final

Pontifícia Universidade Católica do Rio Grande Do Sul
Faculdade de Engenharia - Faculdade de Informática
Engenharia de Computação

Augusto Gosmann Erichsen
Thiago Mânica Monteiro

Desenvolvimento de Interface PCIe para Comunicação com MPSoCs em FPGAs

Volume Final do Trabalho de Conclusão de Curso
Orientador: Prof. Dr. Fernando Gehm Moraes

Porto Alegre 2016

Augusto Gosmann Erichsen
Thiago Mânica Monteiro

Desenvolvimento de Interface PCIe para Comunicação com MPSoCs em FPGAs

O trabalho de conclusão de curso é apresentado como parte das atividades para obtenção do grau de Engenharia de Computação da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul.

Volume Final do Trabalho de Conclusão de Curso
Orientador: Prof. Dr. Fernando Gehm Moraes

Porto Alegre 2016

AGRADECIMENTOS

Eu, Augusto Gosmann Erichsen, agradeço a todos meus colegas e amigos que direta e indiretamente contribuíram com seu apoio para o desenvolvimento deste trabalho. Gostaria de agradecer imensamente aos meus pais pelo apoio e por terem possibilitado meus estudos nesta instituição. Agradeço ainda aos meus pais e a minha irmã por terem me ensinado que o nosso trabalho e esforço trazem conquistas e abrem possibilidades na vida. Agradeço também a minha mulher Ariana Bastos Teixeira e a minha sogra Carmen Bastos por todo apoio proporcionado por elas. Com o apoio de quem nos quer bem tudo se torna mais fácil e com mais confiança buscamos nossos objetivos.

Eu, Thiago Mânica Monteiro, gostaria de agradecer a todas as pessoas que fizeram parte da minha vida, pois todas elas ajudaram um pouquinho a fazer de mim o que sou hoje. Aos meus amigos de infância e faculdade, com eles sempre tive força e alegria para continuar. Quero agradecer, principalmente, a minha família, minha mãe Lair de Castro Mânica, que sempre apoiou todas as minhas decisões, sendo a melhor conselheira, que com muito amor me fez ser o homem engenheiro que sou hoje. Ao meu pai, Geraldo Henrique de Souza Monteiro Filho, que com sua garra e discernimento, me apoiou em momentos difíceis nesse trajeto de formação profissional, que com muito amor me fez ser o homem engenheiro que sou hoje. À minha irmã, Deborah Mânica Monteiro, sem palavras, “a pessoa que mais amo nesse mundo”.

Os barbudos Augusto e Thiago agradecem a todos os colegas e amigos, que não estão se formando conosco, mas que também estiveram sempre presentes nas nossas vidas. Agradecemos aos membros do grupo GAPH que nos ajudaram em diversos momentos na nossa formação profissional. Agradecemos aos professores Ney Calazans e Alexandre de Morais Amory, pela disponibilidade e paciência de corrigir e avaliar nosso trabalho.

Em especial, gostaríamos de agradecer ao professor Moraes, por todas as oportunidades e orientações durante todos estes anos. É indescritível nosso preparo e evolução profissional graças ao conhecimento que nos foi passado. Muito obrigado!

Amigos verdadeiros são os que suportam a tua felicidade!

Padre Fábio de Melo

DESENVOLVIMENTO DE INTERFACE PCIE PARA COMUNICAÇÃO COM MPSOCS EM FPGAS

RESUMO

Simulação e prototipação são métodos comumente utilizados para a validação de sistemas digitais. Contudo, simulações acabam por vezes demorando horas ou dias dependendo do sistema avaliado. Como alternativa, a prototipação em FPGAs é muito utilizada, permitindo acelerar o processo de desenvolvimento e depuração do hardware. O presente Trabalho de Conclusão de Curso apresenta o desenvolvimento de uma interface *PCI Express* permitindo a comunicação entre um computador hospedeiro (*host*) e o projeto prototipado no FPGA. Neste trabalho são empregados dois projetos de referência. O primeiro projeto refere-se ao *core PCIe* e um módulo que faz a interface entre o *PCIe* e a lógica do usuário. O segundo projeto refere-se ao HeMPS-ECS, que compreende um módulo para gerência de admissão de novas aplicações (ECS) e o MPSoC HeMPS. Através do *host*, é feita a inserção de aplicações no sistema, passíveis de serem observadas via *chipscope*. Os resultados deste trabalho apresentam a integração e a prototipação em FPGA do *core PCIe* juntamente ao módulo ECS, sendo esta parte verificada através da ferramenta *chipscope* e com mensagens de depuração enviadas ao *host*. O projeto HeMPS-ECS foi validado apenas por simulação, dadas as restrições de tempo do projeto.

Palavras-chave: FPGA, prototipação, *PCI Express,* MPSoC.

DEVELOPMENT OF A PCIE INTERFACE FOR COMMUNICATION WITH MPSOCS IN FPGAS

ABSTRACT

Simulation and prototyping are methods commonly used to validate digital systems. However, simulations may require hours or days depending on the evaluated system. Prototyping, in FPGAs, is widely used since it allows to accelerate the development and debugging of the hardware. This End-of-Term work presents the development of a PCI Express interface (PCIe) enabling the communication between a host computer and the prototyped design. This work adopts two reference designs. The first design is the Xilinx PCIe core and a module that interfaces the PCIe core with the user logic. The second design is the HeMPS-ECS, which comprises a module for the admission of new applications to execute (ECS) and MPSoC Hemps. The host inserts new applications in the MPSoC, and the internal signals of the design may be observed using the Chipscope tool. The results show the integration and prototyping of the PCIe core with the ECS module, being verified with ChipScope and debug messages sent to the host. The HeMPS-ECS desing was validated only by simulation, due to restrictions related to the End-of-Term work duration.

Keywords: FPGA, prototyping, PCI Express, MPSoC.

Sumário

Lista de Siglas

ASIC	-	*Application Specific Integrated Circuits*
BAR	-	*Base Address Register*
BRAM	-	*Block Random Access Memory*
CC	-	*Completer Completion*
CDR	-	*Clock Data Recovery*
CPU	-	*Central Processing Unit*
CQ	-	*Completer Request*
CRC	-	*Cyclic Redundancy Check*
DMA	-	*Direct Memory Access*
DMA-R	-	*Direct Memory Access - Read*
DMA-W	-	*Direct Memory Access - Write*
ECS	-	*External Control Access*
FPGA	-	*Field-Programmable Gate Array*
GPU	-	*Graphics Processing Unit*
GT/s	-	*Giga Transfer per Second*
HeMPS	-	*Hermes Multiprocessor System*
LMP	-	*Local Master Processor*
LUT	-	*Look up Table*
LVDS	-	*Low-voltage differential signaling*
MPSoC	-	*Multiprocessor System-on-chip*
NoC	-	*Network-on-chip*
PCI Express	-	*Peripheral Component Interconnect Express*
PCI-SIG	-	*Peripheral Component Interconnect Special Interest Group*
PE	-	*Processing Element*
PIO	-	*Programed Input Output*
RAM	-	*Random Access Memory*
RC	-	*Request Completion*
RQ	-	*Requester Request*
RTL	-	*Register-transfer Level*
SP	-	*Slave Processor*
TLP	-	*Transaction Layer Packet*
VHDL	-	*VHSIC Hardware Description Language*
VHSIC	-	*Very High Speed Integrated Circuit*

Lista de Figuras

Lista de Tabelas

1 INTRODUÇÃO

Diversas formas de transmissão de dados - transferência dos dados físicos (fluxo de bits digital ou sinal analógico digitalizado) - são largamente utilizadas nos dias atuais. Estas diferem no sentido de trocas de informações, modo de transmissão e sincronização, além do tipo de meio físico utilizado para a comunicação. Os padrões de transferência, e.g. modo de transmissão, designam o número de unidades elementares de informações (bits) que podem ser transmitidas de forma simultânea pelo canal de comunicação [CCM16]. Estes modos de transmissão são dois: série e paralelo. Por ligação paralela entende-se a transmissão simultânea de *N* bits, os quais são enviados através de *N* vias diferentes. Na ligação em série, os dados são enviados bit a bit na via de transmissão.

Especificamente, a transmissão de dados em barramentos de alto desempenho é um requisito importante para projetos que envolvam unidades de co-processamento, como placas gráficas (e.g. GPUs) ou placas com dispositivos FPGAs. Projetos em FPGAs podem ser utilizados como aceleradores, mas requerem uma interface de comunicação de alta velocidade.

O foco do presente Trabalho de Conclusão é estudar um barramento genérico e de alta velocidade denominado *PCI Express* (*PCIe*), muito utilizado, por exemplo, em placas de expansão, tais como placas de vídeo, som, rede e prototipação de *hardware*. Para que a interface *PCIe* opere adequadamente, o uso de uma aplicação de exemplo em *hardware* se faz necessária. Sendo assim, a essência do presente trabalho se torna o módulo *PCIe* com o projeto de exemplo de uso do *PCIe*.

Inicialmente, o padrão *PCI* surgiu como um barramento utilizado para conexões de placas externas ao computador, visando alta velocidade de processamento, sendo muito utilizado por placas de vídeo. Aplicações 3D, por exemplo, exigem taxas de transferências cada vez maiores. Isto fez com que o padrão *PCI* original passasse por uma evolução com o passar do tempo, desde o seu surgimento na década de 1990 [EME08].

Mesmo com as altas taxas de transferências proporcionadas pelo barramento *PCI*, novas aplicações passaram a exigir maior desempenho das interfaces. Os avanços da tecnologia de fabricação de circuitos integrados permitiram o desenvolvimento de um novo padrão, o *PCI Express*, o qual proporciona taxas de transferências de dados mais altas que o padrão *PCI*, utilizando conexões seriais para transferência de dados [CAS10].

Os projetistas de sistemas digitais enfrentam sempre o desafio de encontrar o balanço correto entre velocidade e generalidade de processamento do seu hardware. É possível desenvolver um *chip* genérico que realiza muitas funções

diferentes, porém com sacrifício de desempenho (por exemplo: microprocessadores), ou *chips* dedicados a aplicações específicas, estes com uma velocidade muitas vezes superior aos *chips* genéricos. Circuitos integrados dedicados (ASICs), têm como características a ocupação mínima de área de silício, maior custo em relação aos *chips* genéricos, rapidez e um menor consumo de potência comparados com microprocessadores. Um fator importante na escolha entre flexibilidade e velocidade é o custo da solução [CAP01].

Uma solução intermediária, que representa um balanço entre custo *versus* flexibilidade, são os FPGAs (*Field-Programmable Gate Array*). Estes circuitos configuráveis podem ser personalizados como diferentes ASICs. Esta tecnologia permite o projeto, teste e correção de circuitos integrados dedicados com um baixo custo de prototipação [CAP01].

O desempenho proporcionado pelo barramento *PCI Express* é uma opção a ser considerada para a comunicação de alto desempenho entre os dispositivos que utilizam FPGAs e o mundo externo ao sistema implementado no FPGA.

O presente trabalho possui como contribuição mais relevante a interface *PCIe* e seu projeto de exemplo de uso. Neste projeto foi desenvolvido um módulo DMA (*Direct Memory Access*) para alta taxa de transferência de dados. A taxa de transferência obtida é um importante resultado deste trabalho, sendo 84710 vezes maior do que a taxa de transferência do projeto original.

1.1 Motivação

Através do protocolo *PCIe* de transmissão de dados, é possível realizar uma interface de alta velocidade entre uma placa de prototipação e um computador hospedeiro. Assim, projetos implementados em FPGAs passam a ter um meio eficiente e rápido para se comunicarem com interfaces externas. Como prova de conceito, será prototipado em FPGA um sistema multiprocessado, conectado a uma interface *PCIe*. Com este projeto, é possível a um computador hospedeiro enviar aplicações para serem executadas no sistema multiprocessado e receber os dados relativos ao processamento da aplicação.

Assim, a motivação do trabalho é o aprofundamento de diversos tópicos vistos ao longo do curso, tais como fluxo de projeto de sistemas digitais, linguagens de descrição de hardware, sistemas embarcados, sistemas integrados de hardware e software, sistemas de comunicação e prototipação.

O projeto de referência utilizado é a união do módulo ECS (*External Control Access*) com a HeMPS (*Hermes Multiprocessor System*) [CAR09]. O ECS é um novo módulo agregado à plataforma HeMPS, responsável pelo gerenciamento dos recursos do MPSoC (*Multiprocessor System-on-chip*). A plataforma HeMPS, desenvolvida no Grupo de Apoio e Pesquisa em Hardware (GAPH), apresenta uma

infraestrutura de hardware e software capaz de gerenciar a execução de múltiplas aplicações paralelas com uma carga de trabalho dinâmica, ou seja, aplicações podem entrar em execução a qualquer momento [MON13]. Pelo fato de haver um volume expressivo de aplicações executando em paralelo, a gerência do sistema se torna complexa. A interface ECS tem a função de gerenciar as aplicações inseridas no MPSoC HeMPS.

1.2 Objetivos

Este trabalho de conclusão tem por objetivo estratégico a integração de um *core PCI Express* com o projeto de referência (HeMPS-ECS). O domínio do protocolo *PCIe*, assim como o domínio da plataforma HeMPS e do módulo ECS, a prototipação destes em hardware e o desenvolvimento de um software de controle representam os objetivos específicos do trabalho.

Este projeto exige o domínio do protocolo *PCIe*, assim como técnicas de prototipação, o estudo de dispositivos FPGA e a plataforma de desenvolvimento associada a este, bem como a integração dos módulos HeMPS-ECS com o *core PCI Express*. A placa de prototipação utilizada neste Trabalho de Conclusão de Curso é a NetSUME-FPGA [DIG15]. Esta placa dispõe de uma FPGA *Xilinx Virtex-7.*

1.3 Estrutura do Documento e do Projeto

1.3.1 Estrutura do Documento

O trabalho é organizado como segue. O Capítulo 2 descreve o protocolo *PCI Express*, suas características e funcionalidades. O Capítulo 3 descreve o projeto de referência para o uso do *core PCI Express.* O Capítulo 4 descreve o projeto de referência (HeMPS-ECS), seu funcionamento, assim como os recursos providos por este sistema multiprocessado. O Capítulo 5 descreve a integração realizada entre *PCI Express* e o projeto de referência HeMPS-ECS. O Capítulo 6 apresenta uma conclusão com análise de resultados.

1.3.2 Estrutura do Projeto

O projeto a ser apresentado neste manuscrito possui o diagrama de blocos apresentado na Figura 1. A Figura 1 apresenta o diagrama de blocos macro do projeto, onde temos a integração entre o *host* (computador hospedeiro) e a placa de prototipação*,* o *core PCI Express,* o módulo *Programed Input Output* (PIO), o módulo *External Control Access* (ECS) e o MPSoC HeMPS. Neste trabalho, o *core PCIe* é representado na cor verde, o *core* ECS na cor laranja, o MSoC HeMPS na cor azul, e as contribuições realizadas nestes módulos pelo triângulo em amarelo.

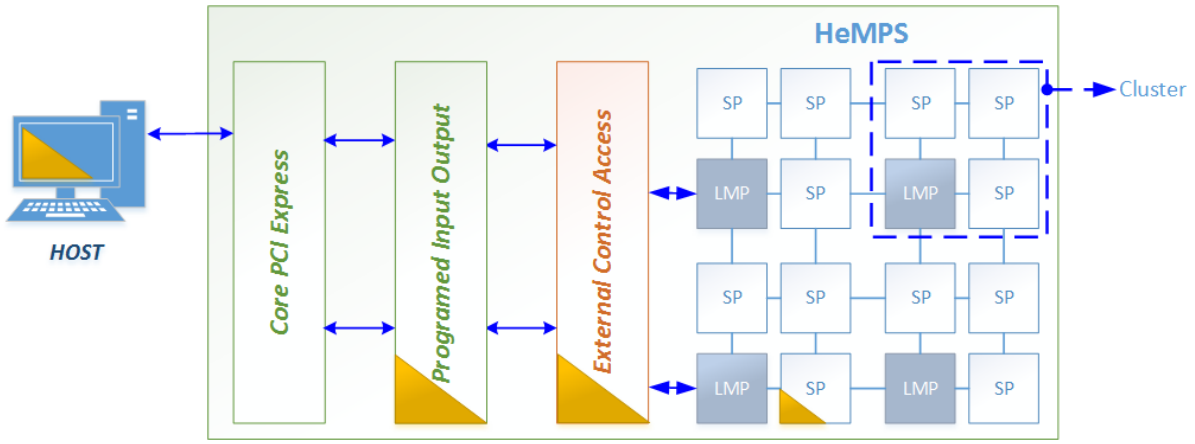

Figura 1 – Diagrama de blocos macro do projeto.

