UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANTONIO DENILSON DE SOUZA OLIVEIRA

INTEGRANDO REDES EM CHIP À PLATAFORMAS MULTIPROCESSADAS: ESTUDO DE CASO STORM

MOSSORÓ – RN

2013

ANTONIO DENILSON DE SOUZA OLIVEIRA

INTEGRANDO REDE EM CHIP À PLATAFORMAS MULTIPROCESSADAS: ESTUDO DE CASO STORM

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação – associação ampla entre a Universidade do Estado do Rio Grande do Norte e a Universidade Federal Rural do Semi-Árido, para a obtenção do título de Mestre em Ciência da Computação.

Orientadora: Prof[a]. Dr[a]. Karla Darlene Nepomuceno Ramos.

MOSSORÓ – RN

2013

Catalogação da Publicação na Fonte.
Universidade do Estado do Rio Grande do Norte.

Oliveira, Antonio Denilson de Souza.

Integrando rede em chip à plataformas multiprocessadas: estudo de caso *Storm*. / Antonio Denilson de Souza Oliveira. – Mossoró, RN, 2013

63 f.
Orientador(a): Profª. Drª. Karla Darlene Nepomuceno Ramos.

Dissertação (Mestrado em Ciência da Computação). Universidade do Estado do Rio Grande do Norte. Universidade Federal Rural do Semi-Árido. Programa de Pós-Graduação em Ciência da Computação.

1. Redes em chip - Dissertação. 2. Interface - Dissertação. 3. MPSoC-Dissertação. I. Ramos, Karla Darlene Nepomuceno. II. Universidade do Estado do Rio Grande do Norte. III.Título.

UERN/BC CDD 004

Bibliotecária: Elaine Paiva de Assunção CRB 15 / 492

ANTONIO DENILSON DE SOUZA OLIVEIRA

INTEGRANDO REDE EM CHIP A UMA PLATAFORMA MULTIPROCESSADA: ESTUDO DE CASO STORM

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.

APROVADA EM: ___ / ___ / _____.

BANCA EXAMINADORA

_____

Karla Darlene Nepomuceno Ramos – UERN
Presidente

_____

Cláudia Maria Fernandes Araújo Ribeiro – UERN
Membro Interna

_____

Márcio Eduardo Kreutz – UFRN
Membro Externo

Dedico esse trabalho a todos que se fizeram presente durante essa caminhada. E principalmente aos meus avôs que Deus os tenham.

AGRADECIMENTOS

A dificuldade de escrever palavras vem sempre que começo, porém nesses parênteses que são os agradecimentos, não deixarei passar. Durante todo o decorrer desta pesquisa e trabalho tiveram momentos difíceis, momentos que cheguei a pensar que não conseguiria.

Todos nós passamos por momentos de dificuldade, porém poucos são aqueles que têm amigos verdadeiros para nos erguer diante dos obstáculos que a vida nos proporciona e foi exatamente isso que me permitiu seguir em frente.

Primeiramente queria agradecer a Profª. Karla Darlene, pelo seu comprometimento, conhecimento e amizade. Acho que seu comprometimento e dedicação foram fundamentais para a produção desse trabalho. Deixo também meus agradecimentos ao Laboratório de Engenharia de Software (LES) da Universidade do Estado do Rio Grande do Norte (UERN), em especial aos colegas do mestrado que motivaram durante todo período do mestrado, apesar de querer citar vários nomes não irei.

Agradeço aos meus pais, meus irmãos e amigos que de certa forma, contribuiram para a finalização dessa caminhada árdua. E por último, e não menos importante, minha namorada a quem amo muito, ALINE ALVES, por ter calma nas horas difíceis e entendimento em momentos oportunos, sem mais, deixo o meu MUITO OBRIGADO.

MUITO OBRIGADO, para esta que estará sempre em minha vida quando pensar na Universidade do Estado do Rio Grande do Norte (UERN), o Programa de Pós-graduação em Ciência da Computação (PPgCC) e seus coordenadores e membros. Peço também minhas sinceras desculpas e que espero um dia retribuir tudo que foi a mim concedido.

RESUMO

A capacidade de integração dos transistores possibilitou a redução das dimensões dos Circuitos Integrados (CIs), e como consequência é possível implementar um sistema completo em uma pastilha de silício, como é o caso de múltiplos processadores em um único chip (Multi Processor *System-on-Chip - MPSoC*). O MPSoC necessita de um mecanismo eficiente de comunicação, que pode ser desempenhado pelas redes em chip. A conexão entre o MPSoC e a rede em chip é realizada por meio de interfaces de rede (NI). Com o objetivo de investigar a utilização de diferentes redes em chip em um MPSoC, este trabalho apresenta a integração de uma rede em chip como alternativa de comunicação a um MPSoC. A integração foi implementada na plataforma STORM em SystemC TLM, utilizando a metodologia de mixagem de linguagens, permitindo a troca de sinais entre VHDL e SystemC. Para validar a integração de uma outra rede em chip à plataforma STORM foi utilizada uma aplicação de multiplicação de matrizes, além de uma estimativa de desempenho da interface. A aplicação foram simuladas em duas instâncias da plataforma STORM, uma com a NoCX4 e outra utilizando a rede em chip HERMES. Os resultados mostraram que é possível integrar diferentes redes em chip a um mesmo MPSoC, e que embora, em alguns casos, possa ocorrer perda de desempenho, em outros, o MPSoC pode utilizar serviços de comunicação distintos oferecidos por diferentes redes em chip.

Palavras-Chave: Redes em Chip, Interface, MPSoC

ABSTRACT

The integration capacity of the transistors allowed to reduce the dimensions of integrated circuits (ICs), and as a result it is possible to implement a complete system on a silicon wafer, such as multiple processors on a single chip (Multi - Processor System on - Chip - MPSoC). The MPSoC require an efficient mechanism of communication, which can be executed by networks on chip. The connection between the MPSoC network chip is performed by means of network interfaces (NI). In order to investigate the use of different networks on chip in a MPSoC, this paper presents the integration of a network on chip communication as an alternative to a MPSoC. The integration was implemented in STORM platform in SystemC TLM, using the methodology of mixing languages, allowing the exchange of signals between VHDL and SystemC. For validating the integration of a chip to another network platform an application STORM matrix multiplication is used, and an estimated performance of the interface. The applications were simulated in two instances of STORM platform, with each other and NoCX4 using the network HERMES. The results showed that it is possible to integrate different networks on the same chip MPSoC, and although in some cases, performance loss can occur, in others the MPSoC can use different communication services offered by different networks on chip.

Keywords: Network on chip, Interface, MPSoC.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SIGLAS

CBDA	*Centrally Buffered Dynamically-Alocated*
CIs	Circuitos Integrados
CMP	*Chip Multiprocessor*
FIFO	*First-In First-Out*
GALS	*Globally Asynchronous Locally Synchronous*
GPP	*General Purpose Processors*
GPU	*Graphics Processing Unit*
HOL	*Head-of-Line*
IP	*Intellectual Property*
Mp-SoC	*Multiprocessor System-on-Chip*
NI	*Network Interface*
OSI	*Open System Interconnection*
RISC	*Reducer Instrution Set Computer*
RT	*Register Transfer Level*
SAF	*Store-and-Forword*
SAM	*System Architeture Model*
SoC	*System-on-Chip*
SPARC	*Scalable Processor ARChiteture*
STORM	*Mp-SoC direcTory-based platfORM*
TLM	*Transaction Level Modeling*
VCT	*Virtual Cut-Through*
VHDL	*VHSIC Hardware Description Language*

SUMÁRIO

1. INTRODUÇÃO

Nos últimos anos, a capacidade de integração dos transistores possibilitou a redução das dimensões dos Circuitos Integrados (CIs), confirmando a Lei de Moore (MOORE, 1965). Como consequência, atualmente é possível inserir um sistema completo em uma única pastilha de silício, também conhecido, em inglês, por *System-on-Chip* - SoC (BERGAMICHI; LEE, 2000).

Um SoC permite a utilização de componentes heterogêneos, tais como GPUs, memórias, barramentos, entre outros. Além disso, é possível que um único SoC seja formado por mais de um elemento de processamento. Os SoCs que empregam vários elementos de processamento são chamados de MPSoC (do inglês, *Multiprocessor System-on-Chip*).

Os MPSoCs podem adotar diferentes infraestruturas de comunicação, tais como barramento e redes em chip, que influenciam no desempenho geral do sistema. Sistemas de uso intensivo de dados, tais como, dispositivos multimídia e plataformas multiprocessadas necessitam de esquemas de interconexão flexível e escalável para lidar com uma quantidade elevada de transações de dados no chip. Quando o barramento é utilizado, à medida que mais módulos são adicionados, o desempenho do sistema como um todo é degradado (REGO, 2006). A infraestrutura de barramento permite apenas uma comunicação de cada vez de acordo com o resultado de arbitragem, assim, a largura de banda média de comunicação de cada elemento de processamento é inversamente proporcional ao número total de núcleos IP (*Intellectual Property*) (GUERRIER; GREINER, 2000). Assim, uma arquitetura baseada em barramento é intrinsecamente não escalável não sendo adequada para um sistema complexo como MPSoC.

Em redes em chip os problemas encontrados nos barramentos não acontecem, porque as conexões são do tipo ponto-a-ponto, e, dessa forma, o desempenho independe do número de módulos conectados à rede. Com o aumento no número de núcleos IP interligados aos atuais sistemas e com o avanço no uso de MpSoCs, torna-se cada vez mais viável usar uma rede de interconexão em substituição aos barramentos, já que

utilizando uma rede em chip é possível obter uma comunicação escalável sem a degradação do desempenho (DALLY; TOWLES 2001).

No contexto dos MPSoC, observa-se características genéricas de sistemas multiprocessados computacionais de propósito geral, que incluem diversos desafios. Destacam-se a dificuldade de programação paralela, a necessidade por balanceamento de carga, visando um melhor aproveitamento das unidades de processamento, além dos mecanismos de comunicação.

No entanto, para que uma rede em chip seja utilizada como infraestrutura de comunicação nos MPSoCs, são necessárias interfaces de rede (NI, do inglês, *Network Interface*) para adaptar o protocolo dos núcleos IP à rede em chip. Além disso, a interface é responsável por empacotar e desempacotar as mensagens e realizar o controle de fluxo entre outros serviços funcionais. Na literatura existem trabalhos que apresentam propostas de interface de rede como apresentado por Ebrahimi et al. (2009), Matos et al. (2010) e Melo (2012), as quais têm suas peculiaridades.

Ebrahimi et al. (2009) propuseram uma interface de rede baseada no padrão AMBA AXI (*Advanced eXtensible Interface*). Nesse padrão os módulos são descritos como mestre (*master*) ou escravo (*slave*). Quando definidos como *mestre* são responsáveis por iniciar as transações com solicitações aos módulos *escravos*, os quais recebem as solicitações e executam cada uma, retornando um sinal de confirmação ou dados. Essa proposta fornece uma alocação dinâmica de *buffer* conforme a variação do tamanho do pacote, já que mensagens de resposta e solicitação podem apresentar tamanhos diferentes.

Matos et al. (2010) desenvolveram um projeto de uma interface de rede como mecanismo de sincronismo de dados. A principal contribuição do trabalho é o sincronismo de dados, que foi projetado para ser expansível e escalável, dependendo da quantidade de núcleos origem que enviam dados para um mesmo núcleo destino. Para a sincronização, são usadas FIFO estáticas para cada núcleo origem, cuja finalidade é a sincronização dos dados por parte dos núcleos.

Melo (2012) desenvolveu uma interface de comunicação extensível para a rede em chip SoCIN, a qual denominou de XIRU (acrônico da expressão: *eXtensible InteRface Unit*). A XIRU é uma interface de rede que utiliza uma arquitetura em

camadas a fim de facilitar a geração de variantes compatíveis com diferentes protocolos intrachip e a inclusão de novos serviços à interface. A especificação da interface foi realizada considerando as características de comunicação e sinais comumente encontrados nos padrões de barramento OCP, AMBA AHB, AMBA AXI, Avalon e CoreConnect.

Assim buscando a utilização de diferentes redes em chip a uma plataforma, este trabalho busca desenvolver uma de interface de rede. A interface promove serviços comumente empregrados nas SoCs, como de adapatação, interfaceamento, empacotamento e desempacotamento, destacando-se pelo seu desenvolvimento modular utilizando camadas de interfaces parametrizáveis.

Na Tabela 1, observa-se que cada interface é compatível com um único padrão de comunicação, com exceção do trabalho de Melo(2012). O desempenho da utilização dessas interface foi expresso com base na frequência de operação máxima atingida e no impacto causado pelas interfaces na latência de comunicação. Neste trabalho foi utilizado a latência como métrica de desempenho.

