FACULDADE DE INFORMÁTICA
PUCRS - Brasil
http://www.inf.pucrs.br

Projeto e Implementação da Arquitetura de Comunicação MERCURY: uma NoC intra-chip com topologia toro, filas centrais compartilhadas e modo de chaveamento virtual-cut-through

Érico Nunes Ferreira Bastos, Celso Soccol,
Ney Laert Vilar Calazans

TECHNICAL REPORT SERIES

Number 050
July, 2005 (rev 1.1 August 2005)

Contato:

calazans@inf.pucrs.br

http://www.inf.pucrs.br/~calazans

É. N. F. Bastos é estudante de Mestrado no CPGCC/PUCRS/Brasil desde 2004. Bastos é membro do grupo de pesquisa GAPH desde 2004, e recebe uma bolsa do governo federal brasileiro através da CAPES de março 2004 a fevereiro de 2006.

C. Soccol é estudante de graduação em Ciência da Computação na FACIN-PUCRS desde 2002. Celso é membro do grupo de pesquisa GAPH desde 2004, e recebe uma bolsa do governo federal através do CNPq, no escopo do projeto Brazil-IP, de 2004 a 2006.

N. L. V. Calazans trabalha na FACIN-PUCRS desde 1986. Ele é professor titular desde 1999. Seus principais interesses de pesquisa são o projeto de sistemas digitais em vários níveis de abstração, arquiteturas de comunicação intra-chip, a prototipação rápida de sistemas, o projeto integrado de hardware e software e aplicações em telecomunicação. Prof. Calazans é o líder do grupo de pesquisa GAPH no PPGCC/FACIN/PUCRS.

Av. Ipiranga, 6681

90619-900 Porto Alegre – RS – Brazil

Resumo

Redes intra-chip ou NoCs são vistas hoje como um meio de contribuir para a solução da crise de projeto de sistemas integrados em um único chip (SoCs). Arquiteturas de comunicação intra-chip baseadas em NoCs prometem escalabilidade e reuso de qualidade superior ao que é ofertado por fios dedicados ou barramentos compartilhados. Este trabalho propõe uma arquitetura de comunicação intra-chip do tipo NoC que pressupõe a organização de um SoC segundo uma topologia regular do tipo toro bi-dimensional. Esta arquitetura, denominada Mercury, possui como principal elemento um roteador construído para dar suporte a um algoritmo eficiente de roteamento, que é livre de *deadlock* e *livelock*. A descrição do projeto abstrato deste roteador, um modelo de referência no nível de transação, junto com um conjunto de ferramentas de apoio ao desenvolvimento da arquitetura de comunicação Mercury são a principal contribuição deste documento.

Palavra-Chave: redes intra-chip, sistemas integrados em um chip, roteadores, algoritmos de roteamento, interconexão intra-chip, arquiteturas de comunicação.

Índice de Assuntos

1 INTRODUÇÃO

Uma crescente densidade no número de transistores, freqüências de operação elevadas, tempos de projeto e ciclo de vida reduzidos caracterizam o cenário da indústria de semicondutores na atualidade [13]. Sob estas condições, projetistas têm desenvolvido circuitos integrados (CIs) que integram um conjunto extenso de elementos funcionais complexos e com alto grau de heterogeneidade, conhecidos como *Sistemas Integrados em um Chip* (do inglês, *Systems on a Chip* ou SoCs). Conforme descrito, por exemplo, por Gupta e outros [12] e Bergamaschi e outros [3], o projeto de SoCs é baseado no reuso maciço de propriedade intelectual implementada e comercializada sob a forma de *núcleos de propriedade intelectual* (do inglês, *intellectual property cores* ou IP cores ou simplesmente IPs). Gupta e outros [12] definem um *núcleo* como um módulo de hardware pré-projetado e pré-verificado que pode ser empregado como bloco construtivo de aplicações complexas e de grande porte sobre um CI. Exemplos de núcleos são controladores de acesso a memórias, processadores, ou controladores de dispositivos periféricos tais como controladores MAC para redes Ethernet ou controladores de barramentos PCI. Núcleos podem ser digitais ou analógicos, ou mesmo ser compostos por blocos que empreguem tecnologias tais como sistemas micro-eletromecânicos ou opto-eletrônicos [17][12]. Núcleos não compõem SoCs de forma isolada, eles devem incluir uma arquitetura de interconexão e interfaces para dispositivos periféricos externos [17]. Ainda que estas arquitetura e interfaces costumeiramente sejam consideradas como núcleos, elas possuem características distintivas. Por exemplo, dificilmente uma arquitetura de comunicação pode ser empregada sem alterações/adaptações em dois CIs distintos, fato comum a ocorrer em núcleos processadores como interfaces MAC Ethernet, PCI ou USB. A arquitetura de interconexão e as interfaces físicas incluem interfaces físicas e mecanismos que habilitam a comunicação componentes do SoC e entre estes e dispositivos externos.

Tradicionalmente, a arquitetura de interconexão intra-chip tem sido implementada seja através do uso de conjuntos de fios dedicados ou barramentos compartilhados. Conjuntos de fios dedicados são efetivos em sistemas com um pequeno número de núcleos, não se aplicando a SoCs e sendo uma escolha que gera problemas sérios ao se considerar a necessidade de agregar escalabilidade a projetos complexos. Esta forma de implementação possui, portanto, características negativas quanto a reuso e escalabilidade. Em última análise, barramentos compartilhados são implementados como um conjunto de fios compartilhado por múltiplos núcleos. Esta abordagem é simples e totalmente reutilizável, mas apresenta desempenho mínimo, pois permite apenas uma comunicação por vez, com todos os núcleos compartilhando a largura de banda de comunicação provida pelo barramento. A escalabilidade é limitada a algumas dezenas de núcleos, tornando a abordagem incompatível com SoCs futuros [14]. O uso de múltiplos barramentos interconectados por pontes passivas ou ativas produz arquiteturas hierárquicas de barramentos, que reduzem algumas das restrições listadas, uma vez que diferentes barramentos podem prover meios de satisfazer diferentes requisitos de largura de banda e/ou protocolos de comunicação, além de aumentar o paralelismo da comunicação. Contudo, o projeto e a implementação de barramentos hierárquicos constituem uma disciplina *ad hoc*, podendo comprometer o reuso e escalabilidade de soluções específicas.

De acordo com vários autores, e.g. os das referências [11], [7], [14] e [2], arquiteturas de interconexão baseadas em barramentos compartilhados não terão condições de prover suporte aos requisitos de comunicação de SoCs de futuro próximo. Ainda, de acordo com o ITRS, CIs contendo bilhões de transistores, com um *min-feature-size* em torno de 50nm e freqüências de relógio em torno de 10 GHz serão uma realidade antes de 2012 [13]. Neste contexto, redes estruturadas mais complexas que barramentos compartilhados aparecem como uma solução possivelmente melhor para implementar arquiteturas de comunicação para tais SoCs. Um conceito que vem evoluindo bastante neste início de século são as denominadas *redes intra-chip* (em inglês, networks on chip ou NoCs) [2]. NoCs são arquiteturas especiais, formadas por fiação de alcance local no interior de um CI e elementos chaveadores (normalmente denominados *roteadores* ou *chaves*) cuja interconexão define uma topologia de comunicação, em cujos extremos conecta-se os núcleos do SoC.

NoCs podem apresentar uma topologia de interconexão regular ou não, podem apresentar diferentes relações entre as cardinalidade de elementos chaveadores e núcleos do SoC, etc. O presente documento descreve a especificação e uma implementação abstrata no nível de transação (ou seja, um modelo de referência) para um arquitetura de comunicação do tipo NoC com topologia toro 2D regular. Esta arquitetura é denominada **Mercury**. Também apresenta-se aqui ferramental básico de projeto e validação associados a esta arquitetura.

O resto deste documento está organizado da seguinte maneira. Nas Seções seguintes desta Introdução, explora-se a motivação e os objetivos do trabalho que compreende a implementação descrita aqui, na Seção 1.1. Em seguida, aborda-se brevemente o estado da arte de NoCs implementadas anteriormente pelo grupo GAPH e propostas de redes com topologia toro, sejam estas extra-chip ou NoCs, na Seção 1.2. A Seção 2 apresenta as linhas gerais da especificação da rede Mercury, com ênfase no algoritmo de roteamento escolhido. Em seguida, a Seção 3 descreve o diagrama de blocos do projeto do roteador e da rede, com algum nível de detalhamento dos blocos componentes. Este diagrama de blocos é a base para a implementação do modelo de referência da rede **Mercury** no nível de abstração de transação (em inglês *transaction-level* modeling, ou TL), assunto abordado na Seção 4. Um conjunto de ferramentas usado para gerar e validar instâncias da NoC Mercury TL são o tema da Seção 5. O documento finaliza com a apresentação de um conjunto de conclusões sobre o trabalho realizado até o momento e uma perspectiva de trabalhos em andamento e futuros, na Seção 6.

1.1 Motivações e Objetivos

Parte da motivação deste trabalho já foi apresentada acima, ao mencionar os compromissos envolvidos na escolha de arquiteturas de comunicação intra-chip, concluindo pela necessidade de transcender o uso barramentos compartilhados e conexões *ad hoc* via fios dedicados. Além desta motivação, existem outras, derivadas da experiência do grupo que propõe o trabalho com projeto, implementação e avaliação de redes intra-chip, discutidas a seguir.

Desde 2002 o grupo GAPH aborda a pesquisa no tema de redes intra-chip. A partir do trabalho seminal no escopo do grupo, de ênfase eminentemente exploratória e de implementação prática [18] [20], atingiu-se a proposta da infra-estrutura **Hermes** de suporte à implementação de NoCs com baixo consumo de área, inicialmente descrita em

[21]. Esta infra-estrutura tem evoluído ao longo dos últimos anos, agregando crescente número de funcionalidades [19][4], métodos de modelagem, projeto, validação e avaliação de NoCs [4][15][16][26] e ferramental de apoio associado [25][31].

Apesar do extenso trabalho já realizado, percebe-se um conjunto de limitações na abordagem adotada. Primeiro, todos os trabalhos citados restringem sobremaneira a estrutura de NoCs investigada, visando tornar a complexidade da pesquisa gerenciável. As redes investigadas em todos os trabalhos citados assumem chaveamento por pacotes, empregam apenas a topologia malha bidimensional regular (malha 2D), usam apenas o modo de chaveamento *wormhole* [8][9] onde o tamanho do *flit* e do *phit* [8][9] são idênticos e assumem armazenamento temporário baseado apenas em filas de entrada. Não obstante, vários graus de liberdade são adotados nos trabalhos, tais como não restringir as dimensões da malha 2D, permitir dimensionar livremente as filas e o tamanho do *flit*/*phit* dos pacotes. Estas opção é razoável, sendo corroborada pela maior parte da investigação atual em NoCs no mundo, que assume parte ou todas as limitações aqui assumidas. Isto decorre em maior escala de um raciocínio natural de mapear características de redes em sistemas distribuídos para o ambiente intra-chip, buscando acessar o melhor custo benefício na implementação de NoCs. Em menor escala, decorre também dos resultados experimentais obtidos com implementações de NoCs. Contudo, ainda não foram suficientemente investigadas as vantagens intrínsecas de usar, por exemplo, outras topologias e modos de chaveamento, seja pelo GAPH ou pela comunidade científica em geral. A motivação principal do trabalho parcialmente descrito aqui é expandir o escopo de pesquisa em NoCs no grupo para abranger NoCs de topologia diversa de malha 2D, modos de chaveamento diversos de wormhole e estratégias de armazenamento distintas de filas de entrada. O trabalho continua sendo de cunho exploratório e eminentemente prático, visando coletar informações que permitam comparar diferentes aspectos ao longo do projeto de NoCs.

Os objetivos deste trabalho são descrever o projeto abstrato da arquitetura de comunicação **Mercury**, bem como descrever o conjunto inicial de ferramentas disponíveis para desenvolvimento e validação da mesma.

1.2 Estado da Arte

Não é objetivo aqui elaborar um levantamento extenso da pesquisa atual em NoCs. Contudo, visando estabelecer o contexto em que este se realiza, propõe-se na Seção 1.2.1 um breve relato do trabalho prévio do grupo GAPH. Em seguida, a Seção 1.2.2 discute a bibliografia de base que descreve redes contendo algumas das escolhas feitas para implementação neste trabalho e cita algumas das principais propostas disponíveis que advogam o uso de redes intra-chip com topologia toro. Uma avaliação abrangente do estado da arte em NoCs com várias topologias até 2004 pode ser encontrada em [21].

1.2.1 A Rede Hermes [21]

A rede Hermes é uma rede direta, formada por estrutura de interconexão de roteadores, segundo uma topologia malha 2D de dimensões arbitrárias [20]. A Figura 1 ilustra uma instância 3x3 da rede Hermes. Cada roteador da rede possui um conjunto de até 4 portas bidirecionais que interconectam roteadores entre si e 1 porta bidirecional local que conecta o roteador ao núcleo processador do SoC. Como se emprega topologia malha 2D, uma dada instância da rede pode apresentar até 9 tipos distintos de roteadores, dependendo da posição deste com relação aos limites da rede. Cada roteador

possui um endereço físico único na rede, expresso em coordenadas XY na malha 2D.

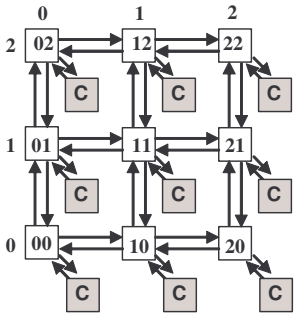

Figura 1 – Exemplo de uma instância da rede Hermes, com dimensões 3x3. C identifica cada núcleo processador do SoC. Os quadrados com números no interior representam os roteadores da rede, e os números indicam a posição de cada roteador na rede, em coordenadas cartesianas (XY).

