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

Modelagem, Descrição e Validação de Redes *Intrachip* no Nível de Transação

por

Edson Ifarraguirre Moreno

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de mestre em Ciência da Computação.

Prof. Dr. Ney Laert Vilar Calazans
Orientador

PORTO ALEGRE

Março, 2004


PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
BIBLIOTECA CENTRAL IRMÃO JOSÉ OTÃO

BIBLIOTECA CENTRAL

Dados Internacionais de Catalogação na Publicação (CIP)

M843m Moreno, Edson Ifarraguirre
Modelagem, descrição e validação de redes intrachip no nível
de transação / Edson Ifarraguirre Moreno. – Porto Alegre, 2004.
136 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.

1. Sistemas Digitais Complexos. 2. Circuitos Integrados
3. Linguagens de Programação. 4. Sistemas de Comunicação.
I. Título.

CDD 004.21
621.3817

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

Av. Ipiranga, 6681 - Prédio 16
CEP 90619-900 - Porto Alegre - RS - Brasil
http://www.pucrs.br/biblioteca

Fone: 0 (xx) 51 3320-3544
Fax: 0 (xx) 51 3320-3548
e-mail: bceadm@pucrs.br

AGRADECIMENTOS

Agradeço a CAPES por ter viabilizado este trabalho através do suporte financeiro, e ao PPGCC-FACIN, pelas instalações oferecidas para o desenvolvimento deste trabalho. Aos funcionários do PPGCC, José Carlos e Leandro pela compreensão e disponibilidade durante os dois anos de desenvolvimento deste trabalho. O trabalho de vocês é sem dúvida fundamental para que o nosso seja tão bem desenvolvido.

Agradeço ao corpo docente do PPGCC, em especial o prof. Ney Calazans pela orientação no mestrado. Ney, seu trabalho é excelente e sem dúvida és um dos meus exemplos de profissional a ser seguido. Agradeço ao prof. Fernando Moraes pelos contínuos conselhos. És sem dúvida um exemplo de profissional competente e perseverante que tentarei seguir. Obrigado por terem me proporcionado o convívio com vocês nestes últimos anos. Aos demais professores também agradeço pela amizade e pelas eventuais partidas de futebol, ainda que muitas vezes utilizassem nosso goleiro.

Agradeço aos amigos do mestrado. Esta jornada seria muito mais difícil sem a ajuda de todos vocês. As invasões ao apartamento do João para assistir as partidas de futebol. As confraternizações regadas com cerveja no XPUK. As partidas de futebol animadas e por vezes fraturadas. Desejo todo o sucesso a todos vocês.

Agradeço também aos colegas do GAPH. Obrigado “Bryan” pela sua incomparável sinceridade. Gostaria de agradecer a meu grande amigo Ewerson pelo convívio nestes últimos dois ou três anos. Obrigado Amory pelos conselhos sobre organização e desorganização no trabalho. Obrigado Noriaki pelo convívio e amizade. Acho que não vou conhecer outro japa tão maluco. Obrigado Moller sei lá por que, mas obrigado (te odeio). Obrigado Aline pela seriedade e exemplo de trabalho. Obrigado Leonel pelo futebol nosso de toda quinta e pela sua amizade. Opa, já ia esquecendo!!! Obrigado Ost pela alegria e eterno entusiasmo que por vezes parecia beirar a loucura. Acho que és a única pessoa que eu posso dizer que já te vi em outra vida ☺.

Agradeço aos bolsistas com os quais tive oportunidade de trabalhar nestes últimos anos. Em especial agradeço o esforço e empenho de Taciano Rodolfo, o “tatá”, e Giovani Zucolotto, o tiradentes do grupo. Sem o envolvimento de vocês pelo trabalho que foi proposto o caminho seria muito muito mais difícil.

Obrigado Mãe! Obrigado Pai! Sei o quanto vocês têm torcido por mim nestes últimos anos. Agradeço por compreenderem, ou não, meu isolamento em vários momentos em que talvez devesse estar muito mais presente. Vocês são meu exemplo do que deve ser feito, do que devo acreditar, do que devo dizer e de como devo agir. Acredito estar aprendendo a lição de como ser uma pessoa digna cada vez mais a cada dia que convivo com vocês. Muito obrigado ao senhor e a senhora, meu pai e minha mãe. Amo vocês.

Obrigado também as minhas irmãs e meu sobrinho. Sei que nosso convívio não é muito fácil muitas vezes, mas continuarmos nos amando e nos zelando é o que nos torna família. Obrigado por fazerem parte da minha vida.

Obrigado a minha noiva, Sílvia Casonato. Assim como meus pais, tu és a pessoa que tem creditado todas as tuas fichas em mim. Contigo aprendi muito, contigo estou aprendendo ainda várias coisas e contigo quero continuar aprendendo e plantando todo amor e felicidade para que um dia venhamos a colher os melhores frutos. Temos passado por várias privações nos últimos tempos, mas se Deus quiser seremos recompensados. Te amo!

RESUMO

O aumento da capacidade de circuitos integrados tem permitido a inclusão de um número crescente de módulos de hardware em um único dispositivo. Apesar das vantagens tecnológicas potencialmente providas por tal característica, a complexidade para a descrição e gerenciamento destes projetos de hardware aumenta na mesma proporção. Isto dificulta atender figuras de mérito tais como o tempo de chegada ao mercado de produtos tecnológicos, escalabilidade e qualidade de projeto. É possível superar ou pelo menos amenizar tal problema pelo aumento do nível de abstração em que se realiza a captura inicial de projeto, pelo incentivo maciço ao reuso e pela implementação de mecanismos mais eficazes de interconexão de componentes no interior de circuitos integrados. O presente trabalho contribui inicialmente com uma proposta de organização para níveis superiores de abstração de projeto a empregar durante a modelagem de sistemas digitais complexos. Com base nesta proposta, mostram-se resultados iniciais da comparação entre modelagens sistêmica e a modelagem RTL tradicional, a partir de um estudo de caso implementado em SystemC e VHDL. O domínio da complexidade do projeto passa pelo tratamento em separado dos aspectos de computação e comunicação do sistema, assumindo importância central neste trabalho. Em particular, endereça-se a preocupação crescente com a comunicação de módulos de hardware em um sistema digital complexo. Propõe-se aqui a modelagem abstrata de uma rede de comunicação intrachip parametrizável denominada Hermes, usando o nível de abstração de transação e empregando a linguagem SystemC. Esta modelagem é validada através de experimentos, e levou à proposta e ao desenvolvimento de ferramental específico de apoio ao projeto da comunicação intrachip. Através do processo de modelagem e dos resultados iniciais obtidos com a rede Hermes, evidencia-se um conjunto potencial de vantagens que pode ser auferido pelo uso de níveis superiores de abstração, durante a modelagem abstrata da comunicação e da computação para sistemas digitais complexos.

Palavras chave: Sistemas em um único chip, projeto de SoCs, níveis de abstração, nível de transação, nível de transferência entre registradores, SystemC, redes intrachip.

ABSTRACT

The increasing capacity of integrated circuits allows a rising number of IP cores to be inserted on a chip. Despite the technological advantage obtained by this, the complexity to describe and manage these projects increases in the same proportion. This threatens the achievement of design performance figures like time to market, design scaling and design quality. It is possible to overcome or at least to lessen the effects arising from design complexity by increasing the abstraction level in which design capture is formally undertaken, by the massive use of design reuse techniques and by implementing more efficient mechanisms for intrachip modules interconnection. This work has as an initial contribution a proposal to organize upper design abstraction levels for the modeling of complex digital systems. Based on this proposal, a case study implemented in both SystemC and VHDL shows examples of figures of merit to use for comparing systemic and traditional RTL modeling techniques. The road to master design complexity passes through the separate treatment of computation and communication aspects of complex digital system design. The concern with this separation and addressing intrachip communication issues are central to this work. Accordingly, the abstract modeling of an intrachip communication network named Hermes is proposed as another contribution. Modeling is performed at the transaction level of abstraction using SystemC as description language. The implementation is validated experimentally, and leads to the proposition of some specific communication design support tools, a third contribution. Through the modeling process and from the initial experimental results obtained with the abstract description of the Hermes network, it is already possible to evidence a set of potential advantages that can be achieved by the use of upper abstraction levels during the design of both communication and computation aspects of complex digital systems.

Keyword: System on chip, SoCs design, abstraction levels, transaction level modeling, register transfer level, SystemC, network on chip.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ASIC	*Application Specific Integrated Circuit*
CI	*Circuito Integrado*
CPI	*Clocks Per Instruction*
FCFS	*First Come First Served*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
FSMD	*Finite State Machine with Datapath*
GAPH	*Grupo de Apoio ao Projeto de Hardware*
HDL	*Hardware Description Language*
ISO	*International Standardization Organization*
LRS	*Least Recently Served*
LUT	*LookUp Table*
NoC	*Network on a Chip*
OCP	*Open Core Protocol*
OSI-RM	*Open Systems Interconnect Reference Model*
QoS	*Quality of Service*
RR	*Round Robin*
RTL	*Register Transfer Level*
SLDL	*System Level Design Language*
SoC	*System on a Chip*
TDM	*Time Division Multiplexing*
TLM	*Transaction Level Modeling*
VCI	*Virtual Core Interface*
VHDL	*Very high speed integrated circuit Hardware Description Language*

1 Introdução

O avanço tecnológico tem permitido cada vez mais a utilização da automatização de atividades cotidianas do homem. Sistemas computacionais são uma das formas modernas de prover esta automatização. Eles são formados por módulos de *hardware* e *software*, cujas operações combinadas provêem um serviço [MIC01]. Sistemas computacionais utilizados em controle de veículos, em comunicação à distância e na automação de residências [EDW97] são cada vez mais importantes para disponibilizar principalmente conforto e segurança.

Sistemas embarcados constituem uma classe de sistemas computacionais [GAJ94]. Apesar de possuírem a mesma estrutura interna de computadores, sendo formados por processadores, memórias, barramentos, e dotados de *software*, sistemas embarcados se caracterizam por não serem percebidos como computadores pelos seus usuários finais [EDW97]. Exemplo dessa característica é dado pela interação do usuário com um telefone celular e carros com freios ABS.

Gordon Moore observou, na década de 60 que a cada 18 meses a capacidade dos circuitos integrados (CIs) duplicava [SCH97], o que vem se mantendo válido até então. A esta observação denominou-se a "lei de Moore", a qual tem funcionado como um acionador da velocidade de evolução da indústria de semicondutores. Nota-se a velocidade de evolução pelo aumento exponencial do número de transistores por unidade de área de CI ocorrido nos últimos 20 anos [WES94].

O contínuo aumento do número transistores por unidade de área de CI permite que sistemas completos sejam montados em um único CI, criando-se o conceito de SoC (em inglês, *System-on-a-chip*) [RAJ00]. O desenvolvimento de SoCs é visto como uma tendência crescente em projetos de *hardware* [MAR01a] pois permite resolver parcialmente problemas tais como redução de área excessiva de sistema, e suavizar gargalo de desempenho na comunicação entre diferentes módulos de *hardware* [ZHU02]. Apesar disto, o contínuo aumento da capacidade dos CIs permite que mais módulos de *hardware* façam parte de um único SoC, o que torna um projeto complexo de descrever, validar e gerenciar [RAJ00]. Estes fatores propiciam o aumento do tempo de projeto, o que tende a invalidar a vantagem do uso de SoC para atender o *time-to-market*. *Time-to-market* é a necessidade que as indústrias têm de lançar um produto no mercado no prazo mais curto possível para obter maior lucratividade [RAU94]. Algumas propostas para superar tal problema são: a elevação do nível de abstração de captura de projeto, técnicas de reuso de projeto e métodos mais eficientes de interconexão de módulos de *hardware* em SoCs. No presente trabalho são propostos níveis de abstração superiores ao de transferência entre registradores (em inglês *Register transfer Level*, RTL) para projeto de *hardware* e é implementada uma rede *intrachip* (em inglês, *Networks on a chip*, NoCs), como mecanismo de comunicação mais eficiente se comparado a barramentos, e que utiliza uma interface padronizada para facilitar seu reuso em projetos.

O presente Capítulo organiza-se da seguinte forma: na Seção 1.1 introduz-se a definição sobre núcleos de propriedade intelectual e SoCs. A Seção 1.2 adianta-se a motivação para este trabalho, tendo por parâmetro os problemas atuais vinculados a projeto de SoCs e algumas alternativas de solução. Na Seção 1.3 listam-se os objetivos estratégicos e específicos, enquanto que na Seção 1.4 a organização do trabalho é mostrada.

1.1 Núcleos de propriedade intelectual e SoCs

Núcleos de propriedade intelectual (núcleos IP ou IPs ou núcleos) são módulos de *hardware* complexos, digitais ou analógicos, estruturados em diferentes formatos e níveis de abstração [GUP97]. Núcleos IP são classificados em três categorias distintas, de acordo com a forma na qual são disponibilizados, quais sejam:

- *Núcleos Duros*: são otimizados para uma tecnologia específica e não podem ser modificados pelo projetista. Estes *núcleos* possuem um *layout* e planta baixa (*floorplanning*) pré-definida.

- *Núcleos Firmes*: *netlist* (lista de conexões) gerado para a tecnologia empregada, que muda de fabricante para fabricante. Este tipo de *núcleo* pode ser parcialmente parametrizado pelo projetista, permitindo que sua arquitetura seja adaptada à necessidade do projeto. Entretanto, como o *netlist* é específico para uma dada tecnologia, existe a dificuldade do uso de componentes fornecidos por fabricantes diferentes.

- *Núcleos Flexíveis*: descritos em linguagens de descrição de *hardware* (em inglês, *Hardware Description Language*, HDL), oferecendo flexibilidade e independência de tecnologia. O *núcleo* pode ser modificado e empregado com tecnologias de diferentes fabricantes.

SoCs são definidos como sistemas complexos que integra diferentes núcleos formando um sistema completo em um único CI [MAR01a]. Os núcleos IP de um SoC podem ser processadores programáveis, memórias, lógica de aplicação específica, além de controladores de entrada e saída, conforme ilustra a Figura 1.

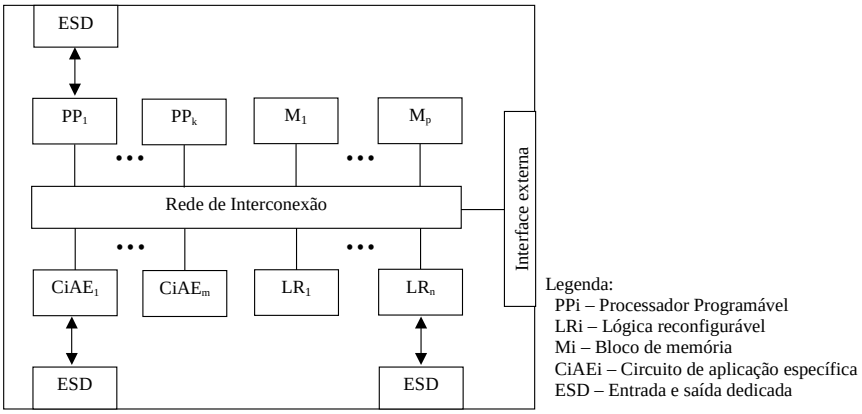


Figura 1 – Estrutura genérica de um SoC. Um SoC pode ser composto por processadores programáveis (PP), memórias (M), circuitos de aplicação específica (CiAE) de alto desempenho, interfaces com periféricos, lógica reconfigurável (LR) e módulos de entrada e saída dedicados(ESD).

A *especificação* de um SoC define quais requisitos o sistema deve atender, tais como funcionalidade a ser atingida, desempenho desejado e consumo máximo de potência. À especificação inicial segue-se a fase de projeto do SoC. Esta fase pressupõe duas atividades principais: a modelagem ou descrição e a validação. A *modelagem* consiste em partir da especificação e produzir uma descrição do sistema ou de parte deste usando um formalismo escolhido. Exemplos de tais formalismos são linguagens ou métodos gráficos de representação de funcionalidade, estrutura, etc. A *validação* consiste em exercitar a descrição

produzida na fase de modelagem para verificar suas propriedades. Assim, pode-se garantir, em maior ou menor grau, o atendimento da especificação pelo resultado da modelagem. Devido à complexidade de SoCs, deve-se sempre iterar as fases de modelagem e validação um certo número de vezes, em um processo de refinamentos sucessivos. Cada iteração está associada a um *nível de abstração de projeto*, onde o resultado da última modelagem realizada representa a especificação do próximo nível de abstração, e assim por diante.

Segundo Cai e Gajski [CAI03a, CAI03b], pode-se classificar métodos de projeto de SoCs em três grupos de classes, quais sejam:

Projeto baseado em componentes (*Component-based design*): Esta abordagem utiliza uma metodologia ascendente (em inglês, *bottom-up*). Nela, são utilizados núcleos IP e *software* pré-definidos e pré-validados. A partir destes módulos, um SoC é montado para atender uma funcionalidade específica ou mesmo definir uma plataforma para projetos futuros.

Síntese em nível de sistema (em inglês, *System-level synthesis*): Esta abordagem utiliza uma metodologia descendente (em inglês, *top-down*). O passo inicial do projeto de SoC se dá pela descrição abstrata do sistema. O passo seguinte são refinamentos sucessivos, finalizando quando um conjunto de núcleos IP sintetizável e a descrição do *software* associado são obtidos.

Projeto baseado em plataformas (*Platform-based design*): Esta abordagem utiliza a metodologia mista ascendente-descendente de geração de projeto (em inglês, *meet-in-the-middle*). Nesta metodologia, o SoC é obtido a partir do mapeamento de módulos de *hardware* e software na plataforma. Neste caso, módulos de *hardware* e suas interconexões já estão presentes na plataforma. Esta abordagem de projeto visa atender a um domínio de aplicações específicas, tal como o processamento de imagens.

A Tabela 1 apresenta uma comparação qualitativa das abordagens de concepção de projeto de SoCs segundo aspectos de descrição de projeto, complexidade de descrição de módulos e de suas interconexões e o tempo de projeto estimado. Aspectos de modelagem e validação específicos não foram levados em conta, por representarem a mesma complexidade independente da abordagem utilizada.

Tabela 1 - Comparativo do esforço de trabalho envolvido no projeto de SoCs em diferentes abordagens.

		Síntese em nível de sistema	**Projeto baseado em plataformas**	**Projeto baseado em componentes**
Descrição de projeto		Refinamentos sucessivos	Mapeamento de módulos	Montagem de módulos pré projetados
Complexidade de:	**Descrição de módulos**	Alta	Média	Nenhuma
	Interconexão de módulos	Baixa	Nenhuma	Depende
Tempo estimado de projeto		Grande	Médio	Depende

A síntese em nível de sistema inicia com a modelagem e, depois de criada uma primeira especificação executável, prossegue com passos de refinamento até chegar a uma descrição válida de entrada para ferramentas de síntese automatizada. Para projetos baseados em plataformas, a funcionalidade do sistema é atingida a partir do momento em que os módulos pré-projetados são mapeados na plataforma com o auxílio de ferramentas de apoio. Para projetos baseados em componentes, a descrição do projeto fica restrita à seleção e interconexão de componentes ou módulos pré-disponibilizados.

Na síntese em nível de sistema, a complexidade de descrição de módulos é considerada alta por envolver muitos passos de refinamento, metodologias de trabalho e ferramentas de apoio. Para projetos baseados em plataformas, esta complexidade é menor, pois as ferramentas de apoio têm como função específica mapear um componente do sistema para estruturas pré-definidas de uma dada plataforma. Projetos baseados em componentes possuem complexidade nula para descrição de módulos, pois tais módulos são reaproveitados de projetos anteriores ou adquiridos de terceiros.

A complexidade de interconexão de módulos tende a ser um dos pontos cruciais para a tomada de decisão sobre a abordagem de projetos do SoC a adotar. Para síntese em nível de sistema, o refinamento de módulos de forma manual e a possibilidade de personalizar a interface de comunicação com os demais módulos do projeto de SoC diminuem a complexidade de interconexão dos módulos implementados. Para projetos baseados em plataformas, a interconexão entre os diferentes módulos já está projetada, bastando apenas ser mapeada por ferramentas de apoio ao projeto. Para projeto baseado em componentes, esta complexidade pode ser alta ou média. Pode ser alta, se caracterizada pelo uso de componentes heterogêneos no que tange a freqüência de operação ou mesmo a largura de banda para comunicação. O mesmo pode ocorrer devido ao não uso de interfaces de comunicação padronizadas, o que exige do projetista o desenvolvimento de lógica de adaptação entre os diferentes módulos. Com o uso de interfaces padronizadas, tal complexidade pode ser reduzida.

O último aspecto a ser analisado é o de tempo estimado de projeto. Para síntese em nível de sistema, o tempo estimado é grande, pois os passos de refinamento e a validação tendem a consumir tempo de projeto considerável. Para projetos baseados em plataformas, este tempo tende a diminuir, pois se pressupõem que estruturas de validação da plataforma pré-existam. Adicionalmente, o uso de ferramentas de mapeamento de módulos pré-projetados na plataforma contribui para a aceleração do projeto. Para projetos baseados em plataformas, o critério tempo de projeto pode ser grande ou médio. Esta variação está principalmente ligada a questões de descrição de projeto e interconexão dos diferentes módulos.

1.2 Motivação

O crescimento do uso de SoCs é visto como uma certeza, devido a vantagens, tais como a possibilidade de diminuição da área e de tempo de projeto, e aumento de desempenho de comunicação. O aumento da capacidade dos CIs habilita a presença de cada vez mais núcleos IP em um único CI e sua heterogeneidade, o que torna complexa a interconexão entre os núcleos IP e a validação global do SoC. Com isto, a concepção e o gerenciamento dos projetos tornam-se tarefas extremamente complexas. Soluções parciais para garantir as vantagens do uso de SoCs e minimizar suas desvantagens podem ser obtidas via três tipos de ação: *(i) emprego de técnicas de reuso de projetos, (ii) fomentar a evolução dos métodos de interconexão de módulos on-chip*, e pela *elevação dos níveis de abstração* das fases de captura de projeto. Cada uma destas ações é discutida a seguir.

1.2.1 Reuso

O reuso de núcleos IP permite a redução do tempo de projeto. Com o aumento da capacidade dos CIs, a possibilidade de inserir cada vez mais núcleos IP em um único SoC também cresce. Todavia, núcleos IP desenvolvidos para uma aplicação específica ainda não seguem, hoje, uma padronização da sua interface externa. O grande número de núcleos IP em um SoC e a uso ineficiente dos padrões de interconexão faz com que a reutilização de módulos torne-se uma tarefa complexa.

1.2.2 Evolução de métodos de interconexões

O uso de meios de comunicação compartilhados, tais como barramentos, para interconectar módulos de um SoC torna-se ineficiente à medida que cresce o número de módulos de *hardware* em um único CI. Este problema deve-se principalmente a fatores elétricos, incluindo o carregamento capacitivo dos fios do barramento, o conseqüente aumento de consumo de energia, além da limitação topológica de apenas permitir uma mensagem trafegando no meio de comunicação a cada instante.

Cada vez mais núcleos IP processadores e outros componentes reutilizáveis têm sido integrados em um único CI. Estudos prevêem que futuramente SoCs sejam compostos por centenas de núcleos IP [BEN01, GUE00]. A comunicação entre núcleos IP costuma ser realizada seguindo duas abordagens distintas: canais dedicados ou barramentos. Canais dedicados não são vistos como uma abordagem eficiente, por dificultarem o reuso de projetos. Barramentos superam este problema, porém apresentam limitações tais como a diminuição da freqüência de operação com o aumento da densidade de SoCs, devido ao uso compartilhado do canal de comunicação [ZEF02] e à escalabilidade limitada. Algumas práticas podem ser adotadas para diminuir estas limitações, tal como a utilização de arquiteturas de barramento hierárquicas. Exemplos destas são *CoreConnect* [IBM02] e *Amba* [ARM05]. Todavia, esta abordagem apresenta problemas, tais como a possibilidade de comunicação bloqueante atingindo todo o barramento, a diferença entre as freqüências de operação e largura de banda entre cada barramento e principalmente a forma desestruturada de montagem destes barramentos. Todos estes fatores afetam a escalabilidade em sistemas de grande porte.

Uma proposta que supere tais problemas e que venha a atender as futuras necessidades de projetos é obtida a partir do uso de esquemas de comunicação *intrachip* mais elaboradas. Conceitos derivados dos domínios de redes de computadores, sistemas distribuídos e arquivos paralelos vêm sendo adaptados para o ambiente interno a um CI complexo estendendo o conceito de redes intrachip (em inglês *network on chip* ou NoCs). O uso de NoCs como mecanismo de comunicação entre núcleos IP de um projeto de SoC [BEN01, KUM02, WIN01]. Tem sido crescentemente por estas apresentarem, quando comparadas a barramentos tradicionais, características de melhora de parâmetros tais como: eficiência energética, confiabilidade, reusabilidade, comunicação não bloqueante e escalabilidade de largura de banda [GUE00, MIC02].

1.2.3 Elevação dos níveis da abstração de captura de projeto

A elevação do nível de abstração de projeto também é vista como uma contribuição para aumentar a eficiência de concepção dos projetos de *hardware*. Novamente, devido ao aumento do número de núcleos IP em um SoC e ao aumento da complexidade da tecnologia de interconexão destes IPs, descrições em RTL são difíceis de se desenvolver e alterar. A elevação dos níveis de abstração de projeto para os chamados *níveis sistêmicos* permite que detalhes sejam ocultados, tornando mais fácil descrever a funcionalidade de cada módulo e suas interconexões. A lacuna existente entre o nível de especificação e os de concepção do projeto usados hoje pode ser preenchida com níveis intermediários como o *nível não temporizado* e o *nível de modelagem de transações* (em inglês, *Transaction Level Modeling*, TLM) conforme proposto, por exemplo, por Arnout em [ARN02].

SystemC é uma linguagem de descrição de projeto em nível de sistema (em inglês, *System Level Design Language*, SLDL) que prove mecanismos para a modelagem projetos em RTL e de níveis mais abstratos. Esta linguagem atende à necessidade de elevação do nível de abstração de captura e validação de projeto. Construída como um conjunto de classes sobre a linguagem C++, SystemC foi desenvolvida com a finalidade de preencher a lacuna entre a especificação do projeto de sistemas computacionais e sua concepção. SystemC permite a descrição de *hardware* e *software* em um ambiente homogêneo, facilitando o processo de compreensão, descrição e validação de projeto. Com isto, a detecção de erros funcionais pode ser realizada nos estágios iniciais de desenvolvimento, diminuindo os esforços e custos de projeto.

1.2.4 Conclusão

Observada a tendência da utilização de SoCs em projetos de sistemas e as vantagens daí advindas, nota-se à complexidade intrínseca de projetos com múltiplos núcleos IP. Assim, a aplicação de técnicas de reuso de núcleos IP, o emprego de novas abordagens para definir a interconexão destes núcleos, e a elevação dos níveis de abstração para a descrição de projetos são vistos como temas relevantes a serem investigados.

1.3 Objetivos

Este trabalho contribui para o avanço da pesquisa em redes de comunicação *intrachip*. Um dos aspectos importantes neste contexto é disponibilizar técnicas que facilitem o processo de projeto e as tomada de decisão sobre características de SoCs que utilizem como meio de comunicação NoCs. Com isto, incrementa-se o desenvolvimento de uma cultura de planejamento de projeto e validação funcional em altos níveis de abstração antes da implementação da comunicação em SoCs em baixos níveis de abstração. O desenvolvimento das técnicas mencionadas teve como alvo a construção de ferramentas que contribuem para o aumento de produtividade e para a eficiência da análise do processo de projeto da comunicação em SoCs que emprega NoCs. Para alcançar esta contribuição, o trabalho partiu do conjunto de objetivos estratégicos e específicos descritos a seguir.

O objetivo estratégico principal foi aumentar o corpo de conhecimentos sobre a modelagem de NoCs em altos níveis de abstração. Ao lado deste, considerou-se também importante aumentar o ferramental teórico e prático disponível para avaliar a relação custo-benefício entre realizar a captura formal de projetos a partir de altos níveis de abstração e fazer o mesmo a partir do nível RTL.

Para atingir os objetivos estratégicos, alguns objetivos específicos foram delineados. O principal destes foi desenvolver pelo menos uma proposta de NoC, incluindo o estudo de diferentes arquiteturas e algoritmos relacionados a estas estruturas de comunicação *intrachip*. Adicionalmente, objetivou-se estudar e empregar ambientes que suportem a modelagem de NoCs em níveis de abstração superiores ao RTL. Estes ambientes permitem não apenas modelar *hardware*, mas também avaliar e prototipar sistemas, ainda que baseado em fluxos apenas parcialmente automatizados de projeto, o que é obtido, por exemplo, através da integração do ambiente em questão com sistemas comerciais de síntese.

Outro objetivo específico delineado deriva diretamente dos anteriores. Este consiste e proporem, e se possível implementa, ferramentas de apoio que capturem as técnicas de projeto desenvolvidas ao longo da implementação de NoCs.

1.4 Organização do documento

O restante deste documento está assim organizado. O Capítulo 2 apresenta a revisão do estado da arte para questões pertinentes ao trabalho: NoCs, SLDL, níveis de abstração de projetos de SoCs e ferramental disponível para modelagem em nível sistêmico. Adicionalmente, este Capítulo inclui as propostas da NoC alvo do trabalho e dos níveis de abstração adotados. O Capítulo 3 apresenta uma comparação inicial entre os compromissos envolvidos no uso de linguagens sistêmicas versus o uso de linguagens RTL para o projeto de *hardware*, através de um estudo de caso. No Capítulo 4 é apresentada a abordagem para a implementação de NoCs em nível de transação. No Capítulo 5 são apresentadas as conclusões sobre o trabalho e perspectivas de trabalhos futuros.

2 Estado da arte

Nos últimos anos, diversas pesquisas têm sido realizadas com a finalidade de aumentar a produtividade através do uso de técnicas que facilitem a modelagem, a descrição, a validação e o gerenciamento de projeto de SoCs [ARN02, KEU00, MAR01a, MIC02]. Diferentes temas de pesquisa são investigados para encontrar soluções parciais ao problema intrínsecos nos projetos de SoCs. Entre estes temas destacam-se ***(i)*** o uso de níveis de abstração para modelagem de projetos, ***(ii)*** linguagens de descrições de sistemas que permitam maior abstração (SLDLs), ***(iii)*** ambientes que viabilizem a modelagem abstrata e ***(iv)*** redes *intrachip*. No presente Capítulo é feita a revisão do estado da arte destes quatro temas.

2.1 Níveis de abstração para modelagem de projeto

A crescente complexidade do processo de projeto de sistemas digitais implica a decomposição hierárquica deste em um conjunto de passos de projeto. Um mecanismo fundamental para conduzir a decomposição do processo de projeto é a abstração de informações. *Abstração* é o nome que se dá ao processo de representar um objeto via um conjunto limitado de informações representando aspectos relevantes deste para um dado tipo de manipulação. Também se pode usar o termo para referir-se ao resultado do processo em si. Segundo Calazans [CAL98], pode-se definir *nível de abstração* no contexto de projeto de sistemas computacionais como um conjunto de descrições de projeto com o mesmo grau de detalhamento, onde uma descrição de projeto é o resultado de um dado processo de abstração.

Durante o projeto de um SoC, um grande número de descrições de projeto com diferentes níveis de detalhamento são manipuladas. Por exemplo, diagramas de portas lógicas contêm muito menos detalhes que uma descrição elétrica do tipo SPICE, mas ambos são descrições do projeto relevantes em algum contexto. Os primeiros podem ser usados para rapidamente validar blocos grandes, com dezenas de milhares de portas lógicas, enquanto a partir de descrições SPICE é possível obter validações com alto grau de detalhamento para pequenos blocos, tipicamente com não mais que alguns milhares de transistores. O número de níveis de abstração manipulados durante o projeto de sistemas complexos tem aumentado à medida que aumenta a complexidade dos projetos. Descrições de projeto RTL são atualmente o estado da arte para ferramentas de síntese, conforme Jerraya et al. [JER01].

Com a crescente complexidade dos projetos de sistemas computacionais e a necessidade de aumento de produtividade de projeto, o uso de níveis de abstração superiores ao RTL tem sido proposto, tal o nível de sistema. Aliada a proposta de modelagem em nível de sistema está a separação de características do projeto, tal como computação e comunicação.

2.1.1 Separação entre computação e comunicação

Um componente essencial de novos paradigmas de projeto de sistemas é a divisão de interesses, ou seja, a separação dos vários aspectos de projeto, para permitir exploração mais eficiente de soluções alternativas para problemas, conforme Keutzer et al. [KEU00]. A principal justificativa para a separação de aspectos de projeto está relacionada à natureza dos problemas. Alguns aspectos de projeto passíveis de separação, conforme [KEU00] são:

- A função e a arquitetura;

- A comunicação e a computação.

No presente trabalho, os principais aspectos de projeto cuja separação será considerada são: a comunicação e a computação. Estes são considerados os principais aspectos a serem separados segundo e.g. Keutzer [KEU00], Cesário [CES02] e Fayad [FAY00]. Entende-se aqui por computação, o comportamento interno de um núcleo IP, sem preocupação de valores serão compartilhados com outros núcleos IP. Entende-se aqui por comunicação o comportamento que descreve a troca de dados entre dois ou mais núcleos IP. Este entendimento pode ser exemplificado por um produtor/consumidor, onde, focando o núcleo IP produtor, a computação limita-se à descrição do comportamento que gera valores aleatórios e a comunicação limita-se à descrição do comportamento de transferência os dados. A abordagem de separação de características permite concentrar-se na solução de problemas de computação, abstraindo detalhes de comunicação. Quanto mais para o final do projeto do sistema a comunicação puder ser refinada, mais maleáveis serão o desenvolvimento e gerenciamento do projeto. Independente da separação de características que se pretende adotar, o uso de níveis de abstração específicos é fundamental.

2.1.2 Níveis de abstração para computação

Arnout [ARN02] propôs dois níveis de abstração superiores ao RTL: o *nível não temporizado*, e o *nível de transação*. No *nível não temporizado*, o objetivo é a descrição da funcionalidade do projeto sem precisão temporal, ou seja, sem a noção do tempo necessário para executar um determinado algoritmo. Neste nível não há separação dos módulos de *hardware* e *software*, apenas a geração de uma especificação executável. No *nível de transação*, a funcionalidade de cada módulo é descrita de forma abstrata, assim como no nível não temporizado. Todavia, como há separação entre os módulos, há necessidade de comunicação, sendo esta baseada em ciclos de transação. Com isto, pode-se garantir o correto funcionamento do sistema via simulações mais eficientes, visto que o detalhamento das descrições é menor.

Outra proposta de níveis de abstração em projetos de sistemas computacionais é apresentada pela Synopsys em [SYN02a]. Nesta referência, são definidos como níveis de abstração superiores ao RTL os níveis *não temporizado, temporizado* e *de transação*.

No *nível não* temporizado [SYN02a], os algoritmos que descrevem o comportamento dos módulos são mais facilmente capturados, verificados e otimizados, devido a alto grau de abstração das descrições. Neste nível, os módulos do projeto que estão sendo desenvolvidos comunicam-se ponto-a-ponto através de canais abstratos do tipo fila, acessados via operações bloqueantes de leitura e escrita. Assim, projetistas são beneficiados de duas formas. Primeiro, a facilidade de desenvolvimento, pois a comunicação é simples e a sincronização é implícita. Segundo, uma maior eficiência de simulação, pois muitos detalhes de implementação da computação são abstraídos neste nível. A comunicação entre os módulos do sistema, neste nível, pode ser validada de forma abstrata otimizando o processo de transmissão e recepção de informações do ponto de vista funcional. O desempenho obtido pode ser avaliado com métricas tais como a taxa de erro de transferência de bits entre a fonte transmissora e o destino. A inserção de erros na transferência dos bits permite avaliar a perda de desempenho quando muitos erros ocorrem, o que contribui para a descrição do algoritmo que consuma menos tempo de execução e corrija erros. Outra vantagem do uso deste nível é a validação de um algoritmo logo nos primeiros estágios do projeto, onde é verificada a real funcionalidade do mesmo. Como exemplo, pode-se descrever um algoritmo de roteamento o qual pode ser garantida a não ocorrência de problemas tais com *deadlock* ou *livelock*.

No *nível temporizado*, empregam-se modelos funcionais dotados de informações de atraso. A modelagem de atrasos pode considerar atrasos de processamento, de comunicação, ou ambos. Este nível de abstração é usado para analisar os efeitos de latência no comportamento do sistema e a arquitetura do sistema nos estágios iniciais do projeto. Através de especificações de atraso de processamento, é possível definir modelos funcionais temporizados para determinar se o sistema vai gerar resultados em tempo hábil ou não, conforme a especificação. Normalmente, empregam-se aqui esquemas de comunicação ponto-a-ponto, a exemplo do que ocorre no nível não temporizado. Modelos funcionais temporizados são geralmente usados para analisar o compromisso entre implementações alternativas de *hardware* e *software* nos estágios iniciais de projeto. Isto é feito pela avaliação do impacto do mapeamento de processos para ambos, *hardware* e *software* [SYN02a]. A idéia de tempo neste nível é normalmente expressa através de estimativas de ciclos de relógio.

Nota-se então que os modelos funcionais descritos tanto no nível não temporizado quanto no temporizado não fazem distinção entre módulos de *hardware* ou *software*. A partir da avaliação do desempenho e/ou custo de projeto para a separação dos módulos em *hardware* ou *software*, é definido o *nível de transação*. No *nível de transação*, os componentes da arquitetura, tais como memórias, unidades aritméticas, e geradores de endereço comunicam-se através de meios compartilhados, como barramentos. Barramentos costumam envolver mecanismos de arbitragem para resolver conflitos de acesso ao meio de comunicação, que ocorrem quando vários componentes requisitam acesso a este simultaneamente. Um grande esforço é necessário para projetar e verificar modelos comportamentais em HDL, pois sua simulação é pouco eficiente. Além disto, não existem técnicas universais para transformar todas as partes de uma descrição HDL puramente comportamental em *hardware*. A depuração de *software* normalmente requer a simulação de *hardware* em níveis próximos ao RTL, o que pode tornar a verificação de um sistema complexo inviável.

Com a modelagem em *nível de transação*, a comunicação não necessita de uma implementação com alto grau de detalhamento. O comportamento desta comunicação é modelado e expresso em termos de transações. O conceito de transação varia de autor para autor. Por exemplo, segundo Pasricha [PAS02] *transação* é uma troca de dados qualquer entre módulos, independentemente do protocolo de troca empregado ou do tempo necessário para efetuar esta troca. Esta definição foi adotada para o presente trabalho visto que o objetivo é exatamente prover interfaces de comunicação sem detalhamento da implementação e sem o uso de conceito de tempo.

Descrições em nível de transação são mais fáceis de desenvolver e usar se comparadas a descrições RTL, devido ao baixo grau de detalhamento de implementação dos módulos. A modelagem no nível de transação cria uma especificação executável do modelo que simula ordens de magnitude mais rápido que modelos RTL. Esta especificação executável provê um modelo abstrato que desenvolvedores de *software* podem utilizar para testar seus códigos no contexto de um SoC. Esta modelagem também provê um mecanismo para análise de comprometimento de *hardware* e *software*, e permite um nível de confiança na verificação funcional do sistema logo nos primeiros estágios do processo de projeto. As diferentes equipes de um projeto devem preferencialmente comunicar-se no nível de transação. Problemas em projetos são bem menos custosos se detectados e resolvidos em nível de transação do que em RTL.

2.1.3 Níveis de abstração para comunicação

No passado, níveis de abstração eram utilizados para descrever a computação dos módulos de um sistema, sendo que a descrição da comunicação entre módulos era vista como acessória. Por este motivo, a descrição da comunicação era realizada da mesma forma que a descrição da computação. Contudo, a

crescente escala de interconexão de módulos em sistemas leva ao aumento da importância de interconexões. A partir deste momento surge então a necessidade de processos de modelagem específica para a comunicação entre módulos, usando um conjunto de níveis de abstração próprios. Exemplos de propostas de níveis de abstração para tratar questões de comunicação são apresentados por Haverinen et al. [HAV02] e Jerraya et al. [JER01].

Na Figura 2é apresentada a abordagem proposta por Haverinen et al. [HAV02]. Esta proposta descreve diferentes níveis de abstração em uma modelagem de comunicação onde o nível mais baixo de abstração, RTL, é descrito pela implementação detalhada do protocolo OCP. O protocolo OCP é uma proposta de padronização de interfaces para a interconexão de núcleos IP [OCP05a].

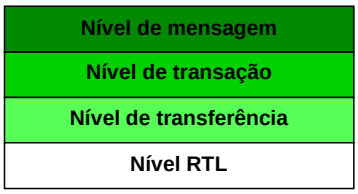


Figura 2 - Níveis de abstração na modelagem de comunicação proposta por Haverinen et al. [HAV02].

O *nível de mensagem* não é temporizado, ou seja, não inclui informações de tempo para executar uma transação. A transferência de dados entre os núcleos envolve a passagem de vários dados, que podem ser tipos abstratos, tais como registros estruturados em *software* (e.g. construções *struct* na linguagem C). A mensagem a ser trocada representa toda a informação necessária para que os núcleos IP possam se comunicar.

O *nível de transação* é temporizado, mas abstrai o conceito de ciclos de relógio. O termo temporização neste caso refere-se a correta ordem das transações que ocorrem entre os diferentes núcleos IP. Uma única transação neste nível pode envolver a troca de um ou diversos dados, assim como no nível de mensagem. O tempo de execução decorrido de uma transação é estimado internamente nos núcleos que trocam dados. Sistemas descritos neste nível são independentes dos protocolos de comunicação específicos. Assim, para a comunicação entre núcleos IPs o dado que será enviado a outro núcleo IP é repassado para um mecanismo que se encarregará de realizar a transferência, sendo que esta transferência pode ocorrer através de um *handshake* ou ainda através da implementação de uma *fifo*, ficando implícito aos núcleos IP.

