Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistemas Multiprocessados Heterogêneos em Redes em Chip utilizando o framework HeMPS

Lukas Ferreira Machado

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Marcelo Grandi Mandelli

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistemas Multiprocessados Heterogêneos em Redes em Chip utilizando o framework HeMPS

Lukas Ferreira Machado

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Marcelo Grandi Mandelli (Orientador)
CIC/UnB

Prof. ª Dr. ª Alba Cristina Magalhães Alves de Melo
CIC/UnB

Prof. Dr. Marcus Vinícius Lamar
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 15 de Julho de 2019

Dedicatória

Dedico esse trabalho a minha família, especialmente a minha noiva, Rosana Rogiski, pelo suporte e companheirismo em todos os momentos. Aos amigos que me aconselharam nos momentos mais difíceis e, especialmente, ao meu orientador, Marcelo Grandi Mandelli, que graças a sua ajuda estou tendo a oportunidade de me formar como um engenheiro da computação.

Agradecimentos

Agradeço a Universidade de Brasília por ter proporcionado momentos incríveis da minha vida. Agradeço a minha noiva, Rosana Rogiski, pelas noites a fio me apoiando e incentivando, mesmo nos momentos mais difíceis. Agradeço aos membros da banca avaliadora pelo tempo que dedicaram a ler esse trabalho que fiz com tanto afinco e, especialmente, agradeço o meu orientador, Marcelo Grandi Mandelli, por toda paciência e dedicação que teve comigo.

Resumo

Devido ao menor custo, melhor performance e eficiência energética, sistemas multiprocessados em *chip* heterogêneos vem se tornando bastante utilizadas para os mais diversos projetos de sistemas. A fim de se entender os diversos aspectos desses sistemas, por meio da plataforma *Hermes Multiprocessor System* (HeMPS) para sistemas multiprocessados em *chip* com elementos de processamento homogêneos, o presente estudo se faz necessário para propor a integração de um módulo de *hardware* para demonstrar a viabilidade e possíveis benefícios de se adaptar um sistema homogêneo em um heterogêneo, contribuindo com uma interface para outros sistemas a serem integrados através da HeMPS. A metodologia escolhida para análise foca na integração e adaptações para tornar a plataforma heterogênea. Espera-se que esse estudo contribua e incentive o desenvolvimento de novas e melhores alternativas para sistemas multiprocessados em *chip*.

Palavras-chave: MPSoC, NoC, HeMPS, MPSoCs heterogêneas, trabalho de conclusão de curso

Abstract

Due the lower cost, better performance and energetic efficiency, heterogeneous multi processor systems in chip are becoming widely used for several systems projects. In order to understand most aspects of these systems, through the plataform *Hermes Multiprocessor System* (HeMPS) for multi processor systems in chip with homogeneous processing elements, this thesis turns necessary to evaluate the integration of a hardware module to demonstrate the viability and possible beneficts of adaptating a homogeneous system into a heterogeneous one, contributing with a new interface to be used for other systems that would be integrated through the HeMPS. The chosen methodology emphasis on the integration and adaptations in order to turn the plataform heterogeneous. It is expected that this thesis contribute and inspire the development of new and better alternatives for multi processor systems in chip.

Keywords: MPSoC, NoC, HeMPS, heterogeneous MPSoCs, thesis

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Interface de Programação de Aplicativos (*Application Programming Interface*).

CPU *Central Processing Unit*.

DMA Acesso Direto à Memória (*Direct Memory Access*).

DMNI Interface de Rede de acesso direto à memória (*Direct Memory Network Interface*).

FIFO Primeiro a entrar, Primeiro a sair (*First-In First-Out*).

flit unidade de controle de fluxo (*flow control unit*).

flits unidades de controle de fluxo (*flow control units*).

FPGA Arranjo de Portas Programáveis em Campo (*Field Programmable Gate Array*).

FSM Máquina de Estados Finitos (*Finite State Machine*).

HDL *Hardware Description Language*.

HeMPS *Hermes Multiprocessor System*.

IP Propriedade Intelectual (*Intelectual Property*).

LST *Least Slack Time*.

MIPS Microprocessador sem estágios intertravados de pipeline (*Microprocessor without interlocked pipeline stages*).

MMR Registrador de mapeamento de memória (*Memory-Mapped Register*).

MPE Elemento de Processamento Gerente (*Manager Processing Element*).

MPSoC Sistemas intra-chip multiprocessados (*Multiprocessor Systems on Chip*).

NI Interface de Rede (*Network Interface*).

NoC Rede em chip (*Network on Chip*).

PE Elemento de Processamento (*Processing Element*).

PH *Packet Handler*.

phit unidade física (*phisical unit*).

RAM *Random Access Memory*.

RISC Computador com um conjunto reduzido de instruções (*Reduced Instruction Set Computer*).

RR *Round Robin*.

RTL *Register Transfer Level*.

SMPE Elemento de Processamento Gerente do Sistema (*System Manager Processing Element*).

SoC Sistema em Chip (*System on Chip*).

SPE Elemento de Processamento Escravo (*Slave Processing Element*).

TCB Bloco de Controle de Tarefas (*Task Control Block*).

UART Receptor/Transmissor Universal Assíncrono (*Universal Asynchrounous Receiver/Transmiter*).

Capítulo 1

Introdução

Avanços tecnológicos nos mais diversos campos levaram a uma alta densidade de transistores em pastilhas únicas de processamento, os chips [3], com a redução constante do tamanho dos mesmos, o que acarretou no surgimento dos sistemas integrados intra-chip, denominados SoCs (*Systems-on-Chip*).

Uma das principais premissas de um projeto que utilize SoCs é a facilidade para reutilizar diferentes *hardwares* acoplados numa mesma arquitetura, ou seja, sua reusabilidade. Com isso, módulos devidamente fundamentados são utilizados nesses sistemas [3], os chamados IP (*Intelectual Property*). Do vasto conjunto de módulos, processadores, bancos de memória, barramentos de comunicação e outras interfaces serão interligados em um mesmo *chipset* de silício. Assim sendo, o arquiteto e projetista do sistema tem a tarefa de melhor escolher como todos os módulos trocam informações por meio do mecanismo de comunicação [3]. Essa modelagem de sistema visa a disponibilização desses sistemas no mercado em um curto período de tempo. A reutilização de módulos proprietários, já disponíveis no mercado, reduz ainda os custos de criação dos mesmos e, com isso, o custo final para o consumidor.

Entre os mais variados sistemas feitos dessa forma, cabe destacar aqueles que integram múltiplos processadores, denominados sistemas multiprocessados intra-chip (MPSoCs, *Multiprocessor Systems-on-Chip*). Esse tipo de sistema foi planejado focando a alta e crescente demanda por performance em máquinas que requerem eficiência energética durante a execução de aplicações embarcadas concorrentes como compressão de vídeo, comunicação *wireless* e jogos, por exemplo [4].

Um grande desafio no projeto desses sistemas é a escolha ou a criação de uma arquitetura de comunicação entre os elementos de processamento (PE) presentes nos MPSoCs. A forma usada atualmente para estabelecer essas comunicações é a que utiliza Redes intra-Chip, as NoCs (*Networks-on-Chip*) [3]. Nessas redes, mensagens são transferidas entre seus PEs de forma a garantir o controle do sistema e a execução correta das aplicações

alvo. O núcleo de processamento presente nesses elementos geralmente são compostos de processadores, roteadores, interfaces de rede entre outros módulos.

Nesse trabalho, será usado como *framework* de referência o HeMPS, desenvolvido pelo grupo *GAPH* [5]. Sua característica é gerar MPSoCs homogêneos baseados em NoCs. Na composição desses MPSoCs, está presente o processador Microprocessador sem estágios intertravados de pipeline (*Microprocessor without interlocked pipeline stages*) (MIPS). Esse processador apresenta uma implementação simples capaz de executar toda a arquitetura de conjunto de instruções (ISA) de inteiros de 32 *bits* do MIPS.

Os diferentes tipos de sistemas multiprocessados intra-chips são classificados em MPSoCs homogêneos e heterogêneos. Os homogêneos usam a mesma arquitetura em todos os seus elementos de processamento. Dessa forma, são ditos simétricos ao usar a mesma implementação de processadores enquanto que são assimétricos no caso de utilizar diferentes implementações [6] com mesma arquitetura. Já os sistemas heterogêneos utilizam diferentes arquiteturas nos elementos de processamento, de forma que possa haver diferentes processadores e *hardwares* dedicados. Assim, é possível aumentar a performance e reduzir o consumo energético em aplicações específicas [7]. Atualmente, devido a essas vantagens, procura-se utilizar MPSoCs heterogêneos para se obter o desempenho desejado em aplicações específicas [8].

Além disso, esse trabalho serve como estudo de caso para se verificar os desafios e requerimentos para se ter elementos heterogêneos dentro de sistemas multiprocessados, entendendo como pode ser feito o controle de fluxo de dados e cuidados ao se adaptar diferentes módulos para se comunicar, tanto a nível de *software* quanto de *hardware*. Para isso, nesse estudo foi criado um módulo de *hardware* para o envio e recebimento de pacotes na HeMPS via *hardware*, diferenciado-se do envio e recebimento via *software* na implementação original da HeMPS com o objetivo de que esse módulo sirva de interface para possíveis novos sistemas com a NoC do MPSoC da plataforma.

1.1 Objetivos

Os objetivos desse projeto são subdivididos em duas subseções. Na subseção 1.1.1, são descritos os objetivos gerais, fundamentando os conhecimentos no processo deste estudo. Os objetivos específicos são exemplificados em 1.1.2, exemplificando tópicos sobre a implementação do estudo de interesse deste projeto.

1.1.1 Objetivos gerais

A partir desse estudo, busca-se adquirir e aperfeiçoar conhecimentos relacionados a SoCs, NoCs, MPSoCs e possíveis melhorias em arquiteturas e processos de comunicação utili-

zados em sistemas multiprocessados em chip. Além disso, almeja-se criar uma interface que possibilite futuras integrações com outros módulos de *hardware* na HeMPS por meio de sua NoC.

1.1.2 Objetivos específicos

Com esse estudo, busca-se fazer mudanças de módulos nos elementos de processamento para se ter componentes mais heterogêneos nas MPSoCs geradas pelo *framework* HeMPS e, para auxiliar na prova de conceito para um MPSoC heterogêneo, criar um módulo que futuramente sirva como uma interface entre novos sistemas e a NoC do MPSoC da plataforma, algumas etapas são necessárias:

1. escolher e implementar um módulo para demonstrar um elemento de processamento heterogêneo, bem como sua validação a nível de *software*;
2. desenvolvimento, modificação e adaptação de diferentes interfaces, tanto a nível de *software* quanto a nível de *hardware*, e criação de um novo módulo de *hardware* que sirva como prova de conceito para uma HeMPS com MPSoCs heterogêneos e futura acoplagem de outros módulos de *hardware*, respeitado-se os requerimentos necessários para se ter elementos de processamento heterogêneos;
3. desenvolvimento, modificação e adaptação de diferentes módulos e interfaces que garantam a comunicação dentro da plataforma, respeitando-se as exigências advindas dessas mudanças [9].

1.2 Justificativa

Este trabalho, por meio de um estudo de caso, tem por objetivo a implementação de um MPSoC heterogêneo utilizando-se como referência a plataforma HeMPS, procurando-se modificar o menos possível sua arquitetura de *hardware* e comportamentos presentes. Para isso, a metodologia utilizada se deu através da inclusão de um módulo de *hardware* na HeMPS que atue fazendo uma interface com a NoC do sistema para envio e recebimento de mensagens via *hardware*.

Também, por meio dessa mudança dentro da HeMPS, objetiva-se mostrar que é possível a migração para uma MPSoC heterogênea e, com isso, uma plataforma heterogênea de melhor performance e melhor acoplamento dos mais diversos módulos [9].

Por fim, deseja-se criar um módulo de *hardware* que, além de tornar a HeMPS heterogênea, sirva em trabalhos futuros como base para a integração de outros módulos de *hardware* com a HeMPS por meio da comunicação em sua NoC.

1.3 Metodologia da pesquisa

Com intuito de conseguir o objetivo nesse trabalho, o conhecimento aprofundado sobre MPSoCs, NoCs e, primordialmente, o entendimento sobre o *framework* HeMPS é fundamental. Para isso, o conhecimento sobre a arquitetura e o sistema operacional desejados da plataforma são fundamentais. O mesmo se aplica aos mecanismos de criação do *framework*, com seus diversos *scripts* e fluxogramas de controle, além dos mecanismos de comunicação e acoplamento dos diferentes módulos que compõem os elementos de processamento do sistema.

Assim sendo, temos que incluir algum módulo para demonstrar a validade de uma arquitetura heterogênea, sendo realizada a partir de códigos nas linguagens de programação *C* e *C++*, o envio, recebimento e validação de mensagens e seus pacotes por um módulo que se interconecta com um banco de memória local ao elemento de processamento. A partir dele, testamos se os mesmos pacotes estão válidos, verificando-se seus campos de controle e a mensagem, que seria sua carga útil, no recebimento. Para isso, serão modificados os *scripts* de geração da plataforma e seus módulos de compilação, uma vez que esse módulo de teste entra no lugar do processador do PE e, com isso, tornam-se necessários cuidados a mais com o controle de processos nos *clusters* e alocação de processos nos mesmos a partir do *kernel* mestre.

Além disso, vale a pena ressaltar que a interface entre os componentes dos PEs será mantida, uma vez que o intuito desse estudo é verificar a validade de módulos heterogêneos sem modificar o comportamento do *hardware* que atualmente faz parte da plataforma, mas adaptando o *software* para lidar com possíveis mudanças de comportamento, como, por exemplo, número de processos por *cluster* e mapeamento de processos.

1.4 Estrutura do texto

O texto desse estudo é divido em 5 capítulos que são apresentados abaixo:

1. *Introdução*: neste capítulo são explicados os objetivos, os desafios e a metodologia de pesquisa do estudo de caso, além do que se almeja com esse trabalho por meio de uma justificativa.

2. *Fundamentação teórica*: nesse capítulo se solidifica os conceitos primordiais sobre a plataforma HeMPS para um melhor entendimento do estudo de caso, os passos de sua evolução e desafios.

3. *Metodologia utilizada para análise do framework HeMPS*: nesse capítulo é descrita a metodologia e etapas seguidas para a realização deste trabalho.

4. *Resultados*: nesse capítulo é descrita a utilidade da plataforma após todas as mudanças feitas com intuito de se ter uma MPSoC heterogênea, além da análise das mudanças com a HeMPS na sua forma original.

5. *Conclusão*: nesta parte fazemos uma sucinta análise do trabalho apresentando um resumo das ideias abordadas e trabalhos futuros.

Capítulo 2

Fundamentação teórica

O objetivo deste capítulo é descrever e fazer uma breve análise dos assuntos essenciais para a compreensão deste trabalho. Dessa forma, as seções subsequentes tem o intuito de fundamentar e caracterizar o conhecimento fundamental para se entender as propostas a serem apresentadas.

Na seção 2.1 são descritos e caracterizados os conceitos e a teoria para sistemas em *chip* ou sistemas integrados *intra-chip* (em inglês, *System-on-chip* ou na sua forma abreviada, Sistema em Chip (*System on Chip*) (SoC).

A seção 2.2 detalha um tipo específico de SoC denominado sistema *intra-chip* multiprocessado (em inglês *Multiprocessor Systems-on-Chips* ou de forma abreviada Sistemas intra-chip multiprocessados (*Multiprocessor Systems on Chip*) (MPSoC)). São descritas as principais categorizações, características e arquiteturas utilizadas com suas principais vantagens, desvantagens e usos.

A seção 2.3 apresentamos o conceito de rede em *chip* com seus principais conceitos, diferentes arquiteturas e modelos empregadas para seu devido funcionamento.

A seção 2.4 apresenta o *framework* para geração de sistemas *intra-chip* multiprocessados parametrizáveis denominado *Hermes Multiprocessor System* (HeMPS) utilizado nesse projeto. Nessa seção, os principais módulos que o compõe bem como seu funcionamento e detalhes de arquitetura são detalhadamente mostrados, além de uma explicação sobre seu funcionamento tanto a nível de *software*, por meio de seu núcleo de processamento, o *kernel*, quanto a nível de *hardware*.

2.1 Sistemas em *Chip* ou *intra-chip*

De acordo com Gonçalves [10], um Sistema em Chip (*System on Chip*) (SoC) pode ser definido como a integração de um sistema completo em um único *chip*. Esse tipo de componente é muito utilizado em sistemas portáteis e embarcados devido a fatores como

baixo consumo de energia, custo final menor se comparado a dispositivos com componentes não encapsulados, bom desempenho que apresentam e a minimização do espaço que permitem.

Os projetos de SoC tendem a ser bastante complexos em diferentes aspectos devido ao fato de que precisam considerar os diferentes componentes integrantes de um SoC tanto em nível de *software* quanto *hardware*. Uma forma de minimizar a sua complexidade e tornar os projetos comercialmente viáveis é a utilização de blocos Propriedade Intelectual (*Intelectual Property*) (IP), uma vez que mesmo que possuam restrições de propriedade intelectual, são compostos por núcleos amplamente conhecidos e foram previamente verificados e testados pela comunidade, o que minimiza os riscos que existiriam caso fosse optado pelo desenvolvimento próprio de tais componentes.

Os blocos IP comercializados podem ser classificados em três grandes grupos, como consequência direta da sua forma de implementação: *Soft-core*, *Firm-core* ou *Hard-core*.

***Soft-core*:** são blocos IP compostos por núcleos escritos em uma linguagem de descrição de *hardware* (*Hardware Description Language* (HDL)) independente de tecnologia e sintetizável [3];

***Firm-core*:** são blocos IP compostos por núcleos com mais informações, o que é expresso, por exemplo, pela presença de *netlists* que armazenam a descrição das conexões dos componentes dos componentes eletrônicos no qual se inserem. Devido ao fato de guardarem informações sobre componentes de circuitos, esse tipo de bloco IP, embora ainda possa ser parametrizável e ter alguma flexibilidade num projeto, é em certo grau, dependente da tecnologia empregada [11];

***Hard-core*:** são blocos IP compostos por núcleos que carregam dentro de si informações de temporização e *layout*, e por isso, estão prontos para serem empregados em um sistema. Por armazenarem dados sobre a topologia do sistema, fornecem uma maior confiabilidade ao sistema, mas, em contrapartida, são fortemente dependentes da tecnologia utilizada [11].

Um exemplo de SoC comercializado é o *Qualcomm Snapdragon 850 Mobile Computer Platformm*, parte da linha de processadores *Snapdragon mobile processors* da Qualcomm. Esse tipo de SoC é capaz gerenciar e executar múltiplas requisições de forma concorrente com baixo custo energético. A Figura 2.1 apresenta um diagrama dos blocos funcionais desse modelo de SoC [12]. Esse modelo em especial integra blocos IP como unidade central de processamento, processador gráfico, módulo de segurança para computação em nuvem, módulos de rede e sistema de gerenciamento de memória, cada um com funções específicas que juntas permitem o funcionamento do sistema geral.

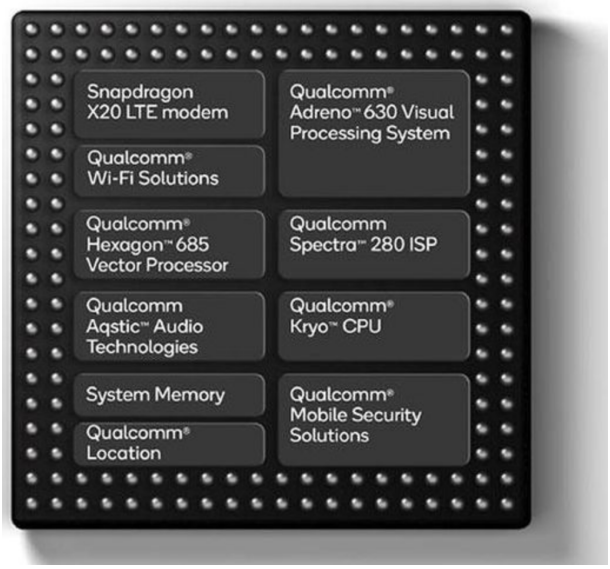


Figura 2.1: Diagrama funcional de blocos do sistema *intra-chip Qualcomm Snapdragon 850 Mobile Computer Platform* (Fonte: [12]).

O exemplo descrito na Figura 2.1 dá uma dimensão do grau de complexidade e tamanho do desafio enfrentado ao tentar-se integrar e interconectar os diferentes módulos de um SoC, o que justifica a escolha usual de se optar por um dispositivo com propriedade intelectual vinculada ao invés de se tentar construir um dispositivo próprio.

2.2 Sistemas *intra-chip* multiprocessados - MPSoCs

Mesmo com os grandes desafios percebidos na integração dos diferentes módulos de um SoC, os requisitos atuais para desenvolvimento de SoCs giram em torno da necessidade de se executar múltiplas tarefas com tamanho e complexidade computacional diversos, de forma paralela, o que levou à criação de sistemas com diversos Elemento de Processamento (*Processing Element*) (PE)s em um único *chip* de silício. Isso gerou componentes com alta densidade de transistores, o que é, então, classificado como um Sistema *intra-chip* multiprocessado, ou MPSoC.

Sistemas *intra-chip* multiprocessados podem ser definidos como uma série de elementos de processamento interconectados por meio de uma arquitetura de comunicação bem definida, que podem interagir e se comunicar de diversas formas dependendo, principalmente, da arquitetura e contexto a elas empregados. Um exemplo de rede MPSoC são as redes em *chip* [6], que serão abordadas com mais detalhes em futuras subseções.

De acordo com [6], MPSoCs podem ser categorizados como *homogêneos* e *heterogêneos*.

Homogêneos: um MPSoC é dito homogêneo se todos os seus elementos de processamento são simétricos, ou seja, possuem a mesma arquitetura. Se todos os PEs do MPSoC possuem, também, a mesma implementação, ou seja, utilizam a mesma estrutura de infraestrutura/arquitetura dizemos que este sistema é *simétrico* [6]. Por outro lado, se os elementos de processamento possuem a mesma arquitetura e implementações diferentes diz-se que o MPSoC é *assimétrico*.

Heterogêneos: um MPSoC é classificado como heterogêneo se a arquitetura dos diferentes PEs é diversa e, portanto, são especializados para determinados requisitos [6].

Devido à sua simetria em termos de arquitetura, MPSoCs homogêneos permitem vantagens no que se refere à melhor ênfase em requisições, ou seja, maior desempenho entre trocas de desempenho e gasto energético, tolerância a falhas, maior simplicidade, generalidade e flexibilidade de uso, além de menor custo durante a criação e desenvolvimento de projetos, devido à similaridade dos elementos, o que permite suporte mais aprimorado para protocolos de requisição dentro dos barramentos de comunicação. Por outro lado, Van Berkel [13] afirma que para sistemas e aplicações com alta demanda de capacidade de processamento, MPSoCs heterogêneos tendem a apresentar maior desempenho com menos gasto energético, o que se torna importante nos sistemas atuais, que exigem a integração de unidades de processamento especializadas quando se buscam determinados níveis de processamento e performance em tarefas específicas.

2.3 Interconexão no *chip*

Cada *chip* com múltiplos núcleos possui, principalmente, dois tipos de componentes primordiais: elementos de processamento (*core*) e outros elementos de não processamento tais como arquitetura de comunicação e memória (*uncore*) [14]. Embora a alta densidade de transistores nos sistemas em *chip* atuais proporcione um grande número de núcleos para uso, a comunicação eficiente, ou seja, com baixa latência, controle de fluxo e recuperação de erros entre os diferentes módulos ainda é um desafio. A arquitetura de comunicação no *chip* controla o fluxo de requisições à memória e controla o tráfego de entrada e saída (I/O), provendo uma interconexão confiável para troca de informações entre tarefas. Logo, uma arquitetura de comunicação com baixa performance pode, quase que completamente, comprometer o desempenho de um sistema mesmo que composto por uma arquitetura com múltiplos núcleos de processamento de alta performance. Dessa forma, prover um meio de comunicação entre tarefas de maneira escalável e de alta performance é um ponto chave para muitos arquitetos de *hardware* [15]. Os principais desafios enfrentados pelos arquitetos na criação de novas implementações SoC são, principalmente:

Comunicação escalável para múltiplos *cores*: Podemos afirmar sem perda de generalidade que a performance dos PEs é limitada pela capacidade de comunicação entre os mesmos [15]. Devido ao aumento crescente na capacidade de processamento dos diferentes módulos dos sistemas, é difícil balancear a taxa de comunicação e a taxa de consumo das requisições no canal de transmissão de forma que não exista um gargalo entre eles. Além disso, com a existência de milhares de componentes em um único *chip*, os sistemas precisam se adequar para suportar múltiplos fluxos de comunicação em paralelo, de modo que não exista latência (mesmo que de apenas um único ciclo de comunicação) para componentes nos extremos do *chip*. Esses desafios podem acarretar maiores custos na definição do layout final dos componentes e na escolha e disposição dos meios de comunicação.

Energia limitada: Em 1974, Dennard [16] previu que a densidade de potência do transistor permanecerá constante à medida que nos movemos para tamanhos menores de nós. Isso é conhecido como *lei de dimensionamento de Dennard* [16]. Porém, nos últimos tempos, notou-se que a redução da área de um transistor é inversamente proporcional à sua redução de potência, de forma que ainda que houvesse uma quantidade gigantesca de transistores à disposição num sistema, os mesmos não poderiam ser acionados todos ao mesmo tempo, o que geraria uma perda de eficiência no que se refere ao uso de componentes para uma aplicação. Portanto, melhorar a eficiência energética dos componentes do *chip* tornou-se o pré-requisito para aumentar o dimensionamento previsto na lei de Moore.

Aplicações Heterogêneas: Diferentes conjuntos de aplicações podem se comunicar com arquiteturas computacionais de diferentes formas, sendo esperado que um *chip multi-core* moderno execute grandes conjuntos dessas aplicações. Logo, dentro das aplicações, a latência de comunicação e o requisito de largura de banda podem variar [17], não sendo difícil notar que a performance está altamente correlacionada com a eficiência na interconexão entre as tarefas. Sendo assim, layouts de interconexões precisam ser planejados considerando-se os piores casos de execução/interação entre diversas aplicações, o que pode gerar *SoC*s ineficientes para sistemas específicos, como no caso de sistemas mono-processados.

Métricas para avaliação de performance em interconexões: Um grande desafio para mensurar performance em interconexão no *chip* é configurar métricas como latência em comunicação e largura de banda disponível para memória, que independam das aplicações ativas na SoC, como seria o caso de métricas pontuais para determinados contextos dentro de aplicações, como tempo de execução. Dessa forma, percebe-se que gerar as métricas e fazer a análise dos seus resultados pode ser uma tarefa

bastante complexa para um conjunto de aplicações já que pode envolver medidas sujeitas ao tipo de processo que está ativo no sistema a ser avaliado [18].

Custo associado à arquitetura e projeto de um *chip*: Os custos para o projeto de um *chip* com múltiplos PEs têm aumentado em um ritmo alarmante devido a custos não associados à engenharia do *chip*. Ou seja, no seu processo de criação e confiabilidade de um *chip*, custos com pesquisa, prototipação e testes para avaliação de desempenho têm crescido consideravelmente. Como consequência, na maioria das vezes, projetistas acabam por reutilizar arquiteturas anteriores bem consolidadas e verificadas ao invés de tentar desenvolver arquiteturas inéditas para os *chip*s, especialmente devido à importância de se incorporar no processo de desenvolvimento ferramentas capazes de explorar o melhor layout (e mensurar isso), no que se refere à tempo-eficiência, para um conjunto de aplicações.

Confiabilidade das interconexões: Com a redução do tamanho dos componentes, passamos a nos preocupar com a confiabilidade dos circuitos digitais, uma vez que condições de operação imprevisíveis podem surgir e causar comportamentos anômalos no resultado final. Devido às restrições de operação dos sistemas, arquitetos devem se preocupar em integrar diversos mecanismos de confiabilidade para garantir que as interconexões entre os componentes ocorre com a qualidade desejada.

2.3.1 Arquiteturas de comunicação em SoCs modernos para Sistemas *intra-chip* multiprocessados (MPSoCs)

Com o aumento do número de PEs em um mesmo sistema em *chip*, a qualidade da comunicação entre esses elementos tornou-se um fator determinante no desempenho dos MPSoCs, uma vez que estes podem ser tão complexos a ponto de inviabilizar canais de comunicação ponto a ponto ou gerar um desempenho muito reduzido nos canais multiponto [3] [19], o que não é interessante para as exigências atuais de desempenho e eficiência energética de componentes.

Devido à sua reutilização, um sistema de comunicação que transfere dados entre componentes por meio de um barramento tende a ser o modo preferencialmente usado na comunicação entre os diversos PEs. No entanto, limitações como consumo de energia, escalabilidade, largura de banda compartilhada pelos núcleos conectados ao barramento e a não existência de paralelismo dentro dos canais de comunicação tornam-se obstáculos para novas implementações do MPSoC.

A Figura 2.2 mostra um exemplo genérico de bloco IP com destaque para os canais de comunicação existentes entre os pares de núcleo. É importante notar que na arquitetura

apresentada existem tanto canais ponto-a-ponto, como aqueles ligando os núcleos mestre aos seus escravos, quanto canais multi-ponto, como aqueles nomeados como endereço, dado e controle. Podemos observar que esses canais multi-ponto são ligados à Unidade de Controle de Barramento (também conhecida como Árbitro).

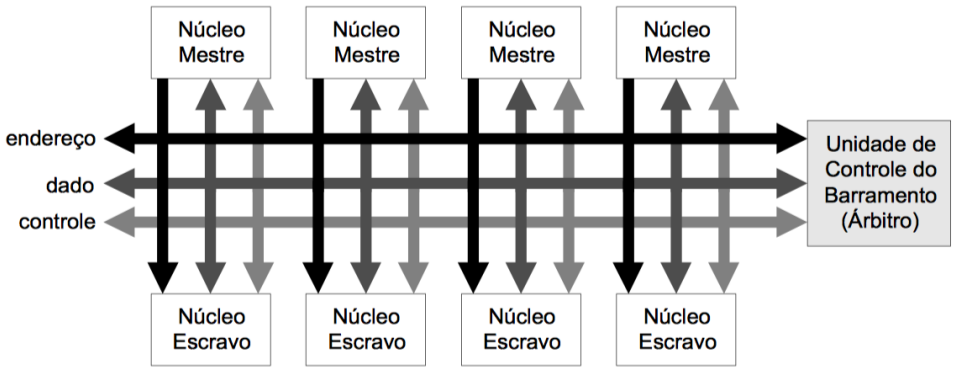


Figura 2.2: Arquitetura genérica de um SoC baseado em barramento (Fonte: [3]).

O exemplo apresentado na Figura 2.2 apresenta uma representação de um sistema com grande número de tarefas concorrentes que exige uma estrutura de controle ao barramento de comunicação. Nesse caso, os componentes que iniciam a comunicação para transferência no barramento são denominados *mestres*, enquanto os receptores recebem a nomenclatura de *escravos*. Para sistemas com diversos componentes mestres, o árbitro utiliza algum critério de escalonamento para conceder o uso ao barramento a um determinado mestre, a fim de evitar colisões que impeçam a troca de informações entre os núcleos. O algoritmo de escalonamento utilizado pelo árbitro pode ser baseado tanto em prioridades estáticas quanto dinâmicas do sistema em questão. Um último ponto importante a se observar diz respeito ao compartilhamento do árbitro: note-se que existe apenas um árbitro gerenciando requisições de comunicação de múltiplos núcleos e, portanto, o aumento no número de mestres no sistema acarreta no incremento da latência na arbitragem [3].

Uma vez que cada bloco IP potencialmente realiza atividades e possui funções diferentes dentro de um sistema, um MPSoC é composto por diferentes tipos de interfaceamento, necessários para que a comunicação entre os blocos IP ocorra de forma satisfatória dentro do *chip*. Uma vez que um bloco IP é composto por diversos núcleos, é, também, necessário que esses núcleos sejam capazes de se comunicar entre si. Há dois tipos principais de meios de comunicação entre os núcleos de um bloco IP: conexões em canais ponto-a-ponto ou em canais multi-ponto, conforme ilustrado na (Figura 2.3):

Canais ponto-a-ponto: Utilizada em sistemas modelados para comunicação de fluxo de dados, como em codificadores/decodificadores de áudio ou de vídeo, nesse tipo

de canal de comunicação os núcleos são interligados por canais dedicados ligando pares de núcleos de forma exclusiva. Devido à exclusividade do canal, fica mais simples otimizar cada canal para o tipo e quantidade de informação que trafega nele, o que possibilita um melhor desempenho na comunicação entre os pares de núcleos. Por outro lado, por possuir estruturas dedicadas, possui uma baixa taxa de reusabilidade de seus componentes. Um exemplo de modelagem de bloco IP utilizando canais de comunicação ponto-a-ponto pode ser visto na Figura 2.3 (a). Pode-se observar que há um canal exclusivo ligando os pares de núcleos de forma que os dados que trafegam por ele somente interessam ao dois núcleos em que estão interligados, motivo pelo qual nesse tipo de configuração os processo de otimização tendem a ser mais simples.

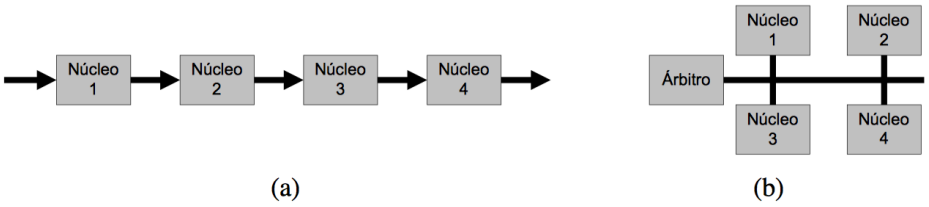


Figura 2.3: Arquiteturas de comunicação: (a) ponto-a-ponto e (b) multiponto (Fonte: [3]).

Canais multi-ponto: Geralmente usado em sistemas cujo foco é a troca de informações de espaço de endereçamento, como em processadores, memórias e dispositivos de entrada e saída, canais multi-ponto são estruturados em um barramento compartilhado ao qual os núcleos do sistema são conectados [20]. Dessa forma, os diferentes núcleos do bloco IP disputam o controle e a banda alocada ao barramento de comunicação, e, portanto, suas requisições de comunicação podem não ser imediatamente atendidas devido as etapas de escalonamento necessárias para que todos os núcleos possam trocar informações. Esse tipo de canal possui uso mais generalista, uma vez que precisa ser capaz de transmitir dados, endereços e controle. Dentro de cada bloco IP existe um módulo central de escalonamento, denominado *árbitro*, que rege o uso e acesso ao barramento [3]. A Figura 2.3 (b) mostra um exemplo simplificado de uma arquitetura de canais multi-ponto. Pode-se observar que nesse modelo existe um único canal compartilhado que provê as trocas de informações entre todos os núcleos do bloco IP.

Cada tipo de canal possui vantagens e desvantagens, não sendo possível afirmar que um deles é melhor que o outro, pois a resposta depende da finalidade do bloco IP. Para fins de

análise, podem ser utilizadas métricas imparciais para caracterizar os dois tipos de canais de conexão: paralelismo, consumo de energia, frequência de operação, escalabilidade, área e reusabilidade [3].

Segundo [3], temos que:

- Canais com conexão ponto-a-ponto fornecem um maior paralelismo visto que num mesmo momento diversas trocas de informações podem ocorrer de forma paralela, uma vez que os núcleos não concorrem pelo uso dos canais de comunicação dentro do sistema. Ao mesmo tempo, eles independem de algoritmos e métricas de escalonamento, o que pode ser identificado nos canais multiponto.

- Acerca do consumo de energia, canais ponto-a-ponto consomem menos energia, uma vez que a taxa de uso tende a ser muito menor do que no caso dos canais multiponto. Além disso, canais ponto-a-ponto cobrem menores distâncias e possuem menos núcleos conectados, o que coopera para o menor consumo de energia.

- A frequência de operação é definida pelos tempos de transição entre componentes no canal de comunicação. Logo, teremos tempos maiores de transição e taxas menores de operação em canais que servem a um número maior de elementos de processamento. Portanto, observa-se uma maior frequência de operação em canais ponto-a-ponto.

- Em relação à escalabilidade, um sistema é dito escalável se sua largura de banda aumenta quando a complexidade de uso e o seu uso propriamente dito também aumentam.Nesse sentido, canais multi-ponto, por serem mais generalistas, podem ser reutilizados em sistemas com propósitos completamente diferentes, o que gera menores custos, tempo reduzido no projeto e implementação dos sistemas, motivo pelo qual os canais multi-ponto são os mais utilizados atualmente [19]. Em resumo, canais multi-ponto permitem melhores índices de reusabilidade. Nos canais ponto-a-ponto, por outro lado, novas estruturas dedicadas de comunicação devem ser criadas para comunicação entre as unidades de processamento já acopladas, aumentando, com isso, o tempo de desenvolvimento e implementação do projeto. Além disso, aumenta-se a complexidade do projeto, uma vez que diferentes pares de núcleos podem requerer canais de comunicação com características únicas e completamente distintas.

- No que se refere a área ocupada, deve-se considerar o tamanho e a complexidade para a inserção dos múltiplos canais de comunicação dentro do *chip*. Em geral, conexões ponto-a-ponto ocupam uma área menor, já que os *layouts* dos blocos IP tendem a alocar os componentes o mais próximo possível com intuito de reduzir a

latência de comunicação no canal, abordagem oposta aos canais muti-ponto onde o barramento precisa englobar todos as unidades lógicas e de processamento contidas no sistema.

Assim sendo, canais ponto-a-ponto são uma opção melhor em termos de concorrência de tarefas, frequência de operação, consumo de energia e escalabilidade. Porém, são componentes de difícil reusabilidade, devido à dependência de propósito e tipo de núcleo a que estão interligadas. Sendo assim, sistemas com canais multi-ponto, embora percam na maioria dos critérios analisados, tornam-se uma opção mais interessante na medida em que permitem redução nos custos de projeto de sistemas de diferentes finalidades [19] e, por isso, atualmente, são mais empregados que os canais ponto-a-ponto.

Uma forma de minimizar as desvantagens inerentes ao sistema de barramento no caso de canais multi-ponto, é pela utilização de múltiplos barramentos e hierarquias de barramentos. Múltiplos barramentos são caracterizados por utilizarem vários canais multiponto que são compartilhados entre os elementos de processamento que são conectados a ele, conforme ilustrado na Figura 2.4 que apresenta dois tipos de arquitetura de blocos IP onde os múltiplos barramentos são conectados de forma distinta. Na Figura 2.4(a) os núcleos estão completamente conectados aos canais de comunicação: embora existam múltiplos canais de comunicação, existe uma ligação dos núcleos com cada um dos diferentes barramentos. Já na Figura 2.4(b) pode-se observar uma estrutura de barramento onde os núcleos estão conectados a apenas uma parte dos canais de comunicação, e, portanto, são nomeados como parcialmente conectados.

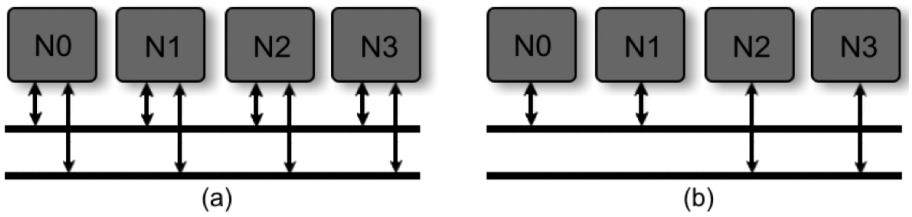


Figura 2.4: Arquitetura de comunicação com múltiplos barramentos: núcleos completamente conectados (a) e núcleos parcialmente conectados (b) (Fonte: [10]).

De maneira geral, blocos IP com núcleos completamente conectados garantem mais nodos ou *links* de comunicação no sistema, o que tende a diminuir a sobrecarga que seria observada no caso de um único barramento. Já no caso de blocos IP com núcleos parcialmente conectados ao barramento, pode-se ter diferentes configurações de interconexão, onde um núcleo pode estar ligado a apenas um ou múltiplos barramentos de conexão. Como consequência direta da existência dos múltiplos canais de comunicação, aumenta-se o nível de vazão de comunicação e largura de banda dentro do sistema.

Uma outra forma de projeto de blocos IP implementa uma hierarquia de barramentos, onde existem dois ou mais barramentos, cada um com características distintas, interconectados por um circuito ponte (*bridge*), como ilustrado na Figura 2.5. Nesta figura pode-se identificar dois canais de comunicação (o barramento rápido e o intitulado lento) conectados entre si por meio de uma ponte. A nomenclatura "rápido"e "lento"encontrada na Figura 2.5 busca mostrar que os dois barramentos em questão tem características físicas de largura de banda ou qualidade de transmissão distintas e podem coexistir em um mesmo bloco IP sem que se perca a viabilidade de uso do bloco.

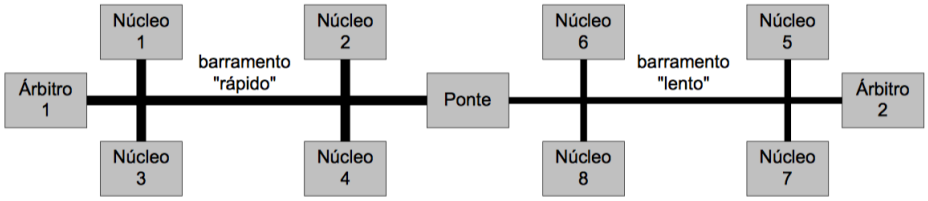


Figura 2.5: Arquitetura de comunicação com hierarquia de barramentos (Fonte: [3]).

