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

HeMPS-V: Um MPSoC com processador de arquitetura RISC-V

André Bezerra Marinho

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

Orientador
Prof. Dr. Marcelo Grandi Mandelli

Brasília
2018

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

HeMPS-V: Um MPSoC com processador de arquitetura RISC-V

André Bezerra Marinho

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

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

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Marcus Vinicius Lamar
CIC/UnB

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

Brasília, 08 de agosto de 2018

Resumo

A partir da alta capacidade de integração de transistores em um único chip, o desenvolvimento de complexos sistemas em pastilhas de silício é cada vez mais comum. Por isso, os sistemas integrados intra-chip (SoCs) têm ganhado cada vez mais espaço no mercado. Basicamente, esses sistemas se caracterizam pela integração de módulos de propriedade intelectual (IP). Por meio desses módulos, pode-se integrar diversos elementos de processamento em um único SoC, o que caracteriza os sistemas multiprocessados intra-chip (MPSoCs). Uma forma de fazer a integração desses módulos é utilizando comunicações via redes intra-chip (NoCs). Dessa forma, o presente projeto aborda a modificação de um framework capaz de gerar MPSoCs baseados em NoCs homogêneas parametrizáveis, HeMPS (Hermes Multiprocessor System-on-Chip). Essa modificação consistirá na substituição do atual núcleo do elemento de processamento dos nós da rede. Atualmente, são integrados a esses núcleos, o processador Plasma que possui arquitetura MIPS e executa o conjunto de instruções de inteiros de 32 bits. A ideia é integrar no sistema uma implementação de um novo processador em diferente arquitetura que seja funcional nesses MPSoCs. A utilização de uma arquitetura RISC-V é viável e possui um conjunto de instruções correspondente ao atual. Mais ainda, além de ter código aberto, caracteriza um conjunto de instruções mais recente que o MIPS. Portanto, esse trabalho serve como estudo de caso e apresenta os problemas e requisitos para o interfaceamento de uma nova arquitetura no sistema de referência, HeMPS. O funcionamento do sistema com a nova arquitetura é detalhado apresentando validações de mecanismos que dependem diretamente desta, como interrupções, chamadas de sistema, salvamento e recuperação de contexto e paginação.

Palavras-chave: Sistemas integrados intra-chip, Redes intra-chip, MPSoCs homogêneos, RISC-V, Hardware Description Language

Abstract

Due to the high density of transistors on a single chip, developing complex systems in silicon wafers is increasingly common. Hence Systems-on-Chip (SoCs) are gaining more space in market. Basically, these systems are characterised by integrating intelectual property (IP) modules. Through these modules, it's possible to integrate several processing elements in a single SoC, which characterises Multiprocessor Systems-on-Chip (MPSoCs). A way of implementing an interconnection between those modules is using as communication Networks-on-Chip (NoCs). This way, the present project discusses the modification of a framework which generates homogeneous parameterizable MPSoCs based on NoCs, HeMPS (Hermes Multiprocessor System-on-Chip). This modification consists of replacing the actual core from the processing elements which are in network nodes. Currently, is integrated in those cores the processor Plasma, which has MIPS architecture and executes a 32 bits integer instruction set. The idea is to integrate on this system an implementation of a new processor with different architecture that will be functional on those MPSoCs. Using RISC-V architecture is viable and has an instruction set that matches the actual. Furthermore, besides having an open-source code, it characterises an instruction set more recent than MIPS. Therefore, this work serves as a case study and shows the problems and requirements for interfacing a new architecture on the reference system, HeMPS. The operation of the modified target system is detailed showing validations of mecanisms that directly relies on new architecture, such as interruptions, system calls, saving and recovering processor context and pagination.

Keywords: Systems-on-chip, Networks-on-chip, Homogeneous MPSoCs, RISC-V, Hardware Description Language

Sumário

1 Introdução **1**
1.1 Objetivos . . . 3
1.1.1 Objetivos gerais . . . 3
1.1.2 Objetivos específicos . . . 3
1.2 Justificativa . . . 4
1.3 Metodologia da pesquisa . . . 4
1.4 Estrutura do texto . . . 5

2 Fundamentação teórica **6**
2.1 Sistemas integrados intra-chip (SoCs) . . . 6
2.1.1 Metodologia de projeto de SoCs . . . 7
2.2 Sistemas intra-chip multiprocessados (MPSoCs) . . . 9
2.2.1 Redes intra-chip (NoCs) . . . 11
2.3 MPSoC baseado em NoC homogênea parametrizável - HeMPS . . . 21
2.3.1 Processador Plasma . . . 22
2.3.2 Rede intra-chip Hermes . . . 24
2.3.3 Interface de rede e acesso direto à memória (DMNI) . . . 25
2.3.4 Modelagem de aplicações . . . 29
2.3.5 Kernel (Sistema operacional distribuído) . . . 30
2.4 Arquitetura RISC-V . . . 36
2.4.1 Outras arquiteturas . . . 37
2.4.2 Processadores RISC-V . . . 38

3 Integração de um novo processador na plataforma HeMPS **40**
3.1 Interface do processador de arquitetura RISC-V no Elemento de Processamento 40
3.1.1 Interface do processador com a memória . . . 44
3.1.2 Mecanismo de acesso ao repositório de tarefas . . . 49

3.2 Adaptação da geração na plataforma HeMPS para arquitetura RISC-V . . . 53
3.2.1 Execução do script hemps-run e carregamento do arquivo de configuração YAML . . . 54
3.2.2 Execução do script testcase_builder . . . 56
3.2.3 Compilação dos sistemas operacionais e das aplicações . . . 56
3.2.4 Execução de *scripts* para realizar a simulação na ferramenta Modelsim 59
3.3 Identificação e adaptação de rotinas do *kernel* dependentes da arquitetura . . 60
3.3.1 Implementação de interrupções de máquina no processador . . . 61
3.3.2 Salvamento e recuperação de contexto . . . 65
3.3.3 Rotinas relacionadas ao tratamento de pacotes (DMNI) . . . 73
3.4 Alterações na implementação do processador Clarvi . . . 76
3.4.1 Extensão do conjunto de instruções do processador para RV32IM . . . 77
3.4.2 Chamadas de sistema . . . 81
3.4.3 Controle do retorno de interrupções . . . 85

4 Resultados **89**
4.1 Casos de testes utilizados . . . 89
4.2 Validação do funcionamento . . . 95
4.2.1 Validação da paginação . . . 96
4.2.2 Validação das interrupções . . . 97
4.2.3 Validação das chamadas de sistema . . . 98
4.2.4 Validação geral do sistema . . . 102
4.3 Comparativo entre os processadores . . . 114

5 Conclusão **116**
5.1 Conclusões . . . 116
5.2 Trabalhos futuros . . . 117

Referências **119**

Lista de Figuras

2.1 Diagrama de blocos do SoC Snapdragon 845 mobile. 7
2.2 Arquitetura genérica de um SoC. 8
2.3 Diagrama da arquitetura ponto-a-ponto. 10
2.4 Diagrama da arquitetura multiponto. 10
2.5 Diagrama da arquitetura multiponto hierárquica com múltiplos barramentos, sendo um rápido de grande largura de banda e outro lento de pequena largura. 11
2.6 Diagrama de blocos de uma rede intra-chip simples formada por enlaces e roteadores. 12
2.7 Divisão de uma mensagem a ser transmitida. 13
2.8 Estrutura física de um enlace SPIN composto por 2 canais unidirecionais opostos e seus sinais auxiliares. 13
2.9 Topologias de NoCs diretas em que a) representa a malha. Em b), tem-se a toróide e em c), o hipercubo 14
2.10 Topologias de NoCs indiretas em que a) apresenta o tipo *crossbar* e b) o multiestágio. 15
2.11 Unidade de roteamento de uma NoC com 2 portas de entrada e 2 portas de saída. 15
2.12 *Buffers* FIFO. 19
2.13 *Buffers* SAFC. 19
2.14 *Buffers* SAMQ. 20
2.15 *Buffers* DAMQ. 20
2.16 Arquitetura atual do MPSoC gerado pelo HeMPS (a). Nesse caso, tem-se um sistema 8×8 gerado. Em (b) é visto o elemento de processamento detalhado. 21
2.17 Diagrama de blocos do processador Plasma. 23
2.18 Arquitetura atual da rede Hermes em um sistema 3×3 em (a). Em (b) é vista a arquitetura detalhada do roteador. 24

2.19 MPSoc 6×6 (a) com duas propostas de implementação do PE, com DMA e NI separados (b) e integrados em uma DMNI (c). 25
2.20 Arquitetura da DMNI proposta. 26
2.21 Máquina de estados finitos do módulo de envio. 26
2.22 Máquinas de estados finitos do módulo de recebimento. 27
2.23 Máquina de estados finitos do módulo de recebimento. 28
2.24 Exemplo do modelo de aplicações a serem executadas na HeMPS. 29
2.25 Hierarquia do *kernel* da HeMPS. 30
2.26 Diagrama do protocolo de admissão de novas aplicações. 31
2.27 Protocolo de comunicação entre tarefas utilizado na HeMPS. 34
2.28 Organização da paginação de memória. 36
2.29 Comparativo entre processadores de instruções reduzidas. 37
2.30 Diagrama de blocos do processador Potato e sua interface *wishbone*. 38

3.1 Estrutura atual do elemento de processamento presente nos MPSoCS gerados pela HeMPS. 41
3.2 Multiplexação de barramentos de memória conforme o comportamento da DMNI. 45
3.3 Extração de página corrente a partir do sinal `page`. 47
3.4 Comparação de recebimento de requisições nas duas versões. Em A, tem-se um caso de sucesso. Na região B, tem-se um caso de falha, pois o valor de requisição acabou em 0 quando não devia. 52
3.5 Passo-a-passo de geração da plataforma HeMPS. 53
3.6 *Banner* mostrado no terminal no início da geração do MPSoC. 57
3.7 Processo de geração do código objeto das aplicações a serem executadas no MPSoC. 57
3.8 Processo de geração do código objeto dos sistemas operacionais executados pelos PEs do MPSoC. 58
3.9 Comparação das ondas que são mostradas pela versão original da HeMPS na esquerda e pela versão modificada na direita. 60
3.10 Fluxo de execução de uma interrupção no sistema. 61
3.11 Registrador de interrupção `mip`. 64
3.12 Registrador de interrupção `mie`. 65
3.13 Registrador de estado da máquina `mstatus`. 65
3.14 Registradores presentes na arquitetura RISC-V (esquerda) e MIPS (direita). 69
3.15 Formato das instruções de multiplicação a) e divisão b) do subconjunto M. 78
3.16 Fluxo de execução de uma chamada de sistema. 81

3.17 Caso em que há interrupção durante uma instrução “lw” seguida de uma adição que depende do que foi carregado. 86

4.1 Grafo da aplicação “hello_world_app”, do tipo produtor/consumidor. . . . 89
4.2 Grafo da aplicação “mpeg”. 90
4.3 Grafo da aplicação “dtw”. 90
4.4 Grafo da aplicação “djikstra”. 91
4.5 Grafo da aplicação “synthetic”. 91
4.6 Verificação de paginação na forma de onda do caso de teste 1. A região A mostra o endereço de instruções paginado enquanto na B, está o endereço de dados. 97
4.7 Ocorrência de interrupção durante execução da tarefa de página 2 do caso de teste 2. 99
4.8 Restauração de contexto após o tratamento da interrupção na tarefa de página 2 do caso de teste 2. 100
4.9 Ocorrência de chamada de sistema durante execução da tarefa de página 2 do caso de teste 2. 102
4.10 Salvamento de parte remanescente do contexto no momento que o escalonador escolhe outra tarefa para a execução. 103
4.11 Escolha por executar a tarefa de página 3 ao fim do tratamento da chamada de sistema. 103
4.12 Restauração de contexto antes de voltar a executar a tarefa que havia sido interrompida ao realizar uma chamada de sistema. 104
4.13 Análise de formas de onda quando ocorre envios do repositório de aplicações para o elemento de processamento destino no caso de teste 2. 108
4.14 Análise de formas de onda da execução do caso de teste 2. 109
4.15 Comparação das instruções buscadas nas formas de onda da execução da tarefa “start” do caso de teste 2, com o seu arquivo “lst”. 109
4.16 Comparação das instruções buscadas nas formas de onda da execução da tarefa “start” do caso de teste 2, com as informações de depuração impressas pelo processador na seção “Transcript” do Modelsim. 110
4.17 Execução da simulação do caso de teste 3 no *software* de depuração da HeMPS. 112
4.18 Verificação do mapeamento de tarefas do caso de teste 3 no *software* de depuração da HeMPS. 112
4.19 Visão geral da comunicação durante execução do caso de teste 3 no *software* de depuração da HeMPS. 113

Lista de Tabelas

2.1 Significado dos sinais auxiliares do barramento da NoC SPIN. 13

3.1 Tabela de instruções do subconjunto M. 79

4.1 Tabela comparativa entre as aplicações executadas pela HeMPS na versão original e na versão com processador Clarvi 114

Lista de Abreviaturas e Siglas

ABI Interface Binária do Aplicativo (do inglês, *Application Binary Interface*).

AGP Porta Gráfica Acelerada (do inglês, *Accelerated Graphics Port*).

ASIP Processador de Conjunto de Instruções Específicas (do inglês, *Application Specific Instruction set Processor*).

BE Melhor Esforço (do inglês, *Best Effort*).

CBDA Buferizado Centralmente, Alocado Dinamicamente (do inglês, *Centrally-Buffered, Dynamically-Allocated*).

CPU Unidade Central de Processamento (do inglês, *Central Processing Unit*).

CSR Registrador de Controle e Estado (do inglês, *Control and Status Register*).

DAMQ Dinamicamente Alocado, Multi-Fila (do inglês, *Dynamically-Allocated, Multi-Queue*).

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

DMNI Interface de Rede com Memória Direta (do inglês, *Direct Memory Network Interface*).

FCFS Primeiro a Chegar Primeiro Servido (do inglês, *First Come First Served*).

FIFO Primeiro a Entrar Primeiro a Sair (do inglês, *First In First Out*).

FLIT Unidade de Controle de Fluxo (do inglês, *Flow Control Unit*).

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

GPIO Entrada/Saída de Propósito Geral (do inglês, *General Purpose Input/Output*).

GPU Unidade de Processamento Gráfico (do inglês, *Graphic Processing Unit*).

HDL Linguagem de Descrição de Hardware (do inglês, *Hardware Description Language*).

HeMPS *Hermes Multiprocessor System*).

HOL Bloqueio de Cabeça-de-Fila (do inglês, *Head-Of-Line blocking*).

IP Propriedade Intelectual (do inglês, *Intellectual Property*).

IRQ Requisição de Interrupção (do inglês, *Interrupt Request*).

ISA Arquitetura do Conjunto de Instruções (do inglês, *Instruction Set Architecture*).

LB Esquerda Abaixo (do inglês, *Left Bottom*).

MMR Registrador Mapeado em Memória (do inglês, *Mapped Memory Register*).

MPE Elemento de Processamento Mestre (do inglês, *Master Processing Element*).

MPSoC Sistema intra-Chip Multiprocessado (do inglês, *Multiprocessor System-on-Chip*).

NI Interface de Rede (do inglês, *Network Interface*).

NoC Rede intra-Chip (do inglês, *Network-on-Chip*).

OCP Protocolo Aberto Centrado em núcleo (do inglês, *Open Core Protocol*).

PBD Plataforma Baseada em Projeto (do inglês, *Platform Based Design*).

PCI-E Interconexão Expressa de Componentes Periféricos (do inglês, *Peripheral Component Interconnect Express*).

PE Elemento de Processamento (do inglês, *Processing Element*).

PHIT Unidade Física (do inglês, *Physical Unit*).

RAM Memória de Acesso Aleatório (do inglês, *Random Access Memory*).

RR *Round Robin.*

RT Tempo Real (do inglês, *Real Time*).

SAFC Estaticamente Alocado, Totalmente Conectado (do inglês, *Statically Allocated, Fully Connected*).

SAMQ Estaticamente Alocado, Multi-Fila (do inglês, *Statically Allocated, Multi-Queue*).

SMPE Elemento de Processamento Gerenciador do Sistema (do inglês, *System Manager Processing Element*).

SoC Sistema intra-Chip (do inglês, *System-on-Chip*).

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

SPIN *Scalable Programmable Integrated Network*.

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

UART Receptor/Transmissor Assíncrono Universal (do inglês, *Universal Asynchronous Receiver/Transmitter*).

USB Barramento Serial Universal (do inglês, *Universal Serial Bus*).

VCI Interface de Componente Virtual (do inglês, *Virtual Component Interface*).

VCT Passagem Direta Virtual (do inglês, *Virtual Cut-Through*).

YAML *YAML Ain't Markup Language*.

Capítulo 1

Introdução

Avanços na tecnologia proporcionaram uma redução no tamanho dos transistores. Consequentemente, a alta densidade de transistores que podem estar em um único chip [1] levou ao surgimento dos sistemas integrados intra-chip, conhecidos como SoCs (*Systems-on-Chip*).

Uma das primitivas do projeto de SoCs é a reusabilidade de componentes. Dessa forma, usam-se módulos previamente disponibilizados e testados [1], os blocos IP (*Intelectual Property*). Esses blocos podem ser processadores, controladoras, memórias entre outras interfaces que serão integradas dentro de uma mesma pastilha de silício [1]. Assim, cabe ao projetista realizar apenas a escolha e a união desses componentes, geralmente garantida por meio de um mecanismo de comunicação [1]. Esse processo visa a disponibilização desses sistemas no mercado em um curto período de tempo. A reutilização de módulos proprietários, já disponíveis no mercado, reduzem ainda os custos de criação dos mesmos.

Entre os diversos tipos de sistemas que podem ser projetados dessa forma, destacam-se os sistemas que integram múltiplos processadores, conhecidos como sistemas multiprocessados intra-chip (MPSoCs, Multiprocessor Systems-on-Chip). O surgimento desse tipo de sistema se deu a fim de serem supridas altas demandas de performance aliadas à eficiência energética durante a execução de aplicações embarcadas concorrentes como compressão de vídeo, comunicação *wireless* e jogos, por exemplo [2].

Esses sistemas 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 [3] com mesma arquitetura. Já os sistemas heterogêneos utilizam diferentes arquiteturas nos elementos de processamento, de forma que possam haver diferentes processadores e hardwares dedicados. Assim, é possível aumentar a performance e reduzir o consumo energético em aplicações específicas [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*) [1]. 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 Plasma de arquitetura MIPS. Esse processador apresenta uma implementação simples capaz de executar toda a ISA de inteiros de 32 bits MIPS I(TM). Logo, busca-se nesse estudo desenvolver uma versão desse framework que possua um processador de arquitetura mais atual e com melhor desempenho, haja vista que a ISA MIPS tem algumas desvantagens para a implementação de um processador com maior performance. Segundo [6], algumas dessas desvantagens são:

1. a ISA apresenta muita otimização em torno do padrão de uso de pipelines com 5 estágios [6]. Dessa forma, tem-se a presença das instruções de salto que são atrasadas em um ciclo, o que complica a implementação de versões superescalares ou com superpipeline;

2. essa arquitetura tem pouco suporte para código independente de posição, como seus destinos de salto, que são pseudo-absolutos;

3. os imediatos de 16 bits usados no MIPS consomem muito espaço de forma que haja pouco desse espaço disponível para extensõs da ISA;

4. multiplicação e divisão utilizam registradores especiais, o que aumenta a quantidade de registradores no contexto;

5. essa ISA pressupõe que a unidade de ponto flutuante é um coprocessador separado, o que não é otimizado para implementações em um único chip;

6. na ABI padrão, dois registradores são reservados ao software do kernel, o que reduz o número de registradores disponíveis ao usuário;

7. tratar instruções de *load* e *save* desalinhadas, que são patenteadas, consome um espaço substancial e complica quase todas as outras instruções;

Já a arquitetura RISC-V, surgiu nos últimos anos e tem ganhado força de forma que empresas como Western Digital e Nvidia já passaram a utilizar essa arquitetura.

Ela se destaca por, além de ter código aberto, apresentar um conjunto de instruções incremental [7]. Isto é, pode-se usar apenas o conjunto de inteiros de 32 bits RISCV32I [7] ou pode-se usar o conjunto de inteiro com multiplicação e divisão, o RISCV32IM [7]. Mais ainda, pode-se usar conjuntos mais completos como RISCV64G [7], que apresenta até mesmo ponto flutuante de precisão dupla em 64 bits. Por esses motivos, nesse estudo será buscado interfacear um processador de arquitetura RISC-V.

O trabalho serve ainda como estudo de caso para entender os problemas e requisitos para o interfaceamento de uma nova arquitetura. Esse estudo é também útil para futuras implementações de MPSoCs heterogêneos, na HeMPS, contendo processadores que executam conjuntos de instruções específicos para uma aplicação, os ASIPs, (*Application Specific Instruction set Processor*).

1.1 Objetivos

Os objetivos do projeto são divididos em duas subseções. Na subseção 1.1.1, são abordados os objetivos gerais. Isto é, objetivos mais abstratos acerca dos conhecimentos adquiridos durante a realização do projeto. Já nos objetivos específicos em 1.1.2, abordam-se pontos relacionados à implementação do projeto

1.1.1 Objetivos gerais

A partir desse estudo, busca-se adquirir e aperfeiçoar conhecimentos relacionados a SoCs, NoCs, MPSoCs, sistemas operacionais embarcados e processadores de arquitetura RISC-V.

1.1.2 Objetivos específicos

Como esse estudo busca fazer a atualização do núcleo de processamento dos MPSoCs gerados pelo framework HeMPS, algumas etapas são necessárias:

1. escolher uma implementação pronta de processador com arquitetura RISC-V;
2. desenvolver e modificar módulos e interfaces do sistema que permitem a comunicação [8];
3. desenvolver ou modificar sistemas operacionais presentes no *framework* que sejam capazes de executar softwares no processador [8];
4. possibilitar a integração de mais de uma linguagem de descrição de hardware;

5. modificar e adaptar a forma de geração de MPSoCs de forma que se adéque à nova arquitetura.

1.2 Justificativa

Este trabalho busca realizar em um sistema, já bem consolidado, a troca do processador, que é um elemento essencial para sua execução. Primeiramente, foi definido que a arquitetura a ser utilizada é a RISC-V. Essa escolha se justifica pois a ideia é utilizar uma arquitetura que tenha código aberto, além de possuir um conjunto de instruções de inteiros de 32 bits (RV32I), semelhante ao que é utilizado atualmente, e suportado pela NoC presente.

Além disso, pode-se garantir, com essa atualização, uma futura migração para o formato de MPSoC heterogêneo na busca por melhor performance [8]. Para que isso seja viável, é necessário, primeiramente, fazer a HeMPS ser capaz de funcionar com diversas implementações de processadores, de forma homogênea. Posteriormente, entre aqueles que sejam mais adequados com a aplicação a ser executada, pode-se estabelecer uma plataforma heterogênea.

1.3 Metodologia da pesquisa

Para alcançar os objetivos estipulados, é necessário estudar e ter amplo conhecimento a respeito da plataforma HeMPS e sua implementação mais recente. Deve-se levar em conta o funcionamento dos mecanismos de comunicação e suas interfaces de integração. Além disso, é necessário conhecer o mecanismo de geração de acordo com a arquitetura atual e o sistema operacional utilizado.

O próximo passo é a escolha de um novo processador a ser integrado. Esse processador deve ter arquitetura RISC-V e características semelhantes ao atual. Entre essas, destacam-se o suporte às interrupções, e um conjunto de instruções completo para operações básicas.

A partir da escolha do processador com a arquitetura alvo, serão realizados testes com programas primeiramente feitos em *Assembly* para essa arquitetura e programas compilados a partir do C, assim como ocorre na plataforma atual. Ao validar a execução desse processador, o mesmo será integrado por partes. Primeiro, serão feitos testes do novo processador executando com a memória que se encontra na plataforma. Quando essa execução estiver em correto funcionamento, o processador será integrado. Serão alterados para isso os arquivos de geração necessários e respectivas ferramentas de compilação (*toolchains*).

Será necessário ainda modificar o sistema operacional para suportar a nova arquitetura e realizar a execução de programas nesse processador.

1.4 Estrutura do texto

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

1. Introdução: nesse capítulo são apresentados os objetivos, o problema e a metodologia de solução proposta além da justificativa.
2. Fundamentação teórica: aborda conceitos necessários para o entendimento do projeto e do seu desenvolvimento.
3. Integração de um novo processador na plataforma HeMPS: capítulo que descreve os passos necessários na execução desse projeto.
4. Resultados: demonstra a validação do sistema proposto após as alterações realizadas. Nele são mostradas, também, comparações de execução com a plataforma na sua forma original.
5. Conclusão: traz conclusões a respeito do trabalho realizado e apresenta ideias de trabalhos futuros.

Capítulo 2

Fundamentação teórica

Este capítulo trata da ambientação de fundamentos e conceitos teóricos necessários para a compreensão e desenvolvimento do projeto. Por meio desses fundamentos, busca-se, nas seções seguintes, estabelecer requisitos e caminhos para a solução de possíveis problemas decorrentes.

Na seção 2.1, é definido o conceito de sistema integrado intra-chip.

A seção 2.2 aprofunda conceitos de projetos de um tipo específico desses sistemas, os sistemas intra-chip multiprocessados.

A seção 2.3 apresenta o framework HeMPS, que é utilizado nesse estudo. Sua função é gerar MPSoCs parametrizáveis.

A seção 2.4 trata da arquitetura RISC-V abordando suas vantagens e comparando com outras arquiteturas.

2.1 Sistemas integrados intra-chip (SoCs)

O Sistema integrado intra-chip é um tipo de sistema formado pelo acoplamento de diversos componentes que buscam superar os exigentes requisitos de computação e comunicação [9], de acordo com uma metodologia de projeto, que é melhor descrita na subseção 2.1.1. Para isso, apresenta estruturas computacionais completas e interligadas em um único chip [1].

Esse tipo de sistema é capaz de incorporar uma grande gama de processadores e outros produtos baseados em eficiência de consumo energético, como aplicações sem-fio e automotivas [10]. Como exemplo, tem-se a *Snapdragon mobile processors*, uma família de SoCs desenvolvida pela Qualcomm que tem como objetivo executar múltiplas aplicações concorrentemente com melhor aproveitamento da capacidade energética [11]. O modelo 845 dessa família, mostrado na Figura 2.1, disponibiliza a integração de processador (CPU, *Central Processing Unit*), processador gráfico (GPU, *Graphic Processing Unit*), memória, módulo de segurança, módulos de rede e módulos de processamentos específicos de sinais

e multimídia [12], que foram escolhidos e integrados de acordo com a metodologia do projeto.

Figura 2.1: Diagrama de blocos do SoC Snapdragon 845 mobile (Fonte: [12]).

2.1.1 Metodologia de projeto de SoCs

O projeto de um SoC deve levar em conta diversos componentes relacionados tanto a software quanto a hardware, o que rapidamente o torna complexo em diversos níveis de abstração [9]. Para que um projeto desse nível seja viável ao mercado, usualmente, são utilizados módulos dedicados que possuem propriedade intelectual, mais conhecidos como blocos IP [1]. Esses blocos são compostos por núcleos verificados e testados previamente que se classificam, de acordo com sua forma de implementação, como *Soft-core*, *Firm-core* e *Hard-core* [1].

- *Soft-core*: são núcleos escritos usando uma linguagem de descrição de hardware (HDL, *Hardware Description Language*) sintetizável e são independentes da tecnologia [1].
- *Firm-core*: núcleos que possuem mais informações, como a presença de netlists [1], que são descrições das conexões dos componentes de um circuito eletrônico. Por isso, dependem mais da tecnologia utilizada [9], apesar de ainda serem parametrizáveis e flexíveis ao projeto [13].
- *Hard-core*: núcleos prontos para serem usados no sistema [1]. Possuem informações de temporização e leiaute [9], como o posicionamento e o roteamento das células

de hardware. Assim, são comprometidos pela tecnologia utilizada, ao passo que garantem maior corretude por parte do fabricante [13].

Esses núcleos são fornecidos separados ou em bibliotecas que usam como base um processador. Para esse, disponibilizam coprocessadores, interface para periféricos de entrada e saída, circuitos analógicos, barramentos e memória (s), de forma que os componentes essenciais são mostrados na Figura 2.2. Dessa forma, não sendo necessário desenvolver cada componente do sistema, os projetistas devem desenvolver ou adaptar a arquitetura de integração e comunicação desses blocos. Além disso, devem realizar a seleção dos mesmos levando em conta os requisitos funcionais e de desempenho que melhor se adéquem ao sistema proposto [1].

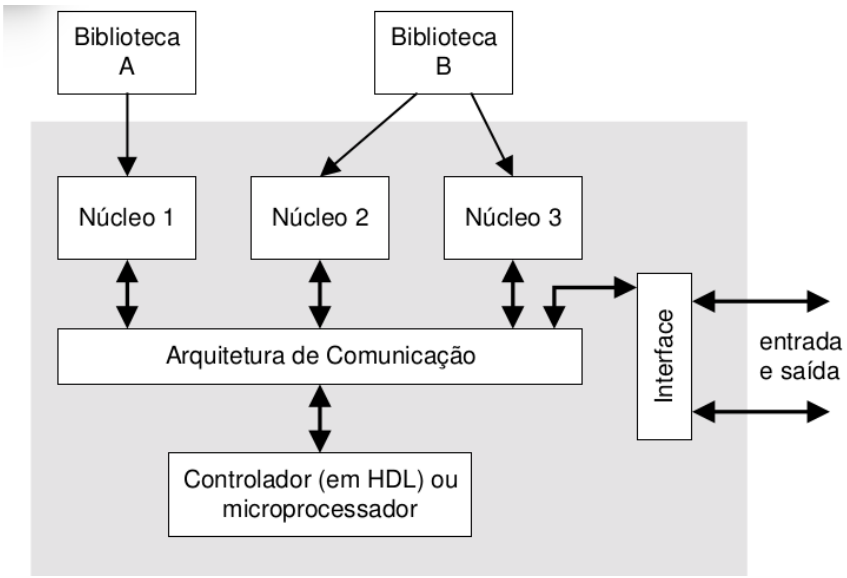

Figura 2.2: Arquitetura genérica de um SoC (Fonte: [1]).

Já a integração *hardware-software*, que também cabe ao projetista, pode ser um problema que afeta o tempo de lançamento para o mercado [1]. Como solução, são propostas interfaces padrões como o protocolo de núcleo aberto (OCP, *Open Core Protocol*) e a interface de componente virtual (VCI, *Virtual Component Interface*) [1]. Além do uso dessas interfaces, outra forma de conceber esse tipo de projeto é usando o conceito de projeto baseado em plataforma (PBD, *Platform Based Design*) [1]. Essas plataformas disponibilizam bibliotecas, ferramentas e componentes parametrizáveis [1], o que otimiza o desenvolvimento do sistema [9].

A atual tendência do desenvolvimento de SoCs de propósito geral, deve possibilitar a execução de um conjunto de diversas tarefas, cujo tamanho e exigência computacional são desconhecidos durante o projeto [3]. Uma forma de garantir os requisitos da atual

tendência é utilizando múltiplos elementos de processamento (PEs, *Processing Elements*) distribuídos nesses sistemas, resultando em sistemas intra-chip multiprocessados [3].

2.2 Sistemas intra-chip multiprocessados (MPSoCs)

Atualmente, a alta densidade de transistores presentes em um único chip de silício permite a utilização de diversos PEs em um único SoC. Dessa forma são definidos os sistemas intra-chip multiprocessados [4]. Esses PEs podem se comunicar por meio de diversas arquiteturas, entre as quais pode-se citar a ponto-a-ponto, multiponto e redes intra-chip. Na subseção 2.2.1, tem-se a conceitualização de redes intra-chips e uma abordagem dessas como arquitetura de comunicação entre os módulos de SoCs. Esse tipo de sistema é capaz de lidar com requisitos de mercado como boa performance computacional, orçamento limitado de gasto energético e tempo de lançamento [4].

Esses sistemas podem ser divididos em MPSoCs homogêneos e heterogêneos.

- Homogêneos: quando se usam as mesmas arquiteturas e mesma implementação em todos os elementos de processamento são simétricos. Podem ser assimétricos no caso de utilizar a mesma arquitetura e diferentes implementações [3].
- Heterogêneos: quando são utilizadas diferentes arquiteturas nos elementos de processamento [3].

Van Berkel [14] afirma que a utilização de arquiteturas de MPSoCs heterogêneos garantem, para aplicações que demandam alta capacidade computacional, maior eficiência entre desempenho e gasto energético [4]. Apesar de grande parte dos MPSoCs conhecidos serem homogêneos para garantir simplicidade, generalidade e flexibilidade, para alcançar determinadas capacidades de processamento, são necessárias integrações de núcleos especializados. Haja vista que um processador RISC suporta menos de uma instrução por ciclo de *clock*, alcançar 100 bilhões de operações por segundo somente seria possível a partir do uso de processadores especializados para determinadas tarefas em MPSoCs heterogêneos [3] [4].

Para escolher os elementos de processamento para um sistema desse tipo, deve-se levar em conta ainda a forma como será implementada a comunicação e o interfaceamento entre os elementos de processamento e os demais componentes. Entre eles, os blocos IP que são característicos em um SoC. A arquitetura de comunicação pode ser escolhida de forma que seja possível uma maior reusabilidade, dado que o reaproveitamento de componentes é uma premissa de projeto de um SoC [9]. Como arquiteturas, podem ser citadas as conexões ponto-a-ponto e multiponto (ou barramento) [9]:

- Ponto-a-ponto: arquitetura que utiliza canais dedicados. Assim, garante-se bom desempenho, pois permite múltiplas conexões simultaneamente, mas uma má reusabilidade [1]. Essa arquitetura é exemplificada na Figura 2.3.

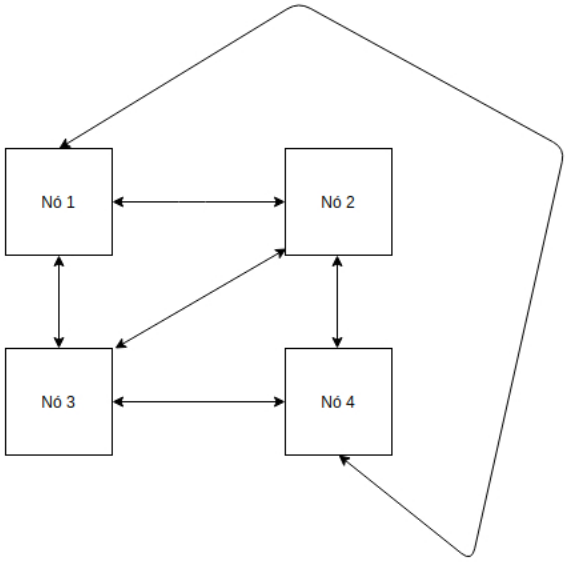

Figura 2.3: Diagrama da arquitetura ponto-a-ponto.

- Multiponto: arquitetura que pode ser implementada via árbitro, como na Figura 2.4, ou via ponte (com árbitro), para arquiteturas hierárquicas [1]. Na abordagem via árbitro, há maior reusabilidade em detrimento de performance e escalabilidade, ao utilizar canais compartilhados com um escalonador central [1]. Nessa abordagem, só pode haver uma comunicação entre dois núcleos por vez, que são escolhidos pelo árbitro.

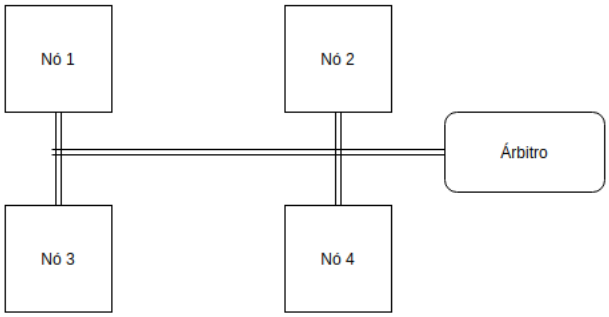

Figura 2.4: Diagrama da arquitetura multiponto.

Na abordagem via ponte, mostrada na Figura 2.5, as desvantagens são amenizadas ao usar barramentos separados para diferentes funcionalidades [1]. Já nessa abordagem, é possível realizar duas comunicações simultâneas. O Árbitro 1 decide o par que vai se comunicar de um lado e o árbitro 2, o par do outro.

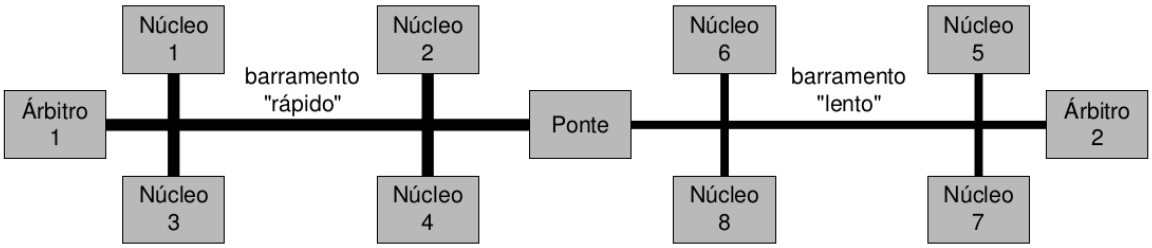

Figura 2.5: Diagrama da arquitetura multiponto hierárquica com múltiplos barramentos, sendo um rápido de grande largura de banda e outro lento de pequena largura (Fonte: [1]).

Entre as arquiteturas apresentadas, a multiponto é preferida devido à sua reusabilidade. Ao utilizar barramentos hierárquicos por meio de ponte, é possível permitir maior escalabilidade de núcleos com menor detrimento de performance [1]. Por exemplo, tem-se a estrutura hierárquica de barramentos dos chipsets atuais. Elas são divididas em uma ponte norte (*northbridge*) e uma ponte sul (*southbridge*), que correspondem à integração de elementos como CPU, RAM e PCI-E ou AGP na primeira e dispositivos de entrada e saída como USB, áudio e serial na segunda [15]. No entanto, como os agrupamentos são específicos, há perda de generalidade e flexibilidade do barramento [1].

Levando em consideração esses problemas, busca-se uma arquitetura que tenha reusabilidade, escalabilidade e paralelismo. Nesse contexto, é necessário utilizar canais ponto-a-ponto compartilhados por meio de chaveamento [1]. Essa solução é alcançada ao utilizar uma rede de interconexão chaveada [1]. A implementação desse tipo de rede em sistemas integrados é chamada de rede intra-chip (NoC).

2.2.1 Redes intra-chip (NoCs)

As redes intra-chip são interconexões entre diferentes módulos em um mesmo chip. Em um SoC, elas são responsáveis por interconectar blocos IP a fim de mantê-los integrados. Como será visto na seção seguinte, o MPSoC utilizado nesse trabalho possui, como nós da rede, seus elementos de processamento. A comunicação entre esses nós é feita pela transmissão de informações por mensagens subdivididas em pacotes por meio de uma estrutura de enlaces e roteadores [9], mostrados na Figura 2.6.

