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

IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO DE UM MPSOC HOMOGÊNEO INTERCONECTADO POR NOC

ODAIR MOREIRA

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, Setembro de 2009

FICHA CATALOGRÁFICA

TERMO DE APRESENTAÇÃO

AGRADECIMENTOS

Quero começar meus agradecimentos com uma frase que meu orientador me disse e que eu vou levar para sempre, “Nunca terminamos. Todo o sucesso é um começo”.

Esse sucesso se deve a uma pessoa que teve grande participação no desenvolvimento deste trabalho. Minha esposa, que desde o inicio foi a responsável por eu ingressar neste mestrado, me apoiar nas horas mais difíceis e suportar meus “surtos” quando as coisas pareciam fugir ao controle. Poderia aqui ficar escrevendo linhas e linhas de agradecimento e eu não conseguiria expressar tudo o que devo a ela, mas ela com certeza sabe o quanto eu agradeço a ela por passar essa etapa ao meu lado.

Não quero aqui ser muito extenso nos meus agradecimentos, mas tenho obrigação de agradecer a todas as pessoas que tiveram uma relação direta ou indireta para o desenvolvimento deste trabalho.

Agradeço a todos os meus colegas de mestrado que me deram uma grande força na hora de cumprir todas as disciplinas do mestrado, pelas madrugadas que estudávamos para apresentar trabalhos no outro dia, ou os fins de sema estudando em POA para fazer as provas e mais trabalhos.

Agradeço a todos os professores da PUC que participaram do MINTER que com certeza contribuíram para este trabalho. Ao professor Ney Calazans, que com suas observações perfeccionistas, comentários e críticas que contribuíram para o desenvolvimento dessa dissertação e também para o meu desenvolvimento pessoal.

A UNEMAT por propiciar este MINTER e a FAPEMAT por contribuir com a bolsa para custeio das idas a Porto Alegre.

A família de minha esposa, que quando tinha que ir para Porto Alegre era na casa deles que eu ficava estudando, comendo e dormindo. Muito obrigado a vocês todos.

Claro que eu não poderia deixar de falar do meu orientador e também não posso deixar de comentar uma coisa interessante que aconteceu, pois o meu orientador inicialmente era para ser o Prof. Ney, mas por algum motivo tive que mudar de orientador e passei a ser orientado pelo Prof. Moraes. Acho que isso é daquelas coisas que a gente não consegue explicar. Uma das pessoas mais incríveis que eu conheci e que me ensinou muito durante esse tempo que convivemos. Agradeço a sua paciência que não tem dimensão, pelas “broncas” por e-mail, por sua dedicação e por suas palavras de incentivo e também a todo o conhecimento que adquiri durante este trabalho. Espero ter atendido as suas expectativas.

Agradeço a Deus por me dar forças quando eu mais precisava.

IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO DE UM MPSOC HOMOGÊNEO INTERCONECTADO POR NOC

RESUMO

O aumento do número de transistores em um único chip trouxe novos desafios de projeto, entre eles como aproveitar efetivamente este elevado número de componentes. Uma das formas encontradas é através do reuso de módulos de hardware. Quando estes módulos de hardware são processadores, temos multiprocessamento em chip, resultando nos MPSoCs (*Multiprocessor System on Chip*). Os MPSoCs estão se tornando elementos comuns em sistemas embarcados, devido ao alto desempenho e à flexibilidade que eles trazem ao projeto. Nos últimos anos universidades e empresas vem desenvolvendo grandes projetos em multiprocessamento. O presente trabalho tem por objetivo desenvolver um MPSoC homogêneo, com interconexão por NoC. A motivação para a adoção de NoCs reside no maior paralelismo e escalabidade desta infraestrutura de comunicação, quando comparado a barramentos. O desempenho do MPSoC desenvolvido é avaliado, demonstrando-se os benefícios em se utilizar este tipo de arquitetura no projeto de sistemas embarcados.

Palavras Chave: MPSoCs, multiprocessamento, NoC, multicore.

DESIGN AND PERFORMANCE EVALUATION OF A NoC-BASED HOMOGENEOUS MPSoC

ABSTRACT

The increase in the number of transistors on a single chip has brought new design challenges, among them, how to design new circuits using this available number of transistors. The reuse of Intellectual Property (IP) cores enables the design of such complex systems. The design of a system through the reuse of processors results in a MPSoCs (Multiprocessor System on Chip). MPSoCs are becoming increasingly popular in embedded systems for the high degree of performance and flexibility they permit. In recent years universities and companies have been developing large projects in multiprocessing. The goal of the work is to develop a homogenous MPSoC, using a NoC as the communication infrastructure. NoC is employed due to its higher degree of parallelism and scalability, when compared to busses. The performance of the implemented MPSoC is evaluated, demonstrating the benefits when this architecture is used to design embedded applications.

Keywords: MPSoC, multiprocessing, NoC, multicore.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CellBE	*Cell Broadband Engine Architecture*
CSIP	*Currently Serviced Input Port number*
DMA	*Direct Memory Access*
DEMAC	*Direct Memory Access Controller*
DSP	*Digital Signal Processor*
EIB	*Element Interconnect Bus*
FIFO	*First In First Out*
FPU	*Floating Point Unit*
GPS	*Global Positioning System*
HDTV	*High-definition television*
IP	*Intellectual Property*
IU	*Instruction Unit*
LUT	*Lookup Table*
MFC	*Memory Flow Controller*
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MMU	*Memory Management Unit*
MPSoC	*Multi Processor System on Chip*
NI	*Network Interface*
NoC	*Network on Chip*
NORMA	*No Remote Memory Access*
NUMA	*Non-Uniform Memory Access*
OCP	*Open Core Protocol*
PDA	*Personal Digital Assistant*
PE	*Processing Element*
PPE	*PowerPC Processing Element*
PXU	*PowerPC execution Unit*
PCB	*Printed Circuit Board*
QNoC	*Quality-of-Service NoC*
SIMD	*Single Instruction, Multiple Data*
SMT	*Simultaneous multithreading*
SPE	*Synergistic Processor Element*
SPU	*Synergistic Processor Unit*
STI	*Sony, Toshiba, IBM*
SoC	*System on Chip*
SoCIN	*System-on-Chip Interconnection Network*
TCB	*Task Control Block*
VLIW	*Very Long Instruction Word*
VSU	*Vector Scalar Unit*
XU	*Fixed Point Execution*

SUMÁRIO

1 INTRODUÇÃO

A indústria de Circuitos Integrados (CI) vem crescendo consideravelmente em função do aumento da demanda do mercado de consumo e, com isso, a evolução dos processadores também vem se acentuando. Quando falamos em demanda de consumo podemos dar inúmeros exemplos de produtos, tanto para consumidores diretos ou indiretos. Um dos mercados de maior crescimento é o de telecomunicações, que também é bastante amplo, envolvendo desde telefonia móvel, internet, HDTV até GPS. Outra área em expansão é a de segurança, não só residencial, mas também industrial, e temos produtos como rastreamento via satélite, aviões, automóveis, etc. Todos estes exemplos têm pelo menos um processador internamente. Quando são agregados mais características a este produto, a possibilidade de maior consumo, maior processamento e miniaturização são inerentes a sua característica. Por isso, tornar os CIs cada vez menores é um dos grandes desafios das indústrias de circuitos integrados.

Em 1965 o então Presidente da Intel predizia que o poder de processamento aumentaria 100% a cada 18 meses, recebendo o nome de “Lei de Moore”. Já em 1975 ele revisou sua predição e aumentou para 2 anos [MOO65]. A tecnologia que nos permite criar processadores cada vez mais rápidos, conforme predito na Lei de Moore necessita de novos métodos de concepção e projeto para resolver importantes desafios, como integração de bilhões de transistores em um único circuito integrado e dissipação de potência.

Em 2005 a *International Technology Roadmap for Semiconductors* já citava a existência de circuitos integrados com mais de 100 milhões de transistores, e que a previsão era que na próxima década circuitos integrados seriam fabricados com bilhões de transistores. Hoje já podemos constatar que em algumas indústrias isso já é possível. Temos como exemplo, o processador Itanium da Intel com mais de 2 bilhões de transistores. Atualmente a principal barreira que os projetistas estão encontrando para desenvolver CIs com bilhões de transistores é a questão das propriedades físicas, ou seja, a miniaturização de um transistor está chegando ao seu limite.

Os projetos de circuitos eletrônicos sofreram avanços, e se direcionam para uma tecnologia que possibilita podermos incluir em um mesmo circuito integrado, módulos analógicos e digitais. É comum encontrarmos ao menos um microprocessador e uma memória nos CIs atuais, além de processadores de sinais, filtros, amplificadores, etc. Esta quantidade de componentes no interior de um chip denomina-se SoC (*System-on-chip*). SoC é portanto, um CI complexo que integra os principais elementos funcionais de um produto completo [MAR01][JER05a].

A integração de vários componentes em SoCs traz avanços comparado com a integração em PCBs (*Printed Circuit Board*), que utiliza diversos circuitos integrados para executarem várias funções. Entre estes avanços podemos destacar:

- Maior velocidade de operação do sistema, uma vez que na integração intra-chip o fluxo de dados entre o processador e outros componentes também intra-chip podem ser maximizados;
- Potência consumida reduzida, devido às baixas tensões requeridas devido ao alto grau de

miniaturização;

- Tamanho e complexidade reduzidas dos produtos para os usuários finais, uma vez que o número de componentes adicionais cai drasticamente;
- Aumento da confiabilidade do sistema.

Enquanto um único processador pode ser suficiente para aplicações de baixo desempenho, um crescente número de aplicações necessita de mais processadores para atingir seus requisitos de desempenho. Por esta razão, é crescente o uso de MPSoCs para a criação de sistemas integrados complexos. Um MPSoC é mais do que um cluster dentro de um chip, pois requisitos de aplicação e restrições de implementação fazem com que os projetistas criem arquiteturas especializadas e heterogêneas [JER05b].

O desenvolvimento de *Sistemas Multiprocessados em Chip*, ou MPSoCs (*Multiprocessor System-on-chip*), é uma tendência no projeto de sistemas embarcados, pois estes sistemas permitem uma melhor utilização dos transistores disponíveis. Em 2005 estima-se que 75% dos projetos dos sistemas embarcados utilizavam pelo menos três processadores [TAN06].

1.1 MPSoCs

MPSoCs são arquiteturas que buscam um compromisso entre restrições da tecnologia VLSI e as necessidades da aplicação. MPSoCs representam uma evolução do conceito de SoC, e com isso se beneficiam das mesmas vantagens adquiridas no projeto de SoCs. Geralmente esse projeto é baseado no reuso de núcleos de hardware (ou simplesmente IPs), ou seja, módulos pré-projetados e pré-validados [GUP97][BER01]. A maioria das indústrias que utilizam sistemas *on-chip* hoje em dia, precisam se preocupar com o desempenho e o consumo de energia. Preocupados com isso essas indústrias visam não só a miniaturização de seus equipamentos, mas também como reduzir o esforço de projeto e consequentemente o *time-to-market.*

Adicionalmente, MPSoCs são tendência porque amenizam a crise de projeto VLSI, através da redução do espaço entre a disponibilidade de transistores por circuito integrado nas tecnologias atuais e a capacidade de projeto de SoCs [HEN03]. Por essas características já citadas, os sistemas multiprocessados em um único chip se tornam em um vasto campo para pesquisas.

A maioria dos sistemas embarcados (*e.g.* Celulares, PDAs, HDTVs, set-top box) hoje em dia já utilizam um sistema em único chip e todos possuem diversos tipos de chips, cada um com suas características específicas. Podemos dizer então que a maioria dos projetos de MPSoCs tendem a ser heterogêneos, podendo ter vários PEs (*Processing Elements*), memórias, etc. Todos esses elementos terão suas próprias características de frequência, potência e voltagem diferentes entre eles.

1.2 NoC – Network on Chip

A forma mais utilizada para interligar os núcleos de um hardware é através de arquiteturas por barramento. Podemos citar o *CoreConect* da IBM e o *WISHBONE* da Silicore. Este último foi adotado pela organização OpenCores como padrão de conectividade em seus núcleos de hardware.

Implementações por barramento podem ser "simples", quando se trata de implementação, porém apresentam algumas desvantagens [BEN02]:

i. Apenas uma troca de dados pode ser realizada por vez, pois o meio físico é compartilhado por todos os núcleos de hardware, reduzindo o desempenho global do sistema;

ii. Necessidade de mecanismos inteligentes de arbitragem do meio físico para evitar desperdício de largura de banda;

iii. A escalabilidade é limitada, ou seja, o número de núcleos de hardware que podem ser conectados ao barramento é muito baixo, tipicamente na ordem da dezena;

iv. O uso de linhas globais em um circuito integrado com tecnologia submicrônica impõe sérias restrições ao desempenho do sistema devido às altas capacitâncias e resistências parasitas inerentes aos longos fios.

Uma alternativa ao uso dos barramentos são as redes intra-chip, ou NoCs (*Network on-Chip*). Uma rede intra-chip possui as mesmas características de uma rede de computadores interconectados paralelamente e pode ser definida como um conjunto de roteadores e canais ponto-a-ponto que interconectam os núcleos de um sistema integrado de modo a suportar a comunicação entre esses núcleos. As principais características das NoCs que motivam o seu estudo são: (*i*) confiabilidade; (*ii*) eficiência no gerenciamento de energia; (*iii*) escalabilidade da largura de banda em relação a arquiteturas de barramento (*iv*) reusabilidade; (*v*) decisões de roteamento distribuídas [MOR04b].

O modelo de comunicação utilizado é o da troca de mensagens, sendo que a comunicação entre núcleos é feita através do envio e recebimento de mensagens de requisição e de resposta.

Cada pacote (Figura 1a) é constituído geralmente por: (*i*) cabeçalho (*header)*, o qual sinaliza o início do pacote e inclui informações necessárias à sua transferência pela rede; (*ii*) carga útil (*payload*), a qual inclui o conteúdo do pacote; e (*iii*) terminador (*trailer)*, o qual sinaliza o final do pacote e pode ser até a última palavra da carga útil, desde que haja um bit especial ativado apenas no final do pacote.

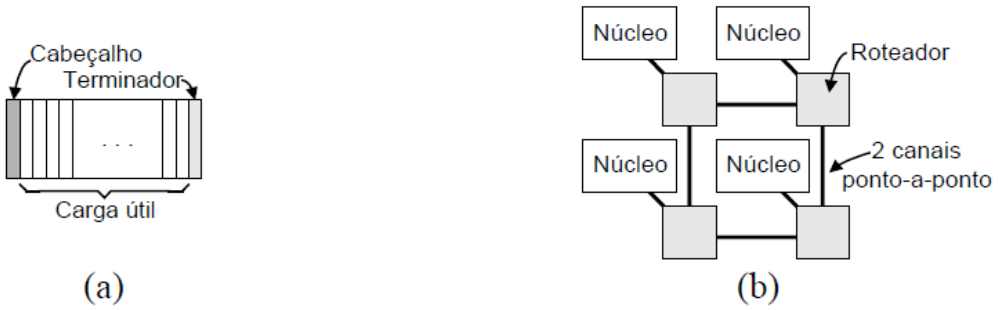

Figura 1 – Exemplo de formato de pacote e topologia de rede intra-chip. (a) formato de pacote. (b) topologia básica de rede intra-chip [ZEF03b].

Redes intra-chip podem ser caracterizadas como segue:

- Controle de fluxo: define a alocação de *buffers* e canais necessários para um pacote avançar pela rede, realizando a regulação de tráfego nos canais.
- Topologia: é o arranjo dos nodos e canais sob a forma de um grafo. Não está diretamente associada aos roteadores, mas sim como eles estão interligados. De acordo com [BAS06] [REG06] estão divididos em redes diretas e indiretas. A Figura 1b ilustra uma topologia direta, do tipo malha.
- Roteamento: define o caminho a ser realizado por um pacote para atingir o seu destino. Determina como um pacote escolhe um caminho deste grafo.
- Arbitragem: resolve conflitos internos na rede, quando duas ou mais mensagens competem por um mesmo recurso (*buffer* ou canal de saída). Determina qual canal de entrada pode utilizar um determinado canal de saída.
- Chaveamento: define como um pacote é transferido da entrada de um roteador para um de seus canais de saída. Os chaveamentos mais utilizados são por circuito e por pacotes [HWA93] [MOH98].
- Tratamento de contenção: determina o esquema de filas utilizado para armazenar um pacote bloqueado na rede quando um canal de saída por ela requisitado já está alocado para outro pacote.

1.3 Motivações do Trabalho

Tendo presente o aumento da demanda de processamento e da utilização de vários processadores para execução de várias tarefas, o uso de sistemas multiprocessados em um único chip está se tornando cada vez mais relevante.

A implementação de projetos MPSoC é também uma importante linha de pesquisa do Grupo de Apoio ao Projeto de Hardware (GAPH). Podemos citar três desses projetos: MultiNoC, MPSoC-H e HeMPS.

1.4 Objetivos do Trabalho

O presente trabalho tem por objetivo estratégico o domínio das tecnologias MPSoC e NoC, desenvolvendo-se um novo MPSoC, com ênfase na integração rede–processador de forma estruturada.

Dentre os objetivos específicos cita-se:

a) Desenvolver módulos de hardware responsáveis pela interface entre a rede intra-chip e o processador;

b) Definir as políticas de acesso à rede, desenvolvendo os *drivers* necessários para atender a estas políticas;

c) Validar a estrutura de hardware-software com pelo menos um processador RISC.

1.5 Organização do documento

O presente documento está organizado como segue. O Capítulo 2 explora o estado da arte em MPSoC e NoCs, relacionando alguns tipos de MPSoCs acadêmicos e comerciais e alguns exemplos de arquiteturas NoC. O Capítulo 3 descreve a arquitetura de hardware e a infraestrutura utilizada para o desenvolvimento do MPSoC proposto. O capítulo 4 apresenta a infraestrutura de software utilizada para o desenvolvimento da implementação do tratamento de interrupções, dos *drivers* de comunicação e a interface de rede para conexão entre o hardware e a NoC. O resultados obtidos e a validação de toda a infraestrutura do MPSoC são apresentados no Capítulo 5, em que a validação se divide em dois momentos, a validação do sistema sem a implementação de interrupção e a validação do sistema com a implementação de interrupção com a inclusão de um aplicação paralela.

2 ESTADO DA ARTE

Este Capítulo apresenta arquiteturas de MPSoCs, tanto comerciais quanto acadêmicas. A segunda parte deste Capítulo revisa arquiteturas de redes intra-chip, como: Xpipes, HERMES, SoCIN, QNoC, AEThereal.

2.1 MPSoCs

2.1.1 MultiNoC

A MultiNoC desenvolvida por [MEL05] baseia-se na rede HERMES [MOR04a] [MOR04b] de tamanho 2x2 com controle de fluxo baseado em *handshake*, dois processadores R8 com 1024 palavras de memória *cache*, uma memória compartilhada de 1024 palavras e uma interface serial com o computador hospedeiro. A Figura 2 mostra os módulos que compõe o sistema MultiNoC.

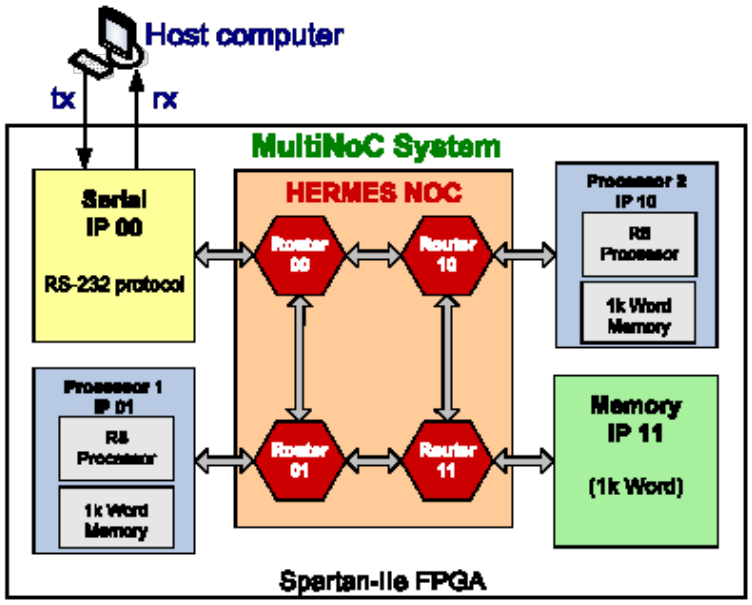

Figura 2 - Sistema MultiNoC [MEL05].

O processador R8 utilizado foi desenvolvido pelo grupo GAPH [GAP07], possui arquitetura load/store de 16 bits, endereçamento a palavra (cada endereço corresponde a um identificador de uma posição em que residem 16 bits de conteúdo), banco de registradores com 16 registradores de uso geral, 4 *flags* de estado e execução de instruções em 3 ou 4 ciclos. A memória (*memory IP*) possui 4 BlockRAMs, resultando em 1024 palavras de 16 bits. A interface serial possui protocolo RS-232 e fornece uma comunicação bidirecional com o computador hospedeiro.

A MultiNoC é uma arquitetura NUMA (*Non-Uniform Memory Access*), em que cada processador tem sua memória local, mas pode ter acesso à memória de outros processadores. Pode ser estendido para qualquer número de processadores assim como o número de memórias. Também pode ser adaptado a protocolos de comunicação mais rápidos, como USB, PCI e *Firewire*.

2.1.2 MPSoC-H

O projeto desenvolvido por [CAR05] implementa um MPSoC utilizando uma rede HERMES a partir do sistema MultiNoC. A Figura 3 ilustra a arquitetura do MPSoC-H (MPSoC-HERMES).

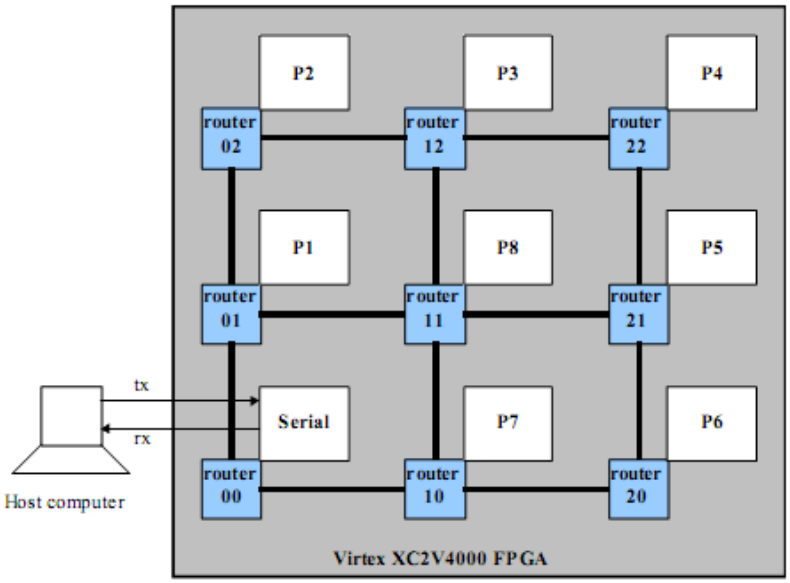

Figura 3 - Sistema MPSoC-H [CAR05].

As principais diferenças entre o sistema MultiNoC e o sistema MPSoC-H são: (1) rede HERMES 3x3 com controle de fluxo baseado em créditos; (2) 8 processadores R8 com 12K de palavras de memória *cache*; (3) ausência de memória compartilhada.

A principal função da interface serial é montar e desmontar pacotes. Quando recebe informações do computador hospedeiro, a interface serial as encapsula em pacotes e envia para os módulos através da NoC. Suas principais funcionalidades são: (*i*) carga e *dump* das memórias do sistema, (*ii*) ativação dos processadores, (*iii*) E/S de dados do sistema e (*iv*) *debug* do sistema.

Foi realizado a criação de um *wrapper* que possibilita a integração entre o processador e o MPSoC. Este *wrapper* tem por função, controlar a execução do processador provocando pausas quando o mesmo executar instruções de leitura, escrita, de entrada ou saída e proporcionar um conjunto de serviços que serve de suporte para comunicação com os demais módulos. Também inclui dentro deste *wrapper*, uma memória que serve como *cache* para o processador.

A memória *cache* foi implementada a partir de 12 módulos de BlockRAMs de 2048 palavras. Todos os serviços do *wrapper* foram mapeados em memória, que são efetuados usando as instruções load/store de acesso à memória.

Da mesma forma que foi feita com o processador, a interface serial também foi encapsulada dentro de um *wrapper* para facilitar a integração com o MPSoC-H. Os serviços implementados no *wrapper* serial são invocados através de um software que acessa a porta serial do computador hospedeiro.

2.1.3 HeMPS

Outro projeto que está relacionado ao tema MPSoC é o de alocação e comunicação entre tarefas em MPSoCs, implementados na plataforma HeMPS [WOS07]. Para a infraestrutura de hardware foi utilizado a NoC HERMES, e o processador de domínio público Plasma [PLA08]. Módulos de hardware foram desenvolvidos, visando conectar o processador à NoC e realizar a alocação de tarefas na memória do processador. Para a infraestrutura de software, foi desenvolvido um *microkernel* multitarefa que é executado em cada processador escravo e a aplicação de alocação de tarefas que é executada no processador mestre. São exploradas duas estratégias de alocação de tarefas: uma estática e outra dinâmica. A Figura 4 mostra a arquitetura proposta por este trabalho.

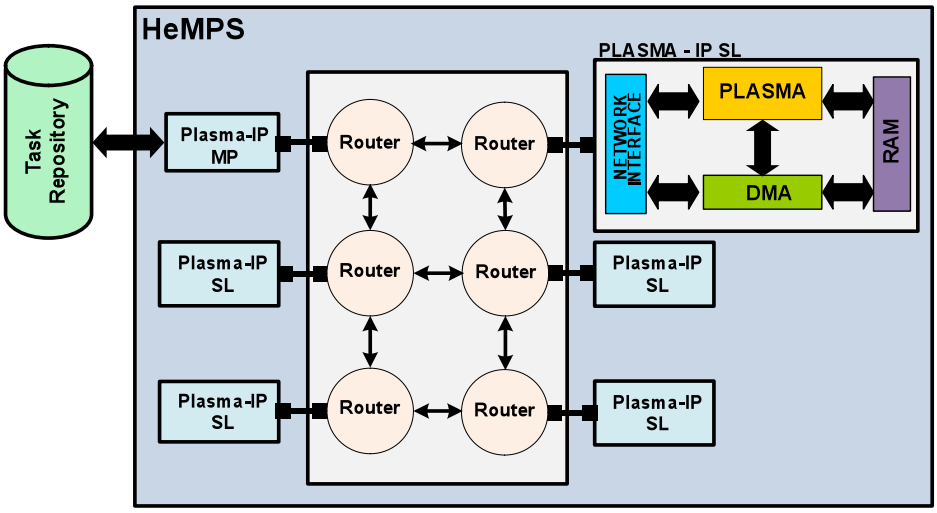

Figura 4 - Arquitetura HeMPS [WOS07].

2.1.4 Arquitetura CellBE

A arquitetura *CellBE* (*Cell Broadband Engeni Architecture*) (Figura 5) [KAH05], foi proposta em 2000 por um grupo denominado STI (Sony,Thoshiba,IBM), para estudar a possibilidade de criar um novo console para jogos. Nota-se que mesmo sendo um processador de uso geral, ele também é direcionado para aplicações em dispositivos embarcados.

O projeto da arquitetura *CellBE* deveria conciliar, além da velocidade de processamento, a redução do consumo de potência. A solução para resolver este problema foi criar um processador com uma arquitetura *multicore* heterogênea [KAH05]. Esta arquitetura permite alcançar alto desempenho, utilizando uma arquitetura paralela entre processadores de características diferentes. Baseado na arquitetura do processador PowerPC, o *CellBE* é constituído de nove núcleos, divido em um núcleo PowerPC e oito núcleos SPE (*Synergistic Processor Element*), ambos de propósito geral. O PPE (*PowerPC Processing Element*) é a unidade central de processamento, enquanto os SPEs são responsáveis por acelerar aplicações específicas. Além dos nove núcleos, a arquitetura possui um barramento principal EIB (*Element Interconect Bus*), um controlador de memória DEMAC (*Direct Memory Acces Controller*), um controlador de memória XDRAM, e uma entrada e saída de Rambus FlexIO.

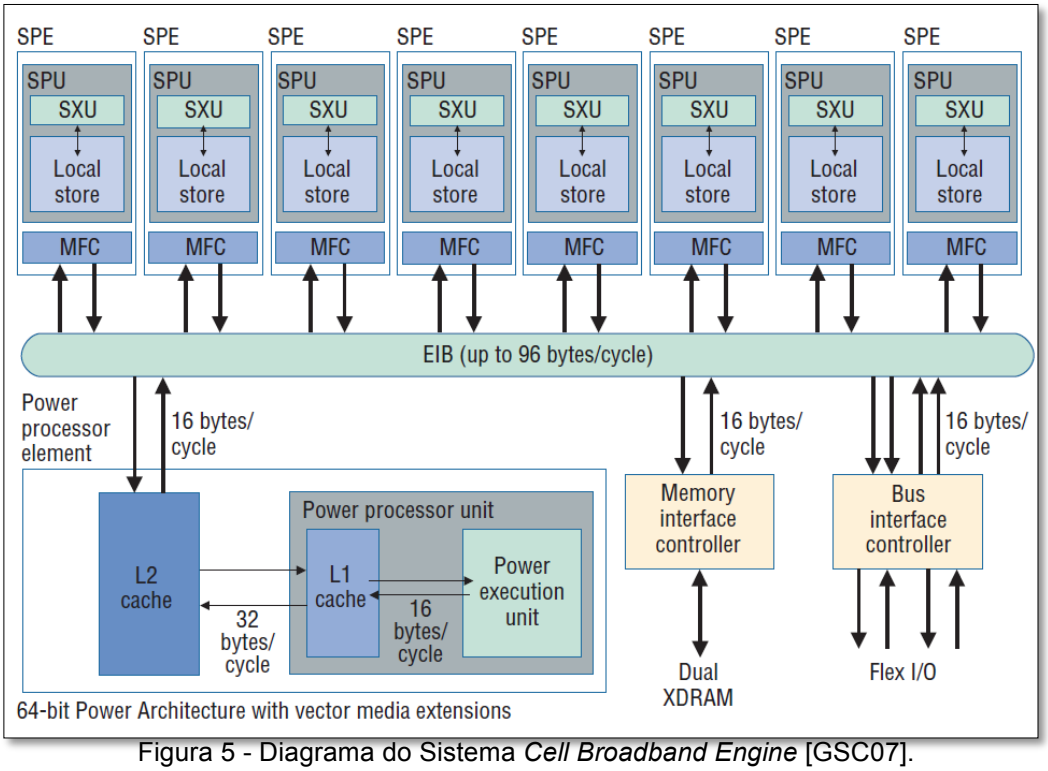

Figura 5 - Diagrama do Sistema *Cell Broadband Engine* [GSC07].

A Figura 6 apresenta o *layout* do processador *Cell*, fabricado inicialmente em tecnologia de 90nm [KAH05], oito camadas de metal, 234 milhões de transistores, e área de 221 mm^2.