Tabela 1. Padrões de Interconexão e métricas utilizada na avaliação dos trabalhos

Trabalho	**Ano**	**Barramento**	**NoC**	**Métricas**
				Desempenho
Ebrahimi et al.	2009	AMBA AXI	Proprietária	Latência
Matos et al.	2010	Proprietário	SoCIN	Frequência de Operação
Melo	2012	Avalon(Extensível)	SoCIN	Frequência de Operação
Este trabalho	2013	Proprietário	HERMES	Latência

A plataforma STORM (MP-**S**oC Direc**T**ory-Based Platf**ORM**)(REGO, 2006) é um ambiente dedicado ao projeto de MPSoC, capaz de permitir a avaliação de desempenho de aplicações embarcadas. A partir da STORM, o presente trabalho integra

uma nova rede em chip à plataforma STORM, a qual pode utilizar uma infraestrutura de comunicação distinta daquela desenvolvida exclusivamente para interligar e realizar comunicação entre os componentes do chip, denominada NoCX4. Portanto, foi estudado como a STORM envia e recebe pacotes com o objetivo de definir uma interface, que utiliza os serviços básicos de empacotamento e desempacotamento. De acordo com as características da plataforma STORM, a interface foi definida em SystemC TLM juntamente com a metodologia de mixagem de linguagens (RUDRA; GAURAV; SACHIN, 2006), para permitir a integração da plataforma com uma rede em chip especificada em VHDL (VHSIC Hardware Description Language).

Como estudo de caso, para validar a integração da rede, utilizou-se a simulação de uma aplicação de multiplicação de matrizes e uma estimativa do custo da interface. O projeto e simulação foram escritos na ferramenta Aldec Active-HDL, realizando uma co-simulação VHDL-SystemC. A partir desta implementação, pôde-se também explorar diversas características e necessidades dos MPSoCs, tais como, otimização das interfaces de rede, serviços adicionais que as interfaces devem oferecer para adequar a requisitos de rede em chip e parametrização.

Os resultados mostram que a interface permite que a STORM utilize diferentes redes em chip e que a mesma possa oferecer suporte a serviços de comunicação anteriormente não suportados pela NoCX4, como por exemplo suporte aos serviços de qualidade de serviço.

1.1. OBJETIVO

Visando investigar os requisitos necessários para uma plataforma multiprocessada utilizar diferentes redes em chip como infraestrutura de comunicação, esta pesquisa tem como objetivo integrar uma rede em chip a um MPSoC que já possui sua própria infraestrutura de comunicação, permitindo uma maior flexibilidade aos MPSoC.

Para atingir o objetivo supracitado foi necessário investigar os serviços de comunicação que as redes em chip podem oferecer e desenvolver uma interface de acordo com os atributos de comunicação definidos pela infraestrutura de interconexão do MPSoC.

1.2. ESTRUTURA DO DOCUMENTO

Além deste capítulo introdutório, o presente trabalho possui mais seis capítulos. O Capítulo 2 apresenta uma visão geral sobre sistemas multiprocessados, redes em chip e interfaces de rede, que constituem a fundamentação teórica desta pesquisa.

O Capítulo 3 apresenta as configurações realizadas na plataforma MPSoC para a integração da rede em chip. É detalhado a plataforma STORM e as redes em chip adotadas no testes, além da interface desenvolvida para a integração da rede em chip.

No Capítulo 4 são apresentados os resultados das aplicações que foram simuladas na plataforma STORM com duas instâncias, uma com a rede em chip HERMES e outra utilizando a NOCX4.

O capítulo 5 apresenta as considerações finais e trabalhos futuros.

2. SISTEMAS MULTIPROCESSADOS

Neste Capítulo serão apresentados os fundamentos relacionados aos sistemas multiprocessados, às redes em chip e interface de rede. A Seção 2.1 trata dos sistemas multiprocessados em um único chip. A Seção 2.2 e 2.3 apresentam respectivamente a definição de redes em chip e os principais mecanismos de comunicação utilizados na infraestrutura de comunicação. Na seção 2.4 é apresentado o conceito de interface de rede e seus serviços.

2.1. MÚLTIPLOS PROCESSADORES EM UM ÚNICO CHIP

Um MPSoC é um sistema, que incorpora muitos ou todos componentes necessários para uma aplicação que utiliza múltiplos processadores como componentes do sistema. Em geral, os sistemas multiprocessados são compostos por processadores embarcados, memórias, dispositivos de entrada e saída, interfaces, entre outros. Os MPSoC constituem um ramo específico e importante de multiprocessadores. São projetados para satisfazer os requisitos específicos de aplicações embarcadas.

Os MPSoC podem ser classificados como: homogêneos ou heterogêneos (JERRAYA; TENHUNEN; WOLF, 2005). Quando os elementos de processamento são todos de mesma natureza são ditos homogêneos. Por exemplo, um sistema composto por processadores idênticos que permitem exclusivamente a execução de tarefas para uma arquitetura específica. Quando esses sistemas são compostos por elementos de processamento diferentes, tais como GPP (*General Purpose Processors*), GPU (*Graphics Processing Unit*), entre outros, é denominado heterogêneo. Enquanto MpSoCs homogêneos simplificam a aplicações de técnicas de migração de tarefas, MpSoCs heterogêneos podem suportar uma maior variedade de aplicações.

O primeiro MPSoC desenvolvido foi o *Lucena Daytona* (ACKLAND et al., 2000). Projetado para estações de base sem fio. O *Daytona* possui uma arquitetura simétrica com quatros processadores ligados ao barramento de alta velocidade. Os processadores são baseados no SPARC V8 melhorado. Os processadores compartilham

um endereçamento de memória e utilizam memórias caches.

Dois MPSoC de destaque nos últimos anos foram o da Intel (VANGAL et al., 2007) e o da Tilera (CORPORATION, 2007), os quais possuem 80 e 64 núcleos idênticos respectivamente. Ambos utilizam redes em chip como infraestrutura de comunicação.

É importante ressaltar que MPSoC não é o mesmo que CMP (*Chip Multiprocessors*). Enquanto que CMP adotam as vantagens do aumento da densidade de transistores para incluir mais processadores, o MPSoC, ao contrário, busca equilibrar as restrições da tecnologia VLSI com as necessidades de aplicações (JERRAYA; TENHUNEN; WOLF, 2005).

2.2. REDES EM CHIP

Uma rede em chip consiste de elementos de processamento computacional (PE), redes, interfaces e roteadores. Os dois últimos compõem a arquitetura de comunicação (Figura 1). A interface de rede é usada para empacotar os dados antes de ser enviado para o roteador. Cada PE está ligado a uma interface de rede, que realiza a comunicação com os roteadores.

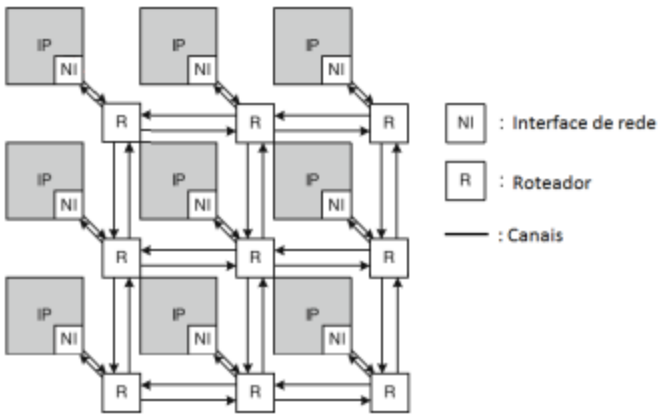

Figura 1. Visão geral de uma rede em chip. Adaptada de (MATOS, 2010)

Cada roteador possui um conjunto de portas usadas para conectar os roteadores vizinhos aos módulos de processamento, memória e dispositivos de entrada e saída do sistema. As portas utilizadas para conectar o roteador a um dos núcleos do MPSoC são

chamadas de portas locais ou terminais. A flexibilidade da rede deve permitir a conexão de núcleos de diferentes naturezas, ou seja, GPUs, memórias, dispositivos de entrada e saída.

O roteador é o principal bloco construtivo de uma rede em chip cuja finalidade é encaminhar mensagens através da rede. É formado por um conjunto de *buffers* e multiplexadores, além de blocos controladores que implementam os mecanismos de comunicação necessários à transferência de mensagens pela rede. Os roteadores podem ser construídos de maneira centralizada ou distribuída.

Os roteadores e núcleos em uma rede em chip são interligados por meio de canais ponto-a-ponto, que transportam os dados da mensagem. As mensagens com tamanhos maiores que a palavra do canal devem ser quebradas e transferidas como sequência de palavras, formando os pacotes.

2.2.1. CARACTERIZAÇÃO DE UMA REDE EM CHIP

Uma rede em chip é caracterizada pela topologia e pelos mecanismos de comunicação utilizados. A topologia define a organização dos roteadores na rede. Os mecanismos de comunicação, a saber: roteamento, chaveamento, controle de fluxo, arbitragem e memorização definem como os pacotes trafegam pela rede. Nesse escopo, os tópicos seguintes conceituam topologia e os mecanismos de comunicação.

Topologia

A topologia de uma rede em chip determina como os recursos de rede estão ligados, refere-se assim ao arranjo estático de canais e roteadores em uma rede de interconexão (KUMAR et al., 2002).

De acordo com a topologia, as redes em chip podem ser diretas e indiretas. Nas redes diretas, cada roteador está conectado a um elemento de processamento, logo todos os roteadores são fontes e destino de tráfego. Em uma topologia indireta, os roteadores são diferentes dos núcleos de processamento, apenas os nodos terminais são fontes e destinos do tráfego. A Figura 2 ilustra as topologias diretas mais

utilizadas. Dentre elas, a topologia malha (Figura 2a) e toro (Figura 2b).

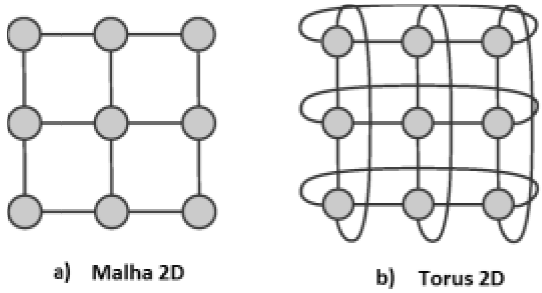

Figura 2. Topologias Diretas. Adaptada de (ZEFERINO, 2003).

As topologias *crossbar* e a rede multiestágio são exemplos de topologias indiretas (PALMA, 2007). C*rossbar* é uma rede onde cada roteador possui uma chave NxN para o chaveamento, porém sua escalabilidade é limitada.

As redes multiestágios são compostas de vários estágios de roteadores idênticos, onde estão conectados a outros roteadores e/ou com nodos do sistema. Essas redes são mais escaláveis, porém não é possível a comunicação ao mesmo tempo com todos os módulos (PALMA, 2007). A Figura 3 mostra uma rede *crossbar* (Figura 3a) composta por roteador 3x3 e uma rede multiestágio 8x8 (Figura 3b).

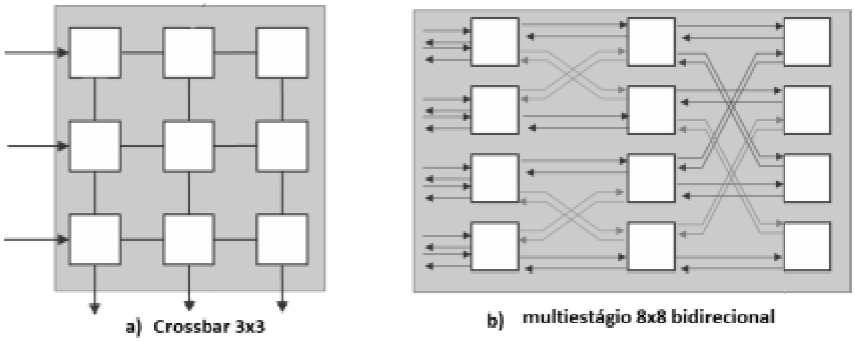

Figura 3. Redes indiretas. Adaptada de (DALLY; TOWELL, 2004).

Controle de Fluxo

O controle de fluxo faz o gerenciamento dos *buffers* e dos canais. Ele determina quando os *buffers* e os enlaces são atribuídos às mensagens, a granularidade da qual elas são alocadas e como esses recursos são compartilhados (JERGER; PEH, 2009). O controle de fluxo deve tomar decisões sobre o destino dos pacotes. Adotando alguma

política de controle para decidir se o pacote deve ser descartado, bloqueado, recebido e armazenado temporariamente ou desviado.

Normalmente as redes em chip realizam o controle de fluxo em nível de enlace. Em cada terminal do enlace, os nodos podem possuir áreas de armazenamento do dado transmitido, normalmente sob a forma de *buffers* FIFO (*First-In First-Out*).