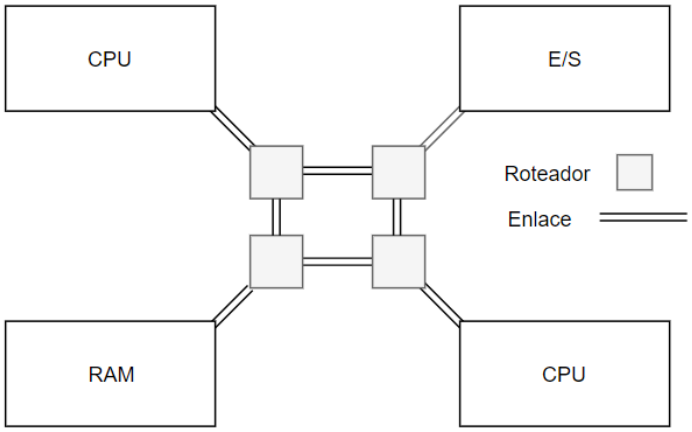

Figura 2.6: Diagrama de blocos de uma rede intra-chip simples formada por enlaces e roteadores.

As mensagens trocadas na NoC são compostas de informações enviadas de um nó (emissor) a outro (receptor) [1]. Essas mensagens têm suas estruturas compostas de três partes: cabeçalho, carga útil e terminador.

- Cabeçalho: possui informações necessárias para o roteamento e de controle de propagação da mensagem [9].
- Carga útil: possui a informação a ser transferida.
- Terminador: possui informações que são usadas para sinalizar o fim da mensagem e detecção de erros [9]. Essa parte pode ser omitida da mensagem, assim como é na NoC da plataforma estudada.

Logo, as mensagens são envelopadas pelo cabeçalho e pelo terminador [9]. Elas são divididas, para a transmissão, em pequenos pacotes de estruturas semelhantes (cabeçalho, carga útil e terminador). Esses pacotes podem ser divididos, ainda, em unidades menores transmitidas pelos canais da NoC, que serão abordados abaixo.

Sabendo que as NoCs são redes de interconexões chaveadas, seus enlaces típicos se formam por dois canais unidirecionais opostos (configuração conhecida como *full-duplex*) [1]. Sendo que um canal é responsável pelo envio e o outro pelo recebimento de pacotes [1]. Esses canais possuem uma largura denominada PHIT (PHisical unIT), que é correspondente à unidade física. De outra forma, PHIT indica o número de bits transmitidos paralelamente por um canal [1]. Logo, um conjunto de PHITs determina o tamanho da unidade de controle de fluxo, FLIT. Sendo essa a menor unidade pela qual se pode controlá-lo [9]. Além disso, um pacote a ser transmitido é composto por um conjunto de FLITs. A relação entre uma mensagem, seu pacote, seus FLITs, e seus PHITs é mostrada na Figura 2.7.

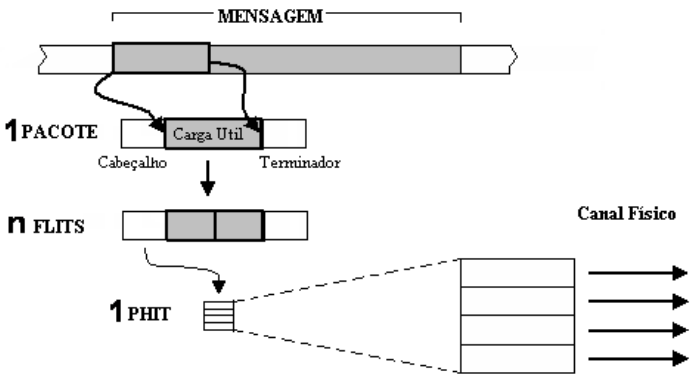

Figura 2.7: Divisão de uma mensagem a ser transmitida (Fonte: [9]).

Os enlaces possuem, em cada canal, um barramento dedicado para a transferência de dados e outras conexões para sinais auxiliares de controle de fluxo, erro, e formatação de pacotes [1]. A Figura 2.8 mostra a estrutura física de um enlace de um projeto de pesquisa em NoCs, chamado SPIN (Scalable Programmable Integrated Network) [1].

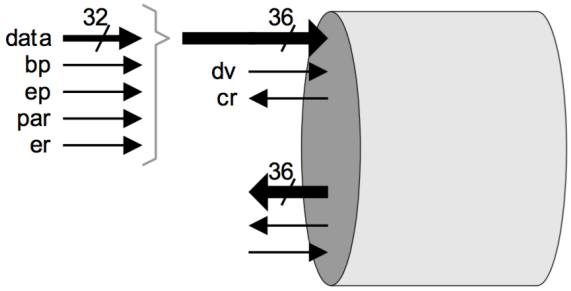

Figura 2.8: Estrutura física de um enlace SPIN composto por 2 canais unidirecionais opostos e seus sinais auxiliares (Fonte: [1]).

Sendo o barramento de dados de 32 bits, representado por "data", os demais sinais "dv", "cr", "bp", "ep", "par" e "er" são os sinais auxiliares, cujos significados estão na Tabela 2.1.

Tabela 2.1: Significado dos sinais auxiliares do barramento da NoC SPIN (Fonte: [1]).

Sinal	**Largura**	**Definição**
dv	1 bit	Indicador de dado válido
cv	1 bit	Retorno de crédito
bp	1 bit	Início do pacote
ep	1 bit	Fim do pacote
par	1 bit	Paridade do phit
er	1 bit	Indicador de erro

Por meio desses enlaces, a forma de interconexão entre os roteadores da NoC define a topologia da rede [9]. Assim como em redes tradicionais, NoCs podem ter diversas topologias. Quanto a essas topologias, essas redes podem ser agrupadas em diretas e indiretas [1].

- Diretas: são redes em que cada nó é formado pelo par entre um componente e um roteador associado [1]. Além disso, os nós possuem conexões diretas ponto-a-ponto com determinados vizinhos [1]. Assim, uma transmissão entre emissor e receptor que não são vizinhos envolve transmitir por meio de nós intermediários [1]. Isso se dá pois, devido a problemas de escalabilidade, não é possível alcançar a topologia ideal, que implicaria em uma rede totalmente conectada. Dessa forma, são propostas redes ortogonais [9] [1]. Essas redes implicam que enlaces entre nós vizinhos produzem deslocamentos em apenas uma dimensão em um espaço de n dimensões [1]. Dessas redes, tem-se como tipos mais comuns, redes em malha, redes toróides e redes hipercubos, presentes na Figura 2.9 em (a), (b) e (c) respectivamente [1].

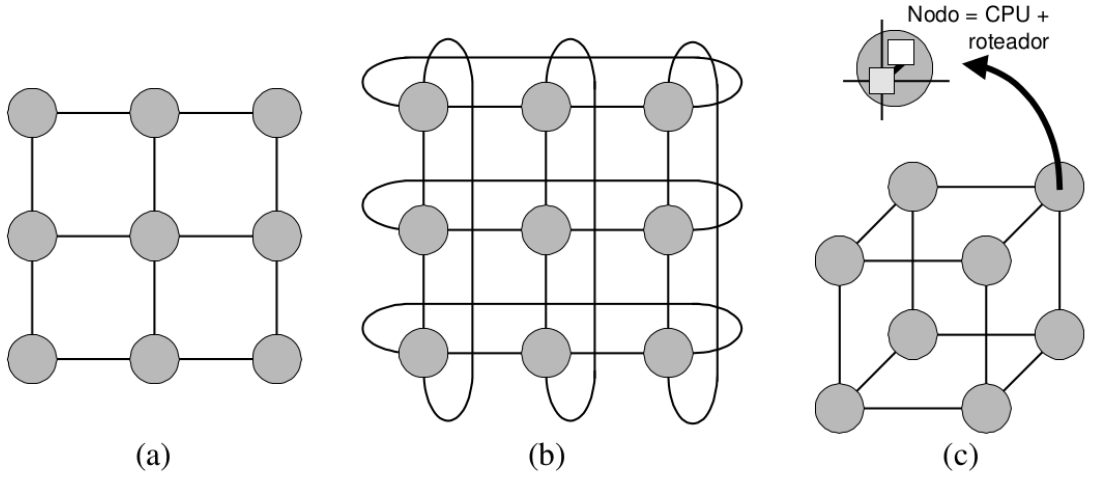

Figura 2.9: Topologias de NoCs diretas em que a) representa a malha. Em b), tem-se a toróide e em c), o hipercubo (Fonte: [1]).

Atualmente, as redes em malha são as mais utilizadas em NoCs. Isso se deve pelo custo-benefício entre o custo de implementação e o desempenho. Nessa topologia, a estrutura regular e amigável em forma de grade facilita as conexões de fios em nível de circuito e, consequentemente, a implementação [16].

- Indiretas: são redes cujos nós são compostos apenas pelos componentes e se conectam, por meio de uma interface [1], a uma rede de roteadores que possuem portas de duas direções para comunicação entre nós ou entre outros roteadores [1]. Além

disso, apenas alguns roteadores se ligam aos nós e, dessa forma, somente esses podem ser emissores e receptores das mensagens. As topologias se definem, então, pelas ligações desses roteadores entre roteadores e entre nós. São mais comuns os tipos *crossbar* e multiestágio, mostradas na Figura 2.10 em (a) e (b) respectivamente [1].

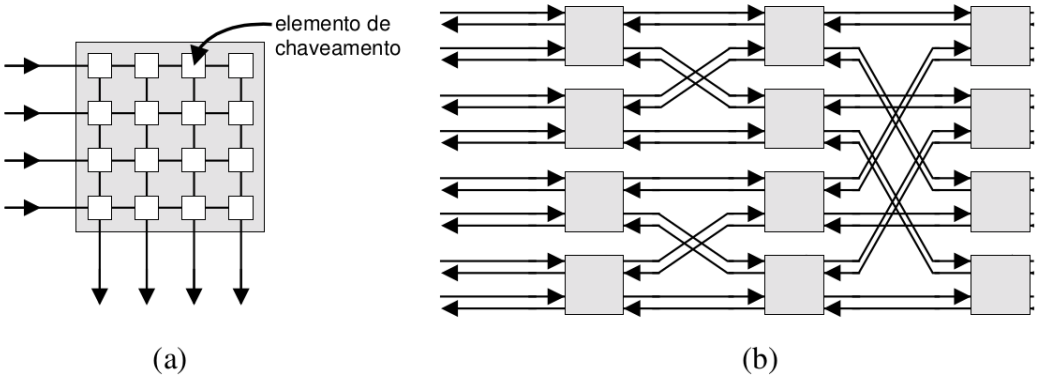

Figura 2.10: Topologias de NoCs indiretas em que a) apresenta o tipo *crossbar* e b) o multiestágio (Fonte: [1]).

Os roteadores da NoC são projetados de acordo com os principais mecanismos e parâmetros [9] como controle de fluxo, roteamento, chaveamento, arbitragem e memorização [1]. Dados esses parâmetros, um roteador é composto por um núcleo de chaveamento, lógica de controle para roteamento e arbitragem, portas de entrada e saída que podem possuir memórias (buffers) para realizar o armazenamento do pacote e, por fim, módulos de controle de fluxo para protocolos físicos [1]. A Figura 2.11 mostra o diagrama de um roteador característico.

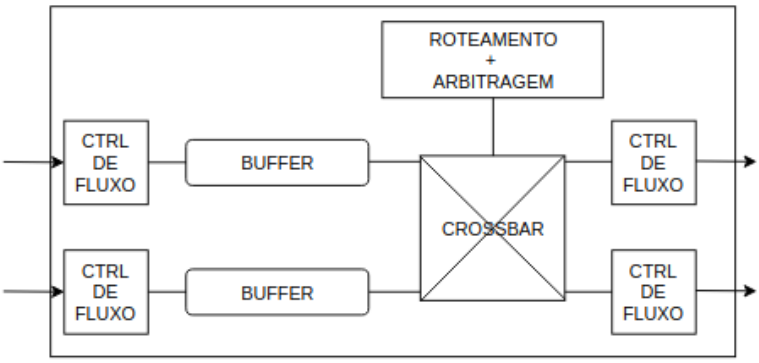

Figura 2.11: Unidade de roteamento de uma NoC com 2 portas de entrada e 2 portas de saída.

Entre os mecanismos apresentados, o controle de fluxo é um protocolo responsável por verificar a disponibilidade do buffer de entrada do receptor (cheio ou vazio) [9]. Isto é, se o receptor está apto a receber uma mensagem. Esse controle é comumente implementado de duas formas: via aperto de mão (*handshake*) ou via créditos [9].

- Implementação via *handshake*: uma mensagem é transmitida ao receptor perguntando se o mesmo está apto a receber o próximo pacote [9].
- Implementação via créditos: o emissor tem conhecimento prévio de quantos pacotes podem estar armazenados no receptor, por meio de um contador que representa os créditos disponíveis, e envia de acordo. Quando o receptor encaminha o pacote para outro nó, ele envia também uma mensagem aos possíveis emissores alertando quantos espaços de pacotes foram disponibilizados (créditos) [1].

O mecanismo de roteamento é responsável pela escolha da rota de acordo com o algoritmo utilizado. Esse algoritmo pode ser escolhido baseado na sua conectividade, adaptatividade, liberdade de *deadlocks* e *livelocks* e tolerância a falhas [1].

- Conectividade: rotear de qualquer origem para qualquer destino.
- Adaptatividade: buscar caminhos alternativos quando o atual apresentar problemas.
- Liberdade de *deadlocks* e *livelocks*: garantir que pacotes não fiquem sem alcançar seus destinos circulando pela rede (*livelock*) ou bloqueados devido a outras condições (*deadlock*).
- Tolerância a falhas: rotear pacotes mesmo na presença de falhas por meio de reenvios.

Quanto às técnicas utilizadas, os algoritmos podem ser determinísticos ou adaptativos [1].

- Determinísticos: independentemente do estado da rede, uma origem sempre vai a determinado destino pelo mesmo caminho.
- Adaptativos: a distribuição é viabilizada de acordo com o estado da rede.

O mecanismo pode ainda ser centralizado, distribuído ou definido pela fonte [1].

- Centralizado: os caminhos são definidos por um controlador central presente na rede.
- Distribuído: cada pacote possui apenas o endereço de destino e cada roteador por onde passa se responsabiliza por determinar o caminho.

- Definido pela fonte: o pacote armazena toda a tabela de roteamento do caminho que deve percorrer antes de ser injetado na rede.

O chaveamento é a parte responsável por multiplexar uma entrada de destino a uma saída de origem no momento certo. Ele pode ser feito baseado em pacotes ou circuitos por meio de caminhos dedicados [1].

- Chaveamento por pacote (ou flits): mensagens são quebradas em pacotes que são repassados por políticas como armazenar e encaminhar (*Store-And-Forward*), política de transpasse virtual (VCT, *Virtual Cut-Through*) ou buraco de minhoca (*Wormhole*) [1].
  - *Store-And-Forward*: roteador só começa o encaminhamento quando o pacote já foi recebido completamente em seu buffer [9].
  - Política VCT (*Virtual Cut-Through*): pacotes são enviados por partes assim que o receptor avisar que há espaço disponível em seu buffer [9].
  - Buraco de minhoca (*Wormhole*): flits de um pacote são encaminhados um a um em modo *pipeline*. O flit de cabeçalho (primeiro a ser transmitido) possui a informação de roteamento e por isso todos os demais devem segui-lo. Caso o tamanho do pacote seja maior que o espaço disponível no buffer de destino, os flits de um pacote bloqueado são mantidos em outros roteadores pelo caminho. Dessa forma, o requisito de tamanho de buffers são menores do que nas outras abordagens citadas [1].
- Chaveamento por circuito: as mensagens são transmitidas por meio de caminhos dedicados que existem durante a transmissão [1]. É necessário criar uma conexão para que esse caminho seja definido.

A arbitragem é responsável por conectar uma entrada a uma saída de acordo com a rota escolhida pelo roteador. Ela pode ser feita de forma centralizada ou distribuída [9].

- Centralizada: roteamento e o mecanismo de arbitragem são processados por um mesmo módulo.
- Distribuída: roteamento e o mecanismo de arbitragem são processados em diferentes módulos tal que os emissores possuem em suas saídas módulos de roteamento e nas entradas módulos de arbitragem.

Além disso, o mecanismo de arbitragem é fundamental para a resolução de conflitos provenientes da concorrência entre pacotes por uma porta de saída, evitando que um

pacote fique indefinidamente esperando por uma oportunidade de ter a posse momentânea dessa porta (*starvation*) [1]. Para chegar a solução desses conflitos, pode-se levar em conta alguns critérios, como análise de prioridades estáticas, prioridades rotativas (RR, *Round-Robin*), primeiro a chegar é o primeiro servido (FCFS, *First-Come-First-Served*), entre outros [1].

- Prioridades estáticas: os pacotes que fazem requisições ao árbitro possuem valores de prioridades fixas pré-definidas. Dessa forma, de acordo com o tráfego, requisições de pior prioridade podem levar seus pacotes a sofrerem *starvation*.
- *Round-Robin*: os pacotes que fazem requisições ao árbitro possuem valores de prioridades que variam. Isto é, após o uso do recurso solicitado, o pacote que fez a requisição recebe a pior prioridade de forma que esse seja o último da fila de próximos.
- FCFS: as requisições serão atendidas sempre na ordem de chegada, o que pode fazer com que longas requisições causem *starvation* em outras.

A memorização é uma técnica presente em roteadores que realizam o chaveamento por pacote. Ela consiste em armazenar os dados no roteador enquanto o receptor não puder recebê-los e, dessa forma, consegue evitar que dados sejam perdidos nas entradas dos *buffers* por falta de espaço [1]. Essa técnica pode ser implementada por meio de estratégias como memorização centralizada compartilhada, memorização na entrada e memorização na saída [1].

- Memorização centralizada compartilhada: um *buffer* central é presente na rede para abrigar pacotes bloqueados. Esse *buffer* deve ser projetado de forma que, tenha 2N portas, em que N representa o número máximo de acessos simultâneos possíveis. Deve ter, ainda, uma porta para leitura e outra para escrita. Além disso, esse *buffer* central tem alocação dinâmica (CBDA, *Centrally-Buffered, Dynamically-Allocated*), de forma que os pacotes bloqueados recebam um endereço dinamicamente.
- Memorização na entrada: o espaço de memória é particionado entre os canais de entrada do roteador. Para isso, utilizam-se *buffers* independentes cujas estratégias podem ser: *buffer* em que o primeiro a entrar é o primeiro a sair (FIFO, *First-In-First-Out*), *buffer* estaticamente alocado e totalmente conectado (SAFC, *Statically Allocated, Fully Connected*), *buffer* estaticamente alocado com multi-filas (SAMQ, *Statically Allocated, Multi-Queue*) e *buffer* dinamicamente alocado com multi-filas (DAMQ, *Dynamically-Allocated, Multi-Queue*):

– *buffers* FIFO: Representados na Figura 2.12, os *buffers* possuem espaço de memória fixo e os dados são lidos na ordem em que são escritos. A desvantagem consiste na possibilidade de um dado enfileirado na frente da fila impedir que outros dados sigam a diante, o que configura um bloqueio por cabeça da fila (HOL, *Head-Of-Line*) [1].

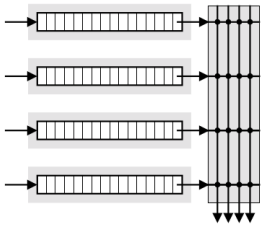

Figura 2.12: *Buffers* FIFO (Fonte: [1]).

– *buffers* SAFC: para solucionar o bloqueio HOL, esses *buffers* são particionados em N partições de tamanhos iguais. Cada uma dessas partições deve ser atribuída de forma estática à uma saída. Para isso, é necessário um *crossbar* $N^2 \alpha N$ ou $N \cdot N \alpha 1$, o que causa um complexo gerenciamento, um complexo controle de fluxo e baixa taxa de utilização [1]. Esse tipo de *buffer* é mostrado na Figura 2.13.

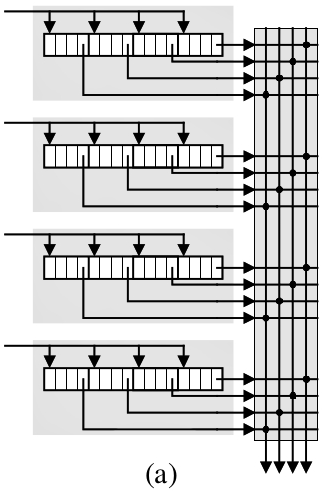

Figura 2.13: *Buffers* SAFC (Fonte: [1]).

– *buffers* SAMQ: apesar de ainda manter a baixa taxa de utilização e o complexo controle de fluxo, a gerência do crossbar é simplificada. Mostra-se na Figura 2.14 que isso se dá pela conexão de partições em multiplexadores o que garantem saídas dinâmicas [1].

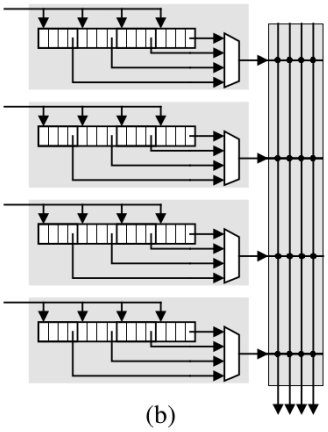

Figura 2.14: *Buffers* SAMQ (Fonte: [1]).

– *buffers* DAMQ: mostrado na Figura 2.15, elimina os problemas quanto à complexidade de gerências ao realizar o particionamento dos *buffers* de entrada de forma dinâmica. Entretanto, sua implementação física é mais complexa [1].

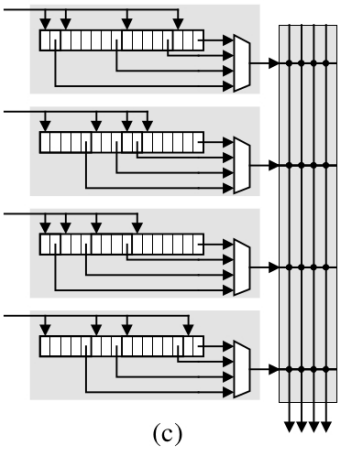

Figura 2.15: *Buffers* DAMQ (Fonte: [1]).

- Memorização na saída: particiona o espaço de memória das saídas na forma de *buffers* FIFO. Dessa forma, se faz necessário que cada *buffer* consiga atender as requisições das N entradas. Uma forma de se garantir isso, é fazendo com que o *buffer* tenha N portas de escrita ou apenas uma porta desse tipo que opere com velocidade N vezes mais rápida que as das entradas [1].

2.3 MPSoC baseado em NoC homogênea parametrizável - HeMPS

Esta seção busca caracterizar o *framework* HeMPS (*Hermes Multiprocessor System*), composto de elementos de processamento interconectados pela NoC Hermes, que é proposto para ser utilizado nesse estudo. A subseção 2.3.1 fala do processador utilizado atualmente. Em 2.3.2 é mostrada a infra-estrutura da NoC Hermes. Já a subseção 2.3.3, aborda a comunicação entre tarefas por meio do módulo de acesso direto à memória e interfaceamento de rede (DMNI, *Direct Memory Network Interface*). As subseções 2.3.4 e 2.3.5 abordam a execução de tarefas e o funcionamento do sistema operacional (*microkernel*), respectivamente.

Desenvolvido pelo grupo de pesquisa GAPH, esse *framework* se caracteriza por ter código aberto e ser usado para avaliar e implementar propostas de projetos de múltiplos processadores [3] que podem ser gerados em VHDL ou SystemC. Se caracteriza, ainda, por ser um MPSoC homogêneo simétrico de propósito geral, apesar de oferecer suporte a configurações heterogêneas [3]. Além disso, tem tamanho parametrizável [4] e os seus elementos de processamento são integrados por meio da NoC Hermes [3]. Na Figura 2.16 é visto em (a) um visão geral da arquitetura desse MPSoC. Já em (b), tem-se a arquitetura do PE presente no sistema.

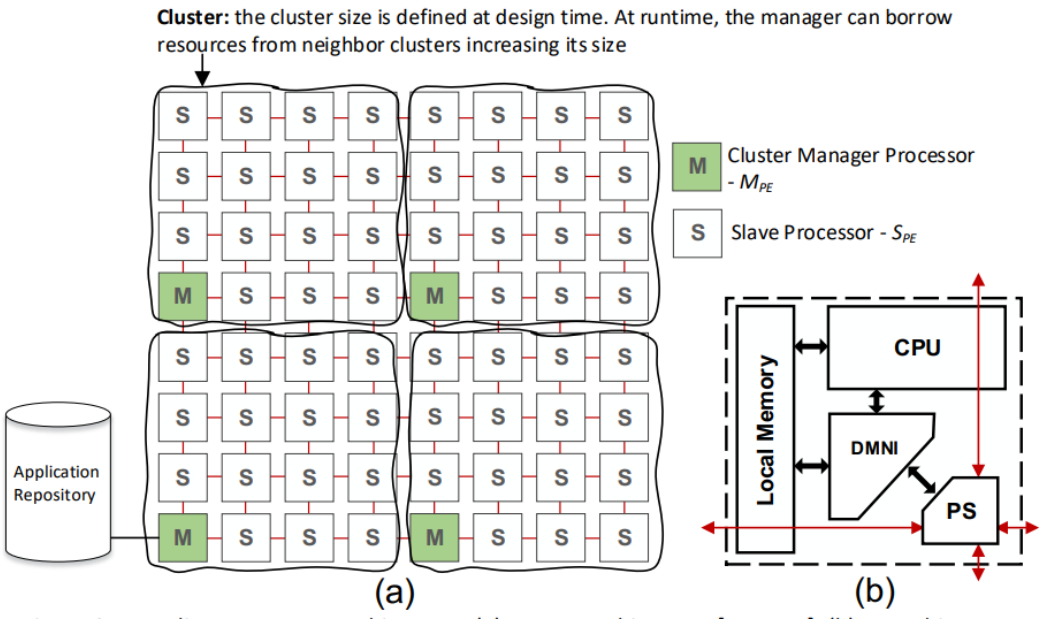

Figura 2.16: Arquitetura atual do MPSoC gerado pelo HeMPS (a). Nesse caso, tem-se um sistema 8×8 gerado. Em (b) é visto o elemento de processamento detalhado (Fonte: [3]).

Nesse sistema, como visto na Figura 2.16 (a), os PEs são organizados em *clusters* que possuem um gerente (MPE, *Manager Processing Element*), ou mestre. Tem-se ainda uma versão específica dos MPEs, os MPEs de sistema (SMPEs, *System Manager Processing Elements*). Esse é o mestre conectado ao repositório de aplicações (*application repository*, na imagem). Ele é o responsável pelo mapeamento de tarefas e pelo controle da entrada de novas aplicações em tempo de execução [3]. Essas aplicações ficam armazenadas em uma memória externa, que é um repositório de aplicações [3]. Os demais componentes desses *clusters* são os elementos de processamento escravos (SPEs, *Slave Processing Elements*). Esses são responsáveis por executar as tarefas de forma distribuída, de forma que suportem a execução multi-tarefa e a comunicação entre elas [3]. Haja vista que a implementação da HeMPS utilizada trata-se de um sistema homogêneo, a diferenciação desses elementos de processamento é dada unicamente no kernel que executam [3].

A arquitetura desses PEs, mostrada na Figura 2.16 (b), inclui um processador, um módulo que integra o acesso direto à memória e o interfaceamento de rede (DMNI), além de uma memória local de duas portas e um roteador [17], que é o roteador original da NoC Hermes [3].

O processador atual é o Plasma [18], de arquitetura MIPS [8]. O módulo de DMNI é responsável por fornecer um caminho direto entre o roteador da NoC e a memória interna [3]. Esse caminho é possível por meio da memória paginada de duas portas. Em uma dessas portas, o endereço, relativo à página contendo as instruções e dados correntes, é definido pelo processador. A outra porta conecta-se diretamente ao DMNI. Mais ainda, o uso do acesso direto à memória conectado à interface de rede permite paralelismo entre a execução de tarefas pelo processador enquanto ocorrem transmissões de dados pela rede [19]. Já os roteadores, são descritos com mais detalhes na seção referente a rede Hermes.

2.3.1 Processador Plasma

O núcleo de processamento presente nos PEs do HeMPS é o Plasma. Trata-se de um processador RISC sintetizável de 32 bits que é capaz de executar o conjunto de instruções de inteiros da arquitetura MIPS - MIPS I (TM), exceto as instruções de load e store desalinhadas [18].

Em seu projeto, é usado um sistema pipeline de 3 estágios e arquitetura de memória Von Neumann, colocando em um mesmo barramento dados e instruções. Para isso, é necessário um módulo de controle (`Mem_ctrl`) para gerenciar os acessos à memória RAM. Esse processador possui ainda uma unidade dedicada de divisão e multiplicação (`Mult`) via somas e subtrações [8].

Para ser interfaceado nesse MPSoC, o processador teve que passar por algumas modificações: inclusão do mecanismo de paginação adequado à memória utilizada, instrução `syscall` e registradores mapeados em memória (MMRs, *Memory-Mapped Registers*) [8].

- Mecanismo de paginação: usado para mapear tarefas em páginas de memória e facilitar o mapeamento e a migração de tarefas [20], esse mecanismo consiste em dividir a memória em partes de tamanho igual. No entanto, apresenta desvantagens ao restringir o tamanho das tarefas a serem executadas ou gerar fragmentação interna.
- Instrução `syscall`: efetua interrupções de software [8].
- Registradores mapeados em memória: para garantir acesso rápido a informações de interrupção e comunicação, há a leitura e a escrita de registradores que são fixos na memória.

Para gerar executáveis com instruções MIPS para esse processador, utiliza-se uma *toolchain* derivada do GCC, o gcc-mips-elf [8]. O código gerado utiliza a arquitetura de memória Von Neumann, com instruções e dados presentes no mesmo segmento.

A arquitetura desse processador pode ser vista na Figura 2.17.

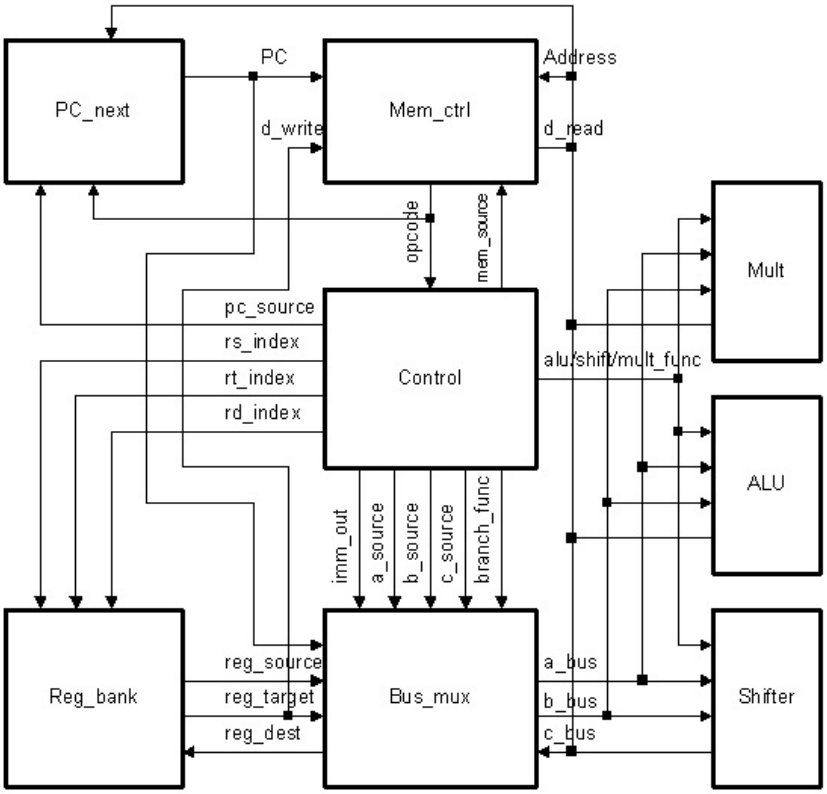

Figura 2.17: Diagrama de blocos do processador Plasma (Fonte: [18]).

2.3.2 Rede intra-chip Hermes

A rede Hermes utilizada pelo HeMPS é uma rede parametrizável cuja topologia é a de malha 2D [21].

Os roteadores dessa rede são formados por: buffers de entrada, uma lógica de controle (*Switch Control*) - para arbitragem e roteamento - compartilhada por todas as portas, crossbar interno e o máximo de cinco portas bidirecionais (norte, sul, leste, oeste e local) [22]. A porta local é usada para realizar a comunicação entre o roteador e o restante do núcleo (PE) em que está. As demais portas servem para conectá-lo aos roteadores vizinhos [21] [22]. Na Figura 2.18 pode-se observar uma hermes instanciada com 9 roteadores (3×3) e a estrutura interna de um deles.

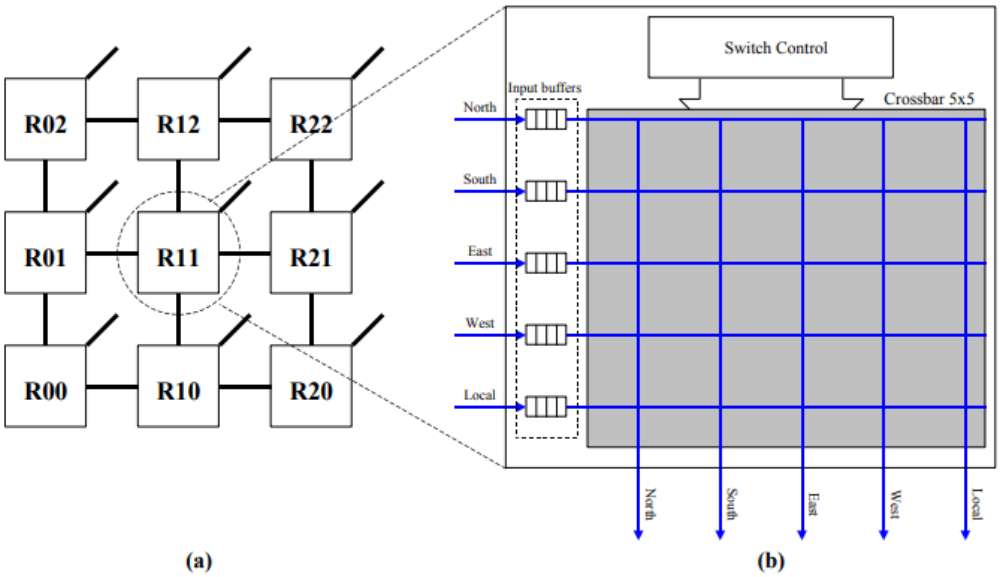

Figura 2.18: Arquitetura atual da rede Hermes em um sistema 3×3 em (a). Em (b) é vista a arquitetura detalhada do roteador (Fonte: [22]).

A comunicação entre esses roteadores usa o chaveamento de pacotes para transmitir *flits* por meio da política *Wormhole* [21], além de controle de fluxo baseado em créditos [22]. Quanto à arbitragem, a fim de se evitar *starvation*, e garantir justiça por meio de prioridades dinâmicas [21], é usado um escalonador do tipo *Round-Robin*. Já o algoritmo de roteamento utilizado é o XY. Nesse algoritmo os roteadores são georreferenciados em um plano cartesiano e conhecem tanto as suas coordenadas quanto as de seu destino. Dessa forma, caminham primeiramente na direção horizontal (eixo x) e depois na direção vertical (eixo y) [23] [21].

2.3.3 Interface de rede e acesso direto à memória (DMNI)

Para implementar o envio e o recebimento de pacotes entre diferentes PEs, é necessário utilizar uma interface de rede (NI, *Network Interface*) [3], para conectar o processador à NoC, e um módulo de acesso direto à memória (DMA, *Direct Memory Access*), para enviar e receber os pacotes [22]. Mais ainda, para melhorar as transações de memória pode-se utilizar um módulo DMNI, que integra o DMA e a NI. Essas duas propostas podem ser vistas na Figura 2.19.

Na Figura 2.19 (b) pode-se observar a proposta de utilização de DMA e NI, comuns em MPSoCs baseados em NoCs [3]. Nesse caso, é necessário que o processador realize o interfaceamento com cada um desses módulos, além de que cada um exige controle individual via software [3]. Essa abordagem funciona da seguinte forma: primeiro, o processador programa o módulo DMA escrevendo o endereço de memória inicial do pacote a ser transferido e seu tamanho em registradores mapeados em memória. Depois, ele retorna à sua execução, que havia sido interrompida, enquanto o DMA transfere os dados da memória e para a memória [3].

Já na Figura 2.19 (c), tem-se a utilização de um módulo DMNI. Esse módulo, que é utilizado no sistema em estudo, garante uma conexão direta entre o roteador e a memória local. Dessa forma, se faz necessário apenas uma interface entre o módulo e o software, além de garantir, por meio de um árbitro de acesso à memória, recepção e transmissão simultânea de pacotes [3]. A interface com o software é dada por meio dos registradores mapeados em memória, que servem para controlar esse módulo.

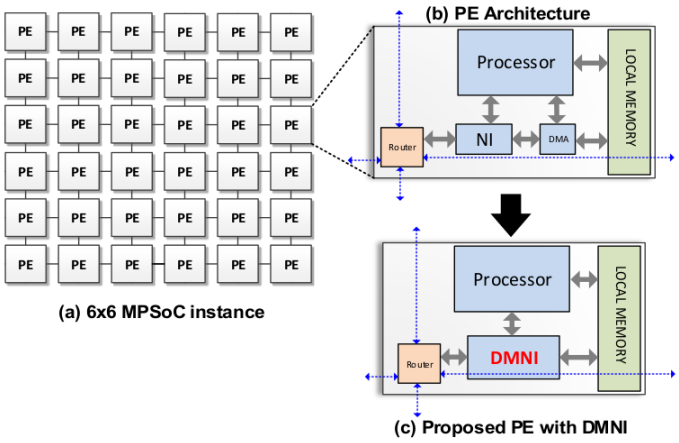

Figura 2.19: MPSoc 6×6 (a) com duas propostas de implementação do PE, com DMA e NI separados (b) e integrados em uma DMNI (c) (Fonte: [3]).

Uma outra funcionalidade relevante desse módulo, é a possibilidade de enviar pacotes em que seus conteúdos estão dividido em duas regiões de memória diferentes [3]. Isso

ocorre, por exemplo, ao formar um pacote com o cabeçalho, que está em uma região, e a sua carga útil, que está em outra. A Figura 2.20 apresenta a arquitetura do módulo, que possui 3 partes principais: os módulos de envio (*send*), recebimento (*receive*) e árbitro (*Memory Access Arbiter*) que se conectam à NoC e à memória local.

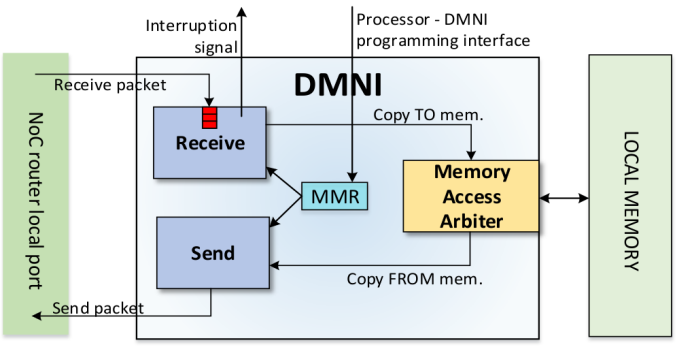

Figura 2.20: Arquitetura da DMNI proposta (Fonte: [3]).

