” ICVerification.com, 2013. [Online]. Available: http://www.icverification.com/BusProtocols/PCIE5.php. [Acesso em 28 Abril 2015].

[25] Terasic Inc., “Terasic TR4 FPGA Development Kit,” Terasic Inc., 2013. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=138&No=683&PartNo=1. [Acesso em 04 Maio 2015].

[26] Terasic Inc., “Terasic TR4 FPGA Development Kit,” Terasic Inc., 2013. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=67&No=704&PartNo=2. [Acesso em 29 Junho 2015].

[27] H. Miller, “Understanding PCI Configuration Space,” BSOD Tutorials, 23 Janeiro 2014. [Online]. Available: http://bsodtutorials.blogspot.com.br/2014/01/understanding-pci-configuration-space.html. [Acesso em 29 Junho 2015].

[28] V. Hindriksen, “GPUDirect and DirectGMA – direct GPU-GPU communication via RDMA,” StreamComputing BV, 18 Abril 2015. [Online]. Available:

http://streamcomputing.eu/blog/2015-04-18/gpudirect-and-directgma-direct-gpu-gpu-communication/. [Acesso em 29 Junho 2015].