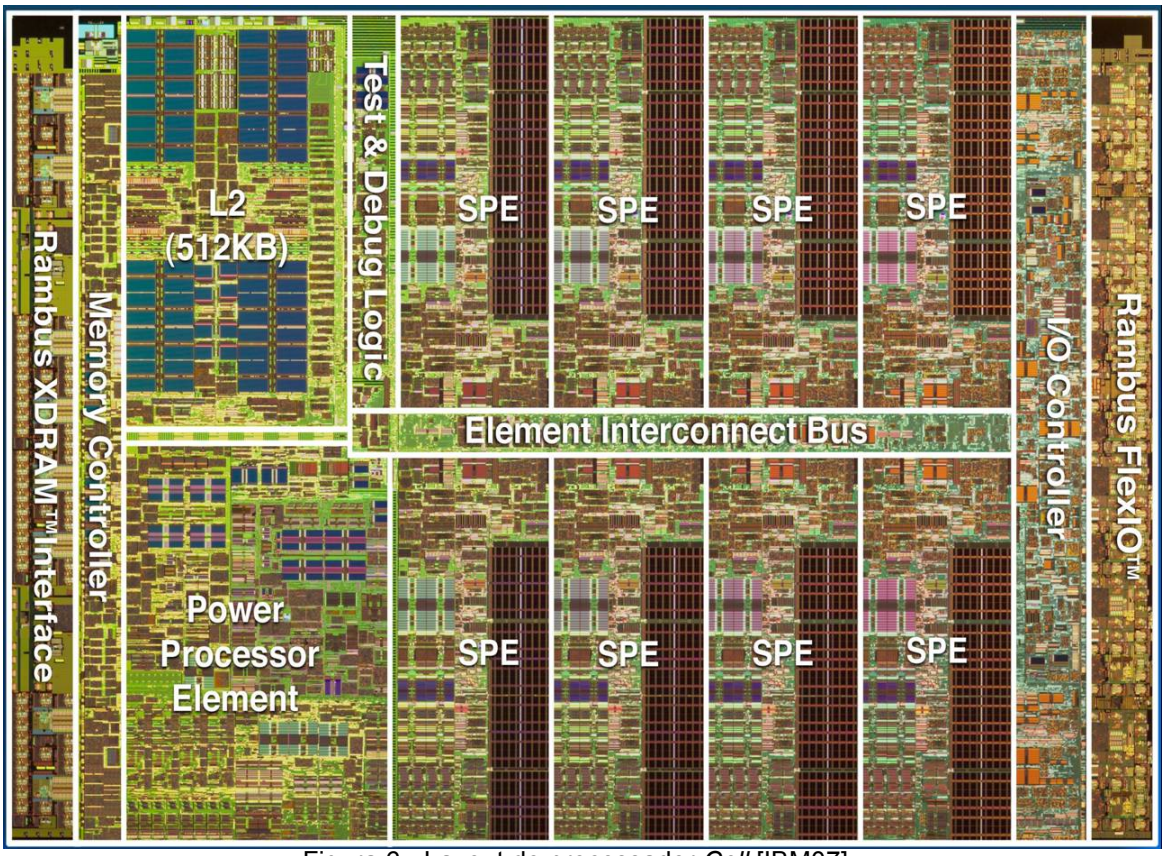

Figura 6 - Layout do processador *Cell* [IBM07].

2.1.4.1 Power Processor Element

O *Power Processor Element* (PPE) é um processador RISC baseado na arquitetura POWER PC, possuindo as mesmas funcionalidades de um processador de propósito geral. Seu principal objetivo é atribuir tarefas para as demais unidades de processamento (SPEs), funcionando como um controlador. Ele é também o responsável por executar o sistema operacional e a maioria das aplicações mais simples, sendo as atividades de computação intensiva atribuídas aos processadores SPEs.

O PPE é um processador de 64 bits. O PPE possui 64KB em sua memória *cache* L1, sendo essa dividida em 32KB de *cache* para instruções e outros 32KB para dados, além do SMT (*Simultaneous Multithreading*) que é um tecnologia similar ao *Hyper Threading*[1]. Há também uma *cache* L2 de 512KB, que foi reduzido para que não ocupasse uma área muito grande, e assim pudesse ser ocupado por outros processadores. Para compensar esta perda são utilizadas memórias XDR da Rambus (Figura 6) de uso externo.

A IBM também inclui uma unidade VMX (Altivec) para fazer cálculos de precisão dupla e trabalhar com duplo processamento (*dual issue*), o que significa que pode executar duas instruções simultaneamente por ciclo. Devido àa sua arquitetura RISC, os PPEs do *Cell* tem um consumo menor de energia do que o PowerPC, inclusive em altas taxas de relógio.

Além da *cache* L2, o PPE ainda possui uma PXU (*PowerPC eXecution Unit*), que capaz de executar instruções de duas threads simultaneamente. Esta PXU é dividida em três outros componentes menores. A IU (*Instruction Unit*), a XU (*Fixed Point Execution*) e a VSU (*Vector Scalar Unit*). A IU é responsável pelo controle do PPE e cuida de tarefas como *fetch* de instruções, decodificação e *branch prediction*. A XU é responsável pela execução das instruções que envolvem números inteiros e de instruções do tipo *Load* e *Store*. A VSU é responsável pela execução de instruções SIMD e também atua como unidade de ponto flutuante.

2.1.4.2 Synergistic Processing Element

O SPE foi criado para preencher uma lacuna existente entre os processadores de propósito geral e os de propósito específico (melhor desempenho em uma única aplicação). O objetivo principal é garantir o desempenho no processamento de grande quantidade de dados de jogos, aplicações multimídia e sistemas de grande largura de banda de dados, como áudio, vídeo, criptografia e simulação de fenômenos físicos.

Cada um dos oito SPEs é um microprocessador RISC SIMD (*Single Instruction Multiple Ddata*) independente constituído de um núcleo de processamento SPU (*Synergistic Processor Unit*) e uma memória local de 256 KB (ele não possui uma memória *cache* própria) e um MFC (*Memory Flow Controller*). O SPE, assim como PPE é capaz de fazer processamento duplo e uma unidade de SPE pode calcular até 25,6 GFlops, resultando em um total de até 200 GFlops.

As SPUs possuem bancos de 128 registradores com 128 bits cada. Dessa forma cada

[1] Hyper Threading - tecnologia desenvolvida pela Intel , utilizada em processadores que o faz simular dois processadores tornando o sistema mais rápido quando se usa vários programas ao mesmo tempo.

instrução pode operar com 16 inteiros de 8 bits, 8 inteiros de 16 bits, 4 inteiros de 32 bits capazes de realizar 32 bilhões de operações por segundo, 4 unidades de ponto flutuante de precisão simples ou ainda 2 de precisão dupla, capazes de realizar 32 bilhões de operações por segundo, todos operando em uma frequência de 4GHz.

Sabe-se que o acesso constante à memória causa um custo muito grande para uma melhor performance, tentando minimizar este problema o SPE possui uma memória própria (*cache*), evitando assim o acesso direto à memória do sistema, ou seja, cada SPE possuí uma região de memória chamada de LS (*Local Store*) com um tamanho de 256KB. Esta memória é extremamente rápida, e é utilizada para guardar quantas instruções forem necessárias. Com essa implementação houve uma redução do tamanho do chip, pois ocorreu consequentemente a redução do número de transistores utilizados. A responsabilidade de gerenciar o movimento de dados entre a memória principal e a LS fica a cargo do software.

O MFC (*Memory Flow Controller*) serve como uma interface para o sistema e o SPU, criando uma polarização do sistema de transferência de dados, proteção e sincronia entre a memória principal e a área local do SPU. Ela é responsável pela comunicação com a memória principal por DMA, onde através de filas fazem a transferência de dados com a memória principal, que pode ser iniciada tanto pelos PPEs quanto pelos SPEs.

2.1.4.3 Element Interconect Bus

O EIB (*Element Interconect Bus*) é o barramento responsável pela interconexão entre os SPEs, o PPE e o controlador de I/O, que permite aos núcleos efetuar leitura e gravação de forma independente e simultânea com relação aos outros núcleos.

Foi implementado na forma de quatro anéis que circulam em sentidos contrários em pares. Este barramento trabalha com a metade do ciclo de *clock* do processador, sendo capaz de transmitir 16 bytes a cada dois *clocks* do sistema. Devido a esta característica este barramento é definido como barramento de oito *bytes* por ciclos.

2.1.5 TILE64

O TILE64 foi desenvolvido pela *Tilera Corporation* [TIL07] baseado em uma arquitetura capaz de gerenciar centenas de núcleos em um mesmo encapsulamento. Inicialmente foi projetado para trabalhar como chip em redes inteligentes e equipamentos de distribuição de mídia. Empresas como 3Com, Codian e GoBackTV já utilizam esta tecnologia. Comparando com o desempenho de um Xeon-Dual Core para sistemas embarcados em uma mesma situação, ele atinge um pico 10 vezes maior.

Este processador possui uma rede de 64 núcleos (*tiles*) de 32 bits (Figura 7) com características idênticas, conectado por uma rede malha, responsável pela comunicação entre os núcleos e com o mundo exterior [CRA07]. Cada *tile* é um processador, incluindo uma memória *cache* integrada com L1 e L2 que interliga o *tile* à malha. A rede malha é de fato constituída por cinco sub-redes independentes, duas redes são totalmente gerenciadas pelo hardware e podem

transportar diferentes tipos de dados de e para cada *tile* ou transferência via DMA, as outras três redes estão disponíveis para uso da aplicação, permitindo a comunicação entre os núcleos e entre os núcleos e dispositivos I/O (Figura 7).

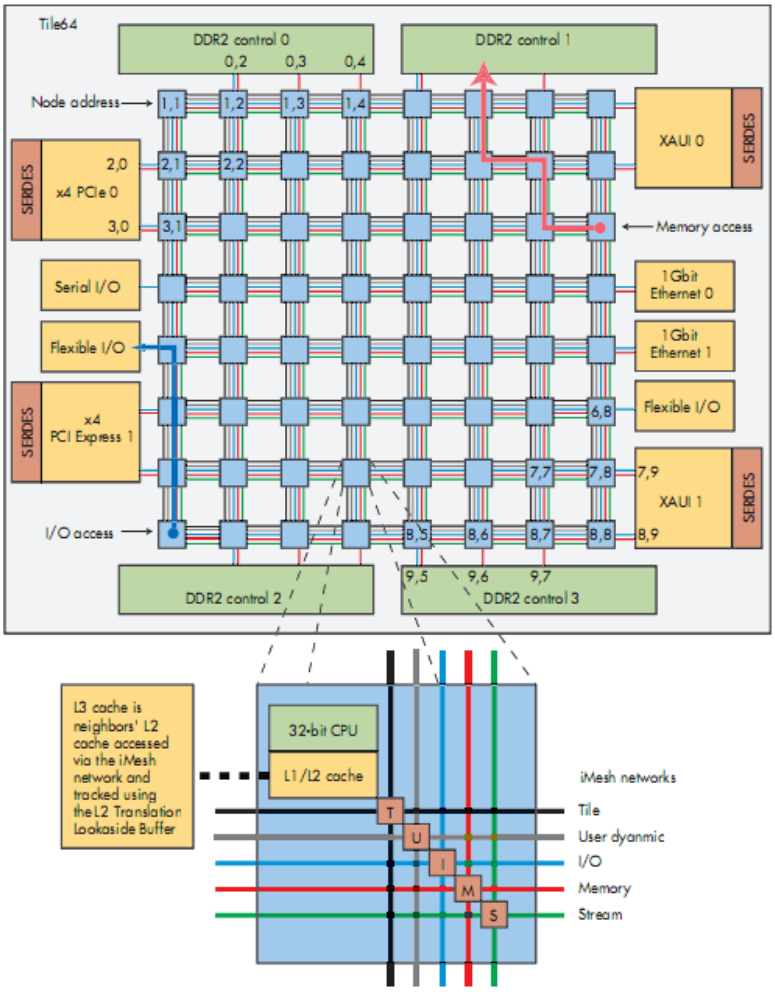

Figura 7: Arquitetura TIL64 com 8x8 tiles [CRA07].

A rede *IMesh* (*Intelligent Mesh*)[WEN07] permite uma comunicação sem interromper as aplicações que estão rodando nos *tiles*. Isso facilita a transferência de dados entre os *tiles*, contendo um total controle para cada conexão de rede.

Ao invés de um barramento centralizado, cada núcleo tem sua própria memória, podendo compartilhar dados com quatro núcleos vizinhos. Como mostra a Figura 8, a memória está dividida em duas *caches* L1 de 8KB e uma L2 de 64KB, com L3 simulado até 5MB quando necessário. O CI também integra uma série de interfaces incluindo XAUI, *gigabit Ethernet*, UARTs, Flexibel I/O e quatro controladores DDR2.

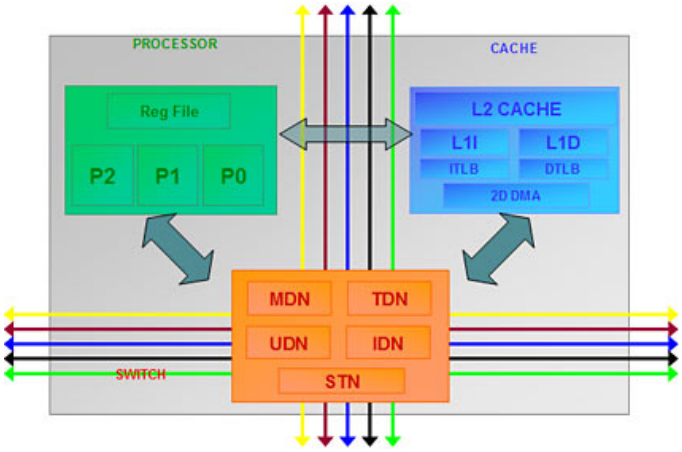

Figura 8 - Layout de um *tile* [AGA07].

A arquitetura do processador TILE64 incorpora um *array* bidimensional homogêneo de núcleos de propósito geral. Para cada processador há um roteador que interliga o núcleo à rede. A combinação de um núcleo e um roteador forma a construção básica de um *tile.* Cada núcleo é um processador totalmente funcional capaz de executar sistemas operacionais completos. Cada núcleo é otimizado para fornecer o maior desempenho com a menor potência, podendo trabalhar a velocidades entre 500 MHz e 1 GHz, com consumo de energia a um nível tão baixo como 170 mW em uma simples aplicação.

2.1.6 Intel Core – 80 (Polaris)

A Intel vem realizando pesquisas em processadores com vários núcleos. Uma das últimas pesquisas em processadores da Intel é o chamado Polaris ou Intel Core-80, que possui 80 núcleos [VAN07]. Este processador, de acordo com a Intel, é capaz de realizar um trilhão de operações em ponto flutuante por segundo, ou 1 Teraflop. Fazendo uma breve comparação, tomemos como exemplo o supercomputador ASCI *Red*, que em 1996 para atingir os mesmos 1 Teraflop utiliza 10 mil processadores Pentium Pro[1] de 200 MHz, seu consumo era de 500 kW e outros 500 kW somente para refrigerar a sala onde se encontrava.

Cada um dos 80 núcleos possui 1,2 milhões de transistores ocupando uma área de 3 mm^2, totalizando 100 milhões de transistores para uma área total de 275 mm^2 [VAN07]. Todos os núcleos compartilham uma região de *cache* de 4 MB. Seu processo de fabricação é de 65 nanômetros. A Figura 9 apresenta a estrutura do processador e a estrutura de cada *tile*.

Em uma primeira versão do processador Polaris, sua frequência de trabalho era de 3,16 GHz, cada núcleo trabalha com uma tensão de 0,95 V e com um consumo total de 62 W. Para obter este consumo a Intel utilizou a tecnologia denominada de *Fine-Grain Power Management*, que coloca em estado ocioso os núcleos não utilizados.

[1] O Pentium Pro internamente funciona como se fosse três processadores em paralelo e é capaz de executar até três instruções por pulso de *clock*. Possui 5,5 milhões de transistores, largura de BUS de 64 bits e *cache* L2 de 256K e 512K e co-processador interno.

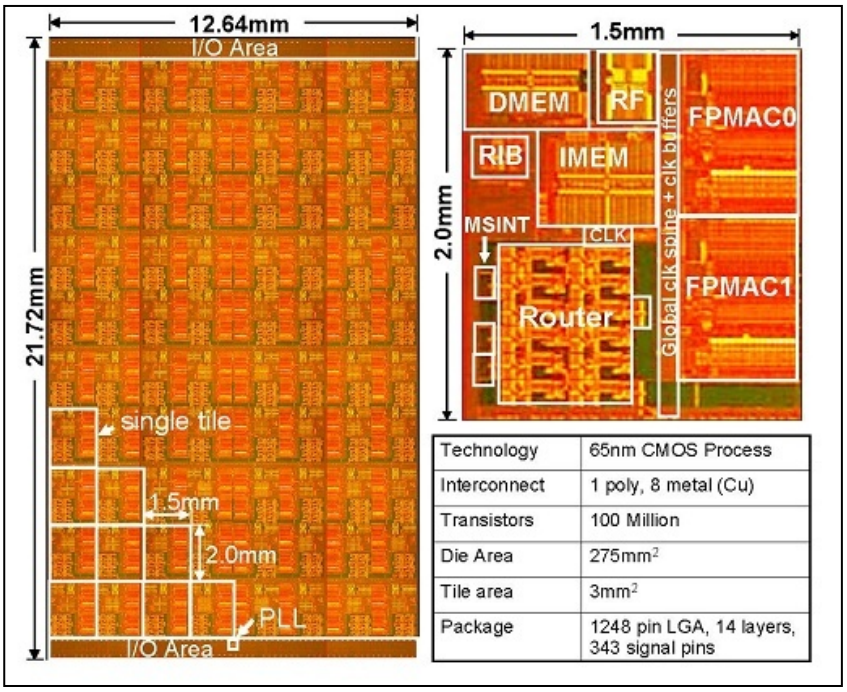

Figura 9 – Protótipo da estrutura do processador POLARIS e de cada *tile* que o compõe [VAN07].

A arquitetura NoC do processador Polaris é apresentada na Figura 10 que contém 80 *tiles* arranjados em uma rede com uma topologia malha 2D de 10x8 e operando à uma frequência de 4 GHz. Cada *tile* consiste de um PE (*processing engine*) com interface mesócrona [VAN08] que direciona os pacotes entre os *tiles* e está conectado a um circuito de roteamento com 5 portas de comunicação, com alta vazão e baixa latência para a troca de mensagens. O roteador da NoC possui chaveamento do tipo *wormhole* e dois canais físicos para transmissão de pacotes.

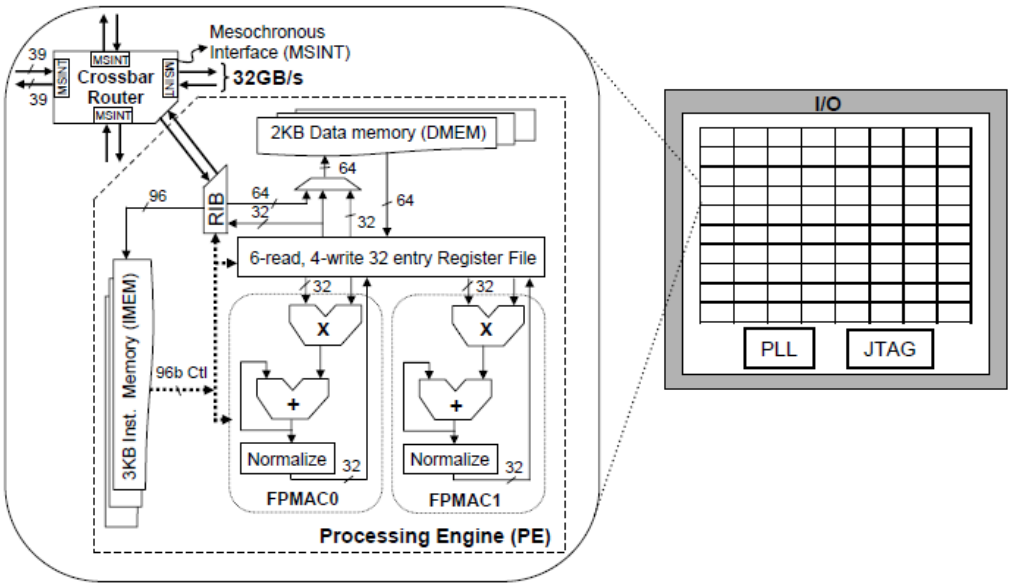

Figura 10 – Diagrama de blocos da NoC e arquitetura do *tile* [VAN07].

A NoC do Polaris habilita uma largura de banda de 256 GB/s. Todos os PEs possuem duas unidades de processamento de ponto flutuante (FPMAC), uma memória de instrução de ciclo único com 3KB (IMEM) e uma memória de dados com 2KB (DMEM). Uma VLIW (*Very Long Instruction Word*) de 96 bits codifica até 8 operações por ciclo. Com um arquivo de registro de 10

portas (6-ler, 4-escrever), a arquitetura permite a programação de ambas FPMACs, carregar e armazenar a DMEM simultaneamente, enviar/receber pacotes da rede malha, controle de programa, e *dynamic sleep instructions.* Um bloco de interface de roteamento (RIB) trata o encapsulamento do pacote entre o PE e o roteador. Uma arquitetura totalmente simétrica permite a qualquer PE enviar/receber instruções e pacotes de dados de qualquer outro tile.

2.2 NoCs

2.2.1 NoC – Xpipes

A rede Xpipes foi proposta por Bertozzi et. al. [BER05] [DAL03] para SoCs multiprocessados. Esta NoC possui roteamento *wormhole* e faz uso do algoritmo de roteamento estático denominado *street sign routing*. As rotas são obtidas através da interface de rede, acessando uma tabela que tem por entrada o endereço destino. Este algoritmo de roteamento permite uma implementação simples do roteador porque nenhuma decisão dinâmica tem que ser tomada nos nodos de processamento.

Uma das principais preocupações no projeto da Xpipes foi a confiabilidade na comunicação. Isto foi alcançado por meio de detecção de erro distribuída, ou seja, a cada roteador. Embora a detecção de erro distribuída cause uma sobrecarga de área nos nodos da rede, se comparada com uma solução fim-a-fim, ela é mais bem capacitada para conter os efeitos da propagação de erro, por exemplo impedindo que um cabeçalho corrompido seja transmitido para um caminho errado.

A rede intra-chip Xpipes possui alto grau de parametrização. A parametrização inclui o tamanho do *flit*, o espaço de endereçamento dos núcleos, o número máximo de roteadores entre dois núcleos, o número máximo de bits para controle de fluxo fim-a-fim, a profundidade do *buffer*, o número de canais virtuais por canal físico, entre outros parâmetros.

Na Figura 11 é apresentado o esquema do roteador, o qual possui 4 entradas, 4 saídas e 2 canais virtuais multiplexados sobre o mesmo canal físico. O roteador adota buferização de saída e a arquitetura resultante consiste de múltiplas replicações do mesmo módulo de saída apresentado na Figura 12, um para cada porta de saída do roteador. Os sinais de controle de fluxo gerados em cada módulo (tal como ACK e NACK para *flits* de entrada) são coletados por uma unidade centralizada do roteador, a qual transmite ao roteador fonte apropriado.

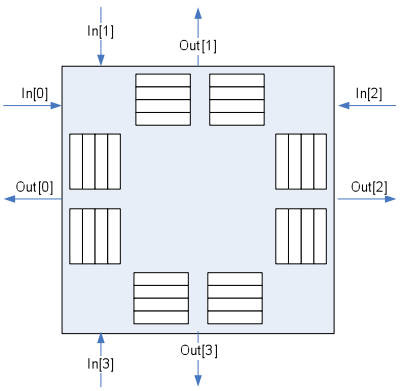

Figura 11 - Arquitetura do roteador Xpipes com 2 canais virtuais [BER04].

Como pode ser observado na Figura 12, cada módulo de saída tem 7 estágios de *pipeline* para maximizar a frequência de operação de *clock* do roteador. Os decodificadores CRC para detecção de erro trabalham em paralelo com a operação do roteador, desse modo ocultando sua latência.

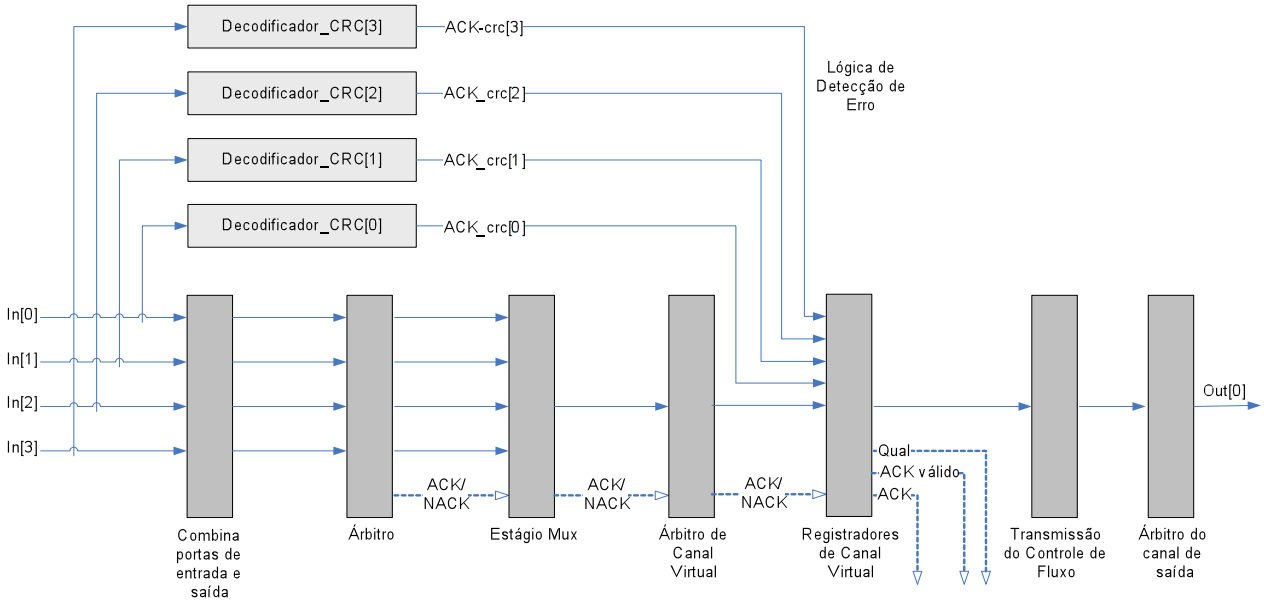

Figura 12 - Arquitetura do módulo de saída para cada porta de saída do roteador Xpipes [BER04].

O primeiro estágio do *pipeline* verifica o cabeçalho dos pacotes nas diferentes portas de entrada para determinar se os pacotes têm que ser roteados através de determinada porta de saída. Somente pacotes compatíveis com a porta de saída são enviados para o segundo estágio, no qual são resolvidas disputas baseadas em uma política *round robin*. A arbitragem é realizada quando o *flit* terminador do pacote antecessor é recebido, de modo que todos os outros *flits* do pacote possam ser propagados sem disputa. O terceiro estágio possui apenas um multiplexador, o qual seleciona a porta de entrada prioritária. O estágio seguinte de arbitragem guarda o status do registrador de canal virtual e determina se *flits* podem ser armazenados no registrador ou não. Um *flit* de cabeçalho é enviado para o registrador com o maior número de posições livres, seguido por sucessivos *flits* do mesmo pacote. O quinto estágio é o estágio de buferização, e a resposta ACK/NACK neste estágio indica se um *flit* foi armazenado corretamente ou não. O estágio seguinte cuida do envio do controle de fluxo: um *flit* é transmitido para o próximo roteador somente quando a porta de saída do roteador destino possuir posições livres disponíveis no registrador adequado. O último estágio de arbitragem multiplexa os canais virtuais no enlace do canal físico.

A interface de rede Xpipes fornece uma interface padronizada OCP para conectar aos núcleos. A NI dos núcleos que iniciam a comunicação (iniciadores) precisa transformar o OCP em pacotes para serem transmitidos através da rede. Ele representa o lado escravo de uma conexão OCP fim-a-fim, e, por isso, é chamado de *network interface slave* (NIS). Sua arquitetura é visualizada na Figura 13.

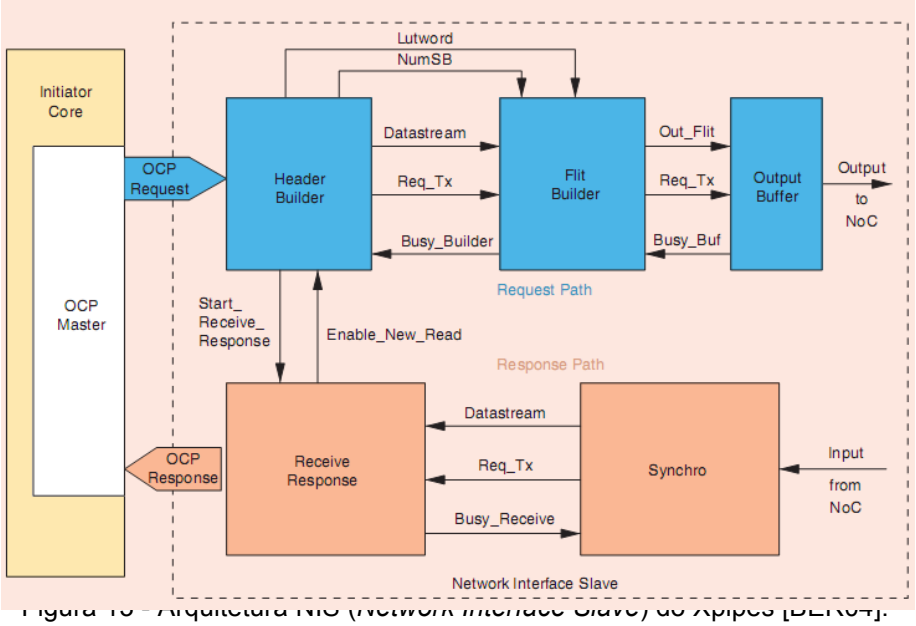

Figura 13 - Arquitetura NIC (*Network Interface Slave*) do Xpipes [DER04].

2.2.2 NoC HERMES

A rede HERMES foi desenvolvida pelo grupo de pesquisa GAPH [GAP07]. A rede HERMES possui um mecanismo de comunicação denominado chaveamento de pacotes, no qual os pacotes são roteados individualmente entre os nodos sem o estabelecimento prévio de um caminho. Este mecanismo de comunicação requer o uso de um modo de roteamento para definir como os pacotes devem se mover através dos roteadores (Figura 14d). A Rede HERMES utiliza o modo de roteamento *wormhole*, no qual um pacote é transmitido entre roteadores em *flits* (Figura 14a), cada pacote é transmitido *flit* a *flit* pelos canais físicos. Apenas o *flit* de cabeçalho possui a informação de roteamento. Assim, os *flits* restantes que compõe o pacote devem seguir o mesmo caminho reservado pelo cabeçalho [WOS07]. A rede HERMES adota a topologia malha (Figura 14c), a qual é justificada em função da facilidade de desenvolver o algoritmo de roteamento, inserir núcleos e gerar o *layout* do circuito.

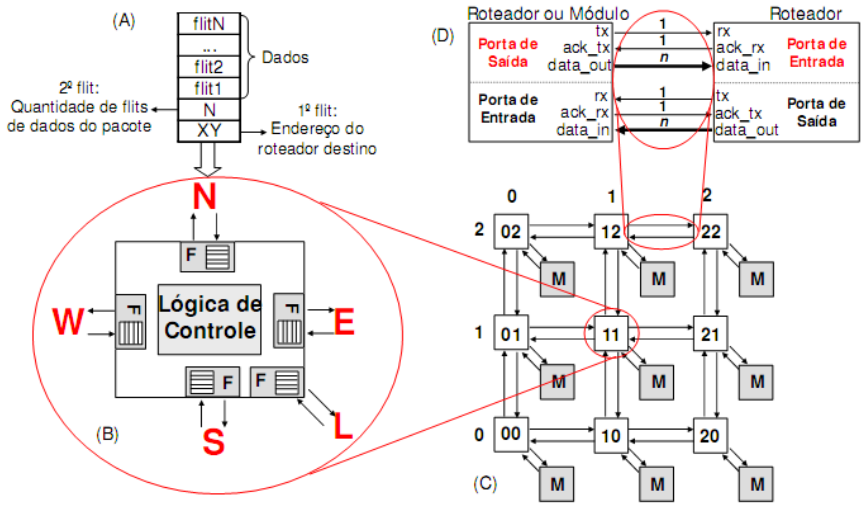

Figura 14 - Estrutura geral da Rede HERMES: (a) formato dos pacotes que trafegam pelos roteadores; (b) diagrama de blocos do roteador; (c) Rede HERMES 3x3; (d) interface roteador-roteador ou módulo-roteador [WOS07].