- Módulo de envio: sua função é injetar pacotes na NoC, sendo que é possível transferir dois blocos de memória em uma única transferência [3]. Sua máquina de estados finita (FSM, *Finite State Machine*) é mostrada na Figura 2.21.

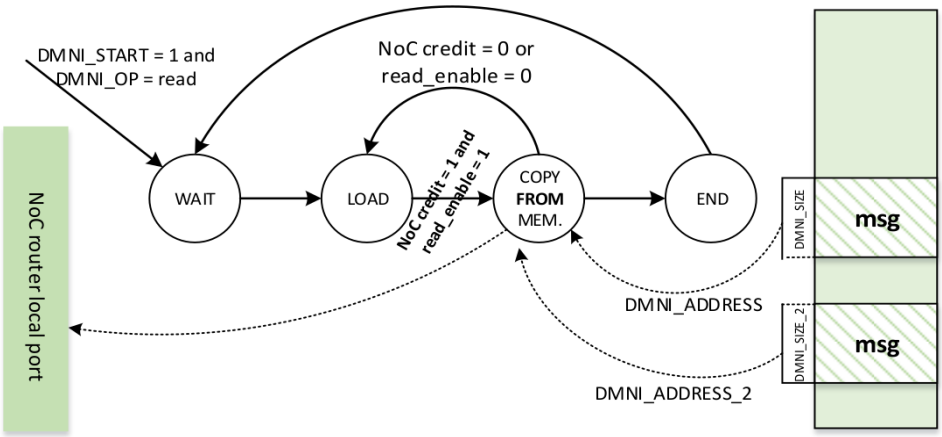

Figura 2.21: Máquina de estados finitos do módulo de envio (Fonte: [3]).

Primeiramente, a máquina aguarda a configuração dos registradores mapeados em memória no estado `Wait`. Quando os registradores `DMNI_START` e `DMNI_OP` recebem os valores 1 e `read` respectivamente, a máquina avança para o estado `Load`, no qual aguarda ter créditos da NoC disponíveis e a operação de leitura permitida pelo árbitro. Se as condições são satisfeitas, o dado é lido pela memória e injetado na porta local do roteador, no estado `Copy from mem`. No caso de haver um

segundo dado a ser lido, a máquina volta ao estado `Load` e o transmite em `Copy from mem`, antes de ir para o estado `End`, que retorna ao `Wait` [3].

- Módulo de recebimento: esse módulo é responsável por receber pacotes da rede e armazená-los na memória. Dessa forma, possui duas FSMs e um *buffer* de 16 *flits*, cujo tamanho é parametrizável no projeto [3].

  Suas máquinas de estados finitas estão mostradas na Figura 2.22 e funcionam da seguinte forma: primeiramente, o estado `Header` recebe o pacote da NoC e gera uma interrupção no processador via *software*, ao passo que armazena todos os *flits* recebidos no *buffer*. Além disso, a NoC para a transmissão caso o *buffer* fique cheio. Em seguida, o estado `Payload size` adquire o tamanho da carga útil do pacote, escreve no *buffer* e avança para o estado `Data`, no qual os demais *flits* são lidos para o *buffer* enquanto o tamanho de carga útil for maior que `'0'` [3]. Então, na segunda FSM, quando os registradores mapeados em memória `DMNI_START` e `DMNI_OP` recebem os valores `'1'` e `read`, respectivamente, o estado `Copy to mem` é habilitado a fazer a escrita dos dados do *buffer* para a memória local, de acordo com sinais de habilitação e desabilitação da escrita [3].

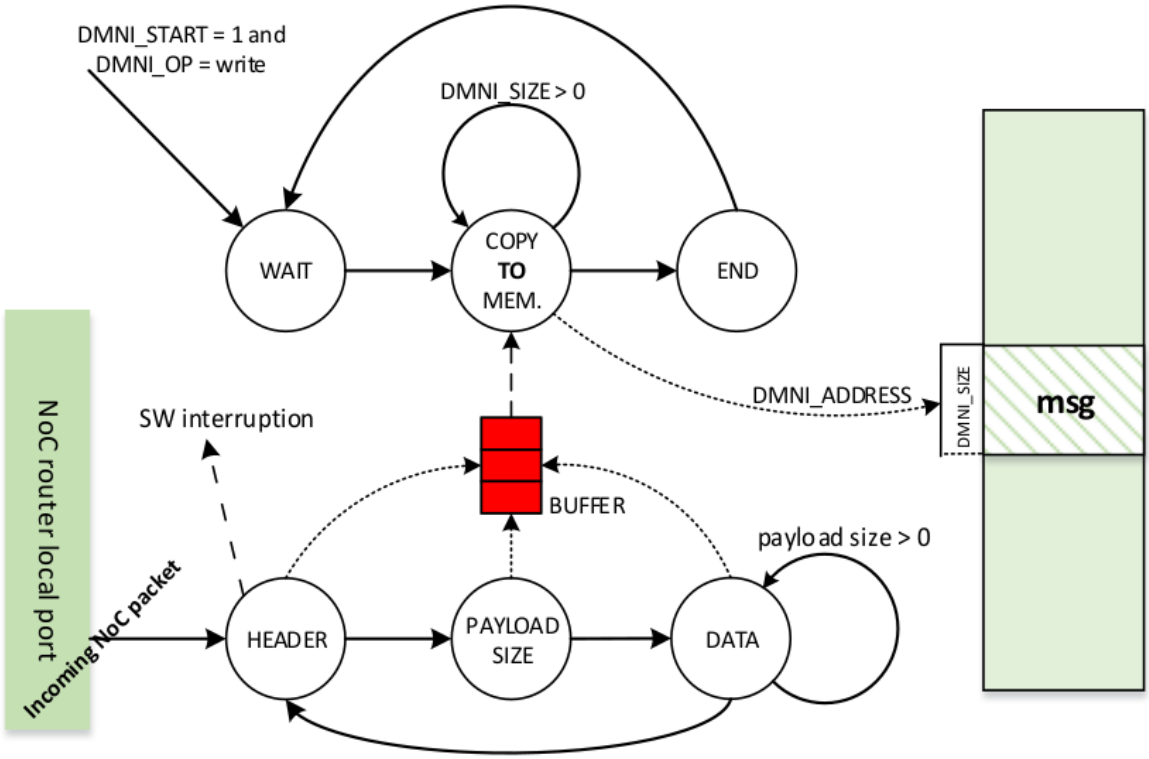

Figura 2.22: Máquinas de estados finitos do módulo de recebimento (Fonte: [3]).

- Módulo árbitro: a função desse módulo é a de garantir concorrência no acesso a memória para enviar e receber pacotes. Essa funcionalidade, é permitida devido a um algoritmo *Round-Robin* que controla dois sinais: o sinal de permissão de leitura (`read_enable`), para envio, e o sinal de permissão de escrita (`write_enable`), para recebimento. O tempo em que cada operação pode ser realizada é definida por um temporizador chamado `DMNI_TIMER` [3].

  A máquina de estados desse módulo é mostrada na Figura 2.23. Um sinal `round` decide qual módulo deve ter acesso à memória, o de recebimento ou o de envio. Se `round` tiver o valor `'1'`, assim como o sinal `send_active`, a máquina vai para o estado de envio. Nesse estado, o módulo de envio tem acesso garantido a memória pelo sinal `read_enable`, de onde consegue buscar as informações para enviar na NoC. A máquina permanece nesse estado enquanto o sinal de envio for ativo, ou até o tempo máximo do temporizador. Se `round` tiver o valor '0' e o sinal `receive_active` for '1', a máquina vai para estado de recebimento. O módulo de envio tem acesso garantido à memória pelo sinal `write_enable`, com o qual consegue armazenar na memória informações vindas da NoC. Analogamente ao estado de envio, a máquina continua nesse estado enquanto o sinal de recebimento for ativo, ou o tempo máximo do temporizador não for atingido [3].

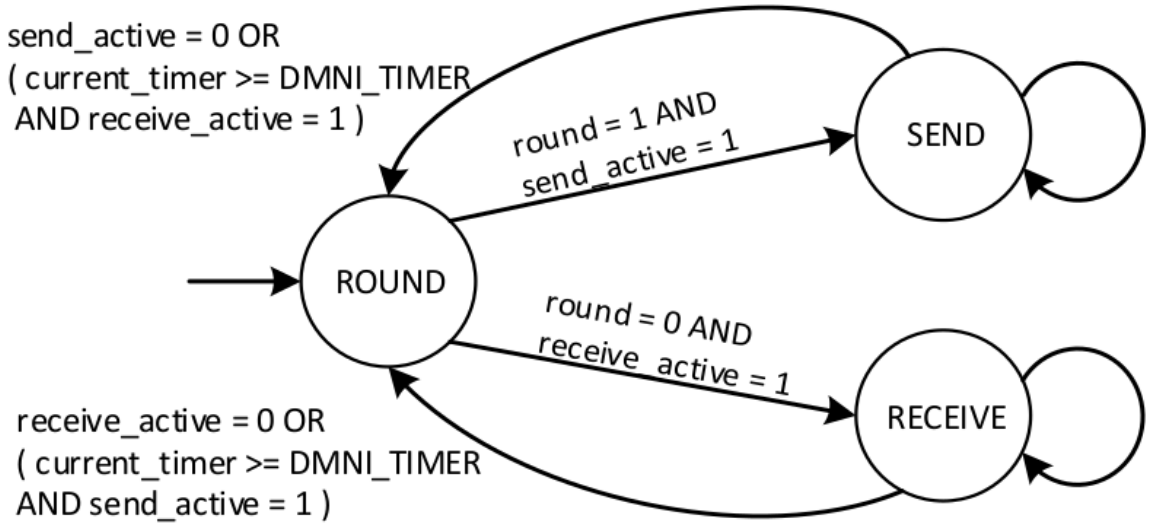

Figura 2.23: Máquina de estados finitos do módulo de recebimento (Fonte: [3]).

2.3.4 Modelagem de aplicações

Uma aplicação a ser executada no sistema é modelada como um grafo de tarefas [3]. Como exemplo, uma aplicação A é o grafo:

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

Nessa aplicação, T é o conjunto de tarefas presentes na aplicação:

$$T = \{t_1, t_2, ..., t_n\}$$

Já D, corresponde ao descritor da aplicação. Esse conjunto contém os pares de tarefas que se comunicam [3]. Na Figura 2.24 tem-se um exemplo de aplicação composta por 4 tarefas, tal que

$$T = \{t_A, t_B, t_C, t_D\}$$

e

$$D = \{t_A \rightarrow t_B, t_A \rightarrow t_C, t_B \rightarrow t_D, t_C \rightarrow t_D\}$$

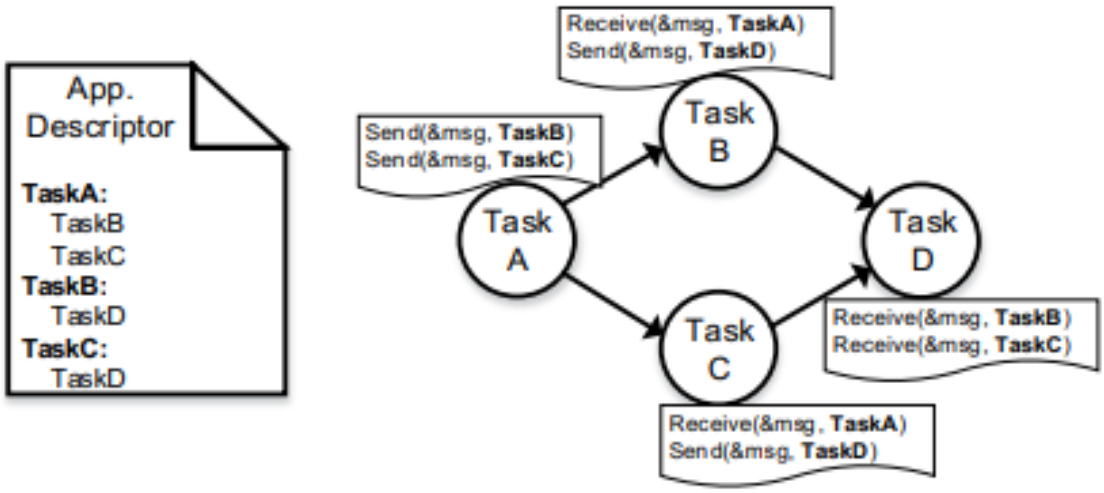

Figura 2.24: Exemplo do modelo de aplicações a serem executadas na HeMPS (Fonte: [3]).

Na simulação da plataforma HeMPS, o usuário não pode definir novas aplicações em tempo real. As requisições de novas aplicações são simuladas em um *testbench* de forma que as aplicações devem ser definidas em tempo de projeto.

2.3.5 Kernel (Sistema operacional distribuído)

O *kernel* presente na HeMPS implementa um gerenciamento distribuído, sendo que o *kernel* pode ser de dois tipos: escravos, presentes nos SPEs, ou mestres, nos MPEs [3]. Na Figura 2.25 pode-se observar a hierarquia do sistema.

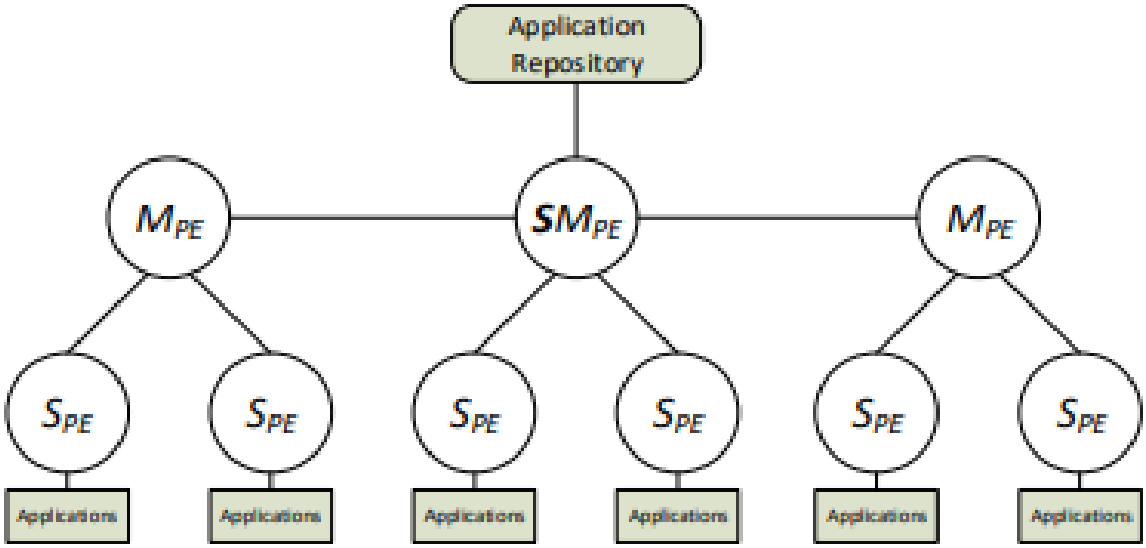

Figura 2.25: Hierarquia do *kernel* da HeMPS (Fonte: [3]).

Uma característica relevante presente no sistema é o tamanho dinâmico dos *clusters*. Ou seja, eles podem ter seus formatos modificados em tempo de execução [3]. Para isso, há nos MPEs um protocolo de reclusterização que realiza o redimensionamento se o número de tarefas for maior que o número de *clusters*. Esse redimensionamento é feito de forma que um *cluster* possa emprestar recursos para outro [3].

Outra característica relevante presente na HeMPS é a migração de tarefas. Essa técnica consiste em fazer o balanceamento da carga no MPSoC e desfragmentar o sistema, tendo, como consequência, redução de energia consumida durante comunicações e otimização do desempenho na execução de aplicações [24].

Abaixo, são detalhados alguns mecanismos e estruturas de gerenciamento usados pelo sistema. São eles: o gerenciamento de novas aplicações, a estrutura do bloco de controle das tarefas dessas aplicações, o gerenciamento das comunicações entre tarefas, as primitivas de comunicação (chamadas de sistema) e a gerência de paginação. Ao fim, é abordado ainda a estrutura dos programas de boot utilizados.

Gerenciamento de novas aplicações

Para gerenciar a inicialização de novas aplicações no sistema, os kernels SMPE, MPE e SPE implementam um protocolo de admissão de novas aplicações [3]. Dessa forma, no

momento que um nova aplicação faz uma requisição para executar no sistema, os seguintes passos são realizados, de acordo com a Figura 2.26:

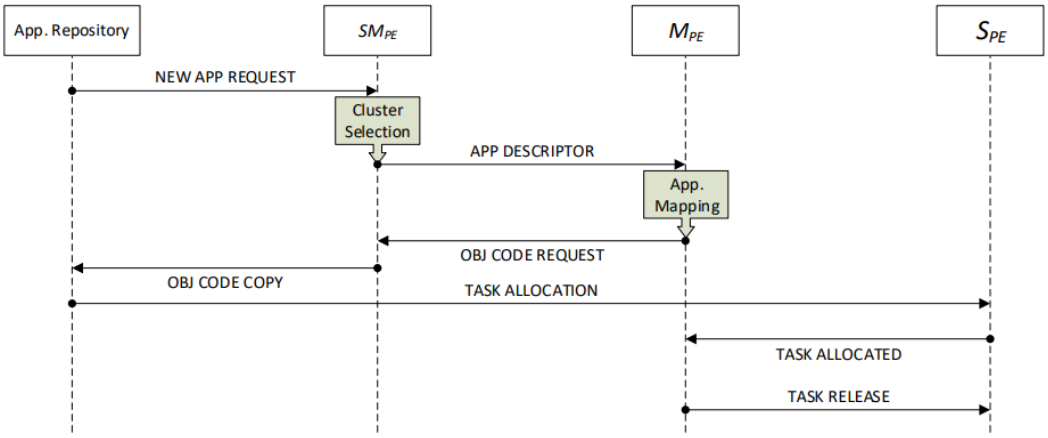

Figura 2.26: Diagrama do protocolo de admissão de novas aplicações (Fonte: [3]).

1. Uma interrupção é enviada do repositório de aplicações ao SMPE [3].
2. A interrupção é tratada de forma que o SMPE obtenha o descritor D e execute um algoritmo de mapeamento de clusters [3].
3. O algoritmo de mapeamento encontra o cluster mais apropriado para receber a aplicação, baseado em sua ocupação, e envia para seu MPE o descritor D [3].
4. O MPE, a partir do descritor, faz o mapeamento de tarefas nos seus SPEs e faz uma requisição ao SMPE pelos códigos objetos dessas tarefas [3].
5. O SMPE trata a requisição fazendo com que o repositório de aplicações transfira os códigos objetos solicitados aos SPEs [3].
6. O SPE então envia uma mensagem informando o sucesso, após o código da tarefa ser recebido e alocado [3].
7. O MPE recebe as respostas de sucesso de cada SPE seu com tarefas da aplicação. A partir desse momento, a aplicação é autorizada a ser executado pelos SPEs [3].

Controle das tarefas em execução

O contexto das tarefas designadas a um SPE para a execução é armazenado em uma estrutura que corresponde ao bloco de controle da tarefa, ou TCB (*Task Control Block*). Essa estrutura contém, como mostrado no Código-fonte 2.1, o armazenamento de 30

registradores. A arquitetura MIPS conta com 32 registradores, sendo que são salvos apenas os registradores de 2 a 31 nesses blocos de controle. Isso se dá por não ser necessário salvar os valores do registrador de constante zero e o registrador `$at`.

Código 2.1: Estrutura do Bloco de Controle de Tarefas

```
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;
```

Nessa estrutura, também é salvo o contador de programa (`pc`) atual da tarefa, o deslocamento (`offset`) da página de memória que contém a tarefa, seu identificador (`id`), os tamanhos em bytes das seções `.TEXT`, `.DATA` e `.BSS` do código *Assembly* gerado, um identificador de processador para realizar migrações dessa tarefa (`proc_to_migrate`), o endereço do mestre associado à essa tarefa (`master_address`) e um ponteiro para a estrutura de escalonamento em `scheduling_ptr`.

Presente na TCB, a estrutura de escalonamento, mostrada no Código-fonte 2.2, apresenta valores que são utilizados para escalonar tanto tarefas de melhor esforço (BE, *Best Effort*), quanto tarefas de tempo real (RT, *Real Time*). A tarefas BEs se caracterizam por serem tarefas sem tempos máximos de execução (*deadlines*) [3] ao passo que as tarefas RTs são tarefas que possuem *deadlines* [3].

Código 2.2: Estrutura usada no escalonamento de tarefas

```
1   typedef struct {
2
3           int             status;
4           unsigned int    execution_time;
5           unsigned int    period;
6           int             deadline;
7           unsigned int    ready_time;
8           int             remaining_exec_time;
9           unsigned int    slack_time;
```

```
10                  unsigned int    running_start_time;
11                  unsigned int    utilization;
12                  unsigned int    waiting_msg;
13
14                  unsigned int    tcb_ptr;
15          } Scheduling;
```

Entre os valores presentes nessa estrutura, destaca-se o estado da tarefa (*status*) que corresponde a um dos seguintes:

- FREE: corresponde a um espaço vazio, isto é, sem uma tarefa alocada;
- READY: uma tarefa pronta para ser executada;
- RUNNING: tarefa em execução;
- MIGRATING: corresponde às tarefas que estão passando pelo processo de migração;
- BLOCKED: tarefa bloqueada esperando o recebimento do pacote `TASK_RELEASE`;
- SLEEPING: tarefas que já executaram até o final e estão esperando pelo fim do período;
- WAITING: tarefa bloqueada esperando receber uma mensagem de outra tarefa. Para esse estado, usa-se a flag `waiting_msg` vista na Código-fonte 2.2.

Outros valores presentes na estrutura são tempo de execução (`execution_time`), período em ciclos de *clock* (`period`), *deadline* para aplicações RT e `ready_time` que corresponde a quantos ciclos de *clock* a tarefa levará para ficar no estado `READY`. O número de ciclos de *clock* restantes (`remaining_exec_time`) na execução é outro valor que só faz sentido para aplicações RT. O tempo de folga (`slack_time`) é a contagem em ciclos de *clock* que ocorre enquanto uma tarefa está ociosa. O tempo de quando a tarefa em execução iniciou é armazenado em `task_running_start_time`. Armazenam-se ainda a porcentagem de utilização da CPU por essa tarefa em `utilization`. Por fim, tem-se o ponteiro da TCB correspondente à essa tarefa em `tcb_ptr`.

Gerenciamento da comunicação entre tarefas

Os SPEs suportam a execução de múltiplas tarefas, interrupções, salvamento de contexto e possuem uma API de comunicação entre as tarefas. A API de comunicação implementa chamadas de sistema, que tem entre elas, as primitivas de envio (*Send*) e recebimento (*Receive*) [3]. A primitiva de recebimento é blocante e é chamada pelas tarefas consumidoras, que geram um pacote de requisição de mensagem (`MESSAGE_REQUEST`) para o PE que executa a tarefa produtora [3].

Cada *kernel* do SPE possui um *pipe*, que é uma fila de mensagens. Essas mensagens são armazenadas no *pipe* quando a tarefa produtora faz uma chamada à primitiva de envio. O *kernel* do PE que possui a tarefa produtora é quem o gerencia. Quando uma tarefa consumidora envia um `MESSAGE_REQUEST`, a produtora verifica se há mensagem no *pipe*. Se houver, esta é enviada para o consumidor por meio de um pacote `MESSAGE_DELIVERY`. Senão, o *kernel* armazena a requisição da consumidora e quando a produtora chamar a primitiva de envio, a mensagem é enviada para o consumidor [3]. A primitiva de envio, em contraste com a de recebimento, só pode ser bloqueada caso não haja mais espaço disponível para ela armazenar a mensagem no *pipe*. Uma operação de envio é mostrada na Figura 2.27 entre a tarefa A, produtora, e a tarefa B, consumidora.

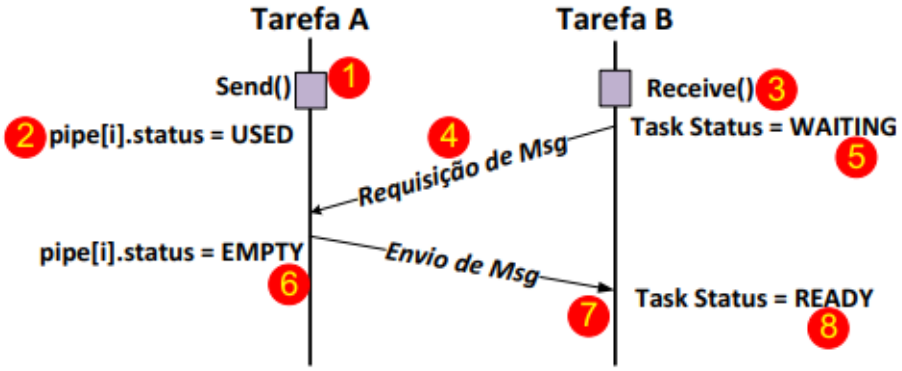

Figura 2.27: Protocolo de comunicação entre tarefas utilizado na HeMPS (Fonte: [20]).

1. Uma mensagem é enviada pela tarefa produtora (A) por meio da primitiva `Send()`;
2. Um espaço (*slot*) do *pipe* é ocupado pela mensagem enquanto a tarefa consumidora (B) não chama a primitiva de recebimento;
3. A tarefa B chama a primitiva de recebimento de mensagens `Receive()`;
4. Há o envio de uma requisição de recebimento de mensagem para a tarefa A;
5. A tarefa consumidora tem seu estado alterado para `WAITING`, no qual espera o recebimento ser finalizado;
6. A tarefa A libera o *slot* ocupado no *pipe*;
7. Há o envio de mensagem da tarefa A para a tarefa B;
8. A tarefa B recebe a mensagem e pode, enfim, voltar para o estado `READY`, no qual aguarda ser executada.

Chamadas de sistema

As chamadas de sistema, que têm como principal objetivo providenciar a infra-estrutura para o estabelecimento da comunicação [8], funcionam como interrupções de software. Uma aplicação utiliza essas chamadas por meio de primitivas. As chamadas e suas respectivas primitivas são detalhadas e mostradas abaixo;

1. EXIT: pela primitiva `exit()` encerra-se uma tarefa. Para isso, impede que a tarefa seja escalonada para a execução. Além disso, espera até que todas as mensagens de seu *pipe* tenham sido consumidas [8].
2. WRITEPIPE: `Send(Message mensagem, int id_destino)` é a primitiva utilizada na aplicação. Essa chamada envia uma mensagem para uma tarefa que tenha requisitado. Caso não haja requisições, armazena a mensagem no *pipe* [8].
3. READPIPE: por meio da primitiva `Receive(Message mensagem, int id_origem)` uma mensagem no *pipe* é buscada. Caso nenhuma seja encontrada, requisição de mensagem é enviada e a tarefa se bloqueia [8].
4. GETTICK: a primitiva `GetTick()` retorna a quantidade total de ciclos de *clock* executados até o momento [8].
5. ECHO: `Echo()` imprime uma *string* no arquivo de *log* da tarefa.
6. REALTIME: por meio da primitiva `RealTime()` definem-se parâmetros para a execução de aplicações de tempo real.

Gerenciamento de paginação

O kernel do SPE oferece, por meio de um mecanismo de paginação, suporte a execução multitarefa. Esse mecanismo divide a memória em páginas de tamanho fixo durante o *boot* do *kernel* [3]. Esse tamanho e o número de páginas são definidos no projeto [3].

A Figura 2.28 apresenta uma divisão lógica da memória, em que a página 0 armazena o kernel, enquanto que as restantes armazenam as tarefas das aplicações. Na parte (b) da Figura 2.28, tem-se a estrutura de cada página detalhadamente. A página é composta por seções de dados estáticos e dinâmicos. A parte estática, apresentada como `text`, armazena o código objeto da tarefa. Enquanto que a parte dinâmica apresenta dados inicializados como `data`, dados não inicializados como `bss` e a pilha de execução (`stack`). Essa estrutura é definida pelo compilador utilizado [3].

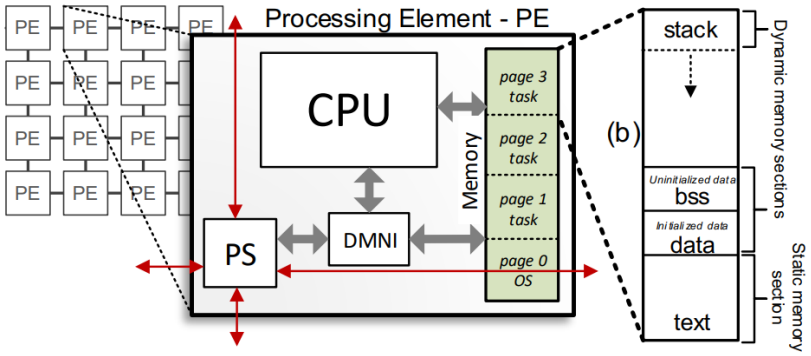

Figura 2.28: Organização da paginação de memória (Fonte: [3]).

Boot

Os dois tipos diferentes de *kernel* utilizados possuem programas de *boot* específicos. Além disso, aplicações que vão ser executadas no sistema necessitam ter suas tarefas carregadas nos PEs assim como o *kernel*. Por isso, há um programa de *boot* genérico que é usado com as tarefas.

Esses programas, que são escritos em linguagem de montagem, estão contidos nas primeiras posições de memória da página do *kernel* e servem para inicializar o sistema operacional [8]. Esses códigos podem possuir ainda rotinas que são necessárias em baixo nível, como por exemplo, o salvamento e a restauração do contexto durante interrupções e chamadas de sistema.

Enquanto o *kernel* mestre apresenta apenas a inicialização do ponteiro de pilha e a chamada da sua função `main` no *boot*, esse programa do *kernel* escravo possui outras funcionalidades além dessas. Nesse *boot*, estão presentes rotinas de tratamento de interrupções e chamadas de sistema. Já o *boot* de tarefas é usado para, além de inicializá-las, realizar a execução da instrução `syscall`. Essa instrução transfere o fluxo de execução do processador ao tratamento das chamadas de sistema quando essas são requisitadas.

2.4 Arquitetura RISC-V

Em um sistema computacional, o interfaceamento mais importante entre o hardware e o software é o conjunto de instruções (ISA) do processador. Ainda assim, as ISAs mais populares comercialmente são proprietárias [6]. A arquitetura RISC-V foi, ao ser criada, disponibilizada de forma gratuita e aberta de forma que ela impulsiona o potencial de inovação em sistemas microprocessados, reduz custos computacionais e facilita o desenvolvimento de dispositivos computacionais especializados [6].

Criada inicialmente com uma proposta educacional e de pesquisa, esse conjunto de instruções é suficientemente completo para ser utilizado em dispositivos de baixo custo e que exigem baixa demanda energética [6]. Além disso, partindo de um conjunto base de inteiros de 32 bits (RV32I), a arquitetura permite que sejam disponibilizadas arquiteturas de 64 e, até mesmo, 128 bits. Pode-se também adicionar extensões com diversas funcionalidades como multiplicação e divisão (M), ponto flutuante (F), ponto flutuante de precisão dupla (D), sincronização multi-processo (A) e um pacote completo delas juntas (G) [6].

2.4.1 Outras arquiteturas

A arquitetura baseada no conjunto de instruções RISC-V se diferencia de outras ISAs justamente por ser uma implementação de código aberto. ISAs como X86, ARM, Power, MIPS e SPARC são patenteadas e comercialmente protegidas [25]. Além disso, a possibilidade de usar um conjunto incremental permite maior flexibilidade de implementação se comparada com outras arquiteturas.

A Figura 2.29 apresenta uma tabela com características e funcionalidades de arquiteturas de instruções reduzidas. Por meio dessa tabela, percebe-se que a arquitetura RISC-V é comparável às demais citadas. Na comparação, usa-se o conjunto RISC-V mais completo de 64 bits, o RV64G. Nela são comparadas a data de lançamento do conjunto base, e a data do lançamento do conjunto em questão.

	ARMv8	**MIPS64 R6**	**Power v3.0**	**RV64G**	**SPARCv9**
Original date (base ISA)	1986	1986	1990	2016	1987
Date of this ISA	2011	2014	2013	2016	2008
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits (flat)	64 bits, flat	64 bits, flat	64 bits, flat	64 bits, flat
Data alignment	Aligned preferred	Aligned preferred	Unaligned	Aligned preferred	Aligned
Data addressing modes	8 (including scaled, pre/post increment)	1 (+1 for FP only)	4	1	2
Integer registers (number, model, size)	31 GPR x 64, plus stack pointer	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits
Separate floating-point registers	32x32 or 32x64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

Figura 2.29: Comparativo entre processadores de instruções reduzidas (Fonte: [6]).

Além disso, são mostrados ainda o tamanho em bits das instruções, o tamanho e o modelo do espaço de endereçamento, se há ou não alinhamento dos dados, a forma de endereçamento, a quantidade e o tamanho dos registradores presentes tanto para inteiros quanto para valores de ponto flutuante e o formato que é adotado para a implementação destes valores.

2.4.2 Processadores RISC-V

São avaliados, nessa etapa, processadores que executam instruções RISC-V e que possam ser compatíveis com o MPSoC estudado. Para isso, realiza-se a análise de processadores embarcados de características semelhantes.

Se faz necessário ainda, levar em conta testes realizados com a *toolchain* oficial do RISC-V, que utiliza a arquitetura Harvard separando segmentos de memória e dados, a fim de que seja gerado uma memória que seja o mais compatível com o *framework* utilizado.

Processador Potato

O processador Potato, cuja estrutura é mostrada na Figura 2.30, implementa uma versão simples de 5 estágios do conjunto de instruções inteiras de 32 bits da arquitetura RISC-V (RV32I [7]) [26]. Sua distribuição oferece ainda um simples SoC no qual é possível testar programas compilados a partir do C, usando a *toolchain* oficial [27] do RISC-V.

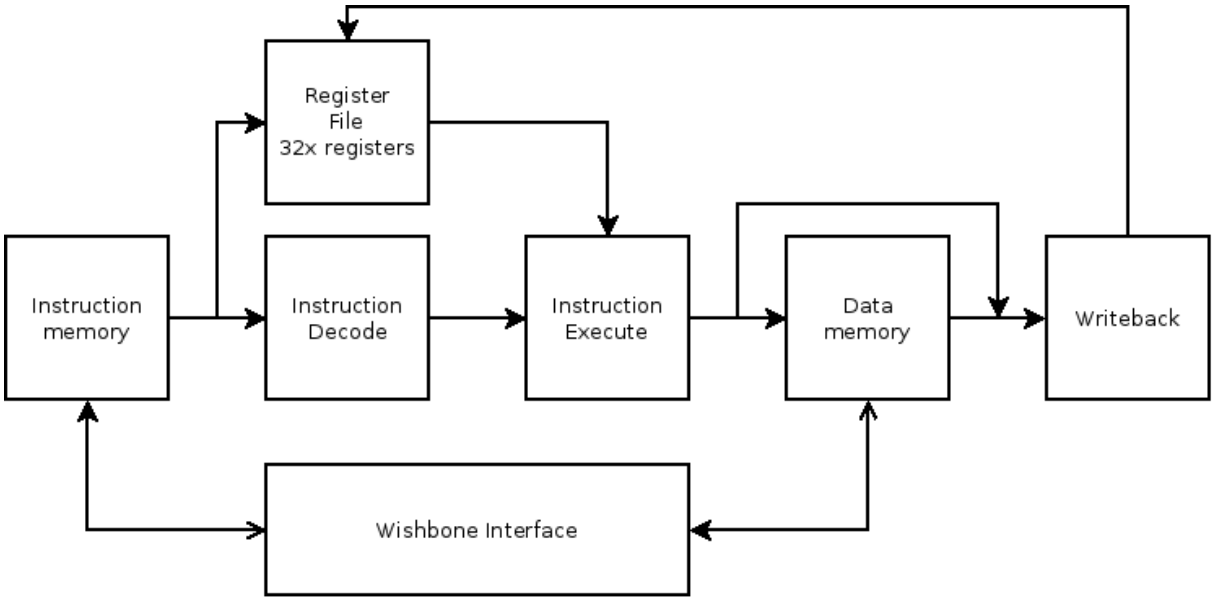

Figura 2.30: Diagrama de blocos do processador Potato e sua interface *wishbone* (Fonte: [26]).

Esse processador apresenta uma interface *wishbone* [28], geralmente usada para a comunicação entre os blocos IP de forma comum e flexível [28], para acessar as memórias de dados e instruções [26]. Além disso, por meio dessa interface é possível usar periféricos como UART, GPIO, memória externa e um temporizador que implementa interrupções por comparação. Além disso, disponibiliza 8 sinais IRQ, que podem ser usados para tratar interrupções individualmente [26].

Processador Boom

O processador Boom foi o mais completo entre os avaliados. Ele apresenta a implementação do subconjunto de instruções RV64G [29]. Esse processador é sintetizável, parametrizável, super-escalável e executa as instruções fora de ordem [30]. Foi um protótipo de base criado com código aberto pela universidade de Berkeley com intuito educacional, industrial e de pesquisa [30]. Escrito na linguagem SCALA, esse processador já foi testado executando aplicações de alta complexidade como o *benchmark* CoreMark [31], uma distribuição Linux e o *benchmark* SPEC CINT2006 [30].

Processador Clarvi

O processador Clarvi é uma implementação mais simples. Feito pela Universidade de Cambridge em SystemVerilog, esse processador executa de forma ordenada, em um *pipeline* de seis estágios, o conjunto de instruções RV32I [32], assim como o Potato. Nele pode-se ainda usar uma memória externa com dados e instruções, além de comunicação com dispositivos por meio de um barramento de entrada e saída mapeado em memória apesar de não oferecer suporte para memórias *cache* [32].

Por ter uma documentação e até mesmo exercícios, que auxiliam no entendimento ao gerar memórias a partir de programas *Assembly* e C, aliados a uma baixa complexidade, opta-se por usar esse processador para substituir o atual processador Plasma presente no HeMPS.

Logo, esse processador foi interfaceado no sistema conforme os passos descritos no próximo capítulo. Primeiramente, é feito um módulo de empacotamento para integrar o processador em SystemVerilog ao sistema em VHDL, depois são feitas modificações externas ao processador, que envolvem a geração dos MPSoCs com a nova arquitetura. Além disso, são descritas modificações nos *kernels* do sistema e no processador.

Capítulo 3

Integração de um novo processador na plataforma HeMPS

Neste capítulo são abordados detalhes da implementação proposta. Isto é, busca-se demonstrar os passos para se interfacear um novo processador no MPSoC proposto. Além disso, são descritos diversos problemas e soluções decorrentes da mudança de arquitetura MIPS para RISC-V.

A seção 3.1 apresenta a integração do processador Clarvi no elemento de processamento do MPSoC de referência.

Como já mencionado, um MPSoC é gerado a partir de parâmetros definidos em tempo de projeto. Dessa forma, a seção 3.2 trata das mudanças necessárias no processo de geração do MPSoC.

A seção 3.3 descreve as modificações necessárias no kernel que dependem da arquitetura usada.