Com a existência dos tipos heterogêneos de barramento num mesmo bloco IP é possível aumentar o número de requisições concorrentes executadas em paralelo, e melhorar o escalonamento das requisições de comunicação, uma vez que componentes de acesso/-processamento mais rápidos podem utilizar barramentos com alta velocidade enquanto componentes com nível de operação mais baixo podem ser afixados aos barramentos com velocidades menores. Entretanto, cabe ressaltar que para a efetiva implementação desse tipo de barramento heterogêneo é preciso isolar elementos de processamento mais lentos daqueles de maior velocidade, pois caso isso não seja feito, pode-se ter uma enorme perda de performance no bloco IP se os núcleos mais rápidos ficarem ociosos ou utilizarem os barramentos mais lentos, incapazes de prover o nível de comunicação requerido.

Assim sendo, devemos buscar uma arquitetura que possa prover ao sistema paralelismo com o menor consumo de energia possível, maior frequência de operação possível, escalabilidade, menor área utilizada e reusabilidade [3]. Uma das implementações de arquitetura de canais de comunicação em sistemas integrados capaz de atender a esses requisitos é a chamada Rede em chip (*Network on Chip*) (NoC), que é composta por uma rede de interconexão com chaveamento. Essa estrutura dos canais de comunicação proporciona maior flexibilidade na comunicação entre os blocos IPs, além de ser capaz de agregar características de confiabilidade de roteamento e protocolos de transferência de dados derivados de redes de interconexão e buscar o equilíbrio de desempenho de arquiteturas ponto-a-ponto com a reusabilidade dos canais multi-ponto.

2.4 *Network on Chip* (NoC) e suas características

NoC pode ser visto como um paradigma para design de SoC que tradicionalmente utiliza topologias baseadas no compartilhamento de canais de comunicação [21]. Esse paradigma surgiu como uma tentativa de redimir os problemas de comunicação encontrados nos SoCs quando se aumenta muito o número de núcleos contidos num único bloco IP. Uma vez que se aumente a integração dos SoCs, a comunicação entre os componentes do chip pode ficar bloqueada em função da concorrência nos canais de comunicação e aumentar a capacidade desses canais nem sempre é uma opção devido às restrições físicas do *chip* em que são inseridos. Sendo assim, a NoC surgiu na tentativa de superar as dificuldades encontradas na escalabilidade do SoC. Na NoC, os canais de comunicação são substituídos por uma interconexão semelhante à rede da *Internet* onde a comunicação é transmitida entre os segmentos do *chip* por meio de pacotes que trafegam pela rede, o que aumenta a escalabilidade do *chip*, embora exija mais energia e *overhead* de área devido ao complexo roteamento exigido para a transmissão das informações [22]. Como no caso da *Internet*, algoritmos de roteamento passam a ter um papel essencial na operação da rede [23].

Em outras palavras, uma NoC é composta por enlaces e roteadores que interligam os blocos IP do *chip* [6] [23]. A Figura 2.6 apresenta um diagrama generalista de uma NoC onde cada bloco IPs ali nomeados como CPU, E/S e RAM está ligado a um roteador que, por sua vez está conectado a outros roteadores por meio de canais de comunicação chamados de enlaces.

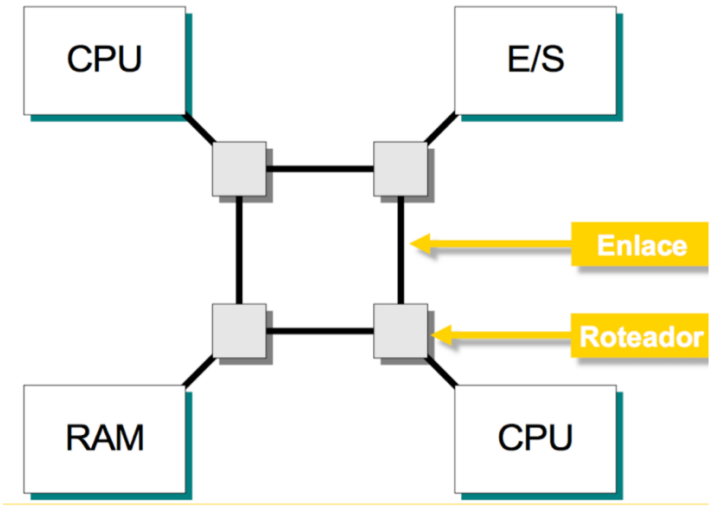


Figura 2.6: Diagrama de blocos de uma rede intra-chip simples composta por roteadores, enlaces e blocos IP (Fonte: [3]).

Os enlaces podem ser definidos como a ligação entre dois elementos de uma rede e

possuem ao menos um meio de interconexão que interliga dois roteadores entre si ou um roteador e um IP *Local* [6] [24]. Para cada canal de comunicação dos enlaces existe ao menos um barramento destinado ao transporte de dados e outros sinais úteis para detecção de erros, controle de fluxo e formatação de pacotes, por exemplo.

Os *links* podem ser do tipo *full-duplex* ou *half-duplex*. No primeiro caso, o enlace é constituído por pelo menos dois canais de comunicação unidirecionais e opostos que permitem a entrada e saída simultânea de informações no enlace (entrada e saída de informações ocorrem em canais distintos). No segundo caso, o enlace é composto por pelo menos um canal de comunicação com a capacidade de transferir pacotes de informações nos dois sentidos ao mesmo tempo (entrada e saída de pacotes ocorre em um único canal) [11]. Na NoC os enlaces mais comuns são do tipo *full-duplex*, onde os canais de comunicação possuem sentido único para transmissão de pacotes [3].

Já o roteador pode ser definido como um componente constituído por quatro elementos fundamentais: núcleo de roteamento, lógica de roteamento e arbitragem, portas e *buffer*s [11], cuja função principal é direcionar os pacotes de mensagens para outros roteadores até que cheguem a seu destinatário. O núcleo de roteamento liga as portas do roteador e é basicamente uma matriz de interruptores. A lógica de roteamento e arbitragem é responsável por definir como os pacotes serão transferidos desde origem até o seu destino. Já as portas são pontos de acesso disponíveis para que outros nodos e nodos terminais possam estabelecer conexão. Por fim, os *buffer*s são os componentes que armazenam temporariamente a informação que trafega na rede.

O compartilhamento e transmissão de informações entre os nós ou elementos dentro da rede se dá através de mensagens que são quebradas em porções menores, chamadas pacotes, e transmitidas pela rede por meio de uma estrutura de enlaces e roteadores [11]. Mais precisamente, quando um bloco *IP* (origem), deseja enviar uma mensagem a um outro elemento (destino), a origem gera os pacotes de mensagem e os transmite para o roteador por meio da sua Interface de Rede (*Network Interface*) (NI). O roteador, então, envia os pacotes para outros roteadores da rede e assim sucessivamente, até que o destino receba as mensagens ou até que se identifique que o destinatário não pode ser alcançado. Tais redes têm mostrado bons resultados quanto ao paralelismo, consumo de energia, frequência de operação e escalabilidade, o que os torna reutilizáveis.

A Figura 2.7 ilustra um exemplo de sistema onde as interconexões entre os elementos de processamento utilizam uma rede NoC. Pode-se observar que o sistema mostrado nesta figura é um MPSoC distribuído numa grade de 6x6 PEs, onde cada elemento de processamento esta ligado ao barramento que utiliza uma estrutura NoC. No caso específico deste exemplo, cada PE possui conexões seus vizinhos adjacentes por meio dos roteadores existentes na rede. Assim, caso um par de elementos diretamente conectado queira trocar

mensagens basta fazer uso do seu roteador que coordenará o envio dos pacotes. Caso as mensagens a serem trocadas sejam entre vizinhos não-adjacentes o roteador enviará os pacotes de mensagem para outros roteadores da rede até que a mensagem chegue ao destino. Sendo assim, cada PE age como um nó capaz de rotear as mensagens nos mesmos moldes do que ocorre com os pacotes que trafegam na *Internet*. Cabe ressaltar que nem todo sistema NoC possui essa mesma estrutura de conexões, já que as interconexões podem ser desenhadas de formas diversas.

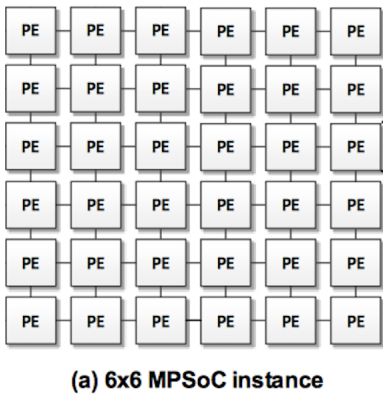


Figura 2.7: Arquitetura de comunicação em NoC organizadas em MPSoC e PE. (Adaptado de [1]).

Assim sendo, as mensagens trafegadas na rede são compostas por três partes bem definidas: cabeçalho (do inglês *header*), a carga útil (ou *payload*) e um terminador (*trailer*) [3] [11]. Vale ressaltar que esta definição é uma apenas uma generalização já que diferente sistemas podem conter outras partes além dessas três básicas:

- **Cabeçalho**: armazena os dados necessários para o controle e roteamento das mensagens, e serão utilizadas pelos roteadores durante a propagação da mensagem pela rede desde o seu remetente até o elemento de destino [11].
- **Carga útil**: campo que contém as informações propriamente ditas que precisam serem enviadas [11].
- **Terminador**: inclui informações usadas para a detecção de erros e sinalizam o fim da mensagem. Em alguns casos, como no da plataforma HeMPS abordada neste trabalho, o terminador pode ser omitido sem perda de generalidade [11].

Logo, o cabeçalho e o terminador acabam por encapsular a carga útil da mensagem [11]. Ao serem divididas e *bufferizadas* para transmissão, a mensagem, conjunto composto por basicamente cabeçalho, carga útil e terminador, são divididas em pacotes que mantém uma

estrutura similar ao da mensagem original, sendo a menor unidade de informação sobre o fluxo de roteamento e o fluxo de dados da informação. Além disso, os pacotes podem ainda ser divididos em unidades menores sobre as quais são realizados os controles de fluxo, denominados unidades de controle de fluxo (*flow control units*) (flits).

Os canais unidirecionais possuem largura física medida em unidades denominadas unidade física (*phisical unit*) (phit)s. Em geral, cada flit tem o tamanho de um a quatro phits, tamanho mínimo necessário para suportar pelo menos a informação necessária ao roteamento do pacote, ou seja, o cabeçalho [3]. Sendo assim, o phit indica a quantidade máxima de bits que podem ser transmitidos no canal de forma paralela. A quantidade de phits presente no canal determina, então o tamanho dos flits. Em resumo, cada pacote transmitido na rede pode ser visto como um conjunto de flits, onde cada flit representa um conjunto de phits.

Como pode-se notar, mensagem, flits e phits estão interligados e a relação entre eles pode ser vista na Figura 2.8. De forma resumida pode-se afirmar que uma mensagem é composta por n pacotes que, por sua vez, são compostos por m flits. Cada flit é, então, formado por k phits que são transmitidos pelo canais físicos de uma rede.

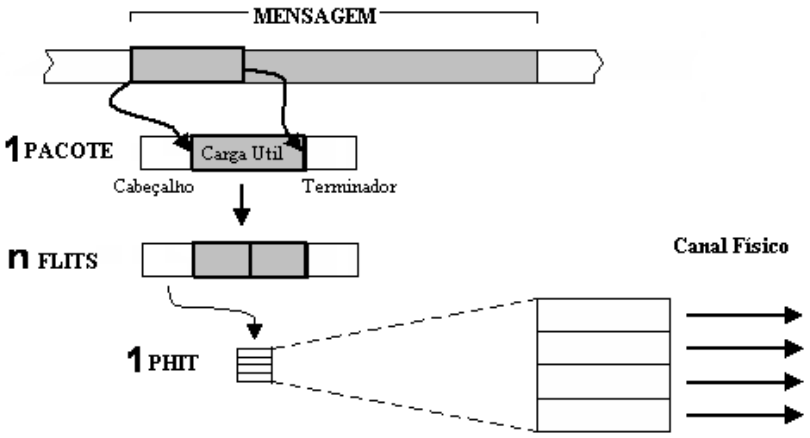


Figura 2.8: Estrutura e hierarquia de uma mensagem (Fonte: [11]).

Os enlaces determinam como os roteadores da NoC são interconectados, o que por sua vez define a topologia da rede, ou seja, a forma como os conjuntos de elementos de processamentos estão conectados nos meios de comunicação na rede [11] definem o padrão estrutural da rede. NoCs podem apresentar diferentes formas de interconexão e, com isso, diferentes topologias, sendo os dois grupos principais classificados como: *redes diretas* e *redes indiretas* [3].

Redes Diretas : nas redes diretas, cada unidade de processamento contido no sistema, geralmente, contêm um roteador que se relaciona a um processador, podendo ser

vistos como um único elemento dentro do sistema, o qual podemos definir pelo termo *nodo* [3].Estes elementos denominados nodos possuem interconexões ponto-a-ponto diretas um número definido de nodos adjacentes, sendo que mensagens entre nodos que não são adjacentes devem passar por um ou mais nodos intermediários, respeitando-se o fluxo de roteamento do pacote dentro da sub-rede gerada para seu envio no que se refere ao seu destino. Apenas os roteadores se envolvem nessa comunicação, sem a presença de outros módulos como processador e sempre procurando se adequar e seguir as métricas estabelecidas pelo algoritmo de roteamento onde o roteador o utiliza para decidir seu o nodo adjacente ou vizinho a ser repassado a mensagem [3].

No que tange a conectividade, a rede direta ideal é completamente conectada, porém, sua escalabilidade é limitada a quantidade de nodos presentes nela, uma vez que seu custo é inviável para um grande número de nodos. Como solução aos problemas e limitações deste tipo de topologia, outras alternativas surgiram com objetivo de manter uma proporção aceitável no que se refere a custo e desempenho. Com isso, surgiram as redes diretas ortogonais, tais como a grelha (*mesh*) n-dimensional, o toróide (*k-ary n-cube*) e o hipercubo, todas exemplificadas na Figura 2.9 [11]. A rede NoC a ser utilizada neste trabalho é uma rede em malha 2-D chamada *Hermes Multiprocessor System* (HeMPS) que será exemplificada e caracterizada em futuros tópicos.

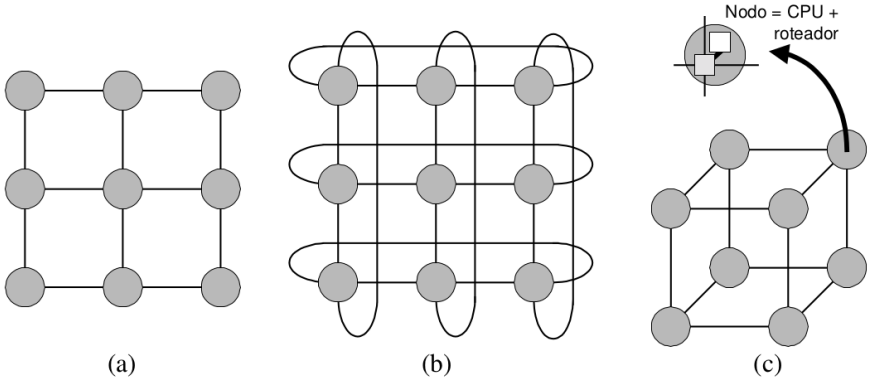


Figura 2.9: Topologias de redes diretas: a) Rede em malha ou grelha 2-D; b) Rede toróide 2-D; c) Rede hipercubo 3-D (Fonte: [11]).

Redes Indiretas : nas redes indiretas, diferente do que ocorre nos nodos das redes diretas, os elementos contidos na rede não possuem seus roteadores acoplados ao seu processador. Nestas redes, os nodos, definidos como a interconexão entre processador, módulos de memória ou computadores completos, possuem uma interface de

rede que é utilizada para comunicação entre outros roteadores conectados à rede do sistema [3]. Cada roteador possui canais bidirecionais, ou seja, que enviam e recebem mensagens em um mesmo canal de comunicação. Porém vale ressaltar que só alguns roteadores se conectam ao nodo desejado e, com isso, apenas um número reduzidos de nós podem atuar como origem ou destino de uma mensagem [3].

A topologia da rede, então, acaba por ser definida pela forma a qual são conectados esses roteadores, sendo mais comuns as topologias de barras cruzadas (*crossbar*) e as redes multi-estágio. Dependendo da forma em que as conexões são realizadas, a topologia da rede também pode ter a classificação de dinâmica ou estática [11]. As redes estáticas são definidas por conexões ponto-a-ponto estáticas entre um número bem definido de nodos. Como exemplo, temos as topologias hipercubo, grelha, e toróide. As redes dinâmicas utilizam canais roteados que são dinamicamente configurados [11]. A Figura 2.10 exemplifica dois tipos de redes indiretas: o modelo *crossbar* e modelo multiestágio.

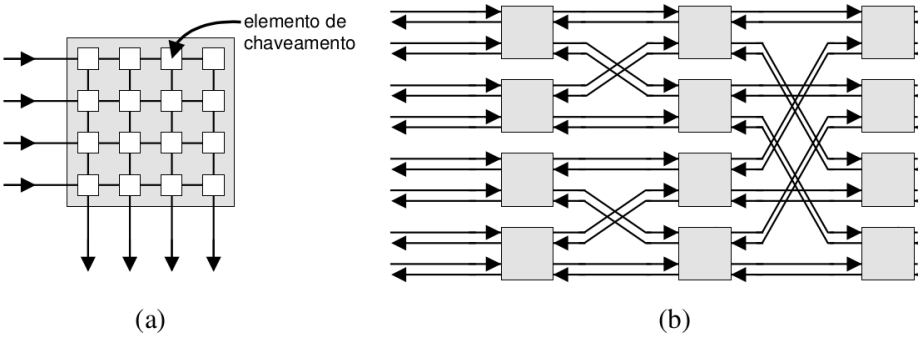


Figura 2.10: Topologias de redes indiretas: a) *crossbar*; b) multiestágio (Fonte: [3]).

Na NoC, os roteadores são responsáveis pelo controle da comunicação, exemplificado na Figura 2.11. São eles que decidem quais caminhos serão tomados pelos pacotes de mensagem até que cheguem a seu destino. Cada roteador está interligado aos roteadores vizinhos por meio de canais de entrada e saída A Figura 2.11 ilustra uma unidade de roteamento de uma NoC composta por 4 entradas e 4 saídas. Genericamente falando, um processador possui controle de fluxo de entrada de mensagens ao qual está associado um componente de memorização nomeado como *buffer*, um elemento de roteamento e arbitragem que controla o envio e recebimento de pacotes e controle de fluxo de saída.

O protocolo do roteador é composto por uma série de regras definidas durante o design e implementadas no componente que definem as ações a ser tomadas quanto aos pacotes que estão chegando no roteador, como controle de fluxo, roteamento, chaveamento,

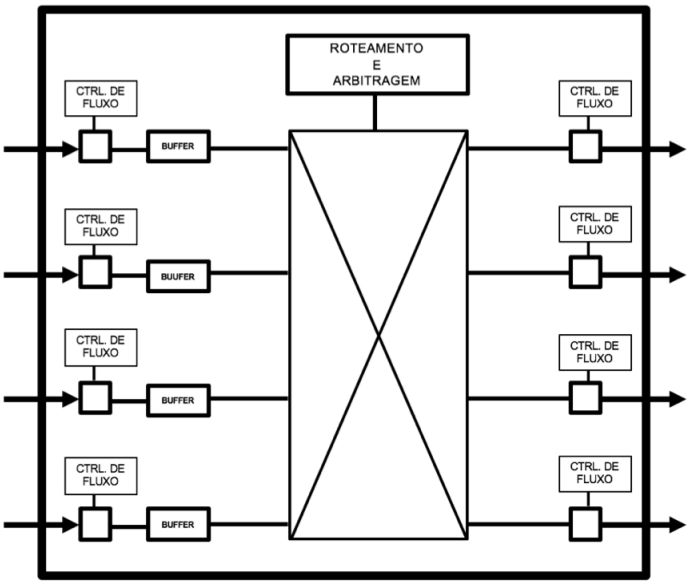


Figura 2.11: Exemplo genérico de uma unidade de roteamento de uma NoC constituída por 4 entradas e 4 saídas (Fonte: [25]).

arbitragem e memorização [3] [11]. Ou seja, o protocolo vai definir como são tratados situações como a chegada de dois ou mais pacotes de informação ao mesmo tempo, disputa de pacotes pelo mesmo canal de comunicação, prevenção de *deadlock* e *livelock*, colisão de pacotes, latência de comunicação e nível de vazão de pacotes, entre outros [26].

Dentre os mecanismos que compõem a NoC, o controle de fluxo é um protocolo responsável por verificar a disponibilidade do *buffer* de entrada do nodo destino, ou seja, ele tem o papel de verificar se o *buffer* do nodo de destino encontra-se vazio ou ocupado e se pode receber novas mensagens [11]. Essa ferramenta pode ser implementada, geralmente, de duas formas: aperto de mão (do inglês *handshake*) ou por meio de créditos [11].

- Implementação por meio de *handshake*: como o nome sugere, uma mensagem denominada *acknowledge* (reconhecimento) é transmitida ao nodo destino para verificar se o mesmo pode receber pacotes [11].
- Implementação por meio de créditos: o nodo de origem possui a informação da quantidade máxima suportada pelo *buffer* de entrada do nodo de destino através de um contador que representa créditos a serem utilizados e envia os pacotes de acordo. O nodo de destino ao re-encaminhar o pacote a outro nodo, ele envia uma mensagem global (*broadcast*) ao possíveis emissores do pacote para que todos saibam a quantidade de espaço livre o *buffer* de entrada ainda possui (créditos) [3].

Os roteadores das redes que adotam essa estruturação possuem uma capacidade limitada para armazenar flits, de tal forma que se o pacote possuir um tamanho maior que o *buffer*, os flits desse pacote bloqueado esperando liberação para envio podem ser armazenados em roteadores ao longo do fluxo de dados na rede. Porém, como a informação de roteamento está contida no cabeçalho dos flits, os mesmos acabam por seguir a rota determinada pelo cabeçalho ao longo da rede, não podendo, assim, ocorrer a multiplexação ou chaveamento de flits de diferentes pacotes em um mesmo canal, já o pacote precisa enviar toda sua informação antes de liberar o canal para envio e recebimento de outros pacotes, aumentando, com isso, os riscos de erros no sistema com a ocorrência de *deadlocks*, *livelocks* e *starvation* [11], problemas que são definidos como:

- *Deadlock*: estado onde existe uma dependência cíclica entre os roteadores presentes na rede que fazem parte do fluxo de dados, bloqueando o canal. Ocorre baseado nas características de fluxo e algoritmo de roteamento dentro da rede, podendo ser evitada como medidas preventivas antes do início da transmissão e possíveis algoritmos para recuperação de estados na rede [11].
- *Livelock*: estado onde um pacote se propaga indefinidamente dentro da rede porque os canais necessários para a conclusão de sua comunicação nunca se encontram livres para uso. Quando isto ocorre, torna-se preciso conter a quantidade de operações de controle com o uso de desvio [11].
- *Starvation*: estado onde dois ou mais *buffer*s de envio possuem um mesmo *buffer* de destino e, devido ao acesso concorrente, podem chegar a nunca conseguir enviar a mensagem, permanecendo permanentemente bloqueados visto que os recursos necessários para o envio de uma mensagem a partir de um determinado elemento podem estar sempre dedicados ao envio de outros pacotes. Para se evitar que pacotes fiquem eternamente esperando liberação para serem transmitidos, algum tipo de controle de fluxo de recebimento deve ser aplicado para melhor selecionar os *buffer*s de envio autorizados a se conectar aos *buffer*s de destino. Exemplos de algoritmos que implementam esse controle são: prioridades fixas, prioridades dinâmicas, escalonamento por idade (*deadline*), FCFS (*First-Come-First-Served*), LRS (*Least Recently Served*) e RR (*Round Robin*) [11]. Os algoritmos citados podem ser definidos como:
  - Prioridades estáticas: uma classificação constante e pré-definida de grau de prioridade é atribuída a todos os pacotes que passam pelo canal de comunicação. Pacotes com prioridade mais alta têm preferência de envio e, portanto, dependendo do fluxo de dados na rede, requisições com prioridade baixa podem vir a sofrer de *starvation*.

- Prioridades dinâmicas: os pacotes que fazem requisições ao árbitro possuem valores de prioridade dinâmicos e parametrizáveis. Dependendo do tráfego observado na rede, podem ocorrer ciclos nas requisições de envio de pacotes enquanto a informação de roteamento ainda está sendo trocada entre os roteadores. Quando isso ocorre, pacotes para um mesmo destino podem ser enviados por rotas diferentes e *links* bidirecionais podem ser tratados de forma distinta, o que pode confundir o gerenciamento da rede e criar tráfego adicional na rede.
- Escalonamento por idade (*deadline*): O processo de *aging*, ou escalonamento por idade, se refere ao tempo de espera e tempo de uso de processamento de uma dada de requisição, onde se procura evitar *starvation* ao permitir que requisições há muito tempo aguardando autorização para envio de pacotes tenham maior prioridade, o que facilita o atendimento de tais requisições.
- *Round-Robin*: os pacotes que possuem valores de prioridades dinâmicos atribuídos e gerenciados pelo arbitro. Assim, ao terminar o tempo de uso de processamento, denominado *quantum*, a requisição recebe uma prioridade baixa e vai para o final da fila de processamento. Torna-se uma boa técnica no que se refere a *fairness*, ou seja, ao uso igualitário dos recursos de processamento, porém ainda pode acarretar em *starvation* se ocorrer um intenso fluxo de pacotes no canal de comunicação.
- FCFS (*First-Come-First-Served*): as ordens vindas de requisições vão ser processadas e manipuladas em ordem de chegada, o que podem acarretar em *starvation* para um fluxo intenso de pacotes visto que um pacote nunca terá chance de ser manipulado.
- LRS (*Least Recently Served*): Similar ao *Round Robin*, tenta tornar igualitário o uso de processamento onde as requisições vão recebendo diferentes prioridades no que se refere ao tempo ou número de vezes que teve acesso ao processamento, porém não está livre de *starvation* para um fluxo intenso de pacotes.

A arbitragem pode ser definida como a tarefa responsável por definir qual elemento da rede pode utilizar os canais de comunicação para envio de mensagem a cada tempo. Sua função principal é garantir que as transmissões de dados ocorram sem conflitos devido à concorrência por recursos e de forma tal que nenhum elemento fique eternamente esperando a sua vez de ocupar os canais de comunicação. Dessa forma, o roteamento atribui uma prioridade a cada requisição de recursos recebida, sendo que aquelas de mais alta prioridade serão as primeiras a serem atendidas.

Nesse contexto, a *arbitragem (input scheduling)* seleciona uma das possíveis portas de entradas conectadas a um roteador de saída ou destino para utilizar os canais de

comunicação, razão pela qual esse componente se torna imprescindível na resolução de conflitos de concorrência quando múltiplos elementos desejam utilizar o mesmo canal de comunicação. De maneira geral, o arbitro soluciona os conflitos por meio de critérios e métricas específicas definidas para os diferentes algoritmos de roteamento.

A Figura 2.12 apresenta um exemplo genérico de arbitragem realizada sobre um grupo de requisições que está aguardando pelo direito de utilizar um determinado canal de comunicação. A figura da esquerda mostra que num determinado momento quatro elementos estão esperando a sua vez de enviar uma mensagem para um mesmo elemento de destino. Como apenas o elemento de destino é capaz de atender a apenas uma das requisições de comunicação a cada tempo, cabe ao algoritmo de arbitragem definir qual dos elementos será o primeiro a ser servido. A figura da direita, por sua vez, mostra que a arbitragem decidiu por conceder prioridade de envio ao terceiro elemento, que, então, pode se comunicar com o nodo de destino, enquanto os demais elementos continuam esperando a liberação do canal de comunicação para que possam enviar seus pacotes.

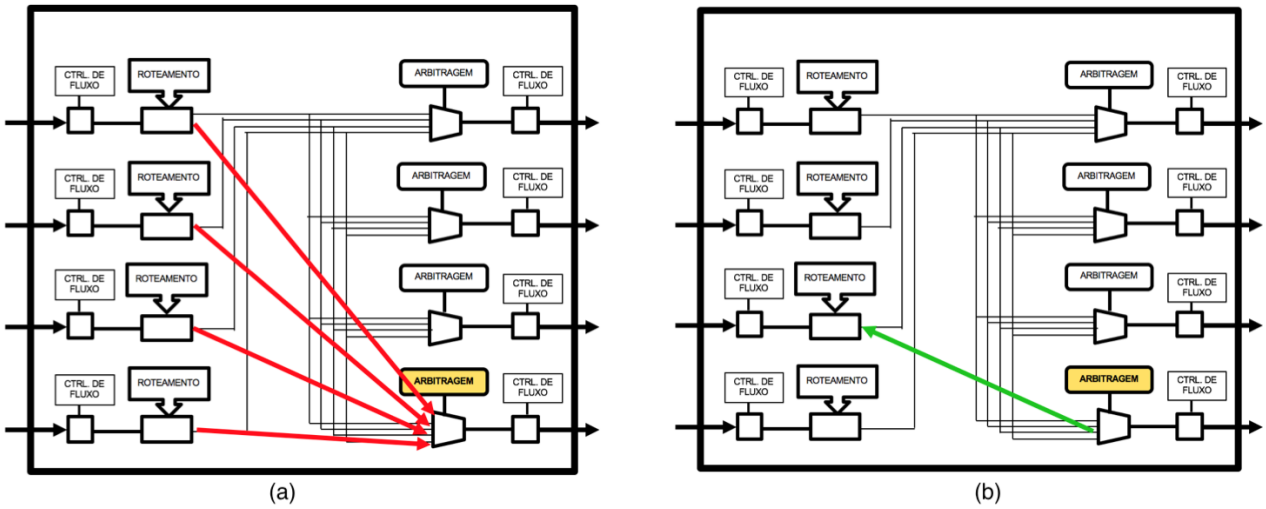


Figura 2.12: Exemplo genérico de uma arbitragem dentro de uma rede (Fonte: [25]).

Dependendo das métricas empregas, um árbitro pode ser classificado como centralizado, distribuído ou definido pelo emissor, a saber [11]:

- *Centralizado*: o roteamento e as métricas de arbitragem são processados no mesmo componente.
- *Distribuído*: o roteamento e as métricas de arbitragem são processados em diferentes componentes com intuito de que diferentes nodos de origem possuam em suas saídas módulos de roteamento e em suas entradas módulos que gerenciam a arbitragem.

- *Definido pela fonte*: o pacote armazena no seu cabeçalho toda a tabela de roteamento necessária para que chegar corretamente ao seu destino. Então antes mesmo de receber o direito ao uso do canal de comunicação, o próprio pacote contém o caminho a ser traçado dentro da rede.

Nas redes baseadas nesta técnica de envio de mensagens por flits, os *buffer*s dos roteadores têm capacidade para armazenar poucos flits, de modo que os flits de um pacote bloqueado são mantidos em diferentes roteadores na rede se o tamanho do pacote for maior que o espaço livre no *buffer* [3]. Como a informação de roteamento é incluída noflit de cabeçalho, os flits de dado devem seguir o flit do cabeçalho através da rede. Como resultado disso, não é possível realizar a multiplexação de flits de diferentes pacotes em mesmo canal lógico. Um pacote deve atravessar completamente um canal antes de liberá-lo para outro pacote. Essa é uma das principais desvantagens desta técnica de chaveamento, pois a probabilidade de ocorrer o *deadlock* é maior [3].

A performance de um roteador depende bastante do algoritmo de roteamento empregado para arbitrar o mesmo. Os critérios utilizados para a arbitragem dependem diretamente das crenças nas quais o algoritmo se baseia e são determinantes para algumas propriedades observadas nas interconexões na rede. De acordo com [25] e [3], algumas dessas características podem ser dadas por:

1. *Conectividade*: capacidade de rotear pacotes de qualquer origem para seu nodo destino.
2. *Sistemas livres de dealock e livelock*: capacidade de prevenir que um pacote não será bloqueado ou começará a se propagar na rede sem atingir seu nodo destino.
3. *Adaptabilidade*: capacidade de rotear pacotes através de rotas alternativas com intuito de prevenir congestionamentos e falhas.
4. *Controle de falhas*: capacidade de rotear pacotes quando ocorre falhas dentro de componentes.

A Figura 2.13 exemplifica o funcionamento do componente de roteamento de rede num determinado momento. Podemos observar que no momento ilustrado na figura, a porta **oeste** requisitou o envio de uma mensagem qualquer para um elemento da rede. Ao receber a requisição de envio, o roteamento analisou as portas de saída disponíveis e decidiu com base nas suas métricas que o melhor naquele dado momento seria fazer a transmissão da mensagem a partir da porta de saída **norte**.

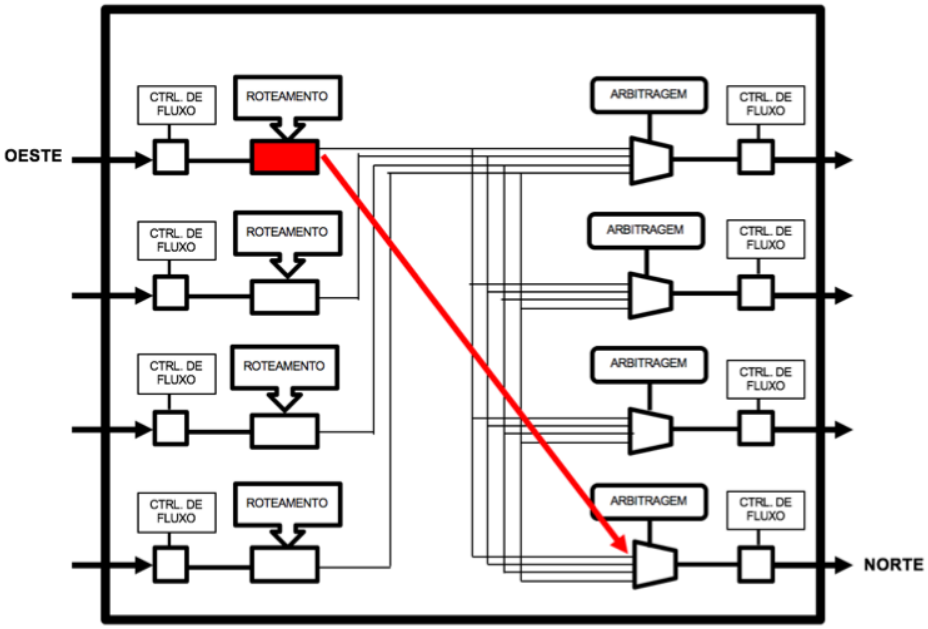


Figura 2.13: Exemplo genérico de um roteamento dentro de uma rede (Fonte: [25]).

A forma como as roteadores estão interconectados é indicada por sua topologia, cuja configuração interfere no desempenho (latência e vazão), consumo de energia, escalabilidade e custo de implementação.

Assim como ocorre numa troca de mensagens na *Internet*, a transferência de pacotes entre os nodos do chip segue um modelo cujas características podem ser associadas as diversas camadas do modelo *OSI*, especialmente as camadas de transporte, rede e enlace.

Na camada de transporte três atributos importantes são os de técnica de chaveamento, controle de fluxo e tipo de rede.

Os pacotes de dados podem ser transmitidos entre origem e destino prioritariamente de duas formas: roteamento por circuito ou roteamento por pacote.

- Roteamento por circuito: Nesse tipo de roteamento o caminho completo entre a origem e o destino das mensagens é estático e previamente estabelecido (roteadores e enlaces). Porém, tal caminho para ser criado deve ser reservado e isso acrescenta tempo de *overhead* no tempo total de envio e recebimento de uma mensagem (*handshake*). A mensagem não é enviada antes que o caminho completo da mensagem seja reservado [26]. Qualquer mensagem entre origem e destino que tente seguir caminho diferente daquele pré-definido é recusado. Esse tipo de roteamento só faz sentido no caso de ocorrerem mensagem muito longas e pouco frequentes, devido a latência inserida na comunicação na definição do caminho a ser seguido pelas

mensagens. No entanto, quando o caminho é definido, a banda de transferência é garantida em um certo sentido e, portanto, traz benefícios na troca de mensagens grandes [26] [11].

- Roteamento por pacote: Nessa abordagem as mensagens são divididas em pedaços (pacotes) e enviadas pela rede. Pacotes diferentes podem seguir caminhos diferentes, uma vez que não ocorre a reserva de canais [11]. A literatura aponta que existem várias estratégias de armazenamento e envio de pacotes, que definem a movimentação dos pacotes entre os roteadores. Os mais conhecidos são:

  - *Store-and-Forward*: o nodo encaminha o próximo pacote apenas quando e se o pacote anterior foi recebido pelo próximo elemento do caminho [11]. Dessa forma, cada nodo deve armazenar o pacote completo antes de enviá-lo ao próximo nodo. Para tanto, cada nodo precisa ter capacidade de armazenagem suficiente para que o pacote completo possa ser temporariamente guardado [26].
  - *Virtual Cut-Through*: na aplicação desse mecanismo, o nodo encaminha o primeiro *flit* do pacote de comunicação para o próximo elemento do caminho após receber deste último a confirmação de que ele pode aceitar o pacote completo que está chegando. Ou seja, os flits do pacote somente são enviados para o próximo nodo se a comunicação entre eles possa ser devidamente estabelecida.Em caso de algum tipo de bloqueio do canal, o pacote é armazenado no nodo e dessa forma, apenas ele é afetado e o canal de comunicação continua livre [26] [10].
  - *wormhole*: nessa técnica, o primeiro flit do pacote de comunicação contém as informações sobre o próximo nodo do caminho e deve ser o primeiro segmento a ser enviado para tal nodo. Uma vez que esse primeiro flit seja recebido, este é encaminhado para o próximo nodo e os flits que se seguem (que contém o corpo da mensagem) são igualmente encaminhados pela mesma rota. Nessa técnica a latência não depende da distância a ser percorrida pelo pacote e sim do tráfego entre os diferentes nodos. Ao mesmo tempo, os nodos intermediários não precisam armazenar o pacote completo, e, portanto, eles experimentam uma redução da fila de pacotes. Em contrapartida, a desvantagem é contenção de recursos em caso de bloqueio de pacotes [11]. Para que a transferência dos pacotes ocorra com sucesso, o canal entre o nodo e o próximo do caminho precisa estar livre, o *buffer* da próxima *crossbar switch* precisa ter capacidade de armazenar o cabeçalho flit e o canal precisa ter banda disponível para trafegar o flit cabeçalho. Se um dos requisitos não for atendido, ocorre o bloqueio do pacote [10].

Todo roteador com uma estratégia de roteamento por pacote, mais especificamente casos como *Store-and-Forward*, deve possuir uma maneira de armazenar dados de pacotes de um dado emissor ou nodo de origem e manter seu controle de fluxo sem perder os dados de pacotes em cada um de seus canais de comunicação de entrada [3]. Logo, uma estrutura para armazenamento temporário de blocos de dados, denominado memorização, torna-se necessário, geralmente aplicando *buffers* em memória e pode ser implementada de diferentes maneiras como memórias compartilhadas centralizadas memórias de entrada e memórias nas saídas [3]. Isto é exemplificado na Figura 2.14 [25] [3] que mostra que num determinado momento a porta de entrada mais abaixo da figura precisou parar o envio de pacotes e guardá-los temporariamente no seu *buffer* para evitar o congestionamento dos canais de comunicação, uma vez que no momento representado esses canais já estavam ocupados ou existiam muitos pacotes trafegando no barramento, o que poderia gerar perda de dados. Assim, o módulo de controle de fluxo considerou prudente aguardar um pouco antes de enviar os pacotes que recebeu.

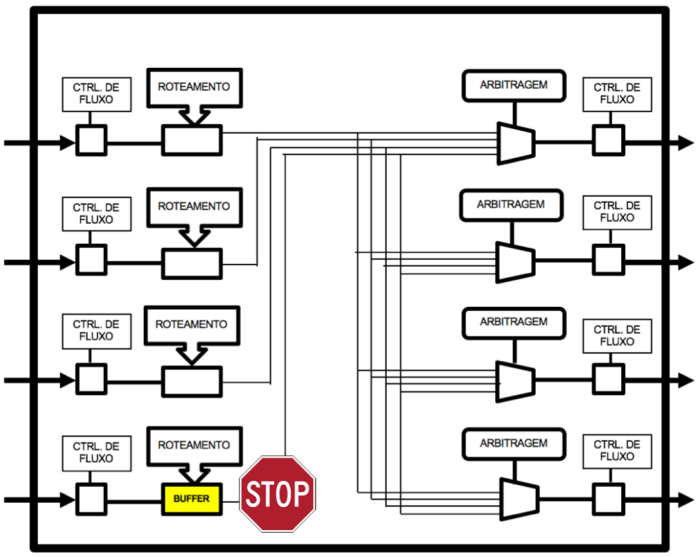


Figura 2.14: Exemplo genérico de um mecanismo de memorização dentro de uma rede (Fonte: [25]).

- Memorização centralizada compartilhada: nesta abordagem, os pacotes bloqueados nos canais de entrada são armazenados em um *buffer* centralizado, onde seu espaço de endereçamento é distribuído de forma dinâmica entre esses pacotes (*Centrally-Buffered, Dynamically-Allocated*). O *buffer* utilizado deve ser modelado para que possua 2N portas, em que N é o limiar de acessos simultâneo, tendo ainda uma porta para leitura e outra para escrita.

