PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTEGRAÇÃO DE NOVOS PROCESSADORES EM ARQUITETURAS MPSOC: UM ESTUDO DE CASO

EDUARDO WEBER WÄCHTER

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Gehm Moraes

Porto Alegre, Brasil
2011

Dados Internacionais de Catalogação na Publicação (CIP)

W114i Wächter, Eduardo Weber
Integração de novos processadores em arquiteturas MPSOC : um estudo de caso / Eduardo Weber Wächter. – Porto Alegre, 2011.
92 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fernando Gehm Moraes.

1. Informática. 2. Mulitprocessadores. 3. Arquitetura de Computador. I. Moraes, Fernando Gehm. II. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"A Integração de Novos Processadores em Arquiteturas MPSoC: Um Estudo de Caso"**, apresentada por Eduardo Weber Wächter, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovada em 23/03/2011 pela Comissão Examinadora:

Prof. Dr. Fernando Gehm Moraes – Orientador — PPGCC/PUCRS

Prof. Dr. Ney Laert Vilar Calazans - — PPGCC/PUCRS

Dr. Alexandre de Morais Amory – — FACIN/PNPD

Homologada em 24/05/11, conforme Ata No. 008 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Gostaria de primeiramente agradecer aos meus pais, talvez meus maiores incentivadores a sempre estudar. Eles que têm me apoiado muito neste caminho (financeiramente também). À minha namorada, Geordana Cornejo Pontelli, por me incentivar e sempre me colocar pra cima e, principalmente, me agüentar este tempo todo! Te amo!

Tenho muito a agradecer ao meu orientador, que é um exemplo de pesquisador e professor. Obrigado pelas ótimas e produtivas reuniões. Valeu Moraes!

Aos colegas e amigos de laboratório, obrigado pelas boas gargalhadas e cervejas! Valeu Thomas, Castilhos, Luigi, Guindani, Raupp, Julian, Ost, Leonel, Edson, Tales, Matheus, Rafael, Carara.

Ao Mandelli, meu companheiro durante o mestrado, valeu pela ajuda e por me agüentar, “Cára”!

Aos integrantes e amigos do FBC, o meu muito obrigado por me tirar da rotina quando eu precisava, e algumas vezes quando eu não precisava também!

Finalmente e não menos importante, gostaria de agradecer a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), que foi o órgão que ofereceu suporte financeiro para o desenvolvimento deste trabalho.

Ao professor Dr. Ney Laert Vilar Calazans e ao Dr. Alexandre de Morais Amory que aceitaram avaliar este trabalho e muito contribuíram no desenvolver deste.

INTEGRAÇÃO DE NOVOS PROCESSADORES EM ARQUITETURAS MPSOC: UM ESTUDO DE CASO

RESUMO

Com o aumento da densidade de transistores em um mesmo circuito integrado, possibilitou-se o desenvolvimento de sistemas computacionais completos em um único chip (*Systems-on-Chip*). Atualmente, o poder computacional exigido pelas aplicações freqüentemente demanda uma arquitetura SoC com mais de um processador. Surgiram então as arquiteturas multiprocessadas, denominadas MPSoCs (*Multiprocessor Systems-on-Chip*). As arquiteturas MPSoCs encontradas na literatura não apresentam grande variedade de tipos de núcleo de processamento. Apesar deste fato, a literatura no tema de MPSoCs argumenta com freqüência que MPSoCs heterogêneos (mais de um tipo de núcleo de processamento) apresentam um melhor desempenho. Um dos problemas para desenvolver arquiteturas heterogêneas é a dificuldade de se integrar processadores em plataformas MPSoC. Este trabalho tem por objetivo suprir a lacuna de integração de processadores pré-validados, adaptando uma plataforma de hardware e software para um novo processador. Como ponto de partida para o trabalho utiliza-se uma plataforma MPSoC estado-da-arte homogênea. Esta plataforma é modificada, tornando-se possível a sua prototipação, resultando na primeira contribuição desta Dissertação. A segunda contribuição reside no desenvolvimento de um novo elemento de processamento para o mesmo MPSoC, utilizando como estudo de caso o processador MB-Lite – que adota a arquitetura Microblaze. Além do desenvolvimento deste módulo, o sistema operacional responsável pela execução multitarefa é portado para este processador, identificando-se os mecanismos dependentes de arquitetura no mesmo. Por fim, são apresentados resultados da integração deste processador, e a avaliação do MPSoC heterogêneo com aplicações sintéticas executando tarefas em processadores distintos (Plasma e MB-Lite), validando-se assim a proposta de integração de novos processadores em arquiteturas MPSoC.

Palavras Chave: MPSoCs heterogêneos baseados em NoC, Processadores Embarcados, Prototipação em FPGAs, MPSoCs em FPGA.

THE INTEGRATION OF NEW PROCESSORS IN MPSOC ARCHITECTURE: A CASE STUDY

ABSTRACT

The increase in transistor density on a single integrated circuit enables the design of complete computational systems on a single chip (*Systems-on-Chip*). Today, the computational power required by applications demands SoC architectures with more than one processor. Appeared then architectures called MPSoCs (*Multiprocessor Systems-on-Chip)*. However, MPSoCs found on literature do not present different kinds of processors cores. Despite this fact, the literature on MPSoCs argues that heterogeneous MPSoCs (more than one kind of processor core) have a better performance. One of the problems to design heterogeneous architectures is the difficulty to integrate processors on MPSoCs platforms. The present work has the objective to fulfill this project gap, the integration of pre-validated processors, adapting a hardware and software platform for a new processor. As start point, this work uses a state-of-the-art homogeneous MPSoC platform. This platform is modified, making possible its prototyping, resulting in the first contribution of this Dissertation. The second contribution lies in developing a new processing element for the same MPSoC, using as case study MB-Lite processor - which adopts the MicroBlaze architecture. Besides the development of this module, the operating system responsible for implementing multitasking is ported to this processor, identifying the architecture dependent mechanisms. Finally, we present results of the integration of this processor, and evaluation of heterogeneous MPSoC with synthetic applications performing tasks on different processors (Plasma and MB-Lite), thus validating the proposed integration of a new processor in an MPSoC architecture.

Key-words: NoC-based heterogeneous MPSoCs, Embedded Processors, FPGA prototyping, MPSoCs in FPGAs.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ALUT	Adaptive Look-Up Table
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ARM	Advanced RISC Machine
ARP	Address Resolution Protocol
BRAM	Block Random Access Memory
CAD	Computer-Aided Design
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DSP	Digital Signal Processor
FF	Flip-Flop
FFT	Fast Fourier Transform
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GPP	General Purpose Processor
GUI	Graphical User Interface
HW	Hardware
IP	Intellectual Property ou Internet Protocol
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
JTAG	Joint Test Action Group
kB	*kiloBytes*
LUT	Look-Up Table
MAC	Media Access Control ou Multiply-Accumulate
MPEG	Moving Picture Experts Group
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
PE	Processing Element
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction, Multiple Data
SO	Sistema Operacional
SoC	System-on-Chip
SW	Software
TCP/IP	Transmission Control Protocol / Internet Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Com o crescente aumento de número de transistores em um único circuito integrado, tornou-se possível desenvolver sistemas completos em um único chip. Estes sistemas são chamados de SoCs (System-on-Chip). Um SoC é um circuito integrado que implementa a maioria, senão todas, as funções de um sistema eletrônico completo [JER05a]. Uma característica de um SoC é a complexidade de projeto e validação. Hoje um SoC contém centenas de milhões de transistores. Um SoC contém vários elementos de processamento, gerando assim um grande poder computacional em um único circuito integrado. Estes elementos de processamento (PEs, *processing elements*) podem ser processadores de propósito geral (GPPs), processadores dedicados (e. g. DSP e VLIW), ou blocos de hardware dedicados (IP, *Intelectual Property*). A arquitetura de um SoC é feita geralmente visando uma aplicação e não um circuito integrado de propósito geral [JER05a].

Os consumidores de sistemas embarcados exigem produtos com desempenho cada vez mais elevado, baixo consumo de energia e baixo custo. Os projetistas devem atender a estes requisitos conflitantes de projeto, além de um *time-to-market* curto. Para suprir estes requisitos de projeto, surgiram arquiteturas com mais de um processador, chamadas de MPSoC (Multiprocessor Systems-on-Chip). MPSoCs já deixaram de ser apenas tema de pesquisa e encontram-se implementações comerciais na área de telefonia celular [LIM09], entretenimento, jogos eletrônicos [KIS06], entre outros. Entre os temas de pesquisa em MPSoCs, atualmente, pode-se citar: mapeamento de tarefas [CAR09b], migração de tarefas [SHE09], modelagem no nível de transação [MON08], modelagem com atores [DEN09], técnicas para estimar potência dissipada [GUI08], entre outros.

MPSoCs podem ser divididos quanto ao meio de interconexão que empregam: *(i)* barramentos ou *(ii)* redes intra-chip (NoCs, do inglês *Networks-on-Chip*). Quando o barramento é utilizado como interconexão, este pode ser monolítico ou hierárquico (com a utilização de pontes). O problema desta arquitetura está diretamente relacionado ao número de módulos conectados nela. Quanto mais módulos são conectados, maior a perda na taxa de comunicação entre eles. Entretanto, um MPSoC possui requisitos rígidos de comunicação que podem não ser atendidos por estruturas baseadas em barramentos.

Por outro lado, NoCs utilizam uma estrutura de comunicação distribuída, com múltiplos caminhos para a transferência de dados. O uso de NoCs permite o uso de MPSoCs para uma maior variedade de aplicações, devido à sua escalabilidade e paralelismo na comunicação entre os módulos conectados à rede. Dentre as vantagens oferecidas por esta estrutura de comunicação, pode-se destacar: *(i)* paralelismo de comunicação entre pares distintos de núcleos; *(ii)* compartilhamento de fios, pelo fato da

largura de banda ser escalável, comportando fluxos concorrentes; *(iii)* possibilita priorizar fluxos com requisitos de QoS (Quality of Service); *(iv)* possibilita a implementação de mecanismos de confiabilidade e gerência do consumo de energia; *(v)* reusabilidade.

Sob o ponto de vista do multi-processamento, um MPSoC é dito homogêneo quando os PEs que o compõem são todos iguais. Por exemplo, um sistema composto por processadores idênticos que permitem exclusivamente a execução de tarefas de software compiladas para tal arquitetura de processador. De outra forma, quando os PEs são de naturezas distintas, o MPSoC é dito heterogêneo. Isso pode ocorrer, por exemplo, quando diferentes tipos de processadores (GPPs, DSPs, etc) fazem parte de um mesmo MPSoC.

Enquanto MPSoCs homogêneos tendem a simplificar a aplicação de técnicas como migração de tarefas, MPSoCs heterogêneos tendem a suportar uma variedade maior de aplicações. Para garantir qualidade de serviço (QoS) e desempenho, um decodificador de TV digital (MPEG4), por exemplo, deve ser heterogêneo o suficiente para integrar vários processadores (e. g. RISC), núcleos de hardware dedicados (e. g. *upsampler*) e memórias (e. g. SDRAM). Cada um desses componentes possui diferentes funcionalidades, área de silício e necessidades de comunicação.

Atualmente existem produtos empregando MPSoCs, como os propostos pela Intel [VAN07] e pela Tilera [TIL09]. O primeiro deles é composto por 80 núcleos de processamento idênticos, enquanto o outro por até 100 núcleos também idênticos. No que diz respeito à comunicação, ambos MPSoCs citados são baseados em NoCs com topologia malha. Como exemplo de MPSoC comercial heterogêneo pode-se citar a arquitetura CELL, apresentada em [KIS06], desenvolvido pela IBM. Este MPSoC visa a execução de diversas aplicações, incluindo processamento científico [BUT07] e multimídia. Sua primeira utilização foi para o videogame PlayStation 3.

1.1 Motivação

Devido à alta complexidade do projeto de um MPSoC e os requisitos de mercado por um time-to-market curto, a sua construção baseia-se no reuso de PEs pré-projetados e pré-validados. Estes PEs podem ser compostos por processadores de diversos tipos, e em alguns casos, módulos de hardware dedicados.

Um dos problemas de projeto reside na integração de processadores a um MPSoC. Entre os problemas na integração podem-se citar: *(i)* desenvolvimento de interfaces entre o processador e a arquitetura de comunicação; *(ii)* desenvolvimento de um sistema operacional (SO) ou µkernel (software básico para execução do SO) para execução de software no processador; (iii) desenvolvimento de métodos para comunicação entre os processadores (que podem ser de outros tipos em arquiteturas heterogêneas).

Autores apontam que as restrições para projeto de MPSoCs conduzem a arquiteturas heterogêneas [JER05b][SHE09]. Por exemplo, Nikolov et al. em [NIK08]

apresentam um fluxo de projeto no nível de sistema onde estimativas de desempenho apontam que arquiteturas heterogêneas atingem um *speed-up* maior se comparado com arquiteturas homogêneas. Desta forma, pode-se inferir que existe a necessidade de um método para que a integração de diferentes tipos de processadores a um MPSoC se torne mais fácil e rápida.

Como ponto de partida para este trabalho, temos a plataforma HeMPS Station (HeMPS-S) [REI09]. Esta plataforma é composta por uma infra-estrutura de hardware e software capaz de gerenciar a execução de múltiplas tarefas simultaneamente. A plataforma é um ambiente dedicado para projeto de MPSoCs homogêneos, capaz de permitir a avaliação do desempenho de aplicações embarcadas distribuídas em determinada arquitetura executando em um FPGA, e também simulável no nível RTL e ISS (apenas para o processador). Como interface para comunicação entre a plataforma e um computador hospedeiro, a HeMPS-S oferece uma interface ethernet com uma pilha TCP/IP implementada em hardware para depuração e avaliação do desempenho.

A partir da plataforma de referência [REI09], o presente trabalho busca definir os requisitos de hardware e software necessários à integração de novos processadores a esta plataforma. A partir desta especificação, utiliza-se o processador MB-Lite como estudo de caso para integração deste à plataforma HeMPS. Além disto, este trabalho visa dar mais flexibilidade na escolha do PE.

1.2 Objetivos

Os objetivos deste trabalho compreendem objetivos estratégicos e específicos, definidos a seguir.

1.2.1 Objetivos Estratégicos

- Domínio da tecnologia de projeto de sistemas multi-processados em chip (MPSoCs);
- Domínio de diferentes arquiteturas de processadores;
- Domínio do SO (*µkernel*) que cada um dos processadores executa;
- Integração de diferentes processadores a um MPSoC.

1.2.2 Objetivos Específicos

- Domínio de pelo menos duas arquiteturas de processadores distintas, para o desenvolvimento de um MPSoC heterogêneo;
- Desenvolver técnicas que facilitem o porte do *µkernel* do MPSoC HeMPS, permitindo para outros processadores;

- Definir os requisitos de hardware e software necessários à integração de novos processadores ao MPSoC HeMPS, sendo esta a principal contribuição da presente Dissertação;
- Integrar o processador escolhido ao MPSoC HeMPS, de forma a obter-se um MPSoC heterogêneo.

1.3 Estrutura do Documento

Este documento é organizado da seguinte forma. O Capítulo 2 apresenta o estado da arte de arquiteturas MPSoCs, apresentando em seu final uma tabela comparativa entre as arquiteturas estudadas. O Capítulo 3 apresenta processadores *soft core* de código aberto disponíveis na literatura. Os processadores são comparados em termos de área ocupada em FPGA e freqüência de operação. O Capítulo 4 apresenta a plataforma HeMPS-S, bem como as modificações necessárias para sua prototipação em FPGA. O capítulo 5 apresenta a infra-estrutura de software (*µkernel*) utilizada no processador Plasma. O Capítulo 6 apresenta as modificações de hardware e software propostas neste trabalho, de forma a portar o *µkernel* para o processador MB-Lite. O Capítulo 7 apresenta exemplos de execução de aplicações no MPSoC heterogêneo. Por último, o Capítulo 8 apresenta conclusões e direção para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Neste Capítulo é apresentado o estado-da-arte em arquiteturas MPSoCs. Ao final deste é apresentada uma tabela com um comparativo entre estas. O objetivo de apresentar o estado-da-arte é mostrar ao leitor as principais arquiteturas MPSoC em desenvolvimento, os aspectos de gerenciamento de cada uma e identificar os PEs utilizados.

2.1 Tomahawk

Os autores em [LIM09] apresentam uma plataforma heterogênea para tratamento de sinais de antena para as próximas tecnologias de telefonia celular, tais como 3GPP LTE e WiMAX. Esta plataforma é gerenciada por: *(i)* dois processadores RISC Tensilica DC212GP que executam o sistema operacional; *(ii)* seis processadores SIMD DSPs ponto fixo para fazer processamento paralelo e vetorial, tais como FFTs e DCTs; *(iii)* dois processadores DSP de ponto flutuante, para processamento de fluxo de dados (*stream*); *(iv)* módulos de hardware para funções específicas como controle de paridade, decodificadores de entropia entre outros. A arquitetura é apresentada na Figura 1. Os blocos em cinza escuro representam IPs externos.

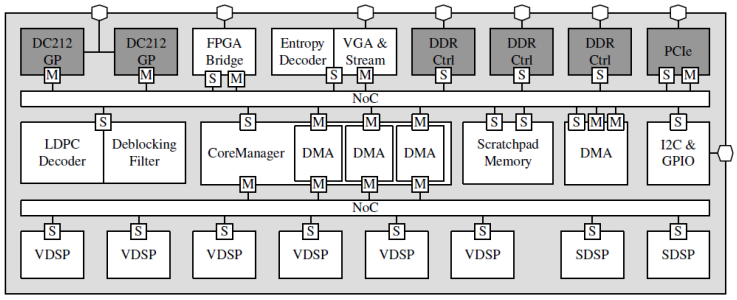


Figura 1 – Arquitetura do MPSoC proposto em [LIM09].

Os blocos são interconectados por duas NoCs com topologia *crossbar*, sendo uma mestre e outra escravo, com largura de *flit* igual a 32 bits e arbitragem por prioridade estática. A NoC suporta transferências em burst de até 63 palavras.

O modelo de programação da plataforma é similar ao utilizado no processador Cell [KIS06]. Segundo os autores, o escalonador implementado na plataforma, o *CoreManager*, por ser implementado em hardware, consegue um melhor desempenho e eficiência em energia quando comparado com o CellSS [BEL06].

O programador da plataforma deve anotar as funções em linguagem C que devem executar como tarefas em cada PE (ou seja, o mapeamento é realizado de forma manual). Além disso, devem-se definir os tipos de entradas e saídas de cada tarefa. Fica a cargo de um *script* fazer as chamadas para os diferentes compiladores para cada PE e mapear o código binário em memória.

O repositório com os códigos binários das tarefas, é gerenciado pelo *CoreManager*. Este também é responsável pelo envio da tarefa, somente quando as dependências de tarefas são resolvidas e quando existir um PE específico disponível. Portanto, se uma tarefa T2 depende do término da execução da tarefa T1, ela só é enviada quando T1 terminar.

O CoreManager então copia as tarefas da memória global (repositório) para a memória local do PE. Quando uma tarefa termina sua execução, a memória local do PE é copiada de volta à memória global.

2.2 MAGALI

Os autores apresentam em [JAL10] três implementações de MPSoCs, comparando-as em termos de área e dissipação de potência, usando benchmarks de aplicações de rede celular 4G. A Figura 2 apresenta estas arquiteturas.

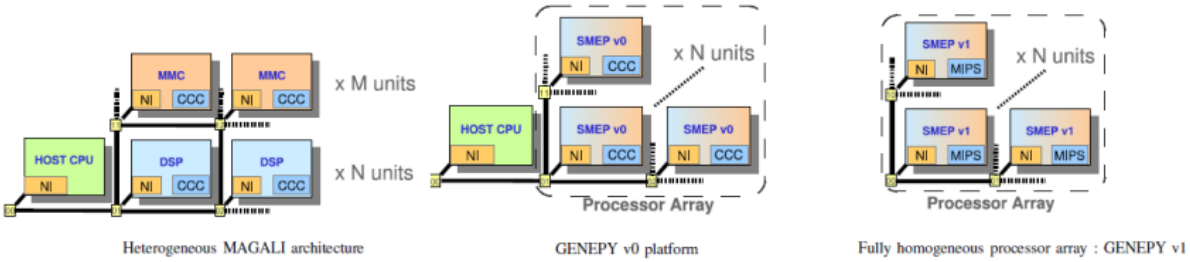


Figura 2 – MAGALI heterogênea, GENEPY v0 e a totalmente homogênea GENEPY v1. [JAL10]

A arquitetura heterogênea, denominada MAGALI, é composta por dois elementos de processamento diferentes: um DSP e um módulo MMC. O DSP é processador VLIW de 32 bits, otimizado para operações com números complexos, contendo ainda um módulo para operações MAC (multiplicação e acumulação). A unidade MMC é um controlador de memória microprogramável para manipulação intensiva de dados, incluindo sincronização, bufferização, duplicação e reordenamento.

A segunda arquitetura é denominada homoGENEous Processor array (GENEPY). Esta é composta pelos módulos Smart ModEm Processors (SMEP) interconectados por uma NoC. O SMEP é implementado em uma tecnologia de 65nm low-power CMOS, executando a 400MHz.

A arquitetura GENEPY v0 é composta de uma matriz de unidades SMEP v0 e um processador hospedeiro. Cada SMEP v0 é composto por um SME (Smart Memory Engine) e um cluster de processamento composto por dois DSPs. O SME gerencia quatro buffers mapeados em uma memória local de 32KB. O DSP é encarregado das leituras de uma FIFO de entrada. Os valores de processamento intermediário são salvos na memória local e os resultados, escritos na FIFO de saída. O fluxo de saída é lido na saída da FIFO pelo SME.

A arquitetura GENEPY v1 é composta apenas por módulos SMEP v1. Esta manteve os mesmos blocos de processamento e gerenciamento, mudando apenas o módulo de controle (CCC). O bloco de controle, nas arquiteturas anteriores era um módulo de hardware responsável por: *(i)* configurações de transferência de dados, *(ii)* armazenagem dos microprogramas para a comunicação de entrada e saída *(iii)* processamento dos dados, *(iv)* escalonamento da NI e do núcleo de processamento. Nesta arquitetura, o módulo responsável por estas é um processador MIPS 32 bits extensível, de forma que é possível a reconfiguração dinamicamente, escalonamento em tempo real e sincronizações. Com a adição deste módulo não é mais necessário o processador hospedeiro.

A implementação homogênea apresentou desempenho melhor comparado com a plataforma MAGALI heterogênea. A homogênea GENEPY v1 é aproximadamente 14% menor em termos de área de silício. Para uma aplicação de telefonia 4G, os comparativos mostraram um speed-up de 3% (v0 e v1), com uma economia de potência de 10% (v0) e 18% (v1). Os autores mencionam que arquiteturas heterogêneas são soluções para os padrões de wireless. Entretanto, argumentam que estas arquiteturas têm uma flexibilidade limitada e reiteram que abordagens homogêneas se justificam para os futuros terminais móveis de telefonia.

2.3 Zhang et al.

Este trabalho [ZHA07] apresenta um MPSoC heterogêneo implementado em um FPGA Altera. O sistema é composto por (Figura 3): *(i)* quatro processadores soft core Nios II; *(ii)* um processador ARM hard core, *(iii)* um subsistema de memória e *(iv)* um barramento do sistema. O processador ARM é o controlador central do sistema e os módulos NIOS II responsável pelo ambiente de processamento. Ambos dividem o acesso a um espaço de endereçamento, que inclui uma memória principal (SDRAM) e um subsistema de comunicação. O módulo contendo o processador ARM é composto por um processador ARM com barramento AMBA para comunicar-se diretamente com uma memória local. O sistema operacional e as aplicações podem ser executados desta memória local. Cada NIOS II contém uma cache de instruções e uma memória local. Cada um destes módulos tem acesso à memória compartilhada e ao barramento principal do sistema. Um dos módulos NIOS II contém uma UART JTAG para comunicação com o computador hospedeiro. O sistema é interligado através de uma arquitetura de barramento hierárquico. Esta arquitetura permite diferentes subsistemas acessarem sua camada do barramento e operar em paralelo se não existir conflito de recursos.

O sistema foi implementado no dispositivo EP2S180. Este dispositivo contém 143.520 ALUTs e 9 Mb de memória RAM embarcada. O sistema executa em uma freqüência de relógio de 60 MHz, utiliza 13% da área de ALUTs e 39% da área de

memória. Como resultado é apresentado uma aplicação utilizando multiplicação de matrizes.

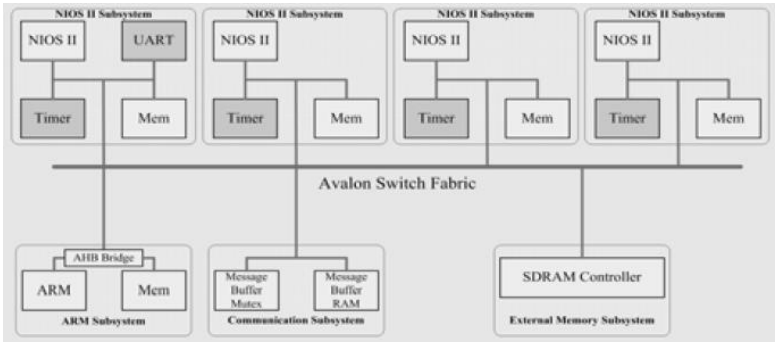


Figura 3 – Organização heterogênea proposta em [ZHA07].

2.4 ZeBu-XXL

Os autores [HAM09] apresentam um MPSoC que tem como alvo a prototipação de uma plataforma multi-FPGA. A arquitetura é composta por 672 PEs, conectados através de uma NoC com topologia malha de tamanho 7x8. Cada roteador é conectado a um cluster de PEs idênticos tratando-se, portanto de uma plataforma homogênea. A Figura 4 apresenta a arquitetura proposta.

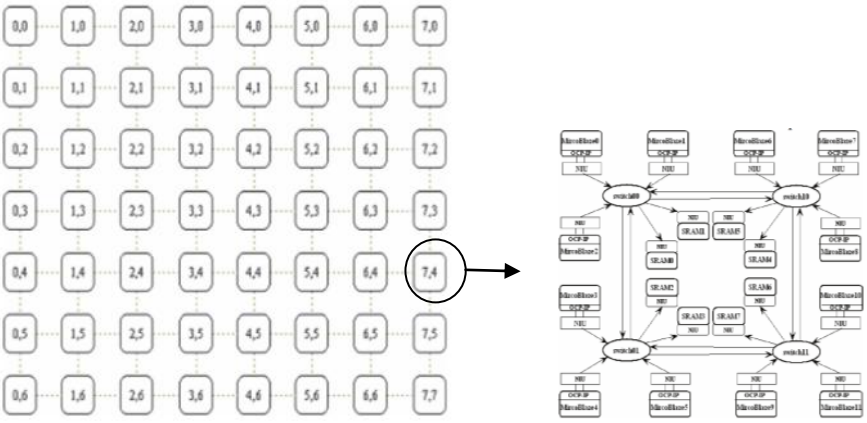


Figura 4 - Arquitetura MPSoC multi-FPGA. No detalhe, um FPGA com 12 PEs.

Cada cluster é de fato um MPSoC interligado por NoC, contendo: *(i)* 12 processadores Xilinx MicroBlaze; *(ii)* 8 memórias geradas a partir do software Xilinx Coregen; *(iii)* 4 roteadores da NoC e *(iv)* um módulo de comunicação entre os clusters (ICM). Estes dois últimos IPs foram gerados a partir da biblioteca VHDL Arteris Danube. Cada cluster é implementado em uma placa e contém os quatro roteadores interconectados por topologia malha. Cada roteador conecta três processadores e duas memórias SRAM.

A arquitetura com 672 processadores requer uma abordagem multi-FPGA. O ICM é responsável por conectar o sistema multi-FPGA. Cada FPGA contem 2 ICMs que conectam os FPGAs em uma topologia malha. A emulação de todo o sistema foi feita em

uma plataforma chamada ZeBu-XXL, contendo 56 FPGAs. Os autores ainda apresentam um fluxo para mapear o projeto nos diversos FPGAs. Como crítica, hoje não existe nenhuma aplicação que necessita um número tão elevado de processadores.

Como o elemento de processamento é uma Microblaze compatível com a norma OCP, os autores afirmam que o processador pode ser substituído por outro também compatível, deixando o restante do projeto idêntico.

2.5 Ninesilica

Os autores [GAR09] apresentam um projeto de MPSoC para aplicações de *soft radio* denominada Ninesilica. A arquitetura é formada por nove PEs, compostos por processadores RISC COFFEE interconectados através de uma NoC. O sistema multiprocessado é baseado no template Silicon Café. Este template provê um modelo configurável na linguagem VHDL para criar um MPSoC baseado em NoC interconectando vários nodos de processamento.

