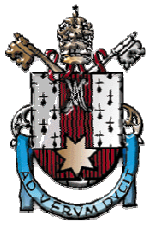


Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Curso de Bacharelado em Ciência da Computação

Uma Exploração Arquitetural de Redes Intra-chip com Topologia Malha e Modo de Chaveamento *Wormhole*

Trabalho de Conclusão II

Autor:

Everton Alceu Carara

Orientador:

Prof. Fernando Gehm Moraes

Porto Alegre, Novembro de 2004.

ÍNDICE

ÍNDICE DE FIGURAS

Índice de Tabelas

1 INTRODUÇÃO

O avanço tecnológico na construção de sistemas digitais complexos é tal que permite implementar em um único circuito integrado mais de 50 milhões de transistores (para microprocessadores). O ritmo desses avanços da tecnologia de fabricação tem se mantido exponencial nas últimas décadas, como atesta a Lei de Moore [SCH97]. A sigla SoC, do inglês *System on Chip* [BER01], designa um sistema computacional completo implementado em um único circuito integrado. SoCs normalmente contêm um ou mais processadores de propósito geral, lógica digital (programável ou não), circuitos analógicos, além de bancos de memória dinâmica e estática. SoCs fornecem como vantagens maior desempenho, menor consumo de potência, menor volume e peso comparado com o projeto baseado em múltiplos circuitos integrados em uma placa de circuito impresso. A Figura 1 ilustra a arquitetura genérica de um SoC.

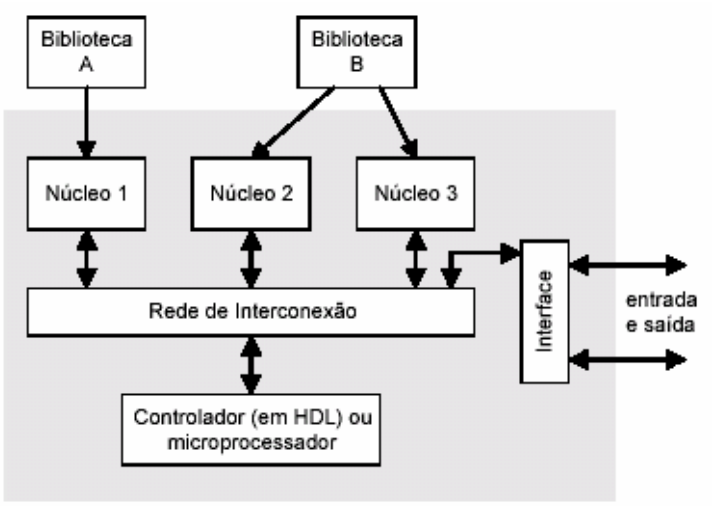


Figura 1 – Arquitetura genérica de um SoC.

Projetar um processador já é uma tarefa complexa. Então, qual a técnica que se deve utilizar para projetar um SoC, contendo múltiplos processadores, módulos de hardware dedicado e software embarcado? Esta é uma questão ainda sem resposta plenamente satisfatória. Considera-se que parte da resposta está no reuso de módulos complexos de hardware, denominados núcleos de propriedade intelectual (*IP cores*), *núcleos IP* ou apenas *núcleos.* Um núcleo é um bloco de circuito pré-projetado e pré-verificado que pode ser usado na construção de uma aplicação maior ou mais complexa em uma pastilha de material semicondutor [GUP97]. Núcleos são pré-projetados, pré-verificados e prototipados em hardware pelo menos uma vez. O principal objetivo de seu uso é a redução do *time-to-market* de produtos. Apesar das vantagens inerentes à utilização de núcleos, identificam-se quatro grandes problemas que devem ser resolvidos para que se possa construir facilmente um SoC [BER01]: (*i*) como integrar núcleos entre si; (*ii*) quais linguagens usar para a descrição de sistemas; (*iii*) como proteger a propriedade intelectual do autor e do usuário do núcleo; (*iv*) como testar projetos baseados em núcleos.

1.1 Estruturas de Interconexão de Núcleos

Nos sistemas integrados atuais, a interconexão entre os núcleos é realizada através de canais ponto-a-ponto ou de canais multi-ponto [ZEF03]. Nos canais ponto-a-ponto (Figura 2a), tipicamente utilizados em sistemas baseados no modelo de comunicação de fluxo de dados (ex: codificadores/decodificadores de áudio ou de vídeo), os núcleos são interligados por canais dedicados, sendo que cada canal é constituído por um conjunto de fios ligando dois núcleos. Nos canais multi-ponto (Figura 2b), geralmente usados em sistemas orientados ao modelo de comunicação baseado em espaço de endereçamento único (ex: processador-memória-E/S), a interconexão é estruturada sob a forma de um barramento compartilhado, multiplexado no tempo, no qual os núcleos do sistema são conectados [GUE00]. Estas duas abordagens podem ser comparadas quanto: (*i*) ao paralelismo; (*ii*) consumo de energia; (*iii*) freqüência de operação; (*iv*) escalabilidade, (*v*) área; (*vi*) reusabilidade.