No caso das redes em chip, existem três mecanismos mais utilizados na implementação do controle de fluxo: *handshake*, *baseado em créditos* e *baseado em canais virtuais.*

O controle de fluxo *handshake* ("aperto de mão") (TOMLINSON, 1975) funciona por meio de diretivas *ack/Nack* onde o emissor ativa um sinal (*tx/valid*) indicando que o *flit* está sendo transmitido através do canal. Ao chegar o dado no receptor, o mesmo sinaliza (*ack*) se houver espaço disponível, caso contrário, envia o sinal de espaço indisponível (*nack*).

No controle de fluxo *baseado em crédito* (YAN; MiN; AWAN, 2008), uma transmissão só ocorre se existir créditos no buffer receptor. O mecanismo funciona da seguinte forma:

1. O receptor envia uma informação de crédito relativa ao espaço de *buffer* para o transmissor.
2. Se existir crédito suficiente, a mensagem é enviada e o número de créditos é diminuído.
3. Se a mensagem for encaminha adiante pelo receptor, o número de créditos aumenta novamente.

O controle de fluxo *baseado em canais virtuais* (DALLY, 1992) consiste em dividir logicamente o *buffer* de entrada em filas independentes de profundidades menores que formaram canais virtuais, visando resolver o problema de bloqueio de cabeça de linha, também chamado de bloqueio HOL (do inglês, *Head of Line*). O HOL ocorre quando o pacote da cabeça da fila fica bloqueado, como consequência da não disponibilidade de créditos no buffer destino. Caso existam pacotes destinados a outras portas, eles ficarão bloqueados até que o primeiro pacote seja transmitido.

Roteamento

O roteamento determina o caminho a ser percorrido pelo pacote através da rede. O objetivo do algoritmo de roteamento é distribuir uniformemente o tráfego entre os caminhos fornecidos pela topologia da rede, de forma a minimizar a contenção, melhorando a latência e a vazão (JERGER; PEH, 2009).

Os algoritmos de roteamento podem ser classificados quanto à localização da implementação do algoritmo: na fonte, centralizado e no roteador (MICHELI; BENINI, 2006).

Quando implementado na fonte, o caminho já é fixado antes do envio do pacote. No roteamento centralizado os caminhos são definidos por um controlador central. Já no roteamento distribuído, o caminho é definido ao longo do percurso pelos roteadores da rede.

A outra classificação é quanto sua adaptabilidade às condições da rede no momento da transmissão, o algoritmo de roteamento pode ser classificado como determinístico ou adaptativo. O determinístico são aqueles que não se adaptam às condições da rede, ou seja, o caminho é conhecido a priori. No roteamento adaptativo, o caminho é definido ao longo do percurso visando a melhor distribuição dos pacotes e evitando congestionamento. Os algoritmos mais simples são os determinísticos como o caso do algoritmo XY. Há também algoritmos parciamente adapatativos, como *west-first* e *negative-first*, que permitem que os pacotes sejam roteados por qualquer caminho curto. Os algoritmos completamente adaptativos, como (GLASS, 1992) e (LI, 2006), são definidos em função do tráfego da rede buscando o caminho mais curto corrente.

No algoritmo de roteamento XY (DUATO, 1997), os pacotes são enviados primeiramente na dimensão X e depois na dimensão Y, sendo que apenas uma mudança de dimensão é permitida. Quando um pacote chega a um nodo, o valor de deslocamento de X é calculado com base no nodo corrente e no destino, caso seja nulo o pacote é roteado na dimensão Y (*North ou South*). Se o deslocamento de Y quanto o de X for nulo o pacote é roteador para a porta local. E se o deslocamento de X for diferente de nulo, o pacote é transmitido através da dimensão X (*West*/*East*) até de o deslocamento seja nulo. São exemplos de redes em chip que utilizam esse roteamento: a QNoC (BOLOTIN et al. 2004) e a NoCX4 (REGO, 2006).

O algoritmo *west-first* é uma adaptação do algoritmo determinístico XY onde os pacotes são roteados dependendo do fonte e destino. Quando o valor do eixo X do destino é menor ou igual ao eixo X da fonte, os pacotes são roteados pelo XY, caso contrário, os pacotes são roteados de forma adaptativa nas direções *East*, *North* ou *South*. A HERMES-TB (SCHERER, 2007), utiliza esse tipo de roteamento além da topologia toro bidimensional.

No algoritmo *negative-first* os pacotes são roteados primeiramente na direção negativa, fazendo o uso da abreviatura X_T para o destino e X_S para origem. Quando $X_T \leq X_S$ e $Y_T \geq Y_S$ ou o inverso, $X_T \geq X_S$ e $Y_T \leq Y_S$ o roteamento é determinístico. Nas outras possibilidades o roteamento é adaptativo.

Arbitragem

A arbitragem é fundamental para resolução de conflitos decorrentes da existência de múltiplos pacotes competindo por uma mesma porta de saída. O mecanismo de arbitragem deve ser capaz de resolver esses conflitos, selecionando um dos pacotes com base em algum critério, evitando que o pacote fique bloqueado indefinidamente (em inglês, *starvation*). A arbitragem pode ser centralizada ou distribuída. Na centralizada, existe um comutador central que recebe todas as requisições dos *buffers* das portas de entrada de cada roteador e realiza a seleção das portas. Na distribuída cada roteador irá dispor de um comutador que será responsável pela escolha das portas para que seja feita a transmissão.

Chaveamento

O Chaveamento define a forma pela qual os pacotes são transmitidos de um canal de entrada de um nodo para um dos seus canais de saída (ZEFERINO, 2003). Algumas técnicas que são utilizadas no chaveamento de pacotes em redes em chip. As principais são: chaveamento por circuito, por pacotes, *Virtual Cut-Through* (VCT) e *wormhole*.

O chaveamento por circuito consiste basicamente na alocação exclusiva do canal entre a fonte e o destino. Essa técnica é mais utilizada para aplicações que requer baixa comunicação entre os módulos e é realizada em duas etapas. Na primeira, o nodo

envia para a rede um cabeçalho de roteamento contendo informações de controle. Assim, o cabeçalho vai percorrendo a rede e alocando os canais para a comunicação. Quando o pacote chega ao destino, um sinal de controle é enviado até o emissor pelo canal alocado. Nesse ponto, a começar a transmissão dos dados (segunda etapa) e ao final da transmissão um terminador enviado pelo emissor começa a desfazer os canais alocados.

Já o chaveamento por pacotes, consiste na divisão das mensagens em várias partes menores, chamados de pacotes, que são enviadas pela rede seguindo vários caminhos diferentes. O VCT segue o mesmo esquema do SAF (*Store-and-Forword*), a única diferença está em direcionar os pacotes para portas que estiverem disponíveis. No pior dos casos, quando todas as portas de saídas estiverem ocupadas, o VCT irá funcionar como o SAF, sendo necessário armazenar todos os pacotes nos *buffers*.

O *wormhole* é uma otimização do VCT que consiste em dividir o pacote em unidade menores chamadas de *flit* (***fl****ow control un****it***) e transmiti-los pela rede. Nesse tipo de chaveamento os *buffers* irão guardar os *flit* ao invés de pacotes, e assim diminuir o tamanho do *buffer*.

Memorização

Em geral, o roteador que utiliza o chaveamento por pacotes, deve ser capaz de armazenar os pacotes destinados a saída que está bloqueada por outros pacotes e, então, realizar o controle de fluxo de modo a evitar a perda de dados.

Existem três técnicas básicas de memorização que podem ser utilizadas em roteadores: Memorização *centralizada compartilhada*, *memorização na entrada* e *memorização na saída*.

Na memorização centralizada compartilhada, utiliza-se um *buffer* centralizado para armazenar os pacotes bloqueados em todos os canais de entrada e o espaço de endereçamento é dinamicamente distribuído entre os pacotes bloqueados. Esse *buffer* é denominado CBDA (do inglês: *Centrally Buffered, Dynamically-Allocated*).

Na memorização realizada na entrada, o espaço de memória é distribuído sob a forma de partições entre os canais de entrada do roteador. Consiste em *buffers* independentes em cada uma das portas de entrada do roteador. Existem várias abordagens para essa técnica, porém a mais utilizada é a FIFO. O *buffer* FIFO (*First In*

Firt Out) é o mais comum e simples, possui um espaço de memória fixo onde os dados são lidos na mesma ordem em que são escritos. O problema desta abordagem é o bloqueio de HOL (*Head of Line)*.

A memorização na saída consiste em colocar *buffers* entre as saídas dos roteadores. Esses *buffers* devem suportar as N entradas simultaneamente. Pode ser implementado com várias portas de escrita ou uma porta escrita mais rápida do que as entradas.

2.3. INTERFACES DE REDE

As interfaces de rede são indispensáveis quando se faz o uso de redes em chip. A Figura 4 ilustra os principais módulos que compõem uma interface de rede (BERTOZZI, 2006). De acordo com MICHELI e BENINI (2006) as funções de uma interface de rede são:

- Realizar o controle de fluxo
- Adaptar o protocolo de envio e recebimento de mensagens entre os núcleos e a rede em chip.
- Dividir as mensagens em *flits* ou pacotes que devem ser enviadas pela rede. Aos pacotes ou *flits* são adicionados todos os controles necessários ao envio dos mesmo.
- Receber os *flits* da rede e reconstruir as mensagens a ser entregues aos núcleos. Removendo informações de roteamento e demais serviços.

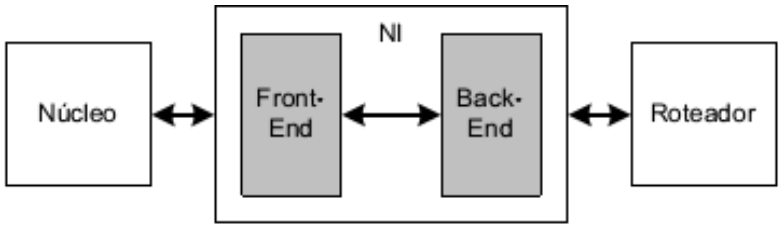

Figura 4. Estrutura de uma interface de rede. (MELO, 2012)

Na Figura 4, os módulos de *front-end* e *back-end* são responsáveis pelas funções descritas por MICHELI e BENINI (2006). O *front-end* realiza a comunicação com os núcleos e *back-end* a comunicação com a rede em chip.

O modelo OSI (*Open System Interconnection*) consiste em uma descrição de sistemas de comunicação baseada em sete camadas hierárquicas, são elas: camada

física, camada de enlace de dados, camada de rede, camada de transporte, camada de sessão, camada de apresentação e camada de aplicação. Considerando o contexto de rede em chip as camadas do modelo OSI se relacionam da seguinte forma:

- Camada física: Define parâmetros elétricos dos sinais, direção dos sinais e largura de dados;
- Camada de enlace: Questões de confiabilidade na transferência de dados e protocolos de comunicação, *handshake*, por exemplo.
- Camada de rede: Define o chaveamento e roteamento. Determina a conexão entre origem e destino
- Camada de transporte: Estabelece o controle de fluxo, define algoritmos de empacotamento e desempacotamento dos pacotes.
- Camada de sessão, apresentação e aplicação: Estratégias de controle de fluxo e negociações de QoS também podem ser implementadas na camada de sessão.

A camada de *front-end* (Figura 5) pode ser vista como a implementação da camada de sessão do modelo de referência. Estratégias de controle de fluxo e negociações de QoS também podem ser implementadas nessa camada.

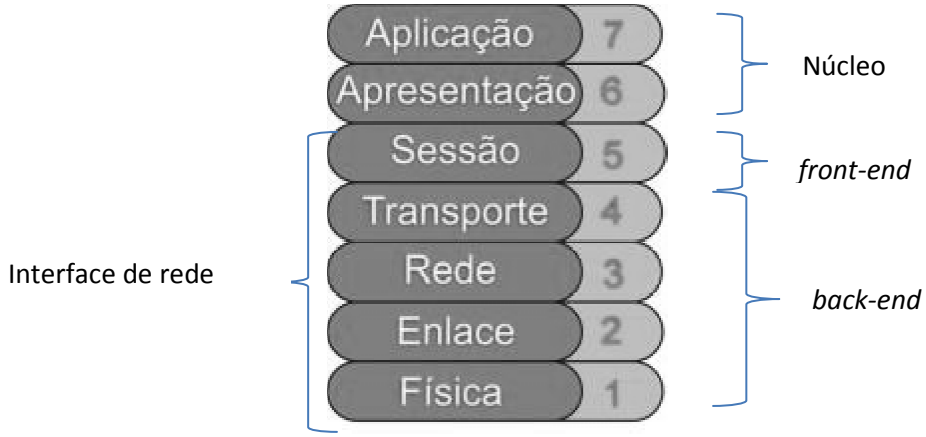

Figura 5. Relação entre modelo OSI e uma interface de rede. Adaptada de Bertozzi(2006)

A camada de *back-end* (Figura 5) integra a camada de transporte, rede, enlace e física do modelo de referência OSI. A camada de transporte suporta a transferência de dados confiáveis às camadas superiores do modelo. A camada de rede realiza a

transferência entre os nodos finais. Esta camada também realiza funções de roteamento e empacotamento de dados.

