Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Mercury:
Uma Rede Intra-chip com Topologia Toro 2D e Roteamento Adaptativo

Érico Nunes Ferreira Bastos

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Ney Laert Vilar Calazans

Porto Alegre
2006

Dados Internacionais de Catalogação na Publicação (CIP)

B327m Bastos, Érico Nunes Ferreira
Mercury : uma rede intra-chip com topologia toro 2D e roteamento adaptativo / Érico Nunes Ferreira Bastos. – Porto Alegre, 2006.
160 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS
Orientador: Prof. Dr. Ney Laert Vilar Calazans

1. Rede Intra-Chip. 2. Topologia Toro 2D. 3. Algoritmo de Roteamento Adaptativo Mínimo. 4. Redes de Computadores. 5. Informática. I. Título.

CDD 004.6

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

Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Substitua esta folha pela ficha de homologacao

Agradecimentos

Agradeço aos meus pais, Leonor e Antônio! Meus grandes incentivadores durante toda minha vida acadêmica. Muitas palavras de apoio e compreensão ditas na hora certa da melhor forma possível, fizeram com que eu chegasse até aqui. Vocês são meus maiores exemplos de vida. Obrigado por tudo, do fundo do coração. Eu AMO vocês.

Agradeço a CAPES por ter viabilizado este trabalho através de um auxílio bolsa. A Sandra Rosa, José Carlos e Thiago Lingener, funcionários do PPGCC pela ajuda disponibilizada em toda papelada necessária durante o mestrado, valeu pessoal!!

Agradeço ao prof. Ney pela orientação de mestrado. Nunca vou esquecer quando tivemos nossa primeira reunião, onde eu sem orientador e oriundo da área de PPD, tentava entender conceitos de *hardware* demonstrados em um FPGA, como o da máquina de refrigerantes. Grande parte do meu conhecimento nesta área se deve ao teu esforço, nas nossas intermináveis reuniões de quinta-feira, nos mínimos detalhes das correções dos volumes entregues e claro no bate papo do dia-a-dia. Sem dúvida, és um profissional para se ter como exemplo a seguir. Agradeço ao prof. Moraes pelos conselhos dados ao meu trabalho e pela disponibilidade em ajudar sempre que necessário. Ao corpo docente do PPGCC pelo ótimo convívio durante estes dois anos, em especial aos da área de SDAC onde um maior contato foi realizado.

Agradeço aos meus amigos, que conviveram comigo nestes dois anos. Ao Rafael Krolow pela amizade e ajuda, como não lembrar das entregas de TIs faltando 30minutos para o prazo final, em plena sexta-feira, e eu com problemas para compilar meu latex!!! Ao Gustavo Bertoldi, pelos momentos de descontração, idas na Cantina, Grelattus, Shopping e claro as compras da quinzena no Nacional. Gilberto, o famoso Giba, como esquecer das idas e vindas nos finais de semana para Pelotas de carona contigo, *golfzeira* guerreira lotada de roupas sujas, *notebook*, comidas e muita vontade de rever o pessoal que por lá ficou. Sem esquecer é claro, Giba mestre cuca com seus carreteiros de segunda-feira, e das tentativas de correr para entrar em forma!!

Agradeço aos meus colegas e amigos do GAPH, pois sem eles este trabalho não chegaria ao fim desta forma. Inicialmente quero agradecer aos colegas de sala de aula e do grupo GAPH, Rafael, Melissa, Leonel, Moller pela constante ajuda e coleguismo. A união deste grupo fez com que não nos sentíssemos sozinhos durante este tempo da dissertação. Agradeço, ao Celso Soccol, meu amigo de todas as horas. Muitas implementações da rede Mercury fizemos juntos no decorrer deste trabalho, aprendi muito trabalhando contigo. Agradeço-te pela constante

ajuda, és um exemplo de determinação. Valeu!! Ao Sérgio, cara o que seria do segundo ano de mestrado sem tua presença no GAPH? Tu és uma figura, muito gente fina mesmo. Obrigado pelo *help* prestado na prototipação da NoC!! Aline, Luigi e Leonel obrigado pela ajuda prestada na finalização deste trabalho.

Agradeço a minha namorada, Paula Arruda, pelas constantes palavras de carinho nestes dois anos. Apesar da distância estivesse sempre presente na minha vida, me apoiando e fazendo com que eu caminhasse pra frente. Obrigado Lovi! :** Agradeço ao meu irmão Eduardo, pelo carinho e apoio prestado sempre que precisei.

"A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original".
Albert Einstein

Resumo

A tecnologia de fabricação de circuitos integrados (CIs) evoluiu ao ponto de inviabilizar alguns dos principais paradigmas subjacentes ao projeto de sistemas digitais complexos. Dentre estes, dois dos mais relevantes são o uso de barramentos como meios de interconexão intra-chip e o projeto totalmente síncrono. Redes intra-chip (em inglês, *Networks On-Chip* ou NoCs) vêm se destacando como uma possível alternativa para substituir barramentos, talvez até suprindo meios de superar as dificuldades de abandonar o projeto totalmente síncrono. Trata-se de arquiteturas de comunicação que adaptam conceitos oriundos de redes de computadores e de sistemas paralelos e distribuídos para o ambiente intra-chip. Elas são constituídas por fiação de alcance local no interior de um CI e elementos chaveadores ou roteadores, cuja interconexão define uma topologia de rede. Nos extremos da rede conectam-se núcleos processadores de um SoC. Uma das questões relevantes colocada pela comunidade de pesquisa em NoCs é qual a melhor topologia a adotar para que um SoC alcance máximo desempenho a um mínimo custo.

O presente trabalho contribui com uma proposta de arquitetura para um roteador toro, partindo de um algoritmo de roteamento mínimo totalmente adaptativo, sugerido por Cypher e Gravano. Com base neste roteador, desenvolve-se aqui o projeto de uma rede com topologia toro denominada Mercury.

Apresentam-se também as modelagens abstrata e concreta da rede Mercury. A rede foi capturada como uma descrição de hardware em VHDL, validada por simulação e prototipada com sucesso em FPGAs. Uma ferramenta parametrizável que dá suporte para a geração automática dos modelos abstrato (em SystemC TL) e concreto (em VHDL RTL) da rede Mercury é uma contribuição adicional do trabalho.

Finalmente, provê-se resultados preliminares de avaliação do projeto, tanto do ponto de vista de ocupação de área em FPGAs como em relação ao desempenho da rede, medindo a vazão e a latência média desta sob diferentes condições de tráfego. Valores obtidos para a rede Mercury são também comparados com a NoC Hermes, uma rede no estado da arte de desenvolvimento.

Palavras-chave: Rede Intra-chip. Topologia Toro 2D. Algoritmo de Roteamento Adaptativo Mínimo.

Abstract

The integrated circuit (IC) fabrication process technology has evolved to a point where some of the main paradigms underlying the design of complex digital systems are no longer usable. Among these, two of the more relevant ones are the use of busses as intra-chip interconnect media and purely synchronous design. Networks on chip (NoCs) have been put forward as a possible alternative to substitute busses, and are also considered as helpful while abandoning purely synchronous design techniques. NoCs are intra-chip communication architectures that adapt concepts from computer networks and distributed and parallel systems to the intra-chip environment. NoCs are composed by intra-chip local wires and switching elements, called routers. The interconnection of routers defines a topology where processing cores are connected to form the SoC.

One of the prominent questions asked by the reserch community that needs to be answered is which topology is best to be adopted, in order to enable the SoC to reach its maximum performance at minimum cost.

The present work contributes to this issue with the proposal of a router architecture adapted to implement torus topology NoCs. The starting point for the architecture is a fully adaptive minimum routing algorithm suggested by Cypher and Gravano. From this router evolves the design of a torus topology NoC called Mercury.

This volume presents the abstract and concrete modeling of the Mercury NoC. The network description has been captured in both SystemC and VHDL. Simulation helped validating the descriptions and the concrete model has been successfully synthesized and prototyped in FPGAs. A parameterizable tool that supports the automatic generation of abstract (in TL SystemC) and concrete models (in RTL VHDL) of the Mercury NoC constitutes an additional contribution of this work.

Finally, a set of preliminary design evaluation results are discussed. Evaluation comprises measuring occupied FPGA area and network performance, considering throughput and average latency figures under different traffic conditions. Mercury NoC measures are compared to what is obtained from the Hermes NoC for similar design parameters. Hermes is a state of the art 2D mesh NoC.

Keywords: Network On-chip. Torus Topology. Fully Adaptive Minimum Routing Algorithm.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

CBDA	Centrally Buffered, Dynamically Allocated	46
CI	Circuito Integrado	27
DAMQ	Dynamically-Allocated, Multi-Queue	44
DVD	Digital Video Disk	27
FIFO	First In First Out	44
Flits	Flow Control Digit	42
GAPH	Grupo de Apoio ao Projeto de *Hardware*	30
HOL	Head Of Line	44
IP	Propriedade Intelectual	27
ITRS	International Technology Roadmap for Semiconductors	29
MP3	MPEG Audio Layer-3	28
MPEG	Motion Picture Experts Group	28
NoC	Networks on-Chip	29
PUCRS	Pontifícia Universidade Católica do Rio Grande do Sul	30
RTL	*Register Transfer Level*	31
SAFC	Statically Allocated, Fully Connected	44
SAMQ	Statically Allocated Multi-Queue	44
SoC	*System on Chip*	27
TL	*Transaction Level*	31
USB	Universal Serial Bus	28

Sumário

1 Introdução

Sistemas digitais estão cada vez mais presentes na sociedade moderna, uma vez que grande parte dos produtos eletrônicos como computadores, telefones celulares, DVDs , aplicações automobilísticas, entre outros, são controlados e operados de forma digital. Sistemas digitais são compostos de três tipos de blocos básicos: lógica, memória e comunicação. A lógica transforma e combina dados, através de operações aritméticas e lógicas; as memórias armazenam dados para uso num tempo futuro; a comunicação transfere os dados de um local para outro [14, 19, 42, 48, 52, 69].

O desempenho da maioria dos sistemas digitais está diretamente ligado à comunicação, já que, com o avanço da tecnologia, memórias e processadores têm diminuído o seu tamanho, se tornado mais rápidos e ao mesmo tempo se tornado mais acessíveis. O grande problema é que o mesmo não acontece com os componentes de comunicação, fazendo com que o principal limite dos sistemas digitais nos dias de hoje, esteja diretamente relacionado com a interconexão entre módulos lógicos e memórias, que realizam a comunicação [28].

A crescente densidade (número de transistores por unidade de área de silício), as freqüências de operação elevadas, o tempo de projeto e o ciclo de vida reduzidos caracterizam o cenário da indústria de semicondutores na atualidade [59]. Em 1980, a maioria dos circuitos integrados (CIs) complexos, eram compostos por dezenas de milhares de transistores. Atualmente, já é possível encontrar CIs com dezenas de milhões de transistores [57]. Esse avanço na tecnologia tem permitido a integração de múltiplos componentes, como processadores, controladores de acesso a periféricos e memória, em um único *chip*, resultando na integração de um sistema completo em uma mesma pastilha [10]. Esses sistemas são conhecidos como SoCs (do inglês, *Systems on Chip*), os quais possuem vários núcleos de propriedade intelectual (do inglês, *IP cores*) que são módulos pré-projetados, pré-verificados e prototipados, utilizados para construção de uma aplicação complexa [42]. A Figura 1 ilustra a arquitetura genérica de um SoC.

Tradicionalmente, arquiteturas de interconexão intra-chip têm sido implementadas seja através do uso de conjuntos de conexões dedicadas 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, caracterís-

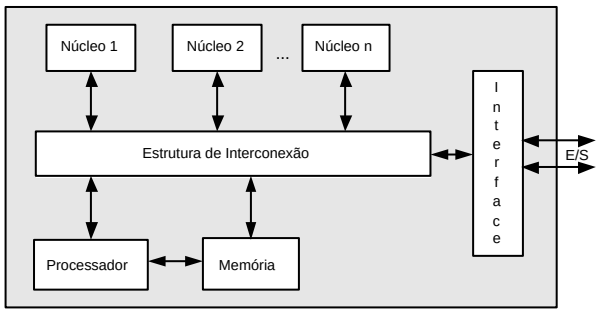


Figura 1 – Arquitetura genérica de um SoC. Pode-se notar que a arquitetura possui núcleos que podem ser módulos específicos, tais como decodificadores MPEG2 ou MP3, interfaces de comunicação Ethernet ou USB , dentre outros. A estrutura de interconexão tem como finalidade realizar a comunicação entre os núcleos e o resto do sistema, podendo ser um barramento ou uma rede intra-chip. Nota-se a presença de interface com o mundo externo (E/S), que é utilizada para interconectar os periféricos, como porta serial, porta USB entre outros.

ticas 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, permitindo apenas uma comunicação por vez, com todos os núcleos compartilhando a largura de banda de comunicação provida pelo barramento. 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.

Atualmente as interconexões entre os núcleos em um SoC são realizadas através de canais ponto-a-ponto ou de canais multi-ponto [69]. Nos canais ponto-a-ponto os núcleos são interligados por canais dedicados (Figura 2 (a)), sendo que cada canal é constituído por um conjunto de fios ligando dois núcleos entre si. Os canais multi-ponto possuem seguidas vezes uma estrutura de interconexão com a forma de um barramento compartilhado (Figura 2 (b)), multiplexado no tempo, no qual os núcleos do sistema são conectados.

Por serem considerados simples e reutilizáveis, canais multi-ponto como os barramentos têm sido preferidos para a construção de SoCs. Apesar de serem os mais utilizados, várias desvantagens são encontradas quando comparados com canais ponto-a-ponto. Algumas delas são:

1. Em estruturas multi-ponto, somente uma comunicação pode ocorrer a cada momento. Em estruturas ponto-a-ponto várias comunicações podem ser realizadas em paralelo.

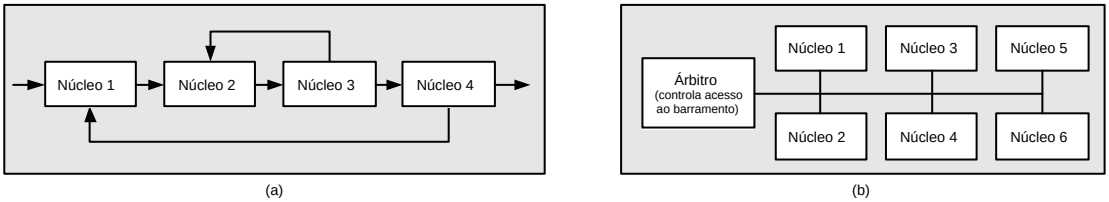


Figura 2 – Estruturas de interconexão: (a) ponto-a-ponto; (b) multi-ponto.

2. Em estruturas multi-ponto, quando um núcleo é adicionado ao sistema ele é conectado a canais já compartilhados por todos os outros núcleos. Já em estruturas ponto-a-ponto, cada novo núcleo exige que novos canais sejam adicionados ao sistema para a comunicação, aumentando a largura de banda total disponível, ou seja, a interconexão cresce com o tamanho do sistema.

3. Em estruturas multi-ponto, os canais de comunicação do barramento devem alcançar todos os núcleos do sistema. Já em estruturas ponto-a-ponto, os núcleos que se comunicam são geralmente posicionados o mais próximo possível uns dos outros, reduzindo assim o comprimento dos canais de comunicação dedicados, conseqüentemente aumentando o desempenho.

De acordo com vários autores [9,18,26,37], 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 [59]. 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 são as denominadas redes intra-chip (em inglês, *networks on chip* ou NoCs) [9]. NoCs são arquiteturas especiais, empregando conceitos oriundos de redes de computadores e de sistemas distribuídos, 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 conectam-se núcleos de um SoC. Os núcleos do sistema são interligados por meio de uma rede de roteadores utilizando canais ponto-a-ponto e canais para transferência de dados entre a origem e o destino.

NoCs podem apresentar uma topologia de interconexão regular ou não, podem apresentar diferentes relações entre as cardinalidades de elementos chaveadores e núcleos do SoC, etc. O presente documento descreve a especificação, 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. Também se descreve a implementação concreta, a validação e a prototipação desta NoC. Esta arquitetura é denominada de NoC **Mercury**.

1.1 Motivação

Parte da motivação deste trabalho já foi apresentada anteriormente, 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 GAPH (Grupo de Apoio ao Projeto de *Hardware*) da PUCRS do qual o autor faz parte, 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 do grupo, de ênfase eminentemente exploratória e de implementação prática [20, 47], atingiu-se a proposta da infra-estrutura Hermes de suporte à implementação de NoCs com baixo consumo de área, inicialmente descrita em [46]. Esta infra-estrutura tem evoluído ao longo dos últimos anos, agregando crescente número de funcionalidades [15,44], métodos de modelagem, projeto, validação e avaliação de NoCs [15,39,40,54] e ferramental de apoio associado [53,63].

Apesar do extenso trabalho já realizado, percebe-se um conjunto de limitações na abordagem adotada. Primeiro, todos os trabalhos citados restringem a estrutura de NoCs investigada, visando tornar a complexidade da pesquisa gerenciável. As redes investigadas em todos os trabalhos citados assumem chaveamento de pacotes, empregam apenas a topologia malha bidimensional regular (malha 2D, Figura 3 (a)), usam apenas o modo de chaveamento *wormhole* [19,22] onde o tamanho do *flit* e do *phit* [19, 22] são idênticos e assumem armazenamento temporário baseado apenas em filas de entrada. Existem ferramentas de apoio também desenvolvidas pelo grupo, que na sua versão presente somente trabalham com topologia malha. Pode-se citar como exemplo as ferramentas Maia [53] e NocGen [49] para geração de rede, TrafficGeneration [63] para geração de tráfego e NocLogView [49] para visualização das estatística de tráfego.

Não obstante, vários graus de liberdade são adotados nos trabalhos mencionados, tais como não restringir as dimensões da malha 2D, permitir dimensionar livremente as filas e o tamanho do *flit/phit* dos pacotes. Esta opção é razoável, sendo confirmada pela maior parte da investigação atual em NoCs no mundo, que assume pelo menos parte das limitações aqui assumidas de forma coerente. Isto decorre em maior escala de um raciocínio natural de mapear características de redes em sistemas distribuídos para o ambiente intra-chip, buscando atingir o melhor custo benefício na implementação de NoCs. Em menor escala, decorre também dos resultados

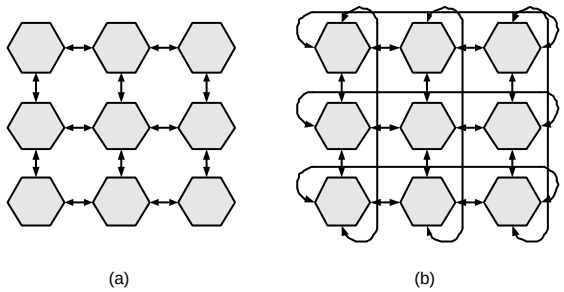


Figura 3 – A Figura ilustra em (a) a topologia malha e em (b) a topologia toro.

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 arquitetura de uma rede toro 2D (Figura 3 (b)) quando comparada a uma rede malha 2D é basicamente a mesma, exceto pelos canais *wraparound*, que conectam os roteadores das extremidades da rede no sentido horizontal e vertical. Desta forma, diferente do que ocorre na rede malha 2D, todos os roteadores possuem 5 portas no toro 2D (Norte, Sul, Leste, Oeste e Local). A topologia toro possui algumas vantagens em relação à malha, como por exemplo, o seu melhor desempenho em redes com alto tráfego e uma melhor diversidade de caminhos [19]. Como ainda pouco se conhece dessa topologia no ambiente intra-chip, informações de como ela se comporta quando utilizada em NoCs não estão totalmente esclarecidas.

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, neste caso usando a topologia toro 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.

1.2 Objetivos

O objetivo principal deste trabalho é obter uma arquitetura de comunicação de rede com topologia toro, denominada **Mercury**, através da implementação da mesma em níveis de abstração TL(do inglês, transaction level) e RTL(do inglês, register transfer level). A maior contribuição do presente trabalho, resultado deste objetivo, é a quantificação da comparação entre a **Mercury** e a já existente arquitetura de comunicação com topologia malha, denominada Her-

mes, comparando as mesmas de forma a avaliar em que aspectos uma é superior a outra em diversos quesitos.

Os objetivos específicos deste trabalho são:

- Estudar propostas de NoCs com topologia toro presentes na bibliografia;
- Modelar de forma abstrata no nível TL e concreta no nível RTL [12, 13, 32, 33, 49, 62] a arquitetura **Mercury**;
- Avaliar, de forma inicial, a implementação da rede **Mercury** em comparação com rede Hermes implementada com topologia malha;
- Prototipar a modelagem concreta da arquitetura de comunicação **Mercury** em *hardware*.

1.3 Organização do Restante do Documento

O restante deste trabalho encontra-se organizado da seguinte forma. No Capítulo 2 são apresentadas definições básicas de redes de comunicação, como topologias, chaveamento, armazenamento temporário, controle de fluxo etc., além de outros conceitos, definindo uma terminologia coerente para o restante do volume. É também apresentada uma introdução a redes com topologia toro e redes intra-chip.

O Capítulo 3 apresenta uma revisão do estado da arte de redes intra-chip com topologia toro. O Capítulo 4 apresenta o algoritmo de roteamento utilizado na arquitetura de comunicação descrita neste trabalho, justificando sua escolha. Um exemplo detalhado de operação do algoritmo é incluído.

O Capítulo 5 apresenta uma proposta de arquitetura para o roteador toro que suporte o algoritmo de roteamento escolhido. São descritos desde a arquitetura geral do roteador até como implementar as interfaces de comunicação, árbitros, filas etc.

O Capítulo 6 apresenta as modelagens da NoC Mercury nos níveis de abstração TL e RTL. Este Capítulo tem como objetivo demonstrar de que forma foi implementado cada módulo da NoC, a estrutura do roteador e da rede, além de demonstrar o seu funcionamento através de validações por simulação.

O Capítulo 7 apresenta as ferramentas desenvolvidas durante a elaboração da dissertação para dar apoio à criação e validação da rede Mercury.

O Capítulo 8 apresenta as estruturas usadas para prototipar a rede Mercury em *hardware*. Neste Capítulo apresentam-se as diferentes etapas de prototipação, passando pela prototipação

inicial do roteador até a de instâncias completas da rede.

O Capítulo 9 apresenta os resultados obtidos ao comparar as arquiteturas de rede Mercury e Hermes do ponto de vista de quesitos tais como área e desempenho. Para finalizar o Capítulo 10, apresenta as considerações finais sobre o trabalho e direções para trabalhos futuros.

2 Definições Básicas

2.1 Introdução

Neste Capítulo são apresentados alguns dos principais conceitos relacionados a redes de comunicação, com o intuito de servir de referência ao leitor para a nomenclatura no restante do texto. Aqui se discute as características e os mecanismos utilizados para o envio e recebimento de dados através de uma rede. Inicialmente o Capítulo apresenta algumas definições sobre redes de comunicação. Após, introduz definições de topologia de rede, roteamento, chaveamento, controle de fluxo, armazenamento temporário e arbitragem. Em seguida, discute-se características de uma rede com topologia toro. Por fim, apresenta-se alguns conceitos sobre redes intra-chip.

2.1.1 Redes de Comunicação

Uma rede de comunicação é formada por um conjunto de nodos capazes de trocar informações e compartilhar recursos, interligados por um sistema de comunicação [19,61]. Nodo é uma entidade que gera informação a ser comunicada a outras entidades e/ou consome informação gerada por outras entidades similares. Por exemplo, imagine quatro nodos conectados a uma rede de comunicação, onde essa rede possui a funcionalidade de receber e transmitir dados de uma origem a um destino qualquer conectado a ela. Quando um nodo deseja comunicar-se com outro, basta ele enviar o dado para a rede que a mesma se encarrega de transmitir o dado.

Como a rede de comunicação é um sistema programável, ela pode refazer suas conexões entre qualquer par de nodos da forma que desejar, fazendo assim com que qualquer nodo se comunique com outro através da sua estrutura. A definição de redes de comunicação é utilizada em vários contextos distintos. Existem redes locais e não locais, redes que conectam outras redes ou que conectam computadores a internet, e até redes dentro de chips que interligam memórias, registradores e unidades aritméticas com o processador. Segundo Zeferino [69], uma rede de comunicação é tipicamente caracterizada pela sua topologia e por um conjunto de

mecanismos que definem a forma como ocorre a transferência de mensagens através da rede.

2.1.2 Topologias de Rede

Uma rede é composta por um conjunto de roteadores e canais, e a topologia de rede é definida pela estrutura de interligação destes roteadores através dos canais. Dally [19] define topologia de rede como um conjunto de nodos N^* conectados a um conjunto de canais C, onde as mensagens enviadas e recebidas pertencem a um conjunto de nodos terminais N onde $N \subseteq N^*$. Se a rede possuir todos os nodos como terminais, pode-se simplificar afirmando que o conjunto de nodos é N. Um canal pode ser definido como $c = (x, y) \in C$, onde o nodo origem, representado por x, e o nodo destino representado por y, pertencem a N^*, sendo $x, y \in N^*$.

Quando se aborda topologias de rede é necessário definir alguns conceitos importantes. São eles:

- Largura de banda: quantidade de informações máxima que pode trafegar de uma origem para um destino em um determinado período de tempo [19, 38].
- Latência: tempo gasto para entregar uma única mensagem no seu destino [38]. Esse tempo é totalmente dependente da rede sob análise.
- *Hop Count*: número de canais que a mensagem deve cruzar para ir do nodo fonte ao nodo destino seguindo algum caminho disponível na rede [19].
- Largura de Bissecção: número mínimo de canais de comunicação da rede que precisam ser cortados para que a rede seja dividida em dois conjuntos de nodos com cardinalidade o mais próxima possíveis [22].

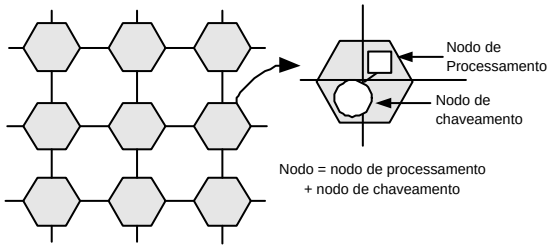


Figura 4 – Nodos de redes diretas [20].

As topologias de rede de comunicação podem ser agrupadas em duas classes principais, as redes diretas e as redes indiretas. Nas redes diretas, cada nodo de chaveamento (responsável

pela transferência de mensagens entre os nodos de processamento) possui um nodo de processamento associado (responsável pelo processamento da mensagem), como ilustra a Figura 4. As redes diretas são também conhecidas como redes estáticas, pois as ligações entre os nodos de chaveamento e processamento não mudam ao passar do tempo.

Entre as redes diretas ortogonais mais utilizadas pode-se citar malha Figura 5 (a), toro Figura 5 (b) e o hipercubo Figura 5 (c).

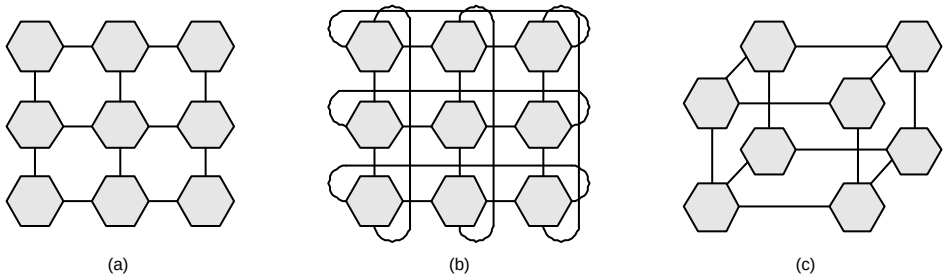


Figura 5 – Exemplos de topologias de rede diretas: (a) malha 2D 3x3; (b) toro 2D 3x3; (c) hipercubo 3D 2x2x2.

Nas redes indiretas os nodos de processamento possuem uma interface para uma rede de nodos de chaveamento. Cada nodo de chaveamento possui um conjunto de portas bidirecionais para ligações com outros nodos de chaveamento e/ou com os nodos de processamento. Somente alguns nodos de chaveamento possuem conexões para nodos de processamento e apenas esses podem servir de fonte ou destino de uma mensagem. Essa característica é o que diferencia as redes diretas das indiretas. Como exemplo de topologia homogênea de redes indiretas pode-se citar o crossbar, ilustrado na Figura 6.

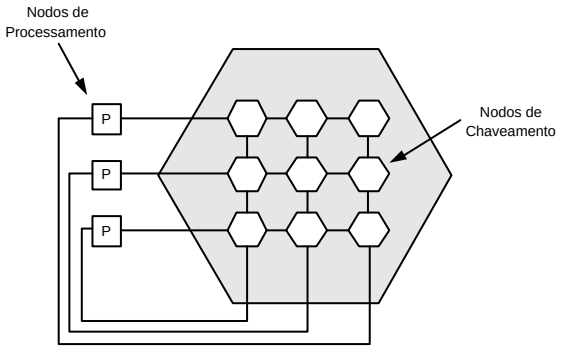


Figura 6 – Exemplo de topologia de rede indireta - Crossbar 3x3 [20].

Dally [19] afirma que não existe uma topologia que seja a mais adequada para qualquer aplicação, pois cada uma é apropriada para uma situação diferente de requisitos e restrições de comunicação.

2.1.3 Roteamento

Roteamento é o método utilizado pelo roteador para decidir, dentre as possíveis portas de comunicação com nodos vizinhos, aquela por onde o dado irá seguir. Geralmente, quando se envia um dado pela rede, existem vários caminhos possíveis para que ele chegue ao seu destino. Um bom roteamento é aquele que consegue escolher um caminho com poucos roteadores entre a origem e o destino, ser tolerante a falhas e adaptativo de forma a conseguir diminuir o *hop count*. Também deve-se considerar o balanceamento de carga através dos diversos caminhos existentes, para que a rede não fique desbalanceada (alguns dos caminhos fiquem sobre-utilizados em relação a outros).

Algoritmo de Roteamento

O roteamento da rede é implementado por um algoritmo que define como é realizada a transferência de dados entre o roteador origem e o destino. O algoritmo é que determina quais dos possíveis caminhos entre os roteadores será utilizado. Além disto, ele deve garantir o correto funcionamento da rede na entrega de pacotes evitando problemas como:

- *Deadlock*: dependência cíclica entre roteadores que solicitam acesso de uma determinada porta de saída, de forma que nenhum consiga obter progresso algum, independente da seqüência de eventos que ocorra. A Figura 7 ilustrada um exemplo de *deadlock*.

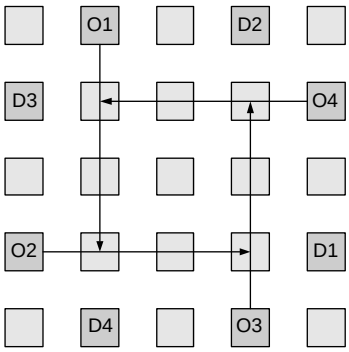


Figura 7 – Ilustrando a situação de *deadlock* (dependência cíclica) em uma rede malha, com um canal físico em cada direção entre cada par de roteadores adjacentes de forma ortogonal, onde quatro pacotes são transmitidos dos roteadores denominados O1, O2, O3, O4 para os roteadores destino D1, D2, D3, D4 respectivamente. Nota-se que todos os pacotes estão trancados esperando por um canal que nunca estará disponível, resultando em uma situação de *deadlock*.

- *Livelock*: ocorre quando pacotes ficam circulando na rede sem conseguir fazer progresso algum na direção do seu roteador destino. Normalmente este problema ocorre em algorit-

mos de roteamento adaptativos não mínimos, podendo ser evitado com estratégias como restrição dos desvios que um pacote pode realizar.

- *Starvation*: situação onde um pacote requisita uma porta de saída, mas nunca é atendido porque ela é sempre alocada para outros pacotes, podendo ocorrer postergação indefinida de entrega do pacote.

Existem vários algoritmos de roteamento propostos na literatura, cada qual visando atender requisitos distintos. É possível classificar algoritmos de roteamento de acordo com alguns critérios, tais como [69]:

- Quanto à adaptatividade: (*i*) Roteamento Determinístico: utiliza sempre o mesmo caminho entre um dado par de roteadores fonte-destino; (*ii*) Roteamento Adaptativo: escolhe o caminho entre um par de roteadores fonte-destino em função de características dinâmicas da rede, tais como congestionamento de canais, tráfego médio pelos canais ou falhas que venham a ocorrer na rede. Os algoritmos adaptativos ainda podem ser classificados quanto:
  - à minimalidade: (*i*) Mínimo: utiliza sempre canais que levam o pacote ao seu destino pelo menor caminho, em número de *hops*. (*ii*) Não Mínimo: utiliza qualquer caminho existente para atingir o destino. Esse roteamento possui uma maior flexibilidade em termos de caminhos disponíveis, porém pode ocasionar problemas de *livelock* na rede.
  - à progressividade: (*i*) Progressivo: o cabeçalho sempre avança pela rede, reservando um novo canal a cada passo de roteamento. (*ii*) Regressivo: o cabeçalho pode retornar pela rede, liberando canais previamente reservados.
  - ao número de caminhos (mínimos): (*i*) Completo: o algoritmo pode utilizar todos os caminhos (mínimos) disponíveis. (*ii*) Parcial: apenas pode utilizar um subconjunto dos caminhos (mínimos).
- Quanto ao lugar onde as decisões de roteamento são tomadas: (*i*) Roteamento Fonte: o nodo transmissor determina a rota que o pacote deverá passar até chegar ao seu destino antes de injetá-lo na rede. O cabeçalho do pacote deve carregar a informação de roteamento completa, aumentando assim o tamanho do pacote; (*ii*) Roteamento Centralizado: possui um roteador centralizado com a função de criar todas as rotas da rede e distribuí-las aos outros roteadores; (*iii*) Roteamento Distribuído: cada roteador possui a responsabilidade de calcular parte da rota com as informações locais que possui, bem como informações que recebe do pacote.

- Quanto à implementação: (*i*) Baseado em Tabela: roteamento é feito baseado em uma consulta a uma tabela de rotas; (*ii*) Baseado em Máquina de Estados: quando a máquina de estados determina o roteamento ou parte deste.
- Quanto ao momento da realização do roteamento: (*i*) Dinâmico: executa o algoritmo em tempo de execução da aplicação. (*ii*) Estático: executa o algoritmo no tempo de compilação da aplicação.
- Quanto ao número de destinos: (*i*) Unicast: pacotes possuem somente um destino. (*ii*) Multicast: pacotes podem ser roteados para vários destinos.

2.1.4 Chaveamento

Para que seja possível realizar a comunicação entre a origem e o destino de um pacote, roteadores devem repassar o pacote recebido para o próximo roteador ou para o destino através de regras definidas pelo chaveamento. O chaveamento define a forma pela qual os dados são transferidos de um canal de entrada de um roteador para um dos seus canais de saída [69].

Segundo Dally e Towles, Ni e McKinley [19, 51], um roteador pode ser definido como um dispositivo que conecta um número de canais de entrada a um número de canais de saída, ou seja, um roteador tem a funcionalidade de transferir informações de uma de suas portas de entrada para uma de suas portas de saída. Na Figura 8, pode-se visualizar a estrutura genérica de um roteador.

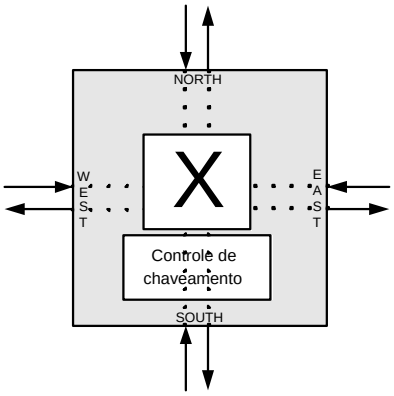


Figura 8 – Estrutura de um roteador, onde visualiza-se um módulo de controle de chaveamento. Neste, pacotes oriundos das portas de entrada são direcionados para as portas de saída, com base na estratégia de chaveamento.

Existem dois métodos básicos utilizados para a transferência dos dados que são: chavea-

mento de circuito e chaveamento de pacotes. No método *chaveamento de circuito* é estabelecido um caminho entre a origem e o destino antes do inicio do envio da mensagem. Quando este caminho é estabelecido a mensagem pode ser enviada, sendo que qualquer outro pedido de comunicação que utilize os canais alocados não será aceito. O caminho somente é desfeito quando toda a mensagem chega no seu destino. Segundo [20] a grande vantagem deste método é que não são necessárias filas nos roteadores intermediários, pois uma vez que a comunicação foi estabelecida a mensagem não é bloqueada. A desvantagem é que esse método causa perda no desempenho da rede como um todo, já que todos os roteadores no caminho reservado para o envio ficam bloqueados durante a transmissão de dados.

No método *chaveamento de pacotes*, a mensagem é dividida em pequenos pacotes. Através do cabeçalho de cada pacote é decidido em cada roteador por qual porta de saída o pacote deve ser roteado, não existindo um caminho pré-definido. Mello e Möller [20] colocam que a grande vantagem do chaveamento de pacotes é que o canal permanece ocupado apenas enquanto o pacote está sendo transferido, enquanto que a desvantagem deste método é que existe a necessidade de filas para o armazenamento temporário de pacotes.

O chaveamento de pacotes pode ser classificado segundo o *modo de chaveamento*, que define a política de repasse de dados entre os roteadores intermediários. Os modos mais usados são: *store-and-forward*, *virtual cut-through* ou *wormhole* [17, 19, 51].

Store-and-Forward

O modo de chaveamento *store-and-forward* garante que todo pacote será recebido e armazenado por inteiro antes de ser enviado ao próximo roteador [58]. A vantagem deste modo é sua simplicidade e o baixo congestionamento dos canais da rede devido ao armazenamento por completo do pacote em cada roteador. Como desvantagem, pode-se citar a maior latência na rede, já que o roteador deve receber todo o pacote para depois iniciar o seu envio ao próximo roteador. Desta forma, o roteador aloca os recursos necessários para que os pacotes possam avançar de nodo em nodo em direção ao destino.

Virtual Cut-Through

O modo de chaveamento *virtual cut-through* é um aperfeiçoamento do modo *store-and-forward* do ponto de vista de eficiência. Sua vantagem é a redução da latência na comunicação, já que somente pressupõe o armazenamento do pacote inteiro caso o canal por ele desejado encontre-se indisponível [69]. Após o roteador receptor garantir que pode receber todo o pacote, o roteador fonte pode iniciar a transferência, mesmo que ainda não tenha recebido o pacote

por inteiro. A desvantagem é que apesar de um pacote somente ser armazenado localmente quando o canal desejado por ele estiver indisponível, o roteador ainda necessita de espaço de armazenamento suficiente para estocar completamente o pacote no caso de bloqueios.

Wormhole

O modo de chaveamento *wormhole* é uma variação do chaveamento *virtual cut-through*, que visa reduzir a quantidade de memória para armazenar pacotes em um roteador. Neste modo, pacotes são divididos em pedaços menores denominados *flits* que trafegam pela rede em modo *pipeline* [21]. A grande vantagem da utilização desse modo de chaveamento é a possibilidade de construção de roteadores pequenos e rápidos [69]. Apenas o *flit* cabeçalho contém informações sobre o roteamento. Com isto, os demais *flits* que compõem o pacote devem seguir o mesmo caminho reservado para o cabeçalho.

Como desvantagem do modo *wormhole*, se o *flit* do cabeçalho encontrar um canal que ele deseja utilizar já em uso, ele é bloqueado até que esse canal seja liberado para que se possa progredir. Quando isso acontece, os *flits* do conteúdo da mensagem ficam armazenados ao longo do trajeto pela rede, bloqueando um número arbitrário de roteadores, dependente do tamanho do pacote. Outra desvantagem é a suscetível ocorrência de *deadlock*, já que mensagens possuem permissão para reter alguns recursos enquanto requisitam outros [23].

2.1.5 Controle de Fluxo

Em uma rede de comunicação podem existir vários pacotes trafegando simultaneamente e concorrendo pelos mesmos recursos com o objetivo de avançar de sua origem até o seu destino. Quando algum recurso está alocado para um pacote e um outro não consegue prosseguir, pode-se dizer que ocorreu um bloqueio de recursos. Quando isso ocorre, é necessário que exista uma política para determinar a melhor maneira de lidar com o pacote que não pode ser atendido. Nesse caso, o pacote pode ser descartado, bloqueado no lugar onde está, recebido e armazenado temporariamente ou então desviado por um caminho alternativo. Quem determina isso é a política do controle de fluxo. Ela é quem aloca os canais e as filas no momento em que o pacote trafega na rede [51]. O controle de fluxo possibilita saber se o receptor está habilitado a receber dados ou não (por exemplo, este pode estar com as filas lotadas, impossibilitando o recebimento). Dentre os controles de fluxo conhecidos, os dois mais utilizados são baseado em créditos e *handshake*.

Baseado em Créditos

Na abordagem do controle de fluxo baseada em créditos não existe descarte de pacotes, já que uma transmissão entre roteadores somente é iniciada quando o receptor garantir que pode armazenar a unidade de informação a ser transmitida do roteador fonte. Essa abordagem leva em consideração que o receptor envia ao transmissor um sinal confirmando a existência de créditos que o último possui para envio dos dados. De posse desta informação o transmissor envia dados apenas se existir crédito para o envio. A cada envio de dados pelo transmissor pode ou não haver mudança no seu estado de crédito junto ao receptor, dependendo do espaço de *buffer* no último.

Handshake

Na abordagem para controle de fluxo denominada *handshake*, o transmissor inicialmente informa ao receptor a intenção de enviar um dado através de um sinal de solicitação. Ao receber este sinal o receptor verifica se existe espaço para receber o dado. Em caso afirmativo, o dado é lido e armazenado, e o receptor envia o sinal de reconhecimento de recepção *ack* ao transmissor. Caso não exista espaço disponível o receptor pode enviar um sinal de não reconhecimento *nack* ou não tomar nenhuma ação. Quando ocorrer o roteador transmissor retransmite o dado até o recebimento de um *ack*.

Algumas desvantagens são encontradas nessa abordagem de controle de fluxo. Segundo Dally [19], o controle de fluxo *handshake* é menos eficiente no uso das filas do que o controle de fluxo baseado em créditos. Esta redução ocorre porque os dados ficam armazenados nas filas por um tempo extra até receberem o sinal de *acknowledge*. Além disto, ele coloca que o *handshake* é ineficiente no uso da largura de banda quando não há espaço em fila, já que se torna necessário reenviar o dado. Carara [15] mostra através de simulação e um caso simples onde o controle de fluxo por créditos alcança ganhos próximos de 100% em velocidade quando comparado com *handshake*. Como principal vantagem para a utilização desta abordagem é a simplicidade para implementação.