O *nível de transferência* se caracteriza pelo uso de descrições comportamentais com temporização baseada em ciclos de relógio, mas sem detalhamento da estrutura interna de hardware, o que o diferencia de em RTL. Resumidamente, neste nível há o detalhamento da interface do núcleo IP, mas o comportamento interno que descreve a estrutura de comunicação permanece em um grau elevado de abstração o que o torna descrição não sintetizável. Neste nível é descrito um protocolo de comunicação específico, mas não sintetizável, como por exemplo, Amba para descrever um barramento mestre-escravo.

Jerraya [JER01] propõem 3 níveis de abstração superiores ao nível RTL para abstrair detalhes de comunicação, quais sejam o *nível de serviço,* o *nível de mensagem* e o *nível de driver*.

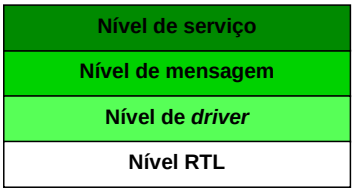


Figura 3 – Níveis de abstração na modelagem de comunicação proposta por Jerraya et al. [JER01].

O *nível de serviço* é definido como a implementação de métodos que descrevem ações de um módulo, denominados *serviços*. Neste nível, um módulo pode requisitar um serviço que é disponibilizado por outro. Questões como camadas de protocolos de comunicação a serem adotadas, estratégias de conexão e questões de temporização não são detalhadas neste nível de abstração. Assim, a funcionalidade é mais facilmente descrita. Exemplos de serviços que podem ser requisitados são pedidos de impressão ou transferência de arquivos. Com este tipo de abstração, a expressão de paralelismo e temporização é facilmente suportada.

No *nível de mensagem*, a comunicação é modelada através de canais ativos com capacidade de interconectar módulos independentes, ocultando questões de protocolos de comunicação específicos e tamanhos de dados. Assim, canais implementando métodos de envio e recebimento de dados são modelados neste nível. Estes canais executam transferências ponto-a-ponto de dados entre módulos.

No *nível de driver*, a comunicação entre os módulos é definida através de uma conexão lógica, que realiza trocas de dados com tipos fixos ou enumerados. O tempo de comunicação é diferente de zero, mas é previsível, visto que o tamanho e a estrutura dos dados são bem definidos, o mesmo ocorrendo com os protocolos de transmissão de dados. Assim, modelos de representação de barramentos mestre-escravo são facilmente definíveis.

Com as características dos níveis de abstração apresentados nesta Seção, é possível observar as vantagens decorrentes de descrever processamento e comunicação em níveis de abstração superiores ao RTL. A definição é empregada em modelagem no *nível de transação* para projeto de *hardware* está sendo ativamente empregado e pesquisado. Alguns trabalhos, tais como [MOR02] e [PAS02], descrevem algumas das vantagens da descrição neste nível de modelagem.

2.1.4 Proposta de organização de níveis de abstração para o domínio de hardware

Para o presente trabalho, foram definidos níveis de abstração que melhor descrevam os possíveis passos de refinamento a adotar na descrição de aspectos de computação e comunicação dos núcleos IP. Neste trabalho, o objetivo não é descrever um SoC, mas sim modelar núcleos IP, em particular NoC. O uso de níveis mais abstratos de modelagem tem como objetivo é apresentar justificativas à proposta de modelagem em níveis mais abstratos em questões tais como funcionalidade, custos, consumo de área, desempenho e potência. A abordagem aqui proposta, contudo, possui potencial para habilitar a modelagem em nível de sistema, onde há necessariamente a definição de um fluxo de projeto completo com uma definição clara das etapas de refinamento. Assume-se o fluxo de projeto de SoC em nível de sistema representado pela Figura 4.

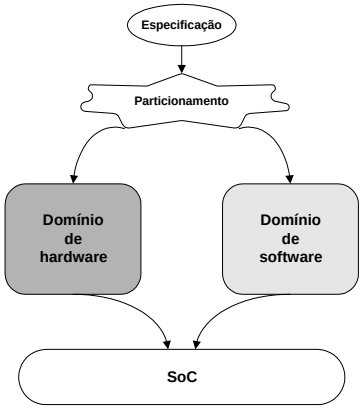


Figura 4 – Estrutura geral de fluxos de projeto de SoCs.

A primeira etapa do fluxo de projeto consiste do processo de *especificação*, tendo como prioridade a criação de um modelo de especificação executável, onde não há particionamento entre módulos, sejam de *hardware* ou *software*. Tal modelo é a seguir particionado em domínios de *hardware* e *software*, conforme análise de requisitos de desempenho e custo a serem atingidos no projeto. Esta ação se dá na etapa de *particionamento* do projeto. O problema de particionamento está fora do contexto deste trabalho. No *domínio de software*, descrevem-se recursos como o sistema operacional embarcado que será utilizado, algoritmos parametrizáveis para atender um dado domínio de aplicação e *drivers* de interfaceamento com módulos de *hardware* descritos. O *domínio de software* também é considerado fora do contexto deste trabalho. No *domínio de hardware*, descrevem-se módulos que executam a computação e/ou a comunicação. Isto inclui a descrição de microprocessadores, controladores de memória, barramentos e NoCs podem ser descritos. O *domínio de hardware* é o foco deste trabalho. O resultado final da junção tanto do domínio de *hardware* e *software* é o sistema completo, ou SoC.

Com base no fluxo de projeto de SoCs apresentado anteriormente, propõem-se níveis de abstração para representar o foco de hardware de um sistema em cada fase do projeto até sua implementação. A Figura 5 representa de forma resumida a proposta. Nota-se que os níveis de abstração para descrição do *software* agregado ao sistema foram desconsiderados nos níveis de abstração propostos.

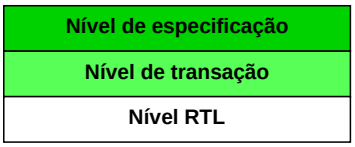


Figura 5 – Níveis de abstração para modelagem de hardware propostos neste trabalho.

No *nível de especificação* é fornecida uma representação abstrata do que se pretende atingir com o sistema, sem a preocupação de particionar esta em módulos de *hardware* ou *software*, ou ainda detalhar sua temporização. Esta descrição se denomina *modelo de especificação* e deve ser executável. O objetivo deste nível de abstração é propor algoritmos que descrevam a solução do problema com algum grau de otimilidade. Nota-se que o objetivo deste nível não é a análise de desempenho ou tomada de alguma decisão do projeto, mas sim a representação da funcionalidade a ser adotada.

O particionamento da especificação executável em módulos de *hardware* e/ou *software* caracteriza o fim do nível de especificação e o início da implementação de um *domínio de hardware* e um *domínio de software*. Dentro do *domínio de hardware* o *nível de transação* é o nível mais abstrato. O *nível de transação* caracteriza-se por ser a primeira fase de particionamento de um projeto. Ali, os núcleos IP são

então modelados seguindo um conjunto de graus de detalhamento, conforme apresentado na Figura 6, tanto para as características de comunicação quanto comunicação. Define-se neste trabalho que o nível de transação não se concentra unicamente na modelagem da computação, mas também na da comunicação. No nível de transação é realizada a busca pela eficiência na descrição de modelos de computação e de comunicação. O principal objetivo deste nível de abstração é dar apoio à implementação de diferentes algoritmos para descrever a solução a ser adotada e, a partir da análise e detecção do algoritmo mais interessante, refiná-lo. Como modelos de computação e comunicação não compartilham hoje um formalismo único eficiente para descrever o detalhamento destes dois aspectos de projetos, propõem-se aqui dois subconjuntos hierárquicos distintos de níveis de abstração dentro do nível de transação.

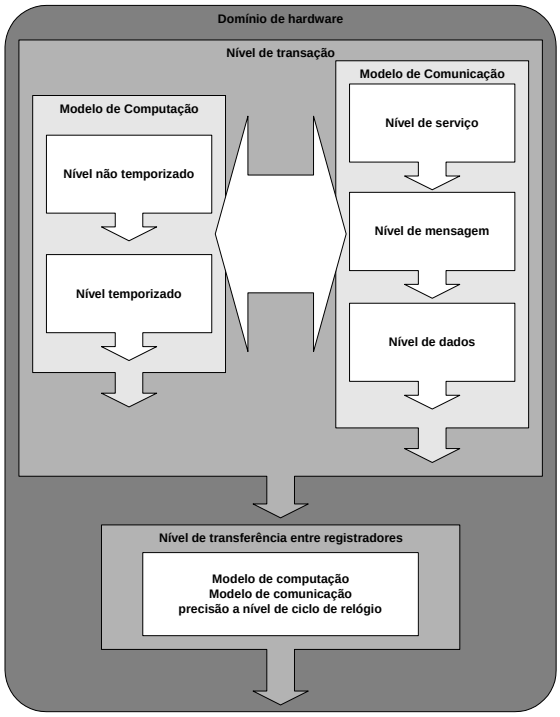


Figura 6 – Níveis de abstração no domínio de hardware. O primeiro nível é o de transação e o nível inferior é o nível de transferência entre registradores.

O *nível RTL* é visto como o último nível de abstração do *domínio de hardware*. Este nível é assumido dentro do fluxo de projeto, pois é nele que as ferramentas de síntese trabalham atualmente, conforme e.g. Jerraya [JER01]. No *nível RTL*, a computação e a comunicação de cada núcleo IP não são vistas separadamente. A computação é representada a partir de lógica combinacional e seqüencial, e a comunicação é realizada a partir de sinais bem definidos, a comunicação entre os diferentes núcleos IP através destes, que implementam o protocolo de comunicação.

A Seção a seguir apresenta o detalhamento da proposta através da comparação desta com outras existentes na literatura.

2.1.5 Comparação de propostas de níveis de abstração para o domínio de hardware

Propostas de níveis de abstração para o domínio de hardware também são adotadas por outros autores. A Tabela 2 resume a comparação de abordagens propostas por Cai [CAI03a], Arnout [ARN02], Haverinen [HAV02] para tais níveis de abstração.

Tabela 2 - Comparação entre propostas de níveis de abstração para o projeto sistêmico.

	Separação entre computação e comunicação	Número de níveis superiores ao RTL	Níveis de abstração diferenciados para comunicação e computação
Lukai Cai	Sim	4	Não
Guido Arnout	Não	2	Não
Haverinen	Sim	3	Sim
Proposta deste trabalho	Sim	5	Sim

Cai [CAI03a] define 3 níveis de abstração para projetos de SoC, quais sejam: não temporizado, temporizado aproximado e temporizado em ciclos de relógio, conforme mostra a . A partir Destes níveis de abstração, os dois primeiros são superiores ao RTL. Apesar de apresentar uma abordagem que separa a comunicação e computação, não há diferença entre os níveis de abstração adotados para as diferentes características. Na abordagem de Cai, a exploração da comunicação fica prejudicada em relação à computação, pois os detalhes de funcionamento da comunicação não se concentram unicamente na precisão temporal. A proposta do presente trabalho apresenta uma abordagem específica para comunicação objetivando o refinamento mais específico para aspectos de comunicação e mais específicos para aspectos de comunicação. Desta forma, a comunicação pode ser mais facilmente refinada, quando comparada à abordagem de Cai.

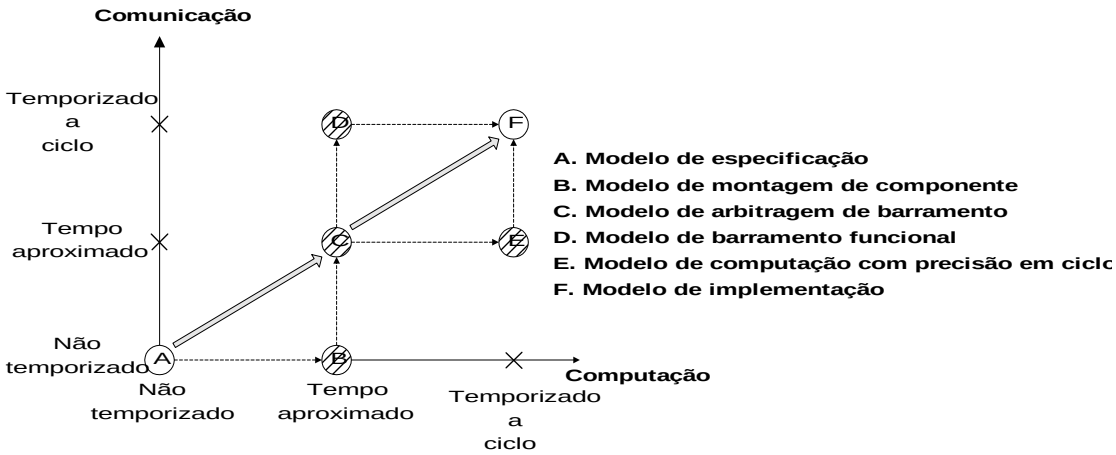


Figura 7 - Grafo de modelagem sistêmica proposta por Cai [CAI03a].

Arnout [ARN02] define 2 níveis superiores ao RTL, quais sejam o nível não temporizado e o nível de transação. Todavia não é apresentada uma separação clara das características de comunicação e computação nestes dois níveis. Desta forma a definição de metodologias de projeto se torna imprecisa, permitindo diversas abstrações para um mesmo grau de detalhamento. No presente trabalho, o nível de transação também é adotado e nele é definido um conjunto de detalhamento a serem assumido para a

modelagem das características de comunicação e da computação de núcleos IP. Desta forma a modelagem permite um fluxo mais preciso quando comparada a abordagem de Arnout.

Haverinen [HAV02] propõem 3 níveis de abstração para comunicação superiores ao RTL, quais sejam, o nível de mensagem, o nível de transação e o nível de transferência. O grau de detalhamento de informações em cada um dos níveis é grande, o que favorece o fluxo de um projeto no que tange a descrição da comunicação dos núcleos IP. Apesar disto, não há uma preocupação explícita com os níveis de abstração para descrição da computação. No presente trabalho, há preocupação tanto com a descrição da comunicação quanto coma a computação. Para tanto, os níveis propostos neste trabalho busca enfatizar as peculiaridades de cada uma das características, conforme melhor detalhado a seguir.

Como a comunicação e para a computação dos núcleos IP possuem características distintas, aqui se trata cada uma delas em separado, neste trabalho definem-se 3 níveis de abstração para comunicação, conforme Figura 8, e 2 níveis de abstração para computação, conforme Figura 9, todos acima do nível RTL. Com esta abordagem, características distintas de cada aspecto podem ser mais bem descritas, facilitando a criação de metodologias de projeto. O principal objetivo dos níveis propostos é favorecer a modelagem de núcleos IPs favorecendo a descrição dos diferentes aspectos de projeto.

Com base nas revisões de propostas realizadas, definiram-se os seguintes níveis de abstração para descrever comunicação nos projetos de SoCs, níveis estes que absorvem as características mais interessantes de cada uma das abordagens já existentes, conforme Figura 8. A proposta apresentada privilegia-se de definições apresentadas por Jeraya e Haverinen, conforme detalhamento a seguir.

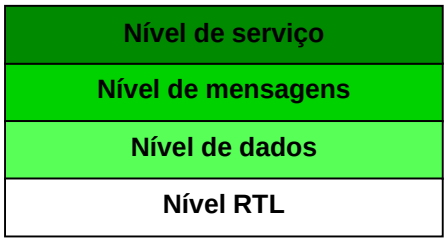


Figura 8 – Níveis de abstração adotados para comunicação.

No *nível de serviço*, o detalhamento da comunicação é abstraído, sendo o principal objetivo descrever a computação desempenhada pelos módulos. Assim, ao invés de detalhar protocolos de comunicação ou criar interpretações dos dados compartilhados, a solicitação de um serviço é implementada. Exemplo de serviço é uma escrita em memória por parte de um processador, onde ao invés de definir um conjunto de sinais a ser ativado, o processador apenas faz uma chamada de um método (e.g. escreveNaMemoria) passando como parâmetro a posição da memória e o dado a ser armazenado. Tal método é definido de forma abstrata em uma interface e implementado pelo canal que interconecta a memória e o processador. A operação de escrita ocorre através da chamada de método de uma interface de comunicação. Neste nível, não há necessidade de interpretação de uma cadeia de bits, apenas a execução de uma requisição de serviço provido por núcleos IP. Este nível é equivalente ao nível de serviço proposto por Jerraya et al., em [JER01] e provê maior abstração do que o nível de mensagem proposto por Haverinen et al. [HAV02]. Questões como temporização e protocolos específicos de comunicação são abstraídos em prol da funcionalidade da comunicação.

No *nível de mensagem*, ainda não há detalhamento de tempo em ciclos de relógio. O objetivo é fornecer um primeiro refinamento da comunicação, que ocorrerá através da interpretação dos dados que são compartilhados entre os diferentes núcleos IP. Os eventos de troca de dados ocorrem de forma

sincronizada. Podem ser descritas interfaces que definem os métodos para envio e recebimento de informações entre módulos, assim como no nível de serviço. As informações repassadas entre os módulos podem representar operações a serem realizadas e os conjuntos de dados relacionados. Por exemplo, no caso de uma comunicação entre o processador e a memória, cria-se uma estrutura (e.g. *struct*) ou mesmo pacotes de dados, contendo o tipo de operação a ser realizada (escrita ou leitura), o endereço referente à posição de memória, e o dado a ser armazenado, caso seja uma operação de escrita. Quando forem usados pacotes de dados para a comunicação, estes devem ser montados no destino para que sejam interpretadas as requisições. Quando forem usadas estruturas de dados para a comunicação, a estrutura contém tipos quem definem que requisição está sendo solicitada. Pacotes de dados permitem a abstração do tamanho do dado, apesar de exigirem a necessidade de remontagem para interpretação, enquanto estruturas de dados já não permitem tanta flexibilidade, porém facilitam a interpretação do dado. Todavia ambas abordagens concentram-se em representar as informações que terão de ser trocadas, como endereço, dado, respostas a requisições e operações a serem realizadas. Esta definição do nível de mensagem apóia-se na proposta de Jerraya et al. [JER01] e de Haverinen et al. [HAV02]. Contudo o foco é permitir uma abordagem que facilite o refinamento para o próximo nível, tendo como alvo a estrutura que será adotada para troca de mensagens. O importante neste nível é definir quais as informações que terão de ser transferidas. Esta definição pode ser realizada tanto através do uso de pacotes de dados quanto através do uso de estruturas de dados bem definidas.

No nível de dados, há precisão de ciclos de relógio, como no nível RTL. Todavia, o que diferencia este do nível RTL é a troca de dados a partir de tipos bem definidos além de mecanismos que descrevam protocolos específicos, como barramentos mestre-escravo, de modo funcional. Assim, podem ser definidos canais de comunicação com a pinagem de endereçamento e operação, por exemplo, são definidos e o que resta é o detalhamento de como ocorre a comunicação internamente ao canal. Esta abordagem é muito próxima da proposta por Haverinen et al. [HAV02] no nível de transferência e por Jerraya et. al. [JER01] no nível de dados.

Com o objetivo de definir um conjunto completo de níveis de abstração a serem adotados para a modelagem de núcleos IP, adicionalmente aos níveis de comunicação propõem-se níveis de abstração superiores ao RTL para descrever computação, conforme Figura 9. Assim como os níveis de abstração assumidos para a comunicação, os níveis assumidos para a computação iniciam logo após o particionamento do modelo de especificação executável.

Figura 9 – Níveis de abstração propostos para representar computação.

O nível de abstração mais alto de um módulo de computação é denominado *nível não temporizado*. Nele descreve-se a funcionalidade dos núcleos IP sem o uso de um sinal de sincronismo nem a idéia de inserção de atrasos temporais. Apesar da despreocupação com o sincronismo interno da atividade dos núcleos IP, a execução dos eventos internos ocorre de forma ordenada total ou parcialmente. Esta organização de execução interna dos núcleos IP, sem precisão temporal, permite garantir a validade do modelo quanto a sua funcionalidade. Adicionalmente esta estrutura contribui para a avaliação de qual o melhor algoritmo para modelar o funcionamento do núcleo IP.

No *nível temporizado*, há a preocupação com o sincronismo das operações realizadas, assim como com o conhecimento do atraso específico de cada tarefa executada. Apesar disto, o nível temporizado se diferencia do nível RTL, devido a abstração da modelagem da comunicação e da computação. Com tal nível de detalhamento, o comportamento do modelo pode ser mapeado e parâmetros de atraso de processamento podem ser analisados. Com esta abordagem é possível validar o desempenho do modelo e definir sua validade ou não de acordo com as especificações do sistema.

O uso de níveis de abstração superiores ao de transferência entre registradores ainda é uma questão em aberto. Durante o desenvolvimento deste trabalho, foram propostas características de modelagem para facilitar o desenvolvimento do trabalho. Dentre as principais vantagens que se busca com a elevação do nível de abstração estão a facilidade de modelagem do sistema, seu gerenciamento, e principalmente a possibilidade de validação do sistema descrito logo nos estágios iniciais do projeto. Com esta abordagem, os domínios de *hardware* e *software* podem ser explorados em paralelo e validados de forma homogênea, diferentemente do que ocorrem em abordagens *ad-hoc* tradicionalmente empregadas hoje.

A definição do nível de transação é uma das contribuições do presente trabalho e tem por finalidade prover mecanismos de refinamento, manuais ou automatizados, que diminuam o tempo de projeto de um sistema completo. Resumidamente, a modelagem no nível de transação permite abstrair detalhes de implementação, usando chamadas de funções que implementam as operações desejadas entre os modelos computacionais. Esta abordagem permite ainda a descrição funcional da operação a ser realizada, o que traz vantagens como o ganho em velocidade de simulação para a validação de um sistema. Como sinais de sincronismo específicos, tais como sinais de relógio, são abstraídos no nível de transação, detalhes como comunicação bloqueante ou não bloqueante podem ser mais facilmente descritos, o que facilita a descrição funcional de barramentos e outros esquemas de comunicação.

2.2 Linguagens de descrição de sistemas

Assumindo-se a necessidade de elevação do nível de abstração de captura de projetos, o uso de HDLs, tais como VHDL e Verilog podem dificultar o processo de modelagem, se estas não suportarem mecanismos que possibilitem a abstração na modelagem de projetos, tanto para comunicação quanto para computação. Uma solução para resolver tal problema é a utilização de linguagens que façam uso de orientação a objetos, tal como ocorre com várias SLDLs.

Devido à exigência da indústria de semicondutores por maior produtividade no projeto de SoCs, é interessante que algumas características sejam providas por linguagens de descrição de projetos em nível de sistema. Uma destas é a facilidade para projetistas aprenderem a usar a linguagem. Subsídios para a descrição do projeto e a velocidade com que o resultado obtido vai ser apresentado são vistos como outras características a serem providas.

Independente do método de projeto de SoC adotado, ou seja, projeto em nível de sistema, projeto baseado em plataforma ou projeto baseado em componentes, o uso de uma SLDL deve apresentar duas características importantes, quais sejam:

- Suportar diversos os níveis de abstração, desde o puramente funcional não temporizado até o de transferência entre registradores;

- Ser executável, de forma que a funcionalidade do projeto e as restrições de temporização possam ser validadas, usando ferramentas que aceitem a SLDL escolhida.

O restante desta Seção apresenta algumas SLDLs. Não serão detalhadas a semântica ou particularidades das linguagens, mas apenas suas origens e propósitos.

2.2.1 OOVHDL

OOVHDL é um grupo de estudo preocupado como aumento da complexidade de projetos de *hardware* e a necessidade do aumento de produtividade na indústria. O principal objetivo deste grupo é adicionar orientação a objetos à linguagem VHDL, para facilitar a descrição e o gerenciamento de projetos complexos. A orientação a objetos e a modelagem genérica oferecem mecanismos para abstração e encapsulamento de descrições de projeto e *testbenches*, provendo maior facilidade no reuso de projetos. Duas propostas de linguagens foram definidas pelo grupo *OOVHDL*, ambas baseadas em VHDL, quais sejam: *Objective VHDL e SUAVE*. Estas foram propostas iniciais, visando suprir um caminho incremental de uma linguagem majoritariamente RTL para uma linguagem SLDL.

Objective VHDL tem como objetivos habilitar a modelagem em alto nível de abstração, facilitar a adição de novas funcionalidades e garantir o reuso de projeto. Tendo por base VHDL, as características que esta proposta busca agregar a VHDL são a inclusão do conceito de classes, herança, polimorfismo, troca de mensagem e chamada de métodos [RAD98].

SUAVE, assim como *Objective VHDL*, tem como base a linguagem VHDL. Conforme [ASH99], para o projeto da linguagem *SUAVE*, foram definidas as seguintes características, que deveriam ser alcançadas para permitir modelagem em alto nível de abstração, quais sejam:

- Prover suporte à modelagem comportamental em alto nível, através do uso de encapsulamento, capacidade de ocultamento de informações, e permitindo o uso de hierarquias de abstração;

- Prover suporte ao reuso incremental de desenvolvimento, através da inclusão de tipos genéricos e do polimorfismo dinâmico;

- Prover formas mais abstratas de comunicação. A comunicação VHDL é baseada na declaração de sinais, sua associação entre os diferentes núcleos IP e a definição de protocolos explícitos;

- Prover a criação e destruição dinâmica de processos;

- Preservar a capacidade de síntese e de formas de análise de projeto, ou seja, apenas adicionar características à linguagem VHDL, sem remover as características atuais;

- Prover abstrações que não sejam tendenciosas para implementações de hardware ou software, permitindo o particionamento e refinamento de sistemas complexos;

- Suportar o projeto concomitante de hardware e software, através da integração com linguagens de programação;

- Suportar o refinamento de modelos, através da elaboração de componentes ao invés de reparticionamento.

Apesar da organização do grupo de estudo OOVHDL, não foi possível encontrar registros de avanço de pesquisa e/ou desenvolvimento das linguagens propostas, tendo o último encontro do grupo ocorrido em maio de 2000, a julgar pela disponibilidade de informações em conferências e na internet.

2.2.2 SystemVerilog

Uma abordagem semelhante à adotada pelo grupo de trabalho OOVHDL é SystemVerilog da Accellera. SystemVerilog foi desenvolvida de dar seqüência a evolução das linguagens de descrição de hardware agregando características novas de desenvolvimento e verificação para os projetistas [FIT04]. Esta linguagem é fortemente baseada na linguagem Verilog, mas inclui também algumas características de VHDL tais como a possibilidade de definição de vetores multidimensionais e comando generate no código. O objetivo desta inclusão de características é possibilitar que projetistas que até então trabalhavam com VHDL possam migrar para SystemVerilog sem ter de encarar mudanças muito radicais na semântica de descrição dos projetos.

Além da combinação das duas linguagens de descrição de hardware, SystemVerilog inclui a possibilidade de modelagem e validação de sistemas em alto nível de abstração [ACC05], provendo mecanismos tal como a possibilidade da modelagem abstrata da comunicação. As classes que definem SystemVerilog provêem um modelo orientado a objeto no qual projetistas podem encapsular métodos, tal como funções ou tarefas e dados. SystemVerilog suporta herança simples, métodos virtuais, dados estáticos, construtores e polimorfismo. As classes definidas em SystemVerilog podem ser instanciadas dinamicamente como objetos, atribuídos valores, desalocadas, e acessadas via manipuladores de objetos. Adicionalmente, a linguagem ainda prevê que a integração da linguagem C com SystemVerilog suporte projetos arquiteturais e comportamentais, verificação de sistemas e tipos de dados definidos na linguagem C, permitindo aos projetistas traduções de e para modelos algorítmicos descritos em C.

2.2.3 Sysjava

Sysjava é uma extensão da linguagem Java utilizada para a descrição de *hardware* [MAR01b]. Diferentemente das extensões de VHDL e *SystemVerilog*, *Sysjava* não evoluiu de uma HDL, mas sim de Java, uma linguagem de programação. Algumas das principais características que permitem utilizar Java como linguagem base para descrição de *hardware* são a portabilidade, a grande quantidade de pacotes nativos para a solução de vários problemas como *threads* e *sockets*, facilidade na geração de interfaces gráficas, sintaxe simples, paradigma de programação orientado a objeto, compilador e interpretador gratuitos. Apesar disto, Sysjava foi desenvolvido, pois Java possui algumas limitações para descrever paralelismo, transição de eventos de sinais entre outras características que são inerentes a *hardware*.

2.2.4 SpecC

A linguagem SpecC é uma notação formal para especificação e projetos de sistemas digitais embarcados incluindo *hardware* e *software* [FUJ01]. Construída sobre a linguagem de programação ANSI-C, a linguagem SpecC atende a conceitos essenciais para projetos de sistemas embarcados tais como hierarquia comportamental e estrutural, comunicação, concorrência, sincronização, transição de estados, tratamento de exceções e temporização.

Um programa SpecC pode ser executado após a compilação com um compilador SpecC que primeiramente gera um modelo de programa intermediário C++, o qual é compilado em seguida por um compilador C++ qualquer para execução do código objeto em um computador hospedeiro.

2.2.5 SystemC

É fato comum que a comunidade de projetistas de *hardware* está familiarizada com programação em C/C++ e em menor escala com a linguagem Java [ARN00]. Assim, a curva de aprendizagem para utilizar uma destas linguagens é menos íngreme que a de aprender uma linguagem totalmente nova. A velocidade de simulação e os recursos disponibilizados pela linguagem C/C++ são superiores aos de Java [MOR98]. Java é uma linguagem interpretada enquanto C/C++ cria um arquivo executável, o que possibilita maior eficiência de execução. Quanto a recursos, C/C++ possui mais mecanismos se comparado a Java, tal como o uso de estruturas genéricas (*templates*). Assim, C/C++ ganha por ser uma linguagem mais tradicional se comparada a Java, uma linguagem considerada nova conforme exposto por Grötker [GRÖ02].

SystemC é uma biblioteca de classes escrita em C/C++, que facilita o processo de modelagem, por possuir características como tipos de dados próprios para definição de *hardware* (*sc_lv*), estruturas (*sc_module*) e processos (*sc_method*) que flexibilizam a descrição de paralelismo, natural em *hardware*. Adicionalmente, SystemC permite a descrição em diferentes níveis de abstração.

2.2.6 Linguagem adotada

SystemC foi escolhido como a linguagem a ser adotada por dois motivos. O primeiro é que SystemC atende as necessidades previamente expostas, tal como menor curva de aprendizagem, maior velocidade de simulação e maior confiabilidade do projeto descrito. Menor curva de aprendizagem está associada ao fato de que durante a formação acadêmica normalmente tem-se maior contato com a linguagem C/C++ do que com HDLs. Quanto a maior velocidade de simulação, há a possibilidade de utilização de níveis de abstração superiores ao RTL. Maior confiabilidade do projeto descrito está relacionada ao fato de que a linguagem permite que modelos gerados em alto nível de abstração sejam usados como estrutura passível de refinamento e validação para as descrições subseqüentes de maior grau de detalhe. O segundo motivo da adoção de SystemC é estratégico, pois SystemC é visto como a mais forte candidata para se tornar padrão para a descrição de *hardware*.

Características adicionais que favorecem o uso de SystemC dá-se ao fato de que ela está vinculada a um organização responsável pela evolução e suporte a linguagem, a qual chama-se *Open SystemC Initiative*. Esta organização disponibiliza uma lista de suporte de alcance mundial. Adicionalmente, SystemC foi sido escolhida por grandes empresas como base para o desenvolvimento de ferramentas de modelagem, validação e síntese de *hardware* e sistemas. Exemplos destas são *Synopsys Inc.* e *CoWare Inc.* Adicionalmente, a captura e a validação de sistemas podem ser obtidas mediante uso de ferramental gratuito.

2.3 Ambientes de suporte ao projeto de sistêmico

Apesar de permitir eficiência em projetos de SoC a partir de seu uso, SLDLs atuais não possuem suporte para a síntese automática de sistemas em altos níveis de abstração e necessitam de intervancao humana para sua interpretação e para gerar código passível de síntese automática. A seguir, são apresentadas

algumas ferramentas atualmente disponíveis para projetos de SoC que viabilizam a modelagem em nível de sistema.

2.3.1 Forge

Forge é um sistema para síntese de alto nível desenvolvido e comercializado pela *Xilinx, Inc*. *Forge* tem como entrada uma descrição comportamental em Java e produz como saída Verilog RTL sintetizável ou ainda um *bitstream* para configurar o FPGA [DON04]. Além da síntese em alto nível, *Forge* permite que a descrição seja direcionada para uma tecnologia alvo do fabricante, tal como FPGAs Virtex e Virtex II.

2.3.2 CoCentric SystemC Compiler e CoCentric System Studio

Ambas as ferramentas *CoCentric SystemC Compiler* e *CoCentric System Studio* são desenvolvidas e comercializadas pela *Synopsys, Inc*. O *CoCentric SystemC Compiler* é um sistema de síntese de alto nível que limitadamente converte descrições comportamentais ou RTL escritas em SystemC para HDL sintetizável ou para EDIF. Algumas restrições são adotadas para que seja possível tal conversão, como por exemplo, a definição detalhada de portas, o uso de métodos do tipo *sc_method* proprietários de SystemC e tipos restritos de dados [SYN02b, SYN02c].

O *CoCentric System Studio* é um ambiente de especificação, modelagem e simulação de projetos que utilizam SystemC como linguagem de descrição de *hardware*. Esta ferramenta possibilita a verificação e a análise de modelos algorítmicos, arquiteturais, *hardware* e *software* em diferentes níveis de abstração [SYN03a, SYN03b]. Adicionalmente, ela suporta a descrição de projetos em diferentes níveis de abstração.

2.3.3 Spark

SPARK é um sistema para síntese de alto nível [GUP03] desenvolvido pelo grupo CECS[1]. *SPARK* tem como entrada uma descrição comportamental em ANSI-C e produz VHDL RTL sintetizável. Para tanto, o sistema usa uma tecnologia de compilador desenvolvido para detectar paralelismo ao nível de instrução. Este mesmo compilador foi adaptado para a síntese em alto nível através da incorporação dos conceitos de exclusão mútua de operações, compartilhamento de recursos e modelos de custo de *hardware*. O sistema *SPARK* provê um mecanismo de transformação de código e suporte a compiladores de transformação, o que permite ao projetista aplicar heurísticas para selecionar e controlar transformações individuais sob modelos de custo realístico para síntese de alto nível. SPARK inclui um sistema completo de síntese de alto nível que provê um caminho a partir da entrada de uma descrição comportamental ANSI-C com restrições ao uso de ponteiros, funções recursivas e o comando *goto*.

[1] CECS - Center of Embedded Computer Systems, University of California, Irvine - http://www.ics.uci.edu/~cecs/

2.3.4 Rose

O sistema de síntese *Rose* trabalha com a síntese da comunicação [DZI03]. Este sistema foi desenvolvido para complementar a funcionalidade alcançada pelo VCC da *Cadence, Inc.*, cuja finalidade é proporcionar mecanismos para exploração do espaço de soluções de projeto. O fluxo proposto pelo grupo que desenvolveu o sistema *Rose* é o de especificação do sistema, exploração de espaço de projeto e particionamento entre *hardware* e *software* utilizando a ferramenta VCC e posterior integração dos módulos heterogêneos através da síntese da comunicação para a implementação de um SoC, utilizando o *Rose*.

2.3.5 Ferramentas de domínio público para SystemC

Uma solução para modelagem em SystemC de projeto de SoCs, que não utiliza ferramentas comerciais, pode ser obtida a partir do uso de compiladores de domínio público, tal como o gcc/g++ disponível na internet (http://gcc.gnu.org), que permite a validação semântica do projeto descrito. Como resultado, obtém-se uma especificação executável do modelo descrito.

Para validação da funcionalidade do modelo descrito em SystemC, é possível utilizar funções da linguagem C++ (e.g. cout, printf) ou gerar arquivos de formas de onda. SystemC é composto por um núcleo de simulação, que permite analisar o conteúdo de sinais utilizados nos módulos em SystemC, a partir da geração de arquivos que respeitam os formatos padrão de forma de onda, tais como VCD, WIF e ISDB. Depois de compilado e executado o modelo implementado em SystemC, estes arquivos de formato de onda são gerados, sendo armazenado os valores contidos nos sinais (sc_signal) e portas (sc_in, sc_out, sc_inout). Arquivos do tipo VCD podem ser lidos pela ferramenta de domínio público GTKWave, sendo gerada representação gráfica da forma de onda rastreada. Esta ferramenta pode ser executada nas plataformas Windows, Unix e Linux, sendo seu código fonte disponível na internet (http://www.gtk.org/).

2.4 Redes de comunicação *intrachip*

Desde a década de 90, cada vez mais núcleos IP processadores e outros componentes reutilizáveis têm sido integrados em um único CI. Estudos prevêem que futuramente SoCs sejam compostos por centenas de núcleos IP [BEN01, GUE00]. A comunicação entre núcleos IP é tradicionalmente realizada seguindo uma ou duas combinações de abordagens: canais dedicados e barramentos. Canais dedicados não são vistos como uma abordagem eficiente, por dificultar o reuso de projetos. Barramentos superam este problema, porém apresentam limitações tais como a diminuição da freqüência de operação com o aumento da densidade de SoCs, devido ao uso compartilhado do canal de comunicação por um número crescente de IPs [ZEF02] e à escalabilidade limitada. Algumas práticas podem ser adotadas para diminuir estas limitações, tal como a utilização de arquiteturas de barramento hierárquicas. Exemplos destas são *CoreConnect* [IBM02] e *Amba* [ARM05]. Todavia, esta abordagem ainda agrega problemas, tais como a possibilidade de comunicação bloqueante atingindo todo o barramento hierárquico, a diferença entre as freqüências de operação e largura de banda entre cada barramento e principalmente a forma desestruturada de montagem destes barramentos. Todos estes fatores afetam a escalabilidade em sistemas de grande porte.

Uma proposta que supere tais problemas e que venha a atender as futuras necessidades de projetos tem sido sugerida a partir do uso de redes complexas de comunicação *intrachip* (ou NoCs, do inglês *Networks on chip*) como mecanismo de comunicação entre núcleos IP de um projeto de SoC [BEN01, KUM02, WIN01]. Esta proposta deve-se ao fato de NoCs apresentam, quando comparadas a barramentos tradicionais, melhorias em: eficiência energética, confiabilidade, reusabilidade, comunicação não bloqueante e escalabilidade de largura de banda [GUE00, MIC02].

2.4.1 Definições básicas

NoCs são estruturas de interconexão de núcleos IP em SoCs, assim como barramentos, porém mais complexos que estes, visto que fazem uso de conceitos mais elaborados, como são vistos a seguir.

Projetos de NoCs adaptam conceitos originalmente surgidos nas áreas de redes de computadores e sistemas distribuídos. Estes incluem conceitos tais como topologias de rede, roteamento, chaveamento, uso de pacotes e uma pilha de protocolos de comunicação. Para tanto, núcleos IP assumem a característica de elementos computacionais e chaves a característica de elementos de comunicação tais como adaptadores de rede, roteadores, etc.

Uma mensagem consiste de um conjunto coerente de informações a ser transmitido entre uma fonte e um destino. Um pacote é a unidade de transmissão de informação empregada pelo meio de comunicação para transmitir mensagens. Pacotes são normalmente constituídos de um cabeçalho (em inglês *header*), corpo (em inglês *payload*) e cauda (em inglês *trailer*).Em geral, o cabeçalho contém dados úteis para o roteamento, o corpo da mensagem contém dados úteis ao núcleo IP destino e a cauda contém dados que garantem a coerência do pacote que está sendo enviado. Em NoCs a comunicação entre os núcleos IP se dá através da troca de mensagens efetivada com freqüência através do envio de pacotes.

Problemas que devem ser evitados em NoCs, assim como em redes de computadores e sistemas distribuídos, são a possibilidade de ocorrência de situações de *deadlock, livelock* e *starvation*. *Deadlock* é definido como uma dependência cíclica entre elementos de comunicação, onde estes solicitam recursos para transmitir informação, mas estes jamais serão concedidos devido à dependência cíclica citada. Uma situação de *deadlock* é ilustrada na Figura 10, para uma rede com topologia malha bidimensional.

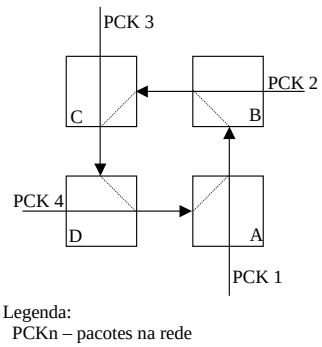


Figura 10 – Situação de deadlock em uma rede malha bidimensional com roteamento *wormhole*. O pacote PCK1 não avança, pois o pacote PCK2 está bloqueando sua passagem. O pacote PCK2 não avança, pois o pacote PCK3 está bloqueando a passagem.

Livelock é definido como uma situação em que uma informação transmitida jamais atinge seu destino, devido ao fato desta percorrer caminhos cíclicos e o ciclo não incluir o destino da informação, conforme

ilustrado na Figura 11. Este problema está normalmente associado à utilização de algoritmos de roteamento não mínimos, definidos mais adiante.

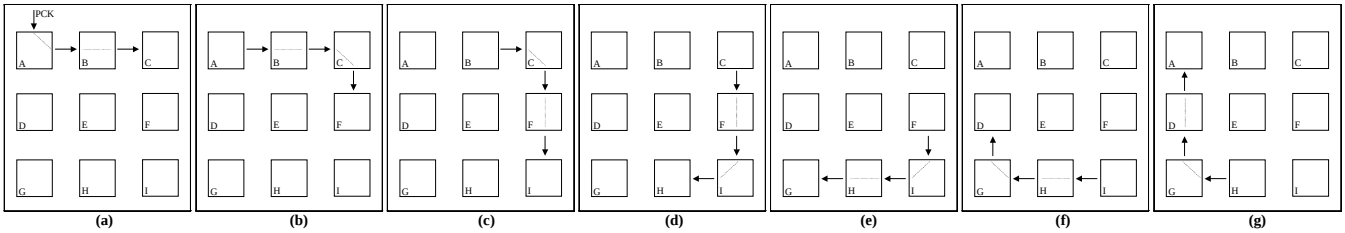


Figura 11 – Exemplo de ocorrência de situação de livelock. Supõe-se um pacote dividido em 4 flits. O pacote PCK entra na rede pela chave A em (a) e tem como destino a chave E. Em (b), (c), (d), (e), (f), (g) o pacote não é roteado para a chave E, executando um caminhamento que não atinge o destino. Ao final, em (g), o ciclo de livelock é fechado.