SERVIÇOS

De acordo com Bertozzi (2006), os serviços apresentados pelas interfaces de rede são classificados em quatros tipos: Adaptação, relógio, Rede, e Funcionais. Conforme apresentado na Tabela 2.

Tabela 2. Serviços básicos oferecidos pela interface de rede.

Serviços	Características
Adaptação	Interfaceamento, Empacotamento e desempacotamento.
Relógio	GALS, Adaptação de fase, Garantia de Latência, Garantia de Largura de banda.
Rede	Reordenamento de pacotes, Controle de integridade, Controle de fluxo.
Funcionais	Coerência de Cache, Métricas de Segurança e Baixa Potência.

Os serviços de adaptação ajustam o protocolo de comunicação do componente ao protocolo de comunicação da rede. Os principais serviços de adaptação são o interfaceamento de núcleo, buscando oferecer reúso e portabilidade, e o empacotamento, que transforma os sinais de entrada provenientes de um núcleo em pacotes construtores para a comunicação com o protocolo da rede.

A distribuição de relógio tornou-se um dos maiores desafios dos futuros SoCs, sendo que a técnica predominantemente adotada é a de sincronismo local e assincronismo global, também conhecida por GALS (*Globally Asynchronous Locally Synchronous*). Ainda que a frequência seja a mesma por todo o chip, adaptações de fase são necessárias para a correta comunicação. Se a rede em chip implementa garantias de latência ou largura de banda, então a interface de rede deve gerenciar esse serviço.

Os serviços de rede são implementados pela camada de transporte na abordagem de redes de computadores. O emprego, ou não, desses serviços depende fortemente das características da rede em uso. Um serviço adotado é o ordenamento de transações para repassar a informação ordenada ao componente, tendo como penalidade o alto consumo

de elementos de memória. Outro serviço possível de adoção é a de transações confiáveis, delegando à interface de rede a função de assegurar um meio de comunicação livre de erros em sistemas com tecnologias de grande complexidade. Também é utilizado o controle de fluxo, responsável por regular o fluxo de pacotes através da rede e tratar os acúmulos localizados de tráfego.

Os serviços funcionais adicionam novas funcionalidades ao sistema. Muitas alternativas podem ser implementadas em outras unidades, mas em alguns casos seu emprego na interface de rede atinge o melhor desempenho. Dentre as diversas alternativas de serviços funcionais disponíveis destacam-se a coerência de cache, técnicas de segurança (manipulação de informação privada, prevenção de pirataria de dados, etc.) e baixa potência (uso de componentes de baixo consumo, técnicas para redução da atividade de chaveamento, etc.).

3. INTREGRANDO UMA REDE EM CHIP A UMA PLATAFORMA MULTIPROCESSADA

Este Capítulo descreve a interface desenvolvida para possibilitar a integração da plataforma STORM a outras redes em chip. Portanto, este capítulo está dividido em seis seções. A primeira seção diz respeito à plataforma STORM apresentando suas características e infraestrutura de comunicação denominada NoCX4. Em seguida, será detalhada a rede em chip HERMES, que será usada como infraestrutura de comunicação em substituição à rede NoCX4. A Seção 3.3 trata da integração da rede em chip HERMES à plataforma STORM. As seções seguintes descrevem a interface desenvolvida para tornar possível a comunicação da STORM com uma outra rede em chip diferente da NoCX4.

3.1. PLATAFORMA STORM

A plataforma STORM foi desenvolvida visando o estudo das características dos MPSoCs para explorar mecanismos de comunicação utilizando as redes em chip, além do processamento paralelo. De acordo com (OLIVEIRA e SILVA, 2010) a STORM possibilita a utilização de qualquer rede em chip, desde que mantida a interface do roteador com os módulos.

A plataforma STORM não possui uma arquitetura definida, mas sim um conjunto de módulos e especificações sobre seu uso, permitindo ao projetista criar diversas instâncias de arquiteturas com características diferentes.

Quanto ao processamento, a plataforma utiliza um processador SPARC (*Scalable Processor ARChitectureI*). Este processador foi desenvolvido pela Sun Microsystems e possui arquitetura RISC (*Reduced Instruction Set Computer*). A plataforma também permite a utilização de outros processadores desde que mantida a interface com a memória cache.

A STORM foi projetada para utilizar tanto o modelo de memória compartilhada como o de memória distribuída. No modelo de memória compartilhada (Figura 6) um ou mais módulos de memória estão distribuídos entre os nodos da rede em chip e fornecem um espaço de endereçamento único. Deste modo os processadores podem

acessar qualquer endereço de memória utilizando variáveis compartilhadas, que são protegidas via *mutexes* para garantir a exclusão mútua.

No modelo de memória distribuída, cada nodo da rede em chip tem um processador com seu próprio módulo de memória. Um processador só tem acesso ao seu endereçamento de memória e a comunicação é realizada por troca de mensagens.

A interface de comunicação utilizada na STORM com memória compartilhada é formada de dois *buffers,* denominados: *Send_buffer* e *Receiving_buffer*. Como os próprios nomes sugerem, o primeiro *buffer* é responsável por enviar os pacotes na rede, e o segundo por recebê-los.

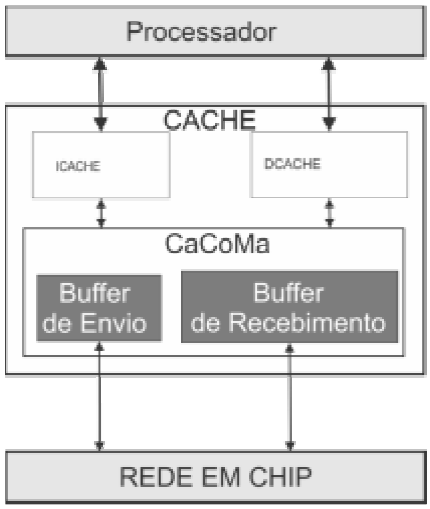

Figura 6. Modelo de Memória Compartilhada. Adaptada de (OLIVEIRA et al. 2010).

O sistema de comunicação é um dos principais aspectos no desenvolvimento dos SoCs. Por esse motivo a plataforma adota uma NoC, denominada NoCX4, como infraestrutura de comunicação(REGO; SILVA; AZEVEDO FILHO, 2004).

NOCX4

A NoCX4 é composta por uma matriz de roteadores. Cada roteador pode possuir até 5 portas: norte, sul, leste, oeste, para a comunicação com outros roteadores vizinhos e uma porta local para conectá-los com um módulo da plataforma, ou seja, consiste de uma topologia direta. Os pacotes que trafegam pela rede são compostos por palavras de 32 bits, que consistem em cabeçalhos e carga efetiva. O formato do pacote é apresentado na Figura 7. O pacote é distribuído da seguinte forma: 8 bits para Origem, 8

para destino, 11 *flags* utilizados para serviços adicionais e 5 bits para o tamanho do *payload*.

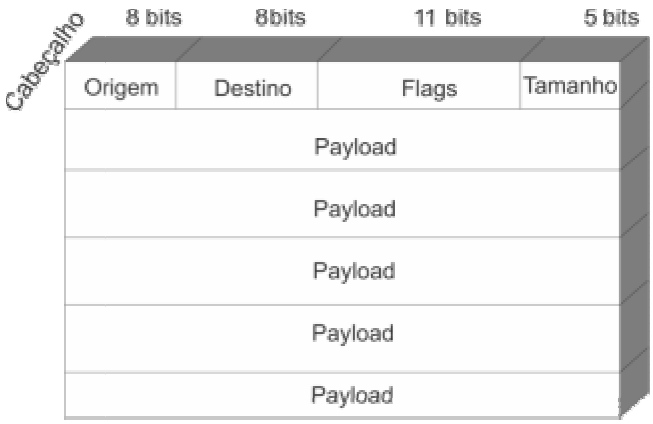

Figura 7. Formato do pacote da NoCX4. Adaptada de (OLIVEIRA et al. 2010).

A rede em chip utilizada na plataforma STORM é a NoCX4 que de acordo com o Capítulo 2 apresenta as seguintes características:

- Controle de fluxo baseado em créditos
- Roteamento dimensional XY determinístico
- Arbitragem Round-Robin;
- Chaveamento VCT;
- Armazenamento FIFO de tamanho 9;

3.2. HERMES

A rede HERMES é caracterizada pelo seu roteador que consiste de um bloco de controle centralizado e cinco portas bidirecionais denominadas NORTH, SOUTH, WEST, EAST, LOCAL ilustrado na Figura 8. A porta local é dedicada à conexão com núcleos IP local e as demais são utilizadas para conectar outros roteadores de acordo com a topologia definida. A lógica de controle (Figura 8) implementa o algoritmo de roteamento e método de arbitragem.

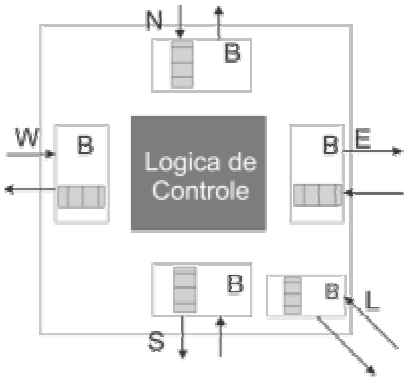

Figura 8. Arquitetura do roteador da HERMES. Adaptada de (MORAES et al. 2004)

A configuração do roteador da HERMES utilizada neste trabalho possui um número diferente de portas, dependendo de sua posição em consideração aos limites da rede. O roteador da HERMES foi configurado utilizando a topologia malha bidimensional, porém outras topologias podem ser utilizadas, como as apresentadas na Seção 2.2.1.

Controle de Fluxo

O controle de fluxo adotado pela HERMES pode ser do tipo *handshake* ou baseado em créditos. Este trabalho utilizou o controle de fluxo baseado em créditos que possui o seguinte mapeamento das portas de entrada e saída:

Portas de entrada:

- clock_rx: recebe o clock de um roteador vizinho;
- data_in: contém o dado (16bits, porém é parametrizável) a ser recebido;
- rx: indica que existe um dado a ser recebido;
- credit_i: recebe a quantidade de créditos no buffer vizinho;
- lane_rx: canal virtual usado para receber os dados.

Porta de saída:

- clock_tx: envia o clock para uma roteador vizinho;
- data_out: contém o dado(16bits, porém é parametrizável) a ser enviado;
- tx: indica que existe um dado a ser enviado;
- credit_o: envia a quantidade de créditos no buffer;
- lane_tx: canal virtual para enviar os dados.

Na Figura 9 pode ser visualizada a forma da ligação desses sinais na rede.

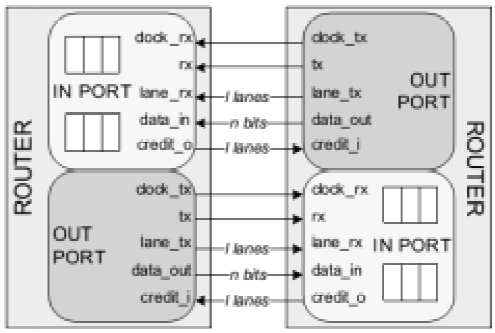

Figura 9. Interface entre roteadores com controle de fluxo baseado em créditos. (MORAES et al. 2004)

Roteamento

A lógica de roteamento implementa diferentes algoritmos. Dentre os apresentados na seção 2.2.1, a rede HERMES utilizada neste trabalho adota o roteamento XY.

Arbitragem

O roteador HERMES pode implementar duas políticas de arbitragem, uma dinâmica utilizando uma fila circular de prioridade (*Round-Robin*) e outra estática por prioridade. A fim de permitir a plataforma utilizar serviços de QoS, foi utilizado a arbitragem por prioridade.

Chaveamento

Dentre os métodos de chaveamento apresentados na seção 2.2.1, o método *wormhole* foi escolhido por oferecer como vantagens: (i) necessita de *buffers* menores para memorizar os dados; (ii) fornece baixa latência de comunicação; e (iii) pode multiplexar mais de um canal virtual em um canal físico. Outro ponto importante é a limitação apresentada pela NoCX4 em relação ao chaveamento, uma vez que utiliza o chaveamento VCT, que apresenta uma maior profundidade dos *buffers*.

3.3. INTEGRANDO A HERMES À STORM

Nesta seção, serão apresentados os aspectos referentes à integração da rede HERMES à STORM. A rede HERMES é descrita em VHDL enquanto que a plataforma STORM é descrita em SystemC.

Na seção 3.3.1 serão apresentadas as características da linguagem SystemC. Na seção 3.2 apresenta as características da rede em chip HERMES. As seções 3.3, 3.4, 3.5 e 3.6 apresentam a interface de rede e como foi realizada a conversão dos pacotes para tornar possível a comunicação entre a rede em chip e a plataforma, descrevendo aspectos importantes do envio e recebimento de pacotes. Na seção 3.7 são apresentado aspectos importantes da integração.