2 PCI EXPRESS

Neste Capítulo apresentamos o *core PCI Express* [XIL15] e suas características. A *Xilinx* oferece este bloco IP implementado em *hardware* como uma solução genérica para uso com FPGAs *Virtex-7*. Dentre algumas características suportadas por este *core*, podemos citar:

- Alta largura de banda;
- Comunicação serial com 1, 2, 4 ou 8 vias, que consiste em um par de sinais diferenciais de transmissão e um par de sinais diferenciais de recepção;
- Vazão de 2,5 GT/s, 5,0 GT/s e 8,0 GT/s, onde GT/s significa *giga transfer per second*, o mesmo que *giga bits per second* em uma via ou *giga transfer per second* em todas as vias [PCI10];
- Interface AXI4-Stream, o protocolo de comunicação da interface, que consiste em interfaces simétricas (mestre e escrava) comunicantes entre si, apresentada detalhadamente na seção 2.1. AXI é um protocolo primeiramente introduzido na família de micro controladores da ARM, denominada ARM AMBA. A primeira versão do AXI foi incluída na família AMBA 3.0. A família 4.0 foi criada em 2010. Dos 3 tipos de interface AXI4 (AXI4, AXI4-Lite e AXI4-Stream), utilizamos a AXI4-Stream, direcionada para alto desempenho na transmissão de dados.

A Tabela 1 apresenta as diferentes configurações de barramento *PCI Express*, que relaciona a geração *PCI Express* e sua respectiva velocidade de transmissão com a largura máxima da via de transmissão, largura do barramento AXI e a frequência de relógio utilizada.

Dentre as diversas configurações disponíveis, trabalharemos neste projeto com *PCIe* de terceira geração de velocidade (Gen3 – padrão utilizado pela *Xilinx*), possuindo um sistema de codificação que transfere 256 bits, com 8 vias de transmissão (x8) e taxa de transferência de 8 GT/s.

A arquitetura utilizada pelo *core PCIe* [XIL15] implementa uma pilha de protocolos dividida em quatro camadas, conforme apresentado na Figura 2, na qual os dados podem ser transferidos entre a memória da CPU, através do *core PCIe,* e a memória localizada na aplicação do usuário em *hardware*. A interface localizada entre o *core PCIe* e a aplicação do usuário é baseada no protocolo *AXI4-Stream* com frequência de operação de 250 MHz. O *core PCIe* também possui duas interfaces para informações de *status* conforme apresentamos na próxima seção.

Tabela 1 - Largura de dados e frequências de clock configuráveis para as interfaces do usuário [XIL15].

Geração PCIe/ Velocidade Máxima das Vias	Largura Máxima das Vias	Largura da Interface *AXI4-Stream*	Frequência de relógio (MHz)
Gen1 (2.5 GT/s)	x1	64 bits	62,5
		64 bits	125
		64 bits	250
	x2	64 bits	62,5
		64 bits	125
		64 bits	250
	x4	64 bits	125
		64 bits	250
	x8	64 bits	250
		128 bits	125
Gen2 (5 GT/s)	x1	64 bits	62,5
		64 bits	125
		64 bits	250
	x2	64 bits	125
		64 bits	250
	x4	64 bits	250
		128 bits	125
	x8	128 bits	250
		256 bits	125
Gen3 (8 GT/s)	x1	64 bits	125
		64 bits	250
	x2	64 bits	250
		128 bits	125
	x4	128 bits	250
		256 bits	125
	x8	**256 bits**	**250**

A arquitetura do protocolo *PCIe* é dividida em camadas, conforme apresentado na Figura 2 [CAS10]:

1. **Camada Física.** Um par de sinais diferenciais de baixa voltagem (do inglês, LVDS) transporta os dados de forma serial. O sinal é codificado com um sinal de relógio próprio, utilizando o esquema de '8b/10b' que significa que cada 8 bits são representados em 10 bits. A codificação é transparente às camadas superiores, como usual em protocolos de comunicação em camadas. A camada física está ligada às camadas de enlace de dados.
2. **Camada de Enlace de Dados.** Responsável pela integridade dos dados, inserindo nos pacotes um número de sequência e *Cyclic Redundancy Check*

(CRC). Esta camada também é responsável por reenviar novamente qualquer pacote corrompido. O controle de fluxo utilizado é baseado em créditos, para garantir que só haja transmissão de pacotes quando há espaço disponível nos buffers de entrada e saída.

3. **Camada de Transação.** Liga a camada do software com a camada de enlace, montando e desmontando os pacotes, que são utilizados para requisições de escrita e leitura. Esta camada também realiza o controle de fluxo de dados utilizando créditos entre as Camadas de Transação nos dispositivos em comunicação. Ela também recebe os pacotes de resposta das requisições da camada de software, entregando-os a seus respectivos remetentes.
4. **Camada de *Software*.** É onde o usuário interage com o Sistema Operacional através de *drivers* para se comunicar com a camada de transação.

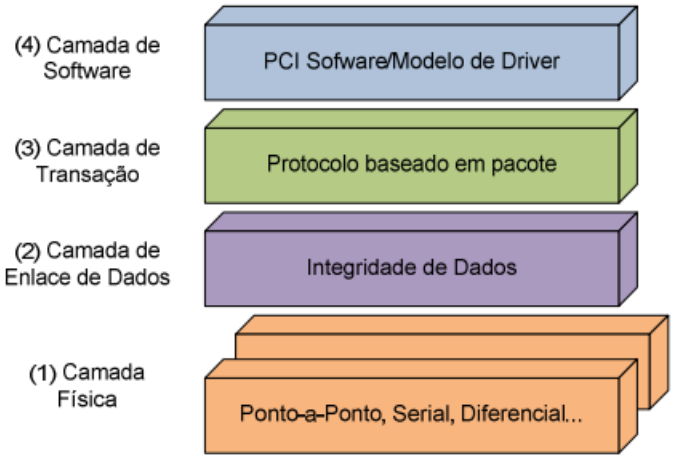

Figura 2 – Pilha de protocolos da arquitetura de interfaces PCIe [NAI09].

O padrão *PCIe* também inclui em sua especificação registradores de configuração e comunicação denominados de *Base Address Register* (BAR) com a especificação permitindo até 6 registradores BAR diferentes, designados BAR0, BAR1, BAR2, BAR3, BAR4 e BAR5. Estes registradores indicam áreas de memória da CPU utilizadas para configuração e comunicação com os dispositivos conectados ao *PCIe* [CAS10].

2.1 Interface AXI-Stream

A interface do usuário é compatível com a interface *AMBA AXI4-Stream* [ARM16], a qual define a comunicação entre o *core PCIe* (lado esquerdo da Figura 3) com a aplicação do usuário em hardware (lado direito da Figura 3). O padrão da indústria *Peripheral Component Interconnect Special Interest Group* (PCI-SIG) denomina de *Completer* a interface na qual as requisições são feitas pela *CPU* e entregues ao periférico (FPGA), *Requester* a interface na qual as requisições são

feitas pelo periférico e entregues a CPU e *Tag* um identificador único de cada requisição pendente. Esta abordagem é apresentada na Figura 3, dividida em quatro interfaces de comunicação e duas de controle: *Completer Request Interface, Completer Completion Interface, Requester Request Interface, Requester Completion Interface, Tag Availability Status, e Flow Control Status* [XIL15].

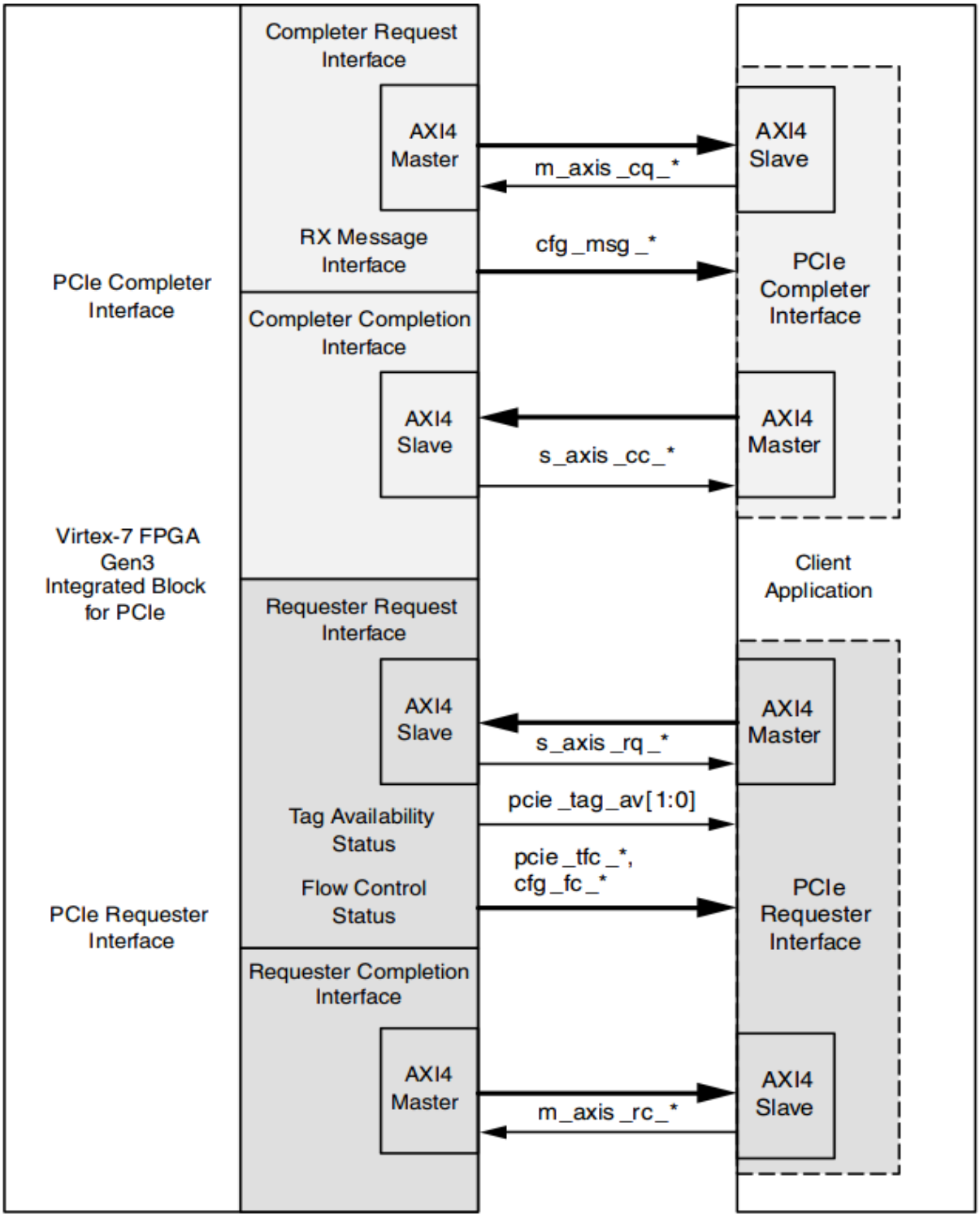

Figura 3 - Diagrama de blocos da interface do usuário do FPGA Virtex-7 [XIL15].

A comunicação é organizada em quatro interfaces, conforme a Figura 3:

- *Completer reQuest* (CQ): entrega requisições para a aplicação do usuário;
- *Completer Completion* (CC): a aplicação do usuário envia respostas às requisições realizadas pelo CQ e pode processar todas as transações como transações separadas (*split*), isto é, pode continuar aceitando novas solicitações do CQ, mesmo sem ter concluído a requisição anterior;
- *Requester reQuest* (RQ): interface através da qual a aplicação do usuário pode gerar requisições para a *CPU*.
- *Requester Completion* (RC): interface através da qual chega o retorno das mensagens RQ para a aplicação do usuário.

Além destas interfaces, o *core PCIe* possui duas interfaces através das quais as informações de *status* são transferidas do *core PCI Express* para a aplicação do usuário [XIL15]:

- *Flow Control Status:* fornece informações sobre transmissões atualmente disponíveis, de modo que a aplicação do usuário possa agendar requisições.
- *Tag Availability Status:* tags servem para identificar pacotes de dados. Porém, antes de detalhar as *Tags,* devemos explicar o conceito de *Posted* e *Non-Posted transactions*. Transações *Posted* são consideradas transações de escrita, pois não esperamos uma resposta, e as transações *Non-Posted* são transações de leitura, pois esperamos uma resposta. Dito isto, as *Tags são* utilizadas para transações *Non-Posted,* para que possamos identificar os pacotes de resposta. Logo, esta interface fornece informações sobre o número de *Tags* disponíveis para atribuir a requisições *Non-Posted* ainda não realizadas, de modo que o usuário possa agendar requisições sem o risco de serem bloqueadas por todas as *Tags* estarem sendo utilizadas, visto que a aplicação do usuário possui uma tabela com a quantidade de *Tags* disponíveis.

2.1.1 Transaction Layer Packet (TLP)

Um *Transaction Layer Packet* (TLP) é transferido em cada uma das interfaces *AXI4-Stream,* composto por um descritor seguido por dados de *payload* (quando o TLP tem dados válidos). No lado de transmissão (para o *host*, através da camada física), o *core PCIe* monta o cabeçalho TLP com os parâmetros fornecidos pela aplicação do usuário (conforme Figura 3). No lado de recepção (para a aplicação do usuário), o *core PCIe* extrai parâmetros dos cabeçalhos dos TLPs recebidos e forma os descritores para a entrega à aplicação do usuário. Para detalharmos os diferentes formatos de TLPs, consideremos as interfaces mencionadas no tópico anterior:

Completer Interface:

Esta interface recebe transações vindas do *host* pela camada física (lado esquerdo da Figura 3) e consiste em duas interfaces separadas, CQ (*Completer reQuest*) e CC (*Completer Completion*) que diferem em relação à direção da transação (*host* para aplicação do usuário ou *vice-versa*). A interface CQ faz a transferência de requisições para a aplicação do usuário e a interface CC faz a transferência da aplicação do usuário para o *host*. As duas interfaces operam independentemente, isto é, o *core PCIe* pode transferir novas requisições através da interface CQ enquanto recebe um pacote CC de uma requisição anterior [XIL15].

O *core PCIe* transfere cada TLP recebido do *host* através da interface CQ como um pacote independente. Cada pacote começa com o descritor (cabeçalho) e pode conter dados nos próximos *bytes*. O descritor é sempre de 16 *bytes* os quais são enviados nos primeiros 16 *bytes* do pacote [XIL15].

A Figura 4 apresenta o formato do descritor CQ do pacote.

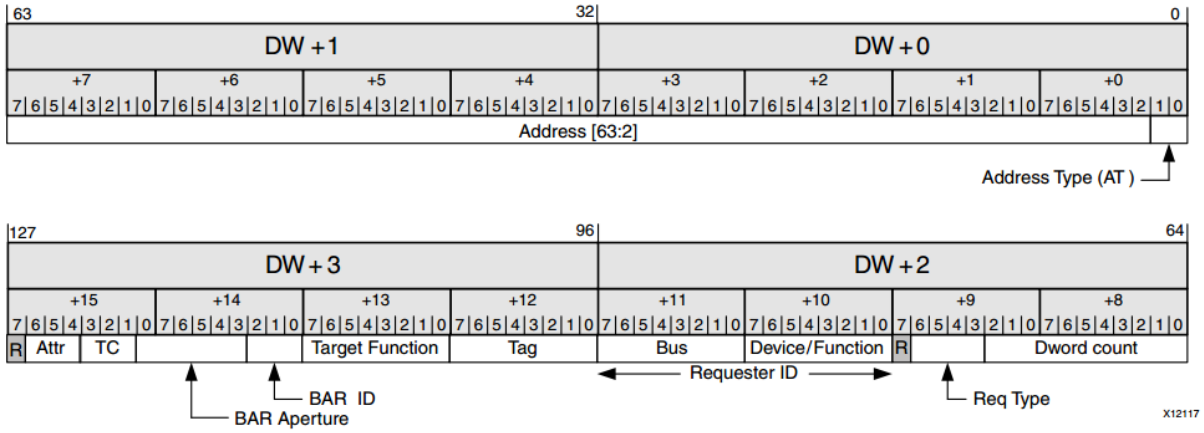

Figura 4 - Formado do descritor *Completer Request Descriptor* [XIL15].

O descritor tem um tamanho fixo de 16 *bytes* nas interfaces de requisição (*request*) e 12 *bytes* nas interfaces de conclusão (*completion*). O campo *Address Type* (AT) é definido para transações de memória e contém os bits extraídos do cabeçalho do TLP de requisição. O campo *Address* indica o endereço da memória instanciada na aplicação do usuário onde a requisição será enviada. *Dword count* indica o tamanho do bloco (em palavras de 32 bits - *Dwords*) a ser lido ou escrito. *Request Type* identifica o tipo da transação, e.g. requisição de leitura ou escrita na memória da aplicação do usuário. *Requester* ID é associado ao barramento onde está conectado o periférico (FPGA). O campo *Tag* é utilizado para transações *Non-Posted* como explicado no tópico 2.1 e pode ser ignorado para transações de escrita. BAR ID contém o número do BAR correspondente. BAR *Aperture* informa quantos bits são válidos no campo de endereço, e.g. indica os bits de endereço que devem ser lidos. *Transaction Class* (TC) é um campo que deve ser retornado no TLP de resposta, quando a requisição é do tipo *Non-Posted*. O campo *Attributes*

informa detalhes para a formação do TLP de resposta, por exemplo, se uma resposta com múltiplos TLPs deve ser enviada de forma ordenada. Os campos não mencionados não são utilizados neste trabalho. Da mesma forma acontece com a interface CC, no sentido contrário (aplicação do usuário para o *host*). O descritor CC é apresentado na Figura 5. O campo *Byte Count* é utilizado quando são necessários múltiplos TLPs para responder a uma requisição. Ele informa a quantidade de *bytes* que faltam ser transmitidos, incluindo ainda os *bytes* que estão sendo enviados. Por exemplo, ao se requisitar uma leitura de 4096 bytes, o primeiro TLP de resposta deve informar o valor de 4096 *bytes* remanescentes, considerando um tamanho máximo de 128 bytes de *payload* de resposta, o segundo TLP deve informar o valor de 4096 – 128 *bytes*, e assim por diante. *Poisoned Completion* informa se a resposta é valida ou deve ser descartada. *Completion Status* informa se a requisição foi atendida ou retorna um código de erro. Os demais campos seguem a nomenclatura da interface CQ.