A seção 3.4 trata de modificações que foram necessárias na implementação do novo processador escolhido.

3.1 Interface do processador de arquitetura RISC-V no Elemento de Processamento

O primeiro ponto a ser abordado no projeto, após a escolha do processador, é como será feita a interface entre o mesmo e o sistema gerado pelo HeMPS. Além das mudanças necessárias na estrutura do PE, a subseção 3.1.1 trata especificamente da interface entre o processador e a memória utilizada. Em 3.1.2 é abordada uma modificação no mecanismo de acesso ao repositório de tarefas.

A versão atual da HeMPS possui duas descrições de seus componentes de hardware. Uma descrição é feita em VHDL enquanto a outra é em linguagem SystemC. Nesse trabalho, é utilizada somente a versão descrita em linguagem VHDL. Nessa versão, está presente o processador Plasma que faz um interfaceamento direto do seu núcleo no elemento de processamento.

A conexão dos módulos no PE, para os MPSoCs gerados em VHDL, é vista na Figura 3.1. Os módulos interfaceados pelo PE são: UART, RAM, Plasma, DMNI e Router. O módulo UART serve para realizar escritas nos arquivos de *log*. O módulo de RAM possui a memória do PE, dividida em páginas. O processador Plasma faz a execução das instruções de arquitetura MIPS. O DMNI permite a interface entre memória e o roteador (*Router*). O roteador é utilizado para interconectar esse elemento de processamento com os demais.

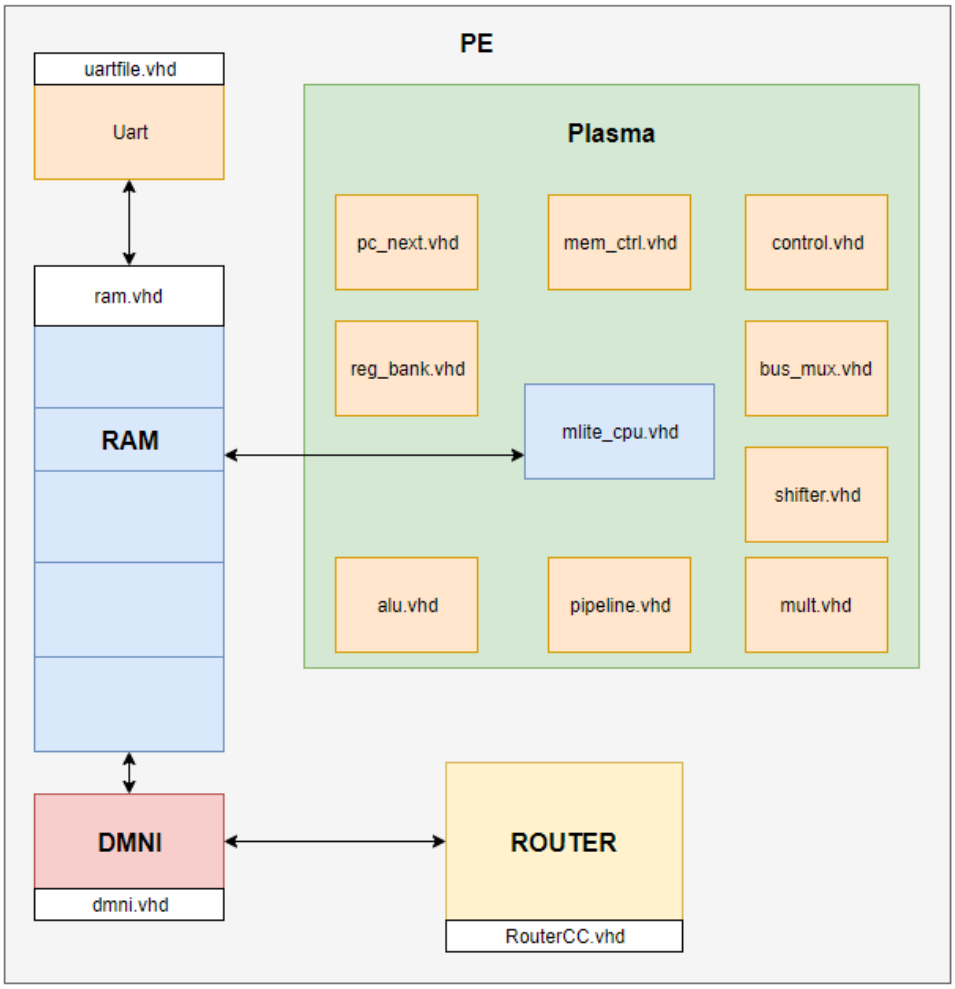

Figura 3.1: Estrutura atual do elemento de processamento presente nos MPSoCS gerados pela HeMPS.

A instância da entidade `mlite_cpu` interfaceada no PE é apresentada no Código-fonte 3.1. Nesse trecho, é visto que a instância recebe o sinal de *clock* (`clock_hold_s`) com o qual é possível realizar uma pausa no processador por completo em casos onde o processador não é necessário. Além disso, possui um sinal para reinício (`reset`), e sinais para interfaceamento com a memória. O primeiro sinal dessa interface é o `cpu_mem_address` pelo qual o processador retorna os endereços de dados ou instruções a serem lidos. O sinal `cpu_mem_data_write` possui um dado que deve ser escrito na memória enquanto que o `cpu_mem_data_read` tem um dado lido dela. O `cpu_mem_write_byte_enable` seleciona, no momento da escrita em memória, o tamanho da palavra a ser gravada e a sua posição. Sendo que para cada endereço, há 32 *bits* e, portanto, 4 *bytes*, uma escrita pode ocorrer em 1 *byte*, uma meia palavra (2 *bytes*) ou em 1 palavra (4 *bytes*). Já o sinal `cpu_mem_pause` serve para pausar as operações referentes à memória no processador. Geralmente essa pausa é habilitada durante acessos ao repositório feitos pelo PE. Por fim, a página que está carregada em memória é definida no processador e retornada ao PE pelo sinal `current_page`.

Código 3.1: Interfaceamento do processador Plasma no elemento de processamento

```
1        cpu : entity work.mlite_cpu
2                port map(
3                        clk          => clock_hold_s,
4                        reset_in     => reset,
5                        intr_in      => irq,
6                        mem_address  => cpu_mem_address,
7                        mem_data_w   => cpu_mem_data_write,
8                        mem_data_r   => cpu_mem_data_read,
9                        mem_byte_we  => cpu_mem_write_byte_enable,
10                       mem_pause    => cpu_mem_pause,
11                       current_page => current_page
12               );
```

No novo processador, que é descrito em SystemVerilog, a interface no elemento de processamento não é feita diretamente. Isto é, há um empacotador descrito em VHDL. Esse empacotador, além de repassar os sinais do elemento de processamento, é responsável ainda por paginar os endereços provenientes do Clarvi e deixá-los com tamanho apropriado para serem utilizados no PE do MPSoC.

Essa interface, mostrada no Código-fonte 3.2, recebe do PE tanto sinais auxiliares para a depuração, como o número da CPU (`cpu_number`) e o sinal de acompanhamento de instruções executadas (`trace`), quanto os sinais necessários ao funcionamento do processador. O processador se conecta à memória, que nessa arquitetura tem-se dados separados de instruções, com um sinal para o endereço de dados e outro para o de instruções que são

cpu_mem_data_address e cpu_mem_instr_address, respectivamente. Os sinais de leitura de dados e instruções são cpu_mem_data_read e cpu_mem_instr_read enquanto que os de escrita são cpu_mem_data_write e cpu_mem_instr_write. Entretanto, a memória de instruções nunca é modificada, e o seu sinal de escrita é sempre nulo (zero). Para definir os *bytes* de escrita cpu_mem_data_write_byte_enable e cpu_mem_instr_write_byte_enable, a interface retorna o *byte* de escrita dos dados caso o processador envie algo no sinal de escrita de dados, ou nulo, caso contrário. O *byte* de escrita do barramento de instruções é sempre nulo.

Código 3.2: Interfaceamento do empacotador do processador Clarvi no elemento de processamento

```
1 cpu: entity work.clarvi_wrapper
2         generic map (
3                 ADDR_WIDTH          => 29,
4                 PAGE_NUMBER_H_INDEX => PAGE_NUMBER_H_INDEX,
5                 PAGE_SIZE_H_INDEX   => PAGE_SIZE_H_INDEX
6         ) port map (
7                 cpu_number          => cpu_number,
8                 trace               => trace,
9                 clock               => clock,

10                reset               => reset,

11                interrupt           => irq,
12                interrupt_ret       => irq_ret,
13    -- Addresses
14                data_address        => cpu_mem_data_address,
15                instr_address       => cpu_mem_instr_address,
16
17    -- Data to be written
18                data_writedata      => cpu_mem_data_write,
19                instr_writedata     => cpu_mem_instr_write,
20
21    -- Data to be read
22                data_readdata       => cpu_mem_data_read,
23                instr_readdata      => cpu_mem_instr_read,
24                cpu_mem_pause       => cpu_mem_pause,
25
26    -- Write enables
27                wbe_data            => cpu_mem_data_write_byte_enable,

28                wbe_instr           => cpu_mem_instr_write_byte_enable,
29                page                => page_to_wrapper,
```

```
30                  ecall                => ecall
31 );
```

Os sinais de relógio (`clock`), reinício (`reset`), pausa de memória (`cpu_mem_pause`) e interrupção (`irq`, `irq_ret` e `ecall`) são os sinais interfaceados por esse módulo que servem para controlar e sincronizar o funcionamento do processador. Diferentemente, do Plasma, o sinal de relógio utilizado não realiza pausas. Para isso, usa-se o sinal de pausa de memória, que basicamente permite o processador continuar executando seus ciclos de acordo com o *clock* enquanto ele ignora as novas instruções adquiridas e invalida seus estágios do *pipeline*. Esse comportamento é necessário quando ocorre uma chamada de sistema. Essas chamadas são detalhadas na seção 3.4.2.

A paginação nessa versão é realizada por meio do elemento de processamento e do *kernel*. O interfaceamento se dá por meio do sinal `page_to_wrapper`, sendo que sua implementação é detalhada em 3.1.1, no tópico referente à paginação.

3.1.1 Interface do processador com a memória

O interfaceamento do processador no elemento de processamento levantou alguns desafios, sobretudo devido a diferença de arquitetura de memória dos processadores. Enquanto tem-se a arquitetura Von Neumann presente na implementação do Plasma, tem-se a arquitetura Harvard no Clarvi. Para lidar com essa diferença, foi necessário modificar a forma como o processador se conectava à RAM.

Adaptação da memória para nova arquitetura

Como o processador Plasma possui arquitetura de memória Von Neumann, uma memória de 2 portas no PE é suficiente. Dessa forma, 1 barramento é usado para dados e instruções e outro para acesso direto do DMNI. Em contrapartida, o processador Clarvi possui arquitetura de memória Harvard, isto é, são necessárias duas memórias, uma para dados e outra para instruções. No entanto, é possível usar uma memória de dois barramentos, ao separá-la em duas seções. No PE estudado, seriam necessários 2 barramentos na memória só para esse processador, sendo 1 para dados e outro para instruções. Logo, para integrá-lo à MPSoC, 2 soluções foram levadas em conta:

1. usar uma memória de três portas, sendo 2 barramentos disponibilizados ao processador e o outro, à DMNI.

2. usar uma memória de duas portas, sendo uma delas compartilhada entre o barramento de dados e a DMNI, como feito em [8];

Inicialmente, por simplicidade e para facilitar a depuração, optou-se por usar a memória com três portas. Mas tendo em mente que uma memória de 3 portas não é comum no mercado, após a validação de funcionamento do sistema, optou-se por realizar a multiplexação entre o sinal de memória do DMNI e o barramento de dados do processador em uma porta, conforme a Figura 3.2. Essa multiplexação é controlada pelos sinais que indicam se a DMNI está ativa, isto é, se esse módulo está enviando ou recebendo algo.

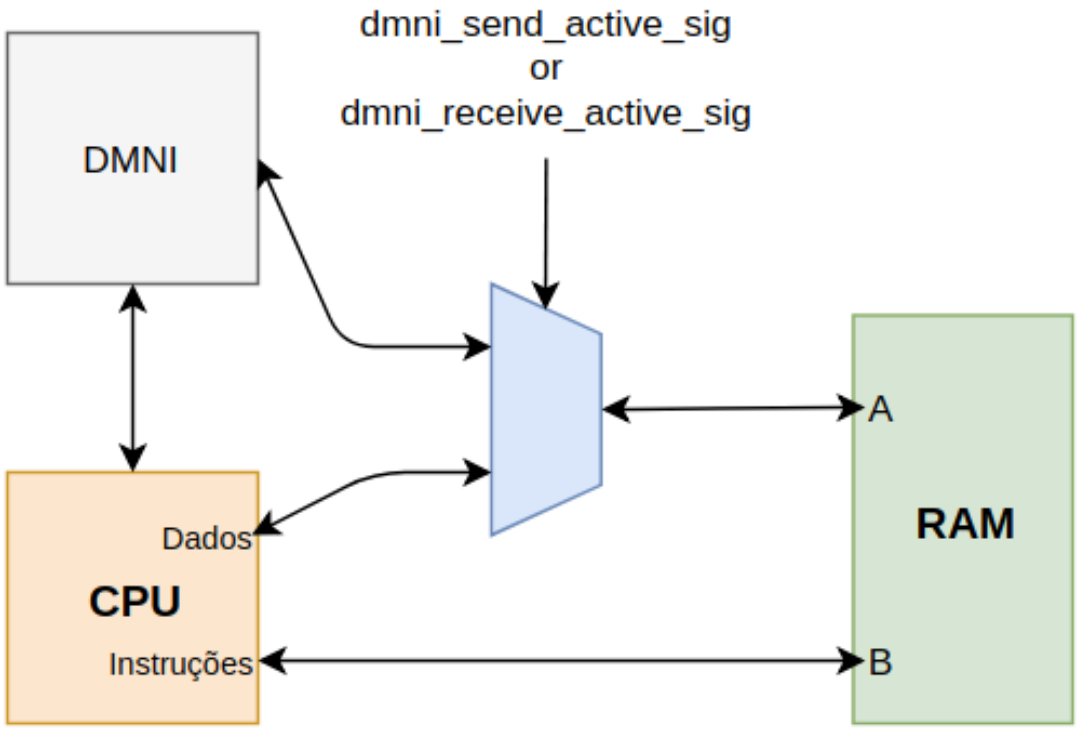

Figura 3.2: Multiplexação de barramentos de memória conforme o comportamento da DMNI.

Implementação de paginação no processador

A paginação é usada no sistema de forma que cada tarefa possua uma página. Assim, a página 0 é sempre reservada ao kernel e, por consequência, ela possui as rotinas de tratamento de interrupções, chamadas de sistema e escalonamento. O escalonador é chamado quando uma tarefa executou por um tempo limite ou quando as rotinas de tratamento assim decidem. Quando esse escalonador escolhe uma nova tarefa para a execução, uma troca de página é necessária.

A versão da HeMPS original implementa esse mecanismo de paginação diretamente no seu processador Plasma. Já na versão que foi modificada para utilizar o processador

Clarvi, a paginação é controlada pelo PE. A tradução de endereços virtuais para endereços em páginas ocorre no encapsulador do processador, o `clarvi_wrapper`.

No Plasma, o registrador $10 do coprocessador 0 é utilizado como registrador de página. Esse registrador armazena a que está em execução, ou seja, o valor do deslocamento (*offset*) de memória onde estão armazenados os dados e as instruções da tarefa corrente. Além disso, o próprio Plasma já realiza a conversão do endereço de memória virtual para o endereço real. Isto é, obtém o endereço na página da tarefa.

A página corrente é definida na rotina responsável por recuperar o contexto das tarefas, que é melhor abordada na seção 3.3.2. Essa rotina salva no registrador de página o *offset* de memória de onde está a tarefa. Esse *offset* está armazenado no TCB e, como pode ser visto no Código-fonte 3.3, é carregado no registrador $27 (`$k1`) nessa rotina. Por fim, antes da tarefa que teve seu contexto recuperado executar, a instrução `mtc0` é usada para transferir o valor do deslocamento contido em `$k1` ao registrador de página.

Código 3.3: Trecho de código no qual é realizada a restauração do deslocamento de página na versão com processador Plasma

```
1        lw      $26,120($27)
2        lw      $27,124($27)
3        li      $1,0x1
4
5        mtc0    $27,$10
6        jr      $26
7        mtc0    $1,$12
```

Internamente, quando o módulo do banco de registradores identifica que há uma escrita com destino ao registrador $10 do coprocessador 0 (indexado no processador com o valor `101010`), o valor que deveria ser escrito ali, é armazenado no sinal de página. Esse sinal é usado para paginar os endereços de memória e sua atribuição pode ser vista no Código-fonte 3.4. Nesse trecho, tem-se o valor que iria para o registrador `101010` sendo armazenado no sinal `page_reg`. Nos casos em que há interrupção ou chamada de sistema, tem-se o valor 0 sendo armazenado nesse sinal.

Código 3.4: Trecho de código em que se tem a atribuição da página no sinal `page_reg`

```
1    --sets page register
2    elsif rd_index = "101010" then
3     page_reg <= reg_dest_new;
4    end if;
5
6    --resets page on interrupts
7    if rs_index = "111110" or  rs_index = "111111" then
```

```
8     page_reg <= (others => '0');
9     end if;
```

No Código-fonte 3.5 tem-se o uso do registrador de página para gerar o endereço de memória paginado (mem_address).

Código 3.5: Trecho de código no qual se tem o registrador page sendo usado para paginar o endereço de memória

```
1 mem_address <= mem_address_wop(31 downto PAGE_NUMBER_H_INDEX+1) &
2         page(PAGE_NUMBER_H_INDEX downto PAGE_SIZE_H_INDEX+1) &
3         mem_address_wop(PAGE_SIZE_H_INDEX downto 2) &
4         "00" when page /= ZERO else
5         mem_address_wop & "00";
6
7         current_page <=
8             page(PAGE_SIZE_H_INDEX+8 downto PAGE_SIZE_H_INDEX+1);
```

No Código-fonte 3.5, o endereço sem paginação mem_address_wop tem seus bits de PAGE_NUMBER_H_INDEX a PAGE_SIZE_H_INDEX + 1 substituídos pelo valor da página corrente. No registrador page, essas mesmas posições correspondem ao valor dessa página. A obtenção dessa página é vista na Figura 3.3, de forma que os valores PAGE_NUMBER_H_INDEX e PAGE_SIZE_H_INDEX são definidos em tempo de geração do MPSoC e correspondem à posição mais significativa da página no endereço e ao seu tamanho, respectivamente.

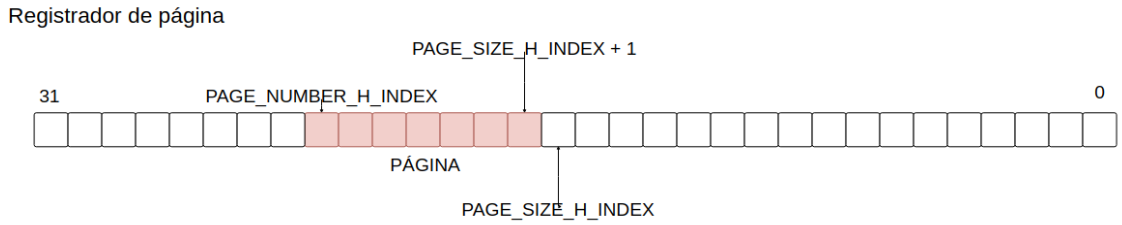

Figura 3.3: Extração de página corrente a partir do sinal page.

Na versão do *framework* que utiliza o processador Clarvi, optou-se por modificar minimamente o processador. Por isso, foi necessário implementar um mecanismo de paginação via software ao usar o registrador mapeado em memória NEXT_PAGE, que armazena a próxima página. Esse mecanismo funciona de forma similar ao que usa o registrador $10 do coprocessador 0 na versão original com processador Plasma.

Dessa forma, não há informação de página corrente no endereço provido pelo Clarvi. Gerar e retornar os endereços de dados e instruções com informações da página corrente ao PE é tarefa do módulo de empacotamento. Esse processo é observado no Código-fonte 3.6.

Código 3.6: Trecho do módulo de empacotamento do processador Clarvi que faz a paginação dos endereços de instruções e dados

```
1 instr_address <=
2     instr_address_full(31 downto PAGE_NUMBER_H_INDEX+1) &
3     page(PAGE_NUMBER_H_INDEX downto PAGE_SIZE_H_INDEX+1) &
4     instr_address_full(PAGE_SIZE_H_INDEX downto 2) &
5         "00" when page /= ZERO else instr_address_full;
6
7 data_address <=
8     data_address_full(31 downto PAGE_NUMBER_H_INDEX+1) &
9         page(PAGE_NUMBER_H_INDEX downto PAGE_SIZE_H_INDEX+1) &
10        data_address_full(PAGE_SIZE_H_INDEX downto 2) &
11    "00" when page /= ZERO else data_address_full;
```

Para isso, esse módulo recebe o sinal `page`, proveniente do elemento de processamento, que contém o valor da página atual. No entanto, essa deve ser extraída com o auxílio dos parâmetros `PAGE_NUMBER_H_INDEX` e `PAGE_SIZE_H_INDEX`, que são passados à esse módulo, da mesma forma que foi mostrada na Figura 3.3.

O valor desse registrador `page` atribuído no PE é obtido do registrador mapeado em memória `NEXT_PAGE`. No Código-fonte 3.7, é mostrado um trecho de código do início da rotina de restauração de contexto. Essa rotina, que vai ser melhor detalhada em 3.3.2, faz recuperação do offset de memória presente no TCB da tarefa que foi escolhida pelo escalonador. Esse valor obtido é, então, escrito no registrador de próxima página, que tem endereço `0x20000190`.

Código 3.7: Trecho de código da rotina de restauração de contexto que atribui o valor do offset da tarefa ao registrador mapeado em memória `NEXT_PAGE`

```
1     lw a1, 124(a7)
2     li a2, 0x20000190
3     sw a1, 0(a2) #NEXT_PAGE
```

Somente quando a instrução `mret` (`0x30200073`) é executada por essa rotina, é que o PE vai autorizar a mudança de página. Dessa forma, a página corrente é alterada imediatamente para a que está no registrador `NEXT_PAGE`, com o auxílio do parâmetro de tamanho de página como pode ser visto no Código-fonte 3.8.

Código 3.8: Trecho de código do PE onde há mudanças de página

```
1 if irq_ret = '1' or ecall = '1' then
2     current_page <= "00000000";
3 end if;
4
5 -- Clarvi mod: get NEXT_PAGE and store it in a register
6 if cpu_mem_data_address_reg = NEXT_PAGE and write_enable = '1' then
7     next_page_reg <= cpu_mem_data_write_reg;
8 end if;
9
10 -- Clarvi mod: whem mret is executed, set change_page flag
11 if cpu_mem_instr_read = x"30200073" then
12     change_page <= '1';
13 end if;
14
15 -- When change_page flag is set, change current page
16 if change_page = '1' then
17     change_page <= '0';
18     current_page <= next_page_reg(PAGE_SIZE_H_INDEX+8 downto
       PAGE_SIZE_H_INDEX+1);
19 end if;
20
21 if cpu_mem_data_address_reg = IRQ_MASK and write_enable = '1' then
22     irq_mask_reg <= cpu_mem_data_write_reg(7 downto 0);
23 end if;
```

3.1.2 Mecanismo de acesso ao repositório de tarefas

De acordo com o protocolo para gerenciamento de novas aplicações no sistema, a introdução de novas aplicações é controlada por dois sinais: o `req_app` e o `ack_app`. O Código-fonte 3.9 apresenta um trecho do *testbench* do sistema. Esse módulo de *testbench* simula requisições de novas aplicações feitas por um usuário por meio do `req_app`. Quando há uma requisição de entrada de uma nova aplicação no sistema, o bit mais significativo desse sinal é ativado em 1, por meio da máscara `0x80000000`. Quando o SMPE reconhece que essa requisição foi recebida, ele ativa o `ack_app`.

Código 3.9: Trecho do testbench do sistema. Esse trecho, mostra a etapa onde são adquiridas as posições de memória iniciais das aplicações no repositório

```
1
2 process (clock, reset)
3
4         variable app_repo_address  : integer := 0;
5         variable app_start_time_ms : integer := 0;
```

```
 6
 7
 8 begin
 9
10
11         if reset = '1' then
12                 req_app <= (others => '0');
13                 app_i <= 0;
14                 current_time <= 0;
15
16         elsif rising_edge(clock) then
17
18                 current_time <= current_time + 1;
19
20                 if req_app = x"00000000" then
21                         if appstart(app_i) /= x"deadc0de" then
22
23                                 app_repo_address :=
24                                         conv_integer(appstart(app_i));
25
26                                 app_start_time_ms :=
27                                         conv_integer(appstart(app_i+1));
28
29                                 dev <= 1;
30
31                                 if (app_start_time_ms * 100000) <=
32                                         current_time then
33                                         dev <= 3;
34                                         req_app <= CONV_STD_LOGIC_VECTOR(
    app_repo_address, 32) or x"80000000";
35
36                                         assert false report "Repository␣
    requesting␣app␣"& integer'image(app_i/2)
37                                         severity note;
38
39                                         app_i <= app_i+2;
40
41                                 end if;
42                         end if;
43                 elsif ack_app = '1' then
44                         dev <= 2;
45                         req_app <= (others => '0');
46                 end if;
47
48         end if;
```

```
49
50 end process;
```

Quando o sinal de requisição req_app é nulo, ocorre a leitura de uma linha do arquivo "appstart.txt". Esse arquivo contém os endereços de memória do início de cada aplicação presente no repositório de aplicações. Sua última linha é sempre composta do valor, em hexadecimal, 0xdeadc0de. Quando esse valor é lido, não haverá mais requisições de novas aplicações.

Após a leitura de uma linha do "appstart.txt", quando o valor do sinal de requisição não for mais nulo, essa etapa aguarda o recebimento do sinal de aviso ack_app, não conseguindo mais ler nada do arquivo enquanto isso. Dessa forma, enquanto houver uma requisição, outras não podem ocorrer até que o PE mestre do sistema envie esse sinal de aviso. Isso se dá quando esse PE recebe uma requisição e conclui o seu tratamento. Essa conclusão ocorre quando ele lê ou escreve no registrador mapeado em memória ACK_APP_REG, fazendo com que seu endereço seja carregado conforme o Código-fonte 3.10.

Código 3.10: Trecho do elemento de processamento que mostra o momento que o SMPE avisa o repositório de aplicações que sua solicitação foi tratada

```
1     if cpu_mem_address_reg = ACK_APP_REG then
2         ack_app <= '1';
3     elsif req_app(31) = '0' then
4         ack_app <= '0';
5     end if;
```

Contudo, na versão que usa como processador o Clarvi, notou-se um problema durante a requisição de novas aplicações. O problema ocorre pois essa versão possui dois barramentos de memória distintos. Então, o endereço do barramento de dados irá permanecer no valor do endereço do registrador ACK_APP_REG. Dessa forma, o sinal ack_app permanecerá em 1 forçando para o sinal req_app o valor 0x00000000, como visto no Código-fonte 3.9, a cada ciclo de *clock* enquanto o endereço de dados não for alterado.

Na versão com o processador Plasma ocorre o mostrado na região A da Figura 3.4. Quando o sinal req_app estava com o valor 0x00000000, houve uma requisição de nova aplicação e o valor 0x00007A0C foi lido do arquivo "appstart.txt" e teve seu bit mais significativo setado, de forma que o endereço resultante foi o 0x80007A0C. Seu valor se manteve, então, constante nos próximos ciclos de *clock*. Já que há apenas um barramento de memória para dados e instruções nessa versão, a próxima instrução foi buscada nele, fazendo com que o ack_app fosse desativado. Já com a versão do processador Clarvi, mostrada na Figura 3.4 em B, tem-se a leitura do endereço 0x00005F44. No entanto,

no próximo ciclo de *clock*, o `req_app` retorna a `0x00000000`, pois o endereço de busca de dados na memória não foi alterado. Dessa forma, a tarefa com início em `0x00005F44` será impedida de ser lida.

Figura 3.4: Comparação de recebimento de requisições nas duas versões. Em A, tem-se um caso de sucesso. Na região B, tem-se um caso de falha, pois o valor de requisição acabou em 0 quando não devia.

Para solucionar esse erro, o PE foi modificado de forma que uma *flag* `invalidate_ack` fosse setada imediatamente quando o sinal de *ack* vai para zero. Assim, esse valor impede que o `ack_app` volte para 1 enquanto não há mudança no endereço de dados. Esse procedimento é mostrado no Código-fonte 3.11.

Código 3.11: Modificação na forma de ativação do sinal `ack_app` do PE que contém o processador Clarvi

```
1  -- modified: Clarvi
2  if cpu_mem_data_address_reg = ACK_APP_REG and invalidate_ack = '0' then
3      ack_app <= '1';
4  end if;
5  if req_app(31) = '0' then
6      ack_app <= '0';
7      invalidate_ack <= '1';
8  end if;
9
10 -- Asserts that: if invalidate flag is set when it shouldn't, it will
       come to the right state...
11 if cpu_mem_data_address_reg /= ACK_APP_REG then
12     invalidate_ack <= '0';
13 end if;
```

3.2 Adaptação da geração na plataforma HeMPS para arquitetura RISC-V

Nessa seção, busca-se abordar o passo-a-passo da geração do MPSoC estudado. Além disso, abordam-se eventuais modificações que foram necessárias nessa etapa para a geração do código objeto, antes gerado para arquitetura MIPS, a ser executado na arquitetura RISC-V. Para gerar um MPSoC por meio da plataforma HeMPS, deve-se escolher entre as aplicações disponíveis ou criar uma própria para ser executada no sistema. A partir dessa aplicação e das configurações definidas pelo usuário em um arquivo YAML (*YAML Ain't Markup Language*), processo que será melhor detalhado mais a frente, o sistema gera um MPSoC customizado.

A Figura 3.5 ilustra a sequência do passo-a-passo de geração, em seguida, os passos e os sub-passos desse processo são detalhados nas subseções seguintes.

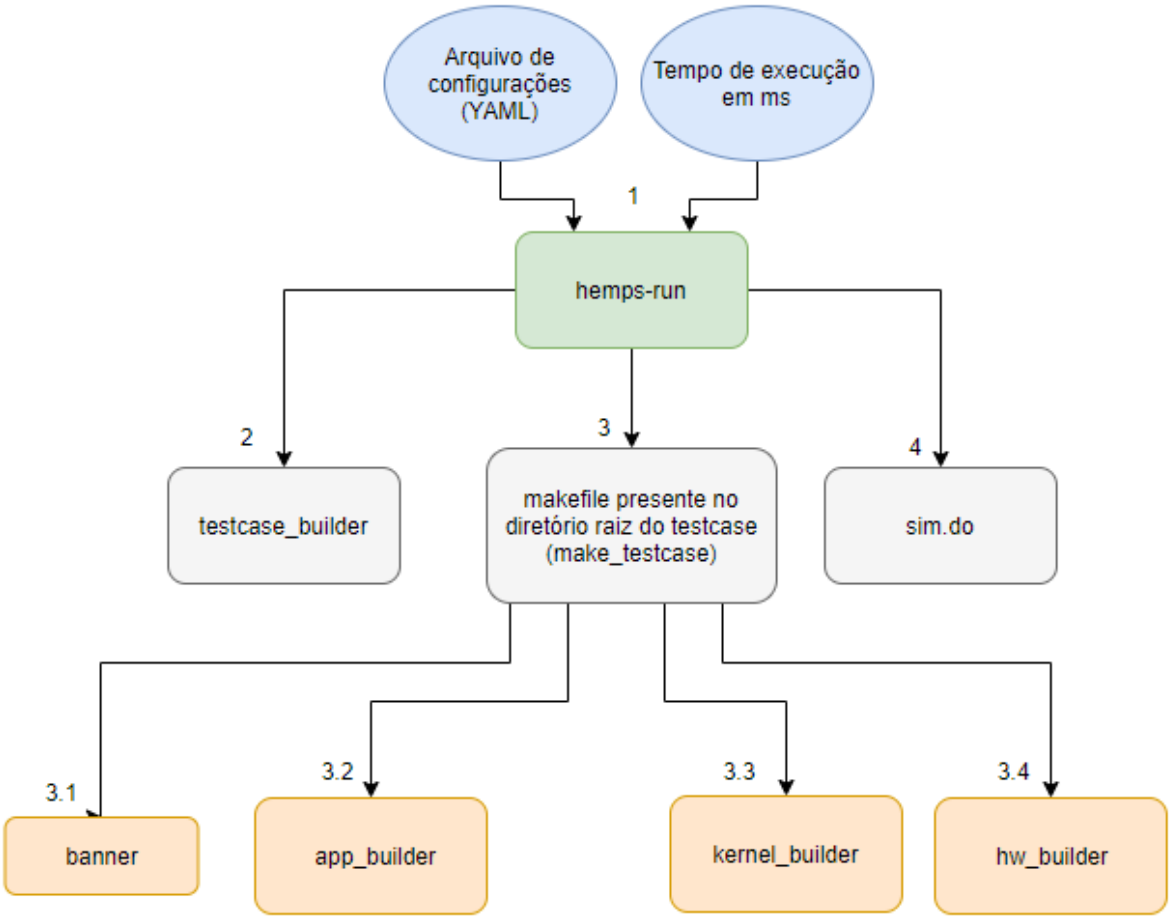

Figura 3.5: Passo-a-passo de geração da plataforma HeMPS.

A subseção 3.2.1 aborda o *script* `hemps-run`, que deve ser executado pelo usuário, e detalhes do arquivo de configuração YAML do MPSoC. Na subseção 3.2.2, o *script* `testcase_builder` é apresentado. Em 3.3.3, são detalhadas etapas relacionadas à compilação das aplicações e dos sistemas operacionais. Na última subseção, 3.3.4, os *scripts* de simulação no software Modelsim da Mentor Graphics [33] são abordados.

3.2.1 Execução do script hemps-run e carregamento do arquivo de configuração YAML

O *script* `hemps-run` é utilizado pelo usuário para a geração e a execução de um MPSoC tendo como parâmetros um arquivo YAML com configurações desse sistema e o tempo de execução em milissegundos do caso de teste a ser executado. Como exemplo, pode-se executar o comando "hemps-run hello_world.yaml 15", em que "hello_world.yaml" é o arquivo de configuração, enquanto que "15" é o tempo de execução.

Os arquivos de configurações devem ter suas estruturas divididas em duas seções: `hw` e `app`. A primeira seção, `hw`, possui as configurações usadas para parametrizar o hardware do MPSoC. Já a seção `app`, lista as aplicações que serão executadas no sistema e define, para cada uma, um parâmetro de início da execução.

O Código-fonte 3.12 mostra, como exemplo, o arquivo de configuração "hello_world.yaml" contendo 5 instâncias da aplicação `hello_world_app` e parâmetros definidos pelo usuário para a geração do sistema.

Código 3.12: Exemplo de script de configuração YAML utilizado no framework HeMPS

```
1 hw:
2   page_size_KB: 64
3   tasks_per_PE: 1
4   repository_size_MB: 1
5   model_description: vhdl
6   noc_buffer_size: 8
7   mpsoc_dimension: [2,2]
8   cluster_dimension: [2,2]
9   master_location: LB
10
11 apps:
12   - name: hello_world_app
13     start_time_ms: 5
14   - name: hello_world_app
15     start_time_ms: 5
16   - name: hello_world_app
17     start_time_ms: 5
18   - name: hello_world_app
```

```
19      start_time_ms: 5
20    - name: hello_world_app
21      start_time_ms: 5
```

Nesse arquivo de configurações, o hardware pode ser personalizado com as seguintes opções:

- `page_size_KB`: tamanho das páginas em quilobyte;
- `tasks_per_PE`: número de tarefas que um elemento de processamento pode executar [34];
- `repository_size_MB`: tamanho do repositório de tarefas em megabytes;
- `model_description`: linguagem de descrição usada para executar o sistema sendo: `sc` para SystemC e `vhdl` para VDHL, quando usando a plataforma com arquitetura MIPS, ou VHDL e System Verilog, quando usando a arquitetura RISC-V. Nesse estudo, são gerados apenas MPSoCs na linguagem de descrição VHDL, não sendo oferecido suporte à gerações em SystemC;
- `noc_buffer_size`: tamanho dos *buffers* utilizados nos roteadores da NoC [34];
- `mpsoc_dimension`: dimensão da MPSoC gerada sendo que o mínimo é a 2×2 [34];
- `cluster_dimension`: dimensão de cada *cluster* do MPSoC sendo que o mínimo é a dimensão do do próprio MPSoC, de forma que seja divisível pelo número de processadores do MPSoC [34];
- `master_location`: localização do mestre global da MPSoC. Atualmente apenas no canto inferior esquerdo (LB, *Left Bottom*) é suportado [34].

As informações das aplicações a serem executadas pelo sistema gerado estão listadas na seção `apps`, não havendo suporte para a adição de novas aplicações em tempo de execução. Dessa forma, pode-se executar várias instâncias de uma mesma ou de diferentes aplicações, definidas em tempo de configuração, se respeitada a mesma estrutura mostrada no Código-fonte 3.12. As configurações usadas para instanciar uma aplicação são:

- `name`: nome do diretório contido em "aplications" que possui as tarefas a serem executadas;
- `start_time_ms`: tempo, que deve ser representados em milissegundos por `ms` ou em microssegundos por `um` [34], que o sistema deve aguardar antes de iniciar a execução dessa instância de aplicação.

A partir do nome do arquivo carregado, um diretório de mesmo nome será criado, para conter o MPSoC gerado. Essa etapa é realizada por um processo filho do `hemps-run` que executa o *script* `testcase_builder`. Ao fim deste *script*, o `hemps-run` é ainda responsável por compilar o caso de teste e fazer a execução do mesmo no simulador Modelsim, conforme é detalhado nos próximos passos.

Como nessa etapa nenhum código objeto ainda é gerado e não há procedimentos que dependem da arquitetura, não foi necessário realizar alterações nela.

3.2.2 Execução do script testcase_builder

A segunda etapa consiste na execução do *script* `testcase_builder` no diretório do caso de teste criado no primeiro passo. Primeiramente, esse *script* faz a análise estática (*parse*) de informações do arquivo YAML de configurações. Depois, copia e gera arquivos que são necessários para a execução do sistema nos diretórios alvos do caso de teste. Entre esses arquivos gerados, estão informações que serão usadas para ambientes de depurações futuras e o *script* para gerar formas de ondas na ferramenta Modelsim.

Diferentemente da etapa anterior, por gerar arquivos que são dependentes da arquitetura utilizada, foram necessárias modificações nessa etapa. Na versão da HeMPS com processador de arquitetura RISC-V, foi necessário alterar o valor do ponteiro de pilha (`sp`) inicial tanto para as páginas. Na versão original, esses valores são definidos no script de geração do arquivo `make` do *kernel*. O tamanho da pilha é definido de acordo com o tamanho em quilobytes da página menos 1. Dessa forma, o endereço ficaria desalinhado com a memória de palavras de 4 bytes causando erro na execução com o processador RISC-V. Então, na nova versão, foi necessário remover essa subtração `- 1`.