3.3.1. SYSTEMC TLM

O SystemC (OSCI, 2006) é uma biblioteca de modelagem do C++ utilizada para descrição de hardware. É também uma metodologia pela qual é possível descrever e validar as características de diferentes tipos de sistemas eletrônicos em diferentes níveis de abstração (KROLL, 2002).

SystemC contém um conjunto de ferramentas de modelagem e sincronização da comunicação de um projeto descrito no nível TLM (*Transaction Level Modeling*): *interfaces*, *canais* e *eventos*.

A modelagem TLM constitui um intermediário entre SAM (*System Architecture Model*) (BLACK; DONOVAN, 2004) e o RTL(*Register Transfer Level*). O nível SAM modela a funcionalidade do sistema, sem considerar arquitetura e implementação. O nível RTL modela com precisão a implementação e a arquitetura utilizando linguagens como VHDL e VERILOG. A TLM permite descrever o sistema separando o comportamento da comunicação do comportamento interno dos elementos. A principal característica do nível TLM é a abstração da comunicação em termos de interface que implementa o conjunto de métodos (MOUSSA; GRELLIER; NGUYEN, 2003).

Uma interface é definida como um objeto funcional que consiste em um conjunto de métodos. Um método é um procedimento, uma declaração de uma tarefa ou uma função. Já o canal é uma estrutura que contém uma ou mais interfaces de uma porta, que o permite ao módulo e processo acessar a interface.

Uma porta somente pode acessar os canais pelas interfaces e unicamente tem acesso ao canal através dos métodos que a interface define.

O SystemC TLM (KROLL, 2002) provê interfaces baseadas nos conceitos de bloqueante ou não-bloqueante e unidirecional ou bidirecional. Os dois tipos básicos de processos no SystemC, *threads* e métodos, diferem na chamada ao método *wait()*. Quando este método (*wait*) é invocado dentro de um método implica em um erro em tempo de execução. Por outro lado, o uso de *threads* implica em mudanças de contexto. Dessa forma, SystemC define como bloqueante uma função que pode conter chamadas ao método *wait()* e como não-bloqueante uma função que não pode conter a chamada ao método *wait()*.

As interfaces de SystemC são classes abstratas de C++. Classes abstratas não podem ser usadas diretamente, mas servem como modelo para criação de outras classes por meio do mecanismo de herança. O mecanismo de polimorfismo de C++ permite que uma classe seja referenciada por meio da classe da qual herda. Dessa maneira, SystemC define a interface básica *sc_interface* que deve ser herdada por qualquer outra classe que define uma interface. Este mecanismo é utilizado para definir as diversas interfaces utilizadas para modelagem em nível TLM em SystemC.

Neste trabalho, foi utilizada a interface bloqueante bidirecional de SystemC, definida pela classe *noc_task_if*, mostrado na Figura 10. A escolha por uma interface bloqueante permite que os módulos realizem chamadas ao método *wait()* durante a transação para emular a passagem do tempo. O uso da interface bidirecional permite o fluxo de dados em ambas as direções numa mesma transação. Assim é possível modificar sinais durante a execução dos métodos de envio e recebimento.

```
class noc_task_if : virtual public sc_interface
{
  public:
        virtual void initialize() = 0;
        virtual void inline send_message () = 0;
        virtual void inline receive() = 0;
        virtual void terminate() = 0;

};
```

Figura 10. Declaração da interface em SystemC

As funções descritas na Figura 10 são caracterizadas a seguir:

- *Initialize()*: método inicial da interface de rede, este é responsável por iniciar a execução da interface.
- *Send_message()*: este método tem como função realizar o envio dos pacotes a rede em chip, além de realizar o controle de fluxo e adicionar informações necessárias a rede alvo.
- *Reveive()*: este método realiza a captura do pacote da rede em chip para a plataforma, além de remover todas as informações de controle adicionada pelo método *send_message()*.
- *Terminate()* : este método irá finalizar a execução da interface de rede.

Nesta interface, os métodos *initialize(), send_message(), receive()* e *terminate()*, são chamados e utilizados pelo método principal da STORM.

3.4. ENVIO DE MENSAGENS

O *send_message* tem por objetivo enviar pacotes oriundos dos processadores para a rede em chip. A STORM já possui uma interface de envio, porém são interfaces exclusivas para cada roteador da NoCX4 e não seria viável sua modificação para adequar a metodologia de mixagem de linguagens. Assim, a interface de envio não necessita de um *buffer* de grande profundidade para envio necessitando apenas guardar o primeiro pacote para realizar as conversões necessárias para a rede HERMES. O processo de envio é realizado em 3 etapas.

1. O processador começa a escrever os dados do pacote no *buffer* de envio da plataforma STORM de tamanho parametrizável.
2. O *buffer* de envio envia um sinal que terminou de receber os dados, ativando o sinal de *ValidDataOut = '1'*.
3. Uma vez que o final da escrita é sinalizada, cada palavra é enviada à rede em chip. Por meio do sinal *tx* é informado a rede que há dados a enviar.

A máquina de estado finito que representa a função de envio de pacotes está descrita na Figura 11.

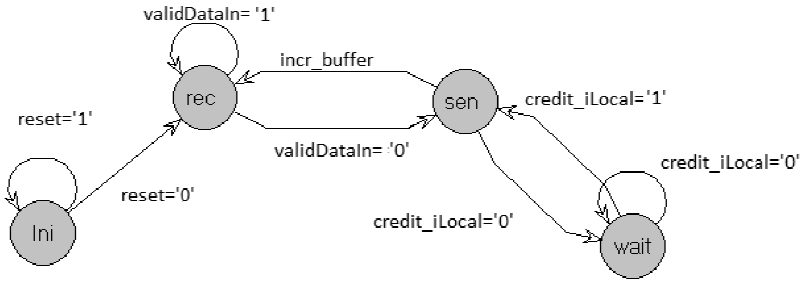

Figura 11. Máquina de estados do método *send_message*.

- Ini: Estado inicial. A máquina de estados inicia e permanece neste estado enquanto o reset não for ativado em '1', se não, avança para o próximo estado.
- Rec: Recebimento. Permanece neste estado enquanto o sinal de *ValidDataIn* estiver ativado. Avança para o estado de *send* quando o sinal *validDataIn* for desativado.
- Send: Transmitindo. Neste estado ativa-se o sinal de *tx* e incrementa-se o contador. A máquina de estados permanece neste estado enquanto que o sinal de *credit_iLocal* estiver ativo, indicando que pode ser enviado o pacote. Quando o sinal de *credit_iLocal* estiver desativado a máquina avança para o estado de *wait*. A informações de *credi_iLocal* é enviado pelo controle de fluxo da rede em chip.
- Wait: Permanece neste estado enquanto o sinal de *credit_iLocal* estiver desativado e avança para o próximo estado quando o sinal for ativado.

3.5. RECEPÇÃO DE MENSAGENS

O *receive* tem por objetivo receber os pacotes da rede em chip, e enviá-los para os processadores da STORM. O método é similar ao anterior, porém possui um *buffer* para armazenar os *flits* de controle da rede HERMES antes de enviá-los a plataforma.

A ideia foi deixar de forma transparente para a plataforma a utilização de interface. Os pacotes são reorganizados para o padrão da NoCX4 e enviados para a interface de recebimento da plataforma, e assim, enviados para a plataforma para retirar as informações pertinentes. Optou-se por utilizar esta estratégia por conta da grande modificação que seria necessário no projeto da STORM em relação a interface de envio e recebimento.

A Figura 12 apresenta a máquina de estado do método *receive*.

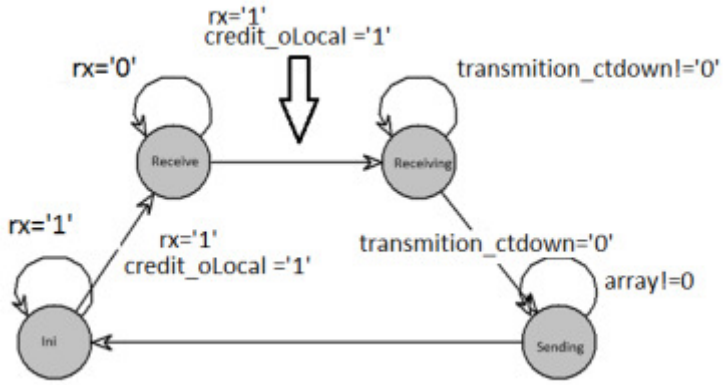

Figura 12. Máquina de estados do método *receive.*

Os estados desta máquina correspondem às seguintes funções:

- Ini: Estado inicial; Quando *rx_local* e *credit_oLocal* estiverem ativos, a máquina de estado avança para o estado *Receive*;
- Receive: Recebimento: Quando o estado atual estiver em S2 e os sinais de *rx_local* e *credit_oLocal* forem ativados, a máquina de estado avança para o próximo estado *Receiving*. Nesse estado é zerado o contador que controla a quantidade de ocupação do *buffer.*
- Receiving: O *buffer* realiza a recepção do pacote. O contador é incrementado neste estado, desde que rx seja igual a '1'. Quando todos os *flits* forem recebidos, avança-se para estado *Sending*;
- Sending: Este estado é responsável por ativar o sinal *ValidDataOut*, que indica a plataforma que há pacotes a receber.

3.6. INTERFACE

O Interface é responsável por integrar as funções *send_message* e *receive* e realizar a comunicação entre o processador SPARC e a rede em chip HERMES. Outra função importante da interface de rede é enviar um sinal de sincronização de *clock* para a rede HERMES, quando a interface é inicializada o sinal é enviado para a HERMES.

A Figura 13 apresenta o interface desenvolvido. Os sinais de interface com a plataforma compreendem: (i) *DataIn*: dados a serem enviados a NoC; (ii) *ValidDataIn*: verifica se ainda existe dados a transmitir; (iii) *AvailableDataIn*: verifica a quantidade

de pacotes a serem recebidos; (iv) *DataOut*: dados de saída para a plataforma STORM; (v) *ValidDataOut*: verifica se existe dados a transmitir para a STORM (vi) *AvailableDataOut*: verifica a quantidade de pacotes a serem enviados a STORM;

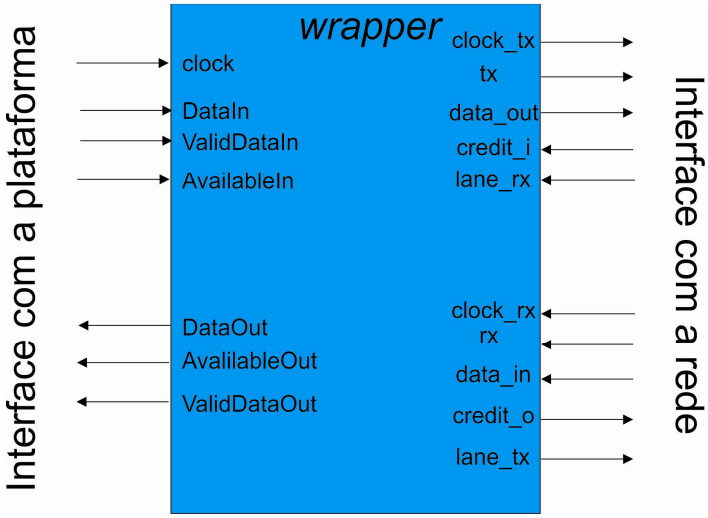

Figura 13. Interface

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*: 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*: dado recebido da NoC; ($viii$) *credit_o*: informa à NoC se pode receber dados. Existem ainda os sinais de *lane_rx*: indica qual canal os dados são transmitidos para a rede, e *lane_rx*: indica qual canal os dados são recebidos pela rede.

3.7. INTEGRAÇÃO

Após realizar o estudo para converção dos pacotes entre a HERMES e os processadores da plataforma STORM e definir o Interface responsável pelo interfaceamento, foi necessário a integração.

O primeiro passo da integração foi adaptar a classe principal da plataforma STORM que é responsável por iniciar a simulação e carregar as aplicações nos módulos

de memória. O primeiro problema encontrado nesse processo foi a sincronização de *clock* (relógio).

A entidade principal da plataforma utiliza uma função de geração de *clock* utilizando controle de processos e as diretivas de *wait()* do SystemC. Esse método gera o *clock* para a utilização da plataforma com a rede NoCX4, porém quando realizada a integração o *clock* não era transmitido para a interface e posteriormente para a rede em chip. Esse problema foi contornado com a criação de uma entidade geradora de clock externo e a criação de portas de entrada de sinais na plataforma.

3.7.1. MODIFICAÇÃO DA CLASSE MAIN

Além das alterações na geração do *clock* foi necessário realizar a mudança no aspecto geral da classe principal. Varias modificações devem ser aplicadas ao código fonte SystemC da plataforma.