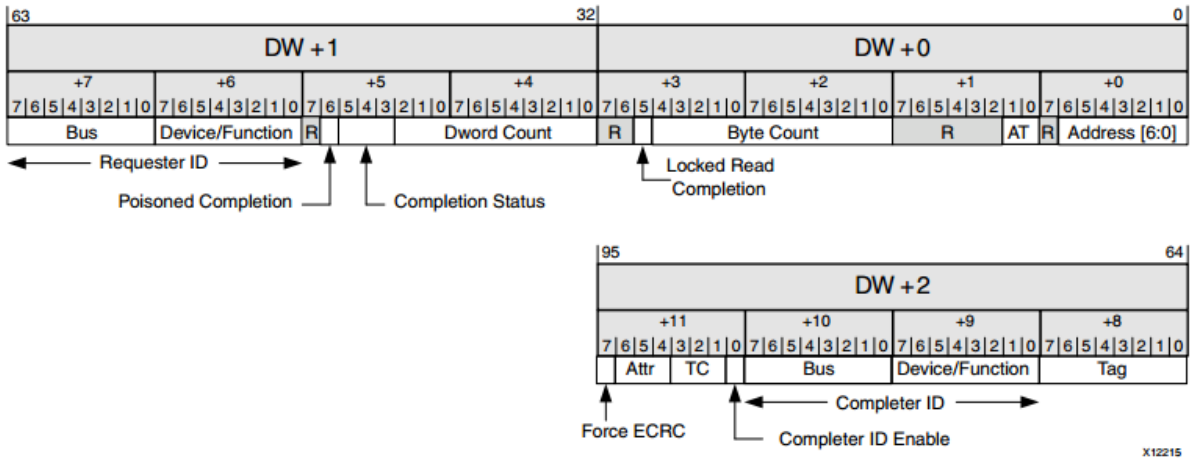

Figura 5 - Formato do descritor *Completer Completion Descriptor* [XIL15].

***Requester Interface*:**

Esta interface permite que a aplicação do usuário em *hardware* inicie transações como mestre, através da camada física para a memória do *host,* e.g. memória do *kernel Linux*, além de iniciar requisições de configuração.

As operações nesta interface são muito semelhantes às interfaces *Completer*, exceto pela inversão das direções tanto do *core PCIe* como da aplicação do usuário em *hardware*. Transações de escrita (*Posted Transactions*) são realizadas de forma única e indivisível enquanto as transações de leitura (*Non-posted Transactions*) são realizadas como transações separadas (*split transactions*). A interface *Requester* também consiste em duas interfaces separadas, RQ (*Requester reQuest*) e RC (*Requester Completion*), que da mesma forma, difere no sentido da transação. A interface RQ destina os dados da aplicação do usuário para o *core PCIe* e a interface RC é utilizada pelo *core* para enviar dados recebidos do *host* (para requisições de leitura) para a aplicação do usuário. Da mesma forma, as duas

interfaces operam de maneira independente, isto é, a aplicação do usuário pode transferir novas requisições através da interface RQ enquanto recebe respostas de requisições anteriores.

A Figura 6 apresenta o descritor RQ do pacote.

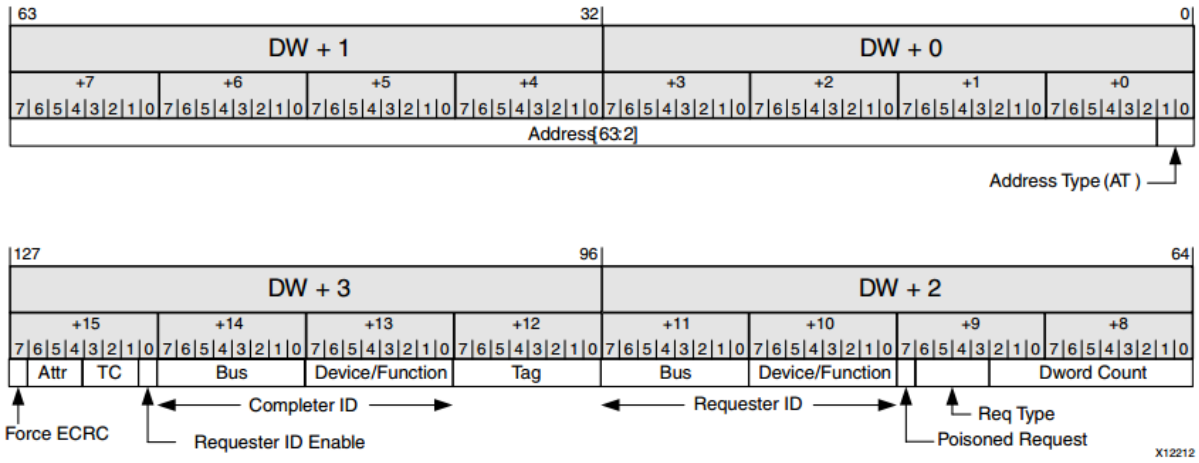

Figura 6 - Formato do descritor *Requester Request Descriptor* [XIL15].

No sentido contrário (dados recebidos do *host* para a aplicação do usuário). O descritor RC do pacote é ilustrado na Figura 7.

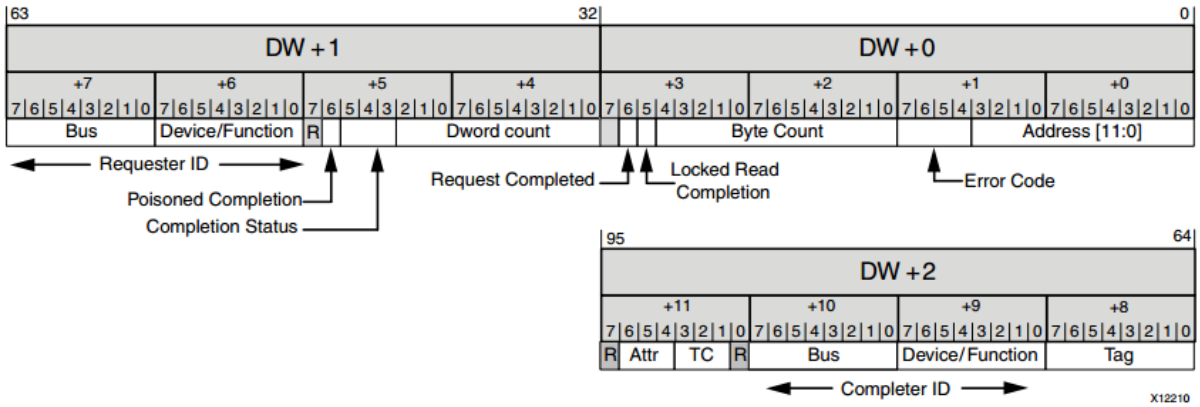

Figura 7 - Formato do descritor *Requester Completion Descriptor* [XIL15].

2.1.2 Opções de Alinhamento de Dados

Cada TLP é transferido como um pacote, conforme visto na seção 2.1.1 e definido na interface de protocolo *AMBA AXI4-Stream* [ARM16]. Por padrão, o alinhamento de dados do payload nas versões anteriores do *core PCIe* era feito por palavra. Para flexibilizar o projeto do usuário, o *core PCIe* da *Xilinx* possibilita que os dados do *payload* sejam alinhados de acordo com o endereço de memória da requisição, simplificando a transferência de blocos de memória. Recebendo o *payload* alinhado, a máquina que faz a leitura do TLP pode escrever os dados diretamente na linha da memória, sendo assim, não perde ciclos de relógio ajustando o alinhamento. No modo de alinhamento por palavra, a máquina de recebimento teria que guardar uma parte do *payload* que está sendo recebido para

ser escrito no ciclo de relógio seguinte, aumentando a complexidade do processo, visto que aumentaria a quantidade de ciclos necessarios para a escrita dos dados do TLP na memória. A Figura 8 apresenta as duas formas de alinhamento. Como podemos ver, no alinhamento por endereço, após o descritor, o *payload* não contém dados válidos.

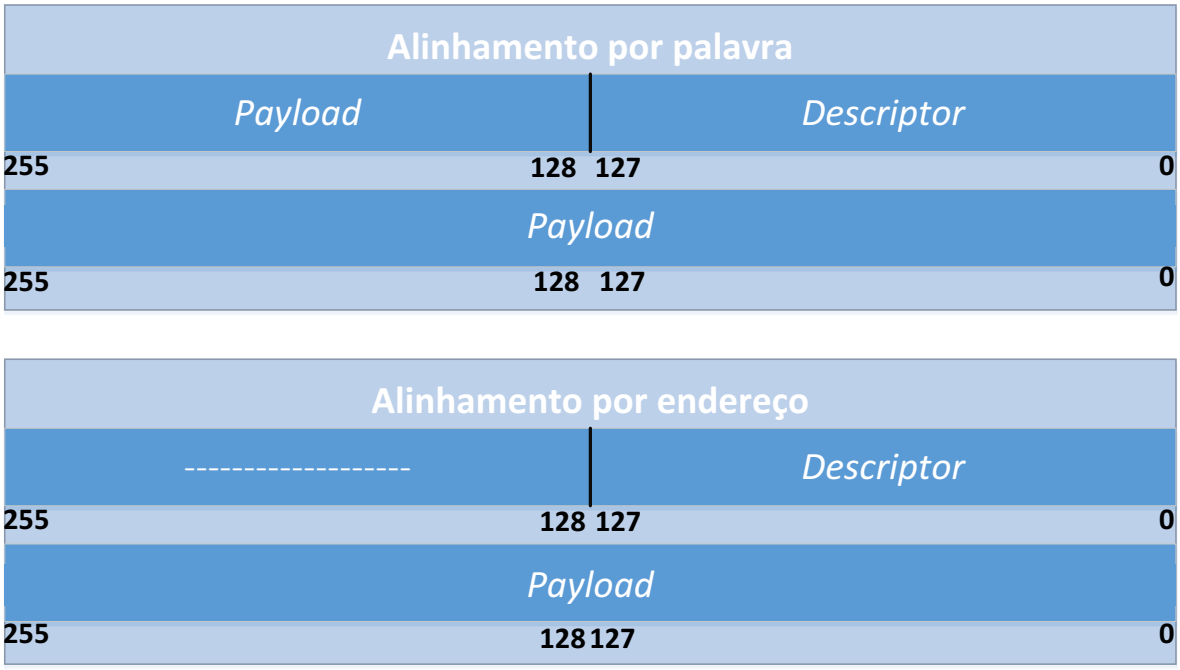

Figura 8 – Formas de alinhamento do *payload* do TLP.

Modos de alinhamento:

- *Dword-aligned* - neste modo, os *bytes* do descritor são imediatamente seguidos pelo *payload*, sempre que dados válidos estiverem presentes. Esta regra vale para todas as interfaces.
- *Address-aligned* – neste modo, os dados podem começar em qualquer posição. Para os dados transferidos entre o *core PCIe* e o *host,* pelos barramentos CQ e CC, a posição da primeira palavra do *payload* no TLP é definida como:

  $N = A / w$

  onde,

  - *A* é o endereço de memória (para mensagens e pedidos de configuração, o endereço é tomado como 0).
  - *w* é a largura da palavra em *bytes*. Qualquer intervalo entre a extremidade do descritor e o início do primeiro *byte* do *payload* é preenchido com *bytes* nulos.

Para dados transferidos a partir do *core PCIe* para a aplicação do usuário, através do barramento RC, o alinhamento da primeira palavra do *payload* é determinado com base no deslocamento do endereço informado na requisição

quando o bloco de dados está destinado à memória instanciada na aplicação do usuário. Para os dados transferidos a partir da aplicação do usuário para o *core PCIe* pelo barramento RQ, a aplicação deve comunicar explicitamente a posição do primeiro *byte* para o *core PCIe* quando o modo *address-aligned* está em uso [XIL15].

No modo *address-aligned*, o *payload* e o descritor não estão autorizados a se sobrepor. Isto é, o transmissor pode enviar uma nova mensagem para iniciar a transferência dos dados depois de ter transmitido o descritor [XIL15].

O modo *address-aligned* possibilita que a lógica responsável pela escrita e leitura em memória seja simplificada, pois assim o *payload* já vem alinhado de forma a ser escrito diretamente na memória. Isto otimiza o tempo necessário para completar as requisições e reduz o custo de área.

3 PROJETO DE REFERÊNCIA PARA USO DO *PCIE*

Para demonstrar o funcionamento do *core PCIe,* a *Xilinx* disponibiliza um projeto de referência descrito em VHDL, denominado *Programed Input Output* (PIO) [XIL15], para utilizarmos como aplicação. Os módulos deste projeto são apresentados na Figura 9 (destaques em amarelo são contribuições deste trabalho de conclusão). A implementação do projeto de referência da *Xilinx* instancia uma memória (*MEMORY*), originalmente de 8KB, e recebe requisições de leitura e escrita do *host* através do *core PCIe*. As requisições feitas pelo *host* são recebidas no barramento CQ que é processado no módulo RX. Quando a requisição é uma leitura de memória, o módulo TX é estimulado a enviar uma resposta usando o barramento CC.

O *projeto de referência* suporta transferências de palavras de memória de até 32 bits, o que torna a aplicação incapaz de utilizar em sua totalidade o barramento de 256 bits disponível no *core PCIe* configurado com 8 vias (*PCIe* x8).

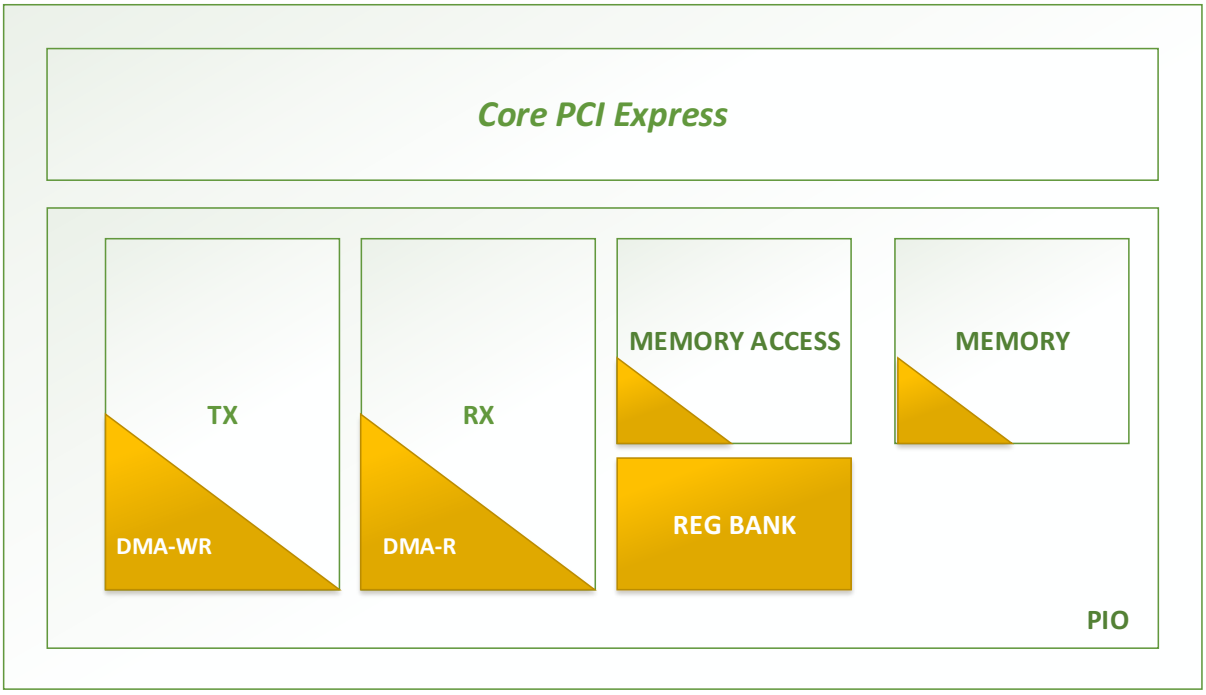

Figura 9 – Design de exemplo do *core PCI Express.*

3.1 Direct Memory Access – Módulos Write Read e Read

Para explorar a máxima capacidade de transferência do *core PCIe*, integramos ao projeto PIO uma lógica para acesso direto à memória (DMA – *Direct Memory Access*), representado em amarelo na Figura 9 como *DMA – Write Read*

(DMA-WR) e *DMA – Read (DMA-R)*. O DMA-WR envia requisições para o *host* pelo barramento RQ, enquanto o DMA-R recebe as respostas das requisições de leitura através do barramento RC.

Para controlar as requisições do DMA-WR, instanciamos um banco de registradores (*Register Bank – REG BANK*). Este banco contém endereços de escrita e leitura da memória física do *host,* quantidade de palavras a serem transferidas, *offset* para operações na memória interna do projeto e sinais de controle como *start, done* e *reset*. O *REG BANK* é acessado via escrita e leitura de memória através do *software* implementado no *host* e descrito na próxima seção (seção 3.2).

