UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

TIAGO AMARO MARTINS

**Utilização de um framework PCI
Express® em um espectrômetro digital
de ressonância magnética**

São Carlos
2017

TIAGO AMARO MARTINS

Utilização de um framework PCI Express® em um espectrômetro digital de ressonância magnética

Dissertação apresentada à Escola de Engenharia de São Carlos da Universidade de São Paulo para obtenção do título de Mestre em Ciências, Programa de Engenharia Elétrica.

Área de Concentração: Processamento de Sinais e Instrumentação

Orientador: Prof. Dr. Homero Schiabel

Trata-se da versão corrigida da dissertação. A versão original se encontra disponível na EESC/USP que aloja o Programa de Pós-Graduação de Engenharia Elétrica.

São Carlos
2017

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.

M379u   Martins, Tiago Amaro
Utilização de um framework PCI Express® em um espectrômetro digital de ressonância magnética / Tiago Amaro Martins; orientador Homero Schiabel. São Carlos, 2017.

Dissertação (Mestrado) - Programa de Pós-Graduação em Engenharia Elétrica e Área de Concentração em Processamento de Sinais e Instrumentação -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2017.

1. Espectrômetro. 2. Ressonância Magnética. 3. FPGA. 4. PCI Express. 5. Device Driver. I. Título.

FOLHA DE JULGAMENTO

Candidato: Engenheiro **TIAGO AMARO MARTINS**.

Título da dissertação: "Utilização de um framework PCI Express® em um espectrômetro digital de ressonância magnética".

Data da defesa: 23/06/2017.

Comissão Julgadora:

Resultado:

Prof. Associado **Homero Schiabel**
(Orientador)
(Escola de Engenharia de São Carlos/EESC)

APROVADO

Prof. Dr. **Alberto Tannus**
(Instituto de Física de São Carlos/IFSC)

APROVADO

Dr. **André Torre Neto**
(Empresa Brasileira de Pesquisa Agropecuária/EMBRAPA)

APROVADO

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica:
Prof. Associado **Luis Fernando Costa Alberto**

Presidente da Comissão de Pós-Graduação:
Prof. Associado **Luis Fernando Costa Alberto**

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.

Aos meus irmãos Rodrigo e Henrique pelos bons momentos durante esses anos de nossas vidas.

Ao Prof. Dr. Homero Schiabel, pela orientação, confiança e pelas oportunidades que me foram dadas.

Ao Prof. Dr. Alberto Tannús, pelo grande suporte, confiança e pelas oportunidades, que a mim, foram apresentadas.

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

A todos os amigos do Centro de Imagens e Espectroscopia *in vivo* por Ressonância Magnética (CIERMag) pela amizade e apoio durante o desenvolvimento desse trabalho.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa concedida.

RESUMO

MARTINS, T. A. **Utilização de um framework PCI Express® em um espectrômetro digital de ressonância magnética.** 2017. 93 f. Dissertação (Mestrado) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2017.

O foco central desse trabalho é a utilização e aprimoramento de um *framework Peripheral Component Interconnect Express* (PCI Express®) para a comunicação de dados em um Espectrômetro Digital de Ressonância Magnética (*Digital Magnetic Resonance Spectrometer*–DMRS) utilizando o conceito de *Field-Programmable Gate Array* (FPGA). Esse trabalho foi desenvolvido para servir como base de comunicação para o Espectrômetro Digital 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. A integração dessa nova comunicação, entre o *software* e o *hardware* do espectrômetro, mantém compatibilidade com as interfaces já existentes possibilitando a execução de todas as sequências desenvolvidas sem nenhuma alteração. A incorporação da comunicação PCI Express provê uma solução com um número menor de etapas por transferência em comparação com a comunicação *Ethernet*. Com isso é possível aumentar o desempenho do sistema e obter taxas de transferência mais elevadas. Para isso, foram feitas mudanças no *hardware* de forma a torná-lo mais eficiente, reduzindo o número de ciclos de *clock* por operação e também a quantidade de lógica sintetizada. Além disso, a latência do *software* durante as transferências também foi reduzida através da utilização de interrupções *Message Signaled Interrupt* (MSI) e do método *Scatter and Gather* usado para reduzir a quantidade de cópias de dados na memória principal do computador. Dessa forma, obteve-se, como resultados reais, uma taxa de transferência efetiva (*throughput*) de 97% do valor máximo da banda possível do barramento PCI Express.

Palavras-chave: FPGA, PCI Express, Ressonância magnética, Espectrômetro Digital.

ABSTRACT

MARTINS, T. A. **Utilization of a PCI Express® framework in a digital magnetic resonance spectrometer.** 2017. 93 f. Dissertação (Mestrado) – São Carlos School of Engineering, University of São Paulo, São Carlos, 2017.

The central focus of this work is the implementation and use of a Peripheral Component Interconnect Express (PCI Express®) framework for data communication on a Digital Magnetic Resonance Spectrometer (DMRS) using the concept of Field-Programmable Gate Array (FPGA). This work is being developed to serve as a communication basis for the magnetic resonance Digital Spectrometer of the Centro de Imagens e Espectroscopia in vivo por Ressonância Magnética (CIERMag) due to demand of high transfer rates of acquired data. The integration of this new communication, between spectrometer software and hardware, keeps compatibility with existing interfaces, making it possible to execute all developed magnetic resonance sequences without any change. The incorporation of PCI Express communication provides solution with a lower number of steps per transfer when compared to Ethernet communication. By this means it's possible to increase system performance and, as result, have higher transfer rates. To accomplish that, the number of clock cycles per operation was reduced, so was the synthesized logic. Furthermore, software latency for data transfer was also reduced consequence of MSI interruption implementation and the use of Scatter and Gather method to remove data movement across the computer main memory. Therefore, it was obtained, as measured real result, a throughput value of 97% the theoretical maximum value for the hardware.

Keywords: FPGA, PCI Express, Magnetic resonance, Digital Spectrometer.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diagrama do fluxo de dados do espectrômetro utilizando Ethernet Gigabit.....	18
Figura 2 - Diagrama do fluxo de dados do espectrômetro utilizando PCI Express.....	18
Figura 3 - Estrutura de uma FPGA.....	22
Figura 4 - Esquemático de um CLB simples.....	23
Figura 5 - Esquemático do encadeamento de várias células lógicas.....	23
Figura 6 - Esquemático de interconexão especial para o carry.....	24
Figura 7 - Esquemático do encadeamento de células lógicas com células de entrada e saída..	25
Figura 8 - Esquemático do bloco de memória com as células lógicas da FPGA.....	26
Figura 9 - Exemplo de utilização de DMA (AMD DirectGMA).....	29
Figura 10 - Comparação entre os padrões PCI e PCI Express.....	32
Figura 11 - Exemplo de topologia PCI Express.....	32
Figura 12 - Diagrama das camadas PCI Express em uma comunicação entre dois dispositivos	33
Figura 13 - Diagrama da comunicação entre duas camadas físicas de dois dispositivos PCI Express.....	34
Figura 14 - Representação de links e lanes na comunicação PCI Express.....	35
Figura 15 - Conector PCI Express para conexão x1.....	36
Figura 16 - Conector PCI Express x1 em placa de circuito.....	37
Figura 17 - Diagrama da comunicação entre duas camadas de vinculação de dados de dois dispositivos PCI Express.....	37
Figura 18 - Diagrama de empacotamento da camada de vinculação de dados.....	38
Figura 19 - Diagrama da comunicação entre duas camadas de transação de dois dispositivos PCI Express.....	39
Figura 20 - Diagrama do empacotamento da camada de transação.....	40
Figura 21 - Exemplo de TLP de requisição de escrita de memória com endereçamento de 32 bits.....	42
Figura 22 - Exemplo de TLP de requisição de leitura de memória com endereçamento de 32 bits.....	44
Figura 23 - Exemplo de TLP de complementação.....	44
Figura 24 - Diagrama do espaço de configuração PCI.....	47
Figura 25 - Diagrama geral do espectrômetro de ressonância magnética. Em laranja está enfatizado o módulo ao qual este trabalho se refere.....	49
Figura 26 - Kit de desenvolvimento Terasic DE4.....	49
Figura 27 - Diagrama completo dos módulos de hardware e software desenvolvidos para a comunicação do computador com o espectrômetro digital de ressonância magnética.....	51
Figura 28 - Diagrama dos módulos de hardware da interface entre o DMRS e o computador	52
Figura 29 - Diagrama dos sinais digitais de transmissão de dados com destaque nos ciclos por pacote.....	57

Figura 30 - Diagrama dos sinais digitais de transmissão de dados com destaque aos ciclos de espera.....	58
Figura 31 - Diagrama dos módulos de software da interface espectrômetro-computador.....	60
Figura 32 - Programa ACQServer utilizando a interface PCI Express.....	71
Figura 33 - Curvas do throughput de dados com as modificações no framework PCI Express	73
Figura 34 - Curvas de comparação entre o throughput escrita do hardware antes e depois desse trabalho.....	74
Figura 35 - Curvas de comparação entre o throughput leitura do hardware antes e depois desse trabalho.....	75
Figura 36 - Ganho percentual de velocidade do hardware.....	75
Figura 37 - Comparação percentual da utilização de recursos da FPGA antes e depois das modificações realizadas.....	77
Figura 38 - Exemplo de código de verificação de fim da transmissão DMA utilizando polling	78
Figura 39 - Curvas de throughput de leitura pelo software para cada método de espera.....	80
Figura 40 - Curvas de throughput de escrita pelo software para cada método de espera.....	81
Figura 41 - Curvas de throughput de leitura com utilização ou não do método de Scatter & Gather.....	82
Figura 42 - Curvas de throughput de escrita com utilização ou não do método de Scatter & Gather.....	82
Figura 43 - Curvas de comparação entre a utilização de diferentes números de lanes para a leitura.....	85
Figura 44 - Curvas de comparação entre a utilização de diferentes números de lanes para a leitura.....	85

LISTA DE TABELAS

Tabela 1 – Desempenho teórico do PCI Express.....	31
Tabela 2 – Descrição do campo Fmt do cabeçalho de um TLP.....	41
Tabela 3 – Descrição da estrutura do registrador de estado.....	55
Tabela 4 – Descrição dos bits do registrador de estado de comandos.....	55
Tabela 5 – Descrição dos bits do registrador de estado de dados.....	55
Tabela 6 – Medidas de tempo de resposta e throughput das transferências no hardware com 4 lanes.....	73
Tabela 7 – Recursos utilizados na FPGA após a síntese utilizando o Quartus Prime para 4 lanes.....	76
Tabela 8 – Tempo de resposta e throughput das transferências no software utilizando o método polling.....	79
Tabela 9 – Medidas de tempo de resposta e throughput das transferências no software utilizando o método de interrupção.....	80
Tabela 10 - Recursos utilizados na FPGA após a síntese utilizando o Quartus Prime para 8 lanes.....	84
Tabela 11 - Medidas de tempo de resposta e throughput das transferências no hardware com 8 lanes.....	84

SUMÁRIO

1 INTRODUÇÃO.....	17
1.1 Justificativa.....	17
1.2 Objetivo.....	19
1.3 Organização do Trabalho.....	19
2 EMBASAMENTO TEÓRICO.....	21
2.1 Field-Programmable Gate Array.....	21
2.1.1 Blocos de Lógica Programável.....	22
2.1.2 Interconexões.....	23
2.1.3 Blocos de Entrada e Saída.....	24
2.1.4 Memória Interna.....	26
2.2 Gerenciamento de Memória em um Sistema Operacional.....	27
2.2.1 Memória Virtual.....	27
2.3 Direct Memory Access.....	28
2.3.1 Transferências Simples.....	29
2.3.2 Scatter & Gather.....	30
2.4 Peripheral Component Interconnect Express.....	30
2.4.1 Topologia.....	31
2.4.2 Camadas do Protocolo.....	33
2.4.3 Outros recursos.....	46
2.4.4 Espaço de Configuração.....	47
3 DESENVOLVIMENTO.....	49
3.1 Visão Geral.....	50
3.2 Hardware.....	52
3.2.1 Encapsulamento.....	53
3.2.2 Aprimoramentos.....	56
3.3 Software.....	60
3.3.1 Classes e Objetos.....	61
3.3.2 Device Driver.....	62
4 RESULTADOS.....	71
4.1 Hardware.....	72
4.1.1 Desempenho.....	72
4.1.2 Recursos utilizados da FPGA.....	76
4.2 Software.....	78
4.2.1 Fim da transmissão.....	78
4.2.2 Scatter & Gather.....	81
4.2.3 Link x8.....	83
5 CONCLUSÕES.....	87
REFERÊNCIAS.....	89

1 INTRODUÇÃO

A proposta deste projeto entra no escopo de um empreendimento científico e tecnológico adotado pelos pesquisadores do Centro de Imagens e Espectroscopia *in vivo* por Ressonância Magnética (CIERMag), o qual tem como objetivo a disseminação, disponibilização e acessibilidade das técnicas de Imagens por Ressonância Magnética (IRM) à comunidade científica em geral. O foco não se restringe apenas ao uso clínico, mas a aplicação dessas técnicas em outras áreas da ciência que se beneficiam com o uso de IRM nas mais diversas aplicações.

A limitação dos equipamentos de IRM origina-se da parametrização dos equipamentos comerciais existentes para a anatomia ou patologia a ser observada e não voltada propriamente ao fenômeno de Ressonância Magnética (RM). Dessa forma, esse trabalho faz parte do desenvolvimento de um Espectrômetro Digital que se destaca por proporcionar meios de fácil adaptação do instrumento às diferentes aplicações existentes.

1.1 Justificativa

O desenvolvimento desse trabalho baseia-se em agregar um novo método de comunicação com o Espectrômetro Digital de Ressonância Magnética (Digital Magnetic Resonance Spectrometer—DMRS) do CIERMag que proporcione altas taxas de transferência dos dados adquiridos. Dessa forma, removem-se as barreiras de comunicação impostas pelos métodos atualmente empregados e, portanto, as funcionalidades e aplicações do dispositivo podem ser expandidas.

O DMRS desenvolvido no CIERMag consiste de duas frentes: o *hardware* desenvolvido em *Field-Programmable Gate Array* (FPGA) e o *software* que é executado em computadores *desktop*. A comunicação entre essas duas frentes era realizada através de *Ethernet Gigabit*, que, para uma expansão do número de receptores do espectrômetro além de 16, não é capaz de suprir as velocidades de comunicação necessárias. A transição da plataforma de comunicação para o *framework* PCI Express®, também desenvolvido no centro (MARTINS, 2015), soluciona o problema provendo uma comunicação de alta velocidade e escalabilidade.

O método de comunicação empregado antes da realização desse trabalho utilizava pacotes *User Datagram Protocol* (UDP), da camada *Transmission Control Protocol/Internet*

Protocol (TCP/IP) do sistema operacional para a transferência dos dados. Para isso, os dados passam por diversos processos de empacotamento e desempacotamento, além da grande movimentação desses pela memória do sistema operacional. Todas essas etapas (Figura 1) introduzem perdas de velocidade de transferência, que para um sistema com um fluxo muito grande de dados, se traduzem em diminuição do desempenho.

Figura 1 - Diagrama do fluxo de dados do espectrômetro utilizando *Ethernet Gigabit*

Fonte: Autoria própria.

O novo método permite velocidades de transferência muito superiores por utilizar o protocolo PCI Express. Com esse protocolo, diversas etapas intermediárias são removidas (Figura 2), portanto, há um ganho de velocidade durante a transmissão desses dados. Em alguns casos, o dispositivo acessa diretamente a memória da aplicação, proporcionando ainda maior redução no tempo total de transferência de dados.

Figura 2 - Diagrama do fluxo de dados do espectrômetro utilizando PCI Express

Fonte: Autoria própria.

1.2 Objetivo

O volume de dados adquiridos por um espectrômetro de ressonância magnética é resultado de uma série de parâmetros do próprio espectrômetro, sendo um desses parâmetros o número de canais de recepção de Radiofrequência (RF). As técnicas mais recentes de ressonância magnética utilizam um número relativamente grande de canais, que aliada a uma alta taxa de digitalização, implica em um grande volume de dados adquiridos e que devem ser transportados para o processamento.

O objetivo desse trabalho foi implantar a comunicação entre o DMRS em FPGA e o *software* do sistema rodando em computadores com sistema operacional (SO) Windows® e Linux®, utilizando para isso o *framework* PCI Express.

Para executar essa tarefa foi necessário desenvolver novos módulos tanto de *hardware* quando de *software*. Os módulos de *hardware* foram necessários para integração do *framework* com o espectrômetro em FPGA e os de *software* consistem, além da interligação do programa de aquisição de dados existente no CIERMag com o *framework*, no desenvolvimento de um *device driver* (módulo que roda no núcleo do SO).

Durante a implantação da comunicação, um estudo detalhado permitiu uma melhoria do *framework* em relação aos recursos utilizados na FPGA, bem como da velocidade de transferência (*throughput*) dos dados. Para isso, a implementação de novos métodos de *Direct Memory Access* (DMA), que resultou na criação de mais módulos em *hardware* e rotinas no *device driver*.

1.3 Organização do Trabalho

Este trabalho foi dividido em cinco capítulos. O primeiro consiste de uma introdução das razões e objetivos desse trabalho. O segundo detalha todo o embasamento teórico necessário pelo projeto. No terceiro encontram-se os métodos e os detalhes de desenvolvimento desse trabalho. No quarto capítulo são apresentados os resultados do desenvolvimento. Por fim, o quinto compõe-se da conclusão referente ao projeto realizado.