O roteador da rede HERMES possui uma lógica de controle de chaveamento centralizada e 5 portas bidirecionais: *East*, *West*, *North*, *South* e *Local* (Figura 14b). A porta *Local* estabelece a comunicação entre o roteador e seu núcleo local. As demais portas ligam o roteador aos roteadores vizinhos. O modo de chaveamento *wormhole* adotado no roteador HERMES permite que cada canal físico seja multiplexado em *n* canais virtuais (*VCs*) [WOS07]. A Figura 15 apresenta o roteador HERMES com dois *VCs* por canal físico .

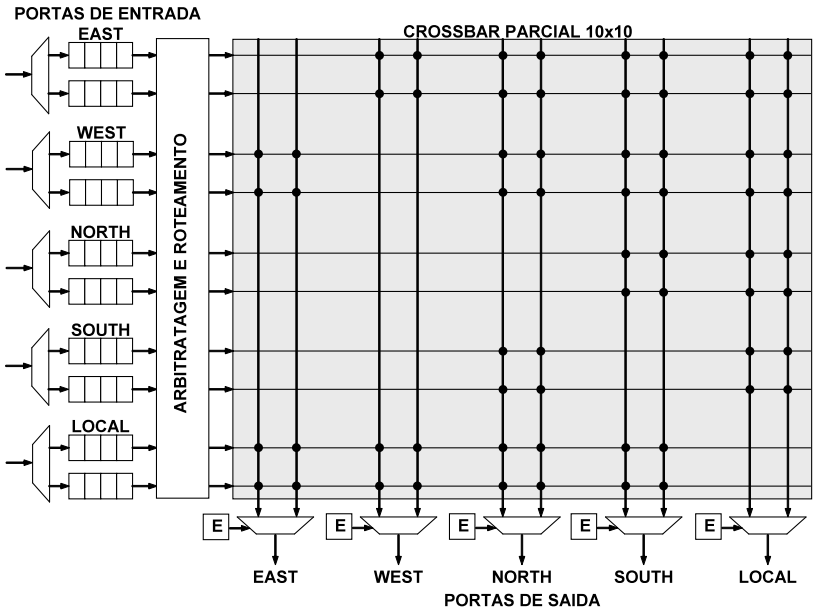

Figura 15 - Estrutura do roteador HERMES com dois canais virtuais. O módulo "E" na porta de saída representa o de um circuito que escalona uma dada porta de entrada para uma dada porta de saída [WOS07].

A cada porta de entrada é adicionado um *buffer* para diminuir a perda de desempenho com o bloqueio de *flits*. A perda de desempenho acontece porque quando um *flit* é bloqueado em um dado roteador, os *flits* seguintes do mesmo pacote também são bloqueados em outros roteadores. Com a inserção de um *buffer* o número de roteadores afetados pelo bloqueio dos *flits* diminui. O *buffer* inserido no roteador da rede HERMES funciona como uma fila *FIFO* (*First in – First Out*) circular, cuja profundidade é parametrizável. Quando o canal físico é dividido em *n VCs*, uma fila *FIFO* de profundidade *p*/*n* é associada a cada *VC*. Por exemplo, o roteador apresentado na Figura 15 possui um espaço de armazenamento de 8 *flits* por porta de entrada, sendo multiplexado em dois *VCs*, com cada *buffer* associado a cada *VC* com profundidade de 4 *flits* (8/2) [WOS07].

A lógica de controle de chaveamento implementa uma lógica de arbitragem e um algoritmo de roteamento. Quando um roteador recebe um *header flit*, a arbitragem é executada e se a requisição de roteamento do pacote é atendida, um algoritmo de roteamento é usado para conectar o *flit* da porta de entrada à correta porta de saída. Cada roteador deve ter um endereço único na rede. Para simplificar o roteamento na rede, este endereço é expresso nas coordenadas XY, onde X representa a posição horizontal e Y a posição vertical [WOS07].

2.2.3 SoCIN

A rede SoCIN (*System-on-Chip Interconnection Network*), foi desenvolvida pelo Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (PPGC-UFRGS) [ZEF03a].

A rede SoCIN possui topologia direta podendo ser configurada uma malha 2-D (Figura 16a) ou toróide 2D. Possui controle de fluxo do tipo *handshake,* roteamento do tipo fonte e determinístico, chaveamento por pacote tipo *wormhole*, arbitragem dinâmica distribuída e memorização de entrada. É baseado em um *soft-core* de um roteador parametrizável denominado RASoC (*Router Architecture for SoC*). Esse *soft-core* é descrito em VHDL e possui quatro parâmetros básicos: (*i*) número de portas de comunicação (até 5); (*ii*) largura da parte de dados do canal físico (*n*); (*iii*) profundidade dos *buffers* de memorização (*p*); (*iv*) e largura da informação utilizada para roteamento das mensagens (*m*), a qual determina a dimensão máxima da rede.

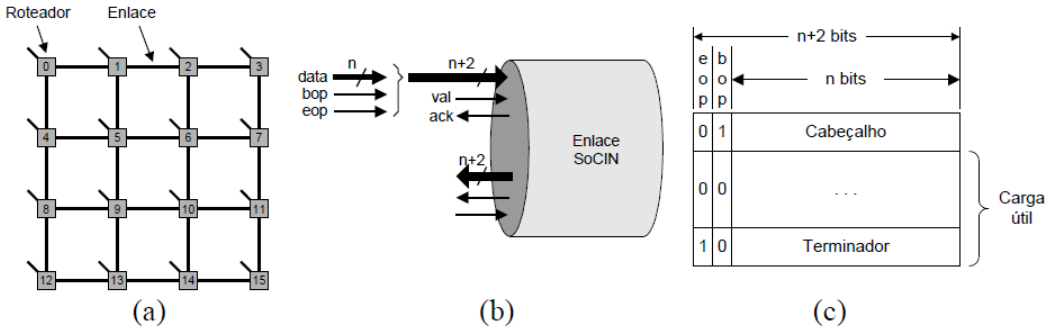

Figura 16 – (a)Topologia básica em grelha, (b) enlace SoCIN , (c) formato do pacote [ZEF03b].

O enlace (Figura 16b) da rede SoCIN possui dois canais unidirecionais em oposição. Cada canal possui *n* sinais de dado, dois sinais de enquadramento do pacote (*bop* e *eop*) e dois sinais de controle de fluxo (*val* e *ack*). O sinal *bop* (*begin-of-packet*) marca o início do pacote, enquanto que o *eop* (*end-of-packet*) marca o seu final. O sinal *val* (*valid*) é utilizado pelo emissor para sinalizar a presença de um dado válido no canal, enquanto que o sinal *ack* (*acknowledgement*) é utilizado pelo receptor para confirmar o recebimento do dado. Os *n* sinais de dado e os sinais de enquadramento constituem o *phit* (*n+2 bits*).

O pacote (Figura 16c) do protocolo da rede SoCIN é composto por um *flit* de cabeçalho e por um número irrestrito de *flits* que compõem a carga útil do pacote, sendo destinados ao enquadramento do pacote que o último *flit* da carga útil é também o terminador do pacote. Cada *flit* possui *n+2* bits, dos quais n bits são reservados ao transporte de informação (ou dado) e dois bits são destinados ao enquadramento do pacote.

2.2.4 QNoC

A arquitetura *Quality-of-Service NoC* (QNoC) proposta por Bolotin et al. [BOL04] está baseada em uma topologia malha com chaveamento de pacotes *wormhole* e com controle de fluxo baseado em créditos. O roteamento *wormhole* reduz a latência e o requerimento de filas nos roteadores. A QNoC oferece quatro classes de serviço:

1. *Sinalização*: nível de serviço com a maior prioridade na rede, para assegurar baixa latência. É utilizado por interrupções e sinais de controle.
2. *Tempo-real*: nível de serviço que garante largura de banda e latência para aplicações de tempo real.
3. *Leitura/Escrita (RD/WR)*: nível de serviço projetado para suportar acessos curtos a memórias e registradores.
4. *Transferência de blocos*: nível de serviço usado para transferência de mensagens longas e blocos grandes de dados, tal como conteúdo de *cache* e transferências DMA.

Os pacotes transportam as informações, *command* e *payload*. Na Figura 17 podemos visualizar o formato básico do pacote. O campo TRA (*Target Routing Address*) contém o endereço solicitado pelo roteamento. O campo *command* indentifica o *payload*, especificando o tipo de operação. O resto da informação é o *payload*, de comprimento arbitrário.

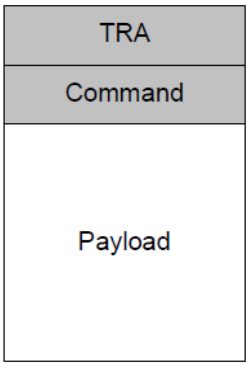

Figura 17 - Formato do pacote [BOL04].

O pacote é dividido em *flits* que são classificados dentro dos seguintes tipos: (*i*) *FP (Full Packet)*, primeiro *flit* do pacote; *EP (End of Packet)*, último *flit* do pacote; *BDY(Body)*, não é o último *flit* do pacote.

O tipo do *flit* e o nível de serviço são indicados em fios separados de controle. A Tabela 1 descreve os sinais de entrada e saída da porta de saída.

Tabela 1 - Sinais da Interface do Roteador.

	Sinais	Largura (*bit*)	Descrição
Sinais de Saída	*Clk*	1	Relógio que sincroniza a transmissão de *flits*
	Data_o	Parametrizável	Dados saindo do roteador
	Type	2	Tipo do *flit*: 00: *Idle* 01: *EP* 10: *BDY* 11: *FP*
	SL	2	Nível de serviço do *flit*
Sinais de Entrada	*Buffer_Credit_SL*	4	Indica se existe espaço disponível em buffer para cada nível de serviço.
	Buffer_Credit_valid	1	Indica se *Buffer_Credit_SL* transporta uma informação de crédito válido.

A Figura 18 ilustra a arquitetura do roteador. O roteador possui até cinco conexões: quatro

para roteadores vizinhos e uma para o núcleo local. O roteador transfere pacotes das portas de entrada para as portas de saída. Dados são recebidos em *flits*. Cada *flit* que chega é primeiro armazenado em *buffers* de entrada. Existem *buffers* separados para cada um dos quatro níveis de serviço. O algoritmo de roteamento determinístico XY é invocado quando o primeiro *flit* do pacote é recebido. O primeiro *flit* contém o endereço destino do pacote e é utilizado para determinar a qual porta de saída o pacote é destinado. O número da porta de saída selecionada para a transmissão do pacote de cada nível de serviço é armazenado na Tabela de Roteamento Atual (*CRT*, do inglês *Current Routing Table*). Quando um *flit* é enviado da porta de entrada para a porta de saída, um *buffer* torna-se disponível e um crédito de *buffer* é enviado ao roteador anterior. Cada porta de saída do roteador é conectada a uma porta de entrada do próximo roteador. A porta de saída gerencia o número de posições disponíveis no *buffer* de cada nível de serviço da próxima porta de entrada. Estes números são armazenados no Estado do Próximo *Buffer* (*NBS*, do inglês *Next Buffer State*). O número é decrementado quando um *flit* é transmitido e incrementado quando um crédito de *buffer* do próximo roteador é recebido.

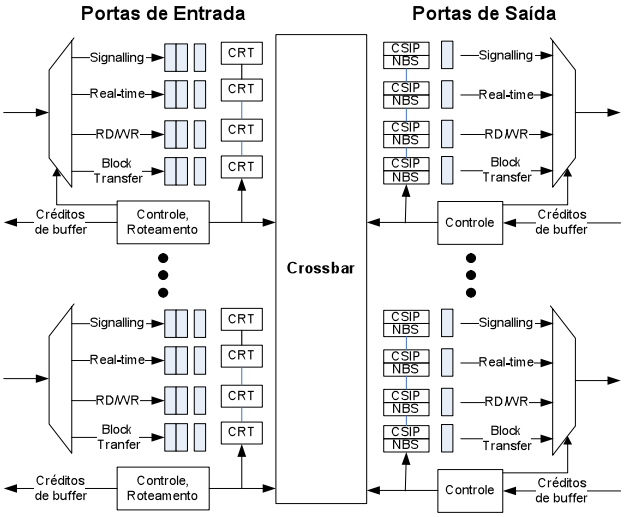

Figura 18 - Arquitetura do roteador [BOL04].

A porta de saída escalona a transmissão de *flits* de acordo com a disponibilidade de *buffers* no próximo roteador, com a prioridade do nível de serviço e com a arbitragem *round-robin* das portas de entrada esperando transmissão de pacotes dentro do mesmo nível de serviço. O número de *buffers* disponíveis no próximo roteador é armazenado na tabela NBS de cada nível de serviço de cada porta de saída. A prioridade dos níveis de serviço é fixa, ordenada com *Signalling* tendo a maior prioridade, *Real-time* tendo a segunda, *RD/WR* a terceira e o *Block-Transfer* a última. O estado atual da arbitragem *round-robin* é armazenado na tabela do número da Porta de Entrada Atualmente Servida (*CSIP*, do inglês *Currently Serviced Input Port number*) para cada nível de serviço de cada porta de saída. Este número avança quando a transmissão de um pacote é finalizada ou se nenhuma transmissão de uma determinada porta de entrada e nível de serviço existe.

2.2.5 ÆThereal

Goossens et. al. propuseram a rede intra-chip ÆThereal [GOO05]. Ela oferece serviços diferenciados com conexão. Uma conexão descreve a comunicação entre um núcleo origem e um ou mais núcleos destino, com um nível de serviço associado. Conexões devem ser criadas expressando o nível de serviço requisitado. A aceitação da conexão pode incluir a reserva de recursos na rede, tais como *buffers* ou o percentual de largura de banda do canal. Depois do uso, a conexão é finalizada e os recursos liberados. Diferentes conexões podem ser criadas ou finalizadas independentemente.

A rede intra-chip ÆThereal possui um roteador que combina vazão garantida (GT) e melhor esforço (BE), como ilustrado na Figura 19. Desse modo é garantido o comportamento de pior caso aliado com uma boa média de uso de recursos. O serviço GT oferece uma latência fim-a-fim fixa e tem a mais alta prioridade, forçada pelo árbitro. O serviço BE usa toda a largura de banda que não está reservada ou não é usada pelo tráfego GT. Recursos são sempre utilizados quando existem dados disponíveis.

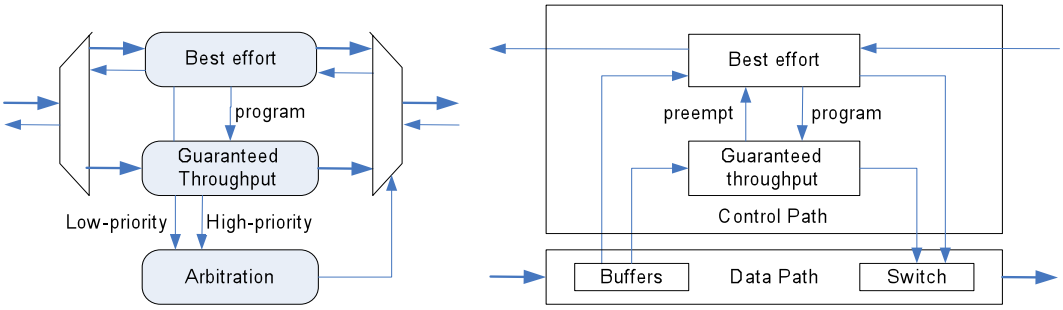

Figura 19 – Divisão do Roteador que combina tráfego melhor esforço (BT) e vazão garantida (GT) [GOO05].

As conexões GT são criadas ou removidas através de pacotes BE que reservam um caminho na rede, dependendo da disponibilidade de recursos (linha "*program*" na Figura 19). Os pacotes BE usam roteamento *wormhole* na origem, e os fluxos GT usam chaveamento por circuito com STDM. Roteadores usam filas virtuais de saída para o tráfego BE.

3 ARQUITETURA DO MPSOC DESENVOLVIDO

Este Capítulo apresenta a primeira contribuição desta Dissertação. Neste Capítulo são apresentados os ambientes de desenvolvimento utilizados para o projeto do hardware, a descrição do processador, assim como os acréscimos realizados na descrição no escopo do trabalho. Esta arquitetura é um MPSoC homogêneo, constituído de vários núcleos de processamento , compostos por processadores MR4 e conectados através da NoC HERMES com uma topologia do tipo malha 2D, como ilustrado na Figura 20.

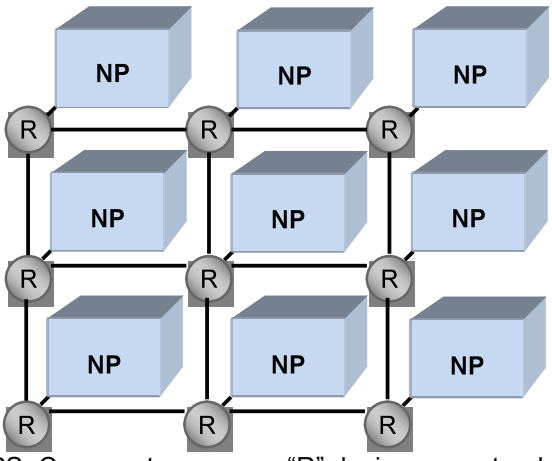

Figura 20 – Arquitetura do MPSoC proposto, em que “R” designa os roteadores da rede HERMES, e “NP” os núcleos de processamento.

Os núcleos de processamento contêm: (*i*) um processador MR4 com memórias de dados e instruções; (*ii*) interface de recepção de dados (*Incoming Unit*); interface de transmissão de dados (*Outcoming Unit*); (*iv*) interface de rede (*Network Iterface*). A arquitetura do NP é apresentada na Figura 21.

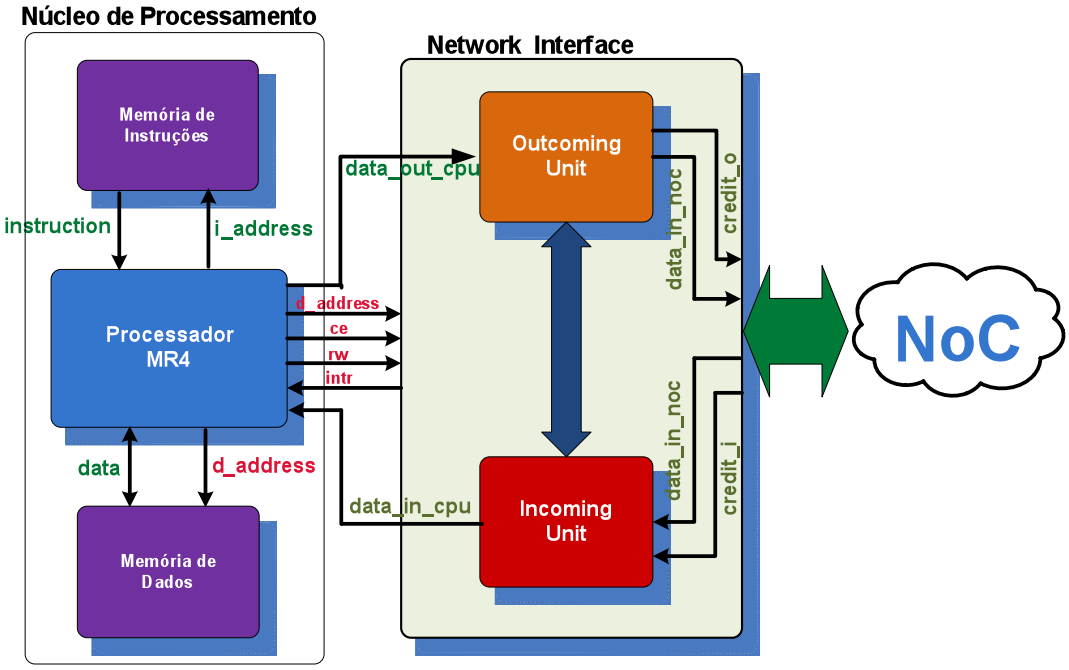

Figura 21 – Diagrama de Blocos do Núcleo de Processamento (NP), formado pelo Processador MR4 e as duas memória e a *Network Interface* (NI) com seus dois *drivers* de comunicação.

Para o desenvolvimento deste MPSoC forma reutilizados módulos já desenvolvidos pelo grupo GAPH, os quais compreendem o processador MR4 e a NoC HERMES. Para o desenvolvimento deste projeto a rede HERMES foi parametrizada utilizando os seguintes parâmetros:

- Controle de Fluxo: *Credit Based*
- Sem canais virtuais
- Dimensão: 3x3
- Largura do *flit*: 32
- Profundidade dos *buffers*: 8
- Algoritmo de roteamento: XY

3.1 Processador MR4

O processador MR4 [MOR08] utilizado neste trabalho foi desenvolvido pelo GAPH e é um dos variantes da família MRx, que tem como características gerais:

- Arquitetura Harvard, *load/store*;
- Dados e endereços de 32 bits;
- Endereço de memória orientado a byte, onde cada palavra ocupa 4 posições consecutivas de memória;
- O banco de registradores possui 32 registradores de uso geral de 32 bits que vão de $0 à $31;
- Formato de instrução regular. Todas as instruções possuem o mesmo tamanho, ocupando uma 1 palavra em memória.

A Figura 22 mostra o diagrama de blocos da estrutura do processador e suas memórias.

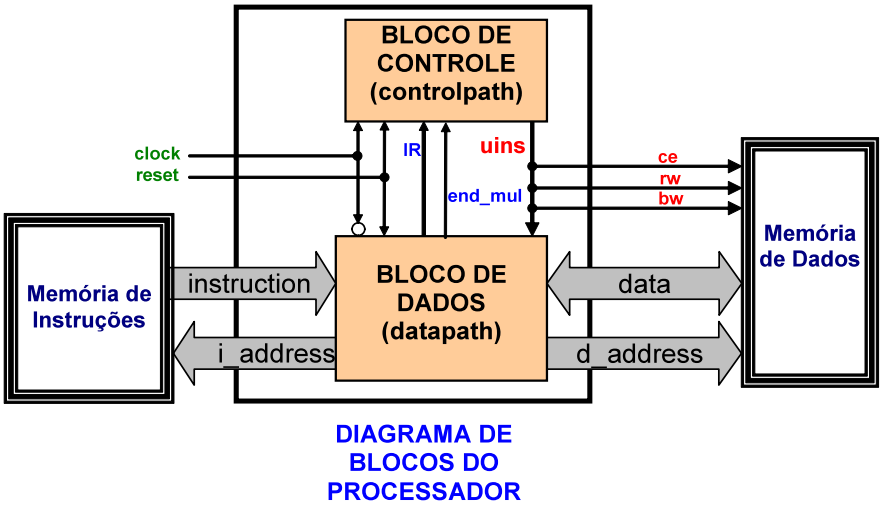

Figura 22 – Diagrama de Blocos do Processador MR4 [MOR08].

Para efetuar a troca de informações entre o processador e as memórias, o MR4 utiliza os seguintes sinais:

- *i_adress*: barramento unidirecional de 30 bits;
- *instruction*: barramento unidirecional de 32 bits;
- *d_adress*: barramento unidirecional de 32 bits;
- *data*: barramento bidirecional de 32 bits.

Na Figura 23 temos a organização do bloco de dados. A execução de qualquer instrução requer de 3 à 5 ciclos,(*i*) busca de instrução, (*ii*) decodificação e leitura dos registradores, (*iii*) operações com a ALU (unidade lógico-aritmética), (*iv*) acesso a memória (*v*) atualização do banco de registradores (*write-back*).

O bloco de dados necessita de 21 sinais de controle, organizados em quatro classes:

- habilitação de escrita em registradores : *wpc*, *CY1*, *wreg*, *CY2*, *walu*, *wmdr*;
- controle de leitura/escrita na memória externa: *ce* , *rw* e *bw*;
- as operações que a unidade lógico-aritmética, executa;
- a seleção da operação do comparador ;
- os controles dos multiplexadores, resultantes da decodificação da instrução .

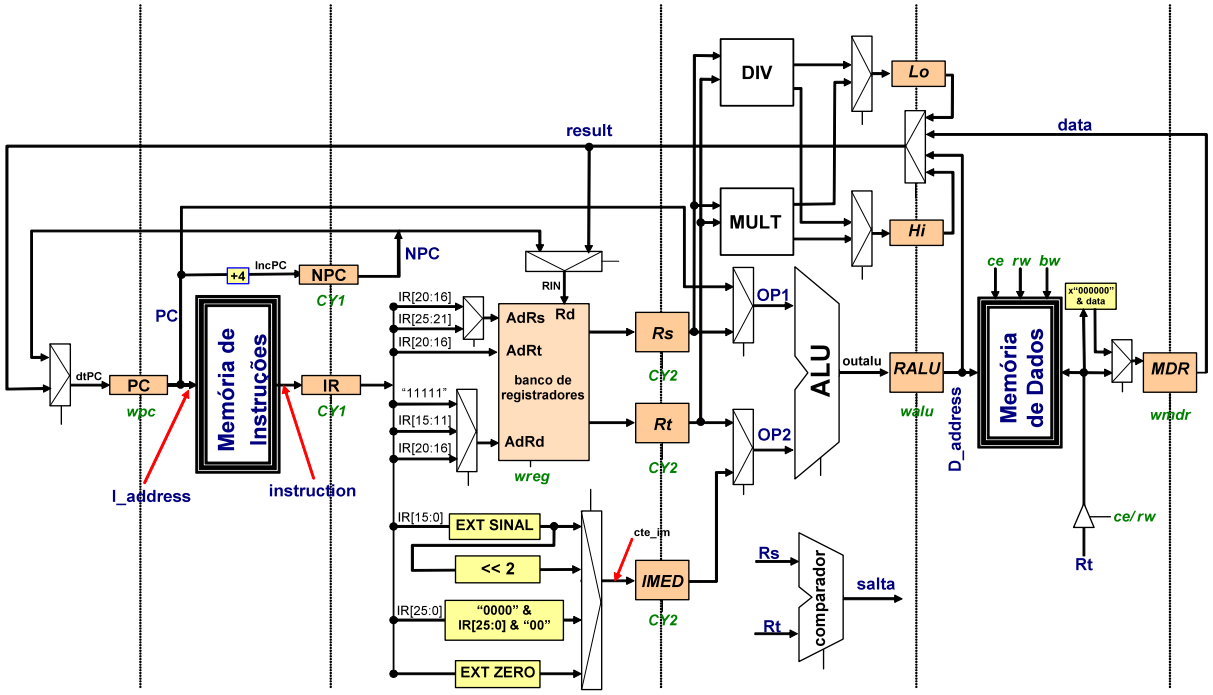

Figura 23 - Bloco de dados completo do MR4 [MOR08].

A execução das instruções é feita através de uma máquina de estados de controle. A Figura 24 mostra a máquina de estados, seus estados e os registradores que são modificados durante sua execução. As funções dos estados são descritas a seguir.

- *Idle*: estado inicial após o reset, serve para garantir que a primeira borda de subida de relógio após este sinal defina o início da operação do processador MRx;
- *Sfetch*: primeiro ciclo, busca de instrução;
- *SReg*: segundo ciclo, leitura dos registradores fonte;
- *SAlu*: terceiro ciclo, operação com a ALU e eventual comparação no comparador (em paralelo);
- *Wbk*: quarto ciclo para a maioria das instruções, onde se escreve o resultado no banco de registradores e atualiza-se o contador de programa (quinto ciclo para as instruções LW e LBU);
- *Sld*: quarto ciclo das instruções LW e LBU, onde se lê um dado da memória de dados externa;
- *Sst*: último ciclo das instruções SW e SB, onde se escreve um dado na memória de dados externa;
- *Ssalta*: último ciclo das instruções de salto condicional ou incondicional, apenas atualiza valor do PC.

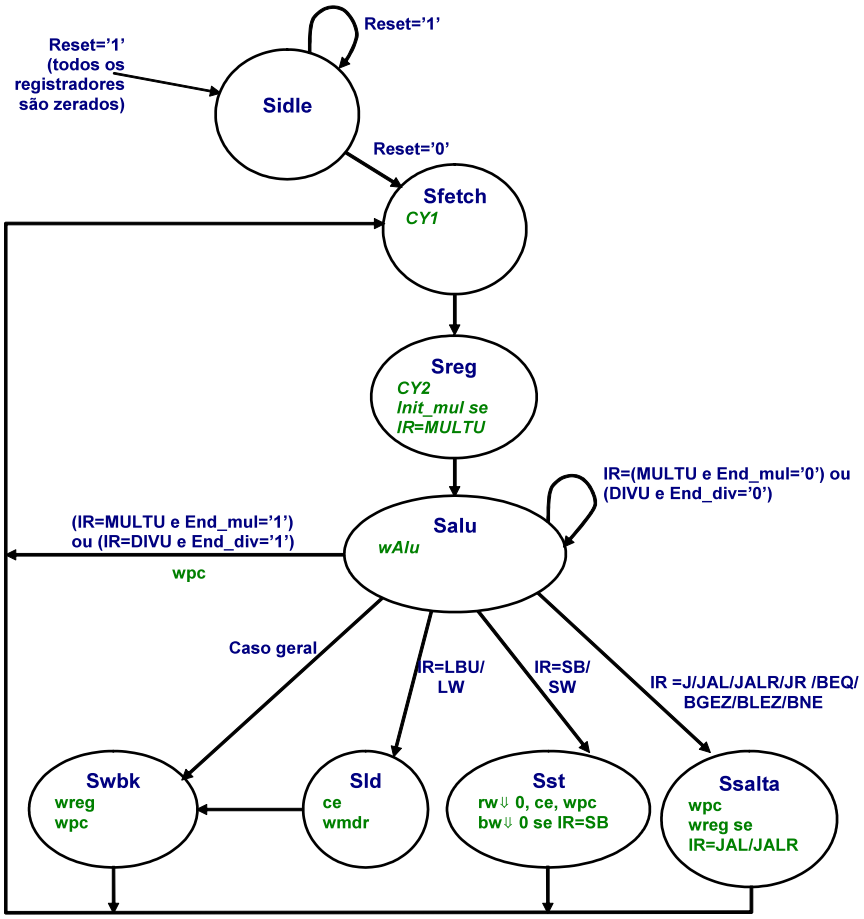

Figura 24 - Máquina de estados de controle para organização MR4 [MOR08].

A principal modificação do processador MR4 para este trabalho inclui o tratamento de interrupção. Esta modificação é descrita na Seção seguinte.

3.2 Inclusão de tratamento de interrupção no processador MR4

A principal justificativa para implementação da interrupção é evitar-se a perda de desempenho na recepção dos dados. Não havendo um mecanismo de interrupção no processador devem-se verificar periodicamente bits de status para a detecção de dado, processo este denominado *polling*. O *polling* é a forma mais simples para um dispositivo de E/S se comunicar com o processador [PAT05].

Neste caso, o principal problema para um processo de *polling* é a sobrecarga do processador, que realiza todo o trabalho de controle de envio e recepção dos dados. Como veremos no Capítulo 5, um dos principais problemas de latência na rede é o tempo gasto pelo processador, verificando se pode enviar dados à NI.

De acordo com [PAT05] há duas formas de endereçar os dispositivos de E/S, mapeamento em memória e instruções de E/S especiais. O método utilizado neste caso foi implementar a interrupção através de mapeamento em memória.

As principais mudanças na arquitetura do MR4 foram: (*i*) a inclusão de novas instruções para o tratamento da interrupção; (*ii)* a inclusão de um novo estado na máquina de estados de controle; (*iii*) novos registradores. Estas modificações, apresentadas na Figura 25, serão detalhadas nas próximas Seções.