O uso de topologia malha 2D em redes intra-chip é justificado, por exemplo, por facilitar as tarefas de posicionamento e roteamento da rede no CI. O roteador proposto pode ser adaptado para construir topologias toro 2D, hipercubos 2D ou 3D e outras topologias que compartilham algumas propriedades com topologias malha 2D. Entretanto, a construção de topologias diferentes de malha 2D pode implicar mudanças na atribuição de coordenadas aos roteadores. Além disto a estrutura do roteador pode depender também da funcionalidade de algoritmos de roteamento.

O principal objetivo de um roteador intra-chip é habilitar a transferência correta de mensagens entre núcleos processadores do SoC. O roteador Hermes original, ilustrado na Figura 2, é bastante simples, sendo composto de uma *lógica de roteamento* unificada, disputada pelas portas de entrada do roteador e uma *lógica de arbitragem* para controlar o acesso destes à lógica de roteamento Estes dois elementos constituem a lógica de controle do roteador (em inglês, *Control Logic*). Completa o roteador um conjunto de *portas de comunicação*, usadas para prover interação entre roteadores ou entre o roteador e seu núcleo processador associado. As portas de comunicação incluem canais de entrada e de saída dotados de meios de armazenamento temporário (*buffers*) de mensagens entrando pela porta.

As até 5 portas bidirecionais do roteador são denominadas East, West, North, South e Local. Cada porta possui um buffer de entrada pra armazenamento temporário. A porta Local estabelece a comunicação entre o roteador e seu núcleo. As demais portas conectam-se a roteadores vizinhos como apresentado na Figura 1.

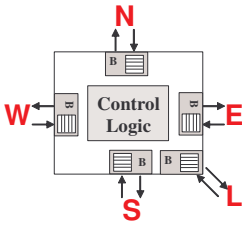

Figura 2 – Diagrama de Blocos do roteador Hermes. O bloco denominado Control Logic inclui a lógica de roteamento e a lógica de arbitragem. B indica a posição dos *buffers* de entrada.

O modo de chaveamento *wormhole* foi escolhido com o objetivo de minimizar o tamanho dos *buffers* do roteador, uma vez que a dimensão destes é o principal fator determinante da área ocupada pelo roteador, e portanto influencia diretamente a

sobrecarga de área dedicada à comunicação no SoC. Além desta vantagem, o modo wormhole tem potencial para reduzir a latência de pacotes na rede e habilita o uso do conceito de *canais virtuais* [6][8][19] que provê meios de melhorar o uso da largura de banda de comunicação total da rede. O modo *wormhole* implica dividir pacotes em *flits*. O tamanho do *flit* é parametrizável na rede Hermes.

Cada pacote que trafega na rede Hermes é formado por um número inteiro de *flits*. Tamanhos de *flit* de 8, 16 e 32 bits têm sido usados em implementações de redes Hermes. O primeiro e o segundo *flits* de qualquer pacote constituem um cabeçalho. O primeiro informa o endereço do roteador alvo do pacote (em coordenadas XY), contendo opcionalmente o endereço do roteador de origem do pacote. Este primeiro *flit* é denominado *header flit*. O segundo *flit* contém o número de *flits* na carga útil do pacote, que inclui todos os *flits* após os dois primeiros. Este tamanho pode ser zero, o que é útil para uso em pacotes de controle. O número de *flits* em um pacote é definido a partir do tamanho do *flit*, sendo, no máximo, $(1 + 2^{(\text{tamanho do flit, em bits})})$.

O roteador Hermes pode estabelecer até 5 comunicações concorrentes independentes entre suas portas de entrada e saída, embora o estabelecimento das conexões precise ser realizado de forma seqüencial, dada a existência de uma lógica de roteamento única por roteador para atender solicitações de todas as 5 portas de entrada. A prioridade de acesso à lógica de roteamento é definida de maneira dinâmica, usando uma estratégia do tipo *round-robin* [20]. O método garante um grau razoável de justiça no acesso ao roteamento para todas as entradas do roteador.

Quando um *flit* é bloqueado em um dado roteador, o desempenho da rede de comunicação é afetado, uma vez que os *flits* que pertencem a um dado pacote são bloqueados, em uma seqüência de roteadores ao longo do caminho entre os roteadores fonte e destino. O número de roteadores envolvidos no bloqueio de um pacote pode ser maior do que 1 simultaneamente, dependendo do tamanho do pacote e do tamanho dos *buffers* dos roteadores intermediários. Os *buffers* de entrada existem e são dimensionados para reduzir a perda de desempenho na arquitetura de comunicação. Os buffers de entrada da rede Hermes são filas circulares simples (FIFOs).

Supondo que não haja bloqueio nem espera por roteamento, pode-se calcular a latência mínima em ciclos de relógio para transferir um pacote de um roteador fonte para um roteador destino, usando a fórmula:

$$latência = \left(\sum_{i=1}^{n} R_i\right) + P \times C_f \,.$$

Nesta fórmula, tem-se: *(i)* n é o número de roteadores no caminho de comunicação incluindo os roteadores fonte e destino. Este valor depende de vários fatores, entre os quais os mais importantes são a posição relativa do fonte e do destino e o algoritmo de roteamento; *(ii)* R_i é o tempo necessário para o algoritmo de roteamento executar a conexão entre uma entrada e uma saída em cada roteador. Na versão inicial da rede este valor é no mínimo 10 ciclos de relógio; *(iii)* P é o tamanho da carga útil do pacote em *flits*; *(iv)* uma constante C_f, cujo valor depende da estratégia de controle de fluxo empregada pela instância da rede Hermes. Na primeira versão da rede, este número valia 2, devido ao uso de um mecanismo de controle de fluxo entre roteadores baseado em um protocolo de *handshake* assíncrono. Neste, dois ciclos de relógio são necessários para cada troca de um dado. Em versões posteriores da rede [4], pode-se escolher entre controle de fluxo baseado em *handshake* assíncrono ou baseado em créditos, o último

mais eficiente, aplicável a redes com *buffers* de entrada. Usar controle de fluxo baseado em créditos faz com que $C_f = 1$, aumentando sobremaneira o desempenho da rede.

Mais detalhes sobre a rede Hermes, seu roteador e a funcionalidade deste, bem como as opções de parametrização da rede podem ser obtidos em [4][18][20] e [21].

1.2.2 Contexto de Propostas de Redes e NoCs com Topologia Toro

Mais uma vez, não é objetivo aqui abordar definições básicas de redes ou de topologias. Assume-se que o leitor tenha conhecimento prévio do tema, podendo-se consultar o Capítulo 5 da referência [8], ou os Capítulos 3 e 4 da referência [1].

A referência [1] apresenta os conceitos fundamentais de redes em geral e redes com topologia toro em particular, no seu Capítulo 3. Além disto, o Capítulo 4 propõe uma revisão parcial do estado da arte em redes e algoritmos de roteamento para topologias toro, citando 11 algoritmos e explorando as características funcionais de 3 destes algoritmos em detalhe. O primeiro dos 3 algoritmos detalhados é um trabalho seminal onde Dally e Seitz descrevem a implementação de um CI que realiza o roteamento em computadores paralelos usando topologia toro de interconexão de processadores [6]. Os dois últimos algoritmos foram propostos por Cypher e Gravano em contexto de aplicação similar ao do primeiro algoritmo [5].

No restante deste documento, os algoritmos propostos por Cypher e Gravano são denominados **Algoritmo 1** e **Algoritmo 2**. O **Algoritmo 1** foi selecionado neste trabalho como base para o mapeamento e implementação de um roteador para redes intra-chip. O roteador em questão é voltado exclusivamente para a implementação de redes com *topologia toro*, usando modo de chaveamento *virtual cut-through* e empregando *filas centrais compartilhadas*. O critério que determinou a escolha do algoritmo foi, como se pode notar a máxima discrepância de características com relação a redes já desenvolvidas pelo grupo GAPH, atendendo à motivação principal apresentada na Seção 1.1.

A proposta de NoCs para arquiteturas de comunicação com topologia toro não é uma iniciativa pioneira deste trabalho. Alguns trabalhos anteriores já descreveram propostas neste sentido. Dentre estes talvez o mais divulgado seja o artigo seminal de Dally e Towles [7], que não menciona uma implementação, mas justifica a escolha de uma topologia toro frente as outras alternativas. Um trabalho que inclui uma implementação de rede toro é o de Marescaux e outros do IMEC [23][24], que propõe um roteador capaz de dar suporte à implementação de redes toro 2D e descreve a prototipação de uma forma restrita deste roteador para implementar uma rede com topologia toro 1D ou seja, um anel. Um terceiro conjunto de trabalhos propõe um roteador e rede associada com topologia toro 1D, implementando um anel bidirecional criando a rede denominada Proteo [27][28][29].

2 ESPECIFICAÇÃO DA REDE MERCURY

O **Algoritmo 1** de Cypher e Gravano [5] é executado em cada roteador da rede **Mercury** e possui as características de ser livre de *deadlock*, *livelock*, e *starvation*. O algoritmo utiliza sempre o menor caminho entre os nodos origem e destino (ou seja, é um algoritmo de caminho mínimo), é totalmente adaptativo e distribuído.

Algumas escolhas foram realizadas para implementação do algoritmo. Dentre estas o método utilizado para transferência de dados, denominado chaveamento de pacotes. Nele não se possui um caminho fixo pelo qual todos os pacotes devem seguir na rede, cada pacote informa a cada roteador a direção que deseja seguir, não existindo um caminho pré-definido, o que o caracteriza como um algoritmo de roteamento distribuído. No chaveamento de pacotes [5][6][11], os algoritmos de roteamento podem ser classificados, segundo o *modo de chaveamento*, em três categorias: *store-and-forward*, *virtual cut-through* ou *wormhole*.

Para a primeira implementação TL deste trabalho foi escolhido o modo de chaveamento *virtual cut-through*, que tem como objetivo reduzir a latência na comunicação quando um pacote chega a um roteador e o canal por ele desejado encontra-se disponível [32]. O roteador receptor do pacote deve garantir que pode receber todo o pacote, para que o roteador fonte inicie a transferência. Um pacote só é armazenado localmente se o canal desejado por ele estiver indisponível, o que faz com que o roteador tenha obrigação de possuir espaço de armazenamento suficiente para estocar completamente o pacote bloqueado.

Quando propuseram o algoritmo, Cypher e Gravano não consideraram características de redes intra-chip, como o paralelismo natural do hardware, o tamanho das mensagens e dos *buffers*, entre outras. Por esses motivos, propõe-se aqui aspectos de implementação para o algoritmo em NoCs, incluindo:

- **Tamanho das mensagens**: na presente implementação, este pode ser variável até o limite máximo da capacidade dos *buffers*, diferente do que acontece com as implementações atuais para malha desenvolvidas pelo grupo dos autores [20][21], as quais utilizam o algoritmo de roteamento em modo *wormhole*. Nesses casos pode-se ter *buffers* de tamanho menor que a mensagem, já que a mesma não fica armazenada por completo em um único nodo;
- **Paralelismo**: como Cypher e Gravano definiram de modo seqüencial seu algoritmo e módulos de hardware podem operar em paralelo, definiu-se e implementou-se um roteador operando em paralelo com todas as filas e portas.
- **Controle de fluxo**: visando simplificar a implementação, evitou-se o uso de controle de fluxo baseado em créditos, já que as filas definidas no **Algoritmo 1** são centrais, e compartilhadas entre todas as entradas do roteador. Cada fila possui um árbitro dedicado para controlar o acesso a elas. Utiliza-se o controle de fluxo por *handshake*.

2.3 Organização do Roteador: portas e estruturas de memorização

O roteador possui como principais características a quantidade de portas de comunicação, o número de filas para armazenamento temporário. O roteador implementado possui 5 portas bidirecionais de comunicação, sendo uma Local para comunicação com o núcleo IP e outras quatro denominadas Norte, Sul, Leste e Oeste para comunicação com os roteadores vizinhos. Todos os roteadores da rede são idênticos em termos de estrutura interna, ao contrário do que o ocorre em redes com topologia malha.

Internamente, o roteador possui três filas centrais, compartilhadas pelas portas de entrada e saída do roteador. Estas filas são denominadas **fila A**, **fila B** e **fila C**. Todas as 5 portas de entrada possuem direito de escrita na fila A. As quatro portas de entrada comunicação Norte, Sul, Leste e Oeste podem escrever também nas filas B e C. Além

disto a fila A possui permissão de escrita na fila B e esta possui permissão de escrita na fila C. Pacotes fluem para fora da rede apenas a partir da fila C, cuja saída está diretamente conectada à porta de saída Local do roteador.

2.4 Organização do Roteador: algoritmo de roteamento

Alguns conceitos e notações básicas devem ser introduzidos antes de se iniciar o estudo propriamente dito do algoritmo.

- O algoritmo de roteamento utilizado pode operar em redes toro com raiz mista k_{d-1} x k_{d-2} x ... x k_0 e dimensão d, onde d >= 1 e k_i >= 2 para todos i, 0 <= i < d.
- Cada nodo na rede toro possui um identificador único com a forma (a_{d-1}, a_{d-2}, ..., a_0), onde 0 <= a_i < ki para todo i, 0 <= i < d.
- Cada nodo da rede (a_{d-1} ... a_{i+1}, ai, a_{i-1} ... a_0) é conectado a todos os nodos com a forma (a_{d-1} ... a_{i+1}, a_i ± mod k_i, a_{i-1} ... a_0) onde 0 <= i < d.
- As linhas que conectam os nodos da forma (a_{d-1} ... a_{i+1}, k_i - 1, a_i-1 ... a0) e (a_{d-1} ... a_{i+1},0, a_{i-1} ... a_0) são chamados de linhas *wraparound*, e todas as outras são denominadas *linhas internas*.