3.2.3 Compilação dos sistemas operacionais e das aplicações

Essa etapa acontece logo que os arquivos necessários já estão presentes no diretório do caso de teste. O processo `hemps-run` executa um *script* que tem origem no `make_testcase` do *framework*. Esse *script* executa as seguintes subetapas:

1. Mostrar no terminal que chamou o `hemps-run` o *banner* do *framework* conforme a Figura 3.6, o que não exige modificação no sistema com arquitetura RISC-V;

2. Executar o *script* `app_builder`. Este realiza o *parse* dos parâmetros de execução das aplicações e compila os casos de teste presentes no diretório "*applications*". Essa compilação se dá ao chamar os *scripts* correspondentes contidos nos subdiretórios de cada aplicação.

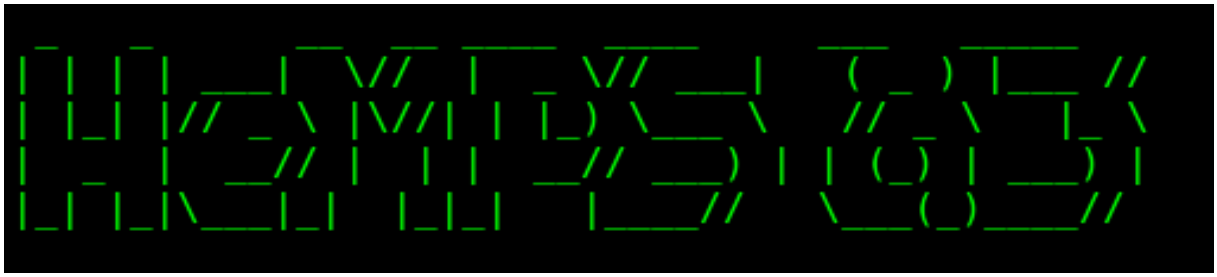

Figura 3.6: *Banner* mostrado no terminal no início da geração do MPSoC.

Os "makefiles" desses diretórios são responsáveis por compilar um programa de *boot* genérico para a aplicação, chamado "boot_task.asm". Esse programa tem o objetivo de inicializar o registrador `$sp` para cada tarefa presente na aplicação. Além disso, faz a chamada à rotina principal (`main`) da tarefa e possui, em sua área global, a rotina de tratamento de chamadas ao sistema (`SystemCall`).

Após essa primeira compilação ser feita, as tarefas são compiladas e têm seus códigos objetos ligados ao do `boot_task`. Por fim, têm suas instruções e dados extraídos no formato hexadecimal para gerar a página de memória da tarefa. Esse processo é ilustrado pela Figura 3.7.

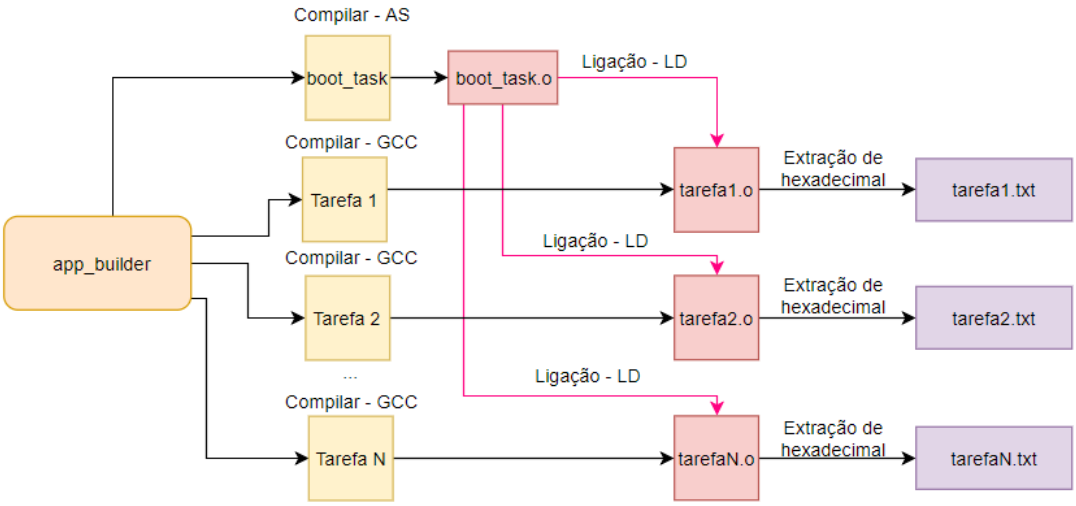

Figura 3.7: Processo de geração do código objeto das aplicações a serem executadas no MPSoC.

Essas compilações, ligações e, até mesmo a extração do código hexadecimal desses programas para a geração de memória, são realizadas pela cadeia de ferramentas (*toolchain*) `gcc-mips-elf`. Entre as ferramentas disponíveis, usa-se a ferramenta

`as` para a compilação do código *Assembly*. A ferramenta `gcc` é usada para a compilação dos programas escritos em C. Já para realizar a ligação entre os códigos objetos gerados, usa-se o ligador `ld`. Por fim, para extrair as informações dos códigos objetos em hexadecimal, usam-se as ferramentas `objdump` e `objcopy`. Para a versão que executa instruções RISC-V, foi necessário alterar essa *toolchain* para a "riscv-gnu-toolchain", que possui as ferramentas correspondentes. Outra modificação necessária foi a conversão do código *Assembly* do programa de *boot* MIPS para RISC-V.

3. Executar o script `kernel_builder`. Na sua execução, os arquivos `kernel_pkg.h` e `kernel_pkg.c` são gerados contendo informações dos *clusters*. Esses arquivos incluem também o mapeamento estático das tarefas a serem executadas. Em seguida, gera as páginas de memória dos sistemas operacionais presentes nos elementos de processamento.

   Por meio da chamada *make* compila, faz a ligação e extrai o código hexadecimal do *kernel* e seu respectivo arquivo de instruções de *boot*. Os códigos de *kernel* são compilados tanto para a versão de PEs mestres quanto para PEs escravos, conforme mostrado na Figura 3.8.

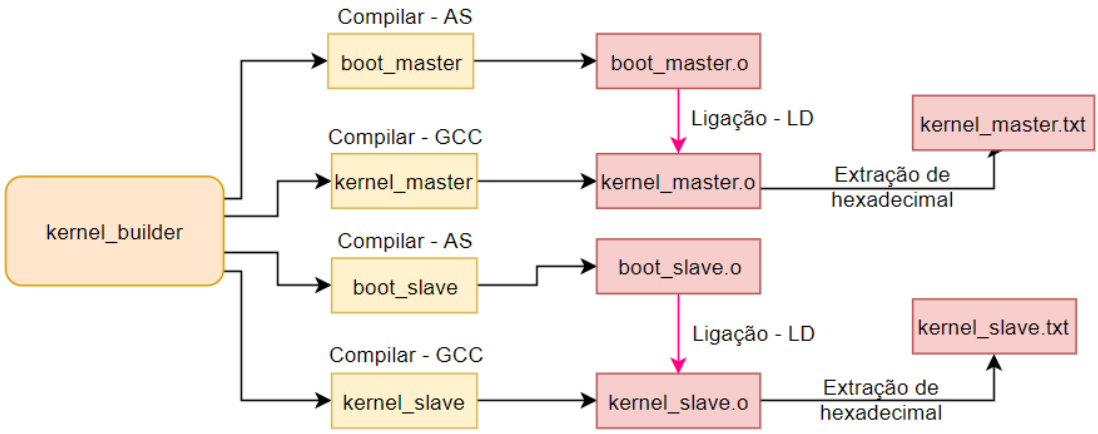

Figura 3.8: Processo de geração do código objeto dos sistemas operacionais executados pelos PEs do MPSoC.

No caso do elemento de processamento do tipo mestre, seu *kernel* implementa rotinas que possibilitam a comunicação via a NoC e disponibiliza rotinas do gerenciamento de execução das aplicações (para o caso do mestre global). Seu arquivo de *boot*

é responsável apenas por inicializar o registrador *$sp* e chamar a rotina principal (*main*).

Já no elemento de processamento escravo, o *kernel* implementa as rotinas de comunicação na NoC, escalonamento e tratamento de interrupções e chamadas de sistemas realizadas na aplicação. Seu *boot*, além de inicializar o registrador `$sp`, é responsável por realizar o salvamento e a restauração de contexto conforme necessário.

Nessa etapa, a *toolchain* de compilação, ligação e extração de código objeto em hexadecimal foi alterada da versão "gcc-mips-elf" para a "riscv-gnu-toolchain", assim como na etapa anterior. De mesma forma, também foi necessário converter os arquivos de *boot* para RISC-V. As conversões desses programas são melhor descritas quando são abordados salvamentos e recuperações de contexto, interrupções e chamadas de sistemas.

4. Executar o `hw_builder`. Esse *script* gera um arquivo com definições necessárias na execução da simulação do hardware. Entre essas definições, destacam-se tamanho de páginas, tamanho da memória, número de processadores, tamanho do *buffer* usado nos roteadores da NoC e o tipo de PE, para cada PE.

   Após gerar esse arquivo, o hardware é sintetizado usando o compilador `vcom` para o MPSoC em linguagem VHDL. Na versão com o processador Clarvi, por conter tanto código em VHDL quanto em System Verilog, foi necessário realizar a compilação em duas partes. Na primeira, faz-se a compilação de arquivos ".vhd" com o `vcom` e, na segunda, a compilação de arquivos ".sv" com a ferramenta `vlog`.

3.2.4 Execução de *scripts* para realizar a simulação na ferramenta Modelsim

Ao fim das etapas de compilação, o processo `hemps-run` chama o *script* de simulação. Esse *script* é responsável por definir as formas de onda que deverão ser mostradas por meio do arquivo `hemps-wave_gen` e executar a simulação do MPSoC gerado na ferramenta Modelsim.

Nessa etapa, não foi necessário alterar nada no *script* de simulação. No entanto, pelas diferenças encontradas no hardware do processador Clarvi, optou-se por modificar o *script* `hemps-wave_gen`. Essas modificações estão basicamente relacionadas à diferença do modelo de memória entre os processadores. O resultado dessa alteração é verificado na Figura 3.9, que mostra na esquerda as ondas geradas na versão com o processador Plasma e na direita, a versão com o Clarvi.

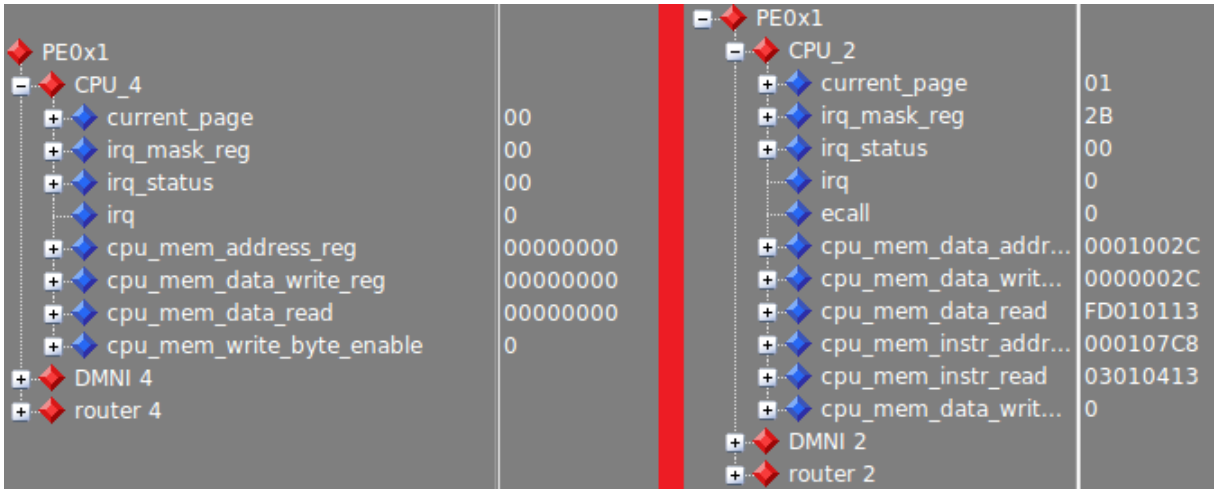

Figura 3.9: Comparação das ondas que são mostradas pela versão original da HeMPS na esquerda e pela versão modificada na direita.

3.3 Identificação e adaptação de rotinas do *kernel* dependentes da arquitetura

Para facilitar a integração de um novo processador na plataforma estudada, a ideia é reaproveitar ao máximo a estrutura do *kernel* original. Isto é, deve-se identificar rotinas presentes no *kernel* e no seu programa de *boot* correspondente que dependam diretamente da arquitetura e do processador escolhido. Geralmente, essas rotinas são as que lidam com interrupções, salvamento e restauração de contexto e tratamento dos pacotes de comunicação.

Na inicialização de um PE no sistema, o arquivo de *boot* correspondente atribui um valor inicial ao registrador do ponteiro de pilha `$sp` e chama a função principal do *kernel*. Nessa função, para o caso do PE ser do tipo escravo, as interrupções são desabilitadas, um TCB é criado para a tarefa de PE ocioso, `OS_Idle( )`, o *pipe* de comunicação e o vetor de mensagens requisitadas são zerados e os blocos de controle de tarefas são inicializados. Todas essas etapas exigem modificações relacionadas à arquitetura, mesmo que indiretamente.

Para utilizar interrupções na versão com RISC-V, foi necessário modificar a forma de habilitação e desabilitação e sua forma de tratamento. Essas modificações são abordadas na subseção 3.3.1. Já quanto à inicialização e a troca de tarefas em execução, são levados em conta os salvamentos e as recuperações de contexto abordadas em 3.3.2. Por fim, procedimentos relacionados à utilização de *pipes* e trocas de mensagens exigiram algumas modificações que são abordadas em 3.3.3.

3.3.1 Implementação de interrupções de máquina no processador

O processador Plasma oferece suporte à interrupções em sua implementação. Isto é, quando há uma interrupção, a execução é desviada para uma rotina de tratamento que está fixada no endereço `0x3C` da página que contém o *kernel*, que é a 0. Na versão com processador Clarvi, optou-se por manter a rotina de tratamento das interrupções também na posição `0x3C` da página do *kernel*. Além disso, o tratamento é feito de forma equivalente. A Figura 3.10 detalha o procedimento adotado quando há uma interrupção tanto na versão com o processador Clarvi quanto na original.

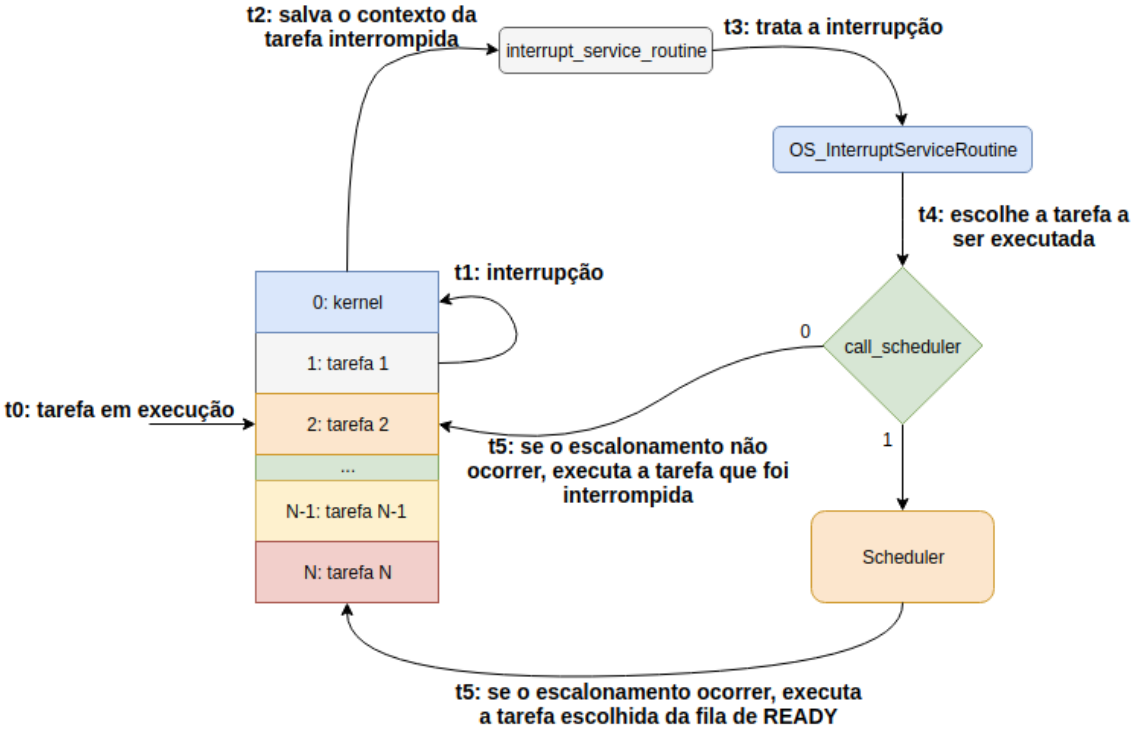

Figura 3.10: Fluxo de execução de uma interrupção no sistema.

No tempo `t0` da Figura 3.10, há uma tarefa na página 2 executando. Em `t1`, há a ocorrência de uma interrupção, o que leva à rotina de salvamento de contexto (desvio para a posição `0x3C` da página do *kernel*) em `t2`. Nessa rotina, o contexto e o contador de programa `$pc` são salvos no bloco de controle da tarefa que foi interrompida, conforme o que é detalhado na subseção 3.3.2. Em seguida, a interrupção é tratada em `t3`, de acordo com o valor armazenado no registrador mapeado em memória `IRQ_STATUS` pela função do *kernel* `OS_InterruptServiceRoutine`. O sistema de referência, HeMPS, suporta interrupções dos seguintes tipos:

- "IRQ_NOC": interrupção realizada pela NoC. Nessa interrupção, um pacote deve ser lido e tratado ou adicionado na fila de serviços pendentes, no caso da DMNI estar enviando algo e o pacote possuir o serviço de migração de tarefas ou de requisição de mensagens;
- "IRQ_PENDING_SERVICE": interrupção para que serviços pendentes na fila sejam tratados;
- "IRQ_SLACK_TIME": interrupção para enviar o pacote de monitoramento de tempo ocioso;
- "IRQ_SCHEDULER": interrupção para que seja chamado o escalonador após uma tarefa ter executado um tempo máximo pré-definido, a fim de se evitar starvation.

Ao fim do tratamento da interrupção, em `t4` na Figura 3.10, o sistema irá executar um possível escalonamento, caso a rotina de tratamento assim determine. No caso da rotina de escalonamento não ter sido chamada, em `t5` há a recuperação de contexto da tarefa que executava em `t1`. Caso o escalonamento tenha sido feito e escolhido uma nova tarefa, o contexto dela será recuperado em `t5`.

Contudo, na versão que executa com o processador Clarvi há diferenças em relação à versão original. Essas diferenças são: a adição de um sinal auxiliar para fazer o PE voltar à página 0, a forma de habilitar e desabilitar interrupções e como é definida a posição de desvio para a rotina de tratamento.

A adição do sinal `irq_ret` é a primeira diferença entre as interrupções na versão original e na nova. Esse sinal é conectado ao processador Clarvi e, no momento que esse vai realizar o desvio para tratar uma interrupção, seu valor é setado de forma que o PE mude a página atual para a página 0, que contém o *kernel*. Como o processador Plasma controla a paginação, esse mecanismo não é necessário, pois ele muda para a página 0 no momento que identifica uma interrupção.

Outro ponto da interrupção que deve ser levado em conta durante a mudança de arquitetura é a função que permite habilitá-las e desabilitá-las. No caso da versão que tem arquitetura MIPS, esse controle é realizado de acordo com o valor presente no registrador `status` ($12) do coprocessador 0. Sendo que o valor 0 as desabilita e 1 as habilita novamente.

A rotina `ASM_SetInterruptEnable`, que é chamada somente na inicialização da função `main` do *kernel* escravo, executa a instrução `mfc0`, armazenando o antigo valor do `status` no registrador `$v0` ($2). Em seguida, atribui o valor do parâmetro passado, que está em `$a0` ($4), ao registrador `status`, por meio da instrução `mtc0`. É importante notar que há uma instrução de salto `jr` para o registrador de retorno `$ra` ($31) antes dessa atribuição. Isso ocorre pois a implementação do processador do tipo *pipeline* de

arquitetura MIPS, é capaz de executar a próxima instrução no momento que realiza um salto. No Código-fonte 3.13 é possível ver a implementação dessa rotina na versão com o processador Plasma.

Código 3.13: Rotina para habilitar interrupções na versão original com processador Plasma

```
1          .globl  ASM_SetInterruptEnable
2
3 ASM_SetInterruptEnable:
4          .set    noreorder
5
6          mfc0    $2,$12
7          jr      $31
8          mtc0    $4,$12          #enable interrupts
9
10         .set    reorder
```

No Código-fonte 3.14, tem-se como essas interrupções são habilitadas na versão com o processador Clarvi.

Código 3.14: Rotina para habilitar interrupções na versão original com o processador Clarvi

```
1          .globl  ASM_SetInterruptEnable
2
3 ASM_SetInterruptEnable:
4          li t0, 0x80 #set mpie
5          csrw mstatus, t0 #0x300
6
7          li t0, 0x800 #set meie
8          csrw mie, t0 #0x304
9
10         li t0, 0x3c
11         csrw mtvec, t0 #0x305
12
13         jr ra
```

O Clarvi, por ter a arquitetura RISC-V, tem uma diferente forma de ser interrompido. Logo, nessa rotina, que somente é chamada uma vez na inicialização, ele deve definir os valores de três registradores de controle e estado (CSR, *Control and Status Registers*), ditos privilegiados, para que possa tratar as interrupções corretamente. Esses registradores só podem ser manipulados pelas instruções do modo privilegiado que estão presentes na ISA base do RISC-V (RV32IM), isto é, as seguintes instruções:

- CSRRW `rd`, `csr`, `rs1`. Essa instrução faz uma troca de valores (*swap*) de forma atômica nos registradores CSR de forma que o valor em `csr` vai para `rd` e o valor em `rs1` vai para `csr` [6];
- CSRRS `rd`, `csr`, `rs1`. Essa instrução lê, de forma atômica, o valor de `csr` para o registrador `rd` e seta os bits de `csr` de acordo com a máscara definida em `rs1` [6];
- CSRRC `rd`, `csr`, `rs1`. Essa instrução lê, de forma atômica, o valor de `csr` para o registrador `rd` e zera os *bits* de `csr` de acordo com a máscara definida em `rs1` [6];
- CSRRWI `rd`, `csr`, `imm5`. Essa instrução lê, de forma atômica, o valor de `csr` para o registrador `rd` e carrega para o `csr` o imediato `imm5` de 5 *bits* [6];
- CSRRSI `rd`, `csr`, `imm5`. Essa instrução lê, de forma atômica, o valor de `csr` para o registrador `rd` e seta os *bits* de `csr` de acordo com a máscara de 5 *bits* definida em `imm5` [6];
- CSRRCI `rd`, `csr`, `imm5`. Essa instrução lê, de forma atômica, o valor de `csr` para o registrador `rd` e zera os *bits* de `csr` de acordo com a máscara de 5 *bits* definida em `imm5` [6];

Primeiramente, o RISC-V suporta três tipos de interrupção [35]: interrupção em nível de usuário ou aplicação, de máquina e de supervisor. O Clarvi oferece somente suporte a interrupções de máquina. Entre essas interrupções, esse processador suporta as externas e as exceções (erros) ocorridas durante a execução. Logo, para habilitar as interrupções externas de máquina, deve-se manipular 3 registradores ditos privilegiados: `mip`, `mie` e `mstatus`.

- `mip`: indica que há uma interrupção pendente. Seu *bit* `MEIP` (*bit* 11), mostrado na Figura 3.11 é o responsável por receber o sinal de interrupção externo;

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	**WIRI**	SEIP	UEIP	MTIP	**WIRI**	STIP	UTIP	MSIP	**WIRI**	SSIP	USIP
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figura 3.11: Registrador de interrupção `mip` (Fonte: [35]).

- `mie`: indica que estão habilitadas interrupções externas. Seu *bit* `MEIE` (*bit* 11), mostrado na Figura 3.12 deve ter seu valor colocado em '1' na inicialização do sistema;

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIE	**WPRI**	SEIE	UEIE	MTIE	**WPRI**	STIE	UTIE	MSIE	**WPRI**	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figura 3.12: Registrador de interrupção `mie` (Fonte: [35]).

- `mstatus`: na Figura 3.13, verificam-se os *bits* `MIE` (*bit* 3) e `MPIE` (*bit* 7). Para que o processador desvie para uma rotina de tratamento de interrupção, o *bit* `MIE` deve estar ativo.

31	30 … 23	22	21	20	19	18	17
SD	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV
1	8	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

Figura 3.13: Registrador de estado da máquina `mstatus` (Fonte: [35]).

O *bit* `mstatus.mpie` armazena o valor do *bit* `mstatus.mie` no momento que há um desvio para a rotina de tratamento de interrupções. Nesse momento o `mstatus.mie` tem seu valor zerado, de forma a evitar interrupções durante o tratamento de outra. Esse *bit*, só recebe o valor do `mstatus.mpie` no momento que há uma retorno pela instrução `mret`, geralmente executada em rotinas de tratamento de interrupções.

Para desabilitar as interrupções durante a execução da `main`, apenas os *bits* `mie.meie` e `mstatus.mpie` tiveram seus valores setados. Dessa forma, quando essa função chamar a rotina de restauração de contexto da tarefa de sistema ocioso, a instrução `mret` será executada, habilitando as interrupções do sistema.

A última diferença nas implementações é que, enquanto o processador Plasma tem o endereço de desvio pré-definido, no Clarvi, esse valor é definido na rotina de habilitar interrupções, `ASM_SetInterruptEnable`, por meio do valor colocado no registrador privilegiado `mtvec`. Para definir um endereço de desvio diferente, basta usar a instrução `csrw`, por exemplo. Logo, a mesma alteração no processador Plasma exigiria do desenvolvedor conhecimento prévio a respeito de seu funcionamento.

3.3.2 Salvamento e recuperação de contexto

No *kernel* PE escravo, o salvamento e a restauração de contexto é realizado pelo seu programa de *boot*. A execução ocorre sempre que há um interrupção ou uma chamada

de sistema. No Código-fonte 3.15 é mostrada a rotina `interrupt_service_routine` responsável por realizar o salvamento de contexto.

Código 3.15: Rotina de salvamento de contexto para interrupções na versão com processador Plasma

```
1 interrupt_service_routine:
2
3         #REGISTERS $26 AND $27 ARE RESERVED FOR THE OS
4         #la      $26,needTaskScheduling
5         #sw      $0,0($26)
6
7         #LOAD TCB MEMORY POSITION IN k1
8         la      $27,current
9         lw      $27,0($27)
10
11        #SAVE ra AND REGISTERS
12        sw      $2,0($27)       # reg[0] = $v0
13        sw      $3,4($27)       # reg[1] = $v1
14        sw      $4,8($27)       # reg[2] = $a0
15        sw      $5,12($27)      # reg[3] = $a1
16        sw      $6,16($27)      # reg[4] = $a2
17        sw      $7,20($27)      # reg[5] = $a3
18        sw      $8,24($27)      # reg[6] = $t0
19        sw      $9,28($27)      # reg[7] = $t1
20        sw      $10,32($27)     # reg[8] = $t2
21        sw      $11,36($27)     # reg[9] = $t3
22        sw      $12,40($27)     # reg[10] = $t4
23        sw      $13,44($27)     # reg[11] = $t5
24        sw      $14,48($27)     # reg[12] = $t6
25        sw      $15,52($27)     # reg[13] = $t7
26        sw      $16,56($27)     # reg[14] = $s0
27        sw      $17,60($27)     # reg[15] = $s1
28        sw      $18,64($27)     # reg[16] = $s2
29        sw      $19,68($27)     # reg[17] = $s3
30        sw      $20,72($27)     # reg[18] = $s4
31        sw      $21,76($27)     # reg[19] = $s5
32        sw      $22,80($27)     # reg[20] = $s6
33        sw      $23,84($27)     # reg[21] = $s7
34        sw      $24,88($27)     # reg[22] = $t8
35        sw      $25,92($27)     # reg[23] = $t9
36
37 #       sw      $28,96($27)     # reg[24] = $gp
38
39        sw      $29,100($27)    # reg[25] = $sp
```

```
40          sw      $30,104($27)    # reg[26] = $s8
41          sw      $31,108($27)    # reg[27] = $ra
42
43          mfhi    $26
44          sw      $26,112($27)    # reg[28] = $hi
45
46          mflo    $26
47          sw      $26,116($27)    # reg[29] = $lo
48
49          mfc0    $26,$14
50          sw      $26,120($27)    # reg[30] = save PC
51
52          li $sp,sp_addr
53
54          #JUMP TO INTERRUPT SERVICE ROUTINE
55          lui     $5,0x2000
56          lw      $4,0x20($5)     #IRQ_STATUS
57          lw      $6,0x10($5)     #IRQ_MASK
58          jal     OS_InterruptServiceRoutine
59          and     $4,$4,$6
```

Já no Código-fonte 3.16, tem-se a restauração do contexto de uma tarefa escolhida pelo escalonador para ser executada em `ASM_RunScheduledTask` na versão do sistema que utiliza o processador Plasma.

Código 3.16: Rotina de restauração de contexto para interrupções na versão original

```
 1          .globl  ASM_RunScheduledTask
 2          .ent    ASM_RunScheduledTask
 3          .set    noreorder
 4
 5 ASM_RunScheduledTask:
 6
 7          move    $27,$4
 8
 9          lw      $2,0($27)       # $v0
10          lw      $3,4($27)       # $v1
11          lw      $4,8($27)       # $a0
12          lw      $5,12($27)      # $a1
13          lw      $6,16($27)      # $a2
14          lw      $7,20($27)      # $a3
15          lw      $8,24($27)      # $t0
16          lw      $9,28($27)      # $t1
17          lw      $10,32($27)     # $t2
18          lw      $11,36($27)     # $t3
```

```
19          lw      $12,40($27)     # $t4
20          lw      $13,44($27)     # $t5
21          lw      $14,48($27)     # $t6
22          lw      $15,52($27)     # $t7
23          lw      $16,56($27)     # $s0
24          lw      $17,60($27)     # $s1
25          lw      $18,64($27)     # $s2
26          lw      $19,68($27)     # $s3
27          lw      $20,72($27)     # $s4
28          lw      $21,76($27)     # $s5
29          lw      $22,80($27)     # $s6
30          lw      $23,84($27)     # $s7
31          lw      $24,88($27)     # $t8
32          lw      $25,92($27)     # $t9
33
34 #         lw      $28,96($27)     # $gp
35
36          lw      $29,100($27)    # $sp
37          lw      $30,104($27)    # $s8
38          lw      $31,108($27)    # $ra
39          lw      $26,112($27)    # $hi
40
41          mthi    $26
42          lw      $26,116($27)    # $lo
43          mtlo    $26
44
45          lw      $26,120($27)    # loads pc of the task that will run
46          lw      $27,124($27)    # loads offset of the task
47          li      $1,0x1
48
49          mtc0    $27,$10
50          jr      $26             # jumps to pc
51          mtc0    $1,$12          # enables interrupts
52
53          .set    reorder
54          .end    ASM_RunScheduledTask
```

Nas rotinas da versão que executa com o processador Plasma, nota-se que essas funcionalidades dependem diretamente da arquitetura devido aos índices dos seus registradores e aos registradores disponíveis. Nelas estão presentes os registradores reservados ao *kernel* ($26 e $27), que correspondem a `$k0` e `$k1`, e os registradores de retorno ($2 e $3) `$v0` e `$v1` que não são definidos na arquitetura RISC-V, por exemplo. No RISC-V, outros registradores são utilizados nesses mesmos índices, como pode ser visto na Figura 3.14. Nela, pode-se comparar as diferenças entre as duas arquiteturas utilizadas e ainda é possí-

vel notar que os registradores utilizados nas instruções de operação e divisão, `$hi` e `$lo`, nem mesmo existem na implementação RISC-V.

Name	Number
zero	0
ra	1
sp	2
gp	3
tp	4
t0	5
t1	6
t2	7
s0	8
s1	9
a0	10
a1	11
a2	12
a3	13
a4	14
a5	15
a6	16
a7	17
s2	18
s3	19
s4	20
s5	21
s6	22
s7	23
s8	24
s9	25
s10	26
s11	27
t3	28
t4	29
t5	30
t6	31
pc	32

RISC-V

Name	Number
$zero	0
$at	1
$v0	2
$v1	3
$a0	4
$a1	5
$a2	6
$a3	7
$t0	8
$t1	9
$t2	10
$t3	11
$t4	12
$t5	13
$t6	14
$t7	15
$s0	16
$s1	17
$s2	18
$s3	19
$s4	20
$s5	21
$s6	22
$s7	23
$t8	24
$t9	25
$k0	26
$k1	27
$gp	28
$sp	29
$fp	30
$ra	31
pc	
hi	
lo	

MIPS

Figura 3.14: Registradores presentes na arquitetura RISC-V (esquerda) e MIPS (direita).

Como na arquitetura RISC-V não há mais registradores reservados para o *kernel* como os `$k0` e `$k1` do MIPS, faz-se necessários usar outra abordagem para armazenar o valor da posição de memória inicial onde os registradores devem ser salvos ou restaurados. O registrador privilegiado `mscratch` no RISC-V tem a mesma função que aqueles registradores reservados ao *kernel* do MIPS. No Código-fonte 3.17 é possível ver a utilização desse registrador.

Código 3.17: Parte da rotina de tratamento de interrupções onde há o salvamento do contexto na versão de arquitetura RISC-V

```
interrupt_service_routine:
        csrrw a7, mscratch, a7  #save a7 in mscratch, then ignore old mscratch...
        la a7, current          #get current value (will be in a7)
        lw a7, 0(a7)

        #SAVE ra AND REGISTERS
        sw x1,    0(a7)         # reg[0]  = ra
        sw x2,    4(a7)         # reg[1]  = sp
        sw x4,    8(a7)         # reg[2]  = tp
        sw x5,   12(a7)         # reg[3]  = t0
        sw x6,   16(a7)         # reg[4]  = t1
        sw x7,   20(a7)         # reg[5]  = t2
        sw x8,   24(a7)         # reg[6]  = s0/fp
        sw x9,   28(a7)         # reg[7]  = s1
        sw x10,  32(a7)         # reg[8]  = a0
        sw x11,  36(a7)         # reg[9]  = a1
        sw x12,  40(a7)         # reg[10] = a2
        sw x13,  44(a7)         # reg[11] = a3
        sw x14,  48(a7)         # reg[12] = a4
        sw x15,  52(a7)         # reg[13] = a5
        sw x16,  56(a7)         # reg[14] = a6
        # sw x17,  60(a7)       # reg[15] = a7 #will be saved later!
        sw x18,  64(a7)         # reg[16] = s2
        sw x19,  68(a7)         # reg[17] = s3
        sw x20,  72(a7)         # reg[18] = s4
        sw x21,  76(a7)         # reg[19] = s5
        sw x22,  80(a7)         # reg[20] = s6
        sw x23,  84(a7)         # reg[21] = s7
        sw x24,  88(a7)         # reg[22] = s8
        sw x25,  92(a7)         # reg[23] = s9
        sw x26,  96(a7)         # reg[24] = s10
        sw x27, 100(a7)         # reg[25] = s11
        sw x28, 104(a7)         # reg[26] = t3
        sw x29, 108(a7)         # reg[27] = t4
        sw x30, 112(a7)         # reg[28] = t5
        sw x31, 116(a7)         # reg[29] = t6

        #ignore x3 (gp)

        # Now that everything is saved, PC will be:
        csrr t0, mepc
        sw t0, 120(a7)
        mv t0, zero
```

```
44
45          # Finally, we can save a7 on memory!
46          csrrw a7, mscratch, a7
47          la t0, current          #get current value (will be in t0)
48          lw t0, 0(t0)
49          sw a7, 60(t0) #mscratch[15] = a7
```

Primeiramente, por meio da instrução `csrrw`, faz-se uma troca de valores entre o que está no registrador `mscratch` e o que está em `a7`. Dessa forma, o registrador `a7` estará salvo em `mscratch` e o valor deste, estará em `a7`. Então, como não importa o valor antigo que estava nesse registrador privilegiado, o ponteiro do TCB que armazena o conteúdo dos registradores, `current`, é colocado em `a7`. Assim, os registradores de 1 a 31 serão salvos a partir da posição de memória contida nesse ponteiro. No entanto, o salvamento do registrador `a7` (x17) só ocorre quando todos os outros, inclusive o `pc`, já foram salvos. Nesse momento, é possível utilizar um temporário (`t0`, por exemplo) para obter o valor que estava no registrador privilegiado `mepc`. Esse valor corresponde ao `pc` que deve ser retornado na tarefa interrompida. O ponteiro `current` é então colocado nesse mesmo registrador temporário e, dessa forma, pode-se salvar o `a7`.

A rotina de recuperação de contexto, que tem um trecho mostrado no Código-fonte 3.18, possui modificações semelhantes àquelas presentes no salvamento.

Código 3.18: Parte da rotina de recuperação de contexto `ASM_RunScheduledTask` na versão de arquitetura RISC-V

```
1           .globl  ASM_RunScheduledTask
2 ASM_RunScheduledTask:
3         lw a7, 60(a0)
4         csrrw a7, mscratch, a7 # stores a7 in mscratch
5                                                   # then we can use a7
    to recover context
6         mv a7, a0
7
8         lw              a0,      120(a7)        # first, set desired pc
    to mepc
9         csrw  mepc,      a0
10                                                        # then, recover
    page offset.
11        lw      a1,  124(a7)
12        li              a2, 0x20000190
13        sw              a1, 0(a2) #NEXT_PAGE
14
15        # Finally: recover scheduled task context
16
```

```
17          lw      x1,    0(a7)    # ra
18          lw      x2,    4(a7)    # sp
19
20          bnez x2, dont_set_initial_sp # if sp is not zero continue, else,
      loads from initial sp
21          # lw      x2,  128(a7)
22          li sp, sp_addr
23
24 dont_set_initial_sp:
25          lw      x4,    8(a7)    # tp
26          lw      x5,   12(a7)    # t0
27          lw      x6,   16(a7)    # t1
28          lw      x7,   20(a7)    # t2
29          lw      x8,   24(a7)    # s0/fp
30          lw      x9,   28(a7)    # s1
31          lw      x10,  32(a7)    # a0
32          lw      x11,  36(a7)    # a1
33          lw      x12,  40(a7)    # a2
34          lw      x13,  44(a7)    # a3
35          lw      x14,  48(a7)    # a4
36          lw      x15,  52(a7)    # a5
37          lw      x16,  56(a7)    # a6
38
39          # lw      x17,  60(a7)    # a7 #recover from csr
40
41          lw      x18,  64(a7)    # s2
42          lw      x19,  68(a7)    # s3
43          lw      x20,  72(a7)    # s4
44          lw      x21,  76(a7)    # s5
45          lw      x22,  80(a7)    # s6
46          lw      x23,  84(a7)    # s7
47          lw      x24,  88(a7)    # s8
48          lw      x25,  92(a7)    # s9
49          lw      x26,  96(a7)    # s10
50          lw      x27, 100(a7)    # s11
51          lw      x28, 104(a7)    # t3
52          lw      x29, 108(a7)    # t4
53          lw      x30, 112(a7)    # t5
54          lw      x31, 116(a7)    # t6
55
56          #lw             x3, 120(a7)
57
58          csrrw a7, mscratch, a7
```

Ao contrário do que acontece no salvamento, o primeiro passo a ser feito nessa rotina

é recuperar o valor do registrador a7, aproveitando que a0 possui o endereço de memória da TCB. O próximo passo necessário é salvar esse valor recuperado de a7 no registrador privilegiado mscratch. A partir daí, pode-se então carregar o valor do ponteiro current, presente em a0, no registrador a7 e realizar a restauração do contexto.