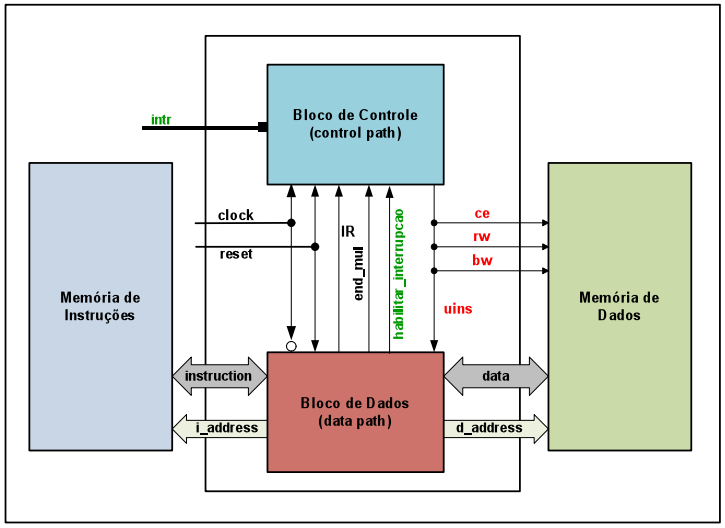

Figura 25 – Diagrama de bloco do processador MR4 após as modificações. Foram incluídos os sinais *intr* e *habilitar_interrupcao*.

No módulo *datapath* foram adicionados um novo pino de saída, *habilita_interrupção* e um novo registrador EPC. Dentro do módulo *control_unit* foram incluídos dois pinos de entrada, *habilitar_interrupcao* e *intr*, um novo estado, *Sint* e duas novas instruções, *mfc0* e *mtc0*. No módulo *mrstd* (descrição de mais alto nível na hierarquia do processador), foi adicionado o pino de entrada *intr*. Além dessas implementações foi adicionado um novo controlador de escrita em registrador, a microoperação *lepc*. Na Figura 26 apresenta-se o bloco de dados completo com as inclusões realizadas durante o projeto.

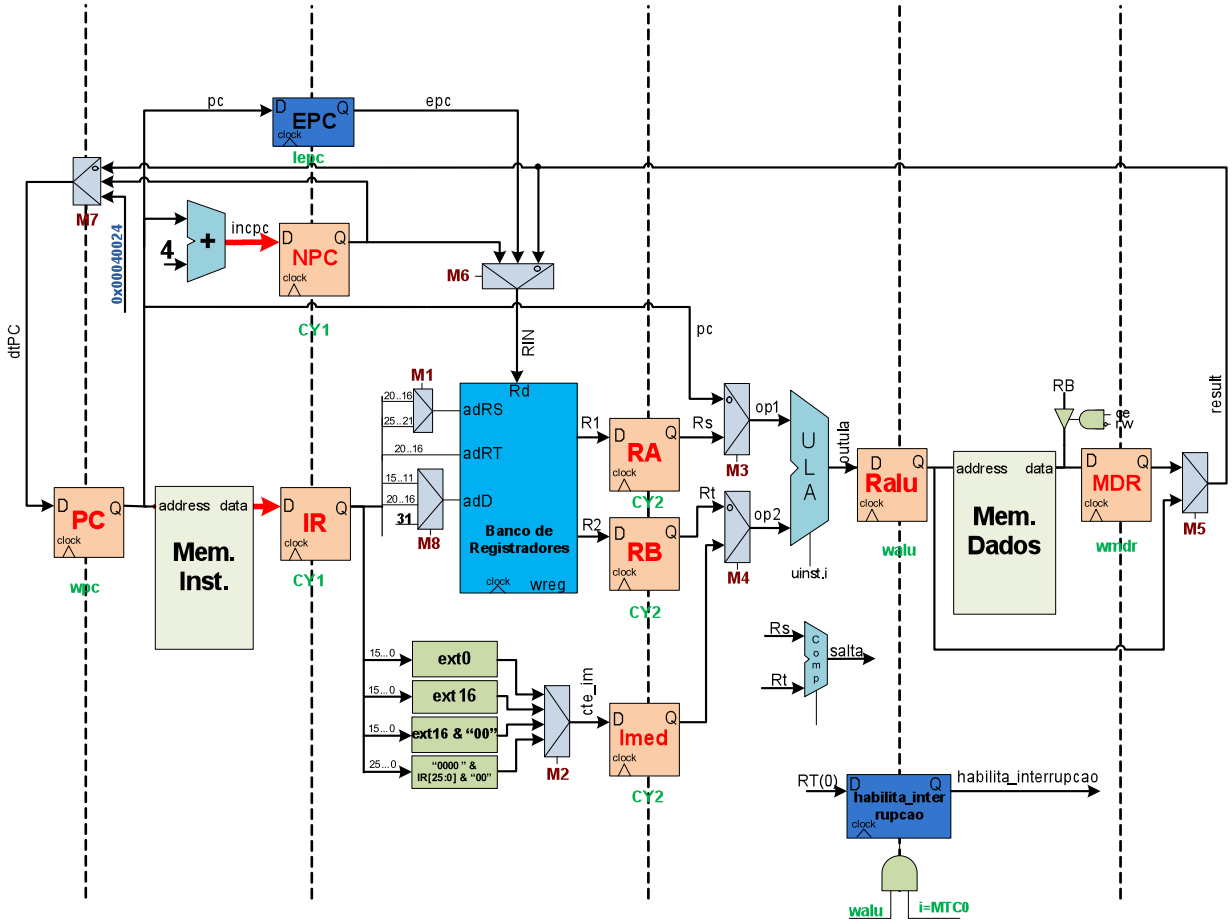

Figura 26 – Bloco de dados completo. Os blocos em azul representam as alterações realizadas com relação ao bloco de dados original.

O sinal *habilita_interrupção* foi criado para indicar quando o estado da interrupção foi alterado. Por *default* a interrupção inicia sempre ativa. O processo de habilitar/desabilitar deste sinal é realizado através da execução da instrução *mtc0*. Ao se iniciar o tratamento de uma interrupção, deve-se desabilitar o tratamento de outras interrupções. No caso de ocorrer uma requisição de interrupção durante o atendimento de uma dada interrupção, o processador não será capaz de recuperar o contexto de suas operações, dado que não há mecanismo de empilhamento de chamadas de interrupção. Com isso foi necessário fazer o mascaramento pelo próprio processador. Para realizar tal mascaramento foi criado o *flip-flop em_atendimento* dentro da entidade *control_unit* e um registrador EPC, que armazena o endereço da última instrução que estava sendo executada quando ocorreu a interrupção. O processador, ao iniciar a execução de um programa coloca o sinal interno *em_atendimento* em zero. Cada vez que o processador entrar no ciclo de busca de instrução é feito o teste:

INTR='1' and em_atendimento='0'

Se este teste for verdadeiro, passa-se para um novo estado na máquina de controle, denominado *SInt,* e realizam-se as seguintes funções:

1) O sinal *em_atendimento* é habilitado em '1';

2) Salva-se o *PC* no registrador especial *EPC*, havendo um salto para a posição de memória

0x00400024 (na prática este é o endereço 4, dado que o espaço de instruções inicia no simulador SPIM, utilizado no presente trabalho, em 0x400020);

3) Volta-se para o ciclo de busca de instrução, para a instrução contida no endereço 0x00400024.

Para esta nova implementação foi necessário realizar alterações na máquina de controle, onde foi adicionado um novo estado chamado *Sint*, controles de novos sinais e novos registradores. A Figura 27 mostra as alterações realizadas. A alteração do estado *Sfetch* para o estado *Sint* ocorre quando houver uma solicitação externa de interrupção (*intr='1'*), e não houver interrupção sendo atendida (*em_atendimento='0'*) e a interrupção estiver habilitada (*habilitar_interrupcao='1'*).

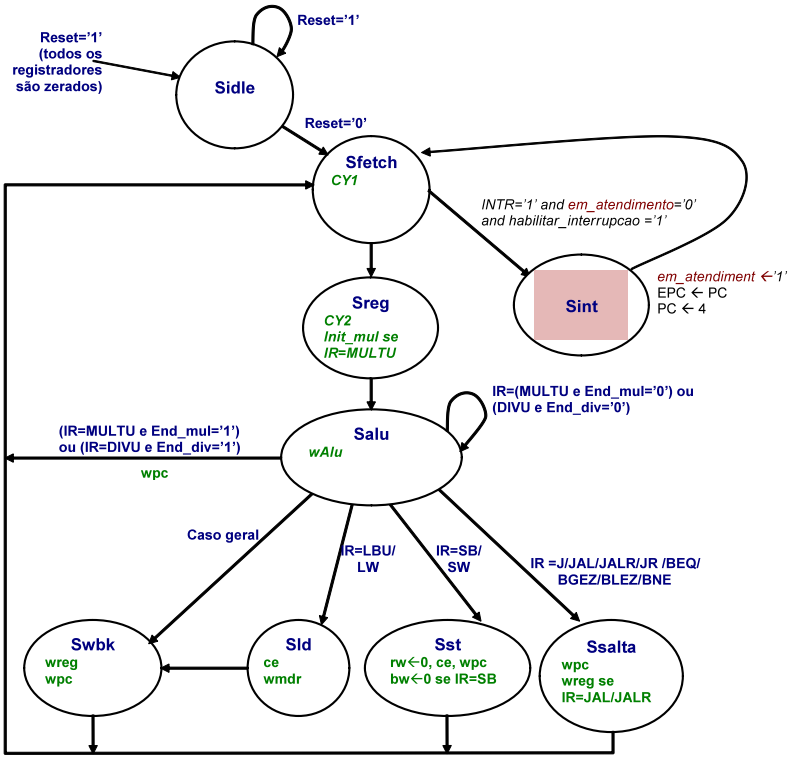

Figura 27 – Máquina de estado de controle do MR4. Em destaque o estado adicional (Sint) para atendimento de interrupção e o controle de novos sinais e registradores.

Além das alterações na máquina estados, foi necessário implementar a decodificação de novas instruções para o tratamento da interrupção, com o formato das mesmas detalhado na Tabela 2:

- *mtc0*: habilita ou não a interrupção. O valor contido no registrador fonte (Rd) é copiado para um registrador denominado *habilitar_interrupcao* de 32 bits;
- *mfc0*: copia o valor do registrador *EPC* para um registrador de uso geral.

Tabela 2 – Formato das novas instruções.

Instrução		Formato da Instrução					Ação
		31 - 26	25 -21	20 -16	15 - 11	10 - 0	
MTC0	*mtc0 Rd, Rt*	10	4	Rt	Rd	0	Rt←Rd
MFC0	*mfc0 Rt, Rd*	10	0	Rt	Rd	0	Rd←Rt

3.3 Elemento de Processamento

Para simplificar a descrição VHDL do sistema, foi criado o módulo elemento de processamento (*elemento_pe.vhdl)* ou PE, o qual possui a entidade do processador MR4 (*MRStd*) , a entidade da memória de dados (*Data_mem*) e a entidade da memória de instruções (*Instr_mem*) . A Figura 28 apresenta as interfaces de cada elemento interno do módulo *elemento_pe*, assim como os sinais para a comunicação com a interface de rede. A conexão do *elemento_pe* com a interface de rede compreende os seguintes sinais:

- ce_perif: quando ativado, indica que o processador fará um operação com a interface de rede;
- readwrite: indica se a operação é de escrita ou leitura;
- data_in_ext: envia os dados para NoC;
- data_out_ext: recebe os dados vindos da NoC;
- address: endereço de memória;
- cont_tick: valor de contagem de pulsos de relógio desde a inicialização do sistema, sendo utilizado para determinar tempo de execução e cálculo de latências de pacotes;
- intr: interrupção, indica à CPU que a NI tem dados a enviar.

Além das alterações para a inclusão das entidades dentro do núcleo de processamento, foi necessário modificar o tamanho da memória dentro do módulo *elemento_pe*. No projeto inicial a memória de dados possuía um tamanho de 2048 palavras. Devido à aplicação utilizada na validação, ordenação *merge sort*, o tamanho foi alterado para 20480 palavras. O registrador $SP (*stack pointer*) é também ajustado de acordo com o tamanho de memória.

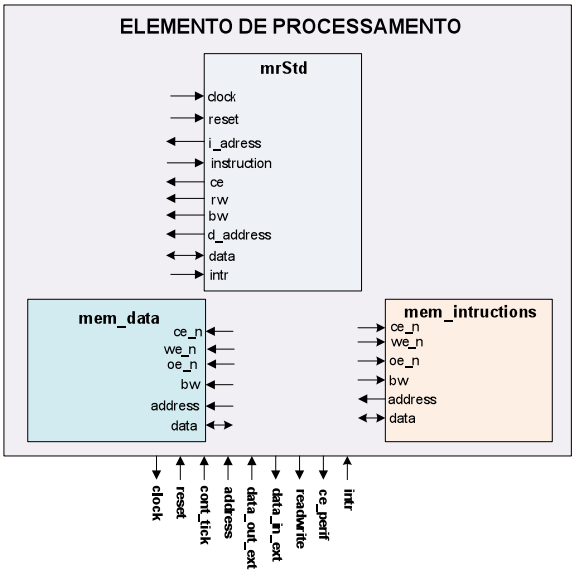

Figura 28 - Bloco do elemento processador.

O código objeto e os dados iniciais da aplicação que serão executados no processador são carregados no início da execução do sistema. Para gerar o código objeto foi utilizado

simulador/montador PCSpim [SPI08].

3.4 Interface de Rede

Para o desenvolvimento da interface de rede (NI, do inglês, *Network Interface*) foi elaborado primeiramente um módulo de transmissão de dados, denominada *outcoming unit* e um módulo de recepção de dados denominada *incoming unit*. Posteriormente, as interfaces são integradas, formando assim a *Network Interface*.

3.4.1 Interface de Transmissão - *Outcoming Unit*

A *outcoming unit* tem por objetivo enviar pacotes oriundos do processador para a NoC. A *outcoming unit* possui um *buffer* (*buf_ni_cpu*) de até 1024 posições (parametrizável em tempo de projeto). O processo de envio é feito em 3 etapas:

1. O processador começa a escrever os dados do pacote no *buffer* quando a faixa de endereçamento for fornecida entre NI_TX_BASE_ADDRESS e NI_TX_HIGH_ADDRESS.
2. O processador avisa a *outcoming unit* que terminou de escrever o pacote no *buffer*, ativando o sinal *end_pack.* Este sinal é ativado através de escrita no endereço NI_TX_END_PACKET e quando os sinais ce='1' , rw='0' e do bit LSB do sinal *data_in_cpu* = '1'.
3. Uma vez sinalizado o final de escrita do pacote cada *palavra* é enviada à NoC. Através do sinal *tx* é informado à NoC que há dados a enviar.

```
tx <= '1' when EA=Stransmiting else '0';
data_out_noc <= buffer(contador_flits);
```

A Figura 29 apresenta a máquina de estados de controle da *outcoming unit.*

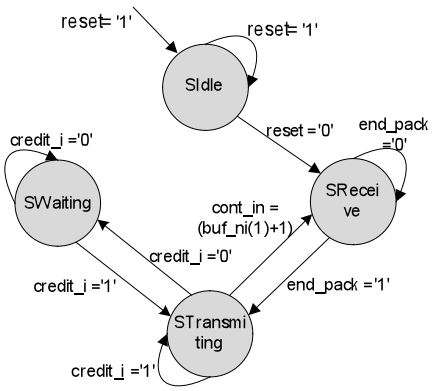

Figura 29 - Máquina de estados da interface *outcoming unit*.

Os estados desta máquina correspondem às seguintes funções:

- SIdle. Estado Inicial. A máquina estados inicia e permanece neste estado enquanto o reset estiver ativado em '1', se não avança para o próximo estado.
- *SReceive*. A máquina de estados permanece neste estado enquanto o sinal de *end_pack* estiver desativado. A máquina de estados avança para o estado *STransmiting* quando o sinal

end_pack for ativado.

- *STransmiting*. Neste estado ativa-se o sinal *tx* e incrementa-se o *contador_flits*. A máquina de estados permanece neste estado enquanto o sinal de crédito *credit_i* estiver ativo, indicando que pode ser enviado o pacote. A máquina de estados avança para o estado *SWaiting* quando o sinal de *credit_i* estiver desativado. O sinal *credit_i* é gerado pelo controle de fluxo da NoC. Ao terminar de enviar para a NoC o pacote (controle pelo número de *flits* transmitidos), a máquina retorna para o estado *SReceive*.
- *SWaiting*: A máquina de estados permanece neste estado enquanto o sinal de *credit_i* estiver desativado e avança para o próximo estado quando *credit_i* for ativado. Neste estado o sinal *tx* é desativado.

Durante os estados *STransmiting* e *SWaiting* um sinal denominado *busy* é ativado. O processador deve ler o sinal *busy* antes de cada escrita no *buffer* para evitar que dados que estão sendo enviados sejam corrompidos por um novo pacote.

3.4.2 Interface de Recepção - *Incoming Unit*

A *incoming unit* tem por objetivo receber pacotes da NoC, e enviá-los para o processador. Esta interface é similar à anterior, possuindo também um *buffer* (*buf_ni_noc*) de até 1024 posições (parametrizável em tempo de projeto). O pacote oriundo da NoC é escrito neste *buffer*. Ao final da escrita do pacote ativa-se um registrador indicando pacote no *buffer*. O processador ao detectar este sinal ativo, realiza a leitura do *buffer*.

A Figura 30 apresenta a máquina de estados da *incoming unit*.

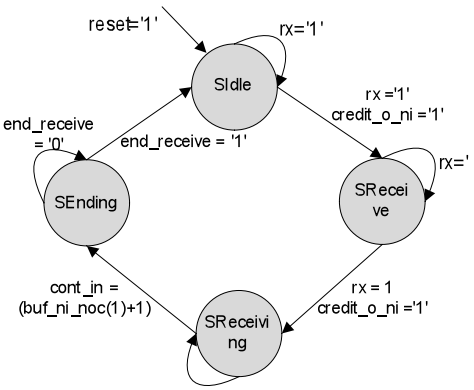

Figura 30 – Máquina de estados da interface *incoming unit*.

Os estados desta máquina correspondem às seguintes funções:

- *SIdle*. Estado inicial. Quando *rx* (aviso da NoC que há novo pacote) e *credit_o_ni* (interface de rede pode receber dados) estiverem ativos, a máquina de estado avança para o estado *SReceive*.
- *SReceive*: Quando o estado atual estiver em *SReceive* e os sinais *rx* e *credit_o_ni* forem ativados, a máquina de estado avança para o próximo estado *SReceiving*. Neste estado é

zerado o contador *cont_in*, que controla a ocupação do *buffer_ni_noc*.

- *SReceiving*: O *buffer* realiza a recepção do pacote. O contador *cont_in* é incrementado neste estado, desde que *rx* seja igual a '1'. Quando todos os *flits* forem recebidos, avança-se para o estado *SEnding*.
- *SEnding*: Este estado é responsável por ativar o sinal *ready_pack*, que indica ao processador que há pacote para receber (ativa o sinal de interrupção *intr*). O processador ao escrever no endereço de *end_receive* indica à esta máquina que o pacote foi totalmente lido, permitindo a volta ao estado *SIdle* e a conseqüente recepção de um novo pacote.

Para que a NI pudesse armazenar todos os *flits* enviados pelo CPU e consequentemente da NoC, foi necessário criar dois *buffers*, um de recepção da CPU e outro para a recepção da NoC com tamanhos iguais de 1024 posições limitando o tamanho do pacote a 1022 palavras de *payload* (lembrando, dois *flits* são reservados para o *header*).

3.4.3 Interface de Rede (NI)

A NI é responsável por integrar as unidades *outcoming unit* e *incoming unit*. A NI realiza a interface entre o processador MR4 e a NoC HERMES e tem por funções: (1) enviar os pacotes para a NoC HERMES, sendo enviado pelo processador, recebido pela NI e depois enviado para a NoC; (2) receber os pacotes da NoC HERMES, sendo recebido pela NI e depois enviado para o processador.

A Figura 31 apresenta a interface externa da NI. Os sinais de interface com a CPU compreendem: (*i*) *ce*: quando ativado indica se está em curso uma operação com a memória de dados da CPU; (*ii*) *rw*: quando ativado indica que está ocorrendo uma operação de leitura e quando desativado uma operação escrita; (*iii*) d_*address*: indica o endereço da memória para leitura ou escrita de dados; (*iv*) *data_in_cpu*: dado recebido da CPU; (*v*) *data_out_cpu*: dado enviado para a CPU; (v*i*) *intr*: sinal de interrupção que vai para a CPU.

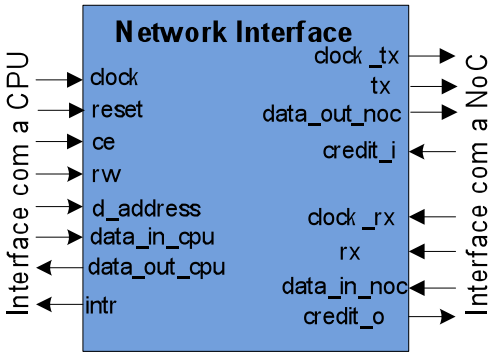

Figura 31 – Network Interface.

Os sinais de interface com a NoC compreendem: (*i*) *clock_tx*: clock para sincronizar o envio de dados; (*ii*) *tx*: informa à NoC que tem dado a enviar; (*iii*) *data_out_noc*: dado a ser enviado à NoC; (*iv*) *credit_i*: indica se a NoC pode receber dados; (v) *clock_rx*: clock para sincronizar a recepção de dados; (*vi*) *rx*: indica se tem dado a receber da NoC; (*vii*) *data_in_noc*: dado recebido

da NoC; (*viii*) *credit_o*: informa à NoC se pode receber dados.

4 INFRA-ESTRUTURA DE SOFTWARE

Neste capítulo é descrito os ambientes para o desenvolvimento deste trabalho e os métodos para a comunicação entre os núcleos de processamento (*elemento_pe*) e a NoC HERMES. Para isso foram desenvolvidos os *drivers* de comunicação e a estrutura que trata das chamadas de interrupção. Também é descrito as primeiras formas de envio de pacotes para teste de validação assim como o desenvolvimento de uma aplicação paralela para avaliação de desempenho da arquitetura.

4.1 Ambientes de Desenvolvimento

4.1.1 Atlas

Atlas [ATL08] é um ambiente para geração e avaliação de NoCs. O projeto de NoC requer novas características, tais como suporte a modos de chaveamento, algoritmos de roteamento e tipos de escalonamento. O grupo GAPH desenvolveu o ambiente ATLAS para automatização dos processos que englobam os projetos de NoCs: geração de rede, geração de tráfego, simulação e avaliação de desempenho.

A geração da NoC permite as seguintes parametrizações: topologia, algoritmo de roteamento, largura dos canais de comunicação, profundidade dos *buffers*, número de canais virtuais, estratégias de controle de fluxo. A partir desta configuração é gerada a rede intra-chip que deve respeitar os requisitos de determinada aplicação. Na geração de tráfego, os cenários de tráfego são gerados para caracterizar aplicações que executarão sobre a rede. Na simulação, os dados de tráfego são injetados na rede. Na etapa de avaliação de desempenho é possível gerar gráficos, tabelas, mapas e relatórios para auxiliar na análise dos resultados obtidos.

A Figura 32 ilustra a interface principal do ambiente ATLAS, a qual permite invocar as ferramentas que compõem as etapas do fluxo de projeto.

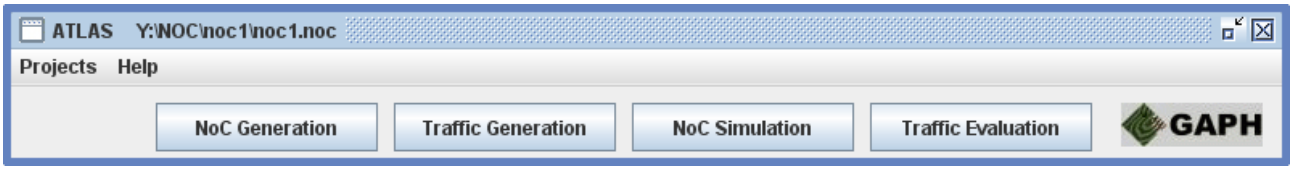

Figura 32 - Interface principal do ambiente Atlas.

As principais ferramentas do ambiente ATLAS estão descritos abaixo com suas respectivas funções:

- NoC Generation – *Maia*: automatização do processo de interconexão dos roteadores.
- Traffic Generation – *Traffic Mpbs*: gera os arquivos de tráfico que serão transmitidos através da rede durante a simulação.
- NoC Simulation: verifica o correto funcionamento da rede, através do simulador ModelSim.

- Traffic Evaluation – *Traffic Measurer*: tem o objetivo de facilitar a análise dos resultados gerados durante a simulação da rede.

4.1.2 PSpim e Mars

Para auxiliar no desenvolvimento, foram utilizados dois simuladores para linguagem de montagem, o PCSpim (Figura 33) para gerar o código objeto e o MARS (Figura 34) para simulação do código. Inicialmente o código é escrito em linguagem de montagem e depois gerado o código objeto final pelo PCSpim, logo após é realizada a adequação do código para executar no processador MR4, visto que é necessário a eliminação de algumas linhas de código para tal adequação. As simulações do código *assembly* foram realizadas no MARS por ter uma estrutura de identificação de erro e um ambiente mais amigável. A Figura 33 apresenta o simulador utilizado para gerar o código objeto.

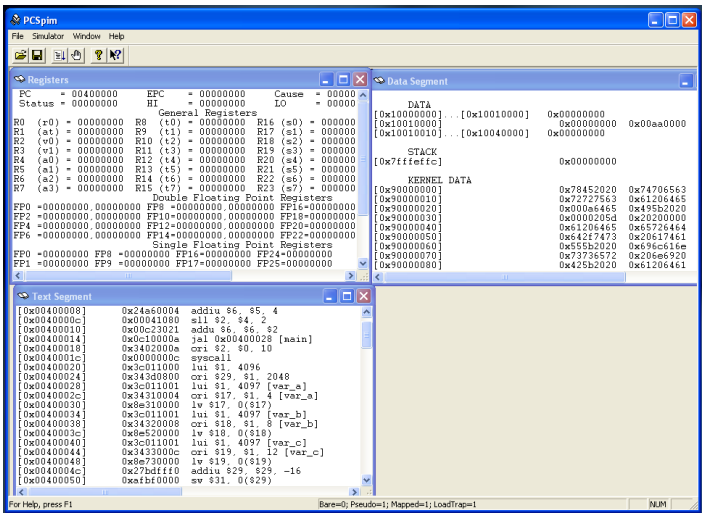

Figura 33 – Ambiente do simulador PCSpim.

O simulador MARS poderia ser também utilizado para gerar o código objeto, mas seria necessário modificar o endereço de inicio de leitura do arquivo objeto, sendo necessário modificar a estrutura vhdl do *mrstd.vhd*. Na Figura abaixo é mostrado ao ambiente do simulador MARS.

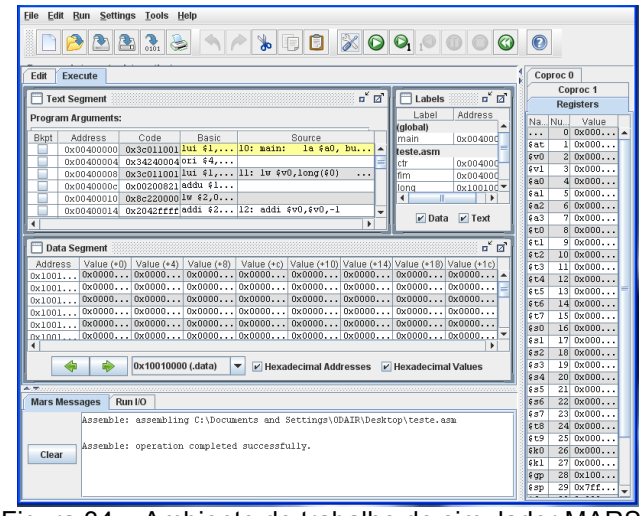

Figura 34 – Ambiente de trabalho do simulador MARS.

4.1.3 Modelsim

Para validar a implementação do MPSoC desenvolvido utilizou-se o simulador Modelsim. Este simulador possibilitou desenvolver os módulos de interface de rede, assim como o núcleo de processamento. Além da inclusão de novos módulos, permitiu-nos realizar toda a parte de simulação e *debug* do projeto.

Todos os gráficos e formas de onda foram gerados através do simulador Modelsim, assim como o acesso à memória em tempo de simulação deu-nos uma visão ampla do conteúdo de cada registrador criado em memória. Na Figura 35 é ilustrado o ambiente que permitiu toda a parte de desenvolvimento de nosso projeto.

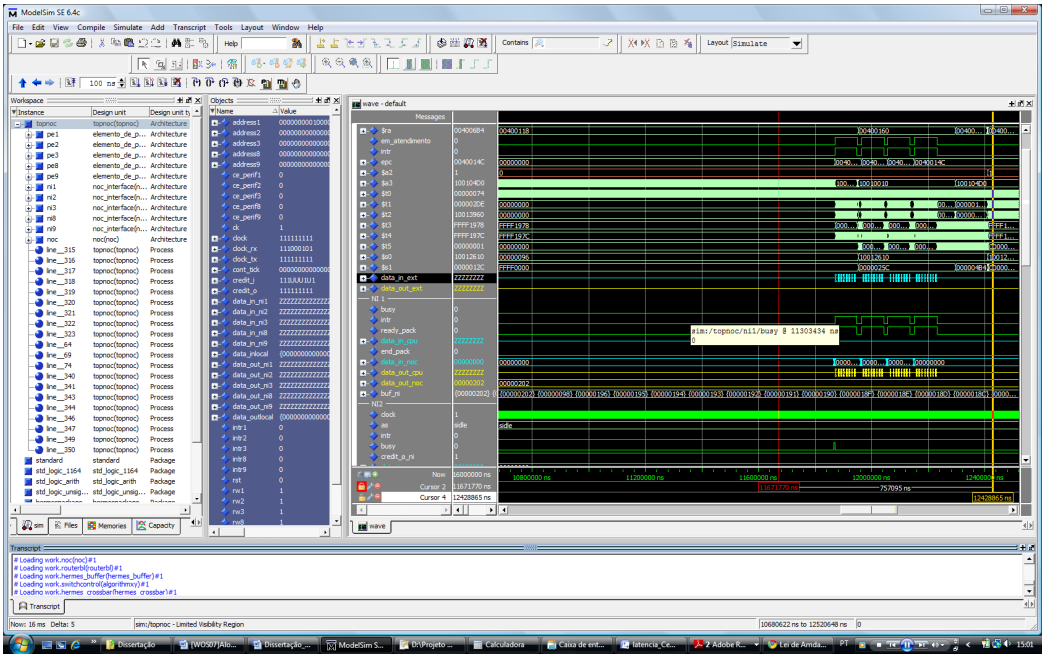

Figura 35 – Ambiente de desenvolvimento Modelsim.

4.2 Tratamento de Interrupção

De acordo com o que foi mencionado no Capítulo 3, quando ocorre uma interrupção o registrador EPC armazena PC + 4, ou seja, o endereço da próxima instrução e PC recebe o endereço da rotina de tratamento de interrupção, denominada *interrupt_handler*. O endereço armazenado em EPC é novamente carregado quando a rotina *interrupt_handler* é finalizada. Os endereços de entrada e saída mapeados em memória utilizados pela *interrupt_handler* compreendem:

- NI_RX_BASE_ADDRESS: registrador que contém o endereço da interface de rede.
- PACKET_BASE_ADDRESS: registrador que retorna o endereço em memória de pacote válido, ou seja, de pacote que já foi recebido e transferido para a memória de dados.
- NI_RX_END_RECEPTION: registrador que informa o fim de recepção do pacote.