2.1.6 Armazenamento Temporário

O roteador de uma rede com chaveamento de pacote deve garantir o armazenamento de pacotes destinados a saídas que já estejam sendo utilizadas por outros pacotes. Além disso, o roteador deve ser capaz de controlar o fluxo proveniente dessa porta a fim de evitar a perda de

dados. Todo roteador é capaz de armazenar um número limitado de pacotes ou uma fração de um pacote em sua memória. Por esse motivo, o bloqueio de pacotes pode ocorrer em maior ou menor escala dependendo da estratégia de armazenamento temporário utilizada. Estas estratégias podem influenciar significativamente o desempenho da rede. Dentre as mais utilizadas pode-se destacar o *armazenamento na entrada*, *na saída* ou *centralizado de forma compartilhada*.

Armazenamento na Entrada

O armazenamento temporário de dados na entrada pressupõem a existência de filas independentes em cada uma das portas de entrada do roteador. As filas podem ser implementadas de várias formas, destacando-se as estratégias *FIFO* (*First In First Out*), *SAFC* (*Statically Allocated, Fully Connected*), *SAMQ* (*Statically Allocated Multi-Queue*) e *DAMQ* (*Dynamically-Allocated, Multi-Queue*).

Filas FIFO

A estratégia de filas *FIFO* é a que possui um funcionamento mais simples e menor custo quando comparada com as demais. Basicamente cada uma das filas possui um espaço da memória fixo onde os dados são escritos e lidos na mesma ordem, conforme a Figura 9 (a). O grande problema desta abordagem é o bloqueio da cabeça da fila (em inglês, *Head Of Line* ou *HOL*), onde um pacote bloqueado por necessitar de uma saída em uso por outro pacote impede que outros pacotes atrás dele avancem para uma porta/saída qualquer que esteja disponível naquele instante.

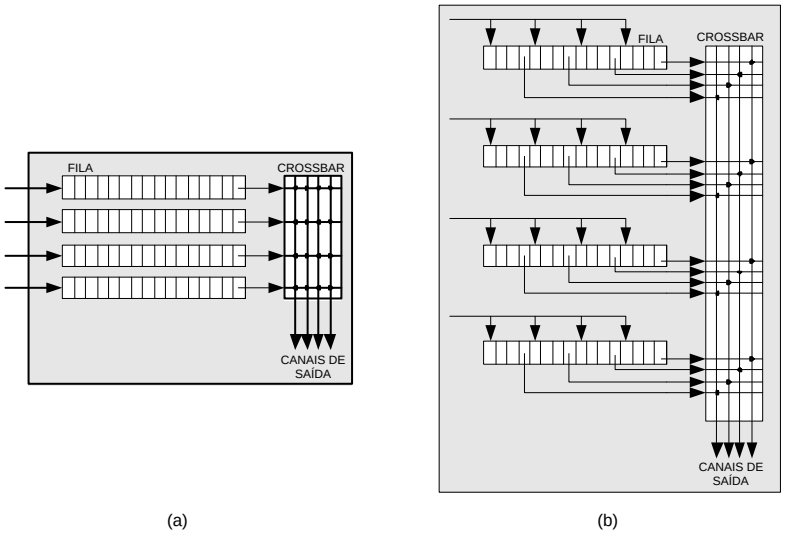


Figura 9 – (a) Roteador com quatro filas *FIFO*. (b) Roteador com quatro filas *SAFC* [69].

Filas SAFC

A estratégia *SAFC* é uma abordagem para amenizar o bloqueio *HOL*. Basicamente, divide-se cada fila de entrada em *N slots* que possuam tamanho igual a 1/*N* do tamanho da fila original. Os *slots* por sua vez devem ser atribuídos individualmente a cada porta de saída, conforme Figura 9 (b). Apesar de simples, essa estratégia possui algumas desvantagens, tais como custo adicional referente ao controle do núcleo de chaveamento e ao controle das *N* sub-filas por porta de entrada. Além disto, a utilização das sub-filas fica restrita a 1/*N* do espaço total, o controle de fluxo torna-se mais complexo, pois é realizado individualmente para cada fila de entrada e o *crossbar* é mais complexo.

Filas SAMQ

A estratégia *SAMQ* foi proposta para simplificar o *crossbar*, fazendo com que as saídas das filas de entrada direcionadas à mesma porta possa ser multiplexadas no tempo, conforme ilustra a Figura 10 (a). Alguns problemas da estratégia *SAFC* são eliminados quando utiliza-se a *SAMQ*, já que o custo do *crossbar* é reduzido. Entretanto, os problemas relacionados com a utilização das filas e o controle de fluxo se mantêm.

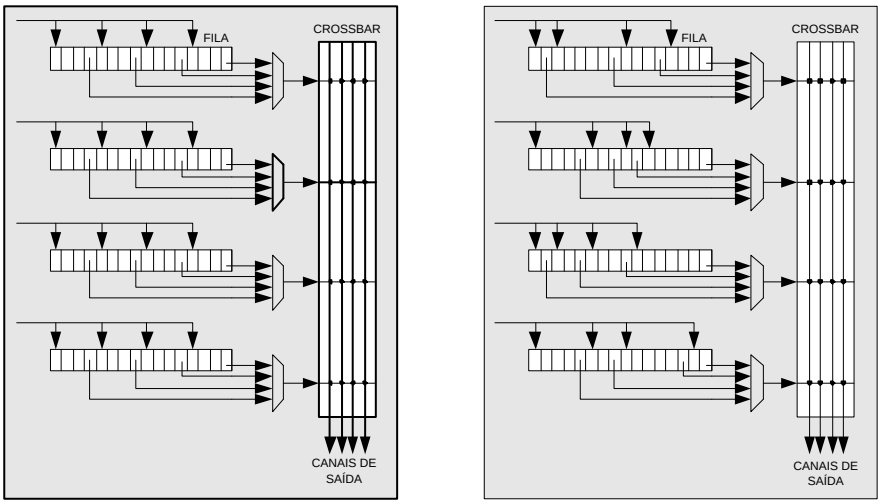


Figura 10 – (a) Roteador com quatro fi las *SAMQ*. (b) Roteador com quatro fi las *DAMQ* [69].

Filas DAMQ

A estratégia *DAMQ* tem como objetivo evitar alguns dos problemas encontrados nas estratégias descritas anteriormente. *DAMQ* propõem um espaço de armazenamento associado a uma porta de entrada e particionado de forma dinâmica entre as portas de saída, conforme a demanda dos pacotes recebidos, o que é ilustrado na Figura 10 (b). Com essa estratégia elimina-se

a possibilidade do bloqueio *HOL* nas filas *FIFO*, aumenta-se a utilização do espaço de memória disponível e o controle de fluxo é mais simples que nas estratégias *SAFC* e *SAMQ*. Como desvantagem, cita-se a implementação física mais complexa para o gerenciamento das filas, baseada em listas encadeadas.

Armazenamento na Saída

O armazenamento temporário de dados na saída implica a inserção de filas nas portas de saída do roteador. O problema desta estratégia é que cada fila deve ser capaz de receber simultaneamente dados das *N* entradas, implicando que a fila de saída possua *N* portas de entrada ou que opere a uma velocidade *N* vezes maior do que as entradas. O uso de armazenamento temporário de saída exige a implementação de um controle de fluxo entre a porta de entrada e de saída, aumentando assim a complexidade do roteador.

Armazenamento Centralizado de Forma Compartilhada

O armazenamento centralizado compartilhado denominado *CBDA* (*Centrally Buffered, Dynamically Allocated*), utiliza filas para armazenamento de pacotes de todas as portas de entrada do roteador. O espaço de memória disponível a ser utilizado é dividido de forma dinâmica entre os pacotes de diferentes entradas. O armazenamento temporário centralizado compartilhado oferece uma melhor utilização de memória do que aquelas proporcionadas pelas abordagens onde este espaço é prévia e estaticamente alocado a portas de entrada.

Segundo Zeferino [69] o *CBDA* deve oferecer no mínimo uma largura de banda igual à soma das larguras de banda de todas as portas, fazendo com que em um roteador *NxN*, a fila possua 2*N* portas de acesso, de modo a permitir *N* acessos simultâneos de leitura e *N* acessos simultâneos de escrita.

Como desvantagem, pode-se citar um problema semelhante ao *HOL* [49, 69]. Onde caso uma porta de saída esteja em uso por uma determinada porta de entrada e ao mesmo tempo outra porta de entrada esteja recebendo dados e também deseje utilizar essa mesma saída, está segunda porta fica com seus dados bloqueados, sendo armazenados na fila centralizada, o que pode acarretar a um preenchimento total da mesma. Quando isso ocorre as outras portas de comunicação são afetadas, pois a fila lotada acarreta a recusa de dados, criando contenção. Este problema pode ser evitado restringindo espaço alocado a cada uma das portas de entrada.

2.1.7 Arbitragem

Suponha a existência de um conjunto de recursos requisitando acesso a um elemento compartilhado por todos. Dá-se o nome de *arbitragem* ao processo de escolha de qual elemento do conjunto de recursos terá acesso ao elemento compartilhado a cada instante. Por exemplo, quando se utiliza chaveamento de pacotes, estes chegam através de canais nas portas de entrada do roteador e são transferidos para os canais das portas de saída. Quando pacotes de diferentes portas chegam simultaneamente ao roteador e requisitam acesso a mesma porta de saída, é necessária uma arbitragem para selecionar quem ganhará o direito a utilizar a porta de saída. O processo de arbitragem é realizado por um *árbitro*. Este tem como função realizar o compartilhamento das portas de saída, garantindo uma utilização balanceada destas pelas portas de entrada, além de assegurar que não existam problemas de *starvation*, ou seja, uma porta de entrada ficar esperando acesso a uma porta de saída indefinidamente.

Existem duas formas de implementação dos árbitros: centralizada ou distribuída. Em um árbitro centralizado, os mecanismos de roteamento e arbitragem costumam ser implementados de forma monolítica, recebendo pacotes de filas de entrada, executando o algoritmo de roteamento e determinando a porta de saída que cada pacote irá utilizar.

Essa abordagem é utilizada em árbitros que procuram maximizar a utilização do *crossbar*, realizando uma arbitragem global, a qual considera todos os pacotes que estejam prontos para transmissão nas entradas, bem como o estado das portas de saída [14, 69].

Em árbitros distribuídos, os mecanismos de roteamento e arbitragem são independentes para cada uma das portas do roteador, existindo um módulo de roteamento em cada porta de entrada e um módulo de arbitragem em cada porta de saída do roteador.

A implementação distribuída permite que sejam construídos roteadores mais rápidos. Entretanto, quando utilizada com algoritmos adaptativos esta abordagem apresenta limitações. Essas ocorrem porque o mecanismo de roteamento pode conseguir rotear o pacote por várias portas de saída, e com isso envia requisições a todos os árbitros dessas portas. Caso mais de um árbitro reserve a sua porta a esse pacote, as portas que não forem escolhidas para envio ficarão ociosas, enquanto poderiam estar enviando pacotes de outras portas de entrada. Por este motivo, roteamentos determinísticos são mais utilizados em conjunto com a arbitragem distribuída.

Quando vários canais de entrada de um roteador estiverem com pacotes prontos para serem transmitidos para as portas de saída, a política de arbitragem irá decidir qual deles irá iniciar a transmissão. Existem vários tipos de política de prioridade, como por exemplo *Round-Robin*, estática, randômica, *First-Come-First-Served*, *Oldest-First*, *Least Recently Served*, entre outras. Cada uma possui características próprias, algumas voltadas para o desempenho, outras para a

simplicidade de implementação.

2.1.8 Redes Toro

Conforme mencionado na Seção 2.1.2, a topologia de uma rede é definida pela estrutura de interligação de seus nodos. Por exemplo, na Figura 11 (a) onde está representada a topologia de uma rede toro bidimensional 3x3, nota-se a presença de 9 roteadores, cada qual conectado a 8 canais (supondo-se que cada enlace da Figura corresponde a dois canais unidirecionais), sendo 1 para cada vizinho e 1 de cada vizinho. Em uma rede toro todos os nodos possuem o mesmo número de enlaces. Isto é a principal diferença entre uma rede com topologia toro e uma com topologia malha, pois até mesmo os nodos localizados nas extremidades da rede possuem ligações com os seus quatro vizinhos. Desta forma uma rede toro consegue rotear de qualquer nodo origem para qualquer nodo destino com enlaces unidirecionais enquanto que em uma rede de topologia malha isto é impraticável.

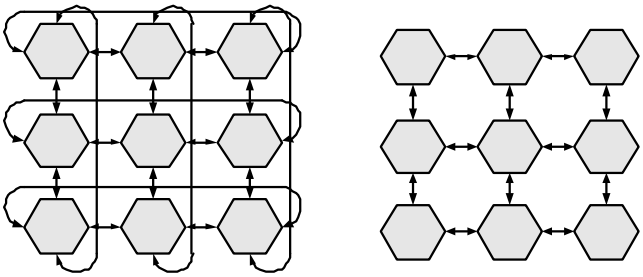


Figura 11 – Exemplos de rede com topologias toro e malha: (a) topologia toro 2D 3x3, mostrando as 8 ligações por roteador; (b) topologia malha 2D 3x3, onde somente o roteador central possui as 8 ligações com seus vizinhos.

Conceitos

Em uma rede toro, os caminhos mínimos são quase sempre fisicamente mínimos também, permitindo que a rede possa explorar a característica de localidade física na comunicação [19]. Um caminho mínimo de um nodo *x* a um nodo *y* é um caminho com o menor *hop count* possível. Uma característica importante de uma rede, é a quantidade de caminhos mínimos disponíveis de um nodo *x* origem até um nodo *y* destino. O conjunto de todos os caminhos mínimos entre *x* e *y* é denotado por *Rxy* [19]. Uma rede que possui múltiplos caminhos mínimos entre a maior parte dos seus nodos, ou seja $|Rxy| > 1$ *para* $x, y \in N$ é mais robusta do que uma rede com somente

um caminho mínimo entre um nodo origem N_1 e um nodo destino N_2, |*Rxy*| = *1* [19]. Quando uma rede possui altos valores de |*Rxy*| diz-se que ela tem uma boa *diversidade de caminhos*, e com isso, é possível obter um bom balanceamento de carga através dos vários caminhos da rede, permitindo inclusive que a rede seja tolerante a falhas dado que o algoritmo de roteamento seja capaz de explorar estas características.

Estrutura

A estrutura de uma rede toro de raiz *k* n-dimensional consiste de *N* = k^n, onde *N* são todos os nodos que estão dispostos em *n* dimensões, tendo *k* nodos em cada uma das dimensões. Todos os nodos em uma rede toro 2D estão interligados com seus vizinhos por um par de canais em cada direção (N/S/E/W), sendo um para envio e outro para recebimento, totalizando 8 canais. O conceito de rede toro pode ser generalizado para incluir toro com raízes mistas onde cada dimensão pode ter um valor de *k* diferente. Esta é a definição mais geral de rede toro, que tem como caso especial redes anel, que nada mais são do que um toro de raiz *k* 1-dimensional.

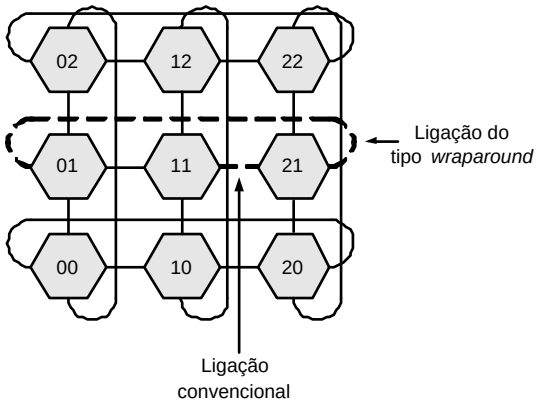


Figura 12 – Topologia de uma rede toro 3x3, salientando as ligações convencionais e as *wraparound* características de um toro.

Em uma rede de comunicação cada nodo possui uma posição relativa na rede. Suponha uma rede toro 2D 3x3, onde a estrutura da rede pode ser caracterizada como tendo além das ligações convencionais dispostas na topologia malha, ligações *wraparound* dos nodos a_{k-1} para os a_0, onde *k* é o número de nodos em cada dimensão da rede. A Figura 12 ilustra esta situação, onde o nodo de número "21" (a_2) possui uma ligação *wraparound* com o nodo "01" (a_0) e uma ligação convencional com o nodo "11"(a_1).

2.2 Redes Intra-Chip

Cada vez mais núcleos IPs e outros componentes reutilizáveis têm sido integrados em um único CI. Benini et al. [9] e Guerrier et al. [26] prevêem que futuramente SoCs sejam compostos por centenas de núcleos IPs.

Tradicionalmente canais dedicados e barramentos são utilizados como abordagem para comunicação entre os núcleos IPs. Os canais dedicados são considerados ineficientes, já que são de difícil reuso. Entretanto, barramentos são reutilizáveis, mas possuem limitações pela diminuição da freqüência de operação com o aumento da densidade de SoCs, devido ao uso compartilhado do canal de comunicação por um número crescente de IPs [69] tornando a escalabilidade limitada.

Algumas abordagens foram utilizadas para tentar resolver essas limitações, como por exemplo, barramentos hierárquicos (CoreConnect [31] e Amba [3]). Entretanto, outros problemas foram agregados, como a possibilidade de operação bloqueante atingindo todo o barramento hierárquico, diferenças entre as freqüências de operação, largura de banda entre cada barramento e principalmente a forma desestruturada de montagem destes [49].

Redes mais complexas para comunicação intra-chip denominadas NoCs, foram propostas para superar os problemas já mencionados, de forma a suprir as necessidades dos projetos atuais [9,18,37]. NoC é um paradigma emergente de projeto, que visa suprir as limitações apresentadas por barramentos tradicionais, trazendo melhorias nos quesitos: (*i*) consumo de energia; (*ii*) escalabilidade da largura de banda; (*iii*) reusabilidade; (*iv*) confiabilidade [26].

NoCs são constituídas por um conjunto de roteadores interligados por meio de canais ponto-a-ponto [27]. Elas surgiram a partir de conceitos (topologias de rede, roteamento, chaveamento, protocolos de comunicação etc.) das áreas de rede de computadores e sistemas distribuídos devidamente adaptados para utilização em ambiente intra-chip. O que diferencia redes intra-chip de redes de computadores são os requisitos do sistema, onde na primeira o número de núcleos e a distância entre eles são inferiores aos da segunda.

Contribuições e revisões conceituais sobre NoCs foram publicadas por vários autores. Destacam-se Benini [9], Dally [18] Guerrier [26], Duato [22], Pande [56] entre outros. Para implementações de NoCs pode-se citar [5,30,41,46,64]. Em [46] é encontrada uma revisão abrangente do estado da arte em redes intra-chip realizada por Moraes et al.

3 Estado da Arte em NoCs Toro

Este Capítulo é um apanhado do estado da arte para redes intra-chip com topologia toro encontradas atualmente na literatura especializada. O resultado deste apanhado está sintetizado através da Tabela 1. Nela, cada linha corresponde a uma rede intra-chip com topologia toro distinta. Um apanhado geral de NoCs com múltiplas topologias representando o estado da arte no ano de 2004 pode ser encontrado na referência [46].

3.1 Características das NoCs Revisadas

Esta Seção descreve brevemente cada uma das NoCs toros descritas na literatura consultada durante a realização deste trabalho. A Seção também revisa uma comparação entre algumas classes relevantes de topologias de NoCs, conforme proposta por Pande e outros.

Dally e Towles [18] propõem uma rede toro dobrado 2D. Para utilização em conjunto com essa topologia é proposto um algoritmo de roteamento XY em conjunto com canais virtuais, de forma a evitar a ocorrência de *deadlocks* na rede. Marescaux et al. [41] utilizam uma rede de comunicação toro 2D com modo de chaveamento *wormhole* especialmente modificada para limitar a área do roteador. Sabe-se que em uma rede de topologia malha o roteador deve realizar o roteamento em todas as direções Norte, Sul, Leste e Oeste de forma bidirecional, para garantir conectividade. Por outro lado, em uma rede com topologia toro, é possível diminuir a complexidade do roteamento e do roteador modificando ambos para utilizar somente dois sentidos de deslocamento, um no eixo das coordenadas X e outro no eixo das coordenadas Y. Entretanto, Marescaux verificou que essa prática aumenta em 15% o consumo de energia em relação a uma topologia malha análoga para uma rede 4x4.

Hölzenspies et al. propõem, em [30], um modelo de comunicação baseado em uma arquitetura toro de *n* dimensões, utilizando modo de chaveamento *wormhole*. Este trabalho tem como principal contribuição a extensão do roteamento introduzido em uma arquitetura de duas dimensões para uma topologia toro n-dimensional.

Em [64], Theocharides et al. propõem uma arquitetura de rede neural reconfigurável, implementada em uma rede intra-chip toro 2D com roteamento XY e modo de chaveamento

wormhole. Essa arquitetura permite um grande número de conexões entre os neurônios da rede, provendo um bom desempenho quando comparado com implementações alternativas. Theocharides et al. acreditam que implementações em NoCs podem resolver o problema de congestionamento de projetos anteriores da mesma equipe de forma econômica, já que podem ser reconfiguráveis e ter alto desempenho.

Chi e Chen [16] propõem um projeto e implementação de um roteador para redes com topologia malha ou toro. Neste são utilizadas filas de entrada e saída e roteamento especial, denominado *look-ahead* para eliminar problemas de bloqueio dos pacotes nas filas. Nas simulações realizadas, o desempenho do roteador proposto se mostrou melhor do que um roteador que utiliza filas FIFO.

Bartic et al. [7] acreditam que diferentes tipos de redes serão necessárias, dependendo do domínio da aplicação visada. Por este motivo, propõem um projeto de rede flexível e escalável, de forma que se possa modificar a rede facilmente para cada necessidade. O projeto é adequado para construção de redes com topologia irregular, baixa latência e alto desempenho. Em [7] é realizada uma simulação com diferentes topologias de rede, entre elas toro e malha, ambas com dimensões 4x4. Nesta simulação, foi constatado que uma rede toro com canais unidirecionais possui tamanho, número de *LUTs* e *slices* menores quando comparado com a topologia malha, por usar roteamento unidirecional em X e Y.

Kim et al. [36] propõem uma arquitetura de roteador que utiliza roteamento adaptativo enquanto mantêm uma baixa latência. Nesta arquitetura, é utilizada uma rede toro 2D de dimensões 8x8, com pacotes de 4 *flits*, modo de chaveamento *wormhole*, filas com 4 *flits* e 6 canais virtuais por canal físico.

O toro diagonal recursivo (em inglês, RDT) é uma proposta oriunda de uma rede de interconexão voltada para uso em computadores maciçamente paralelos [67], onde o objetivo é reduzir drasticamente o diâmetro da rede, visando obter baixíssimas latências de comunicação. Como exemplo, um toro 2D de 1024 vértices (assumindo 32x32) possui diâmetro de 32, enquanto que em um RDT pode-se obter um diâmetro de apenas 7 para o mesmo número de vértices. A topologia é uma rede direta composta pela superposição de n toros de diferentes *graus*. Cada toro envolve uma parte dos P vértices da rede. O toro de grau-1 é um toro 2D com enlaces girados de 45 graus e conectando apenas 1 a cada 4 vértices em X e Y. O toro de grau-2 é um toro 2D girado 90^o (assim equivalendo a um toro convencional) conectando apenas 1 a cada 8 vértices em X e Y. Daí infere-se os toros de grau superior. O algoritmo de roteamento é complexo [68], sendo baseado na decomposição de um vetor resultante XY entre os vértices origem e destino em um conjunto de vetores componentes sobre toros de graus decrescentes. O número de enlaces de cada roteador na rede é proporcional ao número de graus usado. Por exemplo, em

roteador de um RDT com grau máximo 2 possui 8 enlaces, enquanto que se o grau máximo é 0, ele tem 4 enlaces, (idêntico ao toro 2D convencional).

Pande et al. demonstram em [55, 56] um método de avaliação desenvolvido para comparar o desempenho e as características de diversas topologias e arquiteturas de NoCs. Neste trabalho são comparadas NoCs descritas por outros autores, incluindo: (*i*) a NoC SPIN [26], proposta por Guerrier and Greiner a qual utiliza uma arquitetura denominada árvore gorda com 16 nodos; (*ii*) a NoC CLICHÉ [37], proposta por Kumar et al., baseada em uma rede com topologia malha de 16 nodos; (*iii*) a NoC OCTAGON [35], proposta por Karim et al., onde se utiliza uma arquitetura básica de um octágono com 8 nodos e 12 canais bidirecionais; (*iv*) a Arquitetura *Butterfly Fat-Tree* proposta em [56], com 16 nodos; (*v*) a NoC Toro 2D proposta por Dally e Towles [18] com 16 nodos, modificada para um toro dobrado de duas dimensões.

Pande em seu estudo de caso utilizando o método de avaliação proposto em [56] concluiu que a rede toro dobrado em comparação com a rede CLICHÉ, dependendo do tipo de tráfego utilizado, pode obter uma maior ou menor vazão, mas sempre obtendo uma menor latência média. Neste trabalho [56], várias comparações foram realizadas envolvendo todas as arquiteturas de NoCs descritas anteriormente, comparando quesitos como área, vazão, latência e energia. Em outro trabalho [55] Pande verificou que a topologia toro 2D consegue uma maior vazão em relação à topologia malha, quando comparadas em uma rede de mesmas dimensões sujeita a tráfego aleatório uniformemente distribuído.

3.2 Comparação de NoCs Toro

Esta Seção descreve em detalhes cada uma das NoCs toro consultadas durante a realização deste trabalho. Na Tabela 1 é demonstrado um resumo das características de cada rede, sendo que uma maior descrição é realizada nas Seções a seguir.

Tabela 1 – Estado da arte em NoCs com topologia Toro. ND = Informação Não Disponível na literatura consultada.

NoC	Topologia	Algoritmo de Roteamento	Modo de Chaveamento	Canais Virtuais	Armazenamento Temporário	Implementação e/ou Prototipação
Dally e Towles (2001) [18]	Toro 2D	XY	*Wormhole*	Sim	Entrada	Não, apenas conceitual
Marescaux et al. (2002) [41]	Toro 2D	XY	*Wormhole*	Sim (2 canais para evitar *deadlocks*)	Entrada	Virtex XCV800
Hölzenspies et al. (2003) [30]	Toro 2D (extensível para toro n-dimensões)	XY	*Wormhole*	Sim (n+1 onde n = nº de dimensões da rede)	ND	Simulando situações de melhor caso, pior caso e tráfego aleatório
Theocharides et al. (2004) [64]	Toro 2D	XY	*Wormhole*	ND	ND	Sintetizando e Simulando (5 *testbenchs* diferenciados)
Chi e Chen (2004) [16]	Roteador para utilizar em rede Toro 2D	*Look-ahead* para evitar bloqueio HOL	ND	Não	Entrada e Saída	Implementado como um Núcleo IP
Bartic et al. (2005) [6, 7]	Topologia adaptativa incluindo Toro	Determinístico com *Look-up Table* (XY)	Virtual Cut-through	Não	Saída	Virtex-II Pro
Kim et al. (2005) [36]	Toro 2D	Adaptativo, com técnica de *look-ahead* para detecção de congestionamento na rede	*Wormhole*	Sim	ND	Simulando com vários padrões de tráfego
Yang et al. (2005) [68]	Toro Diagonal Recursivo	*Vector Routing*	ND	ND	ND	Não, apenas proposta de estrutura
Mercury (2005)	Toro 2D	Algoritmo CG1 de Cypher e Gravano [17]	Virtual Cut-through	Não	Centralizado Compartilhado	Virtex-II Pro XC2VP30

3.2.1 Topologia de Rede

Apesar de existirem muitas redes de computadores com topologia toro, são ainda escassos os casos de empregado desta topologia em NoCs. A topologia malha é a mais predominante neste tipo de rede no momento. Existem algumas razões para isso. Entre as três maiores vantagens da topologia malha pode-se citar: *(i)* facilidade de implementação; *(ii)* simplicidade na estratégia para roteamento, quando utilizado roteamento XY; *(iii)* escalabilidade [46]. A topologia toro de duas dimensões é utilizada como alternativa à malha com o intuito de reduzir o diâmetro da rede [41], enquanto que o toro dobrado 2D tem como principal vantagem a redução do custo de fios quando comparada com o toro 2D convencional [18].

Analisando a segunda coluna da Tabela 1, onde é considerado o quesito topologia, somente Dally [18] propõem o uso do toro dobrado. Hölzenspies [30] propõem uma topologia toro 2D com possibilidade de extensão para *n* dimensões, Bartic [7] propõem uma rede onde a topologia pode ser substituída, sendo toro 2D uma das opções e Yang [68] propõem um toro diagonal recursivo. Todas as outras NoCs presentes na tabela empregam toro 2D como topologia.

3.2.2 Algoritmo de Roteamento

O próximo parâmetro comparado na Tabela 1 é o algoritmo de roteamento. O algoritmo XY [24] é o mais utilizado em redes toro 2D [18, 30, 41, 64], devido a sua facilidade de implementação. Contudo, trata-se de um algoritmo bloqueante, determinístico e baseado em *hops*. Seu funcionamento é baseado em endereços relativos ao seu destino, não necessitando que os roteadores tenham conhecimento da estrutura da rede como um todo. Inicialmente, o pacote é roteado no eixo cartesiano X até atingir a coordenada correta nesta direção, após, é roteado no eixo Y até alcançar a coordenada correta, atingindo assim o roteador destino. Existem diversas variações possíveis deste algoritmo devido ao uso diferenciado de canais virtuais e métodos de armazenamento temporário.

Chi e Chen [16] utilizam um roteamento denominado "*look-ahead*" onde a decisão do roteamento é realizada um *hop* antes do pacote chegar ao roteador. Este método de roteamento foi desenvolvido especialmente para roteadores que possuem filas de entrada e saída de modo a prevenir perdas de desempenho por causa do problema de bloqueio de cabeça de fila [58]. Kim [36] propõe um algoritmo de roteamento que utiliza a técnica de *look-ahead* para verificar o congestionamento nos roteadores vizinhos, de forma a tornar o roteamento adaptativo baseado no fluxo de dados. Bartic [7] utiliza um algoritmo de roteamento determinístico implementado

como uma *look-up table*. Este algoritmo utiliza uma tabela onde cada roteador da rede possui uma entrada de modo a conter o roteamento para o próximo canal levando ao próximo roteador. O grande problema desta abordagem é a área ocupada em NoCs de tamanho médio a grande, já que quanto maior forem as dimensões da rede, maior será a tabela presente em cada roteador.

Na rede Mercury descrita no contexto deste trabalho é utilizado um algoritmo denominado Algoritmo CG1, proposto por Cypher e Gravano em [17]. Este algoritmo utiliza somente caminhos mínimos e é completamente adaptativo, ou seja, todos os caminhos mínimos entre uma origem e um destino podem ser utilizados para evitar congestionamento. Esse algoritmo será abordado em detalhe no Capítulo 4.

3.2.3 Modo de Chaveamento e Canais Virtuais

No parâmetro modo de chaveamento, descrito na quarta coluna da Tabela 1, as redes propostas por Dally [18], Hölzenspies [30], Kim [36], Marescaux [41] e Theocharides [64], pressupõem o uso de chaveamento de pacotes utilizando o modo *wormhole*. Devido à probabilidade deste modo causar *deadlock* na rede, é utilizado em conjunto canais virtuais para quebrar os ciclos presentes na rede, evitando assim a ocorrência do problema. Desta forma as NoCs propostas em [18, 30, 36, 41] utilizam canais virtuais, cada qual da maneira que melhor lhe convém para sua implementação. A NoC proposta por Bartic [7] e a rede Mercury utilizam o modo de chaveamento *virtual cut-through* e nenhuma delas, nem a NoC implementada por Chi e Chen [16] utilizam canais virtuais.

A escolha do modo de chaveamento *virtual cut-through* para a rede Mercury se deve a diversos fatores. Segundo Shin e Rexford [60], o modo de chaveamento *virtual cut-through* consegue manter uma baixa latência na rede em relação ao modo *store-and-forward*, pois os pacotes são enviados adiante assim que chegam no roteador, caso a porta de saída requerida pelo roteamento esteja disponível. Este modo alcança, com baixo tráfego a mesma latência do modo de chaveamento *wormhole*, enquanto que com alto tráfego alcança a mesma alta vazão do modo *store-and-forward* [60]. Uma rede que utiliza *virtual cut-through* possui uma baixa latência enquanto mantém alto desempenho, ao custo de uma maior quantidade de área necessária para filas.

Banerjee et al. [5] demonstram que o modo *virtual cut-through* consegue baixas latências e altas taxas de aceitação de dados por roteador, quando comparado com o modo *wormhole*, desde quando utilizado com somente 1 enlace entre roteadores com a utilização de até 8 canais virtuais. Ao mesmo tempo, verifica-se que o consumo de energia é praticamente o mesmo para

os dois modos de chaveamento. Desta forma Banerjee et al. afirmam que *virtual cut-through* consegue um melhor desempenho a um custo de energia similar ao *wormhole*.

3.2.4 Armazenamento Temporário

A técnica de armazenamento temporário é o quinto parâmetro analisado na Tabela 1. Algumas NoCs toro analisadas empregam filas nas portas de entrada como as propostas por Dally e Towles [18] e Marescaux et al. [41]. Este tipo de implementação pode ocasionar o já conhecido problema de bloqueio da cabeça da fila. Para solucionar esse inconveniente, pode-se utilizar filas de saída em conjunto com as de entrada, como no caso do roteador proposto por Chi e Chen [16]. Entretanto, essa solução acaba gerando uma maior utilização de área pelo roteador.

Segundo Moraes et al. [46], o tamanho da fila é um importante parâmetro, que implica no compromisso entre a quantidade de contenção da rede, a latência dos pacotes e a área utilizada pelo roteador. Contenção da rede é a medida da quantidade de recursos da rede alocados pelo bloqueio de pacotes. Filas grandes proporcionam uma pequena contenção na rede, uma alta latência de pacotes e roteadores de tamanho grande. Por sua vez, filas pequenas proporcionam uma situação contrária a essa.

Em NoCs, o quesito área é de suma importância. Considerando que o modo de chaveamento escolhido para a rede Mercury não prioriza este aspecto, optou-se por um algoritmo que utilizasse filas centralizadas e compartilhadas entre todas as portas do roteador, de forma a diminuir a área utilizada em comparação com as filas de entrada e/ou saída.

3.2.5 Implementação e/ou Prototipação

Por fim, na última coluna mostra-se a disponibilidade de dados de implementação e/ou prototipação das NoCs revisadas. Marescaux et al. [41] utilizou um FPGA Virtex XCV800 para prototipar sua rede. Hölzenspies et al. [30], somente simulam a rede utilizando-se de três tipos de cenários de testes, enquanto Theocharides et al. [64] simula utilizando cinco tipos de aplicações como *testbenchs*, além de sintetizar sua proposta de arquitetura. Já Chi e Chen [16] implementam somente seu roteador em um IP *Core*, o qual pode ser utilizado em uma rede toro. Bartic et al. [7] prototipam sua NoC utilizando um FPGA Virtex-II Pro e Kim et al. [36] simularam sua rede com vários padrões de tráfego. Para a rede Mercury, vários *testbenchs* foram realizados para validação da NoC e a mesma foi prototipada em um FPGA Virtex-II Pro.

Mais informações sobre a prototipação da rede Mercury serão trazidas no Capítulo 8.

3.3 Evolução da Pesquisa em NoCs

Nos últimos anos tem se destacado o volume de pesquisas em topologias para redes intra-chip em geral, devido a necessidade de um aumento no desempenho da rede e na eficiência do consumo de energia utilizada na mesma. Nesta Seção revisam-se algumas abordagens da evolução destas pesquisas.

Três direções de pesquisa relacionadas a topologias de NoCs envolvem a relação das topologias com a aplicação, com o consumo de energia e com a evolução das tecnologias de fabricação. Murali e De Micheli em [50] apresentam uma ferramenta denominada SUNMAP a qual seleciona de forma automatizada a melhor topologia (dentro das disponibilizadas pela ferramenta) para uma determinada aplicação e realiza o mapeamento dos núcleos IP. Estes Autores acreditam que a escolha da topologia é uma importante fase no desenvolvimento do projeto de NoCs. Hu et al. [29] exploram uma metodologia de síntese física para gerar NoCs com uma boa relação energia/eficiência. Como resultados de experimentos, conseguem reduzir o consumo de energia de forma significativa, otimizando a topologia de rede, localização das células, capacidade dos canais e estilo de ligações dos fios. Wang et al. [65] colocam em seu trabalho que a topologia e a tecnologia empregada nos semicondutores possuem um alto impacto na energia consumida pela rede. Afirmam que uma determinada topologia que hoje é considerada a melhor para determinada aplicação utilizando-se da tecnologia atual, pode deixar de sê-lo com o passar do tempo, devido a novos parâmetros tecnológicos. Os Autores de [65] realizam comparações entre o consumo de energia de determinadas topologias *versus* a tecnologia empregada (70*nm*, 50*nm* e 35*nm*).

Outra direção de pesquisa envolve a relação entre topologias, algoritmos de roteamento e consumo de energia associado. Por exemplo, nas primeiras propostas de roteadores para NoCs, o uso de roteadores simples com algoritmos de roteamentos determinísticos dominavam. Com a evolução da pesquisa começaram a surgir propostas utilizando canais virtuais em conjunto com roteamentos mais elaborados [36]. Notou-se que quando utilizado tráfego não uniforme ou aplicações especificas, como por exemplo multimídia em tempo real, o roteamento determinístico não consegue rotear rapidamente pacotes de modo a evitar congestionamento na rede, resultando em atraso na comunicação. A solução encontrada para resolver esse problema é o emprego de algoritmos adaptativos. Estes porém possuem maior complexidade de hardware e possivelmente consumiram mais energia. Kim et al. [36] acreditam que reduzindo a latência da

rede e aumentando a vazão com a utilização de algoritmos adaptativos, o pequeno gasto extra de energia quando comparado com algoritmos determinísticos, ficará anulado pelo desempenho que o roteamento conseguirá em tráfego não uniforme.

Kim et al. [36] avaliam sua arquitetura utilizando uma rede toro 2D de dimensões 8x8, com pacotes de 4 flits, modo de chaveamento *wormhole*, filas com 4 *flits* e 6 canais virtuais por canal físico. Para validação, foram realizadas simulações de injeções de pacotes utilizando três tipos diferentes de carga: (*i*) tráfego internet auto-similar; (*ii*) tráfego uniforme; (*iii*) tráfego MPEG-2. Em cada uma das simulações foram utilizados quatro diferentes tipos de permutações de tráfego: (*i*) normal-aleatória; (*ii*) transposta; (*iii*) tornado; (*iv*) complemento. Com esses testes, Kim et al. validaram sua hipótese, de que a energia gasta quando se utilizam algoritmos adaptativos é em geral menor do que com algoritmos determinísticos, devido à redução da latência da rede.

Redes intra-chip com topologia toro estão sendo atualmente estudadas, construídas, testadas e comparadas em vários contextos e com objetivos distintos. Isto reforça ainda mais a necessidade de um estudo aprofundado desta topologia. Desta forma, o projeto do roteador presente na rede Mercury foi pensado para ser diferente do que já vem sendo trabalhado no meio acadêmico. Grande parte das implementações toro utilizam algoritmos XY, modo de chaveamento *wormhole*, canais virtuais e filas na entrada, enquanto a implementação proposta da rede Mercury utiliza um algoritmo com modo de chaveamento *virtual cut-through*, filas centralizadas compartilhadas, sem canais virtuais. O algoritmo é livre de *deadlock*, *livelock* e *starvation*. No Capítulo 4 detalha-se o algoritmo escolhido. Para dar suporte ao desenvolvimento da geração da rede Mercury e análise das simulações realizadas, foram propostas duas ferramentas de apoio, as quais serão apresentadas no Capítulo 7.

4 Algoritmo de Roteamento

4.1 Algoritmo de Cypher e Gravano

Cypher e Gravano da IBM propuseram dois algoritmos de roteamento para redes de chaveamento de pacotes voltados para processamento paralelo, utilizando redes de interconexão com topologia toro, de raiz mista e dimensões arbitrárias [17]. Neste Capítulo detalha-se o primeiro destes algoritmos, denominado CG1, o qual foi escolhido para definir as características da NoC Mercury proposta aqui, por ser o mais simples dentre os dois, para implementação em *hardware*.

CG1 é mínimo e completamente adaptativo, ou seja, todos os caminhos mínimos disponíveis podem ser usados para o roteamento de pacotes. Além disso, o algoritmo é livre de *deadlock*, *livelock* e *starvation* e pressupõe roteamento no modo *store-and-forward* ou no modo *virtual cut-through*, não sendo viável seu uso no modo *wormhole*.

4.1.1 Definições e Pressupostos

Algumas definições e pressupostos devem ser introduzidos antes de se iniciar o estudo propriamente dito do algoritmo. Assume-se aqui uma rede com topologia toro onde o número de dimensões é qualquer, embora definido para cada instância da rede. Cada dimensão comporta um número arbitrário de pontos na rede, denominados *nodos*. Um *nodo* é um núcleo de propriedade intelectual associado a um roteador que implementa localmente o algoritmo CG1. Isto caracteriza o uso de redes diretas [19]. A maioria dos exemplos apresentados para ilustrar os conceitos assume uma rede toro 2D. Contudo, todo o desenvolvimento neste Capítulo é genérico, aplicando-se a toros com qualquer número de dimensões. Com relação à conectividade entre nodos, que caracteriza a rede toro, tem-se:

- O algoritmo CG1 opera em redes toro com raízes mistas k_{d-1} x k_{d-2} x ... x k_0 e dimensão *d*, onde $d \geq 1$ e $k_i \geq 2$ para todo *i*, $0 \leq i < d$. Se *d=1*, trata-se de uma rede também conhecida como anel.

- Cada nodo na rede toro possui um identificador único com a forma (a_{d-1}, a_{d-2}, ..., a_0), onde *0* $\leq a_i <$ k_i para todo *i, 0* $\leq$ *i* < *d*.

- Cada nodo da rede, identificado por (a_{d-1} ... a_{i+1}, a_i, a_{i-1} ... a_0), é conectado a todos os nodos identificados por (a_{d-1} ... a_{i+1}, $a_i \pm$ *mod* k_i, a_{i-1} ... a_0), onde *0* $\leq$ *i* < *d*.