A restauração em si, é composta de 3 partes. Na primeira, o valor do pc que deverá ser executado é restaurado. Em seguida, a segunda parte consiste na restauração da página que deverá ser executada. Para isso, seu valor recuperado do TCB é escrito no registrador mapeado em memória NEXT_PAGE, conforme foi descrito em 3.1.1. A terceira e última parte consiste na restauração dos demais registradores que compõem o contexto dessa tarefa. No caso da tarefa estar sendo executada pela primeira vez, seu ponteiro de pilha (sp) será 0 e, por isso, deverá ser inicializado a fim de se evitar erros de execução, como por exemplo, tentativa de acesso a endereços negativos que levam a exceções.

3.3.3 Rotinas relacionadas ao tratamento de pacotes (DMNI)

As rotinas relacionadas ao tratamento de recebimentos e envios de pacotes no *kernel* manipulam registradores do TCB da tarefa em execução diretamente. Por isso, devem ser adequadas à nova arquitetura.

Como exemplo, tem-se no Código-fonte 3.19a rotina handle_packet da versão com o processador Plasma, que foi chamada pela rotina Syscall do *kernel*. No caso do serviço em execução ser do tipo MESSAGE_DELIVERY, durante seu tratamento, há uma leitura e uma escrita nos registradores salvos na posição 3 ($a1) e 0 ($v0), respectivamente, do vetor de registradores do TCB da tarefa consumidora.

Código 3.19: Tratamento de mensagem com o serviço MESSAGE_DELIVERY na função handle_packet na versão original da HeMPS

```
1         case  MESSAGE_DELIVERY:
2
3                 tcb_ptr = searchTCB(p->consumer_task);
4
5                 msg_ptr = (Message *)(tcb_ptr->offset | tcb_ptr->reg[3]);
6
7                 msg_ptr->length = p->msg_lenght;
8
9                 DMNI_read_data(
10                     (unsigned int)msg_ptr->msg,
11                     msg_ptr->length
12            );
13
14                tcb_ptr->reg[0] = 1;
```

```
15
16                  //Release task to execute
17                  tcb_ptr->scheduling_ptr->waiting_msg = 0;
```

Na versão com arquitetura RISC-V, esses mesmos registradores devem ser substituídos por 9 e 8, respectivamente. Assim, o registrador `a1` (`reg[9]`, no TCB) será lido, como na outra versão. Como no RISC-V os registradores de retorno são os do tipo "a", o registrador que armazena o resultado de funções é o `a0` (`reg[8]`, no TCB) nessa arquitetura. No Código-fonte 3.20, estão presentes as modificações necessárias no Código-fonte 3.19.

Código 3.20: Tratamento de mensagem com serviço `MESSAGE_DELIVERY` na função `handle_packet` na versão com o Clarvi

```
1          case  MESSAGE_DELIVERY:
2
3          tcb_ptr = searchTCB(p->consumer_task);
4
5          msg_ptr = (Message *)(tcb_ptr->offset | tcb_ptr->reg[9]);
6
7          msg_ptr->length = p->msg_lenght;
8
9          DMNI_read_data((unsigned int)msg_ptr->msg, msg_ptr->length);
10
11         tcb_ptr->reg[8] = 1;
12
13         //Release task to execute
14         tcb_ptr->scheduling_ptr->waiting_msg = 0;
```

No Código-fonte 3.21, estão presentes essas modificações que foram realizadas na função `write_local_msg_to_task`.

Código 3.21: Função `write_local_msg_to_task` modificada para a versão com o processador Clarvi

```
1          Message * msg_ptr;
2
3          msg_ptr =
4              (Message*)((task_tcb_ptr->offset) |
5              ((unsigned int)task_tcb_ptr->reg[9]));
6
7          msg_ptr->length = msg_lenght;
8
9          for (int i=0; i<msg_ptr->length; i++)
10                 msg_ptr->msg[i] = msg_data[i];
11
```

```
12        //Unlock the blocked task
13        task_tcb_ptr->reg[8] = 1;
14
15        //Release task to execute
16        task_tcb_ptr->scheduling_ptr->waiting_msg = 0;
```

Outro ponto relacionado à comunicação que exige essa adaptação de mesma forma é o módulo de migração de tarefas. Na migração, que consiste em transferir uma tarefa de um PE a outro, todo o contexto do TCB deve ser transferido. Por isso, faz-se necessário realizar adaptações nos registradores que são explicitamente copiados, como exemplo, tem-se os Códigos-fonte 3.22 a 3.23. O Código-fonte 3.22 mostra a versão original enquanto que o Código-fonte 3.23 mostra a função modificada para a execução com o processador Clarvi.

Código 3.22: Função `handle_migration_TCB` original para a versão com o processador Plasma

```
1         void handle_migration_TCB(volatile ServiceHeader * p, TCB *
    migrate_tcb){
2
3         volatile unsigned int tcb_registers[30];
4
5         migrate_tcb->pc = p->program_counter + migrate_tcb->offset;
6
7
8         DMNI_read_data((unsigned int) &tcb_registers, 30);
9
10        for (int i=0; i<30; i++){
11                if (i == 27)
12                        migrate_tcb->reg[i] = tcb_registers[i] +
    migrate_tcb->offset;
13                else
14                        migrate_tcb->reg[i] = tcb_registers[i];
15        }
16
17        if (p->period > 0) //This means that the task have RT parameters
18                real_time_task(migrate_tcb->scheduling_ptr, p->period, p
    ->deadline, p->execution_time);
19
20     #if TASK_MIGRATION_DEBUG
21        puts("\tTCB␣received\n");
22     #endif
23     }
```

Código 3.23: Função handle_migration_TCB modificada para a versão com o processador Clarvi

```
1        void handle_migration_TCB(volatile ServiceHeader * p, TCB *
     migrate_tcb){
2
3        volatile unsigned int tcb_registers[30];
4
5        migrate_tcb->pc = p->program_counter + migrate_tcb->offset;
6
7
8        DMNI_read_data((unsigned int) &tcb_registers, 30);
9
10       for (int i=0; i<30; i++){
11               if (i == 0) // Modified RISC-V
12                       migrate_tcb->reg[i] = tcb_registers[i] +
     migrate_tcb->offset;
13               else
14                       migrate_tcb->reg[i] = tcb_registers[i];
15       }
16
17       if (p->period > 0) //This means that the task have RT parameters
18               real_time_task(migrate_tcb->scheduling_ptr, p->period, p
     ->deadline, p->execution_time);
19
20 #if TASK_MIGRATION_DEBUG
21       puts("\tTCB␣received\n");
22 #endif
23 }
```

3.4 Alterações na implementação do processador Clarvi

Para o correto funcionamento do sistema, foram necessárias modificações diretamente no processador. Em 3.3.1, foram abordadas as mudanças na implementação da interrupção. Naquele caso, foi citada a necessidade de adicionar o sinal irq_ret ao processador. Esse sinal nada mais é do que a cópia de outro sinal ativado quando o processador desvia para a rotina de tratamento de interrupções.

Nessa seção, busca-se detalhar outras mudanças que foram essenciais para implementar todas as funcionalidades necessárias. Essas mudanças consistiram em estender o conjunto de instruções originais do Clarvi para o conjunto RV32IM, conforme a subseção 3.4.1, e em adicionar desvios para chamadas de sistema, o que é abordado na 3.4.2. Assim, as

funções de multiplicação, divisão e resto passaram a ser suportadas na versão cujo sistema tem arquitetura RISC-V.

3.4.1 Extensão do conjunto de instruções do processador para RV32IM

O *kernel* da HeMPS utiliza operações de divisão e multiplicação. O processador Plasma já implementa via hardware essas instruções necessárias. Entretanto, essas instruções não estão disponíveis no processador Clarvi, por implementar originalmente apenas o conjunto de instruções RV32I. Para disponibilizá-las, foi necessário implementar o conjunto padrão de extensão "M". A partir desse conjunto, o processador passa a oferecer essas operações, aceitando códigos compilados para o conjunto RV32IM.

O subconjunto "M" é formado por 8 instruções, 4 de multiplicação, 2 de divisão e 2 de obtenção do resto da divisão. Essas instruções são:

- "MUL": multiplica dois inteiros de 32 bits com sinal. Como o resultado pode retornar um número de 64bits, "MUL" retorna no registrador de destino especificado apenas os 32 bits menos significativos do resultado [6].
- "MULH": multiplica dois inteiros de 32 bits com sinal. Ao contrário da instrução "MUL", "MULH" retorna no registrador de destino especificado apenas os 32 bits mais significativos do resultado [6].
- "MULHSU": multiplica dois inteiros de 32 bits sendo um deles com sinal e outro sem. Assim como na instrução "MULH", retorna os 32 bits mais significativos do resultado [6].
- "MULHU": multiplica dois inteiros de 32 bits sem sinal. Assim como na instrução "MULH", retorna os 32 bits mais significativos do resultado [6].
- "DIV": divisão de dois inteiros de 32 bits com sinal. Nessa instrução apenas o quociente é retornado no registrador de destino, enquanto que o resto é ignorado.
- "DIVU": divide dois inteiros de 32 bits sem sinal. Assim como na instrução "DIV" apenas o quociente é retornado.
- "REM": obtém o resto da divisão entre dois inteiros de 32 bits com sinal. Nessa instrução apenas o resto é retornado no registrador de destino, enquanto que o quociente é ignorado.
- "REMU": obtém o resto da divisão entre dois inteiros de 32 bits sem sinal. Assim como na instrução "REM" apenas o resto da divisão é retornado.

Essas instruções são formadas por 6 campos que são: `funct7`, `rs2`, `rs1`, `funct3`, `rd` e `opcode`. Os campos `rs2`, `rs1` e `rd` são passados por parâmetros, como pode ser visto no Código-fonte 3.24.

Código 3.24: Parâmetros `a5`, `a4`, e `a5`, passados à instrução `remu`

```
1     74: 00a00793 li a5,10
2     78: 02f777b3 remu a5,a4,a5
3     7c: 0ff7f793 andi a5,a5,255
```

Já a Figura 3.15 apresenta em duas tabelas como são organizados os campos nessas instruções. Sendo que na tabela em a), tem-se as instruções que realizam a multiplicação. Já na tabela em b), as instruções que realizam divisões e obtém restos. Nessas tabelas, as instruções com final "W" representam instruções válidas apenas no conjunto de instruções RV64. Por isso, são instruções que não são implementadas no caso desse estudo.

a)

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

b)

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32

Figura 3.15: Formato das instruções de multiplicação a) e divisão b) do subconjunto M (Fonte: [6]).

Para implementar esse subconjunto, foi necessário, primeiramente, modificar o ciclo de decodificação do processador Clarvi. Nesse ciclo, quando é identificada uma instrução de operação lógico-aritmética (instrução cujo `opcode` é `0110011` [36]) ocorre uma verificação nos bits de 31 a 25, representados por `funct7` na imagem acima. Para que a instrução pertença a esse subconjunto, `funct7` deve ter valor igual a "MULDIV", que é uma constante de valor `0000001` [36]. Quando há essa ocorrência, a função de decodificação deve verificar qual das oito instruções disponíveis nesse subconjunto será executada, de acordo com os bits representados por `funct3` (*bits* 14 a 12). Na Tabela 3.1 são deta-

lhadas as instruções desse subconjunto. A partir dela, pode-se verificar a correspondência entre os campos `funct3` e a instrução.

Tabela 3.1: Tabela de instruções do subconjunto M (Fonte: [36]).

MULDIV	**RS2**	**RS1**	**FUNCT3**	**RD**	**OPCODE**	**Instrução**
0000001	rs2	rs1	000	rd	0110011	mul
0000001	rs2	rs1	001	rd	0110011	mulh
0000001	rs2	rs1	010	rd	0110011	mulhsu
0000001	rs2	rs1	011	rd	0110011	mulhu
0000001	rs2	rs1	100	rd	0110011	div
0000001	rs2	rs1	101	rd	0110011	divu
0000001	rs2	rs1	110	rd	0110011	rem
0000001	rs2	rs1	111	rd	0110011	remu

Ainda nesse ciclo, são extraídos os valores dos parâmetros usados nas operações de forma que:

- `rs1`: corresponde ao registrador que possui o multiplicando ou o dividendo;
- `rs2`: corresponde ao registrador que possui o multiplicador ou o divisor;
- `rd`: corresponde ao registrador de destino;

No ciclo de execução, foram usados simples operadores de deslocamento, multiplicação, divisão e módulo de divisão. As instruções foram implementadas das seguintes formas:

- "MUL": multiplicação entre os valores de `rs1` e `rs2` estendidos com sinal;
- "MULH": multiplicação entre os valores de `rs1` e `rs2` estendidos com sinal, sendo retornados apenas os *bits* de 63 a 32;
- "MULHSU": multiplicação entre o valor de `rs1` estendido com sinal pelo valor de `rs2` estendido sem sinal usando 32 deslocamentos para a esquerda seguidos de 32 para a direita. Sendo que são retornados apenas os *bits* de 63 a 32;
- "MULHU": multiplicação entre os valores de `rs1` e `rs2` estendidos sem sinal, sendo retornados apenas os *bits* de 63 a 32;
- "DIV": divisão entre os valores de `rs1` pelo de `rs2` estendidos com sinal;
- "DIVU": divisão entre os valores de `rs1` pelo de `rs2` sem sinal;
- "REM": obtenção de resto da divisão entre os valores de `rs1` pelo de `rs2` estendidos com sinal;

- “REMU”: obtenção de resto da divisão entre os valores de `rs1` pelo de `rs2` estendidos com sinal;

No Código-fonte 3.25, é mostrado o trecho de código responsável por executar instruções do tipo “M”. Entre as linhas 7 e 18 são definidas expansões com ou sem sinal dos valores, de acordo com as instruções. Já entre as linhas 21 e 28 há o retorno da execução, que utiliza o operador `*` para multiplicação, `/` para divisão e `%` para obtenção de resto.

Código 3.25: Trecho da função que mostra a implementação da execução de instruções lógico-aritméticas que corresponde à execução do subconjunto “M” no processador Clarvi

```
1        logic [31:0] rs2_value_or_imm =
2            instr.immediate_used ? instr.immediate : rs2_value;
3
4        // shifts use the lower 5 bits of the intermediate or rs2 value
5        logic [4:0] shift_amount = rs2_value_or_imm[4:0];
6
7        logic [63:0] rst_mulh =
8            instr.op == MULH ? $signed(rs1_value) * $signed(rs2_value) :
     0;
9
10       logic [63:0] rs1_mulhsu =
11           instr.op == MULHSU ? $signed(rs1_value) : 0;
12       logic [63:0] rs2_mulhsu =
13           instr.op == MULHSU ? (rs2_value << 32) >> 32 : 0;
14       logic [127:0] rst_mulhsu =
15           instr.op == MULHSU ? rs1_mulhsu * rs2_mulhsu : 0;
16
17       logic [63:0] rst_mulhu =
18           instr.op == MULHU ? $unsigned(rs1_value) * $unsigned(
     rs2_value) : 0;
19
20       unique case (instr.op)
21           MUL: return $signed(rs1_value) * $signed(rs2_value);
22           MULH: return rst_mulh[63:32];
23           MULHSU: return rst_mulhsu[63:32];
24           MULHU: return rst_mulhu[63:32];
25           DIV: return $signed(rs1_value) / $signed(rs2_value);
26           DIVU: return rs1_value / rs2_value;
27           REM: return $signed(rs1_value) % $signed(rs2_value);
28           REMU: return rs1_value % rs2_value;
```

3.4.2 Chamadas de sistema

Na versão original da plataforma, as interrupções de software, ou chamadas de sistema, foram implementadas a partir da instrução `syscall`. Essa instrução é presente no conjunto de instruções MIPS e não está disponível na versão utilizada do processador Plasma.

Logo, a implementação dessa instrução foi feita de tal forma que sua execução é análoga a uma interrupção. Nela, o processador faz um desvio para uma rotina de tratamento que está fixada no endereço `0x44` da página do *kernel*.

Na versão com arquitetura RISC-V, optou-se por implementar o tratamento de chamadas de sistema de forma semelhante à original, mantendo a rotina de tratamento no mesmo local de memória. A Figura 3.16 detalha o fluxo de execução de quando uma tarefa realiza uma chamada de sistema tanto na versão com o processador Plasma quanto na versão com o Clarvi.

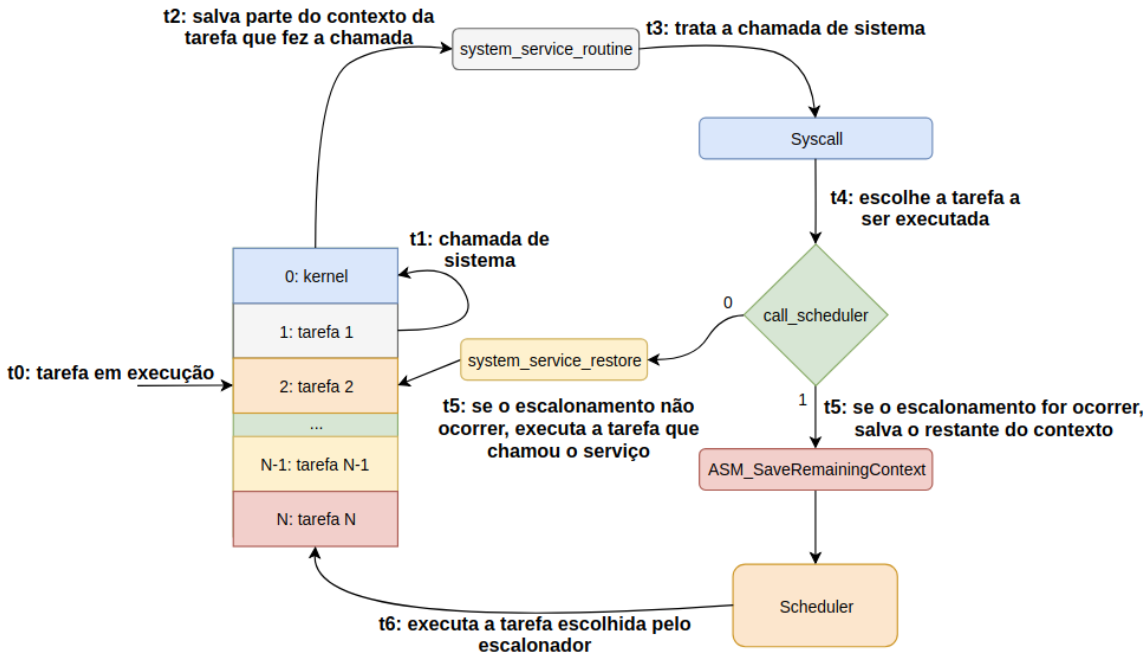

Figura 3.16: Fluxo de execução de uma chamada de sistema.

No tempo `t0` da Figura 3.16, há uma tarefa na página 2 em execução. Em `t1`, essa tarefa executa uma rotina que leva a uma chamada de sistema. Dessa forma, o contexto da tarefa que fez a chamada é salvo parcialmente, no tempo `t2`, e a função de tratamento de chamadas de sistema no *kernel*, `Syscall`, é executada em `t3`.

Após o tratamento da chamada de sistema realizado, o sistema decide no tempo `t4` da Figura 3.16, se uma nova tarefa deverá ser escolhida pelo escalonador. Caso não seja necessário escalonamento, em `t5` o sistema recupera a parte do contexto salvo e volta a

executar a tarefa que fez a chamada em `t1`. Caso seja, o sistema salva parte do restante do contexto da tarefa no tempo `t5` que fez a chamada e, em seguida, chama o escalonador, que no instante `t6` faz a recuperação do contexto da tarefa escolhida para a execução.

A diferença entre as versões com arquiteturas diferentes se dá pela instrução que deve ser executada pelo código *Assembly* do *boot* do *kernel*. Enquanto a versão de arquitetura MIPS usa a instrução `syscall`, em RISC-V usa-se a `ecall`. No processador Clarvi, essa instrução era implementada apenas para parar a execução da simulação. Como a HeMPS já implementa, por meio de um registrador mapeado em memória, a finalização da simulação, a funcionalidade do Clarvi não era necessária nesse sistema.

Logo, a instrução `ecall` foi modificada para fazer com que o PE altere a página corrente para a do *kernel*, invalide as instruções seguintes da página corrente e o `pc` seja alterado para a posição da rotina de tratamento. O Código-fonte 3.26, mostra o momento que o sinal `ex_ecall` é ativado quando não há uma decodificação inválida e a instrução identificada é a `ecall`. O sinal `ecall_signal`, que também é ativado, tem a função de avisar o PE que a página deve ser alterada para a que possui o *kernel*.

Código 3.26: Ativação de uma chamada de sistema no processador Clarvi

```
1        ex_ecall = !de_ex_invalid && (de_ex_instr.op == ECALL);
2        ecall_signal = ex_ecall;
```

Em relação à implementação das interrupções é possível abordar duas diferenças:

1. O processador desvia, em caso de interrupção externa (`mstatus.mie && mip.meip && mie.meie`) ou exceções (`if_exception || ex_exception`), para o endereço presente no registrador privilegiado `mtvec`. Dessa forma, o endereço da rotina de tratamento de interrupções pode ser alterada no código de forma simples. Em contrapartida, o endereço da rotina de tratamento das chamadas ao sistema (`ex_ecall`) é fixado no processador em `0x44`, como mostrado no Código-fonte 3.27.

   Código 3.27: Escolha do contador de programa. Escolhe-se o valor em mtvec para interrupções e exceções. Para chamadas de sistema, o endereço `0x44` é escolhido

```
1 if (if_exception || ex_exception || (mstatus.mie && mip.meip && mie.
    meie)) begin
2     pc <= mtvec;
3 end else if (ex_ecall) begin
4     pc <= 32'h044;
5 end else begin
6     pc <= ex_next_pc;
```

```
7 end
```

2. No caso de exceções ou interrupções (`if_exception || ex_exception || interrupt`), o processador armazena no registrador privilegiado `mepc` o endereço da instrução que foi interrompida. Pode-se armazenar, ainda, o endereço da instrução anterior a interrompida, como é detalhado em 3.4.3. Dessa forma, ao fim do tratamento, a instrução em `mepc` será executada novamente. Já quando há chamadas de sistema (`ex_ecall`), deve-se armazenar o endereço da próxima instrução (`pc + 4`), assim como é feito na versão com processador Plasma. O Código-fonte 3.28 mostra essa diferença no código do processador Clarvi.

Código 3.28: Armazenamento do contador de programa para os casos em que há interrupção ou exceção e para o caso em que há uma chamada de sistema

```
1     if (if_exception || ex_exception || interrupt) begin
2         // Entering a trap handler.
3         // Push 0 onto the mstatus interrupts-enabled stack
4         mstatus.mpie <= mstatus.mie;
5         mstatus.mie <= '0;
6
7         // record the address of the instruction
8         // that caused the trap or the instruction that got
    interrupted
9         if(interrupt_ret_dep) begin
10            interrupt_ret_dep <= 0;
11
12            mbadaddr <= trap_pc - 32'd4;
13            mepc <= trap_pc - 32'd4;
14        end else begin
15            mbadaddr <= trap_pc;
16            mepc <= trap_pc;
17        end
18
19        // set the trap cause
20        mcause <= get_trap_cause();
21    end else if (ex_ecall) begin
22        // Entering a trap handler.
23        // Push 0 onto the mstatus interrupts-enabled stack
24        mstatus.mpie <= mstatus.mie;
25        mstatus.mie <= '0;
26        // record the address + 4 of the instruction that called
    ecall
27        mbadaddr <= trap_pc + 32'd4;
```

```
28          mepc <= trap_pc + 32'd4;
29
30          // set the trap cause
31          mcause <= get_trap_cause();
32      end
33
34      if (!de_ex_invalid && de_ex_instr.op == MRET) begin
35          // Returning from trap handler. Pop the mstatus interrupts-
        enabled stack.
36          mstatus.mie <= mstatus.mpie;
37          mstatus.mpie <= '1;
38      end
39
```

Para parar o *pipeline* e invalidar estágios de novas instruções, no momento que há uma chamada de sistema, o processador adota o mesmo procedimento de interrupções, exceções, saltos e *writebacks*, como visto no Código-fonte 3.29.

Código 3.29: Invalidação de estágios do pipeline que estavam em execução no momento em que há interrupções, exceções, saltos, *writebacks* ou chamadas de sistema

```
1  if (!stall_for_memory) begin
2          if (!stall_for_load_dep) begin
3              if_invalid <= interrupt ||
4                  ex_exception ||
5                  if_exception ||
6                  ex_branch_taken ||
7                  ex_ecall;
8
9              if_de_invalid <= if_invalid ||
10                           interrupt ||
11                           ex_exception ||
12                           ex_branch_taken ||
13                           ex_ecall;
14         end
15
16         de_ex_invalid <= if_de_invalid ||
17                           interrupt ||
18                           ex_exception ||
19                           ex_branch_taken ||
20                           stall_for_load_dep ||
21                           ex_ecall;
22
23         ex_ma_invalid <= de_ex_invalid ||
24                           interrupt ||
```

```
25                          ex_ecall ||
26                          ex_mem_address_error && de_ex_instr.memory_read
    ||
27                          de_ex_instr.op == INVALID;
28          ma_wb_invalid <= ex_ma_invalid;
29 end
30
```

3.4.3 Controle do retorno de interrupções

No momento que ocorrem interrupções, um endereço de retorno deve ser escolhido. Na versão original do processador Clarvi, esse endereço era sempre o que estava executando, ou o que ia começar a ser executado quando ocorreu a interrupção, conforme o Código-fonte 3.30.

Código 3.30: Armazenamento do contador de programa para os casos em que há interrupção ou exceção na versão original do processador Clarvi

```
1 if (if_exception || ex_exception || interrupt) begin
2
3         mstatus.mpie <= mstatus.mie;
4         mstatus.mie <= '0;
5
6         mbadaddr <= trap_pc;
7
8         mepc <= trap_pc;
9
10        mcause <= get_trap_cause();
11 end
```

No entanto, durante uma ocorrência de interrupção, pode-se ter o seguinte cenário: está sendo executada uma instrução de *load*, "i1", seguida por outra instrução que depende do seu resultado, "i2". Nesse caso, a instrução "i2" deve esperar o carregamento realizado em "i1". Na interrupção, o `pc` armazenado é o corrente, que já é o da instrução "i2", enquanto que "i1" ainda não foi finalizada.

A figura Figura 3.17 mostra esse fato para uma instrução de carregamento de palavra "i1" = `lw a4, -52(s0)` seguida de uma adição, com parâmetros dependentes do *load*, "i2" = `add a5, a4, a5`. Na região A dessa figura, tem-se a página da tarefa, que corresponde à 3. Já na região B, percebe-se que há a ocorrência de uma interrupção. Em C, tem-se o endereço `0xFEAC`, da terceira página, que estava sendo buscado da memória. Na região D, observa-se, primeiramente, o registrador `mepc`. Nele, percebe-se que na ocorrência da interrupção o endereço de retorno salvo foi o `0x910`. Vê-se em E, que

mostra as instruções executadas, que esse endereço corresponde à instrução seguinte ao *load word* de endereço `0x90C`.

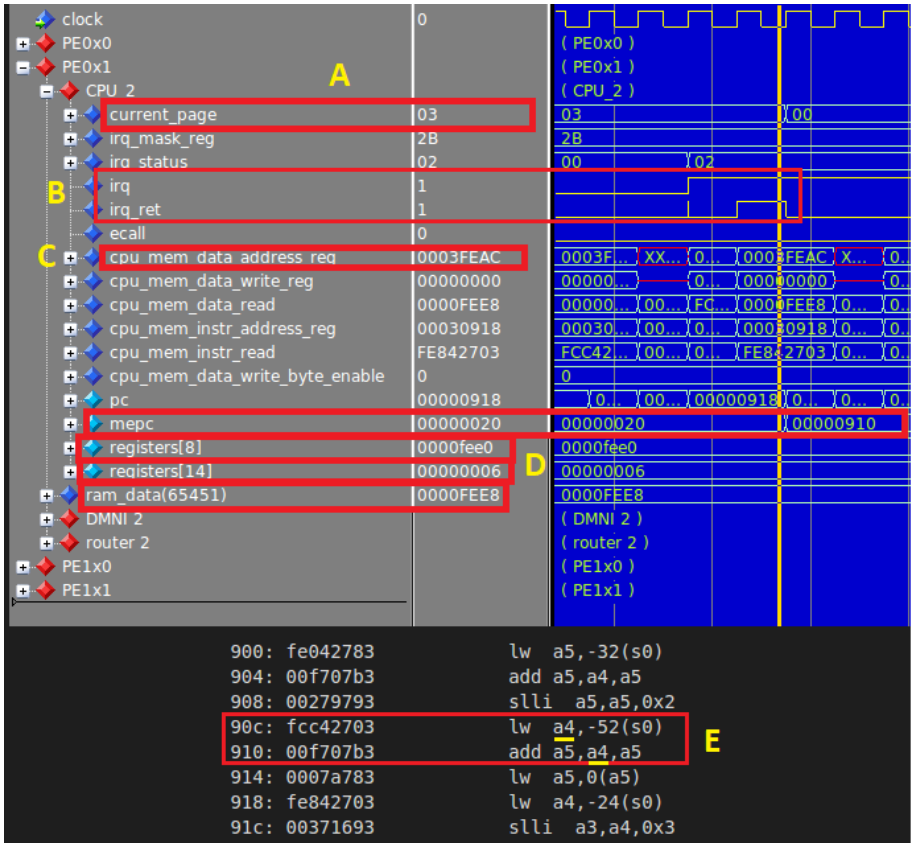

Figura 3.17: Caso em que há interrupção durante uma instrução "lw" seguida de uma adição que depende do que foi carregado.

Nota-se pelas regiões D e E da Figura 3.17, que a instrução load não foi finalizada e deveria ter, portanto, seu endereço salvo no `mepc`. Para que essa instrução tivesse sido completada, o valor no endereço `0x3FEAC` deveria ter sido carregado no registrador de número 14, que corresponde ao `a4`, na região E da Figura 3.17. Esse valor é apresentado em D na posição de memória que corresponde à: `0x3FEAC « 2`, tal que o offset "00" não é considerado, e resulta, em decimal, no valor 65451. Logo, o valor desejado no `a4` é o `0xFEE8` enquanto que o presente é `0x00000006`. Na região D, destaca-se ainda o registrador de número 8, que corresponde ao `s0` na arquitetura RISC-V. Esse registrador é usado como base na operação `lw` em E. Como o registrador `a4` possui um valor incorreto, a continuação da execução dessa tarefa estará comprometida após o tratamento da interrupção.

O Código-fonte 3.31 apresenta o trecho de código do processador com a solução desenvolvida para esse problema.

Código 3.31: Mecanismo para corrigir o pc de retorno das interrupções quando há instruções incompletas

```
1 else if (!stall_for_memory && !stall_for_load_dep) begin
2     if (if_exception || ex_exception || interrupt) begin
3         mstatus.mpie <= mstatus.mie;
4         mstatus.mie <= '0;
5
6         if(interrupt_ret_dep) begin
7             interrupt_ret_dep <= 0;
8
9             mbadaddr <= trap_pc - 32'd4;
10            mepc <= trap_pc - 32'd4;
11        end else begin
12            mbadaddr <= trap_pc;
13            mepc <= trap_pc;
14        end
15
16        mcause <= get_trap_cause();
17    end else if (ex_ecall) begin
18
19        mstatus.mpie <= mstatus.mie;
20        mstatus.mie <= '0;
21
22        mbadaddr <= trap_pc + 32'd4;
23        mepc <= trap_pc + 32'd4;
24
25        mcause <= get_trap_cause();
26    end
27
28    if (!de_ex_invalid && de_ex_instr.op == MRET) begin
29        mstatus.mie <= mstatus.mpie;
30        mstatus.mpie <= '1;
31    end
32
33
34    if (!de_ex_invalid && !interrupt)
35        execute_csr(de_ex_instr, de_ex_rs1_value);
36 end else if (stall_for_load_dep && interrupt) begin
37    interrupt_ret_dep <= 1;
38 end
```

Nas linhas 36 e 37 faz-se a identificação do fenômeno. Quando há um *stall* (uma instrução é parada para esperar outra se completar), durante uma instrução de *load*, o sinal `stall_for_load_dep` é ativado. Isso significa que uma instrução de *load* está em

execução e a próxima instrução depende de seu resultado. Se esse sinal estiver ativado e houver uma interrupção, o fenômeno está presente. Logo, o sinal `interrupt_ret_dep` deve ser setado. No momento que o endereço interrompido, `trap_pc`, vai ser salvo no `mepc`, conforme a as linhas entre 6 e 11, verifica-se o `interrupt_ret_dep`. Se o valor deste for ativo, deve-se armazenar a instrução anterior. Ou seja, `trap_pc - 4`.

Dessa forma, esse mecanismo e as demais funcionalidades modificadas do sistema que foram apresentadas neste capítulo devem ser validados. O próximo capítulo faz a validação dos mecanismos de paginação, interrupção, chamadas de sistema, além do sistema como um todo. Ao fim, são feitas comparações entre as versões que possuem processador Plasma e processador Clarvi.

Capítulo 4

Resultados

O capítulo aborda, de forma geral, como o sistema foi testado. Além disso, busca-se confirmar o funcionamento da plataforma HeMPS com a nova arquitetura de processamento.

Para isso, na seção 4.1, definem-se casos de teste que possuem diferentes complexidades para a validação do sistema e comparação com a versão original.

A seção 4.2 demonstra a validação realizada no sistema proposto.

Na seção 4.3 são comparados pontos relevantes das versões da HeMPS, a que possui o processador Clarvi e a que possui o processador Plasma.

4.1 Casos de testes utilizados

Para realizar a verificação do sistema foram escolhidos alguns MPSoCs gerados a partir de duas aplicações disponibilizadas pelo grupo GAPH [17]. Uma aplicação do tipo produtor/consumidor e outra que realiza uma decodificação mpeg.

A aplicação "hello_world_app", mostrada na Figura 4.1, é do tipo produtor/consumidor e possui 2 tarefas. A tarefa produtora cria 250 mensagens e envia para a consumidora que as lê.

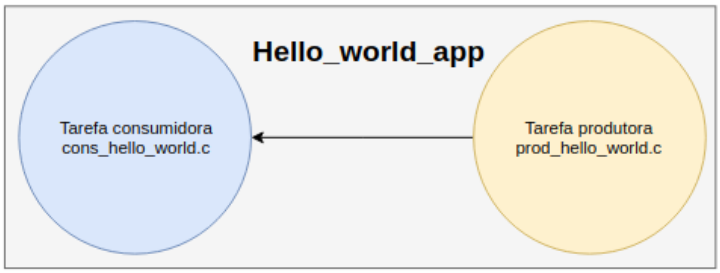

Figura 4.1: Grafo da aplicação "hello_world_app", do tipo produtor/consumidor.

Já a aplicação mpeg é composta por 5 tarefas: tarefa A, tarefa B, tarefa C, tarefa D e tarefa E. Essas tarefas se comunicam de forma sequencial gerando 1 mensagem para cada *frame* definido em mpeg_std.h. A estrutura da aplicação é mostrada na Figura 4.2.

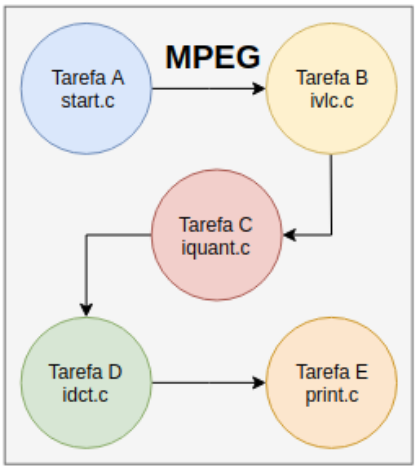

Figura 4.2: Grafo da aplicação "mpeg".

Já para realizar comparações entre as duas versões da HeMPS, são utilizadas outras três aplicações, também disponibilizadas pelo GAPH: a dtw, a djikstra e a synthetic.

A aplicação dtw implementa o algoritmo *Dynamic Time Warping*, que busca a distância entre duas séries temporais. Para isso, usa 6 tarefas. A Figura 4.3 mostra a estrutura e a comunicação dessas tarefas.

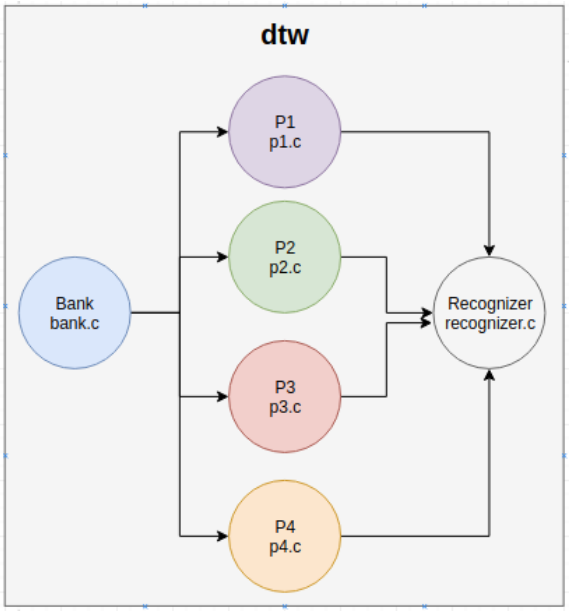

Figura 4.3: Grafo da aplicação "dtw".

Já a aplicação djikstra implementa o famoso algoritmo, de mesmo nome, que busca menores caminhos ou menores custos. Essa aplicação é formada por 7 tarefas e tem seu grafo apresentado na Figura 4.4.

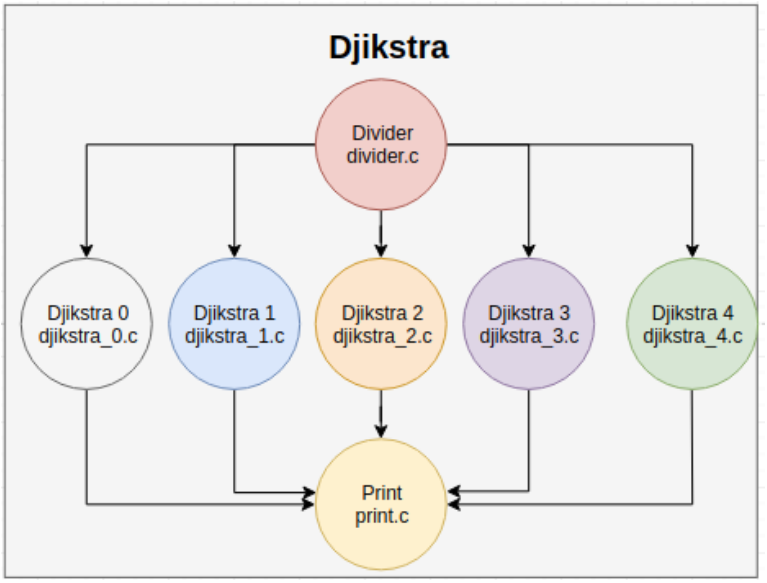

Figura 4.4: Grafo da aplicação "djikstra".