Starvation é uma condição onde, dada uma estrutura de comunicação, não é possível definir, de forma quantitiva ou qualitativa, um limite superior para os recursos necessários para transmitir uma dada informação de uma fonte para um destino (postergação indefinida). Tal situação pode ser exemplificada quando a todo o momento em que um elemento de comunicação solicita roteamento o seu caminho está ocupado por outro pacote. Este problema está diretamente relacionado a algoritmos de arbitragem.

Uma rede de interconexão é composta por duas partes, quais sejam: serviços e sistemas de comunicação. Serviço é o que a rede deve prover, ou seja, a garantia de entrega de mensagens sem erros. Associado a este serviço fundamental existem parâmetros que qualificam o serviço (em inglês, *quality of service* ou QoS), tais como latência média, taxa de recepção de mensagens, largura de banda, taxa máxima e média de erros de transmissão, etc. O sistema de comunicação é a estrutura para que a rede possa prover o serviço com a QoS necessária. Esta estrutura está associada as escolhas de implementação física adotadas tais como topologia, algoritmo de roteamento e modos de chaveamento.

2.4.2 Topologias

Chaves são os elementos básicos de comunicação da NoC. A forma como estes chaves estão interconectados definem a topologia da NoC, normalmente representável através de grafos. Chaves podem ser consideradas os vértices de um grafo e suas ligações os arcos.

De acordo com a topologia, redes de interconexão podem ser classificadas como estáticas ou dinâmicas [HEN96, HWA92]. Redes estáticas são caracterizadas por conexões ponto a ponto entre as chaves, como ocorre para topologias tais como anel, malha, torus e árvore, representadas na Figura 12. Redes dinâmicas usam conexões multiponto entre as chaves, como ocorre em barramentos e chaves *crossbar*.

Além da topologia, mecanismos que definem a comunicação na rede têm de ser descritos. Tais mecanismos devem garantir questões como a entrega da mensagem, o caminho a ser adotado na rede e a validade da mensagem transferida entre um recurso fonte e seu destino. Segundo Guerrier [GUE00], uma NoC transfere dados de um núcleo IP a outro baseado em troca de mensagens, respeitando um determinado protocolo.

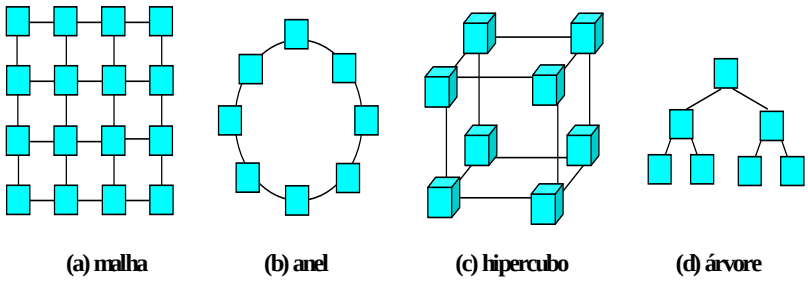


Figura 12 – Algumas topologias estáticas de rede.

Para redes de computadores, protocolos de comunicação são definidos a partir do Modelo de Referência de Interconexão de Sistemas Abertos (em inglês, *Open Systems Interconnect Reference Model*, OSI-RM) proposto pela *International Organization for Standardization* (ISO). O OSI-RM define uma pilha de protocolos, de 7 camadas cujas 4 camadas inferiores são:

- *Camada Física*: Nela é definida a organização de interconexões da rede, ou seja, a distribuição dos canais de comunicação. Esta camada é responsável pelos detalhes referentes à transmissão e recepção de pacotes, sem a preocupação de detecção de erros pelo hardware. Neste nível, são definidos os parâmetros elétricos dos sinais, como nível de tensão, freqüência, forma da onda e ciclo de serviço. Um dos problemas deste nível é a sincronização entre os núcleos, pois os mesmos podem ter domínios de relógio diferentes.

- *Camada de Enlace de Dados*: O principal propósito desta camada é diminuir a falta de confiabilidade na transferência de dados sobre o meio físico. Esta camada define o protocolo para transmitir dados entre um recurso e uma chave e entre duas chaves [TAN96].

- *Camada de Rede*: É responsável por prover conexões e por rotear pacotes na rede de comunicação. Nesta camada são tratados os endereços dos pacotes que chegam e saem das chaves para a definição da direção que o pacote irá percorrer [TAN96].

- *Camada de Transporte*: É responsável por acesso de baixo nível a rede e por transferir mensagens entre os clientes (núcleos IP). Adicionalmente esta camada é responsável por particionar as mensagens em pacotes, gerenciar a ordem dos pacotes, controlar o fluxo e gerar o endereçamento físico [SIL97].

A relação entre as camadas do protocolo OSI-RM com NoCs pode ser definida da seguinte forma:

- *Camada física*: nela são definidos o tamanho do canal de comunicação, os sinais de controle, a topologia;

- *Camada de enlace de dados*: nela são definidas questões de controle do dado que estão sendo transferidos tais como: a forma de armazenamento temporário, controle de fluxo e arbitragem dos pacotes. Nesta camada também podem ser empregados métodos de detecção e controle de erros na transmissão do pacote;

- *Camada de rede*: nela é definido como o pacote fará seu caminhamento na rede. Assim, uma das principais funções é a definição do algoritmo de roteamento baseado no endereçamento definido na rede;

- *Camada de transporte*: nela é implementada a organização de como o pacote deve ser montado na origem ou remontado no destino. Necessariamente esta estrutura está relacionada com a estrutura

do pacote e a semântica de comunicação definida na rede.

2.4.3 Chaveamento

O modo de chaveamento de uma rede de comunicação é definido como a estratégia pela qual o pacote será transmitido de sua fonte até o destino. Os modos de chaveamento podem ser definidos pela reserva de um canal de comunicação entre a origem de um pacote e seu destino ou pelo armazenamento temporário do pacote que permite caminhamento dinâmico na rede. Os principais modos de chaveamento são:

- Chaveamento de circuito (circuit switching): é aquele onde previamente ao envio de mensagens, deve ser estabelecido um caminho da chave fonte até a chave destino denominado conexão e logo após, são enviadas todas as mensagens [TAN96];

- Chaveamento de pacotes (packet switching): é aquele onde a transferência da informação, entre a fonte e o destino, é realizada a partir de uma estrutura padronizada, denominada pacote. Neste modo de chaveamento, o caminho a ser percorrido pelo pacote é definido em tempo de roteamento [TAN96], podendo ser armazenado parcialmente em memória durante o caminho.

Quanto à forma de armazenamento dos pacotes no modo de chaveamento de pacotes, destacam-se três tipos, quais sejam:

- *STORE-AND-FORWARD* (armazenar e passar): A chave não pode repassar o pacote até que o tenha recebido por inteiro;

- *VIRTUAL CUT-THROUGH* (cortar através): A chave pode repassar o pacote assim que a próxima chave der garantias de que pode receber todo o pacote;

- *WORMHOLE*: É uma variante do modo cut-through que permite o uso de estruturas de armazenamento menores que o cut-through visto que não precisa armazenar todo o pacote em uma chave. Nele o pacote é transmitido entre as chaves em unidades chamadas flits (flow control digits), Apenas o flit de cabeçalho tem a informação de roteamento. Os flits restantes seguem o mesmo caminho do header.

2.4.4 Armazenamento temporário

O modo de chaveamento de pacotes pressupõe uma estrutura de armazenamento temporário dos dados na chave. Diferentes estratégias de memorização podem ser adotadas cada qual associada a um controle de fluxo dos pacotes, e isto tem influência marcante no desempenho da rede. Dentre as estratégias de memorização destacam-se o armazenamento na entrada, armazenamento central compartilhado ou na saída.

Armazenamento na entrada consiste em inserir filas na entrada das chaves, sendo esta uma estratégia simples de armazenamento temporário por dois fatores: É definido um espaço fixo de uso da memória e, os dados são lidos na mesma ordem em que são escritos na memória. Apesar de simples, ao uso de armazenamento na entrada está associado um problema denominado bloqueio do *header* do pacote (em inglês, *header of line blocking*, HOL). Supondo duas portas de entrada, A e B, com pacotes estejam

competindo pela mesma porta de saída, por exemplo, a porta C. Somente um dos pacotes será transmitido por vez, enquanto que o outro pacote ficará aguardando na fila de sua porta de origem a liberação da porta C. Supondo que o pacote da porta A está sendo transmitido pela porta C, o pacote da porta B terá de ficar aguardando uma próxima oportunidade de envio à porta C. Se um novo pacote chegar a porta B, e este pacote tiver como destino na porta D, este novo pacote terá de aguardar o envio do primeiro pacote armazenado na fila da porta B. Este novo pacote na armazenado na fila da porta B poderia estar sendo enviado para a porta D, porém também ficará bloquedo na fila. Esta situação descreve o problema de HOL.

Armazenamento temporário central compartilhado faz uso de um *buffer* onde as portas de entrada de uma chave irão depositar os dados a serem transmitidos. Diferente do armazenamento de entrada, o espaço de memória a ser utilizado pelas portas de entrada é distribuído dinamicamente entre os pacotes nele armazenados. Neste caso, o tamanho mínimo do *buffer* deve ser igual ao tamanho do dado que será trocado multiplicado pelo número de portas. Pressupondo uma chave de cinco portas de entrada e o tamanho dos dados é igual a 2 bytes, é necessário um *buffer* de pelo menos 10 bytes. O endereço do *buffer* onde os dados provenientes de uma porta da chave serão armazenados é dinamicamente definido. Armazenamento temporário central compartilhado oferece uma melhor utilização de memória do que aquelas proporcionadas pelas abordagens onde este espaço é prévia e estaticamente alocado às portas de entrada. Apesar disto, o compartilhamento completo do *buffer* pode gerar problemas semelhantes ao HOL. Supor duas portas, A e B, com pacotes destinados à mesma porta de saída C. Se a porta de saída C estiver transferindo o pacote da porta de entrada A e a porta de entrada B continuar recebendo dados, isto poderá levar ao preenchimento do *buffer* centralizado. Esta situação de recebimento de dados pela porta B e conseqüente armazenamento poderá ocasionar o fim do espaço em memória. Conseqüentemente as demais portas de entrada não poderão armazenar seus pacotes, que não serão transmitidos e ficarão bloqueados.

No armazenamento temporário de saída são inseridas filas nas portas de saída, replicadas para cada uma das portas de entrada. Ou seja, cada porta de saída possui N filas, sendo N o número de portas de entrada. O controle de escrita nestas filas pode ser realizado de duas formas. Na primeira, há N portas de escrita na fila operando na mesma velocidade que as portas de entrada. Esta abordagem tem como desvantagem o aumento do tamanho de hardware. Na segunda, há apenas uma porta de escrita na fila operando em uma velocidade N vezes maior que a velocidade de operação da porta de entrada. Esta abordagem tem como desvantagem a queda de desempenho. Adicionalmente, o uso de armazenamento temporário de saída exige a implementação de um controle de fluxo entre a porta de entrada e de saída, aumentando a complexidade desta abordagem.

2.4.5 Arbitragem

O uso de algoritmos de arbitragem proporciona meios para resolver conflitos de requisições simultâneas de serviço, através do gerenciamento de acesso a recursos compartilhados no elemento de comunicação da rede. Ao receber requisições de roteamento simultâneas de diferentes portas de entrada da chave decorrente da chegada de novos pacotes, a arbitragem atribui uma prioridade a cada porta e encaminha pacotes para roteamento conforme esta prioridade.

À arbitragem está relacionado o problema de *starvation* que pode ser amenizado ou mesmo resolvido de acordo com o critério adotado. Dentre os critérios de arbitragem exemplificam-se a prioridade estática, prioridade dinâmica, escalonamento por idade, FCFS (*First Come First Served*), LRS (*Least Recently*

Served) e RR (*Round-Robin*) cujas vantagens e inconvenientes são amplamente discutidos na literatura [SIL97].

2.4.6 Roteamento

Roteamento é o método usado para definir o caminho de pacotes na rede. Dependendo do algoritmo de roteamento implementado, a rede pode apresentar variação no desempenho da transferência do pacote devido principalmente ao maior ou menor caminho percorrido. Algoritmos de roteamento têm influência direta em algumas propriedades da interconexão de rede, quais sejam:

- Conectividade: garantir que qualquer pacote a partir da fonte chegará em seu destino indefinidamente;

- Ausência de *livelock* e *deadlock*: garantir que nenhum pacote ficará bloqueado na rede;

- Adaptatividade: garantir que os pacotes poderão adequar seu percurso quando na presença de problemas tais como congestionamento de rede;

- Tolerância a falhas: garantir a entrega de um pacote mesmo quando alguma parte da rede apresenta defeito (chave, conexão, etc).

Os algoritmos de roteamento podem ser classificados de acordo com diversos critérios. Na literatura existem algumas propostas de taxonomia para algoritmos de roteamento [ASH98, DUA02, NI93]. Alguns critérios e classificações baseadas nestes são:

- O momento de realização do roteamento: *dinâmico* quando realizado em tempo de execução e *estático* quando realizado em tempo de compilação;

- O número de destinos das mensagens: *unicast*, quando os pacotes têm apenas um destino e *multicast* quando os pacotes têm mais de um destino;

- O local onde a decisão de roteamento é tomada: *centralizado* quando um único elemento define o caminho de todos os pacotes, *fonte* quando o caminho do pacote é definido na origem e *distribuído* quando o caminho do pacote é definido pelos elementos presentes ao longo de seu caminho;

- A forma de implementação: *baseado em tabelas*, quando o caminho é definido de acordo com uma tabela armazenada em memória ou *baseado em máquinas de estado*, quando o caminho é definido a partir de um algoritmo implementado em *hardware* ou *software*;

- A adaptatividade: *determinístico*, quando o caminho entre a fonte e o destino é sempre o mesmo ou *adaptativo* quando o caminho entre a fonte e o destino é determinado por fatores da rede, como congestionamento. Roteamentos *adaptativos* podem ser classificados de acordo com a abordagem do algoritmo:

- Quanto à progressividade: *progressivo* se o cabeçalho sempre avança reservando canais ou *regressivo* se o cabeçalho puder retornar liberando os canais previamente reservados;

- Quanto à minimalidade: *mínimo* quando a próxima chave em direção ao destino estiver mais perto que a anterior e *não mínimo* quando o algoritmo de roteamento permitir o afastamento eventual do

pacote do seu destino;

- Quanto ao número de caminhos: *completo* quando todos os caminhos ao destino são possíveis ou *parciais* quando apenas um conjunto limitado de caminhos é viável.

2.5 Estado da arte para proposta de NoCs

NoCs são propostas como um novo paradigma de interconexão de núcleos IP adequado para uso no projeto de SoCs. Diversas pesquisas têm sido desenvolvidas. Nesta Seção, apresenta-se parte do panorama atual de NoCs presente na literatura, baseado em [MOR04], cuja tabelam com um resumo do estado da arte em NoCs é reproduzido na Tabela 3.

Quanto ao modo de chaveamento, uma escolha comum à maioria das NoCs revisadas é a escolha do modo chaveamento por pacotes, informação esta não é explicitada na tabela. A exceção é a NoC aSoC [JIA00], onde a definição do caminho que cada mensagem assume é fixada em tempo de síntese de *hardware*, o que caracteriza um chaveamento por circuito.

A topologia de rede e a estratégia de chaveamento são apresentadas na primeira coluna da Tabela 3. A topologia de rede predominante na literatura é a malha 2D. A razão para a escolha deriva de três vantagens: a facilidade de implementação usando a tecnologia planar de CIs atual, a simplicidade de estratégias de chaveamento XY e a escalabilidade da rede. Outra abordagem é o uso da topologia *toro* 2D, para reduzir o diâmetro da rede [MAR02]. A topologia *toro* 2D dobrado [DAL01] é uma opção para reduzir o custo crescente de fios quando comparado a topologia *toro*. Um problema das topologias malha e *toro* é a latência da rede associada. Duas propostas alternativas de NoCs revisadas são apresentadas para superar este problema. A NoC SPIN [AND03a, AND03b, GUE00] e a chave proposta em [PAN03] empregam a topologia árvore gorda, enquanto a NoC Octagon [KAR01, KAR02] sugere o uso de topologia anel cordal, ambas trabalhando com um diâmetro de rede pequeno, conseqüentemente com uma latência reduzida. Com relação à estratégia de chaveamento, a informação sobre os algoritmos adotados é ainda escassa. Isto indica que mais pesquisa é necessária nesta área. No momento, sabe-se que o algoritmo XY adaptativo é propenso a *deadlock*, todavia há soluções para prover chaveamento XY sem perigo de *deadlock* [GLA94].

O segundo parâmetro quantitativo importante de chaves em NoCs é o tamanho do *flit*. Na Tabela 3 é possível classificar as abordagens em dois grupos, os que focam nas tecnologias futuras de SoCs e os adaptados para as limitações existentes.O primeiro grupo inclui as propostas de Dally [DAL01] e Kumar [KUM02], onde canais de chaveamento possuem barramentos de 300 fios, sem afetar significativamente a área total do SoC. Isto pode ser alcançado e.g. pelo uso de tecnologias futuras de 60nm para construção de CIs de 22mm x 22mm, com uma NoC 10x10 para conectar 100 IPs de 2mm x 2mm [KUM02]. Todavia, isto é inatingível atualmente. O segundo grupo envolve chaves com o tamanho de *flit* variando entre 8 e 64 bits. Este tamanho é similar ao tamanho dos dados com o qual os processadores atuais trabalham. Os trabalhos que provêem um protótipo, Marescaux [MAR02] e [MOR04], tem os menores tamanhos de *flit* quando comparados às demais propostas, quais sejam, 16 e 8 bits respectivamente.

O próximo parâmetro na Tabela 3 é a estratégia de armazenamento das chaves. A maioria das NoCs emprega armazenamento na entrada da chave. Visto que armazenamento na entrada necessita de apenas uma fila na entrada, isto causa pouco consumo de área, justificando a escolha. Todavia, filas de entrada causam o já mencionado problema de bloqueio na cabeça do pacote. Para resolver o problema, filas de saída podem ser usadas, conforme [FOR02], com um custo alto, visto que esta solução aumenta o número de filas de armazenamento na chave. Uma solução intermediária é encontrada via uso de filas virtuais de

saída (em inglês, *Virtual Output Queues*, VOQ) associado com multiplexação temporal de canais, como proposto por [MAR02]. Outro importante parâmetro é o tamanho das filas, importante no desempenho da rede bem como consumo de área da chave. Uma fila grande leva a uma rede com baixa contenção, alta latência de pacotes e chaves grandes. Filas pequenas levam à situação oposta.

O último parâmetro estrutural é o tipo de interface entre o núcleo IP e a chave. O uso de uma interface padronizada para comunicação *intrachip* é uma tendência na indústria e no meio acadêmico. O uso de interfaces padronizadas tem aumentado a possibilidade de reuso, sendo visto como uma característica habilitadora para projetos de SoC futuros. Uma NoC com uma interface entre núcleos IP e chave tal como a proposta por [MAR02], é menos habilitada a agregar núcleos IPs fornecidos por terceiros. As duas principais interfaces padronizadas, VCI e OCP são utilizadas por duas das propostas de NoCs apresentadas na Tabela 3. As NoCs Proteo [SAA02, SAA03, SIG02] e o SPIN [AND03a, AND03b] utilizam interfaces VCI, enquanto a NoC de Sgroi [SGR01] e a Hermes [MOR03a] empregam a interface OCP. Estes padrões hoje evoluem para se integrarem, OCP se tornando o padrão físico para núcleos IP.

A quinta coluna da Tabela 3 coleta resultados sobre o tamanho da chave. É interessante observar que duas abordagens alvo para ASICs [MIC02, RIJ03], ambas com tamanho de *flit* igual a 32 bits, tem dimensões similares, em torno de 0,25mm para tecnologias similares. Adicionalmente, sistemas prototipados [MAR02, MOR03a] produziram resultados de área similares em LUTs de FPGA. Este dado indica a necessidade de estabelecer uma relação entre o tamanho da chave e consumo excessivo de área na parte de comunicação em projetos de SoC. É razoável esperar que a adoção de NoCs por projetistas de SoCs esteja vinculado ao ganho de desempenho na comunicação intrachip. Por outro lado, baixo consumo de área quando comparado com arquiteturas de barramento padrão é outra importante questão. Uma especificação de um projeto de SoC normalmente irá determinar um consumo máximo de área permitido para a comunicação intrachip, assim como o desempenho mínimo esperado. Esta especificação será baseada em características da comunicação entre núcleos IP, no tamanho da chave, no tamanho do *flit* (i.e. largura do canal de comunicação) e na cardinalidade das portas da chave. Estes valores são fundamentais para permitir uma estimativa do consumo de área e do pico de desempenho para comunicação intrachip. A adoção de NoCs está assim vinculada à avaliação quantitativa e à facilidade com a qual os projetistas farão uso da abordagem de NoCs em projetos reais.

O pico de desempenho estimado, apresentado na sexta coluna da Tabela 3, é um parâmetro que necessita análise adicional para prover uma comparação significativa entre diferentes NoCs. Desta forma, esta coluna apresenta diferentes unidade de desempenho para diferentes NoCs. Esta coluna deve ser considerada como meramente ilustrativa para possíveis valores de desempenho. A maioria das estimativas é derivada da produção de três valores: número de portas da chave, tamanho do *flit* e estimativa da freqüência de operação. Nenhuma medida de desempenho de dados foi encontrada em publicações revisadas. O valor associado à NoC proposta por [DAL01] deve ser isto com cautela. A razão é que os dados refletem um limite de tecnologia, que pode ser alcançado pelo envio de múltiplos dados através de um fio em cada ciclo de relógio (e.g. 20 bits a cada cliclo de relógio de 200MHz [DAL01]).

A seguir na Tabela 3 há o parâmetro de suporte à qualidade de serviço (em inglês, *Quality of Service*, QoS). A forma mais comum de QoS em NoCs é através da implementação de chaveamento por circuito. Esta é uma maneira de garantir a vazão e assim a QoS por um dado caminho de comunicação. A desvantagem desta abordagem é que a largura de banda pode ser desperdiçada se o caminho de comunicação não for utilizado a todo o momento durante a avaliação de uma conexão. Adicionalmente, visto que a maioria das abordagens combina o chaveamento por circuito com técnicas de melhor esforço, isto traz como conseqüência o aumento do consumo de área da chave. Este é o caso de propostas de NoCs apresentadas em [DAL01, KAR01, RIJ01].

Tabela 3 – Comparativo do estado da arte em NoCs, extraído de [MOR04]. Posições hachuradas indicam dados não disponíveis na(s) referência(s) consultada(s).

NoC	Topologia / Roteamento	Tamanho do Flit	Armazenamento	Interface IP - NoC	Área da Chave	Desempenho de Pico Estimado	Suporte a QoS	Implementação
***SPIN* - 2000** [AND03a, AND03b, GUE00]	Árvore gorda / Determinístico e adaptativo	32 bits de dado + 4 bits de controle	Fila na entrada + 2 filas compartilhadas na saída	VCI	0.24 mm^2 CMOS 0.13µm	2 Gbits/s por chave		Leiaute ASIC, 4,6 mm^2 CMOS 0,13µm
***aSOC* - 2000** [LIA00]	Malha 2D (escalável) / Determinada pela aplicação	32 bits	Nenhum		50.000 transistores		Chaveamento por circuito (não wormhole)	Leiaute ASIC, CMOS 0,35µm
Dally - 2001 [DAL01]	Toro dobrado 2D/ fonte XY	256 bits de dados + 38 bits de controle	Fila na entrada		0,59 mm^2 CMOS 0,1µm (6,6 % de 1 "tile")	4 Gbits/s por fio	Vazão garantida – (canais virtuais)	Não
***Nostrum* - 2001** [KUM02, MIL04]	Malha 2D (**Pico** escalável) / "hot potato"	128 bits de dados + 10 bits de controle	Filas na entrada e na saída		0,01 mm^2 CMOS 65nm			
Sgroi - 2001 [SGR01]	Malha 2D/ NA	18 bits de dados + 2 bits de controle		OCP				
***Octagon*- 2001** [KAR01, KAR02]	Anel cordal / Distribuído e adaptativo	Dado variável + 3 bits de controle				40 Gbits/s	Chaveamento por circuito	Não
Marescaux - 2002 [MAR02]	Toro 2D (escalável) / Bloqueamento XY, determinístico	16 bits de dados + 3 bits de controle	Fila na entrada	Personalizada	611 slices em FPGAs VirtexII (6.58% da área em 1 XC2V6000)	320 Mbits/s por canal virtual, a 40 MHz	2 canais virtuais (para evitar deadlock)	FPGAs VirtexII / VirtexII Pro
Bartic – 2003 [BAR03]	Arbitrário (links parametrizáveis) / Determinístico, virtual-cut-through	Dado variável + 2 bits de controle por enlace	Fila na saída	Personalizada	552 slices + 5 BRAMs em FPGAs VirtexII Pro (chave com 5 enlaces)	800Mbits/s por canal para flits de 16 bits, a 50 MHz	Controle de taxa de injeção, controle de congestionamento	FPGAs VirtexII Pro
***Æthereal* - 2002** [RIJ01, RIJ03]	Malha 2D / Source	32 bits	Fila na entrada	DTL (Padrão proprietário da Philips)	0,26 mm^2 CMOS 0,12µm	80Gbits/s por chave	Chaveamento por circuito	Leiaute ASIC
***Eclipse* - 2002** [FOR02]	Malha hierárquica 2D esparsa / NA	68 bits	Fila na saída					Não
***Proteo* - 2002** [SAA02a, SAA03, SIG02]	Anel bidirecional / NA	Tamanhos do controle e dados variáveis	Fila na entrada e na saída	VCI				Leiaute ASIC, CMOS 0,18µm
SOCIN - 2002 [ZEF03]	Malha 2D (escalável) / Fonte XY	n bits de dado + 4 bits de controle (parametrizável)	Fila na entrada (parametrizável)	VCI	420 LCs em FPGAs APEX (Estimado, n=8, sem "buffers")	1 Gbits/s por chave, a 25 MHz	Não	Não
***SoCBus* - 2002** [WIK03]	Malha 2D / XY adaptativo	16 bits de dados + 3 bits de controle	Posição única na entrada e armanzenamento na saída	Personalizada		2.4 Gbits/s por enlace	Chaveamento por circuito	Não
QNOC - 2003 [BOL04]	Malha 2D regular ou irregular / XY	16 bits de dados + 10 bits de controle (parametrizável)	Fila na entrada (parametrizável) + Fila na saída (posição única)	Personalizada	0,02 mm^2 CMOS 90nm (Estimado)	80 Gbits/s por chave para flits de 16 bits, a 1GHz	Vazão garantida (canais virtuais, 4 tráfegos diferentes)	Não
T-SoC – 2003 [GRE04, PAN03]	Árvore gorda / Adaptativo	Máximo de 38 bits	Fila na entrada e na saída	Personalizada / OCP	27.000 a 36.000 portas equivalentes		Vazão garantida (4 canais virtuais)	
Xpipes - 2002 [DAL03]	Arbitrário (tempo de projeto) / Fonte estática, "street sign"	32, 64 ou 128 bits	Fila de saída virtual	OCP	0,33 mm^2 CMOS 100nm (Estimado)	64 Gbits/s por chave para flits de 32-bits, a 500MHz	Não	Não
***Hermes* – 2003** [MOR03a]	Malha 2D (escalável) / XY	8 bits de dados + 2 bits de controle (parametrizável)	Fila na entrada (parametrizável)	Nativa / OCP	555 LUTs 278 slices em FPGAs VirtexII	500 Mbits/s por chave, a 25 MHz	Não	FPGAs VirtexII

Canais virtuais são uma forma de atingir QoS sem comprometer a largura de banda, especialmente quando combinado com técnicas de multiplexação por divisão de tempo (em inglês, *time division multiplexing*, TDM). A última técnica, exemplificada em [MAR02] evita que pacotes permaneçam

bloqueados por um longo tempo, desde que *flit*s de diferentes entradas de uma chave sejam transferido de acordo com uma alocação de fatias de tempo pré-definida, associada com cada saída da chave. É esperado que a utilização de SoCs atuais e futuros sejam dominados por aplicações que trabalham com grandes fluxos de dados. Conseqüentemente, suporte a QoS é considerada uma característica fundamental por autores de NoCs.

Finalmente, é possível constatar que os resultados obtidos com implementações de NoCs permanecem ainda muito escassos. Nenhuma das quatro implementações ASIC encontradas na literatura fornecem informações se o projeto corresponde a CIs realmente funcionais. Adicionalmente, três abordagens são apenas projetos esboçados. Por outro lado, duas NoCs foram relatadas como tendo sido prototipadas em FPGA, as propostas por [MAR02] e [MOR04].

2.5.1 Proposta Hermes

Tendo sido apresentado o panorama atual de projetos de NoCs, esta Seção tem por objetivo descrever a proposta de NoC adotada neste trabalho, denominada *Hermes*. As características básicas desta NoC aparecem na última linha da Tabela 3.

2.5.1.1 Chave Hermes

O principal objetivo da chave em uma NoC é proporcionar um mecanismo de transferência de mensagens confiável entre núcleos IP. Para tanto, chaves são formadas por lógica de arbitragem, um algoritmo de roteamento, portas e armazenamento de entrada e/ou saída. As portas conectam-se a núcleos IP e/ou a outras chaves para formar a rede de interconexão.

A chave Hermes contém um bloco de controle centralizado e cinco portas bidirecionais denominadas *NORTH*, *SOUTH*, *EAST*, *WEST*, *LOCAL*. A porta *LOCAL* é dedicada ao núcleo IP local e as demais portas são utilizadas para conexão a outras chaves, de acordo com a topologia adotada. O diagrama de blocos da chave Hermes é mostrado na Figura 13. Na lógica de controle são implementados o algoritmo de arbitragem (centralizado) e o algoritmo de chaveamento de pacotes.

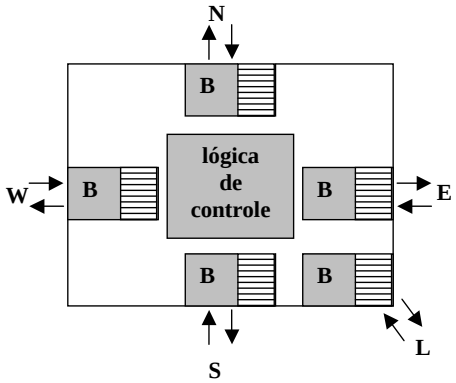


Figura 13 – Estrutura geral da chave da NoC Hermes. A estrutura interna a chave com a letra "B" representa a estrutura de armazenamento da chave.

O modo de chaveamento utilizado é *wormhole*, por proporcionar baixa latência e menor consumo de área, quando comparado aos demais modos de chaveamento por pacote. Associado ao uso de wormhole,

inclui-se filas de armazenamento temporário em cada porta de entrada, com tamanho parametrizável. Esta abordagem contribui para a redução de chaves que ficam bloqueadas para um único pacote que está sendo transmitido pela rede, o que permite maior velocidade de transmissão de dados.

Para ser transmitido pela rede de interconexão, o pacote é dividido em *flits*, que são as menores unidades de transmissão. O tamanho do *flit* é parametrizável, todavia a alteração de tamanho representa necessidades de mudança na forma como os dados devem ser avaliados para roteamento. Atualmente os tamanhos de *flit* com que se trabalha são 8, 16, 32, e 64 bits. A organização do pacote foi definida da como segue, tendo em vista *flits* tamanho igual a 16 bits:

- O primeiro *flit* do pacote armazena os endereços de origem do pacote (8 bits mais significativos) e de destino (8 bits menos significativos);

- O segundo *flit* contém o tamanho do corpo da mensagem, expresso em flits;

- Os demais flits representam o corpo da mensagem.

A lógica de controle é composta por dois módulos, arbitragem é centralizada e roteamento. Quando um novo pacote chega a chave, o cabeçalho da mensagem é armazenado e uma requisição de roteamento é realizada. A arbitragem é definida com prioridade rotativa (*round robin*) para atendimento das requisições de roteamento. Assim, de acordo com a prioridade das portas que estão requisitando roteamento seu pedido é atendido. O algoritmo de roteamento básico é o XY puro. O endereçamento na rede é dado pelas coordenadas XY de cada chave. O cabeçalho do pacote possui endereço XY^{alvo} do destino e a chave onde se encontra o cabeçalho do pacote contém o endereço XY^{local}. Assim, o roteamento com o algoritmo XY puro é realizado da seguinte forma:

- se Xalvo > Xlocal então reserva porta WEST;

- se Xalvo < Xlocal então reserva porta EAST;

- se Xalvo = Xlocal e Yalvo > Ylocal então reserva porta SOUTH;

- se Xalvo = Xlocal e Yalvo < Ylocal então reserva porta NORTH;

- se XYalvo = XYlocal então reserva porta LOCAL.

Ainda referente ao algoritmo de roteamento, a reserva de portas somente é possível se a porta destino estiver livre, ou seja, se a porta pela qual o pacote será enviado não estiver reservada. Por serem portas bidirecionais, estas podem receber e enviar *flits* ao mesmo tempo, conforme mostra a Figura 14.

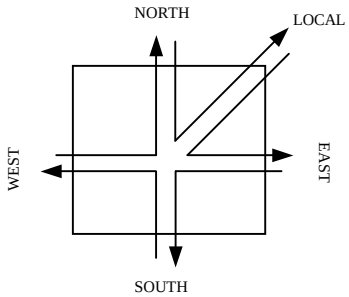


Figura 14 – Exemplo de situação onde a chave Hermes transmite com vazão máxima. Para cada porta de entrada está reservada uma porta de saída.

2.5.1.2 NoC Hermes

A forma como as chaves se interconectam define a topologia da rede de interconexão. Para a proposta da NoC Hermes original foi utilizada a topologia malha, onde cada chave está ligada a um núcleo IP pela porta *LOCAL* e, dependendo da posição da chave na rede, a outras chaves com as demais portas, conforme a Figura 15. Apesar desta topologia ter sido definida, a chave Hermes permite que outras topologias também sejam implementadas tais como *toro* e hipercubo. Para tanto, as chaves têm de ser interconectadas de forma a representar tais topologias e o algoritmo de roteamento tem de ser adequado.

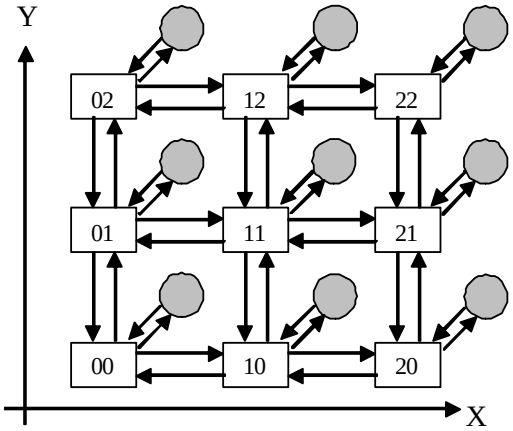


Figura 15 – Representação de uma topologia malha 3x3. Círculos representam núcleos IP ligados às chaves, representadas pelos retângulos. O endereço de cada chave é dado pelo par de coordenadas XY.

3 Modelagem TL e RTL – Um estudo de caso comparativo

O padrão de descrições de entrada suportado hoje pelas ferramentas de síntese automáticas de *hardware* é denominado de nível RTL. Uma interpretação formal do nível de abstração denominado RTL aparece em [BAI01]. Nesta referência, Bailey e Gajski definem o modelo máquina de estados com dados (FSMD) e usam-no como base para estabelecer uma semântica de interpretação precisa para modelos RTL. Ainda naquela referência, também é proposto um método de refinamento de descrições RTL em 5 fases, gerando 5 estilos RTL com níveis de abstração decrescente. O primeiro estilo denomina-se RTL não mapeado, os três seguintes são RTLs mapeados e o último é RTL com exposição de controle. Assim, ao ser analisada mais detalhadamente percebe-se que a própria definição RTL pode incluir um conjunto de níveis de abstração.

Informalmente, RTL consiste na descrição de módulos de *hardware* em termos de um bloco de dados e uma unidade de controle. O bloco de dados é composto estruturalmente usando um conjunto de elementos de memórias (registradores), elementos de processamento (ULA, multiplicadores, etc) e suas interconexões. A unidade de controle é uma descrição comportamental de como o fluxo de dados é comandado. A modelagem RTL utiliza precisão em nível de ciclos de relógio para a descrição de blocos de dados e unidades de controle. Atualmente, a maior parte das ferramentas de síntese aceita modelos RTL como entrada para produção de projetos de alta qualidade.

Apesar de não sintetizáveis, descrições de sistemas partindo de níveis de abstração mais abstratos permitem ter uma visão geral do sistema mais rapidamente. Diminuir a quantidade de detalhes dos passos de implementação e centrar na funcionalidade a ser atingida tende a diminuir a complexidade de descrição, validação e gerenciamento de um sistema. A proposta adotada neste trabalho é a de implementação de projetos iniciando em nível de transação, conforme detalhamento fornecido no Capítulo anterior. Para tanto, a computação é descrita de modo comportamental sem precisão em nível de ciclo de relógio, e a comunicação é descrita de forma abstrata sem detalhamento de protocolos. A partir desta descrição abstrata, um conjunto de passos deve ser adotado até alcançar uma descrição sintetizável. A Seção a seguir apresenta uma proposta de fluxo de projeto partindo de descrições TL.

3.1 Proposta de refinamento partindo de descrições TL

Modelagem TL é considerada importante para análise de desempenho, particionamento de *hardware* e *software*, geração de testes de padrões para vários níveis de abstração, entre outras possibilidades [KUN03, MOU03]. O uso de modelagem TL permite simplificar o esforço de projeto e aumentar da velocidade de validação dos modelos. Conforme discutido no Capítulo anterior, os principais desafios ao se empregar modelagem TL para projetistas RTL são os de abstrair o emprego do sinal de relógio e protocolos de comunicação específicos.

Quando se parte de descrições em mais alto nível de abstração, como as descrições em TL, um conjunto de passos se faz necessário para alcançar uma descrição RTL sintetizável. Atualmente, assim como não há consenso sobre a modelagem TL, o conjunto de passos que leve esta modelagem a uma descrição sintetizável também é ausente. No presente trabalho o ponto inicial da modelagem é TL, sendo a computação descrita de modo funcional sem precisão no ciclo de relógio e a comunicação descrita através de requisição de serviços. Para alcançar uma descrição RTL sintetizável, um conjunto de passos é proposto neste Capítulo.

Para ilustrar o conjunto de passos propostos para o refinamento do modelo TL, um estudo de caso genérico entre um processador e uma memória é apresentado. Este exemplo foi adotado, pois o processador se comunica com a memória apenas para realizar operações de leitura ou escrita. Esta abordagem elimina a representação de refinamento de interfaces mais complexas, que envolvem mais operações, e objetiva a ilustração da estratégia de refinamento.

Através de um conjunto de combinações possíveis de serem alcançadas com as diferentes camadas de abstração definidas para a computação e comunicação, conforme apresentadas no Capítulo anterior, um conjunto de quatro passos de refinamento é aqui proposto. As possíveis combinações citadas e os passos de refinamento são ilustrados na Figura 16.

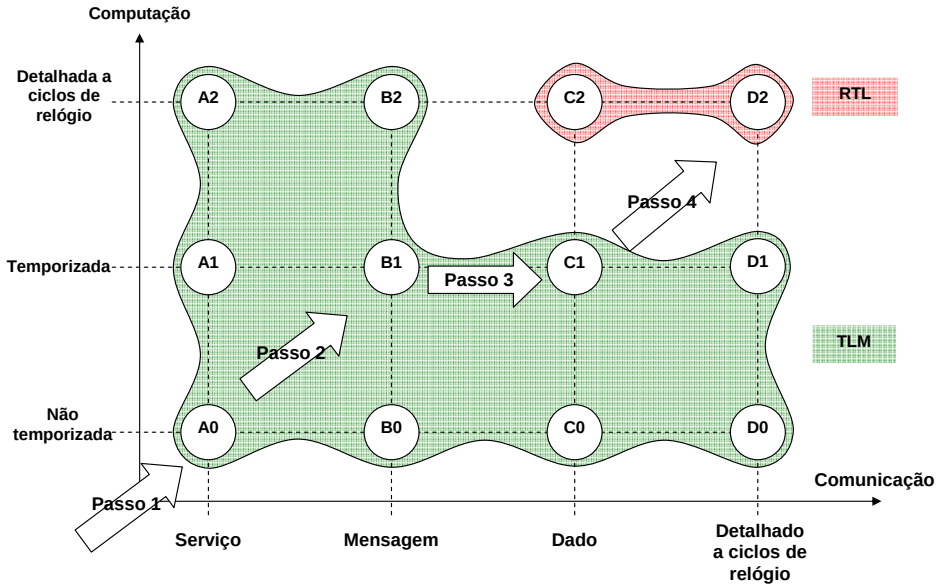


Figura 16 - Combinação de níveis de abstração para computação e comunicação. Ilustração dos possíveis passos de refinamento, iniciando no Passo 1 (descrição abstrata), e finalizando no Passo 4 (descrição sintetizável, ou próxima disto).