- As linhas que conectam os nodos identificados por (a_{d-1} ... a_{i+1}, $k_i - 1$, a_{i-1} ... a_0) e (a_{d-1} ... a_{i+1}, *0,* a_{i-1} ... a_0) são chamadas de *linhas wraparound*, e todas as demais são chamadas de *linhas internas*.

4.1.2 Modelo de Roteamento

O algoritmo proposto por Cypher e Gravano pode utilizar modos de roteamento *store-and-forward* ou *virtual cut-through*. As filas podem ser de qualquer tamanho, as de injeção/entrega não precisam sequer existir, sendo contudo sua definição útil para a apresentação do algoritmo, como ver-se-á adiante. Dados os nodos origem e destino e o nodo onde o pacote está armazenado, o algoritmo computa a seqüência de filas a ser percorrida pelo pacote, para cada posição na rede (nodo, fila) em que se encontra o pacote. O conjunto de filas possíveis para onde o pacote pode-se mover a seguir é denominado *conjunto de espera*. Todas as filas desse conjunto pertencem ao nodo atual ou a vizinhos deste. Nota-se que a fila de injeção não pode pertencer ao conjunto de espera, e o conjunto de espera de pacotes na fila de entrega é vazio, já que a sua única opção de movimento na rede é ser entregue, ou seja, sair da rede. Um conjunto de filas é utilizado em cada um dos nodos que compõem a rede toro. Estas filas se classificam em três tipos, de acordo com a função que desempenham no roteador:

- Fila de injeção: é utilizada por um nodo para inserir pacotes novos na rede;

- Fila de entrega: quando o pacote está no seu nodo destino, ele é retirado da rede a partir desta fila;

- Filas padrão: utilizadas para armazenamento temporário durante o caminhamento do pacote na rede.

Um pacote move-se de uma fila para outra se a fila para onde o pacote está se movendo pertence ao conjunto de espera computado pelo roteador onde o pacote está armazenado. Quando um pacote está sendo movido de uma fila para outra, assume-se que este ocupa as duas filas por uma duração finita de tempo.

CG1 assume que a rede possui as seguintes propriedades: (*i*) o pacote que estiver na fila de entrega do seu nodo destino, será eventualmente removido da rede, em um tempo finito; (*ii*) nenhum pacote permanece para sempre em uma fila se existe alguma fila no seu conjunto de espera; (*iii*) nenhum pacote permanece para sempre em uma fila enquanto um número finito de outros pacotes entra e sai de algumas das filas do seu conjunto de espera.

4.1.3 Ordenamento de Nodos

Definem-se dois tipos diferentes de ordenamento total dos nodos em uma rede toro para utilizar CG1. O primeiro tipo é denominado *crescente à direita*. Trata-se de um ordenamento onde a numeração cresce da esquerda para a direita ao longo das linhas de nodos da rede, conforme ilustra a Tabela 2 para uma rede toro 2D 6x6. O segundo tipo de ordenamento, denominado *crescente à esquerda* é exatamente o oposto do crescente à direita, conforme ilustrado na Tabela 3 para a mesma rede.

Tabela 2 – Ordenamento crescente à direita para uma rede toro 2D 6 x 6.

30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

Tabela 3 – Ordenamento crescente à esquerda para uma rede toro 2D 6 x 6.

5	4	3	2	1	0
11	10	9	8	7	6
17	16	15	14	13	12
23	22	21	20	19	18
29	28	27	26	25	24
35	34	33	32	31	30

Dá-se a seguir um conjunto de definições formais que viabilizam computar estes ordenamentos. Dado um inteiro *i*, $0 \leq i < d$, sendo *d* o número de dimensões do toro ($d \geq 1$), seja a função *g(i)* definida por:

$$g(i) = \prod_{j=0}^{i-1} k_j.$$

Assume-se *g(0) = 1*. O símbolo k_j representa o número de nodos na dimensão *j* sendo $k_j \geq 2$. Dado um rotúlo de um nodo de uma rede toro qualquer, representado por (a_{d-1},a_{d-2}, ..., a_0), define-se a função *Eval* como:

$$Eval(a_{d-1}, a_{d-2}, ..., a_0) = \sum_{i=0}^{d-1} g(i)a_i.$$

Deste modo, a função *Eval* associa um único inteiro entre *0* e *n - 1* para cada nodo da rede, onde *n* é o número total de nodos nesta.

Para esclarecer o uso destas definições, mostra-se a seguir um exemplo de cálculo do índice de um nodo. Seja uma rede toro 2D 6x6 como as ilustradas nas Tabelas 2 e 3. Calcula-se nesta rede o valor da função *Eval* para o nodo localizado na quinta linha e quarta coluna de roteadores. Este nodo possui índices a_1 = 4 e a_0 = 3, dado que a numeração inicia em 0 para todas as dimensões, e que a_0 indica a abscissa e a_1 indica a ordenada. Logo deseja-se computar *Eval(3,4)*. Como d=2 (rede 2D), tem-se

$$Eval(3, 4) = \sum_{i=0}^{2-1} g(i)a_i.$$

Expandindo o somatório, obtém-se:

$$Eval(3,4) = g(0).a_0 + g(1).a_1.$$

Conforme descrito anteriormente, assumiu-se que *g(0)* tem valor *1*, e que *g(1)* é calculado através da equação cujo lado esquerdo é o produtório já descrito. Assim, tem-se que:

$$g(i) = \prod_{j=0}^{0} k_0.$$

Assim, g(1) = k_0 = 6. Logo, computa-se *Eval(3,4)*, como segue:

$$Eval(3,4) = 1.3 + 6.4$$
$$Eval(3,4) = 3 + 24$$
$$Eval(3,4) = 27.$$

Cada um dos diferentes ordenamentos (crescente à direita e crescente à esquerda) pode ser obtido pelo uso da função *Eval* e pelo uso de uma função de translação de índices dos nodos, como descrito a seguir. Dado o inteiro $k_i \geq 2$ e qualquer inteiro a_i onde $0 \leq a_i < k_i$, então tem-se:

$$f_R(a_i,k_i) = a_i$$
$$f_L(a_i,k_i) = k_i - a_i - 1$$

As funções acima serão usadas para gerar os ordenamentos crescente à direita e crescente à esquerda, respectivamente. A Tabela 4 apresenta o resultado de aplicar f_R e f_L a uma linha de uma rede toro com dimensão k_i de tamanho 7.

Tabela 4 – Resultado da aplicação das funções f_R, f_L para k_i = 7.

a_i	0	1	2	3	4	5	6
$f_R(a_i,7)$	0	1	2	3	4	5	6
$f_L(a_i,7)$	6	5	4	3	2	1	0

A partir as definições de *Eval*, f_R e f_L, pode-se agora estabelecer as funções geradoras dos ordenamentos crescente à direita e crescente à esquerda. Dado um nodo de índice (a_{d-1},a_{d-2}, ..., a_0), tem-se:

Right ((a_{d-1},a_{d-2}, ..., a_0) = Eval(($f_R(a_{d-1},kd-1)$,$f_R(a_{d-2},kd-2)$, ... , $f_R(a_0,k0)$)), e
Left ((a_{d-1},a_{d-2}, ..., a_0) = Eval(($f_L(a_{d-1},kd-1)$,$f_L(a_{d-2},kd-2)$, ... , $f_L(a_0,k0)$)).

Adicionalmente, diz-se que a transferência de um pacote de um nodo *a* para um nodo adjacente *b*, ocorreu *para a direita* (respectivamente *para a esquerda*) se e somente se o nodo *a* é menor que o nodo *b*, quando eles são numerados pelo ordenamento crescente à direita (respectivamente para crescente à esquerda). Por exemplo, uma transferência na direção positiva ao longo de nodos internos ocorre para a direita enquanto uma transferência na direção positiva através de um *wraparound* ocorre para a esquerda.

4.1.4 Notação

Para que seja possível entender o algoritmo proposto por Cypher e Gravano, deve-se estabelecer algumas notações resumidas na Tabela 5. Considere que *p* é um pacote arbitrário que está sendo roteado em uma rede toro, utilizando-se do algoritmo descrito na Seção 4.1.5. O cálculo dos caminhos mínimos é conceito fundamental, usado mas não discutido em [17]. Remete-se o leitor interessado ao Capítulo 5 desta dissertação para uma explicação de como este cálculo pode ser feito.

Tabela 5 – Notação utilizada para o Algoritmo CG1. Nesta, o símbolo *p* representa um pacote na rede.

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*, representando as filas para onde o pacote pode ser movido no próximo passo.
neighb(p)	Conjunto de nodos que são vizinhos de *node(p)*.
ok_nodes(p)	Subconjunto de *neighb(p)* que consiste dos nodos vizinhos que estão dispostos sobre algum caminho mínimo entre *node(p)* e *dest(p)*.
ok_queues(p)	Conjunto de filas em *ok_nodes(p)* acessíveis diretamente a partir de *node(p)*.

4.1.5 Algoritmo

O Algoritmo CG1 pressupõe a existência de três filas centrais denominadas *A*, *B* e *C* em cada um dos nodos da rede. CG1 utiliza roteamento de pacotes por qualquer um dos caminhos mínimo da origem até o destino e é livre de *deadlock*, *livelock* e *starvation*. CG1 é descrito a seguir.

Seja *p* um pacote arbitrário que está sendo roteado e que *q = queue(p)*, *x = node(x)*. O conjunto de espera (*wait(p)*) é neste caso geral formado a partir da aplicação das seguintes regras:

- 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 em *x*.
- Caso 2: Se *q* é a fila *A*, então existem duas possibilidades:
  - Se existe um nodo *y* vizinho de *x* que faz parte de algum caminho mínimo tal que *Right(x) < Right(y)*, o conjunto de espera contém todas as filas *A* que pertencem aos nodos acessíveis diretamente a partir de *x* e que pertencem a algum caminho mínimo do pacote, ou seja, todas as filas *A* contidas em *ok_queues(p)*.
  - Caso a condição anterior não se verifique, o conjunto de espera contém unicamente a fila *B* do nodo onde o pacote está armazenado.
- Caso 3: Se *q* é a fila *B*, então existem duas possibilidades:
  - Se existe um nodo *y* vizinho de *x* que faz parte de algum caminho mínimo tal que *Left(x) < Left(y)*, o conjunto de espera contém todas as filas *B* que pertencem aos

nodos acessíveis diretamente a partir de *x* e que pertencem a algum caminho mínimo do pacote, ou seja, todas as filas *B* contidas em *ok_queue(p)*.

  - Caso a condição anterior não se verifique, o conjunto de espera contém unicamente a fila *C* do nodo onde o pacote está armazenado.

- Caso 4: Se *q* é a fila *C*, então existem duas possibilidades:

  - Se *x* não é o nodo destino, o conjunto de espera contém todas as filas *C* que pertencem aos nodos acessíveis diretamente a partir de *x* e que pertencem ao caminho mínimo do pacote, ou seja, todas as filas *C* contidas em *ok_queue(p)*.
  - Se *x* é o nodo destino, o conjunto de espera contém 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.

Para ilustrar a funcionalidade do algoritmo de roteamento, será mostrado a seguir o caminhamento de um pacote numa rede toro 2D 6x6 utilizando o CG1. Considerando uma rede toro 6x6, seja um pacote *p* que deve ser roteado da origem (3,1) para o destino (1,4). Na Figura 13 (a) o roteador origem é indicado pela letra "O" e o destino pela letra "D". O primeiro passo a ser realizado no roteamento é a inserção do pacote na rede e logo após a ordenação dos roteadores. Os pacotes sempre entram na rede pelo IP local no roteador origem na fila de injeção e logo após são injetados na rede na fila *A*. Assim, a ordenação dos roteadores usada é crescente à direita, mostrada na Figura 13 (b).

A seguir, inicia-se o cálculo para saber para onde o roteador (3,1) deve enviar o pacote presente em sua fila *A*. O pacote *p* tem como destino o roteador (1,4), portanto *ok_nodes(p) = {(3,2),(3,0),(2,1)}*. Com os *ok_nodes(p)* calculados e o pacote *p* presente na fila *A*, verifica-se se algum dos *ok_nodes(p)* satisfaz a condição de *Right(nodo atual) < Right(nodo destino)*. Sabe-se, através da ordenação (Figura 13 (b)) o resultado da função *Right* de cada um dos roteadores, onde *Right(3,1)* = "9", *Right(3,2)* = "15", *Right(3,0)* = "3" e *Right(2,1)* = "8". Com base nesta informação deve-se verificar se algum dos roteadores presentes em *ok_nodes(p)* possui o resultado da função *Right* maior que o da função *Right* do roteador onde está armazenando o pacote naquele instante. No exemplo, verifica-se que o *Right(3,1) < Right(3,2)*, *Right(3,1) < Right(3,0)* e *Right(3,1) < Right(2,1)* e constata-se que somente *Right(3,2)* satisfaz essa condição. Apesar de somente um dos três roteadores satisfazer a condição, o Algoritmo CG1 determina que o seu conjunto de espera (*wait(p)*) consiste de todas as filas A que pertencem ao *ok_queue(p)* e ao *ok_nodes(p)*. Diante dessa situação considera-se (neste caso exemplo) que se pode rotear o pacote do roteador (3,1) para os roteadores (3,2), (3,0) ou (2,1).

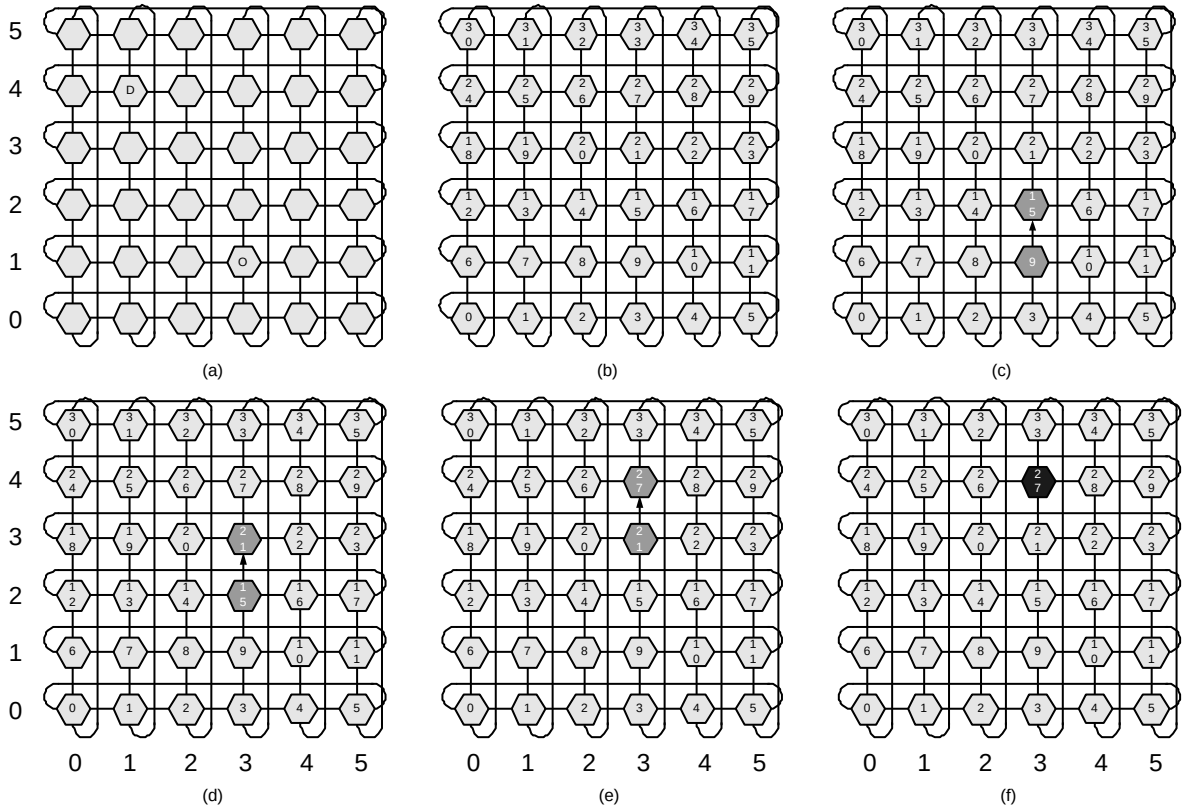


Figura 13 – Rede toro 6x6 com o caminho de um pacote da origem (3,4) até o destino (1,4), segundo o CG1 (Parte 01).

Por questões práticas, define-se aqui que diante de uma situação onde se possa rotear pacotes em várias direções, utilize-se um ordenamento da seguinte forma (Norte, Sul, Leste, Oeste). Com isso, o pacote presente no roteador (3,1) é roteado ao Norte para o roteador (3,2), conforme Figura 13 (c). Utilizando o processo já descrito, calcula-se os *ok_nodes(p)* para cada novo roteador onde o pacote for armazenado, verificando se o *Right* do roteador atual é menor que *Right* do roteador destino, gerando a partir deste resultado o novo *wait(p)*. Com base no conjunto de espera utiliza-se o ordenamento (Norte, Sul, Leste, Oeste) e roteia-se o pacote para o próximo roteador. Seguindo essas regras no exemplo, o pacote *p* é movido do roteador (3,2) para o (3,3) (Figura 13 (d)), e em seguida movido do (3,3) para o roteador (3,4) (Figura 13 (e)), utilizando sempre a fila *A* e o ordenamento "crescente à direita" em cada roteador.

Quando o pacote *p* chega no roteador (3,4) (Figura 13 (e)), o conjunto *ok_nodes(p)* possui somente o roteador (2,4). Sabe-se que *Right(2,4)* é menor que o *Right(3,4)*, fazendo com que o conjunto *wait(p)* seja vazio. Quando isto ocorre, é necessário que o pacote troque de fila. Desta forma o pacote *p* do exemplo, que está na fila *A* do roteador (3,4) move-se para a fila *B*, permanecendo no mesmo roteador (Figura 13 (f)). Isto é ilustrado pela troca de cor padrão

do roteador. A partir deste momento, pelo motivo do pacote pertencer à fila *B* deve-se usar o ordenamento dos roteadores para "crescente à esquerda", conforme ilustra a Figura 14 (a).

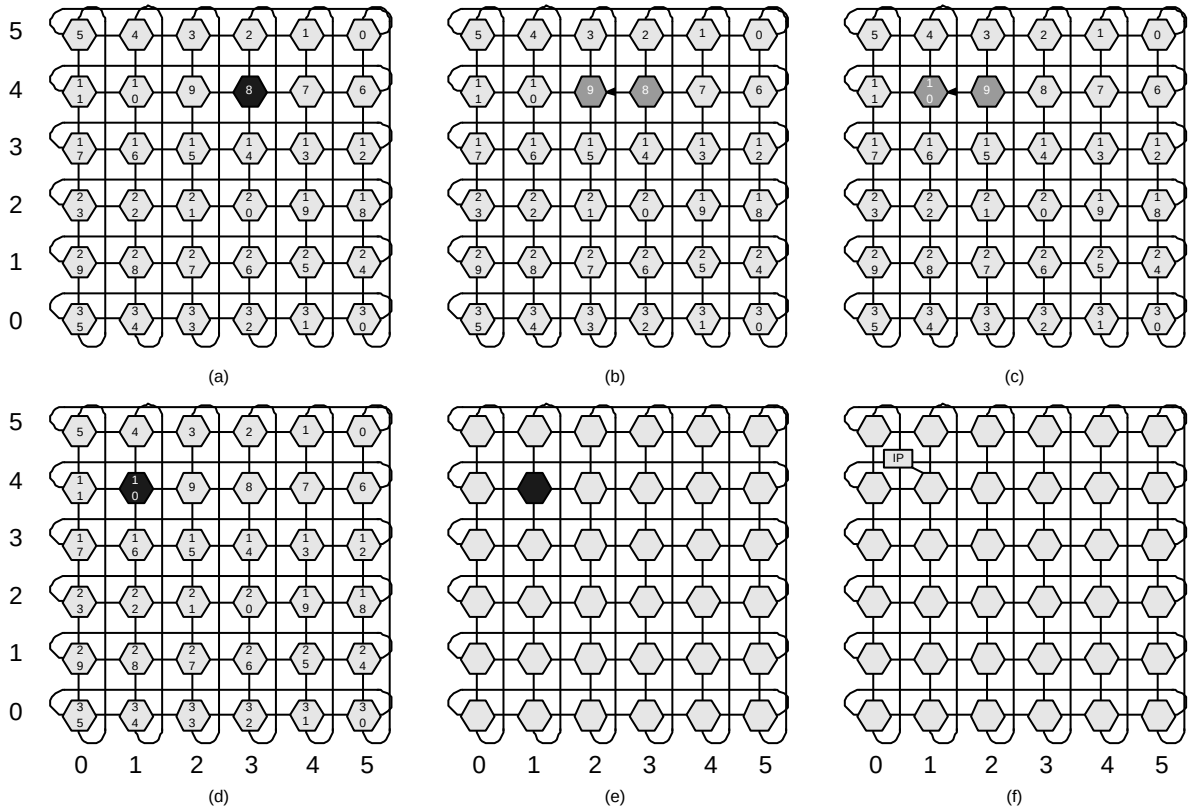


Figura 14 – Rede toro 6x6 com o caminho de um pacote da origem (3,4) até o destino (1,4), segundo o CG1 (Parte 02).

Utilizando-se essa nova ordenação, usa-se a função *Left* definida na Seção 4.1.3. No exemplo, aplica-se a função *Left* verificando se *Left(3,4)* < *Left(2,4)*. Como *Left(3,4)* = "8" e *Left(2,4)* = "9" a condição é satisfeita e o pacote *p* avança em direção ao seu destino (Figura 14 (b)). Desta mesma forma, o pacote *p* é movido do roteador (2,4) para o (1,4), conforme Figura 14 (c).

Quando o pacote *p* chega ao seu destino no roteador (1,4), ele ainda encontra-se na fila *B* (Figura 14 (d)). Desta forma deve-se necessariamente movê-lo para a fila *C* antes de entregá-lo ao IP local. Move-se então o pacote para a nova fila e com isso o ordenamento "crescente à esquerda" não é mais utilizado, deixando a rede sem ordenamento algum, o que ocasiona um possível deslocamento do pacote *p* em qualquer direção no sentido do seu destino (Figura 14 (e)). Como o pacote já se encontra em seu roteador destino e na fila *C*, ele pode sair da rede sendo inserido na fila de entrega de onde sai para o IP local deste roteador, conforme Figura 14 (f).

Em resumo, o pacote *p* será armazenado na fila de injeção no roteador (3,1), na fila *A* nos

roteadores (3,1), (3,2), (3,3) e (3,4), na fila *B* nos roteadores (3,4), (2,4) e (1,4), na fila *C* no roteador (1,4) e na fila de entrega no nodo (1,4), de onde é entregue, saindo da rede.

Para provar que o algoritmo é livre de *deadlock*, *livelock* e *starvartion*, Cypher e Gravano [17] enunciam e provam três Lemas, que são então combinados em um teorema fundamental.

Nos Lemas 1 e 2 prova-se que uma vez que um pacote seja colocado na fila de injeção, ele nunca permanecerá em uma única fila (seja ela qual for) para sempre. Através dessa prova, pode-se garantir que o Algoritmo CG1 é livre de *deadlock* e *starvation*. Se fosse possível o algoritmo de roteamento entrar em *deadlock* (fazendo com que ele não conseguisse mais progredir na rede) ou *starvation*, não se conseguiria provar, através dos Lemas 1 e 2, que o pacote se moveria de fila em fila até ser entregue.

No Lema 3, utilizando-se o que foi provado nos Lemas anteriores (1 e 2), provou-se que um pacote atinge a fila de entrega do seu nodo destino em um tempo finito. Para garantir que o algoritmo é livre de *livelock* para pacotes que entram na rede, usa-se o fato de que nenhum pacote pode permanecer para sempre na fila de injeção de acordo com Lemas 1 e 2, fazendo com que fila de injeção esteja eventualmente voltando a ficar vazia.

Com isso, o CG1 é mínimo, completamente adaptativo e livre de *deadlock*, *livelock* e *starvation*. Mais detalhes sobre a prova de liberdade de *deadlock*, *livelock* e *starvation* são encontrados em [17].

4.1.6 Conclusões

A partir da compreensão da teoria por trás do algoritmo CG1, procurou-se definir como utilizá-lo na prática. Quando propuseram o algoritmo, Cypher e Gravano não levaram em consideração algumas características que existem em redes intra-chip, como o paralelismo natural do *hardware*, o tamanho dos canais de comunicação, o tamanho reduzido necessário para as filas, entre outras características. Por esses motivos, procurou-se definir alguns pressupostos para facilitar a implementação do algoritmo em *NoCs*, enumerados abaixo:

1. Tamanhos das filas: devem ter no mínimo o tamanho de uma mensagem, já que é necessário armazenar uma mensagem por completo em cada roteador, usando o modo *virtual cut-through*;

2. Tamanho das mensagens: pode ser variável até o limite máximo da capacidade das filas, diferente do que acontece com as implementações atuais para malha desenvolvidas pelo grupo do Autor, que utilizam algoritmos de roteamento *wormhole*. Neste último caso,

pode-se ter filas de tamanho menor que a mensagem, já que a mesma não precisa, no pior caso, permanecer armazenada por completo em um único roteador;

3. Paralelismo: como Cypher e Gravano definiram de modo seqüencial seu algoritmo e módulos de *hardwares* trabalham em paralelo, deve-se definir um roteador que consiga operar de forma mais paralela possível com todas as filas e portas, conforme será visto no próximo Capítulo.

4. Controle de fluxo: em NoCs, controle de fluxo baseado em créditos é mais eficiente que *handshake* [15], pois reduz a latência mínima de transmissão de dados entre roteadores de 2 ciclos para 1 ciclo de relógio [15] quando se usa armazenamento na entrada do roteador. Contudo, o uso de filas centralizadas, com um árbitro por fila, conforme suposto pelo algoritmo CG1, muda esta relação. Como cada fila centralizada possui entrada compartilhada entre todas as portas do roteador, a complexidade de arbitragem de acesso para uso de controle de fluxo baseado em créditos possivelmente acarreta aumento do número de ciclos ou afeta a própria duração do ciclo de relógio. Assim, optou-se inicialmente por implementar uma política de controle de fluxo *handshake*.

5 Proposta de Arquitetura para o Roteador Mercury

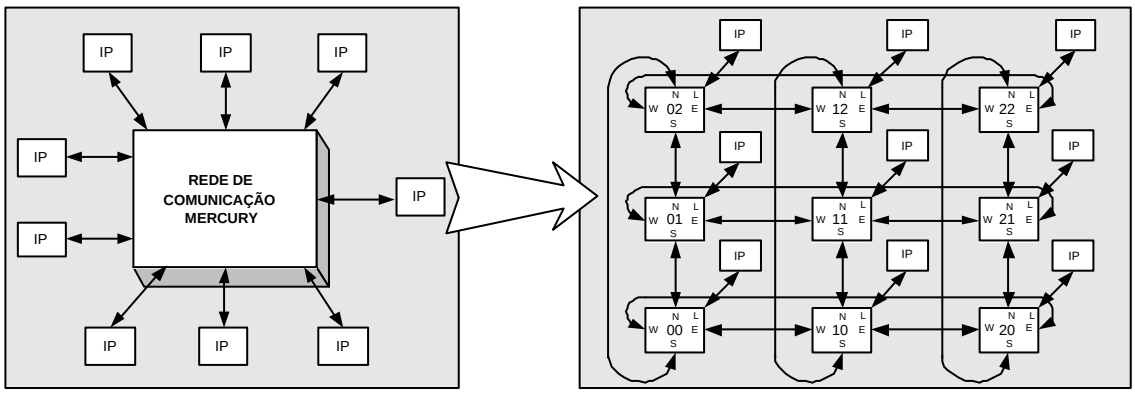
5.1 Introdução

Neste Capítulo propõe-se uma arquitetura de roteador para uma rede toro 2D utilizando o algoritmo CG1. O objetivo aqui é demonstrar a abordagem de modelagem e implementação do algoritmo, de forma a obter uma arquitetura que possa oferecer os requisitos necessários para a operação do CG1 de forma eficiente. O roteador é denominado Mercury.

Inicialmente dá-se uma visão global da rede e do roteador, assim como das interfaces de comunicação entre roteadores na rede e entre o roteador e seu IP local. Em seguida, discutem-se os árbitros de entrada e de saída, que possuem a função de realizar o caminhamento do pacote entre entradas e saídas do roteador. Logo após, é mostrada uma proposta de arquitetura para implementação das filas de armazenamento temporário, que têm como objetivo armazenar o dado a partir do momento em que esse entra no roteador até o momento que é transmitido ao IP local ou a outro roteador da rede. O roteador apresentado aqui assume redes toro 2D apenas. Com a eliminação de estruturas selecionadas, o mesmo roteador pode ser usado também em redes toro 1D (topologia anel).

5.2 Estrutura Geral da Rede Toro

A estrutura geral da rede de comunicação Mercury possui um número arbitrário de elementos de processamento conectados entre si através de uma rede toro, conforme ilustra a Figura 15 para o caso de uma rede de raiz 2,3 e dimensão 2. Essa rede tem como objetivo realizar a comunicação entre os diversos IPs de um SoC, através da execução do algoritmo CG1.



(a) (b)

Figura 15 – Em (a) visualiza-se a estrutura geral da rede, onde nas extremidades os IP *cores* estão presentes e ao centro, em forma de bloco, está a rede de comunicação toro. Em (b) aparece a estrutura interna da rede, onde se visualiza a interconexão de roteadores Mercury entre si, através das suas portas.

5.3 Arquitetura Geral do Roteador Mercury

Todos os roteadores presentes na rede de comunicação toro possuem a mesma estrutura, já que a rede é totalmente simétrica. A arquitetura geral do roteador proposto é ilustrada na Figura 16. Pode-se notar a presença das 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 à entrada 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 três árbitros de entrada, um por fila. A porta Local por sua vez, possui um tratamento distinto, pois segundo o algoritmo CG1, 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, o qual recebe as saídas das três filas, *A*, *B* e *C*. O controle desses multiplexadores é resultado do processo de operação de um árbitro de saída, que recebe como entrada requisições pendentes das três filas e tem como objetivo conectar cada uma delas a uma porta de saída de forma independente e exclusiva.

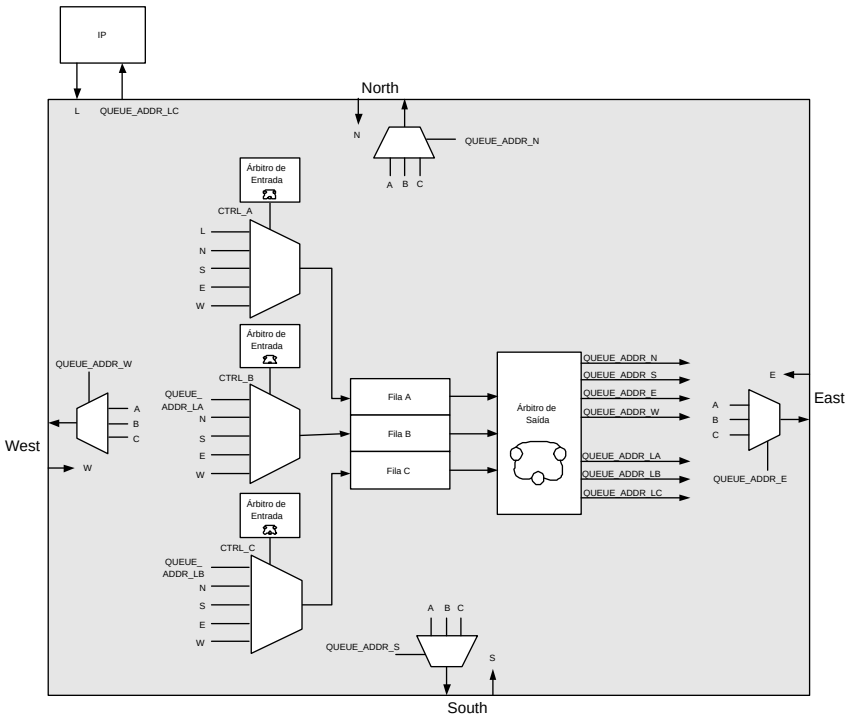


Figura 16 – Estrutura geral do roteador Mercury.

5.4 Interfaces

Para viabilizar a implementação do roteador, propõe-se a utilização de dois tipos de interfaces de comunicação. A primeira delas, ilustrada na Figura 17, é a interface entre alguma das portas do roteador N/S/E/W com alguma das portas N/S/E/W de um roteador vizinho. Nesta interface são definidos quatro sinais por comunicação entre os roteadores, que são *queue_addr*, *size*, *data* e *ack_nack*, correspondendo respectivamente ao endereço da fila de destino, o tamanho do pacote, o dado do pacote em si e o reconhecimento de aceitação ou não do dado pelo roteador receptor.

O sinal *queue_addr*, informa ao roteador destino em qual das filas *A*, *B* ou *C* o pacote que está sendo enviado deve ser armazenado ou dá uma indicação de que não há requisição de transferência de dado. Este sinal pode ser codificado em dois *bits*, representando dado disponível nas fila A, B ou C ou a inexistência de dado a transmitir.

A decisão de projeto de assim estruturar o sinal *queue_addr* vem do fato que o CG1 não permite troca de fila enquanto um pacote trafega de um nodo para outro na rede. O sinal *size* é enviado em separado devido à utilização do modo de chaveamento *virtual cut-through*, que determina que o roteador receptor somente deve receber o dado se tiver condições de armazená-

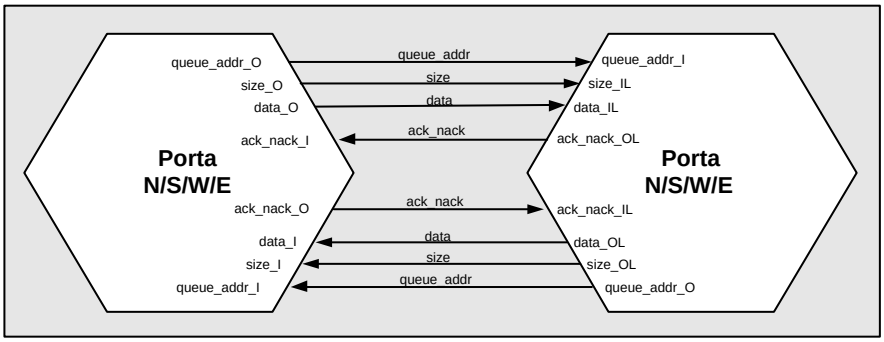


Figura 17 – Interface entre alguma das portas N/S/E/W de um roteador Mercury com alguma das portas N/S/E/W de outro roteador Mercury.

lo por inteiro em sua fila. O dado em si (que representa o *payload*), é enviado através do sinal *data*. O sinal *ack_nack* sinaliza, através de um código *ack* ao roteador transmissor que o receptor já recebeu o dado presente no sinal *data*. Caso não seja possível receber o dado, por motivos de espaço insuficiente na fila, o roteador receptor sinaliza isto através de um código *nack*. Este sinal necessita pelo menos 2 *bits* para codificação, pois deve transportar informações *ack*, *nack* e decisão ainda não formada.

A segunda interface de comunicação, ilustrada na Figura 18, representa a comunicação entre o IP Local e o roteador associado a ele. Essa interface contém a ligação da saída da fila *C*, com os sinais *data_av*, *size*, *data* e *ack_nack*, para o IP Local e a entrada na fila *A* a partir dos mesmos sinais. A principal diferença entre as interfaces de comunicação roteador-roteador e roteador-IP_local, é que o sinal *queue_addr* foi substituído pelo sinal *data_av* por não fazer sentido informar ao IP local em que fila o dado deve ser armazenado. Portanto, o sinal *data_av* tem somente a funcionalidade de sinalizar se existe ou não dado para ser recebido pelo IP local, podendo ser codificado em apenas 1 *bit*.

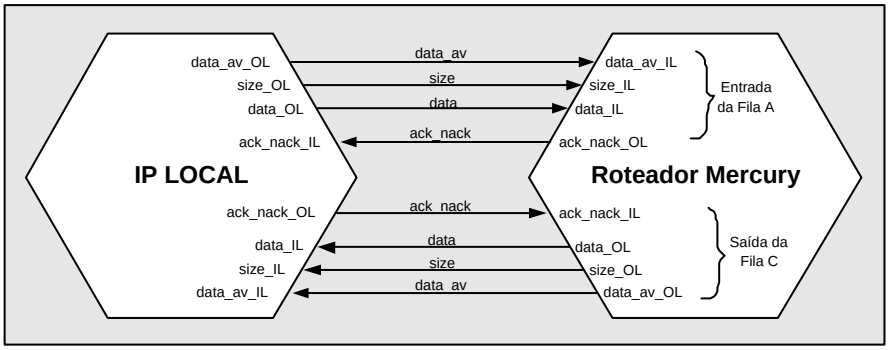


Figura 18 – Interface entre IP local e o roteador Mercury associado a ele.

5.5 Árbitros

Para que o roteador proposto possa implementar o algoritmo CG1, 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 entradas das três filas centrais *A*, *B* e *C* e as portas de saída 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 um árbitro de saída, que controla o acesso das três filas às cinco portas do roteador.

Os árbitros são máquinas de estados que operam de forma concorrente entre si. A funcionalidade dos quatro árbitros, associada ao restante da estrutura do roteador implementa o CG1. As seções 5.5.1 e 5.5.2 dão a estrutura básica de cada um dos tipos de árbitro.

5.5.1 Entrada

O árbitro de entrada, ilustrado na Figura 19, 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 definir a cada instante qual porta de entrada N/S/E/W/L irá acessar a sua fila. Existem três árbitros de entrada trabalhando concorrentemente, um para cada entrada de fila central *A*, *B* e *C*. Não existe restrição de acesso simultâneo a filas distintas, ou seja, pode-se ter três portas quaisquer (distintas), escrevendo em paralelo nas filas *A*, *B* e *C*. 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*.

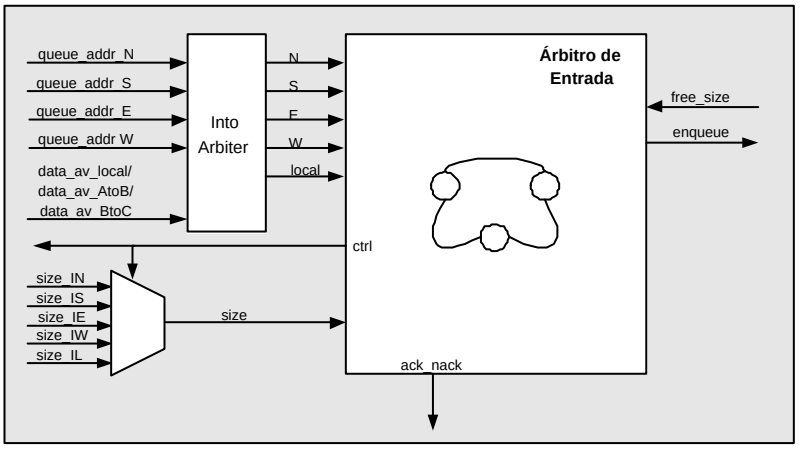


Figura 19 – Diagrama de blocos e sinais de interface do árbitro de entrada do roteador Mercury.

Cada árbitro de entrada recebe até cinco sinais de solicitação de acesso relativos às entradas do multiplexador respectivo. Quatro desses sinais são iguais para todos os árbitros de entrada, que são os sinais das portas de entrada não locais N/S/E/W. 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*.

Percebe-se que externamente cada árbitro de entrada possui um bloco a ele associado denominado *into-arbiter*. O sinal into-arbiter recebe sinais de solicitação de acesso à todas as filas e coloca em sua saída apenas a solicitação específica para a entrada da fila controlada pelo árbitro ao qual into-arbiter está associado. Assim, cada into-arbiter é distinto para cada árbitro de entrada, e os árbitros em si são idênticos.

O árbitro de entrada utiliza um algoritmo de arbitragem de prioridade rotativa, denominado *Round Robin*. Ele foi escolhido porque com ele consegue-se conceder prioridade de acesso de forma justa e simples para todas as entidades concorrentes.

5.5.2 Saída

O árbitro de saída, ilustrado na Figura 20, possui a função de controlar qual dentre as saídas das três filas *A*, *B* ou *C*, terá direito a acessar uma determinada porta do roteador.

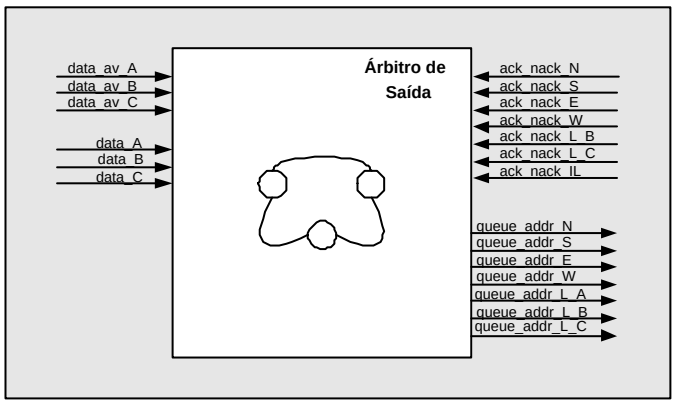


Figura 20 – Diagrama de blocos e sinais de interface do árbitro de saída do roteador Mercury.

O árbitro de saída recebe em paralelo os sinais enviados pelas três filas do roteador, e deve determinar qual delas terá acesso, a uma determinada porta 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.

Como podem existir várias requisições simultâneas das filas para a mesma porta, é necessário utilizar-se de um algoritmo de arbitragem, assim como ocorreu com os árbitros de entrada. Pelos mesmos motivos anteriores, utiliza-se o algoritmo de prioridade rotativa *Round Robin*.