Para o funcionamento do **Algoritmo 1**, Cypher e Gravano [5] definem dois tipos diferentes de ordenamento total dos nodos em uma rede toro. O primeiro tipo de ordenamento, denominado *crescente à direita* é um ordenamento onde a prioridade cresce ao longo das linhas de nodos da rede, conforme Tabela 1. O segundo tipo de ordenamento, chamado *crescente à esquerda* é exatamente o oposto do crescente à direita, conforme Tabela 2.

Tabela 1: Ordenamento denominado crescente à direita para uma rede toro 8 x 9.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71

Tabela 2: Ordenamento denominado crescente à esquerda para uma rede toro 8 x 9.

71	70	69	68	67	66	65	64	63
62	61	60	59	58	57	56	55	54
53	52	51	50	49	48	47	46	45
44	43	42	41	40	39	38	37	36
35	34	33	32	31	30	29	28	27
26	25	24	23	22	21	20	19	18
17	16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1	0

Para que seja possível entender os algoritmos propostos por Cypher e Gravano, devem-se considerar algumas notações utilizadas pelos mesmos. Considere-se que **p** é um pacote arbitrário que está sendo roteado em uma rede toro, utilizando o **Algoritmo 1**. As convenções adotadas para descrever o algoritmo são resumidas na Tabela 3. O algoritmo propriamente dito é sucintamente descrito a seguir. Mais detalhes encontram-se em [5].

As três filas centralizadas (fila A, fila B e fila C) possuem na presente implementação tamanho arbitrário e parametrizável. Além destas filas, o algoritmo pressupõe a existência de uma **fila de injeção**, por onde os pacotes entram na rede e uma **fila de entrega**, por onde os pacotes saem da rede. Estas duas últimas filas existem na descrição apenas para conveniência de descrição do comportamento da rede, e podem, segundo Cypher e Gravano, não corresponder a estruturas fisicamente implementadas no roteador. Neste trabalho opta-se por não implementar fisicamente estas filas.

Tabela 3: Notação utilizada para descrever o Algoritmo 1 [5].

queue(p)	Fila na qual **p** está localizado.
node(p)	Nodo no qual **p** está localizado.
source(p)	Nodo fonte de **p**.
dest(p)	Nodo destino de **p**.
wait(p)	Conjunto de espera de **p**, que possui as filas para onde o pacote pode ser movido no próximo passo.
neighb(p)	Conjunto de nodos que são vizinhos do **node(p)**.
ok_nodes(p)	Subconjunto de **neighb(p)** que consiste dos nodos vizinhos que estão dispostos em algum caminho mínimo entre **node(p)** e **dest(p)**.
ok_queues(p)	Conjunto de filas em **ok_nodes(p)** que são acessíveis diretamente a partir de **node(p)**.

Considerando que **p** seja um pacote arbitrário que está sendo roteado pelo **Algoritmo 1**, e que **q = queue(p)**, **x = node(x)**, o conjunto de espera (**wait(p)**) é formado a partir da aplicação das regras seguintes:

- **Caso 1**: Se **q** (fila onde o pacote p está armazenado) é a fila de injeção, então o conjunto de espera da fila A consiste de **x**.
- **Caso 2**: Se **q** é a fila A, então existem duas possibilidades:
  1. Se existe um nodo **y** vizinho de **x** que faça parte de algum caminho mínimo, tal que **Right(x) < Right(y)**, o conjunto de espera consiste de todas as filas A que pertencem aos nodos acessíveis diretamente a partir de **x** e que pertencem a algum caminho mínimo do pacote.
  2. Caso a condição anterior não se verifique, o conjunto de espera consiste da fila B do nodo onde o pacote está armazenado.
- **Caso 3**: Se **q** é a fila B, então existem duas possibilidades:
  1. Se existe um nodo **y** vizinho de **x** que faça parte de algum caminho mínimo, tal que **Left(x) < Left(y)**, o conjunto de espera consiste de todas as filas B que pertencem aos nodos acessíveis diretamente a partir de **x** e que pertencem a algum caminho mínimo do pacote.
  2. Caso a condição anterior não se verifique, o conjunto de espera consiste da fila C do nodo onde o pacote está armazenado.
- **Caso 4**: Se **q** é a fila C, então existem duas possibilidades:
  1. Se **x** não é o nodo destino, o conjunto de espera consiste de todas as filas C que pertencem aos nodos acessíveis diretamente a partir de **x** e que pertencem ao caminho mínimo do pacote.
  2. Se **x** é o nodo destino, o conjunto de espera consiste apenas da fila de entrega de **x**.
- **Caso 5**: Se **q** é a fila de entrega, então o conjunto de espera é o conjunto vazio.

Informalmente, pode-se dizer que o **Algoritmo 1** move os pacotes da fila de

injeção para a fila A, sem trocar do seu nodo atual. Depois de estar armazenado na fila A, o pacote move-se para a direita sempre que possível. A seguir, ele move-se para a fila B, sem trocar do seu nodo atual. Quando estiver na fila B, o pacote move-se para esquerda sempre que possível. A seguir, ele move-se para a fila C, sem trocar do seu nodo atual. O pacote permanece transitando entre filas C enquanto não chegar ao seu nodo destino. Quando isto ocorrer, o pacote move-se para a fila de entrega sem trocar do seu nodo atual, ou seja, continuando no nodo destino até que saia da rede.

2.5 Propriedades do Roteamento

O algoritmo de roteamento proposto por Cypher e Gravano [5], empregado em cada roteador da rede toro, possui as características de ser livre de *deadlock*, *livelock*, *starvation*. Ele utiliza sempre o menor caminho entre dois nodos origem e destino (classificando-se então como um algoritmo de *caminho mínimo*) e é *totalmente adaptativo*, ou seja, pode usar qualquer um dos caminhos mínimos existentes entre o nodos fonte e destino.

A funcionalidade do algoritmo tem como principio básico a ordenação dos nodos da rede em função da fila em que o pacote se encontra. Diante desta ordenação o pacote primeiramente, enquanto na fila A, trafega pela rede nos sentidos Norte e Leste. Quando não é mais possível avançar nessa direção por algum caminho mínimo, o mesmo troca de fila, passando para fila B. A ordenação dos nodos é então modificada, fazendo com que o pacote trafegue agora somente nos sentidos Sul e Oeste. Quando não for mais possível o pacote trafegar nessas direções sobre um caminho mínimo ele troca de fila novamente, passando a pertencer a fila C, a qual aceita que o pacote trafegue em qualquer das direções.

Este algoritmo, utilizando-se da ordenação dos nodos e das três filas A/B/C centralizadas, quebra os ciclos presentes em uma rede toro, eliminando assim com a possibilidade de *deadlock* na rede.

O objetivo do presente documento não é descrever em detalhe o Algoritmo 1 de Cypher e Gravano, mas descrever a implementação de um roteador e uma NoC que empregam este algoritmo. Para detalhamento maior deve-se consultar [5] e [1].

3 Projeto da Rede e do Roteador

3.1 Estrutura da Rede

A estrutura geral da rede em que o roteador está inserido comporta um número arbitrário de elementos de processamento conectados entre si através de uma rede toro, conforme ilustra a Figura 3 para o caso de uma rede de raiz 3 e dimensão 2. Essa rede tem como objetivo realizar a comunicação entre os diversos módulos de processamento presentes, através da execução do algoritmo de roteamento 1 descrito por Cypher e Gravano [5].

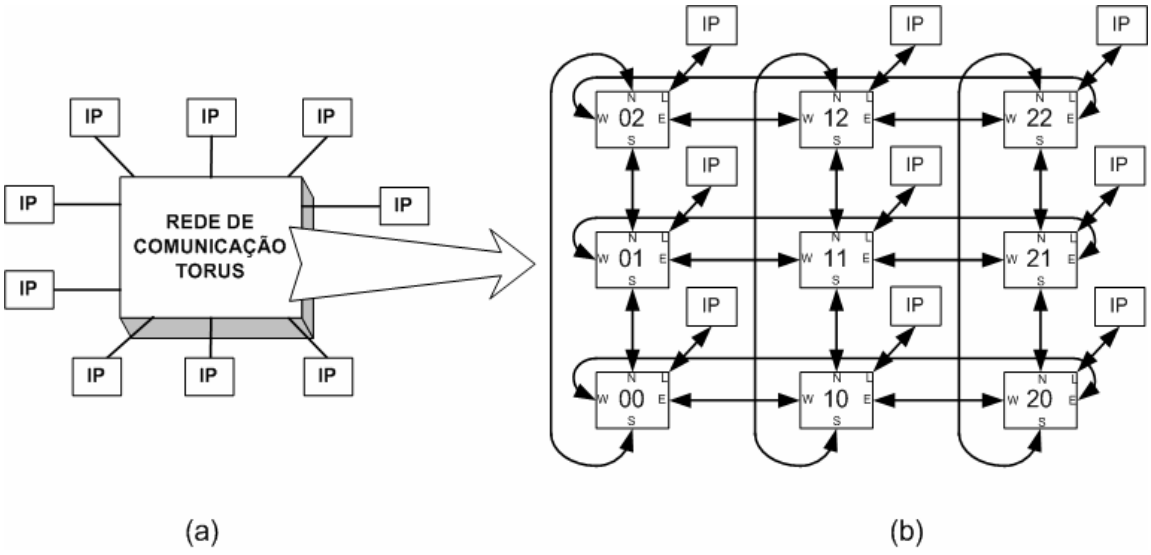

Figura 3 - Em (a) visualiza-se um diagrama de blocos de um sistema que emprega a rede proposta, onde nas extremidades estão os *IP cores* (núcleos de processamento) conectando-se através da rede toro. Em (b) tem-se a estrutura interna da rede toro, mostrando a interconexão dos roteadores entre si. Nota-se também, que em uma rede toro cada roteador emprega todas as dez portas (*N/S/E/W/L*), diferentemente do que ocorre, por exemplo, em uma rede com topologia malha.

3.2 Estrutura do Roteador

Todos os roteadores presentes na rede de comunicação toro possuem a mesma estrutura, já que, conforme visto na Figura 3, a rede é simétrica do ponto de vista destes. A arquitetura geral do roteador proposto aqui aparece na Figura 4. Nela, pode-se notar a presença de dez portas (cinco de entrada e cinco de saída), divididas em pares denominados, Norte, Sul, Leste, Oeste e Local.

As portas de entrada Norte, Sul, Leste e Oeste estão ligadas às entradas de três multiplexadores denominados multiplexadores de entrada, os quais por sua vez, estão ligados cada um a uma das três entradas das filas centrais denominadas *A*, *B* e *C*. Através dessa abordagem, cada uma das portas de entrada não-locais (N/S/E/W) do roteador está ligada diretamente às entradas das três filas. O controle do multiplexador é resultado do processo de operação de 3 árbitros, um para cada fila. A porta Local por sua vez, possui um tratamento distinto, pois segundo o algoritmo de roteamento proposto, ela somente pode enviar pacotes para fila *A*. Por isso, sua conexão na porta de entrada é somente com o multiplexador de entrada da fila *A*. Por motivos similares a saídas das filas *A* e *B* estão conectadas aos multiplexadores de entrada das filas *B* e *C*, respectivamente.

As portas de saída Norte, Sul, Leste e Oeste do roteador, estão ligadas cada uma à saída de um multiplexador de saída, o qual recebe as saídas das três filas, *A*, *B* e *C*. Cada um dos multiplexadores de saída é controlado pela operação de um árbitro, que comanda quando cada multiplexador conecta sua saída a saída de uma dessas três filas. Para a porta Local, o procedimento é um pouco diferente, já que não é necessário a presença de um multiplexador, pois segundo o algoritmo de roteamento ela somente pode receber dados oriundos da fila *C*, e por isso sua porta de saída está diretamente ligada a essa fila.

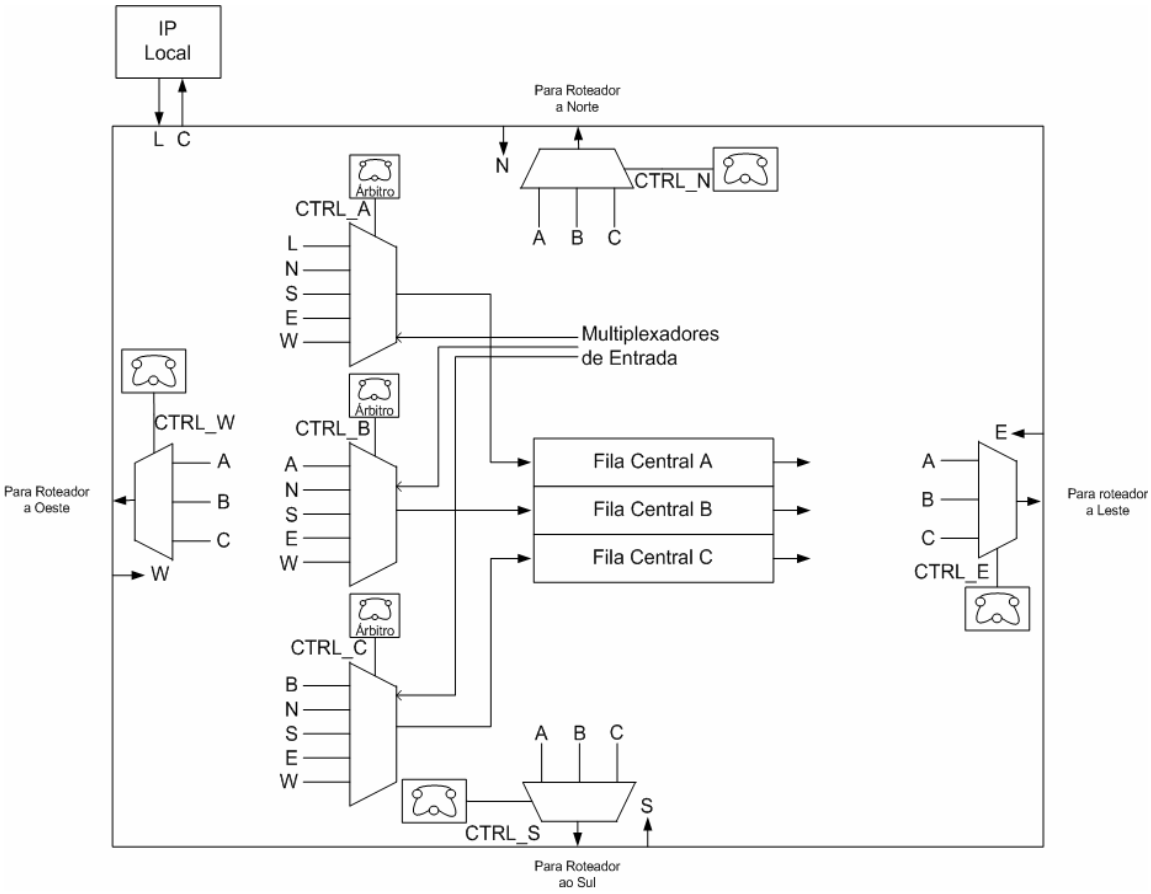

Figura 4 - Estrutura geral do roteador toro proposto.