Para a implementação da rotina *interrupt_handler* foram criadas quatro regiões em memória para armazenar os pacotes recebidos, endereçadas por PACKET_BASE_ADDRESS. A

rotina *interrupt_handler* é executada no momento em que ocorre o salto para o endereço 0x40000024. Depois de salvar o contexto de todos os registradores executa-se um laço (Figura 36), que busca o endereço de uma das quatro regiões criadas para armazenar o novo pacote. Havendo área de memória disponível, pode-se iniciar a leitura dos dados da NI (observar a linha 5), a partir do endereço armazenado no registrador $t1. Se não houver espaço para armazenar um novo pacote, a rotina desabilita a interrupção através da instrução da chamada à instrução *mtc0* e depois restaura o contexto dos registradores. A interrupção é posteriormente habilitada quando do consumo de um pacote.

```
 1 find_packet:  la $t1,PACKET_BASE_ADDRESS
 2               addu $t1, $t1, $t0              # endereço do ponteiro do pacote
 3               lw $t1, 0($t1)                  # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote
 4               lw $t2, 0($t1)                  # carrega a primeira palavra do pacote
 5               beq $t2,$zero, read_packet
 6               addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes
 7
 8               li $t5, 0x10                    ## se o contador chegou a 0x10 é sinal que os 4 espaços para
 9               bne $t0, $t5, find_packet       ## pacotes estão ocupados
10               mtc0 $zero, $12                 ## LOGO DESABILITA AS INTERRUPÇÕES E RECUPERA O CONTEXTO
11               j loop5
```

Figura 36 – Código para encontrar área de dados livre para armazenar pacote recebido.

No momento em que é encontrado um local para armazenar o pacote, executa-se o código a partir do rótulo *read_packet* (Figura 37), ou seja, começa-se a receber os dados da NI. Depois de recebido todos os dados da NI, é enviado um sinal de fim de recepção dos dados através de NI_RX_END_RECEPTION e restaurado o contexto dos registradores. Em seguida depois da finalização da rotina de interrupção salta-se para o endereço armazenado no registrador EPC através da chamada à instrução *mfc0*.

```
read_packet:    la $t2,NI_RX_BASE_ADDRESS       # endereço do periférico (INTERFACE DE REDE)
                addiu $t2,$t2,4                 # posiciona no segundo endereço - o primeiro flit é o endereço
                lw $t3, 0($t2)                  # TAMANHO do payload + 2 flits do cont_tick

                sw $t3, 0($t1)                  # marca  pacote como ocupado
                xor $t0, $t0, $t0               #i=0

loop2:          addiu $t0,$t0,1                 # "i++"
                addiu $t2,$t2,4                 # posiciona o $s0 para o payload (3o, 4o, 5o flits)
                lw $t4, 0($t2)                  # carrega o valor apontado por $t2  (LE DA INTERFACE DE REDE)
                sw $t4, 4($t1)                  # armazeno O valor do flit recebido (GRAVA NA MEM. DE DADOS)
                addiu $t1,$t1,4                 # avança na memória de dados
                bne $t0,$t3,loop2               # se não recebeu todo o pacote, volta a ler os flits

                la $t0, NI_RX_END_RECEPTION     # endereço do registrador que indica fim do recebimento do pacote
                li $t1, 1
                sw $t1, 0($t0)                  # avisa ao hw final de recebimento de pacote

loop5:          lw $ra,0($sp)
                lw $t0,4($sp)
                lw $t1,8($sp)
                lw $t2,12($sp)
                lw $t3,16($sp)
                lw $t4,20($sp)
                lw $t5,24($sp)
                lw $1, 28($sp)
                addiu $sp,$sp,32                ## recuperação do contexto

                mfc0 $a0,$14                    ## recuperação do  EPC ($14),e retorno para a aplicação
                jr $a0
```

Figura 37 – Código para armazenar os *flits* recebidos da NI.

4.3 Drivers de Comunicação

Os *drivers* de comunicação são responsáveis por gerenciar a recepção e o envio de pacotes.

Foram criados dois *drivers*, um de transmissão de pacotes e outro de recepção de pacotes. A estrutura do pacote compreende (Figura 38): (*i*) primeiro *flit* - endereço do NP destino; (*ii*) segundo *flit* - tamanho do *payload*; (*iii*) demais *flits* - *payload*.

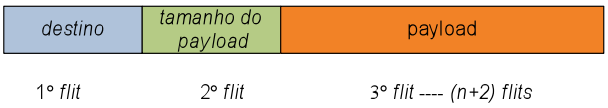

Figura 38 – Formato do pacote enviado pelo processador.

Os objetivos dos *drivers* são montar e desmontar os pacotes, e o envio e o recebimento de pacotes. Para que isso ocorra, são utilizado endereços de entrada/saída mapeados em memória.

4.3.1 Driver de Envio

O *driver* de envio denominado como *outcoming_unit* possui três registradores mapeados em memória. Os endereços de saída mapeados em memória utilizados pelo *driver* de envio compreendem: (*i*) NI_TX_BASE_ADDRESS: registrador que contém o endereço da *outcoming unit*; (*ii*) NI_TX_BUSY: registrador que indica se a NI está ocupada; (*iii*)NI_TX_END_PACKET: registrador que informa o fim de transmissão do pacote.

```
#SALVA  CONTEXTO
SEND:
        addiu $sp,$sp,-12
        sw $ra,0($sp)
        sw $s2,4($sp)
        sw $t1,8($sp)

#CARREGA ENDEREÇO DOS REGISTRADORES EM MEMORIA
        la $s1, NI_TX_BASE_ADDRESS
        la $t3, NI_TX_BUSY
        la $t4, NI_TX_END_PACKET


# VERIFICA SE A NI ESTÁ LIVRE(NI_TX_BUSY)
LOOP: lw $t0, 0($t3)
        bne $t0, $zero, LOOP

# LAÇO PARA TRANSMISSÃO DO PACOTE ORIGINAL SEM A BASE DE TEMPO
LOOP2: addu $s3, $s1, $
        addu $s4, $s2, $t0

        lw $t1, 0($s4
        sw $t1, 0($s3)

        addiu $t0,$t0, 4
        addiu $s0,$s0,-1

        bne $s0, $zero, LOOP2

# ENVIA ESTADO DE NI_TX_END_PACKET PARA INDICAR FIM TRANSMISSÃO DO PACOTE
        li $s0, 1
        sw $s0, 0($t4)

#RESTAURA CONTEXTO
        lw $ra,0($sp)
        lw $s2,4($sp)
        lw $t1,8($sp)
        addiu $sp,$sp,12

        jr $ra
        sw $t0,4($sp)
```

Figura 39 – Código simplificado da *outcoming unit*.

O código simplificado em linguagem de montagem para a transmissão de um pacote (*driver* de envio) de *n flits* é apresentado na Figura 39. O código compreende em salvar o contexto, dois laços principais e recuperação do contexto: (*i*) verificação se a NI está livre, através da leitura do

sinal *busy*, mapeado no endereço NI_TX_BUSY; (*ii*) envio do pacote através da leitura da área de dados e escrita no *buffer*, no endereço NI_TX_BASE_ADDRESS. Ao final do envio do pacote avisa-se à NI que há pacote disponível, escrevendo-se no endereço NI_TX_END_PACKET.

4.3.2 Driver de Recepção

O *driver* de recepção denominado como *incoming_unit* possui somente um registrador mapeado em memória. O PACKET_BASE_ADDRESS é o registrador que retorna o endereço do pacote válido. Este *driver* baseia-se basicamente na rotina bloqueante chamada *get_packet_address*.

O código em linguagem de montagem para a recepção de um pacote (*driver* de recepção) de *n flits* é apresentado na Figura 40. Este código compreende dois laços principais: (*i*) carrega o endereço do ponteiro do pacote e carrega o conteúdo do ponteiro, ou seja, o endereço do pacote mapeado no endereço PACKET_BASE_ADDRESS; (*ii*) restaura contexto.

```
get_packet_adress:
                addiu $sp,$sp,-12
                sw $ra,0($sp)
                sw $t0,4($sp)
                sw $t1,8($sp)

                # LAÇO BLOQUEANTE, ATÉ ENCONTRAR UMA ÁREA DE MEMÓRIA (SÃO 4) COM PACOTE
                xor $t0, $t0, $t0
10:             la $v0,PACKET_BASE_ADDRESS
                addu $v0, $v0, $t0              # endereço do ponteiro do pacote
                lw $v0, 0($v0)                  # carrega o conteúdo do ponteiro, i.e, o endereço do pacote em $v0
                lw $t1, 0($v0)                  # carrega a primeira palavra do pacote
                bne $t1,$zero, l1
                addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes
                andi  $t0, $t0, 0xF
                j 10

l1:             lw $ra,0($sp)
                lw $t0,4($sp)
                lw $t1,8($sp)

                addiu $sp,$sp,12
                jr $ra
```

Figura 40 - Código da *incoming unit*.

4.3.3 Contagem de tempo

Para auxiliar na validação do sistema foi criado um contador de *ticks* (*ciclos de clock*). Os endereços mapeados em memória utilizados para computar o tempo transcorrido desde a inicialização do sistema incluem:

- CONT_TICK_H: registrador que contém a parte alta de contagem dos *ciclos de clock*.
- CONT_TICK_L: registrador que contém a parte baixa de contagem dos *ciclos de clock*.

Para o cálculo da latência, o *driver* de envio lê o tempo atual, armazenando-o no final do pacote. O *driver* de recepção por sua vez subtrai o tempo atual do tempo armazenado no final do pacote recebido, computando assim a latência total do pacote.

As alterações ainda incluem a colocação de mais um pino de entrada (*cont_tick*) para realizar a leitura dos *ciclos de clock* no *elemento_pe.vhdl* e repassar os dados para a CPU através de *data_cpu*.

4.3.4 Impressão dos dados

Para verificar se os dados estavam sendo enviados ou recebidos corretamente pelos *drivers* de envio e recepção, criou-se uma rotina para a impressão dos dados para um arquivo texto. O código simplificado pode ser visualizado na Figura 41. Para realizar esta impressão utilizou-se um registrador mapeado em memória, conforme descrito abaixo:

- VET_TEMP: contém os dados que serão impressos via debug.
- DEBUG: registrador que realiza o *printf* dos dados recebidos, para auxilio da validação do sistema.

```
DEBUG:
#SALVA CONTEXTO
        addiu $sp,$sp,-24
        sw  $ra,0($sp)
        sw  $t0,4($sp)
        sw  $t1,8($sp)
        sw  $t2,12($sp)
        sw  $t3,16($sp)
        sw  $s0,20($sp)

#CARREGA ENDEREÇO DOS REGISTRADORES
        la  $s0, DEBUG
        la  $t3, VET_TEMP

        lw  $t1, 0($t3)
        addiu  $t1,$t1,1
        li  $t0, 0

#IMPRESSÃO DOS DADOS
DEB:
       lw  $t2,0($t3)
       sw  $t2,0($s0)
       addiu  $t3,$t3,4
       addiu  $t0,$t0,1
       bne  $t0,$t1,DEB

#RESTAURA CONTEXTO
       lw  $ra,0($sp)
       lw  $t0,4($sp)
       lw  $t1,8($sp)
       lw  $t2,12($sp)
       lw  $t3,16($sp)
       lw  $s0,20($sp)
       addiu  $sp,$sp,24

       jr $ra
```

Figura 41 – Código de implementação da impressão dos *flits*.

A rotina consiste em um loop principal que executa as seguintes tarefas (*i*) salva contexto, (*ii*) carrega endereços dos registradores VET_TEMP e DEBUG, (*iii*) faz a impressão dos dados a partir do tamanho contido no *header* do pacote e (*iv*) restaura contexto.

Foram necessárias modificações na descrição do *elemento_pe.vhdl* para inclusão do processo que gera o arquivo para *debug* dos dados. Este processo lê os dados no endereço 0xFFFF1A08 e converte para serem escritos no arquivo de impressão.

5 RESULTADOS

Os resultados apresentados a seguir estão divididos em duas fases. Uma fase onde se leva em consideração as primeiras implementações que tem como parte do desenvolvimento as duas interfaces de comunicação (transmissão e recepção), os dois *drivers* de comunicação (*outcoming unit* e *incoming unit*) e a NoC. A segunda fase tem incluídos a implementação e o tratamento das interrupções.

5.1 Validação da Transmissão e Recepção de Pacotes

Esta Seção apresenta os resultados obtidos na primeira fase do projeto, que tem como objetivo validar o envio e a recepção de pacotes. Todas as simulações e validações foram realizadas em VHDL utilizando o Modelsim. Para a validação, foram realizadas simulações que enviam pacotes com o tamanho de 32 *flits*.

O cenário a seguir demonstra a geração de um pacote de 32 *flits*, sendo os 2 primeiros *flits* o destino e o tamanho do pacote, seguidos de 28 *flits* de *payload,* e mais 2 *flits* com a contagem do tempo. Este cenário abrange o envio e a recepção destes pacotes entre três processadores, sendo P1 e P3 os processadores de envio e P2 o processador de destino.

A Figura 42 apresenta os sinais gerados pelo processador PE1 (pacote de dados) para a NI1. Os dados gerados pelo processador são armazenados no *buffer* da NI1, e uma vez o pacote armazenado no *buffer*, a NI1 o injeta na rede.

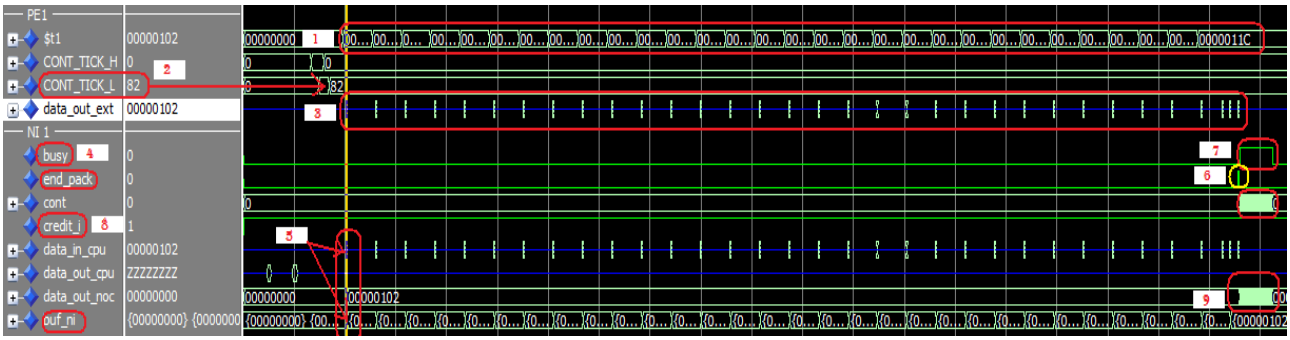

Figura 42 - Pacote gerado pelo elemento de processamento PE1. Sinais gerados pela NI quando está pronta para receber o pacote.

Os eventos gerados compreendem:

1. O processador PE1 gera o pacote com 28 *flits*;
2. É carregada a parte baixa do contador com valor 82 (52H) em CONT_TICK_L;
3. Inicia-se o envio do pacote para o processador de destino pela rede;
4. O sinal *busy* permanece desabilitado, indicando que a NI pode receber o pacote;
5. O pacote é injetado na rede. A NI1 inicia a recepção do pacote, armazenando todos os *flits* no *buffer* da NI1;
6. O sinal *end_pack* indica final de transmissão de pacote;
7. O sinal *busy* é habilitado, indicando que a NI está ocupada;

8. O sinal *credit_i* habilitado, indica que a NoC pode receber dados;
9. Processo de envio do pacote para a NoC.

A Figura 43 apresenta o gráfico dos tempos gerados pelo envio do pacote por PE3, os sinais gerados pelo processador e os sinais gerados pela NI3.

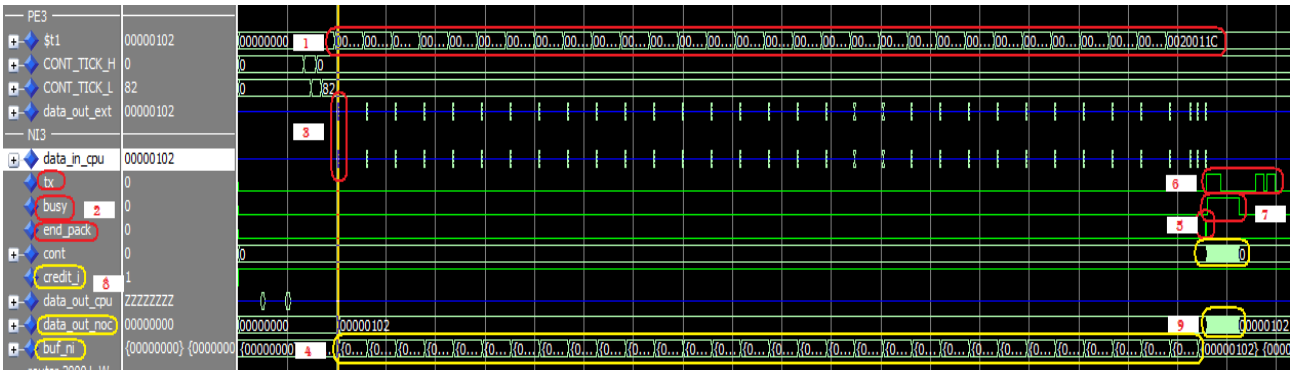

Figura 43 - Pacote gerado pelo elemento de processamento PE3.

Os eventos gerados compreendem:

1. Pacote sendo gerado pelo processador PE3 com tamanho de 28 flits;
2. O sinal *busy* está desabilitado indicando que a NI3 está pronta para receber dados.
3. Momento em que o pacote é enviado pelo processador PE3 e injetado na NI3;
4. Quando injetado na NI o pacote é armazenado no *buffer* (*buf_ni*);
5. Depois de armazenado o pacote, o processador sinaliza a NI3 através do sinal *end_pack* que enviou todo o pacote;
6. O sinal de *tx* está desabilitado indicando à NoC que não há dados para serem enviados. Depois de sinalizado o final do pacote por *end_pack*, a NI3 ativa *tx*, indicando que há pacote a ser enviado;
7. Como a NI3 está ocupada transmitindo o pacote, a NI3 habilita o sinal *busy*;
8. A NoC mantém o sinal *credit_i* habilitado , informando a NI3 que pode receber dados;
9. O pacote é enviado para a NoC.

A recepção dos pacotes é realizada pelo processador PE2 através da NI2. A Figura 44 apresenta os sinais gerados pela NI2 e a recepção do pacote enviado por PE1. Os eventos gerados compreendem:

1. Inicialmente *credit_o* esta habilitado, permitindo assim o envio de pacotes da NoC para NI2. A NI2 só está pronta para receber os dados assim que *rx* estiver habilitado.
2. Momento em que o pacote enviado por PE1 é recebido pela NI2 através de *data_in_noc*;
3. Momento em que o pacote é completamente armazenado no buffer da NI2;
4. Assim que o pacote é totalmente armazenado no *buffer*, a NI2 sinaliza ao processador PE2 que há pacotes a serem enviados através de *ready_pack*;
5. O PE2 lê da NI2 o pacote armazenado no *buffer*;

6. Recepção do primeiro pacote por PE2;
7. Após recebimento de todo o pacote, PE2 sinaliza NI2 através de *end_receive* que finalizou recebimento do primeiro pacote.

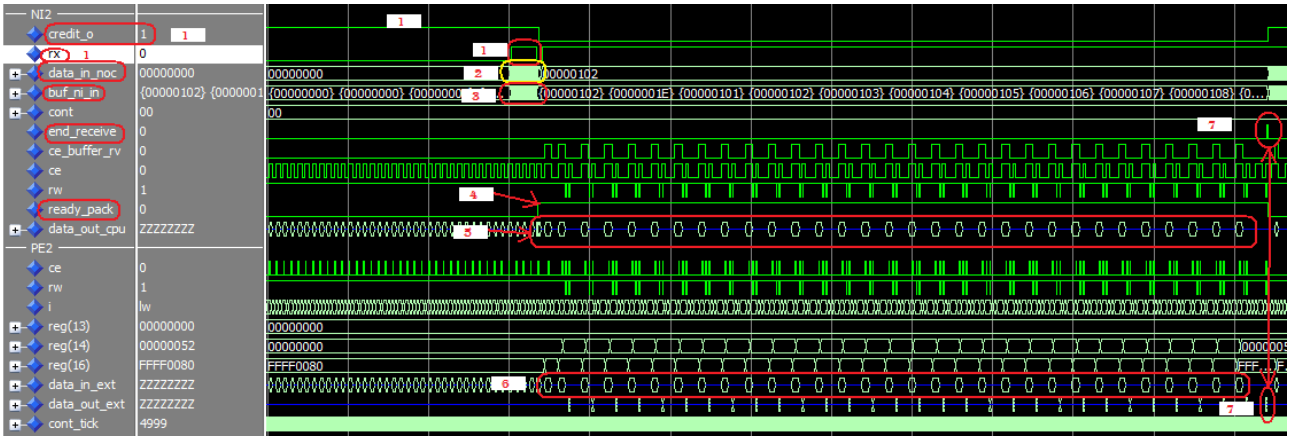

Figura 44 - Recepção do primeiro pacote pela NI2.

A Figura 45 ilustra os sinais gerados pela NI2 e a recepção do segundo pacote gerado por PE3. Os eventos gerados compreendem:

1. A NI2 sinaliza à NoC que pode receber dados habilitando a porta *credit_o*;
2. O sinal *rx* está habilitado, informando a NI2 que há dados a receber da NoC;
3. O pacote é inserido na NI;
4. Momento em que o segundo pacote é completamente armazenado no *buffer* da NI;
5. A NI2 sinaliza ao processador PE2 que há pacotes a serem lidos através de *ready_pack*;
6. A NI2 inicia o envio do segundo pacote para o processador PE2 (destino);
7. O sinal *end_receive* é habilitado, indicando que o segundo pacote foi totalmente lido.

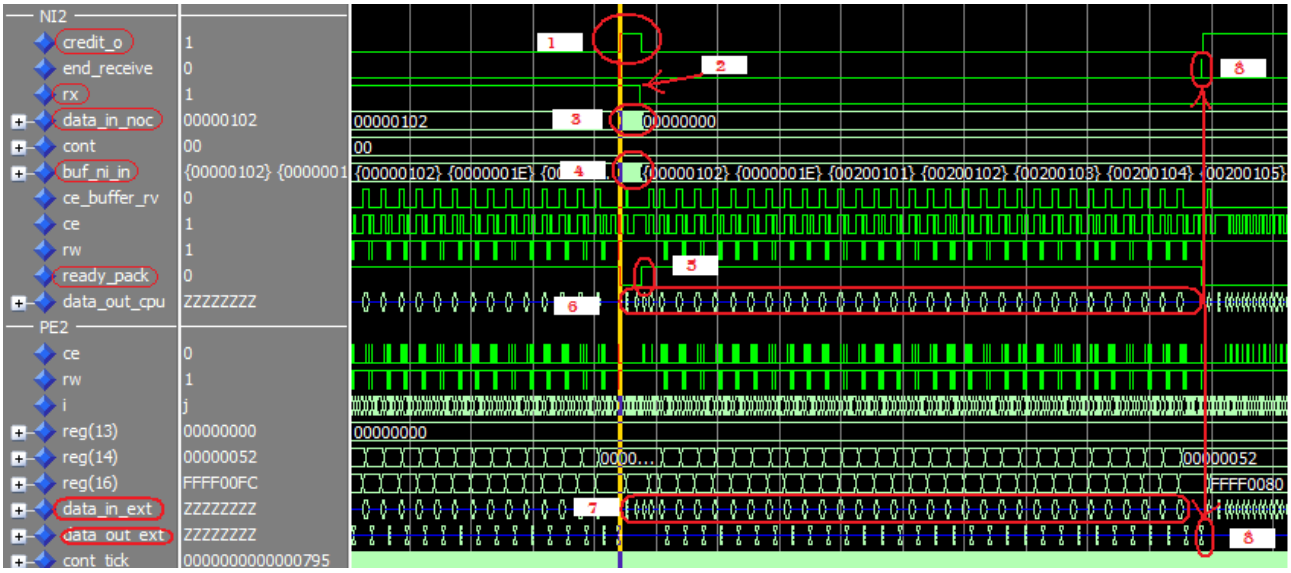

Figura 45 - Recepção do segundo pacote pela NI2.

5.2 Resultados de Latência

Para a avaliação de latência foi utilizada uma configuração com 6 processadores gerando tráfego (pacotes de 32 *flits*) e 3 processadores recebendo este tráfego. A Tabela 3 apresenta a distribuição espacial do tráfego (exemplo: PE1 e PE3 enviam para PE2). Na Figura 46 é apresentada as posições dos roteadores e dos núcleos de processamento na rede NoC.

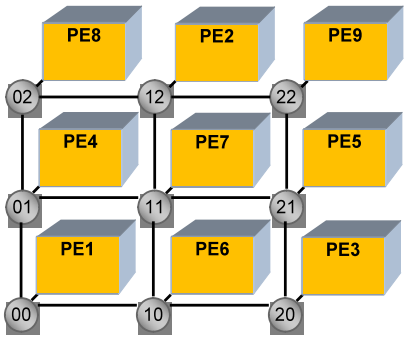

Figura 46 - Posições dos roteadores e núcleos de processamento na rede NoC.

Tabela 3 - Configuração da NoC com distribuição dos NPs (núcleo de processamento), R(roteador).

Configuração dos Núcleos de Processamento			
NP origem	R origem	NP destino	R destino
PE1	00	PE2	12
PE3	20		
PE4	01	PE6	10
PE5	21		
PE7	11	PE9	22
PE8	01		

Os resultados obtidos referem-se apenas às medidas de latência dos pacotes enviados pelos processadores PE1 e PE3 ao processador PE2. As medidas efetuadas levam em conta o tempo que cada instrução leva para ser executada. Analisando o código *assembly* de transmissão podemos constatar o número de instruções executadas. O laço para transmissão possui 7 instruções, como pode se visto na Figura 47.

```
# LAÇO PARA TRANSMISSÃO

loop2: addu  $s3, $s1, $t0
       addu  $s4, $s2, $t0
       lw    $t1, 0($s4)
       sw    $t1, 0($s3)
       addiu $t0,$t0, 4
       addiu $s0,$s0,-1
       bne   $s0, $zero, loop2
```

Figura 47 – Código para transmissão de um pacote com 32 *flits* armazenado em memória.

Cada instrução leva 4 ciclos para ser executada, exceto a instrução *lw* que são 5 ciclos. Assim este laço leva para executar ((6*4) + (1*5)) = 29 ciclos. Como executamos 32 vezes este laço, temos 928 ciclos.

O laço de recepção tem por sua vez o mesmo número de instruções, mas executa só 30

vezes. Logo, leva 870 ciclos. A estes valores vamos adicionar pelo menos 15 instruções (60 ciclos) que são executadas antes e depois do laço, e assim temos aproximadamente:

- 988 ciclos para transmitir um pacote de 32 *flits*;
- 930 ciclos para receber um pacote de 32 *flits.*

A Figura 48 apresenta o início da geração do pacote pelo PE1: (*1*) primeiro *flit* gerado pelo PE1, num total de 32 *flits*. (*2*) tempo decorrido para gerar o início do pacote. (*3*) início da injeção do pacote na rede. Este pacote é injetado na rede com um tempo de 100 ciclos.

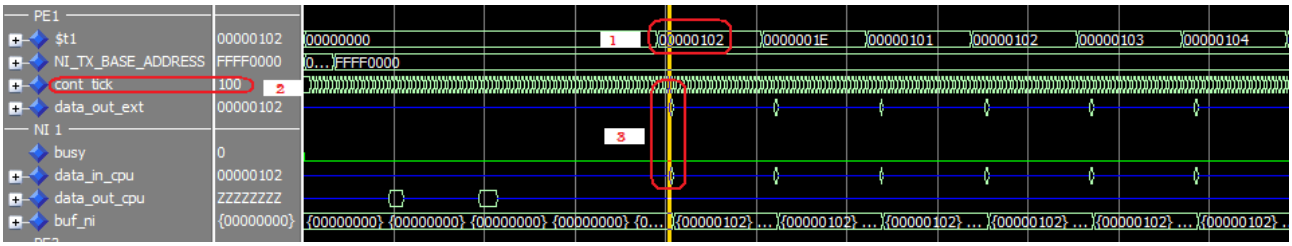

Figura 48 - Tempo de início de geração do pacote.

A Figura 49 mostra o instante em que o pacote é recebido por PE2: (*1*) instante em que o primeiro pacote é recebido da NI2. (*2*) tempo do instante em que o primeiro pacote está sendo recebido – tempo igual a 1042 ciclos.

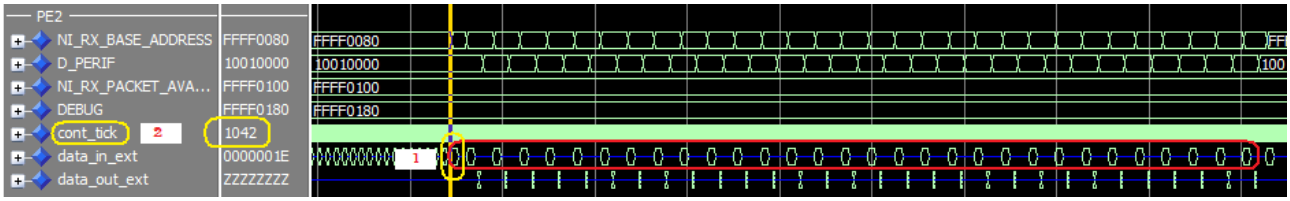

Figura 49 - Tempo de injeção do pacote na rede.

Na Figura 50 é apresentado o resultado da soma dos tempos em que o pacote leva para ser injetado na rede, mais o tempo que ele leva para receber o pacote: (*1*) recepção do pacote. (*2*) tempo total de recepção do pacote – tempo igual a 1900 ciclos.

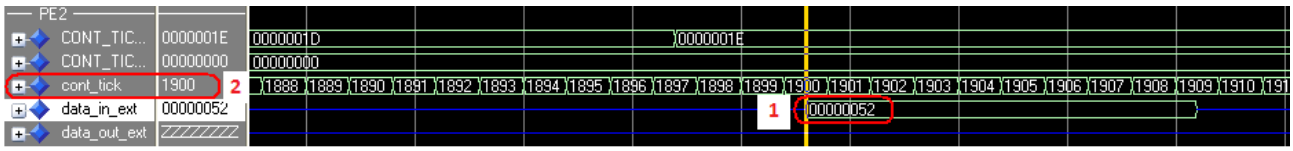

Figura 50 - Tempo de recepção do primeiro pacote.

Estes números conduzem à latência de 1800 ciclos de relógio para o primeiro pacote, de 32 *flits*. Este valor de latência conduz a uma latência por *flit* igual a 56 ciclos de relógio, o que é um valor relativamente alto.

O segundo pacote deve esperar que todo o pacote 1 seja recebido. Logo, a sua latência será o tempo de recepção do primeiro pacote, mais 858 ciclos, que é igual a 2758 *ciclos de clock*, que é aproximadamente o valor apresentado na Figura 51 – 2903 *ciclos de clock*, levando em conta que para iniciar a transmissão foi preciso gastar 100 *ciclos de clock*. A latência do segundo pacote é igual a 2803 *ciclos de clock*, ou 88 *ciclos de clock* por *flit*. Esta maior latência deve-se a colisão dos pacotes, como pode ser facilmente observado no diagrama da Figura 51.

Figura 51 - Tempo de recepção do segundo pacote.

A transmissão dos pacotes dos processadores PE1 e PE3 para o PE2 pode ser esquematizada temporalmente no diagrama da Figura 52. Neste diagrama observa-se que os dois pacotes são simultaneamente inseridos nos *buffers* de suas interfaces de rede. Devido à arbitragem, um é transmitido primeiro, e na sequencia o segundo é transmitido.