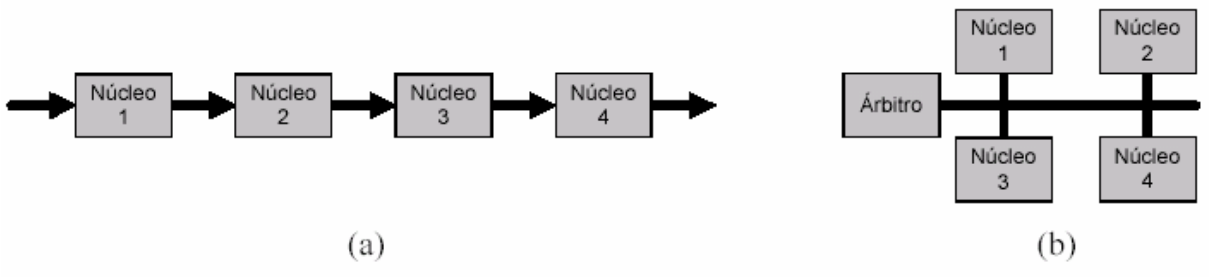


Figura 2 - Estruturas de interconexão nos SoCs atuais: (a) ponto-a-ponto; (b) multi-ponto.

Quanto ao *paralelismo*, os canais ponto-a-ponto permitem que múltiplas comunicações sejam realizadas simultaneamente, pois são independentes entre si. Já em um barramento somente uma comunicação pode ocorrer a cada momento, e os múltiplos núcleos do sistema concorrem pelo uso do barramento, o qual possui um controlador central para escalonar a sua utilização.

Em interconexões baseadas em canais ponto-a-ponto, os fios tendem a ser mais curtos e com uma única carga. Na abordagem multi-ponto, os fios tendem a ser mais longos e com múltiplas cargas. Dado que a potência consumida é proporcional às capacitâncias do circuito, a abordagem multi-ponto tende a apresentar um maior *consumo de energia*. O consumo de energia, além disso, também é proporcional à atividade de chaveamento. Um barramento é responsável por todas as comunicações entre os núcleos do sistema. Desta forma, a sua taxa de utilização é próxima de 100% [LIA00] e a atividade de chaveamento nos seus fios é intensa. Já em canais ponto-a-ponto o consumo de energia em cada segmento é inferior, pois a atividade de chaveamento é muito menor.

A *freqüência de operação* do barramento é menor em relação as canais ponto-a-ponto, devido às altas cargas capacitivas impostas pelo circuito.

No que diz respeito à *escalabilidade*, uma estrutura de interconexão é dita escalável se a sua largura de banda cresce com o tamanho do sistema [HWA93]. Em canais ponto-a-ponto, cada núcleo adicionado ao sistema exige a implementação de novos canais para a comunicação com os núcleos já existentes. A largura de banda desses novos canais é agregada à largura de banda já disponível. Em uma estrutura multi-ponto, um núcleo adicionado ao sistema geralmente é conectado aos canais já compartilhados pelos demais núcleos sem aumentar a largura de banda da estrutura. Pelo contrário, esse núcleo irá competir pela banda disponível, tornando a largura de

banda disponível por núcleo ainda menor.

Quanto à *área*, o custo de uma arquitetura de comunicação é dado pelo custo das interconexões para a implementação dos canais físicos e pelo custo de silício para a realização de funções lógicas como arbitragem, decodificação de endereços, chaveamento, entre outras. Em uma estrutura ponto-a-ponto, cada canal físico é constituído por fios unidirecionais e pode ser otimizado independentemente (eg. largura do canal físico, inserção de *buffers,* etc). Diferentemente, em uma abordagem multi-ponto, a largura física dos canais do barramento deve ser dimensionada de modo a atender aos requisitos de latência e de largura de banda de todos os núcleos do sistema, sendo que a otimização de um barramento compartilhado com múltiplas cargas pode ser muito difícil [HU02]. Em um sistema baseado em canais ponto-a-ponto, os núcleos que se comunicam são geralmente posicionados o mais próximo possível uns dos outros, de modo a reduzir o comprimento dos canais de comunicação. Em um sistema baseado em uma arquitetura multi-ponto, os canais de comunicação do barramento devem alcançar todos os núcleos do sistema, sendo que, tipicamente assume-se que o comprimento total do barramento é igual a metade do perímetro do CI [LAN00].

A *reusabilidade* é um requisito fundamental no mercado de microeletrônica. Se a estrutura de interconexão e os componentes de comunicação a ela associados forem também reutilizáveis, a integração do sistema será ainda mais fácil, ajudando a atingir os requisitos de tempo de projeto do sistema. Uma estrutura de interconexão baseada em canais ponto-a-ponto dedicados é fundamentada em uma abordagem *ad-hoc* e o seu reuso em sistemas diferentes é bastante restrito. Além disso, o tempo para o projeto da estrutura de interconexão tende a ser ainda maior com o aumento da complexidade dos SoCs [DAL01], o que inviabiliza essa abordagem como solução de interconexão em sistemas com requisitos restritos de tempo de projeto.

Estruturas de interconexão multi-ponto, como o barramento, são simples e totalmente reutilizáveis. Por esta razão, barramentos têm sido a abordagem preferida para a construção dos SoCs atuais, apesar das suas inúmeras desvantagens em relação à abordagem ponto-a-ponto.

Uma estrutura de interconexão que pode solucionar os problemas relacionados ao uso de barramentos, simples ou hierárquicos, são as redes intra-chip [BEN02] [BEN01], um conceito denominado *network on chip* – NoC. NoCs herdam das redes de computadores e de sistemas distribuídos as características das camadas de protocolos e o conceito de ligação de nodos à rede. Nas NoCs, os núcleos do sistema são interligados por meio de uma rede composta por roteadores e canais ponto-a-ponto. A comunicação entre os núcleos ocorre pela troca de mensagens transferidas por meio de roteadores e canais intermediários até atingir o seu destino [BEN02].