O bloco de acesso à memória *(MEMORY ACCESS)* controla as operações com a memória e foi modificado para endereçar o banco de registradores e a memória, reservando os primeiros 256 *bytes* endereçados ao banco de registradores e o restante para a memória. Para receber ou transmitir 256 bits de *payload* em apenas um ciclo de relógio, foi necessário modificar a memória para ter 8 palavras de largura, ou seja, 256 bits por linha.

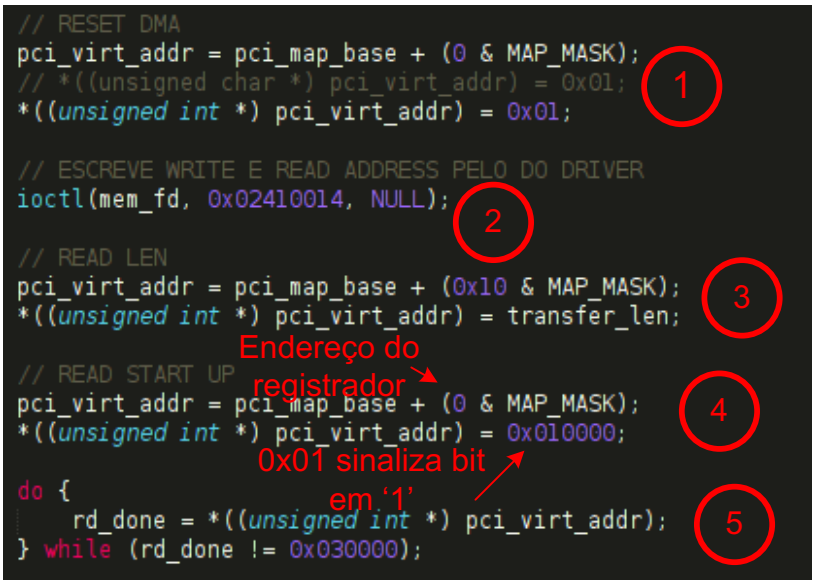

Figura 10 – Sequência para realizar transferência de memória do *host* para o FPGA.

Para transferirmos um bloco de memória do *host* para o FPGA, precisamos configurar os registradores (do banco de registradores) de forma com que o DMA faça uma requisição de leitura de memória ao *host*. Desta forma, o *host* retorna o bloco de memória requisitado ao FPGA. A Figura 10 apresenta a sequência de passos executados no *host* para configurar o DMA (no FPGA). O resultado destes passos no FPGA está apresentado na Figura 11. Iniciamos fazendo o *reset* dos registradores do DMA (1 na Figura 10 e Figura 11) e chamamos o *driver* para que este escreva o endereço de memória no banco de registradores (2). Ao retornar do *driver*, escrevemos a quantidade de palavras de 32 bits a serem requisitadas (3) e colocamos em nível lógico alto o sinal de *start* (4). Neste momento, a lógica do

DMA_WR envia a requisição com o endereço e a quantidade de palavras configurados nos passos anteriores. Após o *host* retornar a resposta, o DMA_R faz a escrita dos dados na memória do PIO e sinaliza com *done* (5), quando a contagem de palavras escritas for a mesma das palavras requisitadas.

A Figura 11 apresenta o comportamento dos sinais de acordo com a execução do *software* descrita acima.

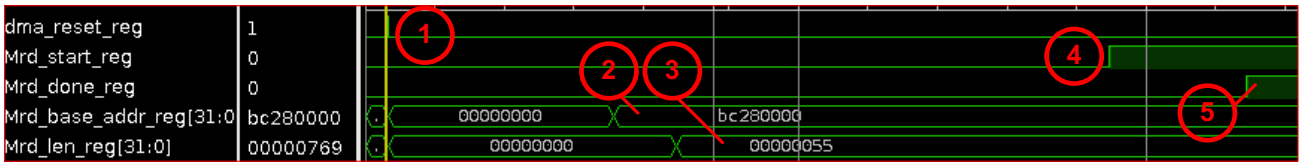

Figura 11 – Comportamento dos sinais da interface *PCIe,* como resultado da execução do *software* no *host.*

3.2 Software de Controle do DMA

Para realizar transferências de múltiplas palavras, é necessário que o PIO inicie transações através do DMA desenvolvido. Para validar as transferências de memória entre *host* e PIO via DMA, escrita e leitura em ambos os sentidos, foi desenvolvido um *software* em linguagem C. Este *software* lê um arquivo de texto, e o escreve em uma região de memória virtual do *kernel*.

Para que o DMA-WR faça requisições, é necessário configurar o endereço de memória física do *host* no banco de registradores. No sistema operacional *Linux,* o *software* que está na área de aplicações não tem acesso ao endereçamento da memória física. Por isto, utilizamos um driver fornecido pela *Xilinx* que reserva uma região de memória física e a mapeia para a memória virtual.

Na Figura 12 temos o *software* abrindo o driver *gboost* no descritor *mem_fd* (1) e fazendo o mapeamento passando o *driver* como argumento (2). O *driver* retornará um endereço virtual para a região de memória física.

```
// ABRE DRIVER
char devname[] = "/dev/gboost";
char* devfilename = devname;
mem_fd = open(devfilename, O_RDWR);

// MAPEIA AREA DE MEMORIA COMPARTILHADA COM FPGA
if (mem_fd < 0)
    PRINT_ERROR;
else {
    mem_map_base = mmap(NULL, MEM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd, 0);
    if (mem_fd == -1 || mem_map_base == MAP_FAILED)
        PRINT_ERROR;
    memset(mem_map_base, 0, MEM_SIZE);
}

memAddr = mem_map_base;
```

1

2

Figura 12 - *Software* abrindo *driver* e realizando chamada para obter endereço virtual.

Os motivos pelos quais o *software* faz chamadas ao *driver* são:

- Obter o endereço da memória virtual;
- Escrever nos registradores do PIO os endereços da memória física (2 na Figura 10).

Com estas chamadas, PIO e *software* têm conhecimento do endereço da mesma região de memória, podendo agora o utilizador prosseguir com a transferência de dados como demonstrado no final da seção anterior (seção 3.1).

É importante citar que para realizar transferências de memória entre o *host* e o PIO original (sem DMA), foi desenvolvido um *software* que faz transferências de apenas uma palavra de 32 bits, isto é, apenas as requisições aceitas pelo PIO original. Este software é utilizado como referência para o cálculo da vazão, a ser apresentado no Capítulo de resultados.

4 HEMPS-ECS

O projeto de referência é denominado HeMPS-ECS. O desenvolvimento deste projeto de referência não foi desenvolvido no escopo do presente Trabalho de Conclusão de Curso. Sua escolha como projeto de referência deve-se à necessidade de se realizar a interface do MPSoC com o mundo externo através de uma interface de alta velocidade.

O projeto HeMPS-ECS possui dois módulos principais. O primeiro corresponde ao sistema multiprocessado denominado HeMPS [WOS07][CAR09]. O segundo módulo corresponde à interface com o mundo externo, denominada ECS (*External Control Access*).

A HeMPS original possui como principal fonte de gerenciamento um elemento de processamento (PE) denominado mestre global. Com o desenvolvimento do módulo ECS, a gerência do sistema passa a ser distribuída, onde cada grupo de processadores (*cluster*) passa a ter um processador mestre local (LMP), os quais se comunicam com a interface ECS.

Dentre as responsabilidades do módulo ECS, cita-se:

- Comunicar-se com o mundo externo para receber novas aplicações;
- Armazenamento das aplicações, e.g. código objeto e grafo de tarefas;
- Comunicar-se com os elementos de processamento do MPSoC;
- Gerenciar os recursos do MPSoC.

A interface ECS oferece maior flexibilidade para interconexão com módulos externos, e.g. memória, e para injeção de aplicações (mundo externo) como podemos ver na Figura 13.

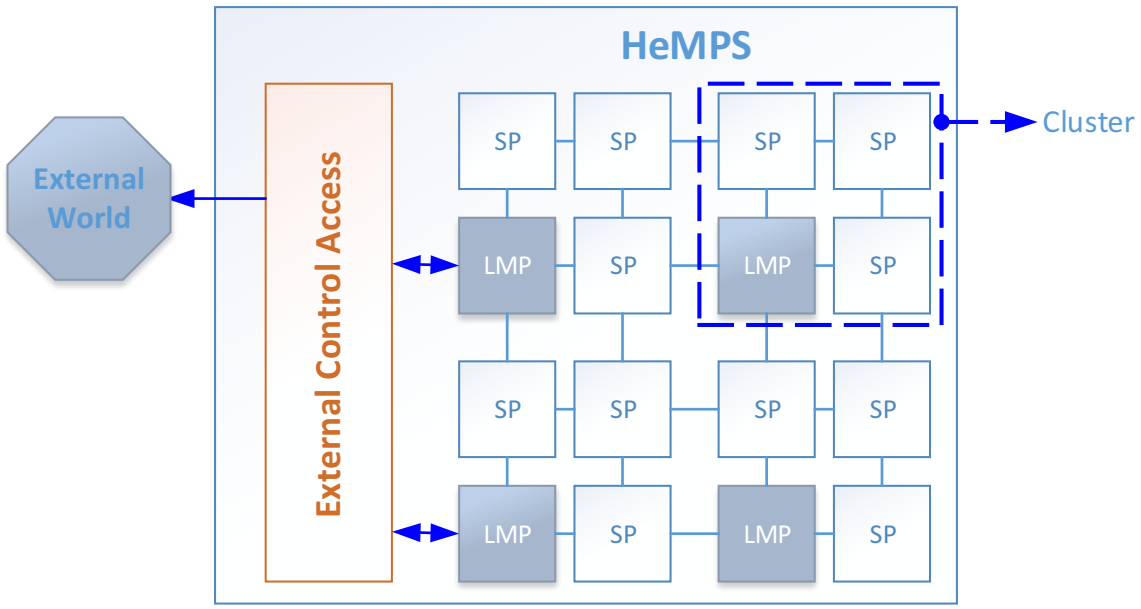

Figura 13 - Projeto de referência - HeMPS-ECS.

O módulo ECS, destacado na cor laranja, se comunica com as portas não utilizadas dos roteadores de borda da NoC para interconectar a rede de processadores a periféricos [ECS16].

Considerando qualquer topologia retangular, há quatro bordas disponíveis para se inserir módulos ECS, conforme ilustra a Figura 14. Desta forma, pode-se transformar a HeMPS em um sistema distribuído com diversos controladores e, então, explorar diversas técnicas de gerenciamento distribuído [ECS16].

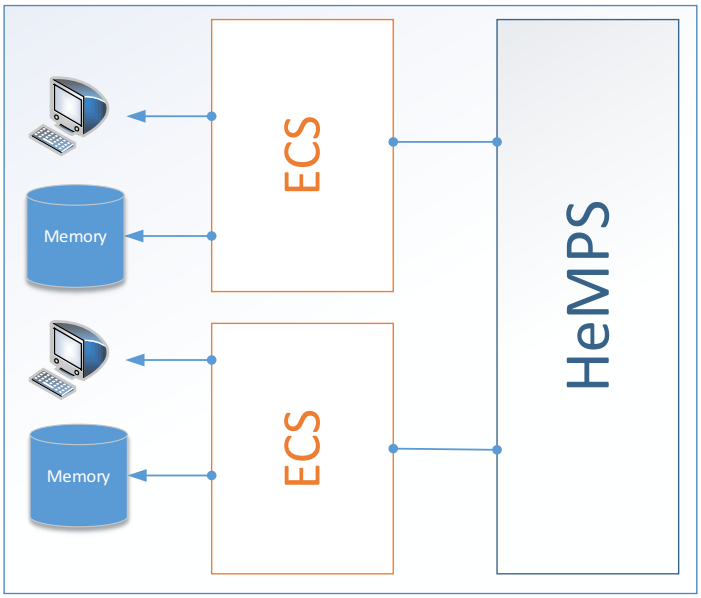

Figura 14 - HeMPS com 2 interconexões ECS.

4.1 A Plataforma HeMPS

Os MPSoCs usados em sistemas embarcados possuem uma maior capacidade de processamento por possuírem maior quantidade de elementos de processamento, podendo estes serem homogêneos (mesma arquitetura para todos os processadores) ou heterogêneos (sistema contendo processadores com arquitetura distintas). O modelo físico de comunicação intra-chip pode ser feito via NoC (*Network-on-chip*) ou barramento. Na HeMPS, utilizada neste trabalho, a conexão é feita via NoC, com topologia em malha 2D. A comunicação, assim, é feita por troca de mensagens. O processamento é homogêneo, ou seja, os elementos de processamentos são idênticos. A rede intra-chip utilizada é a NoC Hermes, também desenvolvida pelo grupo de pesquisa. O mapeamento é feito de forma dinâmica e a memória de cada elemento de processamento é dividida em páginas, podendo cada página ser destinada à alocação do *microkernel* ou tarefas de aplicações do usuário.

A plataforma HeMPS [WOS07][CAR09] é um MPSoC homogêneo baseado em NoC, com arquitetura mestre-escravo. A Figura 15 apresenta um exemplo da plataforma, utilizando uma NoC 4x4 com topologia malha. Os principais módulos que compõem o sistema são a NoC HERMES [MOR04] e os diversos elementos de

processamento (SP e LMP). O elemento de processamento contém o processador Plasma [PLA09] baseado na arquitetura MIPS, uma memória local, uma interface com a NoC e um módulo para acesso direto à memória (DMA – *Direct Memory Access*) [RAU09].

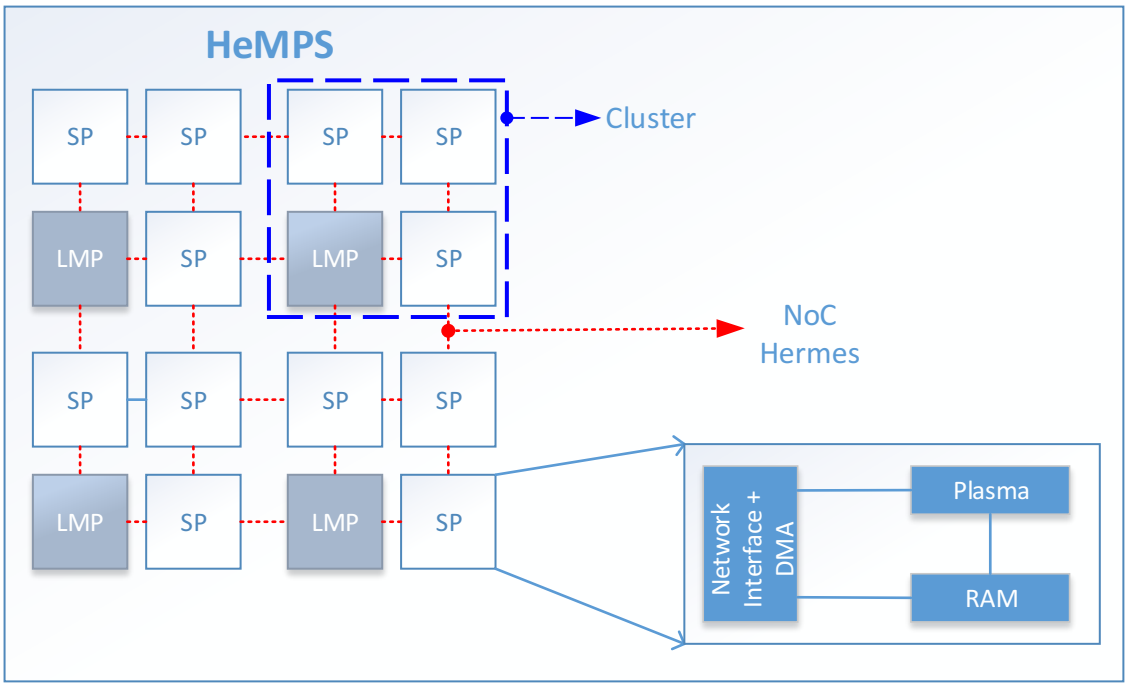

Figura 15 - Instância da plataforma HeMPS utilizando uma NoC 4x4.

As aplicações executadas na HeMPS são modeladas usando grafos de tarefas, e cada aplicação deve conter ao menos uma tarefa inicial. O repositório de aplicações é uma memória externa ao MPSoC que armazena todos os códigos objeto das tarefas necessárias para a execução de aplicações [RAU09]. A Figura 18 apresenta o posicionamento desta memória já no módulo ECS. O sistema original contém um processador mestre global (Plasma GMP), responsável por gerenciar os recursos do sistema, o qual foi substituído pelo módulo ECS, que agora possui o acesso ao repositório de tarefas. Quando a HeMPS inicia sua execução, o processador mestre (agora ECS) aloca a(s) tarefa(s) inicial(ais) nos processadores escravos (SP). Durante a execução, as tarefas são dinamicamente carregadas do repositório para os processadores escravos, conforme os pedidos de tarefas pelos mesmos. Além disso, recursos podem ficar disponíveis quando uma determinada tarefa terminar sua execução [RAU09].

Cada processador escravo possui um *microkernel* que suporta execução multitarefa e comunicação entre tarefas. O *microkernel* segmenta a memória em páginas, alocando a primeira delas para si e as demais para tarefas. Cada processador possui uma tabela de tarefas, com as informações das tarefas locais e remotas. Um escalonamento, implementado utilizando o algoritmo *Round Robin*, provê suporte multitarefa [RAU09].