Os autores utilizaram uma rede 3x3 com uma topologia *star-mesh* com o mestre no centro, sendo o único conectado as entradas e saídas do circuito integrado. Cada escravo, por sua vez, é conectado diretamente ao mestre criando comunicações ponto a ponto entre cada escravo e o mestre (Figura 5).

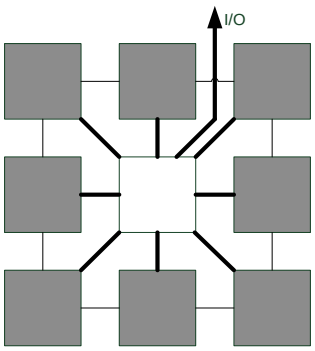


Figura 5- Ninesilica MPSoC: A NoC com uma topolotia star-mesh. Os PEs escravos conectam-se diretamente ao PE mestre.

Este trabalho apresentou resultados para implementações em ASIC e FPGA. A implementação em ASIC utilizou uma biblioteca de 65 nm, com uma ocupação de 630K Gates com uma freqüência máxima de relógio de 200 MHz. A implementação em FPGA teve como alvo um dispositivo Altera Stratix II, utilizando 76.780 ALUTs e 50.482 registradores, com uma freqüência máxima de 75 MHz.

2.6 HS-Scale

Os autores em [SAI07] apresentam um MPSoC homogêneo como um componente de um sistema heterogêneo (Figura 6). O MPSoC é baseado em uma arquitetura escalável com memória distribuída. A arquitetura proposta é feita de um arranjo de PEs interconectados por uma NoC. A partir da premissa que a HS-Scale é um componente de

um sistema, alguns dos PEs são responsáveis de estabelecer a comunicação com o restante do sistema (PEs de interface).

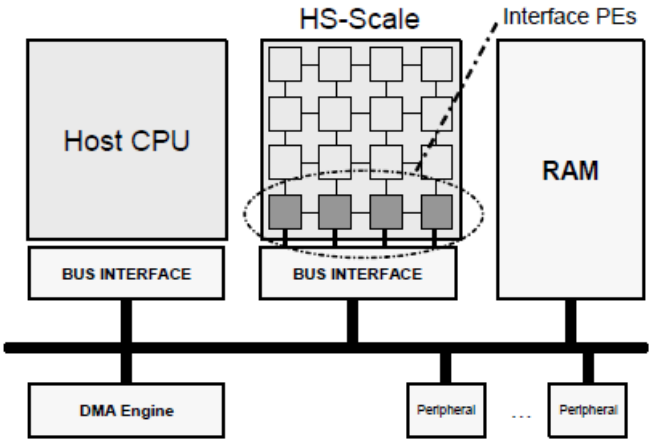


Figura 6 – Exemplo de sistema no qual o MPSoC HS-Scale é componente.

O MPSoC é formado por um arranjo de PEs comunicando-se através de uma rede, denominados de NPU (Network Processing Unit). Cada NPU é composto por: *(i)* um roteador, *(ii)* uma interface de rede (NI), *(iii)* um processador RISC e *(iii)* periféricos (UART, *timers* e controlador de interrupção). Cada PE executa um *µkernel* capaz de executar múltiplas tarefas que são escalonadas através de um time-slice. A comunicação entre os PEs é feita de forma assíncrona.

Como resultados foram apresentados resultados de duas aplicações: um filtro FIR e um decodificador MJPEG. Ainda foram apresentados resultados de ocupação e potência para uma tecnologia de 0,35 µm e ocupação para um dispositivo FPGA Xilinx.

2.7 xENoC

Os autores em [JOV08] apresentam um ambiente de hardware e software para projeto de MPSoCs baseados em NoC. É apresentada uma ferramenta de EDA (do inglês, Eletronic Design Automation) chamada NoCWizard, para gerar modelos de hardware de NoCs a partir de uma descrição XML. A plataforma xENoC é composta por um conjunto de elementos de comunicação (roteadores e NIs) agrupados em uma biblioteca de componentes e um *framework* de software.

O fluxo de projeto xENoC é composto por cinco tarefas para implementar um MPSoC baseado em NoC:

(i) Especificação do sistema: É feita a definição das especificações em nível de sistema (HW-SW) e da composição do sistema. Neste ponto são definidas as restrições do sistema, tais como desempenho do sistema como um todo, latência, largura de banda disponível, consumo de potência e parâmetros da NoC entre outros.

(ii) Exploração arquitetural: O principal objetivo nesta tarefa é realizar um refinamento, a fim de escolher uma arquitetura otimizada de acordo com os requisitos do sistema.

Isto implica na escolha otimizada dos PEs (VLIW, DSP, IPs específicos) e a infraestrutura de comunicação (barramento, NoC ou ponto a ponto), preenchendo assim os requisitos iniciais.

(iii) Projeto da arquitetura e co-projeto de HW-SW: a partir da arquitetura obtida, o próximo passo no fluxo de projeto é a implementação através de alguma HDL, para obter-se um código RTL sintetizável ou criando um protótipo em SystemC. Os autores dão ênfase à tarefa de integração entre HW-SW como chave para explorar as capacidades do hardware através de APIs e *drivers* de software.

(iv) Projeto de Software: Nesta tarefa, os projetistas de software devem otimizar as interfaces com os PEs, drivers de software e o sistema operacional. No caso específico de MPSoCs baseados em NoCs, é importante estudar e definir, através de um grafo acíclico dirigido (DAG, *directed acyclic graph*), a paralelização do código da aplicação seqüencial para tarefas distribuídas e concorrentes. O grafo é então implementado utilizando primitivas da API de software MPI (message passing interface).

(v) Integração e Verificação do Sistema: A última parte do fluxo de projeto é a integração e verificação do sistema todo. Deste modo, a arquitetura de hardware e os componentes de software devem ser verificados concomitantemente, em uma plataforma física ou utilizando um ambiente de co-simulação.

O ambiente para projeto automatizado propõe uma especificação em XML para especificar a NoC. Esta especificação descreve a comunicação das NIs, a localização e composição dos PEs, o tipo de roteador (e seu tamanho de FIFO e tipo de chaveamento) e também a toplogia. Este arquivo XML é a entrada para a ferramenta NoCWizard.

Como estudo de caso, os autores geraram uma NoC com topologia malha com chaveamento por circuito. Esta NoC conecta um número parametrizável de PEs. Cada PE é composto por um processador NIOSII com um barramento AMBA conectando a interface de rede (NI) e a memória do processador como apresentado na Figura 7.

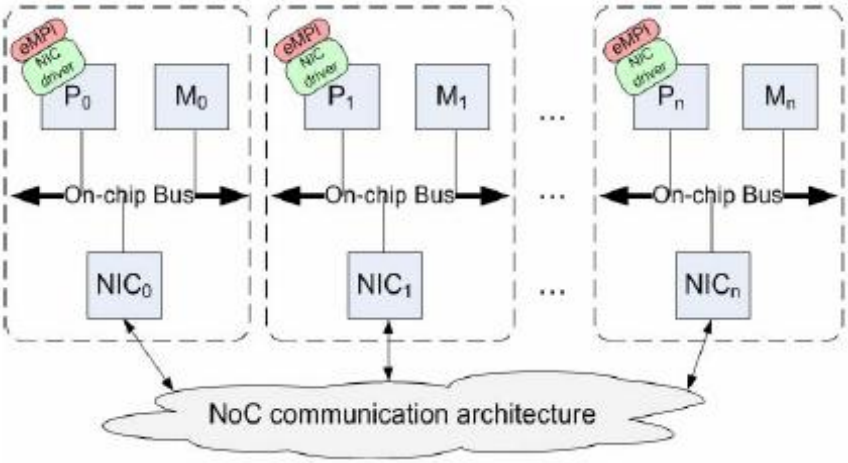


Figura 7 – Arquitetura proposta por [JOV08].

Foram apresentados resultados de área ocupada para diversos tamanhos de NoC e diferentes topologias. Por fim é apresentado um diagrama de seqüência que demonstra as primitivas de comunicação MPI utilizadas na aplicação portada para o MPSoC. Os autores defendem que para a aplicação apresentada, onde os dados gerados são independentes, o overhead de comunicação na NoC é insignificante. Desta forma, o speed-up é sempre próximo ao número de processadores incluídos no MPSoC (e.g. speed-up de 4 para uma rede 2x2).

2.8 HeMPS

O MPSoC HeMPS [WOS07] é composto por uma infra-estrutura de hardware e software capazes de gerenciar a execução de múltiplas tarefas simultaneamente.

A HeMPS é composta por um número parametrizável de PEs interconectados pela NoC HERMES [MOR04]. A NoC utiliza os seguintes parâmetros: *(i)* topologia malha com chaveamento de pacotes tipo wormhole; *(ii)* controle de fluxo baseado em créditos; *(iii)* algoritmo de roteamento XY; *(iv)* 16 bits de tamanho de flit; *(v)* buffer para armazenamento de 16 flits. O tamanho do flit é metade do tamanho da palavra do processador para reduzir a área do MPSoC.

Os PEs são chamados de Plasma-IP e compostos por: *(i)* processador PLASMA; *(ii)* interface de rede (NI); *(iii)* módulo de DMA; *(iv)* memória RAM dupla porta, a qual armazena o código objeto da tarefa e o *µkernel*. Os módulos Plasma-IP (ilustrado na Figura 8) são subdivididos em mestre e escravo. O processador mestre é responsável pelo gerenciamento de recursos do sistema. Quando a HeMPS inicia a execução, as tarefas são dinamicamente carregadas, pelo mestre, do repositório para os processadores escravos. Por este motivo o Plasma-IP mestre é o único processador com acesso ao repositório de tarefas. Quando uma tarefa finaliza sua execução, os recursos são liberados, de forma que outras tarefas possam executar neste processador. Os processadores escravos são responsáveis por executar as tarefas das aplicações.

O módulo Plasma-IP é o que define a interconexão entre os sub-módulos do PE (processador, memória, DMA e NI). Nele podem-se observar diversos registradores no mapa de memória, conectados à entrada de um multiplexador (Figura 8), e este por sua vez à entrada de dados do processador. Há também um módulo, *access repository*, presente apenas no Plasma-IP Mestre, o qual tem por função acessar o repositório externo de tarefas e transmitir via DMA os códigos objeto para os processadores escravos.

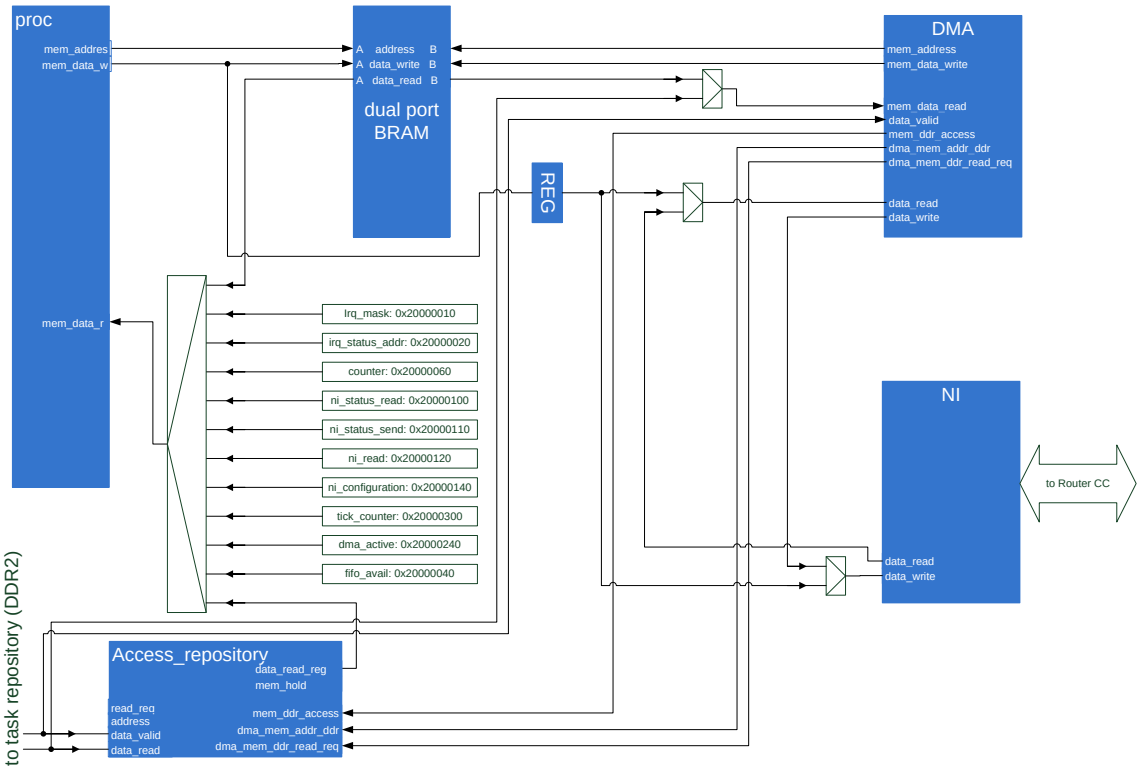


Figura 8 – Diagrama de blocos do Plasma-IP – o elemento de processamento do MPSoC HeMPS.

Para otimizar o desempenho nos PEs, a arquitetura Plasma-IP separa o processo de comunicação e computação. Os módulos NI e DMA são responsáveis por enviar e receber pacotes para a rede, enquanto que o processador Plasma executa o código da tarefa. A RAM local, por ser dupla porta, permite acesso simultâneo do processador e do DMA, evitando uso de hardware extra ou outra técnica, como roubo de ciclo.

Cada processador executa um *µkernel* com suporte a multitarefa e comunicação entre tarefas. O *µkernel* divide a memória em páginas, alocando para ele mesmo a primeira página e para as tarefas as páginas subseqüentes. Cada Plasma-IP contém uma tabela de tarefas, com o endereço de localização de cada uma delas. Um escalonador *round robin* provê o suporte a multitarefa.

A comunicação entre tarefas ocorre através de troca de mensagens. A troca de mensagens é feita através de um *pipe* global localizado no *µkernel* de cada processador. O *µkernel* provê primitivas de comunicação *(Send() e Receive())* para as tarefas. O *µkernel* é escrito em sua maioria em linguagem C e algumas funções especiais em linguagem de montagem (e. g. tratamento de interrupções e salvamento de contexto).

2.9 Conclusão

A Tabela 1 apresenta um comparativo entre as arquiteturas avaliadas. Os trabalhos apresentados mostram que NoC é o meio de interconexão mais utilizado nos MPSoCs revisados. Isto se deve a sua escalabilidade e paralelismo na comunicação. Outro aspecto relevante é que a variedade de processadores co-existindo no mesmo MPSoC

não é pequena. A maior variedade observada é no trabalho proposto por [LIM09], o qual contém três tipos de processadores. Módulos de hardware dedicados são mais raros, apenas presentes quando uma função específica é necessária por questões de desempenho, como módulos para decodificação de vídeo, por exemplo.

Do ponto de vista de gerenciamento, geralmente um dos processadores (Mestre) é encarregado pelo mapeamento das tarefas e gerenciamento de recursos. Em alguns casos o mestre apresenta uma arquitetura diferente dos escravos. O controle centralizado, utilizando um único mestre, pode se tornar um gargalo para redes maiores, visto que apenas um processador ficaria encarregado pelos dados de monitoração, mapeamento e migração de tarefas.

Tabela 1 – Comparativo entre as arquiteturas MPSoC apresentadas.

MPSoC	PE	Número de PEs	Arquitetura	Interconexão	Implementação
MAGALI [JAL10]	DSP ou MMC	1 PE mestre, 2 MMCs e 4 DSPs. Configurável.	Heterogêneo	NoC malha	ASIC
GENEPY v0 [JAL10]	Dois DSPs	1 PE mestre, 4 DSPs. Configurável.	Homogêneo com um PE mestre	NoC malha	ASIC
GENEPY v1 [JAL10]	Dois DSP e um MIPS	4 DSPs e 2 MIPS. Configurável.	Homogêneo	NoC malha	ASIC
Ninesilica [GAR09]	COFFEE	9	Homogêneo	NoC malha-estrela	
ZeBu-XXL [HAM09]	Microblaze	672, organizados em 8x7 clusters (cada cluster com 12 PEs)	Homogêneo	2 NoCs malha, uma para conectar os clusters, e outra para conectar os PEs dentro do cluster.	FPGA/ASIC
Zhang [ZHA07]	ARM ou NIOS II	4 NIOS and 1 ARM	Heterogêneo com um PE mestre	Barramento hierárquico	FPGA
Tomahawk [LIM09]	Tensilica, DSP ponto fixo, DSP ponto flutuante ou HW IP.	2 RISC, 4 DSP ponto fixo, 6 DSP ponto flutuante e 3 HW IP.	Heterogêneo	NoC crossbar	ASIC
HS-Scale [SAI07]	RISC	Configurável.	Homogêneo	NoC malha	FPGA
XeNoC [JOV08]	NIOSII	Configurável.	Homogêneo	NoC malha	FPGA
HeMPS [CAR09a]	MIPS – Plasma	Configurável.	Homogêneo com um PE mestre	NoC malha	FPGA

A plataforma HeMPS contém processadores RISC com baixa ocupação de área, tamanho do MPSoC configurável, interconexão por NoC malha, e uma arquitetura de software para execução multitarefa. Por ser uma arquitetura estado-da-arte, disponível com código aberto, a arquitetura HeMPS é utilizada como base para o presente trabalho.

3 PROCESSADORES SOFT-CORE

Este Capítulo tem por objetivo apresentar processadores *soft core* disponíveis na literatura. Como existe uma grande variedade de processadores de diversos tipos, este trabalho focou-se em processadores que apresentam algumas características específicas para um sistema embarcado. Entre estas, podem-se citar: *(i) tool chain* – disponibilidade de ferramentas para a compilação de código. (de preferência linguagem C): compilador e montador que derivem do gcc seriam ideais; *(ii)* baixo consumo de área de silício e bom desempenho; *(iii)* código aberto: deve ser permitido ao projetista fazer modificações ou até mesmo a inserção de módulos no processador, de forma a otimizar ou adicionar funcionalidades. Apesar disto, um dos objetivos deste trabalho é não exigir do projetista a modificação interna do processador.

Kranenburg e Van Leuken em [KRA10] avaliaram vários processadores, considerando aspectos como: qualidade do projeto, desempenho, documentação disponível e as funcionalidades específicas de cada. Neste trabalho, os autores avaliaram cinco processadores: OpenFire, OpenRISC, LEON3, AeMB, e MB-Lite. O processador MB-Lite foi projetado e proposto pelos autores, como uma implementação da arquitetura Microblaze. Adicionalmente, esta Dissertação avaliou os processadores Plasma [OPE10] (modificado em [WOS07]), COFFEE [KYL03] e o Plasma com uma unidade de ponto flutuante [ROD09]. A avaliação destes oito processadores é detalhada a seguir.

3.1 AeMB

Este processador de 32 bits está sob desenvolvimento pela Aeste Works [AES10]. O processador utiliza uma interface Wishbone para a memória de dados e instruções. O AeMB é composto por um pipeline de cinco estágios e os barramentos de dados e instruções são separados (organização Harvard). A organização é baseada nos cinco estágios clássicos de um pipeline RISC. O processador provê suporte a uma única interrupção externa. O banco de registradores é composto por 32 registradores de 32 bits.

Este processador é uma implementação da arquitetura Microblaze, com exceção das instruções WIC, WDC (escrita na cache), IDIV, IDIVU (divisão). Além disso, o AeMB foi desenvolvido para ser um processador de duas *threads*, podendo fazer uma troca de contexto em um ciclo.

Por ser semi-compatível com a Microblaze, pode-se utilizar a *tool chain* Microblaze para compilar o software para este processador. De acordo com Kranenburg, alguns programas não executam como esperado. Fazendo-se algumas modificações no código, a fim de torná-lo compatível com ANSI C resolvem-se alguns problemas, mas não todos. Por outro lado, a síntese em hardware não apresentou maiores problemas para [KRA10] utilizando-se a ferramenta ISE 10.1.

3.2 LEON3

A família de processadores LEON é uma implementação da arquitetura Sparc V8, a qual não é proprietária e totalmente aberta. O penúltimo processador desta família é o processador de 32 bits LEON3. Sua implementação é baseada na arquitetura Harvard e utiliza o barramento *Amba Advanced High-performance Bus* (AHB).

Uma característica da arquitetura Sparc é o uso de uma janela de registradores para aumentar o desempenho principalmente na troca de contextos de programação. Esta característica é implementada em hardware, ficando transparente para o desenvolvedor de software. Por outro lado, a área ocupada aumenta devido aos requisitos dos registradores. Uma implementação padrão tem oito registradores globais e oito janelas de registradores, cada uma contendo 24 registradores. O número de registradores visíveis é, portanto, 32 (8 globais mais 24 dedicados por janela), enquanto que o total de registradores é 200. Entretanto, a implemetação feita em [KRA10] não obteve benefício da janela de registradores. Um *benchmark* mostra que tanto a MB-Lite quanto a implementação comercial Microblaze da Xilinx tem um tempo de execução menor que o LEON3. Uma explicação é que a *tool chain* aplicada não se beneficia da janela de registradores.

O projeto do processador LEON3 é modular, de tal forma que pode-se desabilitar certas partes do processador para poupar recursos. Uma ferramenta de configuração pode ser utilizada para gerar um SoC que inclua vários periféricos. Até oito processadores podem ser ligados no barramento AHB. Em termos de prototipação, o projeto apresenta várias bibliotecas de hardware para implementações em FPGA e ASIC. Os autores ainda defendem que é difícil, senão impossível de se obter um processador com poucos recursos, pois as unidades de gerência de memória, caches e os controladores do barramento AHB não podem ser desabilitados.

Apesar destas limitações, este processador é utilizado como arquitetura tolerante a falhas, como apresentado em [HAF09] e para aplicações espaciais, como apresentado em [ESA11].

3.3 OpenRisc 1200 [OPE11a]

O processador OpenRisc 1200 faz parte da família de processadores embarcados OpenRisc 1000. Este contém um pipeline de cinco estágios e implementa a arquitetura *OpenRisc Basic Instruction Set* (ORBIS). Esta arquitetura Harvard é composta por instruções de 32 bits e pode operar com dados de 32 e 64 bits. Muitas extensões para esta arquitetura foram implementadas para melhorar, por exemplo: *(i)* o processamento de vetores (*OpenRisc Vector/DSP eXtension* – ORVDX) e *(ii)* instruções de ponto flutuante (*OpenRisc Floating Point* eXtension – ORFPX). O processador avaliado não leva em conta estas extensões. Tanto a memória de dados como a de instruções utilizam o barramento Wishbone.

3.4 OpenFire [OPE11b]

A arquitetura OpenFire 0.3b é uma implementação da arquitetura MicroBlaze 7.10, incluindo instruções e palavras de 32 bits, e todas as operações aritméticas. O processador foi projetado para pesquisa em processadores configuráveis. Características como interrupções, exceções e registradores especiais não foram implementados, pois o objetivo era manter o processador pequeno e simples.

O pipeline é composto por três estágios, diferentemente da especificação da arquitetura Microblaze. Operações de *load* e *store* levam mais de um ciclo para executar, de forma que o processador não se torna compatível no nível de ciclo com a Microblaze.

Este projeto ainda permite interligar vários processadores utilizando o barramento Microblaze FSL (Fast Simplex Link). A documentação do projeto é pobre e não se pode compreender o projeto internamente ao processador. A documentação menciona que a instrução *break* não funciona corretamente, e este problema não é explicado em maiores detalhes.

3.5 COFFEE

O processador COFFEE (Core For FrEE) [KYL03] é um processador de 32 bits baseado na arquitetura MIPS. COFFEE é composto por um pipeline de seis estágios e uma organização de memória Harvard.

O conjunto de instruções foi projetado para evitar *bolhas* no pipeline (e.g. a instrução de divisão foi removida). Fazem parte do processador um controlador de interrupções, uma interface para um coprocessador e o *switch mode*. O switch mode é utilizado para trocar o hardware de decodificação de instruções. Desta forma, pode-se integrar um multiplicador e acumulador (MAC) em hardware. Este MAC será executado quando a instrução adicionada pelo switch mode for executada, por exemplo.

O COFFEE é composto por dois bancos registradores, cada banco contendo de 32 registradores. Ambos os conjuntos são acessíveis no modo privilegiado enquanto que apenas um é acessível no modo usuário.

Os resultados avaliados nesta Dissertação mostram COFFEE é o mais caro em termos de área, comparado com os processadores aqui descritos. A explicação para este fato é que o banco de registradores não está implementado utilizando-se LUTRAMs.

3.6 Plasma

O Plasma é um processador RISC de 32 bits baseado no conjunto de instruções original do processador MIPS. Diferentemente da arquitetura original do MIPS, o Plasma tem três estágios no pipeline e uma organização de memória Von Neumann. O Plasma suporta todas as instruções do ISA (*Instruction Set Architecture*) MIPS-I, exceto as instruções de *load* e *store* desalinhadas, pois estas são patenteadas. A tool chain é

composta por um montador e um compilador de linguagem C; o *mips-elf- assembler* e o *mips-elf-gcc-compiler*, respectivamente.

O Plasma apresenta também uma unidade de multiplicação e divisão em hardware por somas e subtrações sucessivas, respectivamente. Por ser uma arquitetura Von Neumann, existe apenas uma interface de acesso à memória RAM. Por este motivo existe a unidade de controle de memória (Mem_ctrl). A Figura 6 apresenta o diagrama de blocos do processador Plasma.

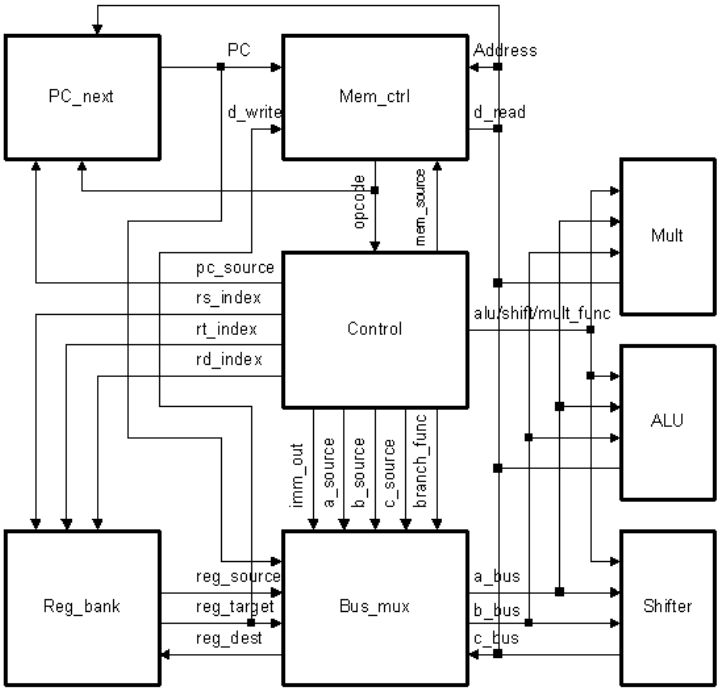


Figura 9 – Diagrama de blocos do processador Plasma.

O Plasma é o núcleo do PE da plataforma HeMPS [WOS07]. Para a utilização na HeMPS, foram feitas modificações na arquitetura V2.0 do Plasma [OPE10], destas podem-se destacar: *(i)* inserção de um mecanismo de paginação interno ao processador; *(ii)* exclusão do módulo da UART, *(iii)* adição de registradores mapeados em memória e *(iv)* inserção da instrução `syscall`. Note que os resultados de área e freqüência mencionados aqui são baseados na versão modificada. Esta versão inclui uma unidade de multiplicação, diferentemente dos outros processadores apresentados, que contém apenas o núcleo do processador.

3.7 PLASMA-FPU

Rodolfo et al. [ROD09] apresentaram implementações de unidades de ponto flutuante com precisão simples executando como co-processador do processador Plasma V3.5 [OPE10]. As unidades de ponto flutuante são organizadas a partir de implementações *(i)* FPU100 do *opencores* e *(ii)* FPU da ferramenta Coregen da Xilinx. Estas unidades de ponto flutuante são co-processadores com três organizações: *(i)* Plasma-HFP100: baseada na FPU100 e *(ii)* Plasma-HFPMin e *(iii)* Plasma-HFPMax:

baseadas na FPU do Coregen, diferindo apenas na parametrização de latência dos módulos de hardware.

Entre as três arquiteturas avaliadas, a arquitetura Plasma-HFPMax apresentou melhor relação custo benefício entre área e desempenho. Devido à diferença entre as freqüências de operação do processador e do co-processador, os autores propuseram uma organização GALS. Na realidade foram propostas duas organizações, com relações de relógio de 4 e 8 vezes, sendo o relógio da FPU mais rápido que o do processador.

Em aplicações com uso intensivo de operações de ponto flutuante, a arquitetura Plasma-HFP-8X apresentou uma melhora de velocidade de execução até 22 vezes em relação à emulação por software.