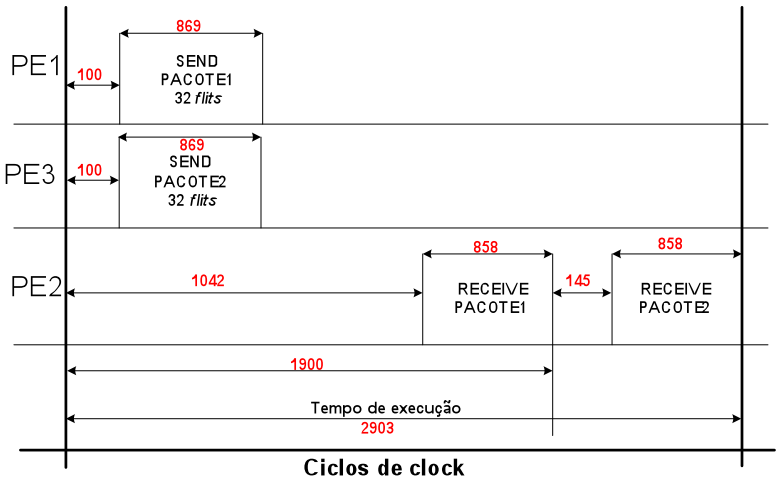

Figura 52 - Tempo para transmissão de 2 pacotes de 32 *flits*, sendo o primeiro sem colisão, e o segundo colidindo com o primeiro.

5.3 Validação da Interrupção

Para efetuar a validação desta segunda fase foi utilizado um aplicação paralela do tipo *merge sort* (código disponível no Apêndice A) utilizando envio de pacotes com tamanhos que variam de 16 à 600 *flits*. Para demonstrar a execução desta aplicação, foram utilizados cinco cenários, que têm por objetivo analisar o desempenho do MPSoC. A aplicação *merge sort* engloba a divisão de um pacote armazenado em memória em um tamanho que dependerá dos *n* processadores que serão utilizados nas simulações. Estes pacotes serão enviados para *n* processadores de destino, os quais executarão a ordenação através de uma aplicação *bubble sort* (código disponível no Apêndice B) dos pacotes recebidos e retornam os pacotes ordenados para o processador de origem. O processador de origem recebe os pacotes ordenados e aplica o *merge* para a ordenação final.

5.3.1 Cenário 1

Este cenário compreende a execução da aplicação *bubble sort* em um único processador. O objetivo deste cenário é verificar os tempos gastos e utilizá-lo com métrica de comparação para os cenários que iremos analisar na seção de resultados. Para esta verificação não foi utilizado em

nenhum momento o acesso à NoC. Para executar um *bubble sort* em um pacote de 600 *flits*, foram consumidos 9.184.058 *ciclos de clock*. O tempo total de execução foi praticamente o tempo de ordenação, desprezando assim o valor gasto para iniciar a rotina.

5.3.2 Cenário 2

Este cenário compreende a execução da aplicação *merge sort* em um único processador (PE1), através da divisão do vetor em duas partes. Comparado o cenário 2 ao cenário 1, obtém a aceleração (*speedup*) pela otimização do algoritmo. Para simplificar a análise de resultados analisamos somente a ordenação de um vetor (pacote) com 600 *flits*. Neste cenário o vetor é dividido em dois outros vetores de tamanhos iguais. A Figura 53 apresenta o gráfico de tempos após a simulação da aplicação *merge sort* executada no processador PE1.

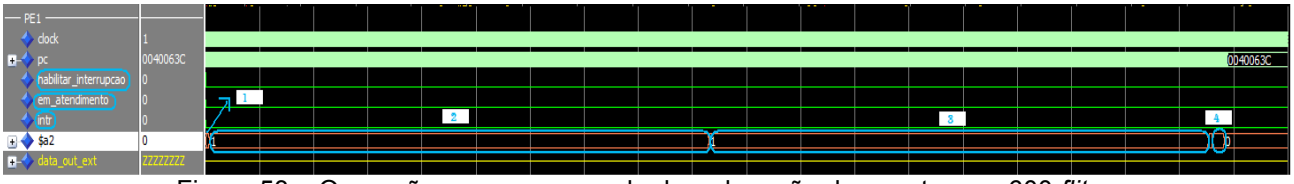

Figura 53 – Operação monoprocessada de ordenação de pacote com 600 *flits*.

1) O pacote armazenado em memória é dividido em duas partes, ou seja, dois pacotes de mesmo tamanho.
2) Ordenação da primeira parte do pacote.
3) Ordenação da segunda parte do pacote.
4) Depois de remontado os dois pacotes, é aplicado o *merge* para ordenação final.

5.3.3 Cenário 3

Este cenário é semelhante ao cenário 2, execução da aplicação *merge sort* em um único processador (PE1), porém divide-se o vetor em quatro partes. A Figura 54 apresenta o gráfico de tempos após a simulação da aplicação *merge sort* executada no processador PE1.

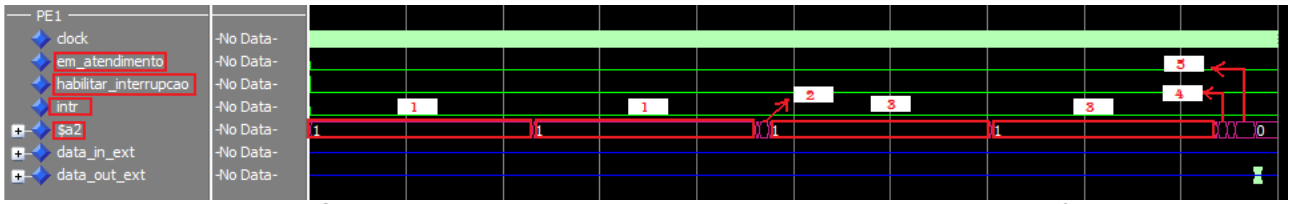

Figura 54 - Operação monoprocessada de ordenação de pacote com 600 *flits*.

1) Tarefa de ordenação dos dois primeiros pacotes.
2) Primeira tarefa de merge aplicada sobre os dois primeiros pacotes.
3) Tarefa de ordenação dos dois pacotes restantes.
4) Segunda tarefa de merge aplicada sobre os dois pacotes restantes.
5) *Merge* final.

5.3.4 Cenário 4

Para a execução deste cenário foi incluído mais dois processadores, PE2 e PE3. O processador PE1 executa a tarefa de *merge* e os dois processadores que foram adicionados à simulação executam a tarefa de ordenação *bubble sort*. Este cenário ordena um vetor de 16 *flits*. A Figura 55 mostra a distribuição espacial dos processadores no MPSoC.

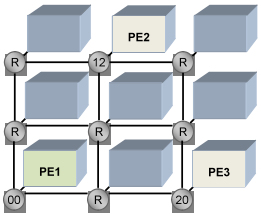

Figura 55 – Posição dos processadores na rede no atual cenário.

A Figura 56 apresenta o gráfico de tempos após a simulação que compreende os três processadores, assim como as principais etapas durante o processo de envio dos pacotes. Nesta figura podemos visualizar os pacotes sendo enviados para os processadores de destino PE2 e PE3. Apesar de não visualizarmos o *header* de cada pacote, os endereços estão nos primeiros *flits*, assim como o tamanho do pacote.

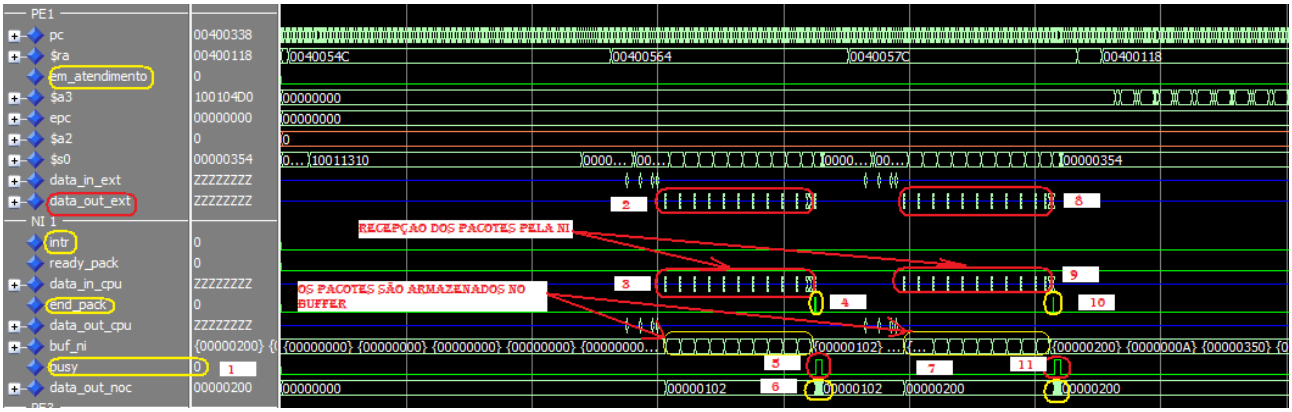

Figura 56 – Processo de envio de dois pacotes.

Descreveremos agora as principais fases durante a transmissão dos dois vetores (pacotes) para os processadores de destino, que executarão o *bubble sort* dos dois vetores.

1) A NI1 sinaliza ao processador através do sinal *busy* que está desocupada e os dados podem ser enviados;
2) Os dados do primeiro pacote são enviados para NI1;
3) Os dados são recebidos pela NI1 e armazenados no *buffer*;
4) Após os dados serem enviados pelo processador, o PE1 sinaliza através de *end_pack* que finalizou a transmissão do primeiro pacote;
5) *busy* sinaliza ao processador que a NI está ocupada, impedindo que o processador envie mais dados. A partir deste ponto a NI inicia o envio do pacote recebido;
6) Depois de armazenados no *buffer* os dados são injetados na rede;

7) Depois de enviado o primeiro pacote, *busy* sinaliza ao processador que a NI está desocupada e pronta para receber um novo pacote;

8) Os dados do segundo pacote são enviados para NI;

9) Os dados são recebidos pela NI e armazenados no *buffer*;

10) Após os dados serem enviados pelo processador, o PE1 sinaliza através de *end_pack* que finalizou a transmissão do segundo pacote;

11) *busy* sinaliza ao processador que a NI está ocupada, impedindo que o processador envie mais dados. A partir deste ponto a NI inicia o envio do pacote recebido. Depois de armazenados no *buffer* os dados são injetados na rede.

A Figura 57 ilustra a forma como a **interrupção** atua no momento que é ativada pela NI. A sequencia de eventos compreende:

1) Os dados são recebidos da NoC pela NI e armazenados no *buffer*, como se trata da recepção de dados via rede, eles são armazenados em *buff_ni_in*;

2) A NI sinaliza ao processador via interrupção que há dados a serem enviados;

3) Após ser sinalizado, o processador imediatamente copia o conteúdo do registrador PC pra o registrador EPC;

4) Dado que o sinail *habilitar_interrupção* está ativo e o sinal *em_atendimento* desativado, a máquina de controle vai para o estado INT.

5) Em estado SInt *lepc* recebe '1', fazendo com que PC receba o valor 0x00400024, saltando para o tratamento da interrupção (rotina *interrupt_handler*).

6) Para sinalizar que o processador está executando a rotina de processamento é ativado o sinal *em_atendimento* enquanto recebe os dados.

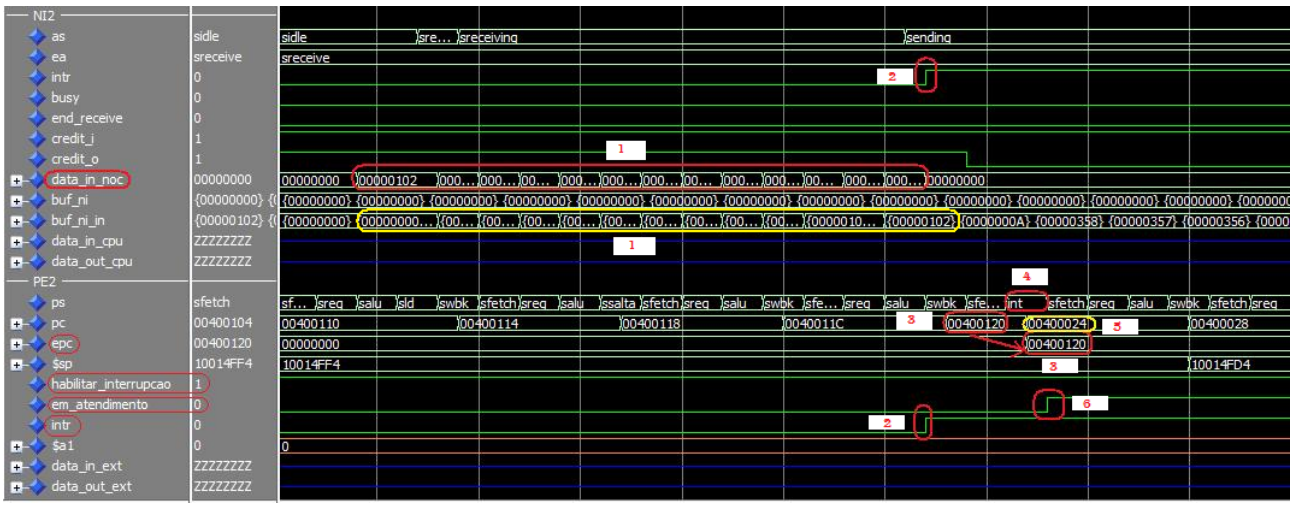

Figura 57 – Momento em que a chamada da interrupção é executada quando a NI necessita enviar um pacote para o processador.

Depois de enviados os pacotes, eles são recebidos pelos processadores que farão o *sort* dos *flits* desordenados. Após ordenação eles são enviados para o processador de destino. A Figura 58 apresenta a recepção do pacote logo após a requisição da interrupção através do sinal *intr* da NI. A requisição da interrupção fica ativa enquanto a NI esta enviando dados para o processador.

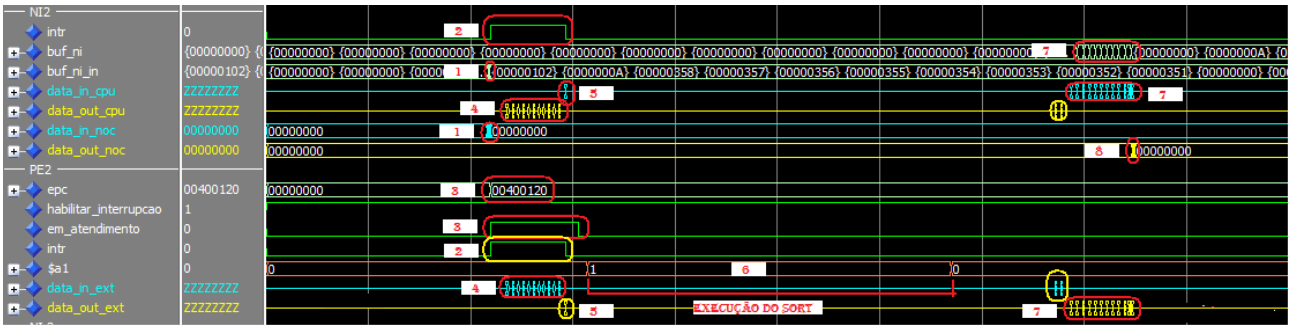

Figura 58 – (4) Recepção do pacote por PE2. (6) Ordenação do pacote. (7) Envio do pacote ordenado.

Os eventos ocorridos para a recepção, ordenação e envio de um pacote são mais bem descritos a seguir:

1) O pacote é armazenado no *buffer* da NI em *buf_ni_in*;
2) A NI interrompe PE2 através do sinal *intr*;
3) PE2 atende a solicitação e habilita o sinal *em_atendimento*;
4) Em *data_in_ext*, PE2 faz recepção do pacote;
5) Após finalização de recepção do pacote PE2 sinaliza a NI, indicando fim de recepção;
6) O registrador $a1 é utilizado para indicar início e término da tarefa de ordenação;
7) PE2 verifica se a NI está ocupada. Envia pacote para NI. Armazena o pacote no *buffer* através de *buf_ni*;
8) Depois de armazenado o pacote no *buffer* a NI envia-o para a rede.

A Figura 59, análoga à Figura 58, ilustra a ordenação do vetor no processador PE3.

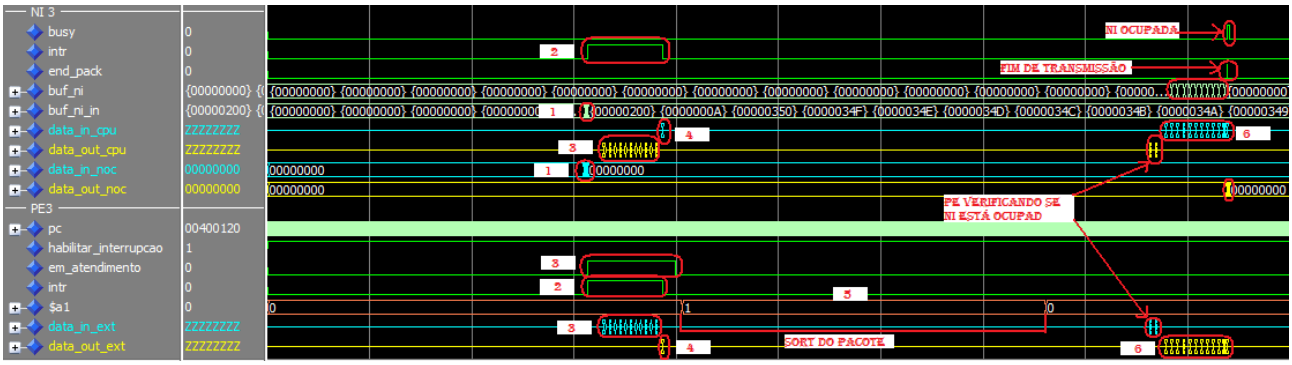

Figura 59 - (3) Recepção do pacote por PE3. (5) Ordenação do pacote. (6) Envio do pacote ordenado.

Para termos uma visão mais ampla do sistema podemos observar a Figura 60, a qual apresenta a aplicação *merge sort* executada em todos os processadores. No envio dos pacotes verificamos que o Pacote 1 é injetado na rede para ser enviado ao processador PE2, somente após ter sido totalmente armazenado no *buffer*. Com isso o Pacote 2 é recebido por PE3 somente após todo o Pacote 1 estar totalmente armazenado em PE2.

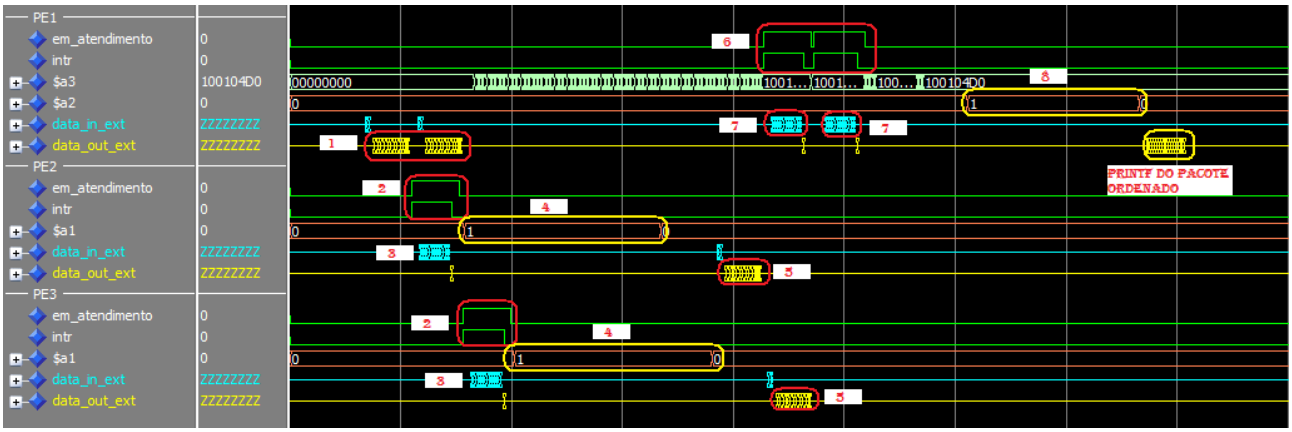

Figura 60 – Aplicação do gráfico das aplicações *merge sort e bubble sort* rodando na rede com 3 processadores.

A sequencia de eventos compreende:

1) Através de *data_out_ext* os pacotes são enviados para os processadores PE2 e PE3;
2) A NI sinaliza o processador mediante o sinal de interrupção intr. O processador habilita o sinal *em_atendimento* para indicar que está atendendo a solicitação da NI;
3) O pacote é armazenado no processador. Após finalização da recepção do pacote, o processador sinaliza fim de recepção;
4) Duração da tarefa de ordenação aplicada aos pacotes recebidos;
5) O processador verifica se a NI está ocupada e envia os pacotes ordenados através de *data_out_ext*;
6) A NI sinaliza o processador PE1 mediante o sinal de interrupção *intr*. O processador PE1 habilita o sinal *em_atendimento* para indicar que está atendendo a solicitação da NI;
7) Em *data_in_ext* ocorre a recepção dos pacotes;
8) Após os pacotes serem remontados é executada a rotina de merge sobre o pacote remontado, tendo como resultado a ordenação final.

5.3.5 Cenário 5

Para realizar o próximo cenário foram incluídos mais dois processadores, PE8 e PE9. Nesta simulação PE1 enviará quatro pacotes para os processadores executarem o *bubble sort* do pacote recebido e retornar para o processador de origem, neste caso o PE1. A Figura 61 ilustra como os processadores estão distribuídos na rede.

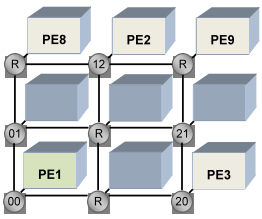

Figura 61 - Posição dos processadores na rede no Cenário 5.

Todas as fases durante a aplicação *merge sort* no sistema com 5 processadores (Figura 62) conectados a rede são descritas a seguir:

1) Injeção dos pacotes na rede. Depois que cada pacote é enviado, envia-se um sinal de fim de transmissão do pacote, habilitando sinal de *end_pack* da NI;
2) Ativação da interrupção (*intr*). Processador sinaliza que está atendendo à requisição habilitando *em_atendimento*;
3) Recepção dos pacotes pelos processadores;
4) Duração da tarefa de ordenação aplicada aos pacotes recebidos;
5) O processador verifica se a NI está ocupada e envia os pacotes ordenados através de *data_out_ext*. Pacotes 1 e 2;
6) O processador verifica se a NI está ocupada e envia os pacotes ordenados através de *data_out_ext*. Pacotes 3 e 4;
7) A NI sinaliza o processador PE1 mediante o sinal de interrupção *intr*. O processador PE1 habilita o sinal *em_atendimento* para indicar que está atendendo a solicitação da NI;
8) Em *data_in_ext* ocorre a recepção dos pacotes;
9) a) Primeiro merge executado sobre os pacotes 1 e 2. b) Primeiro merge executado sobre os pacotes 3 e 4. c) Ordenação final dos pacotes obtido através do merge final.

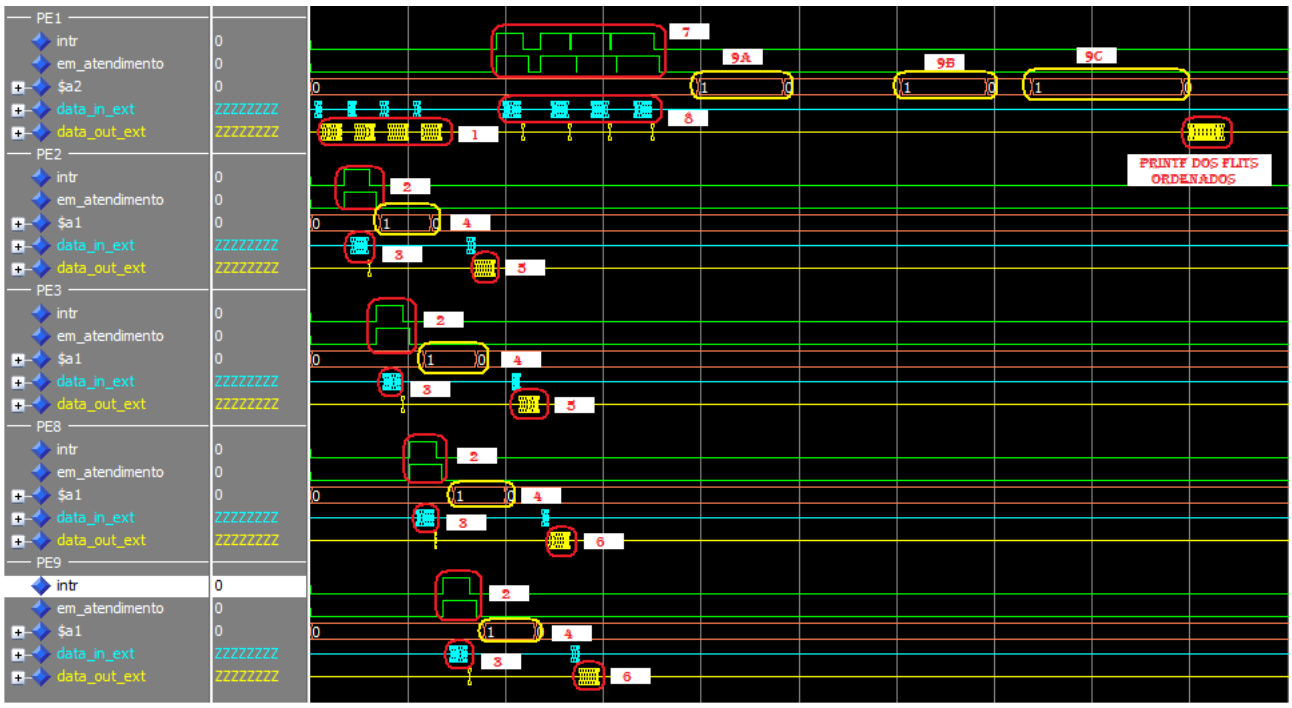

Figura 62 - Apresentação do gráfico das aplicações *merge sort* e *bubble sort* executando em 5 processadores.

5.4 Resultados de Latência e Speedup

Nesta avaliação de resultados temos como principais objetivos analisar os valores de latência e *speedup* dos cenários simulados durante a execução das aplicações *merge sort* e *bubble sort*. Para todos os gráficos dos Cenários simulados, foram analisados os resultados com aplicação do *merge sort* e *bubble sort* em pacotes com 600 *flits*. De acordo com o que vimos na seção anterior, a latência para executar uma ordenação de um pacote com 600 *flits* através de um algoritmo *bubble sort* foi de aproximadamente 9.184.058 *ciclos de clock* em um sistema monoprocessado (Cenário 1). Para executar a mesma ordenação de um pacote de 600 *flits*, no mesmo processador tivemos um tempo de execução de 4.705.002 *ciclos de clock* (Cenário 2).

Na Figura 63 são apresentados os tempos gastos para executar a tarefa de ordenação sobre cada pacote. Podemos observar que dentro do tempo de execução, o maior gasto foi com relação à tarefa de *bubble sort* sobre os pacotes. Convém destacar que neste cenário (Cenário 2) o processador apesar de estar incluído dentro de um MPSoC em momento algum ele faz acesso a rede ou se utiliza de interrupção. Os tempos obtidos neste cenário serão utilizados em conjunto com o Cenário 4 para obtermos os valores de *speedup*.

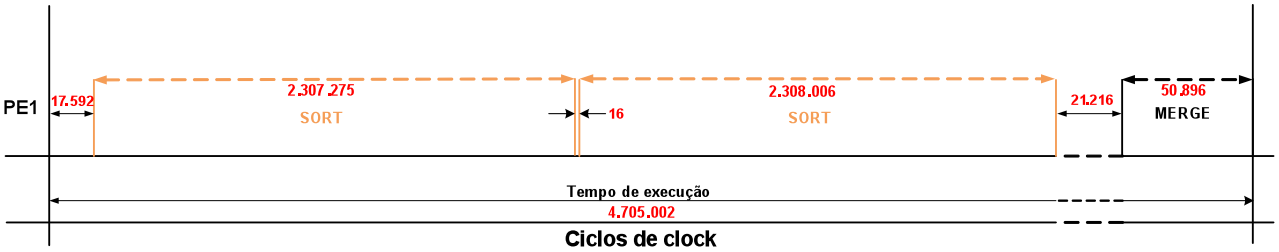

Figura 63 – Cenário 2. Execução monoprocessada das aplicações *merge sort e bubble sort* sobre um pacote com 600 *flits*. Divisão do pacote em duas partes.

Na Figura 64 apresenta-se o tempo de ordenação para o Cenário 3 (sistema monoprocessado, com o vetor dividido em 4 partes iguais). O tempo de execução para este cenário foi de 2.441.433 *ciclos de clock*. A maior contribuição para a redução do tempo gasto deve-se à redução do tamanho de cada pacote, ou seja, em cada ¼ do pacote original. Apesar de utilizarmos três chamadas ao *merge*, o tempo total entre o inicio do primeiro *bubble sort* e o segundo *merge* que é de 2.370.526 *ciclos de clock*, nos dá aproximadamente o tempo de execução. Podemos destacar também o *merge* final que utilizou o mesmo tempo para ser executado em ambos os cenários.

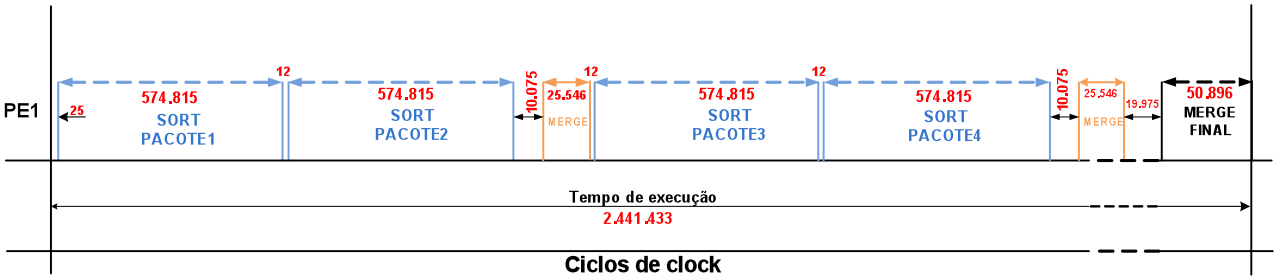

Figura 64 – Cenário 3. Execução monoprocessada das aplicações *merge sorte bubble sort* em um pacote com 600 *flits*. Divisão em quatro partes.

Sendo o cenário 4 a versão multiprocessada doa Cenário 2, verificamos que o tempo para executar a ordenação dos pacotes são aproximadamente os mesmos. Como estamos com os processadores agora conectados à rede, temos um grande tempo em que o processador PE1 fica aguardando que os dois processadores, que neste caso são os processadores PE2 e PE3, executem suas ordenações e enviem seus pacotes. Na Figura 65 podemos visualizar o tempo gasto por PE2 para executar o *bubble sort* no primeiro pacote e enviá-lo para PE1, que é o tempo que PE1 fica aguardando.

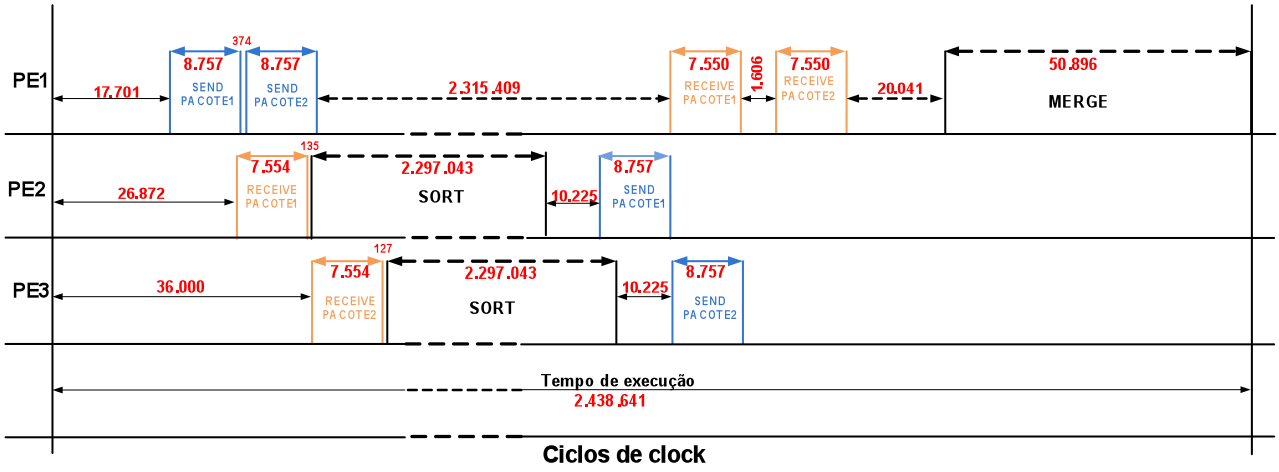

Figura 65 - Cenário 4. *Merge sort* e *bubble sort* executados em um sistema com 3 núcleos de processamento em um pacote com 600 *flits*.