Cálculo dos Caminhos Mínimos

O árbitro de saída também realiza a função de calcular os caminhos mínimos disponíveis entre um roteador origem e um roteador destino pertencentes a rede. Este cálculo é realizado baseado em funções matemáticas. Estas por sua vez, verificam qual a menor distância existente (em *hops*) entre os dois roteadores. A partir deste resultado verifica-se dentre os caminhos do roteador origem nas direções Norte, Sul, Leste e Oeste, qual delas é igual a distância mínima calculada anteriormente. Um pseudocódigo do cálculo para as direções pertencentes ao eixo X é demonstrado a seguir, sendo praticamente o mesmo para o eixo das coordenadas Y.

```
if(destino_x != pos_atual.x)  {
  if(((tam_noc.tam_x % 2) == 0) && (abs(destino_x - pos_atual.x)
       == (((tam_noc.tam_x - 1)/2)+1))){
           door_west = true;
           door_east = true;
  } else {
         if((destino_x - pos_atual.x) < 0) {
           if(abs(destino_x - pos_atual.x) <
             (tam_noc.tam_x - abs(destino_x - pos_atual.x)))
                door_west = true;
           else
                door_east = true;
          } else {
           if(abs(destino_x - pos_atual.x) <
             (tam_noc.tam_x - abs(destino_x - pos_atual.x)))
                door_east = true;
           else
                door_west = true;
           }
  }
}
```

No pseudocódigo descrito acima, verifica-se inicialmente se o roteador destino já está alinhado no eixo cartesiano X com o roteador em que se encontra o dado. Logo após verifica-se a NoC possui um número par de roteadores no eixo X juntamente com a verificação da distância entre o nodo atual e o nodo destino no eixo X. Se a NoC possuir um número par de roteadores no eixo analisado e a distância for igual por qualquer uma das direções, seta-se como caminhos mínimos as direções Leste e Oeste. Caso a NoC não possua um número par de roteadores no eixo X ou a distância por um dos lados (Leste ou Oeste) for menor em *hops*, seta-se somente a direção com menor número de *hops* entre a origem e o destino como caminho mínimo.

5.6 Armazenamento Temporário

O roteador Mercury possui três filas centrais denominadas *A*, *B* e *C*, para armazenamento temporário. Essas filas são circulares e utilizam o método *FIFO* para inserção e remoção de dados. A Figura 21 ilustra a estrutura geral das filas, onde se visualiza os sinais de interface da mesma. Os principais sinais são *data_in*, utilizado para sinalizar a inserção dos dados e *data*, utilizado para retirar um dado e enviá-lo para outra fila ou roteador. Nota-se também a presença dos ponteiros internos de leitura e escrita na fila. Quando a posição de leitura for igual à posição de escrita, considera-se que a fila está vazia, e quando a posição de escrita for a posição de leitura - 1, considera-se a fila cheia.

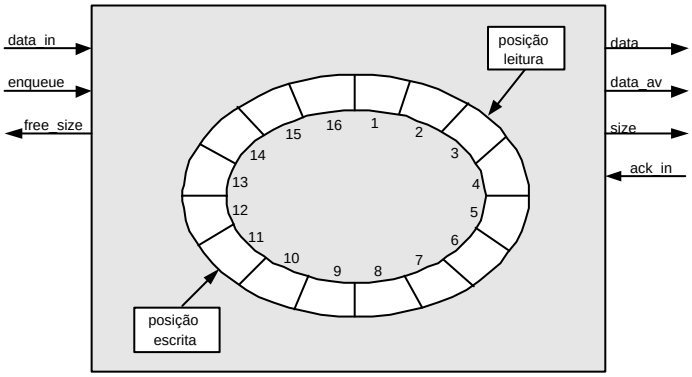


Figura 21 – Estrutura geral das filas do roteador Mercury, exemplificando para uma fila de 16 posições.

A organização interna das filas, ilustrada na Figura 22, detalha de 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*. Saindo da fila, nota-se a presença dos sinais que vão para o árbitro de saída. O sinal

data_av indica ao árbitro que um determinado dado está disponível para ser enviado, o sinal *data* representa o dado propriamente dito, e o sinal *size* representa o tamanho do pacote ao qual esse dado pertence. O roteador destino após receber a requisição de envio, confirma a recepção do dado a fila através do *ack_in*, indicando se foi ou não aceito o dado presente no sinal *data*.

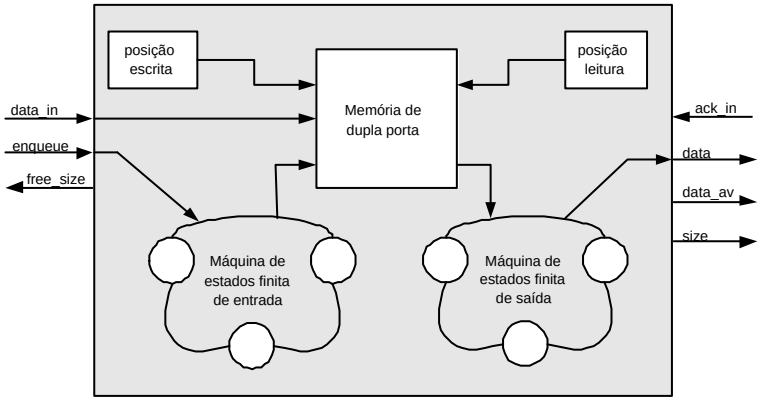


Figura 22 – Estrutura geral das fi las internamente e sua interface os outros módulos do roteador.

Existem duas máquinas de estados finitas associadas a cada fila. Elas 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*. 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 *ack_in* do roteador vizinho para o qual o dado está endereçado. A cada envio realizado, ela verifica novamente se a fila possui dados para serem enviados.

5.7 Conclusões

Nesse Capítulo apresentou-se a proposta do roteador Mercury, que dá suporte à execução do algoritmo de roteamento CG1 de Cypher e Gravano [17]. Definiu-se através de uma abordagem descendente, a arquitetura do roteador Mercury incluindo como são realizadas suas ligações na rede (Seção 5.2), passando pela estrutura interna (Seção 5.3), seus componentes de interface (Seção 5.4), os árbitros (Seção 5.5) e a estrutura geral do armazenamento temporário (Seção

5.6).

Os problemas de implementação desta arquitetura residem sobretudo no tratamento da funcionalidade das filas, devido ao alto grau permitido de concorrência de acesso. Nota-se que todas as portas podem requisitar acesso à mesma fila simultaneamente, pois todos os processos dentro do roteador ocorrem em paralelo. Além disto, pode-se citar outro problema existente em relação às máquinas de estados. Pelo motivo de existirem várias trabalhando em paralelo e concorrendo por recursos compartilhados, o processo de implementação destas torna-se importante, visando garantir a justiça no acesso às filas, portas etc. No Capítulo seguinte será abordada a implementação dos módulos definidos aqui.

6 Modelagem da NoC Mercury

No Capítulo anterior, apresentou-se uma proposta de arquitetura do roteador Mercury, que dá suporte à implementação de redes toro com raiz mista e duas dimensões. Este roteador permite implementar NoCs que empregam o algoritmo de roteamento CG1 proposto em [17] por Cypher e Gravano e apresentado no Capítulo 4. No presente Capítulo, aborda-se na Seção 6.1 a modelagem em nível de abstração de transação (em inglês, *transaction level*, ou TL) do roteador e da rede Mercury utilizando-se da linguagem SystemC [1, 11, 25, 62]. A seguir, mostra-se a modelagem concreta RTL do roteador e da rede Mercury na linguagem de descrição de *hardware* VHDL [4].

Em ambas as Seções, 6.1 e 6.2 usa-se uma abordagem similar, mostrando a modelagem do roteador seguida da modelagem da rede, seguido de uma descrição razoavelmente extensa do processo de validação dos modelos propostos.

6.1 Modelagem TL

O objetivo da modelagem TL [12] é prover uma versão executável da especificação de um roteador para rede toro, e de uma rede intra-chip para topologia toro. Esta especificação executável serve mais tarde para guiar o processo de modelagem concreta, bem como a prototipação *hardware* da rede Mercury facilitando a detecção de possíveis erros de projeto durante o processo de implementação.

Nessa Seção será apresentada a modelagem SystemC em módulos, canais e interfaces. A abordagem será ascendente, iniciando com a modelagem do roteador, seguida da modelagem da rede e sua validação por simulação.

O nível de abstração de transação (em inglês, *transaction level* ou TL) vem se constituindo como uma das formas preferenciais para validar projetos de sistemas complexos em alto nível [12, 33, 49], por permitir rapidamente capturar a funcionalidade de um projeto de grande porte. O nível TL pressupõe descrições em temporização detalhada, onde os módulos principais do sistema interagem através de canais abstratos. Os canais podem prover diversos serviços, além do serviço básico de transferência de informação entre módulos do sistema, incluindo adaptação

de formatos de dados, conversão de protocolo de comunicação, encapsulamento da informação, entre outros. Este nível é muito usado para obter reusabilidade em ambientes de verificação. Descrições mais detalhadas de propostas do nível TL podem ser encontradas em [12, 33].

A linguagem SystemC possui recursos explícitos para a modelagem TL. Módulos são implementados usando a construção SC_MODULE e canais usando a construção SC_CHANNEL. Uma forma de prover a conexão abstrata entre módulos se comunicando através de canais ocorre através do uso de conceitos de interfaces, construções parametrizáveis e reutilizáveis providas pela linguagem para modelar cada um dos pontos de comunicação entre um módulo e um canal. A descrição TL do roteador Mercury foi realizada usando estes conceitos e recursos da linguagem. Atribui-se aqui a esta modelagem abstrata o rápido êxito do processo de validação funcional, bem como do posterior processo de prototipação da NoC Mercury. Apenas para dar idéia do nível de esforço necessário para gerar descrições TL e RTL apresenta-se na Tabela 6 uma comparação de implementações de uma NoC Mercury 3x3 nos dois níveis de abstração.

Tabela 6 – Comparação de implementações de uma NoC Mercury 3x3 nos dois níveis de abstração.

Grandezas / Abstração	TL	RTL
Nº de arquivos	36	14
Total de linhas da descrição	2619	2843
Tempo estimado de implementação	5 meses	8 meses

As Seções a seguir descrevem em algum detalhe a modelagem TL da NoC Mercury, bem como do processo empregado em sua validação, baseado no uso da ferramenta Modelsim, do compilador **gcc** associado a esta e da biblioteca SystemC disponibilizada pela Open SystemC Initiative (OSCI) [62].

6.1.1 Modelagem do Roteador Mercury

O roteador é um conjunto de módulos que tem por objetivo receber e enviar pacotes segundo um algoritmo de roteamento. Os módulos que compõem o roteador realizam a tarefa de armazenar os pacotes na fila à qual ele é destinado e enviá-los pela porta de destino correta. A estrutura geral da modelagem SystemC do roteador Mercury pode ser observada na Figura 23. Os módulos SystemC do roteador na implementação TL são: (*i*) *queue*, fila para armazenamento temporário dos pacotes; (*ii*) *arbiterIn*, árbitro para selecionar a porta que será atendida em cada momento para cada fila, seguindo a prioridade *Round-Robin*; (*iii*) *intoArbiter*, árbitro para selecionar e direcionar pacotes para a fila a qual eles pertencem; (*iv*) *arbiterOut*, árbitro

para definir a porta de saída através da qual o pacote será enviado para chegar em seu destino por um caminho mínimo, baseando-se no algoritmo CG1.

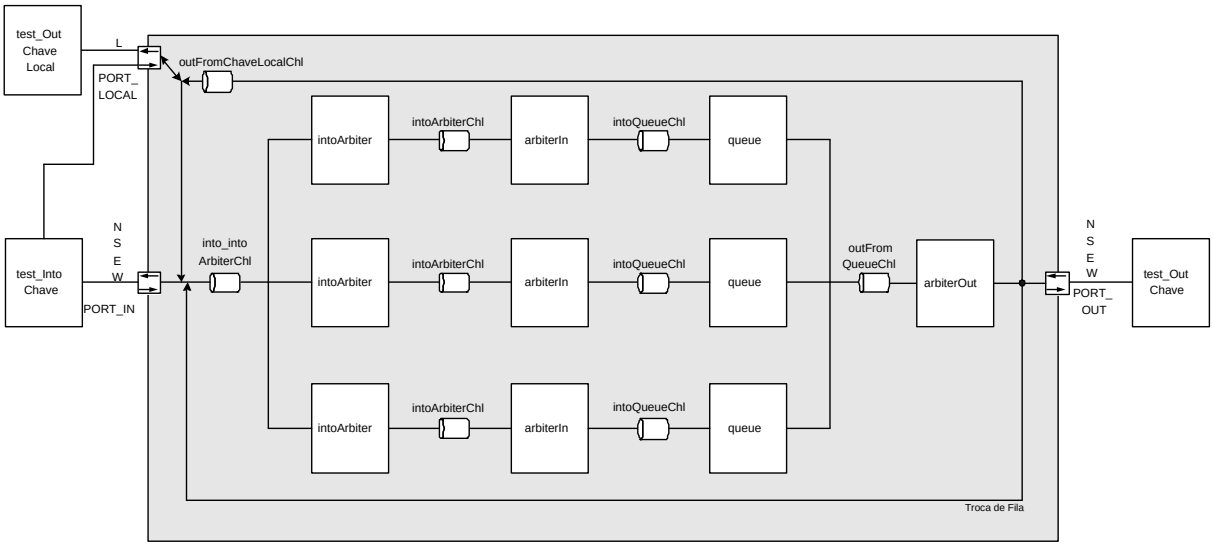


Figura 23 – Visão do roteador completo da NoC Mercury TL.

Para a construção de um roteador são utilizados três módulos *intoArbiter*, três módulos *arbiterIn*, três módulos *queue* e um módulo *arbiterOut*. Para realizar a comunicação entre estes módulos foi utilizado um conjunto de canais: *into_intoArbiterChl*, é o canal de entrada do roteador; *intoArbiterChl*, é o canal de comunicação entre *intoArbiter* e *arbiterIn*; *intoQueueChl*, é o canal de comunicação entre *arbiterIn* e *queue*; *outFromQueueChl*, é o canal de comunicação entre *queue* e *arbiterOut*; e *outFromChaveLocalChl*, é o canal de saída para a porta Local que tem como entrada a saída do módulo *arbiterOut*.

Cada um dos módulos pertencentes ao roteador será descrito a seguir em maior detalhe:

Canal *into_intoArbiterChl*

O canal *into_intoArbiterChl* é o canal de entrada do roteador, ele contém funções necessárias para o recebimento dos pacotes originários das portas Norte, Sul, Leste e Oeste ou da porta Local.

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 sendo recebido através do canal *into_intoArbiterChl* tem uma fila de destino única: se sua origem é um roteador

vizinho, a fila destino será a mesma em que ele estava armazenado no roteador anterior. Se sua origem é a porta Local, a fila destino será sempre a Fila A seguindo o definido no algoritmo CG1.

O módulo *intoArbiter* seleciona os pacotes para uma única fila, de forma que se uma instância particular do módulo estiver conectada à Fila A, este deve ser configurado na sua criação, (via seu construtor), para selecionar apenas os pacotes destinados para a Fila A. Caso a fila destino do pacote não seja a fila na qual este módulo está configurado nada será feito, pois outro *intoArbiter* que está configurado para esta outra fila vai se encarregar de consumir este pacote.

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

Canal *intoArbiterChl*

O canal *intoArbiterChl* está presente entre os módulos *intoArbiter* e *arbiterIn*. Ele é único para cada fila e sua função é prover a transferência dos dados entre os módulos que ele conecta.

Módulo *arbiterIn*

O módulo *arbiterIn* é responsável por ler os pacotes, um por vez, do canal *intoArbiterChl* e enviá-los para o módulo *queue* através do canal *intoQueueChl* com prioridade *Round-Robin*. Este módulo modela o controle de prioridade necessário para que o mesmo atenda todas as portas igualmente, de modo a não permitir que ocorra postergação indefinida de uma porta. Cada vez que a função é chamada, verificam-se as portas na seqüência circular: Norte, Sul, Leste, Oeste e Local. Isto é feito para determinar se existe algum pacote destinado à fila que o módulo opera. Se houver, o pacote será lido, e o endereço dessa porta será armazenado. Na próxima vez que a função for chamada, as portas serão verificadas novamente na mesma seqüência citada, mas começando pela porta posterior à porta em que o pacote foi lido na última vez que a função foi chamada. Dessa forma, nenhuma porta terá mais prioridade do que as outras. Este módulo é único para cada fila, sendo que todos os pacotes que chegam nele, são oriundos do módulo *intoArbiter*, que já os filtrou de acordo com sua fila de destino. No momento do envio de um pacote, o módulo fica aguardando seu armazenamento na fila, para então ler outro pacote, se houver.

Canal *intoQueueChl*

O canal *intoQueueChl* está presente entre os módulos *arbiterIn* e *queue*, sendo único para cada fila. Este canal tem como objetivo realizar a transferência de dado entre os módulos que ele conecta.

Módulo *queue*

O módulo *queue* implementa a fila do roteador e contém funções para controlar o armazenamento de pacotes. A fila implementada é circular e de porta dupla. Nela, os pacotes que chegam são armazenados na primeira posição livre e os pacotes que saem são retirados da primeira posição ocupada. Estas operações podem ser realizadas de forma concorrente. Dois processos implementam a funcionalidade da fila: o processo de entrada e o processo de saída. O primeiro é 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. O processo saída tem como função enviar o primeiro pacote da fila, esperar o sinal que indica que ele foi armazenado no roteador receptor, e calcular a posição em que se encontra o próximo pacote a ser enviado. Uma posição da fila só é acessada depois de verificado o qualificador *fila_vazia*, que informa se existe algum elemento na fila.

Um elemento de exclusão mútua (**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 os processos que querem escrever na fila verificarão se existe espaço livre para isso antes de tentar escrever. Depois de armazenado o pacote na fila, a porta de entrada por onde o pacote chegou é liberada para enviar novos pacotes e a fila pode recomeçar a enviar. Quando a fila contém dados, o processo de saída envia o primeiro pacote da fila através do 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, para então enviar outro pacote.

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 envio de pacotes, sendo que 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 lhe pertence neste canal. Quando um pacote é enviado para uma das posições do vetor, 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 comunica o módulo *queue* que o dado enviado já foi armazenado no seu destino. Com isso a posição ocupada no módulo *queue* já pode ser sobre escrita e a posição no vetor para envio de dados no canal *outFromQueueChl* fica apta para receber um novo pacote. Existe apenas um canal *outFromQueueChl* para as três filas, pois neste ponto todas filas devem concorrer pelas mesmas cinco portas de saída do roteador. Quem seleciona qual porta irá atender qual fila é um único módulo (*arbiterOut*), de forma a evitar que um mesmo recurso seja disponibilizado para mais de uma fila.

Módulo *arbiterOut*

O módulo *arbiterOut* é o responsável por enviar os pacotes seja para os roteadores vizinhos, seja para outra fila no mesmo roteador ou para o IP local. Neste módulo existem três processos de envio, um para cada fila. Cada um envia pacotes distintos, 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 executado/chamado e retornará uma ou mais portas para as quais o pacote poderá ser enviado. Verifica-se, para cada porta seqüencialmente, se é possível enviar um pacote por ela. Em caso afirmativo, o pacote é enviado. Caso contrário, ocorre um processo de verificação da próxima porta retornada pelo algoritmo, de forma circular, até que seja encontrada uma 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 [17], 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 deve ir para a porta Local, ele irá antes para a Fila B, depois para a Fila C, para somente daí ser enviado à porta local, saindo da rede para o IP local.

Canal *outFromChaveLocalChl*

É o canal de comunicação entre a saída do roteador (módulo *arbiterOut*) e a entrada no IP local do mesmo. Somente um pacote da Fila C poderá ser enviado neste canal de cada vez. No momento em que o pacote está sendo enviado, o canal deve aguardar até que o IP local o receba, para que então um novo pacote possa ser enviado por ele.

6.1.2 Modelagem da Rede Mercury

A implementação da comunicação entre os roteadores Mercury, bem como a disposição dos roteadores para implementar uma topologia do tipo toro 2D serão descritos nessa Seção.

A comunicação entre os roteadores 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*:** utilizada pelo módulo *arbiterOut* do roteador transmissor. 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 o 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 CG1. A função *write_local* simplesmente utiliza a função *write_queue_addr* descrita anteriormente, 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 possui o 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 foram implementadas para a troca de filas em um mesmo roteador. Elas são importantes para o funcionamento do algoritmo, já que o mesmo utiliza filas centralizadas e estão localizadas na interface de entrada do canal *into_intoArbiterChl*.

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

- ***write_queue_change***, utilizada pelo módulo *arbiterOut* do próprio roteador para a 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 confirmação do armazenamento na 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.

6.1.3 Validação NoC Mercury TL

Após concluída a modelagem abstrata da rede Mercury em nível de abstração de transação, faz-se necessário validar a mesma. Esta Seção descreve as estruturas utilizadas para validar a NoC TL em SystemC. A validação com a ferramenta Modelsim foi realizada inicialmente de forma *ad hoc*, pela análise de dados impressos na tela e logo após diante da quantidade de dados gerados, desenvolveu-se uma ferramenta para auxiliar na análise que será descrita no Capítulo 7. Vários testes foram realizados para validação da implementação TL durante a execução do trabalho. Nesta Seção serão citados os mais importantes, iniciando com testes das estruturas internas e seguindo em direção a testes da rede completa.

Testbench módulo *queue*

Este *testbench* foi utilizado para validar o funcionamento do módulo *queue* e dos canais de entrada (*intoQueueChl*) e saída (canal *outFromQueueChl*) deste módulo, cuja interligação é ilustrada na Figura 24. O gerador de estímulos é o módulo *test_intoQueue*, que conecta-se ao canal de entrada do módulo. Este contém um processo que insere um determinado número de pacotes na fila. A partir do momento em que o pacote é enviado ao canal e enquanto ele não chega no módulo *queue*, o processo de envio fica bloqueado. No momento em que este é armazenado no módulo, outro pacote pode ser enviado. O capturador de saídas é o módulo *test_outQueue*, que conecta-se ao canal de saída e através de um processo que lê os pacotes do mesmo. Caso não haja pacotes disponíveis, o canal bloqueia o processo até que um pacote seja inserido.

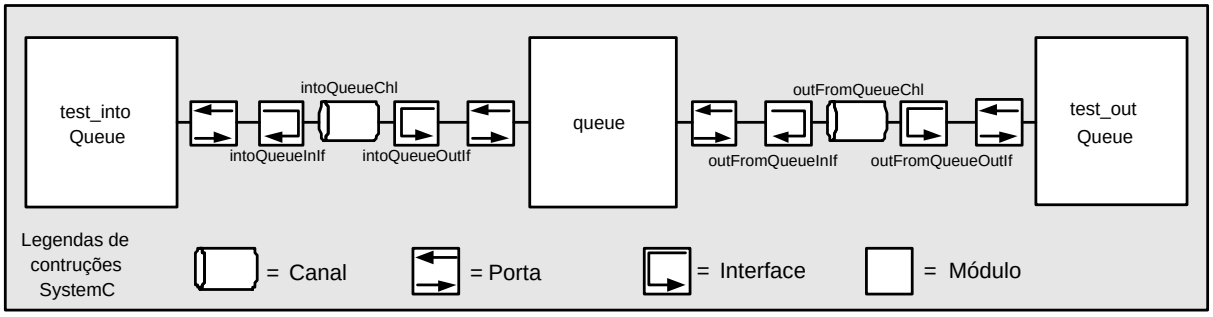


Figura 24 – Estrutura de validação do módulo *queue* e seus canais de acesso para entrada e saída dos dados do módulo.

A partir da estrutura de *testbench* criada para validar o módulo *queue*, o resultando obtido quando simulado na ferramenta Modelsim é mostrado na Figura 25. Cada linha resultante desta simulação foi numerada à esquerda da ampliação do código, para facilitar o acompanhamento dos eventos relatados. Este *testbench* foi realizado com poucos dados (somente três) e tamanho de fila reduzido (filas de tamanho útil 2) para facilitar a explicação neste trabalho. Na prática foram realizados vários testes com filas de tamanho arbitrário assim como número arbitrário de dados inseridos.

Na Figura 25 onde o resultado da simulação é mostrado, *queue0* representa Fila A, e a numeração *[12]* representa o roteador ao qual a fila está associada. Na linha 1, inicia-se o *testbench* a partir da inicialização do processo de entrada de dados. Na linha 2, é ilustrado o armazenamento do dado "0" na Fila A do roteador 1x2, tendo como destino o roteador 0x0. Na terceira linha é relatado o conteúdo atual da Fila A, onde está armazenado o valor "0" referente ao dado inserido na linha 2. Na linha 4, inicia-se o processo de retirada dos dados da fila. Na linha 5, o dado "0" é retirado da Fila A, e na próxima linha é relatado novamente o conteúdo da fila, que após a retirada do dado inserido pela linha 2, está vazia. Deve-se notar que os processos de inserção e retirada de dados das filas são totalmente independentes e operam em paralelo. Ambos são processos bloqueantes, e a operação da fila também se dá em paralelo com as operações de inserção e retirada.

Nas linhas 8 e 10 são inseridos respectivamente os dados "1" e "2" na Fila A. A fila testada possui tamanho dois e neste instante ela está armazenando o número máximo de dados. Com isso, quando se tenta inserir o dado "3" (linha 12) não se consegue, deixando então o dado à espera no canal *intoQueueChl*, até que a fila libere espaço. Na linha 13, o dado "1" é retirado da fila, deixando a mesma com somente um elemento (linha 14). Com espaço disponível, a fila armazena o dado "3" que estava bloqueado no canal a espera de espaço. A partir deste momento os dados são retirados da fila pelo módulo *test_OutQueue*, nas linhas 17 e 19 respectivamente,

Figura 25 – Resultado da simulação do *testbench* do módulo *queue* na ferramenta Modelsim.

deixando a fila vazia a espera de dados (linha 21).

Testbench Parcial do Roteador Mercury

Este *testbench* é utilizado para validar o funcionamento dos módulos *intoArbiter*, *arbiterIn* e *queue* trabalhando em conjunto. Ele possui um gerador de estímulos denominado módulo *test_intoChave* que 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* possui três processos paralelos para retirar pacotes do canal de saída das três filas de forma concorrente (canal *outFromQueueChl*). A estrutura do *testbench* é ilustrada na Figura 26.

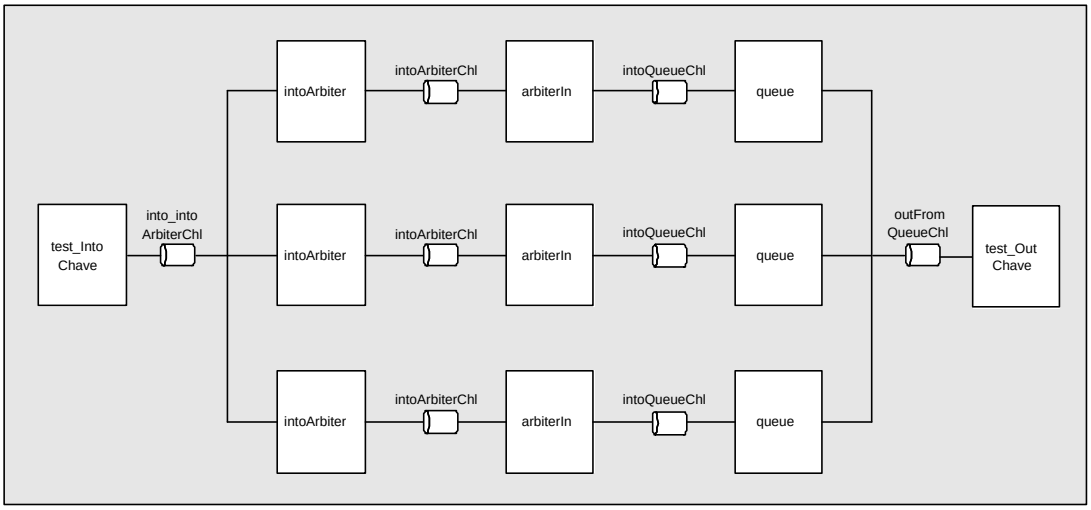


Figura 26 – Estrutura de validação conjunta dos módulos *intoArbiter*, *arbiterIn*, *queue* e dos canais *into_intoArbiterChl*, *intoArbiterChl*, *intoQueueChl*, *outFromQueueChl*

O resultado obtido com a simulação do *testbench* criado na ferramenta Modelsim pode ser visualizado na Figura 27. Este *testbench*, assim como o descrito para o módulo *queue*, foi realizado com o mínimo de dados possíveis, a fim de demonstrar somente a idéia utilizada para validação dos módulos. Neste caso específico, foram definidas filas de profundidade de dois dados e envio de somente um dado por porta de entrada, ou seja, foram enviados cinco dados, sendo cada um originado de uma das portas do roteador, Norte, Sul, Leste, Oeste e Local. Na prática, para validação desses módulos foram feitos testes com filas de tamanho arbitrário assim como o número arbitrário de dados inseridos nestas.

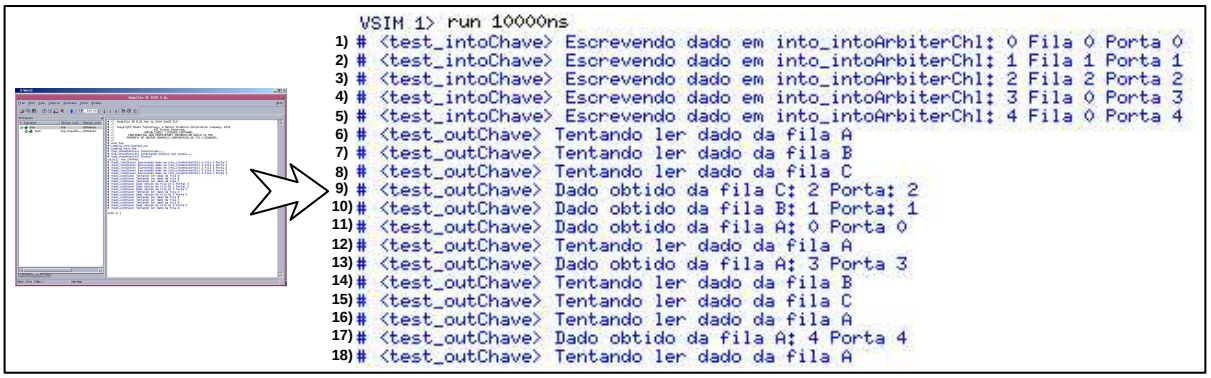


Figura 27 – Resultado da simulação do roteador parcial na ferramenta Modelsim.

Na Figura 27 onde o resultado da simulação é mostrado, nota-se a existência de poucas informações, visto que foi configurado no *testbench* somente a exibição das informações relativas às entradas dos dados no canal *into_intoArbiterChl* e a saída pelo canal *outFromQueueChl*. Nas linhas 1 a 5, escrevem-se os dados numerados de "0" a "4", oriundos das portas N/S/E/W/L nas Filas A, B e C de forma arbitrária. A sintaxe dessas linhas informa primeiramente a numeração dos dados de "0" a "4", logo após a fila para onde cada dado será enviado, sendo a Fila A representando pelo numeral "0" a Fila B pelo "1" e a Fila C pelo "2". Por fim, a seqüência numérica que aparece indica a porta por onde o dado entrou no roteador, sendo a representação 0/1/2/3/4 das portas N/S/E/W/L respectivamente.

As linhas numeradas de 6 a 8 mostram o módulo de saída tentando retirar dados do canal *outFromQueueChl*, e em seguida nas linhas 9 a 11 é mostrado que realmente os dados foram lidos do canal. Este processo se repete mais duas vezes para a Fila A, já que esta possuía 3 dados armazenados, totalizando assim a retirada de cinco dados enviados. Pode-se perceber através deste *testbench* que o funcionamento de *intoArbiter* está correto, já que os dados destinados para a Fila A foram enviados para ela, assim como os das Fila B e C, sem que existisse erros na colocação dos dados na fila apropriada. Por exemplo, o dado "2" pertencia à Fila C no momento que foi injetado no *testbench* pelo módulo *test_intoChave* conforme a linha 3, e este

mesmo dado foi recebido e armazenado na fila correta, já que o módulo *test_outChave* que retira os dados da fila, mostra na linha 9 que retirou o dado "2" da Fila C. Este processo verificou-se para todos os dados injetados e retirados pelo *testbench*, validando assim a estrutura parcial do roteador.

A partir do funcionamento desta estrutura, pode-se então implementar o *testbench* para o roteador completo, para avaliar o correto funcionamento do algoritmo de roteamento presente no módulo *arbiterOut*, o mais complexo do roteador.

Testbench Completo do Roteador Mercury

Este *testbench* é utilizado para validar o roteador completo, em especial o módulo *arbiterOut*, em conjunto com o algoritmo de roteamento funcionando de forma isolada. Neste teste, verifica-se o comportamento do módulo *arbiterOut*, ou seja, se ele está enviando pacotes pela porta correta baseado no algoritmo CG1. Para esta validação, utiliza-se uma roteador completo.

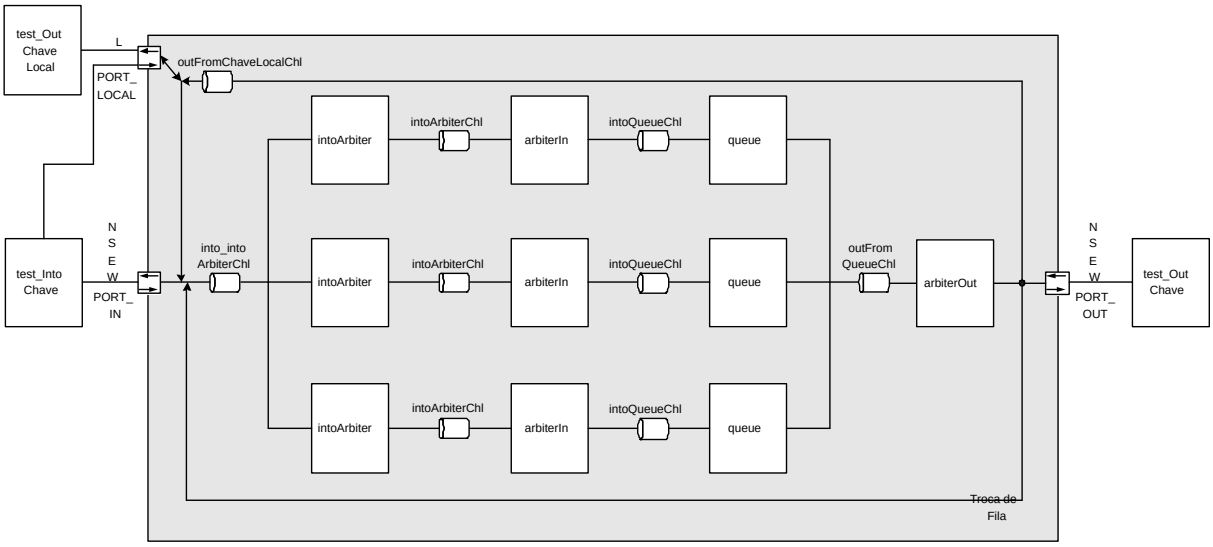


Figura 28 – *Testbench* do roteador completo ilustrando o diagrama de blocos do roteador. Verifi ca-se o funcionamento de todos módulos e canais, dando maior ênfase ao módulo que possui o algoritmo de roteamento denominado *arbiterOut*. A partir da estrutura deste *testbench*, pode-se verifi car o correto funcionamento do roteamento dos pacotes pelas cinco portas do roteador, bem como o roteamento interno para as Filas B e C.

No arquivo *top-level* do *testbench* ilustrado na Figura 28, é possível informar a posição relativa do roteador a ser validado e o tamanho da rede nas coordenadas X e Y na qual este está virtualmente inserido. O gerador de estímulos é o módulo *test_intoChave*, que coloca em cada porta de entrada uma quantidade de pacotes definida, cada pacote com um valor distinto de dado (para viabilizar o acompanhamento do trajeto dos pacotes pelo roteador e sua saída do

mesmo), além de escolher o roteador destino e a fila na qual o pacote irá pertencer, de forma aleatória.

Cada uma da cinco portas recebe um pacote de forma concorrente, devido aos cinco processos paralelos que executam no módulo *test_IntoChave*. Os capturadores de saída são o módulo *test_OutChave* e o módulo *test_OutChaveLocal*. O *test_OutChave* é um módulo do tipo *intoArbiter* com funções diferenciadas, a fim de atender os objetivos do *testbench*. Ele recebe os pacotes que saem do módulo *arbiterOut*, simulando uma recepção em um roteador vizinho, imprimindo na tela o conteúdo 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 deste.

Um exemplo de simulação com a ferramenta Modelsim utilizando o *testbench* apresentado acima gerou o resultado visualizado na Figura 29. Foram definidas filas de profundidade 2 e envio de somente um dado por porta de entrada. Assim, foram enviados cinco dados, cada um proveniente de uma das portas: Norte, Sul, Leste, Oeste e Local. O *testbench* foi realizado em um roteador presente em uma rede de dimensões 3x3, sendo o roteador avaliado o da posição 1x1. Vários outros testes foram realizados com injeção de um número arbitrário de pacotes com diversos tamanhos de fila para efeito de validação, gerando resultados corretos do ponto de vista da funcionalidade. As informações apresentadas na Figura 29 são relativas aos módulos de inserção de dados (*test_IntoChave*) e de remoção dos dados (*test_OutChave*). Nas linhas 1 a 5 mostra-se informações referentes à inserção de dados no roteador. Por exemplo, na linha 1 é inserido o dado "0", na Fila A, pela porta Norte, com destino ao roteador 1x0. Utilizando esta mesma sintaxe seguem as linhas de número 2 a 5.

As linhas de 6 a 10 ilustram a remoção dos dados das filas pelo roteador. Por exemplo, na linha 9 mostra-se o procedimento adotado sobre o dado "0". Com base nas informações de entrada do dado "0" no roteador (linha 1), o módulo *arbiterOut* computa o roteamento do pacote para o seu destino com base no algoritmo CG1. Inicialmente, ele avalia que não existe nenhum caminho mínimo entre o roteador atual do pacote 1x1 e o roteador destino 1x0 quando o pacote está na Fila A. Desta forma, o roteador envia este pacote para a sua Fila B, conforme visualiza-se na linha 9, onde é informado que o pacote "0", está agora na Fila B. Da mesma forma, o dado "1" que entrou no roteador pela porta Sul na Fila B tendo com destino o roteador 0x1, é roteado pela porta Oeste do roteador que está sendo testado. Desta forma, na linha 8 é mostrado que o pacote foi recebido pelo roteador 0x1 na porta Leste, Fila B. Este mesmo procedimento pode ser verificado para os outros 3 dados, inseridos nas linhas 3 a 5 e retirados do roteador nas linhas 8 a 10.

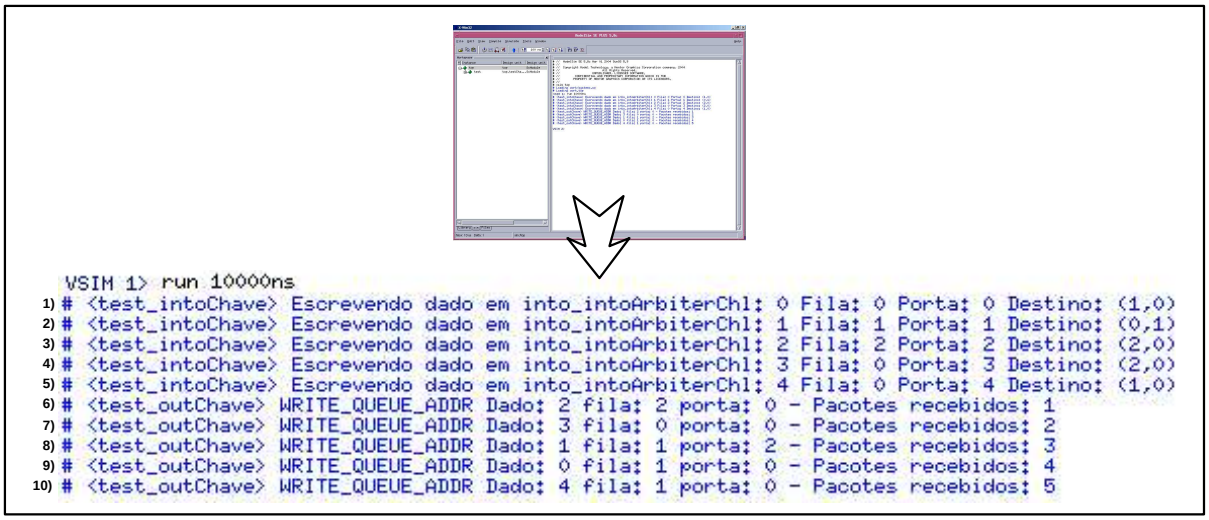


Figura 29 – Resultado da simulação do roteador completo na ferramenta Modelsim.

Através deste processo de teste, verificou-se que o roteador como um todo está em princípio operando de forma correta, já que se pôde validar os seguintes módulos:

- As cinco portas estão aptas a receber e enviar dados;
- O módulo *intoArbiter* seleciona corretamente os pacotes para cada uma das filas;
- O módulo *arbiterIn* recebe direito de acesso à fila;
- O módulo *queue* armazena e envia o dado adiante de forma correta;
- O módulo *arbiterOut*, com base nas informações de controle presentes no pacote, roteia o último de forma correta, aplicando o algoritmo de roteamento;

Sabendo que o funcionamento da estrutura básica do roteador está correto, pode-se então realizar o *testbench* na rede como um todo. A seguir serão mostrados alguns dos vários testes funcionais realizados sobre instâncias completas da rede Mercury TL.

Testbench da NoC Mercury TL

Este *testbench* utiliza vários roteadores conectados, representando uma NoC toro 2D. Vários testes foram realizados nesse ambiente para validar o fluxo correto de dados, o algoritmo de roteamento de forma mais ampla e o funcionamento do projeto em diferentes situações. Como o *testbench* da rede gera um grande volume de pacotes, foi criado um protocolo de comunicação

e uma ferramenta para leitura e formatação dos dados resultantes da simulação Modelsim. A ferramenta e o protocolo desenvolvidos neste trabalho serão apresentados no Capítulo 7.

A rede utilizada durante todos os testes que serão apresentados a seguir é uma NoC Mercury 4x3 toro bidirecional. Nas Figuras 30, 31, 32, 33, 34, 35 é mostrado a leitura e interpretação dos arquivos gerados na simulação pela ferramenta Modelsim, utilizando-se da ferramenta Analisador Júpiter. Essas Figuras estão divididas em quatro partes, sendo mostrado na ampliação em *(a)* o caminhamento de algum(s) pacote(s) típico(s) da simulação, desde a sua origem até o seu destino detalhando o caminhamento do pacote. Em *(b)*, verifica-se se os pacotes chegaram ou não ao seu destino, sendo ilustrado cada pacote em separado para facilitar a localização de possíveis erros, caso algum dos pacotes não consiga chegar ao seu destino. Na ampliação em *(b)* ilustra-se o(s) mesmo(s) pacote(s) que foram demonstrados na ampliação em *(a)* para facilitar o entendimento do *testbench*. Em *(c)* é ilustrado um resumo geral da quantidade de pacotes enviados e recebidos pela rede e em *(d)* dá-se um resumo da atividade individual de cada roteador da rede, em função de quantos pacotes entram e saem por alguma de suas portas.

- **Situação 01**: Roteador 1x1 enviando 150 pacotes para o roteador 2x1. Teste realizado para verificar o funcionamento do envio de pacotes para um roteador vizinho.