O primeiro passo no fluxo de projeto TL, captura de projeto, consiste em descrever o sistema como um conjunto de componentes de alto nível de abstração usando, por exemplo, diagramas de blocos. Componentes são descritos como *módulos* que contêm *processos*, *portas* e *canais abstratos de comunicação*. *Processos* definem o comportamento de um módulo particular e provêem um método para expressar concorrência. *Portas* disponibilizam interfaces de comunicação entre outros módulos, interfaces estas que definem quais serviços estão disponíveis em cada módulo. As portas dos diferentes módulos que se comunicam estão conectadas por canais abstratos de comunicação. Um *canal abstrato de comunicação* implementa uma ou mais interfaces, onde uma interface é simplesmente um conjunto de métodos que definem os serviços disponíveis para os módulos. Canais abstratos de comunicação encapsulam protocolos de comunicação abstratos, tal como protocolos baseados em filas. Para que diferentes módulos possam se comunicar, é necessário que haja um canal abstrato conectando os dois. Este canal deve implementar as interfaces de comunicação de cada módulo. Para concretizar a comunicação entre os módulos, dentro de um processo é necessário chamar um *método*, situação esta ilustrada na Figura 17, onde um processador requisita uma leitura/escrita em uma memória.

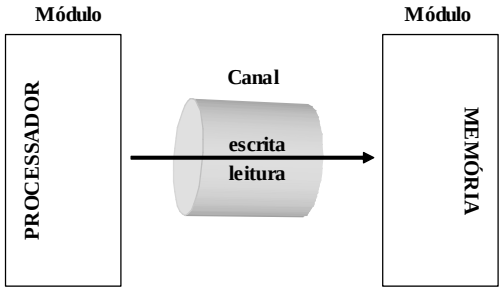


Figura 17 – Primeiro passo de refinamento da comunicação entre 2 módulos TL. Neste passo é definida a interoperabilidade entre os módulos. Processador requisita serviços à memória. A memória responde às requisições do processador. Não há definição de sinais físicos que implementam a comunicação, nem precisão temporal, apenas a definição da funcionalidade do processador e da memória nos módulos respectivos. Ou seja, a comunicação é abstraída e a computação representada de forma algorítmica.

Após garantir o correto funcionamento do projeto descrito, seguem os passos de refinamento do sistema, o passo 2, representado na Figura 16. À parte de computação do módulo é inserida a característica de temporização. São definidas características tal como a pressuposição do tempo de execução de alguns algoritmos descritos na parte computacional. Para garantir uma maior precisão, pode ser inserido o sinal de relógio. Apesar de incluir características de tempo a descrição da computação ainda não é sintetizável. O passo 2 de refinamento, no que se refere a comunicação, mantém o canal abstrato, porém sendo que este apenas implementa métodos de envio e recebimento de dados. Diferentemente do passo 1 de refinamento, onde havia requisições de serviços por parte dos módulos, agora os dados são transferidos entre os módulos e lá são interpretados. O resultado é ilustrado na Figura 18. Nesta situação, por exemplo, ao solicitar uma escrita em memória, o processador envia 3 informações que representam a operação a ser realizada, o endereço e o dado, sendo esta última informação sendo enviada no caso de uma escrita. Outra proposta possível para esta operação é a transferência de tipos estruturados (e.g. struct em C++). Para este último caso, o tipo estruturado deve conter três campos, um com o tipo de operação a ser realizada, um com a posição e um com o dado. Neste refinamento há o detalhamento das informações que devem ser trocados entre os diferentes módulos o que permite um refinamento mais fácil. O resultado do refinamento realizado no passo 2 pode ser apontado na Figura 20 no ponto B1.

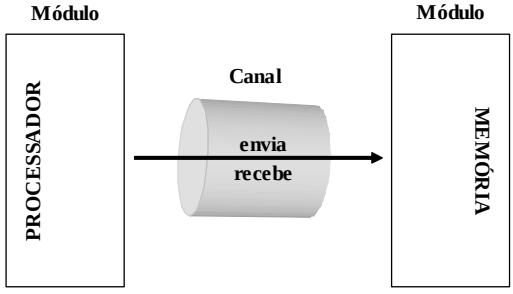


Figura 18 – Segundo passo do refinamento da comunicação.

Sendo novamente validado o projeto, de acordo com um plano de validação previamente elaborado, segue passo 3 de refinamento conforme Figura 20. No passo 3 o protocolo de comunicação entre os módulos e os sinais utilizados para isto são detalhados, sendo eliminado o canal abstrato. Há necessariamente a inclusão de um sinal de sincronismo na descrição dos módulos neste passo de refinamento, conforme ilustrado na Figura 19. A computação dos módulos faz uso deste sinal de sincronismo, sendo esboçada uma primeira versão da máquina de estados de controle da comunicação. Este nível de abstração difere do RTL por ainda permanecer abstraído o tipo de dado que será

compartilhado entre os módulos, o que facilita o uso de alguns passos adicionais de refinamento, caso necessário.

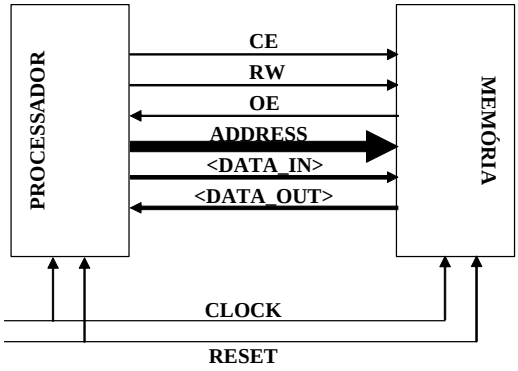


Figura 19 – Terceiro passo de refinamento da comunicação.

O passo final de refinamento, o passo 4, leva a uma descrição RTL sintetizável. Os protocolos de comunicação são definidos e a temporização das ações é definida com precisão em ciclos de relógio. A computação é descrita de forma sintetizável, assim como os tipos de dados que serão compartilhados. A validação do sistema neste nível é realizada por simulação RTL, sendo utilizado, preferencialmente, o mesmo padrão de teste é projetado para o modelo TL. Esta abordagem garante a coerência do projeto desde a fase inicial de modelagem.

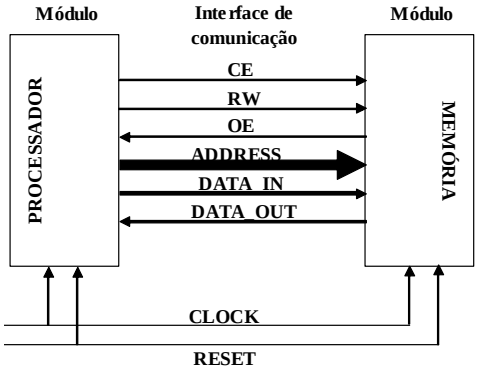


Figura 20 – Passo final do refinamento da comunicação. Descrição RTL com precisão em ciclos de relógio A comunicação entre os módulos ocorre através de sinais interpretados pelos módulos. Nesta Figura, quando o processador necessita comunicar-se com a memória, o sinal CE é posto no valor lógico 1 e o sinal RW em 0 para leitura ou 1 para escrita. O endereço onde ocorrerá a operação é designado pelo sinal ADDRESS. Caso esteja ocorrendo uma requisição de escrita, o dado a ser armazenado está disponível em DATA_IN. Caso esteja ocorrendo uma requisição de leitura, é necessário que o processador aguarde que o sinal OE assuma o valor lógico 1. Após este evento, o dado a ser lido está disponível em DATA_OUT.

Os passos seguintes no fluxo de projeto podem ser executados automaticamente por ferramentas hoje disponíveis e pressupõem: síntese lógica do modelo RTL e síntese física.

O restante deste Capítulo tem por objetivo demonstrar a utilidade do uso da linguagem SystemC para implementar a estratégia de refinamento proposta usando uma seqüência de descrições executáveis do sistema. Adicionalmente, mostra-se que a linguagem possui características que a tornam adequada para lidar com descrições abstratas e concretas de sistemas digitais complexos. Apresenta-se a seguir um estudo de caso de sistemas de *hardware*. A linguagem SystemC é usada aqui para modelagem das

diversas descrições do nível TL ao nível RTL (sintetizável). Para comparações de tamanho de *hardware* ocupado e desempenho em simulação, é utilizado um modelo equivalente, descrito em VHDL. Para a síntese lógica do modelo descrito em SystemC RTL foram utilizadas as ferramentas *CoCentric SystemC Compiler* da Synopsys, para traduzir a descrição SystemC RTL para VHDL, e Leonardo Spectrum, para síntese lógica. Para a síntese física é utilizada a ferramenta ISE da *Xilinx* onde o dispositivo alvo é um FPGA da família VirtexII. Com relação à validação funcional para modelos TL e RTL, é utilizada a ferramenta Synopsys CoCentric System Studio para captura de projeto e simulação. O projeto VHDL RTL é validado utilizando o simulador Modelsim. Este estudo foi inicialmente descrito em [CAL03].

3.2 Estudo de caso

O estudo de caso escolhido é um processador *load-store* de 16 bits denominado R8 [MOR03b]. Este processador apresenta um formato de instruções regular: todas as instruções têm o mesmo tamanho, ocupando apenas uma palavra memória de 16 bits. As instruções contêm o código da operação e a especificação de operandos, caso existam. Há poucos modos de endereçamento. Este processador é quase uma máquina RISC mas lhe faltam características arquiteturais comuns a processadores RISC tal como organização *pipeline*. As principais características organizacionais do processador são:

- Endereços e dados têm tamanho de 16 bits;
- O endereçamento de memória é realizado a palavra;
- O banco de registradores contém 16 registradores de propósito geral;
- Há 4 sinalizadores de estado chamados *negative*, *zero*, *carry*, e *overflow*;
- A execução de uma instrução é realizada no mínimo em 2 e no máximo em 4 ciclos de relógio, ou seja, a média de ciclos de relógio por instrução (em inglês *average clock per instruction*, CPI) para qualquer programa executado é sempre um valor entre 2 and 4.

O processador R8 já foi prototipado em *hardware* e atualmente é utilizado em vários projetos internos ao grupo de pesquisa GAPH. Maiores informações sobre este processador podem ser obtidas em [MOR03a].

3.3 Implementações em SystemC

Nesta Seção será apresentada a descrição do estudo de caso em dois níveis de abstração, usando SystemC 2.0.1 [SYN02c]. O primeiro é um modelo TL e o segundo é um modelo RTL. A descrição TL corresponde ao primeiro passo de refinamento apresentado na Seção 3.1, no qual a computação é descrita de modo funcional e a comunicação é realizada através de requisições de serviço, sem fazer uso de um sinal de sincronismo como relógios, por exemplo. O modelo RTL também é apresentado nesta Seção pode ser obtido pelo refinamento do modelo TL.

3.3.1 Versão em nível TL

Neste nível de abstração, o foco não está na descrição precisa da interface entre a memória e o processador ou no número e natureza dos registradores disponíveis. O foco está na funcionalidade e na comunicação interna e externa de forma abstrata [GRÖ02]. Esta abordagem permite análise do projeto logo nos primeiros estágios, o que facilita encontrar gargalos de arquitetura que venham a comprometer a eficiência do sistema. Para alcançar isto, processos descrevem a funcionalidade arquitetural dos componentes de processamento e canais são responsáveis por descrever a semântica de comunicação entre os componentes. Os canais abstraem a complexidade de protocolos, permitindo a transmissão de tipos de dados desejados, que compreendem desde bits até estruturas complexas [PAS02].

O estudo de caso foi implementado com três módulos e três canais, interconectados segundo o diagrama de blocos mostrado na Figura 21. Os módulos descrevem o comportamento das principais unidades funcionais do processador, da memória e do banco de registradores. Os canais descrevem a comunicação entre o processador e a memória, o processador e o banco de registradores e o processador e os flags.

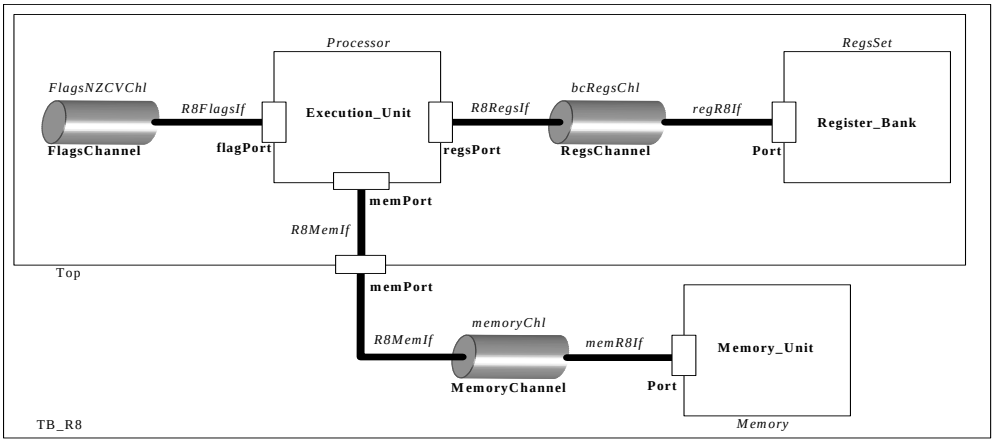


Figura 21 – Estrutura da implementação TL do processador R8. Esta inclui três módulos (Execution_Unit, Register_Bank and Memory_Unit) três canais (FlagsChannel, RegsChannel e MemoryChannel) e cinco interfaces (R8FlagsIf, R8RegsIf, regR8If, R8MemIf e memR8If). Os nomes em negrito são nomes de instâncias, os restantes são nomes de classes na descrição SystemC. O módulo *Top* representa o processador enquanto *TB_R8* representa o *testbench*.

Baseado na especificação do estudo de caso [MOR03b], o processador é modelado para buscar instruções da memória, decodificá-las e executá-las. A comunicação entre o processador e a memória é mapeada no canal *Memory Channel* e sincronizada por eventos. Este canal contém métodos para requisitar instruções à memória e enviar as instruções ao processador. Métodos são definidos pelas interfaces, que são classes contendo objetos virtuais. Interfaces também são usadas para definir as portas do módulo. Por exemplo, quando o processador busca uma nova instrução, ele faz a chamada do método *getNxtWord* através da porta que está conectada ao canal. O método é executado pelo canal, que contém a funcionalidade do registrador PC para endereçar a memória. Assim como o processador, a memória está conectada ao canal por uma porta, onde aguarda uma requisição de leitura. Quando o processador requer a próxima palavra da memória, o canal notifica à memória que o endereço contido em PC terá de ser lido através da chamada do método *getAddress*. A instrução é lida pela memória e enviada de volta para o canal, através da chamada do método *sendWord*. O canal notifica ao processador que a instrução está disponível através de um evento e entra em estado de aguardo de nova requisição. A Figura 22 ilustra a estrutura de métodos usada na comunicação entre o processador e a memória.

```
#include <systemc.h>
#include "R8MemIf.h"
#include "memR8If.h"
class memoryChl: public sc_channel, public R8MemIf,
public memR8If{
public:
typedef sc_lv<16> word_type;
typedef sc_uint<16> ad_type;
void getNxtWord(word_type *word){
      memPosition = PC;
      addressAvailable.notify();
      wait(dataOk);
      *word=localWord;
      PC++; }
void getAddress(ad_type *address){
      wait(addressAvailable);
      *address=memPosition; }
void sendWord(word_type word){
      localWord=word;
      dataOk.notify(); }
private:
  ad_type   PC, memPosition;
  word_type localWord;
  sc_event  addressAvailable, dataOk; };
```

Figura 22 – Descrição parcial TL do processador R8, detalhando a comunicação entre processador e memória.

O nível de abstração de transação permite descrever a funcionalidade do processador, ocultando detalhes dos mecanismos de comunicação. A descrição TL pode ser simulada, e o resultado pode ajudar os projetistas a ajustar, por exemplo, o tamanho da memória, o número de registradores e instruções.

3.3.2 Versão em nível RTL

Seguindo alguns passos de refinamento, tais como os propostos na Seção 3.1, após a validação de um modelo TL, tal como o descrito na Seção 3.3.1, deve-se alcançar uma descrição RTL sintetizável. Tomando-se como base o modelo TL apresentado na Seção 3.3.1, algum detalhamento sobre um possível refinamento é apresentado a seguir.

Como primeiro passo, para a descrição da parte computacional dos módulos, a inclusão do sinal de relógio permite que a implementação do comportamento passe de uma característica algorítmica não temporizada para um comportamento algorítmico temporizado. No entanto, esta descrição permanece não sintetizável, pois as características arquiteturais no módulo não foram ainda implementadas. A comunicação não faz uso do sinal de relógio, todavia há uma substituição da comunicação baseada em serviços para uma baseada em troca de mensagens.

Após serem validados os módulos, o protocolo de comunicação pode ser novamente refinado. Refinamentos e simulações sucessivos permitem que se alcance uma descrição sintetizável. As simulações sobre um sistema que está sendo refinado devem ser cada vez mais demoradas quando comparadas as simulações executada sobre o modelo TL inicial, devido principalmente ao aumento do nível de detalhamento e maior precisão em ciclo de relógio.

No passo seguinte de refinamento, o protocolo de comunicação é detalhado em portas e sinais. Podem ser utilizados tipo de dado *standard logic* ou *boolean* para a implementação destes protocolos na interface destes módulos. O protocolo de comunicação passa da sincronização baseada em eventos para um conjunto de sinais que possivelmente se baseam no sinal de relógio. A exemplo do refinamento adotado no modelo TL do processador apresentado anteriormente, o canal que implementa a comunicação entre o processador e a memória contém o registrador PC. Todavia, na prática este é um registrador interno ao processador. Logo, este registrador é movido do canal para o módulo processador deixando de fazer parte

do canal. O canal de comunicação é substituído por um conjunto de sinais que implementam um protocolo muito similar ao *handshake*.

Finalemente, a computação é descrita de forma a atingir uma característica arquitetural. A descrição de *flip-flops*, tamanho de registradores e máquinas de estados finitos com precisão em nível de ciclo de relógios são definidos. O foco agora muda da modelagem da funcionalidade para a implementação, visto que o que se deseja é uma descrição sintetizável.

Não existem regras finais para conduzir o processo de refinamento. Todavia, nota-se que o uso de uma abordagem hierárquica baseada em refinamentos sucessivos é um método eficiente.

3.4 Comparações

Para comparar as modelagens TL e RTL, foram elaborados alguns experimentos descritos nesta Seção. Como ponto de partida para os experimentos, utilizou-se três implementações equivalentes do processador R8. Duas destas são as descrições TL e RTL realizadas no contexto deste trabalho, descritas na Seção 3.3. A outra é a descrição dita “Gold” do processador R8, implementada em VHDL RTL sintetizável, disponível na página do grupo de pesquisa GAPH. Na Seção 3.4.1 é apresentada uma questão teórica sobre o uso dos níveis de abstração superiores ao RTL a partir de descrições em SystemC e em VHDL, enquanto que nas Seções 3.4.2 e 3.4.3 são apresentados resultado quantitativos da comparação dos diferentes níveis de abstração e linguagens, SystemC e VHDL, tendo como base um programa descrito em *assembly* e executado sobre os as diferentes descrições do processador R8, apresentado anteriormente. Comparou-se o tempo de simulação das versões TL e RTL executando o mesmo programa, com os mesmos dados de entrada. Também se comparou o tamanho do hardware gerado pela síntese automática das versões RTL SystemC e VHDL.

A Seção 3.4.1 apresenta uma comparação qualitativa entre VHDL, uma linguagem tipicamente usada para modelagem RTL, e SystemC, voltada sobretudo, mas não exclusivamente, para níveis de abstração acima de RTL. A seguir, estuda-se a relação de desempenho de simulação de modelos TL e RTL, na Seção 3.4.2. Isto é feito à luz do estudo de caso do processador R8. Finalmente, a Seção 3.4.3 compara os resultados de síntese obtidos a partir das versões RTL nas duas linguagens.

3.4.1 Avaliação qualitativa: VHDL versus SystemC

VHDL é uma linguagem desenvolvida para permitir descrição de *hardware*. Ela contém características específicas para modelar estruturas do nível de abstração RTL, bem como o comportamento de máquinas de estados finitos (em inglês, *Finite State Machine*, FSMs), incluindo estruturas de controle de fluxo e tipos de dados específicos. Realizada a descrição em VHDL, uma ferramenta de simulação, e que obviamente interprete a descrição de *hardware*, pode ser utilizada. Como normalmente *hardware* e *software* interagem em um sistema, co-simulação de *hardware-software* pode ser necessária durante o projeto de um SoC. Com VHDL este processo é possível, mas tem de ser feito através do uso de ferramentas que explicitamente suportam a integração da simulação VHDL com depuradores de *software*, através do sistema operacional de um computador. O simulador Modelsim é um exemplo de ferramenta que tem este suporte. Além deste inconveniente, a descrição de um módulo topo (*testbench*) para validação de um projeto tem de ser realizada usando bibliotecas e comandos específicos de VHDL. Outra opção é novamente recorrer a técnicas de cossimulação para usar uma linguagem com maior capacidade

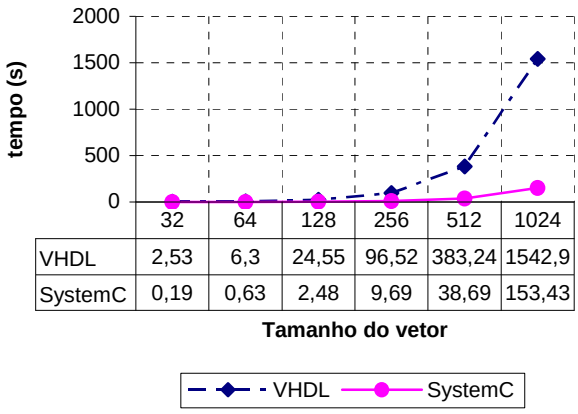
de processar estímulos, tal como C/C++. Para iniciar o trabalho com VHDL, um grande esforço de aprendizagem precisa ser realizado, caso o usuário não conheça a linguagem. Além deste inconveniente, o usuário tem de dominar o conjunto de ferramentas específicas usadas para capturar e validar projetos descritos em VHDL.

SystemC é um conjunto de classes e um núcleo de suporte à simulação que estende a linguagem C++, para permitir modelar *hardware* nesta. SystemC utiliza tipos de dados e controle de fluxo de C++. Adicionalmente, SystemC permite e emprega características de C++ tal como a definição de *templates* que facilitam a declaração e manipulação de variáveis e o uso de herança para definir modelos de *hardware*. O código que descreve o sistema é compilado em um arquivo executável utilizando-se ferramentas tais como GNU gcc\g++ e Microsoft Visual C++. Como um ambiente C++ é utilizado, o esforço para descrever uma simulação envolvendo *software* e *hardware* é muito menor comparado a VHDL, pois a mesma linguagem pode-se descrever tanto *hardware* quanto *software*, sem a necessidade de prover interação entre ferramentas heterogêneas para executar uma cossimulação. Sendo possível utilizar todo o poder de C++ para a simulação, *testbenches* podem ser criados para validar ambos *hardware* e *software*. C++ é uma linguagem muito mais difundida do que VHDL.

3.4.2 Comparação do desempenho de simulação: TL versus RTL

A análise do tempo de simulação foi adotada para comparar a eficiência relativa de modelagens TL na descrição de *hardware*. A descrição VHDL RTL é executada com um simulador VHDL com precisão a ciclos de relógio, enquanto a descrição SystemC é um código compilado gerado pelo compilador GNU g++. A simulação VHDL empregou o simulador Modelsim. Ambas as simulações foram executadas em uma estação Sun Blade 2000, com um processador de 900MHz de freqüência e 1 Gbyte de memória.

Para este processo, o mesmo *testbench* foi utilizado para ambas as descrições, que compreende a execução de uma versão *em código objeto* do algoritmo de ordenação *Bubble Sort*, gerado a partir de fonte em linguagem de montagem, processado pelo programa montador do processador R8. O tamanho dos vetores variou de 32 a 1024 palavras de 16 bits. A Figura 23 apresenta a comparação dos tempos de simulação necessários para executar o algoritmo de ordenação (em segundos).



	32	64	128	256	512	1024
VHDL	2,53	6,3	24,55	96,52	383,24	1542,9
SystemC	0,19	0,63	2,48	9,69	38,69	153,43

Figura 23 – Comparação do tempo de simulação entre as implementações VHDL RTL e SystemC TL. Os valores numéricos do tempo (em segundos) estão disponíveis nas duas linhas inferiores da Figura. Simulação realizada em uma estação Sun Blade 2000, com um processador de 900Mhz de frequência e 1Gbyte de memória principal.

Como mostrado na Figura 23, o tempo de simulação da descrição SystemC é aproximadamente uma ordem de magnitude mais rápido do que a simulação VHDL. A informação é relevante, visto que o tempo necessário para simular grandes circuitos descritos em VHDL (como ocorre em projeto de SoCs) é extremamente alto e constitui o principal fator no caminho crítico de projetos de CIs complexos no tangente a *time-to-market*. Desta forma, a adoção de SystemC pode reduzir significativamente o tempo de validação de projeto, se a validação no nível TL reduzir o esforço de validação no nível RTL ou se a síntese puder proceder diretamente deste ponto para procedimentos de geração de *hardware*.

3.4.3 Comparação do tamanho de *hardware*: SystemC RTL versus VHDL RTL

A última análise realizada foi a comparação do tamanho de *hardware* gerado a partir de descrições SystemC RTL e VHDL RTL fornecidas como entrada de ferramentas automatizadas de síntese. O objetivo principal desta análise é verificar se é possível descrever módulo em SystemC que tenham um tamanho de hardware competitivo com descrições equivalentes em VHDL. Desta forma, a descrição do processador R8 em SystemC RTL baseou-se na descrição VHDL do mesmo processador. Esta abordagem não desqualifica o trabalho desenvolvido, pois como dito anteriormente, o objetivo é garantir que se pode chegar a descrições SystemC com tamanho de hardware comparável ao gerado a partir de VHDL.

Um arquivo EDIF foi obtido a partir da descrição SystemC RTL, utilizando a ferramenta CoCentric SystemC Compiler da Synopsys. O arquivo EDIF e os arquivos fonte VHDL foram importados pela ferramenta Leonardo Spectrum e os resultados da síntese obtidos do mapeamento do projeto para dispositivos Virtex II da Xilinx são mostrados na Figura 24.

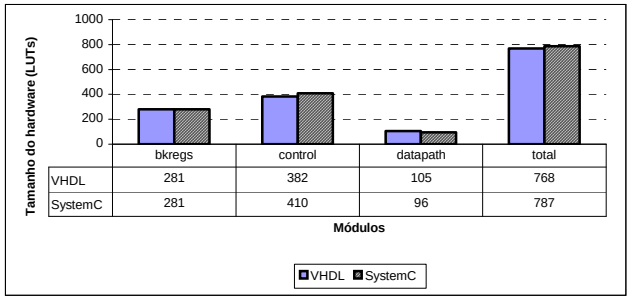


Figura 24 – Comparação do tamanho de hardware entre as implementações VHDL RTL e SystemC RTL. O valor preciso do número de LUTs utilizadas está disponível nas duas linhas inferiores na Figura.

O FPGA destino utilizado foi um VirtexII XC2V1000. A métrica de tamanho utilizada foi a quantidade de LUTs necessária para implementar o *hardware*. Como se pode perceber da observação da Figura 24, o tamanho obtido para cada módulo é similar para VHDL e SystemC. A última barra mostra o tamanho total do *hardware* alcançado por ambos os processos de síntese. A diferença de tamanho entre implementações obtidas a partir de SystemC e VHDL está dentro de 1% do menor resultado.

3.5 Conclusões

Os resultados apresentados na Seção 3.4 fortalecem o ponto de vista da vantagem de descrições de *hardware* utilizando SystemC, por apresentar melhor desempenho de simulação e viabilizar a obtenção de *hardware* competitivo em termos de tamanho com descrições VHDL. O ganho do tempo de simulação chegou a aproximadamente uma ordem de magnitude para o tamanho de *hardware* usado, que representam circuitos de pequeno porte. Como a relação de crescimento é não-linear na complexidade do *software* e no tamanho de *hardware*, ganhos muito maiores podem ser esperados no projeto de SoCs. O tamanho de *hardware* obtido da descrição SystemC RTL é comparável ao obtido a partir da síntese de uma descrição equivalente em VHDL RTL. A descrição completa do processador R8 em VHDL está disponível juntamente com a especificação do processador R8 [MOR03b].

De acordo com os resultados apresentados, SystemC é considerada aqui como uma opção de linguagem interessante para descrição de *hardware* em altos níveis de abstração e descrições sintetizáveis. SystemC é eficiente para simular e validar *hardware* quando comparado com VHDL, e potencialmente permite uma comunidade maior acessar mais facilmente o processo de projeto integrado de *hardware* e *software*.

4 Modelagem e validação da rede Hermes em nível de transação

O Capítulo anterior apresenta uma comparação dos níveis de abstração tradicional (RTL) com o TL, uma tendência emergente na academia e na indústria. Em particular comparam-se alguns aspectos de modelagem nestes níveis usando as linguagens SystemC e VHDL.

Com base nas vantagens apresentadas das comparações apresentadas na Seção anterior sobre o uso de diferentes níveis de abstração para projeto de SoCs propõem-se aqui a modelagem e validação da NoC Hermes a partir de sua descrição TL em SystemC. O objetivo é a validação funcional da NoC em alto nível de abstração. Este objetivo deve permitir a detecção de possíveis erros de projeto antes da implementação [MOR04] no nível RTL. O processo também deve prover um ou mais modelos que possa(m) ser empregado(s) na validação eficiente da comunicação em SoCs contendo a NoC Hermes.

O presente Capítulo descreve a implementação e validação da NoC Hermes em nível de transação nas Seções 4.1 e 4.2.

4.1 Modelagem e descrição da NoC Hermes TL

A presente Seção apresenta o modelo da NoC Hermes implementado em nível TL em linhas gerais.

É interessante relembrar que uma NoC é formada por chaves e por elementos que interconectam estas chaves, conseqüentemente definindo uma topologia. Tais elementos que interconectam a chave são canais, os quais implementam um protocolo de troca de dados como, por exemplo, *handshake*. A chave possui uma estrutura mais complexa quando comparada ao canal que a conecta a outra chave. Ela é composta por N portas, um módulo de arbitragem, um de roteamento e meios de armazenamento temporário que pode ou não estar associados a portas de entrada ou de saída. Na NoC Hermes TL cada chave possui 5 portas de entrada e de saída. O armazenamento temporário está diretamente associado apenas a portas de entrada.

Sendo uma modelagem TL, cabe detalhar que o nível de abstração utilizado para computação é não temporizado e para a comunicação é o de serviços. Esta abordagem permite a visualização de eventuais problemas de modelagem logo nos primeiros passos do projeto. A implementação em alto nível de abstração permite, por exemplo, a validação de algoritmos de roteamento quanto à propensão para apresentar situações de *deadlock*. Obviamente, este problema também é visível em níveis mais baixos de detalhamento, todavia pode não ser tão fácil de detectá-lo e/ou corrigi-lo nestes.

As Seções a seguir apresentam as características gerais da NoC Hermes implementada no nível TL (Seção 4.1.1), seguido de um extenso detalhamento sobre o projeto da chave Hermes e de seus componentes (Seção 4.1.2). Finalmente, apresenta-se os níveis de parametrização do projeto da rede Hermes, bem como a forma de interação entre chaves Hermes e entre uma chave e um IP processador em uma implementação de SoC que use a NoC Hermes como meio de comunicação.

4.1.1 Características da implementação TL

Nesta Seção apresenta-se as características gerais da descrição da NoC Hermes em nível de transação. Estes se referem a itens apresentados no Capítulo 1: pilha de protocolos de comunicação, chaveamento,

memorização e arbitragem. O item roteamento é tratado em Seção a parte durante a discussão da implementação da chave (Seção 4.1.2.5).

Com relação a pilha de protocolos de comunicação, a modelagem arquitetural da chave possui as seguintes características:

- Nível físico: Definição das interfaces físicas da rede externa (ou seja, entre um núcleo IP e uma chave) e interna (ou seja, entre chaves). Na implementação TL, uma transação é suficiente para transmitir exatamente um flit em qualquer interface física, modelando a situação pressuposta da rede Hermes, onde "phits" e flits são idênticos [DAL01];

- Nível de enlace dados: Implementação dos tratamentos realizados sobre os dados. Assim, tarefas como o armazenamento em filas circulares e controles de pacotes são realizados internamente a cada chave. Não foram implementados algoritmos para detecção e correção de erros ou de coerência de dados, apenas reenvio de flits em casos de insucesso no envio de um flit entre chaves ou entre uma chave e um núcleo IP;

- Nível de rede: Foi definida a estrutura do pacote que trafega pela NoC. Esta estrutura é interpretada e um determinado comportamento é adotado em cada instante na chave, visando a comunicação entre chaves, ou chave e núcleo IP;

- Nível de transporte: Com a finalidade de prover a comunicação entre chaves, foram definidos endereços para cada chave. Baseado neste endereçamento, pacotes que trafegam na rede são roteados para que saiam de uma chave fonte e cheguem a uma chave destino. Esta abordagem provê comunicação fim-a-fim.

O modo de chaveamento adotado foi o chaveamento de pacotes. Neste modo, não é reservado um caminho para a transmissão de um pacote antes de ser transferido. O modo de armazenamento adotado para a transmissão do pacote é o *wormhole*, pois consome menos área em hardware, visto que pacotes não precisam ser totalmente armazenados em uma chave e que a área de armazenamento temporário tende a dominar o consumo de área em uma chave [MOR03a].

Ainda associada à questão de armazenamento, a estratégia adotada foi a de *filas* na entrada da chave, por ser mais simples de controlar. As chaves descritas atendem a estas escolhas, pois o critério para recebimento de um pacote na porta de entrada de cada chave é a existência de uma posição livre na fila. Com esta abordagem, o critério para transmissão do pacote não está na garantia do armazenamento total do pacote em cada chave, mas sim na possibilidade de receber cada *flit*. Além disto, um pacote pode estar espalhado por diversas chaves no caminho de transmissão.

As cinco portas de uma chave podem estar requisitando roteamento ao mesmo tempo. Por este motivo foi definido que cada chave deveria possuir controle de acesso ao mecanismo de roteamento. Quanto ao algoritmo de arbitragem, como dito anteriormente, para a NoC Hermes foi adotado o *round robin*. Este algoritmo foi adotado por representar uma forma simples e justa de alternância de prioridade. Nele a prioridade é passada da porta *NORTH* à *SOUTH*, da *SOUTH* à *EAST*, da *EAST* à *WEST*, da *WEST* à *LOCAL*, da *LOCAL* à *NORTH*, conforme requisição de roteamento. Ao iniciar a execução da arbitragem, a porta *LOCAL* ganha prioridade sobre as demais. Caso esta não esteja requisitando roteamento, a próxima porta a ganhar prioridade será a que está requisitando roteamento, segundo a ordem de atribuição de prioridade apresentada anteriormente, ou finalmente a porta *LOCAL*, se nenhuma porta estiver requisitando roteamento. Caso a porta *EAST* esteja requisitando roteamento, por exemplo, a prioridade de acesso ao mecanismo de roteamento é garantida a esta e a definição da próxima porta a ganhar prioridade de acesso somente ocorrerá após a execução do algoritmo de roteamento.

O pacote que transita pelas chaves foi organizado conforme mostra a Figura 25, baseado na proposta original da NoC Hermes [MOR03a]. O primeiro *flit* do pacote contém o endereço de destino na NoC. Neste *flit*, os quatro bits menos significativos representam a coordenada Y e os quatro bits seguintes representam a coordenada X da chave destino. O segundo *flit* representa o número de *flits* com dados válidos disponível no pacote. O tamanho máximo do pacote é $2^{(\text{número de bits para representar um } flit)} + 2$ e o tamanho mínimo é igual a 2. Pacotes com tamanho 2 não transportam *payload* e podem ser utilizados para operações de controle, tais como ativar ou disparar um dispositivo na rede, por exemplo.

posição	0	1	2	...	*size*+1
conteúdo	*header*	*size*	*payload*	*payload*	*payload*

Figura 25 – Estrutura do pacote. O primeiro flit contém o cabeçalho (header) onde está o endereço de destino, o segundo flit contém o tamanho do corpo do pacote (size), e o restante dos flits são o corpo do pacote (payload).

Para permitir a transmissão de pacotes pela NoC Hermes, definiu-se endereços para cada uma das chaves, como apresentados na proposta da NoC Hermes. A topologia adotada inicialmente para o trabalho foi malha, com tamanho parametrizável. Assim, cada chave possui como endereço um valor obtido pela concatenação das posições relativas ao plano cartesiano XY.

4.1.2 Chave Hermes TL

A finalidade da chave Hermes é prover um serviço de transferência de pacotes de uma porta de entrada para uma porta de saída baseado em um critério de roteamento. A modelagem TL da chave expressa esta finalidade sem fazer uso de maiores detalhes de implementação, quando comparada ao modelo RTL. Como dito anteriormente, a interconexão entre as chaves define a topologia de uma NoC. A modelagem TL implementa esta interconexão entre as chaves através de canais, os quais abstraem detalhes de protocolos, tornando a comunicação entre as chaves abstrata.

Na NoC TL modelada, cada chave possui 5 portas. Uma porta é composta por uma entrada e uma saída, sendo que cada chave possui então 5 portas de entrada e 5 de saída. As portas são denominadas *NORTH*, *SOUTH*, *EAST*, *WEST* e *LOCAL*, e são representadas respectivamente pelos números 0 (zero), 1 (um), 2 (dois), 3 (três) e 4 (quatro), conforme ilustra a Figura 26. Como a topologia adotada nesta modelagem TL da NoC Hermes foi a malha, a cada chave podem estar conectadas no máximo 4 (quatro) outras chaves, por qualquer uma das portas com da exceção da porta *LOCAL*. À porta *LOCAL* pode-se conectar um núcleo IP que pode ser uma fonte geradora de pacotes, um consumidor de pacotes ou ambos. Os mecanismos utilizados para a conexão de duas chaves ou entre uma chave e um núcleo IP são idênticos. Ambas utilizam o mesmo tipo de canal de comunicação abstrato.

Cada chave Hermes TL é composta por dois tipos de elemento, quais sejam: portas e lógica de controle, conforme ilustrado na Figura 26. As portas são responsáveis por controlar o recebimento e o envio de pacotes que chegam à chave, sendo compostas por uma entrada, para o recebimento de pacotes, e uma saída, para o envio de pacotes. A lógica de controle implementa, de forma centralizada à chave, a arbitragem e o roteamento de pacotes armazenados em cada porta da chave.

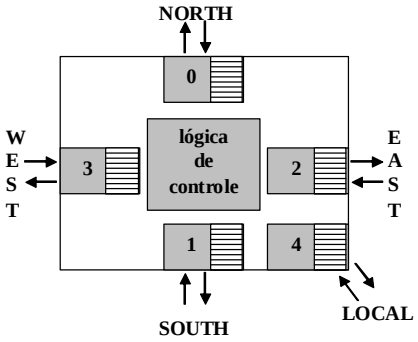


Figura 26 - Estrutura modelada para a chave TL. Os números presentes nas estruturas internas da chave são utilizados para indentificar a porta. Assim, a porta NORTH é identificada por 0 (zero), a SOUTH é indentificada por 1 (um) e assim respectivamente.

A Figura 27 ilustra a modelagem TL da chave Hermes. Nesta Figura é representada a chave, composta por cinco portas e uma lógica de controle, que é um canal de comunicação entre as portas. Cada transferência de pacotes entre uma porta de entrada e uma de saída passa pela lógica de controle. Como esta operação pode estar ocorrendo em paralelo entre vários pares de entrada e saída, cada transferência necessita de uma identificação das portas envolvidas para evitar colisão de pacotes, detalhes estes abordados na Seção 4.1.2.4.

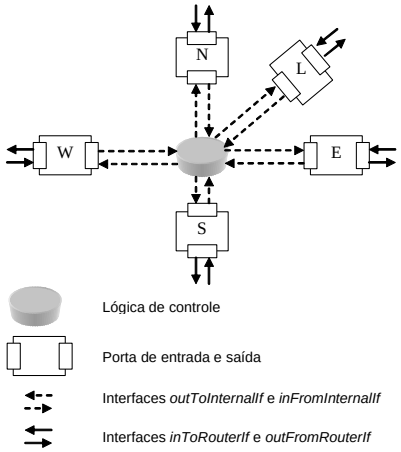


Figura 27 – Estrutura interna da chave Hermes TL. Cinco portas e uma lógica de controle compõem a chave. N é a porta NORTH, S a SOUTH, E a porta EAST, W a porta WEST e L a porta LOCAL.

A chave TL não implementa nenhuma computação diretamente, ou seja, não há nenhuma descrição algorítmica implementada na chave TL. A chave TL é um módulo hierárquico que interconecta portas e lógica de controle. Ao receber um pacote, por uma de suas cinco portas, uma requisição de roteamento é realizada pela porta que armazenou o pacote. Internamente à chave, a porta que recebeu o pacote aguarda ganhar direito para tentar ser ter seu *flit* roteado. Ao receber direito de acesso ao roteamento, a porta que está armazenando o pacote é atendida pela lógica de controle. Caso não seja possível rotear o pacote, este fica aguardando ganhar novamente direito de acesso. Caso tenha sido possível rotear o pacote, este é repassado para a porta de destino que lhe foi atribuída, criando assim uma ligação entre uma porta de entrada e uma porta de saída. Detalhes do comportamento de cada um dos módulos que compõem a chave são apresentados nas Seções a seguir.

4.1.2.1 As portas da chave Hermes

A porta é o módulo utilizado para realizar o controle de entrada e saída de pacotes dentro de uma chave. Neste módulo foram definidas quatro portas abstratas de comunicação, duas para o controle externo à chave (recebimento e envio de pacotes), e duas para controle interno à chave (requisição de roteamento e repasse de *flits* de um pacote para uma porta de saída). O módulo que representa a porta é ilustrado na Figura 28.

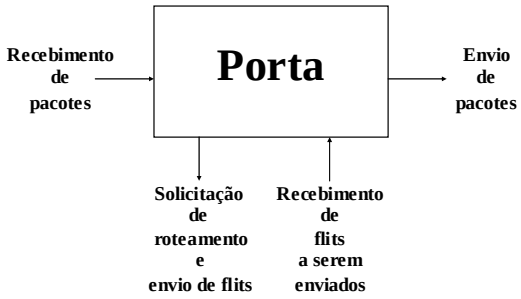


Figura 28 - Representação de uma porta da chave TL.