Podemos dizer que o impacto da comunicação da rede não teve influência direta na aplicação, porque se tomarmos o tempo de execução que foi de 2.428.641 *ciclos de clock* e os tempos de ordenação dos dois pacotes e o *merge* final com relação aos tempos que foram gastos com envio e recepção dos pacotes, temos aproximadamente 1,9 % do tempo de execução.

A Tabela 4 apresenta os tempos de execução que foram obtidos através da simulação de ordenação de pacotes de diversos tamanhos. Estes valores foram obtidos dos Cenários 2 e 4.

A Figura 66 apresenta os tempos de execução para uma aplicação *merge sort* ser executa em um sistema com 1 processador e um outro sistema com 3 processadores. O gráfico obtido refere-se aos valores contidos na Tabela 4.

Tabela 4 – Tempos em ciclos por instrução dos Cenários 2 e 4.

Tamanho do vetor em *flits*	Tempo de execução em *ciclos de clock* para 1 CPU	Tempo de execução em *ciclos de clock* para 3 CPUs
16	7.159	7.664
28	16.159	13.894
100	147.271	89.578
152	324.175	185.380
200	548.671	304.388
252	858.175	466.623
300	1.205.071	646.762
400	2.116.471	1.116.673
600	4.705.002	2.438.641

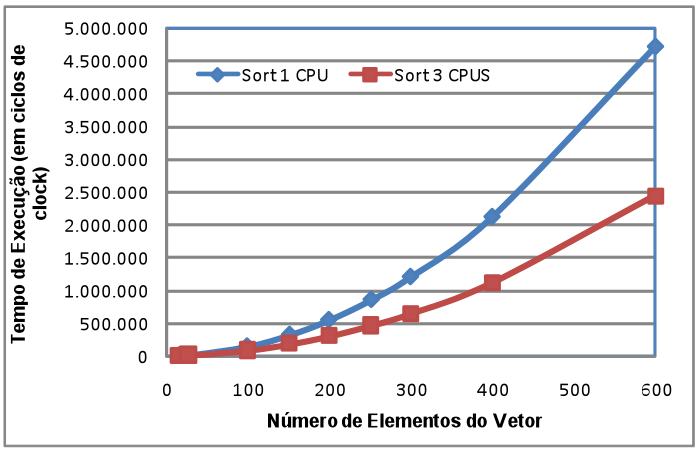

Figura 66 – Gráfico de tempos de execução das aplicações *merge sort* e *bubble sort* em 1 CPU (Cenário 2) e 3 CPUs (Cenário 4).

A Figura 67 apresenta o gráfico de tempos do Cenário 5, que envolve um sistema que possui 5 processadores na rede. Como o pacote é dividido em quatro pacotes assim como foi feito no cenário 3, os tempos de envio, ordenação e recepção são reduzidos. O tempo de envio e recepção dos pacotes por PE1 leva o processador a ficar um tempo ocioso devido ao tempo que o processador PE2 leva para executar a ordenação que é de aproximadamente 574.818 *ciclos de clock*.

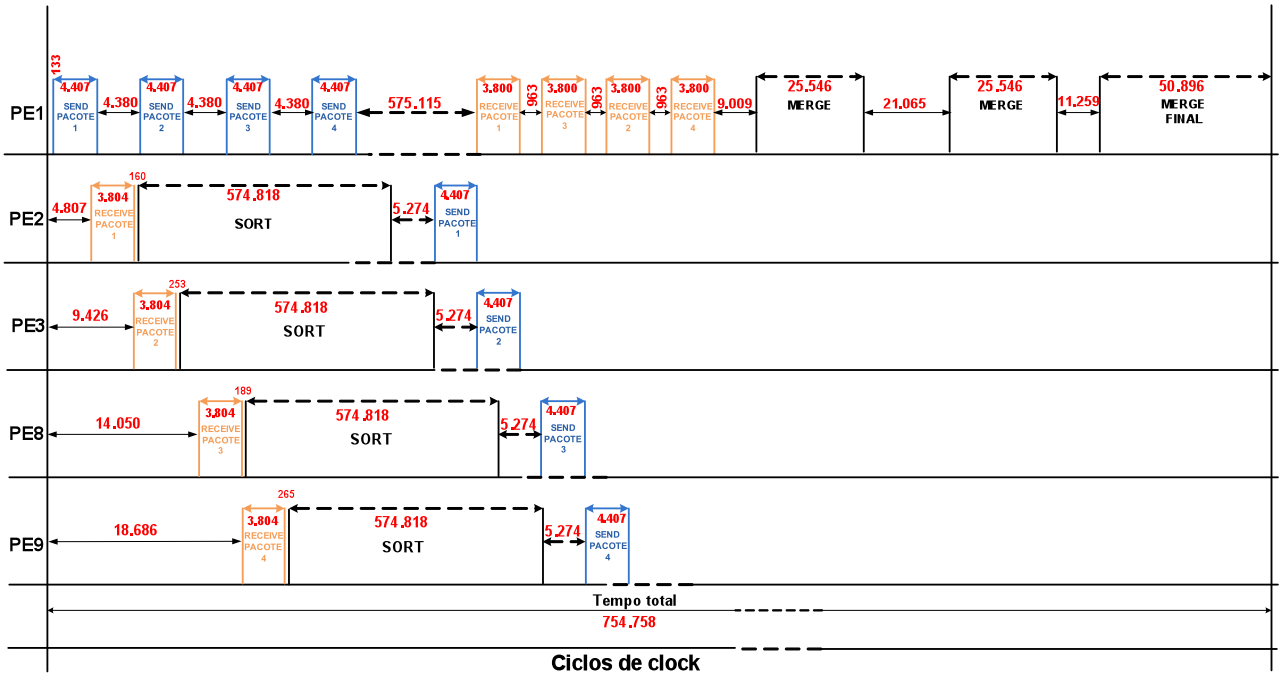

Figura 67 -Cenário 5. *Merge sort* e *bubble sort* executados em um sistema com 5 núcleos de processamento em um pacote com 600 *flits*.

Logo após a recepção de todos os pacotes, há a primeira remontagem, que vem a ser a união dos pacotes 1 e 2 em um único pacote, com o tempo de 9.009 *ciclos de clock*. Depois de remontados os pacotes, é realizado o primeiro *merge* com um tempo total de 25.546 *ciclos de clock*. Na próxima etapa temos um tempo maior de 21.065 *ciclos de clock*. Isso é devido a duas rotinas que são executadas, uma é a segunda remontagem sobre os pacotes 3 e 4 e a outra rotina é a montagem da parte alta do vetor para ser realizado o *merge* final. Logo em seguida é

executado o segundo merge sobre o segundo pacote que foi remontado. Como os pacotes são de tamanhos iguais, obtivemos um tempo igual de 25.546 *ciclos de clock*. Como não há mais pacotes a serem remontados, a próxima etapa a ser executada é a montagem da parte baixa do vetor. Esta etapa leva 11.259 *ciclos de clock*. Como última etapa temos o merge, que executa a ordenação final dos pacotes com tempo final de 50.896 *ciclos de clock*.

Somando os tempos gastos de envio e recepção dos quatro pacotes temos um tempo total de 44.930 *ciclos de clock*. Analisando o tempo total de execução para efetuar a ordenação de todos os pacotes de aproximadamente 754.758 *ciclos de clock*, podemos verificar que tivemos quase 6% de tempo gasto de comunicação na rede. Na Tabela 5 apresentamos os valores obtidos em *ciclos de clock* para os tempos de execução para o Cenário 3, que utiliza um sistema monoprocessado e os valores obtidos para sistemas multiprocessado para 5 processadores (Cenário5).

A Figura 68Figura 66 apresenta os tempos de execução para uma aplicação *merge sort* ser executa em um sistema com 1 processador e um outro sistema com 5 processadores. O gráfico obtido refere-se aos valores contidos na Tabela 5Tabela 4.

Tabela 5 - Tempos em ciclos por instrução 1 CPU (Cenário 3) x 5 CPUs (Cenário 5).

Tamanho do vetor em *flits*	Tempo de execução em *ciclos de clock* para 1 CPU	Tempo de execução em *ciclos de clock* para 5 CPUs
16	7.030	8.924
28	13.294	13.504
100	89.434	49.496
152	185.530	85.556
200	304.834	126.676
252	467.230	179.456
300	647.734	235.776
400	1.118.134	376.975
600	2.441.433	754.758

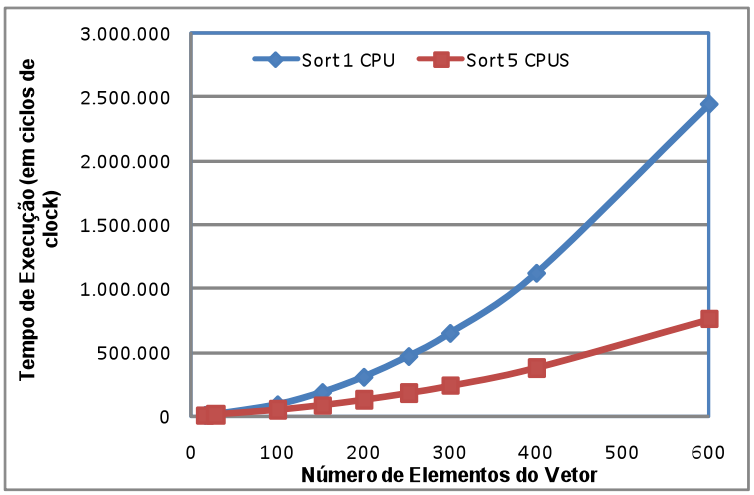

Figura 68 - Gráfico de tempos de execução das aplicações *merge sort* e *bubble sort* em 1 CPU (Cenário 3) e 5 CPUs (Cenário 5).

5.4.1 Speedup

Para que possamos medir as vantagens de incluir em um sistema vários núcleos de processamento, devemos utilizar métricas que mostrem os ganhos obtidos ao executar de forma paralela um algoritmo na arquitetura desenvolvida. Os cenários que foram simulados abrangem essa questão de forma que foi executado o algoritmo de ordenação de forma sequencial e paralela.

Uma tarefa, teoricamente pode ser dividida em *n* tarefas com *n* processadores. Podemos definir o ganho de desempenho como a razão entre o tempo de execução de forma sequencial em um único processador e os *n* processadores.

Conforme [KUM94], a métrica de *speedup* refere-se ao ganho de desempenho obtido pela razão entre o tempo de execução do melhor algoritmo sequencial e sua forma paralela. Vale salientar que o *speedup* não deverá atingir um valor maior que o número de processadores utilizados. Esta análise é definida como Lei de Amdahl.

$$S = \frac{T_1}{T_n}$$

onde: S é o *speedup*, T_1 é o tempo do algoritmo sequencial e T_n é o tempo de execução com n processadores.

Esta lei demonstra o fato de que o gargalo de desempenho de uma aplicação paralela é sua parte sequencial. Por ela, à medida que a parte sequencial do programa aumenta, o *speedup* consequentemente diminui. Portanto para atingir um bom *speedup*, é necessário tornar a parte sequencial do programa a menor possível. Ademais se um problema é constituído de partes paralelizáveis e sequenciais, é necessário fazer com que as partes maiores executem mais rápidas [PAS06].

Os valores de *speedup* na utilização de 2 CPUs e 4 CPUs são apresentados na Tabela 6. Nos valores obtidos para 2 processadores o ganho de desempenho foi praticamente 2, ou seja, o impacto da comunicação a da rotina *merge* foi muito baixo. Já para 4 processadores, o *speedup* ficou abaixo de 4, devido ao tempo de execução das rotinas *merge* e da comunicação.

Tabela 6 – S*peedup - merge sort* e *bubble sort* para 3 e 5 CPUs.

Tamanho do vetor em *flits*	*Speedup* 3 CPUs	*Speedup* 5 CPUs
16	0,93410752	0,78770731
28	1,16298402	0,98441203
100	1,64404765	1,80688338
152	1,74870266	2,16851536
200	1,80253657	2,40640295
252	1,83911745	2,60358807
300	1,86323640	2,74724103
400	1,89533597	2,96606804
600	1,92935389	3,23472292

A Figura 69 apresenta graficamente o s*peedup* observado para as duas implementações.

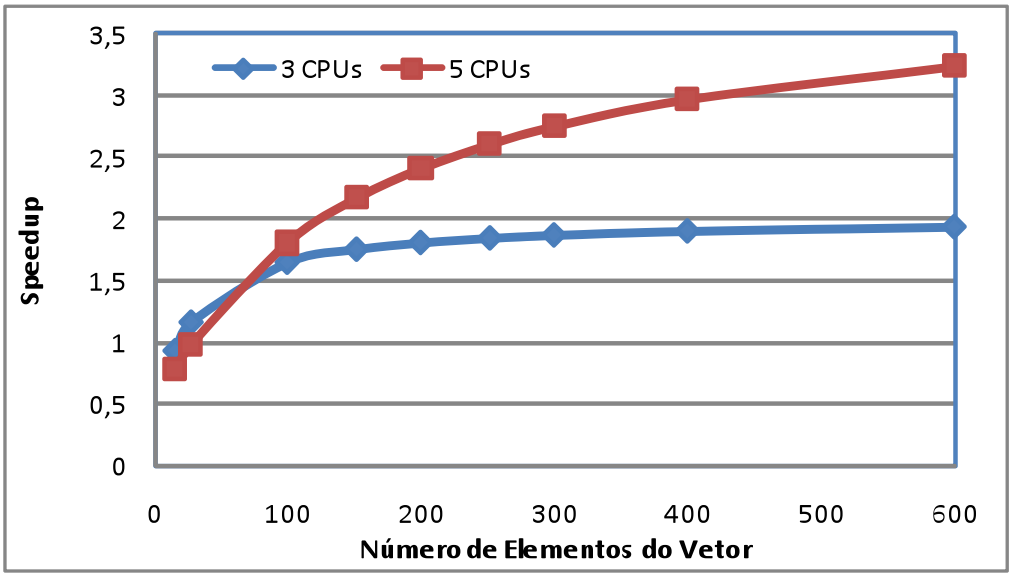

Figura 69 – Gráfico de s*peedup* – *merge sort* e *bubble sort* para 3 e 5 CPUs.

6 CONCLUSÕES

Os atuais projetos de hardware devem aliar as restrições clássicas de projeto, como alto desempenho e baixo consumo de energia, e um reduzido tempo para o produto chegar ao mercado. Uma forma de atender a estas restrições conflitantes é através do reuso de módulos de propriedade intelectual, e em particular através do reuso de processadores, conduzindo assim aos MPSoCs.

Nos primeiros Capítulos foram abordados os principais conceitos relacionados à MPSoCs e NoCs. Atualmente vários métodos vêm sendo desenvolvidos para dar mais poder de processamento aos processadores. O principal deles é colocar vários núcleos de processamento em uma única pastilha de silício. Em 1997 os projetistas da Intel já prediziam que em 2014 teríamos processadores com 8 núcleos, em 2009 isso já é quase uma realidade. Claro que aqui estão sendo considerados processadores para usuários domésticos. Porém se pensarmos que as telecomunicações, os produtos eletrodomésticos, internet, HDTV se utilizam cada vez mais dos recursos de processamento como, por exemplo, acesso a banda larga, onde temos acesso a 100 Mbps e que em alguns países já é possível assistir via internet filmes em alta definição, para que isso seja possível é necessário um grande poder de processamento. Esse contexto comprova o que foi mencionado no Capítulo 1, o grande crescimento da utilização de processadores na indústria, seja ela, de produtos eletrodomésticos, automobilística, médica, telecomunicações, etc. Todas de certa forma vão ter em futuro próximo a interligação de seus produtos. O mecanismo com que os componentes dos sistemas vinham se comunicando não supre mais as necessidades atuais, de alta largura de banda e escalabilidade. Dada as limitações dos barramentos, a interconexão nos MPSoCs está adotando NoCs. O Capítulo 2 apresento diversos projetos de NoCs, e o que mais contribui para o desenvolvimento deste trabalho foi a rede HERMES.

As contribuições deste trabalho compreendem:

1) Desenvolvimento de um MPSoC homogêneo, com reuso dos módulos de processamento e NoC;
2) Desenvolvimento da interface de rede, tendo por característica o uso de *buffers* para injecção/recepeção de dados. Esta característica permite uma simples integração de IPs à rede;
3) Desenvolvimento das camadas de software básico (*drivers*) para a realização da comunicação;
4) Modificação do processador MR4, com a inclusão do mecanismo de interrupção.
5) Avaliação do desempenho do MPSoC proposto utilizando uma aplicação paralela.

A partir dos resultados obtidos através das simulações e das análises realizadas julga-se que os objetivos foram atingidos, ou seja, o desenvolvimento do MPSoc homogêneo, assim como a revisão bibliográfica nos temas abordados.

Como principais sugestões de trabalhos futuros podemos citar: (*i*) integração de um

módulo DMA e execução de simulações com um número maior de processadores para verificar o ganho de desempenho do sistema; (*ii*) inserir outro modelo de processador para analisar qual o impacto na mudança de arquitetura no sistema; (*iii*) pesquisar novos métodos para integrar em um mesmo ambiente a integração da NoC HERMES com “bibliotecas” de IPs diferentes, que venham a possibilitar a redução de tempo de integração de núcleos IPs e aumentar o tempo de análise de resultados .

REFERÊNCIAS BIBLIOGRÁFICAS

[AGA07] Agarwal, A. "The Tile Processor: A 64-Core Multicore for Embedded Processing". TILERA, HPEC, 2007, 18p.

[ATL08] ATLAS - An Environment for NoC Generation and Evaluation. Capturado em: http://www.inf.pucrs.br/~gaph/AtlasHtml/AtlasIndex_us.html , Setembro 2008.

[BAS06] Bastos, E. "Mercury: Uma Rede Intra-chip com Topologia Toro 2D e Roteamento Adaptativo". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2006, 160p.

[BEN02] Benini, L.; De Micheli, G. "Networks on Chips: a New SoC Paradigm". IEEE Computer, Vol. 35 (1), Jan. 2002, pp.70-78.

[BER01] Bergamaschi, R.; Bhattacharya, S.; Wagner, R.; Fellenz, C.; Muhlada, M.; White, F.; Daveau, J.; Lee, W. "Automating the design of SOCs using cores". IEEE Design & Test of Computers, Vol. 18 (5), Set. 2001, pp. 32-45.

[BER04] Bertozzi, D; Benini, L.; "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip". IEE Circuits and Systems Magazine, Vol. 4 (2), Set. 2004, pp. 18-31.

[BER05] Bertozzi, D; Jalabert, A; Murail, S; Tamhankar, R; et al. "NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip". IEEE Trans. on Parallel and Distributed Systems, Vol. 16 (2), Fev. 2005, pp. 113-129.

[BOL04] Bolotin, E.; Cidon, I.; Ginosar, R.; Kolodny, Avinoam; "QNoC: QoS architecture and design process for network on chip". Journal of Systems Architecture, Vol. 50 (2), Dec. 2004, pp. 1-24.

[CAR05] Carara, E.; Moraes, F. G.; "Implementação e Avaliação de Sistema MPSoC Utilizando a Rede HERMES". Technical Report Series, PPGCC-PUCRS, TR051, 2005, p. 29.

[CRA07] Cravotta, R.; "Multicore processor features 64 programmable cores". EDN, Vol. 52 (20), Sep. 2007, p. 20.

[GAP07] GAPH. "Hardware Design Support Group". Capturado em: http://www.inf.pucrs.br/~gaph, Dezembro 2007.

[GOO05] Goossens, K.; Dielissen, J.; Radulescu, A.; "Æthereal Network on Chip: Concepts, Architectures, and Implementations". IEEE Design and Test of Computers, Vol. 22 (5), Set./Out. 2005, pp. 414-421.

[GSC07] Gschwind, M.; et al. "An Open Source Environment for Cell Broadband Engine System Software", IEEE Micro, Vol. 40 (6), Jun. 2007, pp. 37- 47.

[GUP97] Gupta, R. K., Zorian, Y. "Introducing Core-Based System Design". IEEE Design & Test of Computers, Vol. 14 (4), Out./Dec. 1997, pp. 15-25.

[HEN03] Henkel, J. "Closing the SoC design gap". IEEE Computer, Vol. 36 (9), Set. 2003, pp. 119-121.

[IBM07] IBM Research. "The Cell Architecture, Innovation Matters". Capturado em: http://domino.watson.ibm.com/comm/research.nsf/pages/r.arch.innovation.html, Dezembro 2007.

[JER05b] Jerraya, A. A.; Tenhunen, H.; Wolf, W. "Multiprocessor Systems-on-Chips". Computer, Vol. 38 (7), Jul. 2007, pp. 36-40.

[KAH05] Kahle, J.A. ; et al. “Introduction to the Cell Multiprocessor”. IBM Journal Research and Development, Vol. 49 (4/5), Jul./Set. 2005, pp. 589-604.

[MAR01] Martin, G.; Chang, H. “System-on-Chip design”. In: 4th International Conference on ASIC, 2001, pp. 12-17.

[MAR02] Marr, D. T. ; et al. “Hyper - Threading Technology Architecture and Microarchitecture”. Intel Technology Journal, Vol. 6 (1), Feb. 2002, pp. 4 -15.

[MAR02] Marr, D. T. ; et al. “Hyper - Threading Technology Architecture and Microarchitecture”. Intel Technology Journal Q1, 2002, 12p.

[MAR08] MARS. “MIPS Assembler and Runtime Simulator”. Capturado em: http://courses.missouristate.edu/KenVollmar/MARS/, Julho 2008.

[MEL05] Mello, A.; et al. “MultiNoC: A Multiprocessing System Enabled by a Network on Chip”. In: Design, Automation and Test in Europe (DATE '05), 2005, pp. 234-239.

[MEL06] Mello, A. “Qualidade de Serviço em Redes Intra-C: Implementação e Avaliação sobre a Rede HERMES”. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2006, 138p.

[MOH98] Mohapatra, P. “Wormhole Routing Techniques for Directly Connected Multicomputer Systems”. ACM Computing Surveys, Vol. 30 (3), Set. 1998, pp. 374-410.

[MOO65] Moore, E. G.; “Cramming More Components Onto Integrated Circuits”. Electronics, Vol. 38 (8), Apr. 1965, pp. 114-117.

[MOR04a] Moraes, F. G.; et al. “NOCGEN - Uma Ferramenta para Geração de Redes Intra-Chip Baseada na Infraestrutura HERMES”. In: X WORKSHOP IBERCHIP, 2004, pp. 210-216.

[MOR04b] Moraes, F. G.; et al. “HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip”. Integration The VLSI Journal, Vol. 38 (1), Oct. 2004, pp. 69-93.

[MOR08] Moraes, F. G.; Calazans, N. L. V. “Processador Multiciclo – MR4”. Capturado em http://www.inf.pucrs.br/~moraes/ , Janeiro 2008.

[PAS06] Passos, L. B. C. “Avaliação de desempenho de método para a resolução da evolução temporal de sistemas autogravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada”. Dissertação de Mestrado, UNB, 2006, 138p.

[PAT05] Patterson, D. A.; Hennessy, J. L. “Organização e Projeto de Computadores”. Campus, 3ª Edição, 2005, 800p.

[PLA08] PLASMA *Processor*. Capturado em www.opencores.org/projects.cgi/web/mip, Julho 2008.

[REG06] Rego, R. S. “Projeto e Implementação de uma Plataforma MP-SoC usando SystemC”. Dissertação de Mestrado, PPSC-UFRGN, 2006, 144p.

[SPI08] SPIM. “Assembler Simulator for MIPS”. Capturado em: http://www.cs.wisc.edu/~larus/SPIM/pcspim.zip , Julho 2008.

[TAN06] Tanurhan, Y.; “Processors and FPGAs Quo Vadis?”. Computer, Vol. 39 (11), Jan. 2006, pp. 108-110.

[TIL07] TILERA. “The TILE64 Processor Product Brief”. Capturado em: http://www.tilera.com/products/TILEPro64.php , Outubro 2007.

[VAN07] Vangal, S.; et al. “An 80-Tile 1.28 TFLOPS Network on-chip in 65nm CMOS”. In: IEEE International Solid-State Circuits Conference, 2007, pp. 5-7.

[VAN08] Vangal, S.; et al. “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS”. IEEE Journal of Solid-State Circuits, Vol.43(1). Jan. 2008, pp.29-41.

[WEN07] Wentzlaff, D.; et al. “On-Chip In-terconnection Architecture of the Tile Processor”. IEEE Micro, Vol.27(5). Set.-Out. 2007, pp.15-31.

[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.

[ZEF03a] Zeferino, C. A. “Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho”. Tese de Doutorado, PPGC-UFRGS, 2003, 242p.

[ZEF03b] Zeferino, C. A. “Introdução às Redes em chip”. Technical Report, Universidade do Vale do Itajaí – CTTMar/CSED, 2003, p. 12.