2 EMBASAMENTO TEÓRICO

O conteúdo abordado por esse trabalho envolve quatro áreas de conhecimento. A primeira delas é a síntese de lógica digital utilizando *Field-Programmable Gate Array* (FPGA). O entendimento de como é a estrutura interna desse tipo de dispositivo, ajuda a compreender melhor como a linguagem de descrição de *hardware* é convertida em elementos lógicos. A segunda área é o gerenciamento de memória de um sistema operacional. As transferências de dados envolvem leitura e escrita na memória principal do computador, tornando a compreensão desse recurso um conhecimento fundamental para otimização das transferências. Com isso, a terceira área é o acesso direto a memória sem necessidade de intervenção por parte do processador, possibilitando um conjunto de vantagens para movimentação de grandes quantidades de dados. A quarta área de conhecimento abordada é o padrão PCI Express, que permite grandes velocidades de transferências e melhor escalabilidade para o espectrômetro de Ressonância Magnética.

2.1 *Field-Programmable Gate Array*

Os dispositivos semicondutores *Field-Programmable Gate Array* (FPGA) são circuitos integrados que podem ser reconfigurados para uma funcionalidade desejada (XILINX, INC., 2015a).

Antes da invenção da FPGA, para permitir a utilização de dispositivos digitais de forma mais flexível e construção de funções de lógica combinacional, eram utilizadas memórias programáveis de somente leitura (*Programmable Read-Only Memory* - PROM). Porém, por problemas de velocidade e consumo de energia foram substituídas por outros dispositivos de lógica programável (*Programmable Logic Devices* - PLD).

Programmable Logic Array (PLA), *Programmable Array Logic* (PAL), *Generic Array Logic* (GAL), *Complex Programmable Logic Device* (CPLD) (XILINX, INC., 2015b) e FPGA são exemplos de diferentes tipos de PLD criados. As FPGA, porém, se destacaram dos demais por não possuírem portas lógicas, mas conterem tabelas verdade que são configuradas para representar os resultados possíveis do circuito combinacional desejado.

A base desses dispositivos é um conjunto de blocos de lógica programável (*Configurable Logic Blocks* - CLB), que são interconectados por vias reconfiguráveis e blocos

de entrada e saída (I/O Block), como representado na Figura 3 (BROWN; ROSE, 1996). Cada célula de lógica é capaz de executar operações booleanas e armazenamento do resultado.

Figura 3 - Estrutura de uma FPGA

Fonte: Brown, Rose (1996).

2.1.1 Blocos de Lógica Programável

CLB são os núcleos de todas as FPGAs. A base de cada uma dessas células é composta por uma pequena tabela verdade (*Look-Up Table* - LUT), um dispositivo de armazenamento (*flip-flop*) do tipo D (NAVE, 2012) e um multiplexador, tornando opcional a utilização do *flip-flop* no bloco (NICOLLE, 2013c). Cada fabricante e cada família de FPGA possui CLB com características diferentes. Apesar de todas utilizarem o mesmo princípio, algumas possuem LUT com maior ou menor número de entradas e ou saídas. Outras possuem componentes extras como, por exemplo, lógica de vai um (*carry*), registradores de deslocamento (*shifters*) ou somadores.

LUT são memórias de acesso randômico (*Random Access Memory* - RAM), as quais podem ser gravadas com o resultado de qualquer função lógica. E justamente por serem memórias voláteis é que a informação não é retida na FPGA em caso de perda de energia. Comumente as LUT possuem quatro entradas, como representado na Figura 4. Portanto, qualquer função lógica booleana de até quatro entradas pode ser descrita utilizando apenas uma dessas unidades. Para funções com um número superior de entradas utiliza-se uma combinação de CLB (ALTERA CORPORATION, 2006), logo, são necessários caminhos de interconexão para a composição de funções complexas.

Figura 4 - Esquemático de um CLB simples

Fonte: Nicolle (2013c).

Como as tabelas verdade, bem como as interconexões são formadas por memórias Static Random-Access Memory (SRAM) (KUON; TESSIER; ROSE, 2007) as FPGA não são capazes de armazenar sua configuração após uma perda de energia. Dessa forma, outros circuitos normalmente são conectados as FPGA para armazenar as configurações durante o período que o sistema não está energizado e também para reconfigurar a FPGA após o reestabelecimento de energia.

2.1.2 Interconexões

Para obter maior flexibilidade, as vias que conectam os diversos CLB, em uma FPGA (Figura 5), também são programáveis. Esses caminhos são compostos por fios e multiplexadores, que são distribuídos ao redor das células lógicas para formar dutos de comunicação. Tais dutos possuem grande importância no desenvolvimento de sistemas digitais, pois é por esses dutos que as informações são transportadas entre os CLB.

Figura 5 - Esquemático do encadeamento de várias células lógicas

Fonte: Nicolle (2013c).

Além do caminho de dados entre as células, existem caminhos especiais dedicados aos sinais de relógio (*clock*) do sistema. Esses são desenhados de forma a apresentar um atraso menor de transporte para evitar problemas de metaestabilidade.

Caminhos do sinal de *carry* também são comuns entre CLB vizinhas, pois, esses auxiliam a realização de operações aritméticas ao serem cascadeados (Figura 6). A existência e utilização desses caminhos vêm da necessidade de descongestionar as vias de interconexão, consequentemente, reduzindo o número de células lógicas utilizadas e possibilitando aumento da velocidade de operação.

Devido ao grande número de células e estruturas dentro da FPGA, o roteamento dos dados por toda a malha da FPGA é realizado por ferramentas *Computer Aided Design* (CAD). Os atrasos decorrentes da distância que o sinal deve percorrer podem ocasionar erros nos resultados. Dessa forma, o desenvolvimento de toda a lógica deve sempre ser seguido de uma análise de como o circuito será efetivamente mapeado na FPGA.

Figura 6 - Esquemático de interconexão especial para o *carry*

Fonte: Nicolle (2013c).

2.1.3 Blocos de Entrada e Saída

A comunicação entre a FPGA e o mundo externo é feita através de outro tipo de estrutura chamado bloco de entrada e saída, ou célula de entrada e saída. Além de realizar a conexão entre a lógica interna com os pinos do encapsulamento, esses blocos condicionam o sinal elétrico que entra e sai da FPGA (Figura 7).

Figura 7 - Esquemático do encadeamento de células lógicas com células de entrada e saída

Fonte: Nicolle (2013c).

Entre as células de entrada e saída existem algumas com funções específicas. Por isso os pinos da FPGA são divididos em dois grupos: pinos dedicados e do usuário (NICOLLE, 2013b).

2.1.3.1 Pinos dedicados

Os pinos de uma FPGA, que possuem uma função previamente atribuída pelo fabricante e não podem ser alterados, são chamados de pinos dedicados. Esses são classificados em três subcategorias:

- **Pinos de alimentação:** São pinos responsáveis por prover as tensões de operação da FPGA e podem ser divididos em duas classes:
  - o **Núcleo:** Pinos que alimentam os componentes internos como LUT e *flip-flops*.
  - o **Entrada e Saída:** Pinos que alimentam as células de entrada e saída e devem possuir tensões compatíveis com as tensões dos dispositivos externos conectados a essas células.
- **Pinos de configuração:** Esses pinos são utilizados para carregar as informações nas LUT e nas interconexões da FPGA. Por esses pinos são carregados toda configuração binária (*bit stream*) compilada para a FPGA.
- **Pinos de clock ou entradas dedicadas:** Existem pinos conectados a dutos especiais da interconexão da FPGA, com o objetivo de fluir uma corrente maior possibilitando aumento do *fan-out*¹. Estes são utilizados para sinais com várias derivações, por exemplo, sinais de *clock*.

¹*Fan-out*: Número de componentes que podem ser conectados a saída de uma porta lógica (TALA, 2014).

2.1.3.2 Pinos do usuário

Os outros pinos da FPGA, que não tem função previamente especificada, são chamados de pinos do usuário ou pinos de entrada e saída (*Input/Output - I/O*). A direção dos dados por esse pino pode ser configurada como entrada, saída ou bidirecional, utilizando *buffers tri-state*. As células lógicas podem ser conectadas a esses pinos e é com esse tipo de pino que entram e saem todos os dados utilizados pela FPGA.

2.1.4 Memória Interna

A maioria dos circuitos digitais necessita de alguma espécie de armazenamento de dados. As duas principais formas de armazenamento de dados em uma FPGA são: *Blockram* e *Distributed RAM* (NICOLLE, 2013a).

Para facilitar o armazenamento de quantidades razoáveis de dados na FPGA, os fabricantes incluem blocos de memória na malha da FPGA. Esses blocos são chamados *Blockrams*. Normalmente existem em pequena quantidade, na ordem de algumas centenas, porém possuem boa capacidade de armazenamento. Essas memórias são conectadas a estrutura de interconexão da FPGA possibilitando a leitura e escrita pelos outros blocos internos (Figura 8).

Figura 8 - Esquemático do bloco de memória com as células lógicas da FPGA

Fonte: Nicolle (2013a).

O número de portas de um *Blockram* corresponde ao número de acessos simultâneos à memória. As memórias de múltiplas portas tem grande importância em circuitos com

diferentes domínios de *clock*, pois podem ser utilizadas para a transferência de dados entre os domínios.

Distributed RAM, em oposição, são pequenas unidades de memória espalhadas por toda FPGA, esse tipo de memória utiliza as próprias células lógicas para armazenar os dados. As memórias distribuídas são muito versáteis e são utilizadas quando há a necessidade de uma configuração de memória que não é possível ser obtida com os blocos existentes, ou quando o armazenamento desejado é muito pequeno e acarretaria em desperdício dos *Blockrams*. Além disso, a configuração *Distributed* RAM pode utilizar uma grande quantidade de células lógicas, portanto, normalmente evita-se a utilização desse tipo de memória.

2.2 Gerenciamento de Memória em um Sistema Operacional

Sistemas operacionais modernos são capazes de gerenciar e abstrair toda hierarquia de memória presente nos computadores, e realizar a gestão dos programas presentes na memória principal, na memória *cache* e em memórias de massa, além da movimentação dos programas e dados entre os diferentes tipos de memórias existentes nos computadores atuais. A parte do sistema operacional responsável por essas funções é o gerenciador de memória, o qual utiliza-se de elementos de *hardware*, como registradores especiais e/ou unidades de gerenciamento de memória (*Memory Management Unit* – MMU) da unidade de processamento central (*Central Processing Unit* - CPU).

2.2.1 Memória Virtual

O conceito de **memória virtual** surgiu pela necessidade de execução de programas cujo tamanho era maior do que a quantidade máxima de memória principal presente nos computadores. A ideia principal desse conceito é de que cada programa tem o seu próprio espaço de endereçamento, o qual é dividido em segmentos contínuos de memória chamados de páginas (TANENBAUM, 2007).

Dessa forma, o endereço virtual referenciado por um programa não corresponde ao endereço real de memória no qual o programa está localizado. As páginas reais e virtuais são mapeadas por meio de uma tabela, chamada **tabela de páginas**. O gerenciador de memória é responsável pela manipulação dessa tabela e obtenção das páginas requisitadas mesmo que não existentes na memória principal.

O tamanho de um programa pode ser maior do que a quantidade de memória real existente, uma vez que cada programa tem seu próprio espaço de endereçamento. Dessa forma, é possível que as páginas de memórias requisitadas não estejam na memória principal. Nesses casos, a MMU da CPU gera uma interrupção (*trap*) no sistema operacional chamada de *page fault*. O gerenciador de memória seleciona uma página pouco utilizada da memória real, grava-a no disco, fazendo com que essa posição fique “limpa” e possa ser utilizada novamente. Em seguida, o gerenciador de memória obtém do disco a página requisitada, coloca-a na posição limpa de memória, atualiza a tabela de páginas e reinicia a instrução que causou a *trap* (TANENBAUM, 2007).

A tabela de páginas possui entradas que, além do mapeamento entre memória real e virtual, possuem informações a mais sobre a página em questão. A descrição exata dessas informações é dependente da implementação da máquina, entretanto, é comum que haja as seguintes informações: um *bit* de página presente/ausente, *bits* de proteção, um *bit* de página modificada e um *bit* de página referenciada.

A informação de página presente ou ausente indica se a página está localizada na memória ou não. Se a página não estiver na memória a *trap* de *page fault* é ativada. Os *bits* de proteção indicam se a página pode ser escrita e/ou executada, permitindo o controle de acesso às páginas. O *bit* de página modificada é utilizado quando há necessidade de limpar a página e é ativado sempre que ocorre uma escrita na página. Caso o *bit* indique que a página foi modificada é necessário escrever a página no disco, caso contrário pode-se simplesmente abandonar aquela página, uma vez que já existe uma cópia idêntica dela no disco. Por último, o *bit* de página referenciada é ativado sempre que ocorre uma leitura ou escrita na página. Isso serve para que o algoritmo de troca de páginas possa reconhecer quais páginas estão sendo menos utilizadas.

2.3 Direct Memory Access

O método *Direct Memory Access* (DMA) é amplamente utilizado em sistemas de computadores para acesso a memória principal, por parte de subsistemas, sem a necessidade de interferência do processador.

Portanto, a CPU é capaz de realizar outras operações enquanto a memória está sendo utilizada por outras unidades. A CPU apenas realiza a programação das regiões de memória, a qual os dados devem ser transferidos, e o tratamento dos dados após a conclusão da operação.

A utilização de DMA é recomendada quando existe a necessidade de taxas elevadas de transferências ou quando há uma grande quantidade de dados para a transferência de forma que se não empregado DMA, a CPU ficaria indisponível para outras operações por muitos ciclos de *clock*. Esse método é muito empregado em diversos dispositivos como controladores de disco e placas gráficas (HARVEY; DATA ACQUISITION DIVISION STAFF, 1991) por serem sistemas que constantemente necessitam de acesso à memória.

Recentemente o uso de DMA vem sendo empregado não apenas entre os dispositivos e a memória principal, mas também para facilitar o acesso à memória interna de placas gráficas. Um exemplo é o *DirectGMA* da *Advanced Micro Devices* (AMD), que consiste do suporte de *hardware* e *software* para transferências entre a *Graphics Processing Unit* (GPU) e outros subsistemas como representado pela Figura 9 (HINDRIKSEN, 2015).

Figura 9 - Exemplo de utilização de DMA (AMD DirectGMA)

Fonte: Hindriksen (2015).

2.3.1 Transferências Simples

Normalmente, em um sistema operacional, utiliza-se DMA simples para a transferência de dados. Esse tipo de DMA consiste na alocação de um *buffer* contínuo no espaço do núcleo (*kernel*) do sistema para a transferência dos dados do dispositivo. Uma vez completa a transferência, os dados são copiados para o *buffer* do usuário que não é necessariamente contínuo devido à memória virtual e sua divisão em páginas (BRYANT; O'HALLARON, 2010).

Esse método oferece a facilidade de configuração e operação da transferência do controlador DMA. Uma vez que os espaços de dados são contínuos e as informações necessárias consistem somente dos endereços iniciais, da fonte, do destino, e do tamanho a ser transferido (XILINX, INC., 2010). Porém para grande quantidade de dados, a cópia do espaço

do núcleo para o espaço do usuário pode ocasionar uma perda grande de tempo, diminuindo assim o *throughput* do sistema. Além disso, para transferências em que a quantidade de dados é maior do que o tamanho do *buffer* alocado, é necessária a quebra em transferências menores, o que também acarreta em diminuição do *throughput* do sistema, uma vez que o *buffer* não pode ser utilizado para outras transferências por DMA durante a cópia dos dados para o espaço do usuário.

2.3.2 Scatter & Gather

Para resolver o problema causado pelo *overhead* de múltiplas operações DMA, foi criado o método *Scatter & Gather* que consiste da automação do processo de configuração da DMA. Fazendo com que a execução sequencial de múltiplas operações possa ocorrer sem intervalos entre elas. Com isso, é possível utilizar o método de *zero-copy*, que consiste de transferências diretas da e para a memória do usuário.

Como a memória do usuário (memória virtual) pode não ser um bloco contínuo de memória real, não é possível a transferência dos dados somente com o endereço inicial e o tamanho do bloco. Assim, as informações passadas a unidade que irá realizar a operação de DMA, para esse método, correspondem a uma tabela, ou descritor, que indica pequenos blocos de memória contínuos, normalmente um número pequeno de páginas de memória por bloco.

Para evitar o acesso constante à unidade de DMA, o ideal é que a tabela seja montada e enviada de uma vez, porém nem sempre isso é possível. Nesses casos, envia-se a tabela por pedaços para o dispositivo. O método de como a tabela é montada e interpretada depende do *driver* desenvolvido para o dispositivo. Funções para auxílio na construção da tabela são disponibilizadas por cada sistema operacional.

2.4 Peripheral Component Interconnect Express

O *Peripheral Component Interconnect Express* (PCI Express® ou PCIe®) é um padrão de comunicação de alta velocidade (Tabela 1) adotado como duto de expansão para placas em computadores. Historicamente esse padrão é a evolução de padrões anteriores como o *Peripheral Component Interconnect* (PCI™) e o *Accelerated Graphics Port* (AGP).