- Memorização na entrada: nesta abordagem, o espaço de memória é divido em forma particionada entre os canais de entrada do roteador, utilizando-se *buffers* independentes, cada qual com suas próprias entradas e podendo utilizar diferentes estratégias de escalonamento, tais como [3]:

  – *buffers* FIFO (*First-In, First-Out*): Exemplificados na Figura 2.15, esses *buffers* são a alternativa mais simples e de menor custo de implementação. Possuem um espaço fixo e bem delimitado, onde os dados são lidos na mesma ordenação na qual foram escritos. Porém, um grande problema dessa abordagem se origina quando temos pacotes bloqueados no início da fila do *buffer* e isso acaba por impedir os outros dados contidos de continuar seu caminho, mesmo com sua saída desejada disponível para uso (HOL, *Head-Of-Line*) [3], causando, assim, uma subutilização das portas de saída.

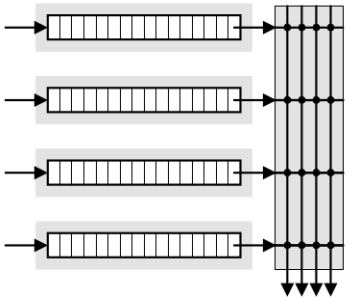


Figura 2.15: *Buffers* FIFO (Fonte: [3]).

  – *buffers* SAFC (*Statically Allocated, Fully Connected*): uma solução proposta para solucionar o bloqueio em série causado pelo HOL, definiu-se os *buffers* SAFC onde os *buffers* de entrada são divididos em N porções de tamanho $1/N$. Cada uma dessas partições deve ser alocada de forma estática numa saída. Para isso, é necessário uma topologia *crossbar* $N^2 \alpha N$ ou $N \cdot N \alpha 1$, acarretando em um controle de fluxo e gerenciamento complexos e baixa taxa de utilização [3]. Esse *buffer* é exemplificado na Figura 2.16 (a).
  – *buffers* SAMQ (*Statically Allocated Multi-Queue*): mesmo mantendo um complexo controle de fluxo e baixo índice de uso provenientes dos *buffers* SAFC, *buffers* SAMQ fazem com que as saídas dos *buffers* de entrada atribuídos a um mesmo canal de saída tenham a possibilidade de serem multiplexadas, simplificando o controle do *crossbar*. Exemplificado na Figura 2.16 (b) que tal característica decorre das interconexões de partições que estão sendo multi-

plexadas, garantindo um número maior de possibilidades de saídas para os dados [3].

– *buffers* DAMQ (*Dynamically Allocated, Multi-Queue*): visto na Figura 2.16 (c), esses *buffers* fazem com que o espaço alocado para memorização seja referenciado a um canal de entrada o qual é subdividido de forma dinâmica entre os N canais de saídas disponíveis dada o fluxo e demanda de pacotes nos canais de saída. Esta implementação é bem complexa de se modelar, porém consegue resolver os problemas inerentes aos outros *buffers* que foram explicados [3].

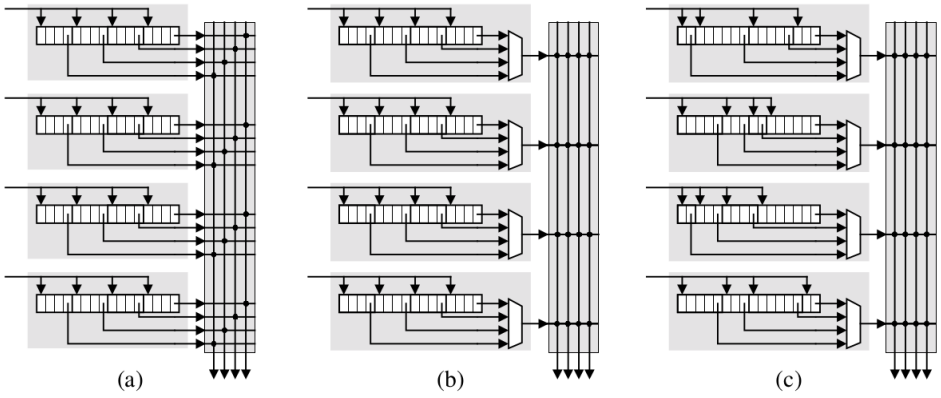


Figura 2.16: *Buffers* SAFC (a), *Buffers* SAMQ (b) e *Buffers* DAMQ (c) (Fonte: [3]).

- Memorização na saída: essa modelagem consiste em subdividir o espaço de alocação da memorização entre suas saídas, podendo ser implementado com *buffers* FIFO. Essa implementação requer um controle de fluxo maior no que se refere ao fluxo internado de dados entre as portas de entrada e saída do roteador, uma vez que cada um desses *buffers* agora devem suportar o fluxo de dados provenientes das N entradas concorrentemente, onde o *buffer* acaba por possuir N portas de escrita ou uma porta de escrita que processa as requisições que nela chegam a uma velocidade N vezes maior do que a velocidade das entradas [3].

2.5 HeMPS - MPSoC modelada por meio de NoC homogênea parametrizável

Esta seção busca caracterizar e explicar os princípios e componentes presentes na plataforma HeMPS (*Hermes Multiprocessor System*) [27], utilizado como arquitetura de MPSoC de referência neste trabalho.

A HeMPS é uma plataforma código aberto desenvolvida pelo grupo de pesquisa *GAPH*[1] [5] com intuito de implementar e avaliar o processo e desempenho de sistemas com múltiplos processadores [6]. Esses sistemas caracterizam-se por possuir uma arquitetura composta por elementos de processamento (PEs) homogêneos (de mesma arquitetura) interconectados pela NoC *Hermes* [28] do tipo malha 2-D. Os PEs são agrupados, á nível de *software*, em *clusters*, definidos em tempo de projeto [7].

A HeMPS permite gerar MPSoCs parametrizáveis definidos em tempo de projeto. Entre os parâmetros, pode-se definir o as dimensões do MPSoC, dimensões do *cluster* e a linguagem de descrição de hardware a qual o sistema será gerado (*VHDL* ou *SystemC* à nível *Register Transfer Level* (RTL) para a descrição do comportamento do sistema em termos do fluxo do seus sinais internos). A Figura 2.17 (a) apresenta um exemplo da arquitetura da HeMPS com dimensão malha 2-D 8×8 e dimensões de *cluster* 4×4.

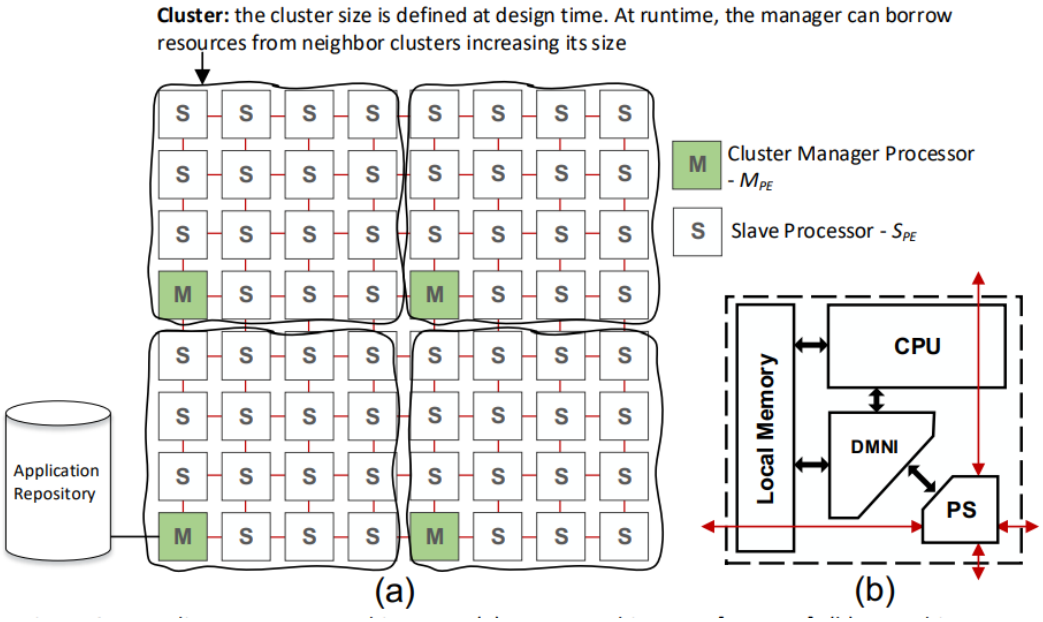


Figura 2.17: Arquitetura MPSoC exemplificando a HeMPS (a) Uma malha *mesh* 2-D 8×8.(b) elemento de processamento e seus principais componentes (Fonte: [6]).

Na Figura 2.17 (a) pode-se ver ainda que existem dois tipos de PEs (á nível de *software*):

- Elementos de processamento escravos (*SPE*s, *Slave Processing Elements*, na Figura 2.17 (a)): responsáveis pela execução de tarefas de aplicações. Possuem suporte à execução multitarefa através do algoritmo de escalonamento *Round Robin* e comunicação entre tarefas;

[1] `http://www.inf.pucrs.br/hemps/`

- Elementos de processamento gerente (MPEs, *Manager Processing Element*, na Figura 2.17 (a)): são responsáveis pela gerência do sistema. Cada *cluster* possui um MPE responsável pela gerência do mesmo. Dentre as funções de gerência de um MPE, pode-se citar o mapeamento de tarefas[2], o controle a localização de tarefas mapeadas, verificação de *deadlines* de tarefas, comunicação entre outros MPEs e SPEs, entre outros [6]. Dentre os MPEs, existe um PE específico denominado elemento de processamento gerente do sistema (SMPE, *System Manager Processing Element*) que possui todas as funções atribuídas ao MPE, porém é o único PE responsável pela gerência global do sistema. Funcionalidades específicas de um SMPE incluem selecionar um *cluster* em que uma aplicação será mapeada, controlar recursos disponíveis nos *clusters*, além de ser o único PE com acesso ao repositório de aplicação. Mais detalhes sobre SPEs e MPEs serão apresentadas na subseção 2.5.5, onde será detalhado o *kernel* da HeMPS.

Também na Figura 2.17 (a) pode-se ver o repositório de aplicações (*Application Repository*), o qual é uma memória externa, somente acessada por um SMPE, usada para armazenamento do código-objeto de tarefas de aplicações que serão executadas no sistema [28] [29]. As aplicações contidas no repositório são alocadas em tempo de projeto, isto é, no momento da geração da plataforma. A plataforma executa aplicações descritas em código *C*, seguindo o modelo descrito na subseção 2.5.4.

A Figura 2.17 (b) apresenta a arquitetura interna de um PE. O PE possui: um processador Plasma(denominado CPU na figura), melhor detalhado na subseção 2.5.1; um roteador da NoC *Hermes* (*PS* (*Packet Switching*) na figura), detalhado na subseção 2.5.2; um módulo denominado Interface de Rede de acesso direto à memória (*Direct Memory Network Interface*) (DMNI), apresentado na subseção 2.5.3 e uma memória local denominada RAM.

As características mais relevantes do MPSoC HeMPS incluem:

Processamento homogêneo : O uso de elementos de processamento homogêneos se dá, principalmente, para simplificar a migração de tarefas e geração de códigos objeto, tarefa mais complexa em elementos heterogêneos onde diferentes tipos de códigos objeto podem implicar o uso de tradução dinâmica [29].

Uso de NoC : Utiliza-se uma rede intra-chip pela escalabilidade e possibilidade de comunicação paralela e concorrente de diferentes módulos [29].

Memória Distribuída : Como cada processador possui sua própria memória privada, responsável por armazenar instruções e dados, que reduz o tráfego de dados na rede intra-chip e o uso de uma memória cache externa [29].

[2]Algoritmo que seleciona um PE do MPSoC para executar uma tarefa.

Comunicação por troca de mensagens : Além de facilitar o uso de maquinas de estados, protocolos e algoritmos para controle e monitoramento de tráfego na rede, justifica-se o uso de comunicação por troca de mensagens por ser um sistema de memória distribuída, visto que memórias compartilhadas são mais indicadas para sistemas com poucas unidade de processamento e que um barramento é utilizado como canal de comunicação [29].

Organização da memória por paginação : Apesar de simplificar o compartilhamento e migração de tarefas no sistema, além do processo de mapeamento, o uso de memória paginada pode acarretar, comparando-se com o uso de memória segmentada, restrição das tarefas ao tamanho máximo permitido pela página e ainda podendo ocorrer fragmentação interna uma vez que podemos ter tarefas menores que o tamanho designado para a página [29].

2.5.1 Processador Plasma

O processador plasma[3] se caracteriza por ser um processador 32 *bits* com instruções do tipo Computador com um conjunto reduzido de instruções (*Reduced Instruction Set Computer*) (RISC), sendo capaz de executar o conjunto de instruções original do processador Microprocessador sem estágios intertravados de pipeline (*Microprocessor without interlocked pipeline stages*) (MIPS), exceto as instruções de *load* e *store* desalinhadas, pois estas são patenteadas [9] [30]. Porém, o processador plasma possui diferenças em relação ao processador MIPS original, visto que utiliza uma organização de memória Von Neumann caracterizada, principalmente, pela utilização do mesmo espaço de memória para instruções e dados presentes no mesmo barramento, além de 3 estágios de *pipeline* para escalonamento de tarefas. Com isso, por existir apenas uma interface de acessos à memória RAM, existe um módulo de controle (*Mem_ctrl*) para melhor gerir o uso à memória RAM. O processador plasma ainda possui ainda uma unidade dedicada de multiplicação e divisão em *hardware* (*Mult*) por somas e subtrações sucessivas [9]. Um diagrama de blocos do processador pode ser visto na Figura 2.18.

Vale a pena ressaltar que o processador plasma presente na MPSoC HeMPS possui modificações se comparado com o processador plasma padrão, sendo alguns deles, manipulação e criação de mecanismos de interrupção, exclusão de módulos (Receptor/Transmissor Universal Assíncrono (*Universal Asynchrounous Receiver/Transmiter*) (UART)), adição de novos registradores mapeados em memória com intuito de prover comunicação rápida a informações de interrupção e comunicação, inserção de um mecanismo de pagi-

[3]Implementação adaptada de https://opencores.org/project/plasma

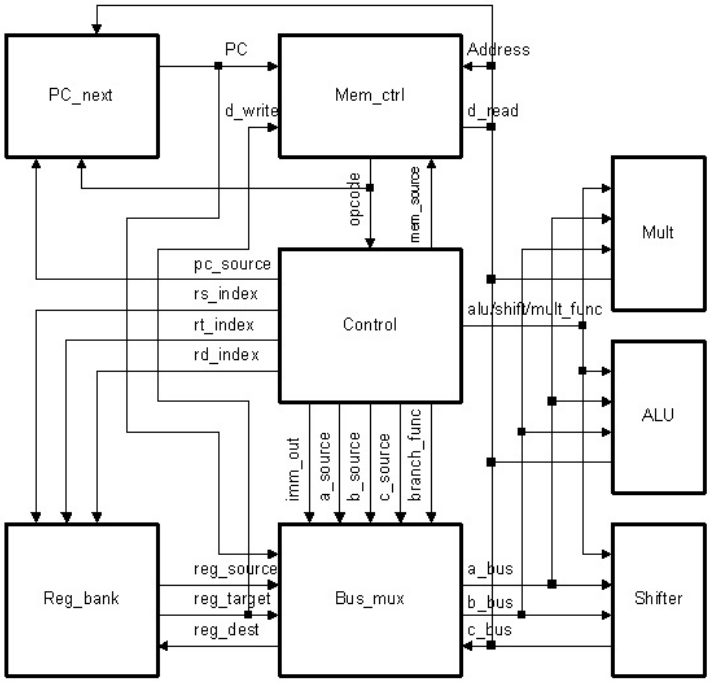


Figura 2.18: Esquematização caracterizando o processador Plasma (Fonte: [30]).

nação interno ao processador, inserção da instrução *syscall* para lançar interrupções via *software*, dentre outros [29].

2.5.2 Infra-estrutura para redes bidirecionais Hermes

A interconexão dentro da MPSoC HeMPS é feita através da rede intra-chip Hermes, caracterizada por ser uma infra-estrutura em malha *2D* parametrizável com chaveamento de fluxo para os pacotes transmitidos entre os roteadores por meio do modo *wormhole*, além de controle de fluxo baseado em créditos e baixa sobrecarga de área, podendo ser aplicada ainda para diversas topologias de redes, tamanhos de flits, profundidades de *buffer* e algoritmos de roteamento [28] [29]. A Figura 2.19 exemplifica a organização e arquitetura interna da Hermes.

A topologia em malha bidirecional é utilizada, principalmente, para torna mais simples as tarefas de posicionamento e roteamento dentro de circuitos integrados ou *FPGAs* [31].

A organização do módulos internos do roteador é imprescindível para a eficiência dentro da infra-estrutura, já que o algoritmo de roteamento escolhido, a memorização e a arbitragem podem afetar o tempo total de envio e resposta de pacotes, onde uma má escolha pode acarretar em problemas como *deadlock*, *livelock* e *starvation*. A arquitetura interna de um roteador Hermes é mostrada em forma de diagrama de blocos na Figura 2.20.

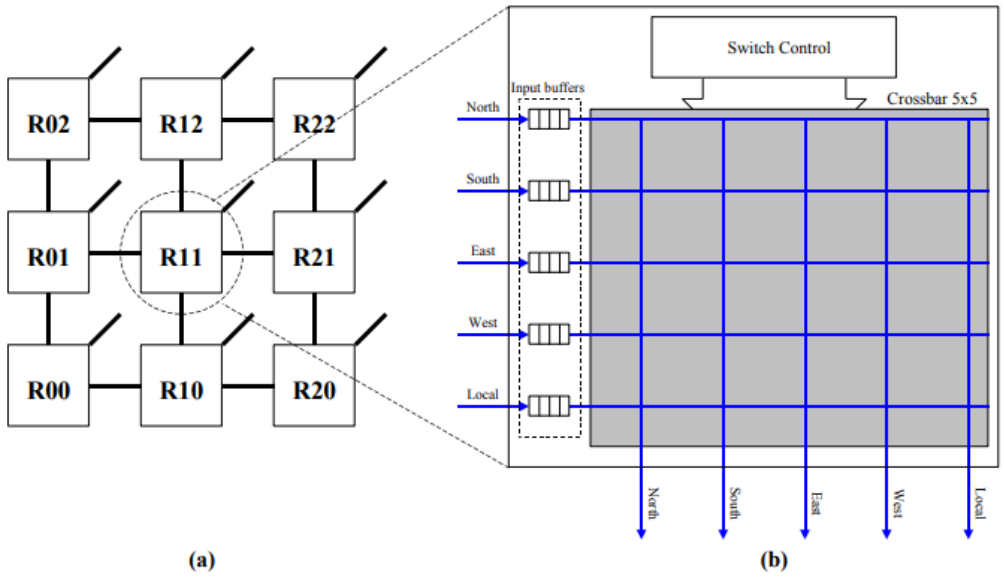


Figura 2.19: Topologia malha bidirecional 3x3. R representa os roteadores em (a). Em (b) é mostrado a organização interna de um roteador. (Fonte: [28]).

Os roteadores, como mostrado na Figura 2.19, possuem uma lógica de roteamento centralizada compartilhada por todas as portas bidirecionais, sendo no máximo cinco delas denominadas como: *East*, *West*, *North*, *South* e *Local* [31] [32]. A portal dita *Local* serve para se estabelecer a comunicação entre o roteador e seu núcleo de processamento, sendo as outras conectadas aos seus roteadores adjacentes [31]. Portas unidirecionais, tanto as de entrada como as de saída, do roteador correspondem a um canal físico [31]. Cada porta, conforme a Figura 2.20, possui um *buffer* de entrada com intuito de diminuir perda de flits de pacotes devido ao fluxo de dados no canal de comunicação, uma vez que quando um flit de um pacote é bloqueado em um roteador esperando sua vez para ser processado, os próximos flits do mesmo pacote são bloqueados nesta porta, em outros núcleos ou no processador *Local*. O *buffer* funciona como uma fila Primeiro a entrar, Primeiro a sair (*First-In First-Out*) (FIFO) de forma circular, sendo seu tamanho parametrizável pelo usuário [31].

Na Figura 2.21, temos a exemplificação da interface de rede entre os roteadores. Nela vemos os seguintes sinais na porta de saída e que, de maneira análoga, podemos deduzir os sinais da porta de entrada:

- *Clock_tx*: sinal que serve para sincronizar o fluxo de dados pelo canal;
- *Tx*: indica a disponibilidade de dados;
- *Lane_tx*: indica o canal virtual transmitindo dado;
- *Data_out*: dados a serem passados na rede;

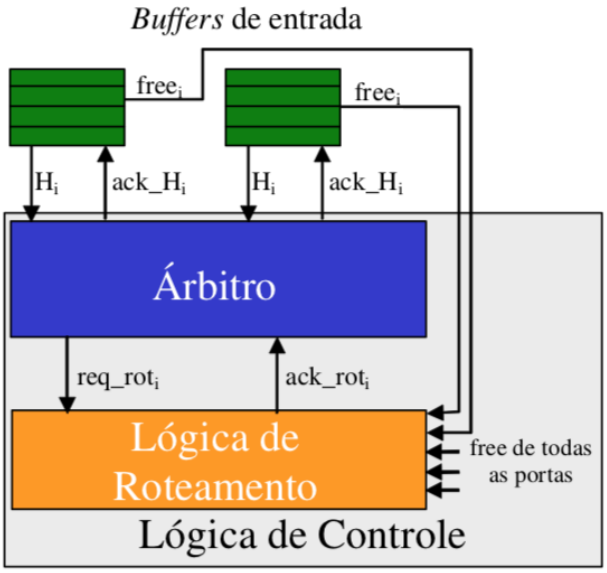


Figura 2.20: Arquitetura interna de um roteador dentro da Hermes. (Fonte: [32]).

- *Credit_in*: informa a disponibilidade de *buffer* no roteador adjacente, para cada um dos canais virtuais.

A quantidade de canais virtuais e a largura para o barramento de comunicação são configurações parametrizáveis, sendo as mesmas funções da disponibilidade de recursos para roteamento e de memória para *bufferização*, onde os *buffers* estão nas entradas dos roteadores [31].

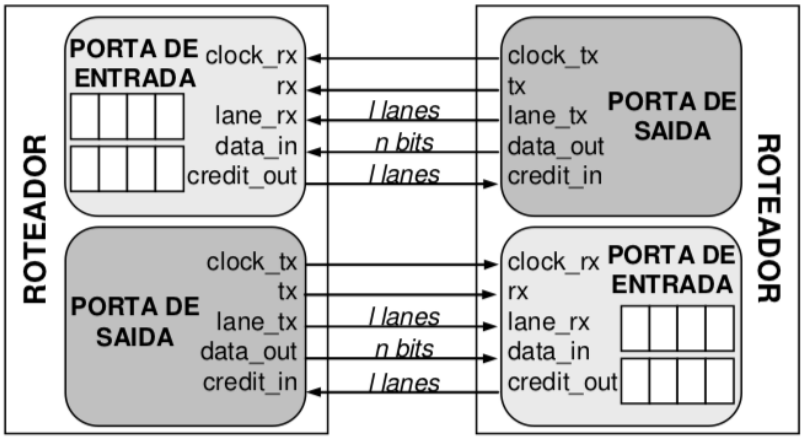


Figura 2.21: Interface de rede entre roteadores na Hermes. (Fonte: [31]).

Como dito, múltiplos flits de diversos pacotes podem chegar a uma mesma porta em um mesmo instante de tempo, com isso, de forma mais detalhada, uma arbitragem dinâ-

mica rotativa centralizada baseada em *Round Robin* garante uma atribuição justa de uso processamento. Dado a aceitação de uso pelo árbitro, o algoritmo de roteamento utilizado será o roteamento XY Determinístico. Assim como um plano cartesiano, os pacotes são referenciados sabendo sua origem e destino dentro da infra-estrutura do sistema, sendo mandados primeiramente através do direção horizontal (eixo x cartesiano) e posteriormente através da direção vertical (eixo y cartesiano), permitindo a implementação de um *crossbar* parcial [3] [31]. Os pacotes recebidos pelas portas *Local*, *East* ou *West* podem ser transmitidos por qualquer porta, exceto pela de recebimento [31]. Pacotes recebidos pela porta *North* podem ser transmitidos exclusivamente pelas portas *Local* e *South*, enquanto pacotes recebidos pela porta *South* transmitem exclusivamente para as portas *Local* e *North* [31]. A implementação de um *crossbar* parcial reduz a área do roteador em 3% comparado a um *crossbar* completo [31].

O algoritmo XY pode ter como retorno uma porta de saída livre ou ocupada. Quando uma porta de saída se encontra ocupada, todos os flits de um mesmo pacote acabam por serem bloqueados. Quando uma porta de saída acaba por ser liberada, o chaveamento (*Packet Switching*) faz com que seja estabelecida uma conexão por meio de um canal virtual entre um par de portas de saída e entrada, onde uma tabela de chaveamento é atualizada para controle e monitoramento [3] [31]. No algoritmo de roteamento XY não se necessita ordenar de alguma maneira os canais virtuais, uma vez que a ordem de percorrer a infra-estrutura, isto é, primeiro a direção horizontal e depois a direção vertical, eixo x e eixo y respectivamente, é suficiente para que não ocorra *deadlock* [31].

Para ilustrar o funcionamento desse algoritmo, na Figura 2.22 de [31] vemos uma rede Hermes 8×8 com dois canais virtuais que iremos denominar *V1* e *V2*. Supondo que:

1. *V2* será utilizado apenas quando *V1* estiver ocupado;
2. Na rede Hermes 8×8 gerada alguns canais podem estar bloqueados (marca x na Figura 2.22);
3. flits de um mesmo pacote sempre irão utilizar um mesmo canal virtual de comunicação em um mesmo canal físico, porém pode ser utilizado canais virtuais diferentes em canais físicos diferentes, demonstrado nas rotas 2, 3 e 4.

Para saber se um canal virtual de uma porta em um roteador está livre ou bloqueada é utilizada uma tabela de chaveamento, exemplificada na Figura 2.23. Basicamente, a tabela de chaveamento é composta por três entradas de valores denominadas *in*, *out* e *free*. Os valores de *in* descrevem *links* entre um canal virtual de entrada, sendo esse seu índice, e um canal virtual de saída, sendo esse seu conteúdo [31]. Os valores de *out*, análogo aos valores de *in*, descrevem *links* entre um canal virtual de saída, sendo esse

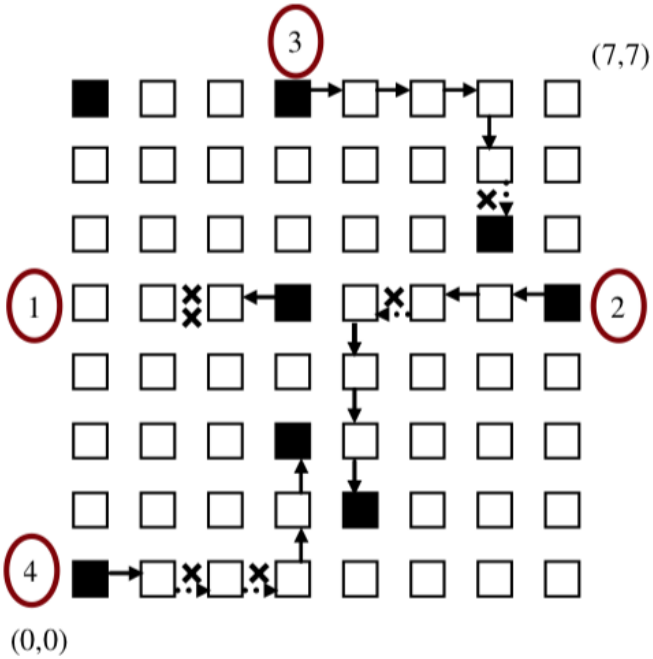


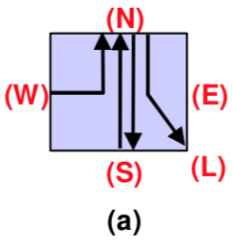
Figura 2.22: Rotas percorridas por 4 pacotes em uma rede Hermes 8x8 com dois canais virtuais (*V1* e *V2*). Quadrados representam roteadores, onde os pretos designam roteadores de origem ou destino de algum unidade de controle de fluxo (*flow control unit*) (flit) de um mesmo pacote, setas contínuas representam o canal virtual *V1*, setas pontilhadas representam o canal virtual *V2* e *x* representam canais bloqueados. (Fonte: [31]).

seu índice, e um canal virtual de entrada, sendo esse seu conteúdo [31]. Os valores de *free* armazenam o estado que se encontram os canais virtuais de saída onde valor 1 o canal está livre para ser utilizado e 0 o canal está ocupado. Dado um identificador único para os valores em *in* e *out*, denominado *id*, construído pela combinação do número ou identificador da porta, denominado *np*, da quantidade de canais virtuais para cada canal físico, denominado *ncv*, e pelo número ou identificador do canal virtual, denominado *cv*, temos a atribuição do identificador *id* apresentado na Equação 2.1.

$$id = (np \times ncv) + cv \tag{2.1}$$

As portas *East*, *West*, *North*, *South* e *Local* recebem os valores de 0 a 4, respectivamente. Já os canais virtuais recebem atribuição de *L1* a *Ln*, sendo *n* a identificação das portas de *0* a *n-1*, respectivamente [31].

Assim sendo, vemos na Figura 2.23 a tabela de chaveamento correspondente a dois canais virtuais denominados *L1* e *L2*. Assim sendo, na exemplificação da Figura 2.23 (a), temos um roteador com dois canais virtuais por canal físico, sendo a porta *North* canal virtual *L1* possui o valor de identificação 4, dado pela equação 2.1 na forma $((2 \times 2) + 0)$,



(a)

Porta	E		W		N		S		L	
VC	0 – L1	1 – L2	2 – L1	3 – L2	4 – L1	5 – L2	6 – L1	7 – L2	8 – L1	9 – L2
Free	1	1	1	1	0	0	0	1	0	1
In			4		8	6	5			
Out					2	6	5		4	

(b)

Figura 2.23: Em (a), vemos um exemplo de chaveamento dentro de um roteador. Já na figura (b), vemos a tabela de chavemento correspondente a (a). (Fonte: [31]).

e o canal virtual *L2* possui o valor de identificação 5, dado pela equação 2.1 na forma $((2 \times 2) + 1)$. Assim sendo, vemos na Figura 2.23 (b) que a saída do canal *L1* está ocupado, já que seu valor *free* está em zero, e está conectado ao canal virtual de entrada *L1* da porta *West*, já que seu valor out se encontra em 2 que seria o identificador de *L1* nessa porta. Já o canal virtual de saída *L2* está ocupado, já que seu valor free está em zero, e conectado ao canal virtual de entrada *L1* pela porta *South*, já que seu valor *out* se encontra em 2 que seria o identificador de *L1* nessa porta. Com isso, a tabela de chaveamento possui informações redundantes com intuito de melhorar a eficiência do algoritmo de roteamento [31].

Após o roteamento, um escalonador aloca a largura de banda entre os canais virtuais de cada porta de saída, onde, para cada um, se existe flits e créditos para transmissão, utiliza-se $1/l$ da largura de banda disponível, sendo l o número de canais disponíveis e se um único canal satisfaz essa restrição, o mesmo aloca toda a largura de banda do canal físico para utilizar [31].

Após o encerramento de transmissão de flits do pacote que estava sendo enviado, a conexão entre os canais virtuais de entrada e saída deve ser encerrada, sendo feito de dois modos distintos: por um *trailer* ou um contador de flits [31]. Na primeira maneira, um *trailer* precisa que um ou mais flits sejam usados como terminadores do pacote, onde uma lógica adicional é utilizada para validar o mesmo. No segundo modo, o contador de flits de um canal virtual específico é inicializado quando o segundo flit do pacote chega, visto que os primeiros são reservados normalmente para o cabeçalho de metadados do pacote, indicando o *payload* ou a carga útil de dados dentro do pacote. O contador, então, subtrai em um para cada flit comunicado sem qualquer problemas durante o processo e, quando o valor dado ao contador atinge zero, o valor *free* correspondente na tabela de chaveamento é posto em um, indicando que o mesmo está livre e, com a desalocação dos recursos utilizados, encerrando a conexão entre as portas [31].

2.5.3 DMNI - Interface de Rede de acesso direto à memória

No *framework* HeMPS, o módulo *Direct Memory Network Interface* (DMNI) é a combinação de um módulo de acesso direto à memória (DMA) com a interface de rede (NI), mostrado na Figura 2.7 (c) onde vemos o módulo integrado a um PE. A ideia fundamental é proporcionar uma interface customizada em um único módulo para sistemas com múltiplos núcleos de processamento que utilizem NoC uma interface direta entre o roteador, no caso dentro da PE, e a memória interna no componente [6]. Além disso, a DMNI habilita o envio e recepção concorrentemente de pacotes que são regidos e gerenciados por um árbitro de acesso à memória que intercala o acesso quando os módulos de recebimento e de envio estão ambos ativos [6].

Uma função importante nesse módulo é o acesso a duas porções diferentes de segmentos de memória para a transferência de pacotes [6], pois há comunicações entre elementos de processamento que podem vir de regiões distintas de memória, como, por exemplo, a estrutura do *header* e o *payload* da mensagem do pacote [6]. A Figura 2.24 mostra um diagrama sobre a arquitetura da DMNI. Como podemos ver no diagrama de blocos, o componente possui três módulos principais: o módulo de envio, módulo de recebimento e módulo árbitro de acesso à memória que fazem comunicação entre a NoC e a memória interna *Local*.

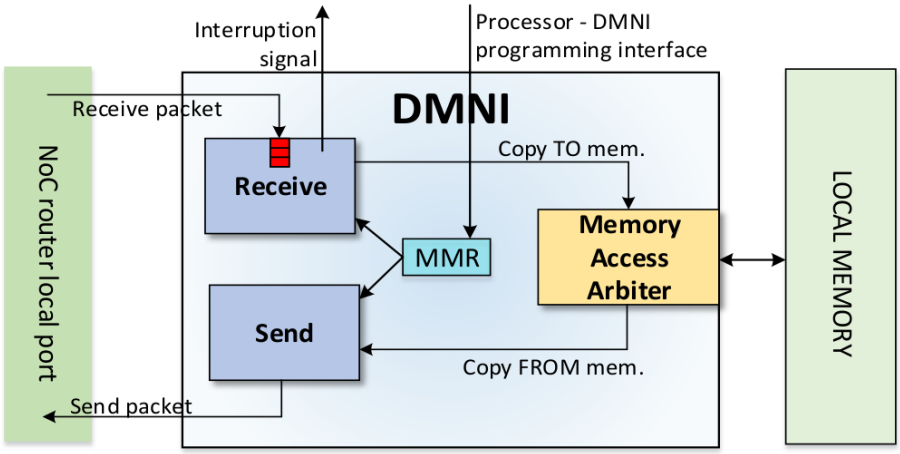


Figura 2.24: Arquitetura da DMNI (Fonte: [6]).

Estrutura do pacote na HeMPS

Como exemplificado anteriormente, para serem enviados pela NoC os pacotes dentro da HeMPS são subdivididos em unidades menores denominadas flits.

Código 2.1: Estrutura ServiceHeader para os pacotes dentro da HeMPS (Fonte: [6]).

```
1 typedef struct {
2     //!<Is the first flit of packet, keeps the target NoC router
3         unsigned int header;
4     //!<Stores the number of flits that forms the remaining of packet
5         unsigned int payload_size;
6     //!<Store the packet service code (see services.h file)
7         unsigned int service;
8         union {
9                     unsigned int producer_task;
10                    unsigned int task_ID;
11                    unsigned int app_ID;
12        };
13
14        union {
15           unsigned int consumer_task;
16           unsigned int cluster_ID;
17           unsigned int master_ID;
18           unsigned int hops;
19           unsigned int period;
20        };
21    //!<Store the packet source PE address
22        unsigned int source_PE;
23    //!<Store the packet timestamp, filled automatically by send_packet
      function
24        unsigned int timestamp;
25    //!<Unused field for while
26        unsigned int transaction;
27
28        union {
29                unsigned int msg_lenght;
30                unsigned int resolution;
31                unsigned int priority;
32                unsigned int deadline;
33                unsigned int pkt_latency;
34                unsigned int stack_size;
35                unsigned int requesting_task;
36                unsigned int released_proc;
37                unsigned int app_task_number;
38                unsigned int app_descriptor_size;
39                unsigned int allocated_processor;
40                unsigned int requesting_processor;
41        };
42
43        union {
```

```
44                  unsigned int pkt_size;
45                  unsigned int data_size;
46                  unsigned int insert_request;
47          };
48
49          union {
50                  unsigned int code_size;
51                  unsigned int max_free_procs;
52                  unsigned int execution_time;
53          };
54
55          union {
56                  unsigned int bss_size;
57                  unsigned int cpu_slack_time;
58                  unsigned int request_size;
59          };
60
61          union {
62                  unsigned int initial_address;
63                  unsigned int program_counter;
64                  unsigned int utilization;
65          };
66          //Add new variables here ...
67  } ServiceHeader;
```

A estrutura do pacote é definida pela estrutura denominada `ServiceHeader`, exemplificada no Código-fonte 2.1. Essa estrutura define o que estará presente em cada flit de cada tipo diferente de pacote. Cada tipo de pacote, bem como os flits utilizados, são definidos por um identificador em *hexadecimal* correspondente aos serviços implementados dentro da plataforma.

No Código-fonte 2.1, vemos a declaração da estrutura de um pacote dentro do sistema, onde sempre o primeiro, segundo, terceiro, quinto, sexto e sétimo flits se referem, respectivamente, ao endereço do roteador de destino dentro da NoC; o tamanho de flits dentro pacote incluindo possível *payload* menos dois, pois o mesmo desconsidera os dois primeiros flits para esse valor; o código do serviço a ser enviado definido no arquivo de módulos do sistema `services.h` cujo serviços são códigos hexadecimais definidas estaticamente como *define*; o endereço do roteador de origem dentro da NoC; o *timestamp* do pacote que é definido pela função *send_packet*() exemplificada no Código-fonte 2.2 e o flit de transação que até a presente versão da plataforma HeMPS não é utilizado por nenhum serviço [6]. Além disso, vale a pena ressaltar que, dependendo da funcionalidade, exite a liberdade de se adicionar mais flits à estrutura, desde que para isso os outros módulos tanto de *hardware* quanto de *software* sejam readequados a essa mudança. Os outros flits

são colocados em *union*, pois, dependendo do serviço, diferentes opções são utilizadas.

Do ponto de vista da NoC, o pacote possui apenas cabeçalho e uma carga útil denominada *payload*. O *header* do pacote possui o endereço do roteador de destino e tamanho do *payload* do pacote. Do ponto de vista do *kernel*, uma mensagem é divida como demonstrado na estrutura no Código-fonte 2.1, onde possui o *header* da mensagem, contendo as mesmas informações do *header* do pacote mais o *header* do serviço. Esse campo do serviço contém o código *hexadecimal* do serviço e outras informações úteis no contexto de diferentes serviços [6]. Alguns serviços possuem também um *payload*, que se caracteriza por ser também o *payload* da mensagem, caso o mesmo exista [6]. Essa estrutura pode ser vista na Figura 2.25.

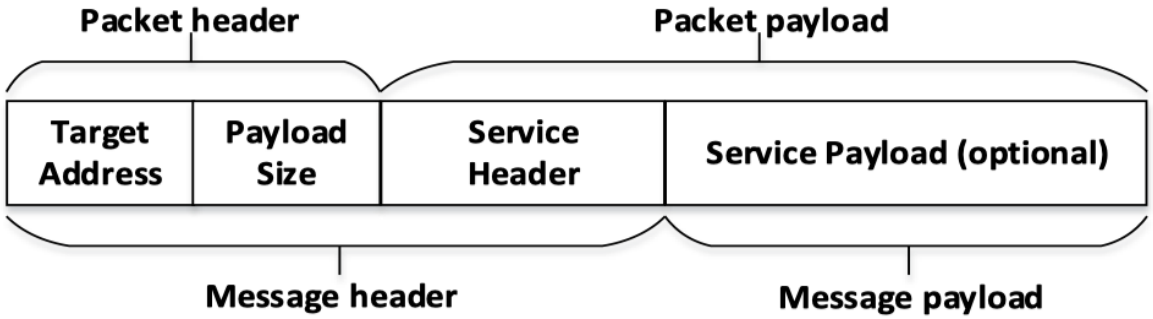


Figura 2.25: Exemplificação da estrutura do pacote e mensagem (Fonte: [6]).

Módulo de Envio de Pacotes

Exemplificado na parte esquerda da Figura 2.24, o módulo de envio se caracteriza particularmente pela possibilidade de transferência de dois blocos de memória em uma única transferência, onde sua principal função é de mandar um pacote pela NoC [6].

No Código-fonte 2.2, vemos a função de envio de pacotes `send_packet()`, utilizada para iniciar a máquina de estados de envio de pacotes da DMNI através de registradores mapeados no sistema. Ela recebe como parâmetros de entrada um ponteiro para a estrutura do pacote `ServiceHeader`, mostrada anteriormente no Código-fonte 2.1, o endereço inicial de memória para o *payload* da mensagem, caso o mesma tenha algum, e o tamanho desse *payload* representado como uma *word* de 32 *bits* [6].

Como visto na linha 11 do Código-fonte 2.2, enquanto a DMNI estiver transmitindo um pacote, dado pelo sinal `DMNI_SEND_ACTIVE` igual a 1, o processo fica ao aguardo para que possa prosseguir com o resto de suas instruções.

Nas linhas de 7 e 9, o tamanho e endereço de memória do primeiro segmento de memória são configurados para escrita, onde se a mensagem tem algum tipo de informação, dado seu *payload*.

Nas linhas de 15 a 18, o segundo bloco de memória é configurado para escrita.

Finalmente, na linha 19 temos a escrita do tipo de operação a ser efetuado pelo envio desse pacote, no caso, uma operação de leitura e na linha 20 a habilitação do sinal de início de envio do pacote para a DMNI [6].