As filas *A*, *B* e *C*, recebem dados através dos três multiplexadores de entrada, os quais podem vir das portas Norte, Sul, Leste, Oeste e Local. Os dados de cada uma das três filas são enviados para os quatro multiplexadores de saída das portas Norte, Sul, Leste e Oeste. A saída da fila *A*, além de enviar dados para os multiplexadores das portas de saída, também pode enviar para o multiplexador de entrada da fila *B*, assim como a fila *B* também pode enviar dados para o multiplexador de entrada da fila *C*. Já a fila *C*, além de enviar dados para os multiplexadores das portas de saída padrões, envia dados diretamente para a porta de saída Local.

3.2.1 Interfaces

Para viabilizar a implementação do roteador, propõe-se aqui a utilização de dois tipos de interfaces de comunicação, ambas utilizando-se de um protocolo de comunicação *handshake*. A primeira delas, ilustrada na Figura 5, é a interface entre uma das portas não-locais, (*N*,*S*,*E*,*W)* de um roteador com uma das portas não-locais de um roteador vizinho. Nessa interface são definidos três sinais na comunicação entre os nodos, que são *queue_ad*, *data* e *ack*, correspondendo respectivamente ao endereço da fila de destino, o transporte de dados do pacote e o reconhecimento de recepção do dado.

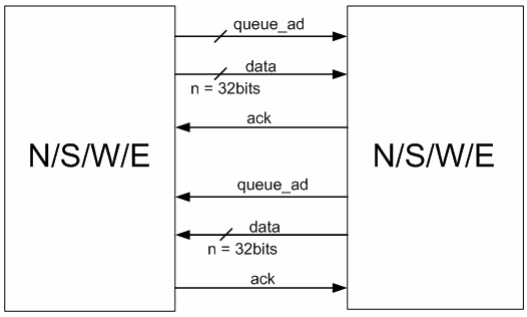

Figura 5 - Interface entre alguma das portas N/S/E/W de um roteador com alguma das portas N/S/E/W de outro roteador qualquer.

O sinal *queue_ad*, informa ao roteador destino para qual das filas (*A*, *B ou C)* o pacote está sendo enviado ou uma indicação de que não há requisição de transferência de dados. Esta decisão de projeto vem do fato que o Algoritmo de Roteamento 1 (descrito por Cypher e Gravano) [5], define que quando um pacote trafega de um nodo para outro na rede, não é possível realizar-se a troca de fila. O dado em si (que representa o conteúdo do pacote), é enviado pelo sinal *data*, ver Figura 5, que possui tamanho parametrizável na implementação proposta. O roteador destino após aceitar e capturar o dado enviado, avisa isto ao roteador origem, através do sinal *ack*.

A segunda interface de comunicação, ilustrada na Figura 6, representa a comunicação entre o *IP* local e o seu roteador. Essa interface contém a ligação da saída da fila *C* para o IP Local e a entrada na fila *A* a partir da saída do IP local. Trata-se de uma interface que pressupõe um protocolo de comunicação assíncrona do tipo *handshake* puro [7].

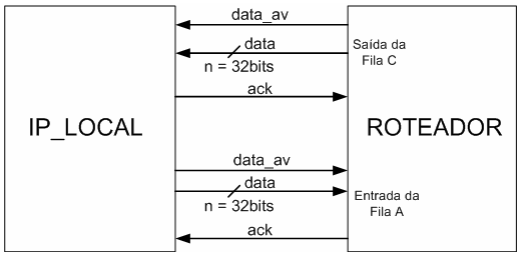

Figura 6 - Interface entre IP local e o seu roteador.

3.2.2 Árbitros

Para que o roteador proposto tenha uma arquitetura capaz de implementar o **Algoritmo 1** [5], propõe-se o uso de dois tipos de árbitro. Os árbitros possuem a função de controlar quem deve e quem não deve acessar um determinado recurso em um dado instante. Os recursos de cada roteador são as três filas centrais (*A*, *B* e *C)* e as saídas do roteador (*N*, *S*, *E*, *W)*. Para o roteador em questão, há necessidade de três árbitros de entrada, um para cada fila central *A*, *B*, *C* e quatro árbitros de saída, um para cada porta não local do roteador (*N*, *S*, *E*, *W)*.

Cada árbitro é uma máquina de estados que opera de forma concorrente com os demais árbitros do roteador. A funcionalidade dos sete árbitros, associada ao restante da estrutura do roteador implementa o Algoritmo 1.

3.2.2.1 Árbitro de Entrada

O árbitro de entrada, ilustrado na Figura 7, possui a função de controlar o multiplexador associado à fila cuja entrada lhe cabe comandar. Por exemplo, imagine-se o árbitro de entrada da fila A, que tem como função dizer a cada instante qual entre as portas de entrada N, S, E, W e L que chegam no seu multiplexador, irá ter direito de acessar a fila. Existem três árbitros de entrada trabalhando de forma concorrente, um para cada entrada de fila central A, B e C. Não existe restrição de acesso simultâneo a filas distintas. Como exemplo, pode-se pensar que simultaneamente se tenha a porta Norte escrevendo na fila A, a porta Sul na fila B e a porta Oeste na fila C.

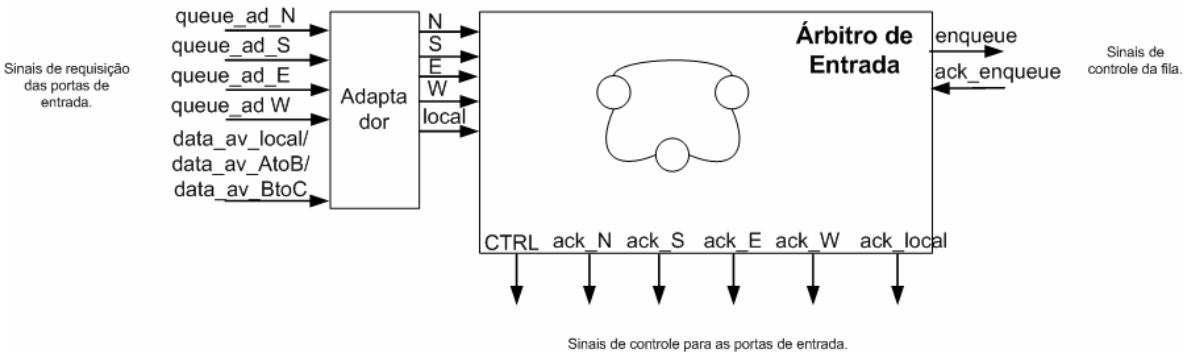

Figura 7 - Árbitro de entrada do roteador.

Cada árbitro de entrada recebe cinco sinais de solicitação de acesso relativos às portas não-locais. Os sinais das portas são os mesmos para todos os árbitros de entrada, que são os sinais das portas de entrada não-locais. O quinto sinal de solicitação de acesso varia de um árbitro para outro. No caso do árbitro da fila *A*, esse sinal é oriundo da porta *L* (local), no árbitro da fila *B*, ele vem da fila *A* indicando uma solicitação de transferência de um dado da fila *A* para fila *B*, e no árbitro da fila *C*, ele vem da fila *B* indicando uma solicitação de transferência de dado da fila *B* para fila *C*.

O árbitro de entrada utiliza um algoritmo de arbitragem com base em prioridade rotativa, denominado *Round Robin*. Ele foi escolhido por questões de simplicidade de implementação em *hardware*. Com esse algoritmo, consegue-se conceder prioridade de acesso de forma simples e na maior parte das vezes justa, para todas as entidades concorrentes.

Quando uma das portas vence o processo de arbitragem, o árbitro envia o sinal *enqueue* para a sua fila (*A*, *B* ou *C)*, solicitando que seja enfileirado o dado especificado pelo algoritmo de arbitragem. Quando a fila realizar a operação de enfileiramento, ela retorna ao árbitro um sinal de *ack_enqueue*, informando que ele já pode selecionar outra porta para acessar a fila.

Finalmente, percebe-se que externamente cada árbitro de entrada possui um bloco a ele associado, denominado adaptador. O adaptador recebe sinais de solicitação de acesso à todas as filas e seleciona gerar na sua saída apenas as solicitações específicas para a entrada da fila controlada pelo árbitro ao qual o adaptador está associado. Assim, cada adaptador é distinto para cada árbitro de entrada. Os árbitros de entrada em si são todos idênticos.

3.2.2.2 Árbitro de Saída

O árbitro de saída (Figura 8), possui a função de controlar qual dentre as saídas das três filas *A*, *B* ou *C*, terá direito a acessar uma das porta do roteador. Cada uma das portas de saída não-locais possui o seu próprio árbitro, sendo que a porta Local por conectar-se somente com a fila *C* não necessita deste.

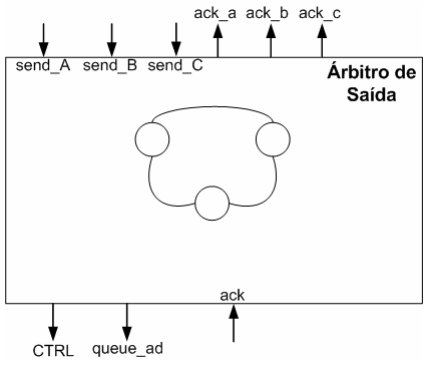

Figura 8 - Árbitro de saída do roteador.

O árbitro de saída recebe em paralelo os sinais enviados pelas três filas do roteador, e deve determinar qual delas terá acesso à porta por ele controlada naquele instante. Nota-se que é possível haver até três portas de saída trabalhando em paralelo, já que se pode ter, por exemplo, a fila *A* enviando dados pela porta Norte, a fila *B* pela porta Sul e a fila *C* pela porta Oeste. O funcionamento de cada árbitro de saída é concorrente, assim como ocorre com os de entrada.

Pelo mesmo motivo adotado anteriormente, utiliza-se um algoritmo baseado em prioridade rotativa, *Round Robin*.

Quando uma das filas ganha o direito de acesso para enviar dados através de uma determinada porta, o árbitro envia um sinal de controle *queue_ad* para o roteador destino, o qual informa a qual fila o dado se destina. Quando o roteador destino armazena o pacote, ele retorna um sinal de reconhecimento *ack* ao árbitro de saída, confirmando o seu recebimento. Nesse momento, a posição ocupada por esse pacote na fila já pode ser liberada. Para que isso aconteça, o árbitro também envia um sinal de *ack* para a fila, informando que o dado já foi enviado.

3.2.3 Armazenamento Temporário

O roteador proposto possui três filas centrais circulares denominadas *A*, *B* e *C*, para armazenamento temporário. A Figura 9 ilustra a estrutura geral de uma fila, onde pode-se visualizar o sinal *data_in* (utilizado para sinalizar a inserção dos dados) entrando na fila, e o sinal *data_out* (utilizado para retirar um dado e enviá-lo para outra fila ou roteador) saindo da fila. Nota-se também a presença dos ponteiros de leitura e escrita na fila. Quando a posição de leitura for igual à posição de escrita, pode-se considerar que a fila está vazia, e quando a posição de escrita for a posição de leitura - 1, a fila está cheia.

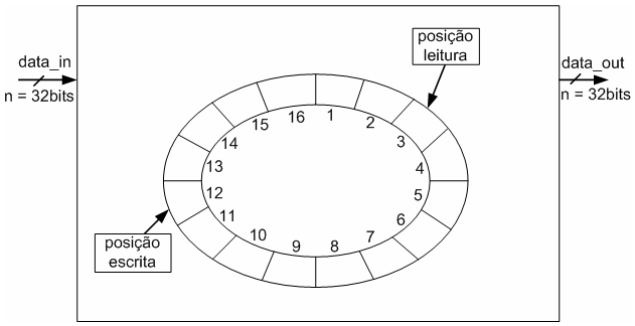

Figura 9 - Estrutura geral das filas, ilustrada para um fila de 16 posições.

A interface de sinais de controle das filas, ilustrada na Figura 10, permite perceber como a comunicação realmente acontece. Nota-se a presença do sinal *enqueue* enviado pelo árbitro de entrada, sinalizando para a fila que existe um dado a ser armazenado, o qual está representado pelo sinal *data_in* de tamanho 32bits. Quando a fila aceita um novo dado, ela confirma esta operação através do sinal *ack_enqueue* para o seu árbitro. Saindo da fila, nota-se a presença dos sinais que vão para o árbitro de saída. O sinal *send_queue* indica ao árbitro que um determinado dado está disponível para ser retirado da fila, e o sinal *data_out* de tamanho 32 bits representa o dado propriamente dito. O árbitro de saída, após enviar o dado recebido pela fila, confirma sua operação através do *ack_queue*, fazendo com que a fila possa descartar esse dado da sua estrutura.