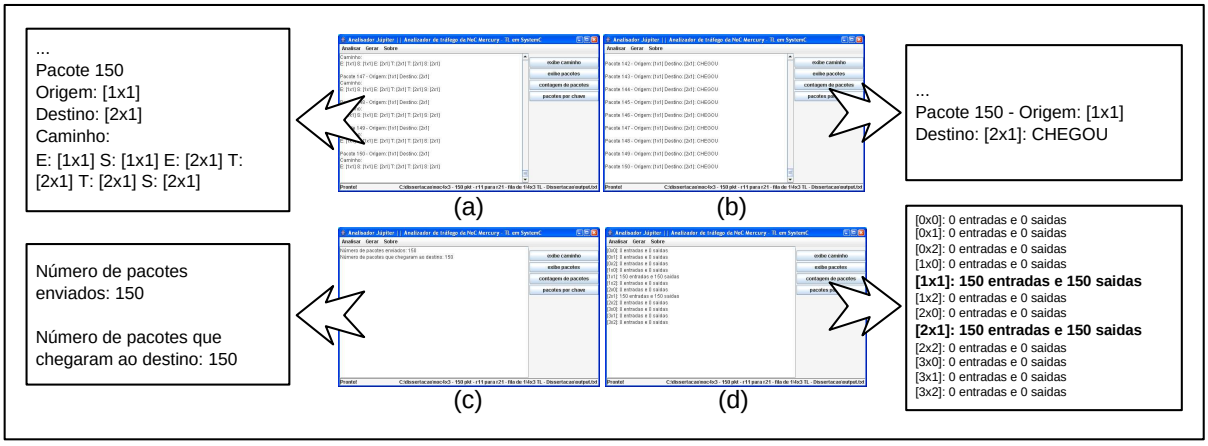


Figura 30 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com 150 pacotes enviados do roteador 1x1 para o roteador 2x1.

Na Figura 30 *(a)* pode-se verificar na ampliação a ilustração de caminhamento de um dos pacotes, no caso o pacote de número 150. Este pacote entrou no roteador origem 1x1 (E: [1x1]), saindo (S: [1x1]) para o roteador destino (E: [2x1]) pela Fila A. Ao chegar no roteador [2,1] o qual é destino do pacote, o mesmo trocou de fila para Fila B (T: [2x1]) e logo em seguida para Fila C (segundo T: [2x1]), sendo entregue após ao IP local. Em *(b)* é mostrado na ampliação que o pacote de número 150, saiu do roteador 1x1 e chegou com

sucesso no seu destino, roteador 2x1. Em *(c)* pode-se visualizar que todos os 150 pacotes enviados chegaram ao seu destino. E em *(d)* é mostrado que 150 pacotes entraram no roteador 1x1 (através do IP local) e que 150 passaram pelo conjunto de portas de saída da rede. Também se pode visualizar na Figura 30 *(d)*, que 150 pacotes entraram e saíram do roteador 2x1. Sabendo-se que foram injetados na rede 150 pacotes ao todo, e que os outros 10 roteadores não receberam nem enviaram nenhum pacote, pode-se afirmar que não se perdeu pacotes ao longo da rede. Também se pode afirmar que nenhum pacote foi roteado para um caminho incorreto.

- **Situação 02**: Roteador 1x0 enviando 150 pacotes para o roteador 3x2. Este teste foi realizado para verificar o funcionamento do *wraparound* vertical através do caminho 1x0 -> 1x2 -> 2x2 -> 3x3. Todos os pacotes injetados na rede chegaram com sucesso em seu destino através de um caminho mínimo, utilizando um canal *wraparound* entre os roteadores 1x0 e 1x2. Desta forma, pôde-se verificar, a partir do *testbench* mostrado na Figura 31 que a interligação entre os roteadores das pontas da rede no sentido vertical funciona corretamente.

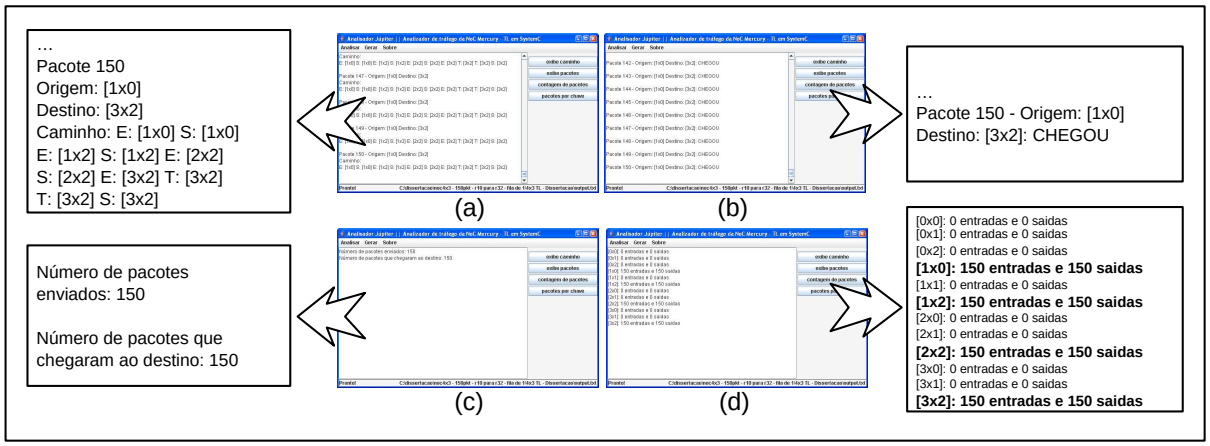


Figura 31 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com 150 pacotes enviados do roteador 1x0 para o roteador 3x2.

Na Figura 31 *(a)* verifica-se na ampliação a ilustração de caminhamento de um dos pacotes, no caso o pacote de número 150. O caminho utilizado para enviar este pacote foi do roteador origem 1x0 pela Fila A para o roteador 1x2, logo após para o 2x2 e em seguida para o 3x2 onde chegou ainda na Fila A. Em seguida, o pacote, trocou duas vezes de fila, para logo após sair do roteador pela Fila C. Em *(b)* mostra-se na ampliação que o pacote de número 150, saiu do roteador 1x0 e chegou com sucesso no seu destino, roteador 3x2. Em *(c)*, visualiza-se que todos os 150 pacotes enviados chegaram ao seu destino. E em

(d) é mostrado que todos os roteadores do caminho entre a origem e o destino, receberam e enviaram os 150 pacotes injetados na rede.

- **Situação 03**: Roteador 0x0 enviando 150 pacotes para o roteador 3x2. Teste realizado para verificar o funcionamento do *wraparound* vertical e posteriormente o horizontal através do caminho 0x0 -> 0x2 -> 3x2. Pode-se verificar, a partir dos resultados do *testbench* mostrado na Figura 32 que a interligação entre os roteadores das pontas da rede no sentido vertical e horizontal está funcionando corretamente.

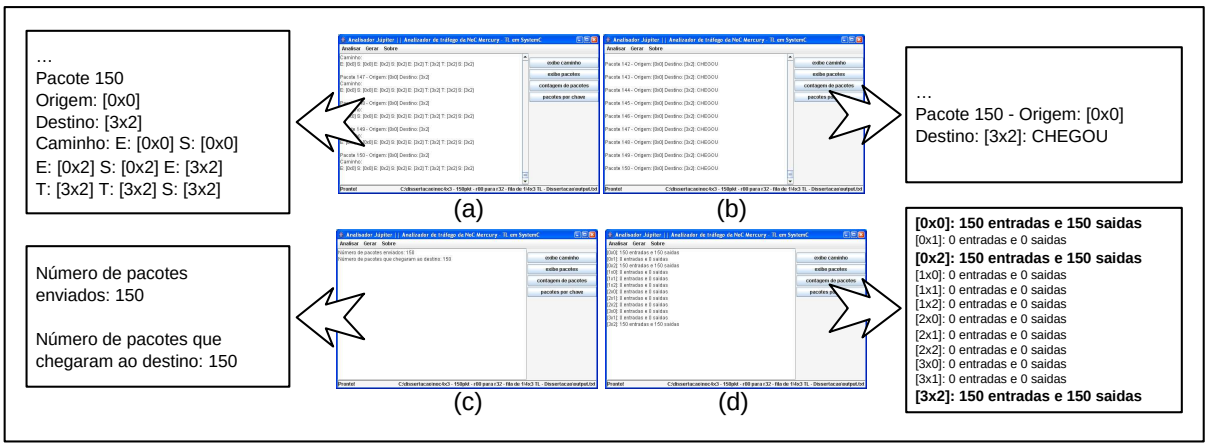


Figura 32 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com 150 pacotes enviados do roteador 0x0 para o roteador 3x2.

Na Figura 32 *(a)* verifica-se na ampliação a ilustração de caminhamento de um dos pacotes, no caso o pacote de número 150. O caminho utilizado para enviar este pacote foi do roteador origem 0x0 pela Fila A para o roteador 0x2 utilizando-se do *wraparound* vertical e logo após para o roteador 3x2 utilizando-se do *wraparound* horizontal. Por fim o pacote muda duas vezes de fila para poder sair da rede para o IP local. Em *(b)* mostra-se na ampliação que o pacote de número 150 saiu do roteador 0x0 e chegou com sucesso ao seu destino, roteador 3x2. Em *(c)*, visualiza-se que todos os 150 pacotes enviados chegaram ao seu destino. E em *(d)* é mostrado que todos os roteadores do caminho entre a origem e o destino, receberam e enviaram os 150 pacotes injetados na rede.

- **Situação 04**: Roteador 1x0 e 1x2 enviando cada 300 pacotes para o roteador 3x2 em paralelo. Este teste foi realizado para visualizar a adaptatividade do algoritmo CG1 com os pacotes saindo do roteador 1x0 e tomando caminhos diferentes dos normalmente utilizados, já que o caminho por onde os pacotes eram roteados (1x0 -> 1x2 -> 2x2 -> 3x2, Situação 02) agora está congestionado. É ilustrada neste *testbench* a utilização de três

caminhos mínimos para envio do roteador 1x0 e dois do roteador 1x2, mostrando que o algoritmo de roteamento é capaz de rotear pacotes por diversos dos caminhos mínimos da rede.

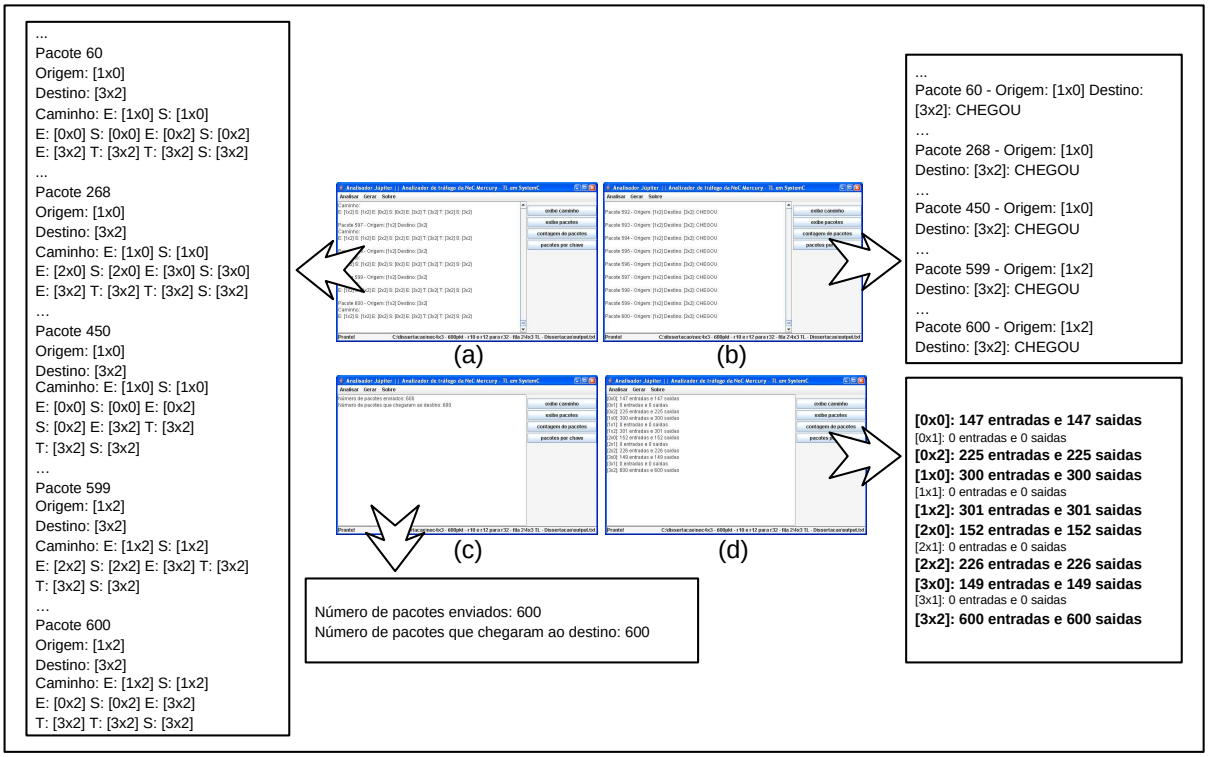


Figura 33 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com 600 pacotes enviados dos roteadores 1x0 e 1x2 para o roteador 3x2.

Na Figura 33 *(a)* verifica-se na ampliação a ilustração de cinco caminhamentos de pacotes. Os pacotes de número 60, 268 e 450 de origem 1x0 e destino 3x2, que no *testbench* da situação 02 eram encaminhados pelo caminho 1x0 -> 1x2 -> 2x2 -> 3x2, agora seguem por caminhos alternativos igualmente mínimos. O pacote 60 utiliza o caminho 1x0 -> 0x0 -> 0x2 -> 3x2, o pacote 268 utiliza o caminho 1x0 -> 2x0 -> 3x0 -> 3x2 e o 450 utiliza o caminho 1x0 -> 0x0 -> 0x2 -> 3x2. O roteador 1x2 envia pacotes utilizando-se de dois caminhos mínimos como mostrado nos caminhos percorridos pelos pacotes 599 e 600. O primeiro utiliza-se do caminho 1x2 -> 2x2 -> 3x2 e o segundo do caminho 1x2 -> 0x2 -> 3x2. Desta forma, fica demonstrado que o algoritmo de roteamento consegue dividir a carga da rede entre os diversos caminhos mínimos disponíveis.

Em *(b)*, mostra-se na ampliação que os pacotes de número 60, 268, 450, 599 e 600 chegaram em seus destinos. Em *(c)* visualiza-se que todos os 600 pacotes enviados chegaram ao seu destino. Finalmente, em *(d)* é demonstrado que todos os roteadores utilizados dos

diversos caminhos entre a origem e o destino, receberam e enviaram os pacotes enquanto que os roteadores que não faziam parte destes caminhos não receberam e também não enviaram pacotes.

- **Situação 05**: Todos os roteadores enviando pacotes para o roteador 1x0. 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.

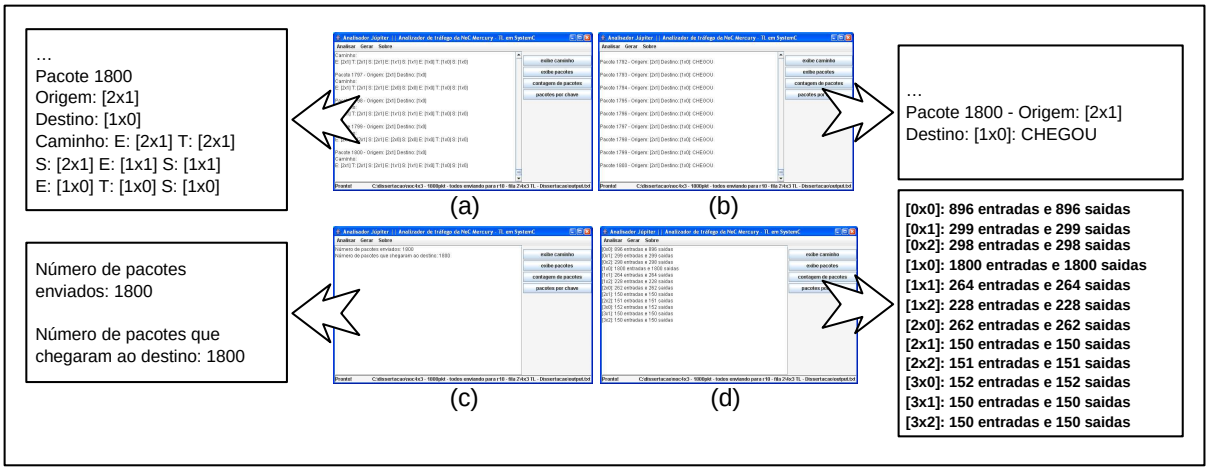


Figura 34 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com todos os roteadores enviando 150 pacotes para o roteador 1x0, totalizando 1800 pacotes trafegando na rede.

Na Figura 34 *(a)* verifica-se na ampliação a ilustração de caminhamento de um dos pacotes, no caso o pacote de número 1800. O caminho utilizado para enviar este pacote foi partir do roteador origem 2x1, inicialmente trocando da Fila A para a Fila B. Posteriormente o pacote vai para o roteador 1x1 e em seguida para o 1x0, onde novamente troca de fila para sair do roteador para o IP local. Em *(b)*, mostra-se na ampliação que o pacote de número 1800, saiu do roteador 2x1 e chegou com sucesso no seu destino, roteador 1x0. Em *(c)* visualiza-se que todos os 1800 pacotes enviados chegaram ao seu destino. Finalmente, em *(d)* é mostrado que todos os roteadores da rede injetaram pacotes e que o roteador destino 1x0 recebeu os 1800 pacotes por todas suas portas e enviou os mesmos para seu IP local.

- **Situação 06**: Todos os roteadores enviando 350 pacotes cada para todos os roteadores, de forma aleatória e uniforme, totalizando 4200 pacotes na rede. Teste realizado para visualizar o comportamento geral da rede em lidar com uma grande variedade de tráfego

em todos os roteadores, conforme ilustra a Figura 35.

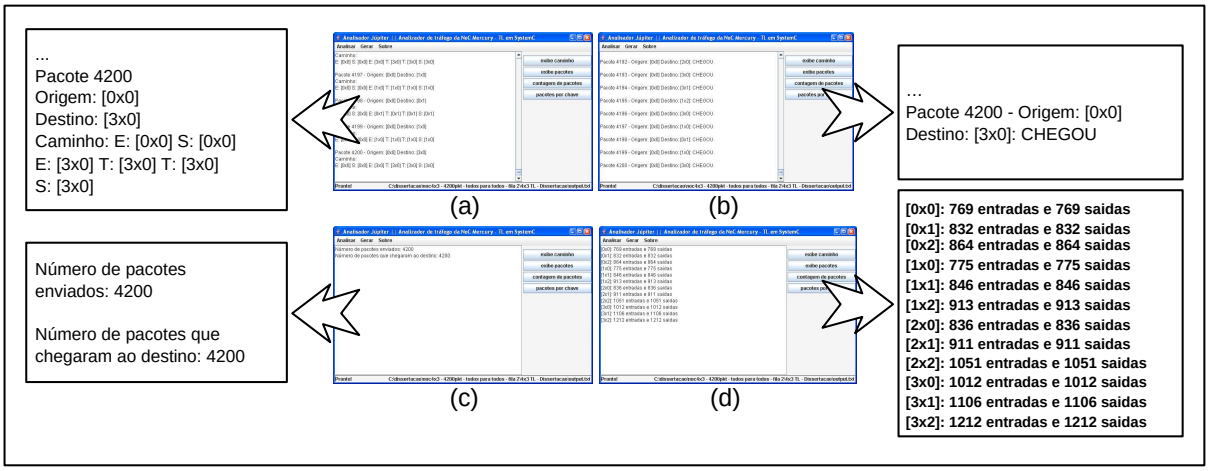


Figura 35 – Resultado parcial da execução do *testbench* para validação em nível TL de uma NoC 4x3 com todos os roteadores enviando 350 pacotes para a rede, totalizando 4200 pacotes.

6.2 Modelagem RTL

Nesta Seção será apresentada a implementação no nível de transferência entre registradores da NoC Mercury mediante emprego da linguagem de descrição de *hardware* VHDL [4]. Para validar a implementação dos módulos, componentes do roteador, do próprio roteador e da rede analisa-se formas de onda parciais obtidas pela execução dos *testbenchs* utilizados na ferramenta Active-HDL [2].

6.2.1 Modelagem do Roteador Mercury

Os módulos que compõem o roteador Mercury RTL são praticamente os mesmo módulos do roteador TL, salvo pela estrutura de interligação. O roteador Mercury RTL possui os seguintes tipos de módulos:

- *intoArbiter*: módulo para selecionar os pacotes referentes a uma determinada fila;
- *arbiterIn*: módulo com a função de selecionar a porta que será atendida naquele instante;

- *queue*: fila para armazenamento temporário de pacotes;
- *arbiterOut*: módulo para definir a porta de saída do roteador para a qual o pacote será enviado, baseado no algoritmo CG1;
- *ack_nackOut*: módulo para realizar o *handshake* entre o roteador e os recursos que desejam enviar pacotes para este;
- *ack_nackIn*: módulo para realizar o *handshake* entre o roteador e os receptores de pacotes deste.

Um roteador completo é composto de um módulo *ack_nackOut*, três *intoArbiter*, três *arbiterIn*, três *queue*, um *ack_nackIn* e um *arbiterOut*. De forma diferente do que ocorre na implementação TL, na versão RTL não existem canais para realizar a comunicação entre os módulos e sim interfaces físicas, compostas por conjuntos de fios de interconexão. A Figura 36 ilustra a arquitetura de um roteador RTL. Essa arquitetura pode ser vista com maiores detalhes no Anexo Apêndice A Figuras 69, 70, 71.

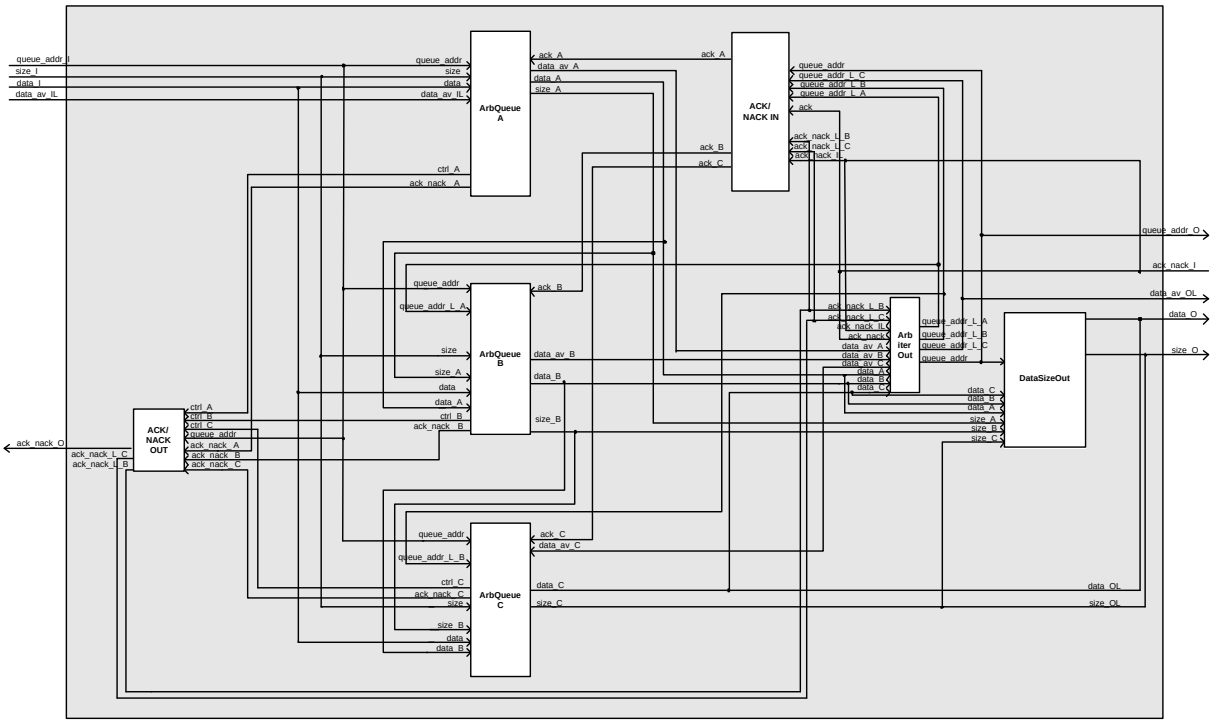


Figura 36 – Visão do roteador completo - NoC Mercury RTL. O módulo denominado *ArbQueue* contém represente os módulos *into_Arbiter*, *arbiterIn* e *queue*. Já o módulo *DataSizeOut* é somente a união de todos multiplexadores de saída dos sinais *size* e *data*.

Cada módulo pertencente ao roteador na implementação RTL será descrito a seguir em maior detalhe:

Módulo *ack_nackOut*

O módulo *ack_nackOut* é responsável por realizar o *handshake* entre o roteador e os recursos que desejam enviar pacotes para este. Ele comunica-se diretamente com um gerador externo de pacotes e com os três módulos *intoArbiter* e *arbiterIn* do roteador, para tentar realizar uma troca de pacotes. Seu funcionamento se dá baseado nos sinais que ele envia para o gerador externo quando este deseja realizar uma comunicação. Esses sinais basicamente podem ter três valores *ack*, *nack* ou *none*, e são recebidos do módulo *arbiterIn*. Através desta interface informa-se ao gerador externo se existe ou não condição de receber o pacote. O gerador externo pode ser um roteador vizinho ou uma porta de saída do IP local do roteador.

Módulo *intoArbiter*

Este módulo é único para cada fila. Assim como na implementação TL, tem como objetivo analisar e separar as requisições dos pacotes que chegam ao roteador de acordo com sua fila de destino. Ele recebe solicitações de armazenamento de todas as cinco portas do roteador e o endereço da fila onde o pacote deve ser armazenado. Com essa informação, o módulo somente repassa para *arbiterIn* os pedidos de armazenamento pertencentes à fila a qual o último está associado.

Módulo *arbiterIn*

O módulo *arbiterIn*, tanto na implementação TL quanto na RTL, é único para cada fila. Ele é responsável por receber requisições de escrita de pacotes e enviá-las para o módulo *queue* aplicando uma prioridade *Round-Robin*. As requisições que chegam neste módulo já foram filtradas pelo módulo *intoArbiter*, de forma que quando este recebe alguma requisição para armazenamento ela pode ser tratada imediatamente. A Figura 37 detalha a funcionalidade deste módulo como uma máquina de estados finita.

O módulo *arbiterIn* recebe as requisições para armazenamento de *intoArbiter*. Ao mesmo tempo, recebe o tamanho do dado a ser armazenado diretamente do gerador externo que está enviando o pacote. De posse dessas informações, *arbiterIn* verifica se existe espaço na fila para o armazenamento completo do pacote. Caso exista espaço suficiente, *arbiterIn* envia ao módulo *ack_nackOut* um sinal de *ack*, caso contrário um sinal de *nack* é enviado, conforme ilustra a máquina de estados da Figura 37.

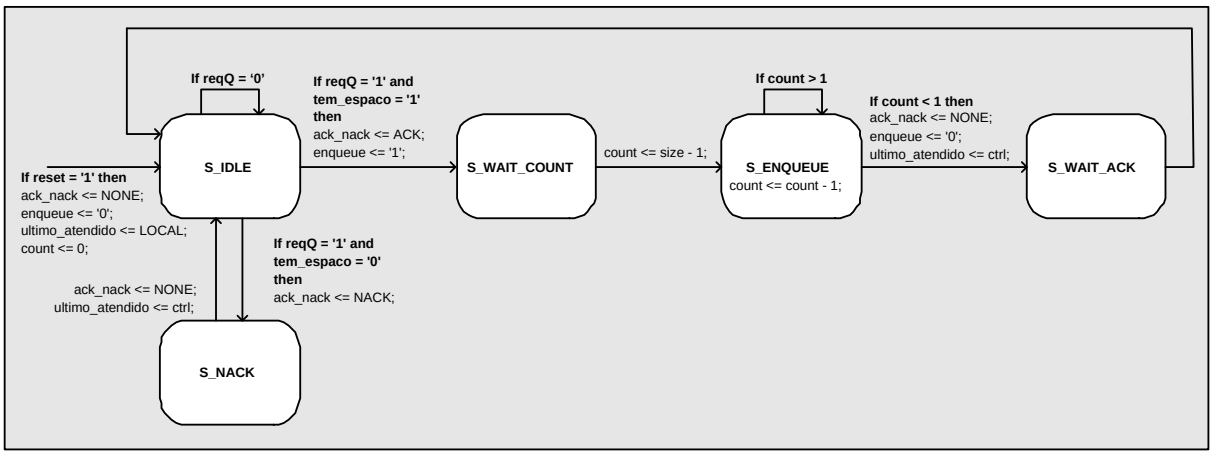


Figura 37 – Máquina de estados do módulo *arbiterIn*.

Módulo *queue*

Este é o módulo que implementa a fila. Assim como na implementação TL, possui funções de controlar o armazenamento e a retirada de pacotes. No roteador, existem três módulos *queue* denominados Fila A, Fila B e Fila C. A fila implementada pelo módulo *queue* é de porta dupla e circular, onde os pacotes que chegam são armazenados na primeira posição livre, e os pacotes que saem são retirados da primeira posição ocupada. Essas operações podem ser realizadas em paralelo.

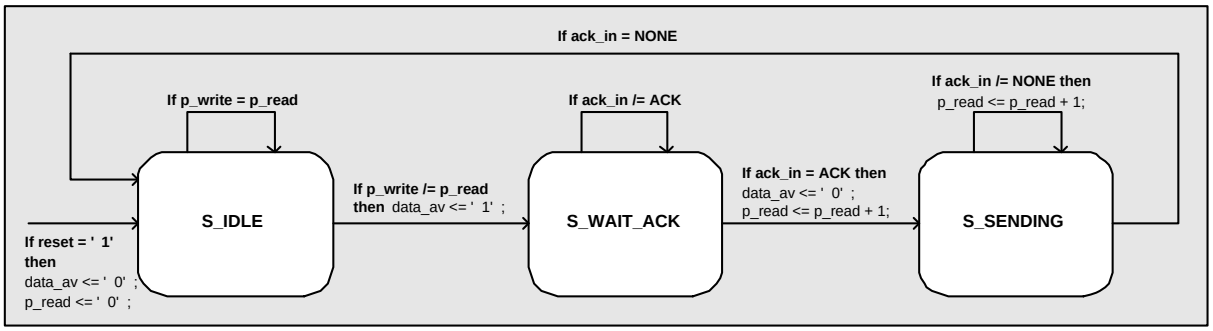


Figura 38 – Máquina de estados fi nita de saída do módulo *queue*.

O módulo *queue* possui um processo para armazenamento dos dados na fila e uma máquina de estados para retirada e envio dos dados presentes nela. O funcionamento do processo de escrita é bem simples, uma vez que exista uma requisição de escrita enviada pelo módulo *arbiterIn*, o processo inicia escrevendo um dado a cada ciclo de *clock*. A máquina de estados finita de saída do módulo *queue* opera conforme ilustrado na Figura 38. Quando o sinal *p_write* for diferente do sinal *p_read*, significa que existe um dado na fila que ainda não foi enviado. Neste

momento é disparado o processo de requisição para envio do dado para o módulo *arbiterOut*, que por sua vez irá verificar por qual porta do roteador este dado será enviado. Os sinais *p_write* e *p_read* informam a posição dos ponteiros de leitura e de escrita respectivamente. Quando os dois apontarem para a mesma posição significa que a fila está vazia.

Módulo *ack_nackIn*

O módulo *ack_nackIn* é o responsável por realizar o *handshake* entre o roteador e algum receptor de pacotes deste. Sua comunicação se dá através dos módulos *queue* e *arbiterOut* do roteador e da interface com o receptor do pacote. Sua função é realizar o controle das filas e das portas de saída do roteador, de forma que todas as filas tenham possibilidade de acessar todas as portas (uma de cada vez) em um determinado momento.

Módulo *arbiterOut*

O módulo *arbiterOut*, tanto na implementação TL quanto na RTL é o responsável por executar o algoritmo de roteamento, selecionando as possíveis portas para envio de um determinado pacote para os roteadores vizinhos, para outra fila dentro do próprio roteador ou para o IP local. Existem três processos paralelos independentes que controlam o envio, sendo um para cada fila (A, B e C), que podem enviar pacotes simultaneamente, desde que por portas distintas. Este módulo está diretamente interligado com os módulos *ack_nackIn* e *queue* do roteador, bem como com recursos externos receptores do pacote. Cada processo de envio de dados possui uma máquina de estados onde é executado o algoritmo de roteamento. Na Figura 39 ilustra-se a máquina de estados da Fila A contida no módulo *arbiterOut*, sendo que para as filas B e C as máquinas possuem estruturas idênticas.

Supor que a máquina de controle de *arbiterOut* está no estado S_IDLE e recebe de *queue* a informação de que existe pelo menos um pacote disponível na fila A para emissão, através do sinal *data_av_A*. Este fato dispara o cálculo de que portas de saída do roteador conectam-se a enlaces participando em algum caminho mínimo para o destino do pacote. Está informação é armazenada no vetor *caminho_minimos_A*. Por exemplo, caso o sinal *caminhos_minimos_A(Norte)* seja igual a "1", significa que o roteador pode enviar o pacote pela porta Norte, pois utilizando-se esta porta o pacote sairá em direção ao roteador destino por um caminho mínimo. Caso o valor do sinal seja "0", isto significa que aquela porta não está apta para rotear o pacote naquele instante, pois não faz parte de um caminho mínimo.

De posse das portas para onde se pode enviar o pacote para seguir por um caminho mínimo, o módulo *arbiterOut* verifica cada porta por vez na ordem Norte, Sul, Leste, Oeste, verificando

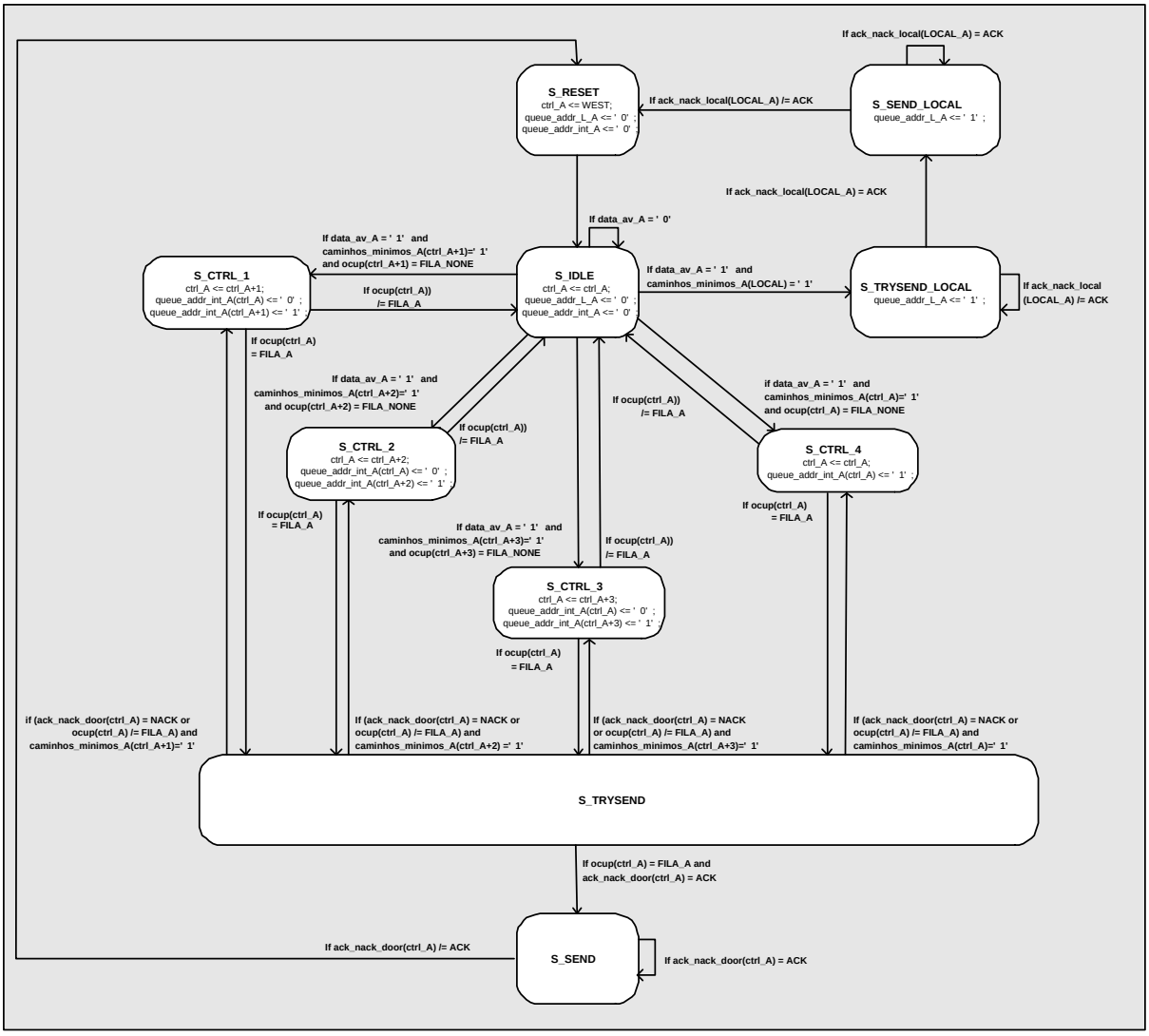


Figura 39 – Máquina de estados de saída do módulo *arbiterOut*.

qual tem possibilidade de receber o pacote. Este processo se repete entre todas as portas de forma cíclica, até que o dado seja enviado. No caso do algoritmo de roteamento retornar como caminho mínimo a porta Local, o pacote deve ser roteado por esta porta prioritariamente, levando em consideração a fila específica onde este se encontra. Se o pacote está na Fila A e existe um caminho mínimo passando pela porta Local, o dado será enviado para a Fila B. Caso o pacote já esteja na Fila B, o dado será roteado para a Fila C e daí para o IP local.

6.2.2 Modelagem da Rede Mercury RTL

A implementação da rede Mercury RTL se dá através da interligação de roteadores para implementar a topologia toro 2D. A Figura 40 descreve as interfaces de comunicação de um roteador. A interface de comunicação entre roteadores vizinhos utiliza os sinais *queue_addr*, *size*, *data* e *ack_nack* para realizar o *handshake* e a troca de dados entre eles. A interface de comunicação para o IP local é um pouco diferente, sendo o sinal *queue_addr* substituído pelo sinal *data_av*, conforme discussão anterior apresentada no Capítulo 5, Seção 5.4. Na Figura 40 ilustra-se um roteador presente em uma rede, detalhando os sinais da porta Oeste ligados ao roteador vizinho e também os sinais da interface com o IP local.

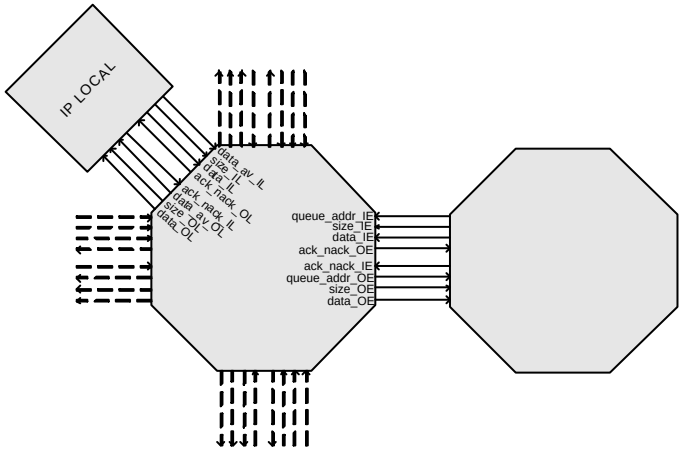


Figura 40 – Interligação de um roteador na rede Mercury, detalhando os sinais da comunicação com seu vizinho pela porta Oeste e a interface com o IP local.

6.2.3 Validação NoC Mercury RTL

Nessa Seção serão descritos os *testbenchs* utilizados para validar a NoC Mercury RTL. A partir da criação de arquivos para geração de tráfego, analisou-se o comportamento de determinados módulos quando submetidos ao tráfego gerado. Como resultado, pôde-se analisar as formas de onda na ferramenta Active-HDL. A seguir serão descritos os principais testes realizados.

Módulo *intoArbiter*

A simulação apresentada na Figura 41 ilustra o funcionamento do módulo *intoArbiter* ligado à Fila A. Este módulo filtra requisições que chegam a ele, repassando ao módulo *arbiterIn* somente as que são direcionadas para a Fila A.

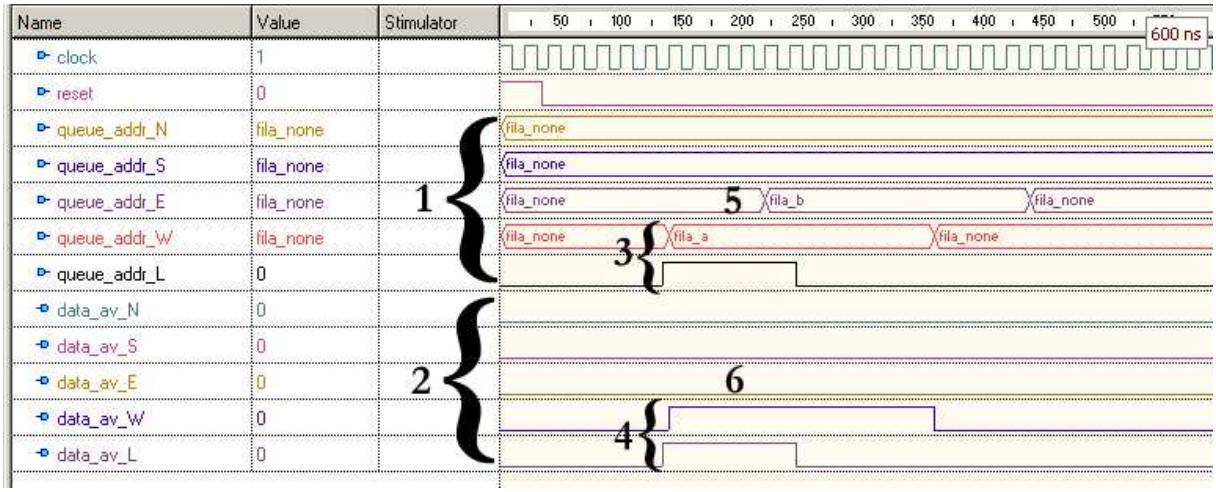


Figura 41 – Exemplo de resultado parcial de simulação do módulo *intoArbiter* associado a Fila A.