Tabela 1 – Desempenho teórico do PCI Express

Versão	Codificação	Taxa de transferência	Largura de Banda por lane	Largura de Banda (link x4)
1.0	8b/10b	2,5 GT/s	2,000 Gb/s (250,0 MB/s)	8,000 Gb/s (1,000 GB/s)
2.0	8b/10b	5,0 GT/s	4,000 Gb/s (500,0 MB/s)	16,000 Gb/s (2,000 GB/s)
3.0	128b/130b	8,0 GT/s	7,877 Gb/s (984,6 MB/s)	31,508 Gb/s (3,938 GB/s)
4.0	128b/130b	16,0 GT/s	15,754 Gb/s (1969,2 MB/s)	63,015 Gb/s (7,877 GB/s)

Diversas melhorias foram feitas aos padrões anteriores ao desenvolver o PCIe tais como: aumento do *throughput*, diminuição do número de pinos de entrada e saída, melhoria na escalabilidade dos dispositivos, adição de sistemas mais eficientes na detecção de erros (BUDRUK, 2007) e adição nativa da funcionalidade *hot-plug*.

Entretanto o PCI Express preserva a compatibilidade de *software* com o PCI, dessa forma é possível a reutilização e configuração de novos dispositivos mesmo quando o *software* não possui suporte ao novo padrão (JACKSON; BUDRUK, 2012).

2.4.1 Topologia

A estrutura de operação do PCI Express é similar à estrutura de operação de redes de computadores (WILSON, 2005). A topologia empregada no PCIe é a ponto a ponto, mas com múltiplas conexões (*links*) seriais (Figura 10), possibilitando que diversos dispositivos sejam ligados simultaneamente a um mesmo módulo raiz (*root complex* ou *host*).

A topologia do antigo PCI é paralela e compartilhada, portanto, todos os dispositivos compartilham as mesmas linhas de dados e somente um dispositivo mestre pode atuar por vez. Entretanto com o PCI Express a comunicação é bidirecional simultânea (*full-duplex*) entre quaisquer duas extremidades (*endpoints*), sem limitações devido à existência de linhas de dados concorrentes, ou seja, os dispositivos podem transmitir e receber concorrentemente sem que ocorra conflito no duto.

A existência de linhas diretas ligando todos os dispositivos a um mesmo *root complex* limita o número de dispositivos no duto. Para resolver essa limitação existem roteadores de dados (*switches*) capazes de aumentar o número de dispositivos conectados a um mesmo duto e rotear os dados para os respectivos destinatários.

Figura 10 - Comparação entre os padrões PCI e PCI Express

Fonte: Adaptada de Wilson (2005).

Os computadores possuem tipicamente módulos *root complex* e *switches* presentes diretamente no circuito integrado (*chipset* da placa mãe) ligado diretamente à CPU chamado de ponte norte (DREPPER, 2007). A Figura 11 representa uma topologia comumente encontrada em computadores. Pela diferença física entre os padrões mais antigos, uma ponte de conversão física é necessária para adaptação entre padrões, caso esses sejam suportados.

Figura 11 - Exemplo de topologia PCI Express

Fonte: Adaptada de Budruk (2007).

2.4.2 Camadas do Protocolo

A estrutura do protocolo PCI Express é dividida em três camadas: a camada de transação (*transaction layer*), a camada de vinculação de dados (*data link layer*) e a camada física (*physical layer*). Todo o dado transportado por esse protocolo é encapsulado e desencapsulado por cada uma dessas camadas como representado pela Figura 12.

Figura 12 - Diagrama das camadas PCI Express em uma comunicação entre dois dispositivos

Fonte: Budruk (2007).

A comunicação é feita sempre entre dois pontos, o transmissor (TX) e o receptor (RX). O lado transmissor é responsável pela geração dos pacotes com os dados internos do dispositivo, que então são armazenados em um *buffer* até que a transmissão aconteça. Os dados a serem transmitidos passam pela camada de transação, em seguida pela de vinculação de dados e por último pela camada física na qual os dados serão enviados pelo *link*. O receptor, por sua vez, é responsável pela decodificação dos pacotes providos pelo *link*. Dessa maneira, os pacotes são submetidos ao procedimento oposto do transmissor, ou seja, passa pela camada física, depois pela camada de vinculação de dados e pela camada de transação, a qual os dados são extraídos para serem utilizados pela parte interna do dispositivo.

2.4.2.1 Camada Física

A camada física definida na especificação do padrão PCI Express (tradicionalmente chamada de PHY) é dividida em duas subcamadas: a elétrica e a lógica. A subcamada elétrica

compreende a especificação de componentes como um serializador / deserializador (SerDes) e outros circuitos analógicos cuja abordagem foge do escopo deste trabalho (TEMBHARE; PATIL, 2014). A subcamada lógica engloba as formas de divisão dos dados em canais, a forma de codificação e também informações para definição da comunicação.

Os dados que chegam para serem empacotados pela camada física são acrescidos de informações de início e fim da transmissão. Além disso, podem ser enviados os pacotes da camada física (*Physical Layer Packets* - PLP) ou também chamados de *Ordered-Set* (Figura 13), que são sempre um número múltiplo de 4 bytes (BUDRUK, 2007). Esses pacotes são decodificados a cada dois pontos, portanto, não são propagados por *switches*. Tais pacotes são utilizados para treinamento, alinhamento dos dados ou mudanças nos estados de energia do *link*.

Figura 13 - Diagrama da comunicação entre duas camadas físicas de dois dispositivos PCI Express

Fonte: Adaptada de Budruk (2007).

2.4.2.1.1 Lanes

Para cada conexão (*link*) PCI Express podem existir uma ou mais faixas (*lanes*) de comunicação. O número de faixas em um *link* é representado por xN onde N é o número de faixas. Cada faixa de comunicação apresenta duas linhas físicas de comunicação, uma para recepção e outra para transmissão de dados, em que cada linha é composta por um par de fios balanceados.

A taxa de transferência para cada faixa é de um *bit* por ciclo e o tempo do ciclo depende da versão utilizada (Tabela 1). Para aumentar a taxa de transferência, um conjunto de faixas paralelas pode ser utilizado, aumentando a taxa proporcionalmente com o número de faixas utilizadas, como representado pela Figura 14.

Figura 14 - Representação de *links* e *lanes* na comunicação PCI Express

Fonte: Wilson (2005).

A comunicação com múltiplas faixas é feita com envio de *bytes* sucessivos em faixas sucessivas. Embora tenha uma maior complexidade de sincronismo dos dados à latência é significativamente reduzida para pacotes com grande quantidade de dados.

Para solucionar problemas de balanceamento DC e de recuperação de *clock*, a transmissão por faixa utiliza codificação 8b/10b (BABU; BABU, 2013) nas versões 1 e 2 do padrão PCI Express e 128b/130b na versão 3. Com a codificação 8b/10b, para cada 8 *bits* de informações são transmitidos 10 *bits* causando um *overhead* de 20%, por esse motivo a nova codificação 128b/130b foi adotada reduzindo o *overhead* para 1,5%, enviando 130 *bits* para cada 128 *bits* de dados.

2.4.2.1.2 Conector

Como o número de faixas de transmissão em paralelo pode diferir de dispositivo para dispositivo, existem vários conectores diferentes. Além disso, a finalidade do dispositivo também influencia no tipo de conector, pois existem conectores internos (Figura 15), uma placa conectada diretamente a outra, e conectores externos, uma placa ligada por um cabo a outra.

Figura 15 - Conector PCI Express para conexão x1

Fonte: Nicolle (2012a).

Apesar das diferenças, todos os conectores internos do PCIe possuem 22 pinos e um dente de separação. Entre esses estão os pinos de alimentação, terra, *reset* e interfaces *System Management Bus* (SMBus) e *Joint Test Action Group* (JTAG). O número de pinos após a separação física depende do número de faixas suportadas. Cada faixa é composta de dois pares de sinais diferenciais (NICOLLE, 2012a), totalizando quatro pinos. Os sinais de recepção (PCI Express *Receive* - PER) são colocados de um lado da placa e os sinais de transmissão (PCI Express *Transmit* - PET) de outro. Os pinos de dados são colocados em ordem crescente das faixas e são intercalados com pinos de terra. O *clock* de referência (*reference clock* - REFCLK) também é transmitido pelo conector em dois pinos diferenciais localizados antes das primeiras linhas de dados como representado pela Figura 15.

Para conexões placa-placa (conectores internos), a lateral de uma das placas é utilizada como conector como representado pela Figura 16. Note também as linhas de dados diferenciais evidenciadas na figura. As curvas realizadas pelas trilhas de dados são feitas para garantir o mesmo comprimento entre todos os sinais de dados de forma a não haver uma diferença de atraso entre os sinais de transmissão, recepção e o *clock*.

Figura 16 - Conector PCI Express x1 em placa de circuito

Fonte: Nicolle (2012a).

2.4.2.2 Camada de Vinculação dos Dados

A camada de vinculação de dados (*Data Link Layer*) é a camada base na qual a integridade dos dados é avaliada e a requisição de nova transmissão é realizada caso necessário. Os pacotes gerados por essa camada são chamados de *Data Link Layer Packets* (DLLP) e são decodificados apenas por camadas de vinculação de dados como representado pela Figura 17.

Figura 17 - Diagrama da comunicação entre duas camadas de vinculação de dados de dois dispositivos PCI Express

Fonte: Budruk (2007).

Para a transmissão dos pacotes, os DLLP são empacotados pela camada física como representado pela Figura 18. Da mesma forma que os PLP, os DLLP são processados a cada

dois pontos e não são propagados por *switches*. Ou seja, os dados a serem transportados são separados das informações presentes nos DLLP e PLP a cada transmissão ponto-a-ponto.

Figura 18 - Diagrama de empacotamento da camada de vinculação de dados

Fonte: Budruk (2007).

As informações anexadas na camada de vinculação de dados possuem um tamanho fixo de 8 *bytes* e são utilizadas para:

- Confirmação de recebimento do pacote;
- Gestão de energia;
- Controle de fluxo.

Para a verificação da exatidão dos dados, no final do pacote são adicionados mais 16 *bits* de *Cyclic Redundancy Check* (CRC) comumente chamado de *Link CRC* (LCRC). Ao ser empacotado pela camada física os DLLP são acrescidos um *byte* no início e um *byte* no fim do pacote.

A confirmação ou não do recebimento, através das informações de ACK (*acknowledged*) e NAK (*not acknowledged*), depende do resultado entre o cálculo do CRC do pacote e o código presente no campo LCRC. No caso de ocorrer algum problema durante a transmissão o receptor envia o sinal de NAK para o transmissor repetir a operação (BUDRUK; ANDERSON; SHANLEY, 2008). Caso contrário o sinal de ACK é enviado e o transmissor esvazia o *buffer* de transmissão para que outro pacote possa ser enviado.

O gerenciamento dos pacotes que podem ser transmitidos é feito pelas informações de fluxo (FCx). Essas são utilizadas pelo transmissor para saber qual a capacidade atual de dados do receptor. Através de um sistema de créditos o transmissor pode reordenar os pacotes a serem enviados ou interromper momentaneamente a comunicação até que exista espaço suficiente para os pacotes no *buffer* do receptor.

2.4.2.3 Camada de Transação

A camada de mais alto nível do protocolo PCI Express é a camada de transação (*Transaction Layer*). É a camada responsável pela transmissão e recepção dos dados e da mesma forma como as outras camadas, os pacotes gerados são chamados de *Transaction Layer Packets* (TLP) e são decodificados apenas por camadas do mesmo tipo como representado na Figura 19.

Os TLP são encapsulados tanto pela camada de vinculação de dados como pela camada física. Dessa forma, para a transmissão de um TLP é montada toda a estrutura pelo transmissor passando por todas as camadas e desmontado no receptor. Como a camada de transação realiza o gerenciamento e endereçamento dos dados, suas informações necessariamente passam por *switches* e são utilizadas para rotear o pacote para o destino correto.

Figura 19 - Diagrama da comunicação entre duas camadas de transação de dois dispositivos PCI Express

Fonte: Budruk (2007).

Os TLP consistem do encapsulamento de todas as camadas do PCI Express como representado pela Figura 20. Juntamente com os dados transmitidos a camada de transação adiciona ao pacote um cabeçalho (*header*) e 32 bits de código CRC chamado de *End-to-end CRC* (ECRC) que é uma verificação opcional da integridade do pacote.

O cabeçalho do pacote existe para definir a função do pacote, a existência de dados, o destinatário e o remetente, o endereço e o alinhamento dos dados. Todas as informações, que o *software* possui para o correto roteamento do pacote são incluídas no cabeçalho.

Figura 20 - Diagrama do empacotamento da camada de transação

Fonte: Budruk (2007),

A quantidade de dados carregados por um TLP é variável, entre 0 e 4096 *bytes*. Essa quantidade é sempre múltipla de 32 *bits* ou um *Double Word* (DW). O tamanho do cabeçalho depende do tipo de endereçamento utilizado pelo pacote. Se o endereçamento for de 32 *bits* então o tamanho do cabeçalho é de 3 DW, se o endereçamento for de 64 *bits* então o tamanho é de 4 DW (BUDRUK, 2007).

2.4.2.3.1 Campos padrões

O cabeçalho é dividido em campos que dependem do tipo de pacote a ser enviado. Porém, alguns campos são padrões para todos os tipos de pacotes e a partir desses campos é que são avaliados os outros campos do cabeçalho.

Fmt

Um desses campos é o *Fmt* (*Format*). Composto por dois *bits*, esse campo representa o tamanho do cabeçalho, consequentemente o tipo de endereçamento e a presença ou não de dados a serem transportados pelo pacote. A lógica utilizada por esse campo está descrita na Tabela 2 (NICOLLE, 2012b).

Tabela 2 – Descrição do campo *Fmt* do cabeçalho de um TLP

<i>Fmt</i>	Formato do TLP
00	Cabeçalho de 3 DW, sem dado
01	Cabeçalho de 4 DW, sem dado
10	Cabeçalho de 3 DW, com dado
11	Cabeçalho de 4 DW, com dado

Type

O campo *Type* descreve o tipo do pacote. Dentre os diversos valores que o campo pode assumir os mais utilizados são 0x00 para operações de acesso a memória e 0x04 para operações de complementação. A definição exata do tipo de pacote utiliza os dois campos, *Fmt* e *Type*, obtendo assim a correta descrição da função do pacote.

TC

Traffic Class (TC) são três *bits* que correspondem ao canal virtual utilizado pelo pacote. Os canais virtuais servem para separar os dados em blocos com diferentes fluxos de controle. A maioria das aplicações não necessita da utilização de canais virtuais e, portanto, apenas o canal zero é utilizado fazendo todos os *bits* de TC zero.

TD

O campo TD (*TLP Digest*) é um *bit* que indica a presença ou não do ECRC no final dos dados. Como a camada de vinculação já possui um código de verificação, normalmente esse *bit* não é ativado.

EP

No caso do dado a ser colocado no pacote estar corrompido ou ser corrompido durante a passagem por outros dispositivos, o campo EP, que corresponde a um *bit*, é marcado e indica que o dado foi envenenado (*poisoned*). O envenenamento de dados é o termo utilizado para descrição da alteração de dados.

Attr

Para descrever os atributos adicionais do gerenciamento do pacote utiliza-se o campo *Attr* (*Attributes*). Esse campo é de dois *bits* que indicam o relaxamento da ordenação dos pacotes e a não obrigatoriedade de correlação de *cache* em *hardware*. Para a maioria das aplicações esse campo não é utilizado e seu valor deve corresponder à zero.

Length

O campo *length* corresponde à quantidade de dados transportada pelo pacote. Como a quantidade de dados é sempre múltipla de 32 *bits*, o número apresentado nesse campo

corresponde ao número de DW de dados. Porém, como são utilizados 10 *bits* para representar esse campo, uma exceção foi criada estipulando o valor zero (00 0000 0000b) como sendo correspondente a 1024 DW.

Reservados

Existem alguns campos no cabeçalho que são reservados para futuras aplicações, sendo preenchidos com zero. A denotação desse tipo de campo é dada pela letra R.

2.4.2.3.2 Pacote de escrita de memória

Os pacotes mais utilizados são os que realizam acesso à memória e os de complementação. O processo de escrita em memória consiste do envio de pacotes do tipo apropriado para o receptor. Nesse caso os campos do cabeçalho correspondem às informações pertinentes a operação de escrita. Um exemplo de pacote de escrita de memória é representado pela Figura 21, no qual se deseja escrever 32 *bits* de dado representado pelo valor 0x12345678 no endereço físico 0xFDAFF040 onde o receptor está mapeado (XILLYBUS LTD., 2015).

Figura 21 - Exemplo de TLP de requisição de escrita de memória com endereçamento de 32 bits

Fonte: Xillybus Ltd. (2015).

Os campos em cinza são campos não utilizados e os em verde correspondem aos campos raramente utilizados. Os números em vermelho correspondem aos valores de cada um dos campos destacados na Figura 21. Dessa forma, os valores transmitidos nesse pacote serão: 0x40000001, 0x0000000F, 0xFDAFF040, 0x12345678. Como o endereçamento do pacote é de 32 *bits*, o tamanho do cabeçalho é de 3 DW, ou seja, os valores de DW 0 a DW 2 correspondem ao cabeçalho do pacote e o valor de DW 3 aos dados.

Requester ID

A identificação do dispositivo que está requisitando o pedido de escrita é feita pelo campo *Requester ID*. Esse número corresponde ao identificador do dispositivo que é um número único atribuído pelo *root complex* durante a etapa de configuração do dispositivo. Os TLP, com exceção dos que realizam acesso à memória ou I/O, são roteados utilizando os

identificadores (*Requester ID* e *Completer ID*). Logo, esse identificador é utilizado no processo escrita de memória somente em casos de erro quando há a necessidade de avisar o dispositivo requerente sobre a situação.