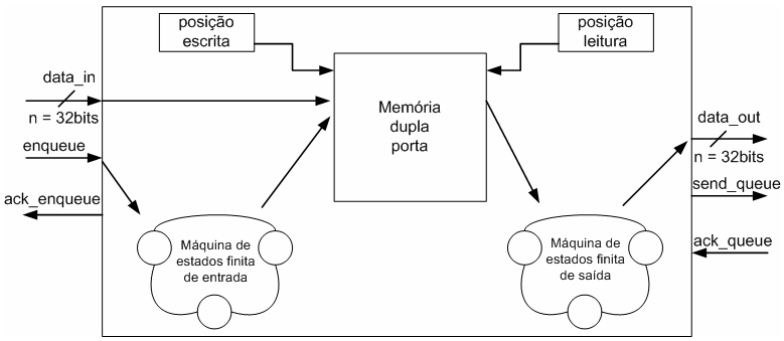

Figura 10 - Interface das filas.

Duas máquinas de estados finitas controlam cada fila. Estas são responsáveis pela liberação ou não do acesso de escrita e leitura. Por exemplo, a máquina de estados de entrada está ligada diretamente ao sinal *enqueue* proveniente do árbitro de entrada, o qual serve como ativador de seu funcionamento. Quando esse sinal indicar que existe um dado para ser armazenado, a máquina de estados deve verificar se a fila está cheia ou não, para que ela possa aceitar o dado e armazená-lo na fila.

A máquina de estados de saída controla os dados da fila que saem através do sinal *data_out*. Ela verifica a cada instante se a fila está ou não vazia. Caso algum dado esteja disponível para ser enviado, ela o disponibiliza e aguarda o recebimento do sinal de *ack_enqueue* do árbitro de saída da porta para qual o dado está endereçado. Após cada envio realizado, ela verifica novamente se a fila possui dados para serem enviados.

4 Implementação no nível de abstração de transação

Esse capítulo descreve a implementação do modelo de referência de uma arquitetura de comunicação do tipo NoC no nível de abstração de transação (em inglês,

transaction level ou TL) mediante emprego da linguagem SystemC . A abordagem será ascendente, iniciando com a implementação do roteador, em seguida descreve-se a implementação da rede e sua validação com Modelsim. Finalmente descreve-se a utilização e proposta de ferramentas de apoio para análise e geração de tráfego.

4.1 Implementação do roteador Mercury-TL

O roteador é um conjunto de módulos que tem por objetivo receber e enviar pacotes baseando-se em um algoritmo de roteamento. Os módulos que compõem o roteador realizam a tarefa de armazenar os pacotes na fila a qual ele é destinado e enviá-los pela porta de destino correta.

Os módulos pertencentes ao roteador são:

- ***queue:*** fila para armazenamento temporário dos pacotes;
- ***arbiterIn:*** seleciona a porta que será atendida em cada momento, seguindo a prioridade Round-Robin;
- ***intoArbiter:*** seleciona os pacotes referentes a fila que eles pertencem;
- ***arbiterOut:*** define a porta na qual o pacote será enviado para chegar em seu destino por um caminho mínimo, baseando-se no algoritmo utilizado.

Para a construção do roteador são utilizados três *intoArbiter*, três *arbiterIn*, três *queue* e um *arbiterOut*. Para realizar a comunicação entre estes módulos foram utilizados canais, de modo a tornar esta comunicação genérica. Os canais são: *into_intoArbiterChl*, que é o canal de entrada do roteador; *intoArbiterChl*, canal de comunicação entre *intoArbiter* e *arbiterIn*; *intoQueueChl*, canal de comunicação entre *arbiterIn* e *queue*; *outFromQueueChl*, canal de comunicação entre *queue* e *arbiterOut*; e *outFromChaveLocalChl*, canal de saída para a porta local que tem como entrada o módulo *arbiterOut*.

Nas próximas subseções será descrito como foram implementados os módulos e canais que compõem o roteador (representada na Figura 11).

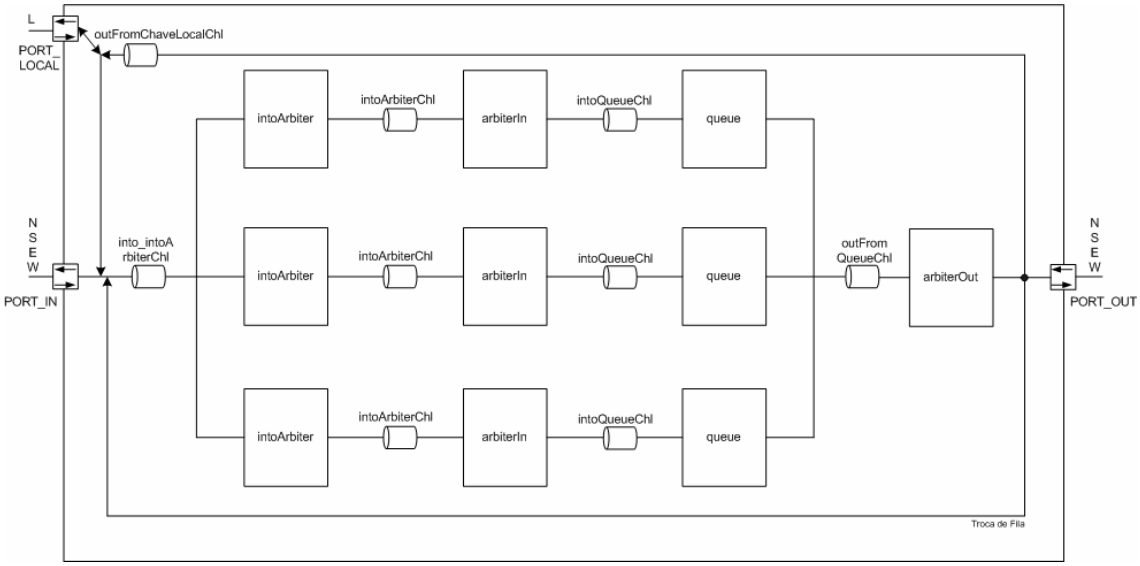

Figura 11 – Estrutura geral do roteador Mercury na implementação TL.

4.1.1 Implementação do canal into_intoArbiterChl

O canal *into_intoArbiterChl* é o canal de entrada do roteador, ele contém funções necessárias para o recebimento e armazenamento temporário de pacotes que vêm de outras chaves ou da porta local.

O canal contém um vetor de quatro posições do tipo TIPODATA (tipo parametrizável que representa o tipo do pacote que circula na rede e que é armazenado nas filas do roteador) chamado *data* para armazenamento temporário de pacotes, cada posição do vetor armazena o pacote de uma única porta do roteador. Quando uma posição é preenchida, a flag *port_busy* indica que a posição está no estado de ocupada (*port_busy* é um vetor de quatro posições do tipo Booleano, que representa as portas do roteador e seta a posição que foi ocupada para verdadeiro) e não permite que a porta correspondente àquela posição seja usada novamente até o armazenamento do pacote em sua fila destino. As quatro posições operam totalmente independentes através do uso de 4 processos paralelos (*threads*) e podem receber pacotes simultaneamente.

Para armazenamento temporário de dados originados da porta local ou de outras filas (no caso de troca de fila) utilizam-se o vetor *data_local*, do tipo TIPODATA e o vetor *local_busy* para marcar as posições utilizadas. Esse vetor contém três posições, cada posição é lida por um *intoArbiter* diferente simultaneamente. Na posição 0 (lida pelo *intoArbiter* correspondente a fila A), é escrito o pacote originado no IP local, na posição 1 (lida pelo *intoArbiter* correspondente a fila B), são escritos pacotes da fila A (para trocas da fila A para fila B) e na posição 2 (lida pelo *intoArbiter* correspondente a fila C) são escritos pacotes da fila B (para trocas da fila B para fila C). Existe uma *thread* diferente para ler cada uma das posições (três *threads*).

Quando o roteador decidir enviar um pacote para um roteador vizinho, este verifica se a porta correspondente do roteador receptor está desocupada e se há espaço disponível na fila destino. Para isso, usa-se a função *isBusy*, que recebe como parâmetro a porta destino do roteador receptor e um valor Booleano passado por referência. Se o valor Booleano retornar falso, significa que a porta está desocupada, existe espaço suficiente para o armazenamento na fila de destino e o pacote pode ser enviado. Caso contrário, a porta já está em uso, ou seja, está em processo de recebimento de outro pacote ou não existe espaço suficiente para o armazenamento do mesmo na fila de destino. O roteador transmissora deve então tentar enviar o pacote por outras portas, se possível (o algoritmo de roteamento determina quais portas estão disponíveis para o envio desse pacote) de maneira circular. Se o algoritmo determinar que existe apenas uma única porta para o envio, o roteador transmissor tentará enviar por essa porta até obter uma resposta permitindo o envio.

Normalmente existe mais de um caminho mínimo possível para enviar um pacote. Portanto tenta-se enviar por um dos caminhos, se este estiver ocupado ou não houver espaço na fila de destino, tenta-se outro, e assim sucessivamente enquanto houver caminhos disponíveis e possíveis, de maneira circular (Round-Robin).

4.1.2 Implementação do módulo *intoArbiter*

O módulo *intoArbiter* é único para cada fila e tem por objetivo analisar e separar os pacotes que chegam no roteador de acordo com sua fila de destino. Cada pacote que está em processo de recebimento no canal *into_intoArbiterChl* tem uma fila de destino

única: se sua origem é um roteador vizinho, a fila destino será a mesma que ele estava armazenado no roteador anterior, se sua origem é a porta local, a fila destino será sempre a fila A. A implementação foi feita desta maneira para possibilitar o uso do algoritmo do Cypher e Gravano [5] para redes Toro. O módulo *intoArbiter* seleciona os pacotes para uma única fila. Se um módulo *intoArbiter* está conectado a fila A, este deve ser configurado na sua criação (através de seu construtor) para selecionar apenas os pacotes destinados para a fila A (lidos do canal *into_intoArbiterChl*) e os enviar para o canal *intoArbiterChl*. Caso contrário nada será feito, pois outro *intoArbiter* (que está conectado com outra fila) vai se encarregar desse pacote.

Resumindo, o objetivo deste módulo é selecionar os pacotes, de acordo com sua fila de destino, para que cada um seja armazenado em sua fila correspondente, eliminando esta tarefa do módulo *arbiterIn*, para que este preocupe-se apenas com o ordenamento do pacotes, segundo a prioridade Round-Robin, recebidos por ele.

4.1.3 Implementação do canal *intoArbiterChl*

O canal *intoArbiterChl* realiza a comunicação entre o módulo *intoArbiter* e o módulo *arbiterIn*, é único para cada fila e sua função é armazenar temporariamente os pacotes que trafegam entre os dois módulos mencionados. O pacote será inserido em uma posição de memória do canal (do tipo TIPODATA) pelo módulo *intoArbiter* e será retirado quando receber um sinal do módulo *arbiterIn* liberando aquela posição, indicando que o pacote já foi inserido em sua fila de destino. É nele também que está implementada a função que realiza o controle de prioridade necessário para que o módulo *arbiterIn* atenda todas as portas igualmente, de modo a não permitir que ocorra postergação indefinida de uma porta. Esta função denomina-se *read_request_port*, ela recebe como parâmetro o pacote, a porta e um valor Booleano, todos passados por referência. No parâmetro pacote, é armazenado o pacote, no parâmetro porta, é armazenada a porta pela qual o pacote entrou no roteador, e no valor Booleano armazena-se se há ou não algum pacote para ser lido no momento da chamada da função. Cada vez que a função é chamada, verificam-se as portas na ordem: norte, sul, leste, oeste e local, para determinar se há algum pacote destinado à fila que o canal opera. Se houver, o pacote será lido, e o endereço dessa porta será armazenado na variável *current_priority_port*. Na próxima vez que a função for chamada, as portas serão verificadas novamente na mesma ordem citada, mas começando pela porta posterior à porta em que o pacote foi lido na última vez que a função foi chamada (usa-se o endereço da porta armazenado na variável *current_priority_port*). Dessa forma nenhuma porta terá mais prioridade do que as outras, esta prioridade é denominada *Round-Robin*.

Este canal é único para cada fila. Todos os pacotes que são armazenados nele, são oriundos do módulo *intoArbiter*, que já os filtrou de acordo com sua fila de destino.

4.1.4 Implementação do módulo *arbiterIn*

O módulo *arbiterIn* é responsável por ler os pacotes, um por vez, do canal *intoArbiterChl* e por enviá-los para o módulo *queue* através do canal *intoQueueChl* com prioridade *Round-Robin*. No momento do envio de um pacote, o módulo fica aguardando seu armazenamento na fila, para então ler outro pacote, se houver.

Existe um único módulo *arbiterIn* para cada fila, que apenas lê e escreve pacotes

destinados para a fila a qual ele pertence, pois o pacotes lidos do canal *intoArbiterChl* já foram filtrados pelo módulo *intoArbiter*.

4.1.5 Implementação do canal *intoQueueChl*

O canal *intoQueueChl* armazena apenas um pacote, e espera até que ele tenha sido armazenado na fila (módulo *queue*), para poder armazenar outro pacote.

4.1.6 Implementação do módulo *queue*

Este é o módulo que implementa a fila e contém as funções para controlar o armazenamento de pacotes no roteador. A fila implementada é uma fila circular de porta dupla, na qual os pacotes que chegam são armazenados na primeira posição livre e os pacotes que saem são retirados da primeira posição ocupada, operações que podem ser realizadas de forma concorrente. Dois processos implementam essa funcionalidade: o processo de entrada, responsável por verificar se a fila está em condições de receber um novo pacote para armazenamento (através do qualificador *fila_cheia*) e calcular a próxima posição de armazenamento. No momento em que o pacote entra na fila, ele estará ocupando duas filas simultaneamente por um tempo finito: a fila transmissora e a fila receptora.