Assim como a chave, a porta (*door.h* e *door.cpp*) é um módulo hierárquico que instancia outros dois módulos. Estes módulos descrevem o recebimento e a saída de um pacote e sendo detalhados a seguir.

4.1.2.2 A porta de entrada da chave Hermes

Um pacote entra por uma das cinco portas de uma chave, após autorização concedida pelo processo controle de fluxo. Como dito anteriormente, a porta é composta por duas partes, a que trata do recebimento de um pacote e a que controla o envio para a próxima chave. Ao receber o pacote, o módulo *inDoor*, que trata o recebimento de um pacote, armazena em *filas* circulares os *flits* que compõem este, até que a porta de saída seja determinada, ou até esgotar o espaço de armazenamento disponível na porta, o que acontecer primeiro. O processo é ilustrado na Figura 29. Ao ser definida e alocada a porta de saída da chave, inicia-se o envio de *flits* até que todos sejam transmitidos.

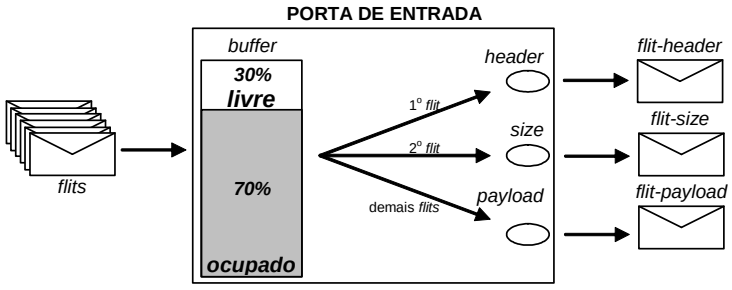


Figura 29 – Funcionamento de uma porta de entrada da chave Hermes TL. À esquerda mostra-se a chegada de flits. Dentro da porta de entrada encontra-se uma fila circular de armazenamento dos flits recebidos (buffer), ilustrada com aproximadamente 70% das posições ocupadas. Internamente à porta de entrada é realizado o controle de repasse do pacote onde o primeiro flit representa o cabeçalho (header), o segundo o tamanho (size), os demais representando a carga útil (payload).

Para a modelagem da porta de entrada, as *filas* circulares têm o número de posições parametrizável e tamanho de palavra equivalente ao tamanho de um *flit*. Cada porta de entrada apresenta comportamento seqüencial para recebimento e para requisição de roteamento e envio de *flits*, ou seja, apenas um flit pode ser armazenado por vez em cada porta de entrada e apenas uma requisição de roteamento e envio de *flits* pode ser realizada por vez na porta de entrada. Todavia, internamente à porta de entrada estas atividades ocorrem em paralelo, ou seja, *flits* podem estar sendo recebidos ao mesmo tempo em que está sendo realizada uma requisição de roteamento ou envio de *flit*. Sendo a chave composta por 5 portas, as operações de controle de recebimento e envio de *flits* nas cinco portas de entradas da chave ocorrem em paralelo.

A implementação do recebimento de pacotes é algorítmica e sem precisão em nível de ciclos de relógio. O comportamento interno do módulo *inDoor* para recebimento de pacotes é ilustrado na Figura 30, sendo que a comunicação com a fonte geradora do pacote ocorre de forma abstrata através da chamada de serviços. Tais serviços são implementados com os métodos SystemC definidos na Tabela 4. Quanto ao comportamento de recebimento de um pacote, inicialmente a chave aguarda que um novo *flit* seja enviado a partir de uma chave fonte. Ao receber este *flit*, o espaço em *buffer* é verificado. Se todas as posições do *buffer* estiverem ocupadas, é informada à chave fonte que o *flit* não pode ser recebido. Caso haja espaço no *buffer*, o *flit* é armazenado em uma posição controlada pelo módulo *inDoor* e a chave fonte é informada que o *flit* foi recebido com sucesso.

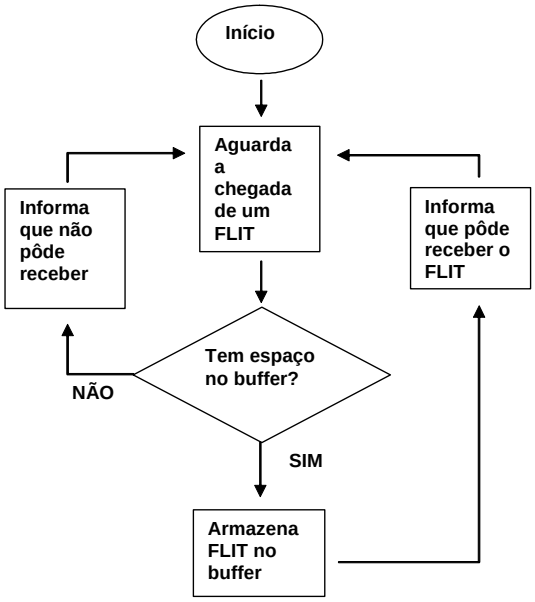


Figura 30 – Fluxograma descrevendo o comportamento de um módulo inDoor para recebimento de flits. Para a chave Hermes este comportamento é replicado cinco vezes, criando independência entre as portas, para que ocorra o paralelismo de recebimento e envio de flits.

A comunicação entre as chaves ou entre uma chave e um núcleo IP se dá através de canais SystemC. Os canais implementam métodos definidos por interfaces além de poder implementar métodos próprios. As interfaces definem os métodos que são utilizados para implementar os serviços a serem disponibilizados pelos módulos. Para a comunicação da porta de entrada através de um canal de comunicação abstrato, foi definida uma interface de comunicação chamada *inToRouterIf*. Nesta interface foram definidos 3 métodos para comunicação durante o recebimento de *flits*, denominados *haveNewFlit*,

couldNotAcceptFlit e *flitAccepted*. O método *haveNewFlit* é chamado e aguarda a notificação de um evento que ocorrerá quando um novo *flit* for enviado pela chave vizinha. Ao receber o *flit*, a porta de entrada verifica a taxa de ocupação do *buffer* circular. Caso o *buffer* esteja totalmente ocupado, o método *couldNotAcceptFlit* é chamado, do contrário o método *flitAccepted* é chamado. Estes dois últimos métodos são chamados para que a chave vizinha, que enviou o *flit*, possa realizar o tratamento da transmissão do *flit*.

Tabela 4 - Métodos de recebimento de um novo *flit* pelo módulo *inDoor*. Métodos definidos pela interface *inToRouterIf*.

Método	Funcionalidade	Parâmetro
void haveNewFlit(FLITTYPE *)	Aguarda recebimento de um *flit*.	• (FLITTYPE*) – Passa variável do tipo do *flit* por referência
void flitAccepted()	Informa que pôde receber o *flit*.	
void couldNotAcceptFlit()	Informa que não pôde receber o *flit*.	

O envio de *flits* para a lógica de controle a partir da porta de entrada tem o comportamento representado na Figura 31.

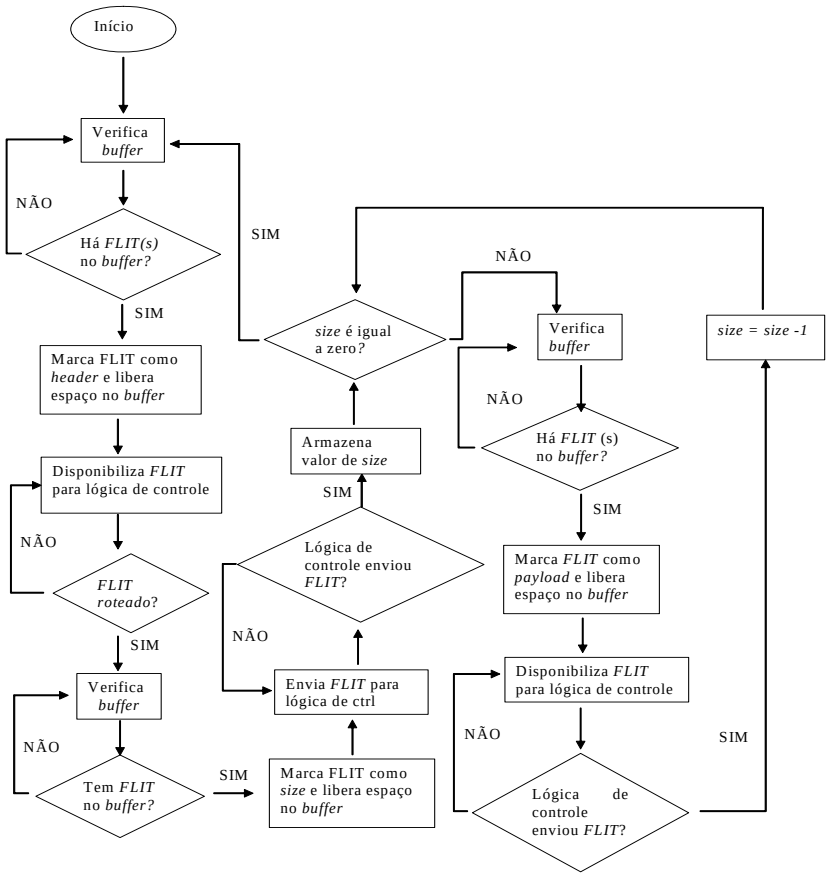


Figura 31 – Fluxograma descrevendo o comportamento do módulo inDoor para o envio de flits à lógica de controle.

Inicialmente, verifica-se a disponibilidade de *flits* na fila de entrada. Havendo *flits* na fila, a porta de entrada marca o primeiro destes como *header* e o envia para a lógica de controle a fim de que seja roteado. Enquanto o roteamento não for realizado a porta de entrada não envia novos *flits* à lógica de controle. Quando o *header flit* for roteado, uma nova verificação do *buffer* é realizada. Caso haja algum *flit*, este é marcado como *size*, armazenado para controle do pacote e enviado para a porta de saída através

da lógica de controle. Quando a porta de entrada for informada que o *flit* pôde ser enviado, o valor de *size* é verificado. Caso o valor seja zero, o pacote foi totalmente enviado e o próximo *flit* representa o *header* de um novo pacote, caso contrário, uma nova verificação de *flit* no *buffer* é realizada, o *flit* capturado é marcado como *payload* e enviado à lógica de controle. Quando o *flit* for enviado, o valor de *size* é decrementado de uma unidade e este é novamente verificado. Se o valor de *size* for zero, o pacote foi totalmente enviado. Caso contrário, o *buffer* será analisado, *flits* serão capturados e o *size* será decrementado até assumir valor zero, para que um novo pacote seja transmitido.

Os métodos utilizados para a comunicação entre a porta de entrada, a lógica de controle e a porta de saída são definidos pela interface *outToInternalIf* e definidos na Tabela 5. Quando o primeiro *flit* é enviado para o roteamento, o método *sendHeader* é chamado, o que define o início da transmissão de um pacote. Este método fica bloqueado até que o *header* seja roteado. Quando o método *sendHeader* é liberado o segundo *flit* é enviado através da chamada do método *sendSize*. O sucesso da transmissão do *size flit* é informado através da variável Booleana passada por referência para o método *sendSize*. Caso o valor desta variável seja *true*, um novo *flit* será capturado no *buffer*, do contrário o *size flit* terá de ser retransmitido. Quando os *payload flits* forem enviados, o método *sendPacketBody* é chamado. O sucesso da transmissão e o tratamento dos *flits* ocorrem como no método *sendSize*. Apesar de todos os métodos terem a mesma funcionalidade, ou seja, a transmissão para a chave destino, o comportamento da lógica de controle é diferenciado para cada um dos tipos de *flit*, o que será mais bem explorado adiante no detalhamento da lógica de controle.

Tabela 5 - Métodos usados para o envio de *flits* à lógica de controle a partir do módulo *inDoor*. Métodos definidos na interface *outToInternalIf*.

Método	Funcionalidade	Parâmetros
void sendHeader(FLITTYPE, int)	Envia o *header* do pacote.	• (FLITTYPE) –*flit* de cabeçalho do pacote • (INT) – A posição da porta no chaveador
void sendPacketSize(FLITTYPE, bool*, int)	Envia o tamanho do pacote.	• (FLITTYPE) – *flit* de tamanho do pacote • (BOOL*) – Passa variável *boolean* como parâmetro: o se *true*: o *flit* foi enviado o se false: o *flit* não foi enviado • (INT) – A posição da porta no chaveador
void sendPacketBody(FLITTYPE, bool*, int)	Envia o corpo do pacote.	• (FLITTYPE) – *flit* corpo do pacote • (BOOL*) – Variável *boolean* para saber se o *flit* foi enviado: o se *true*: o *flit* foi enviado o se false: o *flit* não foi enviado • (INT) – A posição da porta no chaveador

4.1.2.3 A porta de saída da chave Hermes

O módulo *outDoor*, representado na Figura 32, é responsável por enviar *flits* para o núcleo IP, no caso da porta *LOCAL*, ou para uma das chaves vizinhas, no caso das demais portas. Neste módulo, o *flit* recebido é armazenado temporariamente até que a chave vizinha, ou núcleo IP, conectado à porta de saída, informe o sucesso ou insucesso do recebimento do *flit* transmitido. No módulo *outDoor* não há controle do tipo do pacote, ou seja, para a porta de saída é irrelevante se o *flit* que está sendo enviado é do tipo *header*, *size* ou *payload*.

As operações de envio e recebimento de *flits pel*o módulo *outDoor* não ocorrem em paralelo, ou seja, após o recebimento de um *flit*, o próximo somente será recebido depois que a chave ou núcleo IP conectado a esta porta informar se foi possível recebê-lo. Esta abordagem define o comportamento da chave sem armazenamento de saída, conforme a especificação da Seção 2.4.1 para a chave Hermes. O comportamento do módulo é representado na Figura 33. Inicialmente, o módulo aguarda a recepção de

um *flit* para a transmissão. Ao recebê-lo, é realizada a tentativa de transmissão para a chave ou núcleo IP conectado. Caso o *flit* tenha sido enviado, o módulo *outDoor* informa à lógica de controle que conseguiu transmiti-lo, do contrário informa que não foi possível realizar a operação.

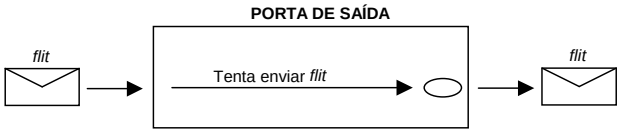


Figura 32 - Funcionamento da porta de saída. À esquerda mostra-se a chegada de flits. À direita ilustra-se o envio dos flits à chave/núcleo IP, flit a flit.

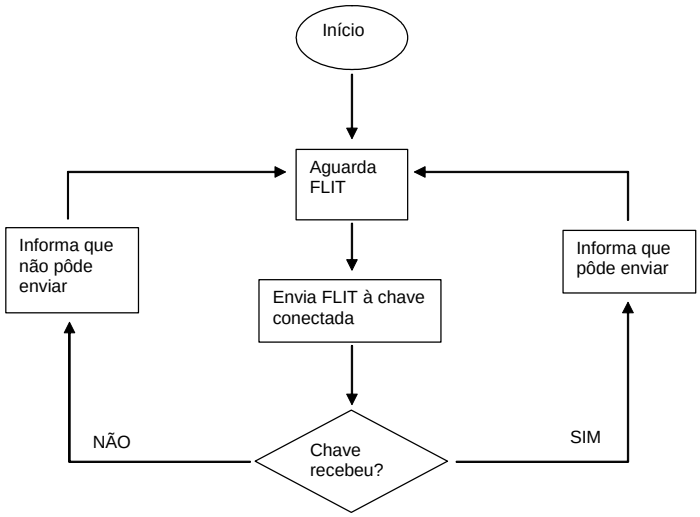


Figura 33 – Fluxograma descrevendo o comportamento do módulo outDoor para recebimento e envio de flits.

Para o recebimento de *flits* vindos da lógica controle, o módulo *outDoor* utiliza os métodos definidos pela interface *inFromInternalIf*, conforme mostra a Tabela 6. Quando método *ogFlit* é chamado, o módulo *outDoor* fica aguardando o recebimento de um *flit* enviado pela porta de entrada. Ao ser recebido este *flit*, procede-se à tentativa de transmissão do *flit*, através do método *try2SendFlit*, definido pela interface *outFromRouterIf*, conforme mostra Tabela 7 e implementado pela lógica de controle. Este método fica bloqueado até que a chave ou núcleo IP receptor do *flit* informa o sucesso, ou não, na operação. A seguir, a informação sobre o sucesso, ou não, da transmissão é repassada à porta de entrada, através da chamada dos métodos *flitSent*, caso tenha sido transmitido o *flit*, ou *couldntSendFlit*, caso não tenha sido transmitido o *flit*.

Tabela 6 - Métodos de requisição de envio de *flits* da porta de entrada à porta de saída.
Métodos definidos pela interface *inFromInternalIf* e implementados pela lógica de controle.

Método	Funcionalidade	Parâmetros
void ogFlit(FLITTYPE*, int)	Aguarda a requisição de envio de *flit* vindos da porta de entrada	• (FLITTYPE *) – *flit* a ser enviado • (INT) – A posição da porta de saída na chave
void flitSent(int)	Informa à lógica de controle que o *flit* foi enviado com sucesso	• (INT) – A posição da porta de saída na chave
void couldntSendFlit(int)	Informa à lógica de controle que o *flit* não pôde ser enviado	• (INT) – A posição da porta de saída na chave

Tabela 7 - Método de envio de *flits* do módulo *outDoor*. Método definido pela interface *outFromRouterIf* e implementado pelo canal de interconexão entre chaves ou entre chave e núcleo IP.

Método	Funcionalidade	Parâmetros
void try2SendFlit(FLITTYPE, bool*)	Tenta repassar *flit* para o núcleo IP (caso a porta de saída seja LOCAL) ou para próxima chave (demais portas)	• (FLITTYPE) – *flit* a ser enviado • (BOOL*) – Resposta do aceite ou não do *flit* enviado

4.1.2.4 A lógica de controle da chave Hermes

A lógica de controle é descrita sob a forma de um canal e é denominada *intraRouterChl*, sendo sua estrutura geral apresentada na Figura 34. O canal *intraRouterChl* implementa os métodos definidos pelas interfaces *outToInternalIf* e *inFromInternalIf*, interconectando as portas da chave além de alguns métodos próprios. Para o presente trabalho, cinco portas foram conectadas ao canal, porém essa quantidade de portas é parametrizável. Além de implementar os métodos definidos nas interfaces, o canal *intraRouterChl* possui outros dois métodos, um para arbitragem e outro para roteamento. Para a arbitragem, foi utilizado o algoritmo de prioridade rotativa (*round robin*) e para o mecanismo de roteamento o algoritmo é parametrizável.

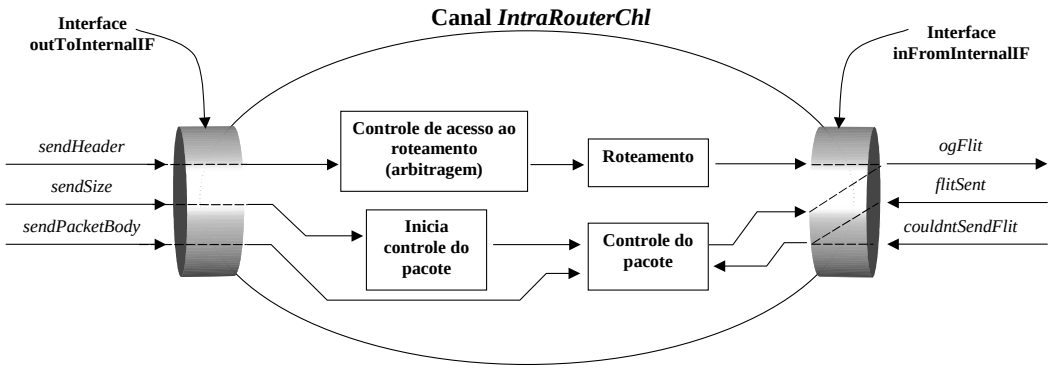


Figura 34 – Estrutura geral da lógica de controle. À esquerda mostra-se a chegada de flits através dos métodos definidos pela interface outToInternalIf. No centro, ilustra-se os métodos envolvidos no processo de roteamento do pacote e os métodos de controle de envio de pacote. À direita, mostra-se o método de transmissão de flit definidos pela interface inFromInternalIf.

O comportamento da lógica de controle é apresentado na Figura 35. Inicialmente, o canal aguarda a chegada de um *flit* do tipo *header*, que representa uma requisição de roteamento. Ao receber este *flit*, a porta de entrada é identificada e o acesso ao mecanismo de roteamento é solicitado. Com a execução do algoritmo de arbitragem dá-se acesso a um dos solicitantes. Quando o acesso é dado a uma das portas que solicitou roteamento, o *flit* que representa o endereço de destino é repassado para o método responsável por realizar o roteamento. O objetivo deste método é definir qual porta de saída será utilizada pelo pacote que está sendo transmitido pela chave. Inicialmente o algoritmo de roteamento utilizado neste trabalho foi o XY Puro. Porém, observou-se ser interessante dispor de outros métodos de roteamento, o que se tornou então uma opção parametrizável. Caso a porta de saída definida no algoritmo de roteamento já esteja em uso, nenhuma saída é atribuída ao pacote. Ao concluir o roteamento, caso não tenha sido atribuída porta de saída ao pacote, a porta de entrada perde a prioridade e tem de aguardar que o algoritmo de arbitragem lhe conceda acesso ao roteamento no futuro.

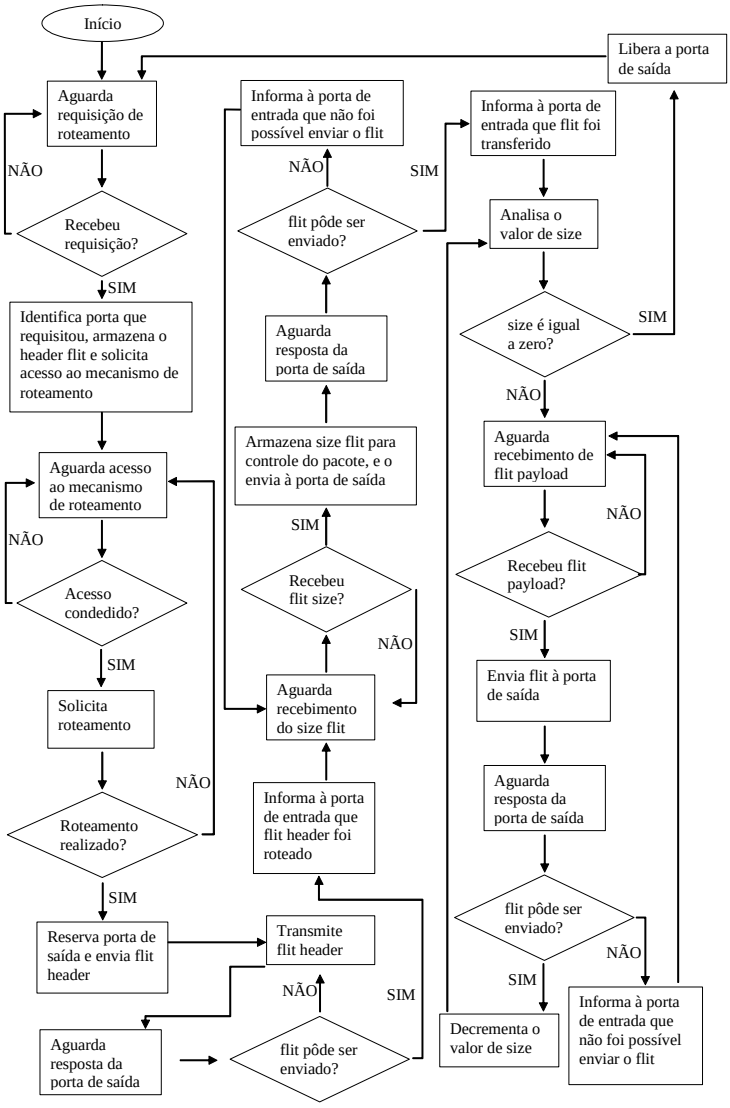


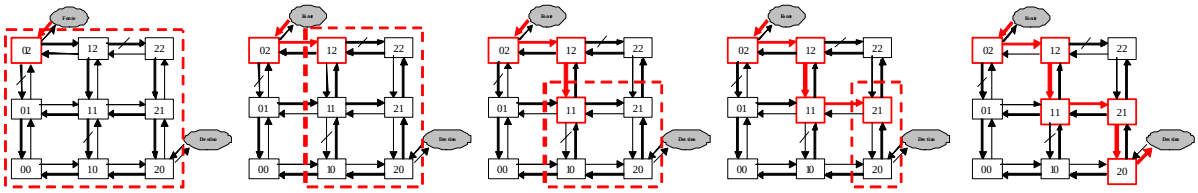
Figura 35 – Fluxograma descrevendo o comportamento da lógica de controle (canal intraRouterChl).

A perda e ganho de acesso ao mecanismo de roteamento ocorrerão até que a porta de saída possa ser alocada. Ao se alocar a porta de saída ao pacote, esta é reservada e o *header flit* é enviado, aguardando-se a resposta do sucesso da transação. Caso a transferência não ocorra, o *header flit* é novamente, enviado, sendo repetida esta operação até que a porta de saída informe que conseguiu repassar o *flit*. Como já apresentado, o controle de envio de *flits* é realizado pela porta de saída. Quando uma porta de entrada é a associada a uma porta de saída, o primeiro *flit* a ser enviado é o *header*, sendo esta requisição de envio solicitada pela lógica de controle. Quando a transferência ocorrer com sucesso, o canal informa à porta de entrada que o *header flit* foi roteado e transferido com sucesso. O canal fica aguardando que a porta de entrada repasse o *size flit*. O valor deste *flit* é armazenado para controle do pacote. Novamente a transferência para a porta de saída é realizada. Contudo, o modelo definido para o canal não se responsabiliza mais pela retransmissão do *flit*, sendo que a lógica de controle apenas repassa a informação sobre o sucesso ou insucesso da transmissão do *flit* à porta de entrada. A seguir, caso o *flit* tenha sido enviado pela porta de saída, o *size flit* é verificado e caso o seu valor seja zero, a porta de saída é liberada e o novo *flit* que se aguarda é do tipo *header*. Caso o *size flit* armazenado não seja igual a zero, aguarda-se o envio de *payload flits*. Ao receber cada um destes, o *flit* é repassado à porta de saída e, assim como o tratamento realizado para o *size flit*, caso não seja possível transmiti-lo, isto é informado à porta

de entrada. Caso o *payload flit* tenha sido transferido, a cópia armazenada do *size flit* é decrementada, o valor é então verificado. Se o valor for igual a zero, a porta de saída é liberada e o novo *flit* que se aguarda é do tipo *header*. Do contrário, a porta de saída permanece reservada e o *flit* que se aguarda é do tipo *payload*. A operação de transferência de *payload flits* repete-se até que o *size flit* armazenado assuma o valor zero.

4.1.2.5 Algoritmos de roteamento

Neste trabalho, foram descritas algumas versões da NoC Hermes TL. Cada versão adiciona uma nova característica ou corrige algum erro de implementação. Na primeira versão da NoC Hermes TL, notou-se a ocorrência de um erro na transmissão de pacotes pela rede. Em alguns momentos a transmissão pela rede congestionava e nenhum pacote era entregue. Foram depurados os elementos que compunham cada chave, bem como a interconexão entre estes. Constatou-se que o problema não estava relacionado a algum erro de implementação, mas sim a um erro de projeto. O algoritmo de roteamento que estava sendo utilizado não era livre de *deadlock*. O algoritmo adotado inicialmente foi o XY adaptativo mínimo, o qual executa um caminhamento em linha e coluna segundo um quadrado envolvente, como mostra a Figura 36. Em virtude deste problema, notou-se interessante poder definir o algoritmo de roteamento como um item parametrizável, ao invés de criar diferentes versões da NoC TL para cada algoritmo de roteamento. Esta abordagem permite flexibilidade ao projeto, além de facilitar explorar e comparar qualitativa e quantitativamente diferentes algoritmos para resolver o mesmo problema.



(a) (b) (c) (d) (e)

Figura 36 – Demonstração do caminhamento de um pacote através de uma NoC 3x3, utilizando roteamento XY adaptativo mínimo. Em (a) observa-se o início do caminhamento ocorrendo pela chave 0x2, sendo definido o maior quadrado envolvente. A partir daquela posição somente é válido caminhamento para a direita (EAST) ou para baixo (SOUTH). Em (c), o cabeçalho do pacote chega à chave 1x1, sendo definido o quadrado envolvente onde a próxima porta válida é para a direita (EAST) ou para baixo (SOUTH). Finalmente o pacote chega ao seu destino em (e), tendo passado por 5 chaves.

Foram descritos oito algoritmos de roteamento para a NoC Hermes, sendo que, o XY adaptativo mínimo é propenso a *deadlock* e o XY adaptativo não mínimo é propenso a *deadlock* e *livelock*. Os outros seis algoritmos de roteamento descritos foram o XY puro, *North Last* mínimo, *North Last* não mínimo, *West First* mínimo, *West First* não mínimo e o *Negative First* não mínimo, todos comprovadamente livres de *deadlock* [NI93]. Estes últimos algoritmos foram descritos por dois motivos. O primeiro foi validar a funcionalidade da NoC Hermes TL sem que erros causados pelos algoritmos de roteamento comprometessem esta tarefa. O segundo motivo foi testar a parametrização do algoritmo de roteamento da NoC Hermes, através do qual diferentes tipos de roteamentos são assumidos em situações de congestionamento da rede. Os algoritmos descritos atenderam as seguintes características para roteamento:

- Quanto ao momento de realização do roteamento: Dinâmico (realizado em tempo de execução);

- Quanto ao número de destinos: Unicast (apenas um destino);
- Quanto ao local onde a decisão de roteamento é tomada: Distribuída (realizada em cada chave);
- Quanto à implementação: Máquina de estados (baseado em algoritmo);
- Quanto à adaptatividade: Determinístico para o algoritmo XY puro e adaptativo para os demais, sendo os adaptativos classificados ainda como:
- Quanto à progressividade: progressivo (reserva caminho durante a transmissão);
- Quanto à minimalidade: mínimo para os algoritmos XY adaptativo mínimo, west-first mínimo. Não mínimo para os algoritmos XY adaptativo não mínimo, west-first não mínimo, north-last não mínimo e negative-first não mínimo.
- Quanto ao número de caminhos: completo para o algoritmo XY adaptativo não mínimo e parcial para os demais.

Os algoritmos de roteamento mínimo fazem uso de quadrado envolvente que limita o caminhamento possível. Os algoritmos não mínimos não são limitados a caminhos dentro do quadrado envolvente. A seguir são apresentados os comportamentos de cada algoritmo utilizado, com exceção do algoritmo XY puro, visto que este já foi apresentado na proposta da NoC Hermes. A implementação dos algoritmos de roteamento é apresentada no Apêndice A.

O algoritmo XY adaptativo procura selecionar portas que se aproximam da chave destino do pacote. Assim, caso o endereço XY da chave destino seja maior que o endereço XY da chave local as portas selecionadas para saída são a *EAST* ou a *SOUTH*. Para a versão com caminhamento mínimo do algoritmo, estas duas portas são a única alternativa de roteamento. Para a versão com caminhamento não mínimo, se nenhuma das duas portas puder ser usada, a porta *NORTH* ou a *WEST* são analisadas como alternativas de roteamento.

O algoritmo *west-first* seleciona o caminhamento pela porta *WEST* antes das demais direções, caso este seja viável. Sempre que um pacote começa a ser transferido pela NoC Hermes que emprega este algoritmo, a seguinte regra é adotada: um pacote poderá seguir no sentido *WEST* apenas se nunca seguiu em nenhum dos outros sentidos (*EAST/NORTH/SOUTH*). Este caminhamento para *WEST* necessariamente ocorre quando o endereço XY da chave destino é menor que o endereço XY da chave local. Os demais casos dependem do tipo de caminhamento, se for mínimo ou não mínimo. No não mínimo o pacote pode caminhar para *WEST* mesmo que o endereço XY da chave destino não seja maior que o endereço XY da chave local, já no mínimo isto não ocorre.

O algoritmo *north-last* usa inicialmente qualquer sentido de deslocamento. Contudo, tendo ido uma vez para *NORTH* movimentos subseqüentes serão apenas para *NORTH*. O comportamento deste algoritmo é similar ao XY adaptativo quanto à liberdade de caminhamento nas direções antes da utilização da porta *NORTH* pela primeira vez.

O algoritmo *negative-first* necessariamente utiliza caminhamento na direção negativa (portas *WEST* e *SOUTH*) da rede sempre que seu destino for menor a origem. No momento em que o pacote evoluir em uma direção positiva (ou seja, passando por uma das portas *EAST* ou *NORTH* em alguma chave), a negativa não pode mais ser utilizada. O caminhamento inicialmente na direção negativa pode não ser utilizado quando a origem do pacote está em um ponto cujo endereço contém somente coordenadas menores que aquelas do destino do pacote.

Inicialmente os algoritmos de roteamento descritos foram validados a partir de teste de mesa. Todavia, esta prática mostrou-se tediosa e propensa a erros para NoCs de maior dimensão e para pacotes acima de determinado tamanho. Foram gerados arquivos com detalhamento dos caminhos percorridos por pacotes durante a simulação da NoC para a validação dos algoritmos. Contudo esta prática, apesar de mais eficiente que a anterior, também se mostrou complicada e trabalhosa. Finalmente, foi utilizada a ferramenta de simulação[2] de algoritmos de roteamento para topologia malha, que permitiu a validação visual dos algoritmos descritos. Esta ferramenta permite verificar se o algoritmo de roteamento adotado para o caminhamento de um pacote pela rede está correto e se a condição de parada de um pacote, no caso de um bloqueio, está correta. Entende-se por condição de parada correta de um pacote a situação em que, ao chegar a uma chave, o pacote não pode ser roteado, pela aplicação do algoritmo, por nenhuma das portas livres, ficando assim o pacote parado nesta chave.

4.1.3 Estrutura geral da NoC Hermes TL

O mecanismo de interconexão entre chaves na NoC Hermes, e entre uma chave e um núcleo IP local é um canal denominado *interRouterChl*. Este canal implementa os métodos definidos pelas interfaces *intoRouterIf* e *outFromRouterIf*, apresentados na Tabela 4 e na Tabela 7. Neste canal não há controle sobre o tipo de *flit* que está trafegando entre as chaves, tampouco mecanismos mais complexos de gerenciamento dos dados. O canal *interRouterChl* implementa o controle de envio e confirmação de recebimento de dados transmitidos, como ilustrado na Figura 37. Este canal de fato implementa a estratégia de controle de fluxo da rede. Para a transferência de dados, são utilizados dados do tipo *sc_event* para sincronismo de comunicação entre chaves, ou entre chave e núcleo IP. A cada porta de uma chave são associados dois canais do tipo *interRouterChl*, um para a entrada e outro para a saída de dados.

Na construção da NoC Hermes alguns parâmetros podem ser informados para alterar algumas características da rede. A primeira característica é a forma como a NoC Hermes pode ser construída. A interconexão das chaves define a topologia da rede. Apesar de possível, a descrição de diferentes topologias da NoC Hermes exige um processo trabalhoso de descrição. Há alguns outros parâmetros que podem ser alterados e que não representam um trabalho tão grande, tal como a o tipo de dado adotado para o flit, o tamanho do flit, o número de portas adotado em cada chave na rede, profundidade das filas de armazenamento temporário e o algoritmo de roteamento. Com exceção do algoritmo de roteamento, os parâmetros citados são informados no arquivo doordefs.h. O algoritmo de roteamento a ser adotado deve ser informado à NoC Hermes no construtor do módulo que interconecta as chaves, como apresentado no apêndice E, onde uma NoC 2x2 está descrita.

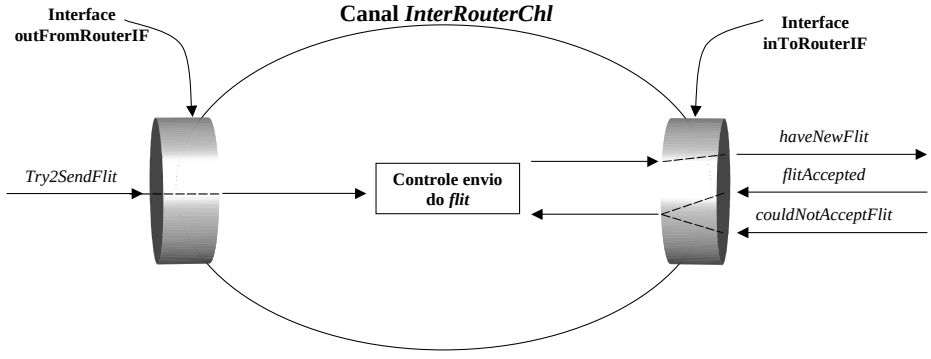


Figura 37 – Estrutura do canal de interconexão entre chaves ou entre chave e núcleo IP.

[2] A ferramenta de simulação do algoritmo de roteamento para topologia *malha* foi desenvolvida pela bolsista Aline Viera Mello.

4.2 Validação da NoC Hermes TL

A validação da NoC Hermes foi dividida em validação da funcionalidade e análise do desempenho. Quanto à funcionalidade, inicialmente validou-se a transmissão de pacotes pela rede. Para tanto, foram inseridos pacotes com fontes e destinos aleatórios na NoC. Esta validação permite garantir o correto funcionamento da rede e de sua interconexão em linhas gerais. A segunda característica de funcionalidade validada refere-se ao paralelismo na transmissão de pacotes. Para este segundo critério de validação, dois conjuntos de pacotes, com fontes aleatórias na rede, foram utilizados. O primeiro com destinos aleatórios e o segundo concentrado em uma mesma chave destino. Com isto pode-se garantir a funcionalidade da concorrência na transmissão dos pacotes.

Para a validação do desempenho uma forma alternativa de análise teve de ser adotada, pois a ausência de um sinal de relógio para ser usado como referência, comum em projetos RTL, não está presente. A necessidade de uma métrica diferenciada para avaliar a simultaneidade ou não do funcionamento da NoC faz-se necessária. A métrica utilizada foi uma ponderação do número de vezes que cada pacote ganhou a prioridade em cada chave até ser roteado e do número de chaves que passou até atingir seu destino. Estas informações permitem analisar detalhes como a carga da rede, se o mecanismo de arbitragem que está sendo utilizado é eficiente e comparar abstratamente algoritmos de roteamento, por exemplo.

4.2.1 Validação básica da transmissão de pacotes

Para a geração de pacotes para a validação da NoC, foi utilizada a ferramenta *TrafficGen*[3] [OST05], pois esta permite a geração de pacotes com tamanhos e destinos aleatórios, a partir de diferentes fontes. Com esta ferramenta, é gerado um arquivo para cada chave da NoC, onde cada linha do arquivo representa um pacote. Complementar a esta ferramenta, dois núcleos IP em SystemC denominados *flitGenerator* e *flitConsumer* foram descritos, os quais são conectados a cada chave da NoC através da porta *LOCAL*. O *flitGenerator* conecta-se à porta *LOCAL* na entrada da rede associada a uma chave. Ele lê o arquivo gerado pelo *trafficGen*, referente a sua posição na rede, interpreta cada linha e gera o pacote a ser transmitido, conforme mostra a Figura 38. Como previamente citado, o *trafficGen* gera diferentes arquivos que podem ser lidos pelo *flitGenerator*. Estes arquivos recebem o nome de *in0.txt*, *in1.txt*, *in(N-1).txt*, sendo N o número de chaves que compõem a NoC. A cada *flitGenerator* é atribuído um número identificador que vai de zero à N-1. Esta identificação é utilizada como base para a leitura dos arquivos gerados pelo *trafficGen*. O *flitConsumer* conecta-se à porta *LOCAL* na saída da rede associada a uma chave. Ele simplesmente consome o pacote que chega a ele.

A primeira funcionalidade a ser testada foi a de entrega dos pacotes. Para esta tarefa foi desenvolvida uma NoC 6x6, onde foram conectados 36 *flitGenerators* e 36 *flitConsumers*, um par destes em cada chave. A ferramenta *trafficGen* gerou 36 arquivos com conteúdos aleatórios. A visualização na tela da partida de um pacote e depois de algum tempo a visualização da chegada no destino mostrou-se uma tarefa complexa. Para superar esta dificuldade a descrição da NoC foi alterada para informar a cada chave a porta que estava requisitando roteamento, o endereço de destino do pacote e a porta destinada a este pacote, a partir da geração de um arquivo de resultados de execução. Esta abordagem facilitou o trabalho de detecção dos pacotes que entravam e saíam da rede. Durante esta validação, foram encontrados problemas de transmissão dos pacotes. Inicialmente alguns comportamentos não haviam sido levados em

[3] A ferramenta *TrafficGen* foi desenvolvida pelo bolsista Leandro Heleno Möller para validação da NoC Hermes descrita em VHDL.

consideração tais como pacotes com tamanho zero e pacotes cujo endereço destino era igual ao endereço origem. Esta validação permitiu a correção destes problemas e a garantia de funcionamento da NoC. Obviamente a linguagem SystemC por si só não apresenta vantagens específicas para a detecção dos problemas anteriormente citados. O ganho está em ter descoberto tal erro ainda nos primeiros passos de desenvolvimento e percebê-lo como um item a ser analisado ao longo do processo de refinamento do projeto. Assim sendo, a implementação em alto nível de abstração suportado pela linguagem SystemC permitiu a detecção do problema logo nos primeiros passos do projeto.