3.8 MB-Lite [KRA10]

O MB-Lite é uma implementação da arquitetura MicroBlaze de 32 bits, baseada na arquitetura MIPS. Este processador contém os mesmos cinco estágios do pipeline do MIPS. Várias modificações na organização MIPS forma efetuadas para obter uma implementação compatível com a MicroBlaze.

O processador MB-Lite implementa um controle distribuído para eliminar a necessidade de um controlador de pipeline complexo e centralizado. Todas dependências como *stall*, *hazards* e *forwards* são resolvidas localmente. Por isso, é relativamente fácil de compreender o projeto, uma vez que as dependências foram feitas de forma clara e simples para entender. O projeto foi descrito seguindo métodos de codificação rigorosos. Caminhos combinacionais e seqüenciais são explicitamente separados para facilitar o entendimento das dependências de temporização. Esta abordagem resultou em um projeto que é fácil de entender, depurar e manter.

3.9 Conclusão

A maioria dos processadores analisados utiliza métodos *ad-hoc* para o desenvolvimento do hardware. Este tipo de metodologia é difícil de manter, entender e depurar. Por este motivo, muitos dos processadores acabaram sendo descartados para utilização como novo PE a ser incluído na HeMPS.

Outra característica apresentada é de que não se consegue extrair facilmente apenas a unidade de processamento. Ou seja, o processador está geralmente ligado a periféricos, controladores de barramento e módulos de hardware específicos, entre outros. Para alguns tipos de aplicações este tipo de abordagem é importante, por exemplo, quando se necessita uma interface padrão de barramento. Entretanto o objetivo do presente trabalho é um processador simples, sem módulos de hardware para manter a ocupação de área pequena.

Devido ao código bem escrito e a documentação disponível, o processador MB-Lite foi escolhido como a arquitetura para ser incluída na implementação heterogênea da HeMPS. A Tabela 2 resume os processadores revisados, comparando-os. Pode-se notar que a arquitetura dos processadores apesar de variar bastante, é muito ligada à arquitetura MIPS, visto que a Microblaze é baseada nesta. Desta forma, dos oito processadores estudados, sete apresentam uma arquitetura similar à do processador MIPS. A exceção é o processador LEON3.

Tabela 2 – Processadores avaliados. Adaptado de [KRA10].

Processador	Arquitetura	Arquitetura de Memória	HDL	Tool-chain	# de estágios do pipeline
AeMB	Microblaze	Harvard	Verilog	mb-gcc	5
LEON3	Sparc V8	Harvard	VHDL	sparc-elf-gcc	7
OpenFire	Microblaze	Harvard	Verilog	mb-gcc	3
OpenRisc	ORBIS	Harvard	Verilog	gcc modificado	5
MB-lite	Microblaze	Harvard	VHDL	mb-gcc	5
Plasma	MIPS	Von Neumann	VHDL	gcc-mips-elf	3
COFFEE	RISC	Harvard	VHDL	gcc modificado	6
Plasma-FPU	MIPS	Von Neumann	VHDL	gcc-mips-elf	3

A Figura 10 apresenta os resultados da síntese dos processadores. Dentre os processadores que consomem pouca área pode-se citar os processadores aeMB, Plasma, OpenFire e MB-Lite. O processador OpenFire foi descartado devido ao problemas de documentação e da instrução *break* relatados anteriormente. O processador aeMB foi descartado devido aos *glitches* apresentados no barramento de endereços, e a não correta execução de alguns programas.

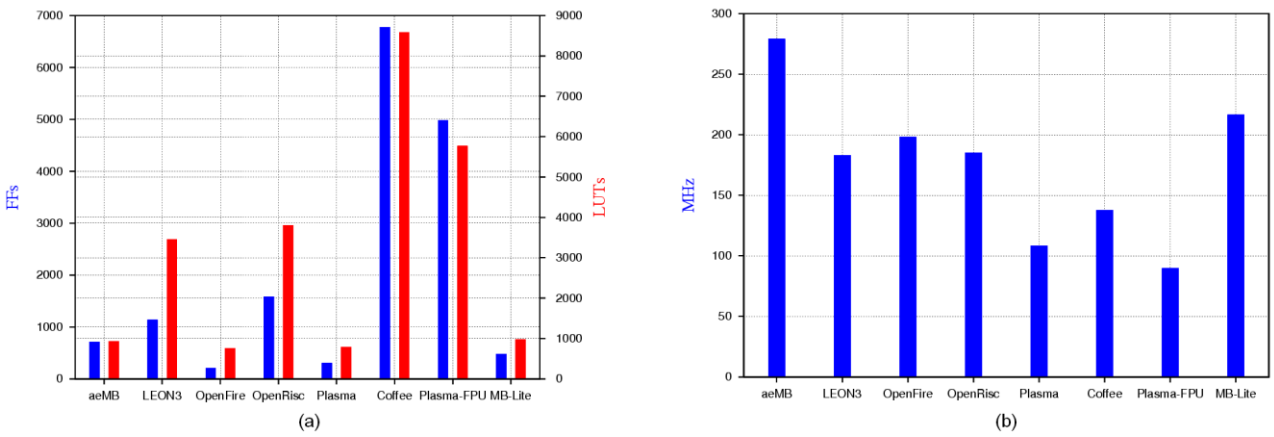


Figura 10 – Resultados de utilização de área (a) e de freqüência (b) dos processadores avaliados, tendo como alvo um o dispositivo xc5vlx110ff1760-3. Adaptado de [KRA10].

4 A PLATAFORMA HEMPS-S E SUA PROTOTIPAÇÃO

A Plataforma HeMPS Station [REI09], utilizada para o projeto de sistemas MPSoC, é utilizada como base para o desenvolvimento deste projeto. A Seção 4.1 descreve a arquitetura de hardware da HeMPS-S. A prototipação da plataforma constitui a *primeira contribuição* desta Dissertação, descrita na Seção 4.2. Por último, a Seção 4.3 descreve a ferramenta HeMPS-S Generator, utilizada para geração e controle da plataforma.

4.1 Descrição da Arquitetura de Hardware

A plataforma HeMPS Station (HeMPS-S) [REI09] é um ambiente que permite a avaliação de aplicações embarcadas distribuídas em MPSoCs. Esta avaliação pode ser feita em uma plataforma FPGA, bem como por simulação no nível RTL. Através das ferramentas executando em um computador hospedeiro, e uma interface de comunicação entre este e o MPSoC é possível acessar a HeMPS-S. Desta forma, pode-se avaliar o sistema em tempo de execução e monitorar o sistema para obtenção de dados de desempenho.

A arquitetura do ambiente de prototipação é apresentada na Figura 11. HeMPS-S é composta pelo MPSoC HeMPS, o módulo ConMe (um controlador de DDR2), o módulo ComEt (Interface de comunicação MAC Ethernet), todos, interconectados pelo módulo *Main Control*.

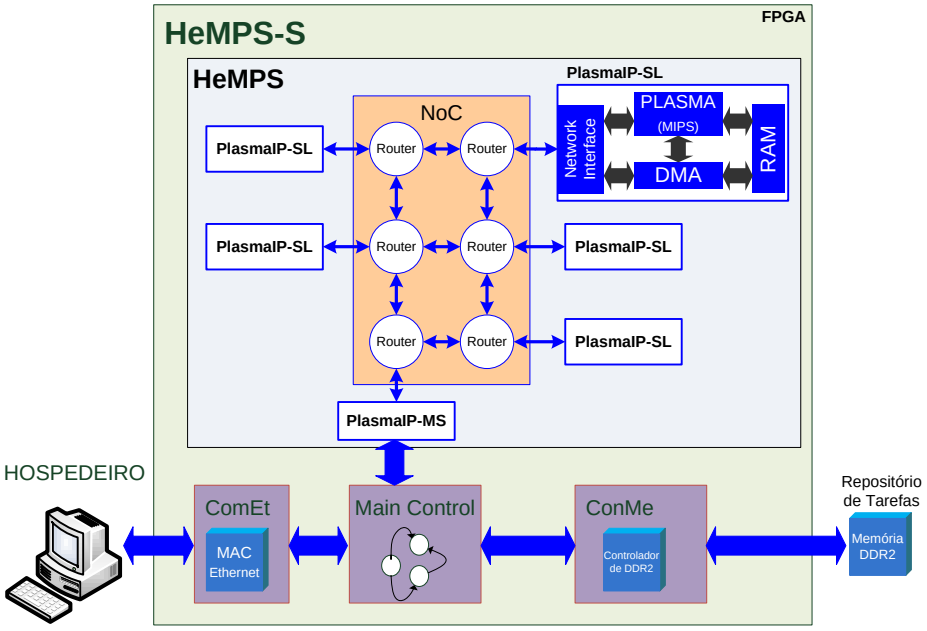


Figura 11 – Visão geral da arquitera HeMPS-S.

A HeMPS-S é controlada pelo usuário no computador hospedeiro. O hospedeiro contém aplicações descritas em linguagem C, modeladas utilizando grafo de tarefas. O

usuário pode escolher em qual processador cada tarefa irá executar, ou pode escolher deixar para que o sistema (processador mestre) dinamicamente mapeie estas tarefas.

O computador hospedeiro compila cada tarefa, e o código binário resultante é armazenado em um arquivo, denominado repositório de tarefas. O repositório é composto pelo cabeçalho e os códigos binários das tarefas. O cabeçalho identifica onde cada tarefa será mapeada (ou se esta será mapeada dinamicamente), o tamanho do código da tarefa e em qual posição do repositório o código da tarefa inicia. A Figura 12 apresenta a estrutura do repositório de tarefas.

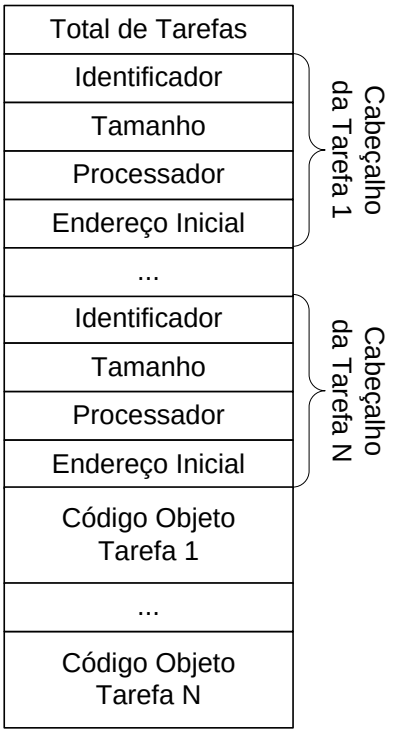


Figura 12 – Estrutura do repositório de tarefas da plataforma HeMPS-S.

A Figura 13 ilustra o processo de armazenamento do repositório na plataforma HeMPS-S. Depois da compilação, o computador hospedeiro envia o repositório de tarefas para o MPSoC. O repositório é enviado através de pacotes UDP pela interface MAC Ethernet (1 na Figura). O módulo ComEt é responsável por processar estes pacotes, através de uma pilha TCP/IP implementada em hardware (2). Esta pilha implementa os protocolos ARP, IP e UDP. O Main Control recebe então os dados do repositório (3) e os armazena na memória DDR2 através do módulo ConMe (4). Depois que todo o repositório de tarefas foi armazenado na memória, o hospedeiro inicia a execução das aplicações no MPSoC HeMPS (5). O início da execução é feito pelo usuário (via ferramentas no computador hospedeiro) através de um comando para o módulo *Main Control*, que controla o MPSoC.

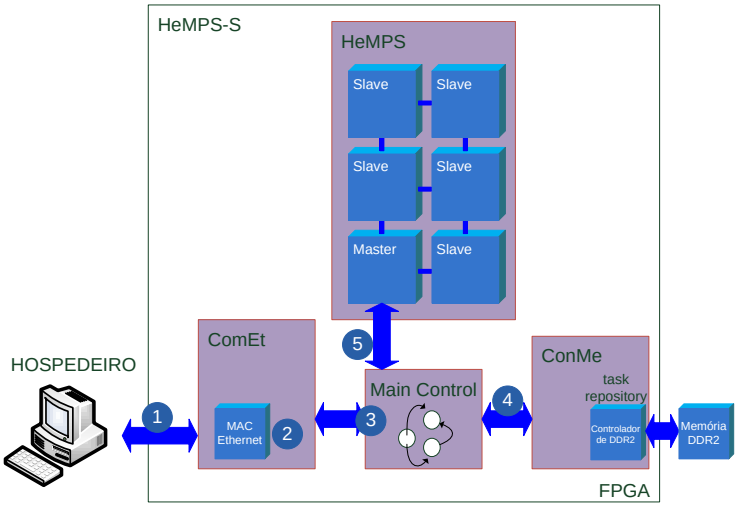


Figura 13 – Exemplo do processo de armazenamento do repositório na plataforma HeMPS-S.

A Figura 14 ilustra o processo de execução de aplicações na plataforma HeMPS-S. Com o repositório armazenado, o processador mestre do MPSoC inicia a leitura deste através do módulos *Main Control* (1) e ConMe (2). O mestre pode então enviar as tarefas para os escravos (3) de acordo com o mapeamento indicado no cabeçalho do repositório. Quando a tarefa é recebida por um escravo, esta é escalonada para ser executada. As tarefas podem comunicar-se entre elas, no mesmo ou em diferentes processadores, e ainda enviar dados de depuração para o mestre. O mestre envia estes dados de depuração para o hospedeiro através dos módulos Main Control (4) e ComEt (5). No hospedeiro, uma aplicação de controle da plataforma interage com o usuário através de interface gráfica. Esta aplicação, denominada *HeMPS-S generator*, mostra ao usuário os dados de depuração (6), habilitando a avaliação de desempenho. A Figura 23, ao final deste Capítulo, apresenta a *HeMPS-S generator*.

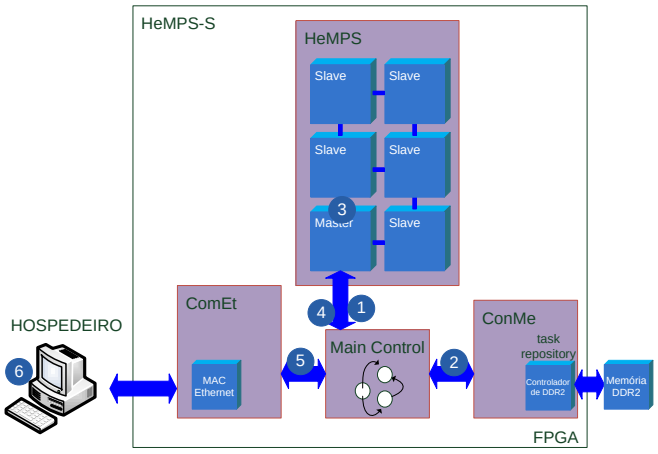


Figura 14 - Exemplo do processo de execução e depuração das aplicações na plataforma HeMPS-S.

4.1.1 ConMe

O módulo ConMe tem por objetivo ser um intermediário entre a HeMPS e um controlador de memória DDR2. O ConMe é dividido em dois blocos: uma interface e um controlador DDR2. O Controlador DDR2 foi gerado pela ferramenta Xilinx Coregen e é responsável por interagir fisicamente com a memória. A interface tem por objetivo simplificar o protocolo de comunicação com o controlador e auxiliar no endereçamento dos dados para o módulo Main Control.

A DDR2 deve armazenar o repositório de tarefas. A escrita é feita com os dados do repositório contidos nos pacotes UDP enviados pelo hospedeiro. A leitura é feita pelo Plasma-IP mestre. Tanto a leitura como a escrita são controladas pelo módulo *Main Control*.

4.1.2 ComEt

Este módulo é responsável pela comunicação entre a HeMPS e o computador hospedeiro. A pilha TCP/IP é implementada em hardware, sem a necessidade da utilização de um processador dedicado ou de uma pilha em software para processar os pacotes com a rede. Esta estratégia é adotada para melhorar o desempenho e reduzir área ocupada.

O ComEt é composto pelos módulos de transmissão e recepção de pacotes Ethernet e um núcleo MAC Ethernet. Estes módulos se comunicam com o MAC e com o módulo Main Control. Para a comunicação com o MAC, é implementada então, uma pilha de TCP/IP com suporte aos protocolos UDP, IP e ARP.

O módulo de recepção remove o cabeçalho dos pacotes IP e transmite a carga útil para o Main Control. O módulo de transmissão é responsável por inserir o cabeçalho nas mensagens de depuração do Plasma-IP mestre para gerar pacotes IP. Estas mensagens, em sua maioria, contêm dados de depuração gerados pelas tarefas que estão executando no MPSoC. Vale salientar que estas mensagens devem ser geradas pelo projetista através de software por ele desenvolvido.

4.1.3 Main Control

Este módulo é responsável por controlar a interação entre os módulos da HeMPS-S. Entre suas funções, podem-se citar: *(i)* processar os comandos recebidos pelo ComEt; *(ii)* escrever e ler dados no ConMe; *(iii)* transmitir o repositório de tarefas para a HeMPS e *(iv)* transferir as mensagens de depuração da HeMPS para o ComEt.

O Main Control recebe apenas a carga útil dos pacotes enviados pelo hospedeiro via ComEt. A carga útil é composta por um comando e os dados, responsáveis por gerenciar a execução da HeMPS-S. Os comandos são detalhados na Tabela 3.

Tabela 3 – Comandos processados pelo módulo Main Control.

Comando	Função	Descrição
CCEECCAA	Connect	Conecta o computador hospedeiro com a HeMPS-S
0000AADD	Write	Escreve o código binário no repositório de tarefas
AAAAAAAA	Start	Inicializa a execução na HeMPS-S
DDCCEECC	Disconnect	Desconecta o computador hospedeiro da HeMPS-S

4.2 Prototipação da Plataforma HeMPS-S

Esta Seção descreve a primeira contribuição deste trabalho, o processo de prototipação em FPGA da plataforma HeMPS-S, indicando as modificações no projeto necessárias para atender às restrições de temporização. O dispositivo utilizado é um FPGA Xilinx Virtex 5 5vlx330tff1738-2, escolhido devido à grande quantidade de blocos de memória embarcada (324 blocos BRAM) e à disponibilidade de núcleos MAC Ethernet no seu interior, o qual é necessário ao módulo ComEt.

As principais ferramentas de CAD empregadas foram Xilinx o ambiente PlanAhead 11.1 e Modelsim da Mentor. O PlanAhead habilita o projeto de planta baixa, inserção de restrições de projeto, depuração e análise dos resultados de área e temporização. Diferentemente do ISE, no PlanAhead pode-se sintetizar múltiplas estratégias de implementação facilmente no mesmo projeto. Deste modo, se torna possível avaliar um maior número de alternativas de projeto, acelerando-se o processo de prototipação. O Modelsim, apesar de não utilizado para prototipação, foi utilizado para simular a plataforma.

4.2.1 Reestruturação da Arquitetura

Para implementar um número maior de PEs, foi necessário modificar a plataforma HeMPS-S original [REI09]. A arquitetura original utilizava a NoC Hermes como um módulo IP. Esta abordagem não é prática, pois todos os roteadores estarão próximos entre si, e não necessariamente próximos aos seus devidos processadores. Os roteadores devem estar fisicamente próximos aos processadores com os quais se comunicam, de forma a reduzir o atraso nas interconexões locais. Por este motivo, um novo módulo foi criado, chamado *Plasma_Router.* O Plasma_Router contem um Plasma-IP e o roteador central da NoC Hermes (*RouterCC*). Esta modificação evita fios longos, o que comprometeria o desempenho do MPSoC.

O RouterCC é um dos nove diferentes tipos de roteadores existentes em qualquer instância de uma NoC Hermes com dimensões maiores que 3x3. Este roteador contém cinco portas, uma porta local para comunicação com o processador e outras quatro para comunicação com os vizinhos (NoC malha).

De acordo com a posição de algum Plasma_Router no MPSoC, algumas portas não serão utilizadas (e.g. porta norte nos roteadores de topo). As portas não utilizadas têm seus sinais de entrada aterrados e os de saída abertos (não conectados a nenhum módulo). A ferramenta de síntese remove a lógica não utilizada, reduzindo a área. A Figura 15 apresenta a organização interna do módulo Plasma_Router.

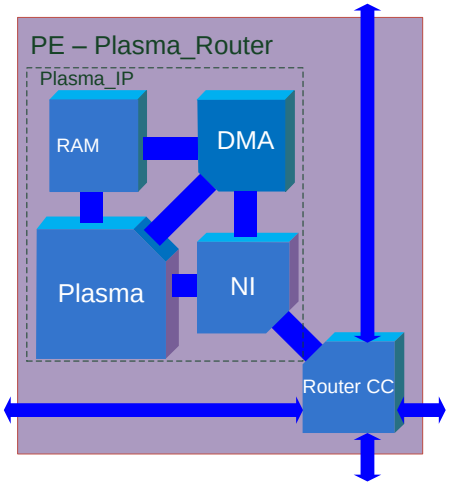
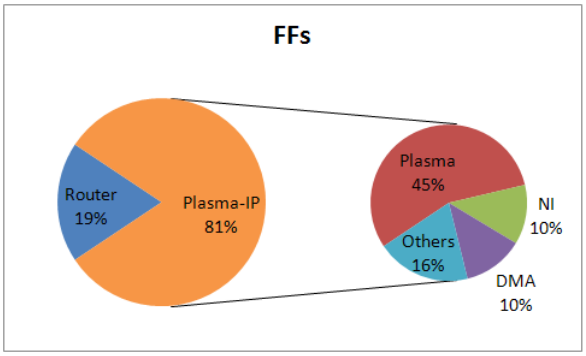
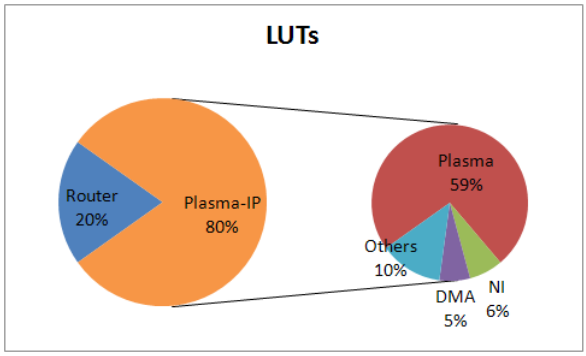


Figura 15 – A organização do módulo Plasma_Router.

A Figura 16 apresenta a ocupação de área em termos de flip-flops e LUTs de cada módulo Plasma_Router. Note que o roteador avaliado tem cinco portas operacionais, sendo portanto o pior caso em termos de área.



(a) Ocupação de Flip-Flops para o Roteador (224) e Plasma-IP (974).



(b) Ocupação de LUTs para o Roteador (621) e Plasma-IP (2541).

Figura 16 – Ocupação de área em termos de flip-flops e LUTs, tendo como alvo o dispositivo 5vlx330tff1738-2.

4.2.2 Restrições de Temporização

A plataforma HeMPS-S prototipada têm três fontes externas de relógio: uma de 50 MHz e duas de 25 MHz. O relógio de 50 MHz é utilizado pelo MPSoC, enquanto que o controlador DDR2 requer quatro sinais de relógio, derivados do relógio de 50 MHz. A Figura 17 ilustra como os sinais de relógio da HeMPS e do controlador DDR2 são obtidos

a partir do relógio externo. Os outros dois relógios externos (de 25 MHz) são gerados pelo PHY externo, o qual comunica-se com o MAC Ethernet.

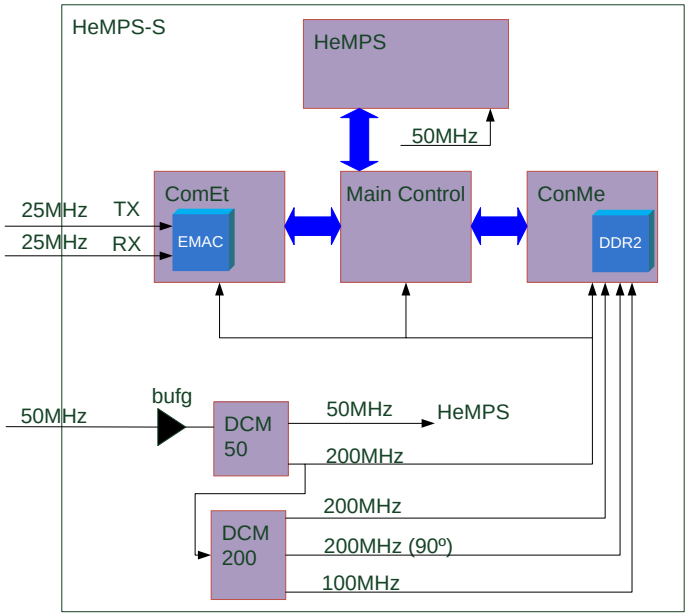


Figura 17 – Organização de relógios da HeMPS-S. Os DCMs (Digital Clock Managers) são componentes do FPGA para gerar sinais de relógio com baixo escorregamento, além de permitir dividir ou multiplicar as freqüências de operação.

A Figura 18 apresenta parte do arquivo de restrições do usuário (*User Constraints File* – UCF), detalhando as restrições aplicadas para cada sinal de relógio. Por exemplo, a linha 1 indica que o sinal `hemps_clk_50` pertence ao `TNM_NET` denominado `SYS_clk50`. O código da linha 2 atribui à `TNM_NET SYS_clk50` uma restrição temporal de um sinal de relógio com período de 20 ns e ciclo de serviço de 50%.

```
1. NET "hemps_clk_50" TNM_NET = "SYS_clk50";
2. TIMESPEC TS_SYS_clk_50 = PERIOD "SYS_clk50" 20 ns HIGH 50 %;
3. NET "clk200_bufg" TNM_NET = "SYS_clk200";
4. TIMESPEC TS_SYS_clk200 = PERIOD "SYS_clk200" 5 ns HIGH 50 %;
5. NET "clk0_bufg" TNM_NET = "SYS_clk0";
6. TIMESPEC TS_SYS_clk0 = PERIOD "SYS_clk0" 5 ns HIGH 50 %;
7. NET "clk90_bufg" TNM_NET = "SYS_clk90";
8. TIMESPEC TS_SYS_clk90 = PERIOD "SYS_clk90" 5 ns HIGH 50 %;
9. NET "clkdiv0_bufg" TNM_NET = "SYS_clkdiv0";
10.TIMESPEC TS_SYS_clkdiv0 = PERIOD "SYS_clkdiv0" 10 ns HIGH 50 %;
```

Figura 18 – Parte do arquivo UCF da HeMPS-S.

4.2.3 Planta Baixa (*Floorplaning*)

O número de PEs no MPSoC neste trabalho é limitado pelo número de blocos de memória (FPGA) disponíveis no dispositivo FPGA. O dispositivo selecionado contém 324 BRAMs. Cada memória RAM local do processador Plasma necessita de 16 BRAMs. Desta forma, o dispositivo alvo pode conter até 20 processadores.

A Figura 19 apresenta o resultado de uma síntese com a ferramenta de planta baixa no ambiente *PlanAhead* para uma instancia 2x3 da plataforma HeMPS-S. Cada PE utiliza

duas colunas de 8 BRAMs. A localização dos módulos ComEt e ConMe é escolhida para tirar partido da proximidade de pinos no dispositivo por eles utilizados.

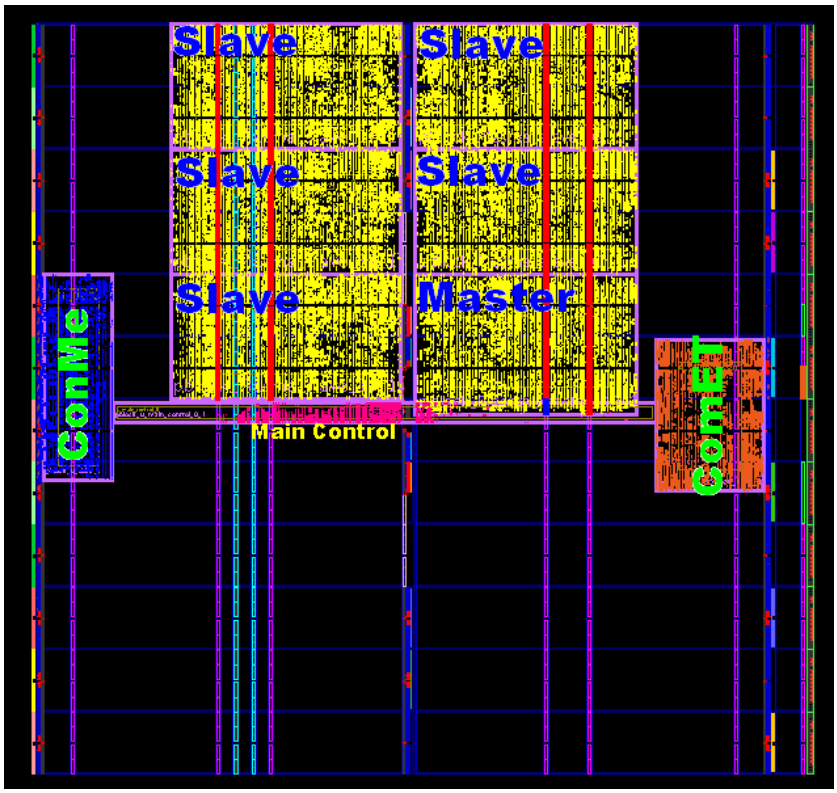


Figura 19 – Resultado de uma síntese com ferramenta de planta baixa da HeMPS-S no ambiente PlanAhead.