O outro processo é o de saída, que tem como função enviar o primeiro pacote da fila, esperar o sinal que indica que ele foi armazenado no roteador receptora e calcular a posição em que se encontra o próximo pacote a ser enviado. Uma posição da fila só é acessada depois de verificada o qualificador *fila_vazia*, que informa se existe algum elemento na fila.

Um *mutex* impede que os processos de entrada e saída alterem ao mesmo tempo os qualificadores que indicam o estado da fila (*fila_cheia* e *fila_vazia*). Uma posição ocupada da fila nunca será sobrescrita antes ser lida, pois processos que querem escrever na fila verificarão se existe espaço livre para isso antes de tentar escrever.

Depois de armazenado o pacote, a porta de entrada pela qual ele chegou é liberada para receber novos pacotes e a fila transmissora pode recomeçar a enviar pacotes. Quando a fila não está vazia o processo de saída escreve o primeiro pacote da fila no canal *outFromQueueChl*, e espera até que ele seja armazenado na fila do roteador seguinte, ou se for destinado à porta local, espera até que se confirme sua chegada ao *IP_local*.

4.1.7 Implementação do canal *outFromQueueChl*

O canal *outFromQueueChl* implementa a comunicação entre os três módulos *queue* contidos no roteador e o módulo *arbiterOut* através de um vetor de três posições para armazenamento temporário de pacotes (do tipo TIPODATA) chamado data. Cada posição do vetor é dedicada a uma fila. Os três módulos *queue*, quando possuem pelo menos um dado armazenado, o enviam para a posição que o pertence neste canal. Quando um pacote é escrito em uma das posições do vetor data, o canal *outFromQueueChl* espera até que a escrita do pacote seja realizada em seu destino (roteador vizinho ou outra fila do mesmo roteador no caso de troca de filas), após isto, o canal está de novo apto para receber um novo pacote naquela posição.

Existe apenas um canal *outFromQueueChl* para as três filas, pois neste ponto, todos os pacotes devem passar pelo módulo *arbiterOut*, que é o módulo que lê os dados deste canal, para serem roteados.

4.1.8 Implementação do módulo *arbiterOut*

O módulo *arbiterOut* é o responsável por enviar os pacotes para as chaves vizinhas, para outra fila na mesmo roteador ou para o *IP_local* através da porta selecionada. Neste módulo existem três processos de envio, um para cada fila, que podem cada um enviar um pacote distinto, desde que para portas distintas. Este módulo contém uma referência ao algoritmo de roteamento que está sendo utilizado. Portanto, antes do envio do pacote, o algoritmo será consultado e retornará uma ou mais portas para as quais o pacote poderá ser enviado. Pergunta-se a cada porta seqüencialmente, se é possível enviar um pacote por ela através da função *isBusy*, implementada no canal de entrada do roteador (*into_intoArbiterChl*). Caso afirmativo, envia-se o pacote. Caso contrário, pergunta-se se é possível enviar o pacote à próxima porta retornada pelo algoritmo (de forma circular), até que seja encontrada uma porta livre.

No caso do algoritmo retornar a porta local como porta destino, o módulo leva em consideração a fila em que o pacote se encontra. Segundo o algoritmo do Cypher e Gravano [5], um pacote só poderá ser entregue no *IP_local* do roteador se estiver na fila C. Então, se o pacote encontra-se na fila A e recebe a ordem de ser enviado para a porta local, significa que ele irá para o fim da fila B. Se estiver na fila B, irá para o fim da fila C. Somente após, ele será direcionado para o *IP_local*.

4.1.9 Implementação do canal *outFromRoteadorLocalChl*

É o canal de comunicação entre a saída do roteador e a entrada do *IP_local* da mesma. Tem apenas uma posição para armazenamento temporário de pacotes (do tipo TIPODATA), pois somente um pacote da fila C poderá ser escrito neste canal de cada vez. No momento que a posição é ocupada, o canal aguarda até que o *IP_local* retire o pacote dessa posição, para que ela fique livre para um novo armazenamento.

4.2 Implementação da rede

Essa Seção descreve a implementação da comunicação entre as chaves bem como a disposição de chaves para implementar uma topologia do tipo Toro.

4.2.1 Interface de comunicação entre as chaves

A comunicação entre as chaves acontece entre o módulo *arbiterOut*, localizado na saída do roteador transmissor e o canal *into_intoArbiterChl* do roteador receptor. Esta comunicação é implementada através de funções de envio e recebimento de pacotes e funções de controle de fluxo. O tipo de transferência de dados é bloqueante, isto é, quando um pacote é enviado, o processo emissor fica aguardando até receber a confirmação de recebimento pelo receptor.

As funções utilizadas para a realização da escrita em um roteador estão localizadas na interface de entrada (*into_intoArbiterInIf*) do canal *into_intoArbiterChl* e são:

- ***write_queue_addr***: é utilizada pelo módulo *arbiterOut* do roteador transmissor e recebe como parâmetros um pacote, o endereço da fila destino e a porta através

da qual o pacote será acolhido no roteador receptor. A fila na qual o pacote está armazenado no roteador transmissor fica bloqueada até que o pacote chegue na fila destino do roteador receptor. Para tanto essa função faz uso de semáforos e, assim que um pacote chega no canal, ele é bloqueado até que receba um sinal da fila destino, indicando sua chegada e armazenamento.

- ***write_local***: é utilizada pelo módulo *IP_local* para escrever no roteador. Como todos os pacotes que vêm do *IP_local* estão sendo injetados na rede, eles devem ser direcionados para a fila A, segundo o algoritmo em [5]. A função *write_local* simplesmente utiliza a função *write_queue_addr* descrita acima, mandando o endereço da fila A como parâmetro de endereço de fila.
- ***isBusy***: essa função é utilizada pelo módulo *arbiterOut* antes de enviar um pacote para outro roteador para saber se a porta a ser utilizada está livre e se a fila que vai receber o pacote tem espaço necessário para acolhê-lo. Recebe como parâmetros o endereço da porta a ser utilizada para o recebimento, o endereço da fila que vai armazenar o pacote e uma variável Booleana (*busy*) por referência. Quando a porta está livre e a fila pode armazenar o pacote, a variável *busy* retorna falso, caso contrário retorna verdadeiro.

Algumas funções não estão relacionadas com a comunicação entre chaves, mas sim com a troca de filas em uma mesmo roteador. Essas funções são importantes para o funcionamento do algoritmo usado, que utiliza filas centrais.

Estas funções também estão localizadas na interface de entrada do canal *into_intoArbiterChl* e são:

- ***write_queue_change***: é utilizada pelo módulo *arbiterOut* da próprio roteador para realização da troca de fila. Recebe como parâmetros um pacote e o endereço da fila destino. Neste caso, como no caso da função *write_queue_addr*, também utilizam-se semáforos para controle da comunicação. A fila transmissora fica bloqueada até receber o sinal de recebimento da fila destino.
- ***isBusyLocal***: tem o mesmo objetivo da função *isBusy*, porém não testa se alguma porta está livre, apenas se há espaço disponível na fila destino. Recebe como parâmetros o endereço da fila destino e uma variável Booleana (*busy*) por referência.

4.3 Validação com Modelsim

Essa seção descreve a estrutura de validação utilizadas para a validação da NoC TL em SystemC. A validação com a ferramenta Modelsim foi realizada inicialmente de forma *ad hoc*, pela análise de dados impressos na tela.

4.3.1 Testbenchs utilizados

▪ ***fila_testbench***: este *testbench* foi utilizado para validar o funcionamento de uma fila (módulo *queue*) e dos canais de entrada (*intoQueueChl*) e saída da fila (canal *outFromQueueChl*), cuja interligação é ilustrada na Figura 12. O gerador de estímulos é o módulo *test_intoQueue* que conecta-se ao canal de entrada da fila. Este contém um processo que insere um determinado número de pacotes no canal (definido pela

constante NUM_PACOTES localizada no arquivo *defs.h*). A partir do momento em que o pacote é escrito no canal e enquanto ele não chega na fila, o processo de envio fica bloqueado. A partir do momento em que este é armazenado, outro pacote pode ser enviado. O capturador de saídas é módulo *test_outQueue* que conecta-se ao canal de saída e tem um processo que retira pacotes do mesmo. Caso não haja pacotes disponíveis, o canal bloqueia o processo até que um pacote seja inserido. As constantes DEBUG_TESTBENCH e DEBUG_QUEUE devem possuir valor 1 no arquivo *defs.h* para visualização da entrada e saída de pacotes no sistema e entrada, saída e permanência de pacotes na fila.

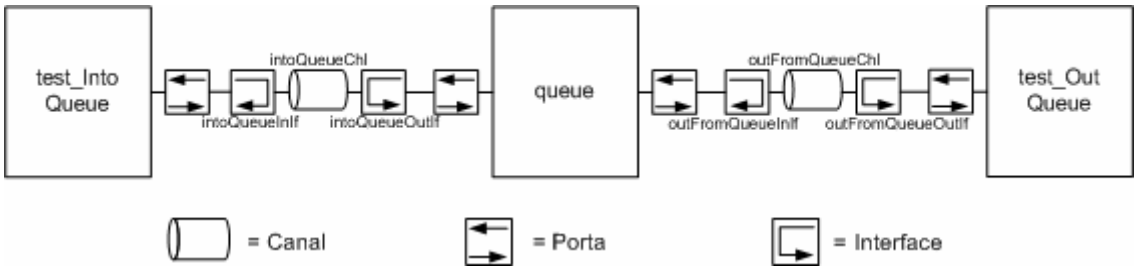

Figura 12 - Estrutura de validação da fila e seus canais de acesso (fila_testbench).

▪ ***roteador_parcial_testbench***: utilizado para validar o funcionamento dos módulos que *intoArbiter* e *arbiterIn* (descritos nas Seções 4.1.2 e 4.1.4). O gerador de estímulos (módulo *test_intoChave*) conecta-se a entrada do roteador (canal *into_intoArbiterChl*) e tem por objetivo simular o envio simultâneo de pacotes por diferentes portas (através de cinco processos, um para cada porta) com diferentes filas de destino. O capturador de saídas (módulo *test_outChave*) tem três processos que retiram pacotes do canal de saída das três filas (*outFromQueueChl*) de forma concorrente.

No arquivo defs.h deve-se ativar (colocar em 1) as constantes: DEBUG_TESTBENCH, para visualizar a entrada e saída de pacotes (nos dois módulos de testbench) e a fila para a qual o mesmo é destinado; DEBUG_CANAL, para verificar o caminho que os pacotes estão seguindo dentro do roteador; e DEBUG_QUEUE, para verificar sua chegada e permanência na fila.

A estrutura deste testbench é ilustrada na Figura 13.

▪ ***roteador_completo_testbench***: utilizado para validar o roteador inteiro, em especial o módulo *arbiterOut* em conjunto com o algoritmo de roteamento funcionando em apenas um roteador, isto é, verifica-se se este está mandando os pacotes pela porta correta segundo o algoritmo. Para esta validação, utiliza-se um roteador completo.

No arquivo *top_testChaveCompleta.h*, que contém o módulo *top-level* da estrutura ilustrada na Figura 14, é possível informar a posição deste roteador em uma rede (através das variáveis *pos.x* e *pos.y*) e o tamanho da rede em x e y (variáveis *tam_noc.tam_x* e *tam_noc.tam_y* respectivamente) na qual ela está inserida. No arquivo *defs.h*, a constante DEBUG_TESTBENCH deve valer 1 e a variável NUM_PACOTES deve conter o número de pacotes que cada porta do roteador irá receber.

O gerador de estímulos é o módulo *test_intoChave*, que coloca em cada porta de entrada a quantidade de pacotes definidos por NUM_PACOTES, cada um com um valor distinto (para viabilizar o acompanhamento do trajeto dos pacotes pelo roteador e sua saída pela mesma). Além disto, o gerador escolhe o roteador destino e a fila de

forma aleatória. Cada uma da cinco portas recebe um pacote de cada vez de forma concorrente devido aos cinco processos que rodam no módulo *test_intoChave*. Os capturadores de saída são o canal *test_outChave* e o módulo *test_outChaveLocal*. O *test_outChave* é um canal do tipo *into_intoArbiterChl* com funções diferenciadas, a fim de atender os objetivos do *testbench*. Este recebe os pacotes que saem do módulo *arbiterOut*, simulando uma recepção em um roteador vizinho, imprimindo na tela o valor do pacote (que funciona como identificador) e o endereço da porta na qual o pacote foi recebido, além de incrementar a quantidade de pacotes recebidos. O módulo *test_outChaveLocal* tem como funcionalidade receber pacotes destinados à porta local do roteador, isto é, pacotes que entraram no roteador com destino ao *IP_local* desta.

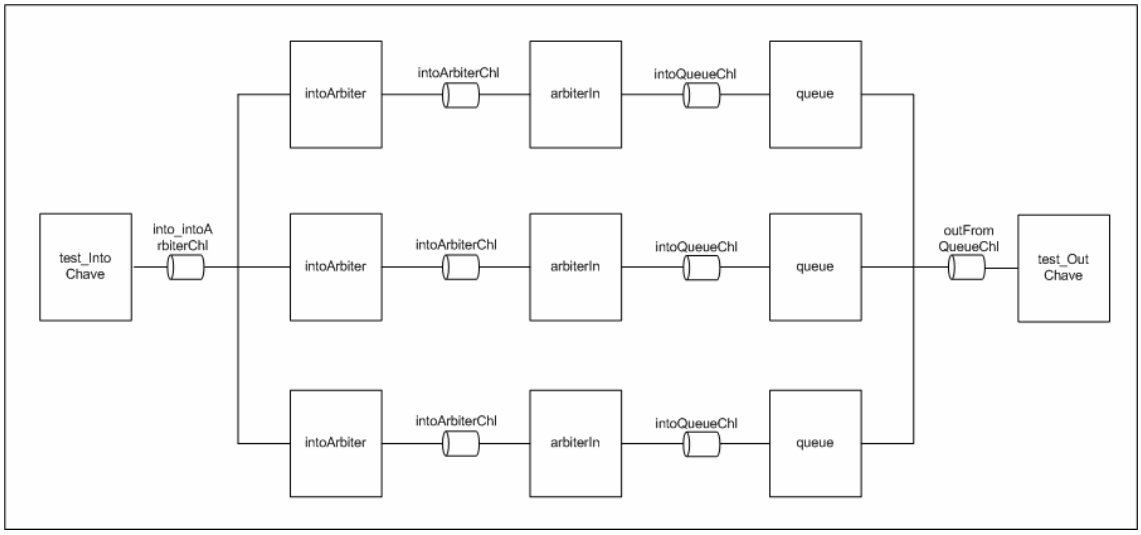

Figura 13 - Estrutura de validação dos módulos intoArbiter, arbiterIn, queue e dos canais into_intoArbiterChl, intoArbiterChl, intoQueueChl, outFromQueueChl em conjunto (roteador_parcial_testbench).