De modo a otimizar o desempenho nos elementos de processamento, a arquitetura do módulo Plasma visa a separação entre computação e comunicação. A interface de rede e o módulo DMA são responsáveis por enviar e receber pacotes, enquanto o processador Plasma realiza a computação de tarefas e gerenciamento entre os demais módulos. A memória local é uma memória dupla porta, permitindo acesso simultâneo do DMA e processador, evitando assim hardware extra para elementos como exclusão mútua e roubo de ciclo [RAU09].

Modificações na *NoC Hermes* presente na HeMPS foram realizadas para habilitar a comunicação entre a NoC e o módulo ECS. Como o módulo ECS está conectado em interfaces de fronteira da NoC, que normalmente estão aterradas e não são utilizadas, foi necessário implementar um novo endereçamento para indicar aos roteadores se o pacote que esta transitando pela NoC será destinado a um PE ou ao novo módulo ECS. O novo endereçamento é baseado no endereçamento XY já existente na *NoC Hermes*, que define as coordenadas de localização do PE destino, porém, com a adição de um bit de sinalização que distingue se o pacote é relacionado ao módulo ECS ou não [ECS16]. Este novo endereçamento da HeMPS-ECS é apresentado na Figura 16.

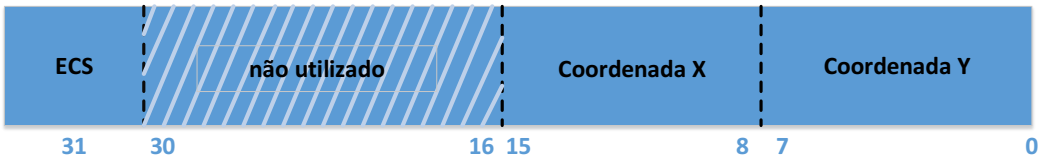

Figura 16 – Novo endereçamento da plataforma HeMPS.

Atualmente, a HeMPS utiliza apenas 16 bits do cabeçalho de endereçamento de pacote para se comunicar via NoC, sendo os 8 primeiros bits dedicado para a coordenada Y, e os 8 seguidos para a coordenada X. Desta forma, o bit de sinalização foi posicionado no bit mais significativo do cabeçalho, deixando ainda 15 bits não utilizados. Este novo bit de sinalização tem as seguintes interpretações [ECS16]:

- **ECS = '0':** pacote original da *NoC Hermes*. Seu endereçamento XY representa o PE destino do pacote.
- **ECS = '1':** pacote ECS. Seu endereçamento XY representa um roteador que possui conexões diretas com o módulo ECS. Este roteador detectará o valor do *bit* de sinalização e destinará o pacote para a interface com o módulo ECS.

Este comportamento implica que apenas os roteadores que fazem interface com o módulo ECS sejam capazes de interpretar o *bit* de sinalização ECS, evitando incompatibilidade com o roteamento original da NoC. Para isso, uma mudança na descrição VHDL fez-se necessária, com finalidade de identificar quais roteadores devem interpretar o *bit* de sinalização ECS. Na estrutura atual da HeMPS-ECS, apenas dois roteadores (00 e 02) possuem interface com o módulo ECS, como indicado na Figura 17, sendo estes os roteadores de borda configurados com o

parâmetro *OUTSIDE* = ‘1’ (parâmetro que permite acesso a portas de fronteiras). Para os roteadores restantes, *OUTSIDE* = ‘0’.

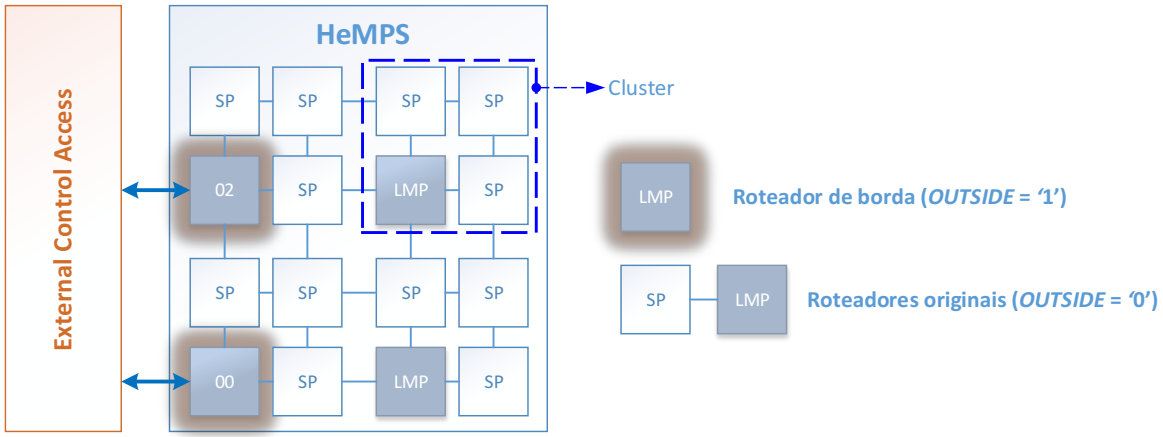

Figura 17 – Conexão dos roteadores da plataforma HeMPS com o módulo ECS.

4.2 Interface ECS

A interface ECS é composta por uma CPU (*Central Processing Unit*) com uma memória interna para o código objeto do controlador, um DMA-W (*Direct Memory Access - Write*) para interconectar o mundo externo (ou periféricos) à memória interna (utilizada para armazenar aplicações), um bloco para receber pedidos da HeMPS chamado de árbitro e outro para enviar pacotes para a HeMPS, chamado de DMA-R (*Direct Memory Access - Read*). A Figura 18 ilustra esta arquitetura com suas principais interconexões.

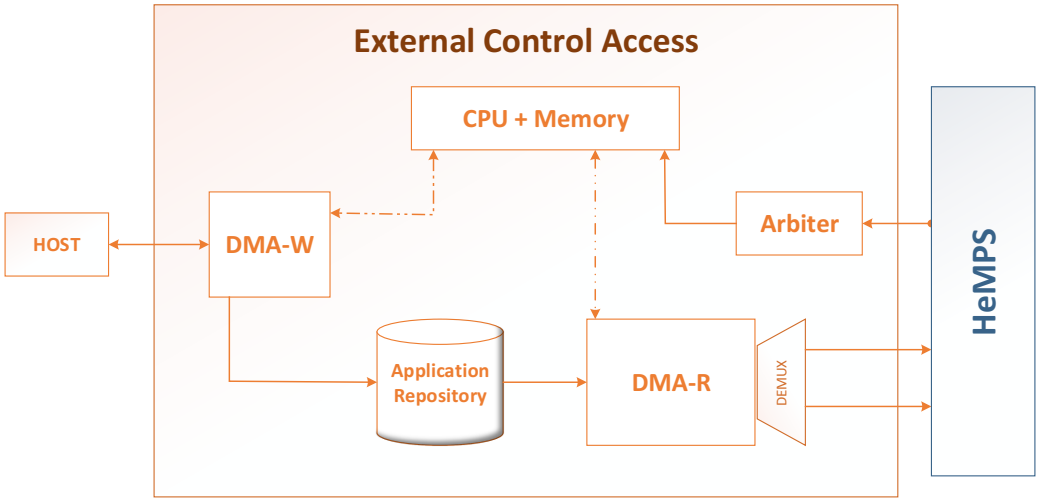

Figura 18 - Arquitetura interna do módulo ECS com suas principais interconexões.

O DMA-W é responsável por receber os dados do mundo externo (os quais podem ser uma aplicação que será executada na HeMPS) e armazená-los na memória (*Application Repository*). Para que isso aconteça, a CPU deve programar o endereço inicial que o DMA-W deve começar a escrever, desde que haja espaço disponível na memória. Além disso, é necessário que exista espaço disponível na

HeMPS, isto é, processadores com páginas livres para executar as tarefas da nova aplicação (a organização de memória na HeMPS é paginada, onde cada PE escravo pode executar simultaneamente *n* tarefas, onde *n* é definido em tempo de projeto). Desta maneira, antes de aceitar os dados do mundo externo, a CPU deve perguntar aos mestres locais a disponibilidade de páginas livres em processadores para a execução de uma nova aplicação. Se o pedido de recursos for menor ou igual à quantidade de recursos disponíveis, então o DMA-W é configurado e os dados do mundo externo são escritos no *Application Repository* [ECS16].

O bloco responsável por enviar pedidos para a HeMPS é o DMA-R. Este bloco contém registradores mapeados em memória que são escritos pela CPU para configurar o tipo de pacote que deve ser enviado. Em alguns tipos de pacotes, o DMA-R deve ler a aplicação na memória para extrair dados, tais como código objeto de uma tarefa que será alocada para construir o pacote. As respostas dos pedidos enviados por este módulo são recebidas e tratadas pelo árbitro. Este módulo também possui registradores mapeados em memória com a CPU, no entanto, ao contrário do DMA-R, o árbitro é quem escreve neles. Cada serviço que o árbitro recebe é tratado e, somente quando for necessário, a CPU é interrompida para tratar algum pedido da HeMPS [ECS16].

4.3 Protocolo ECS

O protocolo que descreve o funcionamento do módulo ECS é estimulado através de um *testbench*. A linguagem *systemC* foi utilizada no desenvolvimento desde módulo, principalmente por apresentar características e vantagens para teste e verificação de *hardware* [ECS16]. O *testbench* é composto por um processo gerador de *clock* e um processo que faz a injeção dos dados na forma de uma máquina de estados, a qual pode ser visualizada na Figura 19, sendo seus estados descritos na Tabela 2.

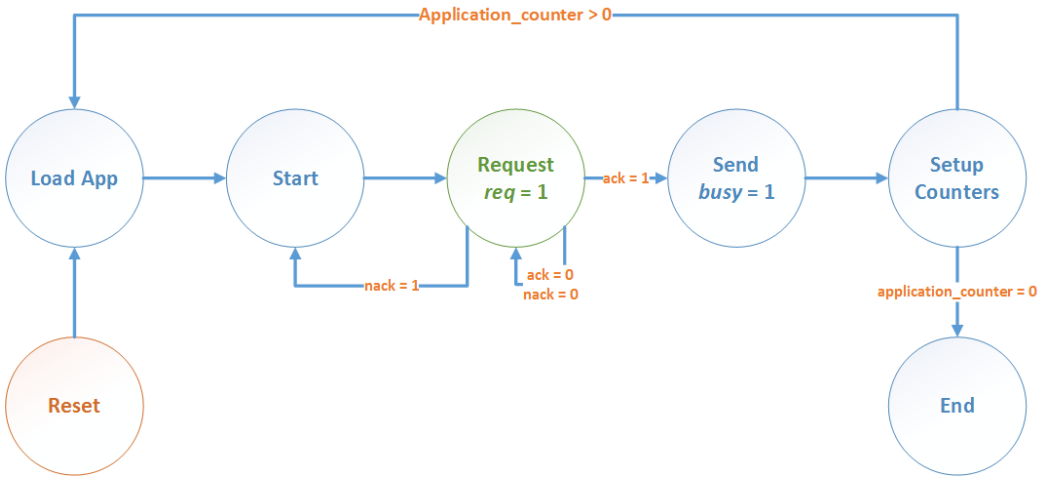

Figura 19 – Máquina de estados do protocolo de inicialização do módulo ECS [ECS16].

Tabela 2 – Descrição dos estados de configuração inicial do módulo ECS [ECS16].

Estado	Descrição
Reset	Inicializa todos os sinais internos do *software* e as entradas do módulo ECS com seus devidos valores.
Load App	É feito o carregamento do código objeto da aplicação para um vetor.
Start	Estado inicial, não há injeção de dados neste ponto.
Request	Realizada a ação de *request* para o módulo ECS, ou seja, o sinal *req* é levantado e o descritor da aplicação é injetado, esperando a análise de disponibilidade (*ack* ou *nack*) do módulo ECS para receber a aplicação.
Send	Injeção do código objeto da aplicação. Neste estado, o sinal *busy* é ativado indicando a operação do módulo DMA-W.
Setup Counters	Ao fim do envio de cada aplicação, os contadores devem ser atualizados, atualizando o número de aplicações no sistema.
End	Fim da execução.

Inicialmente um sinal de requisição (*req*) é colocado em nível lógico alto e logo em seguida um descritor é enviado para a CPU como podemos ver na Figura 20. Neste caso o campo 0xC302 informa 195 (0xC3) de tamanho de página e 2 (0x02) tarefas contidas na aplicação.

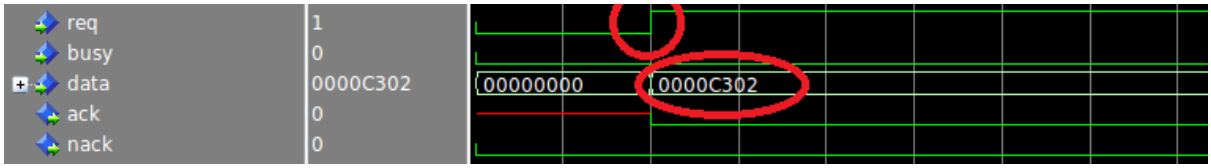

Figura 20 – Sinal de requisição e pacote de configuração.

Neste momento, a CPU verifica se há espaço na HeMPS para a inserção da(s) aplicação(ões) requisitada(s). Como podemos ver na Figura 21, 240 corresponde ao serviço *RESOURCE AVAILABLE*, considerando um MPSoC 4x4 com 4 *clusters*. O DMA-R envia 4 pedidos de disponibilidade de recursos, sendo cada pedido endereçado a um mestre local.

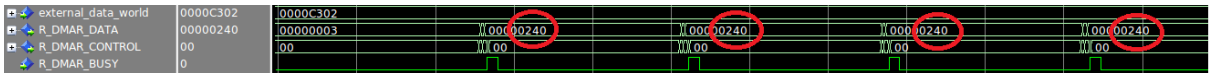

Figura 21 – Verificação de espaço disponível na HeMPS.

Caso a resposta seja positiva, a CPU retorna um sinal de *ack* para o mundo externo e então, o sinal de *busy* do DMA-W é colocado em nível lógico alto. A partir deste momento, a aplicação é lida pelo DMA-W e escrita na memória (*Application Repository* na Figura 18) localizada no módulo ECS, como podemos ver na Figura 22.

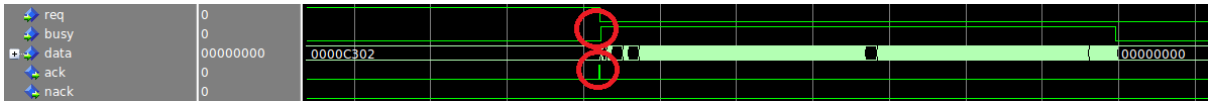

Figura 22 – Resposta *ack* e aplicação sendo inserida na memória.

No instante seguinte, o DMA-R envia para o PE definido (mestre local que possui mais recursos disponíveis em seu cluster) o serviço *New Appliation* (requisição de nova aplicação) e o árbitro, ao receber requisições de tarefas (*Task Request*) dos processadores da HeMPS, interrompe a CPU, a qual configura o DMA-R, que lê as tarefas no repositório de aplicações e as insere na HeMPS através do serviço *Task Allocation*, como podemos ver na Figura 23.

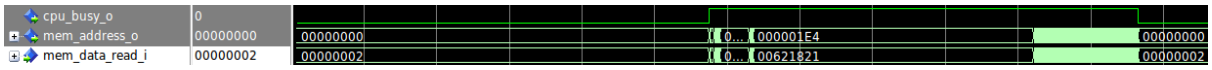

Figura 23 – Aplicação sendo inserida na HeMPS.

4.4 Módulos ECS

Nas próximas seções são analisados os módulos que compõe o *core* ECS.

4.4.1 Direct Memory Access – Read

O DMA-R possui interconexão com três outros blocos: o repositório de aplicações, a CPU e a HeMPS. O repositório é utilizado pelo DMA-R para extrair informações da aplicação, como o código objeto de uma tarefa ou o grafo da aplicação. A CPU comunica-se com o DMA-R através de registradores mapeados em memória para solicitar que uma tarefa seja encaminhada à HeMPS [ECS16].

Inicialmente o DMA-R está em um estado inativo esperando que a CPU faça alguma solicitação (inicialização de *cluster,* alocação de tarefa, requisição de nova aplicação). Uma vez que todos os parâmetros sejam passados para o DMA-R, a CPU deve ativar o sinal de *start* [ECS16].

O DMA-R é responsável pela parte ativa da comunicação com a HeMPS, i.e. envio de pacotes para a rede. Os serviços suportados pelo DMA-R são [ECS16]:

- ***Initialize Cluster***: serviço para inicializar o mestre local de um determinado *cluster*;
- ***New Application***: serviço para encaminhar o descritor (grafo) de uma aplicação para um mestre local de um determinado *cluster*;
- ***Task Allocation***: serviço para encaminhar o código objeto de uma tarefa para um determinado elemento de processamento.

O DMA-R comunica-se com a CPU através de um conjunto de registradores mapeados em memória. Estes registradores têm suas escritas controladas por *flags* (e.g. cpu_maddr_i, cpu_daddr_i e demais na Figura 24) juntamente com o barramento de dados (cpu_cfg_**data**_i). Toda vez que uma operação de escrita for feita em um dos registradores, uma *flag* é sinalizada, indicando que há um dado disponível na entrada do DMA-R. A *flag* identifica o registrador correspondente a ser escrito o dado no barramento de *cpu_cfg_data_i* (CPU *Configuration Data*).