4.2.4 Árvore de Reset

O sinal de reset do MPSoC HeMPS é gerado pelo módulo Main Control, depois de finalizada a transmissão do repositório de tarefas. Um problema encontrado durante o processo de síntese é que a ferramenta não reconhecia o fio de reset para cada processador como um fio especial. Isto ocasionava falha na síntese, pois o atraso nestes fios excedia a restrição temporal. Para evitar erros de temporização, foram utilizados explicitamente componentes Xilinx do tipo buffer BUFG para ligar o sinal de reset para o MPSoC.

4.2.5 Interfaces Assíncronas

O módulo ComEt é composto por dois domínios de relógio. De um lado temos o Main Control lendo/escrevendo à 200 MHz e de outro lado o PHY lê/escreve à 25 MHz. Além disso, não existe nenhuma relação de fase entre os relógios, apesar de as freqüências serem múltiplas.

A solução adotada utiliza uma LUTRAM com escrita síncrona e leitura assíncrona para armazenar os dados e um sincronizador de dois flip-flops para cada sinal de controle. A Figura 20 apresenta o módulo de fronteira para processar a transmissão dos dados UDP, provindos do computador hospedeiro. O sinal *tx_ack* sinaliza que os dados estão

disponíveis para serem consumidos na LUTRAM. O sinal é sincronizado no outro domínio de relógio e a leitura dos dados tem início. Depois de o conteúdo da LUTRAM ter sido lido pelo Main Control, o sinal *tx_done_in* indica o final da leitura no domínio de relógio de 25 MHz, e o sinal de controle é sincronizado com o outro domínio. Uma estrutura análoga é utilizada na direção oposta.

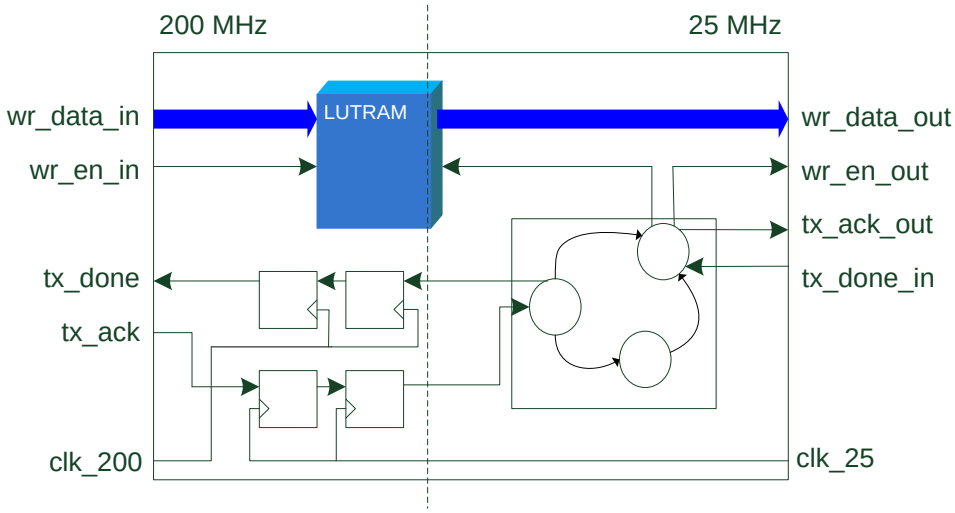


Figura 20 – Parte do módulo de transmissão UDP.

A outra fronteira de relógio se situa na interface entre o Plasma-IP mestre e o Main Control. O mestre requisita os dados do repositório com um relógio de 50 MHz, enquanto o Main Control acessa a memória DDR2 à 200 MHz. Entretanto estes relógios têm a mesma fase, um aspecto garantido pelo DCM. Para sincronizar esta interface um protocolo de quatro fases é utilizado (Figura 21). Em (1) o processador requisita a leitura do endereço `0x04000000` através do sinal *read_req*. O Main Control baixa então o valor de *data_valid*, pois o dado lido ainda não está disponível no barramento. Em (3), após o endereço ser lido na DDR2, pode-se sinalizar sua validade (sobe *data_valid*). Em (4) o mestre sinaliza que a leitura foi feita normalmente (baixa *read_req*). A primeira leitura é mais lenta que as subseqüentes, pois o controlador DDR2 lê palavras de 256 bits, executando portanto uma leitura de 16 palavras de 8 bits simultaneamente e, bufferizando-as.

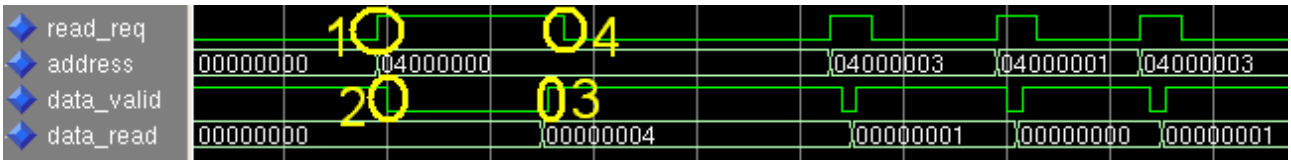


Figura 21 – Protocolo de quatro fases na interface entre o Plasma-IP mestre e o Main Control.

4.3 Ferramenta de Configuração e execução da HeMPS-Station

O usuário pode controlar a plataforma HeMPS-S através do computador hospedeiro, utilizando a ferramenta *HeMPS-S Generator*, cuja janela gráfica inicial aparece na Figura

22. Esta ferramenta é capaz de gerar um MPSoC com *Plasma_Routers*, além de controlar a e processo de execução da plataforma HeMPS-S. Dentre as suas funcionalidades, pode-se citar:

(i) **Configuração da Plataforma**: Pode-se gerar um MPSoC: número arbitrário de Plasma_Routers, através dos parâmetros X e Y; número máximo de tarefas executando por escravo; o tamanho da página (e conseqüentemente o tamanho da RAM); e o nível de abstração da descrição do processador (apenas para simulação). O usuário pode escolher entre versões ISS ou VHDL. Estas são equivalentes, sendo que o ISS simula mais rápido e o VHDL permite observar mais dados de depuração (sinais internos).

(ii) **Inserção de aplicações no MPSoC**: No painel esquerdo da Figura 22 podemos encontrar as aplicações e mapeá-las manualmente nos processadores escravos ou deixá-las no mestre para que este as mapeie. Na Figura 22, como exemplo, apresenta-se as aplicações *MPEG* e *communication* inseridas na Plataforma. A aplicação *communication* tem 4 tarefas, enquanto que a MPEG contém 5 (start, ivlc, idtc, iquant e print).

(iii) **Definição do mapeamento inicial de tarefas**: Note que na Figura 22 a tarefa start (tarefa inicial da MPEG) é mapeada no processador 01, e as tarefas taskA e taskB (tarefas iniciais da *communication*) estão mapeadas nos processadores 11 e 12, respectivamente. As tarefas restantes, designadas para o processador mestre, serão mapeadas dinamicamente durante a execução do MPSoC.

(iv) **Geração do código binário**: Através do botão *Generate*, as tarefas e o *µkernel* são compilados e o repositório é gerado com o mapeamento de cada tarefa. Neste momento o VHDL topo do MPSoC HeMPS é gerado com os dados parametrizados de X e Y para a síntese.

(v) **Definição dos endereços de rede**: IP e MAC. Atualmente não está implementado o protocolo DHCP, no qual a plataforma poderia receber um número de IP dinâmicamente.

(vi) **Botão *Debug***: Este botão gera uma janela gráfica (GUI) com os resultados de simulação, que são lidos a partir de um arquivo texto gerado pelo testbench da plataforma.

(vii) **Depurar a plataforma em FPGA**: O botão *Send to Board* executa quatro ações: *(1)* conecta o hospedeiro com o FPGA; *(2)* preenche a DDR2 com os códigos binários presentes no repositório; *(3)* inicia a execução da HeMPS, inicializando o MPSoC através do comando Start; e *(4)* recebe as mensagens de depuração.

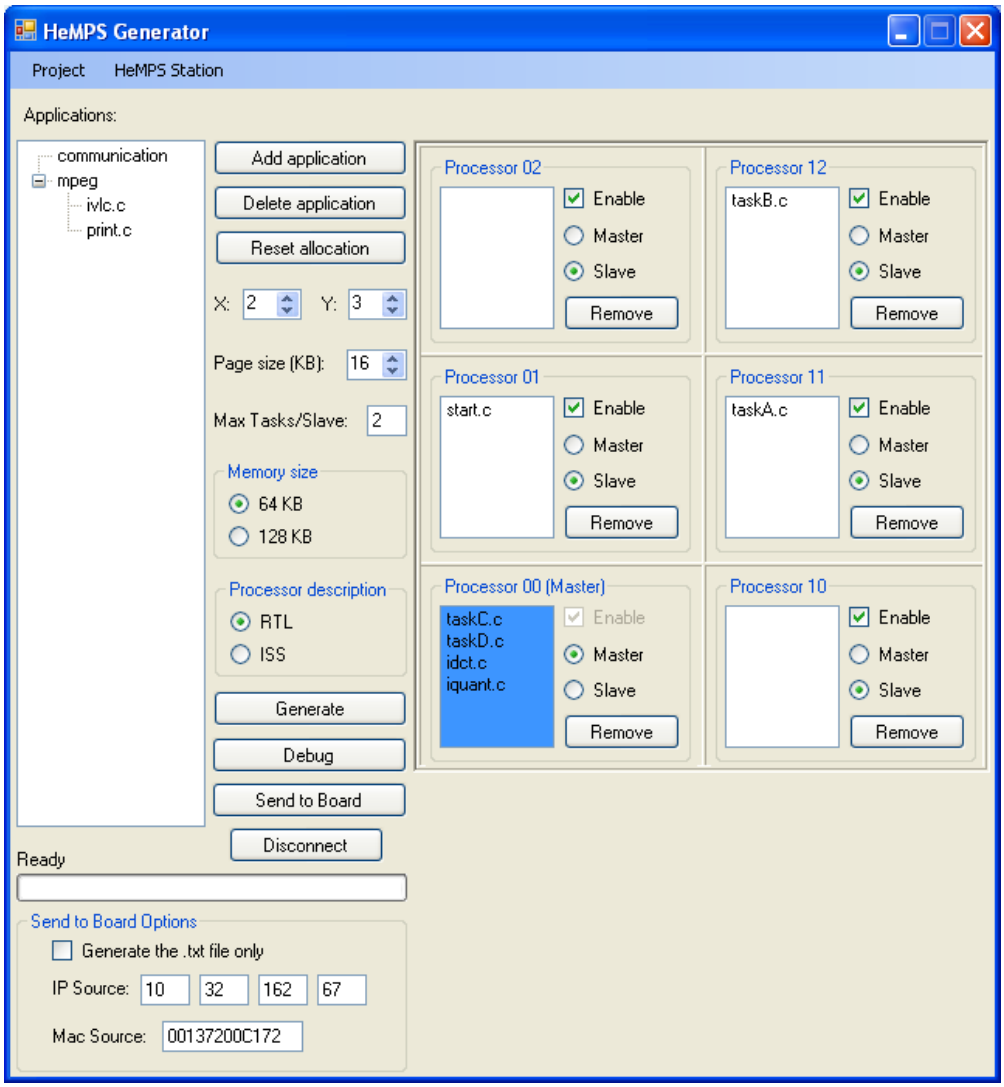


Figura 22 – A Ferramenta *HeMPS-S generator.*

Após inicializar o MPSoC, mensagens de depuração são geradas pelo Plasma-IP mestre. Estas mensagens são exibidas na tela da ferramenta, habilitando avaliação de desempenho em tempo de execução, como ilustra a Figura 23.

A tela com as mensagens de depuração é composta por um painel por processador. Neste exemplo, o painel correspondente ao Plasma-IP mestre (processador 00), exibe duas mensagens do *µkernel*. O processador 11 executa a tarefa A da aplicação *communication* e simultaneamente as tarefas C (*iquant*) e E (*print*) da MPEG. Estas mensagens possibilitam ao usuário verificar o comportamento correto da aplicação bem como medir o desempenho de cada tarefa. Os números depois de *"communication x started"* e antes de *"communication x finished"* correspondem ao número de ciclos de relógio gastos desde o início da execução, obtidos a partir da chamada de sistema `Gettick()`. As chamadas de sistema serão discutidas em detalhe no Capítulo 5.

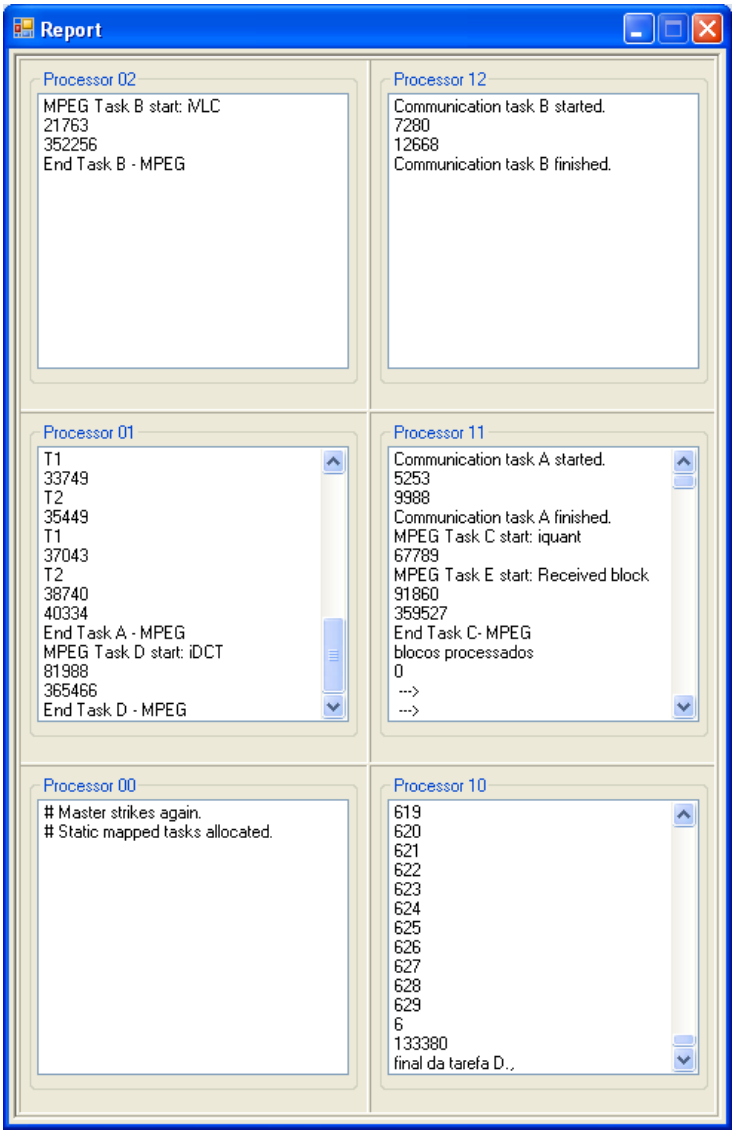


Figura 23 – Janela gráfica da depuração de hardware, exibindo os resultados enviados pelo FPGA.

4.4 Conclusão

O ambiente apresentado neste Capítulo, a plataforma HeMPS-S, mostrou ser um *framework* operacional para execução de aplicações distribuídas. Como contribuição deste Capítulo do trabalho, pode-se citar todo o processo de prototipação, descrito na Seção 4.2.

Este ambiente é capaz de gerar um MPSoC de tamanho parametrizável que pode ser simulado no nível RTL e prototipado em hardware. Além disto, o framework contém uma arquitetura de software para execução multitarefa e a parte de hardware com a arquitetura MPSoC.

Um aspecto importante deste framework é que a plataforma é disponível de forma livre e com código aberto. Por este motivo, o projetista pode utilizar a HeMPS-S como base para o desenvolvimento de novas técnicas tanto de software como de hardware.

Como se trata de plataforma prototipada em hardware, as modificações propostas pelo projetista podem servir como prova de conceito.

Como extensões para a plataforma HeMPS, atualmente estão sendo realizados os seguintes trabalhos:

(i) Generalização do PE, de forma que a execução das tarefas possa ser feita em processadores distintos, generalização esta descrita nos próximos Capítulos;

(ii) Inclusão de uma arquitetura de memória distribuída na HeMPS, com suporte à memórias cache, trabalho em andamento pelo mestrando Tales Marchesan Chaves;

(iii) Controle dinâmico de freqüência dos processadores, trabalho em andamento pelo mestrando Thiago Raupp da Rosa;

(iv) Inclusão de heurísticas de mapeamento dinâmico no *µkernel*, trabalho em andamento pelo mestrando Marcelo Mandelli;

(v) Inclusão de políticas de Qualidade de Serviço na NoC e no *µkernel,* trabalho em andamento pelo doutorando Everton Carara.

5 INFRA-ESTRUTURA DE SOFTWARE DO MPSOC HEMPS

Este Capítulo apresenta a *segunda contribuição* deste trabalho: o detalhamento do *µkernel* da HeMPS utilizado no processador PLASMA, e a identificação dos pontos onde este *µkernel* é dependente da arquitetura do processador.

Assume-se que as aplicações que executam no MPSoC são modeladas como um grafo de tarefas. Neste grafo, os vértices representam as tarefas das aplicações, e as arestas representam a comunicação entre estas. Por exemplo, o grafo da Figura 24 apresenta uma aplicação composta por quatro tarefas, onde as tarefas A e B enviam informações para a tarefa C, e esta por sua vez para a tarefa D.

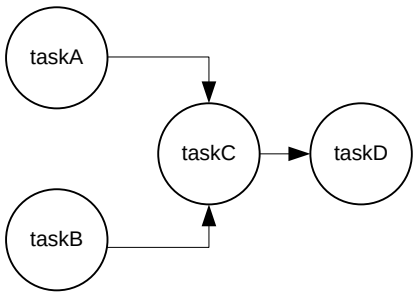


Figura 24 – Grafo de tarefas que modela uma dada aplicação.

Para uma execução multitarefa a memória de cada um dos processadores é dividida em páginas. Cada uma destas páginas pode conter o código de uma tarefa ou o próprio *µkernel*. O tamanho de cada página pode ser parametrizado de acordo com o tamanho da memória disponível. Neste trabalho, devido às restrições de memória na prototipação, a memória utilizada tem tamanho de 16 kB e é dividida em quatro páginas de 4 kB. O *µkernel* escravo atualmente ocupa 6 kB e por esta razão são utilizadas duas páginas para este. As outras duas páginas ficam disponíveis para tarefas do usuário.

O *µkernel* que executa em cada Plasma-IP é um núcleo sistema operacional (SO) o qual tem por objetivo gerenciar e dar suporte à execução de tarefas em cada processador. O *µkernel* da HeMPS é preemptivo, ou seja, cada tarefa utiliza o processador por um período pré-definido de tempo chamado *timeslice*. Existem duas versões do *µkernel*: uma que executa no Plasma-IP mestre, que tem por objetivo coordenar a distribuição e gerenciamento das tarefas, e não a execução propriamente de uma tarefa de uma aplicação. A outra versão é a que executa no Plasma-IP escravo, que tem como responsabilidade o controle da execução multitarefa e o tratamento de interrupções (tanto por software como por hardware) do processador local a ele, ou onde ele executa. Por efetivamente executar o processamento das tarefas, apenas o *µkernel* do Plasma-IP escravo é avaliado quanto à dependência da arquitetura do processador.

A Figura 25 apresenta a estrutura do *µkernel* e a estrutura genérica de dada tarefa, ambas apresentadas em camadas. Estas camadas são discutidas a seguir.

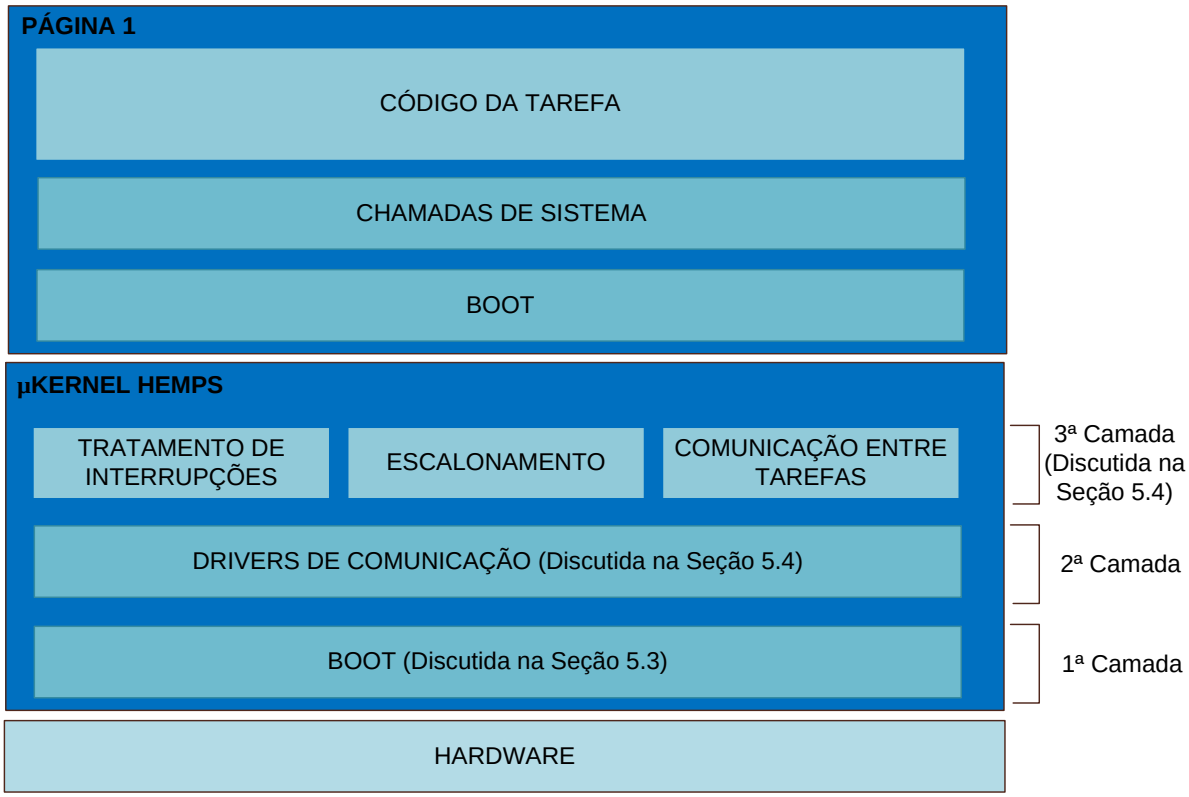


Figura 25 – Camadas de software da tarefa (Página 1) e do *µkernel* do Plasma-IP escravo do MPSoC HeMPS.

A tarefa é composta pelas camadas de “boot”, chamadas de sistema, e o próprio código da tarefa. A camada de boot é responsável pela inicialização da área de dados enquanto as chamadas de sistema apenas executam chamadas para funções externas à página da tarefa. O *µkernel*, por sua vez, é composto pelas camadas de boot, drivers de comunicação a terceira camada com várias funções.

5.1 Estrutura de Dados Específicas do *µkernel*

Para a execução e comunicação entre tarefas, o *µkernel* utiliza estruturas de dados específicas. Estas estruturas são responsáveis pelo controle do contexto da tarefa (*Task Control Block - TCB*), armazenagem e identificação de mensagens (*Pipe* e *Request Message*), e localização das tarefas (*Task Location*).

Task Control Block (TCB)

Esta estrutura (Figura 26) é responsável pelo armazenamento do contexto referente à execução da tarefa. Nela são armazenados os valores dos registradores do processador, o endereço de retorno para a tarefa (pc) após o atendimento de interrupção ou chamada de sistema, o identificador da tarefa (id), o endereço inicial da tarefa na memória (offset), e seu status. Note que a TCB (neste exemplo para o Plasma) contém apenas 30 registradores. Isto se deve ao fato de que o registrador $0 é a constante zero e o registrador $1 é reservado ao montador).

```
typedef struct {
    unsigned int reg[30];       //30 registers (Vn,An,Tn,Sn,RA)
    unsigned int pc;            //program counter
    unsigned int offset;        //initial address of the code
    unsigned int id;            //identifier
    unsigned int status;        //status (READY, WAITING, TERMINATED, RUNNING, FREE, ALLOCATING)
} TCB;
```

Figura 26 – Estrutura da Task Control Block.

Um vetor de TCBs é alocado estaticamente no *µkernel*, sendo que cada posição do vetor corresponde a uma página que pode receber uma dada tarefa. O campo *status* tem por função representar o estado atual da tarefa que está alocada em uma página, podendo esta tarefa encontrar-se em diferentes estados, conforme detalhado na Tabela 4.

Tabela 4 – Descrição dos estados possíveis das tarefas.

Status	Descrição
READY	A tarefa está pronta para executar
RUNNING	Indica que esta tarefa está executando na CPU
TERMINATED	Indica que a tarefa terminou sua execução
WAITING	A tarefa requisitou uma mensagem e está esperando pela resposta
FREE	Indica que a TCB está livre e uma tarefa pode ser alocada
ALLOCATING	Indica que a TCB está sendo alocada

A troca de contexto entre tarefas é feita através de rotinas de salvamento e recuperação dos conteúdos dos campos da TCB. É importante salientar que no momento em que ocorre uma troca de contexto, o conteúdo dos registradores não deve ser modificado até o salvamento destes na TCB. A rotina de salvamento de contexto armazena todos os valores dos registradores na TCB referente à tarefa, enquanto que a recuperação de contexto carrega todos os registradores com os valores armazenados na TCB. Estas duas rotinas são descritas em linguagem de montagem, pois devem ter controle sobre o valor de cada um dos registradores. Se descritas em linguagem C, por exemplo, o compilador poderia modificar o conteúdo de alguns registradores ao longo desta operação, algo indesejado na troca de contexto.

Por exemplo, quando uma tarefa A já executou durante o tempo definido no *timeslice*, se existir outra tarefa com *status* `READY`, esta deve ser escalonada. Neste momento, todos os registradores da tarefa A são salvos na sua TCB e uma tarefa B pode ser escalonada para execução. Os registradores da tarefa B são carregados da sua TCB e então a tarefa B pode iniciar sua execução.

Pipe

O *Pipe* é uma área de memória do *µkernel* destinada à comunicação entre tarefas. Nele ficam armazenadas (até serem consumidas) as mensagens que as tarefas enviam entre si. O *Pipe* é dividido em *slots* cuja estrutura é apresentada na Figura 27. Cada *slot* responsável por armazenar as informações de cada mensagen. O *Pipe* é implementado em software como um vetor com acesso aleatório. Desta forma, problemas como bloqueio por *head-of-line* (FIFO) e *deadlocks* são evitados.

Dentre os campos do *PipeSlot*, diversos são utilizados para identificação da mensagem. A mensagem propriamente dita contém tamanho parametrizável em tempo de compilação de até `MSG_SIZE` palavras (`message`[`MSG_SIZE`]), a indicação se o slot está sendo ou não utilizado (`status`), e a ordem da mensagem (`order`). O campo `order` é importante para garantir a entrega de mensagens na ordem nas quais estas foram geradas.

```
typedef struct {
    unsigned int remote_addr;       /* Remote processor address */
    unsigned int pkt_size;          /* NoC packet size (flits) */
    unsigned int service;           /* Service identifier */
    unsigned int local_addr;        /* Local processor address */
    unsigned int target;            /* Target task */
    unsigned int source;            /* Source task */
    unsigned int length;            /* Message lenght (32 bits words)*/
    unsigned int message[MSG_SIZE];
    enum PipeSlotStatus status;
    unsigned int order;
} PipeSlot;
```

Figura 27 – Estrutura de um slot do Pipe do MPSoC HeMPS.

TaskLocation, Request Message e current

A estrutura *TaskLocation* é responsável por vincular o endereço do processador na rede ao identificador da tarefa. Como as tarefas se comunicam através de um identificador de tarefas, fica transparente para o programador em qual processador a tarefa está alocada.

No modelo de comunicação utilizado na HeMPS as escritas são não bloqueantes, escrevendo-se diretamente no *Pipe*, enquanto que as leituras são bloqueantes. As leituras geram um pacote de *request_message,* e o processador destino possuindo em seu *Pipe* a mensagem, esta é enviada à origem. Caso contrário, estes pedidos de mensagens são armazenados na estrutura *RequestMessage*. No momento em que há uma escrita no Pipe, a estrutura *RequestMessage* é consultada, afim de se realizar, se possível, o envio da mensagem.