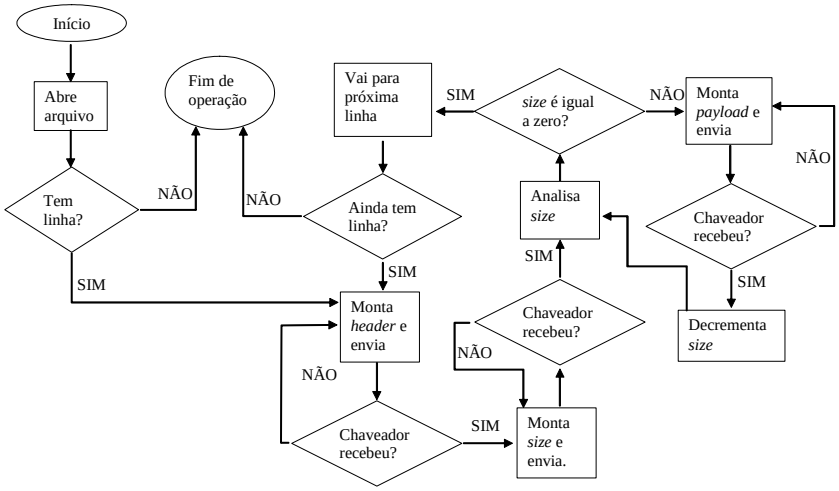


Figura 38 – Comportamento do módulo flitGenerator, auxiliar na validação da NoC Hermes TL.

4.2.2 Validação da transmissão paralela de pacotes

A segunda funcionalidade a ser validada foi a de paralelismo de transmissão de pacotes na rede. Com base nos arquivos de resultados de execução gerados pela descrição da NoC mencionados anteriormente, foi possível analisar que havia paralelismo na transmissão dos pacotes. Os pacotes que trafegavam na NoC possuiam, em média 20 *flits* por pacote, o que poderia bloquear em média 3 a 4 chaves durante o caminhamento pela rede, visto que a profundidade da fila utilizada era de 8 posições. Pôde-se observar, que em determinadas ocasiões um pacote chegava ao seu destino e logo começava a ser consumido pelo *flitConsumer*. Alguns instantes depois, outro pacote chegava a mesma chave para ser consumido pelo *flitConsumer*, porém permanecia ocupando as posições no *buffer* das portas de entrada e aguardando a liberação da porta *LOCAL*, conforme ilustra a Figura 39.

```
[0x0],Porta :[4], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [1], latenciaTotal: [2]
[5x2],Porta :[4], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [0], latenciaTotal: [2]
[0x1],Porta :[0], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [2], latenciaTotal: [3]
[5x1],Porta :[1], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [3], latenciaTotal: [4]
[4x1],Porta :[2], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [4], latenciaTotal: [3]
[1x1],Porta :[3], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [2], latenciaTotal: [3]
[2x1],Porta :[3], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [2], latenciaTotal: [2]
[3x1],Porta :[3], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [2], latenciaTotal: [1]
.[4x1],Porta :[3], HEADER: [4x1],Nao foi possivel rotear o pacote, Tentativas: [17]
[4x1],Porta :[3], HEADER: [4x1],Roteamento realizado com sucesso, Porta de destino: [4], latenciaTotal: [18]
```

Figura 39 – Trecho do arquivo gerado durante a simulação da NoC Hermes TL. Para linhas que iniciam com "[", a primeira informação em negrito são coordenadas da chave atual do pacote, a segunda informação é a porta requisitando roteamento, a terceira a chave destino e a quarta a porta para onde o pacote foi direcionado. Para as linhas que iniciam com ".", a primeira, a segunda e a terceira informações são idênticas às anteriores. As primeiras 4 linhas mostram o caminhamento do pacote da chave 0x0 e 5x2 em direção à chave 4x1 em paralelo. O pacote da chave 5x2 chega primeiro à chave 4x1 na linha 5. As demais linhas apresentam o caminho do pacote proveniente da chave 0x0, até o recebimento na chave 4x1.

Com tal mecanismo foi possível detectar que no mesmo instante que um pacote está sendo consumido em um determinado endereço, os demais podem estar trafegando pela rede. Conforme o trecho do arquivo de log apresentado na Figura 39, é pode-se determinar que há dois pacotes trafegando pela rede tendo como origem as chaves 0x0 e 5x2, ambos com destino à chave 4x1. Com base na Figura 39, o caminhamento de ambos os pacotes é ilustrado na Figura 40. Na Figura 39, a primeira linha representa a partida do pacote PCK1 e a segunda representa a partida do pacote PCK2. Na quinta linha, o pacote PCK2 chega ao destino tendo este pacote percorrido as chaves 5x1 e 4x1. Na última linha o pacote PCK1 chega ao destino, tendo este pacote percorrido as chaves 0x0, 0x1, 1x1, 2x1, 3x1 e 4x1.

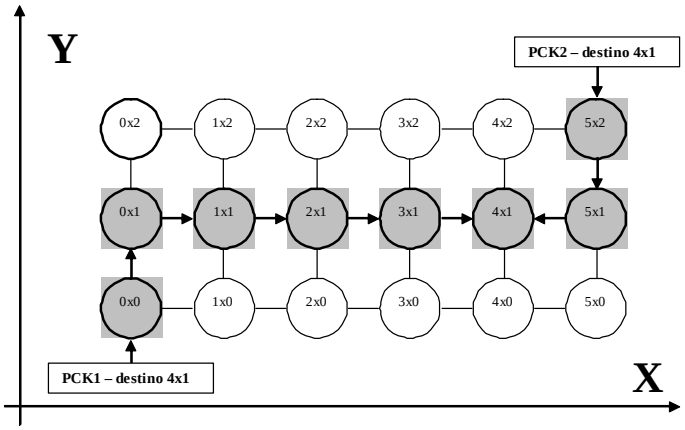


Figura 40 – Caminhamento dos pacotes na rede, de acordo com o trecho do arquivo apresentado na Figura 39.

4.2.3 Avaliação abstrata do desempenho da NoC

Como dito anteriormente, na modelagem TL da NoC Hermes não há o sinal de relógio para ser utilizado como referência para avaliação de desempenho da NoC. Assim sendo, a modelagem da NoC TLM exige o uso de métricas alternativas para a análise de desempenho.

Neste trabalho foram definidas duas métricas ambas utilizando como base o número de tentativas de transmissão de um pacote de uma chave para outra. A primeira reporta o número de vezes que cada porta ganhou acesso ao recurso de roteamento antes de enviar o pacote para a porta destino. Esta métrica foi denominada “tentativas de roteamento”. A segunda reporta o número de vezes que uma porta competiu por acesso ao recurso de roteamento antes de enviar o pacote. Esta métrica foi denominada “concorrência por roteamento”.

De posse destas duas definições de métricas para avaliação de desempenho a descrição da NoC Hermes TL foi alterada para que a informação pudesse ser capturada. Estas informações então foram incluídas no arquivo de resultados de execução gerado pela descrição da NoC Hermes TL.

A métrica de “tentativas de roteamento” permite avaliar critérios que facilitem o fluxo de informações pela rede. Quanto menor forem os valores capturados para esta métrica mais livre está a rede, ou seja, menos canais estarão ocupados. Parâmetros que venham a contribuir para uma rede mais livre podem ser obtidos com o uso de algoritmos de roteamento que não permitam grande liberdade de caminhamento pela rede e filas de armazenamento temporário mais profundas, por exemplo.

A métrica de “concorrência por roteamento” permite avaliar se a latência interna em cada chave está coerente. A entrega de pacotes que respeitam um tempo máximo para transitarem pela rede é interessante. Pacotes que excedam este tempo máximo podem representar latência muito grande na rede. Esta latência pode estar principalmente vinculada a concorrência pelo recurso de roteamento quando muitos pacotes trafegam pela rede. A presente métrica permite avaliar se o algoritmo de arbitragem que está sendo utilizado é o mais eficiente.

De posse destas duas métricas diferentes algoritmos de roteamento e arbitragem podem ser comparados. Em alto nível de abstração é obtido o algoritmo que permite o maior ganho de desempenho. De posse dos algoritmos mais eficientes, uma relação custo-benefício para o tamanho de hardware descrito e o ganho de desempenho pode ser realizada. O critério a ser adotado fica a cargo dos projetistas.

5 Recursos adicionais: interfaces OCP e ferramentas de apoio

NoCs são recursos de comunicação para SoCs complexos. Assim, a padronização de suas interfaces com IPs procesadores do SoC que as contém é recomendável [OCP05b]. A implementação descrita nas Seções 5.1 e 5.2 pressupõem uma NoC com interface nativa. Para que esta possa ser usada e portada para SoCs dotados de IPs processadores com interfaces padronizadas, forma desenvolvidos módulos adaptadores da interface nativa para o protocolo padrão OCP [OCP05a].

A descrição do projeto e validação destes módulos é alvo da Seção 5.1. Como se escolheu implementar estes módulos detalhando o protocolo OCP no nível físico como precisão em nível de ciclos de relógio, lançou-se mão do conceito de *transatores*, que convertem transações em nível TL em sinais com precisão RTL. Tais transatores implementam interface padronizada OCP permitindo o reuso e a validação de núcleos IP descritos em SystemC RTL/TL com a NoC Hermes descrita em SystemC TL.

O processo de validação de projeto da NoC Hermes TL revelou-se complexo e propenso a erros. Para aliviar estas dificuldades, foram desenvolvidas duas ferramentas específicas. A primeira permite gerar descrições de NoCs em nível TL de forma automatizada, a partir de parametrização do usuário. A segunda auxilia no processo de identificação de caminhos percorridos por pacotes durante a execução de simulações de sistemas contendo NoCs. A descrição destas ferramentas está contida na Seção 5.2.

5.1 Interfaces padronizadas OCP em SystemC

O objetivo da NoC TL é prover uma implementação abstrata da comunicação entre diferentes núcleos IP, e eficiente em termos de velocidade de simulação. Os núcleos IP em si não precisam ser descritos no mesmo nível de abstração que a NoC Hermes, ou seja, núcleos IP TLM podem estar sendo simulados juntamente com núcleos IP RTL, utilizando como meio de comunicação a NoC Hermes TLM. Na implementação da NoC Hermes TL não foram utilizadas interfaces detalhadas através de sinais, o que para projetos que venham a utilizar tal detalhamento pode representar um complicador principalmente para projetos baseados em componentes.

Para integrar núcleos IP de diferentes níveis de abstração, fez-se necessário implementar módulos transatores que recebam sinais e convertam os mesmos em comunicação abstrata ou recebam uma comunicação abstrata e a convertam em sinais detalhados. Há duas possibilidades de implementação de interface detalhada. A primeira é a representação da mesma interface da NoC Hermes RTL descrita em VHDL. A segunda era a implementação de uma interface padronizada, tal como a interface OCP [OCP05a]. Em projetos que visam reuso, o uso de interfaces padronizadas ao invés de interfaces proprietárias é vantajoso. Do ponto de vista de implementação, estas interfaces representam apenas uma adaptação externa da NoC Hermes TLM. Internamente, a comunicação da NoC, ou seja, entre as chaves, permanece ocorrendo de forma abstrata.

A descrição de interfaces padronizadas baseadas no padrão OCP provê duas contribuições. A primeira é a possibilidade de validação mista de modelos descritos em RTL e TL, facilitando a garantia da funcionalidade do sistema que se deseja implementar. A segunda contribuição diz respeito ao reuso de projeto e os ganhos que se pode obter frente à eficiência de simulação com a NoC TL. A presente Seção tem por finalidade descrever a implementação dos transatores OCP e uma semântica de comunicação adotada para a NoC Hermes TL.

5.1.1 Transator NoC OCP *Slave*

A interface OCP *Slave* disponibiliza um mecanismo de comunicação entre núcleos IP mestres e dispositivos escravos. Através desta interface, tanto comandos de escrita quanto de leitura podem ser requisitados pelo núcleo IP mestre, conforme ilustra a Figura 41.

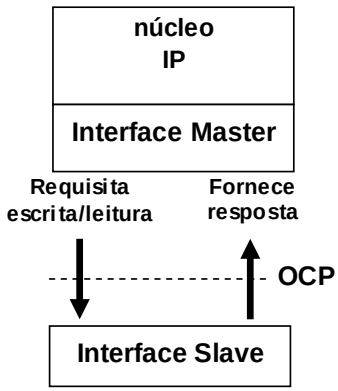


Figura 41 – Ações associadas a um núcleo IP mestre conectado à interface OCP escrava.

O transator NoC OCP *Slave* é uma interface escrava conectada à NoC TL através de uma chave específica via a porta *LOCAL* desta, como mostra a Figura 42. Internamente a este transator não há a implementação de uma máquina de estados, apenas a interpretação dos sinais ou da comunicação abstrata que chega até o transator. A implementação é predominantemente comportamental e não sintetizável.

O transator NoC OCP *Slave* tem duas funcionalidades básicas. A primeira é capturar e repassar comandos de leitura ou escrita do núcleo IP mestre, passados através da interface OCP *Master*, para a chave vinculada. A segunda é repassar ao núcleo IP mestre os dados solicitados em uma operação de leitura. Para implementar a primeira funcionalidade, o transator provê um mecanismo de interpretação de sinais detalhados e gera comunicação abstrata para a rede. Para a interpretação de sinais detalhados, o transator NoC OCP *slave* aguarda comandos provenientes de um núcleo IP mestre.

Um exemplo de núcleo IP mestre pode ser um dispositivo MP3 que faz leitura em uma memória remota. No momento em que o dispositivo MP3 requisitar a leitura, um conjunto de sinais OCP *Master* é gerado, sendo estes sinais capturados pelo transator NoC OCP *Slave*. Tais sinais são interpretados e a um pacote é gerado, sendo enviado à NoC através da comunicação abstrata entre o transator NoC OCP *slave* e a NoC. Note que a memória pode estar descrita em TL ou RTL, estando conectada a outra chave da NoC.

O transator NoC OCP *Slave* possui uma interface OCP *slave* que emprega um mínimo de sinais, e duas interfaces abstratas (*inToRouterIf* e *outFromRouterIf*) definidas para a NoC, conforme ilustrado na Figura 42. A comunicação entre o transator e a NoC ocorre através do canal *interRouterChl*. Já a comunicação entre o transator e o *wrapper* OCP *master* do núcleo IP *master* ocorre através de compartilhamento de sinais. A comunicação entre o transator e a interface OCP *master* ocorre de forma sincronizada através do uso de um sinal de relógio e um sinal de *reset*. O comportamento do transator NoC OCP *slave* é dividido em envio de requisições de leitura ou escrita para a um núcleo IP disponível em um determinado endereço na rede, e o recebimento da resposta, sendo esta definida de acordo com uma semântica de comunicação.

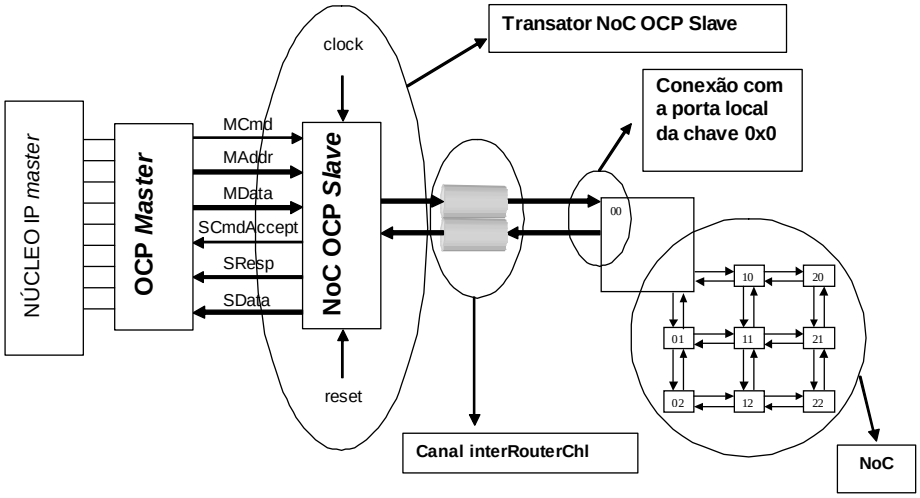


Figura 42 – Estrutura do transator NoC OCP Slave. O transator recebe sinais OCP detalhados e gera comunicação abstrata com a NoC Hermes.

Para a implementação da interface detalhada, os parâmetros apresentados na Tabela 8 foram definidos. Foram utilizados os sinais básicos OCP [OCP05a], como mencionados anteriormente. Sobre estes sinais foi definida uma semântica de comunicação, o que é mais bem detalhado a seguir.

Tabela 8 – Interface OCP básica utilizada no transator OCP Slave da NoC Hermes TL.

Sinais	Tamanho	Sentido	Descrição
MCmd	3 bits	Entrada	Identifica o tipo de requisição
MAddr	16 bits	Entrada	Identifica o endereço a ser lido ou escrito
MData	16 bits	Entrada	Contém o dado a ser escrito
SCmdAccept	1 bit	Saída	Indica que o comando foi aceito pela interface escrava
SResp	2 bits	Saída	Indica a resposta de uma leitura
SData	16 bits	Saída	Contém o dado lido

A semântica de comunicação corresponde ao envio do endereço destino da requisição, do tamanho da mensagem que será enviada e da mensagem em si. O comando é interpretado pelo transator e um pacote é gerado para ser transmitido pela rede. Caso seja gerada uma requisição de leitura, o transator libera o núcleo IP mestre logo que a mensagem pode ser transmitida pela rede, o que evita o bloqueio da execução do núcleo IP mestre. Ao receber a resposta proveniente do endereço solicitado, o transator informa ao *wrapper* OCP *Master* que a resposta à requisição está disponível.

5.1.2 Transator NoC OCP *Master*

A interface OCP *Master* disponibiliza um mecanismo de comunicação entre núcleos IP escravos e dispositivos mestres. Através desta interface, tanto comandos de escrita quanto de leitura podem ser recebidos pelo núcleo IP escravo, conforme ilustra a Figura 43.

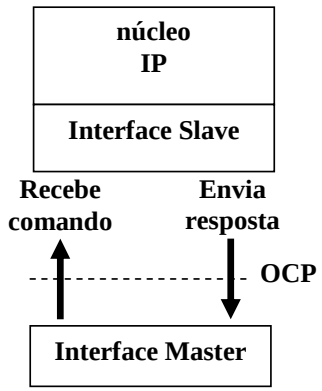


Figura 43 – Ações associadas a um núcleo IP escravo conectado a uma interface OCP mestre.

O transator NoC OCP *Master* é uma interface mestre conectada à NoC TL por uma chave, através da porta *LOCAL*, como mostra a Figura 44. Internamente a este transator, não há a implementação de uma máquina de estados, apenas a interpretação dos sinais ou da comunicação abstrata que chega até o transator. A implementação é predominantemente comportamental e não sintetizável, assim como ocorre com o transator NoC OCP *Slave*.

O transator NoC OCP *Master* tem duas funcionalidades básicas, assim como o transator NoC OCP *Slave*. São elas: repassar para o núcleo IP escravo os comandos de leitura e escrita recebidos através da NoC e enviar através da NoC a resposta a uma requisição de leitura. A primeira funcionalidade é obtida através da interpretação do pacote que trafega pela rede e a conseqüente geração de sinais detalhados pela interface OCP com um núcleo IP escravo conectado a este transator. Assim sendo, a comunicação abstrata através da NoC gera estímulos através da interface OCP. A segunda funcionalidade é obtida a partir dos sinais OCP de resposta gerados pelo núcleo IP escravo. A partir desta resposta um pacote é enviado à rede através da interface abstrata.

Um exemplo de núcleo IP escravo pode ser uma memória que recebe comandos de leitura e escrita. Sempre que algum dispositivo mestre ou mestre/escravo conectado à NoC Hermes requisitar uma operação com esta memória, um pacote contendo um comando será recebido pelo transator NoC OCP *Master*. A partir desta comunicação abstrata, um conjunto de sinais OCP será gerado.

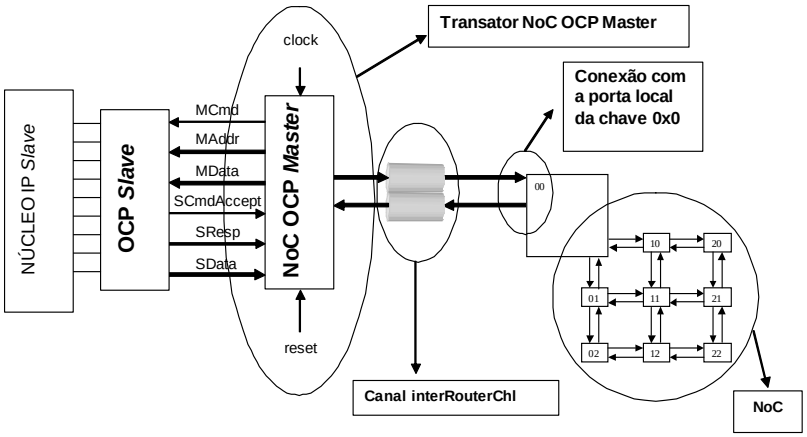


Figura 44 – Estrutura do transator NoC OCP *Master*. A comunicação é recebida da rede. A partir da comunicação abstrata, os sinais OCP do transator NoC OCP *Master* são ativados e transmitidos ao núcleo IP *Slave.*

O transator NoC OCP *Master* possui uma interface OCP *Master* básica, descrita com sinais detalhados, e duas interfaces abstratas *inToRouterIf* e *outFromRouterIf* definida para a NoC, conforme ilustrado na Figura 44. A comunicação entre a NoC Hermes e o núcleo IP escravo ocorre da mesma forma que com o transator NoC OCP *Slave*. A interface detalhada é conectada ao núcleo IP escravo e as portas abstratas são conectadas à NoC, uma para envio de pacotes à rede e outra de recebimento de pacotes da rede.

Para a implementação da interface detalhada, os parâmetros apresentados na Tabela 9 foram definidos. Este conjunto de sinais segue o padrão OCP também definido para o transator NoC OCP *Slave*. Assim como naquele transator, sobre estes sinais foi definida uma semântica de comunicação, detalhado a seguir.

Tabela 9 – Interface OCP básica utilizada nos transatores OCP da NoC Hermes TL.

Sinais	Tamanho	Sentido	Descrição
MCmd	3 bits	Saída	Identifica o tipo de requisição(escrita/leitura)
MAddr	16 bits	Saída	Identifica o endereço a ser lido ou escrito
MData	16 bits	Saída	Contém o dado a ser escrito
SCmdAccept	1 bit	Entrada	Indica que o comando foi aceito pela interface escrava
SResp	2 bits	Entrada	Indica a resposta de uma leitura
SData	16 bits	Entrada	Contém o dado lido

A semântica de comunicação adotada na rede define que ao receber um pacote, o transator descarta o primeiro *flit*, armazenando apenas o endereço de retorno. O segundo *flit* contém o tamanho do *payload* da mensagem, incluindo o comando. O terceiro *flit* contém o comando que será gerado. Caso o valor deste *flit* seja 1, o transator está recebendo uma requisição de escrita, e caso seja 2 o transator está recebendo uma requisição de leitura. No caso de uma requisição de escrita, os dois próximos *flits* representam o endereço de escrita e o dado. Para esta requisição, o tamanho mínimo que deve ter sido recebido no *flit size* é 3. Caso este valor seja maior que 3, mais de uma escrita está sendo requisitada no mesmo pacote, caracterizando uma escrita em rajada. Para o caso de uma requisição de leitura, o tamanho mínimo do

size flit é 2, um *flit* de comando e um de endereço. Caso este valor seja maior que 2, mais de uma requisição de leitura estará sendo realizada no mesmo pacote, caracterizando uma leitura em rajada. A interpretação destes pacotes faz com que um conjunto de estímulos obedecendo ao protocolo OCP seja gerado. Estes estímulos são enviados ao *wrapper* OCP *slave*, que se responsabiliza pela comunicação com o núcleo IP *slave*.

5.1.3 Transator NoC OCP *Master*/*Slave*

A interface OCP *Master/Slave* disponibiliza um mecanismo de comunicação entre núcleos IP mestre escravo e qualquer outro dispositivo da rede. Núcleos IP mestre escravo têm como característica poder ser acionadores ou dispositivos que respondem a requisições de outros dispositivos. Através desta interface solicitações de leitura podem ser executadas por estes ou recebidas. Da mesma forma, respostas podem ser recebidas ou enviadas, conforme ilustra a Figura 46.

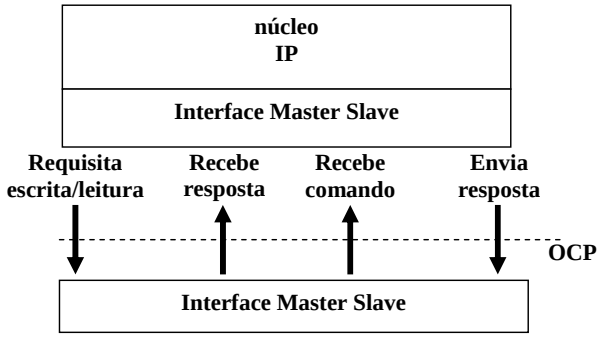


Figura 45 – Ações associadas a um núcleo IP *Master Slave* conectado a uma interface OCP *Master Slave*.

O transator NoC OCP *Master/Slave* é uma interface mestre conectada à NoC TL através de uma chave específica via a porta *LOCAL*, como mostra a Figura 46. Assim como os demais transatores, internamente este transator não implementa uma máquina de estados, apenas realiza a interpretação dos sinais OCP ou da comunicação abstrata que chega até este transator. A implementação é predominantemente comportamental e não sintetizável, assim como ocorre com os demais transatores da NoC.

O transator NoC OCP *Master/Slave* tem quatro funcionalidades. Isto se deve ao fato de que este transator une as funcionalidades dos transatores *Master* e *Slave*. Tais funcionalidades são: recebimento de comandos e repasse de resposta de uma leitura, no lado escravo deste transator, e o repasse de comandos OCP e recebimento de resposta de leitura, no lado mestre deste transator. Tais funcionalidades são alcançadas da mesma forma que as especificadas em ambos transatores *Master* e *Slave*.

Um exemplo de núcleo IP *Master/Slave* é um processador que aguarda receber uma requisição de execução, e, ao receber tal requisição, inicia atividades de leitura e escrita através da NoC. Inicialmente, este processador tem um comportamento escravo, pois aguarda um comando, e logo em seguida assume um comportamento mestre, visto que realiza comandos de leitura e escrita com os demais dispositivos na NoC.

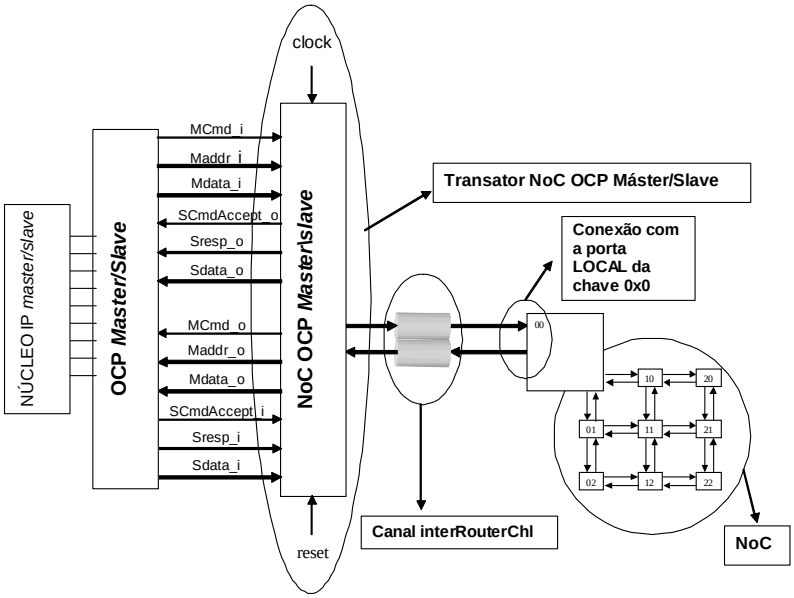


Figura 46 – Estrutura do transator NoC OCP master/slave. A comunicação é recebida da rede. A partir da comunicação abstrata, os sinais OCP do transator NoC OCP *master* são ativados e transmitidos ao núcleo IP *slave.*

O projeto do transator NoC OCP *Master/Slave* é a união duas interfaces, uma interface OCP *Master* e uma OCP *Slave* (descrita com sinais) e duas interfaces abstratas *inToRouterIf* e *outFromRouterIf* definidas para o lado da NoC, conforme ilustrado na Figura 46. A interface OCP é conectada ao núcleo IP mestre escravo e as interfaces abstratas são conectadas à NoC, uma para envio de pacotes à rede e outra de recebimento de pacotes da rede.

Para a implementação da interface OCP, os parâmetros apresentados na Tabela 10 foram definidos. Este conjunto de sinais segue o padrão OCP também definido para o transator NoC OCP *Master/Slave*, e assim como naquele transator, sobre estes sinais foi definida uma semântica de comunicação, detalhada a seguir.

Tabela 10 – Interface OCP adotada para o transator NoC OCP Master/Slave.

Sinais	Tamanho	Sentido	Interface	Descrição
MCmd_i	3 bits	Entrada	Escravo	Identifica o tipo de requisição (escrita/leitura)
MAddr_i	16 bits	Entrada	Escravo	Identifica o endereço a ser lido ou escrito
MData_i	16 bits	Entrada	Escravo	Contém o dado a ser escrito
SCmdAccept_o	1 bit	Saída	Escravo	Indica que o comando foi aceito pela interface escrava
SResp_o	2 bits	Saída	Escravo	Indica a resposta de uma leitura
SData_o	16 bits	Saída	Escravo	Contém o dado lido
MCmd_o	3 bits	Saída	Mestre	Identifica o tipo de requisição (escrita/leitura)
MAddr_o	16 bits	Saída	Mestre	Identifica o endereço a ser lido ou escrito
MData_o	16 bits	Saída	Mestre	Contém o dado a ser escrito
SCmdAccept_i	1 bit	Entrada	Mestre	Indica que o comando foi aceito pela interface escrava
SResp_i	2 bits	Entrada	Mestre	Indica a resposta de uma leitura
SData_i	16 bits	Entrada	Mestre	Contém o dado lido

Ao receber um pacote, o transator descarta o primeiro *flit* (endereço de destino do pacote) armazenando apenas o endereço de retorno. O segundo *flit* contém o tamanho do *payload* da mensagem, incluindo o comando. O terceiro *flit* contém o código do comando a ser gerado na interface OCP. Caso o valor deste *flit* seja 1, o transator está recebendo uma requisição de escrita. Caso seja 2, o transator está recebendo uma requisição de leitura, e caso seja 8 o transator está recebendo uma resposta de uma requisição realizada pelo núcleo IP *Master/Slave*. No caso de uma requisição de escrita, os dois próximos *flits* representam o endereço de escrita e o dado. Para esta requisição, o tamanho mínimo que deve ter sido recebido no *flit size* é 3. Caso este valor seja maior que 3, mais de uma escrita está sendo requisitada no mesmo pacote, caracterizando uma escrita em rajada. Para o caso de uma requisição de leitura, o tamanho mínimo do *flit size* é 2, um *flit* de comando e um de endereço. Caso este valor seja maior que 2, mais de uma requisição de leitura estará sendo realizada no mesmo pacote, caracterizando uma leitura em rajada. A interpretação destes pacotes faz com que um conjunto de estímulos obedecendo ao protocolo OCP seja gerado. Estes estímulos são enviados ao *wrapper* OCP *Slave*, que se responsabiliza pela comunicação com o núcleo IP *slave*. Para o caso em que o comando é uma resposta de requisição anterior, o *size flit* informa quantos dados estarão sendo recebidos.

5.1.4 Validação dos transatores NoC OCP

Para validação das interfaces padronizadas, foram definidos dois núcleos IP, um implementando um módulo descrevendo o algoritmo de cálculo da série de *Fibonacci*, a partir daqui denominado *Fibonacci*, e um descrevendo o comportamento de um núcleo IP memória, a partir deste ponto denominado Memória. *Fibonacci* é um módulo mestre, que requisitam comandos de leitura e escrita, enquanto Memória é um módulo escravo que responde as requisições do núcleo IP mestre. O núcleo IP *Fibonacci* calcula os 100 primeiros números da série de *Fibonacci.* No núcleo IP *Fibonacci* foi definido um processo para realizar o cálculo da série. Inicialmente este processo escreve o primeiro e segundo valores da série. Após isto, são executadas duas requisições de leitura da memória e uma de escrita. Nas duas leituras, os dados capturados da memória são utilizados como base para o cálculo da série de *Fibonacci*. Na escrita, o resultado da soma dos dois valores lidos é armazenado na memória.

Para validar os transatores NoC OCP *Master* e NoC OCP *Slave* a estrutura ilustrada na Figura 47 foi empregada. Nesta validação, a NoC Hermes inicialmente não está presente. Este modelo serve exclusivamente para validar funcionalmente os transatores NoC OCP *Master* e NoC OCP *Slave*. Conforme a funcionalidade do transator NoC OCP Slave, sempre que uma requisição é gerada por *Fibonacci*, um pacote é gerado e enviado na NoC Hermes. Este pacote é transmitido ao transator NoC OCP *Master* onde é interpretado.

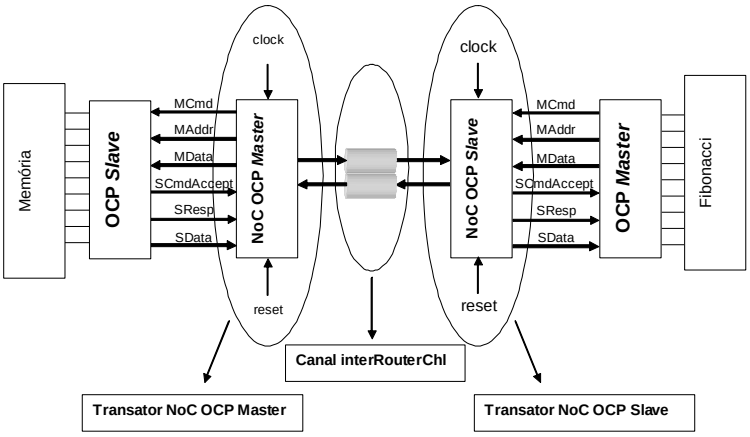


Figura 47 – Ambiente de validação dos transatores NoC OCP *Master* e NoC OCP *Slave*.

Para a validação do funcionamento dos transatores, foram utilizadas duas estratégias. A primeira foi a apresentação dos valores lidos e o resultado do cálculo na própria tela. A segunda forma foi a validação através de formas de onda, tendo então sido gerado um arquivo com formato VCD e utilizado o software gtkWave. A cada execução do ambiente de validação, foram realizadas 196 requisições de leitura e 100 requisições de escrita.

Através deste ambiente de validação, pôde-se constatar que o funcionamento tanto do transator NoC OCP *Master* quanto do NoC OCP *Slave* estava correta. Os estímulos gerados pelo núcleo IP *Fibonacci* foram corretamente capturados pelo transator NoC OCP *Slave*, repassados sob a forma de pacotes para o transator NoC OCP *Master*, ali interpretados e transmitidos para a Memória. Finalizada a validação destes dois transatores, iniciou-se a validação do transator NoC OCP *Master/Slave*.

Para a validação do transator NoC OCP *Master/Slave*, foi utilizado um ambiente de validação similar ao descrito anteriormente. A NoC Hermes inicialmente não foi utilizada no ambiente de validação, o que permitiu concentrar-se na validação dos transatores. O ambiente de validação é ilustrado na Figura 48. Foram utilizados dois núcleos IP *Fibonacci* e dois núcleos IP Memória. Foi assim definido que o núcleo IP Fibonacci_1 executaria operações de leitura e escrita sobre o núcleo IP Memória_2, e o núcleo IP Fibonacci_2 realizaria operações de leitura e escrita sobre o núcleo IP Memória_1. O objetivo deste ambiente é validar tanto o funcionamento de cada uma das partes do transator NoC OCP *Master/Slave* separadamente (a mestre e a escrava) quanto as operações concorrentes de requisição (quando há o envio e o recebimento de comandos pela interface OCP).

Para a validação em separado do transator NoC OCP *Master/Slave*, o núcleo IP Fibonacci_1 requisitava operações de leitura e escrita sobre o núcleo IP Memória_2. Foram utilizados impressão na tela e arquivos com formas de onda para a validação da execução do ambiente. Assim como na validação dos transatores NoC OCP *Master* e NoC OCP *Slave*, o repasse de requisições ocorreu sem problemas, o que demonstrou que em uma situação onde não há concorrência entre as diferentes partes do transator NoC OCP *Master/Slave*, sua funcionalidade é correta.

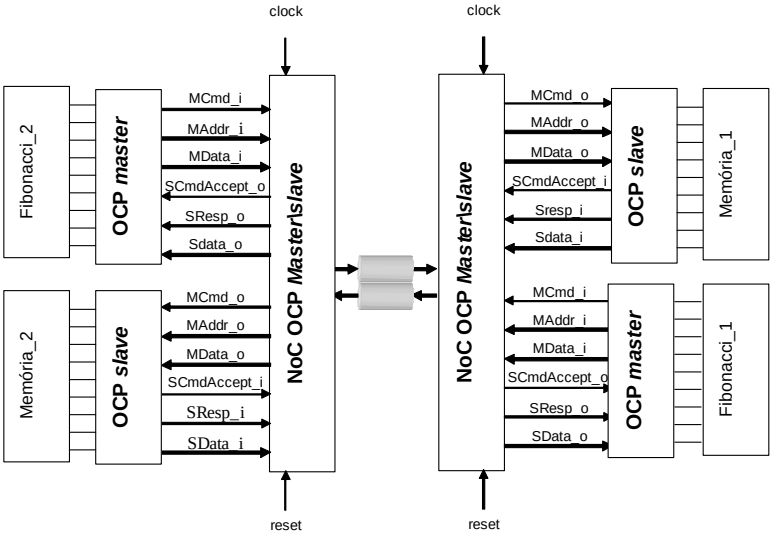


Figura 48 – Ambiente de validação do transator NoC OCP *Master Slave*. O núcleo Fibonacci_1 requisita operações de leitura e escrita no núcleo Memória_2. O núcleo Fibonacci_2 requisita operações de leitura e escrita no núcleo Memória_1.

Para a validação do ambiente com concorrência, o ambiente apresentado na Figura 48 foi utilizado, sendo que os núcleos IP Fibonacci_1 e Fibonacci_2 operavam sobre os núcleos IP Memória_2 e Memória_1 respectivamente. Este ambiente permitiu simular uma situação em que requisições eram enviadas e recebidas pelos transatores simultaneamente. Durante a execução deste ambiente de validação, foi eventualmente detectado bloqueio do processo descrito neste transator. O problema ocorreu porque no mesmo instante em que uma requisição de leitura chegava a um dos transatores *Master/Slave*, o núcleo IP *Fibonacci* conectado a este poderia estar gerando requisições de leitura para a memória no transator vizinho. Foram detectados dois problemas advindos desta situação. O primeiro foi a perda de contexto durante a operação com a memória. O segundo problema foi a ocorrência de *deadlock* entre os transatores, porque os dois transatores estavam recebendo requisições de leitura ao mesmo tempo e para evitar perda de contexto, nenhum dos transator respondia ao outro, ficando ambos aguardando que o outro responde-se. A solução encontrada consistiu em modificar o módulo de forma a garantir que a partir da chegada de uma requisição, seja do núcleo IP mestre ou de um pacote da rede, esta tinha seu atendimento garantido, independente da chegada de novas requisições. Para validar a funcionalidade dos transatores NoC OCP *Master/Slave*, foram utilizados novamente dois mecanismos. O primeiro foi através da impressão na tela das requisições de leitura e escrita e da verificação do sucesso destas operações. O segundo mecanismo consistiu da análise de formas de onda capturada durante a execução do ambiente de validação em arquivo VCD. Depois de detectado e corrigido o problema de bloqueio da execução dos transatores NoC OCP *Master/Slave*, o ambiente foi validado funcionalmente.

O resultado final foi a validação dos 3 transatores, NoC OCP *Master*, *Slave* e *Master/Slave*. Estes foram funcionalmente validados para serem utilizados em projetos. Com esta garantia, núcleos IP descritos em diferentes níveis de abstração podem ser validados juntamente com a NoC abstrata, levando em consideração apenas aspectos funcionais, onde o uso de um sinal de relógio não seja necessário. O desenvolvimento e a validação dos transatores contribuíram para o entendimento da conversão de comunicação detalhada em abstrata e vice-versa. A partir destes, outros transatores foram desenvolvidos e utilizados em projetos de SoCs, tal como aquele desenvolvido no contexto do projeto Fênix[4]. Esta

[4] http://www.brazilip.org.br/fenix/index.asp

abordagem permitiu o desenvolvimento de aplicações que mesclavam núcleos IP em diferentes níveis de abstração, o que facilita encontrar eventuais problemas de projeto logo nos primeiros passos de desenvolvimento e com tempo de simulação muito menor, dado o baixo grau de detalhamento possível de ser utilizado.

5.2 Ferramentas de apoio no projeto de NoCs

Durante o desenvolvimento deste trabalho, algumas versões da NoC Hermes TL foram descritas. Todas as versões utilizaram apenas a topologia malha, tendo como dimensões mínimas uma rede 2x2, ou seja, quatro chaves interconectadas. A maior rede descrita tinha dimensões 6x6, ou seja, trinta e seis chaves interconectadas. A descrição de uma NoC não representa uma tarefa complexa em si, mas devido à quantidade de núcleos IP envolvidos na descrição da mesma e a quantidade de canais utilizados para interconectá-los é uma atividade muito propensa a erros. Assim, além da mera descrição da rede consumir um tempo elevado, muitas vezes o processo deve ser revisado.

Para a validação da NoC Hermes TL, inicialmente foi utilizado um método simples de análise do caminhamento. Era utilizada uma representação em papel da NoC descrita e, a partir das informações impressas na tela, o caminhamento do pacote era rastreado. Assim como a tarefa anterior, não se trata de tarefa complexa, mas devido ao volume de informações esta também é uma atividade propensa a muitos erros. Por vezes, durante a execução do modelo a rede bloqueava. Encontrar o pacote que havia bloqueado na rede era uma tarefa difícil devido ao volume de informações a manipular.

Devido as estas dificuldades, considera-se importante o desenvolvimento de ferramentas que facilitem tanto a descrição esta quanto a validação da NoC Hermes TL. Foram desenvolvidas as ferramentas *nocGen* e *nocLogView*. A primeira permite descrever uma NoC Hermes TL de forma automatizada. A segunda gera relatórios a partir da execução do modelo da NoC Hermes TL. A presente Seção tem por finalidade apresentar a funcionalidade destas ferramentas.