Código 2.2: Estrutura da função de envio de pacotes processada dentro do sistema operacional, o *kernel*, do processador via *software* (Fonte: [6]).

```
1       /**Function that abstracts the process to send a generic packet
    to NoC by programming the DMNI
2       * \param p Packet pointer
3       * \param initial_address Initial memory address of the packet
    payload (payload, not service header)
4       * \return dmni_msg_size Packet payload size represented in memory
     words of 32 bits
5       */
6       void send_packet(ServiceHeader *p, unsigned int initial_address,
    unsigned int dmni_msg_size){
7               p->payload_size = (CONSTANT_PKT_SIZE - 2)+dmni_msg_size;
8               p->transaction = 0;
9               p->source_PE = MemoryRead(NI_CONFIG);
10              //Waits the DMNI send process be released
11              while (MemoryRead(DMNI_SEND_ACTIVE));
12              p->timestamp = MemoryRead(TICK_COUNTER);
13              MemoryWrite(DMNI_SIZE, CONSTANT_PKT_SIZE);
14              MemoryWrite(DMNI_ADDRESS, (unsigned int) p);
15              if (dmni_msg_size > 0){
16                      MemoryWrite(DMNI_SIZE_2, dmni_msg_size);
17                      MemoryWrite(DMNI_ADDRESS_2, initial_address);
18              }
19              MemoryWrite(DMNI_OP, READ);
20              MemoryWrite(DMNI_START, 1);
21      }
```

Na Figura 2.26, vemos a máquina de estados finitos da função descrita de envio.

O estado inicial da máquina denominado *WAIT* é o aguardo da configuração dos registradores mapeadores de memória (Registrador de mapeamento de memória (*Memory-Mapped Register*) (MMR)s) enquanto as linhas 19 e 20 não forem executadas. Quando a configuração se encerra, o estado seguinte é o denominado *LOAD* o qual a máquina pede permissão de acesso à memória ao árbitro de acesso à memória por meio de um sinal

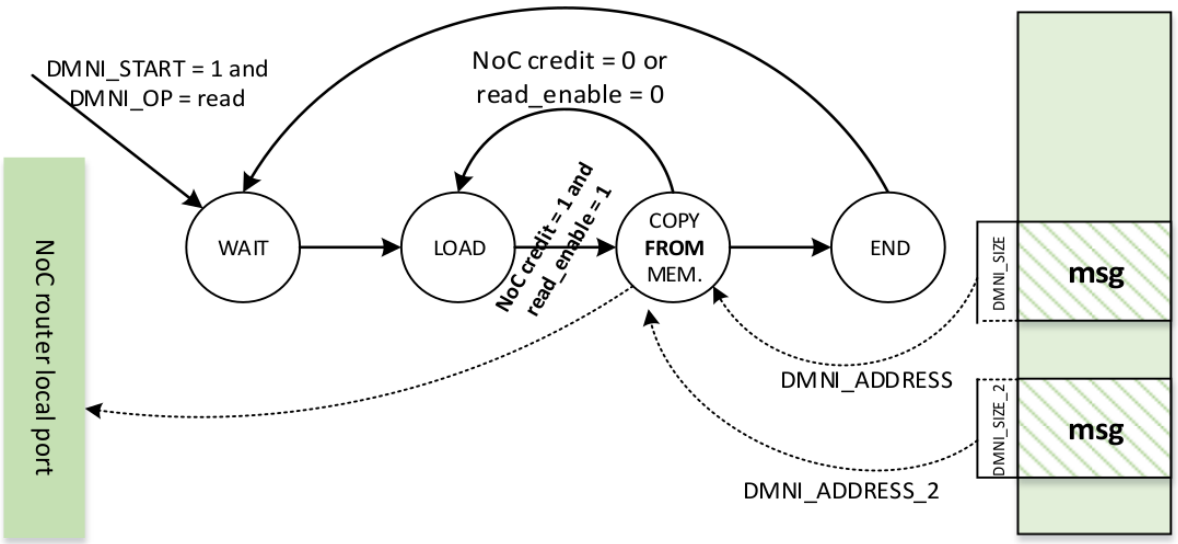


Figura 2.26: Máquina de estados finitos do processo de envio de pacotes pela DMNI (Fonte: [6]).

`send_active`. No estado de *LOAD* é verificado se a porta do roteador está habilitada a receber dados para serem enviados, dados por `credit` igual a 1, e se o árbitro dentro do módulo garantiu o acesso à escrita, visto quando o sinal `read_enable` está em 1. Ambas as condições concluídas, a máquina avança para o estado denominado `COPY_FROM_MEM` onde os dados lidos do banco de memória são mandados a porta do roteador *Local*, sempre prestando atenção que quando o árbitro ou a própria porta de transmissão do roteador *Local* encerram a comunicação, a máquina retorna ao estado de *LOAD*. Com isso, o primeiro bloco de memória é enviado e a máquina atualiza o ponteiro de endereço da memória ao segundo bloco de memória referente ao *payload* da mensagem, obviamente se o mesmo foi configurado anteriormente durante os estados, e transmite o resto necessário para a transmissão do conteúdo do pacote [6].

Módulo de Recebimento de Pacotes

O módulo de recebimento é mostrado por sua máquina de estados na Figura 2.27. Possui duas máquinas de estado com um *buffer* de capacidade de 16 unidades de controle de fluxo (*flow control units*) (flits), porém seu tamanho sendo parametrizável no projeto [6].

Ao receber os flits do pacote na porta *Local* do roteador da NoC, a DMNI armazena-os em um *buffer* local. Quando se inicia o armazenamento desses flits, ocorre um sinal de interrupção via *software* que é utilizado para avisar um PE da chegada de um pacote [6]. A segunda máquina de estados, a superior na Figura 2.27, é responsável por escrever os pacotes armazenados no *buffer* na memória local.

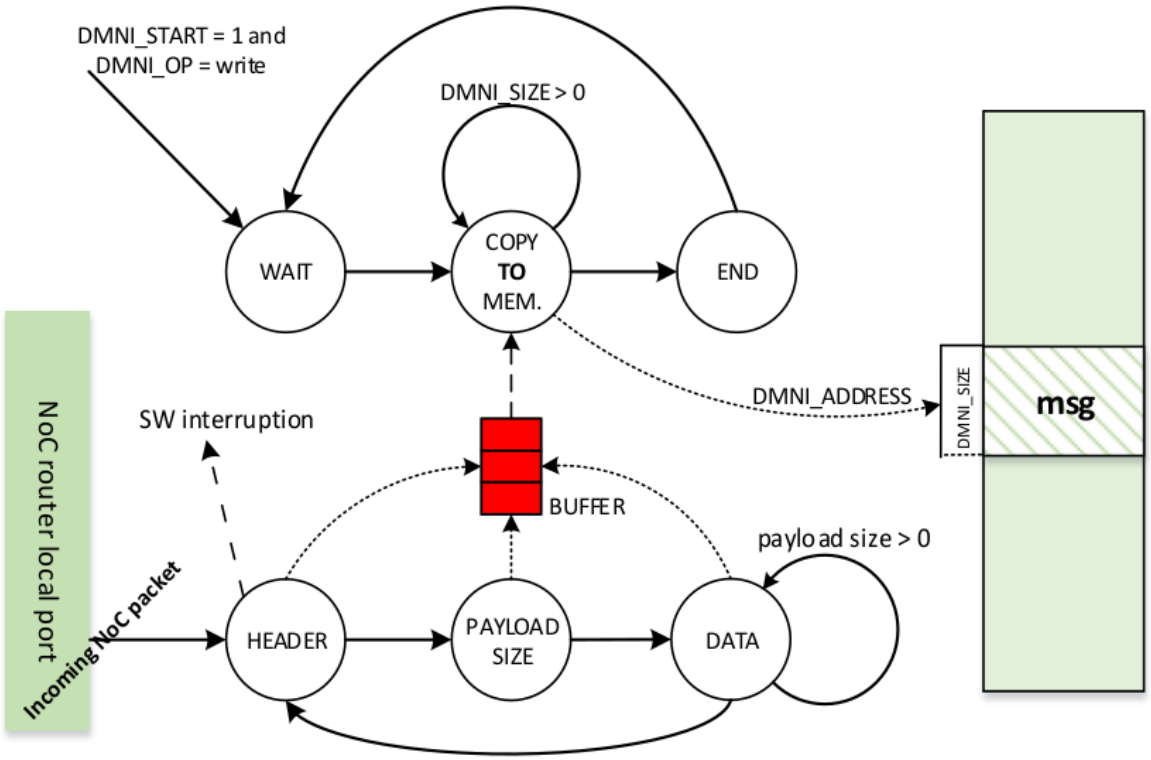


Figura 2.27: Máquina de estados finitos do processo de recebimento de pacotes pela DMNI (Fonte: [6]).

O estado `header` é responsável por armazenar os flits referentes ao cabeçalho da mensagem. O estado *`payload`*`_SIZE` é responsável por ler e atualizar o tamanho da mensagem do pacote, isto é, a carga útil no mesmo para assim avançar par o estado denominado `DATA` que se encarrega de ler os flits do pacote referentes a carga útil ou *payload* da mensagem enquanto os mesmos não alcançarem o valor armazenado em `payload_SIZE`, ou seja, através de um decrementador será lido os flits do pacote enquanto seu tamanho atualizado no estado `payload_SIZE` for maior que zero [6]. Essa máquina é ativada através da programação da DMNI, realizada na função `read packet()`, mostrada no Código-fonte 2.3.

Código 2.3: Função de recebimento de pacotes processada dentro do sistema operacional, o *kernel*, do processador via *software* (Fonte: [6]).

```
1       /**Function that abstracts the process to read a generic packet
    from NoC by programming the DMNI
2        * \param p Packet pointer
3        */
4       void read_packet(ServiceHeader *p){
5               MemoryWrite(DMNI_SIZE, CONSTANT_PKT_SIZE);
```

```
6                   MemoryWrite(DMNI_OP, WRITE);
7                   MemoryWrite(DMNI_ADDRESS, (unsigned int) p);
8                   MemoryWrite(DMNI_START, 1);
9                   //Waits the DMNI copy all data to memory before release
    the software to access it
10                  while (MemoryRead(DMNI_RECEIVE_ACTIVE));
11          }
```

Na segunda máquina de estados, a superior na Figura 2.27, é inicializando o processo de escrita, mostrados no Código-fonte 2.3 da função `read_packet()` nas linhas 5 a 8. Ao transferir os dados lidos do *buffer* a memoria *Local*, definido pelo estado `COPY_TO_MEM`, a máquina manda um sinal de permissão ao árbitro para acessar a memória *Local* por meio de um sinal denominado `receive_active`. Sendo garantido o acesso pelo árbitro através do sinal `write_enable`, o processo começa a escrever a carga útil da mensagem na posição de memória requirida, do contrário caso o acesso seja negado, a máquina de estados se mantêm no estado `COPY_TO_MEM` [6].

Vale a pena ressaltar que as máquinas acabam operando em paralelo, sendo uma que recebe os flits do pacote por meio do roteador da NoC e a segunda por meio do *buffer* para, então, armazenar na posição requirida da memória interna definida pelo ponteiro a estrutura do pacote *ServiceHeader* [6].

Código 2.4: Estrutura para alocação de espaços em memória para pacotes via *software* (Fonte: [6]).

```
1         /**
2          * \brief This structure is in charge to store a ServiceHeader
    slot memory space
3          */
4         typedef struct {
5                   ServiceHeader service_header;
6                   unsigned int status;
7         }ServiceHeaderSlot;
```

Vale a pena ressaltar que o envio de pacotes via *software* é limitado a dois espaços de memória definidos no Código-fonte 2.4 pela estrutura `ServiceHeaderSlot` [6].
Assim sendo, no Código-fonte 2.5 a função *get_service_header_slot*() acaba por retornar um espaço em memória não utilizada pela DMNI e, com isso, evitar que posições de memória sejam mudadas sem que seu conteúdo seja completamente transmitido pela DMNI, diminuindo a possibilidade de erros na aplicação no que se refere a posições corrompidas de memória e evitando erros mais severos durante a execucação da aplicação [6].

Código 2.5: Definição das função `get_service_header_slot()` para alocação de espaços em memória para envio de pacotes via *software* (Fonte: [6]).

```
1         //!<Slots to prevent memory writing while is sending a packet
2         ServiceHeaderSlot sh_slot1, sh_slot2;
3
4         /**Searches for a free ServiceHeaderSlot (sh_slot1 or sh_slot2)
    pointer.
5          * A free slot is the one which is not being used by DMNI. This
    function prevents that
6          * a given memory space be changed while its is not completely
    transmitted by DMNI.
7          * \return A pointer to a free ServiceHeadeSlot
8          */
9         ServiceHeader* get_service_header_slot() {
10                if ( sh_slot1.status ) {
11                        sh_slot1.status = 0;
12                        sh_slot2.status = 1;
13                        return &sh_slot1.service_header;
14                } else {
15                        sh_slot2.status = 0;
16                        sh_slot1.status = 1;
17                        return &sh_slot2.service_header;
18                }
19        }
```

Para exemplificação, no Código-fonte 2.6 temos uma demonstração de como é feita uma chamada de função via *software*. Na linha 3, o *header* é criado com base no endereço do roteador do PE dentro da NoC.

Código 2.6: Exemplificação de chamada de função para envio de pacote pela HeMPS via *software* (Fonte: [6]).

```
1         ServiceHeader *p;
2         p = get_service_header_slot();
3         p->header =  <coordenada_X> * 256 + <coordenada_Y>;
4         p->service = <service dentro de software/include/service.h>;
5
6         <outros campos da estrutura ServiceHeader>
7
8         send_packet(p, 0, 0);
```

Árbitro de acesso à memória

O módulo árbitro propicia o acesso concorrente à memória para o envio e recebimento de pacotes, sendo, assim, possível que os elementos de processamento recebam informação e transmitam novos pacotes para a NoC, intercalando os acessos por meio de uma rotina de escalonamento dinâmico baseado em uma fila circular baseada no algoritmo *Round-Robin*. Para fornecer essa configuração, o árbitro por meio do algoritmo controla dois sinais de controle: um para recebimento denominado `write_enable` e outro para leitura denominado `read_enable` [6]. Além disso, um *timer* denominado `DMNI_TIMER` define o tempo de *quantum* ou de acesso à memória pelo algoritmo.

A Figura 2.28 exemplifica a máquina de estados que controla o árbitro.

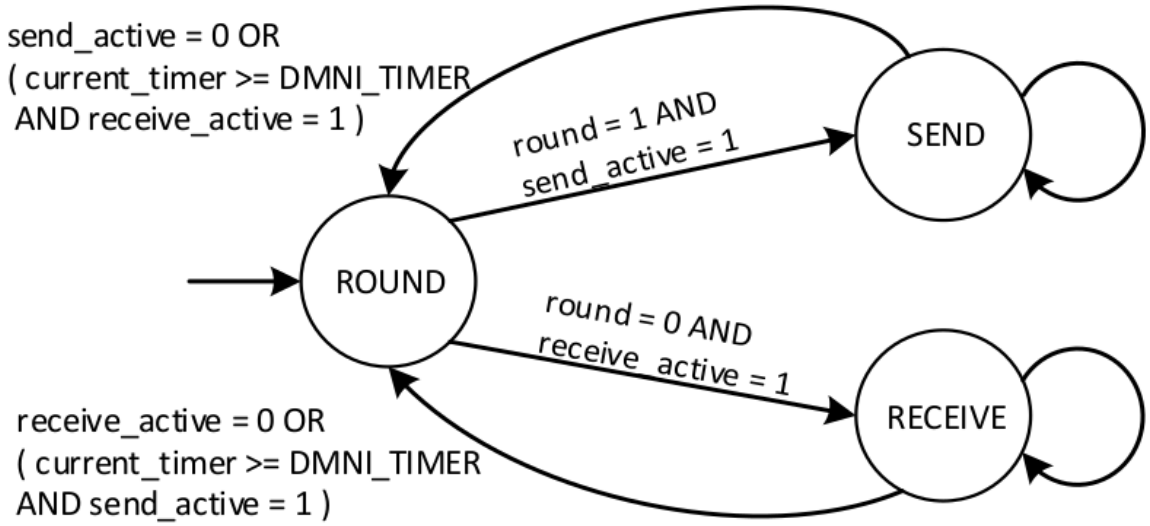


Figura 2.28: Máquina de estados finitos do módulo árbitro de acesso á memória dentro da DMNI (Fonte: [6]).

Por meio de um sinal denominado *round*, é selecionado o módulo que terá acesso a memória. As máquinas de estados que recebem e enviam os pacotes habilitam os sinais de controle `send_active` e `receive_active`, respectivamente. Se os sinais `round` e `send_active` possuem valor igual a 1, a máquina de estados vai para o estado denominado `SEND` e por meio do sinal habilitado `read_enable`, possui acesso ao banco de memória pelo tempo definido por `DMNI_TIMER`. O PH fica neste estado até o sinal `read_enable` ser desabilitado ou o tempo de acesso ter se esgotado. Vale ressaltar que o PE pode ficar no estado lendo por um tempo mais longo que seu *quantum* se nenhum outro módulo requisitar o acesso à memória [6]. O estado denominado `RECEIVE` possui comportamento análogo ao estado `SEND`, onde o mesmo vai para esse estado se

o sinal `round` estiver desabilitado em zero e o sinal `receive_active` para envio de pacotes esteja habilitado. Logo, no estado `RECEIVE` o módulo que o requisitou tem acesso ao banco de memória pelo sinal `write_enable`, e pode guardar informações provenientes da NoC. O PE permanece neste estado enquanto seu *quantum* não acabar e o sinal `write_enable` estiver ativo. Toda vez que a máquina retorna ao seu estado inicial `ROUND`, o sinal `round` recebe um valor oposto ao seu anterior, mudando a sequência de preferência de acesso do módulos à memória [6].

2.5.4 Modelagem de Aplicações

Essa seção descreve como as aplicações são modeladas no MPSoC HeMPS. A Figura 2.29 apresenta um exemplo de modelo de aplicação utilizado na plataforma HeMPS.

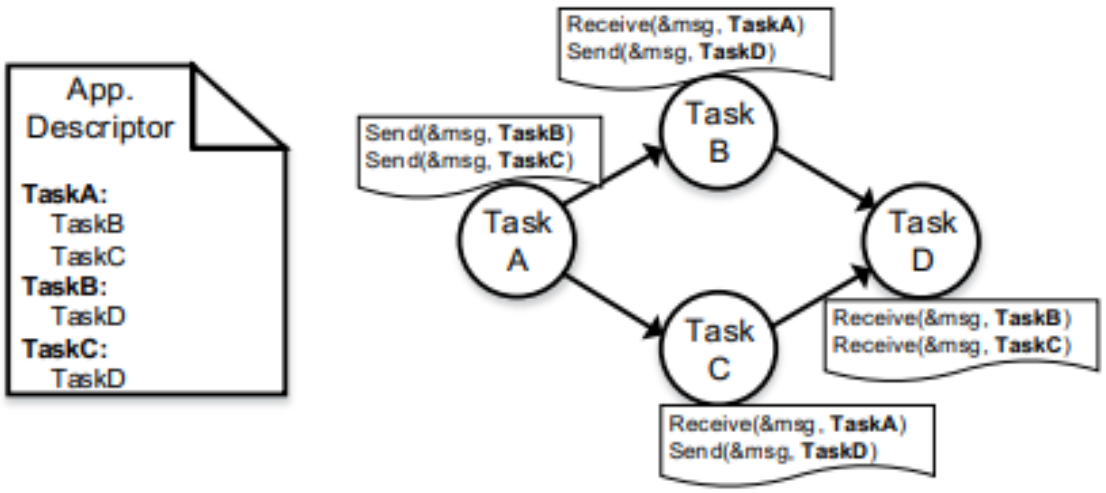


Figura 2.29: Exemplificação da modelagem de uma aplicação estruturada como um grafo na HeMPS (Fonte: [6]).

Uma aplicação pode ser modelada como um grafo de tarefas na forma mostrada na equação 2.2:

$$A = \{T, D\} \tag{2.2}$$

Onde A é um grafo direcionado no qual um vértice define uma tarefa e a direção das arestas indica o sentido da comunicação entre diferentes tarefas. Sendo assim, considere T o grupo de tarefas na forma mostrada na Equação 2.3:

$$T = \{task_1, task_2, ..., task_n\} \tag{2.3}$$

As ligações entre os pares de tarefas ilustrados na Figura 2.29 podem ser expressas pelo conjunto D motrado na equação 2.4. Esta equação é, então, dita o descritor da aplicação que contém os pares de tarefas que se comunicam [6].

$$D = \{task_A \rightarrow task_B, task_A \rightarrow task_C, task_B \rightarrow task_D, task_C \rightarrow task_D\} \tag{2.4}$$

O descritor de uma aplicação possui as informações necessárias para que as escolhas de mapeamento dentro do sistema, sejam realizadas [9]:

Um aplicação deve ser definida em tempo de projeto no *framework* HeMPS, uma vez que o usuário não tem a liberdade de definir novas aplicações em tempo de execução [6].

2.5.5 Micro*kernel* para sistemas distribuídos

Cada processador plasma presente no MPSoC possui um sistema operacional simples, denominado *microkernel* ($\mu kernel$), que dá habilidade de uma hierarquia de controle baseada em um gerenciamento distribuído, como visto na Figura 2.30 [6]. Os $\mu kernel$ dentro da HeMPS são preemptivos, ou seja, as tarefas alocadas possuem tempo de processamento bem definido e podem ter sua execução temporariamente interrompida para o início da execução de outras tarefas. Como mencionado anteriormente, o sistema possui dois tipos de *kernel*: MPEs e SPEs. Há ainda um tipo específico de MPE chamado SMPE.

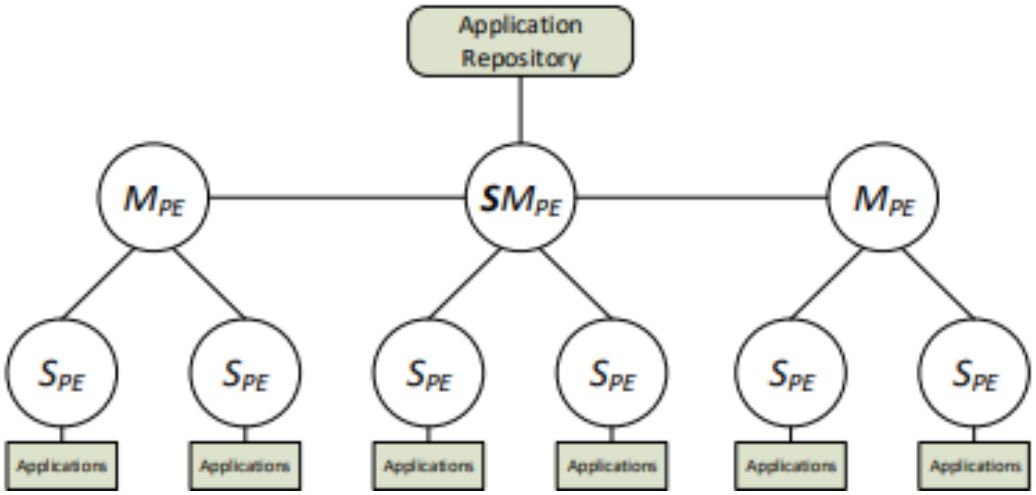


Figura 2.30: Hierarquia dos *kernel*s presente no *framework* HeMPS (Fonte: [6]).

Vale a pena ressaltar que o tamanho dos *clusters* ou conjuntos dentro da HeMPS pode ser mudado de forma dinâmica em tempo de execução se a quantidade de aplicações em execução exceder a capacidade de processamento e recursos dos *clusters* disponíveis, como

o tamanho das páginas em memória dentro de uma memória *Local* de um PE [6]. Assim sendo, essa funcionalidade é apenas possível graças a um protocolo de *re-clusterização* disponível nos MPEs para disponibilizar o empréstimo de recursos entre *clusters* [6].

A hierarquia dos procedimentos dentro do *microkernel* podem ser vistos na Figura 2.31. Nela, podemos ver os diferentes procedimentos suportados, bem como sua ordem, começando pela primeira camada até atingir a terceira camada. Os procedimentos da duas primeiras camadas são comuns para os *kernels* mestres e escravos. Porém, os procedimentos da terceira camada para o tratamento de interrupções, escalonamento e comunicação entre tarefas são feitos pelo *kernel* escravo [6].

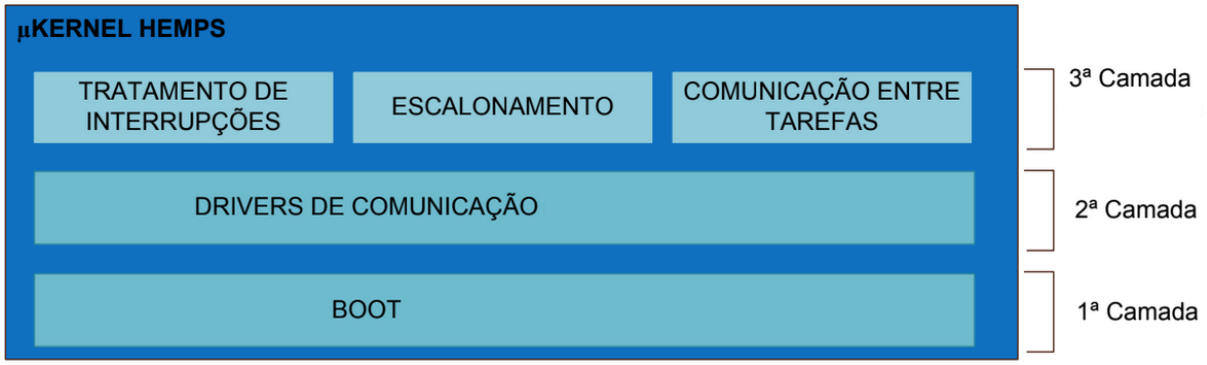


Figura 2.31: Hierarquia das tarefas desempenhadas pelos *kernel*s presente no *framework* HeMPS (Fonte: [9]).

Gerenciamento de admissão para aplicações

Os *kernel*s SMPE, MPE e SPE devem implementar um protocolo para admissão de novas aplicações para inserção de aplicações dentro do sistema, exemplificada na Figura 2.32, seguindo as seguintes etapas:

1. Quando uma aplicação manda uma requisição ao sistema para ser executada, o repositório de aplicações manda uma interrupção ao SMPE para ser tratada [6];
2. Dentro do SMPE, a interrupção é tratada e as informações do descritor D é resgatada para, através de um algoritmo de mapeamento de *clusters* cujo critério é baseado na taxa de recursos alocados no *cluster*, decidir-se qual dos *clusters* dentro do sistema é mais apropriado a receber a aplicação [6];
3. Ao ser selecionado, o MPE recebe o descritor D da aplicação vinda do SMPE e, com isso, mapeia as tarefas aos SPEs e faz uma requisição SMPE para receber os códigos objetos das tarefas [6];

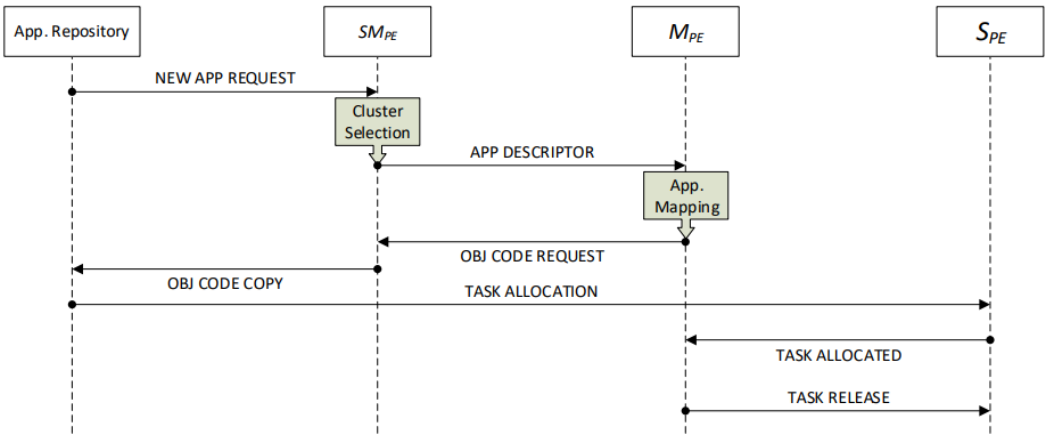


Figura 2.32: Diagrama de blocos caracterizando o funcionamento do protocolo de admissão para aplicações (Fonte: [6]).

4. Dentro do SMPE, a requisição é processada fazendo com que o repositório de aplicações seja configurado para transferir os códigos objetos para os designados SPEs [6]. Assim sendo, quando um SPE recebe um código objeto de uma tarefa, o mesmo manda uma mensagem para seu mestre MPE informando que o carregamento do código objeto foi feito com sucesso [6];

5. Por fim, o MPE recebe as mensagens de todos os seus SPEs acerca das tarefas da aplicação e, quando todas estiverem carregadas, o MPE libera a aplicação a ser executada nos SPEs [6].

Controle das tarefas em execução e estrutura de Dados Específicas do *kernel*

Para o controle da execução e comunicação entre tarefas, o *kernel* armazena o contexto das tarefas alocadas aos SPEs em estruturas denominadas bloco de controle da tarefa ou Bloco de Controle de Tarefas (*Task Control Block*) (TCB), a persistência e identificação de mensagens em estruturas denominadas *Pipe* e *Request Message* e busca de tarefas utilizando a estrutura *Task Location* [9].

Task Control Block (TCB)

Mostrado no Código-fonte 2.7, essa estrutura possui a atribuição de guardar o contexto de uma tarefa. Os valores armazenados do contexto são os valores dos registradores do processador, notando que arquitetura MIPS para o processador *Plasma* possui 32 registradores, porém apenas os registradores 2 a 31 são salvos, visto que o registrador 0 é reservado para a constante zero e o registrador 1 é reservado ao montador [9].

Além disso, são salvos a posição ou endereço de retorno de contexto da tarefa dado pelo seu *program counter* (*pc*) após o tratamento da interrupção ou chamada de sistema que ocasionou seu armazenamento, o identificador da tarefa (*id*), a posição inicial na memória da tarefa dado por *offset* que é utilizado em conjunto com o registrador base e seu *status*, os tamanhos em bytes das porções de código `.TEXT`, `.DATA` e o código em *Assembly* para a tarefa `.BSS`, um identificador para auxiliar o processador fazer requisições da tarefa dado por `proc_to_migrate`, o endereço do elemento de processamento mestre aloca para essa tarefa (`master_address`) e um ponteiro para a estrutura de escalonamento dado por `scheduling_ptr` [9].

Código 2.7: Estrutura em código C do *Task Control Block* (Fonte: [9]).

```
1       typedef struct {
2           unsigned int reg[30];
3           unsigned int pc;
4           unsigned int offset;
5           unsigned int sp;
6           int          id;
7           unsigned int text_lenght;
8           unsigned int data_lenght;
9           unsigned int bss_lenght;
10          unsigned int proc_to_migrate;
11          unsigned int master_address;
12
13          Scheduling * scheduling_ptr;
14      } TCB;
```

Ainda no bloco de controle de tarefas, temos a estrutura de dados fornecida ao escalonamento que é exemplificado no Código-fonte 2.8. Nela, são escalonadas tarefas que, após uma análise, não possuem *deadlines* ou tempo máximo de execução, caracterizando um tentativa por melhor esforço (*Best Effort*), e tarefas em tempo real que, ao oposto das de melhor esforço, possuem *deadlines* ou tempo máximo de execução bem definidos, respeitando-se seu escopo de execução dentro da aplicação e interação entre os elementos de processamento dentro da NoC [6].

Código 2.8: Estrutura usada no escalonamento de tarefas (Fonte: [9]).

```
1       typedef struct {
2               int             status;
3               unsigned int    execution_time;
4               unsigned int    period;
5               int             deadline;
6               unsigned int    ready_time;
```

```
7                int             remaining_exec_time;
8                unsigned int    slack_time;
9                unsigned int    running_start_time;
10               unsigned int    utilization;
11               unsigned int    waiting_msg;
12
13               unsigned int    tcb_ptr;
14       } Scheduling;
```

O *kernel* aloca de forma estática um vetor de bloco de controle de tarefas onde cada valor dentro dele se refere a uma páginas dentro de um sistema de paginação. O campo *status* da estrutura *Scheduling* possui a finalidade de representar o estado atual da tarefa, que de acordo com [6] podem ser:

- *READY*: A tarefa está autorizada a executar;
- *FREE*: Existe um espaço disponível para alocar uma tarefa;
- *MIGRATING*: A(s) tarefa(s) esta(ão) sendo migrada(s);
- *BLOCKED*: A tarefa está bloqueada enquanto o pacote `TASK_RELEASE` não for recebido;
- *SLEEPING*: Representa tarefas que finalizaram sua execução e estão à espera do fim do período de processamento;
- *WAITING*: A tarefa está bloqueada esperando o recebimento de mensagem de outra tarefa com que se comunica, utilizando-se o campo `waiting_msg` do Código-fonte 2.8 para esse controle.
- *RUNNING*: a tarefa está em execução, consumindo poder de processamento e mudando as seções dinâmicas de memória.

Além disso, temos ainda na estrutura o quanto de tempo a tarefa teve de execução dado por `execution_time`, o período da tarefa em ciclos de relógio (*period*), tempo máximo de execução se a aplicação for em tempo real (*deadline*), o números de ciclos para a tarefa ficar pronta para a execução no estado `READY` (`ready_time`), o números de ciclos que faltam para o término da execução (`remaining_exec_time`), o número de ciclos que a tarefa ficou sem processamento ou ociosa (`slack_time`), o tempo inicial de execução `task_running_start_time`, o uso de processador pela tarefa `utilization` e, por fim, o ponteiro para a estrutura de bloco de controle de tarefas referente à tarefa (`tcb_ptr`) [6] [9].

Quando ocorrem as rotinas para troca de contexto entre tarefas, sendo elas de salvamento ou recuperação dos valores dos campos da TCB, o conteúdo dos registradores não

devem ser modificado até ser completa sua atualização dentro da TCB. A rotina de salvamento realiza a persistência do contexto armazenando os valores dentro dos registradores na TCB que referenciam a tarefa atual e a rotina de recuperação recarrega os valores de volta para a TCB, sendo ambas rotinas descritas no montador devido a necessidade de se ter o controle sobre os valores nos registradores [9].

Pipe

O *Pipe* é uma área alocada no *kernel* destinada para a comunicação entre tarefas, sendo nela guardadas todas as mensagens das tarefas que executam em um determinado PE. Essa área é subdivida em *slots* onde cada um tem a função de guardar as informações das mensagens que passam pelo mesmo. Seu código é mostrado no Código-fonte 2.9. O *Pipe* ao ser implementado em *software* é estruturado como um vetor de acessos randômico onde essa alternativa é utilizada para se evitar problemas como *deadlocks* e *head-of-line* (FIFO) [9].

Código 2.9: Estrutura usada no *slot* de um *Pipe* dentro da HeMPS (Fonte: [9])

```
1   typedef struct {
2           unsigned int    remote_addr;
3           unsigned int    pkt_size;
4           unsigned int    service;
5           unsigned int    local_addr;
6           unsigned int    target;
7           unsigned int    source;
8           unsigned int    length;
9           unsigned int    message[MSG_SIZE];
10          enum PipeSlotStatus status;
11          unsigned int    order;
12  } PipeSlot;
```

Dentro dessa estrutura, o campo `remote_addr` se refere ao endereço do processador remoto, `remote_addr` se refere ao tamanho dos pacotes da NoC em flits, *service* o identificador do serviço sendo utilizado, `local_addr` o endereço referente ao processador local, *target* e *source* são as tarefas destino e origem, respectivamente, *length* sendo o tamanho da mensagem em 32 *bits words*, o *payload* ou a carga útil da mensagem dada pelo campo parametrizável `message[MSG_SIZE]`, onde `MSG_SIZE` é definido em tempo de compilação, *status* que demonstra se o *slot* está ocupado ou não e, por fim, a ordem da mensagem *order* para se garantir que os pacotes das mensagens sejam enviados e recebidos da forma esperada [9].

Estrutura TaskLocation

A estrutura *TaskLocation* é utilizada para se referenciar a faixa de endereço do processador na rede em relação ao identificador da tarefa, ficando assim claro ao programador o processador em que a tarefa está, visto que as tarefas conversam entre si por meio de um identificador de tarefas [9].

Estrutura *RequestMessage*

Na modelagem de comunicação dentro da HeMPS, as escritas são rotina que não causam bloqueios de processo, onde a escrita é feita diretamente pelo *Pipe*, enquanto as rotinas de leitura causam bloqueio. Nas leituras, um pacote de requisição de mensagem, denominado `request_message`, é criado e o processador destino envia ao processador de origem a mensagem do pacote que se encontra em seu *Pipe*. Do contrário, as requisições de mensagens são armazenadas dentro da estrutura *RequestMessage*, sendo, assim, consultada quando ocorre uma escrita no *Pipe* para se saber a possibilidade de envio das mensagens [9].

Ponteiro *current*

O ponteiro *current* é um ponteiro localizado na área de dados global cuja função é a indexação da TCB da tarefa em execução, sendo por ele que o armazenamento e carregamento de contexto dos registradores nas TCBs e o escalonamento de tarefas são realizados [9].

Gerenciamento da comunicação entre tarefas no sistema

Os SPEs conseguem prover a execução de várias tarefas, interrupções, armazenamento de contexto e uma interface via API que provem as primitiva de envio (*Send*), sendo a mesma bloqueável apenas se não existir mais espaço disponível para guardar mensagens dentro do *pipe*, e de recebimento (*Receive*) [6]. A primitiva de recebimento é bloqueante e é chamada pelas tarefas consumidoras que geram pacotes do tipo `MESSAGE_REQUEST` para o PE da tarefa produtora cujo o envio se dá pelo pacote `MESSAGE_DELIVERY` quando a tarefa produtora faz uma requisição para a primitiva de envio (*Send*) [6]. Quando uma tarefa consumidora chama a primitiva de recebimento (`Receive`), o mesmo vai para um estado de espera enquanto não receber a mensagem das tarefas do produtor [6].

Cada *kernel* dos SPEs possuem um *pipe* que armazena cada mensagem recebida de tarefas, gerenciando seu controle. Quando uma tarefa produtora recebe um pacote `MESSAGE_REQUEST`, o *kernel* da produtora busca se a mensagem está armazenada em seu *pipe*. Se o mesmo estiver, a mensagem é mandada a tarefa consumidora por meio de um pacote `MESSAGE_DELIVERY`, do contrário o *kernel* guarda a mensagem de requisição

para, quando a tarefa produtora chamar a primitiva de envio (`Send`), a mensagem é então enviada [6].

Vale a pena ressaltar que as primitivas (`Send`) e (`Receive`) estão a nível de aplicação, onde, ao serem chamadas, chamam as rotinas de sistema, que vão ser melhor detalhadas nas próximas subseções, `WritePipe()` e `Readpipe()`, respectivamente, que se encontram dentro do *kernel* do processador *Plasma*, sendo (`Send`) assíncrona e a (`Receive`) síncrona [29]. Essas primitivas são implementadas dessa forma, pois as mensagens acabam por serem transmitidas pela infra-estrutura do sistema apenas se requisitadas e, com isso, reduz o *overload* dentro da rede uma vez que os pacotes não ficam bloqueados [29].

Assim sendo, os tipos de mensagens ou requisições processados pelo *kernel* são [29]:

- *NO_MESSAGE*: Requisição que informa que a mensagem requisitada não existe [9];
- *MESSAGE_REQUEST*: Requisição de uma mensagem para uma tarefa que está sendo processada em outro PE entre SPEs [29];
- *MESSAGE_DELIVERY*: Requisição de uma mensagem que foi requisitado por um pacote `MESSAGE_REQUEST` para ser entregue, sendo a comunicação entre SPEs [29];
- *TASK_ALLOCATION*: Requisição feita para alocar uma tarefa requisitada em um PE no sistema, sendo a comunicação de MPE para SPE [29];
- *TASK_ALLOCATED*: Requisição feita para notificar que uma tarefa foi alocada no sistema, sendo a comunicação de MPE para SPE [29];
- *TASK_REQUEST*: Requisição feita que solicita o mapeamento de uma tarefa onde, caso a mesma já esteja mapeada, um pacote denominado *LOCATION_REQUEST* é retornado contendo a posição da tarefa requirida, sendo a comunicação de SPE para MPE [29];
- *TASK_TERMINATED*: Requisição feita para informar o fim da execução de uma tarefa, sendo a comunicação de SPE para MPE e MPE para SPE [29];
- *TASK_DEALLOCATED*: Requisição feita para informar o fim da execução de uma tarefa e pode ser desalocado a página onde a mesma estava, sendo a comunicação de SPE para MPE [29];
- *FINISHED_ALLOCATION*: Requisição que informa que o MPE terminou a alocação estática inicial das tarefas [9];
- *LOCATION_REQUEST*: Requisição feita para informar a localização de uma tarefa em específico, sendo a comunicação de SPE para MPE e MPE para SPE [29].