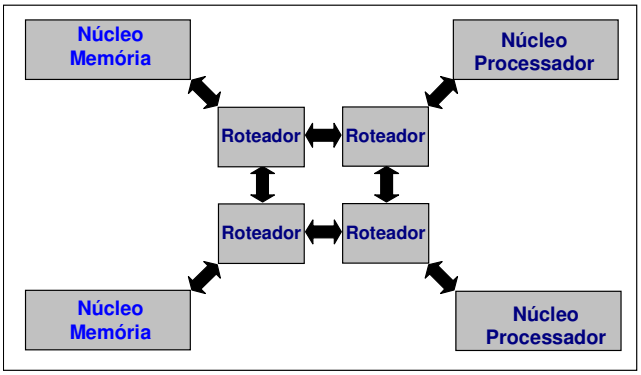


Figura 3 – Exemplo de NoC.

Espera-se que os sistemas baseados em NoCs forneçam boas soluções para o reuso de núcleos [KUM02], já que as NoCs possuem as seguintes características: (*i*) eficiência no consumo de energia [BEN01]; (*ii*) largura de banda escalável, quando comparada à arquiteturas de barramento tradicionais; (*iii*) reusabilidade; (*iv*) decisões de roteamento distribuídas [GUE00]; (*v*) paralelismo na comunicação. Embora tenham como desvantagem maior custo em área de silício e latência na comunicação, esses problemas serão certamente atenuados pela grande disponibilidade de transistores e por soluções arquiteturais que permitirão reduzir a latência da rede e seus efeitos no desempenho da aplicação [ZEF03].

1.2 Objetivos do Trabalho

O presente Trabalho de Conclusão tem três objetivos estratégicos.

O primeiro objetivo é estudar algumas técnicas de roteamento, memorização, controle de fluxo e arbitragem. Em seguida implementar algumas das técnicas de memorização, controle de fluxo e arbitragem estudadas, dando continuidade aos trabalhos de pesquisa relacionados a redes intra-chip. A motivação para este estudo é proporcionar à rede HERMES [MOR04a][MOR04b] a capacidade de ser personalizada em função de uma determinada aplicação.

O segundo objetivo visa integrar os diferentes módulos, criando novas redes com diferentes características.

O terceiro objetivo corresponde ao estudo e implementação da técnica de canais virtuais na HERMES, visando em trabalhos futuros o atendimento a requisitos de Qualidade de Serviço (*QoS – Quality of Service*).

1.3 Contribuições do trabalho

As contribuições deste trabalho de conclusão são:

1. Novas implementações dos módulos que compõem a arquitetura do roteador integrados à ferramenta MAIA [MOR04a].
2. Possibilidade de criar diferentes redes intra-chip compostas pelos novos módulos.
3. Implementação dos canais virtuais na HERMES.
4. Avaliação preliminar de desempenho da rede, em termos de latência e área, utilizando os novos módulos e canais virtuais.

1.4 Organização do Documento

Capítulo 2 - **Redes Intra-Chip**: Apresenta uma visão geral sobre os conceitos relacionados a redes intra-chip e análise das principais características da rede HERMES.

Capítulo 3 - **Controle de Fluxo**: Apresenta as principais funcionalidades de controle de fluxo em redes intra-chip e algumas abordagens como: *handshake*, crédito e *slack buffer*. O Capítulo encerra com uma breve comparação entre *handshake* e crédito.

Capítulo 4 - **Memorização**: Apresenta as principais funcionalidades da memorização e algumas abordagens como: Memorização na entrada, Memorização na saída e Memorização centralizada compartilhada.

Capítulo 5 - **Roteamento**: Apresenta as principais funcionalidades do roteamento e o funcionamento dos seguintes algoritmos: Algoritmo XY, Algoritmo *West-First*, Algoritmo *North-Last* e Algoritmo *Negative-First*, todos mínimos e para redes malha. O Capítulo encerra com uma avaliação dos algoritmos apresentados.

Capítulo 6 - **Arbitragem**: Apresenta as principais funcionalidades da arbitragem e algumas técnicas como: arbitragem centralizada, arbitragem distribuída e *daisy chain*. Também se apresenta algumas políticas de arbitragem, as quais podem ser aplicadas tanto na abordagem centralizada quanto na abordagem distribuída. O Capítulo encerra com uma comparação de arbitragem centralizada versus arbitragem distribuída.

Capítulo 7 - **Canais Virtuais:** Apresenta o conceito de canais virtuais. Mostra duas implementações diferentes de canais virtuais sobre a rede Hermes, encerrando o capítulo com algumas avaliações de desempenho e área.

Capítulo 8 - **Conclusões e Trabalhos futuros**: Apresenta alguns comentários sobre o trabalho desenvolvido bem como as atividades futuras a serem realizadas.

2 REDES INTRA-CHIP

Uma rede intra-chip é composta por nodos de processamento e nodos de chaveamento. Por exemplo, a rede em anel ilustrada na Figura 4 é uma topologia bastante simples e econômica. Cada nodo de chaveamento possui ligações para dois nodos de chaveamento vizinhos e para um nodo de processamento local. Os nodos de processamento são responsáveis pela execução das tarefas do sistema, enquanto que os nodos de chaveamento (também chamados de roteadores) são responsáveis pela transferência de mensagens entre nodos de processamento.