Converter *sc_main()* para um módulo: Para a ferramenta de Active-HDL reconhecer o processo de conversão de sinais e executar o código SystemC, a função *sc_main()* deve ser substituída por um construtor, SC_CTOR(), colocado dentro do módulo no nivel superior do projeto, que neste caso a classe principal da plataforma. Além disso: Qualquer código de *testbench* dentro do sc_main() deve ser transferido para um processo, normalmente SC_THREAD.

Outro ponto importante da modificação é que todas as variáveis do SystemC em *sc_main()* incluindo sinais, portas e módulos devem ser definidos como membros de *sc_module*.

Para projetos SystemC, deve-se exportar todos os módulos de nível superior em seu projeto, no Acitve-HDL é necessário utilizar o macro SC_MODULE_EXPORT(<nome do módulo>). O exemplo pode ser visto na Tabela 3.

Tabela 3. Exemplo de conversão da classe sc_main().

Original código	**Modificado**

int sc_main(int argc, char* argv[])
{
sc_signal<bool> mysig;
mymod mod("mod");
mod.outp(mysig);
sc_start(100, SC_NS);
	SC_MODULE(mytop)
	{
	sc_signal<bool> mysig;
	mymod mod;
	SC_CTOR(mytop)
	: mysig("mysig"),

}	mod("mod")
	{
	mod.outp(mysig);
	}
	};
	SC_MODULE_EXPORT(mytop);

3.7.2. ADAPTAÇÃO DOS ARQUIVOS VHDL

Os arquivos da rede em chip HERMES foi gerado pelo ambiente *Atlas* que permite a geração da rede, contrução de cenários de tráfego (que caracterizam aplicações que executarão sobre a rede) e avaliação de desempenho.

O fluxo de projeto na Atlas é composto das seguintes etapas:

- *NoC Generation*: geração da rede. São escolhidos os seguintes parâmetros estruturais da rede: controle de fluxo; número de canais virtuais; tipo de escalonamento; dimensão; largura do canal de comunicação; tamanho do buffer; algoritmo de roteamento; e a opção da geração do testbench em SystemC. Em VHDL são gerados todos os módulos que compõem os roteadores e são estabelecidas as suas respectivas interconexões.
- *Trafic generation*: geração do tráfego que será transmitido na rede. O usuário pode configurar a rede inteira ou nodo a nodo, com os seguintes parâmetros: taxa de transmissão; padrão de distribuição de tráfego, ou um único destino; número de pacotes; tamanho de cada pacote, em flits; e distribuição de taxas de injeção. Esta etapa gera para cada núcleo transmissor um arquivo com a descrição do tráfego.
- *NoC simulation*: invoca um simulador VHDL externo (ModelSim), utilizando a descrição da rede, e os arquivos de teste e testbench. Os resultados da simulação são salvos em arquivos de saída. Além da gravação de dados das conexões entre os núcleos e os roteadores (para avaliação externa de desempenho), é possível gravar informações de cada canal que conecta os roteadores (para avaliação interna de desempenho).
- *Trafic Evaluate*: captura os dados dos arquivos da simulação, permitindo a análise de parâmetros de entrega das mensagens, como vazão e latência, tanto de forma externa, como interna. São gerados relatórios, curvas, gráficos 3D, mapas

textuais e tabelas que descrevem o comportamento do cenário estabelecido em NoC Generation e Traf ic Generation.

No projeto da interface foi utilizado apenas o primeiro passo para a geração da rede em chip necessária para integração à plataforma STORM. Assim todos os arquivos gerados foram integrados ao projeto da plataforma, porém o primeiro problema encontrado foi a entidade Top da rede HERMES que apresentava os sinais por meio de matrizes.

As matrizes são responsáveis por enviar os sinais para todos os roteadores da rede, mas quando utilizada a metodologia de mixagem é necessário fazer a conversão desses matrizes em vetores com sinais compatíveis. Na Tabela 3, os sinais de compatibilidade são descritos e na última linha o tipo compativel para SystemC para matrizes em VHDL são *arrays*, porém na especificação da metodologia não há um equivalente direto para esse tipo de dado (RUDRA; GAURAV; SACHIN, 2006).

Tabela 4. VHDL-SC compatibilidade de tipo de dados

Tipo VHDL	Tipo SystemC
std_logic, bit, boolean	sc_logic, sc_bit
std_logic_vector, bit_vector	sc_lv, sc_bv
real	double/float
integer	primitive C integer types
record	struct
enum	enum
multi-d array	array of signals*

A interface deve promover essa adaptação de tipos. Além da modificação da classe principal da HERMES, a interface deve manter esse compatibilidade de sinais. Assim para contornar esse problema de sinais foram convetidos de matrizes para vetores, tornando possível a leitura e escrita de sinais de SystemC na entidade top da rede HERMES descrita em VHDL.

4. RESULTADOS

Neste capitulo são apresentados os resultados da integração da rede em chip HERMES à plataforma STORM utilizar uma rede diferente da NoCX4. Além de testar e validar à interface de rede foi realizada um estimativa de desempenho com base na latência e uma a aplicação de multiplicação de matrizes.

4.1. ESTIMATIVA DE DESEMPENHO

A estimativa de desempenho foi realizada com objetivo de analizar o custo associoado na integração da rede em chip HRMES e futuras integrações, pois permite verificar qual o custo de latência para o processo de conversão dos pacotes entre a plataforma e a rede em chip.

A interface de rede consome 2 ciclos entre o incio e fim da operação de recebimento dos pacotes e o envio para a rede. E consome 2 ciclos entre o inicio e fim da operação de recebimento dos pacotes por parte da STORM.

De acordo com Mello et al.(2005) a latência da rede HERMES para o envio de um pacote com *P flits* entre dois nodos separados por R roteadores, é dada por (1), considerando a livre contenção e com controle de fluxo baseado em créditos utilizando canais virtuais.

$$Latência_{rede} = \sum_{i=1}^{n} R_i + P \quad (1)$$

Dessa forma, as latências mínimas para execução de operações recebimento e envio, entre dois nodos separados por R_i roteadores e a latência (arbitragem e routeamento) é pelo menos 6 ciclos de *clock* na implementação usada. Considerando o tamanho dos correpondentes pacotes (16 *flits*), é dadas por (3).

$$Lantência_{interface} = \sum_{i=1}^{n} R_i + 16 + 4 \quad (2)$$

$$Lantência_{interface} = \sum_{i=1}^{n} R_i + 20 \quad (3)$$

Ao considerar dois nodos separados por apena dois roteadores (distância mínima), a latência mínima de uma operação de envio e recebimento é igual a 32 ciclos.

Portando, a contribuição máxima da interface de rede em uma operação de envio e recebimento é igual a 12,5% (4/32) da latência total, decaindo com o aumento da distância entre os nodos, conforme a Figura 14.

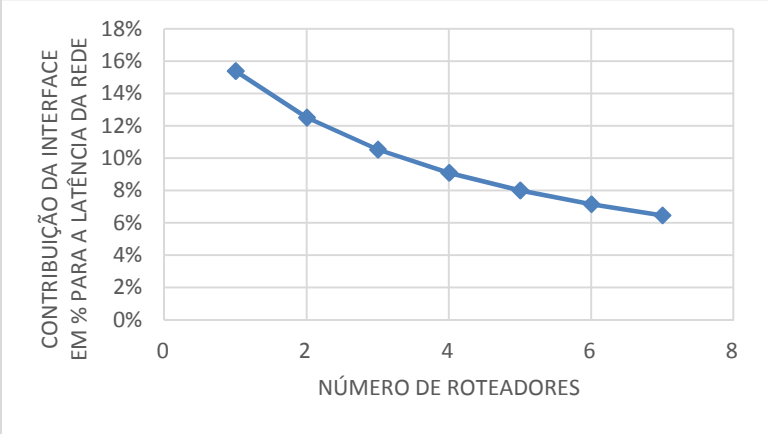

Figura 14. Contribuição da interface de rede.

4.2. SIMULAÇÃO: MULTIPLICAÇÃO DE MATRIZES

Operações sobre matrizes, como a multiplicação de matrizes, são bastante comuns sendo muito utilizadas em áreas como: teoria dos grafos, algoritmos numéricos, processamento de sinais e controle digital (ZIAD; MUSBAH, IBRAHIEM, 2008). Em geral, a multiplicação de matrizes requer esforço computacional devido sua complexidade.

A aplicação consistiu da multiplicação de duas matrizes M_A e M_B de tamanho 8x8, cujo resultado é guardado na matriz de resultado M_C. O algoritmo utilizado, de complexidade $O(n^3)$, percorre as linhas de M_A e as colunas de M_B, multiplicando e acumulando o resultado, e por fim salvando-o na matriz M_C.

Foram utilizadas duas instâncias da plataforma STORM, versão memória compartilhada, para realização dos experimentos. Uma instância utilizando a rede NoCX4 (Figura 15) e outra utilizando a rede HERMES (Figura 16). No modelo de memoria compartilhada, todos os processadores acessam o módulo de memória. Portanto, visando minimizar o tempo médio de acesso à memória, a mesma foi posicionada em um nodo central da malha.

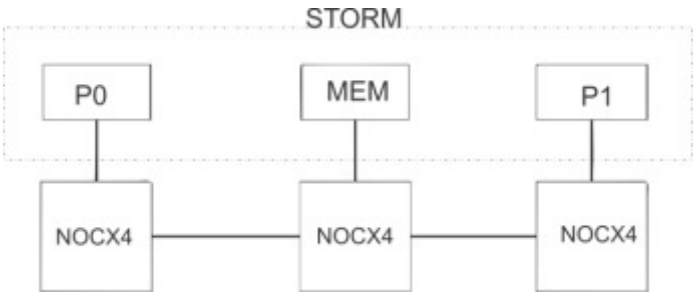

Figura 15. Instância com rede NoCX4.

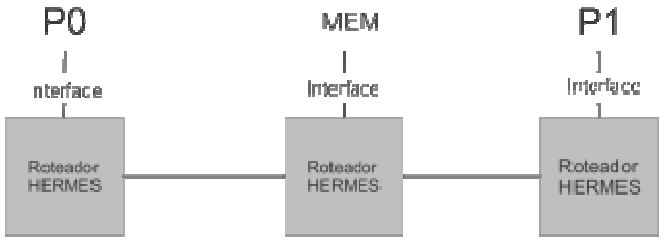

Figura 16. Instância com rede HERMES.

Os resultados apresentados na Tabela 5 correspondem à aplicação de multiplicação de matrizes na plataforma STORM configurada com o modelo de memória compartilhada e redes em chip HERMES e NOCX4. Os resultados apresentados na simulação usando a rede em chip NoCX4 estão de acordo com a dissertação de REGO (2006).

Tabela 5. Comparação das implementações com mesmo número de processadores.

	STORM – NOCX4	STORM-HERMES	Processadores
Ciclos Simulados	58268	59168	2
CPI Médio	0,33	0,33	2
Latência Media	10.57	17,43	2

Os resultados apresentados na Tabela 5 demostram que a interface de rede configurada para o uso na rede em chip HERMES representa um aumento em torno de 2% em ciclos de simulação se comparado com a rede NoCX4. O aumento nos ciclos é justificado pelo número de *flits* de controle que a rede HERMES necessita para a entrega dos pacotes, de acordo com o trabalho de Mello (2006) são necessário 2 *flits* de controle para a entrega dos pacotes e a cada ciclo a interface de rede libera um *flit*.

Com a leve variação dos ciclos simulados, o CPI médio não teve alteração em duas casas decimais visto que é a mesma quantidade de instrução para um número ligeiramente menor de ciclos de simulação.

Dessa forma os resultados de latência da rede terá um aumento 10.57 para 17.43, visto que a latência dos pacotes é calculada com base nos ciclos de *clock* para cada pacote. A latência média dos pacotes na rede é exibida na terceira linha da Tabela 5. Esses valores foram obtidos calculando a média da latência dos pacotes que chegam a cada roteador.

5. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esta pesquisa realizou uma investigação dos requisitos necessários para a integração de diferentes redes em chip a plataformas multiprocessadas. A partir do levantamento bibliográfico e dos estudos realizados, identificou-se o uso de interfaces de rede. Esta pesquisa integrou por meio de um interface uma rede em chip a uma plataforma, que passou a suportar novos serviços de comunicação.

A integração consistiu no uso da rede em chip HERMES pela plataforma STORM, a qual utilizava a rede NoCX4 como estrutura de interconexão. Apesar da utilização de uma rede em chip específica no estudo de caso, o interface pode ser configurado para utilizar outras redes em chip.

Por meio da pesquisa realizada, percebeu-se ainda o aumento no número de serviços suportados pelas redes em chip, que motivou a utilização de uma estrutura de camada, separando os módulos para facilitar a adição de serviços adicionais.

Uma das contribuições deste trabalho foi a modelagem em nível TLM em SystemC, que permitiu abstrair detalhes da comunicação entre os módulos, permitindo reduzir o tempo de desenvolvimento dos modelos. São utilizadas chamadas as funções entre componentes para representar transações dentro do sistema ao invés da modelagem dos sinais e temporização.