5.2.1 Gerador de NoCs – *nocGen*

Uma NoC 6x6 de topologia malha em SystemC, cria 36 instâncias do núcleo IP chave e 2 canais de comunicação externa para cada porta da chave. Para cada instância da chave devem ser passados parâmetros específicos ou genéricos, tais como o endereço da chave na rede, o algoritmo de roteamento a ser utilizado e a dimensão total da rede. Quanto às interconexões, caso a ligação entre as chaves seja realizada de forma errada, o caminhamento pela rede fica comprometido.

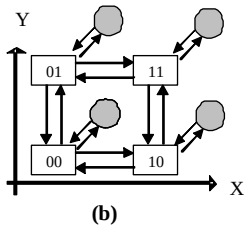
Para superar os problemas derivados de erros oriundos de um processo manual de descrição foi desenvolvida a ferramenta *nocGen*. Esta ferramenta possui duas funcionalidades. A primeira é a capacidade de geração de um arquivo intermediário contendo a informação de interconexão da rede. A segunda é a geração da NoC descrita em SystemC a partir deste arquivo intermediário. Estas duas funcionalidades são fornecidas de forma separada para evitar a limitação da ferramenta de apenas gerar uma descrição fixa da NoC Hermes TL e também para permitir o uso futuro de outras linguagens.

O arquivo intermediário provê informações sobre a interconexão da rede, podendo ser utilizado como entrada para ferramentas, de geração de descrições de redes. O arquivo intermediário é estruturado

conforme ilustrado na Figura 49(a). A primeira linha representa a definição de uma chave. A linha começa com os caracteres “C:”. A seguir são informados o nome da chave e a posição XY desta na rede. As linhas seguintes, que iniciam por “P:” representam as portas desta chave. Há três tipos de descrição diferente para as portas, quais sejam: Ou não está conectada a alguma chave, ou está conectada a outra chave ou é uma porta LOCAL. Quando a porta está não conectada a alguma chave, aparece o valor -1 na linha, como aparece na segunda linha. Quando a porta está conectada a outra chave, aparece o nome da chave a qual ela está conectada e a qual porta desta chave é realizada a conexão, exemplificado pela terceira linha. Caso a porta seja LOCAL, nenhum valor é informado visto que ela é utilizada como entrada da NoC, conforme ilustra a sexta linha da Figura 49(a). Para gerar o arquivo intermediário pela ferramenta, basta digitar o comando “`./nocgen -topology mesh -size 2 2`”. Ao executar este comando será solicitado o nome do arquivo de saída. O resultado da geração de um arquivo intermediário de uma NoC 2x2 pode ser visto na Figura 49(a), correspondendo à NoC na Figura 49(b).

```
C: chv_0x0, 0, 0
P: N, -1
P: S, chv_0x1, N
P: E, chv_1x0, W
P: W, -1
P: L
C: chv_0x1, 0, 1
P: N, chv_0x0, S
P: S, -1
P: E, chv_1x1, W
P: W, -1
P: L
C: chv_1x0, 1, 0
P: N, -1
P: S, chv_1x1, N
P: E, -1
P: W, chv_0x0, E
P: L
C: chv_1x1, 1, 1
P: N, chv_1x0, S
P: S, -1
P: E, -1
P: W, chv_0x1, E
P: L
```

(a)



(b)

Figura 49 – (a) Estrutura do arquivo intermediário. Demonstração de um arquivo intermediário descrevendo uma rede 2x2. (b) NoC correspondente.

A geração de um modelo de NoC Hermes a partir de um arquivo intermediário permite tanto que descrições manuais de interconexões das chaves sejam elaboradas como que se descreva a rede de forma automática. As informações deste arquivo intermediário são interpretadas e um modelo pronto para ser utilizado é produzido.

O uso da ferramenta *NoCGen* permite a geração da NoC Hermes em SystemC dois níveis de abstração. O primeiro nível é a descrição totalmente em nível de transação, onde a rede é composta por chaves cuja computação é descrita de forma comportamental sem ciclos de relógio e a comunicação descrita de forma abstrata através da chamada de serviços. O segundo nível é a descrição RTL. As descrições neste nível estão atualmente limitadas ao uso do algoritmo de roteamento XY puro.

Esta ferramenta opera via linha de comando, informando ao final do processamento se tudo ocorreu como esperado ou se ocorreu algum erro. Os comandos associados ao programa são apresentados na Figura 50.

```
$ ./nocGen.x
./nocGen.x [–capture] [–format[-internal/-rtl/-tl]] –size x y -topology [mesh] –algorithm [adapMin/adapNMin/purexy/wfMin/wfNMin/nlNMin/nfNMin]
  -capture: Capture intermediate format
  -format/f: NoC's description abstraction level
    internal: Generate intermediate format
    rtl: Generate SystemC regiter transfer level NoC Description
    tl: Generate SystemC transaction level NoC Description
  -size/s: NoC size
    x,y: line size and column size
  -topology/t: Define the interconection between switches
    mesh: Actually available only for mesh topology
  -algorithm/a: Define the routing algorithm
    adapMin: XY adaptive Minimal
    adapNMin: XY adaptive non Minimal
    purexy: XY pure
    wfMin: West First Minimal
    wfNMin: West First non Minimal
    nlNMin: North Last non Minimal
    nfNMin: Negative First non Minimal
```

Figura 50 – Opções de execução e parâmetros para a ferramenta nocGen. A opção " - capture" carrega um arquivo intermediário. A opção "-format" gera formato interno da descrição desejada ou descrição SystemC TL da NoC ou descrição RTL da Noc. A opção "-size" informa o número de linhas e colunas da NoC e a opção "-topology" informa a topologia a ser adotada.

O resultado obtido com esta ferramenta foi uma descrição de NoC em SystemC TL confiável e realizada em um tempo muito menor do que se o mesmo fosse produzido manualmente. A NoC Hermes TL 6x6 mencionada anteriormente foi gerada por esta ferramenta. Uma NoC Hermes TL 2x2 gerada a partir desta ferramenta é apresentada no apêndice E deste trabalho.

5.2.2 Gerador de relatórios – nocLogView

Com o objetivo cobrir o máximo de situações de tráfego e bloqueios de canais possíveis de ocorrer na NoC, um número grande de pacotes deve estar trafegando simultaneamente pela rede. Durante a execução do modelo da NoC Hermes TL, informações sobre a passagem de cada um dos pacotes em cada uma das chaves era impressa na tela. Tais pacotes eram gerados com a ferramenta *trafficGen*, que permite a definição de pacotes para destinos aleatórios ou para um destino único.

Inicialmente, as informações eram utilizadas para avaliar se o algoritmo de roteamento estava sendo executado de forma coerente. Para isto, um gráfico impresso em papel representando a NoC sob simulação, era utilizado para validar o caminho realizado pelos pacotes durante a simulação. Quando algum erro no envio de um pacote ocorria (e.g. um pacote que não chega ao destino) a detecção do primeiro pacote que ficou preso era uma tarefa complicada devido à quantidade de pacotes que já trafegavam na rede. Em uma das operações de validação utilizou-se uma NoC 6x6 e na ferramenta *trafficGen* foi definido que cada chave enviaria 50 pacotes à rede. Pressupondo que cada pacote passaria por três chaves até chegar a seu destino, a quantidade média de linhas resultante da execução deste modelo era de aproximadamente 5400 linhas (36 chaves x 50 pacotes x 3 chaves por onde passam os pacotes).

Como alternativa para superar a dificuldade encontrada com a validação em papel planilhas eletrônicas foram utilizadas como mecanismo de validação. Adicionalmente, o uso deste mecanismo permitia maior facilidade na análise de alguns parâmetros, tal como o atraso que um pacote sofre durante o caminhamento pela rede. Apesar de mais eficiente que a validação em papel, esta tarefa ainda consumia tempo considerável, chegando a ocupar 2 horas para o mapeamento de um grupo de cinco pacotes. Além deste problema, a possibilidade de erro ao rastrear um pacote permanecia elevada, o que por vezes punha em dúvida os resultados obtidos, tornando assim este um método pouco confiável de validação.

Para superar esta dificuldade na tarefa de validação foi desenvolvida a ferramenta *nocLogView*. Esta ferramenta sofreu um processo evolutivo de desenvolvimento, atendendo inicialmente à validação dos algoritmos de roteamento, através do rastreamento de pacotes que trafegaram na NoC, e a seguir gerando relatórios sobre o tráfego de pacotes. Como entrada desta ferramenta utiliza-se o mesmo arquivo empregado para validação em papel ou via planilha eletrônica. Como saída, a ferramenta apresenta um conjunto de relatórios, conforme ilustra a Figura 51, que têm como objetivo contribuir para a validação da NoC. Para chamar a ferramenta, basta digitar "`./noclogview -tl nome_do_arquivo_de_log`", e aguardar a apresentação das opções de relatório. Para simulações de grande volume de informações, o processo de geração pode levar em torno de 30 segundos.

```
------------------------
Digite a letra desejada
------------------------
  [A] - Lista chaves
  [B] - Lista pacotes
  [C] - Lista pacotes por chaves
  [D] - Rastro de pacote específico
  [E] - Relatorio
  [X] - Sair
  Opcao:
```

Figura 51 - Menu principal da ferramenta nocLogView.

A opção [A] - *Lista de chaves* informa quantos pacotes foram enviados a partir de cada chave e quantos destes pacotes chegaram ao destino, conforme ilustra a Figura 52. No final é apresentada uma sumarização de quantos pacotes foram enviados à rede e quantos chegaram ao destino. Esta informação permite identificar se a funcionalidade da NoC foi atingida, ou seja, se todos os pacotes enviados a rede foram entregues. Caso nem todos os pacotes tenham sido entregues, esta opção facilita identificar de quais chaves originaram os pacotes que não chegaram ao destino. Com isto fica mais fácil e ágil encontrar os pacotes que trancaram na rede. Na Figura 52 é apresentado o final da lista de chaveadores de uma simulação ocorrida em uma NoC 3x3.

```
Chaveador: 1x0
  Pacotes enviados : 500
  Chegaram ao destino: 500

Chaveador: 2x0
  Pacotes enviados : 500
  Chegaram ao destino: 500

Chaveador: 0x1
  Pacotes enviados : 500
  Chegaram ao destino: 500

Total de pacotes enviados: 4500
Total de pacotes que chegaram no destino: 4500
```

Figura 52 - Resultado da opção Lista de chaves.

A opção [B] - *Lista de pacotes* lista todos os pacotes que foram enviados através da rede, informando qual era seu destino e se este alcançou ou não seu destino. Não há uma relação direta entre a chave de origem com o pacote, todavia esta informação permitiu, nos primeiros estágios de validação, encontrar quais eram os destinos que estavam tendo algum problema para o recebimento de pacotes. Na Figura 53 apresenta-se o resultado final da lista de pacotes.

```
Pacote 4499
  Destino : 2x2
  Chegou: SIM

Pacote 4500
  Destino : 1x2
  Chegou: SIM

Total de pacotes enviados: 4500
Total de pacotes que chegaram no destino: 4500
```

Figura 53 - Resultado da opção Lista de pacotes.

A terceira opção do relatório, [C] - *Lista de pacotes por chave*s, permite uma visualização mais detalhada do percurso percorrido por pacotes enviados de cada uma das chaves. Durante o processo de validação da NoC houveram pacotes que não foram entregues ao destino, ou seja, ficaram presos por algum motivo durante seu caminhamento pela rede. A opção [A] da ferramenta permite visualizar de quais chaves partiram os pacotes que pararam em seu caminhamento pela rede. Na opção [C] o usuário pode detalhar informações sobre os pacotes que partiram de uma chave. Ao escolher uma chave específica, são listados todos os pacotes que foram enviados por esta, informando qual era seu destino, se alcançou o destino, por quantas chaves passou até chegar a seu destino e a latência que obteve até alcançar seu destino. Neste contexto, latência significa quantas vezes o pacote ganhou prioridade para ser roteado. Esta opção permite ainda detectar quais pacotes foram bloqueados em seu caminhamento pela rede. A Figura 54 ilustra o resultado da escolha da opção *Lista de pacotes por chaves* para a chave 0x0.

```
pacote: 498
  destino: 1x2
  chegou: SIM
  Nro de roteadores por q passou: 4
  Latencia: 27
pacote: 499
  destino: 1x0
  chegou: SIM
  Nro de roteadores por q passou: 2
  Latencia: 4
pacote: 500
  destino: 0x2
  chegou: SIM
  Nro de roteadores por q passou: 3
  Latencia: 7
```

Figura 54 - Resutado da opção Lista de pacotes por chaves.

Provavelmente a opção que mais contribuiu para detectar problemas no caminhamento de pacotes na rede tenha sido a [D] - *Rastreamento de pacote específico*. Ao detectar que um pacote havia parado na rede, utilizando a opção [A], [B] e [C], esta nova opção permite detectar mais facilmente todos os pacotes que pararam na rede. Utilizando esta opção da ferramenta foi possível detectar com maior facilidade a ocorrência de situações de *deadlock* para o algoritmo de roteamento XY adaptativo mínimo. Adicionalmente, pôde-se validar se o caminhamento de pacotes na rede estava condizente com o algoritmo de roteamento que estava sendo empregado naquele momento.

A partir do sucesso obtido com estes relatórios, foram definidos outros tipos de relatórios, que possibilitaram a medição do atraso relativo obtido com cada algoritmo de roteamento. Esta análise de atraso, está diretamente vinculada à escolha do algoritmo de roteamento. Apesar disto, sabe-se que problemas de contenção estão principalmente relacionados ao algoritmo de arbitragem adotado.

A opção [E] da ferramenta foi desenvolvida para permitir a análise das informações capturadas durante a execução do modelo da NoC Hermes TL, estendendo assim a ferramenta para uma característica adicional, a da validação. Pode-se observar na Figura 55 algumas informações capturadas durante a execução do modelo e geradas pela ferramenta.

```
Total de pacotes enviados..............4500
Latência total na rede                 111095
Ganho total de prioridade..............94595
Percentual de ganho de prioridade      85
Percentual de roteamento realizado     9%
Total de chaves utilizado pelos pacotes.12569
Número total de chaves na rede         9
Média de pacotes por chave.............500
Latência média por pacote              24
Média de chaves utilizadas por pacote...3
Latencia media por chave               8

============================================================

464 pacotes passaram por 1 chave(s) = 10%
Latência total eh                  5386
Ganho total de prioridade..........4543
Percentual de ganho de prioridade  84
Percentual de roteamento realizado.10%
Latência média por pacote          11
Latencia media por chave...........11

============================================================

1375 pacotes passaram por 2 chave(s) = 30%
Latência total eh                  26104
Ganho total de prioridade..........22556
Percentual de ganho de prioridade  86
Percentual de roteamento realizado.12%
Latência média por pacote          18
Latencia media por chave...........9

============================================================

1519 pacotes passaram por 3 chave(s) = 33%
Latência total eh                  40631
Ganho total de prioridade..........33860
Percentual de ganho de prioridade  83
Percentual de roteamento realizado.13%
Latência média por pacote          26
Latencia media por chave...........8

============================================================

912 pacotes passaram por 4 chave(s) = 20%
Latência total eh                  30242
Ganho total de prioridade..........25565
Percentual de ganho de prioridade  84
Percentual de roteamento realizado.14%
Latência média por pacote          33
Latencia media por chave...........8

============================================================

230 pacotes passaram por 5 chave(s) = 5%
Latência total eh                  8732
Ganho total de prioridade..........8071
Percentual de ganho de prioridade  92
Percentual de roteamento realizado.14%
Latência média por pacote          37
Latencia media por chave...........7
```

Figura 55 – Exemplo de relatório produzido pela opção [E] da ferramenta nocLogView.

A primeira parte deste relatório fornece informações gerais sobre o tráfego total de pacotes pela rede. A primeira linha informa quantos pacotes foram enviados através da rede. A segunda linha informa o somatório do número total de vezes que cada pacote concorreu por prioridade, independente de ter ganhado ou não. Esta informação permite avaliar, por exemplo, se o algoritmo de arbitragem que está sendo utilizado é o mais eficiente quanto ao tempo de espera de um pacote para ser roteado. A terceira linha informa o somatório sobre todos os pacotes de quantas vezes um pacote ganhou prioridade dentro de cada chave, independente de ter sido roteado. A quarta linha informa o percentual médio de vezes que

o pacote ganhou direito de acesso ao roteamento sobre o total de vezes que concorreu. No caso da Figura 55, houve em média 85% de ganho de direito de acesso, ou seja, cada pacote concorreu poucas vezes para ser atendido pelo algoritmo de roteamento, o que representa uma baixa concorrência pelo recurso de roteamento. A linha seguinte, a quinta linha, informa o percentual médio de vezes que um pacote pôde ser roteado após obter direito de acesso ao roteamento. Do universo de 94595 ganhos de direito de acesso, apenas 8514 vezes os pacotes puderam ser roteados, ou seja, 9% das vezes. A sexta linha informa o total de vezes que alguma chave utilizada durante o caminhamento dos pacotes. Este dado permite comparar os diferentes algoritmos de roteamento para avaliar a relação custo-benefício do uso de cada um destes. Para um mesmo volume de pacotes na rede, o uso de mais chaves durante o caminhamento indicar maior quantidade de bloqueio de canais e maior consumo de potência. A sétima e oitava linhas são usadas apenas como parâmetro para avaliar os demais dados. A nona linha faz uma média aritmética da quantidade de vezes que cada pacote concorreu por prioridade durante todo seu caminhamento na rede. A décima linha apresenta uma média aritmética do número de chaves utilizadas pelos pacotes em seu caminhamento na rede. Finalmente, a 11ª linha apresenta a média aritmética do número de vezes que cada pacote tentou sair de cada uma das chaves em que entrou durante seu tráfego pela rede.

Após esta primeira parte do relatório, é apresentada uma seqüência de informações separando os pacotes pelo número de chaves que passou até atingir seu destino. O primeiro conjunto, o de pacotes que passaram por apenas uma chave, não tem sentido em um sistema real, pois representam pacotes que saem de uma chave, por exemplo, a 0x0, e tem como destino a mesma chave. Como os pacotes são gerados pela ferramenta *trafficGen* e seu destino é aleatório, a situação de uma chave enviar um pacote para ela mesma é possível de ocorrer. Os demais conjuntos podem ser utilizados para distribuir de forma mais otimizada os núcleos IP pela rede de forma a diminuir o uso de chaves ou a latência. Com base nos dados capturados, iniciou-se uma investigação sobre a relação da descrição em nível de transação com descrições de mais baixo nível.

Uma descrição da NoC Hermes RTL em VHDL foi desenvolvida anteriormente a este trabalho [MOR03a]. Nesta, a ferramenta *trafficGen*[5] gera pacotes que alimentam a rede para validação da transmissão. A validação é realizada por cosimulação C/VHDL, na qual os arquivos gerados pela ferramenta *trafficGen* são lidos, e ao final da execução é gerado um arquivo de saída para cada chave com a informação do momento em que o pacote entrou na rede, a origem do pacote e o momento em que este saiu. Inicialmente, imaginou-se criar uma relação entre o arquivo de saída da cosimulação e o arquivo de resultados de execução gerado pela NoC TL. Esta abordagem permitiria comprovar que os resultados obtidos de forma abstrata poderiam prever uma relação direta entre as duas descrições, SystemC e VHDL. Infelizmente, o resultado obtido não foi o esperado. Na maioria das vezes, uma grande latência detectada na descrição em nível de transação não se verificava na descrição VHDL. O principal motivo para este problema é a diferença na arquitetura das linguagens. No momento em que um pacote de uma determinada chave entra na rede descrita em SystemC não ocorre na mesma instante que o mesmo pacote no mesmo chave na descrição VHDL. Assim, o resultado é o bloqueio de determinadas portas da chave descrita em SystemC que não necessariamente ocorre na descrição VHDL. Conseqüentemente, o atraso em SystemC não necessariamente representa o atraso em VHDL, ou pelo menos, não há uma relação direta entre a simulação SystemC e a mesma em VHDL.

Apesar disto, a ferramenta *nocLogView* contribuiu em muito para a análise e validação da NoC TL, possbiilitando a diminuição do tempo necessário para rastrear pacotes, detectar erros e situações de *deadlock* no roteamento, capturar latência média, latência total, e medir percursos médio e total.

[5] http://www.inf.pucrs.br/~gaph/homepage/download/software/maia/Maia.htm

6 Considerações finais

6.1 Resumo das contribuições do trabalho

O presente trabalho foi o primeiro ao longo do desenvolvimento da NoC Hermes a prover um modelo abstrato da NoC em nível de transação. Este é parte de um esforço de modelagem abstrata da estruturas de comunicação em SoCs, tendo em vista a crescenete complexidade da parte de comunicação neste tipo de sistemas. O trabalho aportou um conjunto de contribuições originais, cujas principais são detalhadas a seguir.

Devido à necessidade de melhor detalhamento em níveis de abstração da denominação genérica *nível de transação*, e por não haver um modelo amplamente aceito para isto na comunidade de pesquisa, foi proposto um conjunto de níveis de abstração de captura de projeto, apresentados e justificados no Capítulo 2. Os níveis de abstração propostos baseiam-se na crítica de propostas anteriores, também discutidos naquele Capítulo. Para corroborar a proposta de organização de níveis de abstração de projeto acima do RTL, conduziu-se um experimento de modelagem e implementação de um processador especificado pelo grupo GAPH, descrita no Capítulo 3. Para estas atividades utilizou-se as linguagens de descrição de hardware SystemC e VHDL. Comparou-se a área ocupada em *hardware* obtida com cada um dos modelos apresentados. Ainda naquele Capítulo, os níveis de abstração de captura de projeto TL e RTL foram comparados quanto ao desempenho de simulação.

Uma segunda contribuição do presente trabalho portou sobre a modelagem abstrata de comunicação em SoCs, usando como estudo de caso a NoC Hermes desenvolvida no escopo do grupo de pesquisa do Autor. Este estudo de caso, descrito no Capítulo 4, permitiu estabelecer um método informal de descrição de modelos abstratos para a comunicação em SoCs, que vem desde então sendo utilizado por outros membros do grupo do Autor. O modelo abstrato foi também empregado no escopo de um projeto cooperativo multi-institucional, onde desempenha o papel de meio de comunicação na descrição abstrata de uma plataforma de projeto para sistemas embarcados sem fio (plataforma Fênix, no projeto Brazil-IP). Adicionalmente, os primeiros passos foram dados aqui para se obter métodos de avaliar o desempenho da comunicação em níveis de descrição abstratos, através da proposta de técnicas de estimativa do desempenho temporal da comunicação na rede Hermes TL.

A terceira contribuição foi o desenvolvimento das ferramentas de apoio ao projeto, validação e avaliação de NoCs, descrito no Capítulo 5. Uma destas (*nocGen*) automatiza parcialmente o processo de descrição de redes intra-chip no nível de transação, enquanto uma outra ferramenta (*nocLogView*) agiliza o processo de validação e avaliação dos modelos utilizados, conforme apresentado igualmente no Capítulo 5.

Finalmente, empreendeu-se o desenvolvimento de núcleos IP que convertem comunicação abstrata para comunicação detalhada compatível com o protocolo padronizado OCP, conforme descrito no Capítulo 5. Estes núcleos são específicos para o estudo de caso de NoC abordado, mas o domínio de sua implementação é visto como uma contribuição importante. Isto se deve ao fato de que o desenvolvimento dos transatores facilita o reuso de projeto e a validação mista de núcleos descritos nos níveis RTL e TL que venham a utilizar a NoC Hermes TL como mecanismo de comunicação. Estes transatores foram também utilizados na plataforma Fênix do projeto Brazil-IP citado anteriormente.

6.2 Conclusões e trabalhos futuros

Pôde-se observar que não existe uma proposta de organização de níveis de abstração acima do RTL universalmente aceita ou adequada para todo e qualquer contexto de projeto de SoC. Julgou-se necessário definir um conjunto de níveis de abstração superiores ao RTL que fizessem uso da separação de aspectso de computação e comunicação para facilitar a modelagem. Como trabalho futuro, sugere-se a caracterização mais detalhada destes níveis de abstração seguida da deinição de fluxos de projeto coerentes baseados em uma seqüência destes níveis como processo de refinamento ao longo do projeto de SoCs.

Dado a existência de um esfroço de automatização do processo de geração da comunicação através de redes intra-chip no grupo do Autor, concretizado através do arcabouço MAIA [OST04, OST05], considera-se importante como trabalho futuro a integração das funcionalidades das ferramentas nocGen e nocLogView à este arcabouço. Esta atividade habilitaria o arcabouço MAIA a gerar e validar estruturas de comunicação descritas no nível TL, uma vez que hoje este está limitado à geração/validação de descrições no nível RTL. Como benefício no sentido oposto, ou seja, do arcabouço MAIA para o trabalho com níveis asbtratos de projeto, ter-se-á a agregação da capacidade de geração de tipos de tráfego complexos, uma característica de relevância no processo de validação de NoCs.

Este trabalho modelou a NoC Hermes no nível de transação. Tal NoC foi validada funcionalmente através do uso das ferramentas de apoio desenvolvidas especificamente para este fim. Adicionalmente é permitida a parametrização da rede quanto ao dimensionamento e algoritmos de roteamento, através da ferramenta *nocGen*. Contudo, não foram obtidos dados quantitativos detalhados, tais como a latência estimada ou o *throughput* da rede para tráfegos dados, sejam estes realistas ou não. Desta forma, a obtenção de tais dados e sua análise constitui uma interessante sugestão para trabalho futuro. Além disto, divisa-se como relevante estender a capacidade de geração/validação de NoCs para outras topologias (e.g. toro, hipercubo, e árvore gorda), outras estratégias de controle de fluxo (baseada em créditos), outras estratégias de armazenamento (filas centralizadas ou filas de saída ou abordagens mistas) e outros algoritmos de roteamento.

Finalmente, foi desconsiderada aqui a implementação de suporte à qualidade de serviço (em infglês, *quality of service* ou QoS) em NoCs. Como trabalho futuro, necessário a viabilizar a adoção de NoCs em sistemas de alcance comercial, é importante poder gerar NoCs com garantia de QoS em diversos níveis. Todavia, um trabalho anterior a este deve ser a definição de quais as formas de QoS implementar, como modelar QoS em níveis de abstratos. Esta tarefa envolverá necessáriamente análise de como descrever abstratamente sistemas usando chaveamento de circuito e/ou canais virtuais.

7 Referências

[ACC05] Accellera Organization Inc. **"SystemVerilog 3.1a - Language Reference Manual"**. Capturado em: http://www.accellera.org/SystemVerilog_3.1a_LRM.pdf, 2005.

[AND03a] Andriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L. & Zeferino, C. **"SPIN: a Scalable, Packet Switched, on Chip Micro-Network"**. Em: Design Automation and Test in Europe - DATE'03, 2003, pp. 70 –73.

[AND03b] Andriahantenaina, A. & Greiner, A. **"Micro-Network for SoC: Implementation of a 32-Port SPIN Network"**. Em: Design Automation and Test in Europe - DATE'03, 2003, pp. 1128-1129.

[ARM05] ARM Corporation. **"AMBA 2.0 Specification"**. Capturado em: http://www.arm.com/amba_2/IHI0011A_AMBA_SPEC.pdf, 2005.

[ARN00] Arnout, G. **"SystemC Standard"**. Em: Asia and South Pacific Design Automation Conference - ASP-DAC 2000, 2000, pp. 573-577.

[ARN02] Arnout, G. **"EDA Moving Up, Again!"** Em: 6th European SystemC Users Group, 2002, 6 p.

[ASH98] Ashra, F. **"Introduction to Routing in Multicomputer Networks",** ACM Computer Architecture News, vol. 26 (5), Dec. 1998, pp. 14-21.

[ASH99] Ashenden, P. J., Wilsey, P. A. & Martin, D. E. **"SUAVE Language Description"**. Technical Report, Department of Computer Science - University of Adelaide, 1999, 60 p. Capturado em: http://www.ashenden.com.au/suave/suave-descr-july-1999.pdf.

[BAI01] Bailey, B. & Gajski, D. **"RTL Semantics and Methodology"**. Em: International Symposium on System Synthesis - ISSS'01, 2001, pp. 69-74.

[BEN01] Benini, L. & Micheli, G. D. **"Powering Networks on Chip"**. Em: International Symposium on System Synthesis - ISSS'01, 2001, pp. 33 -38.

[CAI03a] Cai, L. & Gajski, D. **"Transaction Level Modeling in System Level Design"**. Technical Report, University of California at Irvine, 2003, 23 p. Capturado em: http://www.ics.uci.edu/~cecs/technical_report/TR03-10.pdf.

[CAI03b] Cai, L., Gajski, D. & Verma, S. **"Comparison of SpecC and SystemC Languages for System Design"**. Technical Report, University of California at Irvine, 2003, 30 p. Capturado em: http://www.ics.uci.edu/~cecs/technical_report/TR03-11.pdf.

[CAL03] Calazans, N. L. V., Moreno, E., Hessel, F., Rosa, V. & Moraes, F. G. **"From VHDL Register Transfer Level to SystemC Transaction Level Modeling: a Comparative Case Study"**. Em: 16th Symposium on Integrated Circuits and System Design - SBCCI'03, 2003, 6 p.

[CAL98] Calazans, N. L. V. **"Projeto Lógico Automatizado de Sistemas Digitais Seqüenciais"**. 1st ed., Rio de Janeiro, Imprinta Gráfica e Editora Ltda, 1998, 346 p.

[CES02] Cesário, W. O., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S. & Jerraya, A. A. **"Multiprocessor SoC Platforms: A Component-Based Design Approach"**. IEEE Design & Test of Computers, vol. 19 (6), Nov.-Dec. 2002, pp. 52-63.

[DAL01] Dally, W. & Towles, B. **"Route Packets, Not Wires: on Chip Interconnection Networks"**. Em: 38th Design Automation Conference - DAC'01, 2001, pp. 684-689.

[DON04] Davis, D. **"Forge-J: High Performance Hardware from Java"**. White paper, Xilinx, Inc, 2004, 5 p. Capturado em: http://www.xilinx.com/ise/advanced/forge_java.pdf.

[DUA02] Duato, J. **"Interconnection Networks"**. revised ed., Morgan Kaufmann, 2002, 624 p.

[DZI03] Dziri, M. A., Samet, F., Wagner, F. R., Cesario, W. O. & Jerraya, A. A. **"Combining Architecture Exploration and a Path to Implementation to Build a Complete SoC Design Flow from System Specification to RTL"**. Em: Asia and South Pacific Design Automation Conference - ASP-DAC 2003, 2003, 6 p.

[EDW97] Edwards, S., Lavagno, L., Lee, E. A. & Vincentelli, A. S. **"Design of Embedded Systems: Formal Models, Validation and Synthesis"**. Em: Proceedings of the IEEE, March 1997, pp 366-390.

[FAY00] Fayad, F. & Khordoc, K. **"An Object-Oriented Refinement Methodology Through the Design of a Settop-Box"**. Em: Canadian Conference on Electrical and Computer Engineering, 2000, pp. 1032 -1036.

[FIT04] Fitzpatrick, T. **"SystemVerilog for VHDL Users"**. Em: Design Automation and Test in Europe - DATE'04, 2004, 6 p.

[FOR02] Forsell, M. **"Scalable High-Performance Computing Solution for Networks on Chips"**. IEEE Micro, vol. 22 (5), Sep.-Oct. 2002, pp. 46-55.

[FUJ01] Fujita, M. & Nakamura, H. **"The Standard SpecC Language"**. Em: 14th International Symposium on System Synthesis -ISSS'01, 2001, pp. 5.4.1-5.4.6.

[GAJ94] Gajski, D., Vahid, F., Narayan, S. & Gong, J. **"Specification and Design of Embedded Systems"**. 1st ed., Upper Saddle River, NJ, Prentice Hall, 1994, 450 p.

[GLA94] Glass, C. & Ni, L. M. **"The Turn Model for Adaptive Routing"**. Journal of Association for Computing Machinery, vol. 41 (5), Sep. 1994, pp.874-902.

[GRÖ02] Grötker, T., Liao, S., Martin, G. & Swan, S. **"System Design with SystemC".** 1st ed., Boston, Kluwer Academic Publishers, 2002, 219 p.

[GUE00] Guerrier, P. & Greiner, A. **"A Generic Architecture for on Chip Packet-Switched Interconnections"**. Em: Design Automation and Test in Europe - DATE'00, 2000, pp.250-256.

[GUP03] Gupta, S., Dutt, N. D., Gupta, R. K. & Nicolau, A. **"SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations"**. Em: International Conference on VLSI Design, VLSI'03, 2003, 6 p.

[GUP97] Gupta, R. K. & Zorian, Y. **"Introducing Core-Base System Design"**. IEEE Design and Test of Computers, vol. 14 (4), October-December 1997, pp. 15-25.

[HAV02] Haverinen, A., Leclercq, M., Weyrich, N. & Wingard, D. **"SystemC Based SoC Communication Modeling for OCP Protocol"**. White paper, 2002, 39 p. Capturado em: http://www.ocpip.org/data/ocpip_wp_SystemC_Communication_Modeling_2002.pdf.

[HEN96] Hennessy, J. & Patterson, D. **"Computer Architecture: A Quantitative Approach"**. 1st ed., San Francisco, Morgan Kaufmann, 1996, 760 p.

[HWA92] Hwang, K. **"Advanced Computer Architecture: Parallelism, Scalability, Programmability"**. 1st ed., New York, McGraw-Hill, 1992, 672 p.

[IBM02] IBM Inc. **"The CoreConnectTM Bus Architecture"**. Capturado em: http://www3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.pdf, 2002.

[JER01] Jerraya, A. A., Svarstad, K., Ben-Fredj, N. & Nicolescu, G. **"A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems"**. Em: Asia and South Pacific Design Automation Conference - ASP-DAC 2001, 2001, pp. 69-77.

[JIA00] Jian, L., Swaminathan, S. & Tessier, R. **"aSOC: A Scalable, Single-Chip Communications Architecture"**. Em: IEEE International Conference on Parallel Architectures and Compilation Techniques, 2000, pp. 37-46.

[KAR01] Karim, F., Nguyen, A., Dey, S. & Rao, R. **"On Chip Communication Architecture for OC-768 Network Processors"**. Em: 38th Design Automation Conference - DAC'01, 2001, pp. 678-683.

[KAR02] Karim, F., Nguyen, A. & Dey, S. **"An Interconnect Architecture for Network Systems on Chip"**. IEEE Micro, vol. 22 (5), Sep-Oct 2002, pp.36-45.

[KEU00] Keutzer, K. F., Malik, S., Newton, A. R., Rabaey, J. M. & Vincentelli, A. S. **"System-Level Design: Orthogonalization of Concerns and Platform-Based Design"**. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, vol. 19 (12),2000, pp 1523-1543.

[KUM02] Kumar, S., Jantsch, A., Soininen, J. P. & Fonsell, M. **"A Network on Chip Architecture and Design Methodology"**. Em: Proceedings of the IEEE Computer Society Annual Symposium on VLSI, ISVLSI'02, 2002, 8 p.

[KUN03] Kunkel, J. **"Embedded Computing - Toward IP-Based System-Level SoC Design"**. Computer, vol. 36 (5), May 2003, pp. 88 - 89.

[MAR01a] Martin, G. & Chang, H. **"Tutorial 2 - System on Chip Design"**. Em: 9th International Symposium on Integrated Circuits, Devices & Systems, ISIC'01, 2001, 6 p.

[MAR01b] Marcon, C. **"Modelagem de Sistemas de Telecomunicação para o Projeto Integrado de Hardware e Software"**. Trabalho de individual II, PPGCC - FACIN - PUCRS, 2001, 136 p. Capturado em: http://www.inf.pucrs.br/~gaph/documents/marcon_ti_2.pdf.

[MAR02] Marescaux, T., Bartic, A., Verkest, D., Vernalde, S. & Lauwereins, R. **"Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs"**. Em: Field-Programmable Logic and Applications, FPL'02, 2002, pp. 795-805.

[MIC01] Micheli, G. D., Ernst, R. & Wolf, W. **"Readings in Hardware/Software Co-Design"**. 1st ed., San Francisco, Morgan Kaufmann, 2001, 697 p.

[MIC02] Micheli, G. D. & Benini, L. **"Network on Chips: A New SoC Paradigm"**. IEEE Computer, vol. 35 (1), Jan. 2002, pp. 70-78.

[MOR02] Moreno, E. **"Modelagem, Descrição e Validação de Hardware em Diferentes Níveis de Abstração usando SystemC"**. Trabalho de individual II, PPGCC - FACIN - PUCRS, 2002, 33 p. Capturado em: http://www.inf.pucrs.br/~gaph/documents/emoreno_ti_2.pdf.

[MOR03a] Moraes, F. G., Calazans, N. L. V., Mello, A. V., Möller, L. H. & Ost, L. C. **"HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip"**. Technical report, PPGCC - FACIN - PUCRS, 2003, 26 p. Capturado em: http://www.inf.pucrs.br/tr/tr034.pdf.

[MOR03b] Moraes, F. G. & Calazans, N. L. V. **"R8 Processor Specification and Design Guidelines"**. white paper, PPGCC - FACIN - PUCRS, 2003, Capturado em: http://www.inf.pucrs.br/~gaph.

[MOR04] Moraes, F. G., Calazans, N. L. V., Mello, A. V., Möller, L. & Ost, L. C. **"An Infrastructure for Low Area Overhead Packet-switching Networks on Chip"**. The VLSI Journal Integration, Elsevier, NV., vol. 38 (1),2004, pp. 69-93.

[MOR98] Moreira, J. E., Midkiff, S. P. & Gupta, M. **"A Comparison of Java, C/C++, and FORTRAN for Numerical Computing"**. Antenas and Propagation Magazine, vol. 40 (5),1998, p. 102-105.

[MOU03] Moussa, I., Grellier, T. & Nguyen, G. **"Exploring SW Performance using SoC Transaction-Level Modeling"**. Em: Design Automation and Test in Europe - DATE'03, 2003, pp.120 - 125.

[NI93] Ni, L. **"A Survey of Wormhole Routing Techniques in Direct Networks"**. IEEE Computer, vol. 26 (2), Feb. 1993, pp. 63-76.

[OCP05a] OCP-IP. **"Open Core Specification. Versão 1.0"**. Capturado em: http://www.ocpip.org/members/ocpspec/OCP-IP_OpenCoreProtocolSpecification-1.0.zip, 2005.

[OCP05b] OCP-IP. **"The Importance of Sockets in SoC Design"**. Capturado em: http://www.ocpip.org/data/sockets_socdesign.pdf, 2005.

[OST04] Ost, L. **"Redes Intrachip Parametrizáveis com Interface Padrão para Síntese em Hardware"**. Dissertação de mestrado, PPGCC - FACIN - PUCRS, 2004, 120 p. Capturado em: http://www.inf.pucrs.br/~moraes/papers/diss_ost.pdf.

[OST05] Ost, L., Mello, A. V., Palma, J. C. S. a., Moraes, F. G. & Calazans, N. L. V. **"MAIA - A framework for Network-on-chip"**. Em: Asia and South Pacific Design Automation Conference - ASP-DAC 2005, 2005

[PAN03] Pande, P., Grecu, C., Ivanov, A. & Saleh, R. **"Design of a Switch for Network on Chip Applications"**. Em: International Symposium on Circuits and Systems - ISCAS'03, 2003, pp.217-220.

[PAS02] Pasricha, S. **"Transaction Level Modeling of SoC with SystemC 2.0"**. Em: Synopsys Users Group Conference, SNUG, 2002, 5 p.

[RAD98] Radetzki, M., Putzke-Röming, W. & Nebel, W. **"Objective VHDL Language Change Specification Version 1.2"**. Language Specification, OFFIS, 1998, 41 p. Capturado em: http://www.eda.org/oovhdl/lcs.frm.ps.gz.

[RAJ00] Rajsuman, R. **"System-on-a-Chip - Design and Test"**. 1st ed., Santa Clara, Artech House, 2000, 277.

[RAU94] Rauscher, T. G. **"Time to Market Problems - The Organization is the Real Cause"**. Em: Proceedings of the IEEE International Engineering Management Conference, 1994, pp. 338-345.

[RIJ01] Rijpkema, E., Goossens, K. & Wielage, P. **"A Router Architecture for Networks on Silicon"**. Em: 2nd Workshop on Embedded Systems - PROGRESS'2001, 2001, pp.181-188.

[RIJ03] Rijpkema, E., Goossens, K. & Rãdulescu, A. **"Trade Offs in the Design of a Router with Both Guaranted and Best effort Services for Networks-on-Chip"**. Em: Design Automation and Test in Europe - DATE'03, 2003, pp.350-355.

[SAA02] Saastamoinen, I., Alho, M., Pirttimäki, J. & Nurmi, J. **"Proteo Interconnect IPs for Network-on-chip"**. Em: IP Based SoC Design, 2002.

[SAA03] Saastamoinen, I., Alho, M. & Nurmi, J. **"Buffer Implementation for Proteo Network-on-Chip"**. Em: International Symposium on Circuits and Systems, ISCAS'03, 2003

[SCH97] Schaller, R. R. **"Moore's Law: Past, Present and Future"**. Em: IEEE Spectrum, June 1997, pp. 52-59.

[SGR01] Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J. & Sangiovanni-Vincentelli, A. **"Addressing the System-on-Chip Interconnect Woes Through Communication-Based Design"**. Em: 38th Design Automation Conference, DAC'01, 2001, pp. 667-672.

[SIG02] Sigüenza-Tortosa, D. & Nurmi, J. **"Proteo: A New Approach to Network-on-Chip"**. Em: IASTED International Conference on Communication Systems and Networks, CSN'02, 2002.

[SIL97] Silberschatz, A. & Galvin, P. B. **"Operating Systems Concepts"**. 5th ed., New York, Addison Wesley, 1997, 888 p.