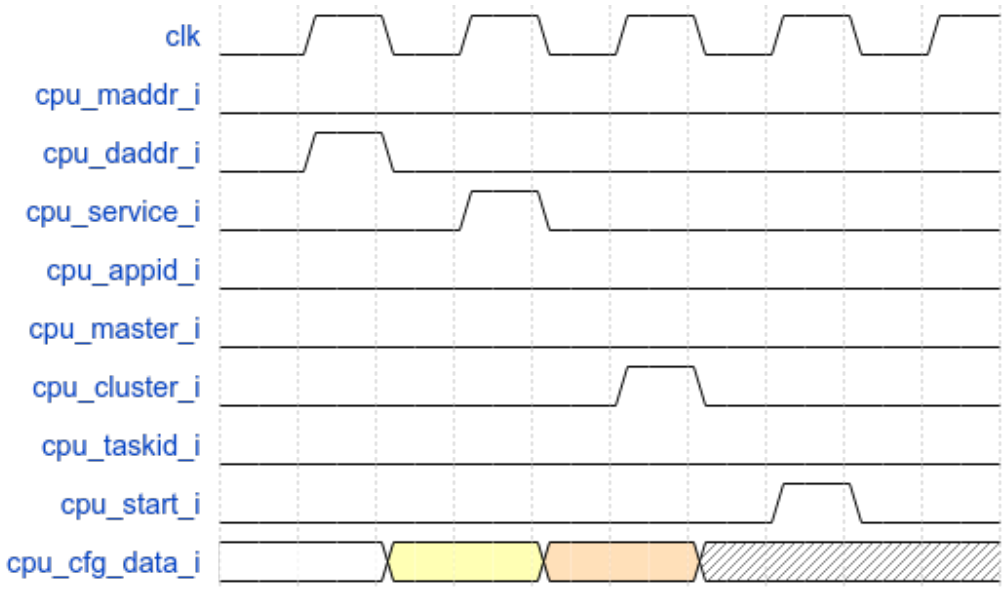

Figura 24 – Exemplo de comportamento do DMA-R com a CPU [ECS16].

4.4.2 Direct Memory Access – Write

O DMA-W é responsável por receber o conteúdo das aplicações vindas do mundo externo e transmitir essas informações para a memória, juntamente com a devida posição de memória e um sinal habilitando sua escrita.

Quando uma nova aplicação deve ser inserida na memória, a CPU, que previamente recebeu informações básicas sobre essa aplicação (tamanho de página e número de tarefas) programa o DMA-W através de seus registradores mapeados em memória. Nesta programação o DMA-W recebe informações relativas ao tamanho da aplicação e a posição inicial em que a aplicação deve ser armazenada. Quando a CPU aceita a aplicação (após configurar o DMA-W) e emite um sinal de *ack* ao mundo externo, informando que a HeMPS, em seu estado atual, é capaz de executar a aplicação, o DMA-W também recebe este sinal e inicia seu processamento [ECS16].

A Figura 25 apresenta o comportamento da interface do DMA-W com a CPU. Este módulo comunica-se com a CPU através de dois registradores mapeados em memória (responsáveis por armazenar *mem_addr* e *app_size*). De forma similar ao DMA-R, sempre que algum dado for escrito em algum destes registradores, uma *flag* (*cpu_mem_addr* e *cpu_app_size*, respectivamente) é levantada indicando que a entrada *cpu_data* contém um dado válido.

Após escrever os dados de *mem_addr* e *app_size*, a CPU levanta o sinal *ack* para indicar ao mundo externo que a HeMPS pode receber a aplicação. Assim, o DMA-W também recebe esse sinal e sabe que poderá enviar os dados diretamente para a memória, sempre que o sinal *busy* (apresentado no estado *Send* da Tabela 2) estiver em nível lógico baixo.

Estas escritas na memória prosseguem até o DMA-W atingir a última posição de memória a ser escrita. No momento da escrita, ele levantará uma *flag* (*cpu_irq*) para indicar para a CPU que a aplicação está totalmente carregada na memória. Assim, o DMA-W está pronto para receber uma nova aplicação e a CPU pronta para transmitir ao MPSoC HeMPS a aplicação que foi escrita na memória (repositório).

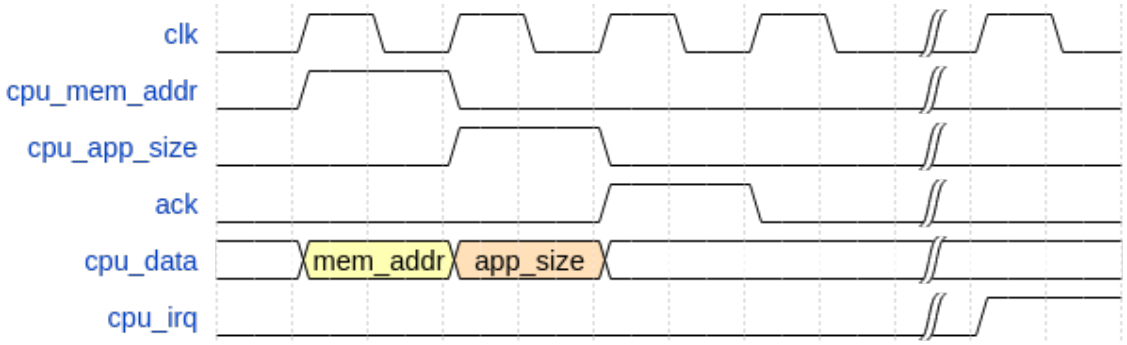

Figura 25 – Exemplo de comportamento do DMA-W [ECS16].

4.4.3 Árbitro

O árbitro possui como funcionalidade principal a comunicação entre HeMPS e ECS. Sendo responsável por tratar os pacotes recebidos pela HeMPS, o arbitro realiza interrupções na CPU do modulo ECS, requisitando a inserção de tarefas *(New Task*), informando recursos disponíveis (*Resource Available*) nos *clusters* da HeMPS ou ainda informando as aplicações que foram finalizadas (*App Terminated*). Para que isto seja possível, 3 conexões se fazem necessárias: com os roteadores de borda da HeMPS; com a CPU, através dos sinais *interrupt* e *grant_interrupt* e com 4 registradores de 32 bits que passam informações específicas para cada serviço [ECS16].

No início da operação do sistema, o árbitro fica alternadamente fornecendo crédito aos roteadores da HeMPs conectados a ele (sinais *credit_out_0* e *credit_out_1*) e verificando se o roteador correspondente requisita o envio de pacotes. Quando o roteador faz uma requisição o árbitro recebe o pacote, interpreta o serviço e escreve nos registradores de comunicação com a CPU, chamando-a através de uma interrupção [ECS16].

Os serviços suportados pelo árbitro são:

- ***Resource Available:*** serviço em que o árbitro recebe todas as respostas de pedido de informações de recursos disponíveis nos *clusters* (informação enviada por cada um dos mestres locais) e, após comparar e encontrar o *cluster* com maior disponibilidade de recursos e informa a CPU, através dos registradores mapeados em memória, o *cluster* (ID do mestre local) com maior disponibilidade;
- ***New Task***: serviço em que o mestre local ou o PE encaminha solicitação de nova tarefa. Após processar o pacote o árbitro encaminha o pedido para CPU através dos registradores mapeados em memória;

- ***App Terminated***: serviço para informar a CPU (através dos registradores mapeados em memória) que determinada aplicação terminou a sua execução.

4.4.4 Mestre Global ECS (CPU)

Na HeMPS-ECS o mestre global foi substituído por um mestre local na estrutura interna da HeMPS. Porém, ainda é necessário uma unidade de processamento dedicada para habilitar a iteração entre o mundo externo e a HeMPS. Desta forma, o módulo ECS possui um microprocessador PLASMA para atuar como mestre global. Este mestre global possui um *kernel* diferente dos PEs internos da HeMPS para atender os seguintes requisitos [ECS16]:

- Atender inserção de aplicações do mundo externo, avaliando se a plataforma suporta a aplicação apresentada;
- Programar o módulo DMA_W para carregar aplicações na memória;
- Programar o módulo DMA_R para realizar três serviços da HeMPS:
  - *Initialize Cluster*;
  - *New App*;
  - *Resource Available*;
- Atender interrupções do Árbitro referente à três serviços da HeMPS:
  - *Resource Available;*
  - *New Task;*
  - *Task Terminated.*

Após sair do estado de *RESET*, o mestre global inicialmente programará o DMA-R para inicializar todos os clusters da HeMPS utilizando o serviço *Initialize Cluster*. Como a HeMPS-ECS utilizada neste trabalho possui 4 *clusters*, o DMA_R é programado 4 vezes, programando cada *cluster* individualmente. Finalizado esta etapa de inicialização, a CPU habilita as interrupções do Árbitro, DMA-W e do mundo externo [ECS16].

Caso a interrupção do mundo externo seja disparada, a CPU processará três informações da aplicação: número de tarefas, tamanho de página e CRC. Estas informações estarão apresentadas em uma palavra de 32 bits como na Figura 26. A atual implementação não utiliza o campo CRC, o qual em versões futuras conterá o CRC correspondente ao CRC de todas as tarefas da aplicação.

Figura 26 – Pacote de configuração.

Os 7 bits menos significativos representam o número de tarefas, e os 20 bits seguintes representam o tamanho de página. O número de tarefas da aplicação corresponde ao número de páginas de memória necessárias para executar a aplicação. Assim, CPU solicita a quantidade de recursos disponíveis em cada cluster programando o DMA-R para enviar pedidos de *Resource Available*. Ao fim da configuração do DMA-R, a CPU desabilita a interrupção do mundo externo, evitando que o mundo externo tente injetar outra aplicação enquanto a aplicação anterior ainda não foi tratada [ECS16].

Caso ocorra uma interrupção do DMA_W, o mesmo informa que já carregou toda a aplicação na memória. Com a aplicação carregada, a CPU pode programar o DMA-R para enviar a aplicação para a HeMPS (*New App*). Após a configuração do DMA-R, a CPU reabilita a interrupção do mundo externo para inserção de uma nova aplicação [ECS16].

5 INTEGRAÇÃO PCIE – HEMPS-ECS

A Figura 27 ilustra o diagrama de blocos dos dois projetos de referência, e o diagrama de blocos do projeto final. A integração dos dois projetos de referência em um módulo *TOP* necessitou a definição dos domínios de relógio. Os módulos *PCIe* e PIO operam em 250 MHz, dada que esta é a frequência do barramento *PCIe*. O módulo HeMPS-ECS opera em 100 MHz devido a caminhos críticos no processador Plasma. Observar que dois módulos originalmente contidos na HeMPS-ECS foram movidos para o *TOP*: DMA-W e *Application Repository.* O DMA-W opera na frequência de 250 MHz permitindo a escrita no repositório na frequência do *PCIe*. É importante destacar que sincronizadores (*two-flops*) foram empregados entre a CPU do ECS com o DMA-W. O *Application Repository* é uma fila bi-síncrona, escrita na frequência de 250 MHz e lida na frequência de 100 MHz. O módulo DEBUG FIFO recebe as mensagens da CPU do ECS, transmitindo-as para o *host*. Este módulo também é uma fila bi-síncrona.

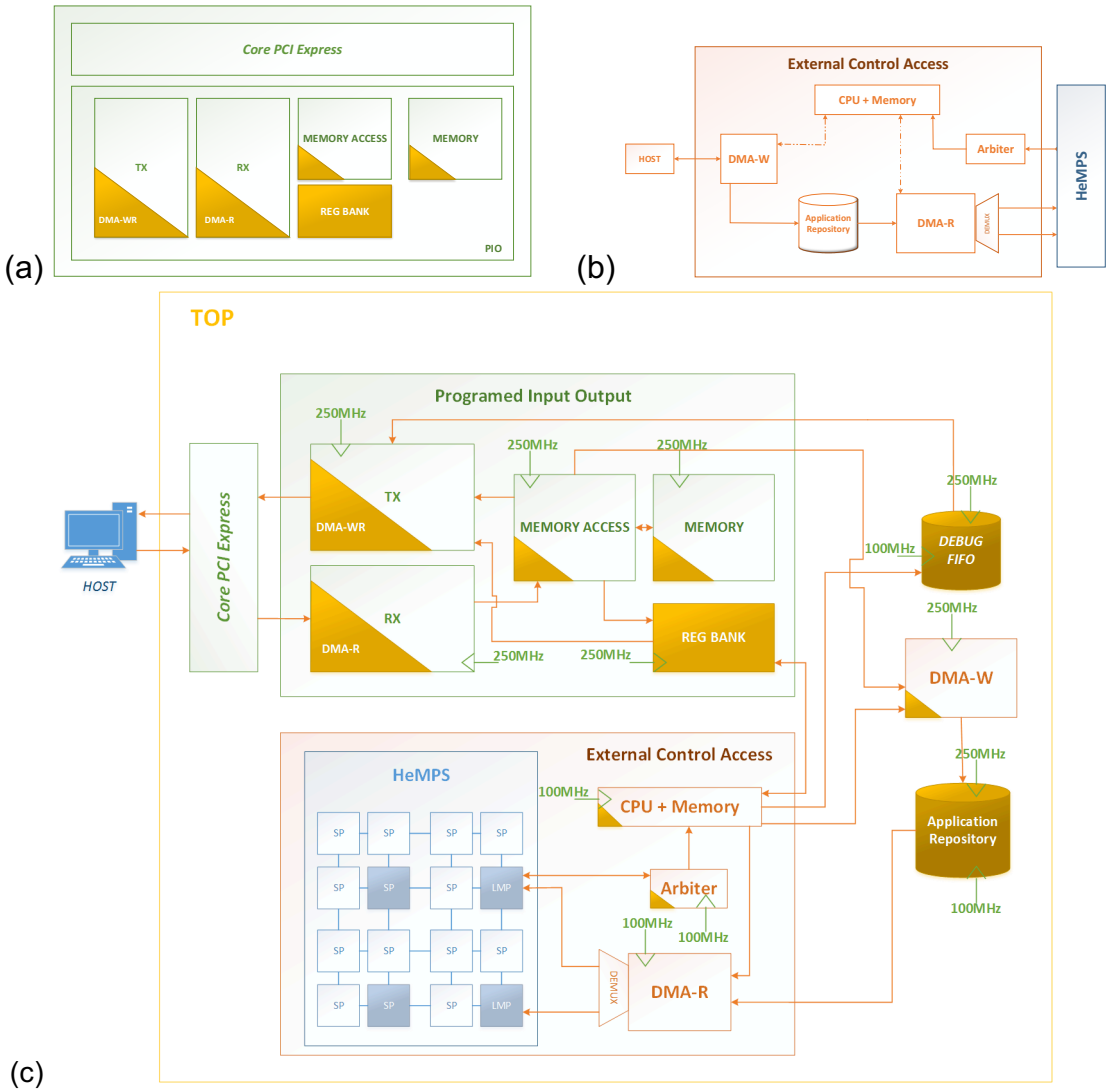

Figura 27 – (a) Projeto de referência PCIe; (b) projeto de referência HeMPS-ECS; (c) diagrama de blocos do projeto final.

Observar que há dois módulos de memória no projeto: *MEMORY* (no PIO) e o *Application Repository* (no *TOP),* ambos com 32 KB, implementados com BRAMs (*Block Random Access Memory*). O uso de BRAMs auxilia no atendimento às restrições de temporização. No projeto original observou-se que a síntese utilizava LUT RAMs (*Look up Table* RAMs) no lugar de BRAMs, gerando violações de temporização. Ao longo do projeto, estas memórias foram reescritas com BRAMs.

O fluxo de recepção de dados do *host* para o FPGA envolve os módulos: *core PCIe*, RX, *Memory Access* e *MEMORY*. Uma vez autorizada a entrada da aplicação pelo módulo ECS, o dado é transferido da *MEMORY* para o *Application Repository,* via DMA-W. Para que este processo de transferência fosse possível, no módulo *MEMORY ACCESS* foi adicionado um processo para realizar estas ações.

5.1 Processo de integração e validação do sistema

Com o objetivo de utilizar o *core PCIe* para injetar aplicações em um MPSoC, adaptamos o projeto do PIO para que este realize o protocolo utilizado no ECS. Para isto, os sinais de controle da CPU do ECS (1 - *req*, 3 - *busy* e 2 - *ack*) foram ligados ao *REG BANK* conforme Figura 28, assim, podem ser configurados pelo *software* operando no *host*. Desta forma, a integração entre PIO e ECS começa ao injetarmos a aplicação via *software* no *core PCIe* conforme apresentado anteriormente na Figura 10 do Capítulo 3.