Além destas, um ponteiro na área de dados global, denominado `current`, é responsável por indexar a TCB da tarefa em execução. Através deste ponteiro é feito o salvamento e recuperação de contexto dos registradores nas TCBs e o escalonamento de tarefas.

5.2 Chamadas de Sistema

Para prover uma infra-estrutura de comunicação entre as tarefas e depuração de código, utilizam-se chamadas de sistema (*system calls*), as quais são de fato interrupções de software. As duas principais rotinas de comunicação são Send() e Receive(), que causam chamadas de sistema – `WRITEPIPE` e `READPIPE`, respectivamente. Para depuração, existe uma rotina de Echo(), que envia mensagens para o Plasma-IP mestre, que faz a comunicação com o mundo externo. A Tabela 5 apresenta as chamadas de sistema implementadas no *µkernel* do Plasma-IP escravo.

Tabela 5 – Chamadas de Sistema presentes no *µkernel* do Plasma-IP escravo.

Tipo	Descrição
EXIT	Indica que uma tarefa terminou sua execução. Não deve mais ser escalonada para execução e espera a leitura de todas as mensagens dela no Pipe,se houver.
WRITEPIPE	Escreve uma mensagem no Pipe, ou se esta já foi requisitada pela tarefa consumida a envia para tarefa destino.
READPIPE	Procura por uma mensagem no Pipe, se esta não existir, envia uma requisição à tarefa com a mensagem de origem e bloqueia a tarefa enquanto a mensagem não for recebida.
GETTICK	Retorna o valor de um contador global de ciclos de relógio, habilitando medidas de tempo de execução.
ECHO	Envia uma mensagem ao mestre, que a repassa para o mundo externo.

Para a execução de chamadas de sistema no Plasma, Woszezenki [WOS07] adicionou uma nova instrução ao processador, `syscall`, que efetua uma interrupção de software (trap). Esta instrução é parte do ISA do MIPS, mas não é nativa na versão 2.0 do Plasma, usada como base para o Plasma-IP. Ou seja, quando a instrução é executada, ocorre um salto para uma posição de memória pré definida (`0x44` no caso do Plasma), onde existe uma instrução de salto para a rotina de tratamento desta interrupção de software. Deve-se ressaltar que para a inserção desta instrução foram feitas modificações internas ao processador, o que é algo indesejado para a inclusão de novos processadores em um MPSoC. Para ilustrar o funcionamento, a Figura 28 apresenta o fluxo da execução da interrupção de software no Plasma.

Em (1) tem-se o código de uma tarefa qualquer executando. A primitiva de comunicação é mapeada através de uma macro em uma chamada da função `SystemCall`. A função `SystemCall` é escrita em linguagem de montagem no próprio código da tarefa na área de chamada de sistema da mesma (2), e executa a instrução `syscall`. A `syscall` executa um salto para a posição `0X044` (3), localizada na página do *µkernel*. Nesta posição existe outro salto para a rotina `system_service_routine` que faz o salvamento de contexto e chama a função `Syscall` (5) do *µkernel*, implementada em C, que efetivamente processa a requisição da chamada de sistema. Neste exemplo, a primitiva de

comunicação Send() é mapeada como uma chamada de sistema `WRITEPIPE`. Depois de executar a chamada de sistema, o *µkernel* recupera o contexto da tarefa (6) e esta retorna a execução (7).

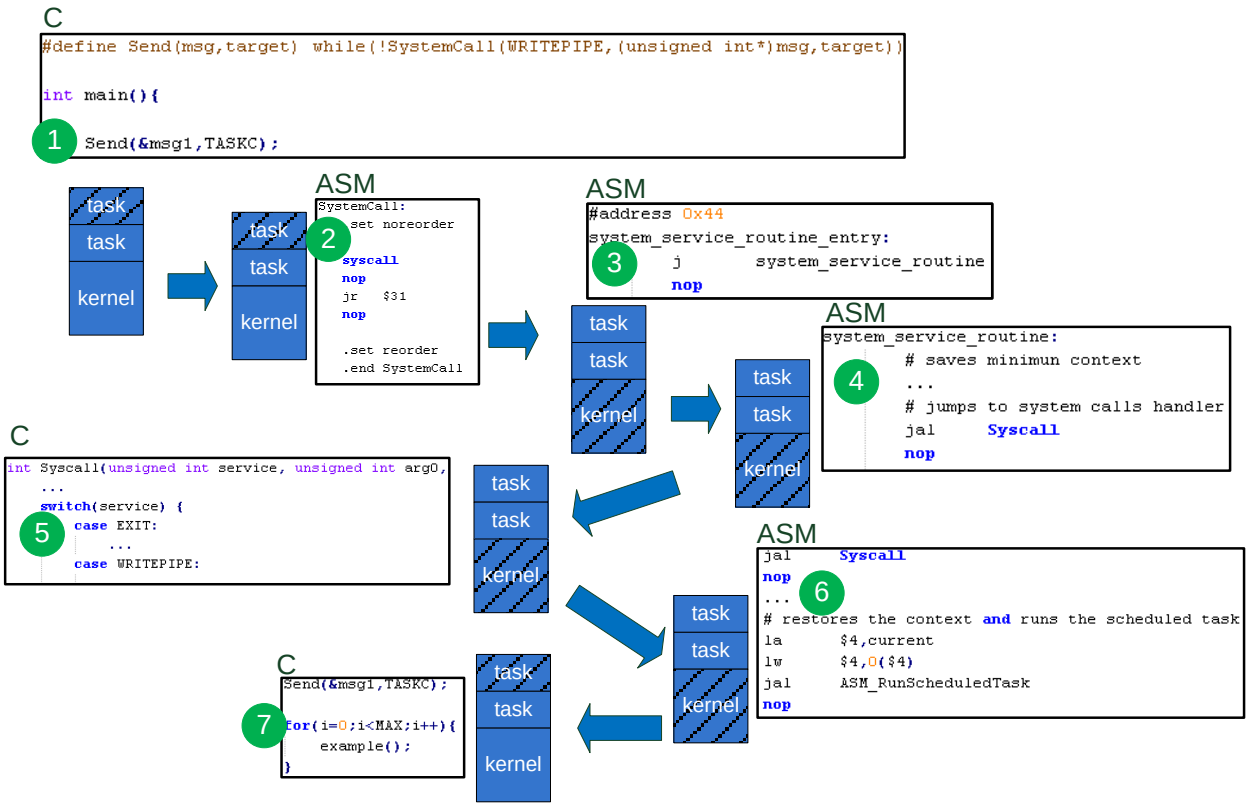


Figura 28 – Fluxo da rotina de chamada de sistema (system call) em uma tarefa qualquer.

5.3 BOOT (Primeira Camada do *µkernel*)

As primeiras posições de memória do código executável de um Plasma na HeMPS contém programação responsável pela inicialização do apontador de pilha (SP) e do ponteiro global (GP). Além disso, uma vez que tanto as tarefas como o *µkernel* são escritos em C, o código de boot é responsável por inicializar toda a seção de dados com zero para todas posições de memória. Este procedimento é adotado para garantir que as variáveis não incializadas pelo desenvolvedor de software conterão apenas zeros (o compilador também garante isso ao programador). Este código está descrito em linguagem de montagem e é dependente da arquitetura do Plasma e da forma como é gerado o código executável do *µkernel*. Depois de finalizado, o código de boot executa a chamada da função main, escrita em linguagem C, presente nas posições de memória subseqüentes.

A inicialização da área de dados, do SP e do GP é comum tanto para o *µkernel*, quanto para as tarefas. Este processo é feito por uma ferramenta (`convert_bin`) que

automatiza o processo. O processo de geração do código da uma tarefa e do *µkernel* é explicado em detalhe na Seção 5.7.

5.4 Drivers de Comunicação (Segunda Camada do *µkernel*)

Estes drivers são responsáveis pela comunicação entre do processador, através do *µkernel*, e os módulos de hardware que compõe o Plasma-IP. Os drivers de comunicação são acessados apenas no espaço de endereçamento do *µkernel*, evitando que as tarefas do usuário se comuniquem diretamente com módulos de hardware.

As principais rotinas desta camada são: *(i)* `NI_Write()` e `NI_Read()` escrevem e lêem dados na NI e *(ii)* `DMA_Send()` configura o DMA para fazer leituras e escritas na memória. Nesta camada, a rotina `DMA_Send()` é a única rotina que tem dependência da arquitetura, devido a arquitetura de memória utilizada. Esta dependência é detalhada na seção 6.3.1.

5.5 Terceira Camada do *µkernel*

Esta camada é responsável pelo tratamento das interrupções, comunicação entre as tarefas, e escalonamento de tarefas.

5.5.1 Tratamento de Interrupções

Dois tipos de interrupção de hardware são implementadas no Plasma-IP: *(i)* interrupção pelo módulo NI: sempre que o PE recebe algum dado da rede, é gerada uma interrupção provinda do módulo NI; *(ii)* interrupção de *timeslice*: indica que a tarefa atual executou durante o tempo pré-determinado e outra tarefa deve ser escalonada.

O tratamento de interrupções no processador Plasma ocorre da seguinte forma: quando o pino de interrupção é ativado, ocorre um salto para a posição de memória `0x04C`, a qual contém um salto para a rotina que executa o tratamento de interrupção. O valor do PC é armazenado no registrador `$ra`. Desta forma depois de executado o tratamento de interrupção, sabe-se o endereço onde ocorreu a interrupção e pode-se voltar a execução neste ponto. A Figura 29 apresenta um exemplo que ilustra o fluxo da rotina de tratamento de interrupção.

Antes de iniciar o tratamento da interrupção é feito o salvamento e no retorno a recuperação de contexto, assim como é feito nas chamadas de sistema. Apesar de serem procedimentos com o mesmo objetivo, no *µkernel* do Plasma-IP foram implementados por Woszezenki [WOS07] dois tipos de salvamentos de contexto: um para chamada de sistema e outro para o tratamento de interrupções. Apesar de semelhantes, são duas implementações distintas, onde o salvamento de contexto para chamadas de sistema é parcial (salvam-se apenas alguns registradores do total), e para interrupções é completo.

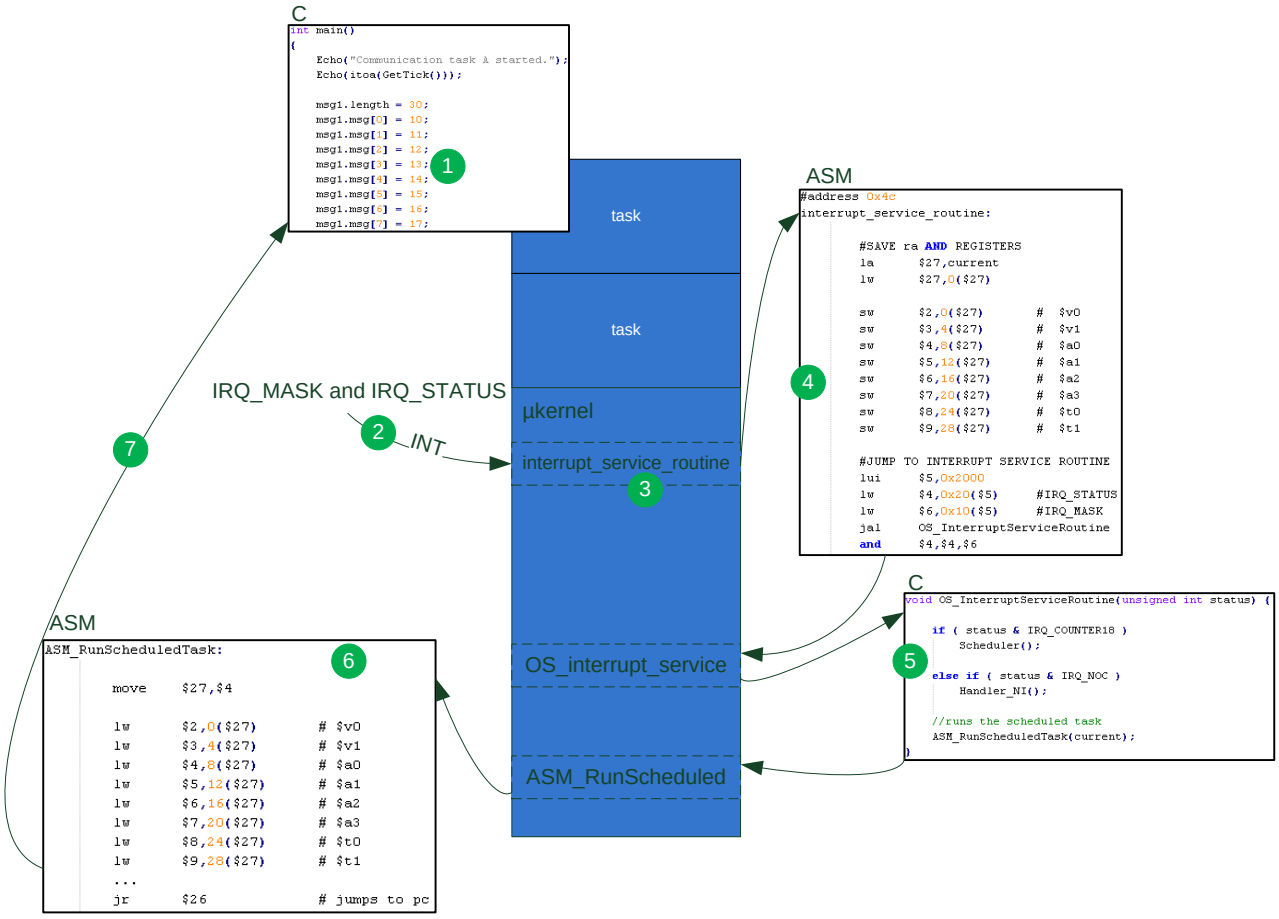


Figura 29 – Fluxo da rotina de tratamento de interrupção.

A interrupção pode ocorrer em qualquer momento da execução do código de uma tarefa (1). Se interrupções não estiverem mascaradas (2), o Plasma interrompe a execução e salta para a posição `0x4C` da memória (3). Nesta posição de memória há um salto para a `interrupt_service_routine` (4), que é a rotina responsável por fazer o salvamento de contexto e chamar a rotina de tratamento de interrupção `OS_interrupt_service` (5). Esta última é a rotina que efetivamente faz o tratamento da interrupção. O retorno da interrupção é feito através da rotina `ASM_RunScheduledTask (6)`, que recupera o contexto da tarefa (que pode ser diferente da que gerou a interrupção, no caso de uma interrupção *timeslice*) e retorna.

Como mencionado anteriormente, um dos tipos de interrupção é a de *timeslice*. Neste tipo, o processador executa a rotina de `Scheduler()`, o escalonador do sistema. Já a rotina `Handler_NI()` tem por responsabilidade o processamento dos pacotes recebidos pela NI. Os pacotes que trafegam na NoC contêm no cabeçalho a espeficicação de um dos serviços, detalhados na Tabela 6.

Tabela 6 – Descrição dos diferentes serviços que podem ser especificados em pacotes recebidos pelo *µkernel*.

Serviço	Código	Descrição
REQUEST_MESSAGE	0x00000010	Requisição de uma mensagem
DELIVER_MESSAGE	0x00000020	Entrega de uma mensagem previamente solicitada
NO_MESSAGE	0x00000030	Aviso de que a mensagem solicitada não existe
TASK_ALLOCATION	0x00000040	Alocação de tarefas: uma tarefa deve ser transferida pelo DMA para a memória do processador
ALLOCATED_TASK	0x00000050	Aviso de que uma nova tarefa está alocada no sistema
REQUEST_TASK	0x00000060	Requisição de uma tarefa
TERMINATED_TASK	0x00000070	Aviso de que uma tarefa terminou a sua execução
DEALLOCATED_TASK	0x00000080	Aviso de que uma tarefa terminou a sua execução e pode ser liberada
FINISHED_ALLOCATION	0x00000090	Aviso que o nodo mestre terminou a alocação inicial das tarefas (alocação estática)

5.5.2 Comunicação Entre Tarefas

A comunicação entre tarefas envolve chamadas de sistema, tratamento de interrupção (no caso de tarefas em processadores diferentes) e a utilização do *pipe* pelo *µkernel*. A comunicação entre tarefas é baseada na troca de mensagens através das primitivas Send() e Receive().

Quando uma dada tarefa executa a chamada de sistema Send(), a mensagem é armazenada no Pipe deste processador, e o processamento da tarefa continua. Isto caracteriza uma escrita não bloqueante. A chamada de sistema Receive() por sua vez, é bloqueante. Se a tarefa cuja mensagem foi requisitada está localizada no mesmo processador, a tarefa executa uma leitura diretamente no Pipe. Se a tarefa está localizada em outro processador, o *µkernel* envia uma requisição através da NoC (utilizando o serviço `REQUEST_MESSAGE`). O escalonador então carrega o status da tarefa com valor `WAITING` e esta não é mais escalonada, ou seja, bloqueia a tarefa aguardando a mensagem.

Quando a mensagem é recebida (identificada pelo serviço `DELIVER_MESSAGE`) pela NoC, acontece uma interrupção e o escalonador muda o status da tarefa de `WAITING` para `READY` e pode assim escalonar novamente a tarefa. A Figura 30 ilustra este processo. Em (a) assume-se que a tarefa t2 escreveu uma mensagem no Pipe, endereçada para a tarefa t4 (Send(&msg,t4)), e a tarefa t4 está requisitando a mensagem da tarefa t2 (Receive(&msg,t2)). Em (b) o processador 1 envia a mensagem requisitada para o processador 2 através da NoC. O sistema assegura o ordenamento na entrega de mensagens pois a primitiva Send() adiciona a cada mensagem a ordem em que elas foram escritas.

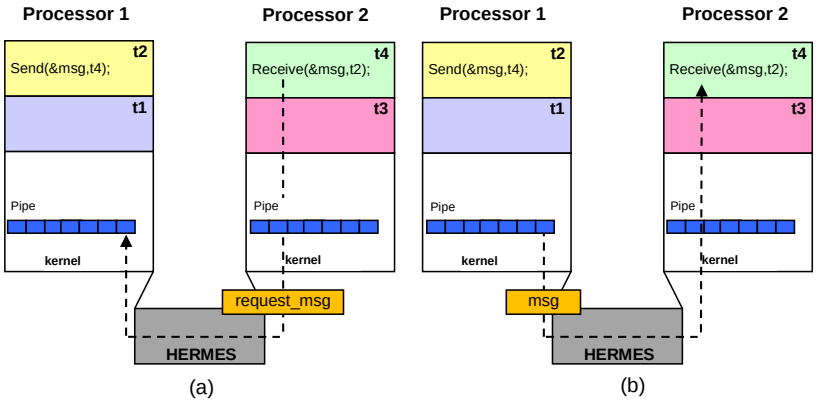


Figura 30 – HeMPS executando uma leitura remota de uma mensagem disponível. Adaptado de [CAR09a].

5.5.3 Escalonamento

A rotina `Scheduler()` (Figura 31) é responsável pelo escalonamento de tarefas no *µkernel*. Esta rotina utiliza um algoritmo *round robin* e é acionada sempre que uma tarefa executou por 16k (2^{14}) ciclos de relógio (este valor é parametrizável). Como mencionado anteriormente, esta rotina também é executada após uma chamada do comando *Receive*(). A chamada do comando *Receive*(), por ser bloqueante, aguarda o recebimento da mensagem requisitada. Enquanto aguarda o recebimento, outra tarefa pode ser escalonada.

Se a tarefa atual encontra-se em execução (`current->status == RUNNING`), ela passa para o estado `READY`. Com esta operação, a tarefa deixa de ser executada, podendo-se escalonar outra para execução. Neste momento, o algoritmo *round robin* entra em ação, procurando seqüencialmente a primeira tarefa pronta para executar (`status == READY`). Esta tarefa pode ser então escalonada e este procedimento é feito através do ponteiro `current`, que aponta para a TCB da tarefa executando.

```
void Scheduler() {
    static unsigned int round_robin = 0;
    int scheduled = FALSE;
    int i;
    if (current->status == RUNNING)
        current->status = READY;

    for (i=0; i<MAXLOCALTASKS; i++) {
        if (round_robin == MAXLOCALTASKS-1)
            round_robin = 0;
        else
            round_robin++;

        current = &(tcbs[round_robin]);

        if (current->status == READY ) {
            //ready to execute
            current->status = RUNNING;
            scheduled = TRUE;
```

Figura 31 – Trecho de código do escalonador do *µkernel*.

5.6 Identificação de Rotinas dependentes da Arquitetura do Processador

Esta Seção tem por identificar as rotinas de software dependentes da arquitetura. Esta identificação é importante, pois quando se quer portar o *µkernel* para outro processador, estas rotinas devem ser modificadas.

5.6.1 Salvamento de Contexto

A Figura 32 apresenta a descrição da rotina de salvamento de contexto nos processadores MB-Lite (b) e Plasma (a). Por serem descritas em linguagem de montagem, estas rotinas devem ser totalmente reescritas de acordo com a linguagem de montagem do processador alvo.

```
swi r18, r0, reg
la r18, r0, current
lw r18, r0, r18
swi r3, r18, 0
swi r4, r18, 4
swi r5, r18, 8
swi r6, r18, 12
swi r7, r18, 16
swi r8, r18, 20
swi r9, r18, 24
swi r10,r18, 28
swi r1, r18, 32
swi r2, r18, 36
swi r11, r18, 40
swi r12, r18, 44
swi r13, r18, 48
swi r14, r18, 52
swi r15, r18, 56
swi r16, r18, 60
swi r17, r18, 64
```

(a)

```
#REGISTERS $26 AND $27 ARE RESERVED FOR THE OS
la      $27,current
lw      $27,0($27)
sw      $2,0($27)       #  $v0
sw      $3,4($27)       #  $v1
sw      $4,8($27)       #  $a0
sw      $5,12($27)      #  $a1
sw      $6,16($27)      #  $a2
sw      $7,20($27)      #  $a3
sw      $8,24($27)      #  $t0
sw      $9,28($27)      #  $t1
sw      $10,32($27)     #  $t2
sw      $11,36($27)     #  $t3
sw      $12,40($27)     #  $t4
sw      $13,44($27)     #  $t5
sw      $14,48($27)     #  $t6
sw      $15,52($27)     #  $t7
sw      $16,56($27)     #  $s0
sw      $17,60($27)     #  $s1
sw      $18,64($27)     #  $s2
```

(b)

Figura 32 – Parte da descrição em linguagem de montagem das rotinas de salvamento de contexto para os processadores Plasma (b) e MB-Lite (a).

A rotina de salvamento de contexto utiliza um registrador para indexar a TCB da tarefa em execução através do ponteiro `currrent`. Note que nas primeiras linhas das duas rotinas de salvamento de contexto, as instruções carregam um registrador (`r18` em (a) e `$27` em (b)) com o valor do ponteiro `current`. No Plasma o registrador `$27` é destinado ao OS, ou seja, não existem dados válidos da tarefa nele, e por isso pode ser sobrescrito sem ser antes salvo. No MB-Lite não existe um registrador destinado ao OS, por este motivo o valor do registrador r18 é armazenado na variável global `reg` na primeira linha do código.

Depois de carregado o valor de `current`, o salvamento de todos os registradores é feito deslocando-se apenas na estrutura da TCB, pois os registradores estão organizados em uma estrutura de vetor (`TCB.reg[]`).

5.6.2 Handler_NI e Syscall

Algumas rotinas, como `Handler_NI()` e `Syscall()`, por exemplo, modificam os registradores da TCB diretamente. Os registradores modificados, como mostra a Figura 33, armazenam o valor de retorno da função e registrador de passagem de parâmetro.

```
case MESSAGE_DELIVERY:

    ...

    /* Points the target message (ReadPipe argument) */
    msg = (Message *)(target->offset | target->reg[3]);

    ...

    target->reg[0] = TRUE; // return true to ReadPipe()
    target->status = READY;

    if (idle)
        need scheduling = TRUE;
```

Figura 33 – Parte de código da função `Handler_NI()` responsável pelo tratamento do serviço `MESSAGE_DELIVERY`.

Por exemplo, quando acontece o recebimento de uma mensagem previamente requisitada por uma chamada de sistema, a tarefa que requisitou esta não está sendo escalonada, pois a leitura é bloqueante. O bloqueio acontece quando é executada a chamada de sistema, que não retorna à execução da tarefa.

A rotina `Handler_NI()` deve retornar para a chamada de sistema, por parâmetro, o ponteiro para a posição de memória onde a mensagem recebida se encontra (no exemplo `target->reg[3]`). O valor de retorno da chamada de sistema deve retornar um valor positivo (`target->reg[0]`). A tarefa, que não estava sendo escalonada, tem seu status modificado (`target->status = READY`) e pode ser escalonada novamente.

Para que não seja necessária a modificação do código C do *µkernel* de acordo com os registradores de passagem de parâmetro e retorno de função, optou-se por convencionar que os registradores da TCB serão salvos na forma apresentada na Tabela 7.

Tabela 7 – Convenção utilizada nos registradores da TCB.

Campo	Tipo de Registrador
TCB.Reg[0]	Valor de retorno 0
TCB.Reg[1]	Valor de retorno 1
TCB.Reg[2]	Argumento 0
TCB.Reg[3]	Argumento 1
TCB.Reg[4]	Argumento 2
TCB.Reg[5]	Argumento 3
TCB.Reg[6-31]	Outros

5.6.3 OS_Init

Esta rotina é responsável pela inicialização de todas as estruturas de dados utilizadas no *µkernel*, incluindo as TCBs. Por este motivo esta rotina também deve ser modificada para o porte do *µkernel*. Dentre suas ações, podem-se citar:

(i) Inicialização do SP e GP do kernel, e armazenamento destes em variáveis globais na área do kernel;

(ii) Leitura do endereço de rede do processador (leitura no registrador `NI_CONFIG`);

(iii) Inicialização do *Pipe*, zerando todas suas posições;

(iv) Inicialização das estruturas TaskLocation e RequestMessage;

(v) Inicialização das TCBs;

5.6.4 ASM_RunScheduledTask

Esta rotina é responsável pela recuperação de contexto e salto para a execução de uma tarefa. A rotina além de recuperar o contexto salvo previamente na TCB, deve carregar o valor do deslocamento na memória (registrador `PAGE`) da tarefa. Totalmente escrita em linguagem de montagem (como mostra a Figura 34), esta rotina deve ser reescrita de acordo com a linguagem de montagem do processador alvo.

```
   ...
lw      $16,56($27)     # $s0
lw      $17,60($27)     # $s1
lw      $18,64($27)     # $s2
lw      $19,68($27)     # $s3
lw      $20,72($27)     # $s4
lw      $21,76($27)     # $s5
lw      $22,80($27)     # $s6
lw      $23,84($27)     # $s7
lw      $24,88($27)     # $t8
lw      $25,92($27)     # $t9
lw      $28,96($27)     # $gp
lw      $29,100($27)    # $sp
lw      $30,104($27)    # $s8
lw      $31,108($27)    # $ra
lw      $26,112($27)    # $hi
mthi    $26
lw      $26,116($27)    # $lo
mtlo    $26
lw      $26,120($27)    # loads pc of the task that will run
lw      $27,124($27)    # loads offset of the task (for paging setup)

li      $1,0x1
mtc0    $27,$10
jr      $26             # jumps to pc
mtc0    $1,$12          # enables interrupts
```

Figura 34 – Parte da rotina de recuperação de contexto e retorno para a tarefa – `ASM_RunScheduledTask`.

Note-se que a recuperação de contexto é análoga ao salvamento de contexto, executando-se cargas indexadas em relação ao registrador `$27` (registrador do SO que aponta para a TCB corrente). Na antepenúltima instrução desta rotina, o registrador `PAGE`

é carregado através da instrução `mtc0 $27,$10` (o registrador `$10` do coprocessador 0 corresponde ao registrador `PAGE`, que no Plasma foi incluído no banco de registradores) [WOS07]. Notar também a carga do valor '1' no registrador `$1`, para posterior reabilitação de interrupções. Ao final da rotina ocorre o salto para onde a tarefa parou a execução antes da interrupção (através do registrador `$26`).

5.7 Processo de Geração de código Binário para o Processador PLASMA

O processo de geração do código que é carregado na memória RAM do processador envolve quatro passos. Este processo é apresentado na Figura 36, que ilustra o fluxo de geração. Esta figura simplifica o fluxo contido no trecho do arquivo *Makefile* mostrado na Figura 35, que ressalta a compilação do *µkernel* e da tarefa.

```
kernel_slave:
    $(AS_MIPS) -o boot_slave.o ..\kernel\slave\boot.S
    $(GCC_MIPS) -o kernel_slave.o ..\kernel\slave\kernel.c
    $(LD_MIPS) -Ttext 0 -eentry -Map kernel_slave.map -s -N -o kernel_slave.bin  boot_slave.o kernel_slave.o
    @$(DUMP_MIPS) --disassemble kernel_slave.bin > kernel_slave.lst
    ..\..\tools\convert_bin kernel_slave.bin kernel_slave.txt 24576

taskA_0:
    $(GCC_MIPS) "C:\wachter\hemps_generatorspecial\HeMPS 3.8x\Platform\software\applications\test\taskA.c" -o taskA_0.o
    $(LD_MIPS) -Ttext 0 -eentry -Map taskA_0.map -s -N -o taskA_0.bin BootTask.o taskA_0.o
    @$(DUMP_MIPS) --disassemble taskA_0.bin > taskA_0.lst
    ..\..\tools\convert_bin taskA_0.bin taskA_0.txt 16384
```

Figura 35 – Trecho do arquivo *Makefile*, responsável pela compilação do código do *µkernel* e da tarefa.