Por fim, a aplicação synthetic realiza iterações e faz envios de mensagens entre suas 6 tarefas. Sua estrutura é mostrada na Figura 4.5.

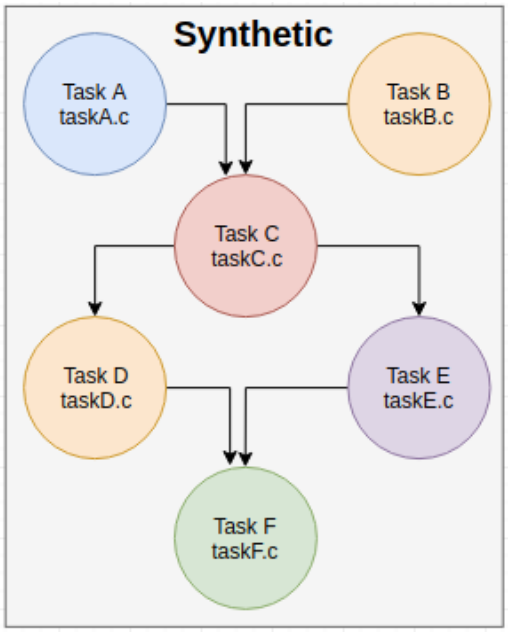

Figura 4.5: Grafo da aplicação "synthetic".

A partir dessas aplicações foram criados 6 diferentes casos de teste.

- Caso de teste 1: 1 aplicação hello_world_app executando em 1 MPSoC com 4 PEs, organizados no formato 2×2, e executando 1 tarefa por PE, conforme o Código-fonte 4.1.

Código 4.1: Arquivo de configuração YAML do caso de teste 1

```
1     hw:
2   page_size_KB: 64
3   tasks_per_PE: 1
4   repository_size_MB: 1
5   model_description: vhdl
6   noc_buffer_size: 8
7   mpsoc_dimension: [2,2]
8   cluster_dimension: [2,2]
9   master_location: LB
10
11 apps:
12   - name: hello_world_app
13     start_time_ms: 0
14
```

- Caso de teste 2: 1 aplicação mpeg executando em 1 MPSoC com 4 PEs, organizados no formato 2×2, e executando 3 tarefas por PE, conforme o Código-fonte 4.2.

Código 4.2: Arquivo de configuração YAML do caso de teste 2

```
1 hw:
2    page_size_KB:  64
3    tasks_per_PE: 3
4    repository_size_MB:  1
5    model_description: vhdl
6    noc_buffer_size: 8
7    mpsoc_dimension: [2,2]
8    cluster_dimension: [2,2]
9    master_location: LB
10
11 apps:
12   - name: mpeg
13     start_time_ms: 0
14
```

- Caso de teste 3: 1 aplicação mpeg e 1 aplicação hello_world_app em 1 MPSoC com 9 PEs, organizados no formato 3×3, e executando 2 tarefas por PE, conforme o Código-fonte 4.3.

Código 4.3: Arquivo de configuração YAML do caso de teste 3

```
1 hw:
2    page_size_KB:  64
3    tasks_per_PE: 2
4    repository_size_MB:  1
5    model_description: vhdl
6    noc_buffer_size: 8
7    mpsoc_dimension: [3,3]
8    cluster_dimension: [3,3]
9    master_location: LB
10
11 apps:
12   - name: mpeg
13     start_time_ms: 0
14   - name: hello_world_app
15     start_time_ms: 5
16
```

- Caso de teste 4: 1 aplicação djikstra, 1 aplicação synthetic e 1 aplicação dtw em 1 MPSoC com 9 PEs, organizados no formato 3×3, e executando 3 tarefas por PE, conforme o Código-fonte 4.4.

Código 4.4: Arquivo de configuração YAML do caso de teste 4

```
1 hw:
2    page_size_KB:  64
3    tasks_per_PE: 3
4    repository_size_MB:  1
5    model_description: vhdl
6    noc_buffer_size: 8
7    mpsoc_dimension: [2,2]
8    cluster_dimension: [2,2]
9    master_location: LB
10
11 apps:
12   - name: mpeg
13     start_time_ms: 0
14   - name: dtw
```

```
15     start_time_ms: 0
16
```

- Caso de teste 5: 1 aplicação mpeg e 1 aplicação dtw em 1 MPSoC com 4 PEs, organizados no formato 2×2, e executando 3 tarefas por PE, conforme o Código-fonte 4.5.

Código 4.5: Arquivo de configuração YAML do caso de teste 5

```
1 hw:
2    page_size_KB:  64
3    tasks_per_PE: 3
4    repository_size_MB:  1
5    model_description: vhdl
6    noc_buffer_size: 8
7    mpsoc_dimension: [3,3]
8    cluster_dimension: [3,3]
9    master_location: LB
10
11 apps:
12   - name: dijkstra
13     start_time_ms: 2
14   - name: synthetic
15     start_time_ms: 2
16   - name: dtw
17     start_time_ms: 2
18
```

- Caso de teste 6: 1 aplicação synthetic e 1 aplicação djikstra em 1 MPSoC com 4 PEs, organizados no formato 2×2, e executando 3 tarefas por PE, conforme o Código-fonte 4.6.

Código 4.6: Arquivo de configuração YAML do caso de teste 6

```
1 hw:
2    page_size_KB:  64
3    tasks_per_PE: 3
4    repository_size_MB:  1
5    model_description: vhdl
6    noc_buffer_size: 8
7    mpsoc_dimension: [2,2]
8    cluster_dimension: [2,2]
```

```
 9    master_location: LB
10
11 apps:
12   - name: synthetic
13     start_time_ms: 0
14   - name: dijkstra
15     start_time_ms: 2
16
```

Esses casos de teste possuem a seguinte estrutura de arquivos:

- diretório "*applications*": possui as aplicações a serem executadas;
- diretório "*build*": contém *scripts* necessários para a compilação e execução do MPSoC gerado;
- diretório "*debug*": armazena os arquivos de configuração necessários para a execução da simulação na ferramenta de depuração "HeMPS_Debugger";
- diretório "*hardware*": onde estão os arquivos de descrição do hardware em linguagens HDL;
- diretório "*include*": guarda definições necessárias para a execução tanto do software quanto para o hardware do MPSoC gerado;
- diretório "*log*": onde os arquivos de *log* gerados pelos PEs durante a inicialização e a execução de tarefas estão;
- diretório "ram_pe": possui arquivos com o conteúdo a ser carregado na memória RAM de cada PE;
- diretório "*software*": abriga os *kernels* mestre e escravo a serem executados pelos PEs.

4.2 Validação do funcionamento

A validação do funcionamento do sistema consiste em confirmar que o sistema é funcional. Ou seja, é a prova de que ele é, no mínimo, capaz de executar as mesmas aplicações que a versão original.

Essa validação é dividida em 4 subseções. Na subseção 4.2.1 faz-se a validação do funcionamento da paginação utilizada. Já na 4.2.2, verificam-se as interrupções e, em 4.2.3, as chamadas de sistema. Por fim, faz-se uma análise geral do funcionamento do sistema de três formas diferentes na subseção 4.2.4. A primeira forma é pela análise dos

arquivos de *log* gerados. A segunda, pela análise das formas de onda durante a execução da simulação no *software* Modelsim [33]. Por último, faz-se uma análise utilizando o *software* de depuração da plataforma "HeMPS_Debugger".

4.2.1 Validação da paginação

A paginação se mostra correta se o valor da página, que é o de deslocamento (*offset*) contido no TCB de uma tarefa, for adicionado ao endereço do registrador contador de programa (`pc`) e ao endereço de dados. Dessa forma, os endereços acessados na memória corresponderão aos endereços daquela tarefa, pois a memória de um PE é dividida em uma página por tarefa.

Nas formas de onda dos casos de teste analisados, é possível observar as transições que ocorrem no sinal de página corrente, `current_page` e, por meio dele, verificar os endereços de instrução e de dados. É observado nas formas de onda do caso de teste 1, mostradas na Figura 4.6, que o endereço de página é adicionado ao endereço do `pc`, de forma a garantir que esse endereço retorne as instruções da tarefa correta em A. Já na região B, observa-se a adição do *offset* de página no endereço de dados. Para esse caso, a página é adicionada na metade mais significativa da palavra do endereço. Enquanto o valor virtual do endereço de instruções corresponde à `0x990` na imagem, o valor real é `0x10990`, por estar na página 1.

Para validar que a instrução buscada nesse endereço está correta, usam-se os arquivos do tipo "lst". Esses são gerados durante a compilação e são encontrados nos diretórios "*applications*" e "*software*" do caso de teste avaliado. Sendo que o primeiro possui os arquivos fontes e objetos das tarefas e o segundo, dos *kernels*. Dessa forma, no Código-fonte 4.7, pode-se ver a instrução que deve ser buscada da tarefa no endereço `0x990` virtual que está na linha 7. Nesse caso, o arquivo em questão corresponde ao da tarefa que executa na página 1 do PE 0x1 (a tarefa "prod_hello_world").

Código 4.7: Trecho da tarefa "prod_hello_world"

```
1
2  97c:   fffff097          auipc   ra,0xfffff
3  980:   6a0080e7          jalr    1696(ra) # 1c <SystemCall>
4  984:   fec42783          lw      a5,-20(s0)
5  988:   00178793          addi    a5,a5,1
6  98c:   fef42623          sw      a5,-20(s0)
7  990:   fec42703          lw      a4,-20(s0)
8  994:   0f900793          li      a5,249
9  998:   f6e7d0e3          ble     a4,a5,8f8 <main+0x78>
10 99c:   00000693          li      a3,0
11 9a0:   00000613          li      a2,0
```

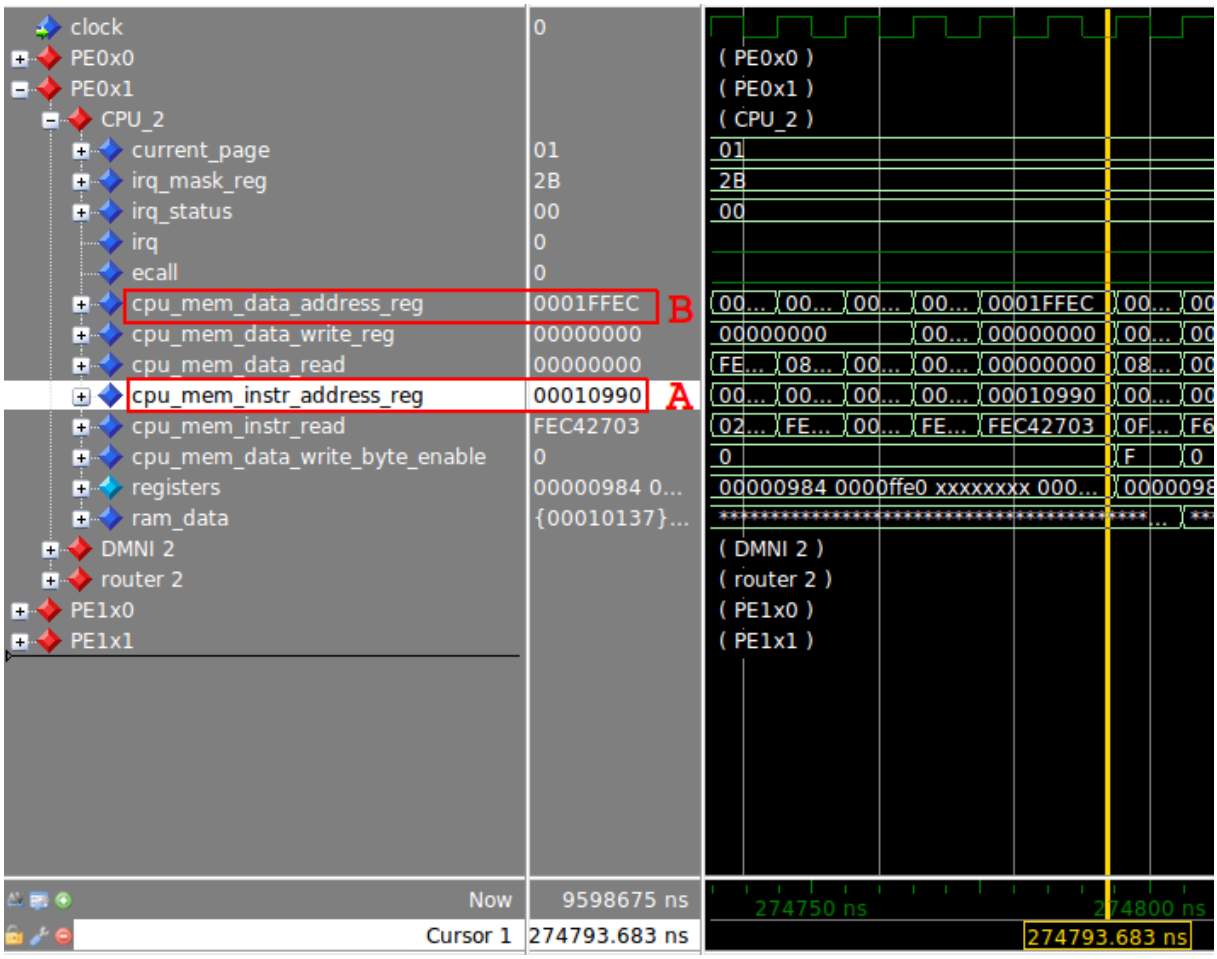

Figura 4.6: Verificação de paginação na forma de onda do caso de teste 1. A região A mostra o endereço de instruções paginado enquanto na B, está o endereço de dados.

```
12  9a4:    00000593                li      a1,0
13
```

Pelo trecho destacado do arquivo "lst" mostrado, é possível notar que a paginação funcionou corretamente e a instrução lida no processador, vista na Figura 4.6, `0xFEC42703`, corresponde àquela do arquivo "lst" em questão.

4.2.2 Validação das interrupções

Para validar o funcionamento das interrupções externas, verifica-se o comportamento da execução do caso de teste 2 por meio de suas formas de onda. Nessa execução dois pontos devem ser levados em conta durante uma interrupção: seu desvio para a rotina de tratamento e seu salvamento e recuperação de contexto.

Logo, espera-se que quando houver uma interrupção, e esta estiver habilitada, a execução vá para a rotina de tratamento de interrupções. Essa rotina está na página do *kernel*,

que é a 0, no endereço `0x3C`. Além disso, é necessário verificar, se o contexto salvo antes da interrupção ser tratada foi recuperado corretamente na execução dessa tarefa após o tratamento.

Na Figura 4.7 é vista a ocorrência de uma interrupção. Na região A, destaca-se a transição de páginas. A aplicação executava nesse PE a tarefa cuja página era a de número 2. No momento que há a interrupção, demonstrada pelo sinal ativo `irq`, a página é mudada para a que possui o *kernel*, que é a 0. Além disso, ocorre o desvio para a rotina de tratamento em `0x3C` que é visto em B. O contexto dos registradores da tarefa que estava executando é visto na região delimitada em C.

O funcionamento do tratamento da interrupção se confirma então pelo apresentado na Figura 4.8. Na região A dessa imagem, vê-se que ao fim do tratamento da interrupção, a página da tarefa que estava sendo executada será retornada. É possível, ainda, ver a restauração do contexto ocorrendo e recuperando aquele mostrado na região C da Figura 4.7.

4.2.3 Validação das chamadas de sistema

Para validar o funcionamento das chamadas de sistema, adota-se um procedimento semelhante ao da interrupção. No entanto, nesse caso busca-se observar a mudança de execução para a rotina de tratamento no endereço `0x44`. Além disso, o contexto não é salvo completamente nessas chamadas, é necessário verificar apenas o salvamento e a restauração do contexto dos registradores mostrados no Código-fonte 4.8.

Código 4.8: Registradores salvos no início de uma chamada de sistema

```
1 system_service_routine:
2         #mscratch must have (old) current value
3         csrrw t6, mscratch, t6  #save t6 in mscratch, then ignore old
    mscratch...
4         la t6, current          #get current value (will be in t6)
5         lw t6, 0(t6)
6
7         # saves minimum context
8         sw x10,  32(t6)         # reg[8]  = a0
9         sw x11,  36(t6)         # reg[9]  = a1
10        sw x12,  40(t6)         # reg[10] = a2
11        sw x13,  44(t6)         # reg[11] = a3
12        sw x14,  48(t6)         # reg[12] = a4
13        sw x15,  52(t6)         # reg[13] = a5
14        sw x16,  56(t6)         # reg[14] = a6
15        sw x17,  60(t6)         # reg[15] = a7
16        sw x1,    0(t6)         # reg[0]  = ra
```

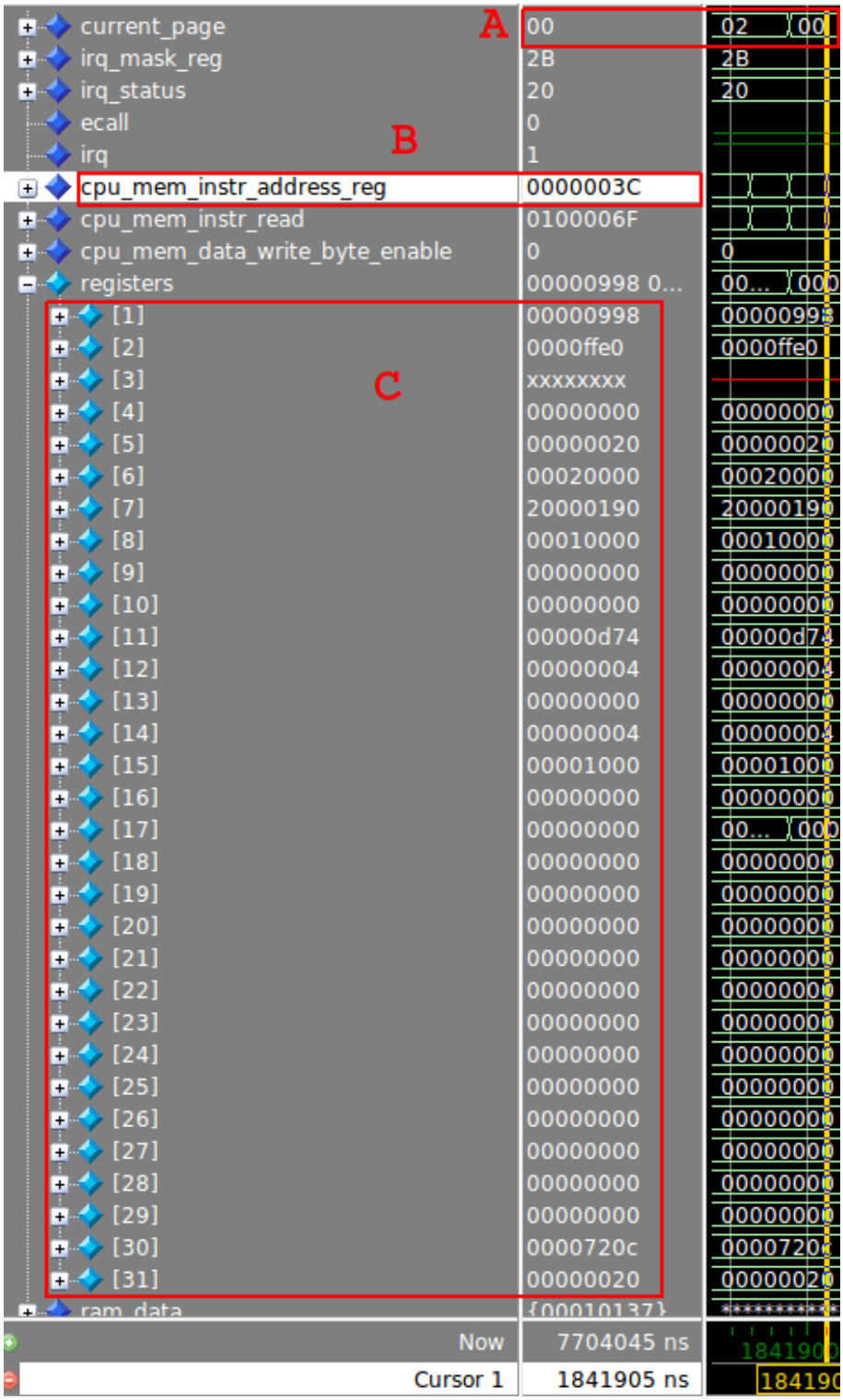

Figura 4.7: Ocorrência de interrupção durante execução da tarefa de página 2 do caso de teste 2.

```
17          sw x2,     4(t6)          # reg[1]  = sp
18          # Save PC:
```

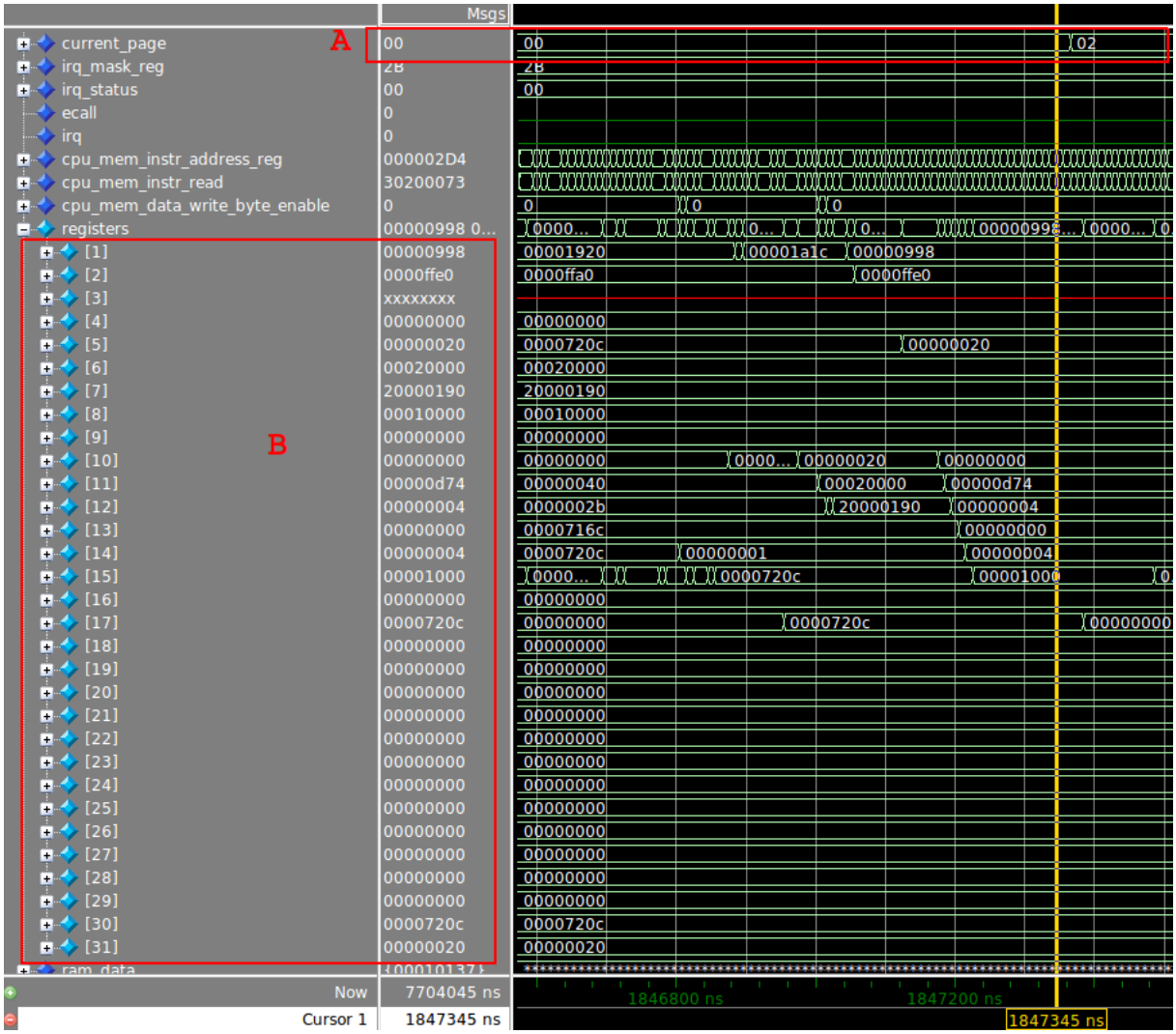

Figura 4.8: Restauração de contexto após o tratamento da interrupção na tarefa de página 2 do caso de teste 2.

```
        csrr t0, mepc
        sw t0, 120(t6)

```

No entanto, uma chamada de sistema pode chamar o escalonador ao fim de sua execução. Quando isso acontece, é necessário salvar outra parte do contexto, como é mostrado no Código-fonte 4.9.

Código 4.9: Registradores possivelmente salvos no fim da execução de uma chamada de sistema

```
.globl  ASM_SaveRemainingContext
ASM_SaveRemainingContext:
        # "current" address is in mscratch
```

```
4         # csrrw a7, mscratch, a7
5         la a7, current
6         lw a7, 0(a7)
7
8         #SAVE ra AND REGISTERS
9         sw x10,  32(a7)         # reg[8]  = a0 # saves SystemCall return
10        sw x8,   24(a7)         # reg[6]  = s0/fp
11        sw x9,   28(a7)         # reg[7]  = s1
12        sw x18,  64(a7)         # reg[16] = s2
13        sw x19,  68(a7)         # reg[17] = s3
14        sw x20,  72(a7)         # reg[18] = s4
15        sw x21,  76(a7)         # reg[19] = s5
16        sw x22,  80(a7)         # reg[20] = s6
17        sw x23,  84(a7)         # reg[21] = s7
18        sw x24,  88(a7)         # reg[22] = s8
19        sw x25,  92(a7)         # reg[23] = s9
20        sw x26,  96(a7)         # reg[24] = s10
21        sw x27, 100(a7)         # reg[25] = s11
22
23        jr ra
24
```

Na Figura 4.9 verifica-se a ocorrência de uma chamada de sistema, no caso de teste 2, durante a execução de uma tarefa que estava na página 2.

Na região A dessa figura, destaca-se a transição de páginas. No momento que há a chamada de sistema, a página é mudada para a página do *kernel*, que é a 0. O desvio da execução para a rotina de tratamento que está no endereço `0x44` é visto na região delimitada em B. O contexto mínimo da tarefa que estava executando é visto nas áreas C e D. Esse contexto armazenado no TCB deverá ser o mesmo ao fim do tratamento da chamada de sistema e execução da tarefa interrompida. Exceto para o caso do registrador x10, que corresponde a `a0`. Esse registrador, por ser utilizado para armazenar o retorno de funções, não tem seu contexto restaurado em caso de retorno à mesma tarefa.

No caso em que o escalonador escolhe para execução uma tarefa diferente da que fez a chamada de sistema, deve-se salvar parte do contexto remanescente da que chamou. Os registradores a serem salvos no TCB estão destacados na Figura 4.10. Esses são os registradores de tipo "s" e o `a0`, que armazena o valor do retorno da função que trata as chamadas de sistema no *kernel*, `Syscall`.

Na Figura 4.11, vê-se que ao fim do tratamento, a tarefa da página 3 foi escolhida para executar. Sendo que quem havia realizado a chamada de sistema, foi a tarefa de página 2.

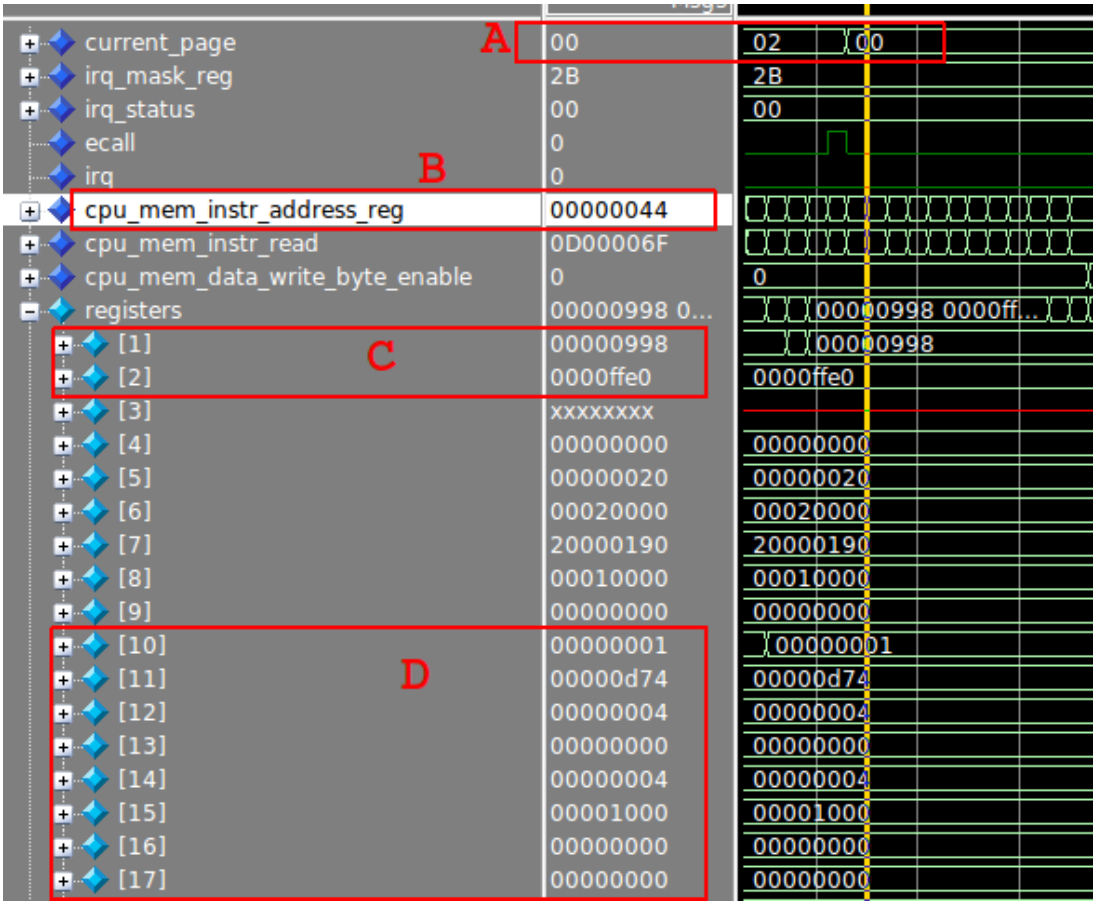

Figura 4.9: Ocorrência de chamada de sistema durante execução da tarefa de página 2 do caso de teste 2.

O funcionamento do tratamento da chamada de sistema se confirma então pelo apresentado na Figura 4.12. A próxima vez que aquela tarefa que havia feito uma chamada de sistema (tarefa de página 2) executa, seu contexto é restaurado com sucesso. A transição para aquela tarefa é vista em A. Já o contexto restaurado, está nas regiões B e C.

4.2.4 Validação geral do sistema

Essa validação consiste, primeiramente, em analisar os arquivos de *log* gerados durante a execução para confirmar o funcionamento do sistema. Depois, trechos das formas de onda dos casos de teste executados são analisadas para verificar o comportamento da simulação. Por fim, faz-se outra análise no funcionamento por meio do *software* de depuração HeMPS_Debugger presente na plataforma.

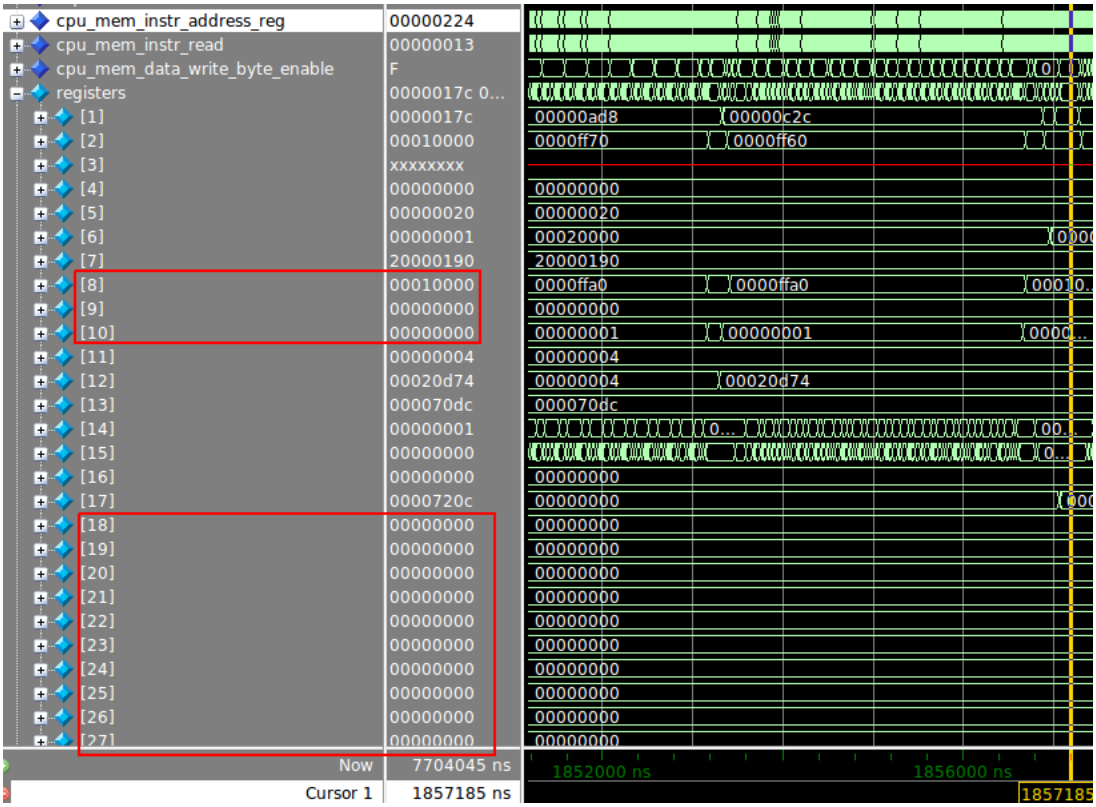

Figura 4.10: Salvamento de parte remanescente do contexto no momento que o escalonador escolhe outra tarefa para a execução.

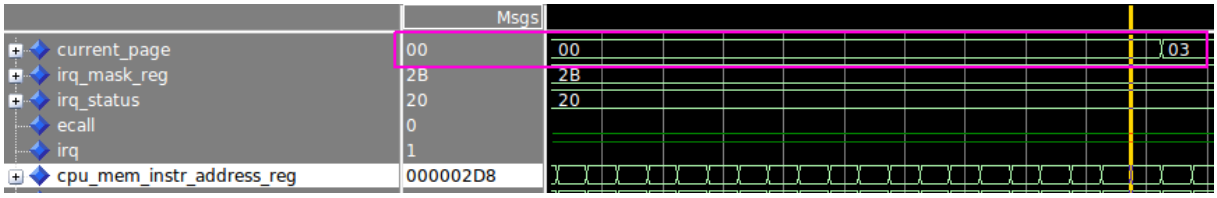

Figura 4.11: Escolha por executar a tarefa de página 3 ao fim do tratamento da chamada de sistema.

Análise dos arquivos de log gerados na execução

Durante a execução de um MPSoC em simulação, um arquivo de *log* é gerado por cada processador. Nele estão contidas informações a respeito do funcionamento de seu *kernel* e dados da tarefa que são impressos por meio da chamada de sistema `Echo`.

Para o caso de teste 1, espera-se, além das mensagens de alocação de tarefas, presentes no *kernel* mestre e nos escravos que foram alocados, o *log* de envio de 250 mensagens pelo processador 0x1 e o recebimento de todas elas pelo processador 1x0, por meio de `Echos` na hora que são enviadas e recebidas. É importante, ainda, analisar o *log* do SMPE que é sempre o processador 0x0. Nessa análise é possível avaliar as alocações de tarefas das

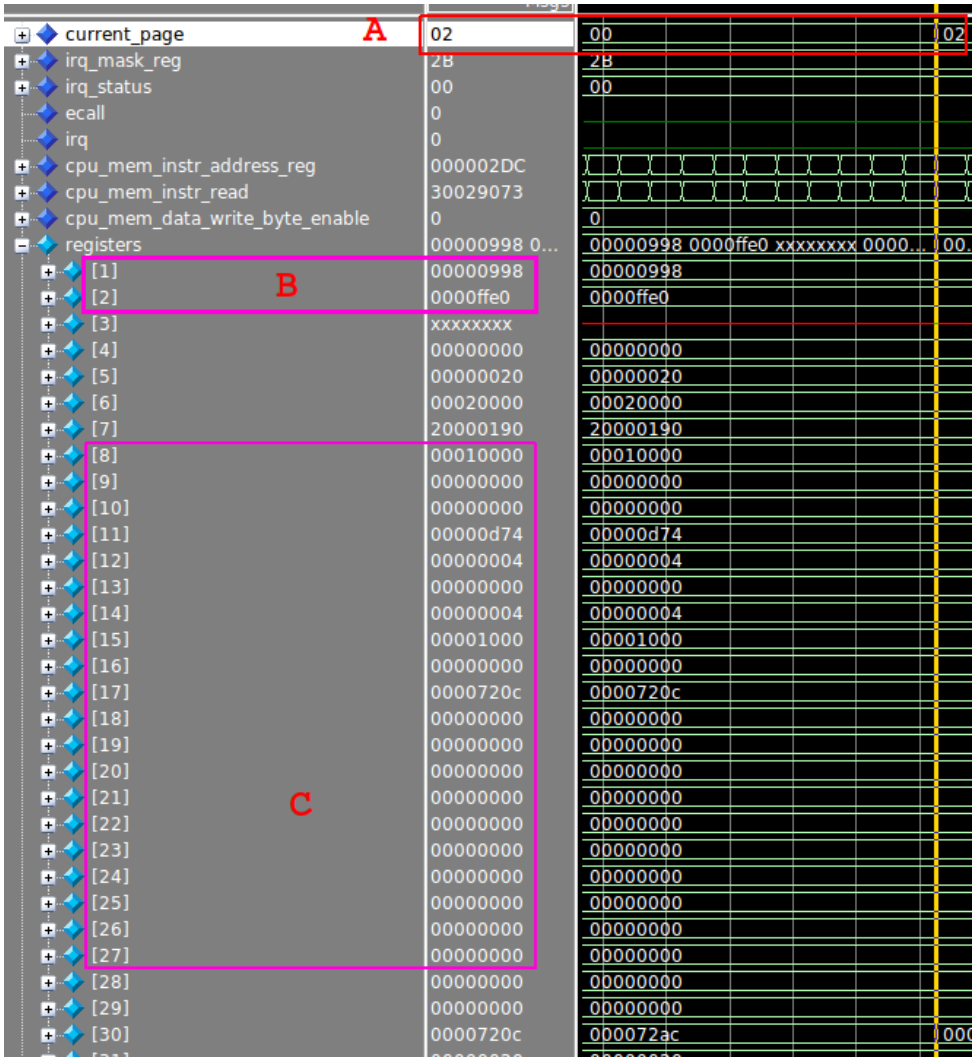

Figura 4.12: Restauração de contexto antes de voltar a executar a tarefa que havia sido interrompida ao realizar uma chamada de sistema.

aplicações de uma simulação.