Apesar dos resultados não apresentarem ganhos de desempenho na plataforma utilizando a rede HERMES como infraestrutura de comunicação, a contribuição está em permitir que a plataforma suporte outras demandas de comunicação que a NoCX4 não oferece, como por exemplo, requisitos de Qualidade de Serviços e Tempo Real.

Como trabalhos futuros, pretende-se adicionar controle de pacotes para o mapeamento de tarefas, permitindo que a plataforma STORM possa reduzir, por exemplo, o número de saltos dos pacotes e a ocupação dos canais. No caso de aplicações de comunicação intensa, tais como de fluxo de dados, a manutenção da ocupação da infraestrutura de comunicação é fundamental. Assim a ocorrência de congestionamento pode prejudicar transmissões de áudio e vídeo. O mapeamento de tarefas pode ser usado para evitar ao máximo o congestionamento dos canais de

comunicação. Outro ponto importante é o balanceamento da carga dos processadores no sistema.

Considerando mapeamento de tarefas, a interface foi projetada de modo que todos os pacotes destinados aos núcleos sejam transmitidos exclusivamente por uma interface global.

Um Outro trabalho futuro será a integrar o controle de integridade em nível de enlace, permitindo que a plataforma possa verificar a integridade dos *flits* a serem transferidos pela rede (verificação de paridade) ou do pacote ser entregue a interface de rede destino (verificação de *checksum*). E por último a integração de outras redes em chip diferentemente da HERMES que contenham outros serviços de comunicação.

REFERÊNCIAS

ACKLAND, B. et al. A single-chip, 1.6-billion, 16-b mac/s multiprocessor DSP. *Solid-State Circuits*, IEEE Journal of, v. 35, n. 3, p. 412 –424, mar 2000. ISSN 0018-9200.

BENINI, L. De MICHELI, G. **Networks on chips:** a new SoC paradigm. Computer, 35(1), Jan. 2002, pp. 70-78.

BERGAMICHI, R. A.; LEE W. R. Designing system-on-chip using cores. In DAC '00: **Proceedings of the 37th conference on Design automation**, pp 420-425, New York, NY, USA. ACM Press.

BERTOZZI, D. The data link layer in NoC design. In: DE MICHELI, Giovani; BENINI, Luca. **Networks-on-Chips**: technology and tools. 2006. Amsterdam: Boston: Elsevier: Morgan Kaufmann Publishers, 2006.

Black, D. C; Donovan, J. **SystemC from the Ground-up**, Kluwer Academic Press, 2004

BOLOTIN, E.; CIDON, I.; GINOSAR, R.; KOLODNY , A.. QNoC: QoS architecture and design process for network on chip. **Journal of Systems Architecture: the EUROMICRO Journal** (Special issue: Networks on chip archive). Vol. 50 Issue 2-3, Feb. 2004.

CORPORATION, T. TILE64™ Processor. August 2007. Santa Clara, CA, EUA. Product Brief Description. Disponível em: <http://www.tilera.com/sites/default/files/productbriefs/PB010 TILE64 Processor A v4.pdf>

DALLY, W. J.; TOWLES, B. Route packets, not wires: on-chip interconnection networks. In: CONFERENCE ON DESIGN AUTOMATION, 38., 2001, Las Vegas. **Proceedings**... New York: ACM Press, 2001. p. 684-689.

DALLY, W. J.; TOWLES, B. **Principles and practices of interconnection networks**. Amsterdan: Morgan Kaufmann, 2004.

DALLY, W. J. Virtual-channel flow control. IEEE **Transactions on Parallel and Distributed Systems**, Piscataway, v. 3, n. 2, p. 194-205, 1992.

DE MICHELI, G., BENINI, L. **Networks on Chip**: Technology and Tools, San Francisco: Editora Morgan Kaufmann, p. 203 – 284, 2006.

DUATO, J. et al., **Interconnection Networks**. Los Alamitos, California: IEEE Computer Society Press, 1997, p. 515.

EBRAHIMI, Masoumeh; DANESHTALAB, Masoud; SREEJESH, Plosila; LILJEBERG, Pasi; TENHUNEN, Hannu. Efficient network interface architecture for Network-on-Chips. In: NORCHIP CONFERENCE, 27., 2009, Trondheim. **Proceedings**... [S.l.: s.n.], 2009. p.1-4.

GLASS, C. J.; NI, L. M. The turn model for adaptive routing. ACM:1992; Vol. 20, pp 278-287.

Guerrier, P., Greiner, A. (2000), "A Generic Architecture for On-Chip Packet-Switched Interconnections", In: Design, Automation and Test on Europe - DATE, 2000, Paris. Proceedings... Los Alamitos: IEEE Computer Society Press. p. 250-256

JERGER , N. E.; PEH, L. S. "**On-Chip Networks**", Synthesis Lecture, Morgan-Claypool Publishers, Aug. 2009

JERRAYA, A.; TENHUNEN, H.; WOLF, W. **Multiprocessor System-on-chips**. Computer, 38 (Issue 7): 36-40, 2005.

KROLL, A. "Transaction Level Modeling (TLM) with *SystemC* and CoCentric System Studio". SNUG Boston. 2002;

KUMAR, S. et al.: A Network on Chip Architecture and Design Methodology. In: Int. Symposium on Very Large Integration Scale, Pittsburg, 2002. Proceedings... Los Alamitos: IEEE Computer Society (2002) 105-112

LI, M., et al. DyXY: a proximity congestion-aware deadlock-free dynamic routing method for network on chip. In: **Proceedings of the 43rd annual conference on Design automation.** ACM: San Francisco, CA, USA. 2006.

MATOS, Débora. **Interfaces parametrizáveis para aplicações interconectadas por uma Rede-em-Chip**. 2010. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós Graduação

Mello, A.; Tedesco, L.; Calazans, N.; Moraes, F., "Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC," *Integrated Circuits and Systems Design, 18th Symposium on* , vol., no., pp.178,183, 4-7 Sept. 2005

M. RUDRA, K. V. GAURAV, K. SACHIN "SystemVerilog-VHDL Mixed Design Reuse Methodology", DVCon 2006

MELO, D. R. **INTERFACE DE COMUNICAÇÃO EXTENSÍVEL PARA A REDE-EM-CHIP SOCIN**, 2012. Dissertação (Mestrado em Computação aplicada) – Curso de Mestrado Academico em Computação aplicada, Universidade do Vale do ITAJAÍ, SANTA CATARINA, 2012.

MOORE, G. E., "**Cramming More Components onto Integrated Circuits**", Electronics, pp. 114–117, April 19, 1965.

MORAES, Fernando Gehm ; CALAZANS, Ney ; MELLO, Aline Vieira de ; MÖLLER, Leandro Heleno; OST, Luciano. HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. **Integration** (**Amsterdam**), Amsterdam, v. 38, n. 1, p. 69-93, 2004.

MOUSSA, I.; GRELLIER,T.; NGUYEN, G. Exploring SW Performance Using SoC Transaction-Level Modeling. In **Proceedings of the conference on Design,**

Automation and Test in Europe: Designers' Forum - Volume 2 (DATE '03), Vol. 2. IEEE Computer Society, Washington, DC, USA, 20120.

OSCI INITIATIVE. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society, p.* 1666-2005, 2006

PALMA, J. C. S**. Reduzindo o Consumo de Potência em Networks-on-Chip através de Esquemas de Codificação de Dados**. Tese (Tese doutorado) — PUCRS, 2007

PASRICHA, S. e DUTT, N.**On chip Communication Architectures: System on chip Interconnect**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

REGO, R. S. de L. S. **Projeto e Implementação de uma Plataforma MP-SoC usando SystemC**. Dissertação (Dissertação de Mestrado) — UFRN, 2006.

TOMLINSON, R. S. **Selecting sequence numbers**. SIGOPS Oper. Syst. Rev., 1975, (3):11-23.

VANGAL, S. et al. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In: Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International. [S.l.: S.N.], 2007. p. 98 –589. ISSN 0193-6530.

YAN, S.; MIN, G.; AWAN, I. "Performance Analysis of Credit-Based Flow Control in InfiniBand Interconnection Networks. **Journal of Interconnection Networks**, v. 7, n. 4, p.535-548, 2008.

ZIAD, A; MUSBAH, A; IBRAHIEM, E.E. "Performance Analysis and Evaluation of Parallel Matrix Multiplication Algorithms", World Applied Sciences Journal, pp. 211 – 214, 2008.

ANEXOS I

#----------

#

Multiplicação de Matrizes

#----------

#

Descrição: Multiplica duas matrizes paralelamente fazendo uso da biblioteca “mutex.h”.

```
#include "mutex.h"
#define SIZE 8
#define PROCESSORS 2
#define STEP (SIZE/PROCESSORS)
#define RESTO (SIZE%PROCESSORS)
int m1[16], m2[16];
#define mutex1 m1[7]
#define mutex2 m2[7]
int M_A[8][8] = {1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8, };

int M_B[8][8] = {1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8,
                 1,2,3,4,5,6,7,8, };
int M_C[SIZE][SIZE];
int main() {
    int i, j, k, soma, fim;
    unsigned noc_addr = _get_noc_addr();
    down(&mutex1);
    unsigned int id=get_id();
    up(&mutex1);

    if (id < RESTO) {
        i = (id * STEP) + id;
        fim = ((id + 1) * STEP) + id + 1;
    } else {
        i = (id * STEP) + RESTO;
        fim = ((id + 1) * STEP) + RESTO;
    }

    for (; i<fim; i++)
      for (j=0; j<SIZE; j++) {
          soma = 0;
          for (k = 0; k<SIZE; k++)
            soma += M_A[i][k]*M_B[j][k];
          down(&mutex2);
          M_C[i][j] = soma;
```

```
        up(&mutex2);
    }
return noc_addr; }
```

ANEXOS II

```
#-------------------------------------------------------------------------------------------------------
#
```