Os passos da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 41.

1. Sinais de entrada do módulo *intoArbiter*.

2. Sinais de saída do módulo *intoArbiter*.

3. Os sinais *queue_addr* das portas Oeste e Local requisitam permissão de escrita na Fila A durante um mesmo ciclo de relógio.

4. Os sinais *queue_addr* das portas Oeste e Local são repassados através dos sinais *data_av* para o módulo *arbiterIn*.

5. O sinal *queue_addr* da porta Leste requisita a escrita na Fila B.

6. Apesar do sinal *queue_addr* Leste requisitar uma escrita, o *data_av* não repassa a mesma para o *arbiterIn* pois a requisição não é direcionada à fila a qual este módulo está associado.

Ao mesmo tempo em que o *intoArbiter* associado à Fila A realiza a seleção de requisição, os outros módulos *intoArbiter* associados às outras duas filas trabalham em paralelo, selecionando dados relativos as suas filas. A Figura 42 ilustra o funcionamento do módulo *intoArbiter* associado a Fila B.

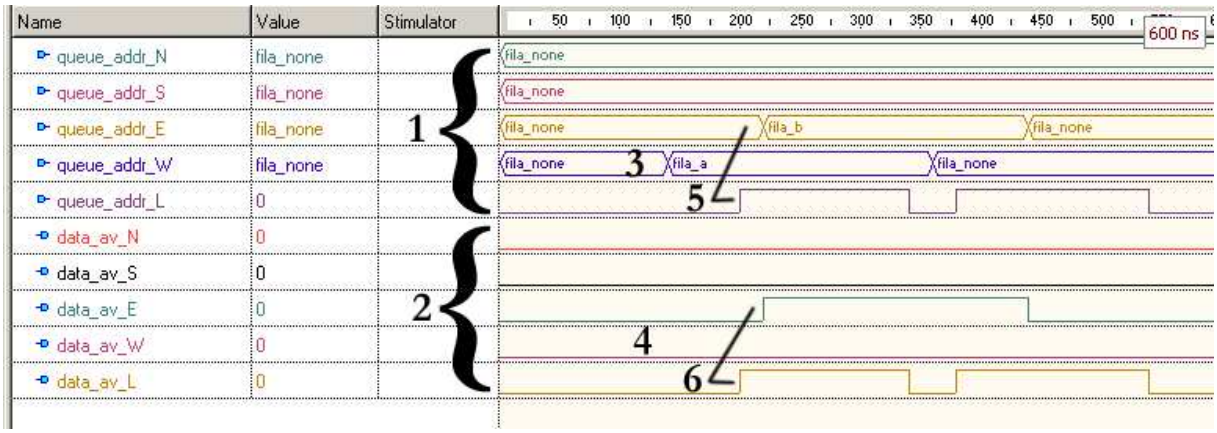


Figura 42 – Exemplo de resultado parcial de simulação do módulo *intoArbiter* associado à Fila B.

Os passos da simulação são descritos abaixo, onde a numeração tem correspondência com a Figura 42.

1. Sinais de entrada do módulo *intoArbiter*, iguais ao do intoArbiter associado à Fila A, sendo a única diferença o *queue_addr_L*.

2. Sinais de saída do módulo *intoArbiter*.

3. Requisição da porta Oeste para escrita na Fila A.

4. O módulo *intoArbiter* associado à Fila B não repassa a requisição ao *arbiterIn* da sua fila.

5. As portas Leste e Local requisitam escrita na Fila B.

6. Ambas as portas que requisitaram escrita na Fila B, são repassadas para o *arbiterIn*.

A partir destas duas simulações visualiza-se um exemplo de funcionamento correto do módulo *intoArbiter* que analisa e separa as requisições dos pacotes que chegam ao roteador de acordo com a fila de destino.

Módulo *arbiterIn*

Uma simulação parcial do funcionamento do módulo *arbiterIn* associado à Fila A é ilustrada na Figura 43. Este módulo recebe requisições de acesso apenas da fila associada a ele, pois *intoArbiter* já filtrou as mesmas. A partir das requisições recebidas, *arbiterIn* envia as mesmas para a fila através de prioridade *Round-Robin*.

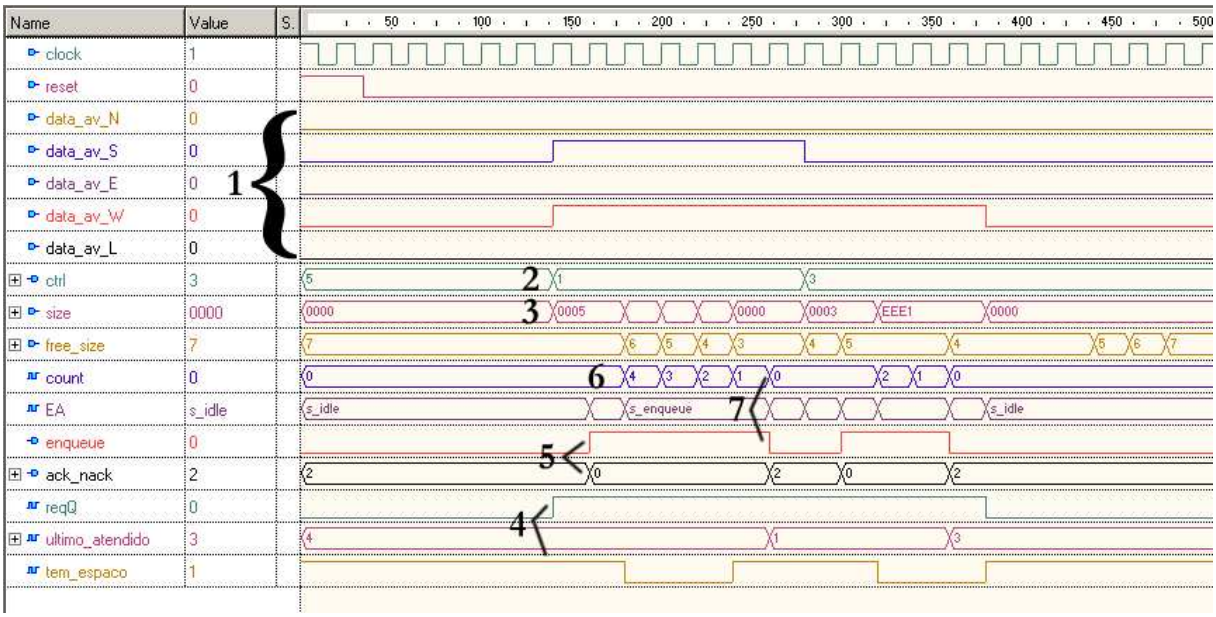


Figura 43 – Exemplo de resultado parcial de simulação do módulo *arbiterIn* associado à Fila A.

Os passos da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 43.

1. Sinais representando requisições de acesso à fila recebidos através da comunicação com módulo *intoArbiter*. Nota-se o surgimento simultâneo de duas requisições das portas Sul e Oeste.

2. O Sinal *ctrl* armazena a última porta atendida. No início da simulação o valor padrão é 5 que representa o valor *port_none*. A prioridade *Round-Robin*, usa a ordem numérica circular 5(port_none) -> 0(Norte) -> 1(Sul) -> 2(Leste) -> 3(Oeste) -> 4(Local) -> 0(Norte) -> 1(Sul). Como as portas Sul e Oeste requisitaram acesso à fila no mesmo ciclo de relógio, e o sinal *ctrl* está com o valor 5, a próxima porta a ser atendida será a porta Sul.

3. Quando o sinal *ctrl* recebe o valor da nova porta que ganhou o acesso a fila, a informação referente ao tamanho do pacote desta porta chega em *arbiterIn* pelo sinal *size*.

4. No mesmo momento, *arbiterIn* verifica se existe espaço para armazenar o pacote, verificando o sinal *tem_espaco*, que informa se a diferença entre o tamanho do pacote e o espaço disponível na fila (*free_size*), é positivo.

5. Caso a condição de espaço seja satisfeita, *arbiterIn* envia para a fila, através do sinal *enqueue*, um aviso de armazenamento, e ao mesmo tempo envia à porta que requisitou a escrita na fila um *ack* através do sinal *ack_nack*, onde 0 representa *ack*, 1 *nack* e 2 *none*.

6. O sinal *count* recebe o tamanho do dado a ser enviado no momento do aceite da requisição, já decrementado de uma unidade, pois o primeiro *phit* já foi enviado no momento da requisição de acesso.

7. Quando o valor do sinal *count* se torna zero, significa que todo o pacote já foi enviado para a fila. Neste instante *arbiterIn* coloca o sinal *enqueue* sinalizando para a fila parar de armazenar dados provenientes daquela porta.

Após o enviado do pacote descrito, a prioridade *Round-Robin* é novamente executada. Desta vez, a porta Oeste que ficou aguardando é atendida e todo o procedimento descrito antes se repete.

Módulo *queue*

A Figura 44 ilustra uma simulação parcial mostrando o funcionamento do módulo *queue*, sendo este equivalente para todas as três Filas, A, B e C. A fila validada possui tamanho de 7 *phits*, conforme mostra o sinal *free_size* no tempo "0"*ns* de simulação.

Os passos da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 44.

1. *queue* recebe o sinal *enqueue* do *arbiterIn*, indicando que deve dar início ao armazenamento de *phits*. O sinal *data_in* contém os dados propriamente ditos provenientes do recurso transmissor do pacote, gerando um novo *phit* a cada ciclo de *clock*. A cada *phit* recebido e armazenado na fila (representada pelo sinal *buf1*), a cabeça de leitura representada pelo sinal *p_write* avança uma posição.

2. A máquina de estados de *queue* detecta que a cabeça de leitura *p_read* é menor que a cabeça de escrita *p_write*.

3. A máquina de estados de saída de *queue* é executada, modificando seu estado para *s_wait_ack* e escrevendo "1" no sinal *data_av*.

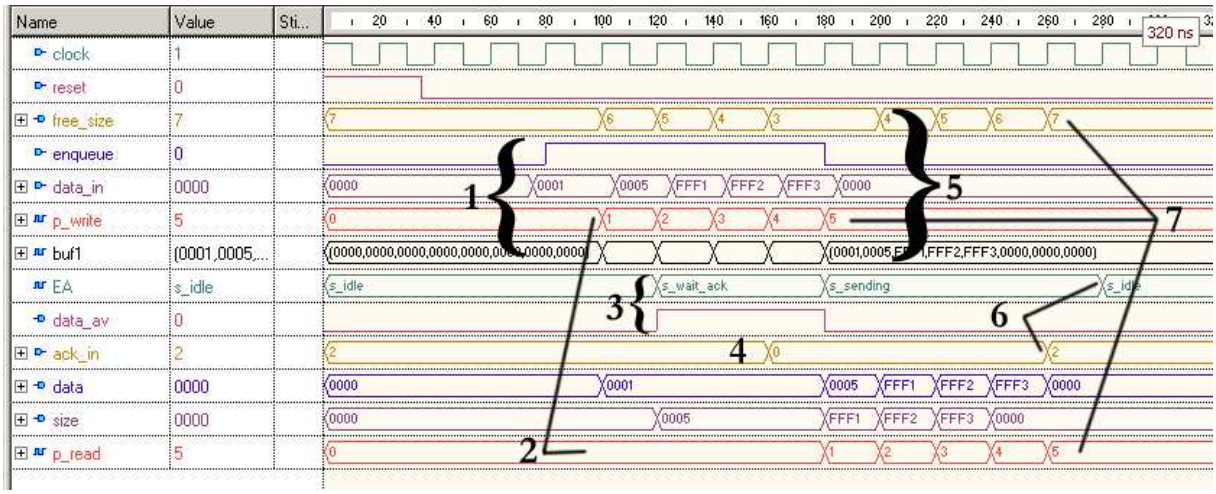


Figura 44 – Exemplo de resultado parcial de simulação do módulo *queue*, mostrando o recebimento e envio de um pacote.

4. Quando a requisição de comunicação é concedida pelo recurso destino do pacote e este aceita receber o pacote, *queue* recebe um *ack* através do sinal *ack_in*, fazendo com que a máquina de estados de saída mude seu estado para *sending* (enviando). Neste momento, já se inicia a transferência do pacote para o recurso seguinte, em paralelo com a recepção do restante do pacote. Nota-se que a cabeça de leitura (p_read) avança em uma posição a cada *clock* durante o envio.

5. Ao mesmo tempo em que *queue* envia *phits* para outra fila, ele continua a receber o pacote mencionado no item "1". Quando o sinal *enqueue* volta novamente ao seu estado inicial "0", o recebimento do pacote terminou. Neste instante, o sinal *buf1* possui armazenado o dado por completo e a cabeça de escrita contêm a última posição escrita do dado. Nota-se que o sinal *free_size* é decrementado até este instante, mas como o envio do dado da fila está em execução, seu valor volta a aumentar novamente.

6. Ao receber o pacote transmitido, o receptor sinaliza através do sinal *ack* modificando seu valor de *ack* para *none*. Com base nessa informação, a máquina de saída de *queue* volta ao seu estado inicial, aguardando um novo dado para enviar.

7. Ao final do recebimento e envio do pacote, os valores dos ponteiros de leitura e escrita avançaram em número igual ao tamanho do dado enviado. Como o pacote já foi enviado, o tamanho livre na fila (sinal *free_size*) volta ao seu valor inicial, indicando que existem novamente sete posições livres.

Módulo *arbiterOut*

O *testbench* para o módulo *arbiterOut* objetiva validar o algoritmo de roteamento. A Figura 45 ilustra o comportamento de *arbiterOut* quando submetido a uma requisição de envio de dados. O roteador onde o módulo *arbiterOut* testado está inserido ocupa a posição 1x1 em uma rede 3x3.

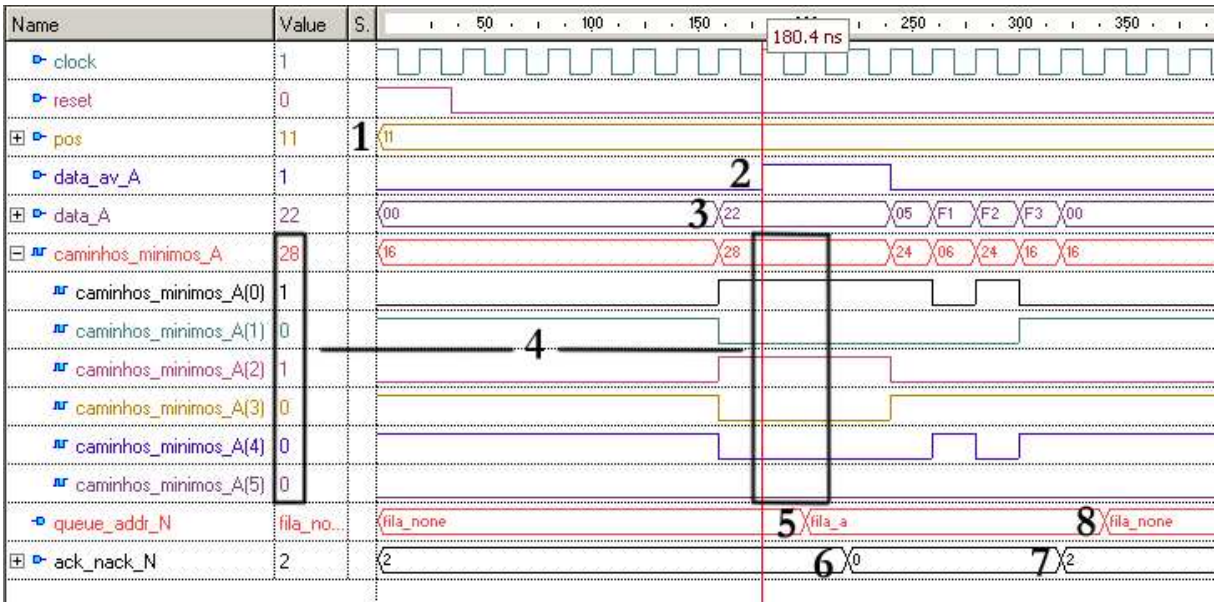


Figura 45 – Exemplo de resultado de simulação do módulo *arbiterOut*.

Abaixo se descreve os passos da simulação, onde a numeração tem correspondência na Figura 45.

1. O sinal *pos* indica a posição do roteador avaliado na rede, neste caso posição 1x1.
2. O sinal *data_av* em "1" ativa o processo de verificação dos caminhos mínimos.
3. O sinal *data_A* contém o primeiro *phit* do pacote a ser enviado que representa o roteador destino (2x2).
4. Com base nos sinais *pos* e *data_A*, quando recebe a ativação através do sinal *data_av*, *arbiterOut* verifica os caminhos mínimos existentes entre o seu roteador e o roteador destino. Aqui o sinal *caminhos_minimos* nas posições "0" (Norte) e "2" (Leste) está ativo (em "1"), informando que o pacote pode ser roteado pelas portas Norte e Leste. O módulo *arbiterOut* sempre tenta rotear os pacotes pelas portas disponíveis no sinal *caminhos_minimos* seguindo o ordenamento Norte, Sul, Leste, Oeste. Sempre que a

posição "4", representando a porta Local estiver ativa (em "1") ela terá prioridade sobre as demais, e o pacote deve ser roteado por este caminho.

5. *arbiterOut* requisita o envio do pacote pela porta Norte para o roteador 1x2 através do sinal *queue_addr_N*. Este sinal é a entrada do recurso receptor, e informa que existe uma requisitação de armazenamento de pacote na Fila A deste recurso.

6. Através de um *ack* (valor "0") enviado via sinal *ack_nack*, informa-se a *arbiterOut* que o recurso destino aceitou receber o pacote.

7. O sinal *ack_nack* retorna ao valor *none* quando todo o pacote foi recebido pelo recurso destino.

8. Neste instante, a porta Norte que estava sendo utilizada pela Fila A é liberada para aceitar requisições de qualquer uma das filas.

Roteador Completo

O exemplo de simulação do roteador completo foi dividido em duas partes, ilustradas nas Figuras 46 e 47. Na primeira parte, demonstra-se o envio do dado pelo IP local para o roteador. Na segunda parte, visualiza-se que o dado percorreu um caminho correto por dentro do roteador e saiu pela porta Norte em direção ao seu destino sobre um caminho mínimo. Cabe ressaltar que o roteador testado ocupa a posição 1x1 em uma rede de dimensões 3x3. O pacote enviado tem como destino o roteador 2x2. Desta forma, o roteamento pela porta Norte pertence realmente a um caminho mínimo, visto que o pacote entra no roteador na Fila A.

Os passos da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 46.

1. O vetor *queue_addr_I* contém cinco posições, cada uma referente a uma porta do roteador, indicando a fila onde o pacote requisitado deve ser armazenado. Nota-se que para a última posição, o valor é *fila_a*. Isto ocorre porque qualquer dado oriundo do IP local deve ser armazenado sempre nesta fila.

2. O sinal *size_I(4)* (relativo à porta Local), indica que o tamanho do dado requisitado para armazenamento neste roteador tem tamanho de cinco *phits*.

3. Neste momento o sinal *data_I(4)* contém o primeiro *phit* do pacote a ser armazenado. Este dado corresponde ao endereço de destino do pacote (roteador 2x2).

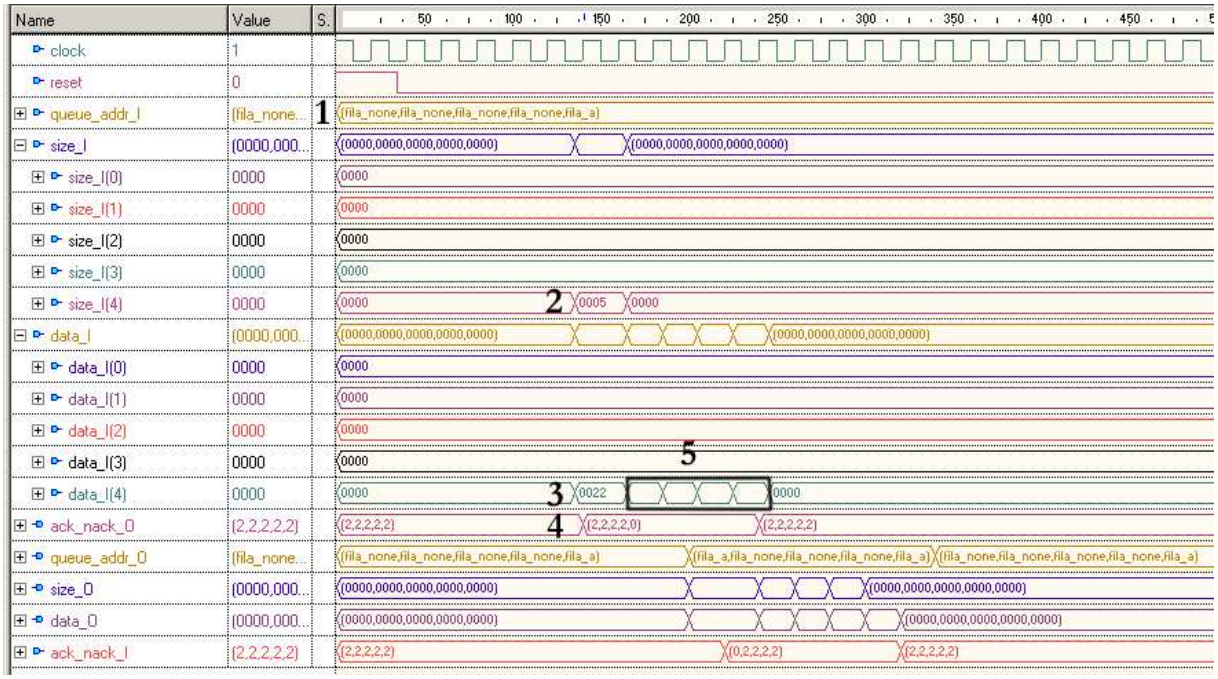


Figura 46 – Exemplo de resultado parcial da simulação do roteador por completo - Parte 1.

4. A partir dos sinais descritos nos itens 1, 2 e 3 é realizada uma verificação se o roteador pode ou não aceitar o pacote. No caso da simulação é sinalizado através de um *ack* no sinal *ack_nack_O(4)* que o pacote já pode ser enviado.

5. Com o envio do *ack*, a cada ciclo de *clock* o sinal *data_I(4)* envia um *phit* do pacote.

Os passos da parte 2 da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 47.

1. O sinal *queue_addr_O(0)* indica que existe um pacote para ser enviado pela porta Norte na Fila A, pois o valor fila_a está presente na posição "0"(Norte) do vetor.

2. O sinal *size_O(0),* indica que o dado a ser enviado possui tamanho de cinco *phits*.

3. O sinal *data_O(0)* mostra o dado propriamente dito sendo enviado após o recebimento do *ack* (ver item 4 a seguir).

4. Na posição referente à porta Norte, por onde a requisição de envio do dado saiu do roteador é enviado um *ack* (ack_nack_I(0) = 0) pelo receptor, confirmando que a requisição foi aceita e que o envio já pode iniciar.

5. Quando o sinal *ack_nack_I(0)*, volta ao estado *none*, o processo de envio é imediatamente concluído.

6. Neste mesmo instante, a porta Norte é liberada para envio de outros dados provenientes de qualquer uma das cinco portas.

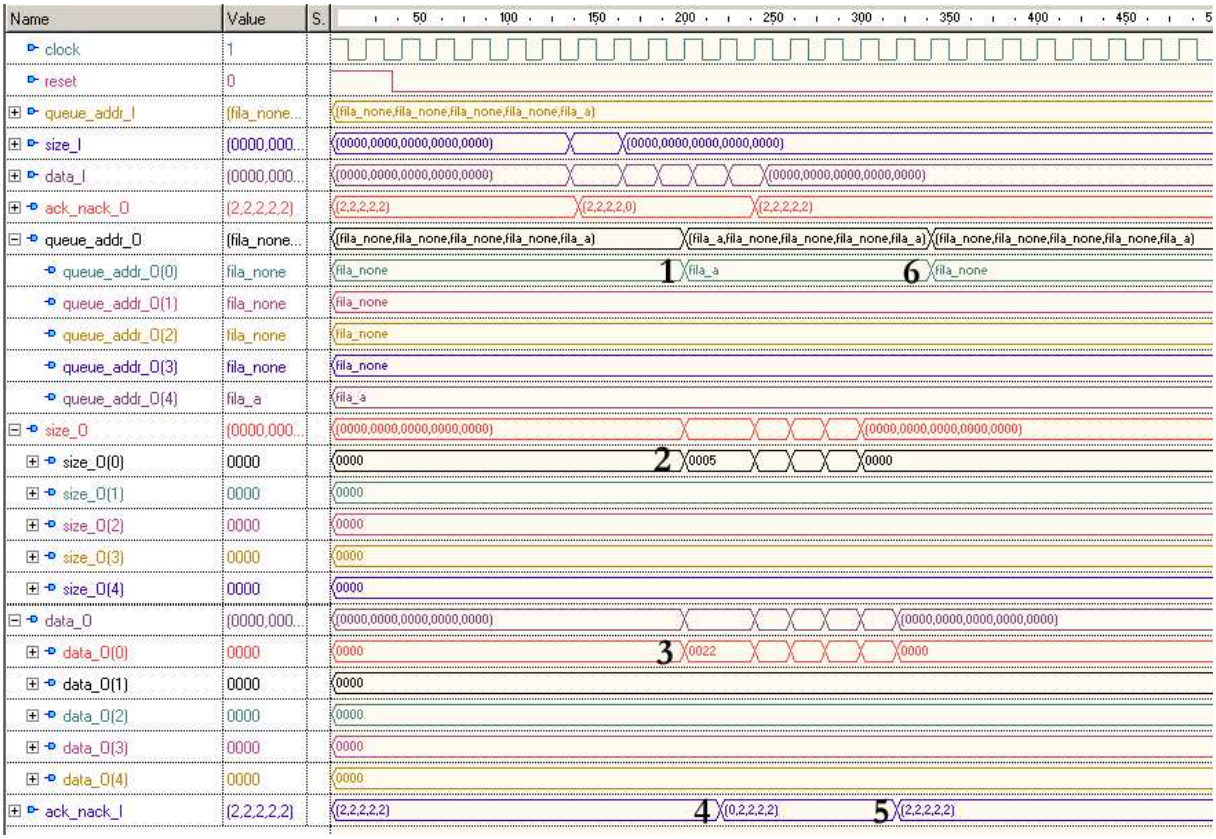


Figura 47 – Exemplo de resultado parcial de simulação do roteador por completo - Parte 2.

NoC Mercury RTL

O exemplo de simulação da rede completa foi igualmente dividido em duas partes para facilitar sua compreensão. A rede simulada é de dimensões 3x3, sendo que o pacote tem origem no IP local do roteador 2x0 e destino no IP local do roteador 2x2. Os passos realizados na parte 1 da simulação são descritos abaixo, onde a numeração tem correspondência na Figura 48.

1. Uma requisição para armazenamento chega ao roteador 2x0 de seu IP local.

2. O sinal *size_int_I20(4)* informa que o pacote possui tamanho de cinco *phits*.

3. O sinal *data_int_I20(4)* apresenta os sucessivos *phits* do pacote.

4. O sinal *ack_nack_int_O20(4)* indica através de um *ack* no índice 4 do vetor que a Fila A tem condições de armazenar o dado proveniente do IP local.

5. Neste instante é realizado um *handshake* entre o roteador 2x0 e o 2x2, de forma que o pacote comece a ser enviado pela porta Sul, conforme mostra a requisição no sinal *queue_addr_int_O20(1)* e a aceitação para envio do dado através do *ack* recebido no sinal *ack_nack_int_I20(1)*.

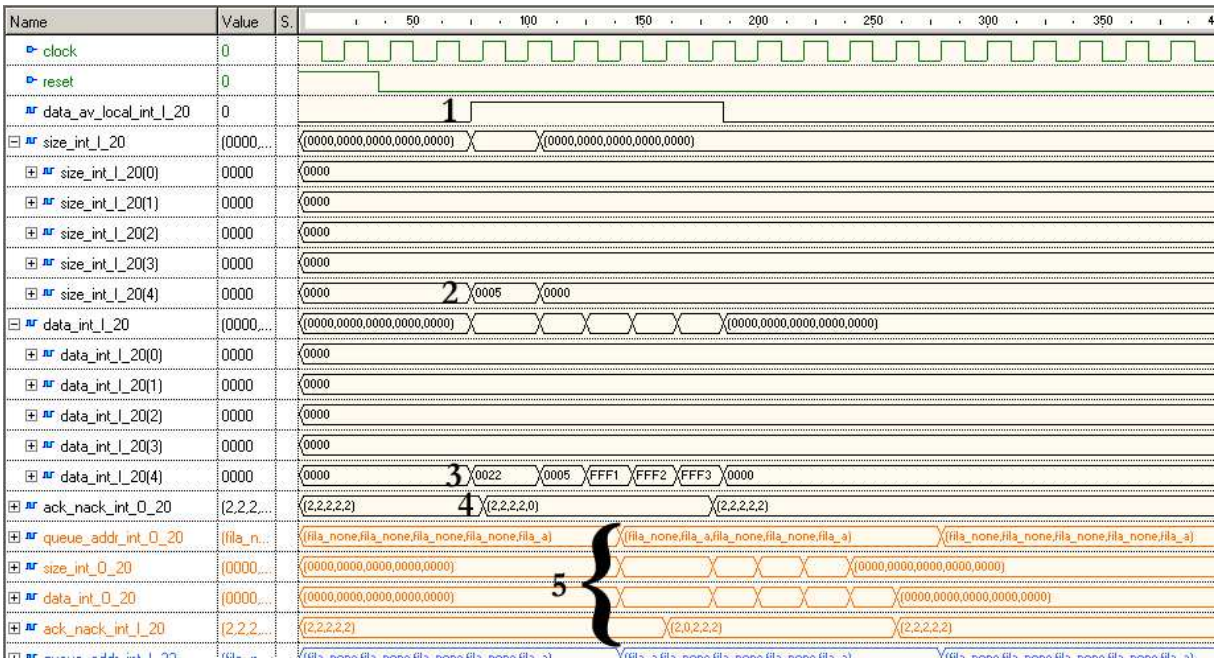


Figura 48 – Exemplo de resultado parcial de simulação de uma NoC 3x3 - Parte 1.

A Parte 2 da simulação, ilustrada na Figura 49, representa o dado saindo da Fila C para o IP local do roteador 2x2. Os passos desta simulação são descritos abaixo, onde a numeração tem correspondência na Figura 49.

1. Quando o dado chega à Fila C e está pronto para ser enviado ao IP local o sinal *data_local_int_O22* tem seu valor alterado para "1", indicado que o roteador deseja enviar um dado ao IP local.

2. Conforme ilustrado na Parte 1 da simulação, o pacote tem tamanho de cinco *phits*, aqui informado pelo sinal de saída do roteador *size_int_O22(4)*.

3. Da mesma forma, o sinal *data_int_O_22(4)* mostra os *phits* sendo enviados para o IP local. Nota-se que o pacote que entrou no roteador 2x0 é o mesmo que saiu do roteador 2x2.

4. O sinal *ack_nack_int_I22(4)* sinalizando um *ack*, informa à porta Local do roteador que o IP local tem condições de receber o dado a ser enviado. Neste instante, inicia-se a transferência.

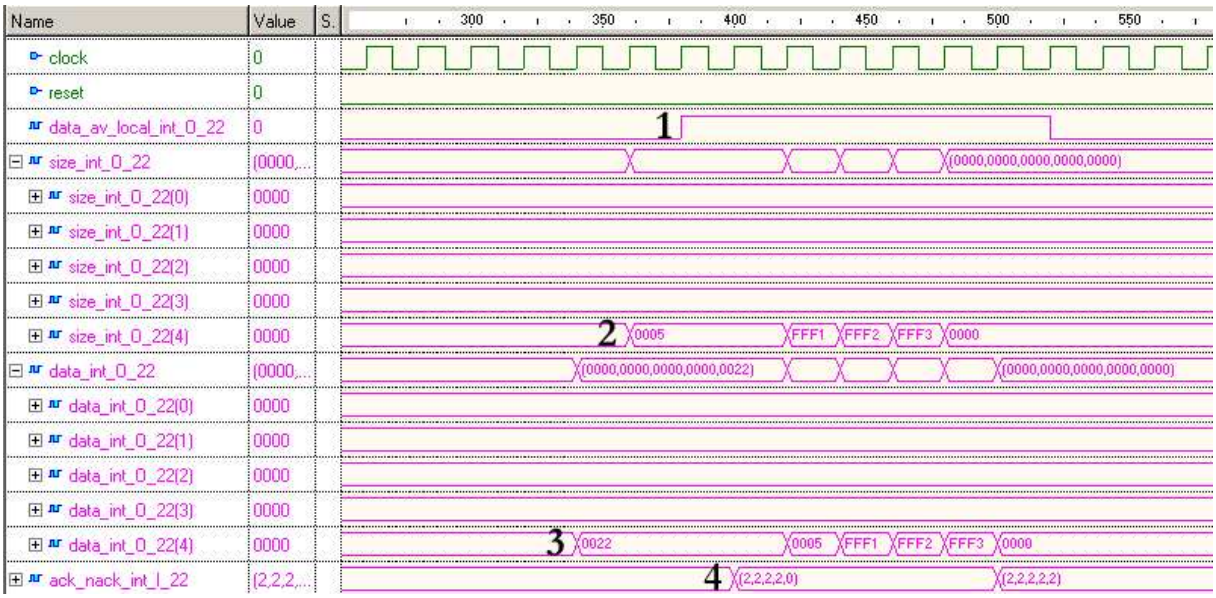


Figura 49 – Exemplo de resultado parcial de simulação de uma NoC 3x3 - Parte 2.

Ilustrou-se através desta simulação da NoC que um dado pacote trafega na rede de forma correta sem perda de *phits* ou modificação nos mesmos, utilizando-se de um caminhos mínimos disponíveis na rede. Também se ilustra que os *wraparound* da rede toro ajudam a rotear os pacotes com menos *hops*, visto que em uma rede malha de mesmas dimensões e com a mesma origem e destino, o dado deveria ser roteado para o roteador 2x1 primeiramente antes de ser roteado para o seu destino, no roteador 2x2.

6.3 Conclusões do Capítulo

As Seções 6.1 e 6.2 descreveram as modelagens abstrata e concreta da NoC Mercury em algum nível de detalhe. O processo de validação destas modelagens foi aqui apenas esboçado. Na realidade, uma extensa quantidade de casos foi gerada e simulada em ambos os modelos, TL

e RTL. Várias falhas sutis foram encontradas apenas após a utilização de casos complexos de múltiplas transmissões simultâneas de pacotes, alguns destes casos sendo descritos no Capítulo 9 desta Dissertação. Devido ao grande volume de informações geradas nas simulações, foi proposta e implementada uma ferramenta para auxiliar a compilação de resultados no nível de abstração TL, conforme será descrito no Capítulo 7.

Considera-se o projeto da rede Mercury suficientemente validado para servir como base de uma NoC real, uma vez que simulações TL e RTL extensas são realizadas sem erros, e que a rede, foi prototipada em *hardware*, gerando resultados de operação absolutamente corretos conforme será descrito no Capítulo 8.

7 Ferramenta de Apoio ao Projeto da NoC Mercury

A ferramenta Júpiter, apresentada neste Capítulo, foi desenvolvida dentro do contexto deste trabalho para dar apoio à geração e validação de NoCs Mercury. Ela é composta de dois módulos denominados **Gerador Júpiter** e **Analisador Júpiter**. O primeiro tem como funcionalidade gerar descrições de redes Mercury a partir de um conjunto de parâmetros selecionados pelo usuário. O gerador Júpiter pode produzir descrições nos dois níveis de abstração em que a rede foi modelada, conforme descrito no Capítulo 6, TL (em SystemC) e RTL (em VHDL). A segunda ferramenta foi desenvolvida para dar apoio ao processo de validação manual dos resultados gerados na ferramenta Modelsim após simulação da NoC em nível TL. Para validação de descrições RTL usa-se formas de onda geradas diretamente pela ferramenta Modelsim.

7.1 Gerador Júpiter

O módulo Júpiter responsável por gerar redes Mercury produz descrições da interconexão entre roteadores e a própria descrição dos roteadores da NoC além da interface externa da rede. Ele foi desenvolvido para facilitar o trabalho mecânico de descrição da rede empregado para geração de NoCs Mercury. Além disto, o gerador Júpiter possui flexibilidade, permitindo que sejam parametrizadas algumas informações antes da geração da rede, hoje incluindo:

- o nível de abstração da descrição;
- as dimensões da rede;
- o algoritmo de roteamento;
- o tamanho do *phit*;
- o tamanho das filas centralizadas A,B,C.

Na opção "Nível de abstração" é informado se a ferramenta irá gerar uma NoC Mercury em SystemC TL ou VHDL RTL. Caso seja escolhida a primeira opção, os parâmetros "Tamanho do

phit" e "Tamanho da fila" serão desconsiderados. No caso da escolha da segunda opção (VHDL RTL), o tamanho do *phit* deve ser escolhido entre as opções disponíveis que variam entre 8 e 64 *bits* e o tamanho da fila entre os valores de 4 a 32 *phits*.

No parâmetro "dimensões da rede" deve-se informar o número de roteadores desejado nas dimensões X e Y da rede a ser gerada. Assume-se uma topologia toro 2D regular e totalmente simétrica. As dimensões da NoC Mercury sempre estão de acordo com os eixos cartesianos, ou seja, a coordenada X cresce para a direita e a coordenada Y cresce para cima. Os valores na ferramenta para essa opção podem variar de 3 até 255, para ambas coordenadas. Atualmente, a escolha do algoritmo de roteamento está restrita a apenas uma opção, CG1 de Cypher e Gravano [17], devido a ele ser o único algoritmo implementado neste trabalho. Uma possibilidade de trabalho futuro inclui implementar sobre o mesmo roteador o segundo algoritmo proposto em [17] sem alterações significativas do *hardware*.

Quando se seleciona o tamanho do *phit* pode-se automaticamente estar restringindo o tamanho da NoC nas coordenadas X e Y. Isto ocorre porque, no caso da utilização de CG1, quando existe comunicação entre roteadores ou IP local, são enviadas junto ao pacote as coordenadas XY do roteador destino. Por este motivo, a parametrização da rede pode ser limitada pelo tamanho do *phit*.

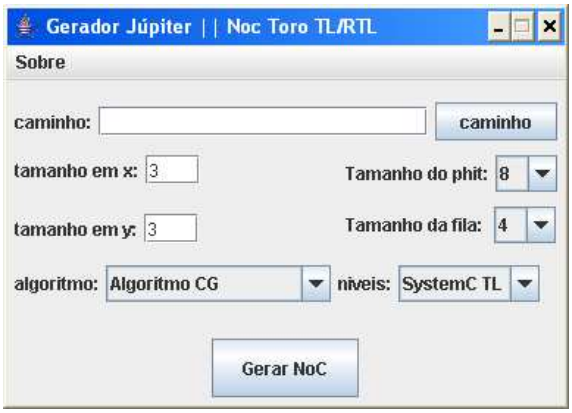


Figura 50 – Tela inicial do Gerador Júpiter

Por exemplo, na implementação de CG1 realizada neste trabalho, o endereço destino (posição XY) do dado a ser enviado, deve estar contido na metade do tamanho de um *phit*. Portanto, caso o *phit* escolhido seja de tamanho 16 *bits*, o endereçamento dos roteadores da rede deve ser passível de representação utilizando 8 *bits* para as coordenadas X e Y juntas. Com isso, se tem 4 *bits* para coordenada X e 4 *bits* para Y. Desta forma, utilizando-se de um *phit* de 16 bits pode-se endereçar roteadores em até 16 posições em X e 16 posições em Y, ou seja, é viável se ter uma NoC de tamanho 16x16 com um *phit* de tamanho 16 *bits*. Na Figura 50, mostra-se a

janela inicial do Gerador Júpiter, totalmente desenvolvido em Java.

7.2 Analisador Júpiter

O processo de validação por simulação de uma NoC, mesmo de pequenas dimensões (3x3 ou 4x4), produz uma quantidade de dados que rapidamente pode se tornar inviável de gerenciar para extrair informações úteis sobre a corretude da implementação ou sobre o desempenho da rede. No caso de descrições RTL, dispõe-se de visualizadores de formas de ondas digitais que facilitam um pouco esta tarefa. Contudo, para simulação TL, os dados são gerados como resultados numéricos de tráfego de pacotes passando por roteadores e interfaces de rede. Assim, propõe-se aqui uma ferramenta de análise de tráfego da NoC Mercury, com a finalidade de permitir ao desenvolvedor efetuar análises específicas de certos aspectos do tráfego de uma simulação em SystemC em nível de transação. Estes aspectos são:

- a quantidade total de pacotes enviados pelos IPs Locais;
- a quantidade total de pacotes recebidos pelos IPs Locais;
- detalhes sobre o caminhamento de pacotes específicos pela rede;
- a verificação de quais pacotes chegaram ao destino dentre todos os enviados durante a sessão de simulação;
- a verificação de quais pacotes não chegaram ao destino dentre todos os enviados durante a sessão de simulação;

Em *testbenches* executados através da ferramenta Modelsim, 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 NoC. Para facilitar tais análises, foi proposto e implementado o Analisador Júpiter e foi definido um protocolo de comunicação entre a simulação da NoC e a ferramenta.

Para o funcionamento e análise de um tráfego através do Analisador Júpiter, necessita-se abrir um arquivo com um formato pré-definido. Esse arquivo é gerado ao longo da simulação da NoC Mercury TL realizada no Modelsim. O formato do arquivo contempla registros em formato texto de quatro tipos, assim definidos:

1. **Informação:** [I] [tamanho da NoC]
2. **Entrada:** [E] [roteador local] [porta entrada] [fila] [roteador destino]/[roteador local]
3. **Troca de fila:** [T] [roteador local] [porta entrada] [fila destino] [roteador destino]
4. **Saída:** [S] [roteador local] [porta entrada] [fila] [porta saída] [roteador destino]

As seguintes abreviações são definidas da seguinte forma:

- [I] identifica que essa linha possui informação sobre as dimensões da rede;
- [E] identifica que essa linha informa a entrada de um pacote em algum roteador;
- [T] identifica que essa linha informa uma troca de fila dentro de algum roteador;
- [S] identifica que essa linha informa a saída de um pacote de algum roteador;
- [tamanho da NoC]: tamanho da NoC em X e Y, 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.
- [roteador destino]: roteador final para o qual o pacote se destina;
- [roteador local]: roteador no qual o pacote se encontra atualmente;
- [fila]: fila na qual o pacote entrou no roteador;
- [fila destino]: fila na qual o pacote foi colocado;
- [porta entrada]: porta através da qual o pacote entrou no roteador;
- [porta saída]: porta que o pacote usou para sair do roteador.