Last BE

O protocolo PCI Express exige que o número de *bits* de qualquer pacote seja múltiplo de 32. Portanto, o campo *Last Byte Enable* (*Last BE*) é utilizado para descrever quais dos *bytes* presentes no último DW são válidos, cada *bit* do campo corresponde a um *byte* em sequencia. Com isso, torna-se possível a transferência de um número de *bits* múltiplo de 8.

First BE

Assim como o campo *Last BE*, o campo *First Byte Enable* (*First BE*) descreve a validade de cada um dos *bytes* do primeiro DW de dados. Isso possibilita que os dados não alinhados a 32 *bits* sejam transmitidos.

Address

O endereço no qual os dados devem começar a serem escritos é indicado pelo campo *Address*. O tamanho reservado para esse campo é de 30 *bits* para cabeçalhos de 3 DW e de 62 *bits* para cabeçalhos de 4 DW. O valor desse campo corresponde ao endereço dado em *Double Words* e não em *bytes*, como normalmente é utilizado. Os dois *bits* que completam 32 ou 64 *bits* de endereçamento são reservados e devem ser utilizados com zero, como representado na Figura 21.

O valor de endereço do exemplo (0x3F6BFC10) corresponde exatamente ao endereço desejado (0xFDAFF040) dividido por 4, que corresponde a conversão de endereçamento a *byte* para endereçamento a DW.

2.4.2.3.3 Pacote de leitura de memória

O processo de leitura de memória é um pouco mais complexo, pois consiste de duas operações. A primeira delas é a requisição de leitura, na qual são especificados todos os parâmetros da leitura como endereço e quantidade de dados. A segunda operação é a resposta à requisição, nessa o dispositivo requisitado envia os dados de volta ao requerente de forma a concluir o processo de leitura.

Como exemplo, supõe-se a leitura de 1 DW do endereço físico 0xFDAFF040 de um dispositivo. A Figura 22 ilustra o pacote enviado para a requisição da leitura de memória (XILLYBUS LTD., 2015).

Figura 22 - Exemplo de TLP de requisição de leitura de memória com endereçamento de 32 bits

Fonte: Xillybus Ltd. (2015).

Os campos apresentam significados similares aos campos do pacote de escrita, porém agora são utilizados para a leitura. Por exemplo, o campo de *Requester ID* é utilizado para o retorno dos dados, não somente para erros como ocorre na escrita. E o campo *Length* corresponde à quantidade de dados desejada. Além disso, o campo *Tag*, que não é utilizado na escrita, é necessário para a leitura.

Tag

O valor utilizado no campo *Tag* não tem significado próprio. Esse campo serve como uma etiqueta ao pacote. Essa etiqueta é utilizada na resposta da requisição de leitura. Assim, é possível enviar múltiplas requisições, com diferentes etiquetas, que as devidas respostas serão etiquetadas de acordo.

2.4.2.3.4 Pacote de complementação

Após o recebimento da requisição de leitura de memória, uma resposta deve ser gerada e enviada de volta ao dispositivo que fez a requisição. A resposta, portanto, é um pacote de complementação (*completion packet*). Continuando com o exemplo anterior, a resposta aquele pacote de leitura poderia ser um pacote como representado pela Figura 23.

Figura 23 - Exemplo de TLP de complementação

Fonte: Xillybus Ltd. (2015).

Nesse caso, os únicos campos que permanecem os mesmos são os campos padrões, como já abordado anteriormente. Os outros campos correspondem às informações específicas de retorno ao requerente e de remontagem dos dados.

Apesar do limite máximo dos TLP ser de 4096 *bytes* de dados por pacote, a maioria dos dispositivos possui um limite bem inferior a esse, comumente 128 *bytes*. Por esse motivo, uma requisição pode ocasionar múltiplos pacotes de complementação.

Byte Count

Cada pacote contém o tamanho de dados sendo transportados, porém no caso de múltiplos pacotes de complementação para uma única requisição, torna-se necessária a informação do número de *bytes* que restam serem transmitidos. Essa informação é representada pelo campo *Byte Count*. Com a mesma lógica utilizada pelo campo *Length*, o valor representado pelo campo *Byte Count* corresponde diretamente ao número de *bytes*, com exceção ao valor zero (0000 0000 0000b) que indica 4096 *bytes*.

As várias respostas de uma mesma requisição são enviadas sempre de maneira sequencial e ordenada. Portanto, o campo *Byte Count* é muito utilizado para verificação de que todos os pacotes de complementação foram recebidos. Para isso, basta-se verificar se o valor do campo *Byte Count* corresponde exatamente ao número de *bytes* de dados do pacote, indicando que não há *bytes* para serem enviados por outros pacotes.

Outra aplicação desse campo é na transmissão de um número não múltiplo de 4 *bytes* de dado, uma vez que o campo *Length* corresponde ao número de *Double Words*.

BCM

O campo *Bit Count Modified* (BCM) é um campo utilizado apenas para dispositivos PCI-X™, portanto, deve permanecer em zero.

Status

Para indicar problemas na complementação utiliza-se o campo *Status*. Situações em que não existem problemas o valor desse campo é retornado como zero. Caso contrário, o valor desse campo indica o tipo de problema como: requisição não suportada, ou impossibilidade de realizar a complementação, ou cancelamento da complementação.

Completer ID

A identificação do dispositivo que gera os pacotes de resposta faz parte do pacote de complementação. Essa informação segue os mesmos princípios do *Requester ID* presente no pacote de leitura.

Lower Address

O campo *Lower Address* representa os 7 bits menos significativos do endereço do primeiro *byte* válido do pacote de complementação. Isso é utilizado para o correto posicionamento dos dados em casos de múltiplos pacotes de resposta a uma única requisição. Além disso, também é utilizado para o correto alinhamento dos dados, uma vez que a unidade básica utilizada é o DW.

Esse campo complementa as informações do campo *Byte Count* possibilitando o correto ordenamento dos dados, nos casos que pacotes de diferentes requisições são retornados fora de ordem.

Tag

Assim como tratado anteriormente o campo *Tag* é utilizado para etiquetar as respostas de forma que elas possam ser montadas corretamente pelo dispositivo que pediu a leitura.

Requester ID

Para indicar o destino do pacote de complementação, o valor do campo *Requester ID* do pacote de requisição deve ser utilizado no pacote de complementação. Com isso, as mesmas atribuições existentes para o pacote de leitura cabem aqui.

2.4.3 Outros recursos

O padrão PCI Express define ainda outros tipos de pacotes, que diferentemente dos pacotes apresentados, não tem utilização tão frequente. Por exemplo, pacotes de I/O, que servem para acesso por duto de dispositivos ao invés do mapeamento em memória. Esses são descritos no padrão por motivos de compatibilidade e seu uso é desencorajado.

Para as interrupções, existem dois tipos básicos de pacotes: sinalização de interrupção (INTx) e Message Signaled Interrupt (MSI/MSI-X). Como MSI-X é uma versão expandida do MSI, esses são similares e normalmente tratados como um único tipo.

O suporte ao tipo INTx existe apenas por compatibilidade com dispositivos PCI. Esse tipo de interrupção não é recomendado por problemas de retenção da interrupção e compartilhamento (XILLYBUS LTD., 2015). Por esse motivo foi criado o MSI, e posteriormente o MSI-X, que permite interrupções por mensagem ao invés de mudança do estado de pinos como feito pelo antigo PCI. Isso facilita não só a identificação do dispositivo que causou a interrupção, mas também elimina a necessidade de gerenciamento do pino de interrupção.

2.4.4 Espaço de Configuração

Para a configuração do dispositivo no sistema foi especificado um conjunto de registradores, formando uma região de memória contínua, que vem desde os dispositivos PCI. Essa região é conhecida como espaço de configuração (*configuration space*).

Em dispositivos PCI o espaço de configuração era de 256 *bytes*, porém esse espaço foi expandido no PCI Express para 4096 *bytes*. Entretanto os registradores mais importantes desse espaço ficam alocados nos primeiros 64 *bytes* chamados de cabeçalho do espaço de configuração (*Configuration Space Header*). Para dispositivos finais (*endpoints*) PCI Express, o cabeçalho é do tipo 0, como representado pela Figura 24 (BUDRUK; ANDERSON; SHANLEY, 2008).

O espaço de configuração de um dispositivo PCI/PCIe serve para o mapeamento e identificação do dispositivo pelo sistema operacional ou *firmware* do sistema. Com isso, é possível que a configuração de dispositivos seja feita sem a utilização de *jumpers*, método *plug and play*.

Figura 24 - Diagrama do espaço de configuração PCI

Fonte: Adaptada de Budruk, Anderson, Shanley (2008).

2.4.4.1.1 Registradores

Como não convém a este trabalho a descrição de todos os registradores de configuração existentes, trata-se aqui apenas dos mais importantes a esta aplicação. Tais campos são: *Vendor ID*, *Device ID*, *Command*, *Status* e os *Base Address Registers*, os restantes encontram sua descrição em (BUDRUK; ANDERSON; SHANLEY, 2008).

Vendor ID e Device ID

Ambos registradores *Device ID* e *Vendor ID* consistem em números padronizados, de 16 *bits*, normalmente gravados no dispositivo pelo fabricante e auxiliam na identificação realizada pelo *software*. O *Vendor ID* é alocado pelo comitê PCI-SIG² para cada fabricante, já o *Device ID* é o próprio fabricante que estipula o valor.

Command

Para que o dispositivo possa acessar a memória, o *Bus Master Enable*, que é um *bit* do registrador *Command*, deve ser habilitado. A maioria dos outros *bits* do registrador não possuem funções aplicáveis ao PCI Express e devem ser fixados em nível lógico zero.

Status

O registrador *Status* é utilizado para identificar estados de erro e de interrupção quando INTx é utilizado. Além disso, por definição para o PCI Express, o registrador *Status* sempre indica a presença da lista expandida de capacidades, permitindo assim a utilização do espaço de configuração expandido.

Base Address Registers

No mapeamento do dispositivo na memória principal do sistema, os endereços disponibilizados para acesso ao dispositivo são armazenados nos *Base Address Registers* (BAR). Qualquer dispositivo PCI pode ter até 6 BAR de 32 *bits*, portanto podem ser mapeados até 6 regiões diferentes de endereços ou utilizar 2 BAR como um único endereço de 64 *bits*.

---

²PCI-SIG website: <https://pcisig.com>

3 DESENVOLVIMENTO

Esse trabalho iniciou-se a partir do *framework* PCI Express para FPGA previamente desenvolvido. O foco, portanto, foi implantar, aprimorar e adaptar o *framework* ao Espectrômetro Digital desenvolvido por pesquisadores do CIERMag.

Figura 25 - Diagrama geral do espectrômetro de ressonância magnética. Em laranja está enfatizado o módulo ao qual este trabalho se refere

Fonte: Adaptada de Pachchigar (2013)

O primeiro passo no desenvolvimento foi a mudança de *kit* de desenvolvimento, para o *kit* DE4 da Terasic, mostrado na Figura 26 (TERASIC INC., 2013). O *kit* possui uma FPGA Stratix® IV EP4SGX230KF40C2 da Altera®, um conector PCI Express x8 e quatro *Ethernet Gigabit*, além de uma grande quantidade de pinos de entrada e saída (*General-Purpose Input/Output* – GPIO). Dessa forma, é uma ótima plataforma para o desenvolvimento do DMRS com comunicação PCI Express.

Figura 26 - Kit de desenvolvimento Terasic DE4

Fonte: Terasic Inc, (2013).

O desenvolvimento do *hardware* na FPGA foi feito utilizando a linguagem VHSIC³ *Hardware Description Language* (VHDL), que é a mesma linguagem utilizada pelo espectrômetro digital do CIERMag e pelo *framework* desenvolvido. A síntese do código VHDL foi feita utilizando o *software* Altera Quartus® Prime v.16.1.0, que inclui as ferramentas de análise, síntese e programação específicas para a família Stratix.

Entretanto, para o *software* desenvolvido foram utilizadas diversas linguagens de programação. O código do *driver* para Linux e a biblioteca de suporte ao *driver* foram feitos em C. As aplicações de teste para Linux foram desenvolvidas em C e Python™; e para o Windows em Visual Basic® .NET e Python. As ferramentas utilizadas foram: o compilador GNU GCC v.6.3.1 para compilação no Linux e o ambiente Microsoft® Visual Studio® 2015 para o Windows.

A estrutura do trabalho foi desenvolvida em duas etapas sendo elas: a comunicação entre o espectrômetro e o computador por PCI Express, e a melhoria do *framework* para obter um aumento no desempenho e diminuir a latência de aquisição dos dados da FPGA.

3.1 Visão Geral

Ao realizar a comunicação entre o DMRS e um computador *desktop* por PCI Express, o foco foi incorporar o *framework* PCIe como um dos módulos de comunicação do espectrômetro. Dessa forma, a compatibilidade não seria quebrada e os métodos de comunicação utilizados anteriormente continuariam funcionais.

A Figura 27 representa o diagrama de conexão entre os módulos de *hardware* e *software* do sistema desenvolvido. A fim de destacar o desenvolvimento realizado, cada uma das cores presentes no diagrama possui um significado específico. Os blocos em cinza correspondem a blocos que não foram desenvolvidos, nem modificados nesse trabalho, apenas estão no diagrama para melhor entendimento do processo. Os blocos em azul correspondem aos blocos do *framework* PCI Express que já tinham sido desenvolvidos, mas sofreram modificações. Os blocos em amarelo também são blocos do *framework* PCIe que já haviam sido desenvolvidos, porém durante o desenvolvimento desse trabalho esses blocos foram completamente reescritos. Por último, os blocos em verde correspondem a blocos novos criados para adicionar funcionalidades que não existiam anteriormente.

---

³Very High Speed Integrated Circuit (VHSIC) foi um programa do governo dos E.U.A. na década de 80 para pesquisa e desenvolvimento de circuitos integrados de alta velocidade.

Figura 27 - Diagrama completo dos módulos de *hardware* e *software* desenvolvidos para a comunicação do computador com o espectrômetro digital de ressonância magnética

Fonte: Autoria própria.

Para facilitar o entendimento da estrutura, o desenvolvimento foi quebrado em duas frentes, o *software* e o *hardware* em que cada frente abrange seus respectivos módulos como representado na Figura 27. Nota-se que para o *software* também são feitas distinções entre os módulos que são alocados na memória no espaço do usuário (*user space*) e os módulos do núcleo (*kernel space*).

3.2 Hardware

Para realizar a comunicação entre o espectrômetro e o computador, foram feitas conexões entre as interfaces de comunicação existentes no *framework* PCI Express e no espectrômetro. O principal módulo do *hardware* do espectrômetro possui duas interfaces internas de comunicação, uma de comandos e outra de dados (Figura 28). Todos os tipos de comunicação que o espectrômetro suporta USB, *Ethernet* e, com esse trabalho, PCI Express são multiplexados para cada uma das interfaces de comunicação.

A interface de comando do espectrômetro é bidirecional de 8 *bits*, portanto, existem dois dutos de 8 *bits* em que um duto realiza a escrita dos comandos e o outro a leitura. Os comandos enviados para o espectrômetro são caracteres ASCII, no qual cada caractere do comando é lido ou escrito em um ciclo de *clock*.

Figura 28 - Diagrama dos módulos de *hardware* da interface entre o DMRS e o computador

Fonte: Autoria própria.

No caso da interface de dados, a situação é diferente. Os dados só precisam ser lidos do espectrômetro, portanto, a interface é somente de leitura apresentando uma largura de 32 *bits* e altas taxas de transferência. Os dados transmitidos por essa interface não possuem codificação, mas possuem um formato específico de organização que indica corretamente o receptor de origem e informações do sistema.

O *framework* PCIe, por sua vez, possui duas interfaces bidirecionais, uma de I/O de 32 *bits* de ponto a ponto, outra de DMA de 128 *bits*. Como a interface DMA é capaz de transferir múltiplos pacotes sequenciais em um único comando, obtêm-se melhores taxas de transferência e menor *overhead* para grandes volumes de dados.

3.2.1 Encapsulamento

Observando as características de cada uma das interfaces decidiu-se utilizar a interface de I/O para comunicação dos comandos do espectrômetro e a interface DMA para leitura dos dados. Como ambas as interfaces do *framework* são 4 vezes maiores do que as interfaces do espectrômetro, foi necessária a criação de um módulo chamado ToRM PCI (Figura 28), para as conversões de largura e encapsulamento do *framework* no *hardware*.

A conversão das larguras dos dutos de dados não foi o único fator que implicou na criação de um encapsulamento do *framework*. Registradores de controle dos dutos e de estruturas para cruzamento entre domínios de *clock* (*First-In First-Out* – FIFO) também ocasionaram a criação de um módulo para englobar toda a estrutura PCI Express. Dessa forma, o espectrômetro recebe e envia comandos e dados sem vinculação com o processo de transferência.

3.2.1.1 Multiplex

Devido aos diferentes tamanhos entre dutos de comunicação, foi criado um módulo capaz de multiplexar e demultiplexar os dados, para converter o número de *bits* de entrada no número de *bits* para a saída. Este módulo foi desenvolvido para ser genérico, com a relação entre a razão do número de *bits* de entrada e de saída, um número inteiro. Dessa forma, mesmo que no futuro o tamanho do duto do espectrômetro seja modificado, este módulo não precisará ser alterado.