INTERFACE DE REDE

```
#-------------------------------------------------------------------------------------------------------
#
#ifndef __INTERFACE_IF_H__
#define __INTERFACE_IF_H__

#include <systemc.h>
#include "noc_args.h"
#include "include/specs.h" // Palavras reservadas da STORM
#define NR 3 // numero de roteadores.
#define Lane 2


class noc_task_if : virtual public sc_interface
{
  public:
              virtual void initialize() = 0;
              virtual bool inline InValid(int) = 0;
              virtual void inline outTx(int, int) = 0;
              virtual unsigned long int inline InData(int) = 0;
              virtual void inline outData(int, unsigned long int) = 0;
              virtual void inline send_message () = 0;
              virtual int inline inCredit ( int , int) = 0;
              virtual void inline laneTx (int,unsigned long int) = 0;
              virtual void inline receive() = 0;
              virtual void terminate() = 0;

};


class noc_port_if : public sc_module
{
       public:

              sc_in<sc_logic > clk;
              sc_out<sc_logic > reset;

              //storm
              sc_in<sc_int<FIFO_WIDTH> > DataIn0;
              sc_in<int> AvailableIn0;
              sc_in<bool> ValidDataIn0;
              sc_in<bool> WritingStartIn0;
              sc_out<sc_int<FIFO_WIDTH> > DataOut0;
              sc_out<int>  AvailableOut0;
              sc_out<bool> ValidDataOut0;
              sc_out<bool> WritingStartOut0;

              sc_in<sc_int<FIFO_WIDTH> > DataIn1;
              sc_in<int> AvailableIn1;
              sc_in<bool> ValidDataIn1;
              sc_in<bool> WritingStartIn1;
              sc_out<sc_int<FIFO_WIDTH> > DataOut1;
              sc_out<int>  AvailableOut1;
              sc_out<bool> ValidDataOut1;
```

```
        sc_out<bool> WritingStartOut1;

        sc_in<sc_int<FIFO_WIDTH> > DataIn2;
        sc_in<int> AvailableIn2;
        sc_in<bool> ValidDataIn2;
        sc_in<bool> WritingStartIn2;
        sc_out<sc_int<FIFO_WIDTH> > DataOut2;
        sc_out<int>  AvailableOut2;
        sc_out<bool> ValidDataOut2;
        sc_out<bool> WritingStartOut2;


        //hermes
        sc_out<sc_lv<NR> > rxLocal;
        sc_out<sc_lv<Lane> > lane_rxLocal0;
        sc_out<sc_lv<Lane> > lane_rxLocal1;
        sc_out<sc_lv<Lane> > lane_rxLocal2;
        sc_out<sc_lv<FIFO_WIDTH> > data_inLocal0;
        sc_out<sc_lv<FIFO_WIDTH> > data_inLocal1;
        sc_out<sc_lv<FIFO_WIDTH> > data_inLocal2;
        sc_in<sc_lv<Lane> > credit_oLocal0;
        sc_in<sc_lv<Lane> > credit_oLocal1;
        sc_in<sc_lv<Lane> > credit_oLocal2;
        sc_in<sc_lv<NR> > clock_txLocal;
        sc_in<sc_lv<NR> > txLocal;
        sc_in<sc_lv<Lane> > lane_txLocal0;
        sc_in<sc_lv<Lane> > lane_txLocal1;
        sc_in<sc_lv<Lane> > lane_txLocal2;
        sc_in<sc_lv<FIFO_WIDTH> > data_outLocal0;
        sc_in<sc_lv<FIFO_WIDTH> > data_outLocal1;
        sc_in<sc_lv<FIFO_WIDTH> > data_outLocal2;
        sc_out<sc_lv<Lane> > credit_iLocal0;
        sc_out<sc_lv<Lane> > credit_iLocal1;
        sc_out<sc_lv<Lane> > credit_iLocal2;

noc_port_if(sc_module_name nm) : sc_module(nm),
              clk("clk"),
              reset("reset"),
              DataIn0("DataIn0"),
              AvailableIn0("AvailableIn0"),
              ValidDataIn0("ValidDataIn0"),
              WritingStartIn0("WrintingStartIn0"),
              DataOut0("DataOut0"),
              AvailableOut0("AvailableOut0"),
              ValidDataOut0("ValidDataOut0"),
              WritingStartOut0("WritingStartOut0"),
              rxLocal("rxLocal"),
              lane_rxLocal0("lane_rxLocal0"),
              lane_rxLocal1("lane_rxLocal1"),
              lane_rxLocal2("lane_rxLocal2"),
              data_inLocal0("data_inLocal0"),
              data_inLocal1("data_inLocal1"),
              data_inLocal2("data_inLocal2"),
              credit_oLocal0("credit_oLocal0"),
              credit_oLocal1("credit_oLocal1"),
              credit_oLocal2("credit_oLocal2"),
              clock_txLocal("clock_txLocal"),
              txLocal("txLocal"),
              lane_txLocal0("lane_txLocal0"),
```

```
                    lane_txLocal1("lane_txLocal1"),
                    lane_txLocal2("lane_txLocal2"),
                    data_outLocal0("data_outLocal0"),
                    data_outLocal1("data_outLocal1"),
                    data_outLocal2("data_outLocal2"),
                    credit_iLocal0("credit_iLocal0"),
                    credit_iLocal1("credit_iLocal1"),
                    credit_iLocal2("credit_iLocal2")
        {
        }

};


#endif // __INTERFACE_IF_H__


#ifndef __INTERFACE_H__
#define __INTERFACE_H__

#include <systemc.h>
#include "interface_if.h"



class noc : public noc_port_if,
            public noc_task_if
{
public:
            noc(sc_module_name nm) : noc_port_if(nm) {};

public:
            virtual void initialize() = 0;
            virtual bool inline InValid(int) = 0;
            virtual void inline outTx(int, int) = 0;
            virtual unsigned long int inline InData(int) = 0;
            virtual void inline outData(int, unsigned long int) = 0;
            virtual void inline send_message () = 0;
            virtual int inline inCredit ( int , int) = 0;
            virtual void inline laneTx (int,unsigned long int) = 0;
            virtual void inline receive() = 0;
            virtual void terminate() = 0;

};

#endif // __INTERFACE_H__


#include "interface.h"
#include <systemc.h>


unsigned long int CurrentTime;
bool active[NR];
sc_int<FIFO_WIDTH> array[NR];
sc_int<FIFO_WIDTH> arrayR[NR]
sc_int<FIFO_WIDTH>  destino;
```

```
sc_int<FIFO_WIDTH>  tamanho;
sc_int<FIFO_WIDTH>  origem;
sc_int<ADDR> Dest;
sc_int<PCK_SIZE_FIELD> tam;
sc_int<ADDR> Orig;
int available[NR];
int availableR[NR]
int Index,i,j,k,cont=0,controle;

int transmition_ctdown[NR];
int transmition_ctdownR[NR]
int total_sent_data[NR];
int total_sent_pcks[NR];
int total_received_data[NR];


bool inline noc::InValid(int Indice){ // da STORM
	if(Indice == 0){
		return ValidDataIn0.read();
	}
	if(Indice == 1){
		return ValidDataIn1.read();
	}
	if(Indice == 2){
		return ValidDataIn2.read();
	}
}
void inline noc::OutValid(int Indice, int Booleano){ para STORM

	if(Indice == 0){
		ValidDataOut0.write(Booleano);
	}
	if(Indice == 1){
		ValidDataOut1.write(Booleano);
	}
	if(Indice == 2){
		ValidDataOut2.write(Booleano);
	}
}

 int inline noc::inTx( int Indice){ // da HERMES
		if(Indice == 0) return txLocal.read().get_bit(0);
		if(Indice == 1) return txLocal.read().get_bit(1);
		if(Indice == 2) return txLocal.read().get_bit(2);


 }
void inline noc::outTx(int Indice, int Booleano){ // para HERMES
		if(Indice == 0) rxLocal = (Booleano != 0)? "001":"000";
		else if(Indice == 1) rxLocal = (Booleano != 0)? "010":"000";
		else if(Indice == 2) rxLocal =(Booleano != 0)? "100":"000";
}


unsigned long int inline noc::inData(int Indice){ // da STORM
		if(Indice == 0) return DataIn0.read();
		if(Indice == 1) return DataIn1.read();
		if(Indice == 2) return DataIn2.read();
}
```

```
void noc::OutData(Int Indice, unsigned long int Valor){// para STORM
            if(Indice == 0) return DataOut0.write(Valor);
            if(Indice == 1) return DataOut1.write(Valor);
            if(Indice == 2) return DataOut2.write(Valor);

}


void noc::OutDataNoC(int Indice, unsigned long int valor){ // para HERMES
            if(Indice == 0) data_inLocal0.write(sc_lv<32>(valor));
            if(Indice == 1) data_inLocal1.write(sc_lv<32>(valor));
            if(Indice == 2) data_inLocal2.write(sc_lv<32>(valor));
}
void long int inline noc::InDataNoC(int Indice){ //da HERMES

            if(Indice == 0) return data_outLocal0.read();
            if(Indice == 1) return data_outLocal1.read();
            if(Indice == 2) return data_outLocal2.read();
 }




void inline noc::laneTx(int Indice, unsigned long int Valor){
            if(Indice == 0)      lane_rxLocal0.write(sc_lv<2>(valor));
            else if(Indice == 1) lane_rxLocal1.write(sc_lv<2>(Valor));
            else if(Indice == 2) lane_rxLocal2.write(sc_lv<2>(Valor));

      }


int inline noc::inCredit(int Indice, int Lane){ //da HERMES

            if(Indice == 0){
                  if(Lane == 0) return (credit_oLocal0.read().get_bit(0) ==
SC_LOGIC_1)? 1 : 0;
                  if(Lane == 1) return (credit_oLocal0.read().get_bit(1) ==
SC_LOGIC_1)? 1 : 0;
            }
            if(Indice == 1){
                  if(Lane == 0) return (credit_oLocal1.read().get_bit(0) ==
SC_LOGIC_1)? 1 : 0;
                  if(Lane == 1) return (credit_oLocal1.read().get_bit(1) ==
SC_LOGIC_1)? 1 : 0;
            }
            if(Indice == 2){
                  if(Lane == 0) return (credit_oLocal2.read().get_bit(0) ==
SC_LOGIC_1)? 1 : 0;
                  if(Lane == 1) return (credit_oLocal2.read().get_bit(1) ==
SC_LOGIC_1)? 1 : 0;
            }
}

void inline noc::outCredit(int Indice, int Valor){ //para STORM
            if(Indice == 0){
                  AvailableOut0.write(Valor);
            }
            if(Indice == 1){
```

```
                AvailableOut1.write(Valor);
            }
            if(Indice == 2){
                AvailableOut2.write(Valor);
            }

}

int inline noc::CreditOut(int Indice){ // da STORM
            if(Indice == 0){
                return AvailableIn0.read();
            }
            if(Indice == 1){
                return AvailableIn1.read();
            }
            if(Indice == 2){
                return AvailableIn2.read();
            }


 }


void noc::initialize()
{
      wait(clk.posedge_event());
      reset.write(sc_logic_1);
      CurrentTime=0;
      wait(clk.posedge_event());
      reset.write(sc_logic_0);
    ++CurrentTime;

}



void inline noc::send_message ()
{

 for(Index=0;Index<NR;Index++){ // ativando transmissões

       if(InValid(Index)==1){
            active[Index] = true;
            //n_active++;
      }else{
            active[Index] = false;
      }
 }    //fim for

 for(Index=0;Index<NR;Index++){

 if(inCredit(Index,0)==1){
      if(active[Index]){
             if(available[Index] <= 0) {
           cout << "**************************" << endl;
             cout << "****** BUFFER CHEIO ******" << endl;
             cout << "**************************" << endl;
             }
       }
```

```
 //PAYLOAD
 if (transmition_ctdown[Index]) { //Packet transmition is in course
        array[Index] = InData(Index);
        OutDataNoC(Index,array[Index]);
        outTx(Index,1); // envia sinal NOC
        laneTX(Index,1) // dois canais: 0 ou 1
        available[Index]++;
        transmition_ctdown[Index]--;
        total_sent_data[Index]++;

       } else{
       // HEAD
        array[Index] =  InData(Index);
        transmition_ctdown[Index] =
PacketSize[(int)array[Index].range(PCK_SIZE_FIELD-1, 0)];
             if (AvailableIn.read() >= transmition_ctdown[Index]) {
             //  cout << "CICLO = " << simulation_cycle
             //       << " TIPO = " << array[Index].range(PCK_FLAGS_FIELD +
PCK_SIZE_FIELD - 1, PCK_SIZE_FIELD)
             //       << " TAMANHO = " <<
PacketSize[(int)array[Index].range(PCK_SIZE_FIELD - 1, 0)]
             //       << " ORIGEM = " << array[Index].range(PCK_WIDTH - ADDR
- 1, PCK_WIDTH - ADDR * 2)
             //       << " DESTINO = " << array[Index].range(PCK_WIDTH - 1,
PCK_WIDTH - ADDR) << endl;


             destino.range(ADDR-1,0) = array[Index].range(PCK_WIDTH - 1,
PCK_WIDTH - ADDR);
             tamanho.range(PCK_SIZE_FIELD-1,0) =
PacketSize[(int)array[queue_head].range(PCK_SIZE_FIELD - 1, 0)];
             origem.range(ADDR-1,0) = array[Index].range(PCK_WIDTH - ADDR -
1, PCK_WIDTH - ADDR * 2);


          OutDataNoC(Index,destino);
          outTx(Index,1);
          laneTX(Index,1)

          wait(clk.posedge_event());
          OutDataNoC(Index,tamanho);
          outTx(Index,1); // envia sinal NOC
          laneTX(Index,1) // dois canais: 0 ou 1

          wait(clk.posedge_event());
          OutDataNoC(Index,origem);
          outTx(Index,1);
          laneTX(Index,1)

          available[Index]++;
        transmition_ctdown[Index]--;
        total_sent_pcks[Index]++;
        total_sent_data[Index]++;

    } else {

    transmition_ctdown[Index] = 0;
    outTx(Index,0);
```

```
    }

    } else { //No packets to transmit = empty queue
        transmition_ctdown[Index] = 0;
        outTx(Index,0); // envia sinal NOC
        }


        outCredit(Index,available[Index]);

    }
    }//fim for
}  // Fim Envio


void noc::receive(){

    for(Index=0;Index<NR;Index++){

         if (transmition_ctdownR[Index]) { //recebendo
          if (CreditOut(Index)) {
               total_received_data[Index]++;
               arrayR[Index] = InDataNoC(Index);
               OutData(Index,arrayR[Index]);
               OutValid(Index,1);
               availableR[Index]--;
               transmition_ctdownR[Index]--;
               }
     } else {  //HEAD

           if(inTx(Index)==1){
             if (CreditOut(Index)) {
                   arrayR[Index] = InDataNoC(Index);

                     Dest.range(ADDR-1,0) = arrayR[Index].range(ADDR-1,0);

                     wait(clk.posedge_event()); // 1 ciclo
                     arrayR[Index] = InDataNoC(Index);
                     transmition_ctdownR[Index] = arrayR[Index];

                     wait(clk.posedge_event()); // 2 ciclo
                     arrayR[Index] = InDataNoC(Index);
                     Orig.range(ADDR-1,0) = arrayR[Index].range(ADDR-1,0);

            OutData(Index,arrayR[Index]);
              OutValid(Index,1);
            availableR[Index]--;
              transmition_ctdownR[Index]--;
        }
     }
              OutData(Index,0);
              OutValid(Index,0);
              transmition_ctdownR[Index] = 0;

    }

   }//fim for
```

```
} //fim receive

void noc::terminate()
{
	for (int i=0; i<2; i++)
	{
		wait(clk.posedge_event());
	}
	sc_stop();
}
```