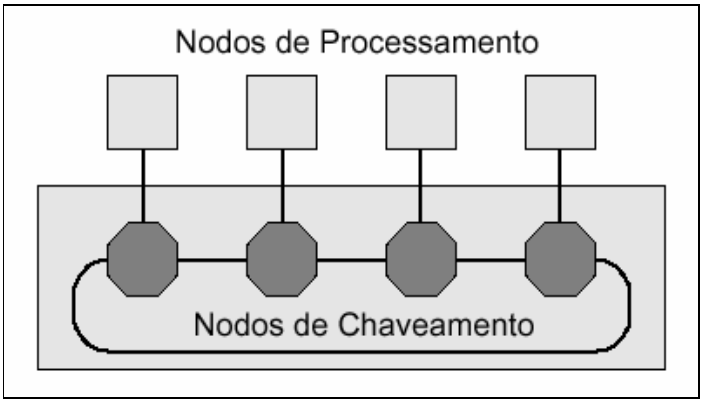


Figura 4 – Rede intra-chip com topologia em anel.

A ligação física entre dois nodos de chaveamento é denominada enlace (*link*). Um enlace possui um ou dois canais físicos de comunicação e é implementado sob a forma de conexões elétricas ou ópticas. Em NoCs os enlaces são implementados com fios, utilizando uma camada de baixa resistividade (alumínio ou cobre). Um canal físico pode ainda possuir dois ou mais canais lógicos, chamados de canais virtuais. Dependendo da topologia da rede, um nodo de processamento também pode ser ligado a um nodo de chaveamento através de um enlace.

Em SoCs, a comunicação entre os núcleos é implementada através do roteamento entre os módulos. É neste quesito, roteamento, que as topologias NoCs superam as topologias de barramento. Na topologia NoC as conexões são locais, entre módulos de chaveamento próximos, o que reduz o comprimento total de roteamento e por conseqüência aumenta o desempenho elétrico. Já nas topologias de barramento, as conexões são globais, o que acarreta perda de desempenho devido aos fios longos.

O nodo chaveador é sem dúvida o principal elemento de uma NoC, pois as principais características de uma rede intra-chip são definidas pela funcionalidade desse. Em geral, os nodos de chaveamento possuem um núcleo de chaveamento, uma lógica para roteamento e arbitragem (abreviado por R&A na Figura 5b) e portas de comunicação para outros nodos de chaveamento e, dependendo da topologia, para um nodo de processamento local, não necessariamente um processador, (Figura 5a). As portas de comunicação incluem canais de entrada e de saída, os quais podem possuir, ou não, *buffers* para o armazenamento temporário de informações.

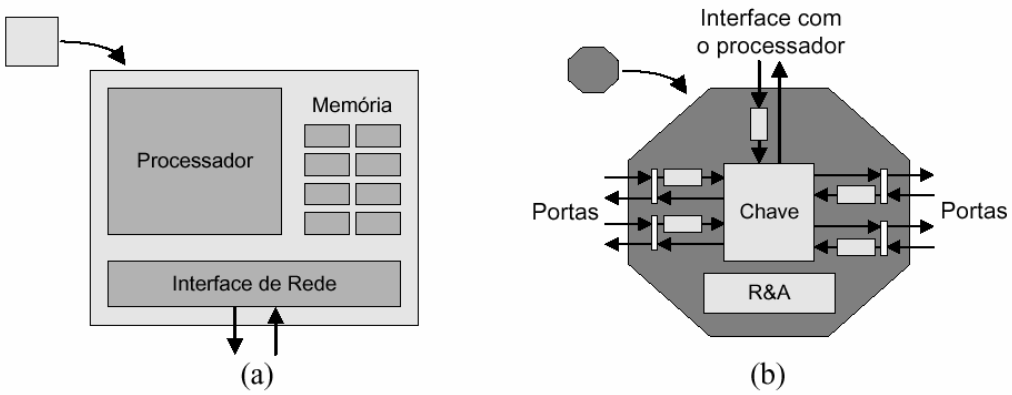


Figura 5 - Nodos: (a) de processamento; (b) de chaveamento.

As informações trocadas entre um nodo fonte e um nodo destino de uma comunicação são organizadas sob a forma de mensagens. Em geral, as mensagens são formadas por três partes: um cabeçalho (*header*), um corpo de dados (*payload*) e um terminador (*trailer*). O cabeçalho e o terminador formam um envelope ao redor do corpo de dados da mensagem. O cabeçalho carrega informações de roteamento e de controle utilizadas pelos nodos de chaveamento para propagar a mensagem através da rede, em direção ao seu destino. Já o terminador, carrega informações utilizadas na detecção de erros e na sinalização do final da mensagem.

Geralmente, as mensagens são quebradas e transmitidas em pacotes. Um pacote corresponde à menor unidade de informação, contendo detalhes sobre o roteamento e seqüenciamento dos dados. Sua estrutura é semelhante à de uma mensagem, contendo um cabeçalho, um corpo de dados e um terminador. Um pacote é constituído por uma seqüência de *flits*, cuja largura depende da largura física do canal. Um *flit* é a menor unidade de dados sobre a qual é realizado o controle de fluxo.

Uma rede intra-chip pode ser caracterizada pela sua topologia e pelas estratégias utilizadas para controle de fluxo, memorização, roteamento e arbitragem que ela utiliza. Essas características são discutidas a seguir.

2.1 Topologias de Redes de Interconexão

Uma rede de interconexão pode ser caracterizada pela forma como seus nodos são interligados. Essa estrutura é tipicamente representada por um grafo G(N,C) onde *N* representa o conjunto de nodos (de processamento e/ou de chaveamento) da rede e *C* representa o conjunto de canais de comunicação, representando o conjunto de arestas do grafo. Quanto à topologia, as redes de interconexão para multiprocessadores podem ser agrupadas em duas classes principais, as redes diretas e as redes indiretas.