Os processos de multiplexação e demultiplexação, para esse caso, consistem em armazenar e concatenar os dados recebidos por um número de ciclos de *clock* e disponibilizá-los uma vez que o tamanho armazenado correspondesse ao tamanho do duto de saída.

3.2.1.2 Domínios de clock

A principal frequência de operação dos módulos do espectrômetro é de 50 MHz. Assim essa é a frequência com que os dados são comutados pelos circuitos internos do espectrômetro. Entretanto, para a comunicação PCI Express o fluxo de dados utiliza a frequência de 125 MHz para operação em *link* x4 e 250 MHz para operação em *link* x8, valores que dependem exclusivamente do *hard* IP PCI Express da FPGA (ALTERA CORPORATION, 2014).

A melhor maneira para o cruzamento dos dados entre os domínios de *clock* é utilizando uma unidade FIFO *dual-clock*. Essa unidade é composta de memórias e contadores. A lógica interna estipula condições que garantem a transmissão do dado diminuindo a ocorrência de metaestabilidade no sistema, entre outras vantagens (LI et al., 2013).

Como são três dutos de comunicação, um de leitura, um de escrita de comandos e um de leitura de dados, foram necessárias três unidades FIFO, uma para passar de 125/250 MHz para 50 MHz e duas para passar de 50 MHz para 125/250 MHz.

3.2.1.3 Registradores

Para que o gerenciamento e controle das FIFO e dos multiplexes seja feito por um programa, foram criados dois registradores capazes de configurar e verificar a situação dessas unidades. A escrita e leitura desses registradores são feitas pelo módulo responsável pelo gerenciamento da comunicação PCI Express no *software* do espectrômetro.

3.2.1.3.1 Registrador de configuração

O registrador de configuração consiste de um *bit* responsável por informar que a comunicação PCI Express está disponível para o espectrômetro. Dessa forma, antes do *software* iniciar a comunicação com o espectrômetro, o registrador de configuração deve ter esse *bit* ativo. Uma vez ativo é garantido ao espectrômetro que o sistema possui o *driver* e as bibliotecas necessárias para a comunicação PCI Express.

3.2.1.3.2 Registrador de estado

Para a verificação do estado das FIFO foi adicionado um registrador de estado. Esse registrador é somente de leitura, portanto, o *software* não é capaz de alterar seu valor. Sua largura é de 32 *bits*, porém é dividido em dois registradores menores de 16 *bits*, como representado na Tabela 3.

Tabela 3 – Descrição da estrutura do registrador de estado

Bit	Nome	Descrição
31:16	Registrador de Estado de Comandos	Registrador que indica a situação da leitura e escrita de comandos.
15:0	Registrador Estado de Dados	Registrador que indica a situação da transferência de dados.

O registrador de estado de comandos contém informações sobre o estado das FIFO de comando, além do indicador de fim de comando. Os detalhes sobre esse registrador estão descritos na Tabela 4.

Tabela 4 – Descrição dos *bits* do registrador de estado de comandos

Bit	Nome	Descrição
15	Final de Comando	Um comando composto por diversos caracteres foi totalmente enviado.
14	FIFO Vazia	FIFO de Comando está vazia, portanto, todos os comandos já foram lidos.
13	FIFO Cheia	FIFO de Comando está cheia, portanto, o espectrômetro não pode mais mandar dados.
12:0	Nenhum	Sem função

Do mesmo modo, o registrador de estado dos dados, detalhado na Tabela 5, indica a quantidade de dados que ainda precisam ser transferidos por PCI Express e também o fim de pacote que notifica sobre a existência de um pacote inteiro que o espectrômetro enviou para a transferência.

Tabela 5 – Descrição dos *bits* do registrador de estado de dados

Bit	Nome	Descrição
15	Final de Pacote	Ativo no final da transmissão de um pacote.
14:6	Nenhum	Sem função
5:0	Dados Disponíveis	Número de dados, em blocos múltiplos de 16 <i>bytes</i> , disponíveis para serem lidos.

3.2.1.4 Endereçamento

Como o acesso ao espectrômetro se dá por duas interfaces de comunicação distintas, são utilizadas duas formas distintas para o endereçamento. Foram reservados no duto de

comando três endereços distintos, um para cada registrador e outro para o envio dos comandos até o espectrômetro. Para os dados, o endereço é único, ou seja, todos os dados são enviados serialmente para uma única FIFO. Como se pode alocar uma quantidade grande de endereços existe também a possibilidade de utilização de múltiplas FIFO.

3.2.2 Aprimoramentos

Durante a implementação do *framework* PCI Express, através de um estudo detalhado, percebeu-se que diversos aspectos poderiam ser melhorados resultando em menor consumo de lógica da FPGA, essa diminuição permite disponibilizar mais recursos para futuros aprimoramentos ou adições de funcionalidades ao espectrômetro. Além disso, a utilização de outra técnica de DMA, como *Scatter & Gather*, também poderia proporcionar benefícios diversos, como redução da latência de obtenção dos dados consequentemente maiores taxas de transferências.

3.2.2.1 Controlador DMA

Os módulos do controlador de DMA foram os mais modificados de todo o *hardware*. Os objetivos dessas mudanças foram: diminuir o número de ciclos de *clock* por transferência; aumentar a frequência de operação; e simplificar o código e a lógica. A fim de corrigir possíveis problemas descobertos, que o controlador anteriormente desenvolvido possuía, e prevenir exceções, diversos módulos foram modificados e em alguns casos reescritos. Também foram feitas algumas modificações na estrutura do código para que o sintetizador pudesse melhor compreender as estruturas e aperfeiçoar a lógica gerada.

Uma modificação importante foi a criação de um seletor por prioridades dos pacotes a serem enviados. A ordem de prioridades estipulada foi a seguinte (da maior para menor): pacotes de complementação, pacotes de requisição de leitura e de escrita. Com isso é possível responder a outros pacotes mesmo no meio de uma transmissão de dados.

Os registradores do controlador DMA foram separados das unidades de recepção e transmissão e reunidos formando um módulo independente. Do mesmo modo os contadores de tempo utilizados para a medição das transferências também foram alocados junto com os outros registradores do controlador uma vez que estes também ocupam espaço de endereçamento do controlador.

3.2.2.1.1 Compensação da latência de leitura

Para diminuir o número de ciclos de *clock* por transferência, foi necessária uma reestruturação no módulo de obtenção dos dados e no de interpretação do padrão Avalon-ST da Altera (atual Intel), utilizado no *framework* desenvolvido anteriormente. Para obter a melhor taxa de transferência possível todos os ciclos de *clock* devem ser utilizados para transferência do cabeçalho e dos dados do pacote. Dessa forma, a operação de obtenção do dado, avaliação da estrutura do pacote (início e fim) e segmentação do dado devem ser feitas em um único ciclo de *clock*.

A leitura dos dados tanto de uma FIFO quanto de memória possui uma latência característica, necessária para a estrutura funcionar, e ao segmentar os dados em diversos pacotes essa latência acaba se repetindo no início de cada pacote. Com isso, o controlador não realiza transferências durante alguns ciclos de *clock*.

A solução implantada foi armazenar os dados de forma que não houvesse necessidade de parar a aquisição, removendo o tempo de espera entre os pacotes. O armazenamento dos dados adquiridos foi realizado com uma FIFO na entrada de dados. Junto foram adicionados diversos controles como contadores de endereço e quantidade de dados para que os dados pudessem ser obtidos de forma independente da geração dos pacotes. A quantidade de dados armazenados na FIFO foi escolhida de forma que mesmo que a FIFO esteja cheia e a aquisição de dados do duto tenha que parar, a geração de pacotes pode prosseguir compensando a latência da aquisição dos dados.

O diagrama representado na Figura 29 demonstra a transmissão de dados com um pacote de 128 *bytes* de dados e mais 16 *bytes* de cabeçalho, com o duto de 128 *bits*. Em destaque estão os nove ciclos utilizados para a transferência desse pacote, delimitado pelos sinais de início (*sop*) e de fim (*eop*). Nota-se que a transferência, graças as FIFO, levam somente o mínimo número de ciclos de *clock* necessários.

Figura 29 - Diagrama dos sinais digitais de transmissão de dados com destaque nos ciclos por pacote

Fonte: Autoria própria.

3.2.2.1.2 Estado de espera

Na teoria, a transferência contínua, sem períodos de inatividade, proporciona taxas de transferência mais elevadas. Na prática, outros fatores também devem ser considerados. O fluxo contínuo de dados pode fazer com que o *root complex* entre em um estado de espera e pare de aceitar pacotes do transmissor por certo período de tempo. Isso pode acontecer por diversos motivos, pois esse módulo deve gerenciar outros dispositivos e comunicar-se com a memória principal.

Uma vez que o *root complex* para de receber os pacotes, toda a cadeia de transferência dos pacotes deve ser congelada. A retomada da geração de pacotes também possui uma latência principalmente entre pacotes de diferentes tipos. Eliminar essa latência pode acarretar em um consumo muito grande de lógica. Assim como feito anteriormente, a solução desenvolvida foi utilizar uma FIFO na saída dos pacotes para armazenar e compensar a latência de geração dos mesmos.

A Figura 30 representa os sinais de saída de dados para o *hard IP PCI Express* e nota-se, o pacote com os ciclos de espera destacado em azul e em verde o pacote sem ciclos de espera. Portanto, o método utilizado não causa mais atrasos na linha de transmissão. O início do pacote possui o sinal com terminação *sop* e o final do pacote é representado pelo sinal *eop*.

Figura 30 - Diagrama dos sinais digitais de transmissão de dados com destaque aos ciclos de espera

Fonte: Autoria própria.

Além disso, essa FIFO resolve outros problemas como, por exemplo, garantir a existência de pelo menos um pacote completo antes de começar a transferência. Dessa forma, as transferências de dados por DMA do *framework* foram isoladas por ambas as FIFO. Isso aumenta a latência do sistema, porém reduz drasticamente o tempo ocioso nas transmissões de grande volume de dados.

3.2.2.1.3 Aumento da frequência

O período de *clock* define a quantidade máxima de lógica combinacional que pode existir entre dois *flip-flops*, isso porque se o tempo de propagação for muito grande o sinal de

clock pode chegar antes em um *flip-flop* do que em outro e provocar assincronismo e metaestabilidade.

Para possibilitar o aumento da frequência de 125 MHz para 250 MHz e acomodar a lógica necessária sem causar problemas de temporizações foi aumentado o número de estágios de *flip-flops*. Além disso, as operações realizadas pelas máquinas de estado foram simplificadas, tanto de recepção quanto transmissão dos pacotes.

Simplificar a lógica sempre ajuda a reduzir os caminhos críticos, porém nem sempre é possível remover ou reduzir a lógica de um caminho. Portanto, em diversos casos as operações foram quebradas em duas ou mais partes com *flip-flops* entre elas. Isso também aumenta a latência do sistema, uma vez que os sinais precisam de um número maior de ciclos de *clock* para atravessarem a lógica. Entretanto, esse aumento da latência é compensado com o aumento da frequência e consequentemente da taxa de transferência de dados.

3.2.2.2 Gerenciamento de Tags

Para a correta identificação dos pacotes de complementação ao retornarem, cada *tag* é relacionada a um endereço e um tamanho de pacote. O endereço é o valor que indica a posição que o pacote deve ir e o tamanho é utilizado para descobrir se toda a requisição, representada pela *tag*, foi atendida.

O método utilizado anteriormente proporcionava uma tabela de endereços e tamanhos, com acesso de leitura e escrita em múltiplas linhas da tabela de forma simultânea. O resultado era uma latência pequena, mas com um consumo de lógica elevado, uma vez que essa tabela não utilizava as estruturas *Blockram* existentes, devido ao grande número de acessos simultâneos necessários.

Utilizando informações do cabeçalho do próprio pacote foi possível obter o tamanho total dos dados da requisição, removendo a necessidade da tabela de tamanhos. Com isso, o número de acessos à tabela de endereços também foi reduzido a apenas dois acessos simultâneos, que atende os requerimentos das estruturas *Blockram* e consequentemente reduz drasticamente o consumo de lógica.

Para o cálculo dos dados remanescentes de uma dada requisição foi utilizada a equação 1, em que *Length* é a quantidade de dados restantes, *LowerAddress* e *ByteCount* são campos do cabeçalho como discutido na seção 2.4.2.3.

$$Length = ((LowerAddress \wedge 0 \times 03) + ByteCount + 3) \gg 2 \quad (1)$$

Portanto, o módulo de gerenciamento de *tags* foi modificado para apenas um vetor de *bits*, em que a posição do *bit* corresponde a *tag* e o valor do *bit* indica se a *tag* está sendo utilizada no momento ou não, e uma memória contendo os endereços que o dado de cada *tag* deve ser escrito.

3.3 Software

A comunicação do *software* do espectrômetro, também chamado de Console, com o *hardware* é feita através de um programa chamado ACQServer (Figura 31). Toda a estrutura de comunicação é descrita no ACQServer que possui duas versões, uma em Visual Basic .NET e outra em Python.

A utilização do PCI Express requer um *driver* no núcleo do sistema operacional e uma biblioteca para simplificação das chamadas ao *driver*. Ambos foram adicionados ao ACQServer através de classes, uma vez que as duas linguagens utilizadas são orientadas a objeto. Essa classe, chamada PCI Express Core (Figura 31), é capaz de abstrair as funções específicas e funcionar da mesma forma para todos os tipos de comunicação suportados, sejam elas: USB, *Ethernet* ou PCI Express.

Figura 31 - Diagrama dos módulos de *software* da interface espectrômetro-computador

Fonte: Autoria própria.

3.3.1 Classes e Objetos

A classe criada para a comunicação entre o *driver* PCI Express e os outros módulos do ACQServer foi elaborada seguindo a mesma interface utilizada pelos outros módulos de comunicação do espectrômetro. Essa interface consiste dos seguintes métodos: abertura e fechamento da comunicação; leitura e escrita de comandos; leitura de dados e limpeza dos *buffers*.

A abertura da comunicação realiza a chamada ao *driver* passando o identificador do dispositivo para que a estrutura interna do *driver* seja preparada para a comunicação e habilite o *bit* do registrador de configuração para que o *hardware* do espectrômetro possa comunicar por PCI Express. Da mesma forma, o fechamento da comunicação realiza a limpeza das estruturas utilizadas para a inicialização, inclusive o registrador de configuração.

A escrita dos comandos para o espectrômetro é feita através de chamadas para o duto de I/O, os comandos são quebrados de 32 em 32 *bits* e enviados escrevendo no duto de I/O. Caso o comando não seja múltiplo de 32 *bits* são adicionados caracteres espaço, que é ignorado pelo espectrômetro, até que se complete o próximo múltiplo. Como o espectrômetro utiliza caracteres ASCII para os comandos é sempre possível obter um múltiplo de 32 *bits*.

Além disso, a cada transferência verifica-se o registrador de estado da FIFO de transmissão de comando. Caso cheia, a tarefa hiberna por 1 milissegundo, depois realiza a verificação novamente. Com isso, é possível a liberação da CPU para outras tarefas enquanto aguarda-se espaço na FIFO de comando para termino da transferência.

Para a leitura o processo é similar, primeiro espera-se o espectrômetro enviar um comando inteiro ou até que a FIFO de leitura fique cheia. Então se lê de 32 em 32 *bits*, removendo caracteres nulos e convertendo novamente para caracteres, até que a FIFO seja esvaziada. Tanto os comandos, quanto os estados das FIFO são lidos utilizando a interface de I/O do *firmware* PCI Express.

No caso dos dados, ambas as interfaces (I/O e DMA) são utilizadas. A leitura dos dados é feita pelo duto DMA com a opção de *streaming* habilitada, ou seja, o endereço não é automaticamente incrementado a cada leitura, mas funciona como um seletor de canais. Com isso possibilita a existência de múltiplos canais e a expansão do número de fontes de dados.

Antes da operação de leitura dos dados, verifica-se a existência de um sinal de fim de pacote, que pode ocorrer caso a FIFO de dados esteja cheia ou o espectrômetro tenha transferido todos os dados. Caso não ocorra o fim de pacote a tarefa hiberna por 1 milissegundo para depois realizar outra verificação. No caso de existir um fim de pacote os

dados existentes são lidos e armazenados até que a quantidade de dados requisitados seja atendida.

Outras funções como as de aborte e de limpeza dos *buffers* também fazem parte da estrutura da classe criada. Com isso, todo o espectrômetro pode ser manipulado com uma simples instânciação do objeto e chamada aos métodos. Facilitando a integração com o resto do programa existente.

3.3.2 Device Driver

Como a comunicação PCI Express depende do acesso direto ao barramento de dados do dispositivo, é fundamental a utilização de um módulo no *kernel* do sistema operacional. Módulos que cuidam do tratamento e utilização de dispositivos são chamados de *device driver*. Com uma busca no código fonte do sistema operacional Linux por métodos e estruturas mais eficientes para transferência de dados, modificou-se o *device driver* do *framework* PCI Express com o objetivo de remover código desnecessário, uma vez que esse código é executado no espaço do *kernel* (núcleo do sistema operacional). Além disso, as alterações também tiveram como foco seguir as convenções estabelecidas para desenvolvimento de *device drivers* para dispositivos PCI no Linux (MARES; GRUNDLER, 2006).