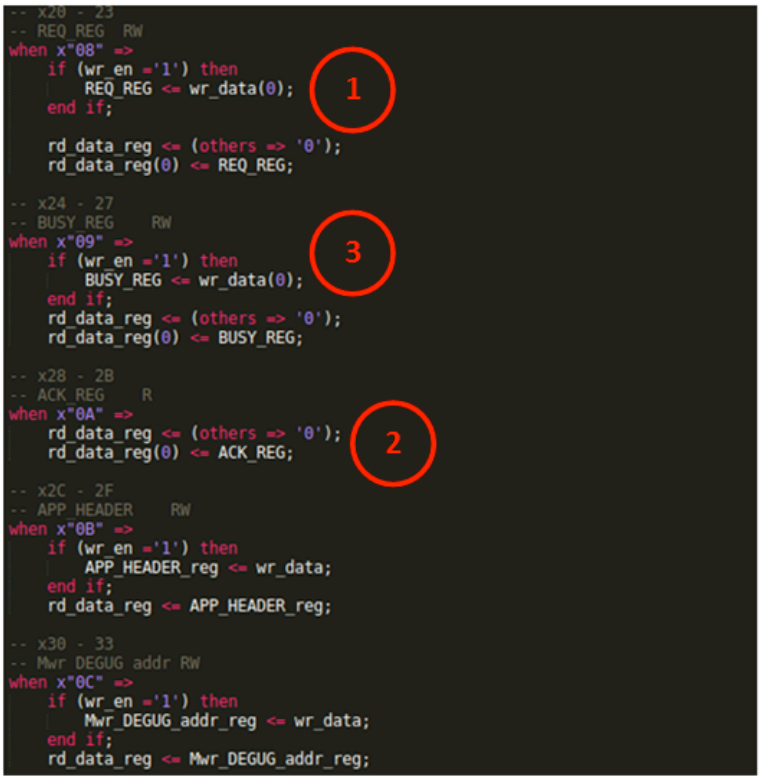

Figura 28 – Instanciação dos registradores utilizados pelo protocolo ECS (módulo *REG BANK*).

Após a inserção da aplicação no *core PCIe,* o sinal *req* é colocado em nível lógico alto (1 na Figura 29 e Figura 30) e é feito *pooling* no registrador *ack* (2 na Figura 29 – não foi possível amostrar o sinal *ack* via *chipscope* na Figura 30). Com a confirmação da CPU do ECS, levantamos o sinal *busy* (3), que estimula o *MEMORY ACCESS* a transferir a aplicação para o *Application Repository* via DMA_W. Em seguida, O ECS prossegue o protocolo descrito no Capítulo 4, seção 4.3.

```
// REQ_REG = 1
pci_virt_addr = pci_map_base + (32 & MAP_MASK);
*((unsigned char *) pci_virt_addr) = 1;

pci_virt_addr = pci_map_base + (40 & MAP_MASK);
do {
    // ACK_REG
    ack_reg = *((unsigned char *) pci_virt_addr);
} while (ack_reg != 1);

// BUSY_REG = 1
pci_virt_addr = pci_map_base + (36 & MAP_MASK);
*((unsigned char *) pci_virt_addr) = 1;
```

1
2
3

Figura 29 – Código do *software* (executando no *host*) responsável por realizar o protocolo ECS, através de escritas no *REG BANK*.

Após o DMA-W terminar a transferência para o repositório de aplicações no ECS, o mesmo interrompe a CPU informando o término da transferência (4 na Figura 30).

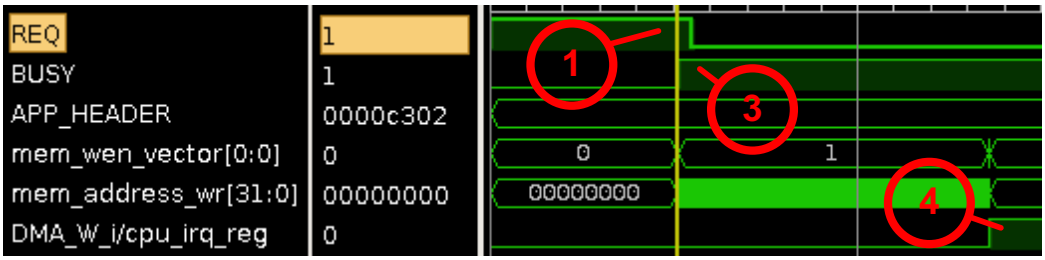

Figura 30 – Comportamento do *hardware* ao realizar o protocolo de inserção de aplicações no FPGA (tela obtida do chipscope).

Através de modificações no *kernel* dos processadores escravos da HeMPS, adicionamos um serviço chamado *ECHO SLAVE*¸ que é um pacote de mensagem de depuração encaminhado aos roteadores de fronteira da NoC, para sabermos quando um processador realiza um *ECHO* ao executar uma tarefa. Este pacote é configurado conforme a Figura 31.

Este pacote é enviado ao ECS, chegando primeiramente ao árbitro, que verifica o serviço *ECHO SLAVE* (representado na Figura 32) e encaminha a mensagem à CPU através de uma interrupção.

```
case ECHO:

    puts(itoa(MemoryRead(TICK_COUNTER))); puts("_");
    puts(itoa(net_address>>8));puts("x");puts(itoa(net_address&0xFF)); puts("_");
    puts(itoa(current->id >> 8)); puts("_");
    puts(itoa(current->id & 0xFF)); puts("_");
    puts((char *)((current->offset) | (unsigned int) arg0));
    puts("\n");

    ack_p = get_service_header_slot();
    // zero the X-axis of the destination and '1' to the most significant bit (put outside of the NoC)
    ack_p->header = (0xffff00ff & (p.header | 0x80000000));

    // reply packet uses the same service code
    ack_p->service = ECHO_SLAVE;

    // get the XY coordinates of this local master
    ack_p->resource_available = net_address;

    send_packet(ack_p, 0, 0);

    break;
```

0x8 = 1 no bit mais significativo, endereçando aos roteadores de borda

Figura 31 - Configuração do pacote com serviço *ECHO SLAVE*, executado no *kernel* dos PEs escravos.

```
when TEST_SERVICE =>
    flit_count <= x"0000000B";
    if (flit_service = NEW_TASK) then
        PE_manager <= SEND_VALUE_0;
    elsif(flit_service = APP_TERMINATED) then
        PE_manager <= RECEIVE_PAYLOAD;
    elsif (flit_service = RESOURCE_AVAILABLE) then
        PE_manager <= CALCULATE_RESOURCES;
    elsif (flit_service = ECHO_SLAVE) then
        PE_manager <= SEND_VALUE_0;
    else
        PE_manager <= SERVICE_UNKNOWN;
    end if;
```

Figura 32 – Verificação feita pelo árbitro ao receber o pacote da HeMPS contendo o serviço *ECHO SLAVE.*

A Figura 33 ilustra a simulação, onde estes pacotes oriundos da HeMPS são recebidos no árbitro, e transmitidos para a CPU-ECS. Podemos verificar no barramento de dados *data_in* a chegada do serviço *ECHO SLAVE* (0x245) e a saída do mesmo pacote no barramento *data_out.*

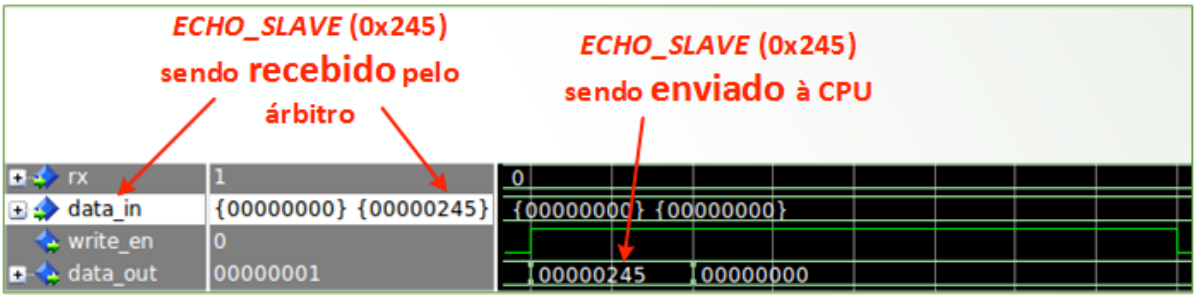

Figura 33 – Pacote contendo o serviço *ECHO_SLAVE* sendo recebido no árbitro e enviado à CPU-ECS.

Por sua vez, a CPU-ECS, interrompida pelo árbitro, recebe o pacote contendo o serviço *ECHO SLAVE,* como podemos verificar na simulação apresentada na Figura 34 (cada palavra da UART contém 4 caracteres ASCII).

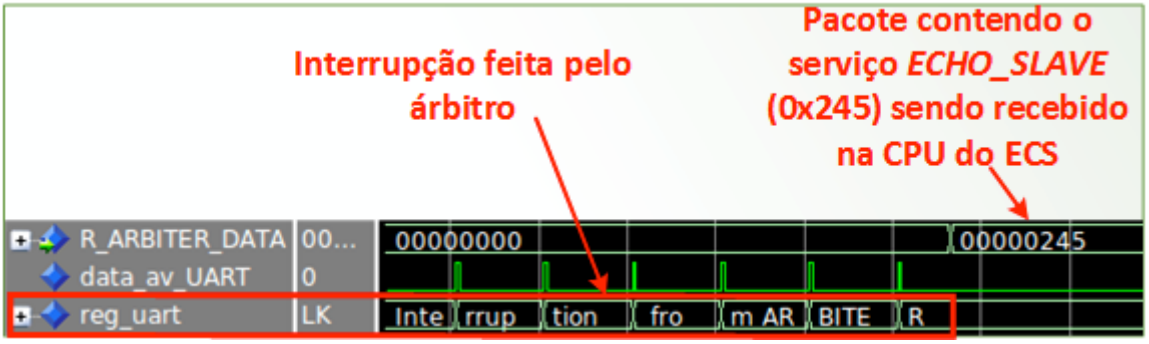

Figura 34 – Pacote contendo o serviço *ECHO_SLAVE* sendo recebido na CPU-ECS por interrupção do árbitro.

Para que os pacotes de depuração sejam mostrados ao usuário (no *host*), é necessário que estes sejam enviados através do PIO e *core PCIe.* Para isto, é descrito no *kernel* da CPU-ECS uma função chamada PUTS. Esta função tem por objetivo realizar a escrita no barramento de *debug*. O barramento de *debug* está conectado ao módulo DEBUG FIFO desenvolvido e apresentado na Figura 27. A porta de leitura deste módulo está ligada ao TX (também apresentado na Figura 27) do PIO. O módulo TX faz a transferência dos pacotes de depuração para a memória física do *host*, como apresentado na Figura 35, onde o módulo TX monta o cabeçalho da mensagem de *debug* no pacote TLP.

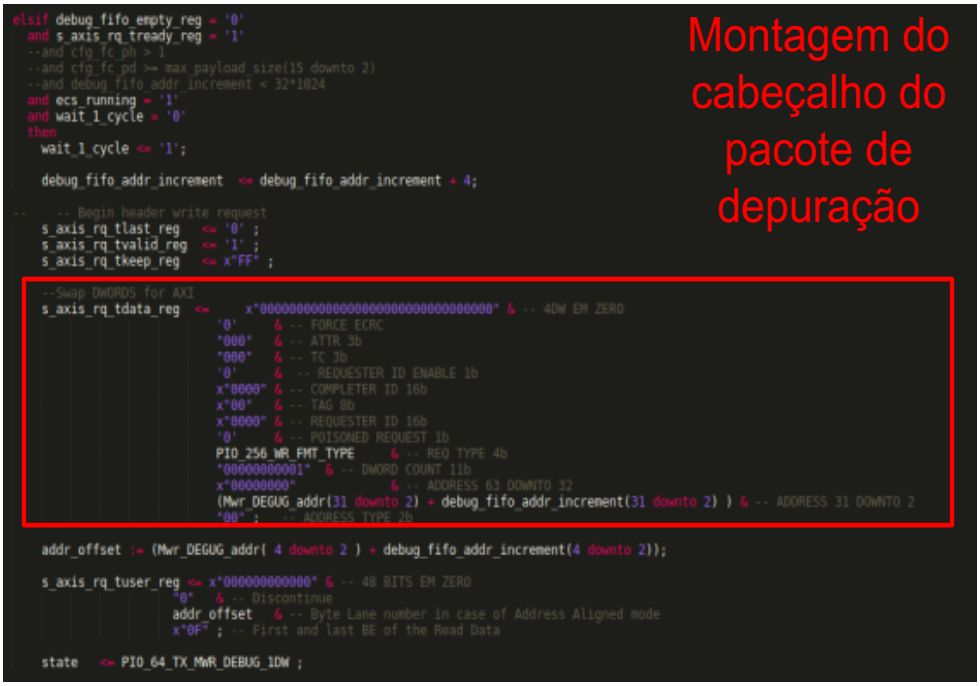

Figura 35 – Montagem do descritor da mensagem (256 bits) com dados de depuração (módulo TX).

A Figura 36 apresenta a execução, verificada por *chipscope,* de um pacote de depuração informando a inicialização dos *clusters* pela CPU e enviado ao módulo DEBUG FIFO, para posterior envio ao *host* através do módulo TX do PIO.

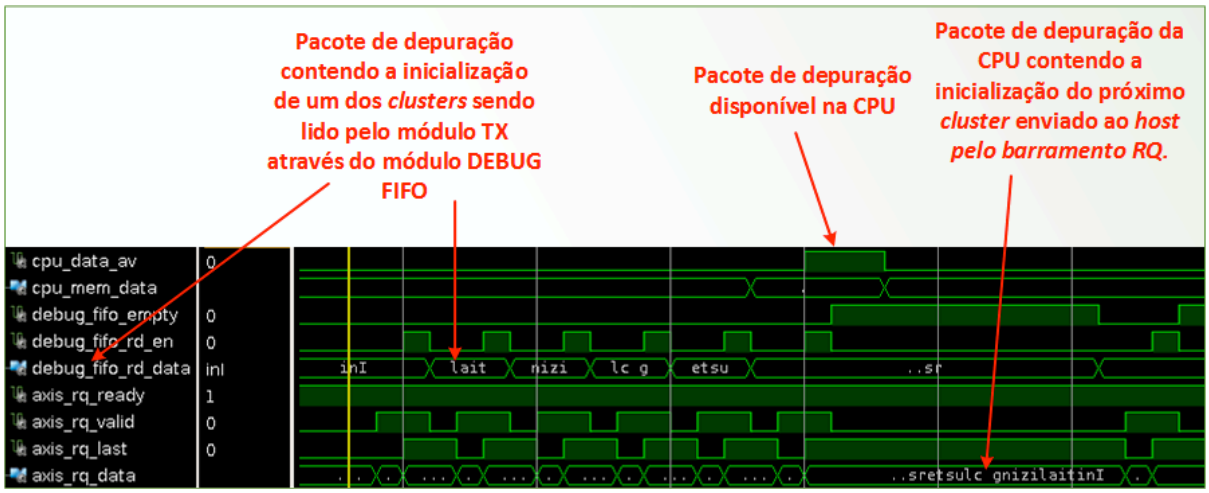

Figura 36 – Validação do recebimento de pacotes de depuração na CPU e envio do mesmo ao módulo DEBUG FIFO.

A Figura 37 apresenta o pedido de recursos disponíveis (serviço Resource Available – 0x240) aos *clusters*. Este pedido é feito pela CPU-ECS e enviado à HeMPS pelo DMA-R.

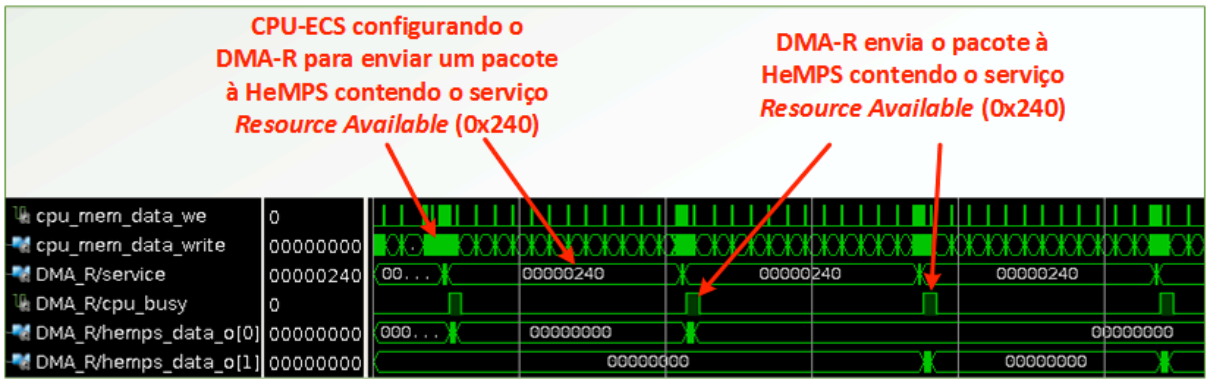

Figura 37 – Pacote contendo o serviço *Resource Available* sendo enviado à HeMPS.

A Figura 38 apresenta a validação final da integração dos módulos, com as mensagens de depuração chegando ao *host.* Primeiramente é feita a inicialização dos *clusters,* seguido da interrupção do mundo externo (1), representando o pedido de inserção de aplicação. Ao obter resposta de recursos disponíveis, o DMA-W insere a aplicação na memória do ECS (4) e é requisitada uma tarefa.