[SYN02a] Synopsys, Inc. **"CoCentric System Studio Enables Verification at Multiple Levels of Abstraction with SystemC"**. Capturado em: http://www.synopsys.com/products/cocentric_studio/cocentric_studio_wp.pdf, 2002.

[SYN02b] Synopsys, Inc; CoWare, Inc; Fujitsu Microeletronics, Inc; Motorola; ST Microeletronics; Cadence Design Systems, Inc. **"Functional Specification for SystemC 2.0, Version 2.0-Q"**. Capturado em: http://www.systemc.org, 2002.

[SYN02c] Synopsys, Inc. **"SystemC Version 2.0 User's Guide - Updated for SystemC 2.0.1"**. Capturado em: http://www.systemc.org, 2002.

[SYN03a] Synopsys, Inc. **"CoCentric System Studio - User Guide Version U-2003.03"**. Capturado em: http://www.synopsys.com, 2003.

[SYN03b] Synopsys, Inc. **"Getting Started With CoCentric System Studio Version U-2003.03"**. Capturado em: http://www.synopsys.com, 2003.

[TAN96] Tanenbaum, A. S. **"Computer Networks"**. 3rd ed., Upper Saddle River, Prentice-Hall Inc, 1996, 813 p.

[WES94] Weste, N. & Eshragian, K. **"Principles of CMOS VLSI Design: a Systems Perspective"**. 2nd ed., Reading, Addison-Wesley, 1994, 713 p.

[WIN01] Wingard, D. **"Micro-Network Based Integration for SoCs"**. Em: 38th Design Automation Conference, DAC'01, 2001, pp. 673-677.

[ZEF02] Zeferino, C. A., Kreutz, M. E., Carro, L. & Susin, A. A. **"A Study on Communication Issues for System-on-Chip"**. Em: 15th Symposium on Integrated Circuits and System Design, SBCCI'02, 2002, pp. 121-126.

[ZHU02] Zhu, Q., Matsuda, A., Shoji, M., Kuwamura, S. & Nakata, T. **"An Object-Oriented Design Process for System-on-Chip Using UML"**. Em: 15th International Symposium on System Synthesis, ISSS, 2002, pp. 249-254.

8 Apêndices

8.1 Apêndice A – Algoritmos de roteamento

```
#ifndef _MESH_H
#define _MESH_H

#include "systemc.h"
#include "doorDefs.h"

//**************************************
//* FORMATO DOS ARQUIVOS DE ROTEAMENTO
//* ----------------------------------
//*   PARAMETROS:
//*     1) LINHA DE DESTINO
//*     2) COLUNA DE DESTINO
//*     3) LINHA DO ROTEADOR
//*     4) COLUNA DO ROTEADOR
//*     5) NRO DE LINHAS DA NOC
//*     6) NRO DE COLUNAS DA NOC
//*     7) VETOR DAS PORTAS LOCADAS
//*     8) PORTA Q ESTA REQUISITANDO ROTEAMENTO
//*
//**************************************

//
//*****************************************************************************
//
int XYPuro(
  FLITTYPE destAddressParam,
  FLITTYPE localAddressParam,
  FLITTYPE nocSizeParam,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddressParam.range(7,4).to_uint();
  sc_uint<4> destY=destAddressParam.range(3,0).to_uint();
  sc_uint<4> localX=localAddressParam.range(7,4).to_uint();
  sc_uint<4> localY=localAddressParam.range(3,0).to_uint();
  sc_uint<4> tamX=nocSizeParam.range(7,4).to_uint();
  sc_uint<4> tamY=nocSizeParam.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if(destY==localY){
    if     ((destX==localX)&&(outputPorts[LOCAL]==FREEPORT)) fwdPort=LOCAL;
    else if((destX>localX) &&(outputPorts[EAST]==FREEPORT))       fwdPort=EAST;
    else if((destX<localX) &&(outputPorts[WEST]==FREEPORT))       fwdPort=WEST;
  }
  else{
    if     ((destY>localY)&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
    else if((destY<localY)&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
  }

  return fwdPort;
};
```

```
//
//*************************************************************************
//
int XYAdaptativoMinimo(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if(destY==localY){
    if     (destX==localX){
      if   (outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
    }
    else if((destX>localX)&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
    else if((destX<localX)&&(outputPorts[WEST]==FREEPORT)) fwdPort=WEST;
  }
  else{
    if     ((destX==localX)&&(destY<localY)&&(outputPorts[NORTH]==FREEPORT))
fwdPort=NORTH;
    else if((destX==localX)&&(destY>localY)&&(outputPorts[SOUTH]==FREEPORT))
fwdPort=SOUTH;
    else if((destX>localX)&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
    else if((destX<localX)&&(outputPorts[WEST]==FREEPORT)) fwdPort=WEST;
  }

  return fwdPort;
};

//
//*************************************************************************
//
int XYAdaptativoNaoMinimo(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if(destY==localY){
    if(destX==localX){
      if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
```

```
    }

    if     ((destX<localX)  &&(requestingPort!=WEST)
&&(outputPorts[WEST]==FREEPORT))  fwdPort=WEST;
    else if((destX>localX)  &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT))  fwdPort=EAST;
    else if((destX!=localX) &&(requestingPort!=NORTH) &&(localY>0)
&&(outputPorts[NORTH]==FREEPORT))  fwdPort=NORTH;
    else if((destX!=localX) &&(requestingPort!=SOUTH) &&(localY<(tamY-1))
&&(outputPorts[SOUTH]==FREEPORT))  fwdPort=SOUTH;

  }
  else{
    if     ((destX==localX)&&(destY<localY)&&(requestingPort!=NORTH)
&&(outputPorts[NORTH] ==FREEPORT)) fwdPort=NORTH;
    else if((destX==localX)&&(destY<localY)&&(requestingPort!=EAST)
&&(localX<(tamX-1))&&(outputPorts[EAST] ==FREEPORT)) fwdPort=EAST;
    else if((destX==localX)&&(destY<localY)&&(requestingPort!=WEST)  &&(localX>0)
&&(outputPorts[WEST] ==FREEPORT)) fwdPort=WEST;

    if     ((destX==localX)&&(destY>localY)&&(requestingPort!=SOUTH)
&&(outputPorts[SOUTH] ==FREEPORT)) fwdPort=SOUTH;
    else if((destX==localX)&&(destY>localY)&&(requestingPort!=EAST)
&&(localX<(tamX-1))&&(outputPorts[EAST] ==FREEPORT)) fwdPort=EAST;
    else if((destX==localX)&&(destY>localY)&&(requestingPort!=WEST)  &&(localX>0)
&&(outputPorts[WEST] ==FREEPORT)) fwdPort=WEST;

    if     ((destX>localX) &&(requestingPort!=EAST) &&(outputPorts[EAST]
==FREEPORT)) fwdPort=EAST;
    else if((destX>localX) &&(destY<localY) &&(localY>0)
&&(requestingPort!=NORTH) &&(outputPorts[NORTH] ==FREEPORT)) fwdPort=NORTH;
    else if((destX>localX) &&(destY>localY) &&(localY<(tamX-1))
&&(requestingPort!=SOUTH) &&(outputPorts[SOUTH] ==FREEPORT)) fwdPort=SOUTH;

    if     ((destX<localX) &&(requestingPort!=WEST) &&(outputPorts[WEST]
==FREEPORT)) fwdPort=WEST;
    else if((destX<localX) &&(destY<localY) &&(localY>0)
&&(requestingPort!=NORTH) &&(outputPorts[NORTH] ==FREEPORT)) fwdPort=NORTH;
    else if((destX<localX) &&(destY>localY) &&(localY<(tamX-1))
&&(requestingPort!=SOUTH) &&(outputPorts[SOUTH] ==FREEPORT)) fwdPort=SOUTH;
  }

  return fwdPort;
};

//
//**************************************************************************
//
int westFirstMinimo(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
```

```
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if((requestingPort==EAST)||(requestingPort==LOCAL)){

    if(destX<localX){
      if(outputPorts[WEST]==FREEPORT) fwdPort=WEST;
    }
    else{
      if(destX==localX){
        if(destY==localY){
          if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
        }
        else if((destY<localY)&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
        else if((destY>localY)&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
      }
      else{
        if     ((destY==localY)&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
        else if((destY<localY) &&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
        else if((destY>localY) &&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
      }
    }
  }
  else{

    if(destY==localY){

      if(destX==localX){
        if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
      }

      if     ((destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
      else if((destX>localX) &&(localY>0)           &&(requestingPort!=NORTH)
&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
      else if((destX>localX) &&(localY<(tamX-1)) &&(requestingPort!=SOUTH)
&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

      if     (destX<localX){
        cout << "ERRO NO ROTEAMENTO WEST FIRST MINIMO: Tentanto ir para oeste após
finalizado o caminhamento para oeste." << endl;
      }
    }
    else{

      if     ((destY<localY)
&&(requestingPort!=NORTH)&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
      else if((destY<localY) &&(destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;

      if     ((destY>localY)
&&(requestingPort!=SOUTH)&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
      else if((destY>localY) &&(destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
    }
  }

  return fwdPort;
```

```
}

//
//**************************************************************************************
//
int westFirstNaoMinimo(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if((requestingPort==EAST)||(requestingPort==LOCAL)){

    if(destX<localX){
      if(outputPorts[WEST]==FREEPORT) fwdPort=WEST;
    }
    else{
      if(destX==localX){
        if(destY==localY){
          if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
        }

        if     ((destY<localY)&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
        else if((destY>localY)&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
        else if((localX>0)      &&(outputPorts[WEST] ==FREEPORT)) fwdPort=WEST;

      }
      else{

        if     ((requestingPort!=EAST)  &&(outputPorts[EAST]==FREEPORT))
fwdPort=EAST;
        else if((destY==localY) &&(localY>0) &&(requestingPort!=NORTH)
&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
        else if((destY==localY) &&(localY<(tamY-1)) &&(requestingPort!=SOUTH)
&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
        else if((destY==localY) &&(localX>0) &&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;

        else if((destY<localY)  &&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;

        else if((destY>localY)  &&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

      }
    }

  }
  else{

    if(destY==localY){
```

```
      if(destX==localX){
        if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
        else fwdPort=FREEPORT;
      }

      if     ((destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
      else if((destX>localX) &&(localY>0)           &&(requestingPort!=NORTH)
&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
      else if((destX>localX) &&(localY<(tamX-1)) &&(requestingPort!=SOUTH)
&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

      if     (destX<localX){
        cout << "ERRO NO ROTEAMENTO WEST FIRST NAO MINIMO: Tentanto ir para oeste
após finalizado o caminhamento para oeste." << endl;
      }
    }
    else{

      if     ((destY<localY)
&&(requestingPort!=NORTH)&&(outputPorts[NORTH]==FREEPORT)) fwdPort=NORTH;
      else if((destY<localY) &&(destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;

      if     ((destY>localY)
&&(requestingPort!=SOUTH)&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
      else if((destY>localY) &&(destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
    }

  }

  return fwdPort;

}

//
//**************************************************************************
//
int northLast(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if(requestingPort==SOUTH){

    if(destY!=localY){
      if(outputPorts[NORTH]==FREEPORT) fwdPort=NORTH;
```

```
    }
    else{
      if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
    }

  }
  else{

    if(destY==localY){
      if(destX==localX){
        if(outputPorts[LOCAL]==FREEPORT) fwdPort=LOCAL;
      }

      if     ((destX<localX) &&(requestingPort!=WEST)
&&(outputPorts[WEST]==FREEPORT)) fwdPort=WEST;
      else if((destX>localX) &&(requestingPort!=EAST)
&&(outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
      else if((requestingPort!=SOUTH) &&(localY<(tamY-1))
&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

    }
    else{

      if     ((destY>localY)  &&(requestingPort!=SOUTH)
&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

      else if((destX>localX) &&(requestingPort!=EAST)  &&(localX<(tamX-1))
&&(outputPorts[EAST]  ==FREEPORT)) fwdPort=EAST;
      else if((destX>localX) &&(requestingPort!=SOUTH) &&(localY<(tamY-1))
&&(outputPorts[SOUTH] ==FREEPORT)) fwdPort=SOUTH;
      else if((destX>localX) &&(requestingPort!=WEST)  &&(localX>0)  &&
(localY<(tamY-1)) &&(outputPorts[WEST]  ==FREEPORT)) fwdPort=WEST;

      else if((destX<localX) &&(requestingPort!=WEST)  &&(outputPorts[WEST]
==FREEPORT)) fwdPort=WEST;
      else if((destX<localX) &&(requestingPort!=SOUTH) &&(localY<(tamY-1))
&&(outputPorts[SOUTH] ==FREEPORT)) fwdPort=SOUTH;
      else if((destX<localX) &&(requestingPort!=EAST)  &&(localX<(tamX-1))
&&(localY<(tamY-1))  &&(outputPorts[EAST]  ==FREEPORT)) fwdPort=EAST;

      else if((destX==localX) &&(destY>localY) &&(outputPorts[SOUTH]==FREEPORT))
fwdPort=SOUTH;
      else if((destX==localX) &&(destY>localY) &&(localX<(tamX-1))
&&(requestingPort!=EAST) && (outputPorts[EAST]==FREEPORT)) fwdPort=EAST;
      else if((destX==localX) &&(destY>localY) &&(localX>0) &&(requestingPort!=WEST)
&&(outputPorts[WEST]==FREEPORT)) fwdPort=WEST;
      else if((destX==localX) &&(destY<localY) &&(outputPorts[NORTH]==FREEPORT))
fwdPort=NORTH;

    }

  }

  return fwdPort;

}

//
//*****************************************************************************
```

```
//
int negativeFirst(
  FLITTYPE destAddress,
  FLITTYPE localAddress,
  FLITTYPE nocSize,
  int* outputPorts, int requestingPort){

  sc_uint<4> destX=destAddress.range(7,4).to_uint();
  sc_uint<4> destY=destAddress.range(3,0).to_uint();
  sc_uint<4> localX=localAddress.range(7,4).to_uint();
  sc_uint<4> localY=localAddress.range(3,0).to_uint();
  sc_uint<4> tamX=nocSize.range(7,4).to_uint();
  sc_uint<4> tamY=nocSize.range(3,0).to_uint();

  int fwdPort=FREEPORT;

  if((requestingPort==EAST)&&(requestingPort==NORTH)&&(requestingPort==LOCAL)){

    //*****************************************************
    //* MESMA LINHA
    //*****************************************************
    if     ((destY==localY)&&(destX==localX)&&(outputPorts[LOCAL]==FREEPORT))
fwdPort=LOCAL;

    else if((destY==localY)&&(destX<localX)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    else if((destY==localY)&&(destX<localX)&&(localY<(tamY-
1))&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

    else
if((destY==localY)&&(destX>localX)&&(requestingPort!=EAST)&&(outputPorts[EAST]==FREE
PORT)) fwdPort=EAST;
    else if((destY==localY)&&(destX>localX)&&(localY<(tamY-
1))&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
    else
if((destY==localY)&&(destX>localX)&&(localX>0)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    //*****************************************************

    //*****************************************************
    //* LINHA ACIMA
    //*****************************************************
    else
if((destY<localY)&&(destX==localX)&&(requestingPort!=NORTH)&&(outputPorts[NORTH]==FR
EEPORT)) fwdPort=NORTH;
    else
if((destY<localY)&&(destX==localX)&&(localX>0)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    else if((destY<localY)&&(destX==localX)&&(localY<(tamY-
1))&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

    else if((destY<localY)&&(destX<localX)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    else if((destY<localY)&&(destX<localX)&&(localY<(tamY-
1))&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;

    else
if((destY<localY)&&(destX>localX)&&(requestingPort!=NORTH)&&(outputPorts[NORTH]==FRE
EPORT)) fwdPort=NORTH;
```

```
    else
if((destY<localY)&&(destX>localX)&&(requestingPort!=EAST)&&(outputPorts[EAST]==FREEP
ORT)) fwdPort=EAST;
    else if((destY<localY)&&(destX>localX)&&(localY<(tamY-
1))&&(outputPorts[SOUTH]==FREEPORT)) fwdPort=SOUTH;
    else
if((destY<localY)&&(destX>localX)&&(localX>0)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    //*****************************************************

    //*****************************************************
    //* LINHA ABAIXO
    //*****************************************************
    else if((destY>localY)&&(destX==localX)&&(outputPorts[SOUTH]==FREEPORT))
fwdPort=SOUTH;
    else
if((destY>localY)&&(destX==localX)&&(localX>0)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;

    else if((destY>localY)&&(destX<localX)&&(outputPorts[SOUTH]==FREEPORT))
fwdPort=SOUTH;
    else if((destY>localY)&&(destX<localX)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;

    else if((destY>localY)&&(destX>localX)&&(outputPorts[SOUTH]==FREEPORT))
fwdPort=SOUTH;
    else
if((destY>localY)&&(destX>localX)&&(localX>0)&&(outputPorts[WEST]==FREEPORT))
fwdPort=WEST;
    //*****************************************************

  }
  else{
    //*****************************************************
    //* MESMA LINHA
    //*****************************************************
    if     ((destY==localY)&&(destX==localX)&&(outputPorts[LOCAL]==FREEPORT))
fwdPort=LOCAL;

    else if((destY==localY)&&(destX>localX)&&(outputPorts[EAST]==FREEPORT))
fwdPort=EAST;
    //*****************************************************

    //*****************************************************
    //* LINHA ACIMA
    //*****************************************************
    else if((destY<localY)&&(destX==localX)&&(outputPorts[NORTH]==FREEPORT))
fwdPort=NORTH;

    else if((destY<localY)&&(destX>localX)&&(outputPorts[NORTH]==FREEPORT))
fwdPort=NORTH;
    else if((destY<localY)&&(destX>localX)&&(outputPorts[EAST]==FREEPORT))
fwdPort=EAST;
    //*****************************************************

    //*****************************************************
    //* LINHA ABAIXO
    //*****************************************************
    //*****************************************************
```

```
  }

  return fwdPort;
}
#endif]
```

8.2 Apêndice B – Transator NoC OCP *slave*

```
#ifndef __ocpSlaveBus
#define __ocpSlaveBus

#include <systemc.h>
#include "intoRouterIf.h"
#include "outFromRouterIf.h"
#include "doorDefs.h"
#include "OCPdefs.h"

// posicoes possiveis nas portas no roteador
SC_MODULE(ocpSlaveBus){

    sc_in<bool > clock, reset_n;

    // Sinais OCP básicos
    sc_in<ocpMCmd >     MCmd;
    sc_in<sc_lv<ADDR_WDTH> >   MAddr;
    sc_in<sc_lv<DATA_WDTH> >  MData;
    sc_out<bool >       SCmdAccept;
    sc_out<sc_lv<DATA_WDTH> > SData;
    sc_out<ocpSResp >   SResp;

    // Porta de comunicacao com a NoC
    sc_port< intoRouterIf >    inAbsPort;
    sc_port< outFromRouterIf > outAbsPort;

    void dtlRcv();
    void absSnd();
    void dtlSnd();
    void absRcv();

    SC_HAS_PROCESS(ocpSlaveBus);
    ocpSlaveBus(sc_module_name _name, FLITTYPE _address):sc_module(_name){

      IPAddress=_address;

      SC_THREAD(dtlRcv);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

      SC_THREAD(dtlSnd);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

    }

    private:
      FLITTYPE IPAddress;
};

#endif

void ocpSlaveBus::dtlRcv(){
  FLITTYPE localData, localSize, localCmd;
  bool localResponse;
```

```
while(true){
  while(!reset_n.read()){
    SCmdAccept.write(false);
    wait();
  }

  // *********************
  // recebe header
  // *********************
  while(MCmd==cmdIdle) wait();
  localResponse=false;
  localData.range(7,0) =MData.read().range(7,0);
  localData.range(15,8)=IPAddress.range(7,0);

  while(!localResponse){
    outAbsPort->try2SendFlit(localData,&localResponse);
    if(!localResponse) wait();
  }
  // responde q conseguiu enviar o header;
  SCmdAccept.write(true);
  wait();
  SCmdAccept.write(false);
  wait();

  // *********************
  // recebe size
  // *********************
  while(MCmd==cmdIdle) wait();
  localResponse=false;
  localSize=MData.read().to_uint()+1;

  while(!localResponse){
    outAbsPort->try2SendFlit(localSize,&localResponse);
    if(!localResponse) wait();
  }
  // responde q conseguiu enviar o header;
  SCmdAccept.write(true);
  wait();
  SCmdAccept.write(false);
  wait();

  // *********************
  // recebe corpo da msg
  // *********************
  while(MCmd==cmdIdle) wait();
  localResponse=false;
  localData=MData.read();
  localCmd=(MCmd.read()==cmdWrite)?1:2;

  while(!localResponse){
    outAbsPort->try2SendFlit(localCmd,&localResponse);
    if(!localResponse) wait();
  }

  localResponse=false;

  while(!localResponse){
    outAbsPort->try2SendFlit(localData,&localResponse);
    if(!localResponse) wait();
```

```
    }
    localSize=localSize.to_uint()-2;
    SCmdAccept.write(true);
    wait();
    SCmdAccept.write(false);
    wait();

    //************************
    // Envia o resto do payload
    //************************
    while(localSize.to_uint()>0){

      while(MCmd==cmdIdle) wait();
      localResponse=false;
      localData=MData.read();

      while(!localResponse){
        outAbsPort->try2SendFlit(localData,&localResponse);
        if(!localResponse) wait();
      }
      // responde q conseguiu enviar o header;
      SCmdAccept.write(true);
      wait();
      SCmdAccept.write(false);
      wait();
      localSize=localSize.to_uint()-1;
    }
  }
}

void ocpSlaveBus::dtlSnd(){
  FLITTYPE localData, localSize, localCmd;
  while(true){
    while(!reset_n.read()){
      SResp.write(rspNULL);
      wait();
    }

    // receber header
    inAbsPort->haveNewFlit(&localData);
    wait();
    inAbsPort->byteAccepted();

    // receber size
    inAbsPort->haveNewFlit(&localSize);
    wait();
    inAbsPort->byteAccepted();

    // receber comando
    inAbsPort->haveNewFlit(&localCmd);
    wait();
    inAbsPort->byteAccepted();
    localSize=localSize.to_uint()-1;

    // receber payload
    if(localCmd[3]==1){

      while(localSize.to_uint()>0){
        inAbsPort->haveNewFlit(&localData);
```

```
        SResp.write(rspDVA);
        SData.write(localData);
        wait();
        SResp.write(rspNULL);
        wait();
        localSize=localSize.to_uint()-1;
        wait();
        inAbsPort->byteAccepted();
      }

    }
    else{
      cout << "Error: OCP slave interface can not receive Write/Read command." <<
endl;
      cout << "  Packet's source: " << localData.range(15,12).to_uint() << "x" <<
localData.range(11,8).to_uint() << endl;
    }



  }
}
```

8.3 Apêndice C – Transator NoC OCP *master*

```
#ifndef __ocpMasterBus
#define __ocpMasterBus

#include <systemc.h>
#include "intoRouterIf.h"
#include "outFromRouterIf.h"
#include "doorDefs.h"
#include "OCPdefs.h"

// posicoes possiveis nas portas no roteador
SC_MODULE(ocpMasterBus){

    sc_in<bool > clock, reset_n;

    // Sinais OCP básicos
    sc_out<sc_lv<ADDR_WDTH> >  MAddr;
    sc_out<ocpMCmd >    MCmd;
    sc_out<sc_lv<DATA_WDTH> > MData;
    sc_in<bool >        SCmdAccept;
    sc_in<sc_lv<DATA_WDTH> >  SData;
    sc_in<ocpSResp >    SResp;

    // Porta de comunicacao com a NoC
    sc_port< intoRouterIf >    inAbsPort;
    sc_port< outFromRouterIf > outAbsPort;

    void absRcv();
    void dtlSnd();


    SC_CTOR(ocpMasterBus){

      SC_THREAD(absRcv);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

      SC_THREAD(dtlSnd);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

    }

    private:
      FLITTYPE sizeStr, cmdStr, headerStr;
};

#endif

// NOC -> (MST) -> SLV -> IP
void ocpMasterBus::absRcv(){

  FLITTYPE localAddress, localData, localSize, localCmd;

  while(true){
    while(!reset_n.read()){
      MCmd.write(cmdIdle);
```

```
      wait();
    }

    MCmd.write(cmdIdle);
    // *********************
    // recebe header
    // *********************

    inAbsPort->haveNewFlit(&localData);
    headerStr=localData;
    inAbsPort->byteAccepted();

    // *********************
    // recebe size
    // *********************

    inAbsPort->haveNewFlit(&localData);
    localSize=localData;
    sizeStr=localData;
    inAbsPort->byteAccepted();

    // *********************
    // recebe comando
    // *********************

    inAbsPort->haveNewFlit(&localData);
    cmdStr=localData;
    localSize=localSize.to_uint()-1;
    inAbsPort->byteAccepted();

    // *********************
    // recebe resto do corpo
    // *********************
    while(localSize.to_uint()!=0){

      inAbsPort->haveNewFlit(&localAddress);

      // comando inválido identificando retorno de leitura
      if(cmdStr[3]==1){
        cout << "ERROR: retorno de leitura inválido no módulo OCP_MASTER_BUS" <<
endl;
      }
      // comando de escrita
      else if(cmdStr.range(2,0).to_uint()==1){
        inAbsPort->byteAccepted();
        inAbsPort->haveNewFlit(&localData);

        MCmd.write(cmdWrite);
        MData.write(localAddress);

        if(SCmdAccept.read()) wait();
        while(!SCmdAccept.read()) wait();

        MData.write(localData);

        if(SCmdAccept.read()) wait();
        while(!SCmdAccept.read()) wait();

        MCmd.write(cmdIdle);
```

```
        localSize=localSize.to_uint()-2;

      }
      // comando de leitura
      else if(cmdStr.range(2,0).to_uint()==2){

        MCmd.write(cmdRead);
        MData.write(localAddress);

        if(SCmdAccept.read()) wait();
        while(!SCmdAccept.read()) wait();

        MCmd.write(cmdIdle);
        localSize=localSize.to_uint()-1;

      }
      // comando nao tratado
      else{
        cout << "ERROR: Comando inválido no módulo OCP_MASTER_BUS" << endl;
      }

      inAbsPort->byteAccepted();

    }

  }
}

// IP -> SLV -> (MST) -> NOC

void ocpMasterBus::dtlSnd(){
  FLITTYPE localData, localHeader, localSize, localCmd;
  bool answer;

  while(true){
    while(!reset_n.read()) wait();

    if(SResp.read()!=rspNULL){
      // monta o pacote
      localHeader=0;
      localHeader.range(7,0)=headerStr.range(15,8);
      localSize=sizeStr;
      localCmd=8;
      localData=SData.read();

      // envia o header
      answer=false;
      while(!answer){
        outAbsPort->try2SendFlit(localData,&answer);
        if(!answer) wait();
      }

      // envia o size
      answer=false;
      while(!answer){
        outAbsPort->try2SendFlit(localSize,&answer);
        if(!answer) wait();
      }
```

```
      // envia o comando
      answer=false;
      while(!answer){
        outAbsPort->try2SendFlit(localCmd,&answer);
        if(!answer) wait();
      }

      // envia o primeiro dado
      answer=false;
      while(!answer){
        outAbsPort->try2SendFlit(localData,&answer);
        if(!answer) wait();
      }

      localSize=localSize.to_uint()-2;

      // envia o dado
      while(localSize.to_uint()>0){

        while(SResp.read()!=rspDVA) wait();
        localData=SData.read();
        answer=false;
        while(!answer){
          outAbsPort->try2SendFlit(localData,&answer);
        }
        localSize=localSize.to_uint()-1;
        wait();
      }

    }

    wait();

  }
}
```

8.4 Apêndice D – Transator NoC OCP *master/slave*

```
#ifndef __ocpMasterBus
#define __ocpMasterBus

#include <systemc.h>
#include "intoRouterIf.h"
#include "outFromRouterIf.h"
#include "doorDefs.h"
#include "OCPdefs.h"

// posicoes possiveis nas portas no roteador
SC_MODULE(ocpMasterSlaveBus){

    sc_in<bool > clock, reset_n;

    // Sinais OCP básicos
    sc_out<sc_lv<ADDR_WDTH> >  MAddr_o;
    sc_in<sc_lv<ADDR_WDTH> >  MAddr_i;
    sc_out<ocpMCmd >    MCmd_o;
    sc_in<ocpMCmd >    MCmd_i;
    sc_out<sc_lv<DATA_WDTH> > MData_o;
    sc_in<sc_lv<DATA_WDTH> > MData_i;
    sc_out<bool >        SCmdAccept_o;
    sc_in<bool >        SCmdAccept_i;
    sc_out<sc_lv<DATA_WDTH> >  SData_o;
    sc_in<sc_lv<DATA_WDTH> >  SData_i;
    sc_out<ocpSResp >    SResp_o;
    sc_in<ocpSResp >    SResp_i;

    // Porta de comunicacao com a NoC
    sc_port< intoRouterIf >    inAbsPort;
    sc_port< outFromRouterIf > outAbsPort;

    void absRcv();
    void dtlSndResp();
    void dtlSndReq();

    SC_HAS_PROCESS(ocpMasterSlaveBus);
    ocpMasterSlaveBus(sc_module_name _name, FLITTYPE _address):sc_module(_name){

      IPAddress=_address;

      SC_THREAD(absRcv);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

      SC_THREAD(dtlSndResp);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

      SC_THREAD(dtlSndReq);
      sensitive_neg << reset_n;
      sensitive_pos << clock;

    }

    private:
```

```
      FLITTYPE sizeStr, cmdStr, headerStr, IPAddress;
};

#endif

// NOC -> (MST/SLV) -> MST/SLV -> IP
//   Receving packet from NoC
//     * decides if it is an answer or a request
void ocpMasterSlaveBus::absRcv(){

  FLITTYPE localAddress, localData, localSize, localCmd;

  while(true){
    while(!reset_n.read()){
      MCmd_o.write(cmdIdle);
      wait();
    }

    MCmd_o.write(cmdIdle);

    // *********************
    // recebe header
    // *********************

    inAbsPort->haveNewFlit(&localData);
    headerStr=localData;
    inAbsPort->byteAccepted();

    // *********************
    // recebe size
    // *********************

    inAbsPort->haveNewFlit(&localData);
    localSize=localData;
    sizeStr=localData;
    inAbsPort->byteAccepted();

    // *********************
    // recebe comando
    // *********************

    inAbsPort->haveNewFlit(&localData);
    cmdStr=localData;
    localSize=localSize.to_uint()-1;
    inAbsPort->byteAccepted();


    // *********************
    // recebe resto do corpo
    // *********************
    while(localSize.to_uint()!=0){

      inAbsPort->haveNewFlit(&localAddress);

      // comando inválido identificando retorno de leitura
      if(cmdStr[3]==1){
        while(localSize.to_uint()>0){
          SResp_o.write(rspDVA);
          localData=localAddress;
```

```
          SData_o.write(localData);
          wait();
          SResp_o.write(rspNULL);
          wait();
          localSize=localSize.to_uint()-1;
        }

      }
      // comando de escrita
      else if(cmdStr.range(2,0).to_uint()==1){
        inAbsPort->byteAccepted();
        inAbsPort->haveNewFlit(&localData);

        MCmd_o.write(cmdWrite);
        MData_o.write(localAddress);

        if(SCmdAccept_i.read()) wait();
        while(!SCmdAccept_i.read()) wait();

        MData_o.write(localData);

        if(SCmdAccept_i.read()) wait();
        while(!SCmdAccept_i.read()) wait();

        MCmd_o.write(cmdIdle);
        localSize=localSize.to_uint()-2;

      }
      // comando de leitura
      else if(cmdStr.range(2,0).to_uint()==2){

        MCmd_o.write(cmdRead);
        MData_o.write(localAddress);

        if(SCmdAccept_i.read()) wait();
        while(!SCmdAccept_i.read()) wait();

        MCmd_o.write(cmdIdle);
        localSize=localSize.to_uint()-1;

      }
      // comando nao tratado
      else{
        cout << "ERROR: Comando inválido no módulo OCP_MASTER_SLAVE_BUS" << endl;
      }

      inAbsPort->byteAccepted();

    }

  }
}

// IP -> MST/SLV -> (MST/SLV) -> NOC
//   Slave answering
void ocpMasterSlaveBus::dtlSndResp(){
  FLITTYPE localData, localHeader, localSize, localCmd;
  bool answer;
```

```
while(true){
  while(!reset_n.read()) wait();

  if(SResp_i.read()!=rspNULL){
    // monta o pacote
    localHeader=0;
    localHeader.range(7,0)=headerStr.range(15,8);
    localSize=sizeStr;
    localCmd=8;
    localData=SData_i.read();

    // envia o header
    answer=false;
    while(!answer){
      outAbsPort->try2SendFlit(localHeader,&answer);
      if(!answer) wait();
    }
    wait();

    // envia o size
    answer=false;
    while(!answer){
      outAbsPort->try2SendFlit(localSize,&answer);
      if(!answer) wait();
    }
    wait();

    // envia o comando
    answer=false;
    while(!answer){
      outAbsPort->try2SendFlit(localCmd,&answer);
      if(!answer) wait();
    }
    wait();

    // envia o primeiro dado
    answer=false;
    while(!answer){
      outAbsPort->try2SendFlit(localData,&answer);
      if(!answer) wait();
    }
    wait();

    localSize=localSize.to_uint()-2;

    // envia o dado
    while(localSize.to_uint()>0){

      while(SResp_i.read()!=rspDVA) wait();
      localData=SData_i.read();
      answer=false;
      while(!answer){
        outAbsPort->try2SendFlit(localData,&answer);
        if(!answer) wait();
      }
      localSize=localSize.to_uint()-1;
    }

  }
```

```
    wait();

  }
}

// IP -> MST/SLV -> (MST/SLV) -> NOC
//   Master requesting
void ocpMasterSlaveBus::dtlSndReq(){
  FLITTYPE localData, localSize, localCmd;
  bool localResponse;
  while(true){
    while(!reset_n.read()){
      SCmdAccept_o.write(false);
      wait();
    }

    // *********************
    // recebe header
    // *********************
    while(MCmd_i.read()==cmdIdle) wait();
    localData.range(7,0) =MData_i.read().range(7,0);
    localData.range(15,8)=IPAddress.range(7,0);

    localResponse=false;
    while(!localResponse){
      outAbsPort->try2SendFlit(localData,&localResponse);
      if(!localResponse) wait();
    }
    // responde q conseguiu enviar o header;
    SCmdAccept_o.write(true);
    wait();
    SCmdAccept_o.write(false);
    wait();

    // *********************
    // recebe size
    // *********************
    while(MCmd_i.read()==cmdIdle) wait();
    localSize=MData_i.read().to_uint()+1;

    localResponse=false;
    while(!localResponse){
      outAbsPort->try2SendFlit(localSize,&localResponse);
      if(!localResponse) wait();
    }
    // responde q conseguiu enviar o header;
    SCmdAccept_o.write(true);
    wait();
    SCmdAccept_o.write(false);
    wait();

    // *********************
    // recebe corpo da msg
    // *********************
    while(MCmd_i.read()==cmdIdle) wait();
    localData=MData_i.read();
    localCmd=(MCmd_i.read()==cmdWrite)?1:2;
```

```
    localResponse=false;
    while(!localResponse){
      outAbsPort->try2SendFlit(localCmd,&localResponse);
      if(!localResponse) wait();
    }
    wait();

    localResponse=false;
    while(!localResponse){
      outAbsPort->try2SendFlit(localData,&localResponse);
      if(!localResponse) wait();
    }
    localSize=localSize.to_uint()-2;
    SCmdAccept_o.write(true);
    wait();
    SCmdAccept_o.write(false);
    wait();

    //************************
    // Envia o resto do payload
    //************************
    while(localSize.to_uint()>0){

      while(MCmd_i.read()==cmdIdle) wait();
      localData=MData_i.read();

      localResponse=false;
      while(!localResponse){
        outAbsPort->try2SendFlit(localData,&localResponse);
        if(!localResponse) wait();
      }
      // responde q conseguiu enviar o header;
      SCmdAccept_o.write(true);
      wait();
      SCmdAccept_o.write(false);
      wait();
      localSize=localSize.to_uint()-1;
    }
  }
}
```

8.5 Apêndice E – NoC Hermes TL 2x2 em SystemC

```
#ifndef _noc
#define _noc

#include "systemc.h"
#include "interRouterChl.h"
#include "../../algoritmos/roteamento/mesh.h"
#include "router.h"

#define ROW 2
#define COL 2

SC_MODULE(noc){

  sc_port< intoRouterIf    > inPort_0x0;
  sc_port< outFromRouterIf > outPort_0x0;

  sc_port< intoRouterIf    > inPort_0x1;
  sc_port< outFromRouterIf > outPort_0x1;

  sc_port< intoRouterIf    > inPort_1x0;
  sc_port< outFromRouterIf > outPort_1x0;

  sc_port< intoRouterIf    > inPort_1x1;
  sc_port< outFromRouterIf > outPort_1x1;

  router *chv_0x0;
  router *chv_0x1;
  router *chv_1x0;
  router *chv_1x1;

  interRouterChl *tmpChl_0;

  interRouterChl *Chl_0x0S_0x1N;
  interRouterChl *Chl_0x0E_1x0W;
  interRouterChl *tmpChl_1;

  interRouterChl *Chl_0x1N_0x0S;
  interRouterChl *tmpChl_2;

  interRouterChl *Chl_0x1E_1x1W;
  interRouterChl *tmpChl_3;

  interRouterChl *tmpChl_4;

  interRouterChl *Chl_1x0S_1x1N;
  interRouterChl *tmpChl_5;

  interRouterChl *Chl_1x0W_0x0E;
  interRouterChl *Chl_1x1N_1x0S;
  interRouterChl *tmpChl_6;

  interRouterChl *tmpChl_7;

  interRouterChl *Chl_1x1W_0x1E;

  SC_CTOR(noc){
```

```
chv_0x0 = new router("chv_0x0", 0x0000, 0x0022, XYPuro);
chv_0x1 = new router("chv_0x1", 0x0001, 0x0022, XYPuro);
chv_1x0 = new router("chv_1x0", 0x0010, 0x0022, XYPuro);
chv_1x1 = new router("chv_1x1", 0x0011, 0x0022, XYPuro);

tmpChl_0 = new interRouterChl("tmpChl_0");
chv_0x0->inPort[0](*tmpChl_0 );
chv_0x0->outPort[0](*tmpChl_0 );

Chl_0x0S_0x1N = new interRouterChl ("Chl_0x0S_0x1N");
chv_0x0->inPort[1](*Chl_0x0S_0x1N);
chv_0x1->outPort[0](*Chl_0x0S_0x1N);

Chl_0x0E_1x0W = new interRouterChl ("Chl_0x0E_1x0W");
chv_0x0->inPort[2](*Chl_0x0E_1x0W);
chv_1x0->outPort[3](*Chl_0x0E_1x0W);

tmpChl_1 = new interRouterChl("tmpChl_1");
chv_0x0->inPort[3](*tmpChl_1 );
chv_0x0->outPort[3](*tmpChl_1 );

chv_0x0->inPort[4]( inPort_0x0);
chv_0x0->outPort[4]( outPort_0x0);

Chl_0x1N_0x0S = new interRouterChl ("Chl_0x1N_0x0S");
chv_0x1->inPort[0](*Chl_0x1N_0x0S);
chv_0x0->outPort[1](*Chl_0x1N_0x0S);

tmpChl_2 = new interRouterChl("tmpChl_2");
chv_0x1->inPort[1](*tmpChl_2 );
chv_0x1->outPort[1](*tmpChl_2 );

Chl_0x1E_1x1W = new interRouterChl ("Chl_0x1E_1x1W");
chv_0x1->inPort[2](*Chl_0x1E_1x1W);
chv_1x1->outPort[3](*Chl_0x1E_1x1W);

tmpChl_3 = new interRouterChl("tmpChl_3");
chv_0x1->inPort[3](*tmpChl_3 );
chv_0x1->outPort[3](*tmpChl_3 );

chv_0x1->inPort[4]( inPort_0x1);
chv_0x1->outPort[4]( outPort_0x1);

tmpChl_4 = new interRouterChl("tmpChl_4");
chv_1x0->inPort[0](*tmpChl_4 );
chv_1x0->outPort[0](*tmpChl_4 );

Chl_1x0S_1x1N = new interRouterChl ("Chl_1x0S_1x1N");
chv_1x0->inPort[1](*Chl_1x0S_1x1N);
chv_1x1->outPort[0](*Chl_1x0S_1x1N);

tmpChl_5 = new interRouterChl("tmpChl_5");
chv_1x0->inPort[2](*tmpChl_5 );
chv_1x0->outPort[2](*tmpChl_5 );

Chl_1x0W_0x0E = new interRouterChl ("Chl_1x0W_0x0E");
chv_1x0->inPort[3](*Chl_1x0W_0x0E);
chv_0x0->outPort[2](*Chl_1x0W_0x0E);
```

```
    chv_1x0->inPort[4]( inPort_1x0);
    chv_1x0->outPort[4]( outPort_1x0);

    Chl_1x1N_1x0S = new interRouterChl ("Chl_1x1N_1x0S");
    chv_1x1->inPort[0](*Chl_1x1N_1x0S);
    chv_1x0->outPort[1](*Chl_1x1N_1x0S);

    tmpChl_6 = new interRouterChl("tmpChl_6");
    chv_1x1->inPort[1](*tmpChl_6 );
    chv_1x1->outPort[1](*tmpChl_6 );

    tmpChl_7 = new interRouterChl("tmpChl_7");
    chv_1x1->inPort[2](*tmpChl_7 );
    chv_1x1->outPort[2](*tmpChl_7 );

    Chl_1x1W_0x1E = new interRouterChl ("Chl_1x1W_0x1E");
    chv_1x1->inPort[3](*Chl_1x1W_0x1E);
    chv_0x1->outPort[2](*Chl_1x1W_0x1E);

    chv_1x1->inPort[4]( inPort_1x1);
    chv_1x1->outPort[4]( outPort_1x1);


  }
};
#endif
```