Para conferir se todos os pacotes foram roteados, ou seja, que todos que entraram no roteador também saíram, basta multiplicar o valor de NUM_PACOTES por cinco (são cinco processos que enviam pacotes) e comparar o resultado com a soma dos últimos valores de cada porta no módulo *test_outChave* e o último valor no módulo *test_outChaveLocal*, que são os únicos módulos de saída do sistema neste *testbench*.

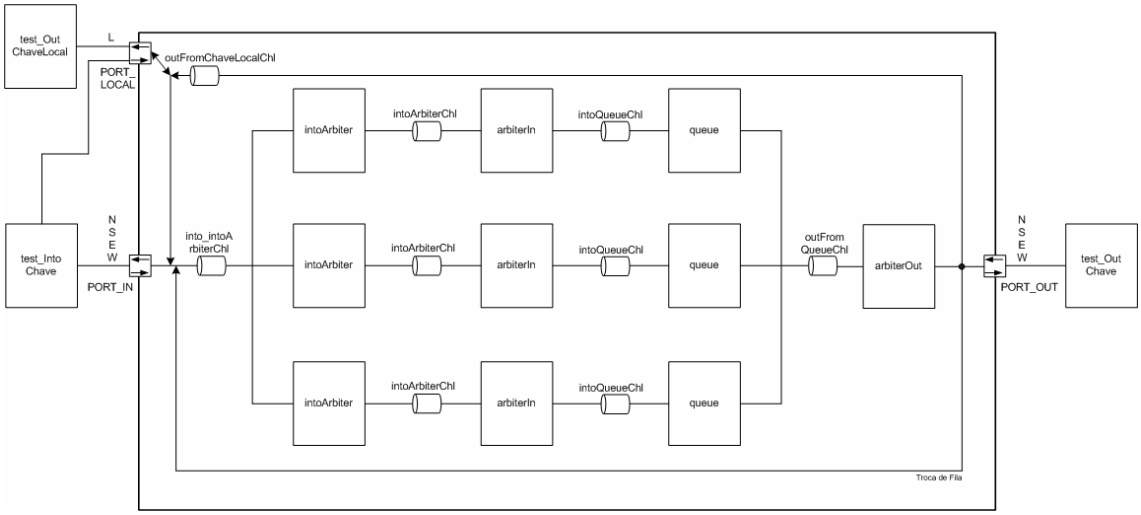

Figura 14 - Testbench do roteador completo.

▪ ***noc_tamX_x_tamY***: *testbenches* utilizando várias chaves conectadas representando uma NoC toro bidimensional X por Y. Estes *testbenches* servem para validar o correto fluxo de dados na rede, testar o algoritmo de roteamento mais a fundo e verificar se o projeto funciona em diferentes situações (listadas na Seção 4.3.2). Como estes tipos de *testbench* geram um grande volume de pacotes, foi criado um protocolo de comunicação que é interpretado por uma ferramenta, a ser apresentada na Seção 5. O gerador de estímulos e capturador de saída é o módulo *IP_local*, que está conectado à entrada e saída da porta local de cada roteador e simula um IP que gera e recebe pacotes, além de definir o destino dos pacotes gerados. A quantidade de pacotes gerados no sistema é a constante NUM_PACOTES (definida no arquivo *defs.h*) multiplicada pelo número de chaves.

Apenas a constante SAIDA_ANALIZADOR deve valer 1 para que seja gerado o arquivo de entrada da ferramenta de análise.

4.3.2 Testes realizados

Nesta Seção serão listados seis dos testes realizados em uma NoC de tamanho 4 por 3 bidirecional com fila de profundidade de 5 elementos.

- Roteador [1,1] enviando 150 pacotes para o roteador [2,1]. Teste realizado para verificar o funcionamento do envio de pacote para um roteador vizinho.
- Roteador [1,0] enviando 150 pacotes para o roteador [3,2]. Teste realizado para verificar o funcionamento do *wraparound* vertical através do caminho [1,0] → [1,2] → [2,2] → [3,3] ou horizontal através do caminho [1,0] → [0,0] → [3,0] → [3,2].
- Roteador [0,0] enviando 150 pacotes para o roteador [1,2]. Teste realizado para verificar o funcionamento do *wraparound* horizontal e também de um caminhamento do pacote pela rede de forma horizontal e vertical através dos caminhos [0,0] → [1,0] → [1,2] ou [0,0] → [0,2] → [1,2].
- Roteador [1,0] e [1,2] enviando 150 pacotes cada para o roteador [3,2]. Este teste foi realizado para visualizar os pacotes que saem do roteador [1,0] tomando diferentes caminhos, já que o roteador por onde eles sempre passavam (roteador [1,2]) agora está congestionado.
- Todas as chaves enviando para o roteador [1,0]. O teste totaliza 150 pacotes por roteador, e 1800 pacotes na rede como um todo. Teste realizado para visualizar o comportamento de um roteador lidando com pacotes chegando de todas as direções e fontes de dados da rede.
- Todos os roteadores enviando para todas as chaves 350 pacotes cada, totalizando 4200 pacotes na rede. Teste realizado para visualizar o comportamento geral de uma rede.

5 Ferramentas para a NoC Mercury

5.1 Gerador Júpiter

A ferramenta de geração da NoC Mercury tem por finalidade permitir ao desenvolvedor gerar NoCs Mercury tanto em nível TL quanto RTL de forma simples.

Ela permite que sejam escolhidas as dimensões da NoC desejada (nos eixos X e Y) e também o nível de abstração, conforme se pode visualizar na Figura 15. Até o presente momento somente roteamento usando a opção “Algoritmo CG” está incluído.

Para gerar uma NoC Mercury, basta selecionar o diretório onde se deseja colocar os arquivos resultantes, escolher a dimensão X e Y e o nível de abstração.

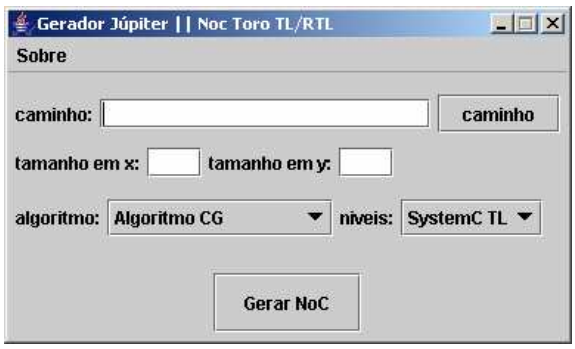

Figura 15 - Gerador Júpiter – Tela inicial da ferramenta para geração de NoCs Mercury nível TL e RTL.

5.2 Analisador Júpiter

A ferramenta de análise de tráfego da NoC Mercury é uma ferramenta que tem por finalidade permitir ao desenvolvedor efetuar análises específicas de certos aspectos do tráfego de uma simulação em SystemC em nível de transação. Estes aspectos são: quantidade de pacotes enviados e recebidos, caminhamento de pacotes, quais pacotes chegaram e quais não chegaram ao destino, entre outros.

Esta ferramenta foi gerada devido à necessidade de analisar grandes quantidades de pacotes trafegando em uma instância da NoC Mercury. Em *testbenches*, é possível visualizar os pacotes trafegando dentro de um roteador, nas filas e até mesmo na rede. Contudo é inviável analisar o tráfego em uma rede sem um processo sistemático de tratamento do grande volume de dados intermediários gerados durante uma simulação realista da rede. Para facilitar tais análises, foi proposta e implementada uma ferramenta de análise e um protocolo de comunicação (que será explicado mais adiante) entre a simulação da NoC e a ferramenta. A Figura 16 mostra a janela inicial da interface gráfica da ferramenta.

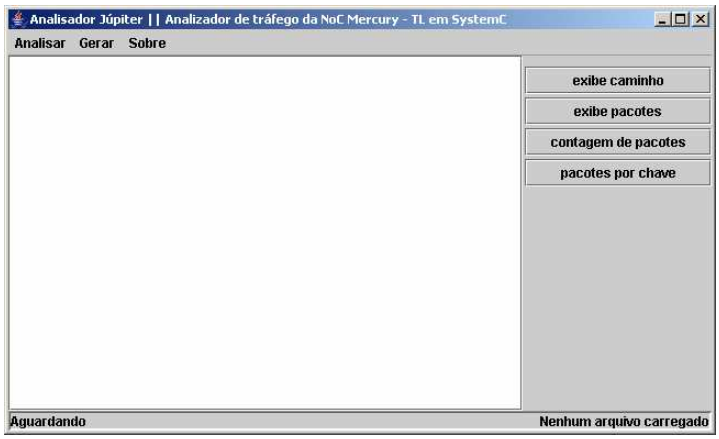

Figura 16 - Tela inicial da interface gráfica da ferramenta de análise de tráfego da NoC Mercury.

O arquivo de entrada para a ferramenta é o arquivo gerado pela simulação da NoC Mercury com as constantes no arquivo *defs.h* configuradas para criar o arquivo para a ferramenta (ver Seção 4.3.1).

As funções para manipulação de arquivos estão no menu *Arquivo*. A opção *abrir* desse menu serve para carregar o arquivo de entrada, que será analisado com as funções dos botões à direita da tela principal; O botão *exibe caminho* mostra o caminho seguido por um pacote na rede, desde a sua criação até sua saída da rede mostrando todas as trocas de fila internas ou entre chaves. O botão *exibe pacotes* mostra os pacotes que chegaram e os que não chegaram ao seu destino. O botão contagem de pacotes mostra a quantidade de pacotes que foram gerados e a quantidade que chegou ao destino. O botão *pacotes por roteador* mostra a quantidade de pacotes que entrou e saiu de cado roteador. A opção *salvar saída* do menu *Arquivo* salva em um arquivo a tela corrente. A opção *fechar* fecha o arquivo que está sendo analisado.

O menu *Gerar* contém funções destinadas à geração de uma instância da NoC Mercury. A versão atual da ferramenta (0.1) contém apenas uma chamada para a ferramenta *Gerador Júpiter*, apresentada na Seção 5.1.

Para o funcionamento e análise de um tráfego através da ferramenta *Analisador Júpiter* necessita-se abrir um arquivo de entrada com um padrão pré-definido. Esse arquivo é gerado ao final da simulação da NoC Mercury TL realizada no Modelsim. Ele deve possuir um padrão pré-definido através de um protocolo que consiste em quatro tipos de informações, as quais são:

1. **Entrada**: [E] [roteador local] [porta entrada] [fila] [roteador destino]/[roteador local]
   - [E] identifica que essa linha informa a entrada de um pacote no roteador;
   - [roteador local] identifica o endereço do roteador na qual o pacote se encontra atualmente;
   - [porta entrada] identifica a porta através da qual o pacote entrou no roteador;
   - [fila] identifica a fila na qual o pacote entrou;
   - [roteador destino] identifica o roteador para a qual o pacote se destina. Este comando é impresso no instante em que o pacote identifica que pode entrar no roteador (verificação de espaço de fila e liberação da porta);
   - [roteador local] identifica que o pacote está destinado à porta local do roteador. Essa sintaxe quando presente substitui a [roteador destino].

2. **Troca de fila**: [T] [roteador local] [porta entrada] [fila destino] [roteador destino]
   - [T] identifica que essa linha informa uma troca de fila dentro do roteador;
   - [roteador local] definido anteriormente;
   - [porta entrada] definido anteriormente;
   - [fila destino] identifica a fila na qual o pacote deve ser colocado. Este comando é impresso no árbitro de saída (*arbiterOut*), no momento que o pacote é escrito na entrada do canal de entrada (*into_intoArbiterChl*) do (mesmo) roteador;

- [roteador destino] identifica o roteador para o qual o pacote está destinado.

3. **Saída**: [S] [roteador local] [porta entrada] [fila] [porta saída] [roteador destino]
   - [S] identifica que essa linha informa a saída de um pacote do roteador;
   - [roteador local] definido anteriormente;
   - [porta entrada] definido anteriormente;
   - [fila] definido anteriormente;
   - [porta saída] é a porta que o pacote usou para sair do roteador. Este comando é impresso logo após a verificação de que existe espaço livre na fila do roteador vizinha e que a porta está livre para transmissão;
   - [roteador destino] definido anteriormente.

4. **Informação**: [I] [tamanho da NoC]
   - [I] identifica que essa linha possui informação sobre as dimensões da rede;
   - [tamanho da NoC] identifica o tamanho em x e em y da NoC, é a primeira linha impressa no arquivo de saída. Ex: [I] [34] representa que esta é uma NoC de tamanho 3 em x e 4 em y.