As portas de entrada e de saída são identificadas numericamente. A porta Norte é representada pelo algarismo "0", a porta Sul por "1", a porta Leste por "2", a porta Oeste por "3" e a porta Local por "4". O mesmo estilo de representação é usado para as filas: A fila A é representada pelo algarismo "0", fila B por "1" a fila C por "2". O protocolo de entrada da ferramenta pode ser visualizado através do exemplo 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]
[E] [20] [2] [0] [20]
[T] [20] [2] [1] [20]
[T] [20] [4] [2] [20]
[S] [20] [4] [2] [4] [20]
...
```

As linhas explicitamente representadas no exemplo acima mostram o caminho de um pacote com origem no roteador 03 indo para o destino, no roteador 20 em uma rede com dimensões X = 3 e Y = 4, conforme pode-se visualizar na Figura 51. Essas linhas foram retiradas diretamente de um arquivo de saída gerado pela ferramenta Modelsim. Entre essas linhas existiam outras que foram ocultadas, para facilitar a leitura do caminhamento desse pacote.

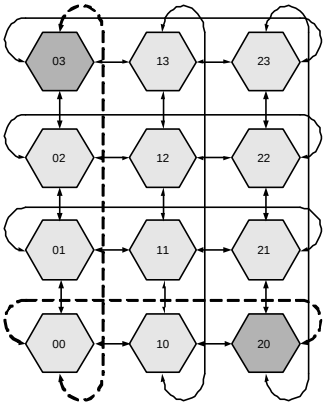


Figura 51 – Caminho percorrido pelo pacote demonstrado no exemplo, do nodo 03 ao nodo 20 em uma NoC 3x4.

O Analisador Júpiter demonstra o caminho decorrido por este pacote de uma forma um pouco diferente, porém mais clara, como demonstra a Figura 52.

Na Figura 53 é demonstrado as outras três opções da ferramenta Analisador Júpiter para este exemplo.

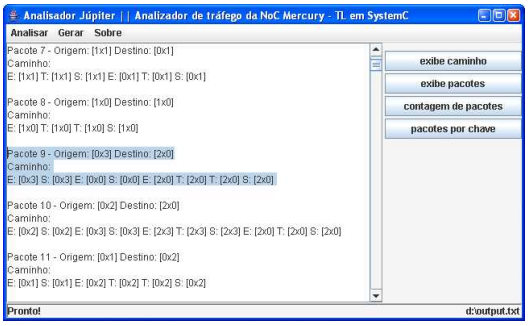


Figura 52 – Ferramenta Analisador Júpiter - Opção "exibe caminho" - Exemplo de saída gerada pelo Analisador Júpiter. O texto salientado mostra o caminho do pacote 9, seguindo o caminho entre a origem, no roteador 03 e o destino, no roteador 20. No caso, o pacote entrou na rede pelo roteador 03 (fi la A) e saiu deste, para a fi la A do roteador 00. Em seguida, o pacote saiu do roteador 00 para a fi la A do destino, roteador 20. Neste último, o pacote troca duas vezes de fi la (da A para a B e desta para a C). Após isto, o pacote sai da rede por este mesmo roteador.

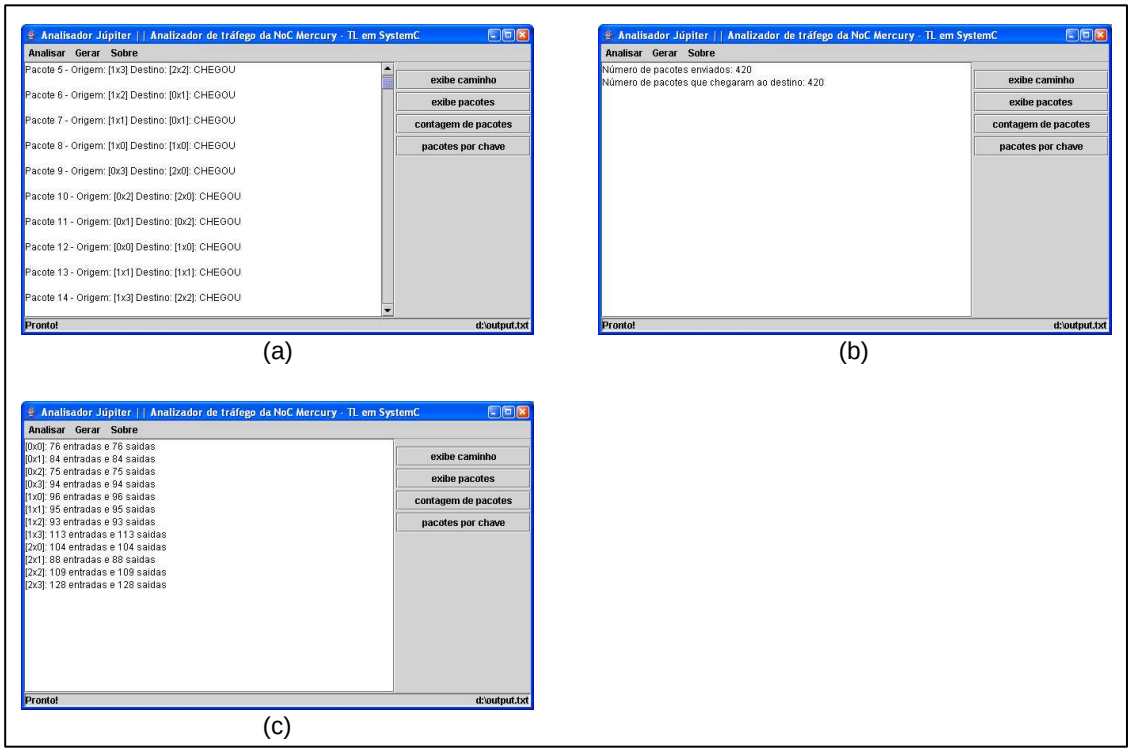


Figura 53 – Ferramenta Analisador Júpiter - Em (a) visualiza-se a opção "exibe pacotes" onde são listados todos os pacotes utilizados pela simulação demonstrando sua origem, destino e estado, sendo que este último informa se o pacote chegou ou não ao destino. Em (b) demonstra-se a opção "contagem de pacotes" onde a quantidade total de pacotes enviados e a quantidade total de pacotes que chegaram ao destino são informadas. Em (c) na opção "pacotes por chave" são listadas todas as chaves utilizadas na simulação, neste exemplo são 12 ao total, sendo exibido para cada uma delas a quantidade de pacotes que entraram e saíram por qualquer uma de suas portas.

8 Prototipação

Em Capítulos anteriores mostrou-se em detalhes o algoritmo de roteamento empregado neste trabalho, a proposta de arquitetura para o roteador e a modelagem TL e RTL da rede Mercury. Neste Capítulo, aborda-se a prototipação da rede Mercury em *hardware* a partir de sua descrição RTL em VHDL.

Para uma prototipação inicial foi utilizado um módulo serial [43] descrito em VHDL conectado à rede através das portas Locais, com a utilização de *wrappers*. Este módulo recebe informações pela porta serial do computador *bit a bit*, e as envia para a NoC em conjuntos de *bits*, como por exemplo 8, 16, 32, 64 *bits*. De forma inversa, o módulo serial também recebe dados da NoC em conjuntos de *bits* e os transmite para a porta serial *bit a bit*.

Para utilização do módulo serial, foi necessária a criação de um *wrapper* de entrada e um de saída. Estes *wrappers* foram descritos em VHDL e realizam a interligação entre a NoC e o módulo serial. Basicamente, os *wrappers* foram criados porque a velocidade de envio e recepção dos dados da NoC é muito mais alta do que a da porta serial. Desta forma, o *wrapper* de entrada espera todo o pacote chegar até ele, para então realizar o *handshake* com o roteador através da porta Local, enquanto que o *wrapper* de saída, espera que o módulo serial consuma todo o pacote enviado, para então deixar o roteador enviar outro pacote pela porta Local.

Serão descritos deste Capítulo quatro montagens utilizadas para prototipação. São elas:

- Dois roteadores interligados, como se estivessem em uma NoC Mercury 3x3;
- NoC Mercury 3x3, com o roteador 0x0 enviando dados para o 0x2;
- NoC Mercury 3x3, com o roteador 0x2 enviando dados para o 2x2;
- NoC Mercury 3x3, com o roteador 1x1 enviando dados para o 0x0.

Todas as montagens descritas acima foram prototipadas no FPGA Virtex-II Pro XC2VP30 da plataforma de prototipação XUPV2P da Digilent, InC (Figura 54). As sínteses lógica e física foram realizadas com a ferramenta ISE 7.1 da Xilinx [66].

Todas as montagens descritas neste Capítulo são preliminares e visam tão somente mostrar a correta operação de uma implementação de *hardware* obtida automaticamente a partir da descrição em VHDL RTL.

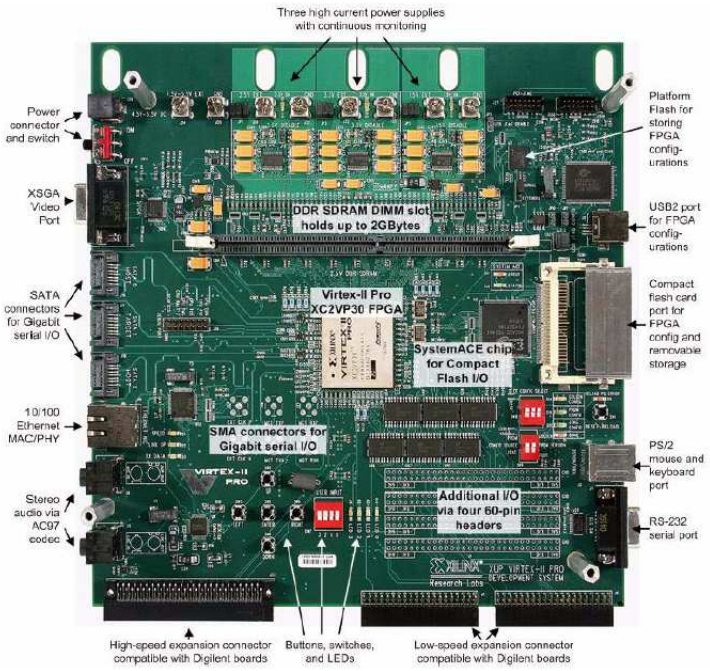


Figura 54 – Plataforma Xilinx University Program Virtex-II Pro Development System (XUPV2P) da Digilent, Inc.

8.1 Prototipação de dois roteadores

A primeira montagem prototipada interliga dois roteadores, correspondendo às posições 1x1 e 0x1, de uma NoC 3x3, conforme ilustra a Figura 55. Cada roteador prototipado possui filas de tamanho 8 *phits* sendo cada *phit* de 8 *bits*. Essa prototipação com dois roteadores foi realizada com o intuito de verificar o comportamento básico do roteador.

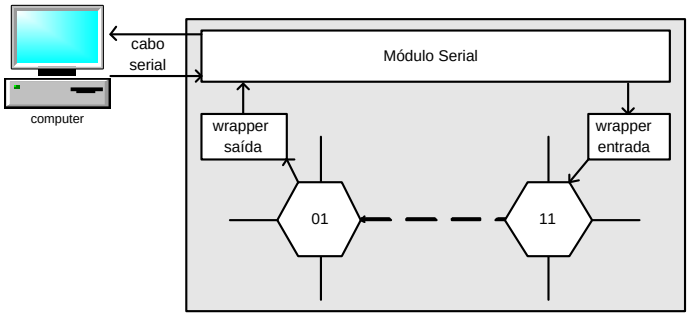


Figura 55 – Estrutura geral da montagem e caminhamento do pacote com dois roteadores comunicando-se. Os pacotes entram pelo *wrapper* de entrada no roteador 1x1 e saem pelo *wrapper* de saída no roteador 0x1.

Para validar a montagem prototipada "*módulo serial + wrappers + 2 roteadores*", realizou-se uma simulação através da ferramenta Active-HDL. Os passos desta simulação são descritos abaixo, onde a numeração tem correspondência na forma de onda de simulação da Figura 56.

1. Sinais *tx_data_in* e *tx_av_in* recebendo dados do módulo serial;
2. Após o recebimento de todo pacote do modulo serial, o *wrapper* de entrada envia os dados para a NoC. (Ver ampliação na forma de onda da Figura 57).
3. Os sinais *data_av_l_in*, *size_in*, *data_in*, *ack_nack_out* mostram o *wrapper* de saída recebendo os dados da NoC. (Ver ampliação na forma de onda da Figura 57).
4. Após receber o pacote, inicia-se a transferência para o módulo serial.

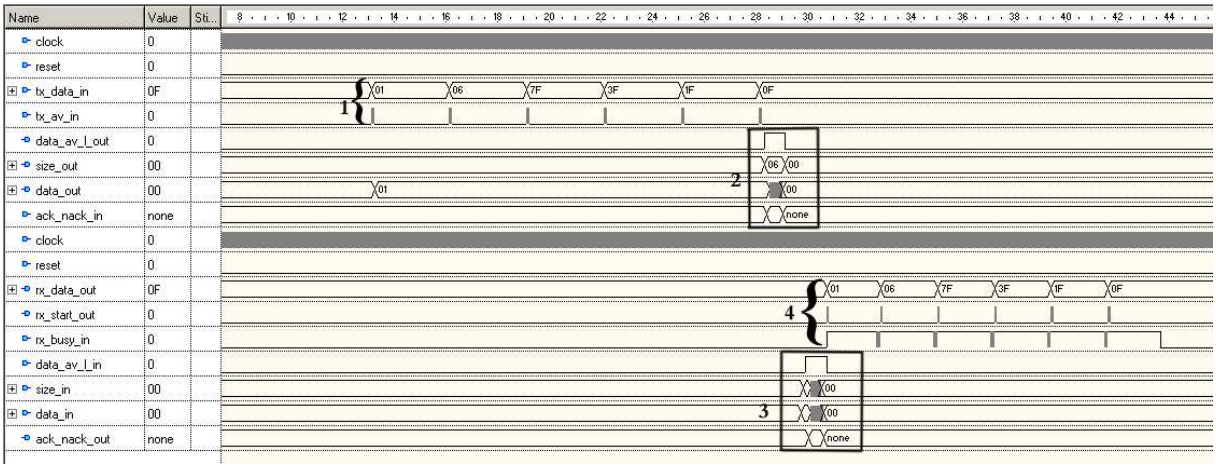


Figura 56 – Simulação na ferramenta Active-HDL da montagem prototipada com dois roteadores.

Abaixo são descritos os passos da ampliação de alguns sinais da simulação da Figura 56, onde a numeração tem correspondência na forma de onda da Figura 57.

1. Sinal indicando dado disponível na porta Local do roteador 1x1;
2. Tamanho do pacote a ser enviado para a porta Local do roteador 1x1;
3. Confirmação de aceite para envio do pacote pela porta Local do roteador 1x1;
4. Pacote sendo enviado pelo *wrapper* de entrada para a porta Local.
5. Sinal indicando dado disponível para o *wrapper* de saída, na porta Local do roteador 0x1;

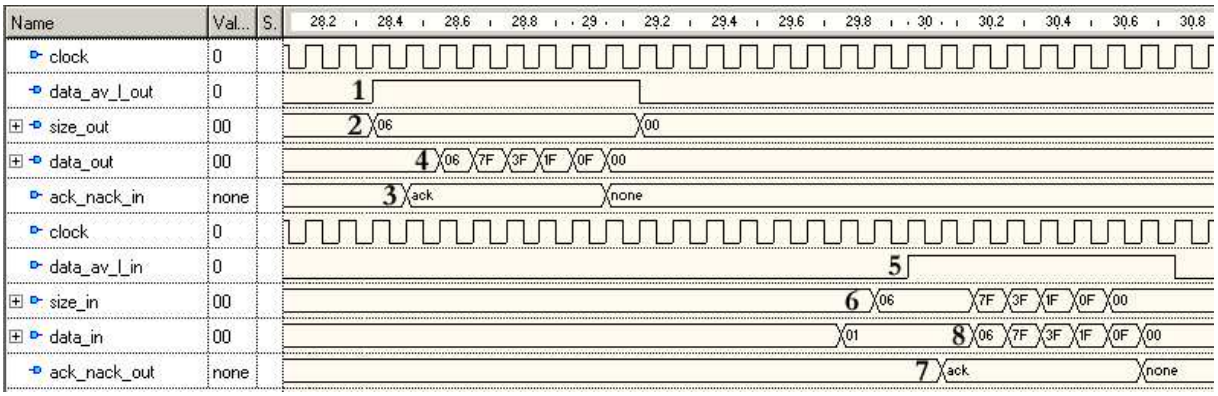


Figura 57 – Ampliação dos sinais da montagem prototipada com dois roteadores.

6. Tamanho do pacote a ser enviado ao *wrapper* de saída, pelo roteador 0x1;

7. Confirmação de aceite pelo *wrapper* de saída para receber o pacote;

8. Pacote sendo enviado da porta Local do roteador 0x1 para o *wrapper* de saída.

Nota-se que a comunicação entre os *wrappers* e a NoC se dá de forma muito mais rápida do que com o módulo serial. Por este motivo é importante a presença dos *wrappers* na montagem a ser prototipada. Analisando o pacote enviado pelo *wrapper* de entrada, através do sinal *tx_data_in* à porta Local do roteador 1x1, nota-se ser este o mesmo enviado pela porta Local do roteador 0x1 ao *wrapper* de saída no sinal *rx_data_out*. A partir daí pode-se concluir que a montagem funciona corretamente.

A montagem validada por simulação com somente dois roteadores, foi prototipada no dispositivo *XC2V1000* da plataforma V2MB1000 da Insight Memec, bem como no dispositivo *XC2VP30* da plataforma XUPV2P da Digilent, Inc. obtendo como resultados da síntese os valores monstrados na Tabela 7.

Tabela 7 – Resultado da síntese de dois roteadores para os dispositivos XC2V1000 e XC2VP30.

Dispositivo XC2V1000 - Plataforma V2MB1000			
	Utilizado	Disponível	% de Utilização
Número de Slices	2079	5120	40%
Número de LUTs	3730	10240	36%
Número de Flip Flops	1637	10240	15%
Dispositivo XC2VP30 - Plataforma XUPV2P			
Número de Slices	2056	13696	15%
Número de LUTs	3646	27392	13%
Número de Flip Flops	1713	27392	6%

Através da Figura 58, pode-se verificar o correto funcionamento em *hardware* da estrutura prototipada. Trata-se da interface gráfica de um aplicativo Java que interage com o protótipo de *hardware* através da interface serial de um computador hospedeiro do tipo PC. Todos os dados das janelas são representados em hexadecimal. O espaço na janela de cima da Figura é reservado para envio das informações através da porta serial do computador. Essas informações são enviadas para o módulo serial, descrito em VHDL, que após receber um conjunto de *bits* comunica-se com o *wrapper* de entrada para envio dos dados pela porta Local do roteador 1x1. O espaço na janela de baixo da Figura é onde são visualizados os pacotes recebidos pela porta serial. Eles são oriundos da porta Local do roteador 0x1, que através do *wrapper* de saída e do módulo serial comunicam-se com o computador hospedeiro.

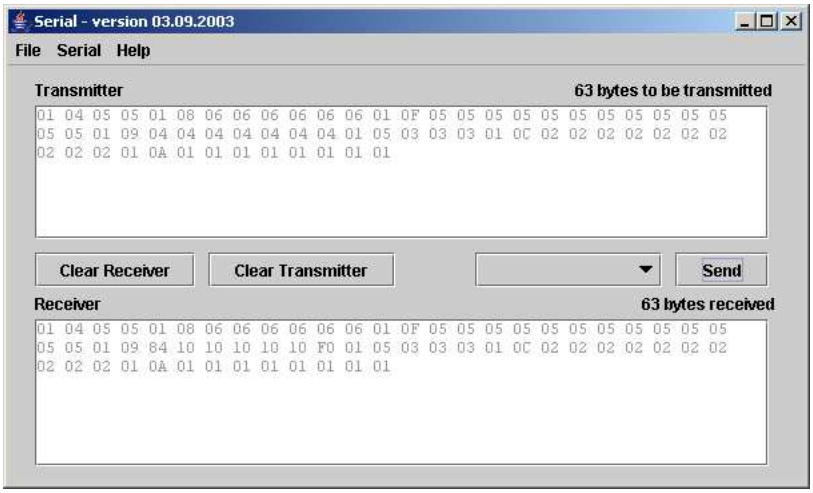


Figura 58 – Envio e recebimento de dados pela porta serial. Experimento com os roteadores 1x1 e 0x1. Cada pacote é composto de 3 partes. O primeiro *phit* é o endereço destino, valendo sempre 01 nesta montagem; o segundo *phit* é o tamanho do pacote, valendo 04 na primeira mensagem; o campo final é a carga útil do pacote, valendo 05 05 para a primeira mensagem. Na Figura, mostra-se a transmissão/recepção de 7 pacotes.

8.2 Prototipação Mercury 3x3

No segundo experimento foi prototipada uma NoC Mercury completa, com dimensões 3x3, tamanho de fila 16 *phits*, e cada *phit* de 8 *bits*. Na prototipação, utilizou-se o módulo serial e os *wrappers*, da mesma forma que na prototipação descrita na Seção 8.1. Três protótipos foram realizados para validar os diferentes caminhos percorridos por diferentes pacotes. Devido à área ocupada pela rede Mercury 3x3, não foi possível utilizar o dispositivo XC2V1000 para

prototipação. Em seu lugar foi utilizado o dispositivo XC2VP30.

Para validar a estrutura a ser prototipada "*serial + wrappers + mercury 3x3*", realizou-se uma simulação através da ferramenta Active-HDL da mesma forma descrita na Seção 8.1. Após ser validada por simulação, a estrutura foi prototipada na plataforma XUPV2P, obtendo como resultado da síntese com a ferramenta LeonardoSpectrum [45], os valores mostrados na Tabela 8.

Tabela 8 – Resultado da síntese da Mercury 3x3, com fila 16 *phits* e *phits* de 8 *bits*, com *wrappers* de entrada e saída e módulo serial, no dispositivo XC2VP30 da plataforma XUPV2P.

XC2VP30 - Plataforma XUPV2P			
	Utilizado	Disponível	Porcentagem
Número de Slices	7847	13696	57.29%
Número de LUTs	15693	27392	57.29%
Número de Flip Flops	6027	29060	20.74%

8.2.1 Roteador 0x0 enviando pacotes para o roteador 0x2

Esta prototipação foi realizada com a NoC Mercury 3x3, onde o *wrapper* de entrada injeta pacotes no roteador 0x0 oriundos do módulo serial e o *wrapper* de saída recebe os pacotes enviados pela porta Local do roteador 0x2. Através da realização desta prototipação, pode-se testar o caminhamento de pacotes através de um canal *wraparound* no sentido vertical, conforme ilustra a Figura 59.

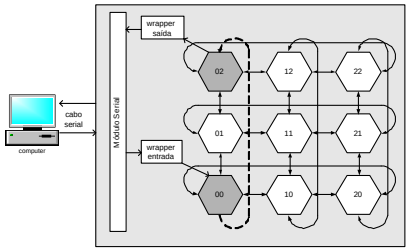


Figura 59 – Estrutura da montagem e caminhamento do pacote na primeira montagem da NoC Mercury 3x3. Os pacotes entram pelo *wrapper* de entrada no roteador 0x0 e saem pelo *wrapper* de saída no roteador 0x2. A linha tracejada indica o caminho exercitado pela prototipação.

Na Figura 60, pode-se verificar o correto funcionamento da NoC Mercury em *hardware*. Na janela de cima da Figura são enviados pacotes através do módulo serial para o *wrapper* e este, através da porta Local do roteador 0x0 injeta os mesmos na rede. Após percorrer o caminho do

roteador origem até o destino, os pacotes saem da rede na porta Local do roteador 0x2, onde está conectado o *wrapper* de saída, que envia as informações para o módulo serial. Desta forma pode-se visualizar na parte de baixo da Figura os pacotes saindo da rede.

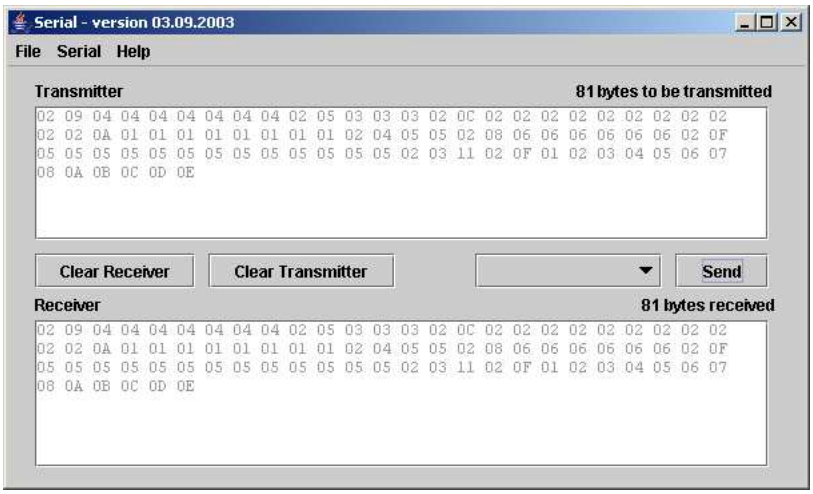


Figura 60 – Envio e recebimento de pacotes pela porta serial em uma NoC Mercury 3x3. Experimento com os roteadores 0x0 e 0x2. Mostra-se a transmissão/recepção de 9 pacotes.

8.2.2 Roteador 0x2 enviando pacotes para o roteador 2x2

Outra prototipação da Mercury 3x3 foi realizada com modificações na injeção e leitura de pacotes na rede. Neste experimento o *wrapper* de entrada injeta pacotes recebidos do módulo serial no roteador 0x2 e o *wrapper* de saída recebe dados enviados pela porta Local do roteador 2x2. Com essa prototipação pode-se verificar o caminhamento de pacotes através de um canal *wraparound* horizontal, conforme ilustra a Figura 61.

O funcionamento do canal *wraparound* horizontal na NoC Mercury 3x3 é validado através da visualização do funcionamento desta prototipação na Figura 62. Vários pacotes são enviados através da serial para a porta Local do roteador 0x2 da rede. Estes, após percorrer o caminho da origem até o destino, saem pela porta Local do roteador 2x2, conforme visualiza-se na Figura 62.

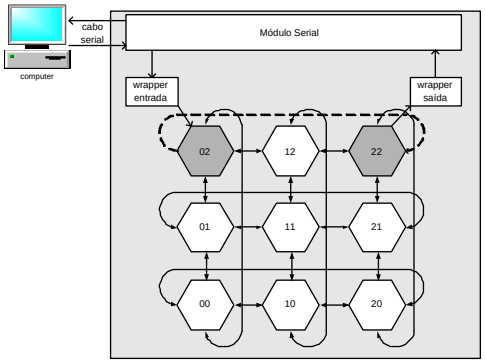


Figura 61 – Estrutura da montagem e caminhamento do pacote na segunda prototipação realizada da NoC Mercury 3x3. Os pacotes entram pelo *wrapper* de entrada no roteador 0x2 e saem pelo *wrapper* de saída no roteador 2x2. A linha tracejada indica o caminho exercitado pela prototipação.

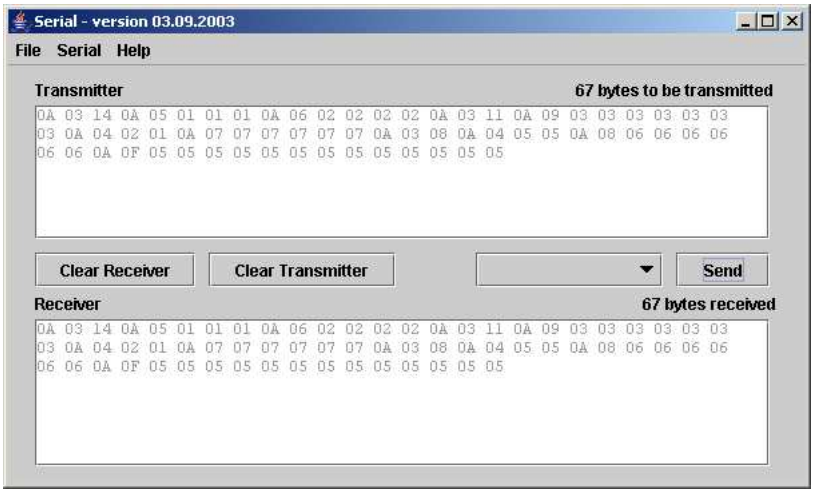


Figura 62 – Envio e recebimento de pacotes pela porta serial em uma NoC Mercury 3x3. Experimento com os roteadores 0x2 e 2x2. Mostra-se a transmissão/recepção de 11 pacotes.

8.2.3 Roteador 1x1 enviando pacotes para o roteador 0x0

O último experimento realizado tem forma similar às duas montagens anteriores. Este experimento visa verificar o caminhamento de um pacote passando por dois roteadores na rede. Desta forma, o *wrapper* de entrada injeta pacotes recebidos do módulo serial no roteador 1x1 e o *wrapper* de saída recebe os dados enviados pela porta Local do roteador 0x0. A Figura 63 ilustra a estrutura da prototipação utilizada neste experimento.

A validação do funcionamento desta prototipação é realizada através da verificação dos pacotes enviados e recebidos pela ferramenta Serial, conforme ilustra a Figura 64.

Com base nestes três experimentos prototipados na plataforma XUPV2P, pode-se considerar

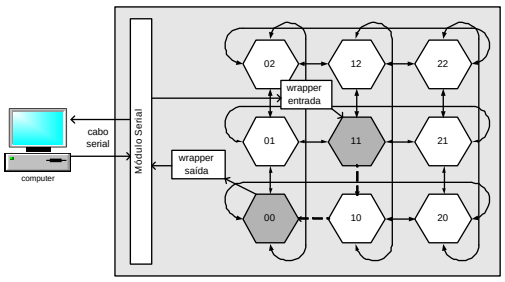


Figura 63 – Estrutura geral da montagem e caminhamento dos pacotes na Mercury 3x3, saindo do roteador 1x1 com destino ao 0x0. A linha tracejada indica o caminho exercitado pela prototipação.

a rede Mercury funcional tanto em nível de simulação, através dos *testbenchs* realizados no Capítulo 6, quanto em *hardware* através das prototipações realizadas com sucesso e descritas neste Capítulo.

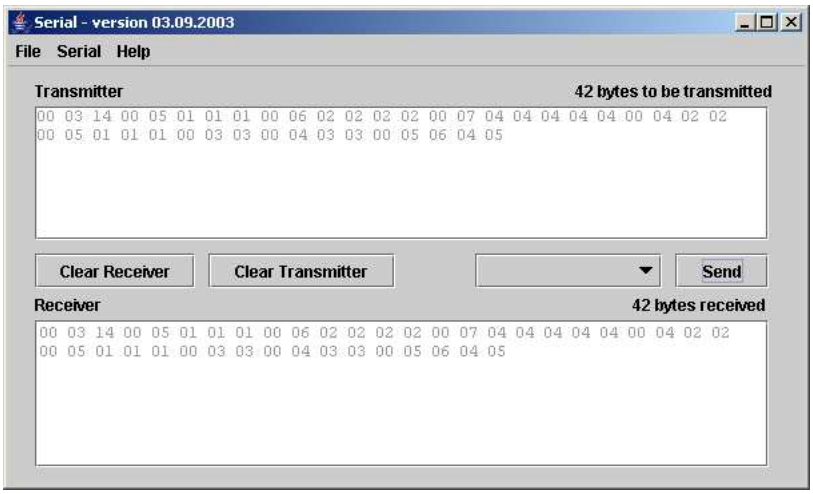


Figura 64 – Envio e recebimento de pacotes pela porta serial em uma NoC Mercury 3x3. Experimento com os roteadores 1x1 e 0x0. Mostra-se a transmissão/recepção de 9 pacotes.

8.3 Evolução das montagens

As montagens descritas neste Capítulo são preliminares, visando somente mostrar o correto funcionamento da rede Mercury quando prototipada. Uma evolução dos *wrappers* está sendo realizada, de modo a aumentar a funcionalidade dos mesmos. Pretende-se que os *wrappers* identifiquem para que roteador deve-se enviar os pacotes oriundos do PC. Desta forma, viabilizar-se-á validar todos os caminhos disponíveis na rede com uma única montagem.

9 Comparação Mercury x Hermes

Neste Capítulo são relatados um conjunto de experimentos preliminares de comparação entre a rede Mercury e a rede Hermes [46,48], também desenvolvida no âmbito do Grupo de Apoio ao Projeto de Hardware (GAPH). São analisados os quesitos de área, vazão e latência, sendo estes dois últimos somente de forma externa às redes. Os experimentos são realizados através da inserção de tráfego nas portas Locais dos roteadores de ambas NoCs e de uma posterior análise do tráfego de saída nas porta Locais.

A rede Hermes possui a topologia malha 2D bidirecional com modo de chaveamento *wormhole*, filas de entrada e lógica de arbitragem/controle central, compartilhada entre todas as portas do roteador. Nesta rede, os valores do *phit* e do *flit* são idênticos. Para comparação com a rede Mercury, escolheu-se a versão da rede Hermes com dois canais virtuais, que possui desempenho e área maiores que a versão sem canais virtuais. A seleção da configuração da rede Hermes deve-se ao fato desta ser considerada aquela com o melhor resultado em termos de custo-benefício para fins de área, vazão e latência, sendo então esta comparação a de pior caso para a rede Mercury.

A rede Hermes já vem sendo evoluída desde sua criação, há quatro anos. Desta forma, sua estrutura e seu código estão otimizados para corresponderem a uma menor área e um ótimo desempenho. A rede Mercury é uma proposta nova, descrita neste trabalho que ainda necessita otimização no código para melhorar seu tamanho de área ocupada e o seu desempenho.

9.1 Comparação de Área

Várias prototipações para o dispositivo FPGA Virtex-II 2V6000 foram realizadas através da ferramenta LeonardoSpectrum [45] para verificação e comparação do número de *Slices*, *Flip Flops* e *LUTs* ocupados pelas redes Mercury e Hermes. Parâmetros como tamanho de fila e tamanho de *phit* utilizados pelas NoCs foram parametrizados de forma distinta a cada prototipação entre os valores aceitáveis para ambas as redes. As configurações escolhidas para comparação de área, são dadas abaixo:

- NoC 3x3 - *Phit* de 8 *bits* variando tamanho da Fila entre 4, 8, 16 e 32 *phits*;
- NoC 3x3 - *Phit* de 16 *bits* variando tamanho da Fila entre 4, 8, 16 e 32 *phits*;
- NoC 3x3 - *Phit* de 32 *bits* variando tamanho da Fila entre 4, 8, 16 e 32 *phits*.

As Tabelas 9, 10 e 11 mostram os resultados obtidos após a síntese lógica das redes Mercury e Hermes. Cada Tabela mostra valores referentes à NoC com um valor diferente para o tamanho do *phit*. Na Tabela 9 é mostrado o resultado de área para NoCs com tamanho de *phit* 8 *bits*, na Tabela 10 de 16 *bits* e na Tabela 11 de 32 *bits*.

Tabela 9 – Comparação de área entre as NoCs Mercury e Hermes, para *phit* de 8 *bits*.

Dispositivo 2V6000									
Fila (em *phits*)	**4**	**8**	**16**	**32**	**Disp.**	**4**	**8**	**16**	**32**
Hermes 3x3									
Slices	4132	4103	4274	4417	**33792**	12.23%	12.14%	12.65%	13.07%
LUTs	8263	8206	8548	8833	**67584**	12.23%	12.14%	12.65%	13.07%
Flip Flops	2163	2220	2334	2448	**70056**	3.09%	3.17%	3.33%	3.49%
Mercury 3x3									
Slices	5862	6685	7941	10020	**33792**	17.35%	19.78%	23.50%	29.65%
LUTs	11723	13370	15881	20039	**67584**	17.35%	19.78%	23.50%	29.65%
Flip Flops	2826	3744	5526	9063	**70056**	4.03%	5.34%	7.89%	12.94%

Tabela 10 – Comparação de área entre as NoCs Mercury e Hermes, para *phit* de 16 *bits*.

Dispositivo 2V6000									
Fila (em *phits*)	**4**	**8**	**16**	**32**	**Disp.**	**4**	**8**	**16**	**32**
Hermes 3x3									
Slices	5516	5430	5801	5943	**33792**	16.32%	16.07%	17.17%	17.59%
LUTs	11031	10860	11601	11886	**67584**	16.32%	16.07%	17.17%	17.59%
Flip Flops	2619	2619	2790	2904	**70056**	3.74%	3.74%	3.98%	4.15%
Mercury 3x3									
Slices	7913	9263	11261	15297	**33792**	23.42%	27.41%	33.32%	45.27%
LUTs	15825	18525	22521	30594	**67584**	23.42%	27.41%	33.32%	45.27%
Flip Flops	3690	5499	9009	16029	**70056**	5.27%	7.85%	12.86%	22.88%

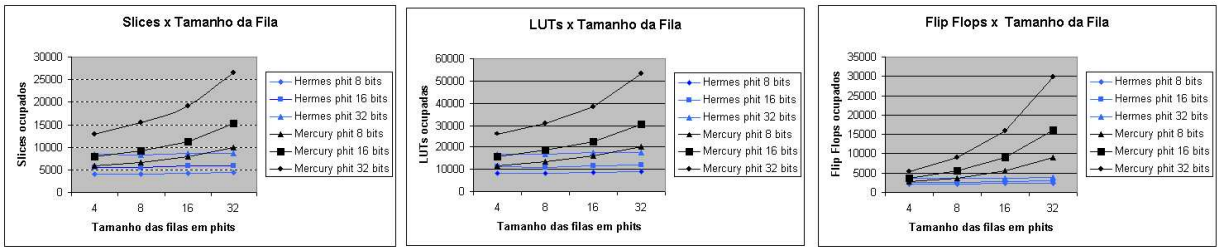
Analisando as tabelas nota-se uma pequena diferença de área entre as duas redes quando comparadas com pequeno tamanho de fila (4-8 *phits*). Conforme se aumenta a profundidade das filas, a diferença de área entre as NoCs aumenta também. Para filas de 32 *phits* a rede Mercury ocupa mais do que o dobro de área do que a rede Hermes. Acredita-se que o aumento na diferença das áreas se deve à utilização de LUTRAMs na rede Hermes, enquanto que o mesmo não ocorre na rede Mercury comparada.

Tabela 11 – Comparação de área entre as NoCs Mercury e Hermes, para *phit* de 32 *bits*.

Dispositivo 2V6000									
Fila (em *phits*)	**4**	**8**	**16**	**32**	**Disp.**	**4**	**8**	**16**	**32**
Hermes 3x3									
Slices	8400	8315	8685	8742	**33792**	24.86%	24.61%	25.70%	25.87%
LUTs	16800	16629	17370	17484	**67584**	24.86%	24.60%	25.70%	25.87%
Flip Flops	3553	3553	3724	3838	**70056**	5.07%	5.07%	5.32%	5.48%
Mercury 3x3									
Slices	12951	15408	19188	26559	**33792**	38.32%	45.60%	56.78%	78.59%
LUTs	25901	30815	38375	53117	**67584**	38.32%	45.60%	56.78%	78.59%
Flip Flops	5445	9009	15975	29772	**70056**	7.77%	12.86%	22.80%	42.50%

O principal fator levando a maiores áreas ocupadas pela rede Mercury em qualquer situação, se deve ao processo de roteamento utilizado pela rede. Conforme descrito neste trabalho, a rede Mercury possui um roteamento adaptativo complexo, onde os pacotes podem ser roteados por qualquer um dos caminhos mínimos disponíveis, enquanto que a rede Hermes utiliza um algoritmo determinístico mínimo, o mais eficiente em termos de área. O grau de complexidade do algoritmo utilizado neste trabalho, torna o roteamento da rede mais eficiente às custas de uma maior área do roteador.

A Figura 65 mostra graficamente as informações de área ocupadas em *Slices*, *LUTs* e *Flip Flops* dispostas pelas Tabelas 9, 10 e 11.



(a) (b) (c)

Figura 65 – Em (a) mostra-se os *slices* ocupados, em (b) as *LUTs* ocupadas e em (c) os *Flips Flops* ocupados em função do tamanho da fi la.

9.2 Comparação de Desempenho

A comparação de desempenho entre as redes Mercury e Hermes foi realizada baseada nos quesitos vazão e latência. Esta comparação usou experimentos onde é inserido e lido tráfego nas

portas Locais dos roteadores. Para tanto, foram utilizadas ferramentas propostas em [63]. Para geração dos arquivos de tráfego a ser injetado na rede, foi utilizada a ferramenta *Traffic Mbps* [46, 48]. Esta ferramenta é parametrizável pelo usuário, permitindo a este definir grandezas da rede tais como: (*i*) as dimensões da rede; (*ii*) o tamanho do *phit*; (*iii*) a técnica de controle de fluxo. Algumas grandezas relacionadas ao tráfego podem e devem ser informadas pelo usuário, incluindo: (*i*) o destino dos pacotes; (*ii*) o número de pacotes que cada roteador deve enviar a rede; (*iii*) o tamanho do pacote em *phits* e (*iv*) a carga oferecida à rede. Algumas destas grandezas podem estar associadas a variáveis probabilísticas, permitindo a geração de tráfego probabilístico.

Para simulação das redes, foi utilizado um ambiente de simulação desenvolvido dentro do GAPH. Neste ambiente, a rede descrita em VHDL é estimulada com o tráfego gerado pela ferramenta *Traffic Mbps*. Este ambiente injeta pacotes pela porta Local de entrada dos roteadores e aguarda a chegada dos mesmos pelas portas Locais de saída. Ao injetar e receber pacotes, a ferramenta cria arquivos com marcas de tempo (em inglês, *timestamps*), informando o tempo de entrada e saída dos pacotes na rede. Após simular as redes Hermes e Mercury, utilizou-se a ferramenta de medição de tráfego denominadas *TrafficAnalysis* para verificar a vazão e a latência média que os pacotes inseridos na rede obtiveram. Como o tráfego injetado em ambas as redes é o mesmo, pode-se traçar um comparativo inicial dos quesitos analisados.

Até o momento, foram realizados quatro estudos de caso comparando as duas NoCs, descritos a seguir. Para todos os casos, utilizou-se redes com dimensões 3x3, filas de profundidade 32 *phits*, tendo cada *phit* largura de 16 *bits*. A rede Hermes utilizada na comparação possui dois canais virtuais e utiliza o algoritmo de roteamento determinístico XY.

Nos três estudos de caso foram realizados 11 experimentos, onde a taxa de carga oferecida para cada roteador se modifica em cada um deles. Os valores de carga máxima submetidos à rede foram 5, 10, 15, 20, 30, 40, 50, 60, 70, 80 e 90%. Considerando que a largura dos canais de transmissão de dados é de 16 *bits* e que a freqüência da rede é de 50Mhz (período de ciclo de relógio de 20*ns*), pode-se afirmar que a taxa máxima de injeção (100%) é de 800Mbps. Conseqüentemente, sabe-se que as taxas de carga oferecidas à rede foram de 40, 80, 120, 160, 240, 320, 400, 480, 560, 640 e 720Mbps.

9.2.1 Estudo de Caso 1

No primeiro Estudo de Caso, os roteadores enviam dados a uma taxa constante, utilizando padrão de tráfego denominado "*target* 22", onde todos roteadores da rede enviam pacotes para

o mesmo destino (neste caso, o roteador 2x2). Em ambas as NoCs foram injetados os mesmos 1000 pacotes com tamanho de 15 *phits*, totalizando 9000 pacotes ao todo em cada rede. Na Tabela 12 pode-se visualizar um resumo do comportamento nos quesitos vazão e latência das NoCs em questão.