O arquivo *Makefile* é responsável por compilar o código C, montar o código em linguagem de montagem e ligar as funções de ambos. Após estes passos, é gerado um *dump* do arquivo binário final (`DUMP_MIPS`) para depuração e a ferramenta `convert_bin` é executada. Os números (24572 e 16384) passados como parâmetros indicam o tamanho máximo do código para uma página (ou mais no caso do kernel) da memória RAM.

A Figura 36 apresenta o fluxo utilizado no Makefile através de um diagrama. Em (1) o código objeto de BOOT da primeira camada é gerado pelo montador. Em (2) o código objeto restante das camadas do *µkernel* é gerado pelo compilador C. Estes dois códigos são unidos (3), de forma que o código de BOOT ocupe as primeiras posições da memória. Neste momento têm-se um código no formato ELF (*executable and linking format*) (4).

O formato ELF identifica seções como a área de dados e a área de instruções. Desta forma, a ferramenta `convert_bin` pode extrair as instruções e dados sem os cabeçalhos e determinar onde a área de dados inicia. É importante determinar onde a área de dados inicia, pois a `convert_bin` é responsável por inicializar esta com zeros, bem como inicializar o apontador de pilha (SP) e o ponteiro global (GP). Depois que estes foram inicializados, a ferramenta `convert_bin` pode gerar um formato em hexadecimal pronto para ser carregado na memória RAM (5).

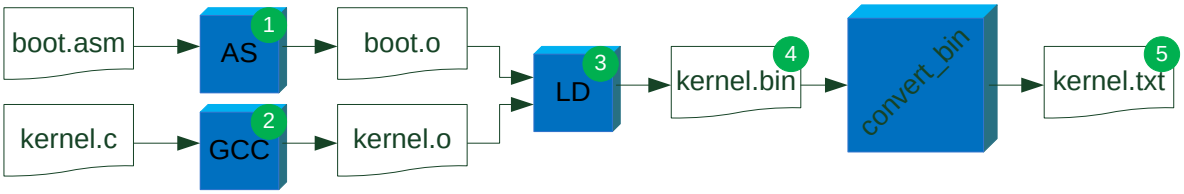


Figura 36 – Fluxo de geração do código binário do *µkernel* carregado na memória RAM.

A ferramenta `convert_bin` também deve verificar se o código da tarefa não ultrapassa o tamanho da página. Este cálculo é feito levando em consideração uma pilha com no mínimo 128 bytes.

6 INTEGRAÇÃO DE UM NOVO PROCESSADOR À PLATAFORMA HEMPS-S

O objetivo deste Capítulo é tornar o processo de inclusão de um processador diferente no MPSoC HeMPS mais fácil e rápido para o projetista, de acordo com as restrições abaixo. Para atingir este objetivo, primeiramente é preciso identificar os requisitos mínimos do processador. Estes requisitos, neste trabalho especificamente, são pertinentes à arquitetura do MPSoC HeMPS. Depois de definidos os requisitos, são apresentadas as modificações necessárias para o porte do µkernel para o processador MB-Lite.

O requisitos mínimos para incluir suporte a um novo processador na plataforma HeMPS-S envolvem:

(i) **Uma interface processador - memória local** (seja por organização Von Neumann ou Harvard): esta interface e o software associado, deve permitir mapear registradores em memória. Estes registradores são utilizados pelo *µkernel* para comunicação com os módulos de hardware do PE.

(ii) **Um processo de tratamento de interrupções**: Do ponto de vista de hardware é necessário no mínimo um pino de interrupção no processador. Do ponto de vista de software é necessária uma rotina em software para executar o tratamento de interrupção.

(iii) **Uma cadeia de ferramentas de geração de código**: O *µkernel* que executa nos PEs é praticamente todo escrito em linguagem C. Algumas rotinas são descritas em linguagem de montagem. Para tanto é necessário um compilador, um montador, e algumas ferramentas de desmontagem para análise do código gerado, para depuração. Uma suíte GCC é o ideal.

(iv) **Largura da palavra**: 32 bits. O MPSoC utilizado supõe que o processador conectado à rede tenha palavra de 32 bits.

6.1 Interface do Processador no Elemento de Processamento

O objetivo é manter a arquitetura do novo PE o mais similar possível do PE da HeMPS-S. Entretanto, a organização de memória do processador interfere na organização interna do PE, como ilustra a arquitetura apresentada na Figura 37. Em (a) tem-se uma organização para um processador com arquitetura Von Neumman (e.g. Plasma) e em (b) para organização Harvard (e.g. MB-Lite). A diferença entre estas é o acesso à memória. Enquanto que em (a) o processador tem apenas um barramento para acesso a RAM, em (b) tem-se dois barramentos, e um deles deve ser compartilhado com o DMA. A Figura que apresenta a arquitetura Harvard é de fato uma simplificação da

organização real, pois o DMA pode escrever tanto na área de instruções quanto na de dados.

Estas diferenças de implementação devem ficar transparentes para o desenvolvedor de software. O Anexo I apresenta a descrição do hardware do elemento de processamento com o processador MB-Lite. É importante ao leitor acompanhar como os módulos NI, DMA e RAM são interligados com o processador.

Como dito anteriormente, o principal fator que influencia na organização interna do PE é a arquitetura de memória do processador, pois nesta dissertação decidiu-se não modificar os módulos NI, DMA e RAM internamente. As modificações de acordo com a arquitetura de memória são detalhadas no decorrer deste Capítulo.

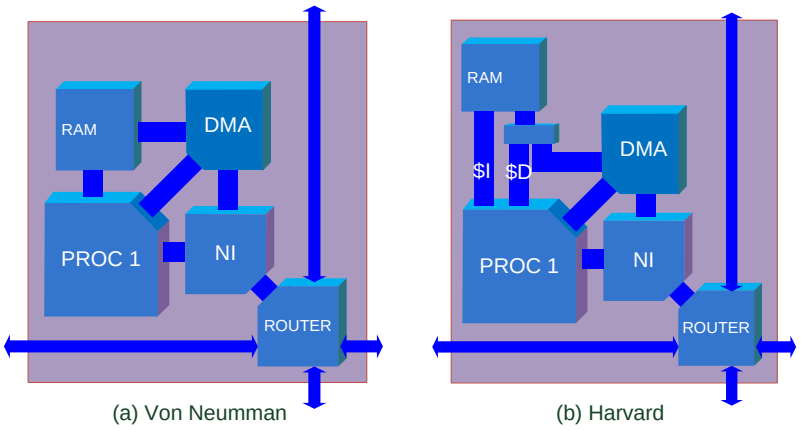


Figura 37 – Arquitetura do elemento de processamento, dependente da arquitetura de memória do processador.

6.2 Organização do PE Dependente da Arquitetura

Esta Seção tem por objetivo apresentar a organização interna do elemento de processamento, identificando as modificações executadas para o porte do *µkernel* para o MB-Lite.

6.2.1 Paginação

O *µkernel* da plataforma HeMPS-S deve implementar um sistema onde um processador possa executar mais de uma tarefa, ou seja, um sistema multitarefa. Nesta plataforma, o que possibilita esta operação é o mecanismo de paginação. Este método permite que os códigos das tarefas sejam compilados independentemente. Durante a execução de cada tarefa, o endereço gerado pelo compilador, endereço lógico, é concatenado com o conteúdo de um registrador (neste caso, o registrador `PAGE`), resultando em um endereço físico, como ilustrado na Figura 38. Este mecanismo habilita que uma memória seja dividida em páginas, onde cada uma destas páginas pode conter o código de uma tarefa. Este método implica em uma proteção de acesso aos dados das tarefas, pois uma tarefa não tem mecanismos para acessar áreas de outras tarefas.

O mecanismo de paginação é controlado pelo *µkernel* (controle do registrador `PAGE`). A troca entre páginas ocorre através de uma interrupção. Por exemplo, assuma-se um processador com tarefas mapeadas nas páginas 2 e 3. A tarefa da página 2 executa durante um período de tempo, definido pela constante *timeslice*, e após este período gera-se uma interrupção. O tratador de interrupção, no *µkernel*, escalona a próxima tarefa, alterando o registrador `PAGE` para o valor 3.

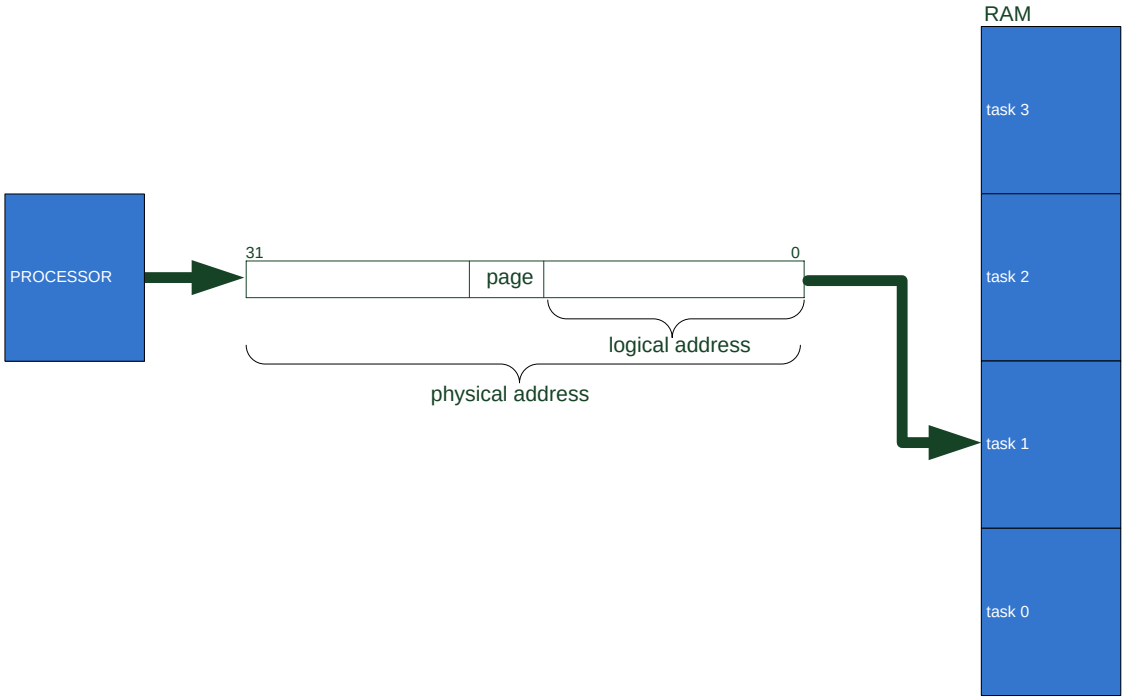


Figura 38 – Mecanismo de paginação utilizado pelo *µkernel* da HeMPS. Neste caso, o processador utiliza um endereço lógico qualquer, mas na realidade está acessando a memória deslocada pelo conteúdo do registrador `PAGE`.

A paginação no processador Plasma foi feita da seguinte forma. Ao banco de registradores foi adicionado o registrador, `PAGE`. A configuração da página é realizada através da instrução `mtc0 reg,$10`. Nesta instrução, o endereço inicial de uma tarefa da memória (offset) contido no registrador `reg` é carregado para o registrador `$10` do CP0 (co-processador 0), que corresponde ao registrador `PAGE`.

Esta abordagem exige do projetista do hardware o conhecimento do funcionamento interno do processador e a validação destas modificações em hardware. Para evitar que seja necessário alterar-se o processador para implementar este procedimento em outras arquiteturas, optou-se por efetuar uma abordagem no nível de software, que consiste em mapear o registrador `PAGE` em memória. Sendo assim, toda vez que há a necessidade de troca de página na memória, chama-se a função: `MemoryWrite(PAGE,value)`.

O problema desta abordagem é que a função `MemoryWrite()` modifica alguns registradores na sua execução, pois deve-se passar parâmetros à esta função. Desta forma, depois que a recuperação de contexto é feita, deve-se executar a troca para a

página da tarefa. Visto que a função `MemoryWrite()` modifica alguns registradores, a recuperação de contexto é perdida.

Para resolver este problema, criaram-se dois registradores mapeados em memória: `CURRENT_PAGE` e `NEXT_PAGE`. O `CURRENT_PAGE` é o registrador que efetivamente é usado na concatenação nos endereços de memória em hardware, enquanto que o registrador `NEXT_PAGE` é utilizado pelo `CURRENT_PAGE` apenas no retorno da rotina de interrupção. A Figura 39 apresenta um trecho de código `ASM_RunScheduledTask()`, responsável pela recuperação de contexto. Note que o registrador `NEXT_PAGE` é carregado antes da recuperação de contexto. O conteúdo de `CURRENT_PAGE` recebe o valor de `NEXT_PAGE` no retorno à tarefa, saltando para a faixa de endereços da tarefa.

```
void ASM_RunScheduledTask(){
    MemoryWrite(NEXT_PAGE,current->offset);
    __asm__ volatile (  "la r18, r0, current;"
                        "lw r18, r0, r18;"
                        " lwi r1, r18, 4;"
                        " lwi r2, r18, 8;"
                        " lwi r3, r18, 12;"
                        " lwi r4, r18, 16;"
                        " lwi r5, r18, 20;"
                        " lwi r6, r18, 24;"
                        " lwi r7, r18, 28;"
                        " lwi r8, r18, 32;"
                        " lwi r9, r18, 36;"
                        "lwi r10, r18, 40;"
                        "lwi r11, r18, 44;"
                        "lwi r12, r18, 48;"
                    ...
```

Figura 39 – Escrita no registrador `NEXT_PAGE` antes da recuperação de contexto.

6.2.2 Interrupções

Outro requisito para a execução do *µkernel* da HeMPS é a existência de rotinas para o tratamento de interrupção. Na parte de hardware é necessária a existência de no mínimo um pino de interrupção de E/S no processador.

O tratamento de interrupção no MB-Lite é semelhante ao processo executado no Plasma, onde o *µkernel* é responsável pelo tratamento de interrupções. Como uma interrupção pode acontecer quando o processador está executando o código de uma tarefa, deve-se saltar para a página que contém o *µkernel*. Por este motivo o tratamento de interrupção é bastante atrelado ao mecanismo de paginação.

No MB-Lite, semelhantemente ao Plasma, quando o pino de interrupção é ativado, ocorre um salto para a posição de memória `0x010`, a qual contém um salto para a rotina que executa o tratamento de interrupção (`OS_interrupt_handler`). Paralelamente a esta operação, o valor do PC é armazenado no registrador `r14`. Desta forma depois de executado o tratamento de interrupção, sabe-se o endereço onde ocorreu a interrupção e pode-se voltar à execução neste ponto.

Quando o processador acessa o endereço `0x010`, o hardware carregado com o valor zero, ocorrendo um salto para a faixa de endereços do *µkernel* (este mecanismo de hardware é apresentado na linhas 134-135 da Figura 40). Em seguida, o *µkernel* do MB-Lite carrega o registrador `r14` com o contador de programa (PC) atual da tarefa. Então a rotina `OS_interrupt_handler` pode executar o tratamento da interrupção.

```
134  if imem_o.adr_o = x"00000010" then
135      current_page_reg <= (others => '0');
136  elsif imem_i.dat_i = x"b62e0000" then
137      current_page_reg <= next_page_reg;
138  end if;
```

Figura 40 – Trecho do hardware de controle de paginação do processador MB-Lite (apresentado no Anexo I) que carrega o registrador `CURRENT_PAGE` com o valor da página do *µkernel*.

No processador MB-Lite, o retorno da rotina de interrupção é feito através da instrução `rtid rA, IMM`, de forma que o processador execute um salto para a o endereço de memória especificado pelo conteúdo do `rA` somado com a constante `IMM`. Por convenção utiliza-se aqui o registrador `r14` como `rA`.

Para o retorno da interrupção à página da tarefa, o hardware do PE monitora o barramento de instruções. Quando o *opcode* `rtid r14, 0` é detectado (valor `0xb62e0000` - linhas 136-138 da Figura 40), o hardware provoca o retorno à tarefa que estava executando, carregando-se o PC com o valor de r14 e `CURRENT_PAGE` com o valor de `NEXT_PAGE`.

O procedimento de tratamento de interrupção, juntamente com a paginação é ilustrado na Figura 41. Em (1) ocorre uma interrupção e o registrador `CURRENT_PAGE` é carregado com zero, causando acesso à área do *µkernel*. Nesta posição existe um salto para a rotina de salvamento de contexto (denominada `interrupt` na figura) (2). O tratamento da interrupção é executado pela rotina `OS_interrupt_handler` (3) de forma equivalente à implementação no Plasma.

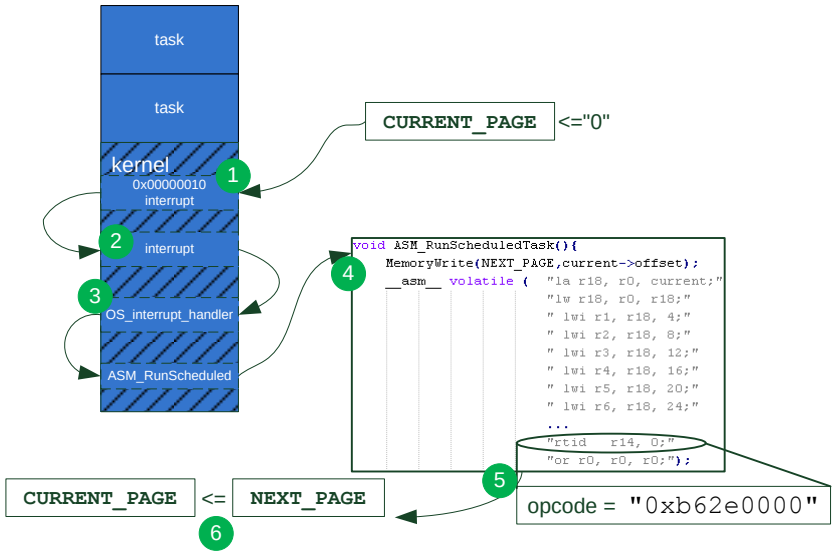


Figura 41 – Esquema de entrada e retorno de interrupção com paginação.

Depois de executado o tratamento de interrupção, pode-se retornar à execução da tarefa, através da rotina `ASM_interrupt_handler` (4) que executa a recuperação de contexto. Como descrito na Figura 39, o registrador `NEXT_PAGE` recebe o valor da página na qual a tarefa se encontra. Os valores dos registradores são recuperados e em (5), ao acessar o opcode de retorno de interrupção, o registrador `CURRENT_PAGE` é carregado com o valor de `NEXT_PAGE` e retorna à faixa de endereços da tarefa.

6.3 Chamadas de Sistema

Como explicado no Capítulo anterior, a comunicação entre os PEs é feita através de chamadas de sistema implementadas como interrupções de software – *traps*. No processador Plasma esta interrupção foi implementada adicionando-se uma instrução específica (`syscall`) que executa um salto para a rotina de chamada de sistema.

Esta abordagem foi descartada para o MB-Lite, pois implicaria na modificação interna do processador. No processador MB-Lite, a chamada de sistema é implementada como uma nova interrupção de hardware. Desta forma, ao executar uma chamada de sistema, a tarefa força a execução de uma interrupção. Esta abordagem, além de não necessitar a modificação do processador, elimina a necessidade de utilização de duas rotinas de salvamento de contexto (parcial e completo), como implementado no Plasma.

Para implementação da chamada de sistema como uma interrupção, adicionou-se ao PE do MB-Lite o registrador mapeado em memória, denominado `SYS_CALL`. O bit menos significativo deste registrador é ligado ao registrador `IRQ_STATUS`. A escrita de um valor ‘1’ neste bit do registrador, gera uma interupção de hardware. A Figura 42 apresenta a nova configuração do registrador `IRQ_STATUS`, onde o bit 5 adicionado identifica uma interrupção de chamada de sistema. Os bits 4 e 3 identificam interrupção de NI e *timeslice*, respectivamente, já existentes na implementação do Plasma.

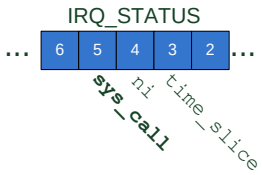


Figura 42 – Configuração do registrador `IRQ_STATUS` para o processador MB-Lite. Em destaque, o bit de interrupção `SYS_CALL` adicionado.

Como exemplo do fluxo de chamadas de sistema no MB-Lite, a Figura 43 ilustra uma chamada de sistema. Em (1) temos a chamada de sistema `Send()` que é mapeada em uma chamada da rotina `SystemCall()` em (2). A `SystemCall()` apenas carrega os parâmetros nos registradores e escreve o valor ‘1’ no registrador `SYS_CALL`. Esta escrita gera uma interrupção em hardware no processador (3), ocorrendo um salto para a página do *µkernel*. Em (4) ocorre o salto para a rotina em linguagem de montagem (`interrupt`)

responsável pelo salvamento de contexto (5). Neste ponto, é executada a rotina `OS_interrupt_handler()` que é responsável por identificar o tipo de interrupção (6) – neste caso de chamada de sistema. Então a rotina `Syscall()`, responsável pelo tratamento de chamadas de sistema é executada (7). Feito isto, pode-se recuperar o contexto da tarefa (8) e retornar para a tarefa (9). O passo 9 é equivalente aos passos 5-6 da Figura 41.

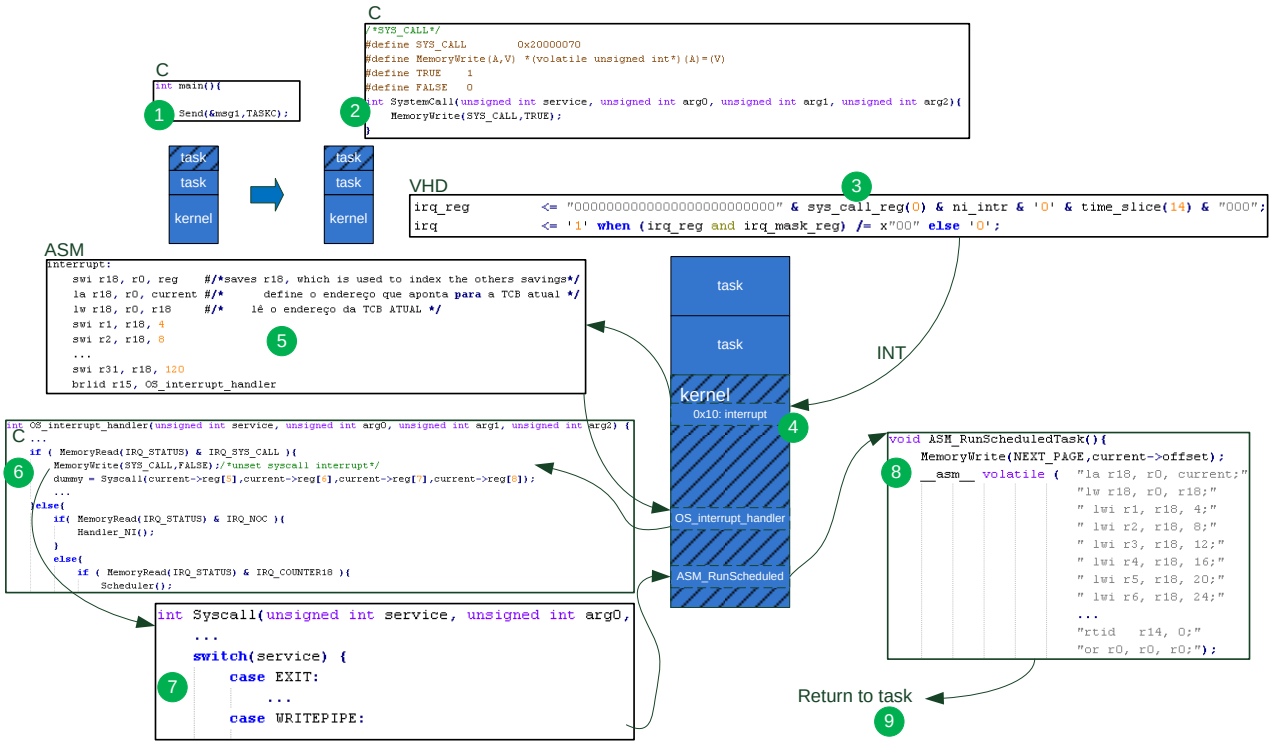


Figura 43 – Fluxo de execução de uma chamada de sistema para o processador MB-Lite através de interrupção.

6.3.1 Controle de Acesso ao DMA

O DMA é o módulo de hardware responsável por controlar a escrita/leitura de grandes blocos de informação da memória. Por exemplo, quando o *µkernel* recebe o código de uma tarefa, este código deve ser escrito em uma página da memória. Esta operação não requer computação, apenas escritas na memória. Desta forma, comunicação e a computação são operações separadas.

A arquitetura do Plasma-IP foi projetada utilizando memórias BRAM com porta dupla, presentes nos FPGAs da Xilinx. Estas memórias possuem dois barramentos que podem ser usados simultaneamente para escrita ou leitura. Note-se que a arquitetura do processador Plasma é Von Neumman, ou seja, existe apenas um barramento para a memória de instruções e de dados. Por este motivo, o processador ocupa apenas um dos barramentos de acesso a memória RAM, disponibilizando o outro barramento para o módulo de DMA.

O processador MB-Lite, por sua vez, apresenta uma arquitetura Harvard, que supõe a existência de duas memórias: uma para instruções e outra para os dados. Dado que a memória RAM utilizada é dupla porta, uma porta é conectada ao barramento de instruções e a outra porta ao barramento de dados. Desta forma, a existência de duas memórias disjuntas é virtualizada pelo uso de uma memória de dupla porta. Uma vantagem desta abordagem é que se consegue carregar o conteúdo das instruções e dados por um único barramento, dado que qualquer um dos barramentos tem acesso a todo o espaço de memória.

Nesta abordagem, nenhuma das portas da memória RAM fica disponível unicamente para o DMA. Por este motivo foram necessárias modificações no acesso à memória pelo DMA. Um dos barramentos da memória RAM teve seu acesso compartilhado conforme apresentado na Figura 44. A memória de instruções é utilizada com muita freqüência, visto que a cada ciclo uma nova instrução é lida da memória. Por este motivo, o barramento da memória de dados (porta A da Figura 44) foi selecionado para ser compartilhado com o DMA. Esta modificação em hardware é descrita nas linhas 58 a 64 do Anexo I.

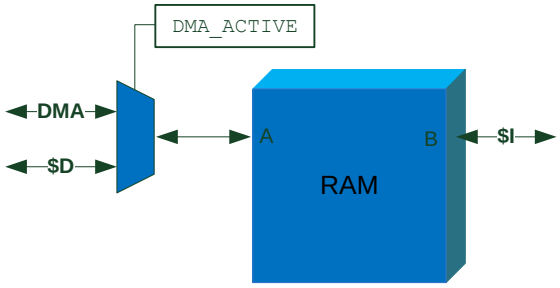


Figura 44 – Estrutura de acesso à memória RAM, mostrando o compartilhamento de acesso pelo DMA e pelo barramento de dados do MB-Lite.

É importante salientar que a configuração do DMA é feita pelo processador, mas quem efetivamente controla o acesso à memória é o DMA (através do sinal `DMA_ACTIVE`). Desta forma, deve-se garantir que quando o DMA está acessando a memória, o processador não tente executar acessos à memória de dados.

A solução encontrada foi adicionar depois das operações de *start* no DMA uma pausa na leitura do barramento de dados. Esta pausa é implementada através de um laço de leitura de um registrador mapeado em memória que contém o próprio sinal `DMA_ACTIVE`. A Figura 45 apresenta o código da função `DMA_Send()` onde este laço foi adicionado. Note que depois de ativar o DMA (`MemoryWrite(DMA_START, 1)`) o laço (`while (MemoryRead(DMA_ACTIVE));`) aguarda que o conteúdo do registrador DMA_ACTIVE assuma o valor 0. Este procedimento é equivalente à uma suspensão do processador (*hold*) durante o DMA. Dado que nos requisitos dos processador não há esta funcionalidade prevista, buscou-se desenvolver um método que evitasse a alteração do código do processador.