As portas de entrada e de saída são identificadas numericamente, sendo a representação feita com a forma descrita a seguir. A porta Norte é representada pelo algarismo “0”, a porta Sul por “1”, a porta Leste por “2”, a Oeste por “3” e a porta Local pelo número “4”. O mesmo tipo de representação é atribuído às filas da seguinte forma: A fila A é representada pelo algarismo “0”, a fila B por “1” e a fila C por “2”. O protocolo de entrada da ferramenta pode ser visualizado através do exemplo listado abaixo:

```
[I] [34]
...
[E] [03] [4] [0] [20]
...
[S] [03] [4] [0] [0] [20]
...
[E] [00] [1] [0] [20]
...
[S] [00] [1] [0] [3] [20]
...
[T] [20] [2] [1] [20]
...
[T] [20] [4] [2] [20]
...
[S] [20] [4] [2] [4] [20]
...
```

As linhas acima mostram o caminho de um pacote com origem no roteador 03 e destino no roteador 20 em uma rede com dimensões x = 3 e y = 4, conforme se pode visualizar na Figura 17. Essas linhas foram retiradas diretamente de um arquivo gerado pelo Modelsim, e entre elas existiam outras que foram ocultadas, através da colocação de elipses (...) no lugar das mesmas, para facilitar a leitura do caminho desse pacote em demonstração.

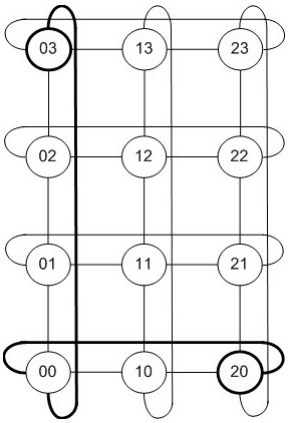

Figura 17 - Caminho percorrido pelo pacote demonstrado no exemplo acima, do nodo 03 ao nodo 20 em uma NoC 4x3.

O *Analisador Júpiter* demonstra o caminho decorrido por este pacote de uma forma um pouco diferente, porém mais clara como demonstra a Figura 18.

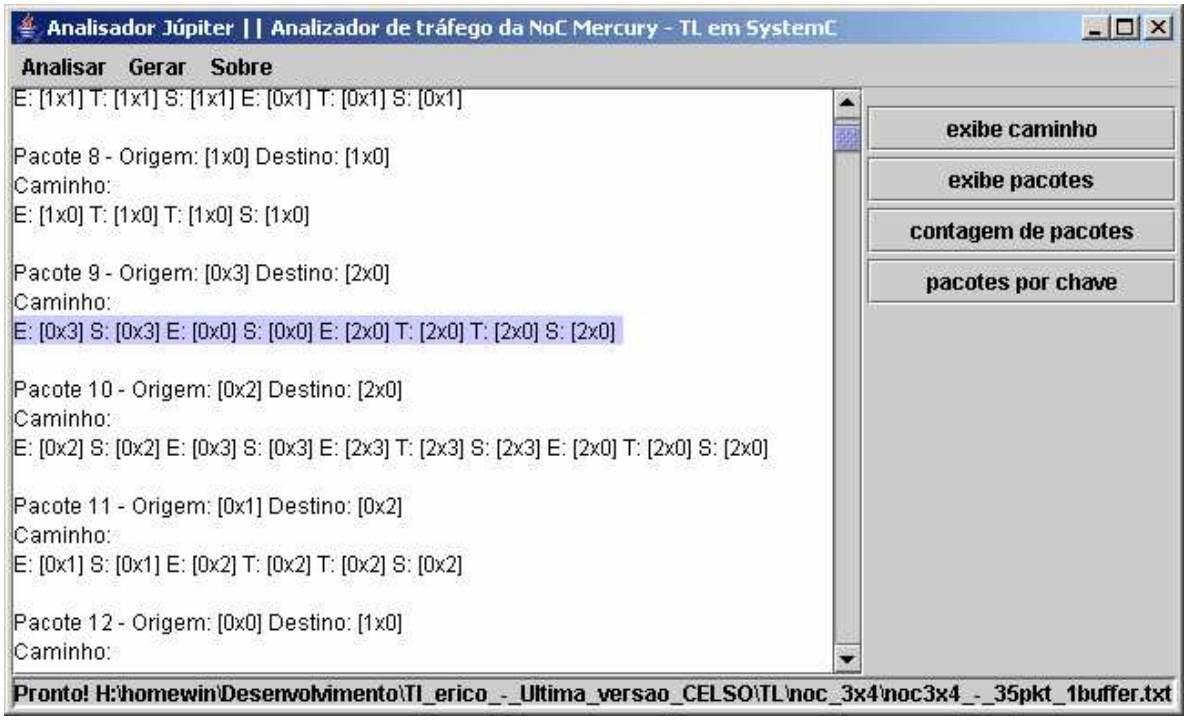

Figura 18 – Exemplo de saída gerada pelo Analisador Júpiter. O texto salientado mostra o caminho do pacote 9, mostrando o caminho seguido entre a origem, no roteador 03 e o destino, roteador 20. No caso, o pacote entrou na rede pelo roteador 03 (fila A) e saiu deste, sem trocar de tipo de fila, para o roteador 00. Em seguida, mais uma vez sem trocar de tipo de fila, o pacote saiu do roteador 00 para o destino, roteador 20. Neste último, o pacote troca duas vezes de tipo de fila (da A para a B e desta para a C). Após isto, pacote sai da rede neste mesmo roteador.

6 CONCLUSÕES E TRABALHO FUTURO

Este documento apresentou a especificação e implementação do modelo de referência da NoC **Mercury**, bem como um conjunto inicial de ferramentas para sua geração e validação. O modelo de referência foi implementado em SystemC e validado através de simulação com a ferramenta Modelsim.

O trabalho aqui descrito revelou a viabilidade de implementar redes intra-chip eficientes com topologia toro. O roteador possui uma estrutura razoavelmente simples, sendo composto por três filas circulares e um conjunto de máquinas de estado. Isto indica que implementações em hardware devem gerar módulos eficientes, de tamanho aceitável e com alto desempenho. A capacidade de parametrizar as dimensões das filas, do *flit* e do *phit* de forma independente provê uma maneira eficaz de avaliar compromissos de área e desempenho.

No momento da escrita deste documento, a implementação no nível de transferência entre registradores (em inglês *register transfer level*, ou RTL) em linguagem VHDL está completa e encontra-se em fase de validação funcional.

Um trabalho futuro importante é a validação ampla da versão RTL, através do uso da ferramenta de geração/avaliação de tráfego descrita em [31], e técnicas de verificação funcional e co-validação descritas em [30].

Outro trabalho previsto é a implementação de uma outra versão de rede com topologia toro, desta vez o mais similar possível em escolhas à rede Hermes, implicando a adoção do modo de chaveamento *wormhole*, e filas de entrada. Para tanto, apenas pequenas adapatações deverão ser realizada no roteador Hermes original, para suportar um algoritmo de roteamento simples, provavelmente do tipo *dimension ordered* proposto em [6] e [8]. Este trabalho já está em andamento.

Um trabalho futuro posterior a todos os anteriores e fundamental para a pesquisa ora iniciada é a comparação das 3 redes geradas: Hermes malha 2D, toro 2D com roteamento Cypher-Gravano e toro 2D com roteamento *dimension ordered*. A comparação visa estabelecer a aplicabilidade e limitações de cada uma destas arquiteturas de comunicação. Comparações de desempenho em termos de latência e vazão de redes similares do ponto de vista de consumo de recursos (tipicamente área e/ou potência) serão fundamentais para determinar dar relevância a estes resultados.

7 REFERÊNCIAS

[1] Bastos, É. **Um Estudo de Redes Intra-chip com Topologia Toro**. Trabalho Individual II. PPGCC, Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brasil, 2004. (In Portuguese).

[2] Benini, L.; De Micheli, G. **Networks on chips: a new SoC paradigm**. IEEE Computer, v. 35(**1**), Jan. 2002, pp. 70-78.

[3] Bergamaschi, R.; et al. **Automating the design of SOCs using cores.** IEEE Design & Test of Computers, v. 18(**5**), Sept.-Oct. 2001, pp. 32-45.

[4] Carara, E. **Uma Exploração Arquitetural de Redes Intra-chip com Topologia Malha e Modo de Chaveamento *Wormhole***. Trabalho de Conclusão II. Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brasil, 2004. (In Portuguese).

[5] Cypher, R.; Gravano, L. **Storage-Efficient, Deadlock-Free Packet Routing Algorithms for Torus Networks**. IEEE Transactions on Computers, 43(12):1376-1385, 1994.

[6] Dally, W.; Seitz, C. **The Torus Routing Chip**. Journal of Distributed Systems, 1(3):187–196, 1986.

[7] Dally, W.; Towles, B. **Route packets, not wires: on-chip interconnection networks**. In: 38th Design Automation Conference (DAC'01), Jun. 2001, pp. 684-689.

[8] Dally, W.; Towles, B. **Principles and Practices of Interconnection Networks**. Elsevier Inc, Amsterdam, 2004.

[9] Duato, J.; Yalamanchili, S.; Ni, L. **Interconnection Networks**. Morgan Kaufmann, Revised Edition, 2002, 624 p.

[10] Grötker, T.; Liao, S.; Martin, G.; Swan, S. **System Design with SystemC**. Springer, Berlin, 2002.

[11] Guerrier. P.; Greiner. A. **A generic architecture for on-chip packet-switched interconections**. In: Design Automation and Test in Europe (DATE'00), Mar. 2000, pp. 250-256.

[12] Gupta, R.; Zorian, Y. **Introducing Core-Based System Design**. IEEE Design & Test of Computers, v. 14(**4**), Oct.-Dez. 1997, pp. 15-25.

[13] International Sematech. **International Technology Roadmap for Semiconductors - 2002 Update**, 2002. Available at http://public.itrs.net.

[14] Kumar, S.; Jantsch, A.; Soininen, J.-P.; Forsell, M.; Millberg, M.; Öberg, J.; Tiensyrjä, K.; Hemani, A. **A Network on Chip Architecture and Design Methodology**. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), Apr. 2002, pp. 105-112.

[15] Marcon, C.; Calazans, N.; Moraes, F.; Susin, A.; Reis, I.; Hessel, F. **Exploring NoC Mapping Strategies: An Energy and Timing Aware Technique**. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'05), Mar 2005. pp. 502-507.

[16] Marcon, C.; Palma, J.; Calazans, N.; Susin, A.; Reis, R.; Moraes, F. **Modeling the Traffic Effect for the Application Cores Mapping Problem onto NoCs**. In: IFIP International Conference on Very Large Scale Integration (VLSI-SOC'05), Oct 2005. Approved for publication.

[17] Martin, G.; Chang, H. **System on Chip Design**. In: 9th International Symposium on Integrated Circuits, Devices & Systems (ISIC'01), Tutorial 2, 2001.

[18] Mello, A.; Möller, L. **SoC multiprocessing architectures: a study of different interconnection topologies**. End of Term Work, FACIN-PUCRS, Jul. 2003, 120p (in Portuguese), available at http://www.inf.pucrs.br/~moraes/papers/tc_multiproc.pdf.

[19] Mello, A.; Tedesco, L.; Calazans, N.; Moraes, F. **Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC**. In: 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), Sep 2005. Approved for publication.

[20] Moraes, F.; Mello, A.; Möller, L.; Ost, L.; Calazans, N. **A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping**. In: IFIP International Conference on Very Large Scale Integration (VLSI-SoC'03), Dec 2003, pp. 318-323.

[21] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. **Hermes: an Infrastructure For Low Area Overhead Packet-Switching Networks on Chip**.

Integration The VLSI Journal, Amsterdam, 38(-): 69-93, October, 2004.

[22] Ni, M. and McKinley, P. **A Survey of Wormhole Routing Techniques in Direct Networks**. IEEE Computer Magazine, 26:62–76, 1993.

[23] Marescaux, T.; Bartic, A.; Verkest, D.; Vernalde, S.; Lauwereins, R. **Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs.** In: Field-Programmable Logic and Applications (FPL'02), Sep 2002, pp. 795-805.

[24] Marescaux, T.; Mignolet, J.-Y.; Bartic, A.; Moffat, W.; Verkest, D.; Vernalde, S.; Lauwereins, R. **Networks on chip as hardware components of an OS for reconfigurable systems.** In: Field-Programmable Logic and Applications (FPL'03), Sep 2003.

[25] Ost, L.; Mello, A.; Palma, J.; Moraes, F.; Calazans, N. **MAIA - A Framework for Networks on Chip Generation and Verification**. In: Asia South Pacific Design Automation Conference (ASP-DAC'05), Jan 2005, pp. 49-52.

[26] Palma, J.; Marcon, C.; Moraes, F.; Calazans, N.; Reis, R.; Susin, A. **Mapping Embedded Systems onto NoCs - The Traffic Effect on Dynamic Energy Estimation**. In: 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), Sep 2005. Approved for publication.

[27] Saastamoinen, I.; Alho, M.; Pirttimäki, J.; Nurmi, J. **Proteo Interconnect IPs for Networks-on-Chip.** In: IP Based SoC Design, Oct 2002.

[28] Saastamoinen, I; Alho, M.; Nurmi, J. **Buffer Implementation for Proteo Networks-on-Chip.** In: International Symposium on Circuits and Systems (ISCAS'03), May 2003, pp II-113 - II-116.

[29] Sigüenza-Tortosa, D.; Nurmi, J. **Proteo: A New Approach to Network-on-Chip.** In: IASTED International Conference on Communication Systems and Networks (CSN'02), Sep 2002.

[30] Silva, K.; Melcher, E.; Araujo, G. **An automatic Testbench Generation Tool for a SystemC Functional Verification Methodology**. In: 17th Symposium on Integrated Circuits and Systems Design (SBCCI'04), Sep 2004, pp. 66-70.

[31] Tedesco, L.; Mello, A.; Calazans, N.; Moraes, F. **Traffic Generation and Performance Evaluation for Mesh-based NoCs**. In: 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), Sep 2005. Approved for publication.

[32] Zeferino, C. **Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho**. PhD thesis, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil, 2003. (In Portuguese).