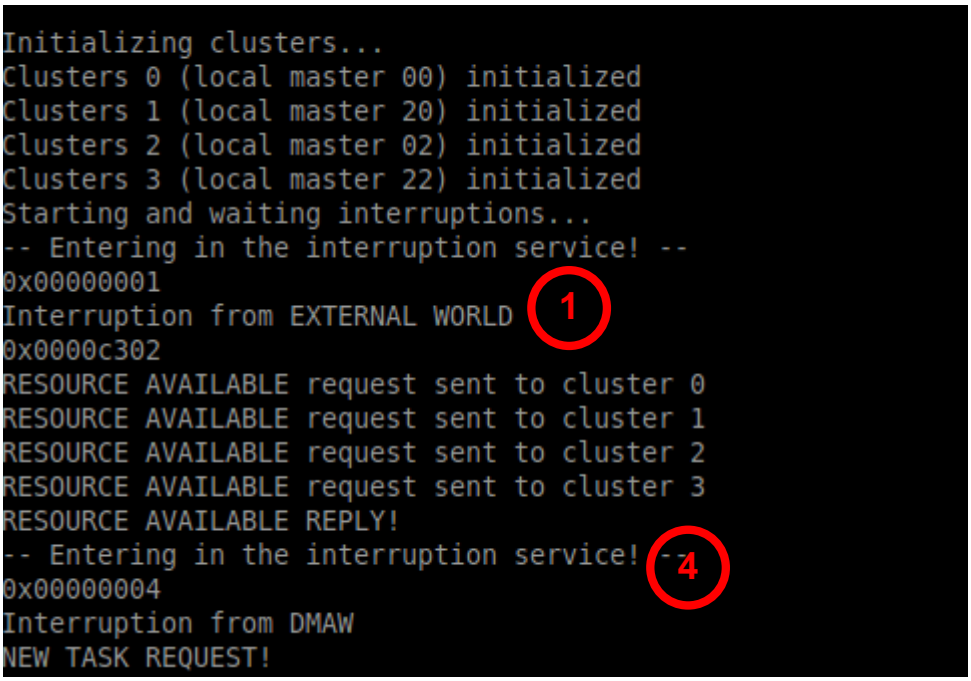

Figura 38 – Mensagens de depuração no *host.*

Dado a limitação de tempo para o Trabalho de Conclusão de Curso, a interface entre o ECS e a HeMPS foi emulada. Assim, um módulo de hardware emula as respostas esperadas pelo ECS, e recebe os dados enviados para a HeMPS. Desta forma, validou-se em FPGA toda a infraestrutura do *core PCIe* juntamente ao módulo ECS. Isto é, o projeto *core PCIe* + PIO + ECS foi validado por prototipação e verificado com *chipscope.* O projeto HeMPS-ECS foi validado por simulação.

6 RESULTADOS

Este Capítulo apresenta resultados de desempenho e ocupação de área.

6.1 Desempenho do DMA

Para comparar o desempenho ao fazer transferências de memória, incluímos nos *softwares* desenvolvidos (apresentados na seção 3.2) o cálculo de tempo durante as transações. A taxa média de transferência alcançada com o DMA foi de aproximadamente 2 GBps enquanto que sem DMA a taxa média foi de aproximadamente 24 KBps, conforme apresentado na Tabela 3.

O método de transferência sem DMA necessita que o *host* realize a transferência (escrita ou leitura) de cada palavra de memória individualmente, assim como é feita a configuração dos registradores do DMA. Desta forma, cada transferência na memória do *host,* faz com que o *kernel* (do *host*) realize uma requisição ao FPGA, assim, deixando de utiilizar o barramento de 256 bits disponibilizado pelo *core PCIe,* o qual comporta 8 palavras de 32 bits.

O tempo de transferência de um bloco de memória entre *host* e MEMORY do PIO, utilizando o DMA, foi contabilizado a partir do momento do *reset* do módulo DMA até o sinal *done,* informando o fim da transferência.

Tabela 3 – Taxa de transferência ao enviar dados da memória do *host* para o FPGA.

Modo de Transferêcia	Volume de dados.	Repetições	Tempo médio	Vazão
DMA	~400kB ou 409487 Bytes	10	0,186 ms	2,05 GBps
PIO original		12	16,15 ms	24,2 kBps

6.2 Características da Implementação no FPGA

Esta sessão analisa as características do projeto do *hardware* desenvolvido, i.e., área e *timing*.

6.2.1 Timing

Para alcançar um as restrições de temporização, foi necessário utilizar a opção de não manter a hierarquia, fazendo com que a síntese reconstrua a hierarquia RTL originalmente descrita. Esta opção fez-se necessária para a síntese

otimizar o caminho crítico do projeto, situado entre a memória de instruções e a unidade lógica aritmética (ULA) da CPU-ECS.

A Tabela 4 apresenta os valores de *Worst Negative Slack* (WNS) e *Worst Hold Slack* (WHS) obtidos após a síntese física para ambas as frequências de relógio utilizadas no projeto.

Tabela 4 – Resultado do pior *Timing Slack* obtido para as duas frequências de relógio utilizadas no projeto.

Frequência	WNS	WHS
250 MHz	+0,106 ns	+0,061 ns
100 MHz	+0,084 ns	+0,276 ns

6.2.2 Área

A ocupação de recursos no FPGA dos projetos envolvidos neste Trabalho de Conclusão de Curso está apresentada na Tabela 5.

Tabela 5 – Ocupação de área do FPGA.

	LUT	**FF**	**BRAM36**
PIO original	5087	4175	17
PIO-DMA	6667	5331	17
PIO-DMA + ECS	10412	6863	42

O *core PCIe* utiliza 9 BRAMs de 36Kb e a memória do PIO utiliza 8 BRAMs de 36Kb. A CPU do ECS utiliza 16 BRAMs de 36Kb para sua memória, enquanto que o *Application Repository* ocupa 8 BRAMs de 36Kb. A DEBUG FIFO utiliza uma BRAM36.

As figuras abaixo apresentam a área ocupada no FPGA. A Figura 39 apresenta a disposição das células no FPGA com o projeto de referência PIO original.

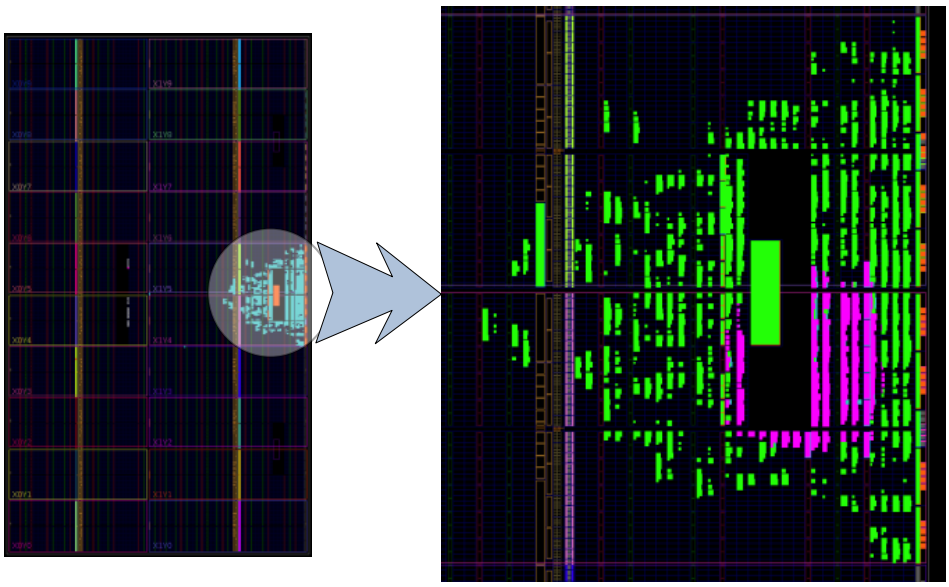

Figura 39 – O *core PCIe* está destacado em verde, enquanto o PIO está destacado em rosa.

A Figura 40 apresenta a ocupação de área do projeto de referência (PIO) modificado com o DMA (destacado em rosa) e o *core PCIe* destacado em verde.

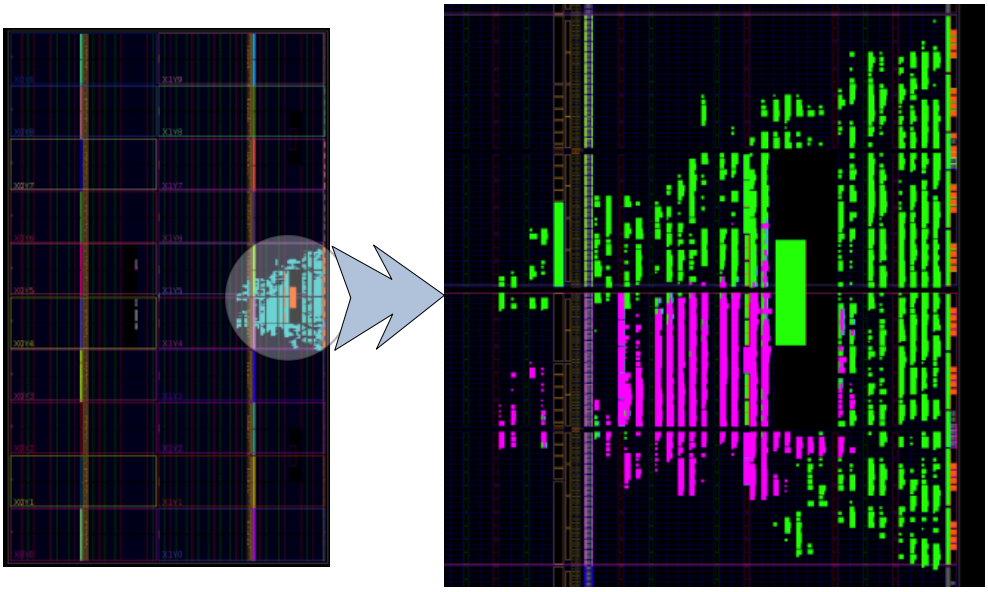

Figura 40 – Destacado em rosa o PIO com DMA e em verde o *core PCIe*.

A Figura 41 apresenta o projeto final, com o módulo ECS destacado na cor marrom. Apesar do módulo DMA-W e do *Application Repository* terem sido movidos para fora do módulo ECS, estes também estão representados em marrom.

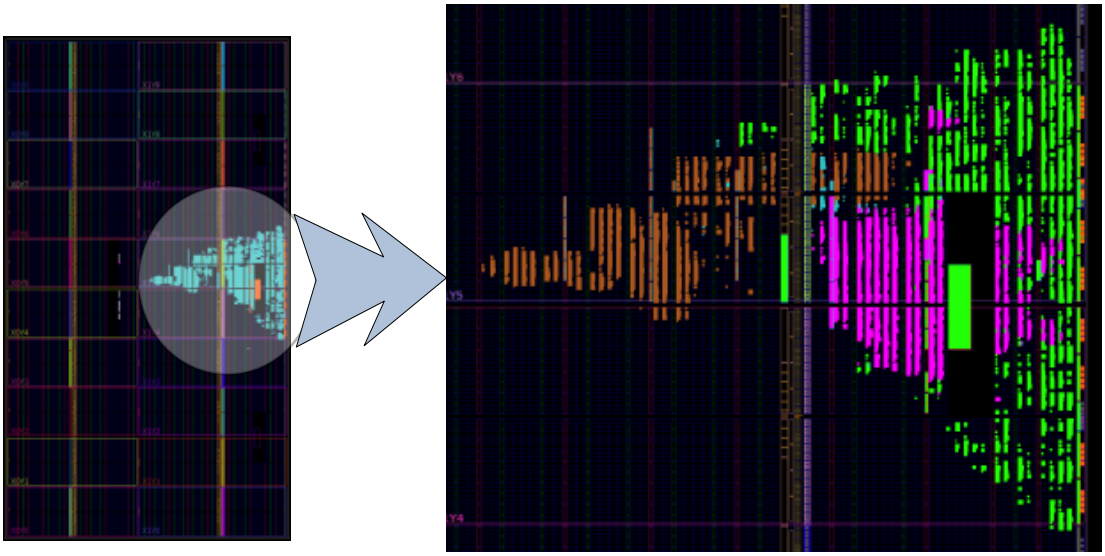

Figura 41 – Em marrom as células do módulo ECS, incluindo o DMA-W e *Application Repository*.

7 CONCLUSÃO

O trabalho descrito neste documento apresentou o desenvolvimento de uma interface *PCI Express* para comunicação com MPSoCs em FPGAs. Além disso, o sistema como um todo é ligado a um computador hospedeiro capaz de inserir aplicações em tempo real.

O processo para projetar esta interface *PCI Express* de comunicação, somado aos objetivos do projeto incluiu:

i. Compreender o *core PCI Express* utilizado;

ii. Compreender uma tecnologia moderna de FPGA;

iii. Compreender as técnicas de comunicação entre *software* e *hardware* descritos em FPGAs;

iv. Realizar alterações nos módulos da *PCI Express*, para que fosse possível a inserção de aplicações.

v. Projetar a interface de integração entre *core PCI Express e* ECS;

vi. Implementar o *software* de controle para realizar o protocolo necessário para a execução.

O projeto alcançou o principal objetivo inicial, projeto e validação de uma interface de integração entre *PCI Express* e *ECS*, além de realizar as inserções de aplicações em FPGA através do *software* desenvolvido. Para alcançar este objetivo, foi necessário dominar dois protocolos: o protocolo de comunicação *PCI Express* e o protocolo de comunicação da plataforma HeMPS, e como a mesma se comunica com o módulo ECS. Também foi necessário implementar uma chamada de sistema no *kernel* da CPU-ECS para depurar o software, compreender o projeto exemplo da *Xilinx* para uso da *PCI Express* e nele desenvolver o DMA para inserção de aplicações reais.

Através das contribuições realizadas no projeto, obtivemos resultados importantes como o aumento na vazão de dados, com uma velocidade na taxa de transferência 84710 vezes maior do que com o projeto original. Outro dado importante notado é a limitação do processador Plasma (CPU-ECS) no que se refere a restrições de temporização, visto que o caminho crítico do projeto contido nele necessitou de grande esforço lógico durante todo o projeto, para que fosse possível a prototipação.

Como trabalhos futuros, a prototipação e validação de aplicações sendo executadas na plataforma HeMPS, além de modificar o *software* de controle, para inserção de mais aplicações de forma parametrizável tornará o sistema completo e pronto para expandir as tecnologias inseridas neste projeto, como por exemplo, o

módulo ECS. A inserção de mais destes módulos torna possível o sistema conectar-se a mais periféricos, sendo assim tolerante a falhas, dado que passa a ser possível inserir aplicações a partir de diferentes origens. Outra ideia de trabalho futuro, é a utilização do projeto de referência PIO com o DMA desenvolvido para outras aplicações, dado que o PIO-DMA tornou-se um projeto de referência genérico.

8 REFERÊNCIAS

[ARM16] ARM Inc. “AMBA AXI4-Stream Protocol Specification v1.0”. Capturado em: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html, Janeiro 2016.

[CAP01] Cappelatti, E.; Moraes, F.; Calazans, N.; Oliveira, L. “Barramento de Alto Desempenho para Interação Software/Hardware”. In: IBERCHIP, 2001.

[CAR09] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. “HeMPS - A Framework for NoC-Based MPSoC Generation”. In: ISCAS, 2009, pp. 1345-1348.

[CAS10] Castilhos, G.; Luigi, L.; Grechi, T. “Emulação de Sistemas Digitais Síncronos em Dispositivos FPGAs”. Trabalho de Conclusão de Curso, Engenharia de Computação, PUCRS, 2010.

[CCM16] CCM Group. “Transmissão de dados – Os modos de transmissão”. Capturado em: http://br.ccm.net/contents/689-transmissao-de-dados-os-modos-de-transmissao, Março 2016.

[DIG15] Digilent Inc. “Product Description”. Capturado em: http://store.digilentinc.com/netfpga-sume-virtex-7-fpga-development-board/, Março 2015.

[ECS16] Referência interna. “External Control Access”. In: GAPH – Grupo de Apoio e Pesquisa em Hardware, 2016.

[EME08] Infowester. “Tecnologia PCI Express”. Capturado em: http://www.infowester.com/pciexpress.php, Agosto 2008.

[MON13] Monteiro, T.; Ruaro, M.; Moraes, F. “Parametrização de Memória e Monitoramento da Plataforma HeMPS”. In: XIV Salão Iniciação Científica da PUCRS, 2013.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Mello, A.; Moller, L.; Ost, L. “HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip”. Integration the VLSI Journal, vol. 38-1, 2004, pp. 69-93.

[NAI09] National Instruments. “PCI Express - An Overview of the PCI Express Standard”. Capturado em: http://zone.ni.com/devzone/cda/tut/p/id/3767, 2009.

[PCI10] PCI-SIG Inc.”PCI Express Base Specification Revision 3.0”. Capturado em: http://composter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf, Novembro 2010, pp. 40.

[PLA09] Rhoads, S. “PLASMA Processor”. Capturado em: http://www.opencores.org/?do=project&who=mips, 2009.

[RAU09] Raupp, T.; Rodolfo, C.; Scartezzini, G. “Desenvolvimento de um Ambiente de Aplicações Embarcadas para a Plataforma Multiprocessada HeMPS”. Trabalho de Conclusão de Curso, Engenharia de Computação, PUCRS, 2009.

[WOS07] Woszezenki, R. C. “Alocação De Tarefas E Comunicação Entre Tarefas Em MPSoCS”. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2007, 121p.

[XIL15] Xilinx Inc. “Virtex-7 FPGA Gen3 Integrated Block for PCI Express v4.1”. Capturado em: http://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v4_1/pg023_v7_pcie_gen3.pdf, Setembro 2015.