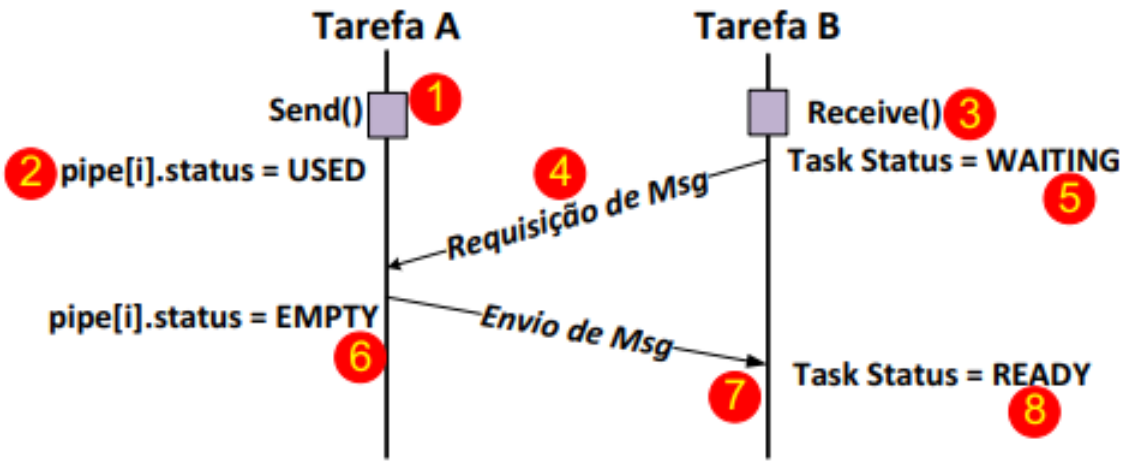


Figura 2.33: Protocolo de comunicação utilizado dentro da HeMPS, exemplificado entre duas tarefas, A e B, que se encontram em PE distintos (Fonte: [29]).

A Figura 2.33 exemplifica a inter-comunicação entre duas tarefas denominadas A e B que se encontram em diferentes pontos de processamento no sistema. Em (1), quando a tarefa A chama a rotina `Send()` para envio de mensagem, a mesma vai ser alocada no pipe e sinaliza em (2) que o mesmo está em uso, porém sendo um escrita assíncrona não-bloqueante [29]. Em (3), quando a tarefa B chama a rotina `Receive()`, podemos ter dois casos possíveis de leitura pela tarefa: no primeiro caso, a tarefa que fez o envio da mensagem se encontram no mesmo elemento de processamento e a tarefa que requisitou a rotina `Receive()` executa uma leitura no pipe local. No segundo caso, demonstrado na Figura 2.33, a tarefa de destino, denominada A, encontra-se em outro PE, logo o *µkernel* requisita um *MESSAGE_REQUEST* pela NoC (4) e a tarefa B fica bloqueada em estado de espera (5), denominado *WAITING*, sendo uma rotina de leitura bloqueante [29]. Depois, ao receber o *MESSAGE_REQUEST*, a tarefa A envia pela NoC a mensagem, liberando espaço no pipe local para uso (6) e, ao chegar a mensagem no elemento de processamento que se encontra a tarefa B (7), o *µkernel* aloca e armazena a mensagem no banco de memória contido na tarefa B e, com isso, possibilitando a execução imediata ou posterior da tarefa com sua mudança de estado para pronta, denominada *READY* (8).

Chamadas de Sistema (*System Calls*)

Chamadas de sistema são interfaces com o sistema operacional, que são ativadas por meio de interrupções de *software* [9]. Um sistema ou aplicação dentro de um SPE que utiliza essas chamadas as faz por meio das seguintes primitivas [9]:

- `EXIT`: chamada pela rotina *exit()* que termina a execução de uma tarefa. Assim sendo, fica em espera para a possível leitura de mensagens em *pipe* da tarefa e impede que a tarefa finalizada volte para fila de escalonamento [9];
- `WRITEPIPE`: chamada pela rotina de comunicação *Send(Message mensagem, int id_destino)* que envia uma mensagem para a tarefa de destino dado por *id_destino* ou escreve no *pipe* caso a mesma ainda não foi requisitada [9];
- `READPIPE`: chamada pela rotina de comunicação *Receive(Message mensagem, int id_origem)* que busca uma mensagem no *pipe* caso a mesma ainda exista, do contrário é enviado um pedido à tarefa de origem com identificador id_origem e a tarefa é bloqueada enquanto tal pedido não chegar [9];
- `GETTICK`: a chamada `GetTick()` devolve o número absoluto de ciclos de *clock* dados por um contador global [9];
- `ECHO`: a chamada `Echo()` imprime uma mensagem em forma de *string* no log da tarefa e ao mestre do *cluster*, onde o mesmo a repassa aos outros elementos [9];
- `REALTIME`: chamada pela rotina `RealTime()`, são instanciados parâmetros para processamento e execução em tempo real [9].

Procedimento de *boot*

O procedimento de *boot* para um processador Plasma dentro da HeMPS é responsável pela inicialização do sistema operacional e suas variáveis, sendo a primeira chamada feita pelo *µkernel*. Esse mesmo processo de *boot* é utilizado também para o carregamento de tarefas, visto que as mesmas precisam ser carregadas nos PE, bem como o *kernel* específico para o tipo de elemento de processamento, ou seja, um *kernel* mestre e um *kernel* escravo possuem programas de *boot* específicos [9].

Esses programas são diferenciados, pois o código de *boot*, descrito em linguagem de montagem, gerado depende da arquitetura do processador utilizada e da forma de criação do executável para os diferentes *kernel*s [9].

Vale a pena ressaltar que enquanto o *kernel* mestre é responsável por executar a chamada da função que inicializa a execução do sistema que faz as chamadas via *software* para a inicialização dos *clusters*, escravos e subsequentes chamadas e rotinas de controle para os mesmos, o *kernel* escravo possui como tarefa a execução das tarefas alocadas pelo usuário. O *kernel* escravo comunica-se com o *kernel* mestre para receber novas tarefas, notificar o término delas e em outras rotinas de controle para o fluxo de tarefas, e o su-

porte para o tratamento de interrupções e chamadas de sistemas, as chamadas *syscalls*. Nesse contexto, as *syscalls* são chamadas a funções definidas dentro de um processo de tarefa do usuário, e são capazes de mudar a rotina de execução do processador quando necessário.

Gerência de memória para multiprocessamento

O *kernel* para os SPEs, através do uso de paginação em memória, suporta a execução de múltiplas tarefas em seu interior, dividindo-as em memória em páginas de tamanho fixo durante o *boot* do *kernel* [6].

A Figura 2.34 mostra a subdivisão lógica da memória, onde a primeira página, denominada página 0, guarda o *kernel* e as outras páginas são utilizadas pelas tarefas das aplicações dentro do sistema.

O interior de uma página em memória, como pode ser visto na Figura 2.34 (b), pode ser dividida em:

- **Dados estáticos**: alocados na seção *text*, contendo o código objeto da tarefa;
- **Dados dinâmicos**: possuem dados inicializados na seção *data*, dados não inicializados na seção `bss` e a pilha de execução (*stack*). A forma que esses dados são estruturados depende do compilador utilizado no sistema [6].

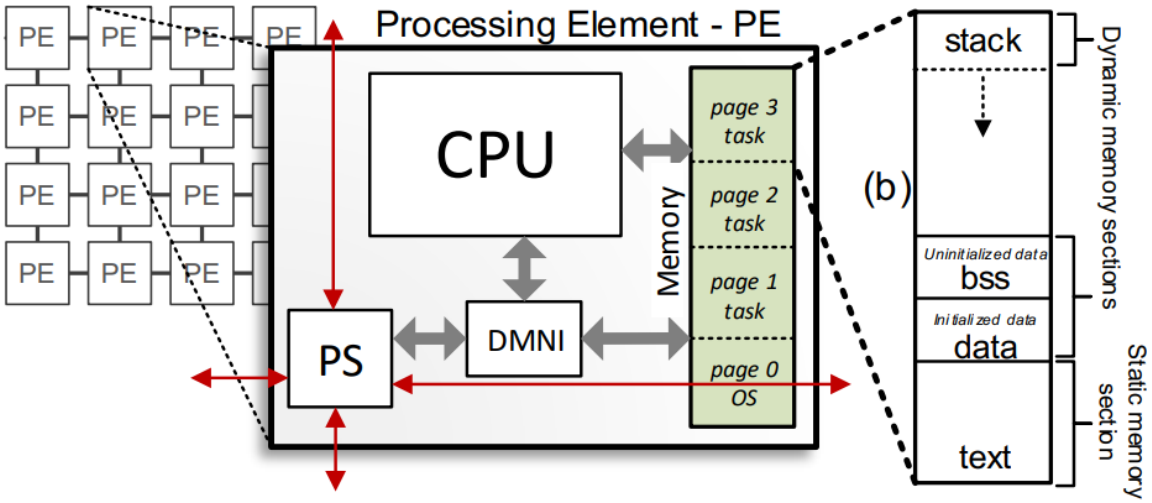


Figura 2.34: Sudivisão da memória por paginação para suporte a multiprocessamento (Fonte: [6]).

Capítulo 3

Metodologia utilizada para análise no framework HeMPS

O objetivo deste capítulo é descrever e fazer uma breve análise da metodologia de pesquisa utilizada neste trabalho. Logo, busca-se apresentar as etapas e estratégias utilizadas para se adaptar a plataforma HeMPS para o uso de um MPSoC heterogêneo por meio de um módulo de *hardware*, que neste trabalho é denominado *packet handler* (PH).

O módulo PH é capaz de fazer a interface com a DMNI para recebimento e re-envio de pacotes que são definidos em tempo de projeto entre o SPE com esse novo módulo e o SMPE. Assim sendo, sua inserção torna o sistema heterogêneo. Nas seções subsequentes, têm-se o intuito de fundamentar e caracterizar os passos para a demonstração e possíveis conclusões para problemas dessa proposta.

Em sua versão mais recente, os componentes de *hardware* da plataforma HeMPS podem ser descritos na linguagem *VHDL* e na linguagem *C++* utilizando uma biblioteca para modelagem e simulação de *hardware* em alto nível chamada *SystemC*. Para esse trabalho a versão a ser utilizada será a versão em *SystemC* por prover tempos de simulações menores que na linguagem *VHDL*, visto sua simplicidade quando comparado ao *VHDL*, e facilidade de programação [33].

Esta primeira seção abordará a interface entre o módulo *packet handler* e os outros módulos do elemento de processamento da HeMPS. Serão detalhadas a implementação original da HeMPS, as mudanças feitas com o propósito de mostrar que é possível implementar heterogeneidade na plataforma e, por fim, será descrito o resultado final após as mudanças, descrevendo a nova configuração e os relacionamentos entre os diferentes módulos existentes na estrutura inicial.

Em sua implementação original, um PE é composto, basicamente, por quatro módulos inter-relacionados: CPU, RAM, DMNI e *Router*. Cada um desses componentes está relacionado com os demais da seguinte forma:

- CPU: comunica-se diretamente com os módulos de RAM e DMNI.
- DMNI: possui estruturas de comunicação direta com os três outros componentes da HeMPS, sendo utilizada, principalmente, para fazer a interface do processador (CPU) e memória (RAM) com a NoC.
- RAM: consegue trocar mensagens diretamente com os módulos de *CPU* e DMNI, tendo portas de entrada e saída dedicadas a cada módulo operando de forma concorrente.
- *Router* (Roteador): tem ligação direta apenas com o módulo DMNI.

A Figura 3.1 ilustra a configuração e inter-relacionamentos entre os diversos componentes da HeMPS: as setas em cinza indicam os pontos onde há interconexão entre os módulos e as linhas tracejadas em vermelho representam as portas de comunicação com a NoC.

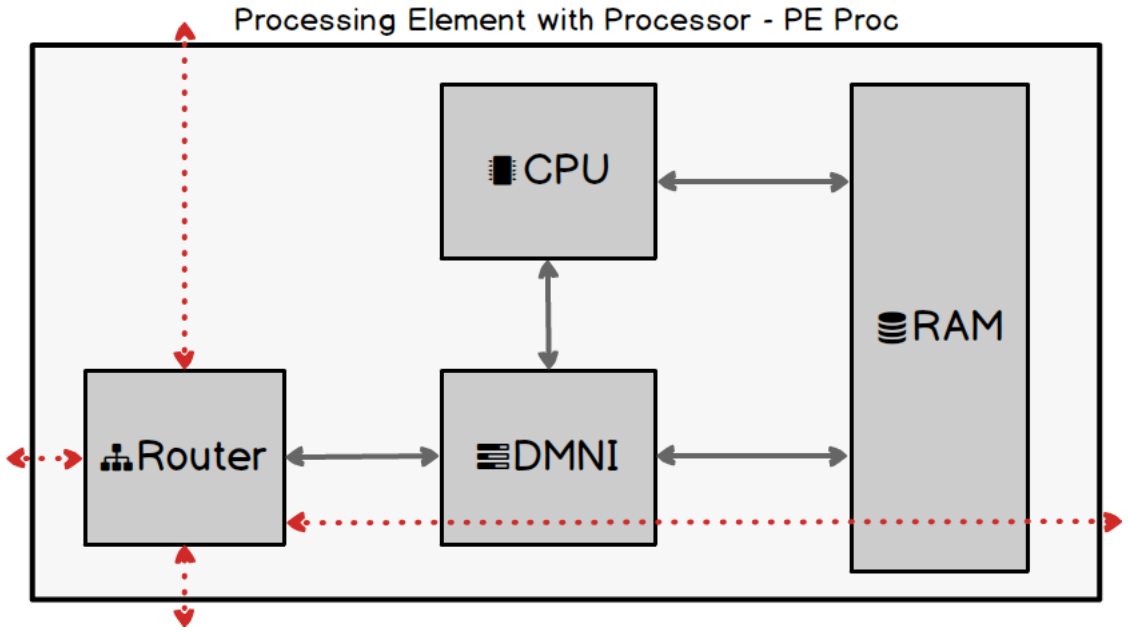


Figura 3.1: Diagrama dos módulos e interconexões originais da HeMPS. (Fonte Própria).

Na versão atual da HeMPS, o empacotamento e desempacotamento de pacotes ocorre via *software*. Para implementação de novos módulos essa característica pode ser um empecilho, por isso buscou-se um módulo que faça isso em *hardware*. No futuro, servirá de interface para outros módulos de *hardware* que possam vir a ser integrados com a NoC.

Nessa parte do desenvolvimento, foi feita a substituição do módulo CPU por um componente de controle que atua sobre a DMNI ao receber um estímulo do *Router*, programa a DMNI para ler o conteúdo da mensagem de entrada e armazená-lo na RAM. Essa substituição foi feita, pois se deseja dar a liberdade dentro do sistema para a integração de

sistemas completos que possam vim a ter seus próprios módulos centrais de processamento (CPU), mantendo PEs com o módulo PH estritamente como uma interface entre esses sistemas ou outros módulos de *hardware* e o resto do MPSoC gerado pela HeMPS.

Posteriormente, o mesmo módulo de controle, chamado *packet handler* (PH), reprograma a DMNI de forma que ela leia o conteúdo armazenado na RAM e a envie para o remetente por meio do *Router*.

Na implementação original da HeMPS, o estímulo (sinais de controle) recebido pela DMNI, acarreta a leitura ou escrita na memória local, até o limite de sua capacidade. Em outras palavras, o estímulo recebido pela DMNI é refletido para a CPU, que pode ativar uma determinada lógica de processamento e controle dos dados armazenados na RAM, seja para a execução de tarefas quanto para a manipulação dos dados recebidos pela NoC.

Com a substituição da CPU pelo PH, não existe mais um componente que utiliza os dados recebidos para processamento de instruções de tarefas, mas um controlador da DMNI, capaz de disparar pela DMNI comandos para envio ou leitura de pacotes de mensagem. Ao mesmo tempo, pode-se afirmar que o PH funciona como um controlador para a leitura e escrita na RAM, compartilhando esta função com a DMNI.

As interconexões entre os módulos após a inclusão do módulo PH permanecem inalteradas, como pode ser visto na Figura 3.2.

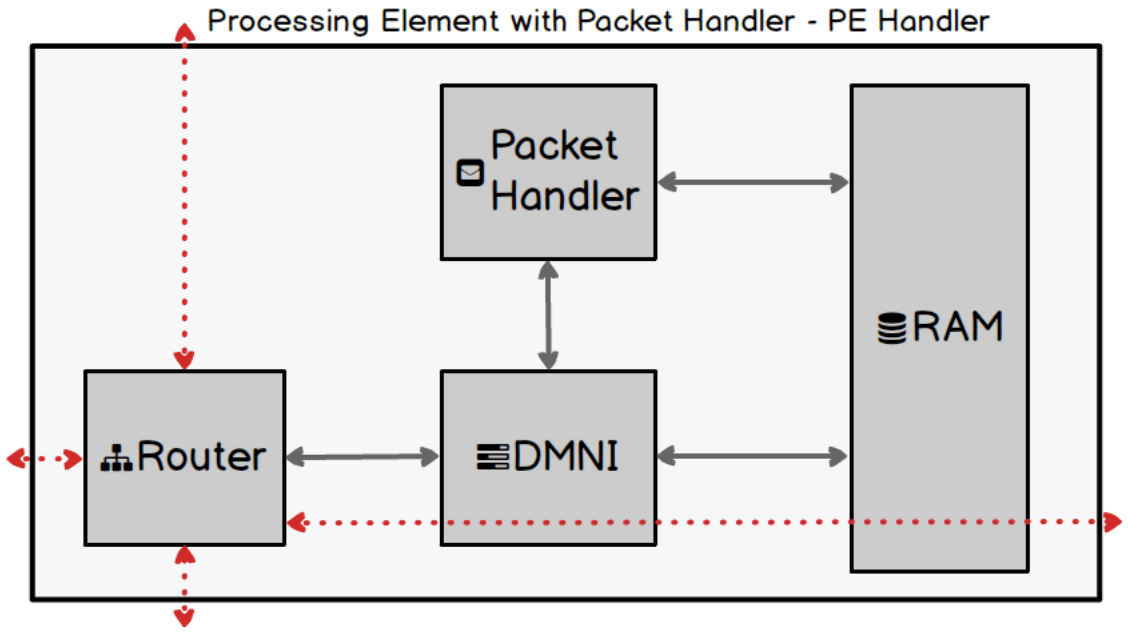


Figura 3.2: Diagrama dos módulos e interconexões após a troca do módulo de CPU. (Fonte Própria).

O módulo (PH) possui mínimas alterações na sua estrutura de ligação com os diferentes componentes no elemento processamento, adaptando as ligações já existentes (entradas,

saídas e sinais de controle do processador) para operar como um controlador RAM, sendo que algumas novas conexões foram adicionadas para permitir o funcionamento desejado. O objetivo de se manter basicamente a mesma estrutura de ligações se fundamenta no desejo de manter a forma de operação da plataforma HeMPS tão intacta quanto possível, uma vez que a mesma é conceitualmente bem fundamentada e aceita na literatura. Dessa forma, não faz sentido exigir uma mudança profunda na forma como ela funciona apenas para torná-la heterogênea. A ideia é que uma pessoa interessada em alterar a HeMPS seja capaz de acoplar módulos sem alterar profundamente o seu funcionamento do ponto de vista de *hardware*, atendo-se apenas a mudanças de *software* e possíveis alterações em interconexões e lógicas de controle dos módulos quando estritamente necessário. Além disso, é importante ressaltar que os elementos de processamento descritos na Figura 3.1 e na Figura 3.2 são gerados em conjunto na plataforma, sendo a forma que são gerados, características e desafios a serem abordados nas seções subsequentes.

Para que a plataforma HeMPS passasse a funcionar conforme esperado após a troca do módulo de CPU pelo PH, foram necessárias várias alterações no código fonte da plataforma: *scripts* de inicialização da plataforma, rotinas de controle dentro do *Kernel* mestre (exemplo: recebimento de novos serviços e inicialização de módulos escravos), adição de novos serviços dentro da aplicação e por fim, trechos de código do *hardware* do PE para readaptar o envio e recebimento de pacotes no PE.

As próximas seções vão detalhar cada categoria de alteração aplicada nos fontes da plataforma HeMPS. A seção 3.1 descreve a implementação do PH e modificações do *hardware* existente, bem como todo o processo de recebimento e envio de pacotes. A seção 3.2 apresenta as modificações na geração da plataforma e do *software*.

3.1 Implementação do módulo *packet handler*

Uma das partes mais desafiadoras na substituição do processador pelo *packet handler* diz respeito à restruturação das interações entre a DMNI e a memória RAM com o novo módulo. Assim sendo, para que o módulo do *packet handler* fosse incluído com sucesso no PE, a lógica para controle de sinais, o funcionamento das máquinas de estados finitos e o comunicação entre os diferentes módulos precisaram ser adaptados para permitir que as funções do *packet handler* fossem implementadas.

3.1.1 Definição dos sinais de controle e inicalização do novo módulo

Para adaptar o novo IP para recebimento e envio de pacotes através da configuração da DMNI, bem como para leitura e escrita pela RAM, as interfaces dos demais blocos se mantiveram inalteradas. Para a inserção do *packet handler* na HeMPS foram readaptados os sinais de controle provenientes do processador com os módulos, respeitando-se o máximo possível o seu propósito original dentro do sistema, conforme mostrado no Código-fonte 3.1 onde muitos sinais se mantêm no início a parte CPU. Algumas entradas e saídas, bem como sinais de controle dentro do elemento de processamento mostrado no Código-fonte 3.2, foram adicionados com o intuito de permitir um gerenciamento mais apurado das comunicações com os demais componentes no elemento de processamento, facilitando o controle e a sincronização de cada Máquina de Estados Finitos (*Finite State Machine*) (FSM) dos diferentes IPs.

Código 3.1: Trecho de código com a interface e sinais de controle do processador (CPU) (Fonte: [6]).

```
1         cpu = new mlite_cpu("cpu", router_address);
2         cpu->clk(clock);
3         cpu->reset_in(reset);
4         cpu->intr_in(irq);
5         cpu->mem_address(cpu_mem_address);
6         cpu->mem_data_w(cpu_mem_data_write);
7         cpu->mem_data_r(cpu_mem_data_read);
8         cpu->mem_byte_we(cpu_mem_write_byte_enable);
9         cpu->mem_pause(cpu_mem_pause);
10        cpu->current_page(current_page);
```

Código 3.2: Estrutura de inicialização e sinais de controle para o bloco IP *packet handler* (Fonte: Adaptado de [6]).

```
1         ph = new packet_handler("packet_handler", (unsigned int)
      router_address);
2         ph->clock(clock);
3         ph->reset(reset);
4         ph->set_address(cpu_set_address);
5         ph->set_address_2(cpu_set_address_2);
6         ph->set_size(cpu_set_size);
7         ph->set_size_2(cpu_set_size_2);
8         ph->set_op(cpu_set_op);
9         ph->start(cpu_start);
10        ph->cpu_mem_byte_we(cpu_mem_write_byte_enable);
```

```
11          ph->cpu_mem_address(cpu_mem_address);
12          ph->cpu_mem_data_write(cpu_mem_data_write);
13          ph->cpu_mem_data_read(cpu_mem_data_read);
14          // New fields added for PH
15          ph->config_data(dmni_data_read);
16          ph->intr(ni_intr);
17          ph->send_active(dmni_send_active_sig);
18          ph->receive_active(dmni_receive_active_sig);
19          ph->mem_address(dmni_mem_address);
20          ph->mem_data_write(dmni_mem_data_write);
21          ph->mem_data_read(packet_handler_data_read);
22          ph->mem_byte_we(dmni_mem_write_byte_enable);
23          ph->enable_send_into_pe_handler(enable_send_into_pe_handler);
24          ph->credit_dmni(credit_i_ni);
25          ph->tx_dmni(tx_ni);
```

Assim sendo, os campos da estrutura do novo módulo de *hardware* e suas funções gerais para o sistema são:

- *clock*: recebe o sinal `clock` responsável por controlar a execução de funções do MPSoC;
- *reset*: recebe o sinal `reset` responsável por reiniciar e manter as FSM dos IP em seu estado inicial;

Para os sinais referentes a interconexão e gerenciamento do módulo PH com a DMNI e suas funções, temos:

- *set_address*: por meio do sinal de saída *cpu_set_address* que tem como função permitir a inicialização de uma variável local dentro da DMNI referente ao endereço inicial para leitura ou escrita na memória local para o *header* de uma mensagem, sendo também utilizada nas lógicas de controle para as FSM dentro da DMNI;
- *set_address_2*: canal de saída de dados que atribui ao sinal *cpu_set_address_2* a função de permitir a inicialização de uma variável local dentro da DMNI referente ao endereço inicial para leitura ou escrita na memória local para o *payload* de uma mensagem, sendo também utilizada nas lógicas de controle para as FSM dentro da DMNI;
- *set_size*: canal de saída de dados que atribui ao sinal *cpu_set_size* a função de permitir a inicialização de uma variável local dentro da DMNI referente ao tamanho do *header* de uma mensagem, por padrão sendo 13 flits, e também utilizada nas lógicas de controle para as FSM dentro da DMNI;

- *set_size_2*: canal de saída de dados que atribui ao sinal *cpu_set_size* a função de permitir a inicialização de uma variável local dentro da DMNI do tamanho do *payload* de uma mensagem, caso o mesmo exista, sendo também utilizada nas lógicas de controle para as FSM dentro da DMNI;

- *set_op*: canal de saída de dados que atribui ao sinal *cpu_set_op* a função de permitir a inicialização de uma variável local dentro da DMNI referente a escrita ou leitura na memória. Um serviço é um de endereço em formato hexadecimal que está definido no *software* da plataforma. O sinal também é utilizado nas lógicas de controle para as FSM dentro da DMNI;

- *start*: canal de saída de dados que atribui ao sinal *cpu_start* o controle de inicializar as FSM de envio e recebimento de pacotes da DMNI, bem como sua leitura e escrita na memória local;

- $config\_data$: canal de saída de dados que atribui ao sinal $dmni\_data\_read$ os dados a serem passados em uma das etapas da configuração da DMNI;

- *intr*: canal de entrada de dados que recebe o sinal *ni_intr* responsável por notificar o início da memorização dos pacotes de uma mensagem dentro da DMNI por meio de uma interrupção de *hardware* e, com isso, inicializando a FSM de recebimento de pacotes dentro do módulo *packet handler*;

- *send_active*: canal de entrada de dados que recebe o sinal *dmni_send_active_sig* responsável por notificar que a FSM de envio de pacotes da DMNI foi inicializada. Está ativo quando a DMNI está enviando dados da memória, *header* e *payload* das mensagens, via NoC;

- *receive_active*: canal de entrada que recebe o sinal *dmni_send_active_sig* responsável por notificar que a FSM de recebimento de pacotes da DMNI inicializou a escrita na memória local. Está ativo quando a DMNI está recebendo dados da memória. Usada também para a sincronização da FSM de recebimento do PH para que, caso exista um *payload* na mensagem recebida, o endereço do mesmo na memória local seja corretamente configurado pelo PH através da DMNI;

Os sinais para a interface da DMNI com o roteador e, consequentemente, a NoC bem como suas funcionalidades são:

- *credit_dmni*: canal de entrada de dados que recebe por meio do sinal de controle *credit_i_ni* se há espaço suficiente no *buffer* da porta local do roteador do PE para a memorização e posterior envio de pacotes pela NoC;

- *tx_dmni*: canal de entrada de dados que recebe por meio do sinal de controle *tx_ni* se a transmissão de pacotes do roteador local com outros roteadores adjacentes está disponível ou não;

Os sinais para a interface da DMNI com a memória local bem como suas funcionalidades são:

- *mem_address*: canal de entrada de dados que recebe do sinal *dmni_mem_address* a posição da memória local desejada pela DMNI para se realizar a leitura ou a escrita, caso a mesma esteja habilitada, de pacotes de uma mensagem;
- *mem_data_write*: canal de entrada que recebe do sinal *dmni_mem_write* os dados da memória local na posição de memória dada por *dmni_mem_address*;
- *mem_data_read*: canal de saída que por meio do sinal *packet_handler_data_read* tem a função de transmitir os pacotes a serem enviados do *buffer* do PH para a DMNI e, subsequentemente, para o PE de destino escolhido;
- *mem_byte_we*: canal de entrada de dados que recebe por meio do sinal de controle *dmni_mem_write_byte_enable* se há permissão de escrita de dados na RAM, sendo essa permissão definida na FSM de recebimento de pacotes da DMNI;

Os sinais para a interface do PH, aproveitados da conexão com a *CPU*, com a RAM, utilizada para a memorização das mensagens no novo IP, bem como suas funcionalidades são:

- *cpu_mem_byte_we*: canal de saída de dados que por meio do sinal de controle *cpu_mem_write_byte_enable* permite a escrita de dados na RAM pelo módulo PH;
- *cpu_mem_address*: canal de saída de dados que por meio do sinal de controle *cpu_mem_address* define a posição da memória local para leitura ou escrita de dados, sendo utilizada pelo PH para a leitura e envio de pacotes;
- *cpu_mem_data_write*: canal de saída de dados que por meio do sinal de controle *cpu_mem_data_write* manda os dados que vão ser escritos na memória local na posição definida pelo sinal *cpu_mem_address*;
- *cpu_mem_data_read*: canal de entrada herdado da interface do processador e sem utilização dentro do PH e na lógica de controle dentro do PE, mas mantido para manter a interface original;

Por fim, a interface de controle com o PE para escolha do módulo para a transmissão dos dados armazenados para leitura e escrita das mensagens é feito por:

- *enable_send_into_pe_handler*: canal de saída de dados que por meio do sinal de controle *enable_send_into_pe_handler* controla o seletor de um multiplexador de duas entradas cuja função é escolher se os dados lidos pela DMNI serão buscados diretamente na RAM ou pelo *buffer* presente no PH. Esta lógica de escolha sobre a busca dos dados é necessária apenas nas FSM de envio de pacotes do sistema uma vez que no envio o novo módulo executa verificações e modificações nos pacotes que garantem a sincronia das essas máquinas e, portanto, evitam que pacotes sejam perdidos ou enviados erroneamente durante os ciclos no sistema;

3.1.2 Adaptação do módulo *packet handler para o recebimento de pacotes*

Sabendo que as correlações entre os diferentes módulos e o controle dos sinais de suas FSM agora são dependentes de como ocorre o gerenciamento para pacotes, o fluxograma de envio de pacotes teve sua lógica adaptada e será abordada nessa seção.

O módulo PH faz uma interface com a FSM da DMNI para o recebimento de pacotes. Ao receber um pacote, seu conteúdo é salvo na memória local (RAM) pela FSM de recebimento de pacotes da DMNI e suas informações relevantes serão utilizadas para a *"programação"* de um módulo de hardware a ser inserido no sistema utilizando a DMNI.

Na Figura 3.3, temos a exemplificação das FSMs de recebimento de pacotes com o módulo *packet handler* incluso no contexto dos outros elementos do PE, onde os sinais com linhas tracejadas em preto são os resultados daquele estado a ser utilizado na FSM da DMNI.

Primeiramente, como pode ser visto na parte (a), ao receber um pacote da porta local do roteador, o primeiro estado da FSM, denominado `HEADER`, gera uma interrupção de *hardware* com o intuito de sinalizar que recebeu o pacote, desde que o recebimento de pacotes pelo roteador esteja habilitado e que exista espaço para memorização. Assim, já na parte (b), a FSM do PH é inicializada, saindo do estado de espera `WAIT_INTR`. Com isso, o pacote é memorizado dentro da DMNI e no PH ocorre a configuração da DMNI. O *buffer* utilizado pela DMNI para memorização tem a função de guardar os pacotes recebidos da NoC para o uso na FSM vista em (c). Caso o *buffer* não possua mais espaço para armazenamento, a FSM em (a) fica em espera até FSM em (c) carregar os pacotes no *buffer* e escrevê-los na RAM e, com isso, liberando espaço dentro do *buffer*.

Primeiro, o tamanho do pacote é posto no sinal `DMNI_SIZE` para definir o espaço ocupado pelo cabeçalho da mensagem ou, se houver *payload*, ao tamanho da carga útil da mensagem após a memorização do *header* na RAM.

O endereço de início, definido pelo usuário para a escrita na RAM, é colocado no sinal DMNI_ADDRESS.

O tipo de operação a ser feita na memória local, no caso escrita, é colocado no sinal denominado DMNI_OP. Por fim, a configuração é feito pelo sinal DMNI_START que inicializa a FSM da DMNI vista em (c). Toda essa configuração é feita nos estados PROG_SIZE, PROG_ADDRESS, PROG_OP e PROG_START, respectivamente.

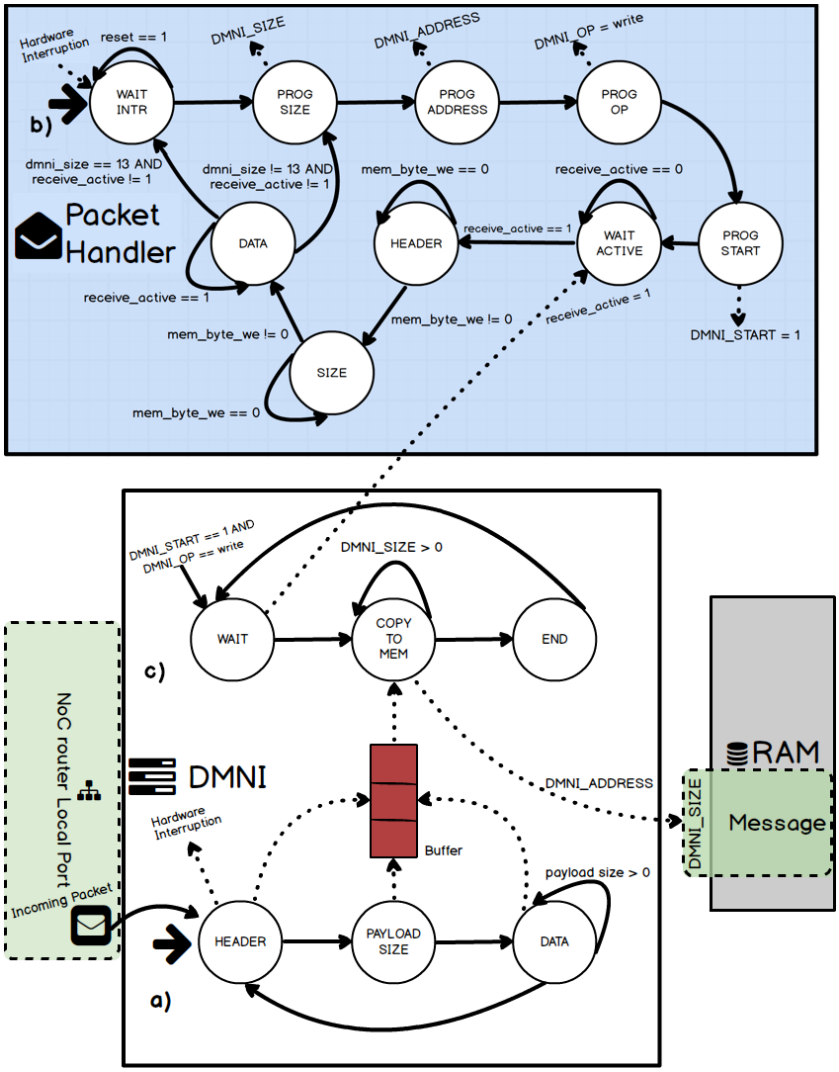


Figura 3.3: Exemplificação das máquinas de estados finitos da tarefa de recebimento de pacotes tendo o módulo *packet handler* integrado ao PE, sendo em (a) e (c) o fluxograma para a DMNI e (b) para o novo módulo (Fonte: Adaptada da Figura 2.27).

Terminada a configuração da DMNI, a FSM fica em repouso no estado WAIT_ACTIVE até a entrada *receive_active* receber a sinalização da DMNI sobre o início da escrita na memória local e, com isso, acontece a contagem de flits da mensagem, onde os estados do PH denominados HEADER e SIZE são para contagem e verificação dos flits contendo

o endereço do roteador de origem da mensagem e o tamanho da mensagem, desconsiderando as duas primeiras unidades de controle de fluxo, respectivamente e exemplificados no Código-fonte 3.3. Além disso, no estado SIZE é verificado se a mensagem contém apenas cabeçalho ou possui carga útil e, caso tenha, setando o tamanho da mensagem, considerando cabeçalho e carga útil, no sinal interno dmni_size para ser utilizado na reprogramação da DMNI para escrita do *payload* na memória local.

Código 3.3: Trecho de código referente a leitura dos dois primeiros flits flits da mensagem pelo módulo *packet handler* (Fonte: Fonte própria).

```
1         case HEADER:
2         if(mem_byte_we.read() != 0) {
3             packet_counter.write(packet_counter.read()+0x00000004);
4             CS.write(SIZE);
5             }
6         else CS.write(HEADER);
7         break;
8
9         case SIZE:
10        if(mem_byte_we.read() != 0) {
11                if(dmni_size.read() == 13) {
12                 packet_counter.write(packet_counter.read()+0x00000004);
13                 payload_size.write(11);
14                 if(mem_data_write.read() == 11)
15                  dmni_size.write(13);
16                 if(mem_data_write.read() > 11) {
17                 dmni_size.write(mem_data_write.read() - 11);
18                 my_packet_total_size.write(mem_data_write.read() + 2);
19                 }
20                } else {
21                packet_counter.write(packet_counter.read()+0x00000004);
22                payload_size.write(dmni_size.read() - 2);
23                dmni_size.write(13);
24                }
25                CS.write(DATA);
26        } else CS.write(SIZE);
27        break;
```

Por fim, no estado DATA, exemplificado no Código-fonte 3.4, é feita a contagem dos flits restantes. Se a terceira unidade de fluxo de controle que se refere ao serviço requerido pela mensagem for o MY_PACKET (serviço criado para esse estudo), é emitida uma notificação para que a FSM de envio do módulo PH seja inicializada. Ao mesmo tempo, ocorre a memorização, em forma de sinal, do endereço inicial da mensagem na RAM, e verifica-se a existência de *payload* na mensagem. Caso exista um *payload*, o PH reprograma a DMNI

para recebimento da carga útil da mensagem e o fluxograma de configuração é reinicializado. Como consequência, a máquina de recebimentos do PH fica em repouso até ocorrer outra interrupção de *hardware* que exija a inicialização dos processos de recebimento de mensagens a ser requerido pela DMNI.

Código 3.4: Trecho de código referente a leitura dos flits restantes da mensagem pelo módulo *packet handler* (Fonte: Fonte própria).

```
1         case DATA:
2         if(mem_byte_we.read() != 0) {
3          if(mem_data_write.read() == 0x300){
4           activate_send.write(1);
5           mypacket_init_addr.write(packet_counter.read()-0x00000008);
6          }
7          packet_counter.write(packet_counter.read()+0x00000004);
8          payload_size.write(payload_size.read() - 1);
9         }
10        if(receive_active.read() == 1)
11         CS.write(DATA);
12        else
13         if(dmni_size.read() == 13)
14          CS.write(WAIT_INTR);
15         else CS.write(PROG_SIZE);
16        break;
```

3.1.3 Adaptação do módulo *packet handler* para o envio de pacotes

Nesta seção é mostrado como é realizado o envio de pacotes pelo módulo PH.

O novo IP faz uma interface com a FSM da DMNI para o envio de pacotes. Ao enviar um pacote, seu conteúdo é lido na memória local e suas informações relevantes são memorizadas para, então, a DMNI ser *"programada"* para enviá-lo pela NoC.

Na Figura 3.4, temos a exemplificação das FSMs de envio de pacotes com o módulo *packet handler* incluso no contexto dos outros elementos do PE.

Primeiramente, descrito na parte (a), o estado inicial denominado `WAIT_SEND` recebe a sinalização via *hardware* de que a máquina de estados de recebimento dentro do módulo PH passou por todos os flits da mensagem e, com isso, o re-envio para o PE de destino começa. Logo, no estado `WAIT_SEND`, são inicializados os sinais internos para controle dos outros estados e, no estado `FULLFILL_BUFFER`, a memorização da mensagem internamente no PH, sendo isso feito pra melhorar a sincronização entre as FSMs do PH e da

DMNI. O *buffer* interno tem tamanho total equivalente ao número de flits do *header* e do *payload* dado pela variável local *buffer_total_size*.

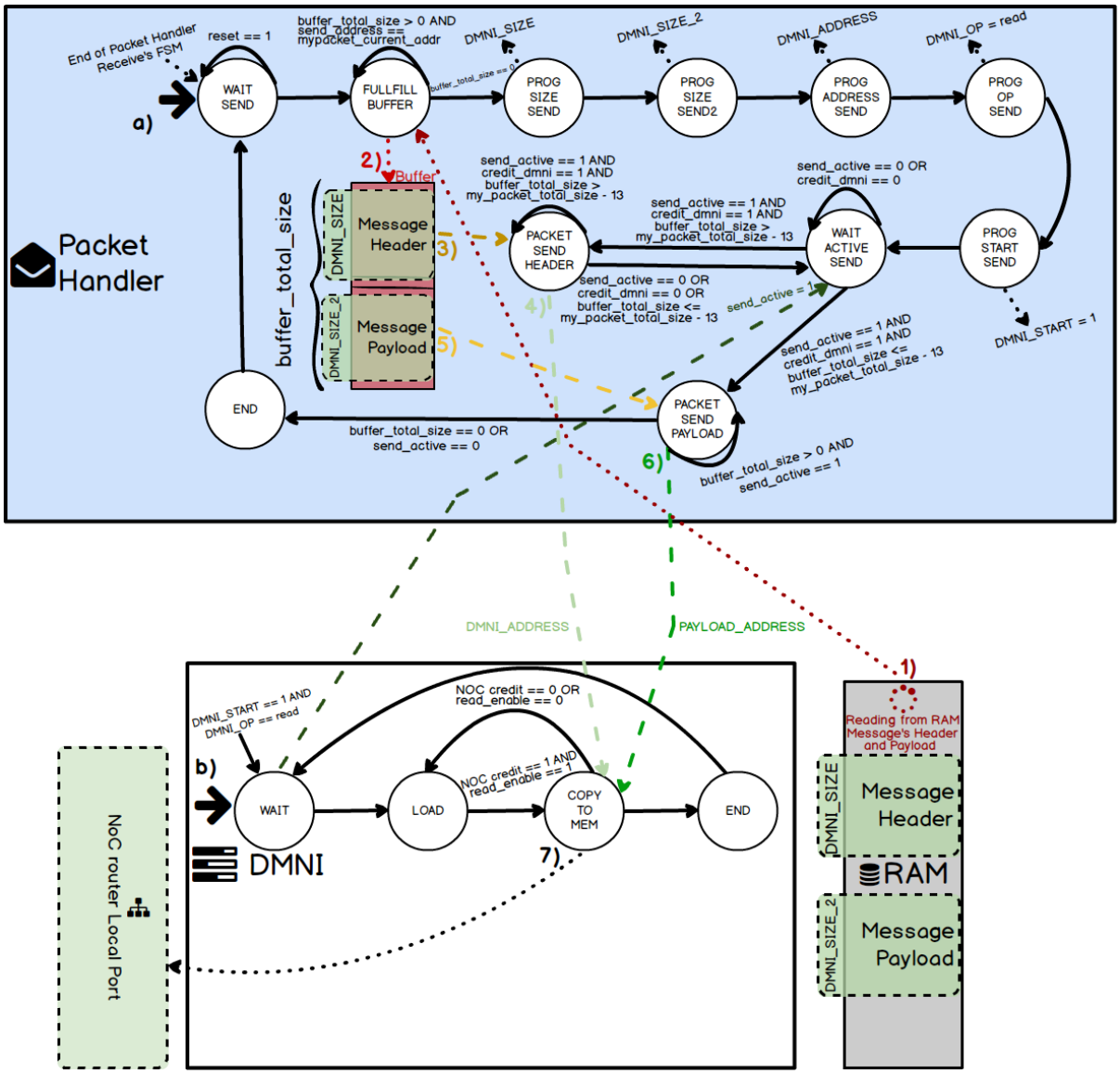


Figura 3.4: Exemplificação das máquinas de estados finitos da tarefa de envio de pacotes tendo o módulo *packet handler* integrado ao PE, sendo em (b) o fluxograma para a DMNI e (a) para o módulo *packet handler* (Fonte: Adaptada da Figura 2.26).