Nas redes diretas, cada nodo de chaveamento possui um nodo de processamento associado, e esse par pode ser visto como um elemento único dentro do sistema, tipicamente referenciado pela palavra nodo, como ilustra a Figura 6. Pelo fato de utilizarem nodos de chaveamento tipo roteador, as redes diretas são também chamadas de redes baseadas em roteadores [DUA02]. Uma outra denominação utilizada é a de redes estáticas, pois as ligações entre os nodos não mudam durante a execução de uma dada aplicação [HWA93].

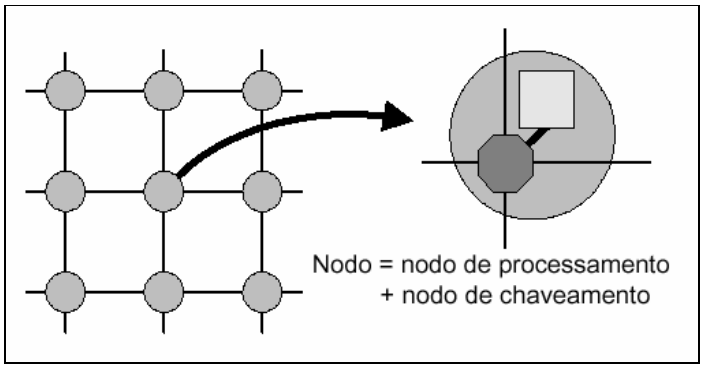


Figura 6 - Nodos de redes diretas.

As topologias de redes diretas estritamente ortogonais mais utilizadas são a grelha (ou malha) (Figura 7a), o toróide (Figura 7b) e o hipercubo (Figura 7c).

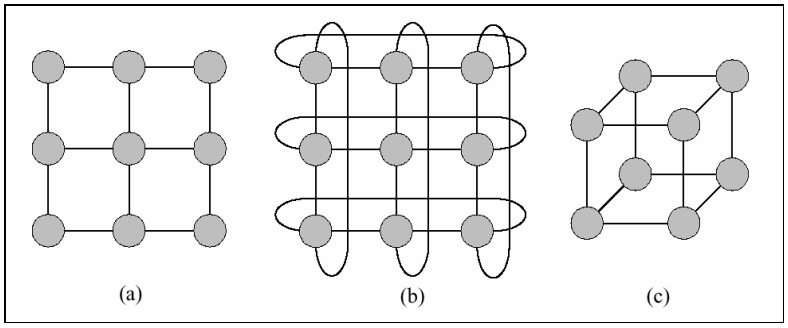


Figura 7 - (a) Malha 2D 3x3; (b) Toróide 2D 3x3; (c) Hipercubo 3D.

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 redes indiretas de diretas. A topologia da rede é definida pela estrutura de interconexão desses nodos.

Duas topologias clássicas de redes indiretas se destacam: o *crossbar* e as redes multiestágio. Para conexão indireta de N nodos de processamento, o *crossbar* (Figura 8) é a topologia ideal, pois consiste de um único nodo de chaveamento $N \times N$. O *crossbar* possui uma complexidade da ordem de N^2, o que torna o seu custo de área proibitivo para redes grandes.

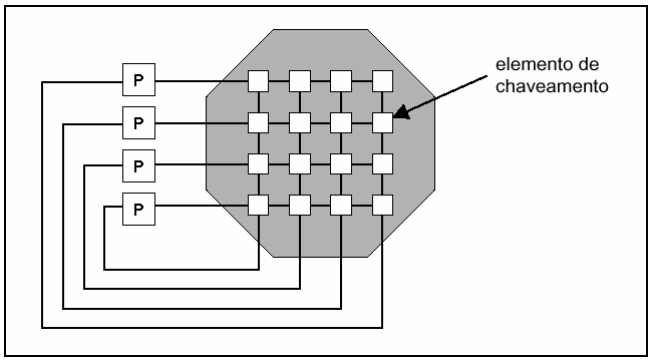


Figura 8 - Crossbar 4 x 4.

2.2 Controle de fluxo

Trata-se do mecanismo que determina o momento em que o pacote deve ser transmitido para o roteador seguinte, logo após ter sido roteado. Serve basicamente para ajustar a taxa de saída de um roteador origem à taxa de entrada de um roteador destino, evitando assim a perda de pacotes durante a transmissão. O controle de fluxo é necessário sempre que dois ou mais pacotes necessitam de um mesmo recurso simultaneamente. Quando isto ocorre, um dos pacotes pode se ser bloqueado, armazenado em um *buffer*, desviado por um caminho alternativo ou simplesmente descartado [CUL98].

As redes de interconexão realizam o controle de fluxo no nível de enlace. A sua implementação depende da estrutura do enlace, mas o procedimento é semelhante para os diversos métodos. O roteador destino envia um sinal para a origem, indicando que ele está pronto para receber mais dados. O roteador origem não transmite enquanto não receber o sinal do roteador destino.

Alguns métodos para o controle de fluxo, os quais serão detalhados posteriormente no Capítulo 3:

- Controle de fluxo baseado em *handshake*;
- Controle de fluxo baseado em crédito;
- Controle de fluxo baseado em *Slack Buffer*.

2.3 Memorização

Em redes que utilizam chaveamento por pacotes, os roteadores devem ser capazes de armazenar os *flits* dos pacotes destinados às saídas que estejam sendo utilizadas por outros pacotes e realizar o controle de fluxo para evitar a perda de dados. Para isto, é necessário utilizar um esquema de memorização para armazenamento temporário dos pacotes bloqueados no roteador. O caso ideal seria ter roteadores com capacidade de armazenamento infinita e garantir que nenhum pacote fosse bloqueado por outro quando sua saída fosse liberada. Todavia, a memória do roteador é limitada e o bloqueio de um pacote pode ser inevitável. A organização dos *buffers* de memória e sua localização dentro de roteadores interferem criticamente no desempenho deste componente.