Esta abordagem tem por desvantagem uma perda de desempenho se comparado com o processador Plasma, pois o processador MB-Lite fica ocioso enquanto o DMA está operando, diferentemente do Plasma. Entretanto, há ganho de desempenho se comparado à uma execução com acessos a memória por software.

```
void DMA_Send(PipeSlot* slot) {
    /* Waits the DMA availableness */
    while ( MemoryRead(DMA_ACTIVE) );

    /* Sets the block size in 32 bits words */
    MemoryWrite(DMA_SIZE, (slot->pkt_size>>1) + 2);

    /* Sets the DMA operation (Memory read) */
    MemoryWrite(DMA_OP, READ);

    /* Sets the block start address */
    MemoryWrite(DMA_ADDRESS, (unsigned int)slot);

    /* Fires the DMA */
    MemoryWrite(DMA_START, 1);

    /* Waits the DMA write */
    while ( MemoryRead(DMA_ACTIVE) );
}
```

Figura 45 – Rotina `DMA_Send()` com laço de pausa para leitura do registrador mapeado em memória `DMA_ACTIVE` após o start do DMA.

6.4 Processo de Geração do Binário para o Processador MB-Lite

O processo de geração do código binário para o processador MB-Lite é semelhante ao processo do processador Plasma. Os dois processadores têm uma parte do código em linguagem de montagem e outra em linguagem C. A geração de código objeto para o MB-Lite requer apenas um arquivo com descrição em linguagem de montagem, contendo o salvamento de contexto. O restante das rotinas em linguagem de montagem (recuperação de contexto, inicialização das TCBs, entre outras) estão contidas no código C através de pragmas.

Poder-se-ia utilizar o salvamento de contexto e o salto para a rotina `interrupt_handler` gerados pelo compilador mb-gcc. Porém, esta rotina realiza modificações na pilha e em registradores, causando perda no contexto da tarefa (TCB). A solução adotada foi escrever o salvamento de contexto e o salto para a rotina de tratamento de interrupção (`OS_interrupt_handler`) em um rotina escrita em linguagem de montagem, denominada `interrupt` e descrita no arquivo `interrupt.s`.

A Figura 46 apresenta o fluxo para geração do código binário, a carregar memória RAM do processador MB-Lite. Após a montagem (1) e a compilação (2) dos códigos em linguagem de montagem e C, respectivamente, uma ferramenta desenvolvida durante no escopo desta Dissertação (denominada `convert`) modifica (4) o código binário gerado a

partir do arquivo ELF (3). Esta ferramenta escreve o endereço da rotina `interrupt` na posição `0x10` da memória. Desta forma, quando ocorrer uma interrupção, haverá um salto para esta rotina de salvamento de contexto em linguagem de montagem (5).

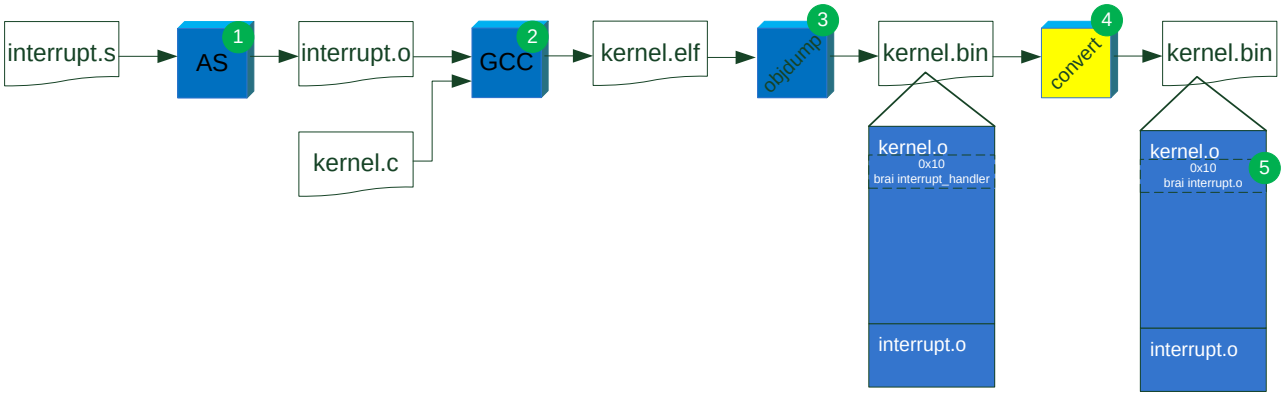


Figura 46 – Fluxo de geração do código binário do *µkernel* carregado na memória RAM do processador MB-Lite.

Diferentemente da abordagem utilizada pelo Plasma, neste fluxo a área de dados não é inicializada pela ferramenta `convert`. Este processo é feito em tempo de execução, por rotinas geradas automaticamente pelo compilador `mb-gcc`. Desta forma, quando o *µkernel* ou uma tarefa tem sua execução iniciada, antes do início da rotina `main`, o processador executa a inicialização da área de dados. Esta inicialização é executada somente no ínicio do *µkernel* e para cada tarefa. Esta abordagem tem como vantagem a não necessidade do desenvolvimento de uma ferramenta específica para inicialização de área de dados. Como desvantagem, temos a ocupação da memória por rotinas que não executam processamento propriamente dito, além do aumento no tempo do início da execução do *µkernel*/tarefa.

6.5 Repositório de Tarefas com Códigos para Processadores Distintos

O repositório de tarefas para o MPSoC heterogêneo não necessitou modificações. Isto se deve ao fato de o processador mestre enviar o código binário das tarefas independentemente da arquitetura. Sendo assim, não há diferenciação se o código que este está enviando é relativo a um processador MB-Lite ou Plasma. A responsabilidade de mapear as tarefas de acordo com a arquitetura do processador é do usuário.

O fluxo de geração de um repositório é feito de forma automática para arquiteturas homogêneas Plasma ou MB-Lite. A criação de um repositório contendo códigos objetos de ambos processadores de arquiteturas heterogêneas também é feita de forma automática.

Ressalta-se que não há mapeamento automático de tarefas no presente trabalho. O projetista deve indicar a posição e o tipo de cada processador conectado à rede, e

mapear manualmente as tarefas nos processadores. Existe também a necessidade de o processador mestre ser do tipo Plasma, pois como dito anteriormente, portou-se apenas o *µkernel* escravo.

6.6 MPSoC HeMPS-S Heterogêneo

O MPSoC heterogêneo proposto é formado por elementos de processamento cujo núcleo pode ser o processador Plasma ou o MB-Lite. Cada um destes PEs é composto pelo (i) roteador e pelo (ii) núcleo de processamento (MB-Lite-IP ou Plasma-IP). Os módulos MB-Lite-IP e Plasma-IP são compostos pelo (i) processador (MB-Lite ou Plasma), (ii) DMA, (iii) NI e (iv) RAM.

A geração de um MPSoC homogêneo é feita de forma automatizada. Depois de gerado o arquivo de topo com o MPSoc, o usuário deve escolher manualmente qual o tipo de núcleo de processamento de cada PE. A escolha é feita através de um atributo *generic* no código VHDL. A hierarquia com os tipos de PE é apresentada na Figura 47.

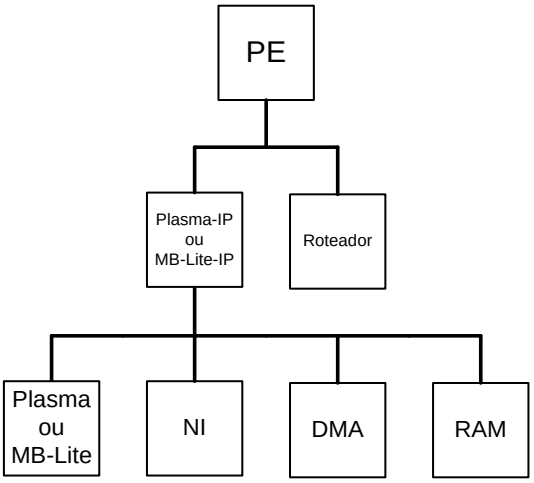


Figura 47 – Hierarquia de módulos e nomenclatura para o PE do MPSoC.

6.7 Comparação Plasma-IP - MB-Lite-IP

Esta Seção de conclusão deste Capítulo tem por objetivo identificar as características de cada um dos processadores, bem como das implementações e compará-los. A Tabela 8 apresenta um comparativo entre estes.

Entre as características de cada um dos processadores, pode-se citar a unidade de multiplicação/divisão do Plasma, que não existe no MB-Lite. Por outro lado o MB-Lite apresenta uma arquitetura Harvard, com barramentos dedicados para a memória instruções e para a memória de dados, diferentemente do Plasma.

Por utilizar as duas interfaces da memória RAM, o processador MB-Lite compartilha o barramento de dados com o módulo de DMA. Esta abordagem causa perda de desempenho se comparado com o Plasma, mas um ganho de desempenho se comparado com uma implementação em software.

Em termos de implementações do elemento de processamento, o Plasma apresenta modificações internas do hardware do processador, que atrapalham a portabilidade do processador. Entre estas modificações pode-se citar a adição da instrução `syscall`, do registrador `PAGE` (juntamente com a instrução `mtc0 reg,$10`). Em termos de software, o Plasma apresenta duas rotinas para salvamento de contexto, uma quando acontece uma interrupção de software e outra quando de uma chamada de sistema. A implementação do elemento de processamento com MB-Lite por sua vez não modifica o hardware interno do processador, visto que o registrador `PAGE` é um registrador mapeado em memória, e a chamada de sistema foi implementada como uma interrupção de hardware.

Tabela 8 – Comparativo entre as características dos processadores Plasma e MB-Lite e de suas implementações.

Feature	**Plasma**	**MB-Lite**
Syscall	Interrupção de software	Interrupção de hardware
Multiplicacão	Hardware	Software
Paginação	Registrador interno ao processador	Registrador externo ao processador
Acesso do DMA à memória	Direto	Compartilhado com o barramento da memória de dados do processador
Salvamento de contexto	Duas rotinas: uma para chamadas de sistema e outra para interrupções	Apenas uma
Modificações de hardware internas ao processador	Adição de syscall, registrador Page	Nenhuma
Inicialização da área de dados	Executada por uma ferramenta específica em tempo de compilação	O compilador mb-gcc cria rotinas de inicialização da área de dados em tempo de execução
Mapeamento de tarefas	Dinâmica	Manual
Geração do MPSoC	Automática, pela ferramenta *HeMPS-S generator*	Semi-automática

Destaca-se como trabalhos futuros a necessidade de automatização de alguns processos, como o mapeamento dinâmico de tarefas e a geração automática do MPSoC heterogêneo.

7 AVALIAÇÃO DA ARQUITETURA HEMPS-S HETEROGÊNEA

Este Capítulo apresenta os resultados da implementação da arquitetura HeMPS-S heterogênea. São apresentados dois cenários de execução: o primeiro onde se apresenta uma execução multitarefa nas arquiteturas MB-Lite e Plasma, e outro com a execução em um MPSoC heterogêneo com diversas organizações de heterogeneidade mostrando comunicação entre o Plasma e o MB-Lite. Por fim, um comparativo de ocupação de área e freqüência de operação é apresentado.

7.1 Execução Multitarefa

Este cenário tem por objetivo validar a execução multitarefa no processador MB-Lite, utilzando um MPSoC de tamanho 3x3. Os processadores 22 (escravo) e 10 (mestre) têm como núcleo de processamento o processador Plasma e o restante destes o processador MB-Lite (Figura 49). Uma aplicação simples, composta por duas tarefas (Figura 48), executa em uma instância de cada tipo de processador. Nesta aplicação a tarefa A envia uma mensagem M1 à tarefa B, que ao recebê-la responde com uma mensagem M2.

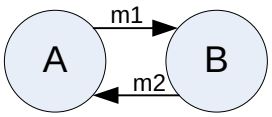


Figura 48 – Aplicação de teste para execução multitarefa.

Uma instância da aplicação é mapeada no processador MB-Lite (02), e outra instância da aplicação no processador Plasma (22), como ilustra a Figura 49. Este cenário permite validar as principais funcionalidades do *µkernel*, como: chaveamento de contexto, troca de página e troca de mensagem (apenas local).

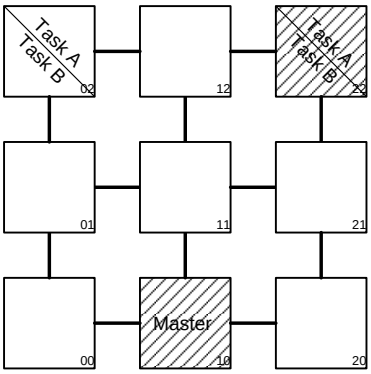


Figura 49 – Mapeamento para execução multitarefa. Os PEs hachurados contém como núcleo o processador Plasma.

A Figura 50 apresenta o código para a tarefa A. Note que existem duas chamadas da função `GetTick()`, uma antes do envio e outra depois do recebimento das mensagens. Cada tarefa executa 10 repetições (`NUM_MSG=10`) de envio de 10 pacotes (`SIZE_MSG=10`).

Assim, cada tarefa envia 100 pacotes, totalizando 200 pacotes trocados entre as tarefas em cada instância da aplicação.

```
int main(){
    int i,j,k;
    int time;

    msg1.length = SIZE_PKT;

    for(i=0;i<NUM_MSG;i++){
        for(j=0;j<SIZE_MSG;j++){
            time = GetTick();
            for(k=0;k<SIZE_PKT;k++){
                msg1.msg[k] = i+0x10;
            }
            Send(&msg1,TASKB);
            Receive(&msg2,TASKB);
            time = GetTick() - time;
            Echo(itoa(time));
        }
    }
```

Figura 50 – Código da tarefa A. Antes do envio e depois do recebimento das mensagens é executada uma chamada da rotina `GetTick()`.

A Figura 51 apresenta o número de ciclos de relógio entre a execução de cada envio e o recebimento de pacotes pelas tarefas. O processador MB-Lite apresenta 12% a mais no número de ciclos de relógio se comparado com a execução do Plasma. Este número se deve aos fatores anteriormente citados, tais como: multiplicação por software no processador MB-Lite; compartilhamento do barramento de acesso à memória de dados. Apesar deste desempenho, todas as funcionalidades o *µkernel* foram validadas, permitindo assim a validadação de cenários mais complexos.

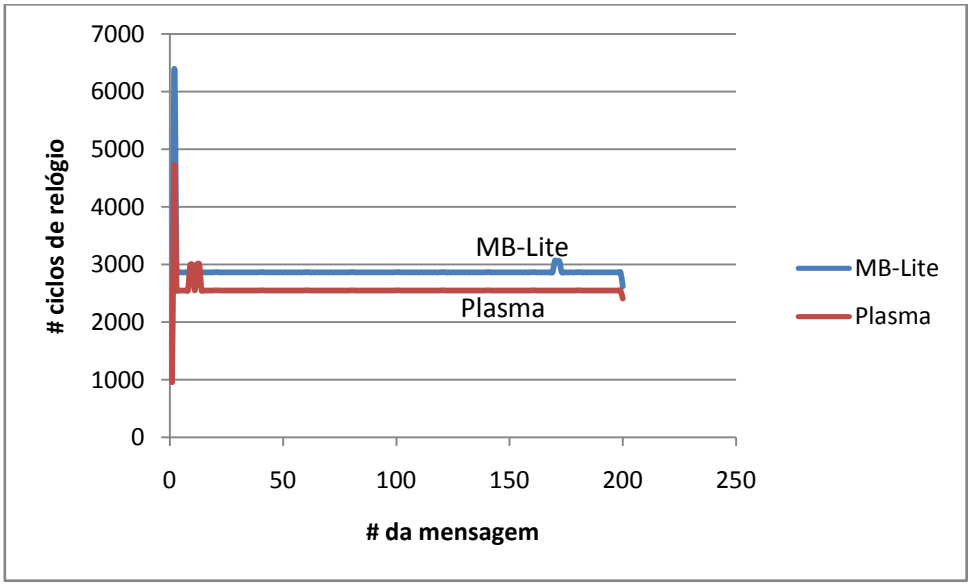


Figura 51 – Comparação do número de ciclos de relógio para envio e recebimento de pacotes durante a execução multitarefa.

Para efeitos comparativos, o binário para o Plasma contém 181 palavras, enquanto que o binário do MB-Lite contém 688 palavras. Isto mostra o impacto das rotinas de inicialização das variáveis da área de dados no código executável do MB-Lite.

7.2 Execução Heterogênea

Esta execução tem por objetivo demonstrar a funcionalidade da plataforma heterogênea, com comunicação entre os processadores independente da arquitetura destes. Desta forma, utilizou-se uma aplicação sintética, cujo grafo é apresentado na Figura 52. As constantes N e M da Figura são configuradas em tempo de compilação e representam o número de mensagens que serão enviadas. A tarefa A, por exemplo, executa 100 rodadas de envio de 10 (N) mensagens de 16 palavras (cada palavra contém 32 bits). A tarefa B, por sua vez, envia 100 rodadas de 20 (M) mensagens de 16 palavras.

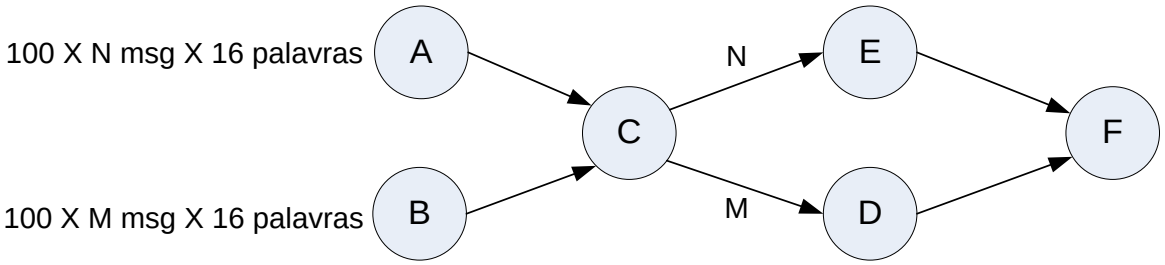


Figura 52 – Grafo de tarefas para à execução heterogênea.

Para complexificar a execução, utilizou-se o mapeamento das tarefas de acordo com a Figura 53. Este mapeamento propõe a divisão dos recursos com execução multitarefa (processadores 02 e 11), e distribuída, utilizando 4 processadores para a aplicação. Este cenário também apresenta uma concorrência pelos canais de comunicação, visto que mensagens de depuração são enviadas ao processador mestre (as mensagens de depuração dos processadores B, C, D, E e F irão trafegar entre os processadores 11 e 10). Por estes motivos, o mapeamento proposto pode ser considerado ruim, de forma que esta execução valida, além das funcionalidades da Sessão anterior, a comunicação entre os processadores.

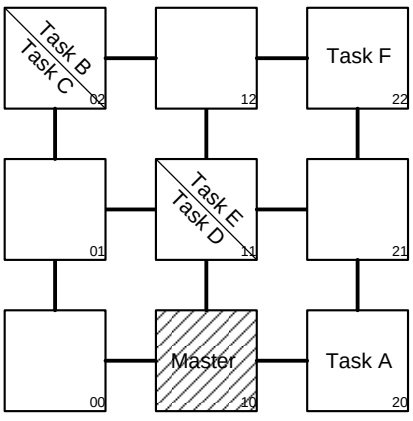


Figura 53 – Mapeamento para o cenário heterogêneo.

Para demonstrar a funcionalidade do sistema, independentemente da arquitetura do processador, foram propostos 8 cenários de teste, com cada um dos processadores configurado como apresentado na Tabela 9. Vale salientar que um mesmo binário não é

equivalente aos dois processadores. Por este motivo, as tarefas foram compiladas separadamente para os dois processadores, utilizando-se o mesmo código fonte. Portanto, no nível de software o repositório varia entre os cenários. Já em hardware o que varia é a configuração dos tipos de PE (MB-Lite ou Plasma).

Tabela 9 – Cenários de teste com configurações heterogêneas.

Cenário	Processador 02	Processador 11	Processador 20	Processador 22
1	Plasma	Plasma	Plasma	Plasma
2	MB-Lite	MB-Lite	MB-Lite	MB-Lite
3	MB-Lite	Plasma	Plasma	Plasma
4	Plasma	MB-Lite	Plasma	Plasma
5	Plasma	Plasma	MB-Lite	MB-Lite
6	MB-Lite	MB-Lite	Plasma	Plasma
7	MB-Lite	Plasma	MB-Lite	Plasma
8	Plasma	MB-Lite	Plasma	MB-Lite

Para mensurar as diferenças de implementação entre os PEs, o tempo de execução de cada rodada da tarefa C foi medido através da rotina `GetTick()`. A escolha da tarefa C se deve ao fato de que esta se comunica com o maior número de tarefas neste exemplo, executando tanto chamadas de `Send()` como `Receive()`. Semelhantemente à execução anterior, os resultados com o processador Plasma apresentaram um número menor de ciclos de relógio para execução da tarefa C, como apresenta a Figura 54. A execução do cenário 4, obteve os melhores resultados, levando em média 80 mil ciclos de relógio para execução de uma rodada.

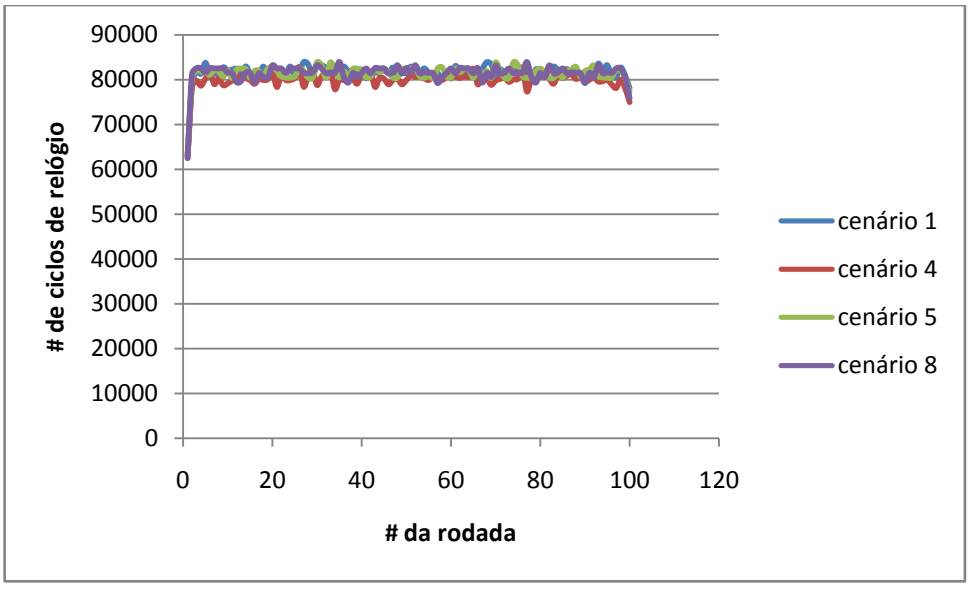


Figura 54 – Número de ciclos de relógio para execução da tarefa C no PE Plasma.

Os cenários com PEs do tipo MB-Lite (Figura 55) apresentaram mais ciclos de relógio para a execução, se comparados com o PEs do tipo Plasma. No pior caso (cenário 2), a diferença foi de 10% no número de ciclos de relógio. Este cenário, com o MPSoC homogêneo com PEs do tipo MB-Lite, apresentou em média 89 mil ciclos de relógio para execução de uma rodada.

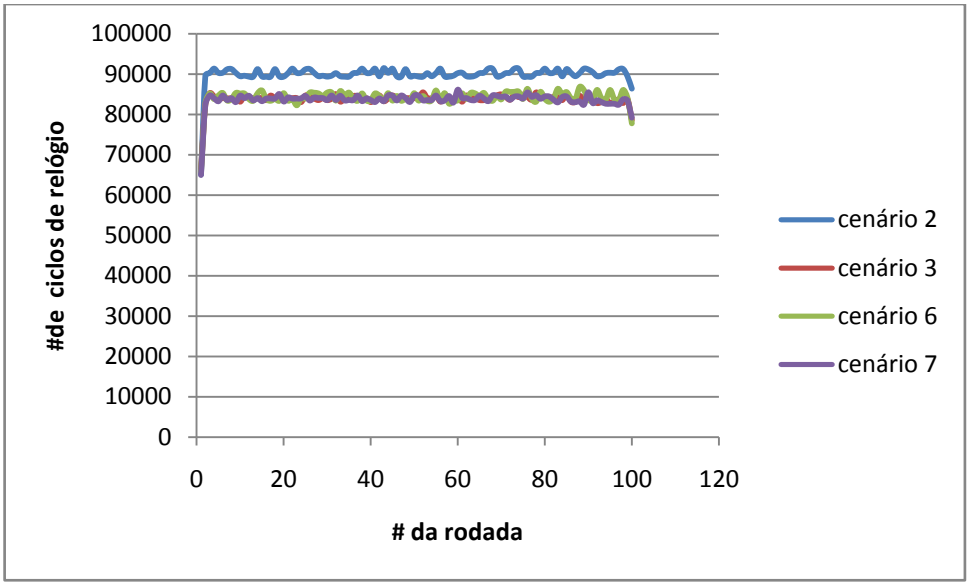


Figura 55 – Número de ciclos de relógio para execução da tarefa C no PE MB-Lite.

A execução destes 8 cenários de teste permitiu avaliar de forma extensa o µkernel. Todos os 8 cenários concluíram a execução de forma correta. Ressalta-se que apesar dos cenários com o processador MB-Lite apresentarem resultados inferiores, o objetivo deste trabalho é o porte tanto do hardware como do software do MPSoC HeMPS para uma nova arquitetura, apenas utilizando o MB-Lite como estudo de caso.

7.3 Comparativo dos Resultados entre PE Plasma x PE MB-Lite

Os resultados apresentados nesta Seção se referem à implementação do elemento de processamento do MPSoC HeMPS heterogêneo. Este elemento de processamento é composto pelo núcleo de processamento (Plasma-IP ou MB-Lite-IP, NI e DMA). A memória RAM por utilizar apenas módulos BRAMs foi excluída dos resultados. Os resultados referentes ao Plasma, apresentados anteriormente, foram dispostos novamente nesta Seção para facilitar a comparação entre com o MB-Lite. A Figura 56 apresenta os resultados de ocupação do PE contendo o MB-Lite e o Plasma.

O que se pode notar é que as duas arquiteturas ocupam praticamente a mesma área em termos de flip-flops. Entretanto em termos de LUTs, o MB-Lite ocupa aproximadamente 50% a mais. Por outro lado em termos de freqüência de operação o MB-Lite apresenta um ganho de 70%. Isto se explica pelo fato de o MB-Lite apresentar 5 estágios de pipeline, diferentemente do Plasma que apresenta apenas 3 estágios.

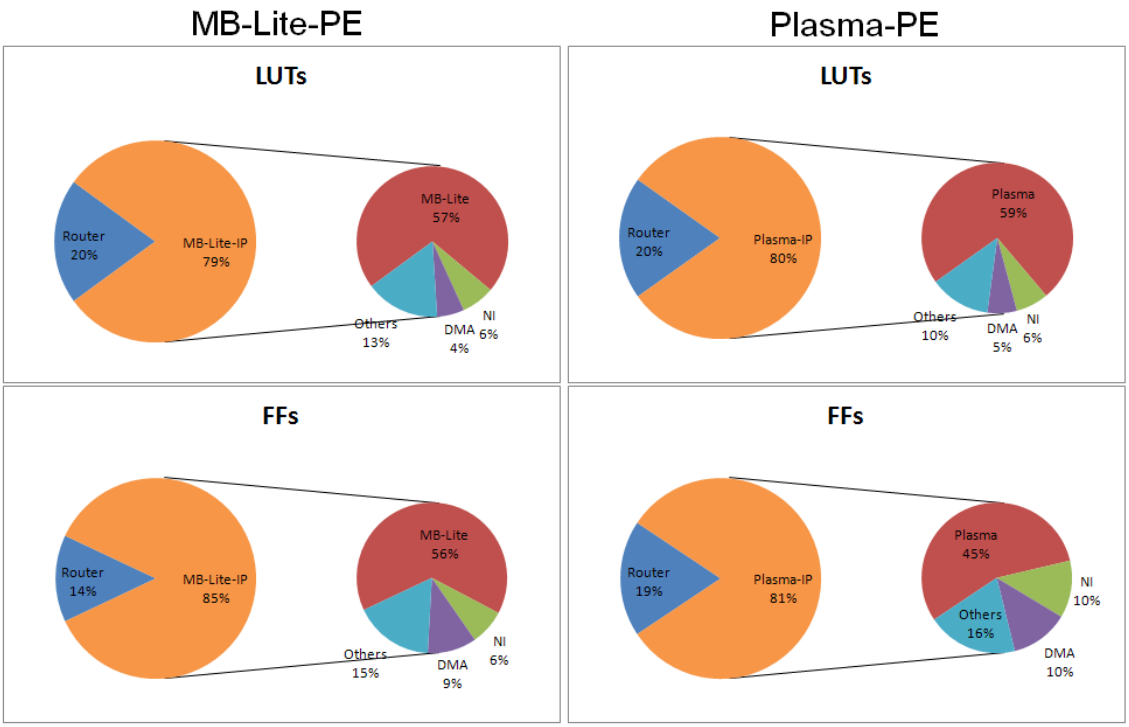


Figura 56 – Resultados de ocupação para os módulos internos do MB-Lite-PE e Plasma-PE para o dispositivo 5vlx330tff1738-2.