Em `FULLFILL_BUFFER`, primeiramente os dados da memória local são lidos, conforme mostrado no passo (1), e continuam a ser carregados, como visto no passo (2), enquanto não forem totalmente carregados e o módulo PH e a DMNI não estiverem sincronizadas. Após toda a mensagem ser carregada, análogo ao recebimento de pacotes, o PH configura a DMNI para o envio de pacotes através dos seguintes estados que tem por função:

- `PROG_SIZE_SEND`: define o tamanho do *header* da mensagem;
- `PROG_SIZE_SEND2`: define o tamanho do *payload* da mensagem;
- `PROG_ADDRESS_SEND`: configura o endereço inicial da mensagem na memória local;
- `PROG_OP_SEND`: configura o modo de operação, no caso de leitura, da RAM pela DMNI, tendo como opções o valor 1 para escrita e o valor 0 para leitura na memória;
- `PROG_START`: inicializa a FSM de envio da DMNI.

Finalizadas todas essas etapas, a FSM espera em `WAIT_ACTIVE_SEND` até que a entrada *send_active* receba a sinalização da DMNI sobre o início da leitura dos dados.

Assim que a FSM inicia seu funcionamento na DMNI, o PH, por meio do sinal de controle `enable_send_into_pe_handler`, habilita um multiplexador localizado no PE, mostrado na Figura 3.5, cujo propósito é definir de qual fonte o fluxo de dados virá, ou seja, se os flits a serem passados para a DMNI e, posteriormente, para NoC vão ser fornecidos pela RAM ou pelo PH. No caso, os dados serão fornecidos diretamente pela RAM no recebimento de mensagens e pelo PH no envio de mensagens, pois, como foi dito anteriormente, precisamos memorizar os flits dentro do PH para melhor sincronizar com a DMNI e, com isso, mandar corretamente toda a mensagem para o seu destino, além de operações para montar os pacotes e validação dos mesmos.

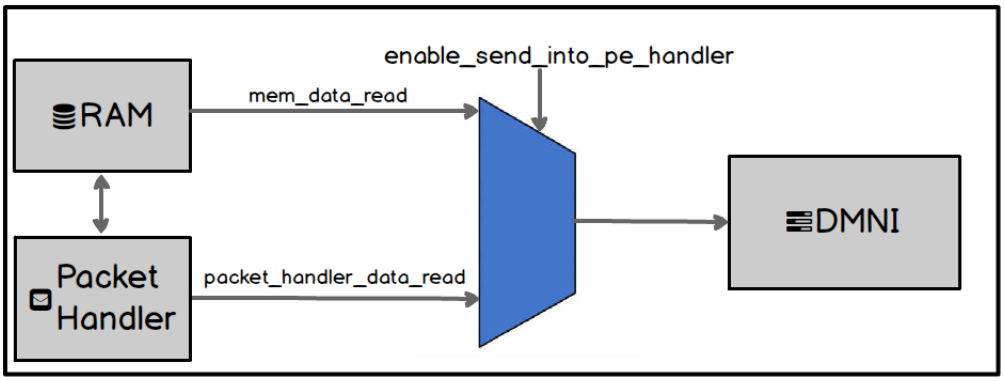


Figura 3.5: Exemplificação do fluxo de dados enviados para a leitura dos pacotes de uma mensagem pela DMNI (Fonte: Fonte Própria).

Assim sendo, nos passos (3) e (4), temos no estado `PACKET_SEND_HEADER`, respectivamente, a leitura do cabeçalho de dentro do PH e envio para a DMNI, exemplificada no Código-fonte 3.5, ficando nesse estado enquanto tiver flits para enviar e o seu envio para a DMNI estiver disponível, do contrário volta para o estado `WAIT_ACTIVE_SEND`.

Código 3.5: Trecho de código referente ao envio de flits do cabeçalho da mensagem do módulo *packet handler* para a DMNI (Fonte: Fonte própria).

```
1        case PACKET_SEND_HEADER:
2        if(buffer_total_size.read() > (my_packet_total_size.read() - 13)
3           && send_active.read() == 1) {
4         if(credit_dmni.read() == 1 && send_active.read() == 1) {
5          // Sending Package into DMNI
6          mem_data_read.write(packet.at(my_packet_total_size.read() -
7           buffer_total_size.read()));
8          buffer_total_size.write(buffer_total_size.read()-1);
9          } else {
10           if (credit_dmni.read() == 0) {
11            if(tx_dmni.read() == 1){
12             buffer_total_size.write(buffer_total_size.read()+1);
13                }
14              } else {
15                 if(tx_dmni.read() == 1) {
16                  buffer_total_size.write(buffer_total_size.read()+1);
17                 }
18              }
19              SendState.write(WAIT_ACTIVE_SEND);
20         }
21       } else SendState.write(WAIT_ACTIVE_SEND);
22       break;
```

Terminado o envio do cabeçalho e o envio de pacotes para a DMNI habilitado, nos passos (5) e (6), temos no estado `PACKET_SEND_PAYLOAD`, respectivamente, a leitura do *payload* de dentro do PH e envio para a DMNI, exemplificada no Código-fonte 3.6.

Código 3.6: Trecho de código referente ao envio de flits do cabeçalho da mensagem do módulo *packet handler* para a DMNI (Fonte: Fonte própria).

```
1        case PACKET_SEND_HEADER:
2        if(buffer_total_size.read() > 0 &&
3           send_active.read() == 1) {
4         if(credit_dmni.read() == 1 && send_active.read() == 1) {
5          mem_data_read.write(packet.at(my_packet_total_size.read()
6          - buffer_total_size.read()));
7            buffer_total_size.write(buffer_total_size.read()-1);
8          } else {
9             if (credit_dmni.read() == 0) {
10             if(tx_dmni.read() == 1){
11              buffer_total_size.write(buffer_total_size.read()+2);
12             }
13            } else {
```

```
14                  if(tx_dmni.read() == 1) {
15                   buffer_total_size.write(buffer_total_size.read()+1);
16                  }
17              }
18          }
19      } else SendState.write(END);
20      break;
```

Enquanto existirem flits a serem transmitidos e o seu envio para a DMNI estiver disponível, a FSM do módulo PH fica no estado `PACKET_SEND_PAYLOAD`, sendo enviado no passo (7) da DMNI para a NoC, e, após o termino, indo para o estado `END` para reconfiguração dessa FSM e volta para o estado de início `WAIT_SEND` para a espera de subsequentes mensagens.

3.2 Mudanças e ajustes no *software* da plataforma HeMPS para o módulo *packet handler*

Além de mudanças no *hardware*, adaptações em *software* foram necessárias para a correta geração da MPSoC pela HeMPS com o novo IP. Logo, as seções seguintes descreverão melhor todas essas etapas, exemplificadas previamente na Figura 3.6.

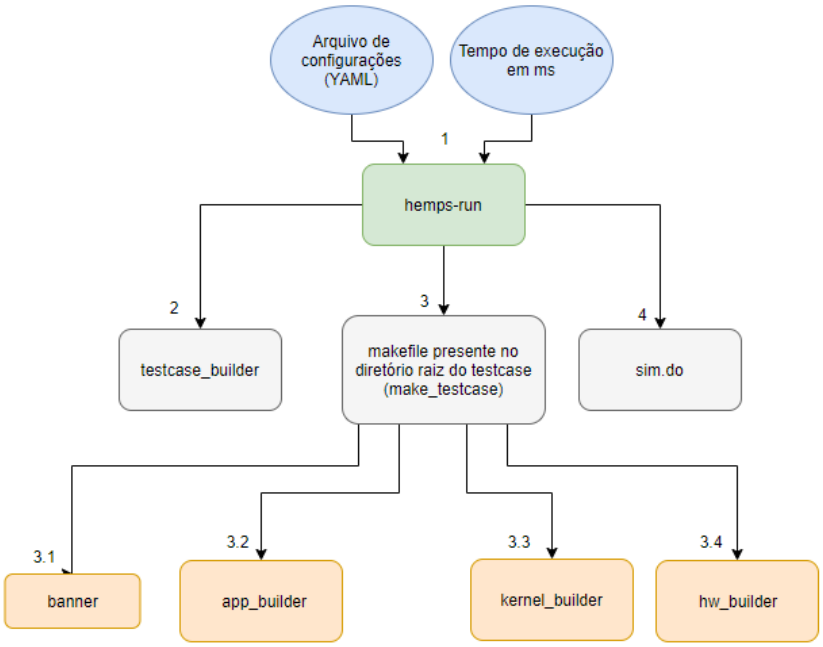


Figura 3.6: Fluxograma da criação da plataforma HeMPS com todas suas etapas (Fonte: [2]).

3.2.1 Execução inicial da plataforma HeMPS por meio do *script hemps-run* e configuração através do arquivo *YAML*

O primeiro passo da geração e execução de um MPSoC dentro da plataforma parte por meio do *script* `hemps-run` o qual, por meio dos campos de configuração da plataforma definidos em um arquivo de configuração do tipo *YAML* (do inglês *YAML Ain't Markup Language*), que define todo o escopo da plataforma bem como o tempo de execução, em milissegundos, das tarefas definidas pelo usuário com intuito de testá-las. Por exemplo, a chamada ao *script* é feito pelo comando *hemps-run example.yaml 20*, sendo o primeiro parâmetro a chamada a rotina de execução do *script*, o segundo parâmetro *example.yaml* o nome definido para o arquivo de configuração e o terceiro parâmetro *20* o tempo de execução.

O arquivo de configuração é subdividido originalmente em três partes:

- `hw` (do inglês *hardware*): usada para as definições de *hardware*;
- `sw` (do inglês *software*): utilizada para as definições de *software*;
- `apps` (do inglês *applications*): define as tarefas das aplicações que vão ser executadas e o seu tempo de início.

Para definir o escopo de execução e criação do PH, foi inserida uma quarta parte denominada `chw`, sigla do inglês *custom hardware*. No Código-fonte 3.7 vemos um exemplo desse arquivo com todas essas seções.

Código 3.7: Exemplo de arquivo de configuração YAML para a geração do sistema na HeMPS, bem como os parâmetros adicionados para o módulo PH (Fonte: Adaptado de [6]).

```
1 hw:
2    page_size_KB: 32
3    tasks_per_PE: 4
4    repository_size_MB: 1
5    model_description: sc
6    noc_buffer_size: 8
7    mpsoc_dimension: [6,6]
8    cluster_dimension: [2,2]
9    master_location: LB
10 sw:
11    mapping_algorithm: WithLoad
12    task_scheduler:  round_robin
13 apps:
14   - name: mpeg
```

```
15      start_time_ms: 0
16    - name: mpeg
17      start_time_ms: 2
18    - name: dijkstra
19      start_time_ms: 2
20    - name: synthetic
21      start_time_ms: 2
22  chw:
23    - name: packet_handler
24      number_pe: 2
25      init_addr: 0x00000064
26      first_payload_size: 20
27      second_payload_size: 15
28      payload:
29        first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751, 63, 23,
        345, 145, 543, 56745, 435, 54, 90]
30        second: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
31      static_addrs:
32        0: [3,3] # packet handler at address X=3, Y=3
33        1: [0,1] # packet handler at address X=0, Y=1
```

Assim sendo, a parte de *hardware* pode ser parametrizada pelo campos:

1. `page_size_KB`: definição do tamanho, em *quilobytes*, da página na memória local dos PE [34];
2. `tasks_per_PE`: quantidade de tarefas que podem ser executadas em um mesmo elemento de processamento, respeitando o escalonamento definido [34];
3. `repository_size_MB`: definição do tamanho, em *megabytes*, para o repositório local de tarefas a serem executadas dentro da plataforma [34];
4. `model_description`: definição da linguagem de descrição de *hardware* a ser utilizada dentro da plataforma, sendo as opções `sc` para a linguagem *SystemC*, `vhdl` para suporte a linguagem *VHDL* e `scmod` para habilitar a simulação da linguagem *SystemC* no simulador *Questa*;
5. `noc_buffer_size`: definição do tamanho para a memorização interna dos roteadores da NoC, ou seja, dos seus *buffers* [34];
6. `mpsoc_dimension`: definição da quantidade de PEs na MPSoC em topologia malha 2D, com dimensão mínima 2×2 ou a dimensão definida para o MPSoC [34];
7. `cluster_dimension`: análogo ao `mpsoc_dimension`, porém se referindo a dimensão dos *cluster*s na MPSoC, sendo a dimensão máxima de cada um o tamanho

máximo da MPSoC e o número de total de PEs em um *cluster* deve ser um número divisível do número do total de PEs do MPSoC. [34];

8. `master_location`: posição dentro da MPSoC do mestre global do sistema, sendo *LB*, do inglês *Left Bottom*, o PE inferior esquerdo localizado na posição x=0 e y=0 e, na versão atual da plataforma, o único espaço possível de ser alocado [34].

No trecho de código que se refere às configurações de *software*, trecho intitulado `sw` no código, tem-se a definição das estratégias de gerenciamento das tarefas em execução e como elas são alocadas nos diferentes PEs. Nesse pedaço de código são definidas algumas variáveis que podem ser descritas como:

1. `mapping_algorithm`: definição do tipo de algoritmo a ser utilizado de mapeamento de tarefas utilizado (*WithLoad*);
2. `task_scheduler`: definição do tipo de algoritmo de escalonamento de tarefas desejado para gerenciar o sistema, sendo as opções disponíveis os algoritmos *LST* (*Least slack time*), sendo definido no arquivo de configuração como *lst*, e *RR* (*Round Robin*), sendo definido no arquivo de configuração como *round_robin* [6] [11] [34];

A seção `apps` contem a lista das aplicações a serem executadas na MPSoC, sendo todas definidas em tempo de projeto uma vez que não se pode definir outras em tempo de execução, mas podendo se ter diversas aplicações de um mesmo código fonte ou de diferentes aplicações. Assim sendo, elas são configurada com os campos:

1. `name`: nomeação dada ao diretório dentro da pasta local *aplications* referente as tarefas da aplicação desejada a serem processadas na plataforma [34];
2. `start_time_ms`: *delay* definido em tempo de projeto em milissegundos `ms` referente ao início do processamento das tarefas da aplicação escolhida [34].

Por fim, foi inserida a seção `chw` referente as configurações de *hardware* e definições dentro da plataforma para a geração correta de uma instância contendo o módulo PH, onde esses campos de configuração são:

1. `name`: denominação do módulo a ser instanciado em um PE para sua melhor identificação no código interno;
2. `number_pe`: quantidade de PEs que receberão o módulo PH, sendo para esse estudo limitado a dois devido ao número de espaços livres simultâneos em memória para leitura e escrita na DMNI sem que se corrompa seus dados internos [6];

3. `init_addr`: endereço da memória local para escrita ou leitura na memória local do PE, sendo que na lógica de controle da DMNI o endereçamento é reduzido em quatro. Por exemplo, `init_addr` com valor *0x00000064* será convertido para o endereço *0x00000060* e indicara que a escrita ou leitura em memória será iniciada nesta posição;

4. Na subdivisão *payload* temos dois campos disponíveis para os *payloads* enviados pela NoC. A quantidade de *payloads* enviados é limitada a dois devido ao número de espaços de memória disponíveis para que uma mesma tarefa do SMPE envie, via *software*, uma mensagem sem que a a memória local manipulada pela DMNI seja corrompida. Assim, temos:

   - `first_payload_size`: campo contido na subdivisão *payload*, define o tamanho da carga útil da primeira mensagem a ser mandada do SMPE ao primeiro SPE contendo o módulo PH;
   - `second_payload_size`: análogo ao campo `first_payload_size`, define o tamanho a carga útil da segunda mensagem a ser mandada do SMPE ao segundo SPE contendo o módulo PH;

5. Na subdivisão *static_addrs* são listadas as posições de rede de cada um dos PHs. Cada PH tem um índice iniciado no número zero para o primeiro, onde, para exemplificação, temos no Código-fonte 3.7:

   - `0`: campo contido na subdivisão *static_addrs*, define a posição estática x e y do primeiro SPE que possui o módulo PH na MPSoC. Obrigatoriamente, essa posição não pode conter nenhum PE mestre, seja ele algum mestre local (MPE) ou o mestre do sistema (SMPE);
   - `1`: campo contido na subdivisão *static_addrs*, define a posição estática x e y do MPSoC do segundo SPE que possui o módulo PH. Obrigatoriamente, essa posição não pode conter nenhum PE mestre, seja ele algum mestre local (MPE) ou o mestre do sistema (SMPE).

De maneira similar, outros SPE podem ser adicionados aumentado-se o número de seu campo, ou seja, *2* para o terceiro SPE e assim por diante.

Além disso, as tarefas de uma aplicação podem ser definidas em tempo de projeto, ou seja, escolher em qual PE serão alocadas as tarefas da aplicação. Com isso, torna-se necessário a definição manual de todas suas tarefas organizadas por início de tempo de inicialização. No Código-fonte 3.8, por exemplo, é definido que a tarefa *print* será mapeada no PE 1×1 e a tarefa *start* no PE 0×1.

Código 3.8: Exemplo de arquivo de configuração YAML para a geração do sistema na HeMPS, bem como os parâmetros adicionados para o módulo PH (Fonte: Adaptado de [6]).

```
1 #--------- Application definitions -----------
2 #Example of an application defining static mapping for two tasks
3 - name: mpeg
4     start_time_ms: 0 #any unsigned integer number
5     static_mapping:
6       print: [1,1] # Task print from app mpeg will be mapped as static at
      address X=1, Y=1
7       start: [0,1] # Task start from app mpeg will be mapped as static at
      address X=1, Y=1
8 #
9 # Attention: When using static mapping all application must be manually
     sorted by start time
```

Com isso tudo definido, uma pasta com o nome do arquivo *YAML* é criada contendo os resultados, configurações, códigos fontes e a MPSoC é criada. Essa parte da compilação é gerenciada por um sub-tarefa do comando `hemps-run` denominado `testcase_builder`. Essa etapa de geração da plataforma não sofreu alterações visto que os seus procedimentos independem da arquitetura escolhida.

3.2.2 Inspeção estática e criação dos arquivos para execução da plataforma pelo *script testcase_builder*

A etapa seguinte de geração acontece na execução do *script* `testcase_builder`.

O início do processo se dá pela inspeção estática dos campos contidos nas configurações do sistema do arquivo *YAML*. Subsequentemente, são criados os arquivos fontes da plataforma e copiados muitos outros arquivos necessários para a execução da simulação. Exemplos das funcionalidades desses arquivos seriam depurações, análises de formas de ondas dos sinais da simulação e arquivos fontes contendo informações sobre configurações do *hardware* e *software* na HeMPS [2].

3.2.3 Compilação das aplicações alocadas e do *software*

Após todos esse arquivos estarem disponíveis no diretório gerado para a simulação, a rotina `hemps-run` executa um arquivo *makefile* de nome `make_testcase` localizado na raiz do projeto. Suas etapas de execução em terminal ou ambiente de desenvolvimento são da seguinte maneira:

1. A tarefa executada por `hemps-run` executa os arquivos na linguagem *python* contidos no diretório *scripts*, sendo a primeira chamada para o *banner* da plataforma contendo o nome da plataforma e a versão utilizada [2].

2. Através do *script* `app_builder`, é realizado a transcrição (*parse*) das configurações necessárias para a chamada dos *scripts* e, consequentemente, das tarefas das aplicações que foram declaradas no arquivo de configuração do sistema e seus códigos fontes localizados na pasta *applications*, exemplificado na Figura 3.7 [2].

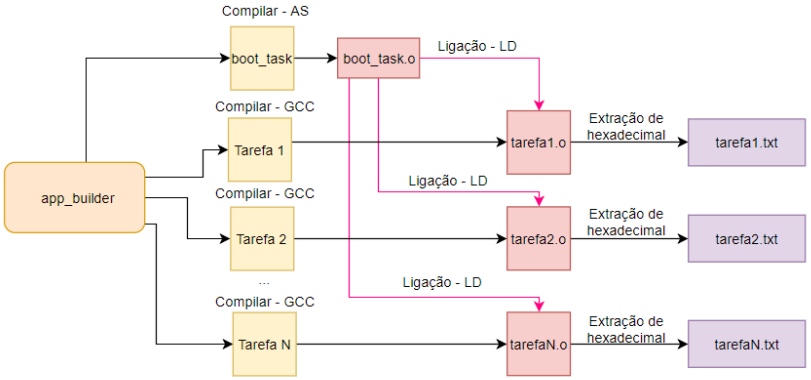


Figura 3.7: Esquematização da compilação e geração dos códigos objetos relativos as aplicações a serem simuladas dentro da HeMPS (Fonte: [2]).

Os conjuntos de diretivas dos arquivos *makefiles* contidos em cada diretório correspondente a uma aplicação passam por um rotina de *boot*, denominado *boot_task.asm*. Por ela, são inicializados o contexto das tarefas de cada aplicação, sua execução pela rotina de entrada (`main`) de cada tarefa e, se necessário, o usos de comunicação entre tarefas por meio de interrupções de *hardware* `SystemCall`. Funcionam também como uma API entre os PEs dentro da MPSoC [2].

A compilação das tarefas é feita pela ferramenta `as` para os códigos gerados em linguagem de montagem (*Assembly*), como pode ser visto na Figura 3.7.

Os códigos objetos dessa primeira compilação são combinados (*linked*) em um executável referente a sua aplicação pelo ligador `ld`.

Por fim, todos os arquivos objetos gerados tem seu conteúdo extraído e convertido para a base hexadecimal para gerar, em formato de texto, a página da memória utilizadas pelas tarefas. Essa etapa é criada pelas ferramentas `objdump` e `objcopy`. Todas as etapas passam pela cadeia de ferramentas fornecida pelo `gcc-mips-elf`

e a compilação dos códigos fontes de *hardware* e *software* nas linguagens `C` e `C++` pelas ferramentas `gcc` e `g++`, respectivamente [2].

Para todo esse tarefa não foram feitas mudanças para adequação com o PH, já que o enfoque das mudanças é no comportamento do *kernel* mestre e mudanças nas definições dos *clusters* e do ambiente de simulação [2].

3. A execução do *script* `kernel_builder`, exemplificada na Figura 3.8, cria os arquivos `kernel_pkg.h` e `kernel_pkg.c` tendo neles os parâmetros dos *clusters* e informações utilizadas apenas pelo *kernel* mestre sobre o mapeamento estático dos SPEs, tarefas e dimensões dentro sistema. Em seguida, são criadas as páginas de memória para os *kernels* dos PEs [2] [6].

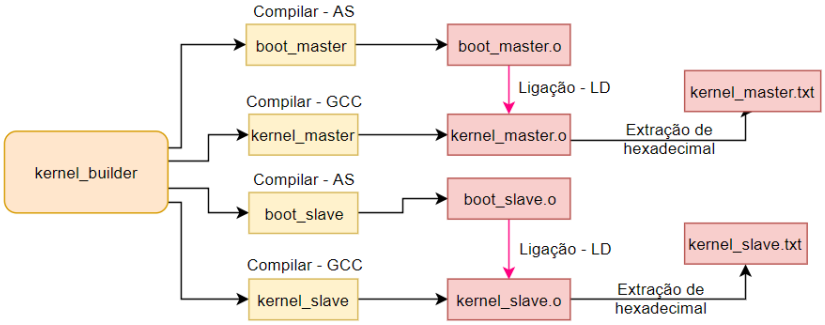


Figura 3.8: Esquematização da compilação e geração dos códigos objetos dos núcleos dos sistemas operacionais para PEs mestres e escravos dentro da HeMPS (Fonte: [2]).

Análogo ao *script* `app_builder`, através da chamada da rotina `make` de um arquivo `makefile`, ocorre a compilação, ligação e extração do código objeto gerado para hexadecimal para os núcleos dos sistemas operacionais e instruções de inicialização tanto para PEs mestres quanto PEs escravos [2].

O *kernel* mestre é responsável pela implementação e gerenciamento dos *clusters* e seus PEs, provendo uma comunicação com elementos escravos e, no caso do mestre global, mestres globais através da NoC, além de controle e gerenciamento de tarefas. Seu *boot* inicializa o contexto da tarefa e início de execução pela chamada a rotina *main* [2] [6].

O *kernel* escravo, análogo ao mestre, também implementa chamadas para a comunicação via NoC, porém seu enfoque é, principalmente, no controle a execução de tarefas por meio do algoritmo de escalonamento definido no sistema e tratamento de interrupções vindas de *hardware* e *software* por meio e chamadas de sistema [2] [6].

Nessa etapa foram adicionadas diversas mudanças para a inserção do módulo PH. No *script* `kernel_builder`, mostrado no Código-fonte 3.9, temos a definição de erros de compilação caso o módulo PH não seja configurado corretamente.

Código 3.9: Trecho de código que mostra como se evita a compilação para os casos em que o módulo PH é mal definido no arquivo de configuração *YAML* para o *script* `kernel_builder` (Fonte: Fonte própria).

```
1 if packet_readers_number == 0:
2    sys.exit("\nError compiling kernel source code. Provide at least
     one PE with packet handler's module!\n")
3
4 if packet_readers_number == 2 and len(
     static_mapping_packet_reader_list) != 2:
5    sys.exit("\nError compiling kernel source code. Provide two PEs
     with packet handler's module or check your static_addrs
     definition!\n")
6
7 if packet_readers_number == 1 and len(
     static_mapping_packet_reader_list) != 1:
8    sys.exit("\nError compiling kernel source code. Provide one PE
     with packet handler's module or check your static_addrs
     definition!\n")
9
10 if packet_readers_number == 2 and messages_payload[0][0] == 0 and
     messages_payload[1][0] == 0:
11    sys.exit("\nError compiling kernel source code. Provide two
     payloads for packet handler's modules or check your payload
     definition!\n")
12
13 if packet_readers_number == 1 and (messages_payload[0][0] == 0 or
     messages_payload[1][0] != 0):
14    sys.exit("\nError compiling kernel source code. Provide one
     payload with packet handler's module or check your payload
     definition!\n")
15
16 if packet_readers_number != len(static_mapping_packet_reader_list):
17    sys.exit("\nError compiling kernel source code. Provide the
     correct number of PEs!\n")
18
19 if messages_payload[0][0] != len(messages_payload[0][1]):
20    sys.exit("\nError compiling kernel source code. Provide the
     correct size for the first payload!\n")
21
22 if messages_payload[1][0] != len(messages_payload[1][1]):
```

```
23    sys.exit("\nError compiling kernel source code. Provide the
      correct size for the second payload!\n")
24
```

Outras verificações de consistências do arquivo de configuração são feitas no*script* `yaml_intf`, exemplificadas no Código-fonte 3.10, que é utilizado para a leitura e estruturação do dados dentro do arquivo *YAML*.

Código 3.10: Trecho de código que mostra como se evita a compilação para os casos em que o módulo PH é mal definido no arquivo de configuração *YAML* para o *script* `yaml_intf` (Fonte: Fonte própria).

```
1 if x_address > int(yaml_reader["hw"]["mpsoc_dimension"][0]) - 1 or
    y_address > int(yaml_reader["hw"]["mpsoc_dimension"][1]) - 1:
2       return "\nError compiling kernel source code. Provide a valid
     position for PE %d x %d!\n" % (x_address, y_address)
3 if int(first_payload_size) == 0 and "first" in
    static_mapping_payload:
4       return "\nError compiling kernel source code. Provide the
    size of payload for the first message!\n"
5 if int(second_payload_size) == 0 and "second" in
    static_mapping_payload:
6       return "\nError compiling kernel source code. Provide the
    size of payload for the second message!\n"
7
8 if int(first_payload_size) > 0 and "first" not in
    static_mapping_payload:
9       return "\nError compiling kernel source code. Provide the
    payload for the first message!\n"
10
11 if int(second_payload_size) > 0 and "second" not in
    static_mapping_payload:
12      return "\nError compiling kernel source code. Provide the
    payload for the second message!\n"
13
```

Além disso, caso tudo esteja configurado corretamente, são incluídos definições no arquivo `kernel_pkg.h` que contém informações relevantes sobre a plataforma utilizada para o gerenciamento do sistema pelo *kernel*, onde podemos citar o número de PEs com o módulo PH; a lista referente ao endereço na NoC dos PEs com o módulo PH e o tamanho, caso existam, do primeiro e segundo *payloads* das mensagens a serem enviadas pela NoC e suas respectivas listas, mostradas no Código-fonte 3.11.

Código 3.11: Trecho de código mostrando os campos adicionados no arquivo `kernel_pkg.h` (Fonte: Adaptado de [6]).

```
1 #define PACKET_HANDLER_NUMBER 5
2 #define FIRST_PAYLOAD_SIZE  20
3 #define SECOND_PAYLOAD_SIZE 15
4
5 extern const int packet_handler_routers_addr[PACKET_HANDLER_NUMBER];
6 extern const int first_payload[FIRST_PAYLOAD_SIZE];
7 extern const int second_payload[SECOND_PAYLOAD_SIZE];
8
```

A função de cada um desses campos adicionados é:

(a) `PACKET_HANDLER_NUMBER`: campo adicionado no arquivo para definir o número de PEs com o módulo PH no sistema;

(b) `FIRST_PAYLOAD_SIZES`: número de flits do *payload* da primeira mensagem enviada para um PH;

(c) `SECOND_PAYLOAD_SIZES`: número de flits do *payload* da segunda mensagem a ser enviada para um PH;

(d) `packet_handler_routers_addr`: lista de tamanho definida pelo campo `PACKET_HANDLER_NUMBER` que guardará os endereços dentro da NoC de todos os PEs com módulo PH;

(e) `first_payload`: lista contendo os dados dos flits do *payload* da primeira mensagem enviada para um PH;

(f) `second_payload`: lista contendo os dados dos flits do *payload* da segunda mensagem a ser enviada para um PH;

No arquivo `kernel_pkg.c`, são adicionados os valores as listas incluídas no arquivo `kernel_pkg.h` e modificado o último valor da lista da estrutura `ClusterInfo` referente ao número de páginas livres em cada *cluster*, exemplificado anteriormente na Figura 2.34. Esse número é decrementado no *script* `kernel_builder` caso um *cluster* tenha um ou mais PEs com o módulo PH. O número de páginas se traduz no número de tarefas que podem ser alocadas em um *cluster*.

Por exemplo, no trecho de código Código-fonte 3.12 temos uma MPSoC de dimensão 6×6 com 9 *clusters* de dimensão 2×2, 5 PEs com o módulo PH e 4 tarefas por processador. Assim, vemos que, do total de 12 páginas de memória no total disponíveis entre os três SPEs de cada *cluster*, os *clusters* 4, 6, 7 possuem um PE com o módulo PH em cada um dos seus elementos. Por isso, cada um deles perde

3 páginas de memória que poderiam estar disponíveis uma vez que a ausência de processador no PE com o módulo PH implica menos tarefas que o *cluster* pode ter em execução. Cada *cluster* possui, portanto, 8 páginas disponíveis para a utilização. Já o último *cluster* possui dois PEs com o módulo PH, ficando com um total de 4 páginas em memória para tarefas referente ao SPE com processador que restou.

Código 3.12: Trecho de código mostrando os campos adicionados e modificados no arquivo `kernel_pkg.c` (Fonte: Adaptado de [6]).

```
1 const int packet_handler_routers_addr[PACKET_HANDLER_NUMBER] = {259,
      1285, 261, 1284, 1027};
2 const int first_payload[FIRST_PAYLOAD_SIZE] = {30, 876, 76, 46,
     1859, 77, 78, 876, 6352, 763, 751, 63, 23, 345, 145, 543, 56745,
      435, 54, 90};
3 const int second_payload[SECOND_PAYLOAD_SIZE] = {0, 1, 1, 2, 3, 5,
     8, 13, 21, 34, 55, 89, 144, 233, 377};
4 ClusterInfo cluster_info[CLUSTER_NUMBER] = {
5         {0, 0, 0, 0, 1, 1, 12},
6         {2, 0, 2, 0, 3, 1, 12},
7         {4, 0, 4, 0, 5, 1, 12},
8         {0, 2, 0, 2, 1, 3, 8},
9         {2, 2, 2, 2, 3, 3, 12},
10        {4, 2, 4, 2, 5, 3, 8},
11        {0, 4, 0, 4, 1, 5, 8},
12        {2, 4, 2, 4, 3, 5, 12},
13        {4, 4, 4, 4, 5, 5, 4},
14 };
15
```

4. A execução do *script* `hw_builder` cria o arquivo `hemps_pkg.h` contendo as definições necessárias para os *hardwares* no sistema, mostrado no Código-fonte 3.13.

Código 3.13: Trecho de código mostrando todos os campos gerados do arquivo `hemps_pkg.h` (Fonte: Fonte própria).

```
1 #define INIT_ADDR_MEM           100
2 #define PAGE_SIZE_BYTES           32768
3 #define MEMORY_SIZE_BYTES      163840
4 #define TOTAL_REPO_SIZE_BYTES   1048576
5 #define APP_NUMBER           15
6 #define N_PE_X              6
7 #define N_PE_Y              6
8 #define N_PE                36
```

```
 9  const int pe_type[N_PE] = {1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
      0, 1, 0, 1, 0, 0, 2, 0, 0, 2, 0, 1, 0, 1, 0, 1, 2, 0, 2, 0, 0,
      0, 2};
10
```

A função de cada campo é:

- `INIT_ADDR_MEM`: campo adicionado no arquivo para definir o endereço da memória local para escrita ou leitura na memória local do PE, sendo seu valor dentro da lógica de controle da DMNI reduzido em quatro. Logo, por exemplo, *init_addr* com valor *100* será convertido para o endereço *96*, *0x00000060* em hexadecimal, para o início da escrita ou leitura em memória;
- `PAGE_SIZE_BYTES`: tamanho, em *bytes*, das páginas de memória;
- `MEMORY_SIZE_BYTE`: tamanho, em *bytes*, da memória local;
- `TOTAL_REPO_SIZE_BYTES`: tamanho, em *bytes*, do repositório local de aplicações;
- `APP_NUMBER`: o número total de aplicações no sistema;
- `N_PE_X`: as dimensões da MPSoC no eixo x;
- `N_PE_Y`:as dimensões da MPSoC no eixo y;
- `N_PE`: número total de PEs;
- `pe_type`: lista de tamanho N_PE contendo os diferentes tipos de PEs presentes no sistema.

Além disso, os tipos de PEs criados no sistema definidos em `pe_type` são:

- *0*: cria um MPE, sendo o primeiro PE sempre o SMPE;
- *1*: cria um SPE contendo um processador;
- *2*: definido para qualquer SPE que não contém o processador, nesse estudo sendo utilizado para SPEs com o módulo PH. Definido durante a execução do *script* `hw_builder`.

O campo `INIT_ADDR_MEM` é validado no *script* `hw_builder` em tempo de compilação interrompendo-se a execução da aplicação caso o endereço de memória não seja suficientemente grande para armazenar um flit, conforme mostrado no Código-fonte 3.14.

Código 3.14: Trecho de código do *script* `hw_builder` que mostra compilação é evitada caso o endereço de memória inicial para a manipulação da memória local seja inválido (Fonte: Fonte própria).

```
1 if init_addr % 4 is not 0:
2     sys.exit("\nError compiling hardware source code. Invalid
     address for packet handler's memory.\n");
3
```

Após gerar o arquivo `hemps_pkg.h`, o *hardware* do sistema é gerado para a MPSoC na linguagem *C++* utilizando a biblioteca *SystemC* e os arquivos de *software* na linguagem *C*.

3.2.4 Mudanças nos módulos de *software*

Os módulos de *software* da HeMPS são bibliotecas na linguagem *C* que fornecem todas as funcionalidades e implementações necessárias para que os *kernels* mestre e escravo executem suas funções.

Esses módulos de *software*, suas funções e quais núcleos os utilizam são:

- *Applications* (do inglês Aplicações): implementa as chamadas que gerenciam a inserção, acesso e remoção da estrutura definida para aplicações. Utilizada apenas pelo *kernel* mestre [6];
- *Communication* (do inglês Comunicação): fornece as chamadas para o controle das estruturas responsáveis pela requisição de mensagens e do *PIPE*. Utilizada apenas pelo *kernel* escravo [6];
- *New Task* (do inglês Nova Tarefa): implementa as chamadas necessárias para a inserção de uma nova tarefa em uma fila de controle de tarefas *FIFO*. Utilizada apenas pelo *kernel* mestre [6];
- *Packet* (do inglês Pacote): fornece as chamadas responsáveis por programar a DMNI via *software* para o envio e recebimento de pacotes, sendo uma abstração da utilização da NoC pelos outros módulos de *software*. Esse módulo é utilizado tanto pelo *kernel* mestre quanto pelo *kernel* escravo [6];
- *Pending Service* (do inglês Serviço Pendente): implementa a chamadas para o controle da fila de tarefas *FIFO* dos pacotes que chegam nos PEs escravos, porém não podem ser imediatamente manipulados. Utilizada apenas pelo *kernel* escravo [6];
- *Processors* (do inglês Processadores): este módulo implementas as funcionalidades responsáveis pela gerência dos processados dos SPEs. Utilizada apenas pelo *kernel* mestre [6];

- *Reclustering* (do inglês Re-aglomeração): fornece as chamadas para o *reclustering*, funcionalidade que habilita o sistema a realocar tarefas de um *cluster* em outro como mais recursos, no caso maior número de páginas em memória livres. Utilizada apenas pelo *kernel* mestre [6];
- *Resource Manager* (do inglês Gerente de Recursos): implementa as chamadas que selecionam onde executar tarefas e aplicações. Sendo assim, esse módulo lida com a gerência de recursos dos *clusters* e mapeamento de tarefas, aplicações e heurísticas de migração de tarefas. Utilizada apenas pelo *kernel* mestre, mais precisamente o SMPE [6];
- *Task Control* (do inglês Controle de Tarefa): fornece as funções e suas chamadas para o controle da estrutura do *task control block* (do inglês bloco de controle de tarefa (TCB)). Utilizada apenas pelo *kernel* escravo [6];
- *Task Location* (do inglês Localização de Tarefa): fornece as chamadas relativas a estrutura de localização de tarefas, utilizada pelo *kernel* escravo para saber o endereço dentro da NoC de uma outra tarefa [6];
- *Task Migration* (do inglês Migração de Tarefa): implementa todas as funcionalidades e chamadas responsáveis para a migração de tarefas entre processadores. Utilizada apenas pelo *kernel* escravo [6];
- *Task Scheduler* (do inglês Agendador de Tarefa): esse módulo é responsável por implementar um algoritmo para escalonamento de tarefas segundo uma abordagem baseada no algoritmo *LST* (*Least Slack Time*), caracterizando-se por otimizar o uso de processadores no sistema uma vez que define o processador a executar as tarefas de acordo com o tempo restante para o fim de uma tarefa após o início de sua execução (*slack time*). Utilizada apenas pelo *kernel* escravo [6].

Além disso, temos os seguintes módulos que possuem definições utilizadas dentro do sistema:

- Interface de Programação de Aplicativos (*Application Programming Interface*) (API): implementa serviços para troca de mensagens entre tarefas [6];
- *Plasma*: define parâmetros para a utilização do processador *Plasma* utilizado pelos PEs na HeMPS [6];
- *Services* (do inglês Serviços): define todos os serviços utilizados pelos *kernels*. Esses serviços são utilizados para identificar os diferentes tipos de pacotes do sistema [6].

Dentre todos os módulos descritos, foram modificados os módulos de *software Services*, *Processors* e *Resource Manager* para adequar o módulo PH ao sistema.

No módulo *Services* foi adicionado um novo serviço para o envio e recebimento de mensagens entre PEs com o novo IP e o SMPE. Esse serviço foi denominado `MY_PACKET` e sua definição mostrada no Código-fonte 3.15.

Código 3.15: Trecho de código do módulo de *software Services* com a definição do novo serviço `MY_PACKET` (Fonte: Fonte própria).

```
1 #define MY_PACKET 0x00000300
```

No módulo *Processors* foram modificadas as funções para atualização e retorno do *slack time*, além da função para inserção de um processador na lista de processadores em uso.