Existem três opções básicas de memorização, as quais serão detalhados posteriormente no Capítulo 4:

- Memorização na entrada;
- Memorização na saída;
- Memorização centralizada compartilhada.

2.4 Roteamento

Roteamento é o método utilizado pelo roteador para decidir por qual porta de saída um pacote que chegou deve ser enviado. O algoritmo que implementa o método de roteamento tem uma forte influência no desempenho da comunicação na rede. Em geral, o algoritmo de roteamento visa atender alguns objetivos específicos como:

- Conectividade: capacidade de rotear pacotes de qualquer nodo origem para qualquer nodo destino;
- Assegurar a correta funcionalidade da rede em termos de entrega de pacotes evitando *deadlock* e *livelock* [DUA02][HEN96];
- Adaptatividade: capacidade de rotear pacotes através de caminhos alternativos quando ocorre congestionamento ou falha em algum componente em algum ponto do caminho planejado;
- Tolerância a falhas: capacidade de rotear pacotes na presença de falhas em componentes;

Os algoritmos de roteamento que serão detalhados posteriormente, no Capítulo 5, são:

- XY;
- XY *West-First*;
- XY *North-Last*;
- XY *Negative-First*.

2.5 Arbitragem

A arbitragem é responsável por definir qual porta de entrada (ou *buffer* de entrada) poderá utilizar uma determinada porta de saída (ou *buffer* de saída) em um determinado momento. Este mecanismo é essencial para resolver conflitos causados pela existência de múltiplos pacotes competindo por uma mesma saída. Ele deve ser capaz de resolver esses conflitos, selecionando uma saída com base em algum critério, sem levar qualquer pacote a sofrer *starvation* (não atendimento da solicitação).

Os esquemas de arbitragem tentam balancear dois fatores na escolha da próxima entrada a ser atendida. Primeiro, cada uma das entradas tem sua prioridade (dinâmica ou estática), sendo que a de mais alta prioridade deve ser escolhida primeiro. Segundo, deve ser garantido que toda entrada, mesmo aquela com a prioridade mais baixa, seja escolhida em algum momento. A observância desse princípio garante que toda entrada que quiser transmitir dados será selecionada [PAT98].

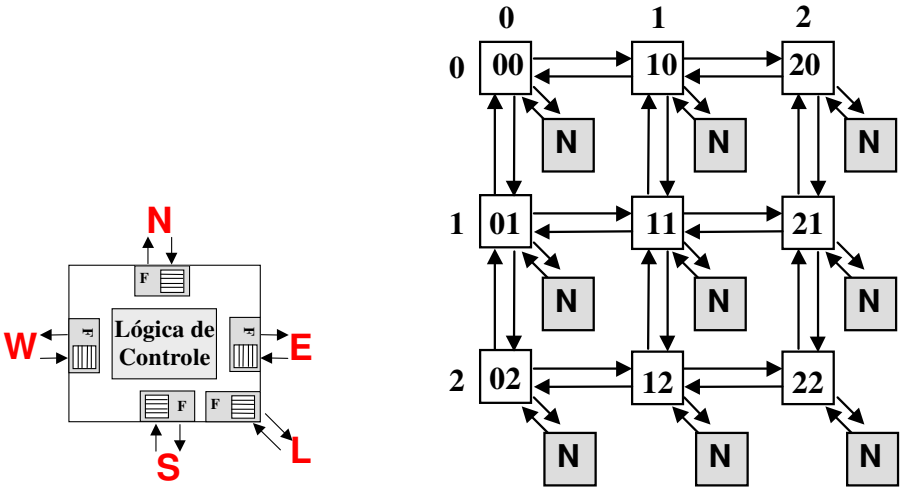
Alguns esquemas de arbitragem, os quais serão explicados, no Capítulo 6, são:

- Arbitragem centralizada;
- Arbitragem distribuída;
- *Daisy chain*.

2.6 Estudo de Caso – a Rede Hermes

HERMES [MOR04b] é uma infra-estrutura usada para gerar NoCs com chaveamento por pacotes para diferentes topologias, tamanhos de *flits*, profundidades de *buffer* e algoritmos de roteamento. HERMES implementa três níveis hierárquicos do modelo de referência OSI: (*i*) físico – corresponde à definição da interface de comunicação entre os roteadores, (*ii*) enlace - adota o protocolo *handshake* para o envio e recebimento de dados de forma confiável, supondo que o meio físico seja confiável, e (*iii*) rede – no qual é implementado o modo de chaveamento *wormhole*.

O principal componente desta infra-estrutura é o roteador (Figura 9a). Este roteador possui uma lógica de controle e pode ser composto por até 5 portas bidirecionais: *East*, *West*, *North*, *South* e *Local*. Cada porta possui uma fila de tamanho parametrizável para o armazenamento temporário de *flits*. A porta *Local* estabelece a comunicação entre o roteador e seu núcleo de hardware. As demais portas são todas opcionais e ligam os roteadores aos seus roteadores vizinhos.



a) Arquitetura do roteador HERMES. b) Rede 3 *x* 3 topologia malha HERMES.

Figura 9 – Roteador e exemplo de NoC Hermes.