Tabela 12 – Estudo de Caso 1 - 9000 pacotes - *Phit* 15 *bits* - Alvo roteador 2x2.

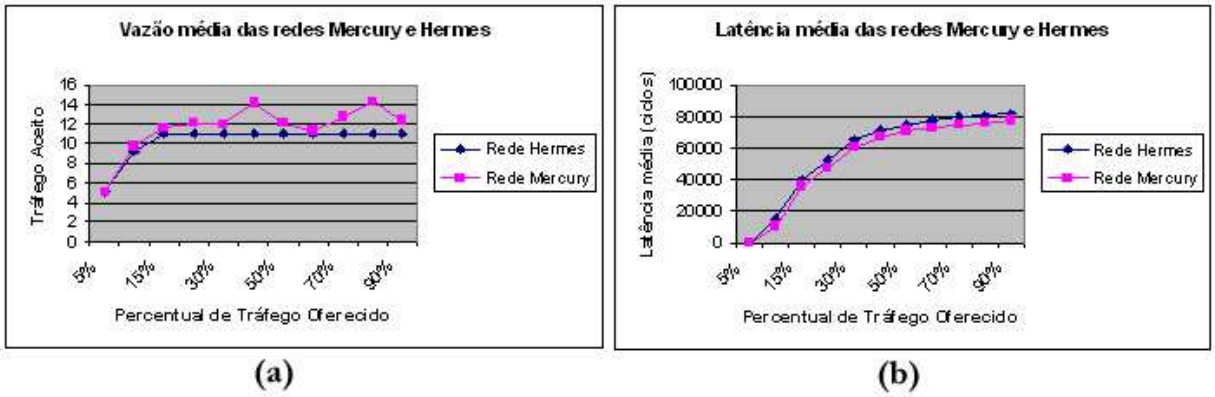
	Hermes		**Mercury**	
Carga Oferecida	**Vazão**	**Latência (ciclos)**	**Vazão**	**Latência (ciclos)**
5%	5.00	101	5.00	105
10%	9.33	15086	9.70	10594
15%	10.91	40060	11.65	35569
20%	10.91	52548	12.03	48057
30%	10.96	65036	11.95	60544
40%	10.92	71281	14.08	66788
50%	10.91	75026	12.07	70534
60%	10.91	77524	11.36	73032
70%	10.91	79308	12.70	74816
80%	10.91	80646	14.31	76154
90%	10.91	81687	12.34	77195

Devido ao tipo de tráfego oferecido à rede (9000 pacotes enviados ao roteador 2x2), o ponto de saturação aparece com pouca carga oferecida, conforme ilustra a Figura 66. Analisando a Tabela 12, nota-se que tanto as duas redes saturam para valores similares da carga oferecida, estando estes entre 10 e 15% da carga máxima. A rede Mercury em todas as cargas oferecidas proporciona uma maior vazão (tráfego aceito) em relação à rede Hermes. Desconsiderando a primeira medição com carga de 5%, a rede Mercury sempre mantém um valor menor do que a Hermes no quesito latência, sendo que a diferença de valores entre as redes segue praticamente a mesma, independente da carga oferecida.

Para este estudo de caso, a rede Mercury se mostrou mais eficiente, pois conseguiu proporcionar uma maior vazão e uma menor latência aos pacotes na rede.

9.2.2 Estudo de Caso 2

O segundo estudo de caso envolve roteadores enviando dados a um taxa constante e utilizando padrão de tráfego "*target* 22" novamente. Ao invés de pacotes com tamanho de 15 *phits* utilizados no Estudo de Caso 1, utiliza-se aqui pacote de tamanho 20 *phits*. Tanto na rede Mercury quanto na Hermes foram injetados desta vez 200 pacotes (de mesmo conteúdo nas duas



(a) (b)

Figura 66 – Gráfi co representando a latência e a vazão média da rede para o Estudo de Caso 1.

redes), totalizando 1800 pacotes em cada uma das redes. A Tabela 13 ilustra os resultados de desempenho obtidos da simulação.

Tabela 13 – Estudo de Caso 2 - 1800 pacotes - *Phit* 20 *bits* - Alvo roteador 2x2.

	Hermes		**Mercury**	
Carga Oferecida	**Vazão**	**Latência (ciclos)**	**Vazão**	**Latência (ciclos)**
5%	5.02	130	5.02	130
10%	9.41	3594	9.95	1722
15%	11.11	10183	11.23	8355
20%	11.28	13468	15.50	11672
25%	11.29	15452	12.47	13662
30%	11.28	16779	12.95	14989
40%	11.28	16779	12.95	14989
50%	11.29	19432	12.47	17642
60%	11.29	19432	12.47	17642
70%	11.29	20570	12.43	18779
80%	11.29	20570	12.43	18779
90%	11.29	21202	20.36	19411

Novamente, neste estudo de caso, a rede Mercury consegue prestar um melhor serviço, com menores médias de latência quando comparadas às médias da rede Hermes (conforme ilustra Figura 67 (b)). Acompanhando no gráfico as medidas de latência a vazão média também nota-se serem estas sempre maiores na rede Mercury, independente da quantidade de carga oferecida à rede. O ponto de saturação da rede Hermes inicia com um pouco menos de tráfego oferecido a rede, próximo aos 15%, enquanto que a saturação na rede Mercury inicia com aproximadamente 20% de carga oferecida. A Figura 67, ilustra os dados contidos na Tabela 13.

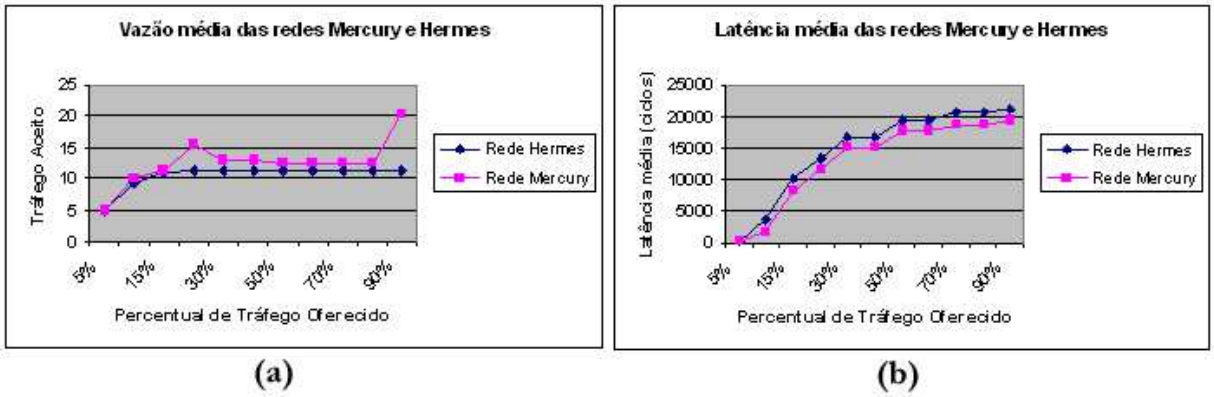


Figura 67 – Gráfi co representando a latência e a vazão média da rede para o Estudo de Caso 2.

9.2.3 Estudo de Caso 3

No terceiro e último estudo de caso, modifica-se novamente o tamanho dos pacotes. Desta vez são injetados 200 pacotes em cada roteador, todos tendo como destino o roteador 2x2, com tamanho de 25 *phits*. Ao todo, são injetados 1800 pacotes idênticos nas duas redes. A Tabela 14 mostra os resultados obtidos através da simulação, para os quesitos latência média e vazão.

Tabela 14 – Estudo de Caso 3 - 1800 pacotes - *Phit* 25 *bits* - Alvo roteador 2x2.

	Hermes		**Mercury**	
Carga Oferecida	**Vazão**	**Latência (ciclos)**	**Vazão**	**Latência (ciclos)**
5%	5.02	149	5.02	155
10%	9.88	1420	9.96	1249
15%	11.73	9586	11.84	9541
20%	12.62	13683	13.32	13687
30%	12.66	17830	22.02	17833
40%	12.68	19902	20.87	19906
50%	12.66	21146	26.93	21149
60%	12.65	21975	15.51	21979
70%	12.65	22568	13.78	22571
80%	12.65	23012	13.37	23015
90%	12.65	23357	12.92	23361

Neste terceiro estudo de caso, as redes Mercury e Hermes atingem praticamente a mesma média de latência, enquanto que a rede Mercury tem uma vazão ligeiramente maior. A Figura 68, ilustra os dados contidos na Tabela 14. Nota-se em (b) que as latências médias de ambas as redes são praticamente idênticas, demonstrando assim um comportamento similar no tempo de entrega dos pacotes.

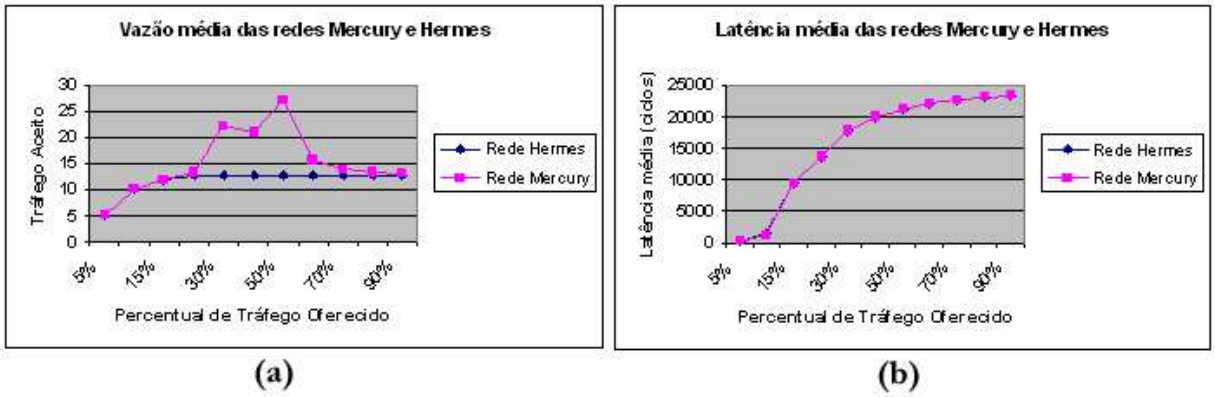


Figura 68 – Gráfi co representando a latência e a vazão média da rede para o Estudo de Caso 3.

9.3 Conclusão

Através dos três Estudos de Caso realizados pôde-se observar que a rede Mercury para o tipo de tráfego injetado consegue uma leve vantagem nas médias de vazão e na latência global. Atualmente, modificações estão sendo realizadas no ambiente que injeta e lê os pacotes da rede Mercury, visando viabilizar a inserção de pacotes com destinos aleatórios (de qualquer origem para qualquer destino na rede), sendo este um outro padrão de tráfego interessante de se comparar quando aplicado a NoCs.

Acredita-se que para pacotes de tamanho pequeno, devido à utilização do modo *virtual cut-through*, a rede Mercury obtenha um melhor desempenho, enquanto que para os pacotes de tamanho grande sua utilização não seja interessante. Quanto maior o tamanho do pacote que trafega pela rede, maior deve ser o tamanho das filas na rede Mercury, resultando acréscimos de área que rapidamente podem se tornar proibitivos, conforme demonstrado na Seção 9.1 deste Capítulo.

Com base nesta primeira abordagem da comparação das duas redes desenvolvidas pelo GAPH, acredita-se que a rede Mercury é uma alternativa para ser utilizada em redes de grandes dimensões, diferente do que foi avaliado neste Capítulo. Acredita-se que quanto maior as dimensões da rede, maior se torna a diferença do número de *hops* a ser percorrido em ambas as redes. Por exemplo, em uma rede de dimensões 8x8, para se rotear um pacote do roteador 0x0 para o 7x7 em uma rede com topologia toro o caminho mínimo possui dois *hops count*, enquanto que em uma rede malha um caminho mínimo possui catorze *hops count*, acentuando assim a diferença de latências entre as duas redes.

Outro fator importante a ser mencionado aqui é que a utilização de pacotes pequenos faz com que a área ocupada pela rede Mercury seja próxima da ocupada pela Hermes, tornando

assim sua utilização uma alternativa viável. De forma geral, acredita-se que para utilização em redes de grandes dimensões com pacotes de poucos *phits*, a rede Mercury é a mais indicada em relação a Hermes. Esta por sua vez se torna mais indicada para redes de pequenas dimensões com pacotes de grandes dimensões.

Está sendo construído um novo cenário de comparação entre as duas redes, onde a dimensão destas é 8x8. Neste novo cenário, espera-se que a rede Mercury obtenha maior desempenho em relação à rede Hermes. O cenário do experimento realizado e apresentado neste Capítulo é considerado o de pior caso, visto que para uma rede 3x3 a diferença da média dos *hop counts* é praticamente desprezível, enquanto que em uma rede de maiores dimensões a *hop count* médio da rede Mercury é bem menor do que o da rede Hermes.

10 Considerações Finais

Este trabalho apresentou as etapas de proposta de arquitetura, modelagem, prototipação e avaliação de uma rede intra-chip com topologia toro e algoritmo de roteamento mínimo adaptativo, a NoC Mercury. Adicionalmente, implementou-se um conjunto inicial de ferramentas de apoio ao projeto para esta NoC.

O levantamento do estado da arte em propostas de NoCs revelou que a topologia toro, embora tendo grande respaldo na área de redes de interconexão para sistemas multiprocessados, ainda é pouco explorada como topologia de redes intra-chip, malgrado sua aparente adequabilidade para este contexto. Isto constituiu uma motivação para o presente trabalho. O estudo de redes de interconexão com topologia toro empregadas em sistemas multiprocessados levou à seleção de um dentre os numerosos algoritmos de roteamento disponíveis na literatura para implementação em ambientes intra-chip. Com a escolha do algoritmo de roteamento, chegou-se a uma proposta de arquitetura específica para um roteador de rede intra-chip que suporte o algoritmo, o que constitui uma primeira contribuição deste trabalho, uma vez que a proposta de Cypher e Gravano [17] não incluía a definição de tal arquitetura. A proposta de arquitetura incluiu um diagrama de blocos definindo a estrutura geral do roteador, a definição das interfaces roteador-roteador e roteador-IP local, bem como a distribuição das partes da funcionalidade do algoritmo sobre os elementos da arquitetura, a partir da propostas de *árbitros* de entrada e de saída. Implícito na arquitetura do roteador e do algoritmo adotado encontra-se a proposta de estrutura da rede de interconexão como um todo.

A partir das propostas de roteador e rede definiu-se realizar a modelagem destes em dois nível de abstração, TL e RTL. A modelagem nestes níveis de abstração constitui uma segunda contribuição do trabalho. Intuitivamente, considera-se que modelar em dois níveis distintos de abstração trouxe como mérito a diminuição do tempo de validação de descrições menos abstratas, produzindo rapidamente resultados corretos na simulação RTL e na prototipação.

A NoC Mercury foi prototipada com sucesso em FPGAs, efetivando a prova de conceito do projeto como um todo, o que constitui uma terceira contribuição do trabalho.

Visando reduzir o trabalho de geração de descrições da NoC Mercury TL e RTL, bem como facilitar o processo de validação de descrições TL, implementou-se ferramentas de apoio ao projeto, o gerador Júpiter e o analisador Júpiter, respectivamente. Isto constitui a quarta contri-

buição do trabalho.

Finalmente, foram gerados resultados iniciais de comparação da NoC Mercury com uma NoC no estado da arte, a NoC Hermes. Estes resultados iniciais constituem uma quinta contribuição do presente trabalho.

A partir do trabalho desenvolvido, pode-se enumerar diversas direções para pesquisas futuras. A seguir, discute-se um conjunto de propostas de trabalhos futuros, apresentados em ordem de relevância decrescente para acrescentar contribuições ao presente trabalho.

No Capítulo 9 foram apresentados alguns estudos de caso de comparação entre a NoC Mercury e a NoC Hermes. Claramente este conjunto de estudos de caso é limitado. De fato, este conjunto não pôde ser mais abrangente devido a problemas na geração de resultados de simulação. Após realizar diversos experimentos com a NoC Mercury 3x3, passou-se a tentar simular uma versão 8x8 desta. Contudo, simulações nesta última rede apresentaram problemas quando a carga selecionada foi superior a 20%. O problema pode estar na rede ou no ambiente de simulação, e encontra-se em investigação. Descobrir onde reside este problema e solucioná-lo é hoje uma tarefa de alta prioridade.

Segundo, o Capítulo 9 mostrou que a taxa de crescimento da área da NoC Mercury excede em muito a taxa de crescimento de uma rede Hermes de dimensões similares. Buscar uma explicação e eventualmente otimizar a descrição da NoC Mercury para reduzir sua taxa de crescimento de área é um trabalho futuro importante. Um dos possíveis caminhos para se obter resultados melhores em FPGAs é otimizar o código VHDL buscando aproveitar a capacidade de inferir estruturas de memória específicas do FPGA durante a síntese lógica/física. Isto é de fato já aproveitado na rede Hermes.

Em terceiro lugar, uma análise mais aprofundada do desempenho da rede Mercury é necessária. O objetivo desta análise é prover explicações para o comportamento da rede, incluindo mostrar porque a simulação revelou algumas não linearidades de comportamento (ver e.g. Figura 66 e seguintes), e explicar os resultados da comparação com a rede Hermes com dois canais virtuais. Como resultados desta análise, poder-se-á sugerir melhorias na estrutura da rede Mercury, visando aumentar seu desempenho e/ou reduzir a área de seu roteador.

Em quarto lugar, comparações com outras redes são um aspecto relevante para posicionar o presente trabalho. Visualiza-se como interessante, por exemplo, comparar a NoC Mercury a uma versão sem canais virtuais da rede Hermes. Outra possibilidade é comparar com uma versão toro da rede Hermes. Esta rede foi inicialmente proposta no escopo do trabalho do Autor [8] e encontra-se no momento com duas versões em implementação, uma com roteamento mínimo determinístico e uma com roteamento mínimo adaptativo [34].

Adicionalmente, divisa-se outros trabalhos futuros que podem ter interesse, incluindo:

- Realizar a prototipação da rede Mercury utilizando conexões em todos os IPs Locais da rede, como por exemplo memórias para validar o funcionamento completo e paralelo em *hardware*.
- Usar técnicas para melhorar o projeto físico da NoC, de forma a posicionar os roteadores estrategicamente conseguindo assim um desempenho melhor em *hardware*.
- Empregar fluxo de tráfego real a rede Mercury para avaliar seu comportamento quando estimulada em um ambiente real.

Referências

[1] E. Aboulhamid, M. Baird, B. Bhattacharya, D. Black, D. Dumlogal, A. Ghosh, A. Goodrich, R. Graulich, T. Groetker, M. Jannsen, E. Lavelle, K. Kranen, W. Mueller, K. Schwartz, A. Rose, R. Ryan, M. Shoji, and S. Swan. *SystemC 2.0.1 Language Reference Manual*. Open SystemC Initiative, 1177 Braham Lane, San Jose, 1.0 edition, 2003.

[2] Aldec - The Design Verification Company. *Active-HDL*. http://www.aldec.com/products/active-hdl. Consultado em Junho/2005.

[3] ARM Corporation. *AMBA 2.0 Specification*. http://www.arm.com/products/solutions/AMBA_Spec.html. Consultado em Junho/2005.

[4] P. Ashenden. *The designer's guide to VHDL*. Morgan Kaufmann Publishers, 1996.

[5] N. Banerjee, P. Vellanki, and K. Chatha. A Power and Performance Model for Network-on-Chip Architectures. In *Design Automation and Test Conference in Europe (DATE)*, February 2004.

[6] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Highly Scalable Network on Chip for Reconfigurable Systems. In *International Symposium on System-on-Chip (SOC)*, March 2003.

[7] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology Adaptive Network-on-chip Design and Implementation. In *Computers and Digital Techniques*, pages 467–472, July 2005.

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

[9] L. Benini and G. De Micheli. Networks on Chips: A New Soc Paradigm. *Computer*, 35:70–78, January 2002.

[10] R. Bergamaschi and J. Cohn. The A to Z of SoCs. In *International Conference on Computer Aided Design (ICCAD)*, pages 791–798, 2002.

[11] J. Bhasker. *A SystemC Primer*. Star Galaxy Publishing, Allentown, PA, 2002.

[12] L. Cai and D. Gajski. Transaction Level Modeling: an Overview. In *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign & System Synthesis (CODES/ISSS)*, pages 19–24. ACM Press, 2003.

[13] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara. From VHDL Register Transfer Level to SystemC Transaction Level Modeling: A Comparative Case Study. In *16th Symposium on Integrated Circuits and Systems Design (SBCCI)*, page 355. IEEE Computer Society, 2003.

[14] E. Carara. Arquiteturas para Roteadores de Redes Intra-chip. Trabalho de Conclusão I, Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN, Porto Alegre, Brasil, 2004. (In Portuguese).

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

[16] H. Chi and J. Chen. Design and Implementation of a Routing Switch for on-chip Interconnection Networks. In *IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (AP-ASIC)*, pages 392–395, August 2004.

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

[18] W. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *38th Design Automation Conference (DAC)*, pages 684–689, 2001.

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

[20] A. de Mello and L. Möller. Arquitetura Multiprocessada em SoCs: Estudo de Diferentes Topologias de Conexão. Trabalho de Conclusão II, Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN, Porto Alegre, Brasil, 2003. (In Portuguese).

[21] J. Duato, A. Robles, F. Silla, and R. Beivide. A Comparison of Router Architectures for Virtual Cut-Through and Wormhole Switching in a NOW Environment. In *13th International and 10th Symposium on Parallel and Distributed Processing (SPDP)*, pages 240–247, March 1999.

[22] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks - an Engineering Approach*. Elsevier Science USA, San Francisco, USA, 2003.

[23] E. Fleury and P. Fraigniaud. A General Theory for Deadlock Avoidance in Wormhole-Routed Networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):626–638, July 1998.

[24] C. Glass and L. Ni. The Turn Model for Adaptive Routing. In *International Symposium on Computer Architecture (ISCA) - 25 Years: Retrospectives and Reprints*, pages 441–450, 1998.

[25] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers - 3º Edição, Norwell, USA, 2003.

[26] P. Guerrier and A. Greiner. A Generic Architecture for on-chip Packet-switched Interconections. In *Design Automation and Test in Europe (DATE)*, pages 250–256, March 2000.

[27] J. Henkel, W. Wolf, and S. Chakradhar. On-chip Networks: A Scalable, Communication-centric Embedded System Design Paradigm. In *17th International Conference on VLSI Design*, page 845. IEEE Computer Society, 2004.

[28] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[29] Y. Hu, H. Chen, Y. Zhu, A. A. Chien, and C-K. Cheng. Physical Synthesis of Energy-Efficient Networks-on-Chip Through Topology Exploration and Wire Style Optimizationz. In *International Conference on Computer Design*, pages 111–118, October 2005.

[30] P. Hölzenspies, E. Schepers, W. Bach, M. Jonker, B. Sikkes, G. Smit, and P. Havinga. A Communication Model Based on an n-Dimensional Torus Architecture Using Deadlock-Free Wormhole Routing. In *Euromicro Symposium on Digital Systems Design (DSD)*, page 166. IEEE Computer Society, 2003.

[31] IBM Inc. *CoreConnect Bus Architecture*, 2005. http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture. Consultado em Julho/2005.

[32] C. Ip and S. Swan. Using Transaction-Based Verification in SystemC. White Paper, 2002.

[33] C. Ip and S. Swan. A Tutorial Introduction on the New SystemC Verification Standard. White Paper, 2003.

[34] C. Scherer Jr. Implementação de uma Rede Intrachip com Topologia Toro 2D Wormhole. Trabalho Individual II, Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN, Porto Alegre, Brasil, 2005. (In Portuguese).

[35] F. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems on Chips. *IEEE Micro*, 22(5):36–45, September 2002.

[36] J. Kim, D. Park, T. Theocharides, N. Vijaykrishnan, and Das C. R. A Low Latency Router Supporting Adaptivity for On-chip Interconnects. In *42nd Design Automation Conference (DAC)*, pages 559–564, June 2005.

[37] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A Network on Chip Architecture and Design Methodology. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 105–112, April 2002.

[38] J. Kurose and R. Keith. *Rede de Computadores e a Internet: Uma Nova Abordagem*. 1. edição - Addison Wesley, São Paulo, Brasil, 2003.

[39] C. Marcon, N. Calazans, F. Moraes, A. Susin, I. Reis, and F. Hessel. Exploring NoC Mapping Strategies: An Energy and Timing Aware Technique. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 502–507, March 2005.

[40] C. Marcon, J. Palma, N. Calazans, A. Susin, R. Reis, and F Moraes. Modeling the Traffic Effect for the Application Cores Mapping Problem onto NoCs. In *IFIP International Conference on Very Large Scale Integration (VLSI-SOC)*, October 2005.

[41] T. Marescaux, T. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. In *Field-Programmable Logic and Applications (FPL)*, pages 795–805, 2002.

[42] G. Martin and H. Chang. System on Chip Design. In *9th International Symposium on Integrated Circuits, Devices Systems (ISIC) - Tutorial 2*, pages 12–17, 2001.

[43] A. Mello, L. Moller, N. Calazans, and F. Moraes. MultiNoC: A Multiprocessing System Enabled by a Network on Chip. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 234–239, March 2005.

[44] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC. In *18th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 178–183, September 2005.

[45] Mentor Graphics Corporate. *LeonardoSpectrum*. http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/index.cfm. Consultado em Setembro/2005.

[46] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an Infrastructure for Low Area Overhead Packet-switching on Chip. *Integration the VLSI Journal*, 38(-):69–93, October 2004.

[47] F. Moraes, A. Mello, L. Möller, L. Ost, and N. Calazans. A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping. In *IFIP International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 318–323, December 2003.

[48] F. Moraes, L. Ost, A. Mello, J. Palma, and N. Calazans. NoCGen - Uma Ferramenta para Geração de Redes Intra-Chip Baseada na Infra-Estrutura HERMES. In *X Workshop Iberchip*, pages 210–216, 2004.

[49] E. Moreno. Modelagem e Validação de Redes Intra-chip no Nível de Transação. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul - PPGCC - FACIN, Porto Alegre, Brasil, April 2004. (In Portuguese).

[50] S. Murali and G. De Micheli. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. In *41st Design Automation Conference (DAC)*, pages 914–919, June 2004.

[51] M. Ni and P. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 26(2):62–76, 1993.

[52] L. Ost. Redes Intra-Chip Parametrizáveis com Interface Padrão para Síntese em Hardware. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brasil, April 2004. (In Portuguese).

[53] L. Ost, A. Mello, J. Palma, F. Moraes, and N. Calazans. MAIA - A Framework for Networks on Chip Generation and Verification. In *Asia South Pacific Design Automation Conference (ASP-DAC), Beijing, China*, pages 49–52, 2005.

[54] J. Palma, C. Marcon, F. Moraes, N. Calazans, R. Reis, and A. Susin. Mapping Embedded Systems onto NoCs - The Traffic Effect on Dynamic Energy Estimation. In *18th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 196–201, September 2005.

[55] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R Saleh. Effect of Traffic Localization on Energy Dissipation in NoC-based Interconnect Infrastructures. In *International Symposium on Circuits and Systems (ISCAS)*, pages 1774–1777, May 2005.

[56] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, August 2005.

[57] R. Reis. *Concepção de Circuitos Integrados*. Editora Sagra Luzzatto, Instituto de Informática da UFRGS - Porto Alegre, 2000.

[58] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. In *Design Automation and Test Conference in Europe (DATE)*, March 2003.

[59] International Sematech. International Technology Roadmap for Semiconductors - 2002 Update. http://public.itrs.net, 2002.

[60] K. Shin and J. Rexford. Support for Multiple Classes of Traffic in Multicomputer Routers. In *Parallel Computer Routing and Communication Workshop*, pages 116–130, May 1994.

[61] L. Soares, G. Lemos, and S. Colcher. *Redes de Computadores - Das LANs, MANs e WANs às Redes ATM*. Campus, Rio de Janeiro, 1995.

[62] S. Swan. An Introduction to System Level Modeling in SystemC 2.0. White Paper, 2001.

[63] L. Tedesco, A. Mello, N. Calazans, and F. Moraes. Traffic Generation and Performance Evaluation for Mesh-based NoCs. In *18th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 184–189, September 2005.

[64] T. Theocharides, G. Link, N. Vijaykrishnan, M. Irwin, and V. Srikantam. A Generic Reconfigurable Neural Network Architecture Implemented as a Network on Chip. In *IEEE International SOC Conference*, pages 191–194, 2004.

[65] H. Wang, L. Peh, and S. Malik. A Technology-Aware and Energy-Oriented Topology Exploration for On-Chip Networks. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1238–1243, March 2005.

[66] Xilinx - The Programmable Logic Company. *Ise 7.1*. http://www.xilinx.com/ise. Consultado em Junho/2005.

[67] Y. Yang, H. Amano, H. Shibamura, and T. Sueyoshi. Recursive Diagonal Torus: An Interconnection Network for Massively Parallel Computers. In *IEEE Symposium Parallel and Distributed Processing*, pages 591–594, December 1993.

[68] Y. Yang, A. Funahashi, A. Jouraku, H. Nishi, H. Amano, and T. Sueyoshi. Recursive Diagonal Torus: An Interconnection Network for Massively Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):701–715, July 2001.

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

Apêndice A Arquitetura do roteador NoC Mercury versão RTL.

Neste Apêndice, ilustra-se a arquitetura do roteador da rede Mercury em três níveis de detalhamento. Na Figura 69 uma versão com todos os sinais é demonstrada. Na Figura 70 uma versão um pouco mais simplificada mas ainda com detalhes internos dos sinais é ilustrada. A Figura 71 demonstra o roteador em blocos de modo a facilitar o entendimento do mesmo.

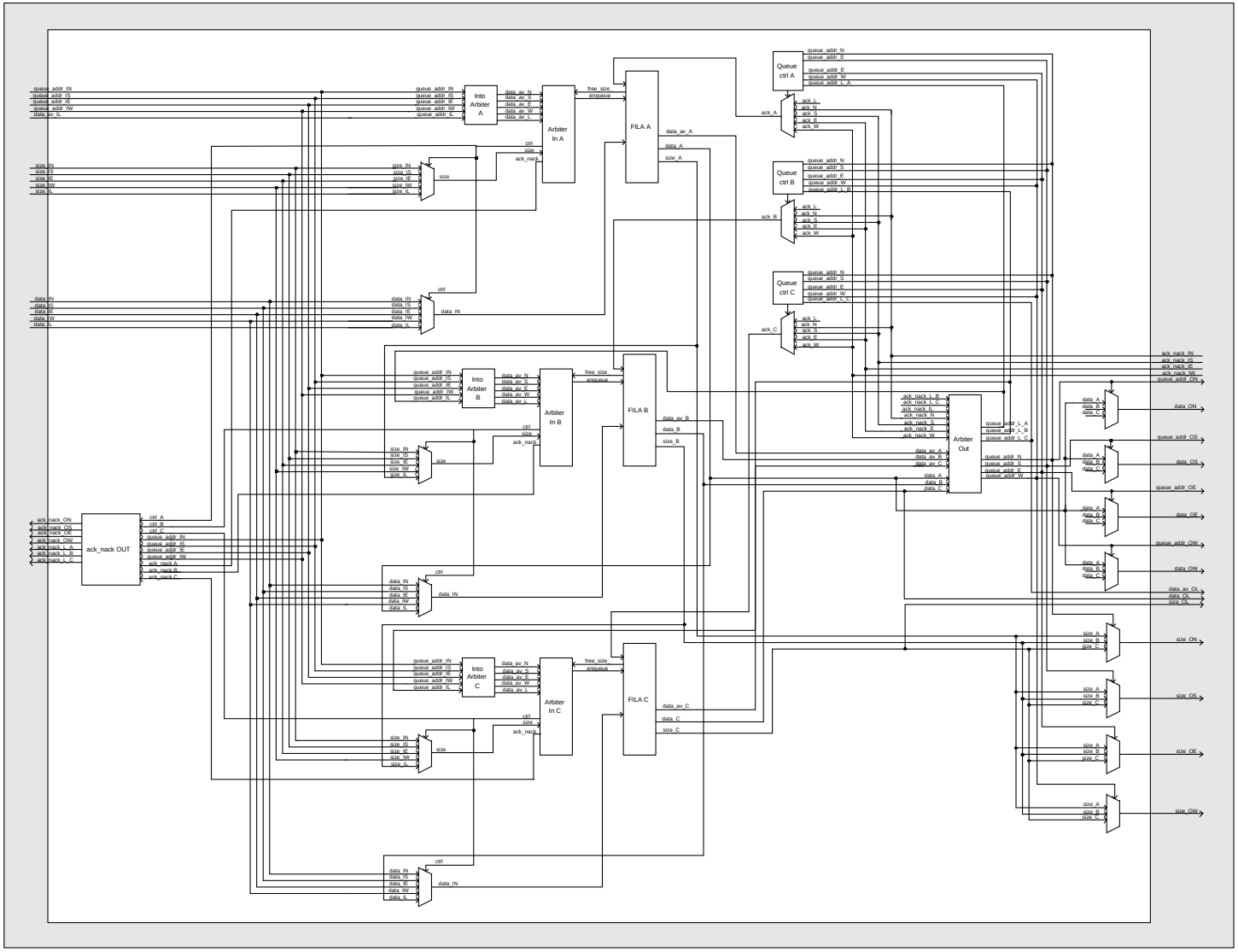


Figura 69 – Roteador da rede Mercury na versão RTL de forma detalhada.

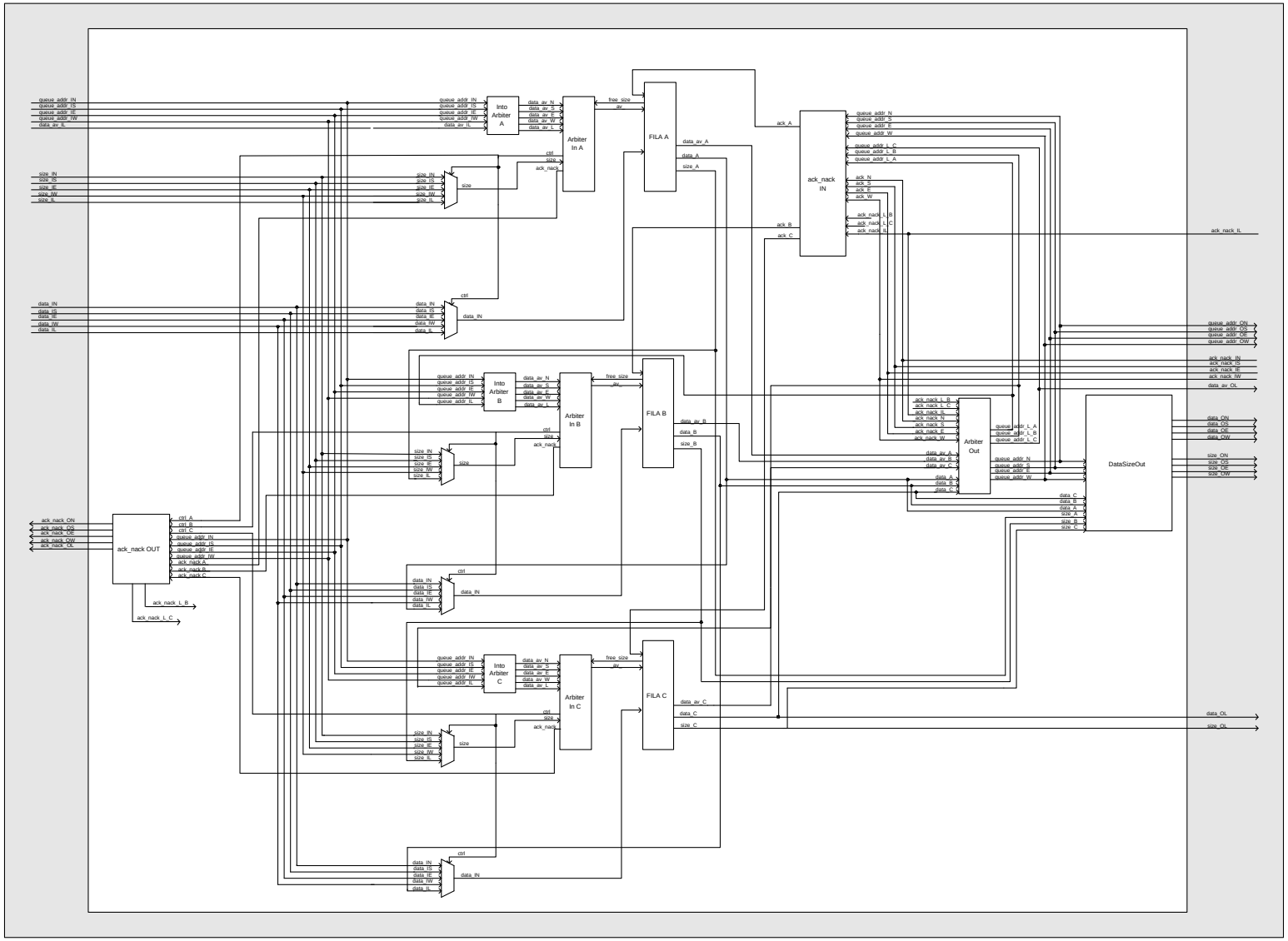


Figura 70 – Roteador da rede Mercury na versão RTL de forma semi-detalhada.

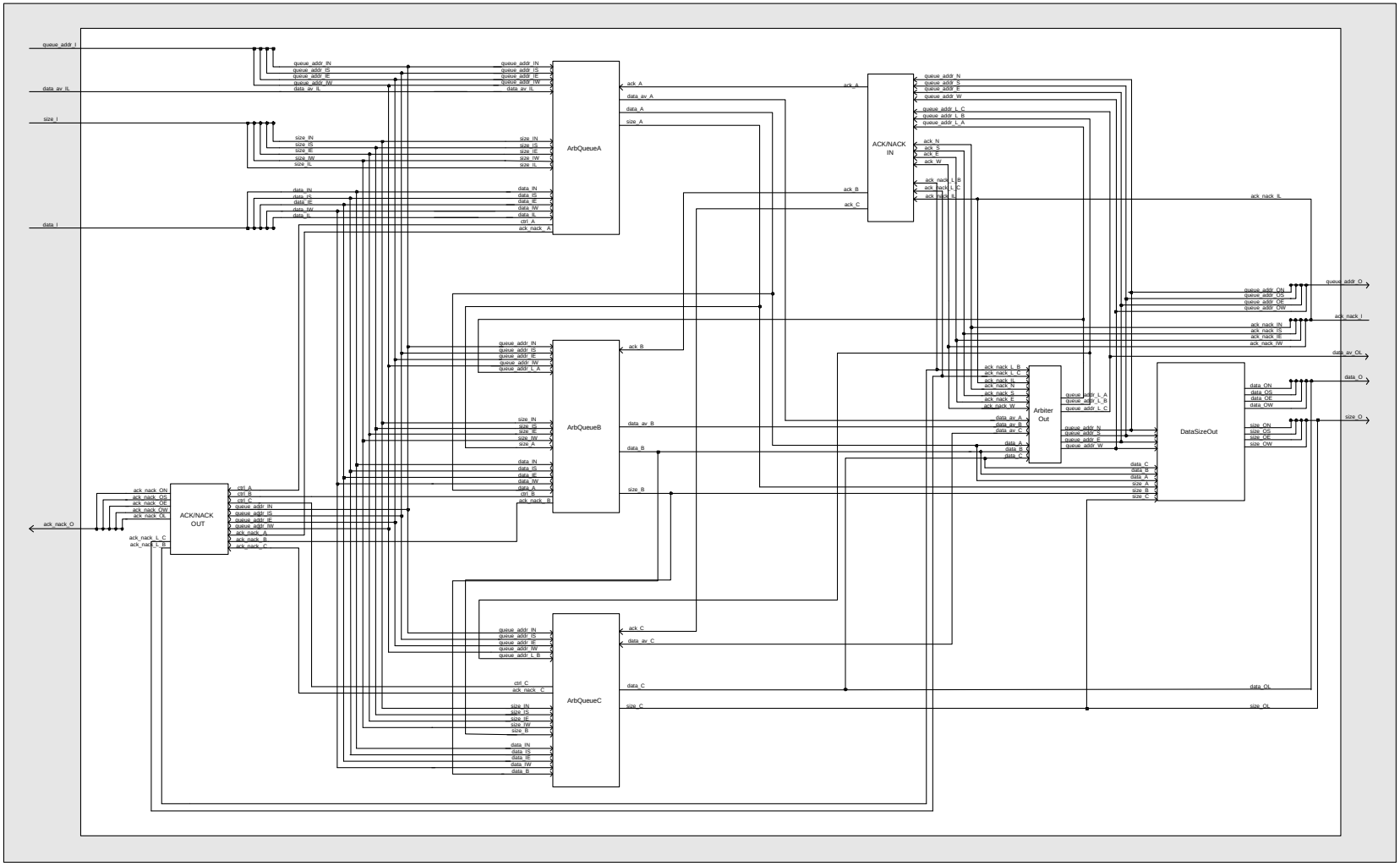


Figura 71 – Roteador da rede Mercury na versão RTL em alto nível.