Na função de atualização do *slack time*, mostrada no Código-fonte 3.16, verifica-se na linha 13 se o endereço do processador na NoC está dentro da lista de endereços da NoC dos PEs com o módulo PH definida em `packet_handler_routers_addr` do arquivo `hemps_pkg.h`. Se não estiver, o retorno da função chamada na linha 11 será -1 e, com isso, o endereço do processador passado não possui um módulo PH e, assim, a lógica original da função é aplicada na linha 14 e 15, do contrário o seu *slack time* é atribuído para zero, significando para o algoritmo de alocação de tarefas que o processador em questão está sempre ocupado, comportamento desejado visto que PEs com o módulo PH não possuem um processador e a alocação de tarefas nesse módulo gera erros na execução do sistema.

Código 3.16: Trecho de código do módulo de *software Processors* com a função modificada para a atualização do *slack time* (Fonte: Adaptado de [6]).

```
1 /**Updates the processor slack time
2  * \param proc_address Processor address
3  * \param slack_time Slack time in percentage
4  */
5 void update_proc_slack_time(int proc_address, int slack_time){
6
7         Processor * p = search_processor(proc_address);
8
9         int valid_proc;
10
11        valid_proc = find_index(packet_reader_routers_addr,
      packet_readers_number_of_pes, proc_address);
12
13        if(valid_proc == -1){
14                p->slack_time += slack_time;
```

```
15                  p->total_slack_samples++;
16          } else {
17                  p->slack_time = 0;
18                  p->total_slack_samples = 0;
19          }
20 }
```

As funções de retorno de *slack time* e inserção de um novo processador na lista de processadores em uso foram mudadas de maneira análoga, verificando-se se o endereço do processador na NoC passado como parâmetro de entrada não consta na lista de PEs que têm um módulo PH. Se não tiverem, a lógica original da função é mantida, do contrário na função de retorno do *slack time* é sempre retornado o valor 0 significando para o sistema que o PE com o módulo PH sempre estará ocupado e nunca receberá uma tarefa. Já na função para inicialização de um novo processador, o número de páginas em memória para o PE com o módulo PH tem seu valor inicializado em 0, uma vez que esse elemento não receberá tarefas, e o endereço do processador na NoC inicializado em -1, garantindo que nenhuma tarefa será mandada a esse PE, uma vez que na lógica de controle de processadores aquele endereço é inválido.

Os códigos para as função de retorno de *slack time* e inserção de um novo processador são mostradas no Código-fonte 3.17 e no Código-fonte 3.18, respectivamente.

Código 3.17: Trecho de código do módulo de *software Processors* com a função modificada para o retorno do *slack time* (Fonte: Adaptado de [6]).

```
1 /**Gets the processor slack time
2  * \param proc_address Processor address
3  * \return The processor slack time in percentage
4  */
5 int get_proc_slack_time(int proc_address){
6
7         Processor * p = search_processor(proc_address);
8
9         int valid_proc;
10
11        valid_proc = find_index(packet_reader_routers_addr,
      packet_readers_number_of_pes, proc_address);
12
13        if(valid_proc != -1){
14                return 0;
15        }
16
17        if (p->total_slack_samples == 0)
18                return 100;
```

```
19
20         return (p->slack_time/p->total_slack_samples);
21 }
```

Código 3.18: Trecho de código do módulo de *software Processors* com a função modificada para a inserção de um novo processador na lista de processadores em uso (Fonte: Adaptado de [6]).

```
1 /**Add a valid processor to the processors' array
2  * \param proc_address Processor address to be added
3  */
4 void add_procesor(int proc_address){
5
6         Processor * p = search_processor(-1); //Searches for a free slot
7
8         int valid_proc;
9
10        valid_proc = find_index(packet_reader_routers_addr,
      packet_readers_number_of_pes, proc_address);
11
12        p->free_pages = (valid_proc == -1 ? MAX_LOCAL_TASKS : 0);
13
14        p->address = (valid_proc == -1 ? proc_address : -1);
15
16        p->slack_time = 0;
17
18        p->total_slack_samples = 0;
19
20        for(int i=0; i<MAX_LOCAL_TASKS; i++){
21                p->task[i] = -1;
22        }
23
24        if(valid_proc != -1){
25                putsv("WARNING: Setting Processor number of pages to 0
      into PE reader proc ", proc_address);
26        }
27 }
```

No módulo *Resource Manager* temos a mudança das funções de mapeamento de tarefas e aplicações, onde, seguindo a mesma lógica aplicada nas funções dos outros módulos, se o endereço do processador na NoC estiver na lista de PEs com o módulo PH sua lógica é alterada, do contrário mantida.

O Código-fonte 3.19 mostra o código de retorno da função de mapeamento de aplicação. Seu comportamento original é o retorno do endereço do processador encontrado

para mapear a aplicação e, caso o endereço recebido seja de um PE com o módulo PH, uma mensagem de aviso que acontecerá o *reclustering* das tarefas da aplicação que fez a requisição, uma vez que o processador não possui recursos para a execução de tarefas.

Código 3.19: Trecho de código do módulo de *software Resource Manager* utilizada na lógica para o mapeamento de uma aplicação em um processador de um *cluster*(Fonte: Adaptado de [6]).

```
1 valid_proc = find_index(packet_reader_routers_addr,
     packet_readers_number_of_pe, proc_address);
2         if (valid_proc == -1){
3             return proc_address;
4         } else {
5             putsv("WARNING: no resources available in cluster to map task
      ", task_id);
6                 return -1;
7         }
```

O Código-fonte 3.20 mostra o trecho de código da função de mapeamento de tarefa. Seu funcionamento original é atualizar o processador escolhido para mapear a tarefa na linha 5, bem como seu *slack time* na linha 6 para, se possível, ao varrer todos os processadores do sistema o algoritmo escolherá o melhor candidato possível para alocar a tarefa. Porém, caso o endereço para o processador contenha um módulo PH, o algoritmo ignorará esse PE em questão para a alocação da tarefa, exemplificado na linha 4. Consequentemente, caso não seja encontrado nenhum PE com um processador disponível, acontecerá o *reclustering* das tarefas da aplicação que fez a requisição.

Código 3.20: Trecho de código do módulo de *software Processors* com a função modificada para o mapeamento de uma tarefa em um processador de um *cluster* (Fonte: Adaptado de [6]).

```
1 if (get_proc_free_pages(proc_address) > 0){
2         slack_time = get_proc_slack_time(proc_address);
3
4         if (max_slack_time < slack_time && valid_proc == -1){
5                 canditate_proc = proc_address;
6                 max_slack_time = slack_time;
7         }
8 }
```

Por fim, foram feitas modificações para a inicialização dos SPEs dentro do *kernel* mestre para controle de como os SPEs são criados e seus processadores, se existentes, inicializados no módulo de *software Processors* e suas páginas livres controladas e alocadas pelo módulo de *software Resource Manager*, exemplificado no Código-fonte 3.21.

Código 3.21: Trecho de código do *kernel* mestre para a inicialização dos SPEs e seus processadores, desde que os SPEs não tenham um módulo PH (Fonte: Adaptado de [6]).

```
1 proc_address = i*256 + j; //Forms the proc address
2 valid_proc = find_index(packet_reader_routers_addr, packet_readers_number
      , proc_address);
3 for(int z = 0; z < CLUSTER_NUMBER; z++){
4 cluster_master_address = (cluster_info[z].master_x << 8) | cluster_info[z
      ].master_y;
5    if(cluster_master_address == proc_address){
6           valid_proc = -1;
7           break;
8    }
9 }
10
11 if( proc_address != net_address ) {
12         //Fills the struct processors
13         add_procesor(proc_address);
14         if(valid_proc == -1){
15            //Sends a packet to the slave
16            p = get_service_header_slot();
17
18            p->header = proc_address;
19
20            p->service = INITIALIZE_SLAVE;
21
22            send_packet(p, 0, 0);
23         }
24
25         index_counter++;
26 }
```

Capítulo 4

Resultados

O objetivo deste capítulo é descrever, de maneira geral, os diferentes testes feitos no sistema para verificar seu comportamento com a inserção do módulo PH.

Na seção 4.1, são explicadas as diferentes aplicações utilizadas para demonstrar o comportamento da plataforma com o módulo PH, contrastando com o comportamento na implementação original, além dos casos de teste para exemplificar essas diferenças e os objetivos a serem vistos em cada teste.

A seção 4.2 aborda com mais detalhes os casos de teste feitos para validar o sistema com o novo módulo. Além disso, é mostrado o funcionamento dentro do sistema por meio dos resultados gerados e interpretados durante simulações da plataforma.

Na seção 4.3, são analisadas as principais diferenças observadas entre a versão original e a versão alterada com o módulo PH, bem como os cuidados necessários que devem ser analisados para a integração de outros módulos de *hardware* na plataforma.

4.1 Aplicações de teste

A plataforma HeMPS disponibiliza um diretório denominado *applications* com diferentes aplicações que servem como casos de teste a serem utilizadas em diferentes configurações do sistema.

Foram utilizadas 5 delas para se fazer análises com o novo IP. As análises de teste englobam casos de validação do arquivo de configuração para criação da plataforma; exemplos funcionais de envio e recebimentos de pacotes dentro do módulo PH e comparações observadas em relação a plataforma e sua implementação original. As 5 aplicações são:

- A aplicação *prod_cons*, mostrada na Figura 4.1, possui 2 tarefas em um modelo produtor-consumidor. Neste modelo, uma das tarefas denominada produtora fica encarregada de produzir 50 mensagens, definidas no arquivo fonte *prod_cons_std.h*

que serão enviadas para uma segunda tarefa denominada consumiidora que, ao recebê-las, manda uma outra mensagem reconhecendo a chegada de cada uma dessas mensagens;

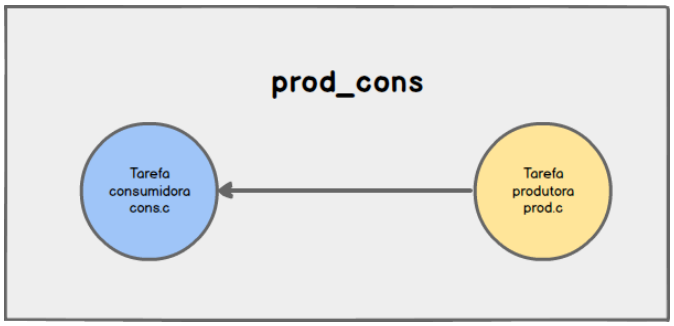


Figura 4.1: Fluxograma da aplicação *prod_cons* com suas tarefas atuando em uma organização produtor-consumidor com o nome das tarefas e seus respectivos arquivos fontes (Fonte: Adaptada de [2]).

- A aplicação *djikstra* implementa o algoritmo de mesmo nome que procura o menor caminho entre cada um dos pares de vértices do grafo. Essa aplicação possui 7 tarefas que são organizadas de acordo com fluxograma ilustrado na Figura 4.2;

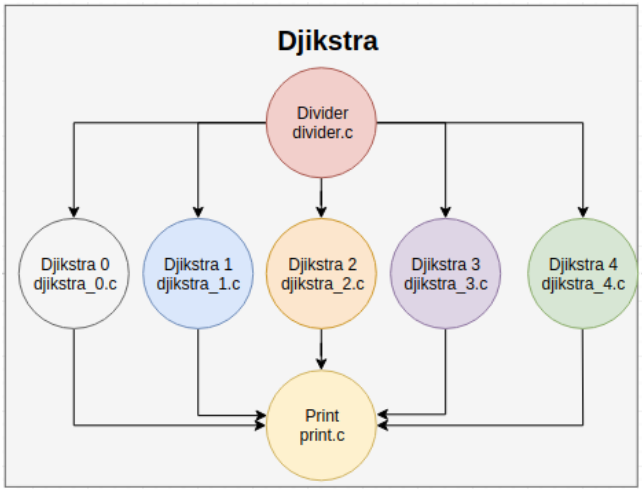


Figura 4.2: Fluxograma da aplicação *djikstra* com o nome das tarefas e seus respectivos arquivos fontes (Fonte: [2]).

- A aplicação *mpeg* produz cinco tarefas, como mostrado na Figura 4.3, para decodificação de arquivos *mpeg* e *jpeg* que interagem entre si de maneira sequencial por troca de uma mensagem por tarefa, correspondente a 50 quadros (*frames*) estipulados e definidos no arquivo fonte *mpeg_std.h*;

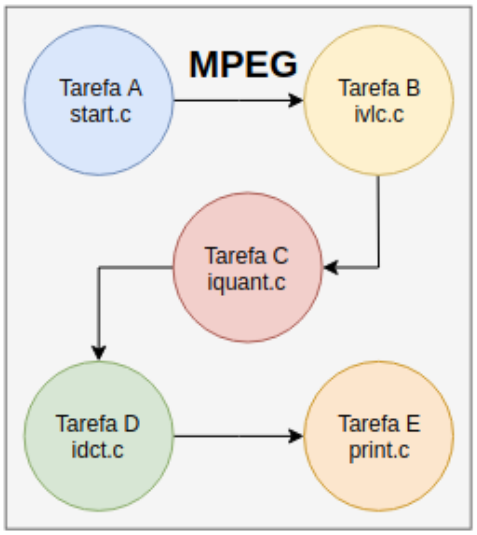


Figura 4.3: Fluxograma da aplicação *mpeg* com o nome das tarefas e seus respectivos arquivos fontes (Fonte: [2]).

- A aplicação *dtw* executa o algoritmo *Dynamic Time Warping* que procura o menor caminho entre duas séries temporais. O intuito desta busca é manter as séries sincronizadas e encontrar padrões entre suas distâncias. A Figura 4.4 mostra a estrutura e a comunicação das 6 tarefas dessa aplicação;

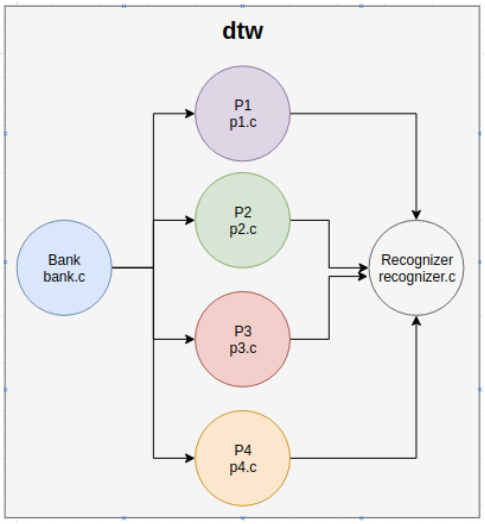


Figura 4.4: Fluxograma da aplicação *dtw* com o nome das tarefas e seus respectivos arquivos fontes (Fonte: [2]).

- A aplicação *synthetic* é uma aplicação sintética criada para fazer comunicações por meio de mensagens entre 6 tarefas em um estrutura exemplificada na Figura 4.5.

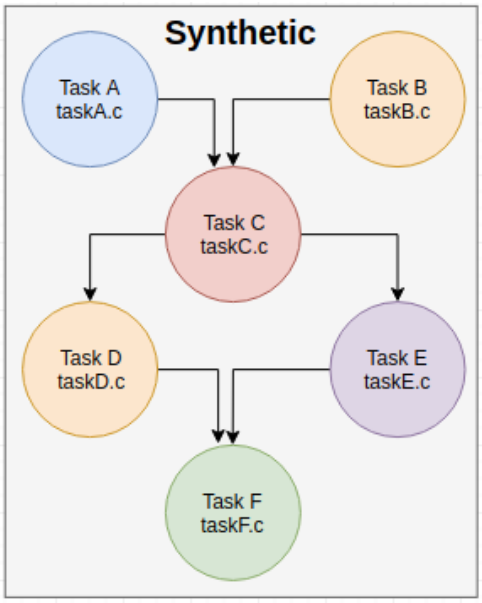


Figura 4.5: Fluxograma da aplicação *synthetic* com o nome das tarefas e seus respectivos arquivos fontes (Fonte: [2]).

Para se fazer as análises, foram definidos 9 casos de teste com tempo de simulação de 75 milissegundos, observando-se sua performance e erros de compilação. Os testes são:

- *Primeiro caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição x = 1 e y = 1 possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.1.

  O intuito desse teste é verificar as mudanças no mapeamento de tarefas, comunicação dentro do MPSoC e demonstrar que tarefas não são mapeadas em SPEs com o módulo *PH*.

  Código 4.1: Configuração do arquivo *YAML* para o primeiro teste (Fonte: Adaptado de [6]).

```
1 hw:
2    page_size_KB:  32
3    tasks_per_PE: 4
4    repository_size_MB:  1
5    model_description: sc     # sc (gcc) | scmod (questa) | vhdl
6    noc_buffer_size: 8        # must be power of 2
7    mpsoc_dimension: [2,2]      # for while, must be a square shape
```

```
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 20 #any unsigned integer number
23     second_payload_size: 0 #any unsigned integer number
24     payload:
25       first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27       0: [1,1] # packet handler at address X=1, Y=1
28
```

- *Segundo caso de teste*: esse cenário utiliza uma instância da aplicação *mpeg*, uma instância da aplicação *dtw*, uma instância da aplicação *dijkstra* e uma instância da aplicação *synthetic*. O MPSoC utilizado possui dimensão 4×4 com um total de 16 PEs. Cada SPE que possua processador é limitado a executar 4 tarefas. O sistema possui 4 *clusters* de dimensão 2×2 para um total de 24 tarefas alocadas.

  Os SPEs nas posições $(x = 3, y = 3)$ e $(x = 0, y = 1)$ possuem o módulo PH. Sua configuração é exemplificada no Código-fonte 4.2.

  O intuito desse teste é demonstrar o uso da técnica de *re-clustering* em um sistema saturado, ou seja, que utiliza todos os seus recursos disponíveis.

  Código 4.2: Configuração do arquivo *YAML* para o segundo teste (Fonte: Adaptado de [6]).

```
1 hw:
2    page_size_KB:  32
3    tasks_per_PE: 4
4    repository_size_MB:  1
5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
6    noc_buffer_size: 8       # must be power of 2
7    mpsoc_dimension: [4,4]     # for while, must be a square shape
```

```
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  lst # round_robing | lst
13 apps:
14   - name: mpeg
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 0 #any unsigned integer number
18   - name: dijkstra
19     start_time_ms: 0 #any unsigned integer number
20   - name: synthetic
21     start_time_ms: 0 #any unsigned integer number
22 chw:
23   - name: packet_reader
24     number_pe: 2 #any unsigned integer number
25     init_addr: 0x00000064
26     first_payload_size: 20 #any unsigned integer number
27     second_payload_size: 15 #any unsigned integer number
28     payload:
29       first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
       63, 23, 345, 145, 543, 56745, 435, 54, 90]
30       second: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
       377]
31     static_addrs:
32       0: [3,3] # packet handler at address X=3, Y=3
33       1: [0,1] # packet handler at address X=0, Y=1
34
```

- *Terceiro caso de teste*: esse cenário utiliza três instâncias da aplicação *mpeg*, uma instância da aplicação *dtw*, duas instâncias da aplicação *dijkstra*, três instâncias da aplicação *synthetic* e duas instâncias da aplicação *prod_cons*. O MPSoC utilizado possui dimensão 4×4 com um total de 16 PEs. Cada SPE que possua processador é limitado a executar 4 tarefas. O sistema possui 4 *clusters* de dimensão 2×2 para um total de 63 tarefas alocadas.

  O SPE na posição x = 3, y = 3 possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.3.

  O intuito desse teste é demonstrar o comportamento do MPSoC com SPEs com o módulo PH em casos do da técnica de *re-clustering* em um sistema saturado, ou seja, que utiliza todos os recursos disponíveis.

Código 4.3: Configuração do arquivo *YAML* para o terceiro teste (Fonte: Adaptado de [6]).

```
hw:
   page_size_KB:  32
   tasks_per_PE: 4
   repository_size_MB:  1
   model_description: sc    # sc (gcc) | scmod (questa) | vhdl
   noc_buffer_size: 8       # must be power of 2
   mpsoc_dimension: [4,4]     # for while, must be a square shape
   cluster_dimension: [2,2] # for while, must be a square shape
   master_location: LB      # LB
sw:
   mapping_algorithm: WithLoad  # WithLoad
   task_scheduler:  lst # round_robing | lst
apps:
  - name: mpeg
    start_time_ms: 0 #any unsigned integer number
  - name: dtw
    start_time_ms: 0 #any unsigned integer number
  - name: dijkstra
    start_time_ms: 0 #any unsigned integer number
  - name: synthetic
    start_time_ms: 0 #any unsigned integer number
  - name: mpeg
    start_time_ms: 2 #any unsigned integer number
  - name: prod_cons
    start_time_ms: 2 #any unsigned integer number
  - name: synthetic
    start_time_ms: 2 #any unsigned integer number
  - name: dijkstra
    start_time_ms: 2 #any unsigned integer number
  - name: mpeg
    start_time_ms: 3 #any unsigned integer number
  - name: prod_cons
    start_time_ms: 3 #any unsigned integer number
  - name: synthetic
    start_time_ms: 3 #any unsigned integer number
chw:
  - name: packet_reader
    number_pe: 1 #any unsigned integer number
    init_addr: 0x00000064
    first_payload_size: 20 #any unsigned integer number
    second_payload_size: 0 #any unsigned integer number
    payload:
```

```
43       first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
44     static_addrs:
45       0: [3,3] # packet handler at address X=3, Y=3
46
```

- *Quarto caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição x = 1 e y = 1 possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.4.

  O intuito desse teste é demonstrar o erro de compilação quando a posição do módulo PH não é definida como sendo a mesma daquela definida para o SMPE.

  Código 4.4: Configuração do arquivo *YAML* para o quarto teste (Fonte: Adaptado de [6]).

```
 1 hw:
 2    page_size_KB:  32
 3    tasks_per_PE: 4
 4    repository_size_MB:  1
 5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
 6    noc_buffer_size: 8       # must be power of 2
 7    mpsoc_dimension: [2,2]     # for while, must be a square shape
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 20 #any unsigned integer number
23     second_payload_size: 0 #any unsigned integer number
24     payload:
```

```
25        first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27        0: [0,0] # packet handler at address X=0, Y=0
28
```

- *Quinto caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição $x = 1$ e $y = 1$ possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.5.

  O intuito desse teste é demonstrar o erro de compilação quando o tamanho do *payload* da primeira mensagem a ser mandada ao SPE com o módulo PH não é definida propriamente.

  Código 4.5: Configuração do arquivo *YAML* para o quinto teste (Fonte: Adaptado de [6]).

```
 1 hw:
 2    page_size_KB:  32
 3    tasks_per_PE: 4
 4    repository_size_MB:  1
 5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
 6    noc_buffer_size: 8       # must be power of 2
 7    mpsoc_dimension: [2,2]     # for while, must be a square shape
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 19 #any unsigned integer number
23     second_payload_size: 0 #any unsigned integer number
24     payload:
```

```
25       first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27       0: [1,1] # packet handler at address X=1, Y=1
28
```

- *Sexto caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição $x = 1$ e $y = 1$ possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.6.

  O intuito desse teste é demonstrar o erro de compilação quando o número de SPEs com o módulo PH no sistema não é definido propriamente.

  Código 4.6: Configuração do arquivo *YAML* para o sexto teste (Fonte: Adaptado de [6]).

```
 1 hw:
 2    page_size_KB:  32
 3    tasks_per_PE: 4
 4    repository_size_MB:  1
 5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
 6    noc_buffer_size: 8       # must be power of 2
 7    mpsoc_dimension: [2,2]     # for while, must be a square shape
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 2 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 20 #any unsigned integer number
23     second_payload_size: 0 #any unsigned integer number
24     payload:
```

```
25        first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27        0: [1,1] # packet handler at address X=1, Y=1
28
```

- *Sétimo caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição $x = 1$ e $y = 1$ possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.7.

  O intuito desse teste é demonstrar o erro de compilação quando a posição de um SPE com o módulo PH no sistema não é definido propriamente, ultrapassando as dimensões definidas para a MPSoC.

  Código 4.7: Configuração do arquivo *YAML* para o sétimo teste (Fonte: Adaptado de [6]).

```
1 hw:
2    page_size_KB:  32
3    tasks_per_PE: 4
4    repository_size_MB:  1
5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
6    noc_buffer_size: 8       # must be power of 2
7    mpsoc_dimension: [2,2]     # for while, must be a square shape
8    cluster_dimension: [2,2] # for while, must be a square shape
9    master_location: LB      # LB
10 sw:
11   mapping_algorithm: WithLoad  # WithLoad
12   task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 20 #any unsigned integer number
23     second_payload_size: 0 #any unsigned integer number
24     payload:
```

```
25        first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
      63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27        0: [3,3] # packet handler at address X=3, Y=3
28
```

- *Oitavo caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição $x = 1$ e $y = 1$ possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.8.

  O intuito desse teste é demonstrar o erro de compilação quando o tamanho do *payload* da segunda mensagem a ser mandada ao SPE com o módulo PH não é definida propriamente, sendo nesse caso a mensagem não foi definida e seu tamanho foi configurado para ser maior que zero.

  Código 4.8: Configuração do arquivo *YAML* para o oitavo teste (Fonte: Adaptado de [6]).

```
 1 hw:
 2    page_size_KB:  32
 3    tasks_per_PE: 4
 4    repository_size_MB:  1
 5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
 6    noc_buffer_size: 8       # must be power of 2
 7    mpsoc_dimension: [2,2]     # for while, must be a square shape
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000064
22     first_payload_size: 20 #any unsigned integer number
23     second_payload_size: 1 #any unsigned integer number
```

```
24     payload:
25       first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
       63, 23, 345, 145, 543, 56745, 435, 54, 90]
26     static_addrs:
27       0: [1,1] # packet handler at address X=1, Y=1
28
```

- *Nono caso de teste*: esse cenário utiliza uma instância *prod_cons* e *dtw* em um MPSoC 2×2 com um total de 4 PEs. Seu único *cluster* possui dimensão 2×2 para um total de 8 tarefas em execução. Cada SPE que possua processador é limitado a executar 4 tarefas.

  O SPE na posição $x = 1$ e $y = 1$ possui o módulo PH. Sua configuração é exemplificada no Código-fonte 4.9.

  O intuito desse teste é demonstrar o erro de compilação quando o endereço definido para a leitura e escrita em memória das mensagens a serem enviadas não é definido propriamente, estando desalinhado em relação as outras posições da memória local do SPE.

  Código 4.9: Configuração do arquivo *YAML* para o nono teste (Fonte: Adaptado de [6]).

```
 1 hw:
 2    page_size_KB:  32
 3    tasks_per_PE: 4
 4    repository_size_MB:  1
 5    model_description: sc    # sc (gcc) | scmod (questa) | vhdl
 6    noc_buffer_size: 8       # must be power of 2
 7    mpsoc_dimension: [2,2]     # for while, must be a square shape
 8    cluster_dimension: [2,2] # for while, must be a square shape
 9    master_location: LB      # LB
10 sw:
11    mapping_algorithm: WithLoad  # WithLoad
12    task_scheduler:  round_robing # round_robing | lst
13 apps:
14   - name: prod_cons
15     start_time_ms: 0 #any unsigned integer number
16   - name: dtw
17     start_time_ms: 5 #any unsigned integer number
18 chw:
19   - name: packet_reader
20     number_pe: 1 #any unsigned integer number
21     init_addr: 0x00000063
22     first_payload_size: 20 #any unsigned integer number
```

```
23      second_payload_size: 0 #any unsigned integer number
24      payload:
25        first: [30, 876, 76, 46, 1859, 77, 78, 876, 6352, 763, 751,
       63, 23, 345, 145, 543, 56745, 435, 54, 90]
26      static_addrs:
27        0: [1,1] # packet handler at address X=1, Y=1
28
```

4.2 Análise do funcionamento do sistema

As análises sobre o funcionamento do sistemas foram elaboradas para demonstrar a validade da nova configuração para a MPSoC, ou seja, que, pelo menos, o envio de mensagens de novos serviços pela NoC e casos com diversas tarefas mapeadas no sistema são capazes de serem concluídos corretamente [2]. Toda essa demonstração será subdivida em 4 partes.

Na subseção 4.2.1 verificamos como o sistema lida quando a inclusão de um SPE com o módulo PH, além de explicações pertinentes sobre comportamentos observados durante a simulação da plataforma.

Na subseção 4.2.2 explicaremos o remapeamento das tarefas dentro do sistema.

Na subseção 4.2.3 verificamos como o sistema lida com o mapeamento e execução de diversas tarefas em um contexto onde as páginas de memória são totalmente utilizadas e, com isso, como mesmo assim ele readapta sua lógica de gerenciamento de tarefas para acomodar SPEs com módulos PH.

Por fim, a subseção 4.2.4 aborda casos de erros de configuração do sistema durante sua compilação, explicando como os *scripts* de geração lidam com esses erros.

Em todas essas subseções, a análise será melhor exemplificada por meio do depurador da plataforma denominado *HeMPS_Debugger* [2] e mensagens dos sistema geradas durante sua geração e simulação dos testes.

4.2.1 Análise do mapeamento de tarefas e comunicação do sistema com a inclusão do módulo PH

A execução do sistema criado pela configuração feita no primeiro teste, visto no Código-fonte 4.1, ocorre como esperado durante toda a simulação. Primeiramente, vemos na Figura 4.6 o momento em que os PEs do sistema são gerados por meio de mensagens imprensas durante sua criação.

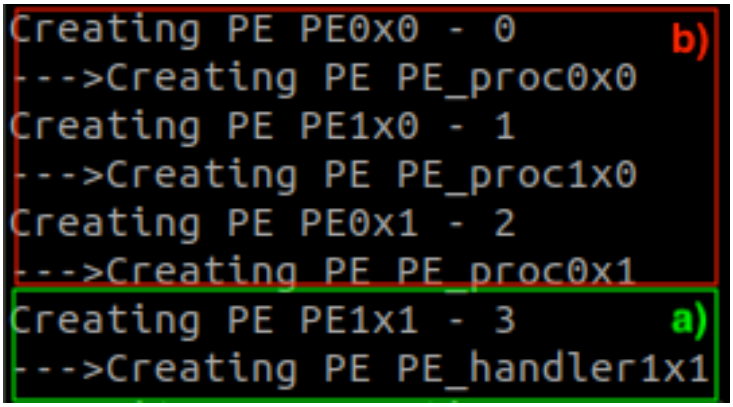


Figura 4.6: Mensagens da compilação do sistema durante a inicialização dos PEs. Em (a), temos a criação do PE com o módulo PH, denominado *PE_handler*, e em (b) os PEs com o processador incluso, denominados *PE_proc* (Fonte: Fonte Própria).

Em seguida, a simulação do sistema é inicializada onde, por meio do depurador *HeMPS_Debugger* mostrado na Figura 4.7, temos a visão geral da MPSoC criada de dimensão 2×2 na parte a), uma tabela das tarefas alocadas no sistema na parte b) e uma tabela com todos os serviços do sistema na parte c).

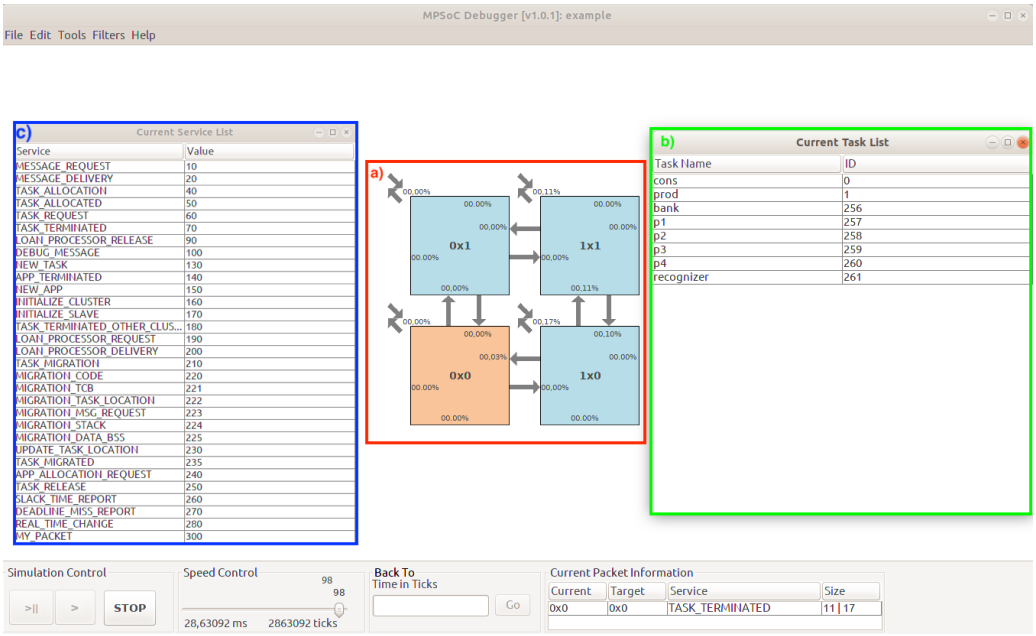


Figura 4.7: Visão geral do sistema gerado. Em (a) temos o MPSoC criado. Em (b) temos todas as tarefas a serem mapeadas para execução. Em (c) temos todas os serviços com seus códigos de identificação dentro do sistema (Fonte: Fonte Própria).

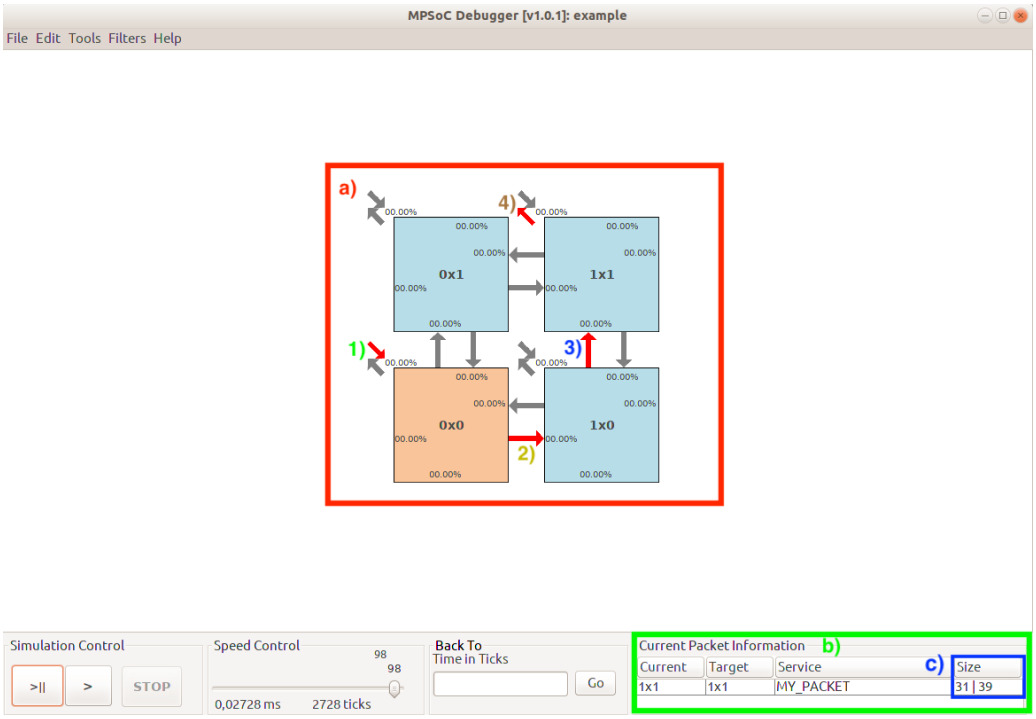


Figura 4.8: Envio do serviço *MY_PACKET* pela NoC do SMPE para o SPE com o módulo PH. Em (a) temos os passos do envio da mensagem listados de 1) a 4) e em (b) o tipo de serviço enviado pela NoC, com sua origem, destino, tipo de serviço e tamanho da mensagem mostrada na parte esquerda de (c) (Fonte: Fonte Própria).

Antes do mapeamento de todas as tarefas, temos o envio da mensagem do serviço *MY_PACKET*, sendo o caminho percorrido pela NoC. Como pode ser vista na Figura 4.8 a), temos primeiro em 1) o envio da mensagem do SMPE para a NoC, percorrendo-a nos passos 2) e 3) até o recebimento no SPE no passo 4). A mensagem recebida pelo SPE pode ser vista na Figura 4.9, sendo a parte a) referente ao *header* da mensagem e a parte b) referente ao *payload* da mensagem.

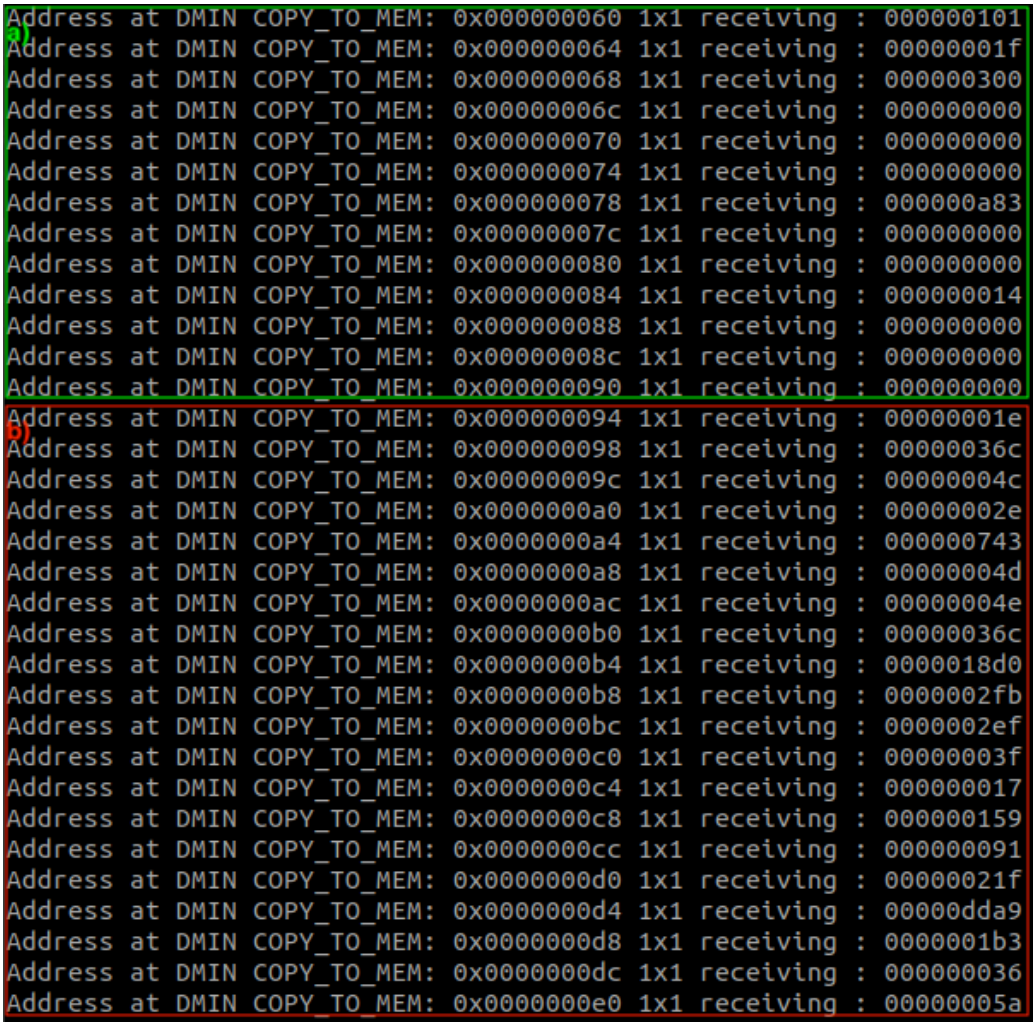


Figura 4.9: Mensagem do serviço *MY_PACKET* recebida pelo SPE com o módulo PH. Em (a) temos o *header* da mensagem com seu endereço de escrita em memória e o conteúdo de cada flit em hexadecimal. Em (b), temos o *payload* da mensagem com seu endereço de escrita em memória e o conteúdo de cada flit em hexadecimal (Fonte: Fonte Própria).

O re-envio da mesma mensagem pela NoC pode ser visto na Figura 4.10, com todo o processo listado do passo 4) ao passo 7), seguindo o algoritmo de roteamento determinístico XY definido na NoC. A mensagem recebida pelo SMPE pode ser vista na Figura 4.11, sendo a parte a) referente a mensagens do sistema sobre a inicialização do SPE com o módulo PH e a parte b) referente a informações do *header* e *payload* da mensagem.

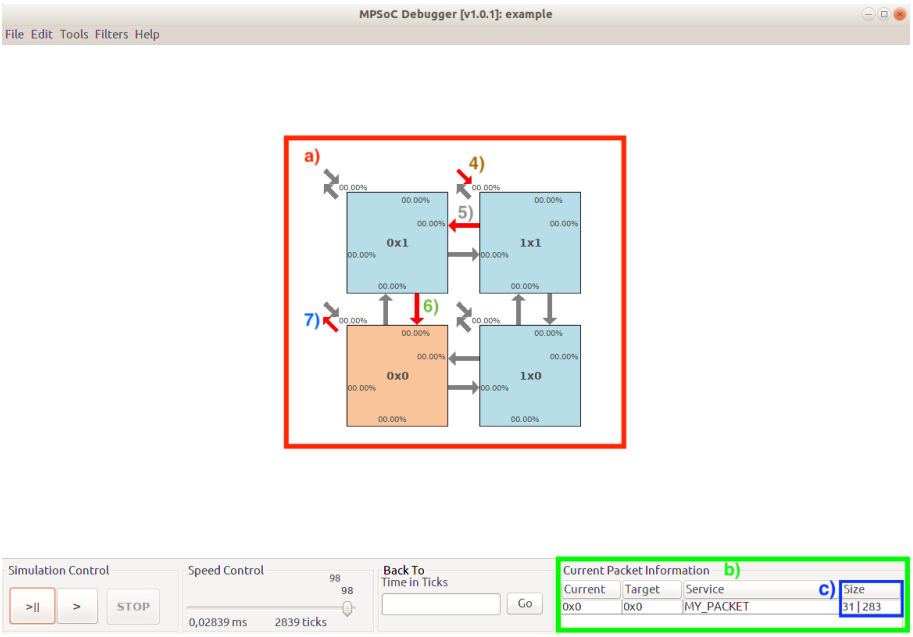


Figura 4.10: Envio do serviço *MY_PACKET* pela NoC do SPE com o módulo PH para o SMPE . Em (a) temos os passos do envio da mensagem listados de 4) a 7) e em (b) o tipo de serviço enviado pela NoC, com sua origem, destino, tipo de serviço e tamanho da mensagem mostrada na parte esquerda de (c) (Fonte: Fonte Própria).