A primeira das alterações realizadas foi a adição de diretivas para impressão de mensagens de erro e de *debug*. Com isso, as ações e erros são registrados pelo sistema e podem ser observados utilizando a ferramenta *dmesg* que imprime as mensagens do núcleo do sistema. Essas diretivas permitem maior controle sobre a verbosidade do sistema e uniformidade na identificação do *driver*.

3.3.2.1 Inicialização

Na inicialização do *driver* deixou-se de adicionar os identificadores de dispositivos suportados (*Device ID* e *Vendor ID*) na tabela de registro e passou-se a utilizar uma tabela de dispositivos estática, na qual se consome menor número de instruções e não apresenta desvantagem em termos de flexibilidade.

As funções do *device driver* que são responsáveis pela manipulação do PCI Express, tais como *probe* e *remove*, são associadas ao sistema operacional através da criação de uma estrutura estática (`struct pci_driver`), na qual, os ponteiros para cada uma dessas funções

são preenchidos com os respectivos endereços, juntamente com as informações de nome do módulo e os identificadores do dispositivo. Para o registro de todas essas informações junto ao *kernel*, deve-se utilizar a função de registro do recurso, no caso `pci_register_driver`, passando a estrutura `struct pci_driver`.

3.3.2.2 Sondagem

Uma vez o *driver* registrado, o sistema realiza a sondagem dos dispositivos existentes e durante esse momento todas as estruturas necessárias para a comunicação devem ser alocadas e inicializadas. O sistema utiliza o ponteiro `probe` da estrutura registrada (`struct pci_driver`) para chamar a função de sondagem do dispositivo.

Na sondagem, o dispositivo é habilitado e sua região de memória é reservada para que não ocorra conflito entre os *device drivers* existentes no sistema. Para realizar o acesso ao dispositivo, os endereços dos BAR são mapeados, de endereços reais para virtuais, possibilitando o acesso pelas funções de leitura e escrita do sistema. Para a utilização de DMA é necessário informar ao *kernel* o número de *bits* (máscara de *bits*) de endereçamento que o dispositivo suporta. Como o *hardware* desenvolvido pode operar tanto em 64, quanto em 32 *bits*, a primeira opção é ativação da máscara de 64 *bits* para DMA e apenas no caso do sistema operacional não suportar essa máscara é que se faz a ativação da máscara de 32 *bits*. As interrupções do sistema também são alocadas e mapeadas durante a sondagem do dispositivo. Isso permite que o dispositivo possa, por meio de MSI ou MSI-X, executar as funções de interrupção do *device driver*.

A estrutura utilizada internamente pelo *driver* para representação da placa, nesse caso `struct torm_dev`, também é alocada e inicializada em memória. Nessa estrutura são associados os recursos e os ponteiros necessários para a operação de todas as funcionalidades presentes no *hardware*. O espectrômetro sempre terá uma comunicação direta com o computador (duto de entrada/saída) e um número inteiro de canais de DMA para a transferência dos dados dos receptores.

Após toda a inicialização básica do *hardware*, aloca-se uma estrutura para a comunicação com o duto de entrada e saída (`struct torm_io`) e *N* estruturas (`struct torm_dma`), em que *N* é o número de canais obtido a partir da leitura de um registrador existente na placa. Aloca-se uma região para um dispositivo do tipo caractere, que é o tipo utilizado para dispositivos como este, para cada canal DMA e uma para o duto de

entrada/saída. Essas regiões são inicializadas e vinculadas com a estrutura de funções do *driver* (struct `file_operations`) para que o duto de entrada/saída e cada canal DMA apareça no sistema operacional como um dispositivo diferente, mesmo pertencendo ao mesmo *hardware*.

Por último, adiciona-se os dispositivos de caractere ao núcleo do sistema operacional. Portanto, é nesse momento que o *driver* realmente entra em operação e todas as outras funções de operação do dispositivo estão disponíveis (CORBET; RUBINI; KROAH-HARTMAN, 2009). Com isso, cada placa cria 1 dispositivo `torm_io` e N dispositivos `torm_dma`, em *userspace*, no diretório `/dev`. Cada dispositivo acompanha um número inteiro (e.g. `torm_io0`, `torm_dma0` ou `torm_dma1`), utilizado para diferenciação dentre os dispositivos. Assim também, no caso de múltiplas placas contendo o *hardware* desenvolvido, o *driver* mantém uma lista com as estruturas `torm_dev`, de forma que, o número de cada dispositivo de caractere criado é sequencial. Por exemplo, uma placa com 4 canais de DMA fará com que sejam criados os dispositivos: `/dev/torm_io0`, `/dev/torm_dma0`, `/dev/torm_dma1`, `/dev/torm_dma2`, `/dev/torm_dma3`. Ao reconhecer mais uma placa existente com 2 canais de DMA, o sistema ainda criará os dispositivos: `/dev/torm_io1`, `/dev/torm_dma4`, `/dev/torm_dma5`. Portanto, independente do número de placas contendo o *hardware* desenvolvido, bem como o número de canais por placa, o *driver* reconhece e ajusta a criação de dispositivos.

3.3.2.3 Abertura e Fechamento dos recursos

Para a utilização dos recursos do *hardware* por programas, no espaço do usuário, realiza-se a chamada da função de abertura de arquivo do sistema operacional. O usuário deverá fazer a abertura do dispositivo correspondente ao recurso que se deseja utilizar. Por exemplo, para a utilização do primeiro canal de recepção, que utiliza DMA, o programa em *userspace* deverá fazer a abertura do dispositivo `/dev/torm_dma0`.

A abertura do dispositivo pelo sistema operacional, faz com que a função `open` da estrutura `file_operations`, que foi vinculada anteriormente ao dispositivo de caractere, seja executada. Nessa função faz-se a vinculação da estrutura do dispositivo em questão (struct `torm_io` ou struct `torm_dma`), com a estrutura de arquivo (struct `file`) através do ponteiro `private_data`. Para os dispositivos de DMA, aloca-se um *buffer* em um espaço de memória capaz de realização de DMA.

Com o fim da utilização do dispositivo, realiza-se a operação de fechamento através da rotina `close` do sistema operacional. Essa rotina executa a função `release` da estrutura `file_operations`. Para os dispositivos `torm_io` não é feita nenhuma alocação durante a abertura do dispositivo, portanto, não há a necessidade de realizar nenhum tipo de liberação e a função apenas retorna zero. Entretanto, para os dispositivos `torm_dma`, a liberação do dispositivo realiza a liberação do *buffer* alocado na abertura, assim como garante que nenhum recurso do dispositivo, como *timers* e *tasklets*, encontram-se em um estado de execução. Isso garante que uma vez realizado o fechamento do dispositivo não há mais nada pendente.

3.3.2.4 Escrita no duto de Entrada/Saída

Ao realizar a chamada de escrita ao dispositivo de entrada e saída, a função `write` associada a estrutura do dispositivo é executada. A primeira coisa a se realizar para a escrita de dados no duto, é a alocação de um *buffer* no espaço do núcleo do sistema, utilizando a função `kmalloc`. Em seguida, copiam-se os dados do espaço do usuário para o espaço do núcleo (`copy_from_user`). Esses espaços representam regiões distintas de memória virtual e possuem atribuições específicas dependendo de seu propósito. O espaço do núcleo corresponde à região com acesso privilegiado em que são executados o *kernel* e os *device drivers*. O espaço do usuário corresponde à região não privilegiada das aplicações que não o núcleo do sistema operacional.

Com isso, todas as informações necessárias são obtidas e a escrita no dispositivo pode acontecer. Utilizando a função `iowrite32`, os dados são escritos em passos de 32 *bits*. Caso a transferência não seja múltipla de 32 *bits* utiliza-se as funções `iowrite16`, para escrever de 16 em 16 *bits*, ou a função `iowrite8` para escrever *byte* a *byte*. Uma vez que a escrita é realizada o *buffer* temporário é descartado (`kfree`) e o número de *bytes* escritos com sucesso é retornado pela função.

3.3.2.5 Leitura do duto de Entrada/Saída

A leitura do dispositivo segue os mesmos princípios da escrita, porém de maneira reversa. Entretanto, o *buffer* temporário, no espaço do núcleo, continua sendo alocado logo no início, pois caso não seja possível a alocação desse *buffer* a função deve retornar falha e não realizar a leitura.

Em seguida, lê-se o dispositivo em passos de 32, 16 ou 8 *bits* utilizando as funções `ioread32` e `ioread8`, respectivamente. Passam-se, então, os dados lidos do espaço do núcleo para o espaço do usuário (`copy_to_user`) para que a aplicação possa utilizá-los. Novamente, antes do retorno da função desabilita-se o *buffer* temporário com a função `kfree`.

3.3.2.6 Mudança de endereço do duto de Entrada/Saída

Tanto a escrita quanto leitura do duto de entrada e saída acontecem de forma sequencial a partir do primeiro endereço do BAR correspondente ao duto. Para leitura ou escrita em determinado endereço, deve-se mudar primeiramente o ponteiro de acesso para em seguida realizar as operações.

A mudança do ponteiro de acesso é realizada pela função `llseek`. Essa atribuição pode ser realizada de duas formas: pela atribuição direta do novo endereço ou pela soma de um *offset* a posição atual.

3.3.2.7 Inicialização do DMA

Durante a sondagem do duto PCI Express, também são realizadas as inicializações de cada uma das estruturas de canal de DMA (`struct torm_dma`). Essa estrutura contém os endereços de base para a leitura e escrita dos registradores, *timers*⁴ e *tasklets*⁵ utilizados para a execução das operações de DMA, bem como as informações do estado do *hardware*, filas de espera e contadores para auxiliar a correta transferência de dados em ambos os modos de operação do canal de DMA.

3.3.2.8 Comandos dos canais de DMA

Para alteração e/ou leitura de parâmetros dos canais de DMA, devem ser realizadas requisições através da função `ioctl`⁶. Essa função é capaz de aceitar múltiplos comandos diferentes, que consistem de números aos quais são mapeadas rotinas diferentes. Dessa forma,

---

⁴Estrutura de temporização existente no *kernel* (CORBET; RUBINI; KROAH-HARTMAN, 2009, p. 198)

⁵Estrutura de deferimento de trabalho existente no *kernel* (CORBET; RUBINI; KROAH-HARTMAN, 2009, p. 202)

⁶Chamada do *kernel* (*syscall*) utilizado para ação em dispositivos (CORBET; RUBINI; KROAH-HARTMAN, 2009, p. 140)

a função `ioctl` dos canais de DMA realiza as operações de verificação do comando e execução da sub-rotina desejada.

Juntamente com os comandos, um ponteiro para outros argumentos pode ser passado possibilitando a execução de sub-rotinas mais complexas. Os diversos argumentos passados para a função `ioctl` dependem exclusivamente do comando a ser utilizado. Para comandos que necessitam que os argumentos sejam escritos ou lidos do *hardware*, as funções `put_user` e `get_user` são utilizadas para passar tais argumentos do espaço do usuário para o espaço do *kernel*.

Três comandos foram estipulados através da função `ioctl`. O primeiro deles consiste da habilitação ou desabilitação de interrupções pelo canal, ou seja, com esse comando é possível fazer com que o canal utilize ou não interrupções para notificação da conclusão das operações. Os outros dois comandos são referentes ao controlador DMA. Um para ler o registrador, do *hardware*, que conta o tempo gasto durante a operação de DMA e o outro para reinicialização desse registrador.

3.3.2.9 Transferências por DMA

Assim como o duto de entrada/saída, os dispositivos correspondentes aos canais de DMA também possuem as funções, `read`, `write` e `llseek`, sendo que a última possui a finalidade de alterar o endereço do *hardware* utilizado para as outras duas funções. Como as operações de leitura e escrita seguem a mesma abordagem, diferindo apenas o sentido de movimentação dos dados, ambas serão discutidas como um único caso.

As transferências por DMA, para esse projeto, foram concebidas de duas formas diferentes. A primeira como uma operação mais simples, na qual o *hardware* é responsável por apenas uma transferência entre a memória interna da placa e uma região de memória do *kernel* capaz de ser acessada por dispositivos externos. Isso, porém, possui suas limitações, como discutido em 2.3.1. A outra, como uma operação mais complexa, que utiliza técnicas de acesso direto ao usuário, diminuindo a cópia de dados pela memória, que, para esse projeto, foi nomeada de *Scatter & Gather*. Em ambos os casos, o objetivo é informar o *hardware*, das informações necessárias para que a transferência de dados possa ocorrer. Tais informações são: endereço dos dados no *hardware*, endereço dos dados na memória do computador, a quantidade de dados e como a transferência ocorrerá, ou seja, o sentido da transferência,

utilização ou não de interrupções e se a transferência será realizada utilizando o modo simples ou *Scatter & Gather*.

3.3.2.9.1 Operação Simples

Para as transferências de dados que utilizam o método simples, existe a dependência de um *buffer*, chamado de *bounce buffer*, nos quais os dados são copiados antes de ir para o destino final (2.3.1). Dessa forma, caso a operação em execução corresponda a transferência de uma quantidade de dados maior do que o tamanho do *buffer* alocado, a operação é quebrada em transferências menores. Para a escrita de dados no *hardware*, inicialmente, executa-se o travamento do *bounce buffer*, para evitar condições de corrida⁷, e a cópia dos dados do espaço do usuário para o *buffer*, que foi alocado durante a inicialização do canal. Em seguida, ocorre a transferência dos endereços de fonte e destino, do número de dados e de *bits* de controle para os registradores da DMA utilizando a função `iowrite32`. Os *bits* de controle indicam a direção da transferência e a utilização ou não de interrupções.

Assim que os registradores DMA são carregados, o *driver* entra em modo de espera até a conclusão da operação. Essa espera pode ser feita em ciclos de leitura do registrador de estado até que esse indique o término da operação. Esse método é comumente referido como *polling* do registrador de estado. Porém, há outro método que não consome tempo do processador enquanto espera pela conclusão da transferência, que é colocar o processo para “dormir”, ou seja, o processador é comutado para execução de outro processo até que uma interrupção seja causada pelo dispositivo. As duas opções estão disponíveis para o *driver* desenvolvido e a escolha do modo é feita pelo usuário ativando ou desativando o modo de interrupções.

Uma vez concluída a transferência, realiza-se o destravamento do *buffer* para que este possa ser utilizado em outra transferência. No caso das operações de leitura, a cópia dos dados é realizada antes do destravamento do *buffer* e com o sentido contrário (do *buffer* para o espaço do usuário).

Por fim, os ponteiros são incrementados e o processo inicia-se novamente caso a operação tenha sido quebrada em múltiplas transferências menores.

3.3.2.9.2 Operação Scatter & Gather

A cópia os dados do *buffer* do usuário para o *bounce buffer* do *kernel*, no qual foi feito o mapeamento para o duto PCI Express, é um processo demorado para grandes quantidades

---

⁷Condições de corrida acontecem quando duas ou mais tarefas tentam, simultaneamente, alterar dados compartilhados. O escalonador de tarefas do sistema operacional pode comutar entre as tarefas a qualquer momento, logo, não é possível prever o resultado do acesso aos dados.

de dados. Essa cópia pode ser totalmente eliminada através do acesso direto aos dados do *buffer* do usuário. Mas, devido ao fato de que os dados em espaço de usuário podem estar fragmentados na memória real, utiliza-se o método *Scatter & Gather* que consiste em passar uma lista de páginas de memória para o dispositivo ao invés de um endereço único.

O acesso aos dados em espaço do usuário é feito obtendo as páginas de memória do *buffer* do usuário com a função `get_user_pages`. Com isso, obtêm-se um vetor de estrutura de páginas de memória (`struct page`) que contêm os endereços reais das posições de memória nas quais os dados estão armazenados.

A partir do vetor de páginas, monta-se uma lista de estruturas (`struct scatterlist`) as quais contêm informações da página (`struct page`), como o tamanho e o deslocamento da página (*offset*). Com a lista de estruturas completa, faz-se o mapeamento das páginas para transferência por DMA utilizando a função `dma_map_sg`. Esse mapeamento faz utilização da unidade de gerenciamento de memória de entrada e saída (*Input/Output Memory Management Unit* – IOMMU) para a correlação entre as posições de memória reais e os endereços do duto PCI Express para a operação de DMA. Esse mapeamento é capaz de agrupar páginas consecutivas de memória em um único segmento a ser acessado. Dessa forma, após o mapeamento, o número de segmentos de memória não necessariamente corresponde ao número de páginas de memória que serão acessadas.

Os endereços e tamanhos dos segmentos de dados mapeados são obtidos com as macros `sg_dma_address` e `sg_dma_len`, os quais são carregados na tabela de registradores do *hardware*, um a um, utilizando a função `iowrite32`. Essa tabela de registradores possui um número fixo de elementos que podem ser armazenados. Para os casos em que a quantidade de segmentos de dados mapeados excede o número máximo de registros que o *hardware* é capaz de armazenar, o carregamento da tabela de registros é feito por partes.

Após o preenchimento dos registros, o *driver*, entra em estado de espera, que como discutido anteriormente pode ser executada de duas formas diferentes, realizando *polling* do registrador de estado ou por interrupção.

Ao concluir o conjunto de transferências, o *hardware* causa uma interrupção no processador, ou ativa o *bit* de “pronto” no registrador de estado, fazendo com que a rotina de transferência volte a ser executada. Caso, anteriormente, não tenha sido possível carregar todos os segmentos de dados no *hardware*, a rotina volta a carregar mais uma quantidade de segmentos e novamente aguarda até a conclusão das transferências. Entretanto, se todos os

segmentos foram carregados a rotina finaliza todo o processo de transferência e retorna a CPU para o processo do usuário.