A lógica de controle, ilustrada na Figura 10, é composta por dois módulos: roteamento e arbitragem. O módulo de roteamento implementa um dos algoritmos de roteamento disponibilizados pela infra-estrutura HERMES. O árbitro determina qual o pacote deve ser chaveado primeiro quando mais de um chegarem no roteador no mesmo tempo. A arbitragem dinâmica rotativa (*Round-Robin*) é utilizada para permitir o chaveamento de pacotes com um grau eqüitativo de justiça. A prioridade de cada porta a cada instante depende da última que obteve permissão para realizar chaveamento.

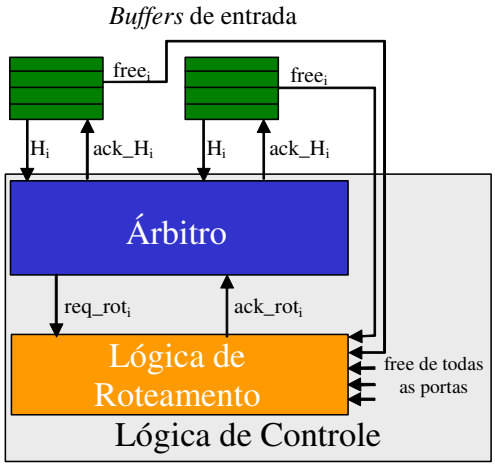


Figura 10 – Principais módulos do roteador: *buffers*, árbitro e lógica de roteamento.

As principais características da rede HERMES no início do presente trabalho são resumidas na Tabela 1.

Tabela 1 – Quadro resumo das características da infra-estrutura HERMES.

Nível Físico	Caminhos bidirecionais com largura parametrizável e suporte à protocolo *handshake*. Número de portas por roteador parametrizável de 3 a 5.
Nível Enlace	Protocolo assíncrono de comunicação usando a técnica de *handshake*.
Nível Rede	Comunicação ponto-a-ponto, utilizando chaveamento por pacotes.
Nível Transporte e superiores	Implementados de forma *ad-hoc*.
Pacote Físico	1 *flit* com destino. 1 *flit* com tamanho do *payload*. tamanho do *payload* parametrizável.
Chaveamento	*Packet Switching*.
Modo de chaveamento	*Wormhole*.
Roteamento	Distribuído, XY Determinístico, Mínimo.
Topologia	Parametrizável: Malha, *Torus* e *Torus Dobrado*. Apenas a topologia malha possui algoritmos operacionais de arbitragem e roteamento.
Filas	Na entrada de cada porta, com tamanho parametrizável.

A arquitetura do roteador é o principal responsável pelo desempenho da rede como um todo, pois diferentes combinações de técnicas de roteamento, memorização e arbitragem podem melhorar ou piorar consideravelmente o tempo de resposta da NoC para uma dada topologia ou aplicação. Uma má escolha da combinação dessas técnicas pode resultar em problemas como *deadlock*, *livelock* e *starvation*.

A geração manual de redes a partir da infra-estrutura HERMES é um processo passível de erros, devido à grande quantidade de fios que ligam os roteadores entre si e ao núcleo local, como ilustra a Figura 11.

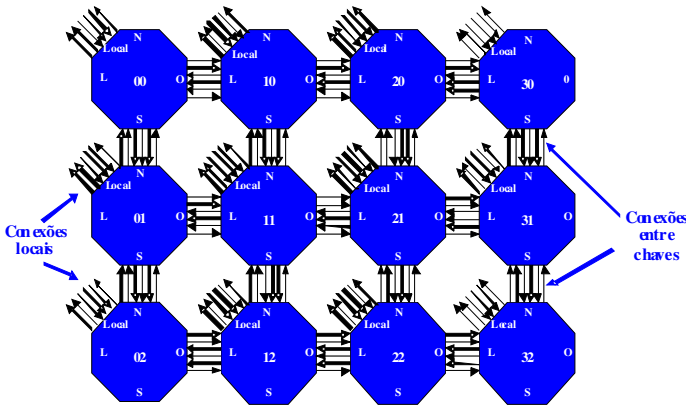


Figura 11 - Interconexões entre roteadores em uma rede intra-chip com dimensão 3 x 4.

Para automatizar esse processo foi desenvolvida a ferramenta MAIA[1] [MOR04a], a qual permite a escolha do algoritmo de roteamento e a configuração de parâmetros como: largura do *flit*, profundidade das filas de entrada dos roteadores e dimensão da rede. Outra possibilidade é utilizar a ferramenta para gerar diversas configurações, possibilitando a análise de qual configuração tem melhor desempenho para uma dada aplicação. Uma outra funcionalidade da ferramenta é a remoção de filas de entrada de canais não utilizados no roteador (Figura 12), o que reduz a área da HERMES em hardware. Além disso, a ferramenta é capaz de automatizar a produção de tráfego e *testbenchs*, permitindo com isso validar a rede gerada.

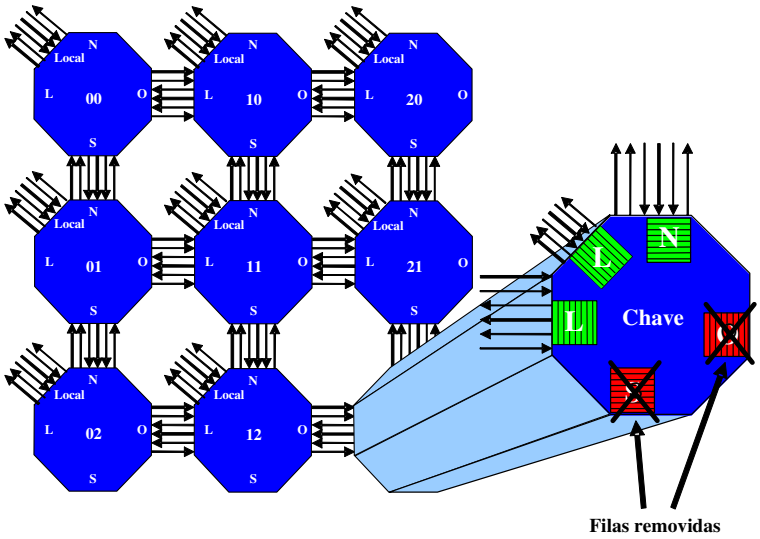


Figura 12 – Remoção de filas de entrada de canais não utilizados no roteador.