APÊNDICE A – APLICAÇÃO MERGE SORT

```
#-------------------------------------------------------------------------------------------------------------------------------------------#
#                                                                                                                                           #
# Recepção e envio de dados, utilizando rotina de interrupção                                                                               #
# Monta dois pacotes a partir de um pacote principal armazenado                                                                             #
# em memória e envia-os para serem ordenados.                                                                                               #
# Recebe os pacotes ordenados e monta-os em único pacote (vetor).                                                                           #
#                                                                                                                                           #
# Autores: Odair Moreira e Fernando Moraes                                                                                                  #
#                                                                                                                                           #
# Revisões:                                                                                                                                 #
# 21/mai/09 - Moraes - Ordenação do código e correção no salvamento de contexto                                                             #
#                                                                                                                                           #
#-------------------------------------------------------------------------------------------------------------------------------------------#
            .text
            j main
#-------------------------------------------------------------------------------------------------------------------------------------------#
# ROTINA PARA TRATAMENTO DE INTERRUPÇÃO - ativada pelo pino externo INTR                                                                     #
#                                                                                                                                           #
#  Esta rotina deve ser "montada" no endereço 0x00004024                                                                                    #
#-------------------------------------------------------------------------------------------------------------------------------------------#
INTERRUPT_HANDLER:
            addiu $sp,$sp,-32
             sw $ra,0($sp)
             sw $t0,4($sp)
            sw $t1,8($sp)
            sw $t3,16($sp)
            sw $t4,20($sp)
            sw $t5,24($sp)

             .set noat
             sw $1, 28($sp)

 #-------------------------------------------------------------------------------------------------------#
 #   laço que busca o endereço de uma área para armazenar o novo pacote                                  #
 #   retorna em $t1 o endereço de onde gravar os dados do pacote                                         #
 #   importante:  este laço DEVE DESABILITAR AS INTERRUPÇÕES SE NÃO HOUVER                               #
 #   espaço para armazenamento de novo pacote                                                            #
 #-------------------------------------------------------------------------------------------------------#
             xor $t0, $t0, $t0
             xor $t2, $t2, $t2
 FIND_PACKET:
             la $t1,PACKET_BASE_ADDRESS
             addu $t1, $t1, $t0                  # endereço do ponteiro do pacote
             lw $t1, 0($t1)                       # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote
             lw $t2, 0($t1)                      # carrega a primeira palavra do pacote
             beq $t2,$zero, READ_PACKET
             addiu $t0, $t0, 4                    # contador "circular": 0,4,8,C, pois só são 4 pacotes

             li $t5, 0x10                        ## se o contador chegou a 0x10 é sinal que os 4 espaços para
            bne $t0, $t5, FIND_PACKET            ## pacotes estão ocupados
             mtc0 $zero, $12                     ## LOGO DESABILITA AS INTERRUPÇÃOES E RECUPERA O CONTEXTO
            ]j loop5


#-------------------------------------------------------------------------------#
#                          Lê pacotes da NI                                     #
#-------------------------------------------------------------------------------#
READ_PACKET:
            la $t2,NI_RX_BASE_ADDRESS       # endereco do periferico
            addiu $t2,$t2,4                 # posiciona NI_RX_BASE_ADDRESS no segundo endereço
             lw $t3, 0($t2)                 # armazena em $t3 o tamanho do pacote
             sw $t3, 0($t1)                 # marca  pacote como ocupado, gravando O TAMANHO do mesmo na primeira posição
                                            # da área de dados escolhida para receber o pacote
             xor $t0, $t0, $t0              #i=0

POSIC:
             addiu $t0,$t0,1                # "i++"
             addiu $t2,$t2,4                # posiciona o $t2 para o payload (3o, 4o, 5o flits)
             lw $t4, 0($t2)                 # carrega o valor apontado por $t2
             sw $t4, 4($t1)                 # armazeno na memória  valor do flit recebido
             addiu $t1,$t1,4                # avança na memória de dados
             bne $t0,$t3,POSIC              # se não recebeu todo o pacote, volta a ler os flits
             la $t0, NI_RX_END_RECEPTION    # endereço do registrador que indica fim do recebimento do pacote
             li $t1, 1
             sw $t1, 0($t0)                 # avisa ao hw final de recebimento de pacote

LOOP5:
             lw $ra,0($sp)
             lw $t0,4($sp)
             lw $t1,8($sp)
             lw $t2,12($sp)
             lw $t3,16($sp)
```

```
            lw $t4,20($sp)
            lw $t5,24($sp)
            lw $1, 28($sp)
            addiu $sp,$sp,32

            mfc0 $a0,$14                    ## reservar o $a0 para o mfco
            jr $a0

#-------------------------------------------------------------------------------------------#
#               Remonta os pacotes para o merge em VET_TEMP             #
#-------------------------------------------------------------------------------------------#
REMONT:
            addiu $sp,$sp,-40
            sw $ra,0($sp)
            sw $t0,4($sp)
            sw $t1,8($sp)
            sw $t2,12($sp)
            sw $t3,16($sp)
            sw $t4,20($sp)
            sw $t5,24($sp)
            sw $t6,28($sp)
            sw $s0,32($sp)
            sw $s1,36($sp)

            jal GET_PACKET_ADRESS

            lw $t1,0($a3)                   #armazena tamanho do pacote
            addiu $t1,$t1,-2                #refaz tamanho do pacote
            la $s0, VET_TEMP                #local onde sera remontado os pacotes
            sll $t1,$t1,1
            sw $t1,0($s0)                   #armazena o tamanho do pacote em VET_TEMP

            li $t0,4                        #i=4 inicia após o tamanho do pacote

            addu $s1,$t1,$0
            sll $s1,$s1,1
            addiu $s1,$s1,4

REM_1:
            slt $t2,$t0,$s1                 #$t1 <-1 se i<((tamanho/2)+1)
            beq $t2,$0,REM_2                #salta para fim se i<((tamanho/2)+1)

            addu $t4,$t0,$a3
            addu $t5,$t0,$s0
            lw $t3,0($t4)                   #$t3 <- conteudo do pacote
            sw $t3,0($t5)                   #armazena em VET_TEMP o conteudo de $t3

            addiu $t0,$t0,4                 #i++

            jal REM_1

REM_2:
            xor $t6, $t6, $t6               #release packet
             sw  $t6, 0($a3)                # para não pegar o mesmo pacote

            jal GET_PACKET_ADRESS           #retorna o endereço do proximo pacote

            sll $s1,$s1,1                   #tamanho do pacote (28*4)
            addiu $s1,$s1,-4
            li $t6,4                        #indice do vetor recebido DEPOIS DO TAMANHO DO PACOTE

REM_3:
            slt $t2,$t0,$s1
            beq $t2,$0,exit_2

            addu $t4,$t6,$a3                #atualiza endereço da posição do vetor recebido
            addu $t5,$t0,$s0                #atualiza endereço da posição de VET_TEMP
            lw $t3,0($t4)
            sw $t3,0($t5)

            addiu $t0,$t0,4                 #i++
            addiu $t6,$t6,4                 #j++

            jal REM_3

EXIT_2:
            lw $ra,0($sp)
            lw $t0,4($sp)
            lw $t1,8($sp)
            lw $t2,12($sp)
            lw $t3,16($sp)
            lw $t4,20($sp)
            lw $t5,24($sp)
            lw $t6,28($sp)
            lw $s0,32($sp)
            lw $s1,36($sp)
            addiu $sp,$sp,-40

            jr $ra
```

```
#-------------------------------------------------------------------------------------#
#                Executa o merge no pacote remontado                                  #
#-------------------------------------------------------------------------------------#
MERGE:
        addiu $sp,$sp,-28
        sw $ra, 0($sp)
        sw $s0, 4($sp)
        sw $s1, 8($sp)
        sw $s2,12($sp)
        sw $s3,16($sp)
        sw $t1,20($sp)
        sw $t2,24($sp)

        la $s0,VET_TEMP                 #vetor onde está remontado pacote
        la $t9,MERG_TEMP                #vetor temporario para o merge

        lw $s1,0($s0)
        addu $s3,$s1,$0
        sll $s3,$s3,2
        addu $s7,$s3,$0
        addiu $s3,$s3,4

        li $t0, 4                       #i = inicio

#--Armazena em um vetor temporário o vetor remontado

MERG_1:
        slt $t5, $t0, $s3               #$t5<-1 se i<=fim
        beq $t5, $0,MERG_24             #salta se i<=fim

        addu $t1,$s0,$t0                #posiciona em VET_TEMP
        lw $t1,0($t1)                   #$t1<-conteudo de VET_TEMP
        addu $t2,$t9,$t0                #posiona em MERG_TEMP
        sw $t1,0($t2)                           #temp[i] <- vetor[i] <-> MERG_TEMP[i]<-VET_TEMP[i]

        addiu $t0, $t0, 4               #i++

        j merg_1

MERG_24:
        li $t0,4                        #i=inicio=4
        addu $t1,$s1,$0
        sll $t1,$t1,1
        addiu $t1,$t1,4
        li $t2,4                        #posição=inicio
        addu $t3,$t1,$0
        addu $s5,$t1,$0
        li $s6,0                        #$s6 <- k=0


MERG_2:
        slt $t4,$t0,$t3                 #$t4<-1 se i<=meio
        beq $t4,$0,MERG_3                       #salta para merg_3 se i<=meio

        slt $t4,$t1,$s3                 #$t4<-1 se j<=fim
        beq $t4,$0,MERG_3                       #salta para merg_3 se j<=fim

        addu $t5,$t9,$t0                #posiciona no vetor o primeiro setor MERG_TEMP
        lw $t5,0($t5)                           #$t5 <- temp[i]
        addu $t6,$t9,$t1                #posiciona no vetor o segundo setor vMERG_TEMP
        lw $t6,0($t6)                           #$t6 <- temp[j]

MERG_21:
        slt $t4,$t5,$t6                 #$t4 <-1 se temp[i]<temp[j]
        beq $t4,$0,MERG_22                      #salta para merg_21 se temp[i]<temp[j]

        addu $t7,$s0,$t2                #posiciona no vetor VET_TEMP
        sw $t5,0($t7)                           #VET_TEMP[posição] <- temp[i]

        addiu $t0,$t0,4                 #i++
        jal MERG_23

MERG_22:
        addu $t7,$s0,$t2
        sw $t6,0($t7)                   #VET_TEMP[posição] <- temp[j]

        addiu $t1,$t1,4                 #j++
        jal MERG_23

MERG_23:
        addiu $t2,$t2,4                 #posição++
        jal MERG_2

MERG_3:
        bne $t0,$s5,MERG_32             #salta se i== meio +1
        addu $s6,$t1,$0                 #k <-j

MERG_31:
```

```
        slt $t4,$s6,$s7                 #$t4 <- 1 se k<=fim
        beq $t4,$0,EXIT_MERG

        addu $t8,$t9,$s6                #
        sw $t6,0($t8)                   #VET_TEMP[posicao)<-MERG_TEMP[k]

        addiu $t2,$t2,4                 #posição++
        addiu $s6,$s6,4                 #k++
        jal MERG_31

MERG_32:
        addu $s6,$t0,$0                 #$s6 <-i <-> k=i

MERG_33:
        slt $t4,$s6,$t3                 #$t4 <-1 se exit_merg se k<=meio
        beq $t4,$0,EXIT_MERG                    #salta para exit_merg se k<=meio

        addu $t8,$t9,$s6                #$t8=&MERG_TEMP atual
        lw $t6,0($t8)                           #$t6 recebe conteudo de MERG_TEMP
        addu $t0,$s0,$t2                #$t0 recebe posição atual de VET_TEMP
        sw $t6,0($t0)                           #VET_TEMP[posicao)<-MERG_TEMP[k]

        addiu $t2,$t2,4                 #posição++
        addiu $s6,$s6,4                 #k++
        jal MERG_33

EXIT_MERG:
        lw $ra,0($sp)
        lw $s0,4($sp)
        lw $s1,8($sp)
        lw $s2,12($sp)
        lw $s3,16($sp)
        lw $t1,20($sp)
        lw $t2,24($sp)
        addiu $sp,$sp, 28

        jr $ra
#-------------------------------------------------------------------------------------------------------------------------------------------------------------#
#   Retorna endereço do pacote válido no registrador $a3 - ROTINA BLOQUEANTE                                                                                  #
#-------------------------------------------------------------------------------------------------------------------------------------------------------------#
GET_PACKET_ADRESS:
        addiu $sp,$sp,-12
        sw $ra,0($sp)
        sw $t0,4($sp)
        sw $t1,8($sp)

        xor $t0, $t0, $t0
l0:
        la $a3,PACKET_BASE_ADDRESS
        addu $a3, $a3, $t0              # endereço do ponteiro do pacote
        lw $a3, 0($a3)                  # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote em $v0
        lw $t1, 0($a3)                  # carrega a primeira palavra do pacote
        bne $t1,$zero, l1
        addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes
        andi  $t0, $t0, 0xF
        j l0

l1:
        lw $ra,0($sp)
        lw $t0,4($sp)
        lw $t1,8($sp)

        addiu $sp,$sp,12
        jr $ra

#-----------------------------------------------------------------------------#
#       Cria os dois vetores para serem enviados                              #
#-----------------------------------------------------------------------------#
MONT_VETOR:
        addiu $sp,$sp,-28               #
        sw $ra,0($sp)                   #
        sw $s0,4($sp)                   #
        sw $s3,8($sp)                   #salva contexto
        sw $s4,12($sp)                  #
        sw $s5,16($sp)                  #
        sw $t1,20($sp)                  #
        sw $t2,24($sp)

        la $s0,D_PERIF
        la $v0,VET_TX_1                 #vetores onde serão montados
        la $v1,VET_TX_2                 #os pacotes a serem ordenados

        lw $s5,0($s0)                   #TAMANHO do payload
        addu $a1,$s5,$0
        srl $a1,$a1,1                   #TAMANHO/2
        addu $t6,$s5,$0
        sll $t6,$t6,1                   #TAMANHO*2
        addiu $t6,$t6,4                 #soma 4 a (TAMAMNHO*2), o tamanho foi incluido no header do pacote(D_PERIF)
```

```
        addu $s4,$0,$t1                 #fim = tamanho-1
#
### CRIA OS DOIS VETORES PARCIAIS A PARTIR DO PACOTE ARMAZENADO EM MEMÓRIA
#
        li $t0,4                        #i=0
        addu $t8,$t6,$0
        li $t7,8                        #j=8 (as duas primeiras posições são reservados para o header)

LOOP4:
        slt $t3,$t0,$t6 #salta de i>tamanho/2
        beq $t3,$0,exit

        addu $t4, $s0, $t0              # $t4 = &D_PERIF[i] - armazena o endereço do vetor
        lw $t4, 0($t4)                  # $t4 = D_PERIF[i] - armazena o valor contido no endereço em $t4
        addu $t5, $v0, $t7              # $t5 = &VET1[i]
        sw $t4, 0($t5)                  # VET1[i] = D_PERIF[i]

        addu $t4, $s0,$t8
        lw $t4 0($t4)                   # $t4 <- conteudo no endereço contido em $t4
        addu $t5, $v1, $t7              # $t5 <- endereço  do segundo vetor temporario
        sw $t4, 0($t5)                  # $t5 <- conteudo no endereço contido em $t5 - VET2[i] = D_PERIF[i]

        addiu $t0,$t0,4 #i++
        addiu $t8,$t8,4 #i++
        addiu $t7,$t7,4 #j++

        j LOOP4

EXIT:
        lw $ra,0($sp)                   #
        lw $s0,4($sp)                   #
        lw $s3,8($sp)                   #
        lw $s4,12($sp)                  #recupera contexto
        lw $s5,16($sp)                  #
        lw $t1,20($sp)                  #
        lw $t2,24($sp)                  #
        addiu $sp,$sp,28                #

        jr $ra

#-------------------------------------------------------------------------------#
#                       Rotina Envia Pacote                                     #
#-------------------------------------------------------------------------------#
SEND:
        addiu $sp,$sp,-12               #
        sw $ra,0($sp)                   #
        sw $s2,4($sp)                   #
        sw $t1,8($sp)


        la $s1, NI_TX_BASE_ADDRESS      # carrega $rede
        la $t3, NI_TX_BUSY              # carrega $registrador  que indica NI ocupada
        la $t4, NI_TX_END_PACKET        # carrega &registrador  que indica fim do pacote



        lw $s0, 4($s2)                  # $s0 <- carrega o tamanho do pacote
        addiu $s0, $s0,2                # incrementa de duas unidades o $s0, pois envia o header (2 flits)

        sw $s0, 4($s2)                  # TICK: MUDA O TAMANHO DO PACOTE, POIS IREMOS INSERIR O TEMPO
                                        # (para cálcula da latência)

LOOP:
        lw $t0, 0($t3)                  # verifica se a NI está livre(NI_TX_BUSY)
        bne $t0, $zero, LOOP

        li $t0, 0                       #i=0  $t0 será usado como deslocamento no vetor do pacote e do
                                        #endereço do periférico

#                                                               ########################################
# CARGA DO TEMPO ATUAL, A SER INSERIDO NO FINAL DO PACOTE  ########################################
#                                                               ########################################
        la $s5, CONT_TICK_H             # carrega o endereço da parte alta do contador de clock
        la $s6, CONT_TICK_L             # carrega o endereço da parte baixa do contador de clock
        lw $s5, 0($s5)
        lw $s6, 0($s6)                  # CARREGA EM $s5 e $s6 a base de tempo atual

#                                                                       ######################################
# LAÇO PARA TRANSMISSÃO DO PACOTE ORIGINAL SEM A BASE DE TEMPO ######################################
#                                                                       ######################################
LOOP2:
        addu $s3, $s1, $t0              #$s3 <- &temp_rede(posição atual)
        addu $s4, $s2, $t0              #$s4 <- &temp_pacote armazenado em memoria (posição atual)


        lw $t1, 0($s4)                  # leio o elemento do pacote que está em memória do processador
        sw $t1, 0($s3)                  # gravo o elemento lido na rede

        addiu $t0,$t0, 4                # incrementa indice de deslocamento no vetor
        addiu $s0,$s0,-1                # decrementa size( tamanho do pacote atual)
```

```
            bne $s0, $zero, LOOP2


#                                                                                                          ########
# COLOCA O TEMPO PRÉ-CARREGADO NO FINAL DO PACOTE, lembrar: o size já foi incrementadio acima ########
#                                                                                                          ########
            addiu $s3, $s3, 4                   #incrementa &periferico
            sw $s5, 0($s3)                      #carrega parte alta do contador no final do pacote(penúltimo flit)
            addiu $s3, $s3, 4                   #incrementa &periérico
            sw $s6, 0($s3)                      #carrega parte baixa do contador no final do pacote(último flit)


            li $s0, 1
            sw $s0, 0($t4)                      # envia estado de NI_TX_END_PACKET para indicar fim transmissão do
                                                            #pacote


#
#  IMPORTANTE: RESTAURA O TAMANHO ORIGINAL DO PACOTE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#
            la $s2, D_PERIF                     # carrega &pacote armazenado em memoria
            lw $s0, 4($s2)                      # $s0 <- tamanho do pacote
            addiu $s0, $s0, -2                  #
            sw $s0, 4($s2)

            lw $ra,0($sp)                        #
            lw $s2,4($sp)                        #
            lw $t1,8($sp)                        #
            addiu $sp,$sp,12                     #

            jr $ra
#-------------------------------------------------------------------------------------------------------------------#
#       IMPRESSÃO DO VETOR ORDENADO NA ÁREA DEBUG                #
#-------------------------------------------------------------------------------------------------------------------#
DEBUG:
            addiu $sp,$sp,-24
            sw $ra,0($sp)
            sw $t0,4($sp)
            sw $t1,8($sp)
            sw $t2,12($sp)
            sw $t3,16($sp)
            sw $s0,20($sp)

            la $s0, DEBUG
            la $t3, VET_TEMP

            lw $t1, 0($t3)                      # primeiro elemento de $v0 é o tamanho do pacote lido
            addiu $t1,$t1,1                     # tamanho + 1 para incluir o tamanho pacote
            li $t0, 0                           # começa em -1 pois vamos fazer o printf do tamanho do pacote também
DEB:
            lw $t2,0($t3)
            sw $t2,0($s0)
            addiu $t3,$t3,4
            addiu $t0,$t0,1                     # i++
            bne $t0,$t1,DEB

            lw $ra,0($sp)
            lw $t0,4($sp)
            lw $t1,8($sp)
            lw $t2,12($sp)
            lw $t3,16($sp)
            lw $s0,20($sp)
            addiu $sp,$sp,24

            jr $ra

#-----------------------------------------------------------------------------------#
#                       Rotina Principal                                            #
#-----------------------------------------------------------------------------------#
            .globl main

main:
#-----------------------------------------------------------------------------------#
#                       Interrupções são habilitadas                               #
#-----------------------------------------------------------------------------------#
            li $1,1
            mtc0 $1,$12

            jal MONT_VETOR

            la $s2, VET_TX_1                    #carrega &pacote armazenado em memoria
            addiu $t0,$0,0x102                  #&processador destino
            sw $t0, 0($v0)                       #carrega endereço de destino
            sw $a1,4($v0)                        #carrega tamanho do pacote

            jal SEND                             #envia pacote a ser ordenado

            la $s2,VET_TX_2
```

```
        addiu $t0,$0,0x200              #&processador destino
        sw $t0,0($v1)                   #carrega endereço de destino
        sw $a1,4($v1)                   #carrega tamanho do pacote

        jal SEND

        jal REMONT                      #chama a rotina para remontar o pacote

        li $a2,1

        jal MERGE                       #chama a rotina para ordenar o pecote remontado

        li $a2,0

        jal DEBUG                       #faz print do vetor ordenado

FIM:
        j FIM


#-------------------------------------------------------------------------------#
#                       Área de dados                                           #
#-------------------------------------------------------------------------------#
.data
PACKET_BASE_ADDRESS:    .word  pck1 pck2 pck3 pck4
pck1:   .space 0x4c0
pck2:   .space 0x4c0
pck3:   .space 0x4c0
pck4:   .space 0x4c0

D_PERIF:  .word      0x0000001C 0x00000010 0x00000357 0x00000356 0x00000355 0x00000354 0x00000353 0x00000352 0x00000351 0x00000350 0x0000034f
0x0000034e 0x0000034d 0x0000034c 0x0000034b 0x0000034a 0x00000349 0x00000348

VET_TX_1:  .space 0x4c0

VET_TX_2:  .space 0x4c0

VET_TEMP:  .space 0x970

MERG_TEMP: .space 0x970

#-------------------------------------------------------------------------------#
#                       Recepção de dados                                       #
#-------------------------------------------------------------------------------#
.data 0xFFFF1000
NI_RX_BASE_ADDRESS: .space 0x970

.data 0xFFFF1984
NI_RX_PACKET_AVAILABLE: .space 4

.data 0xFFFF1980
NI_RX_END_RECEPTION: .space 4
#-------------------------------------------------------------------------------#
#                       Envio de dados                                          #
#-------------------------------------------------------------------------------#
.data 0xFFFF0000
NI_TX_BASE_ADDRESS: .space   0x970

.data 0xFFFF1978
NI_TX_BUSY: .space 4

.data 0xFFFF197C
NI_TX_END_PACKET: .space 4
#-------------------------------------------------------------------------------#
#                                 Debug                                         #
#-------------------------------------------------------------------------------#
.data 0xFFFF1988
CONT_TICK_L: .space 0x40

.data 0xFFFF19C8
CONT_TICK_H: .space 0x40

.data 0xFFFF1A08
DEBUG: .space  0x970
```

APÊNDICE B – APLICAÇÃO BUBBLE SORT

```
#-----------------------------------------------------------------------------------------------------------------------#
#                                                                                                                       #
# Recepção e envio de dados, utilizando rotina de interrupção                                                           #
# Recebe dois pacotes desordenados, ordena-os e envia os dois                                                           #
# pacotes para montá-los e aplicar um rotina merge                                                                      #
#                                                                                                                       #
# Autores: Odair Moreira  e  Fernando Gehm Moraes                                                                       #
#                                                                                                                       #
# Revisões:                                                                                                             #
# 21/mai/09 - Moraes - Ordenação do código e correção no salvamento de contexto                                         #
#                                                                                                                       #
#-----------------------------------------------------------------------------------------------------------------------#
        .text
        j    main

#-----------------------------------------------------------------------------------------------------------------------#
# ROTINA PARA TRATAMENTO DE INTERRUPÇÃO - ativada pelo pino externo INTR                                                 #
#                                                                                                                       #
# esta rotina deve ser "montada" no endereço 0x00000004                                                                 #
#-----------------------------------------------------------------------------------------------------------------------#
interrupt_handler:

        addiu $sp,$sp,-32
        sw $ra,0($sp)
        sw $t0,4($sp)
        sw $t1,8($sp)
        sw $t2,12($sp)
        sw $t3,16($sp)
        sw $t4,20($sp)
        sw $t5,24($sp)

        .set noat
        sw $1, 28($sp)

#-------------------------------------------------------------------------------------------------#
#   laço que busca o endereço de uma área para armazenar o novo pacote                            #
#   retorna em $t1 o endereço de onde gravar os dados do pacote                                   #
#   importante:  este laço DEVE DESABILITAR AS INTERRUPÇÕES SE NÃO HOUVER                         #
#   espaço para armazenamento de novo pacote                                                      #
#-------------------------------------------------------------------------------------------------#

        xor $t0, $t0, $t0
        xor $t2, $t2, $t2
find_packet:
        la $t1,PACKET_BASE_ADDRESS
        addu $t1, $t1, $t0              # endereço do ponteiro do pacote
        lw $t1, 0($t1)                  # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote
        lw $t2, 0($t1)                  # carrega a primeira palavra do pacote
        beq $t2,$zero, read_packet
        addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes

        li $t5, 0x10                    ## se o contador chegou a 0x10 é sinal que os 4 espaços para
        bne $t0, $t5, find_packet       ## pacotes estão ocupados
        mtc0 $zero, $12                 ## LOGO DESABILITA AS INTERRUPÇÃOES E RECUPERA O CONTEXTO
        j loop5


#-----------------------------------------------------------------------#
#                    Lê pacotes da NI                                   #
#-----------------------------------------------------------------------#

read_packet:
        la $t2,NI_RX_BASE_ADDRESS       # endereco do periferico

        addiu $t2,$t2,4                 # posiciona NI_RX_BASE_ADDRESS no segundo endereço - o primeiro flit é o endereço
        lw $t3, 0($t2)                  # TAMANHO do payload + 2 flits do cont_tick

        addiu $a3,$t3,-2                #tamanho do PAYLOAD usado pela rotina sort

        sw $t3, 0($t1)                  # marca  pacote como ocupado, gravando O TAMANHO do mesmo na primeira posição
                                        # da área de dados escolhida para receber o pacote
        xor $t0, $t0, $t0               #i=0

loop2:
        addiu $t0,$t0,1                 # "i++"
        addiu $t2,$t2,4                 # posiciona o $s0 para o payload (3o, 4o, 5o flits)

        lw $t4, 0($t2)                  # carrega o valor apontado por $t2
        sw $t4, 4($t1)                  # armazeno na memória  valor do flit recebido
        addiu $t1,$t1,4                 # avança na memória de dados

        bne $t0,$t3,loop2               # se não recebeu todo o pacote, volta a ler os flits
```

```
            la $t0, NI_RX_END_RECEPTION     # endereço do registrador que indica fim do recebimento do pacote
            li $t1, 1
            sw $t1, 0($t0)                  # avisa ao hw final de recebimento de pacote

loop5:
             lw $ra,0($sp)
             lw $t0,4($sp)
             lw $t1,8($sp)
             lw $t2,12($sp)
             lw $t3,16($sp)
             lw $t4,20($sp)
             lw $t5,24($sp)
             lw $1, 28($sp)
             addiu $sp,$sp,32

            mfc0 $a0,$14                    ## reservar o $a0 para o mfco
            jr $a0
 #-------------------------------------------------------------------------------------------------------------------------------------------------#
#    Retorna endereço pacote válido no registrador $v0 - ROTINA BLOQUEANTE                                                                        #
#-------------------------------------------------------------------------------------------------------------------------------------------------#
get_packet_adress:
            addiu $sp,$sp,-12
            sw $ra,0($sp)
            sw $t0,4($sp)
            sw $t1,8($sp)

            xor $t0, $t0, $t0
l0:         la $v0,PACKET_BASE_ADDRESS
            addu $v0, $v0, $t0              # endereço do ponteiro do pacote
            lw $v0, 0($v0)                  # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote em $v0
            lw $t1, 0($v0)                  # carrega a primeira palavra do pacote
            bne $t1,$zero, l1
            addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes
            andi  $t0, $t0, 0xF
            j l0

l1:         lw $ra,0($sp)
            lw $t0,4($sp)
            lw $t1,8($sp)

            addiu $sp,$sp,12
            jr $ra
#-----------------------------------------------------------------------------#
#                  Ordena o vetor recebido                                    #
#-----------------------------------------------------------------------------#
bubble:
            addiu $sp,$sp,-20
            sw $ra, 0($sp)
            sw $s0, 4($sp)
            sw $s1, 8($sp)
            sw $s2,12($sp)
            sw $s3,16($sp)

            addiu $s2,$v0,0                 # $v0 endereço do vetor a ser ordenado, neste cenario (VETOR)
            addiu $s3,$a3,0                 # $s3 = $a3 = qtd(tamanho do pacote)(passando 14)

            addu $s4,$s3,$0
            sll $s4,$s4,2

            li $s0, 0                       #i=0

for1:       slt $t0, $s0, $s3               #$t0=1 se $s0 < $s3 (i<qtd)
            beq $t0, $0, exit1              #salta se i<qtd

            li $s1,4                        #j=8 ->indice que vai indicar o inicio pacote valido

for2:
            slt $t0,$s1,$s4                 #$t0=1 se $s1<$s4 (j<k)
            beq $t0,$0,exit2                #vai para exit2 se j<k

            addu $t1,$s2,$s1                #$t1 <- &v[j]
            lw $t2,0($t1)                   #$t3 = v[j]
            lw $t3,4($t1)                   #$t4 = v[j+1]

            addiu $s1,$s1,4                 #j++

            slt $t0,$t3,$t2                 #$t0=1 se $t3>$t2 (v[j+1]>(v[j])
            beq $t0,$0,for2                 #vai para for2 se (v[j+1]>(v[j])

            lw $t0, 0($t1)                  #aux=v[j]
            sw $t3, 0($t1)                  #v[j] = v[j+1]
            sw $t0, 4($t1)                  #v[j+1]=aux


            j for2

exit2:      addiu $s0,$s0,1                 #i++
            addiu $s4,$s4,-4                #$s4 = k--
```

```
            j for1

exit1:      lw $ra, 0($sp)
            lw $s0, 4($sp)
            lw $s1, 8($sp)
            lw $s2,12($sp)
            lw $s3,16($sp)
            addiu $sp,$sp, 20

            jr $ra
#-------------------------------------------------------------------------------#
#                       Envia dados                                             #
#-------------------------------------------------------------------------------#
send:
            addiu $sp,$sp,-24
            sw $ra, 0($sp)
            sw $s0, 4($sp)
            sw $s1, 8($sp)
            sw $s2,12($sp)
            sw $s3,16($sp)
            sw $s4,20($sp)


            la $s1, NI_TX_BASE_ADDRESS      # carrega &rede
             la $t3, NI_TX_BUSY             # carrega $registrador  que indica NI ocupada
             la $t4, NI_TX_END_PACKET       # carrega &registrador  que indica fim do pacote

            la $t5,VET_TEMP

            lw $s0, 4($t5)                  # $s0 <- carrega o tamanho do pacote

l2:
            lw $t0, 0($t3)                  # verifica se a NI está livre(NI_TX_BUSY)
             bne $t0, $zero, l2

             li $t0, 0                      #i=0  $t0 será usado como deslocamento no vetor do pacote e do endereço do periféric
#                                                                    ##########################################
# CARGA DO TEMPO ATUAL, A SER INSERIDO NO FINAL DO PACOTE  ##########################################
#                                                                    ##########################################
            la $s5, CONT_TICK_H             # carrega o endereço da parte alta do contador de clock
            la $s6, CONT_TICK_L             # carrega o endereço da parte baixa do contador de clock
             lw $s5, 0($s5)
             lw $s6, 0($s6)                 # CARREGA EM $s5 e $s6 a base de tempo atual



#                                                                                 ####################################
# LAÇO PARA TRANSMISSÃO DO PACOTE ORIGINAL SEM A BASE DE TEMPO  ####################################
#                                                                                 ####################################
l3:
            addu $s3, $s1, $t0              #$s3 <- &rede(posição atual)
            addu $s4, $t5, $t0              #$s4 <- &temp_pacote armazenado em memoria (posição atual)

            lw $t1, 0($s4)                  # leio o elemento do pacote que está em memória do processador
            sw $t1, 0($s3)                  # gravo o elemento lido na rede

            addiu $t0,$t0, 4                # incrementa indice de deslocamento no vetor
            addiu $s0,$s0,-1                # decrementa size( tamanho do pacote atual)

            bne $s0, $zero, l3


#                                                                                          ########
# COLOCA O TEMPO PRÉ-CARREGADO NO FINAL DO PACOTE, lembrar: o size já foi incrementadio acima ########
#                                                                                           ########
            addiu $s3, $s3, 4               #incrementa &periferico
            sw $s5, 0($s3)                  #carrega parte alta do contador no final do pacote(penúltimo flit)
            addiu $s3, $s3, 4                #incrementa &periérico
            sw $s6, 0($s3)                  #carrega parte baixa do contador no final do pacote(último flit)


            li $s0, 1
            sw $s0, 0($t4)                  # envia estado de NI_TX_END_PACKET para indicar fim transmissão do pacote
#-------------------------------------------------------------------------------------------------------------#
#    As duas linhas abaixo fazem o papel do RELEASE_PACKET                                                    #
#-------------------------------------------------------------------------------------------------------------#
            xor $t0, $t0, $t0
            sw  $t0, 0($v0)

            lw $ra, 0($sp)
            lw $s0, 4($sp)
            lw $s1, 8($sp)
            lw $s2,12($sp)
            lw $s3,16($sp)
            lw $s4,20($sp)
            addiu $sp,$sp, 24

            jr $ra
```

```
#-------------------------------------------------------------------------------#
#                    Monta pacote para envio                                    #
#-------------------------------------------------------------------------------#
mont_packet:

        addiu $sp,$sp,-36
        sw $ra, 0($sp)
        sw $s0, 4($sp)
        sw $s1, 8($sp)
        sw $t0,12($sp)
        sw $t1,16($sp)
        sw $t2,20($sp)
        sw $t3,24($sp)
        sw $t4,28($sp)
        sw $t5,32($sp)

        addu $s0,$v0,$zero              #carrega o endereço do vetor que efetuado o sort
        la $s1,VET_TEMP

        lw $t3,0($s0)                   #tamanho do payload
        sw $t3,4($s1)                   #gravo o tamanho do payload em VET_TEMP

        addiu $t1,$0,0x000              #gravo o endereço do processador PE1
        sw $t1,0($s1)                   #de destino em VET_TEMP


        li $t0,4                        #indice de PACKET
        li $t2,8                        #indice de VET_TEMP

ret:
        addu $t4,$s0,$t0                #avança no vetor
        lw $t1,0($t4)                   #$t1<- o conteudo valido do payload
        addu $t5,$s1,$t2                #avança em VET_TEMP
        sw $t1,0($t5)                   #VET_TEMP <- conteudo valido do payload

        addiu $t0,$t0,4
        addiu $t2,$t2,4
        addiu $t3,$t3,-1                #decrementa tamanho para o loop

        bne $t3,$zero,ret

        lw $ra, 0($sp)
        lw $s0, 4($sp)
        lw $s1, 8($sp)
        lw $t0,12($sp)
        lw $t1,16($sp)
        lw $t2,20($sp)
        lw $t3,24($sp)
        lw $t4,28($sp)
        lw $t5,32($sp)
        addiu $sp,$sp,36

        jr $ra
#-------------------------------------------------------------------------------#
#                    Rotina Principal                                           #
#-------------------------------------------------------------------------------#
        .globl main

main:
#-------------------------------------------------------------------------------#
#          Interrupções são habilitadas                                         #
#-------------------------------------------------------------------------------#
        li $1,1
        mtc0 $1,$12

        jal get_packet_adress           # retorna o endereço do pacote em $v0

        li $a1,1

        jal bubble

        li $a1,0

        jal mont_packet

        jal send

        j fim

        addu $t5, $v0, $zero            # coloca o $v0 em $t5
#-------------------------------------------------------------------------------------------#
#               IMPRESSÃO DO VETOR NA ÁREA DEBUG                                            #
#-------------------------------------------------------------------------------------------#
debug:
        la $s0, DEBUG
        lw $t1, 0($t5)                  # primeiro elemento de $v0 é o tamanho do pacote lido
        li $t0, -1                      # começa em -1 pois vamos fazer o printf do tamanho do pacote também
loop4:
        lw $t2,0($t5)
```

```
            sw $t2,0($s0)
            addiu $t5,$t5,4
            addiu $t0,$t0,1         # i++
            bne $t0,$t1,loop4

            xor $t0, $t0, $t0                   #release packet
            sw  $t0, 0($v0)                     #para não pegar o mesmo pacote

fim:
            j main                              # reinicia o procedimento, VERIFICANDO se tem novo pacote
#------------------------------------------------------------------------------#
#                       Área de dados                                          #
#------------------------------------------------------------------------------#
.data

PACKET_BASE_ADDRESS:    .word  pck1 pck2 pck3 pck4
pck1:   .space 0x4c0
pck2:   .space 0x4c0
pck3:   .space 0x4c0
pck4:   .space 0x4c0

VET_TEMP:           .space 0x970

#------------------------------------------------------------------------------------#

.data 0xFFFF1000
NI_RX_BASE_ADDRESS: .space 0x970

.data 0xFFFF1984
NI_RX_PACKET_AVAILABLE: .space 4

.data 0xFFFF1980
NI_RX_END_RECEPTION: .space 4


.data 0xFFFF0000
NI_TX_BASE_ADDRESS: .space  0x970

.data 0xFFFF1978
NI_TX_BUSY: .space   4

.data 0xFFFF197C
NI_TX_END_PACKET: .space   4

.data 0xFFFF1988
CONT_TICK_L: .space 0x40

.data 0xFFFF19C8
CONT_TICK_H: .space 0x40

.data 0xFFFF1A08
DEBUG: .space  0x970
```