4 RESULTADOS

O foco principal desse trabalho é sua utilização no Espectrômetro Digital (DMRS) do CIERMag, integrando uma nova interface de comunicação e transferência de dados em alta velocidade. Isso foi conseguido com sucesso como representado pela Figura 32. Com o novo método de comunicação reconhecido e os comandos sendo transmitidos por ela, foi possível a obtenção de dados de um experimento de Ressonância Magnética, executado no espectrômetro na FPGA. Nota-se que na Figura 32 além da interface encontrada, o texto “*ToRMCORE NMR build 1.2F (C) 2009-2016 by CIERMag.*” é enviado pelo espectrômetro para indicar a versão do *hardware* do sistema.

Figura 32 - Programa ACQServer utilizando a interface PCI Express

Fonte: Autoria própria.

A aquisição de dados realizada foi feita utilizando apenas um receptor do espectrômetro, pois ainda estão em desenvolvimento, no CIERMag, sequências de ressonância magnética multicanal e o interpretador de dados para múltiplos canais. A utilização multicanal do espectrômetro envolve, não somente este trabalho, mas outros trabalhos ainda em desenvolvimento no centro, como a criação de bobinas, pré-amplificadores e rotinas de tratamento dos dados coletados.

Apesar de ainda não terem sido feitos experimentos multicanal, a comunicação PCI Express já está preparada e não requer nenhuma alteração para utilização de multicanais. As

mudanças realizadas por esse trabalho ao *framework* PCIe garantem velocidade suficiente para a transmissão dos dados de múltiplos receptores. Para verificar a velocidade da comunicação, bem como o funcionamento de múltiplos canais, foram desenvolvidas aplicações de teste.

Como a proposta do espectrômetro é funcionar em sistemas Linux e Windows, os testes de velocidade foram executados em um computador *desktop* com o sistema operacional Arch Linux™ executando o *kernel* versão 4.10.13. Os testes feitos com o *software* do espectrômetro foram feitos tanto com a distribuição Linux, descrita anteriormente, quanto com o Windows 10™.

4.1 Hardware

Para testar o *hardware* foram utilizados contadores sintetizados na FPGA para obter o número de ciclos de *clock* que as operações de DMA gastam entre a entrada do comando de execução e a conclusão das transferências. Com esses dados pode-se avaliar o desempenho do *hardware* do sistema de forma isolada e comparar com os valores obtidos pelo *framework* PCI Express antes das melhorias feitas.

4.1.1 Desempenho

Os dados representados na Tabela 6 foram obtidos utilizando a comunicação PCI Express x4 e as transferências se deram por DMA simples com um *buffer* de 32 KBytes alocado pelo *driver*. O tamanho máximo de dados em cada pacote PCIe foi limitado em 128 Bytes e imposto pelo *root complex* (veja 2.4.1) da placa mãe utilizada.

Para evitar confusão de interpretação, as direções das transmissões de dados, referidas nesse trabalho, tem como referência o computador, ou seja, a operação de leitura corresponde ao envio dos dados da FPGA para a memória principal e a escrita corresponde ao envio no sentido oposto, da memória principal para a FPGA.

Utilizando os dados da Tabela 6, foram criadas as curvas da Figura 33, as quais permitem uma avaliação e comparação das velocidades de leitura e escrita.

Nota-se que para transferência de pequenos blocos de dados, o cabeçalho do pacote e os ciclos de *clock* gastos para inicialização de todo o processo no controlador DMA causam

uma diminuição significativa de desempenho. Entretanto, no caso de grandes blocos, razão de se utilizar DMA, esse *overhead* tem uma menor influência.

Tabela 6 – Medidas de tempo de resposta e *throughput* das transferências no *hardware* com 4 *lanes*

Tamanho do bloco de dados [B]	Tempo gasto para leitura [μ s]	Tempo gasto para escrita [μ s]	<i>Throughput</i> de leitura [Mbits/s]	<i>Throughput</i> de escrita [Mbits/s]
262.144	158,68	161,08	13.220,3	13.019,0
131.072	79,02	81,43	13.269,6	12.905,8
65.536	39,81	40,44	13.182,5	12.970,5
32.768	19,72	20,12	13.291,8	13.028,2
16.384	10,11	10,37	13.017,1	12.643,6
8.192	5,08	5,75	12.902,5	11.669,4
4.096	2,64	3,05	12.409,3	10.743,0
2.048	1,42	2,00	11.524,3	8.218,3
1.024	0,81	1,40	10.103,5	5.918,4
512	0,50	1,13	8.127,0	3.713,2
256	0,35	0,79	5.803,3	2.609,5
128	0,28	0,69	3.645,6	1.485,8

Figura 33 - Curvas do *throughput* de dados com as modificações no *framework* PCI Express

Fonte: Autoria própria.

Vale também salientar que a velocidade de escrita é menor que a de leitura, pois também são considerados os tempos que o *root complex* leva para responder a requisição de dados e enviar os pacotes de complementação. Justamente por isso é que nas transferências de pequenos blocos de dados há uma grande redução na velocidade, uma vez que o tempo de requisição ao *root complex* impacta mais no desempenho da transmissão.

Com a limitação de apenas 128 bytes de dados por pacote, a velocidade máxima teórica, para as condições testadas, seria de 13.760 Mbits/s. Esse é 86% do máximo possível

no duto (LAWLEY, 2014). Essa perda deve-se ao *overhead* causado pelo cabeçalho do pacote e o encapsulamento das camadas física e de vinculação de dados.

Os resultados descritos na Tabela 6 são aproximadamente 97% do valor máximo teórico de 13.760 *Mbits/s*, o que é condizente com o esperado, uma vez que na situação real o *root complex* pode ficar ocupado e outros pacotes como DLLP e PLP também trafegam pelo duto.

Comparando o *throughput* de escrita do sistema atual e antes das modificações feitas (Figura 34), pode-se verificar um ganho na velocidade apesar de não ser elevado.

Figura 34 - Curvas de comparação entre o *throughput* escrita do *hardware* antes e depois desse trabalho

Fonte: Autoria própria.

Entretanto, ao compararem-se os resultados do *throughput* de leitura, nota-se um aumento significativo da velocidade da mesma (Figura 35).

A compensação dos estados de espera foi responsável pelo aumento do *throughput* de escrita, e por um aumento parcial no caso da leitura. O grande aumento do *throughput* de leitura deve-se a compensação da latência de leitura. Com isso foi possível remover de 2 a 3 ciclos de *clock* por pacote e, portanto, para uma quantidade grande de pacotes esses ciclos fazem a diferença no desempenho.

As diferenças percentuais dos resultados, entre o original e as alterações feitas, são representadas na Figura 36, na qual se pode notar uma melhoria de até 21% na escrita e de 66% na leitura. Em ambos os casos os resultados garantem a velocidade necessária para a aplicação desejada e serão explorados durante a utilização multicanal do espectrômetro. Com

os valores obtidos é possível a expansão do espectrômetro para um número maior de canais de aquisição de radiofrequência e/ou realização do aumento da largura de banda de aquisição.

Figura 35 - Curvas de comparação entre o *throughput* leitura do *hardware* antes e depois desse trabalho

Fonte: Autoria própria.

Um dos resultados apresentados na Figura 36 apresenta um ganho negativo para a leitura de um pacote de 128 bytes. Isso deve-se ao acréscimo de *flip-flops* para reduzir o caminho crítico de lógica combinacional para que o *hardware* fosse capaz de trabalhar com 8 lanes. Portanto, introduziu-se uma pequena latência no início da transferência.

Figura 36 - Ganho percentual de velocidade do *hardware*

Fonte: Autoria própria.

Ao comparar o ganho de velocidade entre os resultados anteriores e os atuais, há uma pequena perda de desempenho, cerca de 3%, apenas para quantidades de dados bem pequenas (128 *bytes*). Além de ser um decréscimo pouco expressivo, as transferências de pequenas quantidades de dados não são o foco da utilização de DMA.

4.1.2 Recursos utilizados da FPGA

Normalmente o aumento de desempenho em um circuito de uma FPGA, está associado ao aumento de recursos ou de lógica utilizada pela mesma. Porém, durante o desenvolvimento das modificações para melhoria na velocidade, tanto da transmissão quanto da recepção, foram estudados métodos de obter o menor aumento do consumo de recursos.

Os resultados obtidos após a síntese do *framework* PCI Express com as alterações descritas podem ser observados na Tabela 7.

Tabela 7 – Recursos utilizados na FPGA após a síntese utilizando o Quartus Prime para 4 *lanes*

	Quantidade utilizada	Quantidade disponível	Porcentagem de utilização
ALUTs combinacionais	2.720	182.400	1%
ALUTs de memória	80	91.200	~0%
Lógica dedicada a registro	3.889	182.400	2%
Total de blocos de memória (bits)	81.054	14.625.792	1%
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%

Ao contrário do esperado, a quantidade de lógica utilizada foi reduzida, assim como a quantidade de blocos de memória utilizados. Essa diminuição deve-se principalmente às mudanças feitas no módulo de gerenciamento de *tags*. Para uma demonstração mais clara da redução de recursos de *hardware*, a Figura 37 mostra a comparação percentual de utilização dos principais recursos da FPGA antes e depois das alterações.

É possível observar, na Figura 37, uma redução significativa da quantidade de blocos de memória utilizados, uma vez que também as FIFO utilizadas foram reduzidas de tamanho. A análise das latências de cada trecho do circuito foi utilizada para definir os novos tamanhos.

Isso significa que as FIFO estão colocadas para compensar latências específicas. Nos casos que as FIFO não são capazes de compensar as latências, todo o circuito foi desenhado para entrar em um estado de espera até que se possa continuar com o processo de transferência.

A quantidade de Altera LUT (ALUT) combinacionais também foi reduzida, devido a otimização do circuito lógico, proporcionando uma queda razoável nas células lógicas utilizadas. As mudanças no gerenciamento de *tags* fizeram com que fosse utilizada uma quantidade menor de lógica, uma vez que o circuito de controle foi simplificado.

Figura 37 - Comparação percentual da utilização de recursos da FPGA antes e depois das modificações realizadas

Fonte: Autoria própria.

Entretanto, a quantidade de lógica dedicada a registro teve um pequeno aumento apesar da simplificação dos circuitos lógicos. Isso deve-se ao fato de que foi necessário um maior número de *flip-flops* para reduzir o comprimento dos caminhos da lógica combinacional, uma vez que para a utilização com 8 *lanes* a frequência do sinal de *clock* é maior (250 MHz).

O número de LUT utilizadas para memória também aumentou, apesar de ser um aumento pouco significativo. Isso ocorreu, pois, algumas FIFO foram reduzidas em profundidade ao ponto em que a utilização de um bloco dedicado de memória tornou-se desperdício, com isso, a ferramenta de síntese utilizou as LUT para esse armazenamento. Vale a pena ressaltar que esse aumento não significa um aumento de lógica de uma forma geral, uma vez que a soma total de LUT foi reduzida.

4.2 Software

As alterações realizadas no *software* não têm um impacto direto no desempenho do *hardware*. Porém, tais mudanças são importantes para a redução da latência das tarefas de obtenção de dados. Dessa forma, obtém-se um *throughput* maior do *software*, tornando-o capaz de obter um número maior de informações em um tempo menor, o que para a Ressonância Magnética significa resultados mais rápidos.

4.2.1 Fim da transmissão

No processo de transmissão de dados por DMA, o *software* não realiza a transferência. Apenas informa os parâmetros necessários e aguarda o fim da operação. A informação de que as transferências por DMA terminaram é armazenada no registrador de estado do controlador DMA. O *driver* é o componente responsável por descobrir que as transferências foram concluídas e retomar controle dos dados. Dois métodos são comumente utilizados para essa tarefa: *polling* e interrupções.

Primeiramente, utilizou-se o método *polling* que consiste em ler o registrador de estado e verificar se o *bit* correspondente ao fim de transferência está ativo, caso contrário repete-se ciclicamente esse processo até um número de iterações máximo. O algoritmo do código executado para essa operação pode ser observado na Figura 38.

Figura 38 - Exemplo de código de verificação de fim da transmissão DMA utilizando *polling*

```
count = DMA_POLLING_TIMEOUT;
while (count--) {
    // lê o registrador de estado do controlador DMA
    status = ioread32(&channel->regs->status);

    if (status & DMA_STATUS_DONE_MASK)
        return 0;
}
dev_warn(dev, "DMA Polling method timed out.\n");
return -1;
```

Fonte: Autoria própria.

Um problema com esse método é a utilização do processador durante a operação de DMA. O *driver*, que é executado pelo núcleo do sistema e possui maior prioridade do que as aplicações do usuário, fica constantemente executando requisições à placa. Portanto, consome-se tempo de processamento da CPU e da placa, uma vez que as requisições devem ser respondidas com um pacote de complementação.

Para a medição do tempo gasto entre a chamada das operações de DMA, feita pela aplicação em espaço do usuário, e o retorno dos dados utilizou-se a função *clock_gettime*. A Tabela 8 representa os tempos e velocidades obtidos para um *link* x4 PCI Express limitado a 128 bytes por pacote e utilizando 32 KBytes de *buffer* no *driver* com o método *polling*. As transferências foram feitas com diversas quantidades de dados e juntamente com o tempo obtido de cada transferência foi possível calcular o *throughput* como na Tabela 6.

Tabela 8 – Tempo de resposta e *throughput* das transferências no *software* utilizando o método *polling*.

Tamanho dos dados [B]	Tempo de leitura [μ s]	Tempo de escrita [μ s]	Throughput de leitura [Mbits/s]	Throughput de escrita [Mbits/s]
262.144	202,19	195,99	10.493,3	10.795,1
131.072	106,54	100,56	10.173,1	10.600,8
65.536	50,39	47,79	10.519,8	11.025,0
32.768	24,18	23,40	10.841,7	11.258,5
16.384	13,54	12,61	9.816,8	10.500,6
8.192	7,76	7,50	8.583,6	9.036,2
4.096	6,86	6,09	5.146,4	5.943,5
2.048	4,33	4,09	3.902,5	4.182,3
1.024	3,70	3,70	2.382,3	2.491,5
512	3,82	3,19	1.302,5	1.379,3
256	3,42	2,91	694,6	728,6
128	1,93	1,77	541,0	586,1

O processo de DMA utilizando interrupções provê menor utilização do processador durante as transmissões, uma vez que a interrupção avisa o término das operações e o *driver* não precisa ficar constantemente verificando o estado da transmissão, podendo “hibernar” e assim ceder o processador para outro processo. Entretanto, há um pequeno *overhead* causado pela chamada da função de interrupção e depois pela ativação do processo em hibernação.

Ao se fazer os mesmos testes de tempo de resposta com o método de interrupções, foram obtidos os resultados representados na Tabela 9. Devido a requisição de apenas um sinal de interrupção para o canal DMA executando as transferências, não foi necessária a construção de toda a tabela MSI-X, logo, somente interrupções do tipo MSI foram utilizadas.

Tabela 9 – Medidas de tempo de resposta e *throughput* das transferências no *software* utilizando o método de interrupção.

Tamanho dos dados [B]	Tempo de leitura [μ s]	Tempo de escrita [μ s]	Throughput de leitura [Mbits/s]	Throughput de escrita [Mbits/s]
262.144	200,11	198,15	10.499,6	10.598,7
131.072	98,85	97,26	10.616,8	10.783,8
65.536	51,42	49,74	10.235,8	10.614,4
32.768	24,19	23,91	10.837,0	10.962,3
16.384	13,36	13,22	9.808,2	9.915,0
8.192	7,98	8,07	8.215,5	8.125,0
4.096	5,55	6,14	5.910,6	5.370,2
2.048	4,04	4,87	4.068,1	3.630,6
1.024	3,94	3,68	2.370,1	2.228,9
512	3,28	3,75	1.356,9	1.190,2
256	2,80	4,65	739,4	570,9
128	3,66	5,04	300,2	264,0

Para uma comparação entre os métodos de espera pelo fim da transferência os dados de *throughput* foram representados nos gráficos da Figura 39 e Figura 40.

Figura 39 - Curvas de *throughput* de leitura pelo *software* para cada método de espera

Fonte: Autoria própria.

A leitura de dados utilizando ambos os métodos não apresentaram diferenças significativas. Dessa forma, o *overhead* da operação de interrupção não foi significativo com relação ao tempo das operações e a utilização do processador para a verificação do estado da transferência também não foi relevante.

Figura 40 - Curvas de *throughput* de escrita pelo *software* para cada método de espera

Fonte: Autoria própria.

Os resultados obtidos para as operações de escrita na placa (Figura 40) foram muito similares aos de leitura. Com isso, adotou-se o método de interrupção como o padrão já que durante o período de transferência não ocorre utilização da CPU. O método de *polling* foi colocado no *driver* como uma medida de segurança caso ocorra que a mensagem de interrupção não seja recebida dentro de um intervalo de tempo.

4.2.2 Scatter & Gather

Apesar das altas taxas de transferências do *hardware* e da baixa latência na conclusão da transmissão, notou-se que o *throughput* para as aplicações rodando em modo usuário poderia ser aumentado ainda mais. Analisando-se as transferências de grandes quantidades de dados, o maior tempo gasto deve-se à movimentação do dado da região de memória do núcleo do sistema para a região de memória da aplicação e vice-versa. Isso envolve a participação do processador para a leitura e escrita dos dados na memória principal.