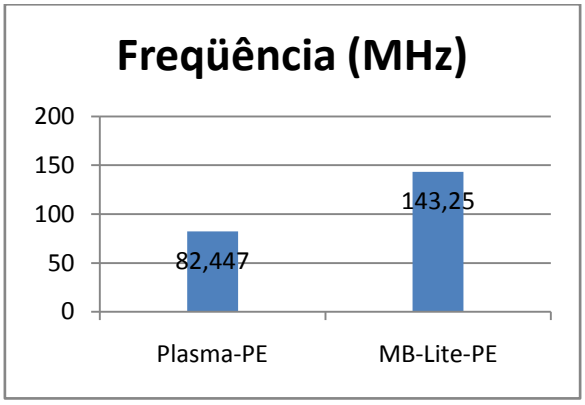


Figura 57 – Resultados de freqüência para Plasma-PE e MB-Lite-PE o dispositivo 5vlx330tff1738-2.

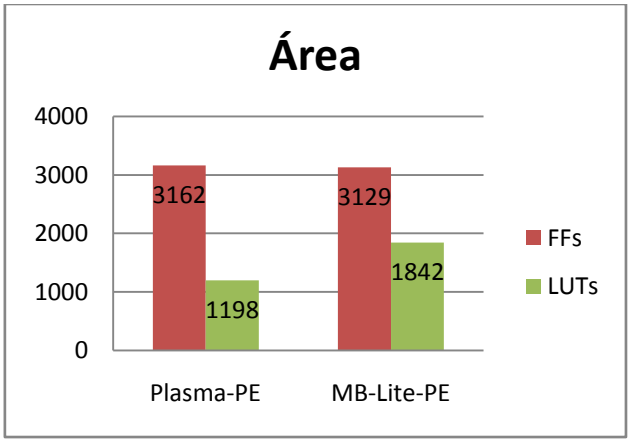


Figura 58 – Resultados de Ocupação de área para o Plasma-PE e MB-Lite-PE o dispositivo 5vlx330tff1738-2.

Desta forma, os resultados das execuções do MB-Lite devem ser reavaliados. Enquanto que número de ciclos de relógio das execuções para o MB-Lite são maiores em 10% na média, a freqüência de operação deste é 70% maior se comparado com o Plasma. Assim, o tempo de execução para uma mesma tarefa executando no MB-Lite será menor se comparado com o Plasma.

Como exemplo, utilizaremos a média do número de ciclos de relógio das rodadas do cenário 4 e 2 da execução heterogênea (o melhor e o pior resultado em termos de ciclos de relógio, respectivamente). Para cada um destes cenários, utilizaremos os valores de freqüência de operação dos processadores Plasma e MB-Lite. Desta maneira pode-se calcular o tempo de execução da mesma tarefa para os dois processadores, como apresentado na Tabela 10.

Tabela 10 – Tempo de execução para os cenários 4 e 2.

	Cenário 4 – Plasma	**Cenário 2 – MB-Lite**
Freqüência de Operação	82,447 MHz	143,25 MHZ
Período	12,129 ns	6,98 ns
Média de # de ciclos de relógio das rodadas	79865	89878
Tempo de Execução	**0,968 ms**	**0,627 ms**

A Tabela 10 mostra que apesar de obter um número de ciclos menor para execução, o Plasma, por operar em uma freqüência menor, obtém um tempo de execução maior se comparado com o MB-Lite. O Processador MB-Lite apresenta um tempo de execução 35% menor se comparado com o Plasma.

8 CONCLUSÃO E TRABALHOS FUTUROS

Esta Dissertação teve dois principais objetivos: (i) a prototipação em FPGA da plataforma HeMPS-S e (ii) o porte da arquitetura de software e hardware do PE do MPSoC HeMPS para uma nova arquitetura de processador.

Como ponto de partida para este trabalho, temos a plataforma HeMPS-S. Esta plataforma foi modificada afim de ser prototipada em FPGA como apresentado no Capítulo 4. Dentre as modificações, a principal situa-se na reestruturação da arquitetura, de forma a unir o módulo Plasma-IP e o roteador, facilitando o processo de síntese em hardware. O processo de prototipação da plataforma HeMPS-S é a primeira contribuição desta Dissertação.

Depois da plataforma prototipada, apresentou-se a integração de um novo processador à plataforma HeMPS-S: o processador MB-Lite. A integração de novos processadores envolve modificações de software (porte do kernel) e de hardware (adaptação do elemento de processamento). Estas modificações foram discutidas durante os Capítulos 5 e 6 desta dissertação. O importante deste processo é o fato de que nenhum dos módulos do PE foram modificados (NI, DMA, MB-Lite) e que o processo de porte do µkernel foi desenvolvido pensando-se no porte para uma arquitetura genérica de processador. Por estes motivos, o trabalho de integração de uma nova arquitetura de processador ao MPSoC é facilitado. Neste processo situa-se a segunda contribuição deste trabalho.

No final do Capítulo 6 uma comparação entre as vantagens e desvantagens de cada arquitetura é apresentada. Modificações internas ao processador, como na implementação do Plasma-IP, foram excluídas. Desta forma o porte para um processador genérico não requer conhecimento e validação da arquitetura interna do processador.

O Capítulo 7 apresenta execuções com o MPSoC heterogêneo. O processador Plasma é 10% mais rápido que o MB-Lite em termos de número de ciclos de relógio executados por uma mesma tarefa. Em termos de área ocupada em FPGA os dois são equivalentes. Já em termos de freqüência de operação, o MB-Lite apresenta uma freqüência 70% maior que o Plasma. Em um dos cenários de teste propostos, o processador MB-Lite apresentou um tempo de execução 35% menor que o Plasma, levando-se em conta o pior caso do MB-Lite comparado com o melhor caso do Plasma.

A contribuição mais relevante deste trabalho é que a plataforma heterogênea está validada e operacional. Ou seja, os dois processadores executam o mesmo *µkernel* multitarefa com suporte a comunicação entre tarefas.

De forma a facilitar o porte do µkernel para novos processadores, a Tabela 11identifica as funções do µkernel que exigiram modificações para o porte ao processador MB-Lite.

Tabela 11 – Diferenças entre as funções do µkernel do MB-Lite e Plasma.

Função	Diferenças
GetFreeSlot()	Igual
`DMA_Send()`	Para MB-Lite ou outro processador Harvard, adiciona-se um laço de espera no final
`InsertMessageRequest()`	Igual
`SearchRequestingMessage()`	Igual
`SearchTCB()`	Igual
`GetSlotFromPipe()`	Igual
`Syscall()`	Igual
`Scheduler()`	Igual
`Handler_NI()`	Igual
`OS_Init()`	Diferente
`ASM_RunScheduledTask()`	Plasma está em assembly, MB-Lite em C com Pragmas
`OS_InterruptServiceRoutine()`	Diferente, com passagem de parâmetros, devido à chamada de sistema
`OS_InterruptMaskClear()`	Igual
`OS_InterruptMaskSet()`	Igual
`OS_Idle()`	Igual
`main()`	Diferente

8.1 Trabalhos Futuros

Como trabalhos futuros podem-se enumerar as seguintes atividades:

- Excluir o código de inicialização gerada pelo compilador do MB-Lite: A inicialização da área de dados pode ser feita em tempo de compilação. A parte de dados pode ser inicializada de forma semelhante àquela do processador Plasma. Esta atividade pode ser realizada de duas formas: ou o desenvolvimento de uma ferramenta para exclusão das rotinas de inicialização ou a utilização de alguma ferramenta presente na *tool chain* Microblaze que exclua estas.
- Portar o µkernel para outras arquiteturas: Da forma como foi desenvolvido o elemento de processamento (conforme Anexo I), o processador pode ser tratado como um IP genérico. Serão necessárias modificações nas leituras/escritas dos registradores mapeados em memória, pois o barramento pode variar de acordo com o processador.
- Automatizar a geração de repositórios heterogêneos: Seriam necessárias modificações ao software *HeMPS-S Generator* de forma que a compilação, a escolha do tipo de nodo de processamento e a geração do binário fosse automatizada.

- Prototipar a arquitetura HeMPS heterogênea em FPGA: Vale salientar que tanto o processador MB-Lite quanto seu elemento de processamento estão descritos em VHDL no nível RTL, facilitando este processo.
- Adicionar um método que permita o mapeamento dinâmico de tarefas no MPSoC heterogêneo. Para isto, o mestre deve conhecer qual o tipo de processador em todos os endereços de rede e o tipo de processador na qual a tarefa mapeada pode executar. Uma abordagem simples seria adicionar no cabeçalho de cada tarefa do repositório um identificador com o tipo de tarefa ou o tipo de processador. Já no MPSoC, os processadores poderiam enviar uma mensagem identificando seu tipo no início da execução do *µkernel*.

REFERÊNCIAS

[AES10] Aeste Works. Capturado em: http://www.aeste.my, Jul. 2010.

[BEL06] Bellens, P.; Perez, J. M.; Badia, R. M.; Labarta, J. "*CellSS: a Programming Model for the Cell BE Architecture*". In: ACM/IEEE Supercomputing Conference, 2006, 11p.

[BUT07] Buttari, A.; Luszczek, P.; Kurzak J.; Dongarra, J.; Bosilca, G. "A Rough Guide to Scientific Computing On the PlayStation 3", Technical Report UT-CS-07-595, University of Tennessee Knoxville, 2007.

[CAR09a] Carara, E. A.; Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. *"HeMPS - a framework for NoC-based MPSoC Generation"*. In: ISCAS, 2009, pp. 1345-1348.

[CAR09b] Carvalho, E. L. S. "*Mapeamento Dinâmico de Tarefas em MPSoCs Heterogêneos baseados em NoC*". Tese de Doutorado, Programa de Pós-Graduação em Ciências da Computação, PUCRS, 2009, 168p.

[DEN09] Densmore, D. ; Simalatsar, A.; Davare, A.; Passerone, R.; Sangiovanni-Vincentelli, A. *"UMTS MPSoC Design Evaluation Using a System Level Design Framework"*. In: DATE, 2009, pp. 478-483.

[ESA11] European Space Agency – LEON2-FT. Capturado em: http://www.esa.int/TEC/Microelectronics/SEMUD70CYTE_0.html Jan, 2011.

[GAR09] Garzia, F.; Airoldi, R.; Ahonen, T.; Nurmi, J.; Milojevic, D. *"Implementation of the W-CDMA Cell search on a MPSoC Designed for Software Defined Radios"*. In: SiPS, 2009, pp. 30-35.

[GUI08] Guindani, G.; Reinbrecht, C.; Raupp, T.; Calazans, N.; Moraes, F.G. *"NoC Power Estimation at the RTL Abstraction Level"*. In: ISVLSI, 2008, pp. 475-478.

[HAF09] Hafer, C.; Griffith, S.; Guertin, S.; Nagy, J.; Sievert, F.; Gaisler, J.; Habinc, S. *"LEON 3FT Processor Radiation Effects Data"*, In: Radiation Effects Data Workshop, 2009, pp.148-151.

[HAM09] Hammami, O.; Li, X.; Larzul, L.; Burgun, L. *"Automatic Design Methodologies for MPSOC and Prototyping on Multi-FPGA Platforms"*. In: ISOCC, 2009, pp. 141-146.

[JAL10] Jalier, C.; Lattard, D.; Jerraya, A. A.; Sassatelli, G.; Benoit, P. and Torres, L. *"Heterogeneous vs Homogeneous MPSoC Approaches for a Mobile LTE Modem"*. In: DATE, 2010, 6p.

[JER05a] Jerraya, A. A.; Wolf, W. *"Multiprocessor Systems-on-Chips"*. Morgan Kaufmann, 2005, 602p.

[JER05b] Jerraya, A. A.; Tenhunen, H.; Wolf, W. "Multiprocessor Systems-on-Chips". *IEEE Computer*, vol 38-7, Julho 2005, pp. 36-40.

[JOV08] Joven, J.; Carrabina, J.; Font-Bach, O.; Castells-Rufas, D.; Martinez, R.; Teres, L. *"xENOC – An eXperimental Network-on-Chip Enviroment for Parallel Distributed Computing on NoC-based MPSoC Archtectures"*. In: Euromicro, 2008, pp. 141-148.

[KIS06] Kistler, M.; Perrone, M.; Petrini, F. "Cell Multiprocessor Communication Network: Built for speed". *IEEE Micro*, vol. 26-3, Mai-Jun 2006, pp. 10–23.

[KRA10] Kranenburg, T.; van Leuken, R. *"MB-LITE: A Robust, Light-Weight Soft-Core Implementation of the MicroBlaze Architecture".* In: DATE, 2010, pp. 997-1000.

[KYL03] Kylliainen, J.; Nurmi, J.; Kuulusa, M. *"COFFEE - a Core for Free"*. In: International Symposium on System-on-Chip, 2003, pp. 17-22.

[LIM09] Limberg T.; et. al. *"A Heterogeneous MPSoC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio"*. In: DAC University Booth, 2009. 6p.

[MON08] Monchiero, M.; Palermo G.; Silvano, C.; Villa O. "*A Modular Approach to Model Heterogeneous MPSoC at Cycle Level*". In: EUROMICRO, 2008, pp. 158-164.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". *Integration, the VLSI Journal*, Vol. 38(1), 2004, pp. 69-93.

[NIK08] Nikolov, H.; Thompson, M.; Stefanov, T.; Pimentel, A.; Polstra, S.; Bose, R.; Zissulescu, C.; Deprettere, E.; *"Daedalus: Toward Composable Multimedia MP-SoC Design"*. In: DAC, 2008, pp. 574-579.

[OPE10] OpenCores, *"Plasma-most MIPS I (TM) opcodes: overview"* capturado em: http://opencores.org/project,plasma, Jul. 2010.

[REI09] Reinbrecht, C. R.; Scartezzini, G.; Da Rosa, T. R. *"Desenvolvimento de um Ambiente de Execução de Aplicações Embarcadas para a Plataforma Multiprocessada HeMPS"*. Trabalho de Conclusão de Curso, Engenharia de Computação, PUCRS, 2009, 109p.

[ROD09] Rodolfo, T.A.; Calazans, N.L.V.; Moraes, F.G. *"Floating Point Hardware for Embedded Processors in FPGAs: Design Space Exploration for Performance and Area"*. In: ReConFig, 2009, pp. 24-29.

[SAI07] Saint-Jean, N.; Sassatelli, G.; Benoit, P.; Torres, L.; Robert, M. *"HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for Embedded Systems"*. In: ISVLSI, 2007, pp. 21-28.

[SHE09] Shen, H.; Pétrot, F. *"Novel Task Migration Framework on Configurable Heterogeneous MPSoC Platforms"*. In: ASP-DAC, 2009, pp. 733-738.

[TIL09] Tilera Corporation. *"TILE64™ Processor"*. Product Brief Description. EUA, Agosto, 2007, 2p.

[VAN07] Vangal, S.; Howard, J.; Ruhl, G.; Dighe, S.; Wilson, H.; Tschanz, J.; Finan, D.; Iyer, P.; Singh, A.; Jacob, T.; Jain, S.; Venkataraman, S.; Hoskote, Y.; Borkar, N. *"An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS"*. In: ISSCC, 2007, pp. 5-7.

[WOS07] Woszezenki, C. *"Alocação de Tarefas e Comunicação entre Tarefas em MPSoCs"*. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2007, 121p.

[ZHA07] Zhang, W; Geng, L.; Zhang, D.; Du, G.; Gao, M; Zhang W.; Hou, N.; Tang Y. *"Design of Heterogeneous MPSoC on FPGA"*. In: ASICON, 2007, pp. 102-105.

[OPE11a] Opencores, "OpenRisc 1200: Overview" capturado em: http://opencores.org/project,or1200_hp, Mar. 2011

[OPE11b] OpenFire Processor Core : Overview" capturado em: http://opencores.org/project,openfire_core, Mar. 2011

ANEXO I – DESCRIÇÃO DO MB-LITE-IP

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.std_logic_unsigned.ALL;
4
5   LIBRARY mblite;
6   USE mblite.config_Pkg.ALL;
7   USE mblite.core_Pkg.ALL;
8   USE mblite.std_Pkg.ALL;
9
10  use work.HermesPackage.all;
11
12  ENTITY mblite_soc IS
13      GENERIC(
14          address_router  : regmetadeflit:= (others=>'0')
15      );
16      PORT(
17          sys_clk_i   : in STD_LOGIC;
18          sys_rst_i   : in STD_LOGIC;
19          -- NoC Interface
20          clock_tx    : out std_logic;
21          tx          : out std_logic;
22          data_out    : out regflit;
23          credit_i    : in  std_logic;
24          clock_rx    : in  std_logic;
25          rx          : in  std_logic;
26          data_in     : in  regflit;
27          credit_o    : out std_logic
28      );
29  END mblite_soc;
30
31  ARCHITECTURE arch OF mblite_soc IS
32
33      procedure set_register(signal value_set: out std_logic_vector; signal dmem_o : in dmem_out_type;
34  constant REG : in std_ulogic_vector(31 downto 0)) is
35      begin
36          if dmem_o.adr_o = REG and dmem_o.we_o = '1' then
37                  value_set <= std_logic_vector(dmem_o.dat_o);
38          end if;
39      end set_register;
40
41      TYPE data_ram_type IS RECORD
42          addr    : std_logic_vector(31 downto 2);
43          en      : std_logic;
44          we      : std_logic_vector(3 downto 0);
45          dataw   : std_logic_vector(31 downto 0);
46          datar   : std_ulogic_vector(31 downto 0);
47      END RECORD;
48
49        ---- SIGNAL AND CONSTANT DECLARATION OMITEED
50
51  BEGIN
52
53      mem_enable      <=  '1' when sys_rst_i = '0' AND dmem_o.ena_o = '1' AND dmem_o.adr_o(31 downto
54  28) /= x"2" else '0';
55      sel_o           <= dmem_o.sel_o WHEN dmem_o.we_o = '1' ELSE (OTHERS => '0');
56      dmem_i.ena_i    <= '1';
57
58      --DMA/data bus mux
59      dram.addr       <= dma_mem_address(31 downto 2) when dma_active_sig = '1' else dmem_adr;
60      dram.en         <= dma_enable_internal_ram when dma_active_sig = '1'   else mem_enable;
61      dram.we         <= dma_mem_write_byte_enable when dma_active_sig = '1' else
62  std_logic_vector(sel_o);
63      dram.dataw      <= dma_mem_data_write when dma_active_sig = '1'else
64  std_logic_vector(dmem_o.dat_o);
65      dmem_o_data     <= dram.datar;
66      mem_data_read   <= std_logic_vector(dram.datar);
67
68      ram: entity work.mblite_ram
69          port map(
70              clk             => sys_clk_i,
71              --data/dma memory bus
72              address_a       => dram.addr,
73              enable_a        => dram.en,
74              wbe_a           => dram.we,
75              data_write_a    => dram.dataw,
```

```
76              data_read_a     => dram.datar,
77
78              --instruction memory bus
79              address_b       => imem_adr,
80              enable_b        => imem_o.ena_o,
81              wbe_b           => "0000",
82              data_write_b    => "00000000000000000000000000000000",
83              data_read_b     => imem_i.dat_i
84          );
85
86      imem_adr <=     std_logic_vector(imem_o.adr_o(rom_size - 1 DOWNTO 2)) when imem_o.adr_o =
87  x"00000010" else
88                      std_logic_vector(imem_o.adr_o(rom_size - 1 DOWNTO 16)) & current_page_reg(15
89  downto 14) & std_logic_vector(imem_o.adr_o(13 DOWNTO 2)) when imem_o.adr_o(rom_size - 1 DOWNTO 16) =
90  "0000000000000000" else
91                      std_logic_vector(imem_o.adr_o(rom_size - 1 DOWNTO 2));
92
93      dmem_adr <=     std_logic_vector(dmem_o.adr_o(ram_size - 1 DOWNTO 2)) when imem_o.adr_o =
94  x"00000010" else
95                      std_logic_vector(dmem_o.adr_o(ram_size - 1 DOWNTO 16)) & current_page_reg(15
96  downto 14) & std_logic_vector(dmem_o.adr_o(13 DOWNTO 2)) when dmem_o.adr_o(ram_size - 1 DOWNTO 14) =
97  "00000000000000000000" else
98                      std_logic_vector(dmem_o.adr_o(ram_size - 1 DOWNTO 2));
99
100     imem_adr_32_debug <= std_logic_vector(imem_o.adr_o(rom_size - 1 downto 2));
101     page_debug <= current_page_reg(15 downto 14);
102
103     core0 : entity mblite.core PORT MAP
104     (
105         imem_o => imem_o,
106         dmem_o => dmem_o,
107         imem_i => imem_i,
108         dmem_i => dmem_i,
109         int_i  => irq,
110         rst_i  => sys_rst_i,
111         clk_i  => sys_clk_i
112     );
113
114     process(sys_clk_i)
115     begin
116         if sys_rst_i = '1' then
117             dmem_reg.adr_o  <= (others => '0');
118             dmem_reg.dat_o  <= (others => '0');
119             dmem_reg.ena_o  <= '0';
120             dmem_reg.we_o   <= '0';
121             current_page_reg<= (others => '0');
122             next_page_reg   <= (others => '0');
123             time_slice      <= (others => '0');
124             irq_mask_reg    <= (others => '0');
125             sys_call_reg    <= (others => '0');
126             tick_counter    <= (others => '0');
127             rtid <= '0';
128         elsif sys_clk_i'event and sys_clk_i='1' then
129             dmem_reg.adr_o  <= dmem_o.adr_o;
130             dmem_reg.dat_o  <= dmem_o.dat_o;
131             dmem_reg.ena_o  <= dmem_o.ena_o;
132             dmem_reg.we_o   <= dmem_o.we_o;
133
134             if imem_o.adr_o = x"00000010" then
135                 current_page_reg <= (others => '0');
136             elsif imem_i.dat_i = x"b62e0000" then
137                 current_page_reg <= next_page_reg;
138             end if;
139
140             if imem_i.dat_i = x"b62e0000" then
141                 rtid <= '1';
142             else
143                 rtid <= '0';
144             end if;
145
146             if imem_o.adr_o = x"00000010" then
147                 enter_int <= '1';
148             else
149                 enter_int <= '0';
150             end if;
151
152             if imem_o.adr_o = x"00000008" then
153                 enter_exception <= '1';
154             else
```

```
155                 enter_exception <= '0';
156             end if;
157
158             if imem_o.adr_o = x"00000260" then
159                 return_exception <= '1';
160             else
161                 return_exception <= '0';
162             end if;
163
164             if current_page_reg /= x"00" then
165                 time_slice <= time_slice + 1;
166             end if;
167
168             --set the irq_mask_reg
169             set_register(irq_mask_reg,dmem_o,IRQ_MASK);
170             --kernel sets the next page to return from a interrupt call in a task
171             set_register(next_page_reg, dmem_o, NEXT_PAGE);
172             -- reinitialize counter
173             set_register(time_slice,dmem_o,COUNTER);
174             -- sys_call register
175             set_register(sys_call_reg,dmem_o,SYS_CALL);
176             tick_counter <= tick_counter + 1;
177         end if;
178     end process;
179
180     --irq_status
181     irq_reg         <= "00000000000000000000000000" & sys_call_reg(0) & ni_intr & '0' &
182 time_slice(14) & "000";
183     irq             <= '1' when (irq_reg and irq_mask_reg) /= x"00" else '0';
184
185
186     dmem_i.dat_i <= std_ulogic_vector(irq_reg)             when dmem_reg.adr_o = IRQ_STATUS and
187 dmem_reg.ena_o = '1'  else
188                     ((0) => ni_read_av, others => '0')     when dmem_reg.adr_o = NI_STATUS_READ and
189 dmem_reg.ena_o = '1' else
190                     ((0) => ni_send_av, others => '0')     when dmem_reg.adr_o = NI_STATUS_SEND and
191 dmem_reg.ena_o = '1' else
192                     std_ulogic_vector(ni_data_read)        when dmem_reg.adr_o = NI_READ and
193 dmem_reg.ena_o = '1' else
194                     std_ulogic_vector(ni_config)           when dmem_reg.adr_o = NI_CONFIGURATION
195 and dmem_reg.ena_o = '1' else
196                     ((0) => dma_active_sig, others => '0') when dmem_reg.adr_o = DMA_ACTIVE and
197 dmem_reg.ena_o = '1' else
198                     std_ulogic_vector(irq_mask_reg)        when dmem_reg.adr_o = IRQ_MASK and
199 dmem_reg.ena_o = '1' else
200                     std_ulogic_vector(tick_counter)        when dmem_reg.adr_o = TICK_COUNTER_ADDR
201 and dmem_reg.ena_o = '1' else
202                     dmem_o_data;
203
204     ni: entity work.network_interface
205     generic map ( address_router => address_router)
206     port map(
207         clock       => sys_clk_i,
208         reset       => sys_rst_i,
209
210         -- NoC interface
211         clock_tx    => clock_tx,
212         tx      => tx,
213         data_out    => data_out,
214         credit_i    => credit_i,
215         clock_rx    => clock_rx,
216         rx      => rx,
217         data_in     => data_in,
218         credit_o    => credit_o,
219
220         -- CPU/DMA interface
221         hold        => plasma_hold,
222         send_av     => ni_send_av,
223         read_av     => ni_read_av,
224         intr        => ni_intr,
225         send_data   => ni_send_data,
226         read_data   => ni_read_data,
227         data_write  => ni_data_write,
228         data_read   => ni_data_read,
229         config      => ni_config
230     );
231
232
233
```

```
234         -- CPU -> NI
235         cpu_read_data       <= '1' when dmem_reg.adr_o = NI_READ and dmem_reg.ena_o = '1' else '0';
236         cpu_send_data       <= '1' when dmem_reg.adr_o = NI_WRITE and dmem_reg.we_o = '1' else '0';
237         cpu_packet_ack      <= '1' when dmem_reg.adr_o = NI_ACK and dmem_reg.we_o = '1' else '0';
238         cpu_packet_nack     <= '1' when dmem_reg.adr_o = NI_NACK and dmem_reg.we_o = '1' else '0';
239         cpu_packet_end      <= '1' when dmem_reg.adr_o = NI_END and dmem_reg.ena_o = '1' else '0';
240
241         -- NI inputs with DMA
242         ni_send_data            <= dma_send_data or cpu_send_data;
243         ni_read_data            <= dma_read_data or cpu_read_data;
244         MUX_NI: ni_data_write   <= dma_data_write when dma_send_data = '1' else
245     std_logic_vector(dmem_reg.dat_o);
246
247         dma: entity work.dma
248         port map(
249             clock           => sys_clk_i,
250             reset           => sys_rst_i,
251
252             -- NI interface
253             read_av         => ni_read_av,
254             read_data       => dma_read_data,
255             send_av         => ni_send_av,
256             send_data       => dma_send_data,
257
258             -- Configuration pins
259             set_address     => cpu_set_address,
260             set_size        => cpu_set_size,
261             set_op          => cpu_set_op,
262             start           => cpu_start,
263
264             -- Input/Output data ports
265             data_read       => dma_data_read,
266             data_write      => dma_data_write,
267             active          => dma_active_sig,
268
269             -- Interrupt management
270             intr            => dma_intr,
271             intr_ack        => cpu_ack,
272
273             -- Memory interface
274             mem_address     => dma_mem_address,
275             mem_data_write  => dma_mem_data_write,
276             mem_data_read   => dma_mem_data_read,
277             mem_byte_we     => dma_mem_write_byte_enable
278         );
279
280         -- CPU -> DMA
281         cpu_set_size        <= '1' when dmem_reg.adr_o = DMA_SIZE and dmem_reg.ena_o = '1' else '0';
282         cpu_set_address     <= '1' when dmem_reg.adr_o = DMA_ADDR and dmem_reg.ena_o = '1' else '0';
283         cpu_set_op          <= '1' when dmem_reg.adr_o = DMA_OP and dmem_reg.ena_o = '1' else '0';
284         cpu_start           <= '1' when dmem_reg.adr_o = START_DMA and dmem_reg.ena_o = '1' else '0';
285         cpu_ack             <= '1' when dmem_reg.adr_o = DMA_ACK and dmem_reg.ena_o = '1' else '0';
286
287         dma_enable_internal_ram     <= '1' when dma_mem_address(30 downto 28) = "000" else '0';
288
289         -- DMA inputs
290         MUX_DMA1: dma_data_read     <= std_logic_vector(dmem_reg.dat_o) when cpu_set_op = '1' or
291     cpu_set_size = '1' or cpu_set_address = '1' else ni_data_read;
292         MUX_DMA2: dma_mem_data_read <= mem_data_read when dma_enable_internal_ram = '1' else data_read;
293
294     END arch;
295
```