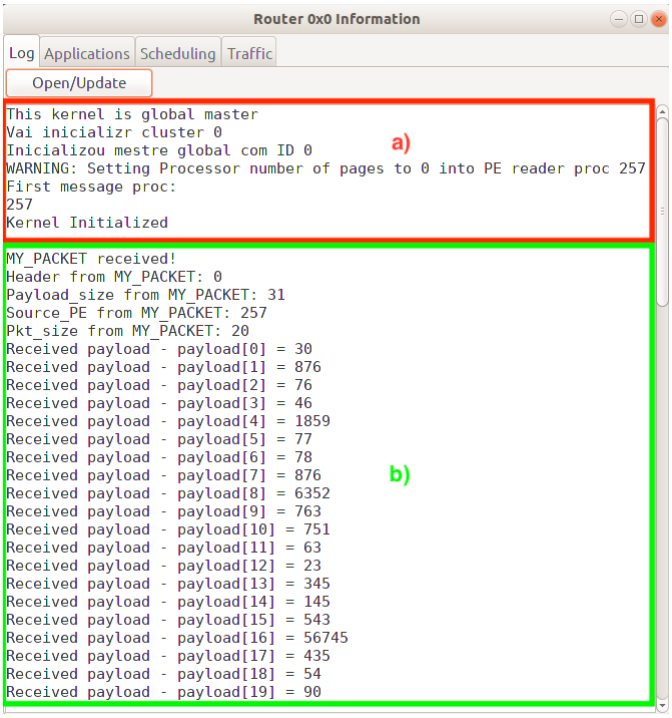


Figura 4.11: Mensagem do serviço *MY_PACKET* recebida pelo SMPE. Em (a) temos informações da inicialização do SPE com o módulo PH . Em (b), temos descrito dados sobre o *header* e *payload* da mensagem (Fonte: Fonte Própria).

Em relação ao mapeamento de tarefas, a implementação original possui uma melhor performance uma vez que, com mais páginas em memória para mapeamento de tarefas, tarefas são alocadas em seus respectivos SPEs mais rapidamente e com menor tendência em ficarem aguardando a serem mapeadas. Porém, tendo-se um tempo de simulação e páginas em memória suficientes ou tarefas sendo mapeadas em diferentes momentos durante a aplicação visando maximizar o uso de páginas em memória por aplicação, a presença do SPEs com o módulo PH não influencia consideravelmente a execução das tarefas dentro do sistema.

Na Figura 4.12 e na Figura 4.13 temos as diferenças no mapeamento das tarefas sem e com o módulo PH incluso na implementação, respectivamente. Percebe-se que o SPE x = 1, y = 1 na Figura 4.13, diferentemente da Figura 4.12, não possui tarefas mapeadas que, por consequência, são mapeadas no SPE x = 0, y = 1.

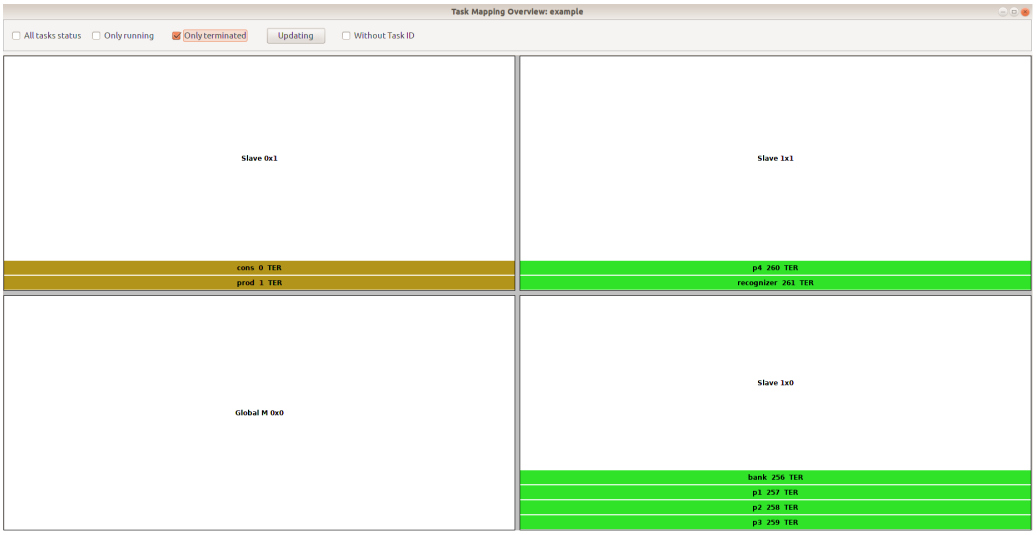


Figura 4.12: Mapeamento das tarefas definidas no primeiro teste para a implementação original da HeMPS (Fonte: Fonte Própria).

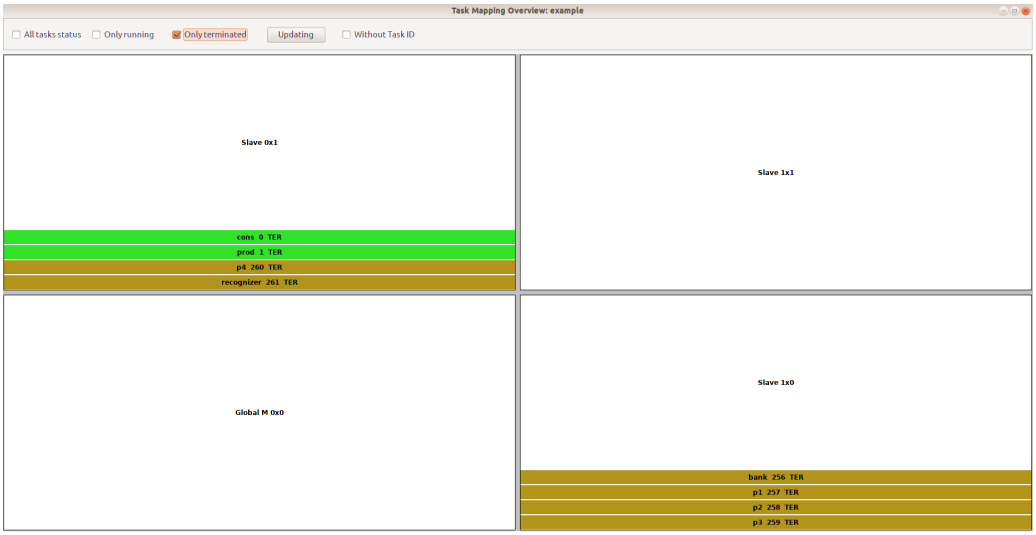


Figura 4.13: Mapeamento das tarefas definidas no primeiro teste para a implementação com o módulo PH. (Fonte: Fonte Própria).

Na Figura 4.14 e na Figura 4.15 temos a visão geral da comunicação entre os PEs para cada implementação. Fazendo uma pequena comparação, não temos mudanças muito significativas no volume de transmissão de flits pelo sistema, respeitando o mapeamento de tarefas em cada implementação. Intuitivamente, temos um maior volume de flits no SMPE da implementação com o módulo PH do que a implementação original visto o envio e recebimento de mensagens do serviço *MY_PACKET* e pouco volume de transmissão de flits durante toda a simulação para o SPE com o módulo PH. Vale a pena ressaltar que o SPE x = 0 e y = 1 da implementação original possui baixo volume de transmissão de flits, pois as tarefas mapeadas a ele referente a aplicação *prod_cons* são de rápida execução e não exigem por um longo período recursos do sistema para serem executadas.

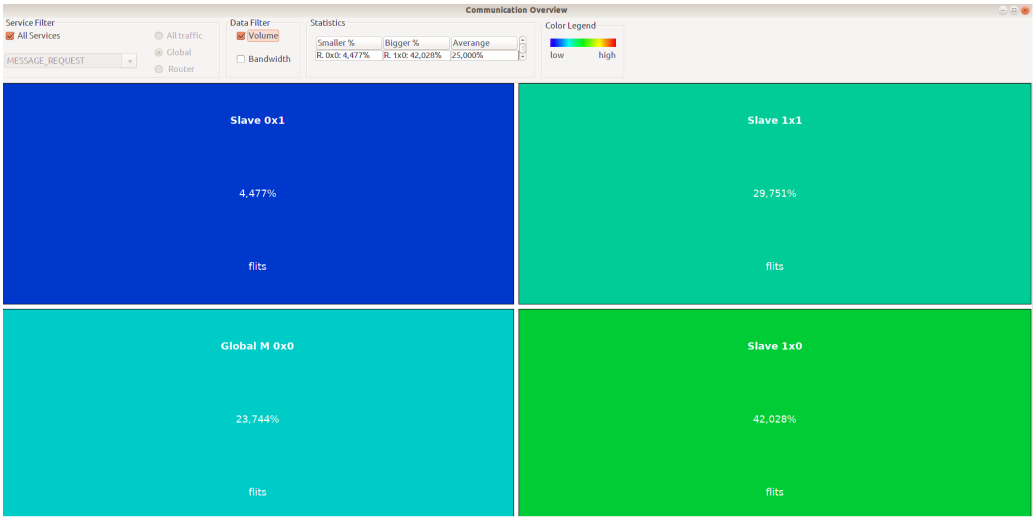


Figura 4.14: Visão geral da comunicação das tarefas por volume total de flits transmitidos no sistema na implementação original da HeMPS para o primeiro caso de teste (Fonte: Fonte Própria).

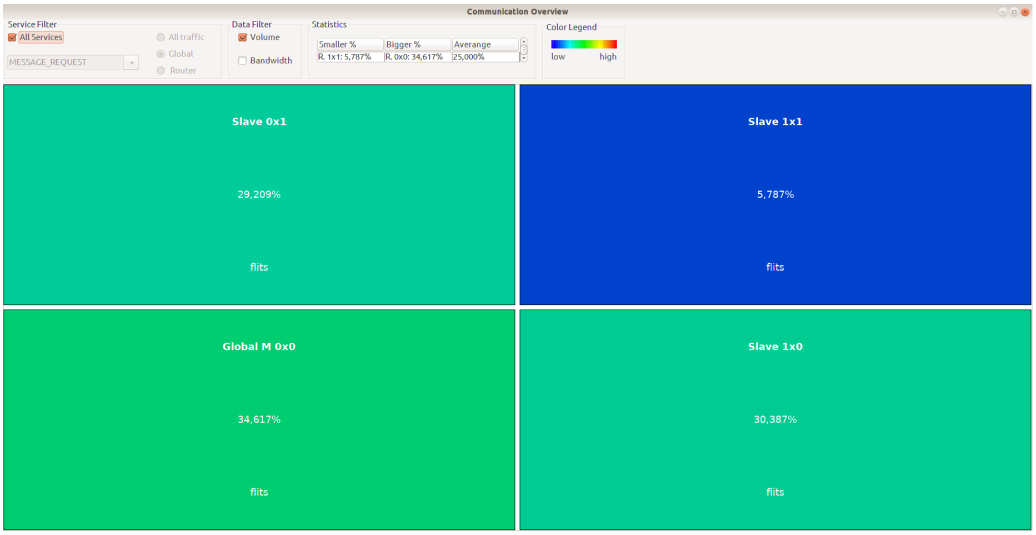


Figura 4.15: Visão geral da comunicação das tarefas por volume total de flits transmitidos no sistema para uma implementação contendo o módulo PH para o primeiro caso de teste (Fonte: Fonte Própria).

4.2.2 Análise do remapeamento de tarefas dentro do sistema com a inclusão do módulo PH

Os passos para a simulação do segundo teste, visto no Código-fonte 4.2, são análogos ao primeiro teste no que se refere a geração dos PEs do sistema e envio e recebimento de

mensagens do serviço *MY_PACKET*. Porém, há uma diferença significativa na gerência de tarefas visto o remapeamento de algumas delas durante a simulação, visto na Figura 4.16.

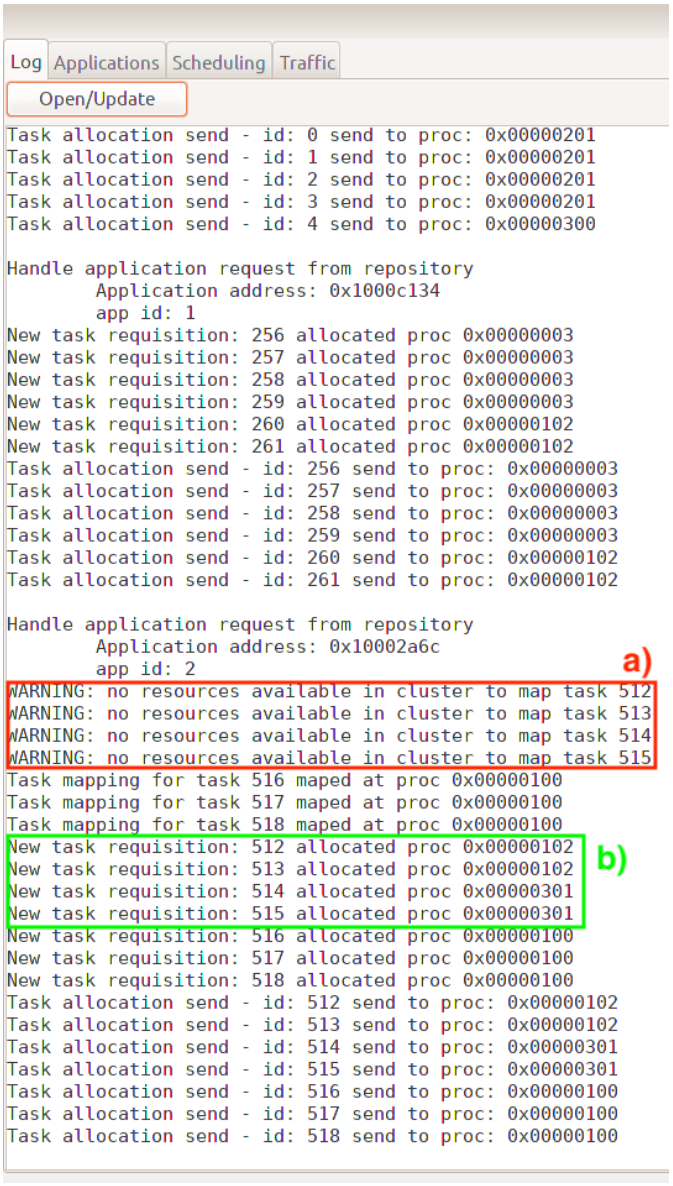


Figura 4.16: Remapeamento de tarefas dentro da aplicação. Em (a) temos as notificações que o *cluster* escolhido para a execução das tarefas não possui mais páginas em memória para receber as tarefas. Em (b) temos o resultado do remapeamento junto com o endereço dentro da NoC dos SPEs que receberam as tarefas (Fonte: Fonte Própria).

Isso se deve ao fato de que *clusters* com SPEs com o módulo PH possuem menos páginas disponíveis em memória se comparado a *clusters* com PEs que tem apenas processadores. Logo, o impacto mais significativo do menor número de recursos é a maior probabilidade de que sejam observadas tarefas em espera.

Para minimizar esse problema, o algoritmo de mapeamento de tarefas procura SPEs que possuam páginas disponíveis em memória, preferencialmente em *clusters* com o maior número total de páginas livres em memória, e que estejam a muito tempo sem executar tarefas.

Também é importante ressaltar a importância de uma configuração adequada para o sistema no que se refere a quantidade de recursos disponibilizados e, principalmente, a dimensão definida para os *clusters*. Após análise e observações com outras simulações, constatou-se que dimensões maiores tendem a mitigar ou até tornar desprezível a falta de páginas em memória nos *clusters* para o mapeamento de tarefas. Dimensões menores podem acarretar em um maior tempo de execução do sistema, devido ao menor número de páginas para mapear tarefas.

4.2.3 Análise do desempenho do sistema com a inclusão do módulo PH para execução e gerenciamento de múltiplas tarefas

Os passos para a simulação do terceiro teste, visto no Código-fonte 4.3, são análogos ao primeiro teste no que se refere a geração dos PEs do sistema e envio e recebimento de mensagens do serviço *MY_PACKET*. Porém, nesse teste pretende-se apenas exemplificar que para sistemas com mais PEs e com mais tarefas disponibilizando recursos suficientes e sendo configurado para suportar todas as tarefas atribuídas, temos o mesmo comportamento visto no primeiro teste para o mapeamento de tarefas. Esse resultado é visto na Figura 4.17, para a primeira metade do sistema, e na Figura 4.18, para a segunda metade do sistema.

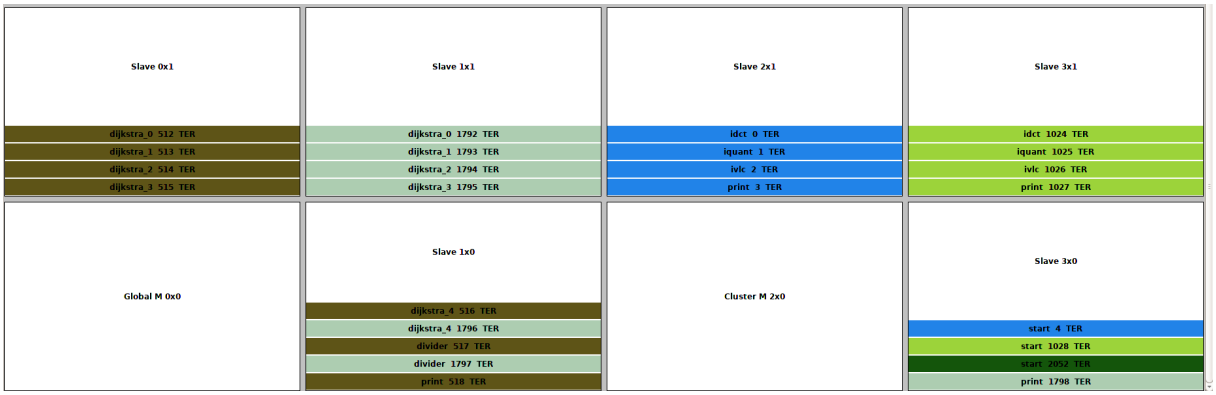


Figura 4.17: Mapeamento das tarefas da parte inferior do sistema do terceiro teste para a implementação com o módulo PH. Nela, vemos que todos os SPEs tem tarefas mapeadas em suas páginas de memória (Fonte: Fonte Própria).

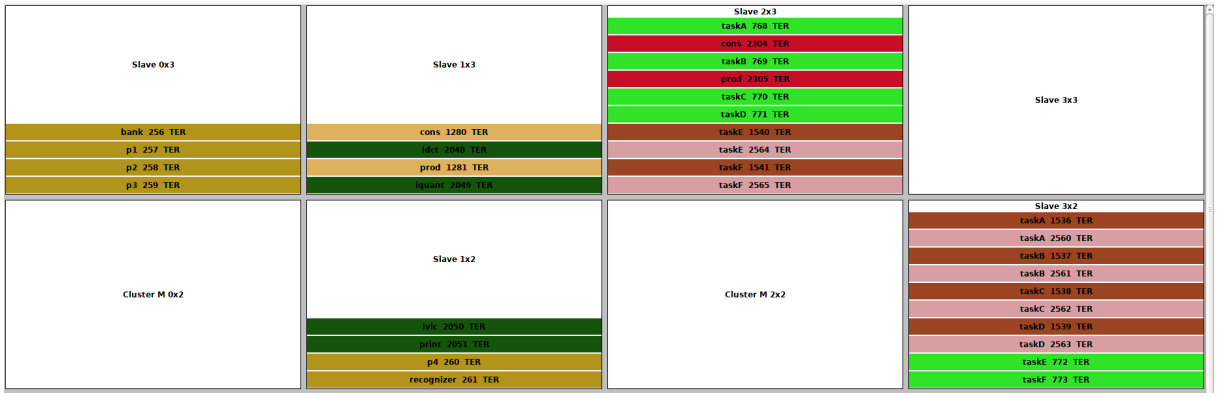


Figura 4.18: Mapeamento das tarefas da parte superior do sistema do terceiro teste para a implementação com o módulo PH. Nela, vemos que, para o SPE x = 3, y = 3, não temos tarefas mapeadas, análogo ao comportamento visto no sistema gerado no primeiro teste (Fonte: Fonte Própria).

4.2.4 Demonstração dos casos de teste para erros de compilação dentro da plataforma

Para garantir que o sistema esteja propriamente configurado com o módulo PH, os *scripts* `kernel_builder` e `yaml_intf` foram modificados para interromper a criação do MPSoC. Os casos de teste criados para validar a geração do sistema são:

- O quarto caso de teste possui a mesma configuração do primeiro caso de teste exceto pela definição da posição dentro do MPSoC atribuída ao SPE com o módulo PH. Como visto na linha 27 do Código-fonte 4.4, a posição $x = 0, y = 0$ é atribuída ao SPE. Porém, devido às definições da plataforma, essa posição sempre está reservada ao SMPE e, portanto, essa é uma configuração inválida para a criação do sistema. A Figura 4.19 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares* devido a um endereço inválido definido para o espaço de endereçamento do módulo PH.

```
Error compiling hardware source code. Invalid position for packet handler.

makefile:21: recipe for target 'hw' failed
make: *** [hw] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.19: Mensagem de erro gerada durante a falha na criação do sistema para o quarto teste (Fonte: Fonte Própria).

- O quinto caso de teste possui a mesma configuração do primeiro caso de teste exceto para definição do tamanho do *payload* da primeira mensagem a ser enviada ao SPE com o módulo PH. Como visto na linha 22 do Código-fonte 4.5, foi definido um tamanho menor do que o número de elementos do *payload* e, portanto, sendo uma configuração inválida para a criação do sistema. A Figura 4.20 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares*, pois o tamanho da primeira mensagem a ser enviada não foi definido corretamente.

```
Error compiling kernel source code. Provide the correct size for the first payload!

makefile:18: recipe for target 'kernel' failed
make: *** [kernel] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.20: Mensagem de erro gerada durante a falha na criação do sistema para o quinto teste (Fonte: Fonte Própria).

- O sexto caso de teste possui a mesma configuração do primeiro caso de teste exceto para a definição do número de SPEs com o módulo PH. Como visto na linha 20 do Código-fonte 4.6, foi definido um número menor de PEs com o módulo PH do que configurado e, portanto, sendo uma configuração inválida para a criação do sistema. A Figura 4.21 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares* e que será necessário corrigi-lá redefinindo o número de SPEs ou redefinindo a quantidade de SPEs com o módulo PH.

```
Error compiling kernel source code. Provide two PEs with packet handler's module
 or check your static_addrs definition!

makefile:18: recipe for target 'kernel' failed
make: *** [kernel] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.21: Mensagem de erro gerada durante a falha na criação do sistema para o sexto teste (Fonte: Fonte Própria).

- O sétimo caso de teste possui a mesma configuração do primeiro caso de teste exceto pela definição da posição dentro do MPSoC atribuída ao SPE com o módulo PH que ultrapassa as dimensões definidas para o MPSoC. Como visto na linha 27 do Código-fonte 4.7, o endereço x = 3, y = 3 ultrapassa o tamanho da MPSoC cujo último elemento se encontra na posição x = 1 e y = 1 e, portanto, sendo uma configuração inválida para a criação do sistema. A Figura 4.22 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares* e que será necessário corrigi-lá redefinindo o SPEs cuja posição foi definida tendo o endereço x = 3, y = 3.

```
Error compiling kernel source code. Provide a valid position for PE 3 x 3!

makefile:18: recipe for target 'kernel' failed
make: *** [kernel] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.22: Mensagem de erro gerada durante a falha na criação do sistema para o sétimo teste (Fonte: Fonte Própria).

- O oitavo caso de teste possui a mesma configuração do primeiro caso de teste exceto para o tamanho do *payload* da segunda mensagem a ser enviada ao SPE com o módulo PH. Como visto na linha 23 do Código-fonte 4.8, foi definido um tamanho maior do que o número de elementos do *payload* da mensagem e, portanto, sendo uma configuração inválida para a criação do sistema. A Figura 4.23 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares*, pois o tamanho da segunda mensagem a ser enviada não foi definido corretamente.

```
Error compiling kernel source code. Provide the payload for the second message!

makefile:18: recipe for target 'kernel' failed
make: *** [kernel] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.23: Mensagem de erro gerada durante a falha na criação do sistema para o oitavo teste (Fonte: Fonte Própria).

- Por fim, o nono caso de teste possui a mesma configuração do primeiro caso de teste exceto pela definição do endereço de leitura e escrita na memória local do SPE com o módulo PH. Como visto na linha 21 do Código-fonte 4.9, foi definido um endereço que, se deixado, poderá corromper a memória visto que cada posição da memória local deve conseguir armazenar um flit e, portanto, com intuito de otimizar o número de posições em memória por página, os endereços devem ser múltiplos de 4 ou começarem a partir da posição 0 da memória. Assim sendo, o endereço que foi definido no teste é uma configuração inválida para a criação do sistema. A Figura 4.24 mostra a mensagem de erro mostrada em linha de comando durante a falha da compilação do sistema. Nela o usuário é informado na primeira linha da mensagem que ocorreu uma falha ao compilar o código fonte dos *hardwares*, pois o endereço para a escrita e leitura na memória local não foi definido corretamente.

```
Error compiling hardware source code. Invalid address for packet handler's memory.

makefile:21: recipe for target 'hw' failed
make: *** [hw] Error 1
make: Leaving directory '/media/sf_Ubuntu_Folder/hemps/hemps9/testcases/example'

*** Error: hemps-run stopped !!!
```

Figura 4.24: Mensagem de erro gerada durante a falha na criação do sistema para o nono teste (Fonte: Fonte Própria).

4.3 Comparação das diferentes implementações

Nessa seção é apresentada uma breve comparação entre os tempos de execução observados para a implementação original e a implementação com o módulo PH. O tempo de execução se refere ao tempo total para se executar todas as tarefas atribuídas ao sistema pelo usuário.

Foram simulados os três primeiros casos de teste em um sistema com um *clock* de 100 MHz. A Tabela 4.1 apresenta um comparativo entre tempos médios de execução dos casos de teste entre as aplicações executadas na HeMPS em sua implementação original e na implementação com o módulo PH no caso em que o algoritmo de gerenciamento de tarefa *Least Slack Time* (LST) é utilizado. Já a Tabela 4.2 apresenta o mesmo comparativo para o caso onde o algoritmo de gerenciamento de tarefas aplicado é o *Round Robin* (RR). Em ambos os casos, foram realizadas um total de 45 simulações, 15 para cada um dos casos de testes. A coluna denominada *Aumento*, presente nas duas tabelas citadas, denota a diferença percentual dos tempos médios observados na comparação entre a implementação

com o módulo PH e a implementação original. Os tempos de execução em ambas tabelas é dada em milissegundos.

Casos de teste	**Número de tarefas**	**Tempo de execução com LST**		
		Original	**PH**	**Aumento**
1	8	28.6448ms	28.6475ms	0.0094%
2	24	23.984ms	24.2141ms	0.96%
3	63	24.0095ms	24.1536ms	0.60%

Tabela 4.1: Tabela comparativa dos tempos médios de execução implementação original × implementação com o módulo PH utilizando os algoritmo de gerenciamento de tarefas *Least Slack Time* (LST). (Fonte: Fonte Própria)

Casos de teste	**Número de tarefas**	**Tempo de execução com RR**		
		Original	**PH**	**Aumento**
1	8	28.6447ms	28.6477ms	0.0105%
2	24	23.9837ms	24.2139ms	0.96%
3	63	24.0093ms	24.1533ms	0.60%

Tabela 4.2: Tabela comparativa dos tempos médios de execução implementação original × implementação com o módulo PH utilizando os algoritmo de gerenciamento de tarefas *Round Robin* (RR). (Fonte: Fonte Própria)

Analisando os valores apresentados nas tabelas, vemos que, ao menos para os casos testados, a escolha do algoritmo de gerenciamento de tarefas não interfere significativamente nos valores de simulação e os valores obtidos refletem corretamente o comportamento do sistema com o acréscimo do módulo PH. Como esperado, vemos que a implementação com o novo módulo aumenta o tempo final de execução, uma vez que, além do envio e tratamento de mensagens, os *clusters* possuem menos páginas em memória. Logo, isso leva a um mapeamento menos eficiente das tarefas dentro do sistema, um aumento no tempo total de execução e menor paralelismo.

Apesar desse aumento, vemos que mesmo para o pior caso descrito no teste 2 onde temos remapeamento de tarefas, o aumento no tempo total de simulação fica em, aproximadamente, $0,96\%$. Isso nos leva a crer que o sistema ainda continua com uma boa performance para a execução e gerenciamento de tarefas, mesmo levando mais ciclos de processamento para execução de todos os casos de teste.

Assim sendo, através das validações e comparações feitas nesse capítulo, podemos concluir que mesmo com uma maior taxa de gerenciamento e uso de recursos no sistema, a implementação com o novo bloco IP se mostra bem eficiente e consegue manter a

confiabilidade da HeMPS para o uso com outros módulos de *hardware* e, como será descrito no próximo capítulo, a possível integração de outros sistemas na plataforma.

Capítulo 5

Conclusões e considerações finais

O objetivo deste capítulo é apresentar as conclusões geradas da metodologia mostrada no capítulo 3 e resultados mostrados no capítulo 4.

Na seção 5.1, é feito uma breve comparação dos objetivos apresentados no estudo e conclusões encontradas.

Na seção 5.2, são abordados tópicos pertinentes para o desenvolvimento e análise em trabalhos futuros.

5.1 Conclusões do estudo e considerações finais

O objetivo principal desse estudo foi a adaptação da plataforma HeMPS com intuito de torná-la heterogênea. Para isso, foi criado um módulo de *hardware* que não só serviu para provar o estudo de caso dessa tese, mas gerou uma interface de comunicação para diferentes módulos de *hardware* e sistemas que, em trabalhos futuros, possam ser acoplados a plataforma.

A metodologia consistiu em duas etapas. Na primeira etapa foi apresentado o módulo desenvolvido, explicitando seu funcionamento e comportamento no sistema. Na segunda etapa foram mostradas todas as etapas de criação da plataforma por meio de *software* e o que teve que ser mudado para que fosse possível a integração do módulo denominado *Packet Handler* (PH) nos PEs, levando em consideração a interação e funcionamento com os módulos DMNI e a *RAM*.

A integração do novo módulo acarretou em uma implementação relativamente simples visto as mudanças feitas no *hardware* e *software* da HeMPS e a abordagem intuitiva para o seu desenvolvimento e elaboração. Porém, vale a pena ressaltar a perda de performance no mapeamento e execução de tarefas no sistema visto que se retirou o processador dos PEs para a integração com o módulo PH. Com isso, foi necessário conhecer toda a estrutura e implementação de *hardware* e *software* para se tentar manter a plataforma com o melhor

desempenho e confiabilidade possíveis. Além disso, exigiu-se uma re-estruturação e um conhecimento amplo da lógica de controle de todos os módulos do sistema para tornar todo esse processo viável, porém se mantendo o funcionamento original dos outros módulos de *hardware*.

Além disso, foi necessário modificar o *software* presente na plataforma para, entre outros motivos, evitar os SPEs com módulo PH estejam na lista de PEs disponíveis para alocação de tarefas, melhorar o controle de fluxo de flits e mapeamento de tarefas.

Por fim, todos os passos para a validação do comportamento e geração da plataforma aconteceram conforme sua implementação. De resultados, o capítulo 4 exemplifica os passos feitos para a validação do sistema, tanto seu funcionamento como criação. Muito mais do que isso, a nova versão implementada mostra-se completamente funcional, com um desempenho vistos em testes comparáveis a implementação original da plataforma e tendo contribuído para trabalhos futuros com a implementação de um módulo de *hardware* que serve como interface entre a HeMPS e outros módulos de *hardware* e sistemas.

5.2 Trabalhos futuros com a nova implementação

Para futuros projetos utilizando-se a implementação criada nesse estudo, alguns pontos interessantes de melhorias e casos de uso seriam:

- melhorar o gerenciamento de leitura e escrita na memória local pelo módulo PH com intuito de otimizar seu espaço;
- desenvolvimento de testes com uma maior cobertura de casos de uso e testes para a uma maior validação do sistema;
- adaptar a nova implementação para o uso na linguagem *VHDL*;
- adaptar a nova implementação para geração do sistema em Arranjo de Portas Programáveis em Campo (*Field Programmable Gate Array*) (FPGA);
- avaliar o desempenho da nova implementação com sistemas distribuídos;
- acoplar um módulo de *hardware* ao módulo PH. O módulo PH servirá de interface de envio e recebimento de pacotes para novos blocos IP ou sistemas;
- integração do módulo PH com sistemas complexos para avaliar seu desempenho como módulo de interface, como, por exemplo, redes neurais.

Referências

[1] Marcelo Ruaro, Felipe B. Lazzarotto, César A. Marcon Fernando G. Moraes: *Dmni: A specialized network interface for noc-based mpsocs*. PUCRS University, Computer Science Department, Porto Alegre, Brazil, 2016. https://ieeexplore.ieee.org/abstract/document/7527462/. ix, 19

[2] Bezerra Marinho, André: *Hemps-v: Um mpsoc com processador de arquitetura risc-v*. Tese de Graduação em Engenharia da Computação, Universidade de Brasília, 2018. xi, 79, 84, 85, 86, 100, 101, 102, 112

[3] Zeferino, Cesar Albenes: *Redes-em-chip : arquiteturas e modelos para avaliação de área e desempenho*. Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Instituto de Informática, 2003. http://hdl.handle.net/10183/4179. 1, 7, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 27, 30, 31, 32, 39

[4] Grandi Mandelli, Marcelo: *Exploration of runtime distributed mapping techniques for emerging large scale mpsocs*. Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2015. http://tede2.pucrs.br/tede2/handle/tede/6317. 1

[5] *Porto alegre, grupo de apoio ao projeto de hardware (gaph), pontifícia universidade católica do rio grande do sul*. https://corfu.pucrs.br/tikiwiki/tiki-index.php. Acessado em 28/11/2018. 2, 33

[6] Ruaro, Marcelo: *Self-adaptive qos at communication and computation levels for many-core system-on-chip*. Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica do Rio Grande do Sul, 2018. 2, 8, 9, 17, 18, 33, 34, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 60, 63, 68, 80, 82, 84, 86, 89, 90, 92, 93, 94, 95, 96, 97, 98, 102, 103, 105, 106, 107, 108, 109, 110, 111

[7] G. Moraes, Fernando, Adelcio Biazi e Eduardo Weber Wächter: *Hemps-s: A homogeneous noc-based mpsocs framework prototyped in fpgas*. 6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), Montpellier, páginas 1–8, 2011. 2, 33

[8] Hassan, Mohamed: *Heterogeneous mpsocs for mixed criticality systems: Challenges and opportunities*. Cornell University, Computer Science Department, New York, United States, 2017. https://arxiv.org/pdf/1706.07429.pdf. 2

[9] Wächter, Eduardo Weber: *Integração de novos processadores em arquiteturas mpsoc: um estudo de caso*. Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2011. `http://tede2.pucrs.br/tede2/handle/tede/5138`. 3, 35, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62

[10] JUNIOR, NELSON ANTONIO GONÇALVES: *Análise e simulação de topologias de redes em chip*. Master thesis, Universidade Estadual de Maringá, Nupélia, Maringá, PR, Brasil, 2010. 6, 15, 29

[11] Sepúlveda Flórez, Martha Johanna: *Projeto de estruturas de comunicação intrachip baseadas em noc que implementam serviços de qos e segurança*. Tese de Doutorado em Microeletrônica, Escola Politécnica (EP). Universidade de São Paulo (USP). São Paulo, SP, Brasil, 2011. 7, 18, 19, 20, 21, 22, 23, 24, 26, 29, 82

[12] *The evolution of mobile computing has arrived: Qualcomm snapdragon 850 mobile computer platform*. `https://tiny.cc/qualcomm`, Acessado em 09/06/2018. 7, 8

[13] Berkel, C H. van: *Multi-core for mobile phones*. Em *Proceedings of DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, páginas 1260–1265, 2009. 9

[14] Cheng HY, Zhan J, Zhao J Xie Y Sampson J Irwin MJ: *Core vs. uncore: the heart of darkness*. 52nd ACM/EDAC/IEEE design automation conference (DAC), 1(1):1–6, 2015. 9

[15] Borkar, Shekhar; Chien, Andrew A.: *The future of microprocessors.* Commun ACM, 54(5):67–77, 2011. `http://dx.doi.org/10.1145/1941487.1941507`. 9, 10

[16] Dennard RH, Gaensslen FH, Rideout VL Bassous E LeBlanc AR: *Design of ionimplanted mosfet's with very small physical dimensions*. IEEE J Solid-State Circuits, 9(5):256–268, 1974. 10

[17] Das R, Ausavarungnirun R, Mutlu O Kumar A Azimi M: *Application-to-core mapping policies to reduce memory system interference in multi-core systems*. IEEE 19th international symposium on high performance computer architecture (HPCA2013), 1(1):107–118, 2013. 10

[18] R, Das: *Application-aware on-chip networks*. Ph.d. thesis, The Pennsylvania State University, Old Main, State College, PA 16801, EUA, 2010. 11

[19] JANTSCH, Y.R SUM; S. KUMAR; A.: *Simulation and evaluation for a network on chip*. Proceedings of 20th NORCHIP Conference, 1(1):7–12, 2002. 11, 14, 15

[20] GUERRIER, P.; GREINER, A.: *A generic architecture for on-chip packet-switched interconnections*. DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXIBITION, 1(1):250–256, 2000. 13

[21] Salminen, Erno, Ari Kulmala e Timo D. Hämäläinen: *On network-on-chip comparison*. 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007), 2007. 17

[22] Alazemi, F., A. AziziMazreah, B. Bose e L. Chen: *Routerless network-on-chip*. Em *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, páginas 492–503, Feb 2018. 17

[23] Rantala, Ville, Teijo Lehtonen e Juha Plosila: *Network on chip routing algorithms*. University of Turku, Department of Information Technology Joukahaisenkatu 3-5 B, 20520 Turku, Finland, 2006. 17

[24] Axel Jantsch, Hannu Tenhunen: *Networks on Chip*, volume 1. Springer, 2003. 18

[25] Mandelli, Marcelo Grandi: *Notas de aula sobre redes em chip*. UNB University, Computer Science Department, Brasilia, Brazil `https://drive.google.com/file/d/1DcNR2FrvdfWXwf20YzJIDgcqyfKXr8x0/view?usp=sharing`. Acessado em 21/07/2019. 23, 26, 27, 28, 30

[26] Cota, Érika, Alexandre de Morais Amory e Marcelo Soares Lubaszewski: *Reliability, Availability and Serviceability of Networks-on-Chip*. Springer US, 1ª edição, 2012, ISBN 978-1-4614-0791-1. 23, 28, 29

[27] *Hemps multiprocessor system on chip*. `http://www.inf.pucrs.br/hemps/index.html`. Acessado em 28/11/2018. 32

[28] Carara, Everton Alceu: *Serviços de comunicação diferenciados em sistemas multiprocessados em chip baseados em redes intra-chip*. Tese de Doutorado em Ciência da Computação, Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2011. `http://tede2.pucrs.br/tede2/handle/tede/5142`. 33, 34, 36, 37

[29] Castilhos, Guilherme Machado de: *Gerenciamento térmico e energético em mpsocs*. Tese de Doutorado em Ciência da Computação, Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2017. `https://www.inf.pucrs.br/moraes/docs/teses/tese_castilhos.pdf`. 34, 35, 36, 60, 61

[30] *Plasma - most mips i(tm) opcodes*. `https://opencores.org/project,plasma`. Acessado em 28/11/2018. 35, 36

[31] Mello, Aline Vieira de: *Qualidade de serviço em redes intra-chip implementação e avaliação sobre a rede hermes*. Tese de Mestrado em Ciência da Computação, Porto Alegre, Programa de Mestrado em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2006. `http://tede2.pucrs.br/tede2/handle/tede/5290`. 36, 37, 38, 39, 40, 41

[32] Carara, Everton Alceu: *Uma exploração arquitetural de redes intra-chip com topologia malha e modo de chaveamento wormhole*. Tese de Graduação em Ciência da Computação, Porto Alegre, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2004. `https://www.inf.pucrs.br/moraes/docs/tcc/tc_carara.pdf`. 37, 38

[33] Carlos A. Petry, Eduardo W. Wächter, Guilherme M. de Castilhos Fernando G. Moraes Ney L. V. Calazans: *A spectrum of mpsoc models for design space exploration and its use*. 2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP), 2012. `https://ieeexplore.ieee.org/document/6380687/`. 64

[34] Ruaro, Marcelo, Guilherme Madalozzo, Alzemiro Silva, Anderson Sant'Ana e Fernando Gehm Moraes: *Hemps 8.x tutorial*, agosto 2017. Porto Alegre, Grupo de Apoio ao Projeto de Hardware, Pontifícia Universidade Católica do Rio Grande do Sul. 81, 82