O Código-fonte 4.10 traz uma análise do arquivo de *log* gerado pelo *kernel* mestre que é executado no processador do PE 0x0. O seu conteúdo é melhor detalhado abaixo.

Código 4.10: Análise do arquivo de *log* do PE 0×0

```
1 This kernel is global master
2 Vai inicializr cluster 0
3 Inicializou mestre global com ID 0
4 Kernel Initialized
5
6 Handle application request from repository
7         Application address: 0x10000000
8         app id: 0
```

```
9
10 Task mapping for task 0 maped at proc 0x00000001
11 Task mapping for task 1 maped at proc 0x00000100
12 App have all its task mapped
13 Task allocation send - id: 0 send to proc: 0x00000001
14
15  -> TASK ALLOCATED from task 0
16 Allocated tasks: 1
17 Task allocation send - id: 1 send to proc: 0x00000100
18
19  -> TASK ALLOCATED from task 1
20 Allocated tasks: 2
21
22  -> send TASK_RELEASE to task 0
23
24  -> send TASK_RELEASE to task 1
25 SIMUL_PROGRESS 100%
26 FINISH 955125
```

Nas linhas de 1 a 4, tem-se informações da inicialização do *kernel* mestre nesse elemento de processamento, além de sua definição como mestre global (SMPE). Ou seja, somente ele tem acesso ao repositório de tarefas. Ainda nessa etapa, o *cluster* de processadores é iniciado. Entre as linhas 6 e 8, vê-se o recebimento de uma requisição de uma aplicação presente no repositório de tarefas. Da linha 10 à linha 13, o mapeamento de tarefas é realizado entre os processadores escravos disponíveis. Como o arquivo de configuração define que seja executada 1 tarefa por elemento de processamento, a tarefa de identificador `id` 0 ("prod_hello_world") é alocada no elemento 0x1 e a tarefa de `id` 1 ("cons_hello_world") é alocada em 1x0, como visto entre as linhas 15 e 20. Após a alocação ser feita, os processadores escravos enviam uma mensagem de resposta e mantém suas tarefas bloqueadas até que, nas linhas 22 e 24, a mensagem `TASK_RELEASE` permita suas execuções. Na região F, tem-se o fim da execução de todas as tarefas dessa aplicação, mostrado em `SIMUL_PROGRESS`, seguido pela quantidade de ciclos de *clock* necessários na execução após `FINISH`.

O Código-fonte 4.11, mostra a análise do arquivo de *log* da tarefa executada no PE escravo 0x1 no caso de teste 1. Nessa análise o conteúdo desse arquivo é melhor detalhado abaixo.

Código 4.11: Análise do arquivo de *log* do PE escravo 0×1 gerado pelo caso de teste 1

```
1 Initializing PE: 0x00000001
2 Slave initialized by cluster address: 0
3 Task id: 0 allocated at 12311
```

```
4 $$$_0x1_0_0_task prod_hello_world started.
5 $$$_0x1_0_0_22592
6 $$$_0x1_0_0_Message sent
7 $$$_0x1_0_0_Message sent
8 ...
9 $$$_0x1_0_0_Message sent
10 $$$_0x1_0_0_948809
11 $$$_0x1_0_0_task prod_hello_world finished.
12 Task id: 0 terminated at 951820
```

Nas duas primeiras linhas do Código-fonte 4.12, tem-se a inicialização do *kernel* escravo nesse elemento de processamento. Nas duas seguintes, tem-se a alocação da tarefa 0 ("prod_hello_world") nele e a impressão, por meio da chamada de sistema `Echo`, de sua inicialização. As linhas de 5 a 10 correspondem à execução da tarefa. Em seu início, a tarefa imprime, pelo `Echo`, o "Tick de inicío" por meio da chamada de sistema `GetTick`. Ou seja, imprime quantos ciclos de *clock* foram executados até então. Em seguida, as mensagens que são enviadas emitem avisos no *log* por meio da chamada `Echo`. Como 250 mensagens são enviadas por essa tarefa, parte desse *log* foi omitido e é denotado por: "...". Ao fim da execução, o "Tick de fim" é impresso de forma a permitir analisar quantos ciclos foram necessários em toda a execução. Por fim, nas duas últimas linhas, a tarefa é finalizada a partir de um aviso de encerramento mostrado via `Echo`. Em seguida, uma mensagem do próprio *kernel* disponibiliza a quantidade de ciclos de *clock* até aquele momento e mostra que a tarefa foi terminada.

No Código-fonte 4.12, tem-se o *log* gerado pelo *kernel* escravo em 1x0. Seu conteúdo é detalhado abaixo.

Código 4.12: Análise do arquivo de *log* do PE escravo 1×0 gerado pelo caso de teste 1

```
1 Initializing PE: 0x00000100
2 Slave initialized by cluster address: 0
3 Task id: 1 allocated at 16525
4 $$$_1x0_0_1_task cons_hello_world started.
5 $$$_1x0_0_1_23322
6 $$$_1x0_0_1_Message received
7 $$$_1x0_0_1_Message received
8 ...
9 $$$_1x0_0_1_Message received
10 $$$_1x0_0_1_949669
11 $$$_1x0_0_1_task cons_hello_world finished.
12 Task id: 1 terminated at 952911
```

Nesse *log*, as duas primeiras linhas correspondem à inicialização do *kernel* escravo no elemento de processamento. Nas linhas 3 e 4, tem-se a alocação da tarefa de `id`

1 ("cons_hello_world"). As linhas de 5 a 10 correspondem à execução da tarefa. No início, a tarefa imprime, o "Tick de inicío", assim como na tarefa alocada no PE 0x1. Em seguida, as mensagens recebidas emitem avisos no *log* por meio da chamada `Echo`. Assim como na tarefa produtora, parte desse *log* foi omitido, devido às 250 mensagens mostradas. Ao fim da execução, o "Tick de fim" é impresso. Por fim, nas duas últimas linhas, a tarefa é finalizada a partir de um aviso de encerramento mostrado via `Echo` seguido da mensagem do próprio *kernel*.

Análise de trechos das formas de onda

Pelas formas de onda geradas no *software* Modelsim durante a simulação do sistema, pode-se analisar seu comportamento durante a execução de um caso de teste. No caso de teste 2, a aplicação "mpeg" é executada e as formas de onda de seus elementos de processamento são analisadas em 2 etapas: na requisição e mapeamento de tarefas que estão no repositório de aplicações (PE 0x0) e na execução dessas tarefas pelos dois elementos de processamento de destino (PEs 0x1 e 1x0).

Primeiramente, há a requisição e mapeamento de tarefas no SMPE. Na Figura 4.13 é observado o envio de uma aplicação do repositório para o elemento de processamento 0x1, por meio do mestre global 0x0.

No tempo `t0` da Figura 4.13, o PE mestre lê o dado `0xFEC42783` do repositório de aplicações. Em `t1`, esse dado é enviado por meio de sua DMNI. Finalmente, em `t2`, o dado é recebido pela DMNI do elemento de processamento destinado.

A execução das tarefas nos elementos de processamento definidos pelo SMPE é dada nesse caso de teste da seguinte forma: como trata-se de uma aplicação com 5 tarefas e no sistema foi definido que cada PE pode executar até 3 tarefas simultaneamente, 2 PEs escravos são usados para executá-la. A Figura 4.14 mostra esses dois elementos de processamento e divide as formas de onda durante a execução em três partes: A, B e C. Na região A, ambos PEs passam pelos procedimentos de inicialização e atribuição de tarefas, que ocorrem na página do *kernel* (página 0), vista no `current_page`. No momento que as aplicações são liberadas para a execução pela mensagem `TASK_RELEASE`, as suas tarefas são inicializadas. Essa execução é vista em B, onde é possível ver a alternância nas páginas em execução, ver chamadas de sistema por meio do sinal `ecall` e ver interrupções por meio do `irq`. Em C, as tarefas já foram finalizadas e o sistema operacional volta para a página do *kernel* onde fica executando a rotina `OS_Idle( )`.

As formas de onda, apesar de permitirem a análise do comportamento do sistema, não são suficientes para comprovar o funcionamento por si só. Para isso, usa-se para depuração os arquivos ".lst" das tarefas analisadas e do *kernel*, juntos dos arquivos de *log*. Na Figura 4.15, compara-se a instrução buscada da memória por meio do sinal

Figura 4.13: Análise de formas de onda quando ocorre envios do repositório de aplicações para o elemento de processamento destino no caso de teste 2.

`cpu_mem_instr_address_reg` com a esperada no arquivo "lst", assim como foi feito para verificar o funcionamento da paginação.

Na região A da Figura 4.15, tem-se o arquivo da tarefa "start", que apresenta a instrução `0xFEC42783` para o endereço `0xD4`. Pelo arquivo de *log*, mostrado no Código-fonte 4.13 do PE em análise, foi verificado que "print" era a tarefa de `id` 0 e, por isso, foi alocada na página 1, já que a página 0 se destina ao *kernel*. A tarefa "start" com `id` 1, foi alocada na página 2 e por último, "iquant", com `id` 2, foi alocada na página 3. Pela forma de onda analisada na área B da Figura 4.15, percebe-se que a busca pela instrução com endereço `0x000200D4`, corresponde ao endereço `0xD4` da página 2, que é a instrução esperada da tarefa "start".

Código 4.13: Arquivo de *log* do PE 0x1 do caso de teste 2

```
1 Initializing PE: 0x00000001
2 Slave initialized by cluster address: 0
3 Task id: 0 allocated at 22322
4 Task id: 1 allocated at 26637
5 Task id: 2 allocated at 31355
6 $$$_0x1_0_0_MPEG Task PRINT start:
```

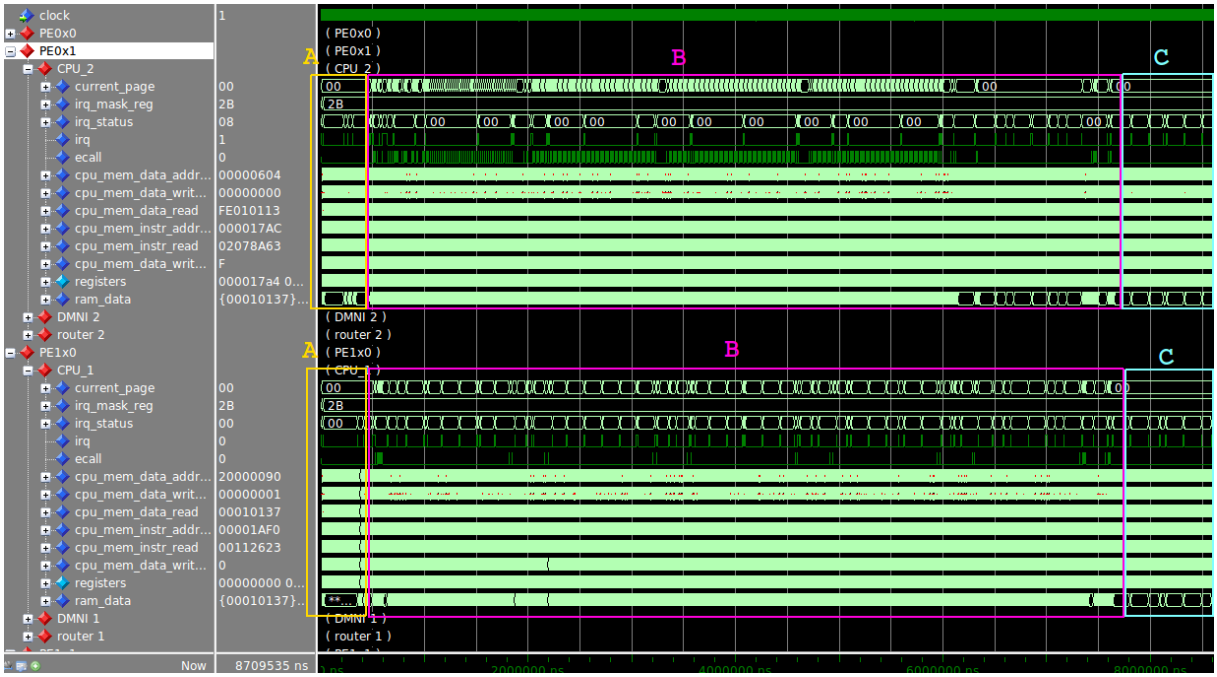

Figura 4.14: Análise de formas de onda da execução do caso de teste 2.

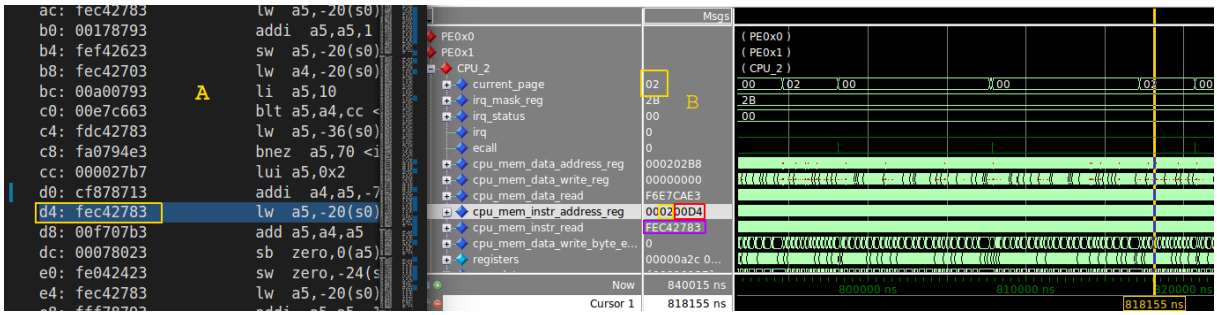

Figura 4.15: Comparação das instruções buscadas nas formas de onda da execução da tarefa “start” do caso de teste 2, com o seu arquivo “lst”.

```
$$$_0x1_0_1_MPEG Task A start:
$$$_0x1_0_1_53896
$$$_0x1_0_1_T1
$$$_0x1_0_1_59196
$$$_0x1_0_1_T2
$$$_0x1_0_1_64893
$$$_0x1_0_2_MPEG Task C start: iquant
$$$_0x1_0_2_71483
$$$_0x1_0_1_T1
$$$_0x1_0_1_74133
$$$_0x1_0_1_T2
$$$_0x1_0_1_78448
```

Mais ainda, o processador Clarvi possui um módulo de depuração “clarvi_debugger”, pelo qual pode-se verificar as instruções que estão sendo executadas por um processador definido nesse módulo. Essa análise é mostrada na Figura 4.16.

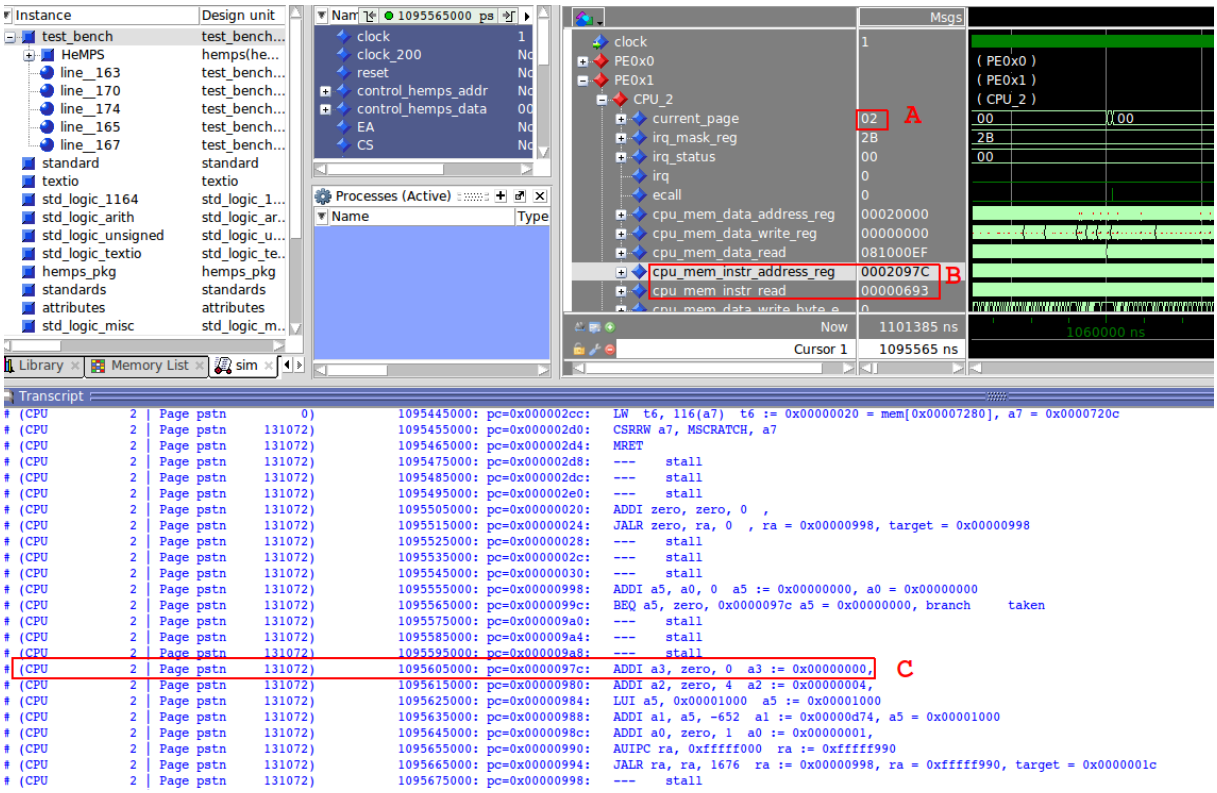

Figura 4.16: Comparação das instruções buscadas nas formas de onda da execução da tarefa “start” do caso de teste 2, com as informações de depuração impressas pelo processador na seção “Transcript” do Modelsim.

Na região A da Figura 4.16, tem-se a página corrente da execução, que corresponde à tarefa “start”. Na B, tem-se a instrução corrente, que está na posição `0x97C` da página 2 no instante 1.095.565ns. Na linha marcada em C, tem-se 40ns depois, a execução dessa instrução. A CPU escolhida para imprimir informações é a de número 2. O *offset* da página é de “131.072” bytes, sendo cada página de 64*kbytes*. A instrução é a da posição `0x97C`, no tempo 1.095.605ns, que é a `addi a3, zero, 0`. Essa instrução corresponde, à `li a3, 0` mostrada no arquivo “lst” dessa tarefa na linha 10 do Código-fonte 4.14.

Código 4.14: Arquivo “lst” da tarefa “start” destacando a instrução na posição `0x97C`

```
958:    00000693        li      a3,0
95c:    00000613        li      a2,0
960:    00000593        li      a1,0
964:    00300513        li      a0,3
```

```
968:   fffff097          auipc   ra,0xfffff
96c:   6b4080e7          jalr    1716(ra) # 1c <SystemCall>
970:   00050793          mv      a5,a0
974:   fef42423          sw      a5,-24(s0)
978:   00000013          nop
97c:   00000693          li      a3,0
980:   00400613          li      a2,4
984:   000027b7          lui     a5,0x2
```

Verificação do funcionamento usando o software de depuração da plataforma

Por meio do *software* "HeMPS_Debugger", é possível ver o mapeamento de tarefas, o envio e o recebimento de mensagens em tempo de simulação, e os logs gerados pelos PEs. Além disso, pode-se ver uma execução com um nível mais alto de abstração de forma a facilitar depurações e entender melhor o seu funcionamento.

Na Figura 4.17, o caso de teste 3 é executado no software de depuração. Para isso, foi inicializada uma seção de depuração no programa com o arquivo "platform.cfg" contido na pasta "*debug*" desse caso de teste. Nesse momento, foram executados 3.724 ciclos de *clock*, vistos no destaque da área A. Na região B, vê-se que o serviço em execução nesse momento, é o de inicializar os PEs escravos por meio do `INITIALIZE_SLAVE`. Na região C da imagem, são mostrados os PEs do caso de teste em análise. As setas vermelhas indicam as comunicações que estão ocorrendo entre os PEs.

Na Figura 4.18, pode-se observar como as 7 tarefas das duas aplicações executadas pelo caso de teste 3 são distribuídas pelos elementos de processamento que podem executar até 2 tarefas de forma simultânea. Os PEs 0x1, 0x2 e 1x0 executam as tarefas da aplicação mpeg, enquanto que o pe 1x1 executa as duas tarefas da aplicação "hello_world_app".

Outro dado interessante que pode ser analisado na execução usando essa ferramenta de depuração é a visão geral da comunicação entre os PEs. A Figura 4.19 apresenta a porcentagem de *flits* que passaram por cada PE. É notório que a maior quantidade está presente nas tarefas da aplicação "mpeg" que iniciaram sem tempo de atraso, de acordo com o arquivo de configuração. Mais ainda, como a aplicação "hello_world_app" está sendo executada inteiramente dentro do PE 1x1, não se espera muita comunicação desse PE com outros, a não ser pelo reencaminhamento de *flits*.

Figura 4.17: Execução da simulação do caso de teste 3 no *software* de depuração da HeMPS.

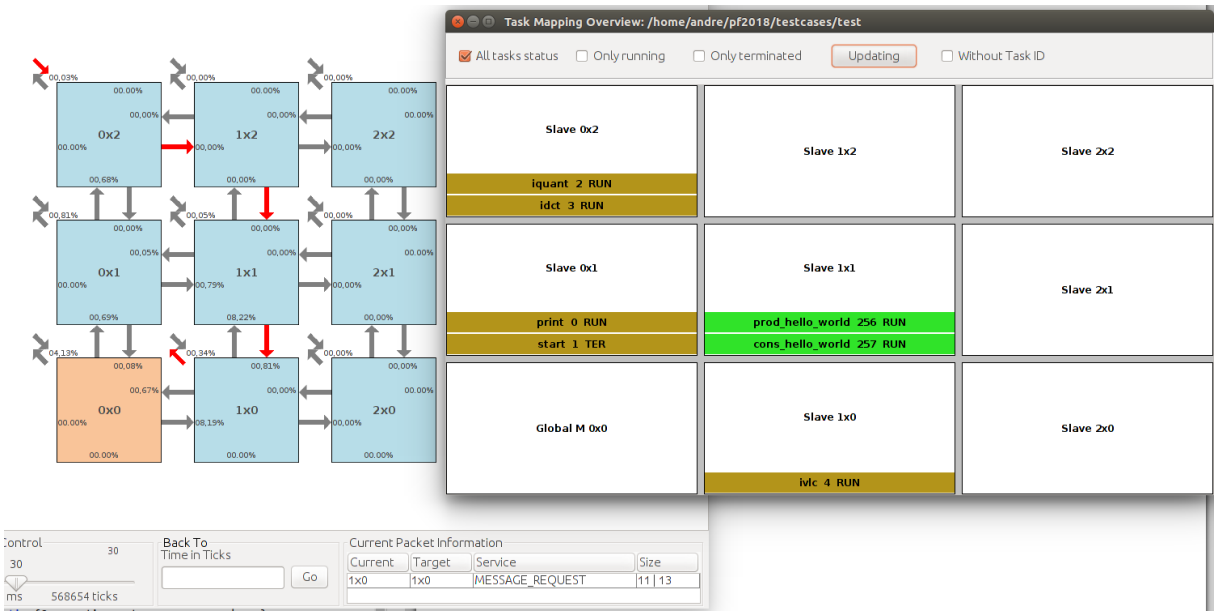

Figura 4.18: Verificação do mapeamento de tarefas do caso de teste 3 no *software* de depuração da HeMPS.

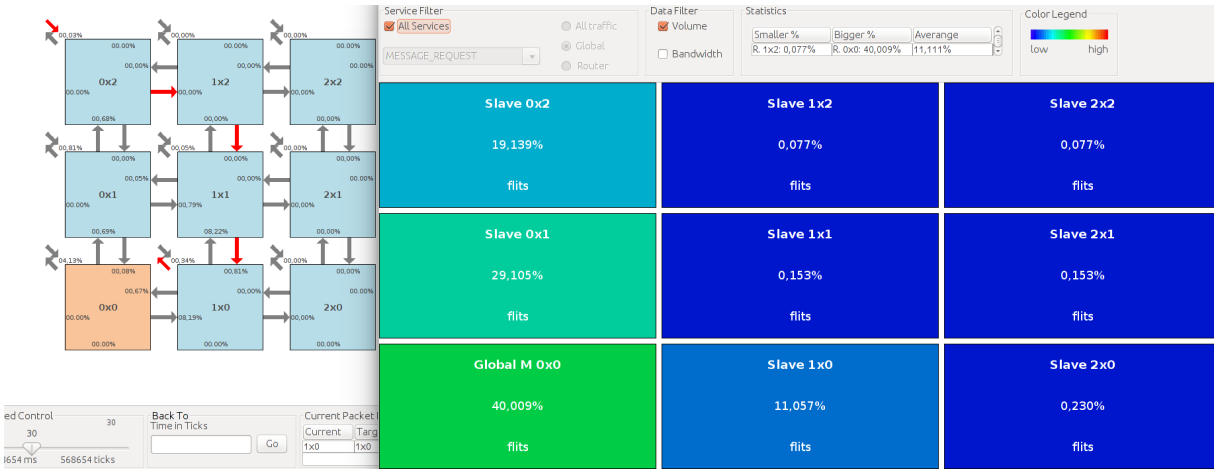

Figura 4.19: Visão geral da comunicação durante execução do caso de teste 3 no *software* de depuração da HeMPS.

4.3 Comparativo entre os processadores

Nessa seção faz-se um comparativo entre as duas versões da HeMPS, isto é, a versão original que possui processador Plasma e a nova versão com processador Clarvi.

A Tabela 4.1 compara a execução dos 6 casos de testes nas duas versões. As execuções foram realizadas em uma máquina com processador Intel i7-4720HQ de 2.6GHz e 12GB de RAM DDR3 no sistema operacional Ubuntu 16.04. Na tabela, o tempo de execução refere-se à quantos ciclos de *clock* o processador leva para executar determinada aplicação, sendo que um ciclo de *clock* no *testbench* da HeMPS tem 10ns. Ou seja, esses processadores são testados considerando uma frequência fixada, para fins de simulação, em 100MHz. Já o tempo de simulação, refere-se ao tempo, em minutos e segundos, que a máquina utilizada levou para simular essa execução até seu fim. A coluna Ganho representa, em porcentagem, a diferença de resultados do processador Clarvi em relação ao processador Plasma.

Casos de teste	**Tempos de execução (x 10ns)**			**Tempos de simulação**		
	Plasma	**Clarvi**	**Ganho**	**Plasma**	**Clarvi**	**Ganho**
1	1.201.148	955.125	20.48%	1m 37,288s	1m 17,999s	9.55%
2	981.704	765.291	22.04%	1m 22,062s	1m 7,570s	17.66%
3	2.841.153	2.060.435	27.48%	4m 11,637s	5m 38,379s	-34.47%
4	10.158.916	6.268.169	38.30%	14m 6,538s	8m 9,858s	42.13%
5	9.582.293	5.879.486	38.64%	22m 25,007s	16m 40,938s	25.58%
6	3.945.643	3.339.749	15.32%	6m 34,159s	4m 24,988s	32.77%

Tabela 4.1: Tabela comparativa entre as aplicações executadas pela HeMPS na versão original e na versão com processador Clarvi

Pela análise dos resultados dessa tabela, percebe-se que o processador Clarvi levou menos ciclos para executar todos os casos de teste. Essa justificativa deve-se a uma quantidade de maior de estágios no pipeline desse processador. Enquanto o plasma possui um pipeline de 3 estágios, o Clarvi possui 6 estágios.

Apesar de que é esperada uma simulação mais demorada utilizando duas linguagens diferentes ao mesmo tempo, em quase todos os casos de teste o Clarvi levou menos tempo, o que se explica pelo tempo de execução menor. A exceção ocorreu no caso de teste 3, no qual o processador Plasma foi cerca de 1 minuto e 26,742 segundos mais rápido na simulação, apesar de possuir 780.718 ciclos de *clock* a mais na execução. Essa ocorrência é explicada pois o processador Plasma possui um mecanismo que o coloca em estado de espera, interrompendo seu *clock*, quando nenhuma tarefa está executando. Nesse momento, há menos uso de recursos da máquina hospedeira, na simulação. O Clarvi no entanto, além de executar em uma plataforma que integra duas linguagens de descrição de *hard-*

ware diferentes, não possui esse mecanismo e acaba consumindo recursos do hospedeiro, o que pode tornar sua simulação mais lenta.

Logo, a partir das validações e comparações realizadas nesse capítulo e do detalhamento da migração de arquitetura exposto no capítulo 3, o próximo capítulo conclui as ideias do projeto e cita possíveis trabalhos futuros.

Capítulo 5

Conclusão

Esse capítulo traz conclusões a respeito do desenvolvimento e dos resultado apresentados nos capítulo 3 e 4, respectivamente. Na seção 5.1, comparam-se os objetivos propostos e a solução final. Em 5.2, são apresentados pontos que merecem atenção e podem ser abordados em trabalhos futuros.

5.1 Conclusões

Como objetivo central desse estudo, buscava-se alterar a plataforma HeMPS. Essa modificação consistiu em gerar MPSoCs contendo processadores de arquitetura RISC-V. Como metodologia, foram propostas 4 etapas. A primeira etapa foi a escolha do novo processador a ser integrado. Depois, foi necessário fazer a integração do processador no PE, levando em conta o funcionamento com a DMNI e a memória presente. Em seguida, foram feitas mudanças na geração de código objeto a ser executada na nova arquitetura. Por fim, houveram alterações no sistemas operacionais presentes nos PEs.

A escolha do novo processador levou a uma implementação que apesar de ser simples e de ter um bom viés acadêmico com a parte de mensagens de depuração, possui um ponto negativo que é a falta das instruções de multiplicação, resto e divisão, que foram implementadas. Dessa forma, foi necessário conhecer a estrutura de funcionamento do processador. Além disso, foi necessário modificar o processador para a integração de funcionalidades básicas como paginação, interrupção e chamadas de sistema. Em [8], são abordadas formas de implementar os mecanismos de paginação e chamadas de sistema via software nessa mesma plataforma para um diferente processador.

A integração do processador Clarvi na plataforma alvo exigiu um empacotador para interfacear o sistema que é escrito em VHDL e o processador, que é implementado em SystemVerilog. Essa etapa exigiu atenção em alguns pontos. A mudança na quantidade de ciclos executados para uma mesma instrução fez com que o acesso ao repositório ficasse

incorreto em alguns casos. Então, um mecanismo de controle desses acessos foi definido no elemento de processamento. Outra mudança necessária foi a interface entre o processador e a memória. Devido à diferença de arquiteturas Von Neumann e Harvard presentes no processador original e no novo, respectivamente, foram necessárias implementações de mecanismos no PE para controlar essa interface. Outra mudança considerada nessa etapa foi a paginação. Na versão original, esta era controlada pelo próprio processador. Na nova versão, foi usado um registrador mapeado em memória de forma que esse controle ficasse a cargo do empacotador do processador.

Quanto à etapa de modificação na geração de código objeto a ser executado, foi usada uma *toolchain* que apresentava as mesmas ferramentas presentes na original. A diferença se dá apenas na arquitetura alvo para a qual o código objeto é gerado.

As alterações nos sistemas operacionais foram mais evidentes no entanto. Por haver diversas rotinas que dependiam da arquitetura utilizada, foi necessária a reescrita do seu *boot* e modificações em seu *kernel*.

Logo, todas as etapas foram executadas como o planejado. Como resultado, pode-se observar, no capítulo 4, a validação do sistema completo e em pontos específicos. Mais ainda, além dessa nova versão se mostrar funcional, foi obtida uma melhora de desempenho quando comparada à original. Essa melhora é explicada pela diferença de quantidade de estágios de pipeline entre as duas versões, o que configura uma maior vazão no processador Clarvi para uma mesma frequência de *clock*.

5.2 Trabalhos futuros

Como trabalhos futuros podem-se citar os seguintes projetos:

- criar mais casos de teste para garantir a validade de todas as funcionalidades;
- implementar uma versão em SystemC de processador com arquitetura RISC-V;
- permitir a parametrização para gerar o sistema em outras linguagens. A versão com o novo processador não é capaz de gerar versões em SystemC, que antes era presente no sistema;
- permitir a geração parametrizada de MIPS ou RISC-V. Isto é, por meio do arquivo YAML o usuário deverá ser capaz de escolher com qual arquitetura ele quer gerar o MPSoC;
- incorporar outros processadores de arquitetura RISC-V. A incorporação de novos processadores com mesmas características pode permitir a geração de MPSoCs heterogêneas;

- permitir a geração de plataformas heterogêneas contando com a presença de processadores de arquiteturas MIPS e RISC-V;
- prototipar essa versão do sistema em FPGA;
- fazer uma implementação otimizada das instruções de multiplicação e divisão no processador Clarvi;

Referências

[1] 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, 2, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

[2] 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

[3] Ruaro, Marcelo: *Self-adaptive qos at communication and computation levels for many-core system-on-chip.* Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2018. `http://tede2.pucrs.br/tede2/handle/tede/7946`. 1, 8, 9, 21, 22, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36

[4] 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. 1, 9, 21

[5] *Grupo de apoio ao projeto de hardware.* `https://corfu.pucrs.br/tikiwiki/tiki-index.php`. Acessado em 25/07/2018. 2

[6] Waterman, Andrew: *Design of the RISC-V Instruction Set Architecture, Volume I: UserLevel ISA, Version 2.0.* Electrical Engineering and Computer Sciences University of California at Berkeley, janeiro 2016. 2, 36, 37, 64, 77, 78

[7] Waterman, Andrew, Yunsup Lee, David A. Patterson e Krste Asanovic: *The RISC-V Instruction Set Manual, Volume I: UserLevel ISA, Version 2.0*, volume 1. Electrical Engineering and Computer Sciences University of California at Berkeley, maio 2014. 3, 38

[8] 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, 4, 22, 23, 35, 36, 44, 116

[9] Sepúlveda Flórez, Martha Johanna: *Projeto de estruturas de comunicação intrachip baseadas em noc que implementam serviços de qos e segurança.* São Paulo, Escola Politécnica, Universidade de São Paulo, 2011. http://www.teses.usp.br/teses/disponiveis/3/3140/tde-04112011-140055/pt-br.php. 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17

[10] Bansal, Deepti: *Comparative study of various systems on chips embedded in mobile devices.* National Conference on Emerging Trends in Electrical, Instrumentation & Communication Engineering, Volume: 4, Número: 7, 2013. 6

[11] *Why snapdragon processors?* https://developer.qualcomm.com/get-started/why-snapdragon-processors. Acessado em 17/04/2018. 6

[12] *Qualcomm® snapdragon™ 845 mobile platform.* https://www.qualcomm.com/documents/snapdragon-845-mobile-platform-product-brief. Acessado em 17/04/2018. 7

[13] Saleh, Resve, Steve Wilton, Shariar Mirabbasi, Alan Hu, Mark Greenstreet, Guy Lemieux, Partha Pratim Pande, Cristian Grecu e Andre Ivanov: *System-on-chip: Reuse and integration.* Proceedings of the IEEE, Volume: 94, Número: 6, Junho 2006. 7, 8

[14] 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

[15] Masaca Soccol, Celso e Giovani Zucolotto: *Implementação do gnu chess em uma plataforma multiprocessada com arquitetura de comunicação baseada em rede intrachip.* Porto Alegre, Curso de Bacharelado em Ciência da Computação, Faculdade de informática, Pontifícia Universidade Católica do Rio Grande do Sul, 2006. http://www.inf.pucrs.br/~calazans/publications/2006_TCC_Celso_Giovani.pdf. 11

[16] Hojabr, R., M. Modarressi, M. Daneshtalab, A. Yasoubi e A. Khonsari: *Customizing clos network-on-chip for neural networks.* IEEE Transactions on Computers, 66(11):1865–1877, Nov. 2017, ISSN 0018-9340. doi.ieeecomputersociety.org/10.1109/TC.2017.2715158. 14

[17] *Hemps multiprocessor system on chip.* http://www.inf.pucrs.br/hemps/index.html. Acessado em 04/12/2017. 22, 89

[18] *Plasma - most mips i(tm) opcodes.* https://opencores.org/project,plasma. Acessado em 04/12/2017. 22, 23

[19] Fochi, Vinicius Morais: *Técnicas de tolerância a falhas aplicadas a redes intrachip.* 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/6140. 22

[20] Castilhos, Guilherme Machado de: *Gerenciamento térmico e energético em mpsocs*. 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. 23, 34

[21] Castilhos, Guilherme Machado de: *Gerência distribuída de recursos em mpsocs - mapeamento e migração de tarefas*. Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2013. http://tede2.pucrs.br/tede2/handle/tede/5208. 24

[22] Carara, Everton Alceu: *Serviços de comunicação diferenciados em sistemas multiprocessados em chip baseados em redes intra-chip*. 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. 24, 25

[23] Chawade, Shubhangi D., Mahendra A Gaikwad e Rajendra M. Patrikar: *Review of xy routing algorithm for network-on-chip architecture*. International Journal of Computer Applications, 43(21), abril 2012. 24

[24] Guindani, Guilherme Montez: *Mecanismo de controle de qos através de dfs em mpsocs*. Porto Alegre, Programa de Pós-Graduação em Ciência da Computação, Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, 2014. http://tede2.pucrs.br/tede2/handle/tede/5262. 30

[25] Yan, Yonghong: *Lecture 07: Risc-v isa*. Oakland CA, EUA, CSE 564 Computer Architecture Summer, Department of Computer Science and Engineering, Oakland University, 2017. 37

[26] Skordal, Kristian Klomsten: *The potato processor*. https://github.com/skordal/potato. 38, 39

[27] Nguyen, Quan M.: *Software tools - risc-v foundation*. https://riscv.org/software-tools/. 38

[28] OpenCores: *Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*. OpenCores, 2010. 39

[29] *The berkeley out-of-order risc-v processor*. https://github.com/ucb-bar/riscv-boom. Acessado em 29/03/2018. 39

[30] Celio, Christopher, David A. Patterson e Krste Asanović: *The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor*. Relatório Técnico UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html. 39

[31] *Coremark an eembc benchmark*. https://www.eembc.org/coremark/index.php. Acessado em 21/07/2018. 39

[32] *Clarvi - a risc-v processor.* https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/exercise-clarvi.html. Acessado em 29/03/2018. 39

[33] *Mentor, a siemens business, leads in electronic design automation software - mentor graphics.* https://www.mentor.com/. Acessado em 23/06/2018. 54, 96

[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. 55

[35] Waterman, Andrew e Krste Asanovic: *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 1.10*, volume 2. Electrical Engineering and Computer Sciences University of California at Berkeley, maio 2017. 64, 65

[36] Patterson, David e Andrew Waterman: *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta. Primera Edición, 1.0.5.* Strawberry Canyon, 2018. 78, 79