A utilização do método *Scatter & Gather*, consiste do acesso direto ao espaço de memória do usuário, removendo a necessidade de movimentação dos dados após a transferência por DMA. Com isso, o tempo entre a chamada de transferência de dados e seu retorno é reduzido, portanto o *throughput* efetivo é aumentado, ou seja, maior quantidade de dados pode ser adquirida no mesmo intervalo de tempo.

Entretanto, o processo de mapeamento e obtenção de múltiplos segmentos para cada transferência causa um aumento do *overhead*, o que pode se refletir em um desempenho pior para pequenas quantidades de dados. A Figura 41 representa a comparação entre a utilização do método de *Scatter & Gather* para a leitura de dados.

Figura 41 - Curvas de *throughput* de leitura com utilização ou não do método de *Scatter & Gather*

Fonte: Autoria própria.

Nota-se que para transferências acima de 32KiB o acesso direto a memória do usuário apresenta valores de *throughput* superiores fazendo com que se possa obter resultados tão bons quanto os apresentados diretamente pelo *hardware*.

A Figura 42 representa o *throughput* em *software* para a operação de escrita.

Figura 42 - Curvas de *throughput* de escrita com utilização ou não do método de *Scatter & Gather*

Fonte: Autoria própria.

Como a divisão da memória em segmentos, depende de como o sistema operacional realiza a alocação de memória, nem sempre o mesmo número de segmentos é alocado para uma mesma quantidade de dados. O mapeamento para a leitura ou escrita também influencia no número de segmentos passados ao *hardware*. Dessa forma, para pequenas quantidades de dados, como representado pela Figura 42, é possível que a utilização do acesso direto a memória do usuário não seja vantajoso. Entretanto, pode-se notar que para grandes quantidades de dados, acima de 1 MiB, o método de *Scatter & Gather*, pode ser utilizado para obter um desempenho superior do sistema.

4.2.3 Link x8

O novo *kit* de desenvolvimento (Terasic DE4) suporta a utilização de 8 *lanes* PCI Express, portanto a expansão para essa nova quantidade faz parte desse trabalho. Como o *framework* já foi desenhado para operar com um número genérico de *lanes* a mudança de 4 para 8 *lanes* consiste simplesmente do aumento da frequência de trabalho de 125 MHz para 250 MHz. Com o aumento da frequência, o período do sinal de *clock* é reduzido pela metade, o que significa que os caminhos dos sinais de lógica devem ser encurtados.

A operação em 250 MHz faz com que a temporização da lógica sintetizada nem sempre fique dentro das condições ótimas de operação. A validação da temporização fica dependente das sementes iniciais utilizadas pelo algoritmo de síntese. Diversos esforços foram feitos para diminuir essa dependência, como reestruturação do código que descreve o *hardware*, bem como alteração dos parâmetros do sintetizador. A validação da temporização pode ser obtida através da exploração das sementes utilizando a ferramenta *Design Explorer*, que vem com o Quartus II.

4.2.3.1 Hardware

A operação em 8 *lanes* também aumenta a quantidade de recursos utilizados da placa. Entretanto, como pode-se verificar com os dados da Tabela 10 e da Tabela 7 a diferença de lógica sintetizada não é expressiva, fazendo com que seja vantajoso a utilização com maior número de *lanes*.

Os únicos recursos que aumentaram linearmente com o acréscimo de *lanes* foram os receptores e transmissores GXB. Porém, a FPGA utilizada possui um número mais do que suficiente desse tipo de dispositivo.

Tabela 10 - Recursos utilizados na FPGA após a síntese utilizando o Quartus Prime para 8 *lanes*

	Quantidade utilizada	Quantidade disponível	Porcentagem de utilização
ALUTs combinacionais	2.752	182.400	2%
ALUTs de memória	88	91.200	~0%
Lógica dedicada a registro	3.934	182.400	2%
Total de blocos de memória (bits)	81.054	14.625.792	1%
Total de blocos de DSP de 18 - bits	0	1.288	0%
Total de receptores GXB canal PCS	8	24	33%
Total de receptores GXB canal PMA	8	36	22%
Total de transmissores GXB canal PCS	8	24	33%
Total de transmissores GXB canal PMA	8	36	22%
Total de PLLs	1	8	13%
Total de DLLs	0	4	0%

Como a frequência de operação dos circuitos lógicos desenvolvidos foi dobrada, o desempenho do *hardware* também é dobrado como pode-se observar pelos resultados apresentados na Tabela 11. Os valores obtidos seguiram as mesmas condições dos valores da Tabela 6 apenas com o número de *lanes* modificado para 8.

Tabela 11 - Medidas de tempo de resposta e *throughput* das transferências no *hardware* com 8 *lanes*

Tamanho dos dados [B]	Tempo de leitura [μ s]	Tempo de escrita [μ s]	Throughput de leitura [Mbits/s]	Throughput de escrita [Mbits/s]
262.144	78,92	85,09	26.573,10	25.186,02
131.072	39,44	41,02	26.584,34	25.571,84
65.536	19,72	20,49	26.585,74	25.592,85
32.768	9,86	10,71	26.583,15	24.813,94
16.384	4,98	5,30	26.319,70	24.734,24
8.192	2,54	2,86	25.792,10	22.941,85
4.096	1,32	1,64	24.839,88	19.943,83
2.048	0,71	1,03	23.100,29	15.890,60
1.024	0,40	0,72	20.277,20	11.335,62
512	0,25	0,57	16.215,18	7.246,54
256	0,18	0,48	11.636,40	4.233,09
128	0,14	0,45	7.306,16	2.276,18

4.2.3.2 Software

Foram realizados os mesmos testes de medição do *throughput* para o *software* como anteriormente. Pode-se observar pelas Figura 43 e Figura 44 a comparação das transferências de leitura e escrita, respectivamente, entre a utilização de 4 ou 8 *lanes*, com e sem o método de *Scatter & Gather*.

Figura 43 - Curvas de comparação entre a utilização de diferentes números de lanes para a leitura

Fonte: Autoria própria.

A partir dos dados representados na Figura 43, nota-se uma grande vantagem na utilização de *Scatter & Gather* em 8 lanes, possibilitando atingir valores de *throughput*, em *software*, próximos aos do *hardware*, em torno de 25 Gbits/s.

Figura 44 - Curvas de comparação entre a utilização de diferentes números de lanes para a leitura

Fonte: Autoria própria.

Para a escrita, novamente, o acesso direto à memória do usuário causa um *overhead* na transferência de pequenas quantidades. Porém, assim como no caso da leitura, para grandes quantidades de dados, obteve-se em torno de 25 Gbits/s.

5 CONCLUSÕES

Enquanto a comunicação por *Ethernet Gigabit* é capaz de suportar até 16 canais de aquisição de radiofrequência em um espectrômetro de Ressonância Magnética, já existem aplicações comerciais de 32 e experimentais de 64, 96 e 128 canais. A nova interface de comunicação do DMRS do CIERMag, por PCI Express, proporciona a expansão para uma quantidade maior de canais de aquisição e é um diferencial em relação aos espectrômetros comerciais. As velocidades de transferência obtidas por PCI Express e o baixo *overhead*, em comparação com *Ethernet*, são características desejadas na comunicação de dados de um espectrômetro de Ressonância Magnética possibilitando a expansão do número de canais de aquisição de radiofrequência para mais de 300.

A execução dos testes de aquisição de dados utilizando todo o espectrômetro com a comunicação PCI Express proporcionou os resultados esperados, uma vez que toda operação funcionou exatamente como no caso do *Ethernet Gigabit*, consolidando os objetivos desse projeto.

Para manter a característica do DMRS em FPGA de ampla abrangência às aplicações, foram realizadas modificações no controlador DMA e melhorias no *device driver* do *framework* PCI Express previamente desenvolvido. Com isso, foi possível atingir taxas de transferências muito elevadas, aproximadamente 12 Gbits/s com 4 lanes e 25 Gbits/s com 8 lanes, possibilitando a utilização de um número de canais de aquisição de radiofrência muito superior do que os utilizados em equipamentos comerciais. Além disso, a flexibilidade proporcionada pela FPGA faz com que essa comunicação possa ser utilizada por todo o sistema como o tronco principal de comunicação com um computador.

Juntamente com o aumento do *throughput* de dados, reduziu-se a lógica utilizada na FPGA, devido à otimização do sistema de comunicação como um todo. Essa redução de recursos é muito importante, pois ao agregar o módulo PCIe com o resto do espectrômetro deseja-se causar o menor impacto possível, deixando o máximo de recursos disponíveis para expansão e melhoria do espectrômetro em si.

O tempo de um exame de imagens por Ressonância Magnética é fortemente impactado pelo tempo de aquisição e processamento dos dados. Dessa forma, um estudo cauteloso de todo o código do *device driver* foi feito com o objetivo de reduzir etapas e utilização do processador, a fim de diminuir o impacto da comunicação no tempo total utilizado pelo *software*.

As técnicas empregadas nesse trabalho, como transferência por DMA, *Scatter & Gather* com acesso à memória do usuário e utilização de interrupção, normalmente são utilizadas em aplicações que necessitam de um elevado desempenho, apesar de consumirem muito tempo de desenvolvimento. As informações sobre essas técnicas não são facilmente obtidas, nem possuem a clareza necessária para o rápido entendimento. Portanto, foi necessário um estudo do código de *device drivers* do *kernel* do Linux, que também não apresentam documentação dessas técnicas, mas serviram como base para o desenvolvimento desse trabalho.

Embora a comunicação por PCI Express tenha sido feita mantendo compatibilidade com as interfaces existentes, um melhor aproveitamento de toda a capacidade implantada nesse trabalho, como a existência de canais de DMA independentes, seria obtido com a utilização de um canal de DMA por receptor de radiofrequência, sem necessidade de multiplexação. Para isso, alterações no *hardware* e nas camadas de *software* do DMRS poderão ser realizadas, em trabalhos futuros, dando continuidade ao desenvolvimento do espectrômetro.

Os resultados obtidos a partir do desenvolvimento desse trabalho abrem, contudo, novas possibilidades de aplicação do DMRS do CIERMag, mantendo a linha de desenvolvimento de um equipamento altamente parametrizável.

Vale a pena citar que embora a aplicação principal desse trabalho seja no espectrômetro do CIERMag, o qual necessita de um canal de alta velocidade com um computador, esse trabalho pode ser utilizado em qualquer sistema que também tenha a mesma necessidade. Como na sequência natural de aplicação em equipamentos que demandam cada vez um maior volume de dados transferidos e maior velocidade de aquisição, podemos citar espectrômetros de Ressonância Paramagnética Eletrônica Pulsada, tomógrafos de Raio-X e aceleradores de partículas.

REFERÊNCIAS⁸

- ALTERA CORPORATION. **FPGA Architecture**. San Jose, CA, USA: Altera Corporation, jul. 2006. Disponível em: <https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf>. Acesso em: 13 jun. 2016.
- ALTERA CORPORATION. **IP Compiler for PCI Express**. San Jose, CA, USA: Altera Corporation, ago. 2014. Disponível em: <https://www.altera.com/en_US/pdfs/literature/ug/ug_pci_express.pdf>. Acesso em: 13 jun. 2016.
- BABU, N. K.; BABU, P. S. S. Design of Physical Coding Sublayer using 8B/10B Algorithm. **International Journal of Recent Technology and Engineering**, v. 2, n. 2, p. 114–117, maio 2013.
- BROWN, S.; ROSE, J. Architecture of FPGAs and CPLDs: A Tutorial. **IEEE Design & Test of Computers**, v. 3, n. 2, p. 42–57, 1996.
- BRYANT, R. E.; O'HALLARON, D. R. **Computer Systems: A Programmer's Perspective**. 2. ed. Boston, MA, USA: Pearson, 2010.
- BUDRUK, R. **PCI Express® Basics**. In: PCI-SIG DEVELOPERS CONFERENCE. , 2007. Disponível em: <http://www.uio.no/studier/emner/matnat/ifi/INF5063/h14/slides/03_29_pci_express_basics.pdf>. Acesso em: 13 jun. 2016
- BUDRUK, R.; ANDERSON, D.; SHANLEY, T. **PCI Express System Architecture**. 9. ed. [s.l.] MindShare, Inc., 2008.
- CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers**. 3. ed. Sebastopol, CA, USA: O'Reilly Media, 2009.
- DREPPER, U. What Every Programmer Should Know About Memory. 21 nov. 2007.
- HARVEY, A. F.; DATA ACQUISITION DIVISION STAFF. **DMA Fundamentals on Various PC Platforms**. [s.l.] National Instruments Corporation., abr. 1991. Disponível em: <<http://www.cs.umb.edu/~bobw/CS341/DMAFundamentals.pdf>>. Acesso em: 15 jun. 2016.
- HINDRIKSEN, V. **GPUDirect and DirectGMA – direct GPU-GPU communication via RDMA**. 18 abr. 2015. Disponível em: <<http://streamcomputing.eu/blog/2015-04-18/gpudirect-and-directgma-direct-gpu-gpu-communication/>>. Acesso em: 14 jun. 2016.
- JACKSON, M.; BUDRUK, R. **PCI Express Technology: Comprehensive Guide to Generations 1.x, 2.x and 3.0**. 1. ed. [s.l.] MindShare, Inc., 2012.
- KUON, I.; TESSIER, R.; ROSE, J. FPGA Architecture: Survey and Challenges. **Foundations and Trends® in Electronic Design Automation**, v. 2, n. 2, p. 135–253, 2007.

---

⁸De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

- LAWLEY, J. **Understanding Performance of PCI Express Systems**. [s.l.] Xilinx, Inc., 28 out. 2014. Disponível em: <http://www.xilinx.com/support/documentation/white_papers/wp350.pdf>. Acesso em: 15 jun. 2016.
- LI, H. et al. A Wrapper of PCI Express with FIFO Interfaces based on FPGA. **arXiv:1306.5501 [cs]**, 23 jun. 2013.
- MARES, M.; GRUNDLER, G. **How To Write Linux PCI Drivers**. [s.l.] Linux Kernel Organization, Inc., 2006. Disponível em: <<https://www.kernel.org/doc/Documentation/PCI/pci.txt>>. Acesso em: 13 jun. 2016.
- MARTINS, T. A. **Desenvolvimento de um framework PCI Express para FPGA**. Trabalho de Conclusão de Curso—São Carlos, SP, Brasil: Universidade de São Paulo, 23 jun. 2015.
- NAVE, C. R. **The D Flip-Flop**. 2012. Disponível em: <<http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/dflipflop.html>>. Acesso em: 13 jun. 2016.
- NICOLLE, J. P. **PCI Express - Connector**. 15 fev. 2012a. Disponível em: <<http://www.fpga4fun.com/PCI-Express1.html>>. Acesso em: 13 jun. 2016.
- NICOLLE, J. P. **PCI Express - The transaction layer**. 15 fev. 2012b. Disponível em: <<http://www.fpga4fun.com/PCI-Express4.html>>. Acesso em: 13 jun. 2016.
- NICOLLE, J. P. **Internal RAM operation**. 20 maio 2013a. Disponível em: <<http://www.fpga4fun.com/FPGAinfo3.html>>. Acesso em: 13 jun. 2016.
- NICOLLE, J. P. **FPGA pins**. 21 maio 2013b. Disponível em: <<http://www.fpga4fun.com/FPGAinfo4.html>>. Acesso em: 13 jun. 2016.
- NICOLLE, J. P. **How FPGAs work**. 14 out. 2013c. Disponível em: <<http://www.fpga4fun.com/FPGAinfo2.html>>. Acesso em: 13 jun. 2016.
- PACHCHIGAR, M. High-Performance Data-Acquisition System Enhances Images for Digital X-Ray and MRI. **Analog Dialogue**, v. 47, 10 out. 2013.
- TALA, D. K. **Digital Logic Families Part-I**. 9 fev. 2014. Disponível em: <<http://www.asic-world.com/digital/logic1.html>>. Acesso em: 16 jun. 2016.
- TANENBAUM, A. S. **Modern Operating Systems**. 3. ed. Upper Saddle River, N.J: Pearson, 2007.
- TEMBHARE, A. R.; PATIL, P. B. Design & Implementation of PCI Express BUS Physical layer using VHDL. **International Journal on Recent and Innovation Trends in Computing and Communication**, v. 2, n. 7, p. 1883–1886, jul. 2014.
- TERASIC INC. **Altera DE4 Development and Education Board**. 2013. Disponível em: <<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=138&No=501>>. Acesso em: 13 jun. 2016.
- WILSON, T. V. **How PCI Express Works**. 17 ago. 2005. Disponível em: <<http://computer.howstuffworks.com/pci-express.htm>>. Acesso em: 13 jun. 2016.

XILINX, INC. **Channelized Direct Memory Access and Scatter Gather**. [s.l.] Xilinx, Inc., 25 fev. 2010. Disponível em:
<http://www.xilinx.com/support/documentation/ip_documentation/chan_dma_sg.pdf>.
Acesso em: 14 jun. 2016.

XILINX, INC. **Field Programmable Gate Array (FPGA)**. 2015a. Disponível em:
<<http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm>>. Acesso em:
13 jun. 2016.

XILINX, INC. **What is Programmable Logic?** 2015b. Disponível em:
<<http://www.xilinx.com/company/about/programmable.html>>. Acesso em: 13 jun. 2016.

XILLYBUS LTD. **Down to the TLP: How PCI express devices talk (Part I)**. 2015.
Disponível em: <<http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1>>.
Acesso em: 13 jun. 2016.