[1] O nome original da ferramenta era NocGen.

3 CONTROLE DE FLUXO

As redes de interconexão realizam o controle de fluxo no nível de enlace. O problema a ser tratado é transferir um dado de uma porta de saída de um nodo origem, através de um enlace, para uma porta de entrada de nodo destino, sem que haja perda do mesmo. O armazenamento no nodo destino pode ser um simples *latch* ou um *buffer* organizado na forma de *FIFO*. Independente do tipo do enlace, problema é que o armazenamento na entrada do nodo destino pode não estar disponível para aceitar a transferência, devendo o dado ser mantido no nodo fonte até que o nodo destino esteja pronto para recebê-lo.

3.1 Handshake

Quando o transmissor deseja transmitir um dado, ele disponibiliza-o em um barramento e indica que há um dado disponível através de uma linha que vai até o receptor. O transmissor mantém o dado no barramento até que o receptor sinalize que o dado foi armazenado. O receptor, quando detecta que o transmissor tem um dado disponível, verifica se há disponibilidade para o seu armazenamento. Se houver, o receptor armazena o dado e sinaliza através de uma outra linha que vai até o transmissor. Caso contrário, o receptor não sinaliza enquanto não houver disponibilidade para o armazenamento.

A Figura 13 ilustra os sinais do protocolo *handshake*.

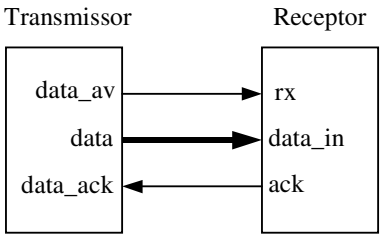


Figura 13 – Sinais do protocolo Handshake.

A Figura 14 apresenta o diagrama de tempos dos eventos ocorridos durante a transmisão de dados.

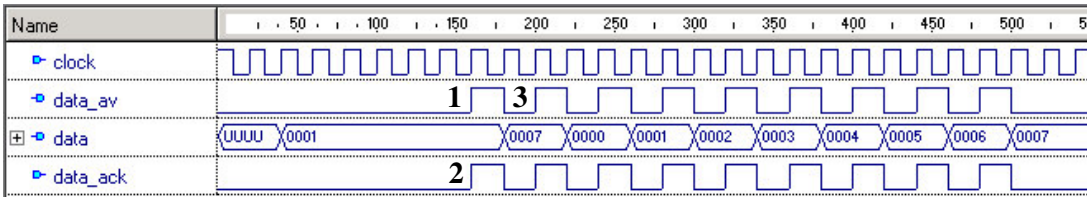


Figura 14 - Diagrama de tempo dos eventos de uma transmissão que usa o protocolo *handshake*.

1. O transmissor coloca o sinal *data_av* em nível alto, indicando que tem um dado válido no barramento *data* (0001) para ser transmitido.

2. No mesmo ciclo de *clock*, o receptor detecta que o transmissor quer enviar um dado. O receptor armazena o dado e sinaliza para o transmissor colocando em nível alto seu sinal *ack*, o qual está ligado à entrada *data_ack* do transmissor. Apesar de na simulação ocorrerem no mesmo instante de tempo, o sinal *data_ack* apresenta um atraso combinacional em relação ao sinal *data_av*.

3. O transmissor detecta que o receptor armazenou o dado (*data_ack* em nível alto*)* e coloca o sinal *data_av* em nível baixo.

Os passos 1, 2 e 3 se repetem enquanto o transmissor tiver dados para enviar.

Com esse protocolo são necessários pelo menos 2 ciclos de *clock* para a transmissão de cada *flit*.

3.2 Crédito

Quando o transmissor quer enviar um dado, ele o disponibiliza em um barramento, indica que há um dado disponível através de uma linha que vai até o receptor e envia também um sinal *clock* para sincronizar a transmissão [BOL04]. O receptor, quando detecta que o transmissor tem um dado disponível, verifica se há disponibilidade para o seu armazenamento e enquanto houver espaço para armazenar dados, ele mantém um sinal de crédito para o transmissor.

Quando o receptor não tiver mais espaço para o armazenamento dos dados, ele sinaliza para o transmissor através do sinal de crédito. O transmissor então, mantém o dado no barramento até o receptor indicar, através do sinal de crédito, que está apto a receber dados novamente.

A Figura 15 ilustra os sinais do protocolo baseado em crédito.

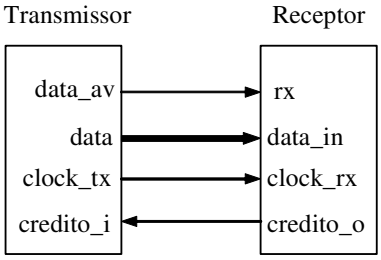


Figura 15 – Sinais de protocolo baseado em crédito.

A Figura 16 apresenta o diagrama de tempos dos eventos ocorridos durante a transmissão de *flits*.

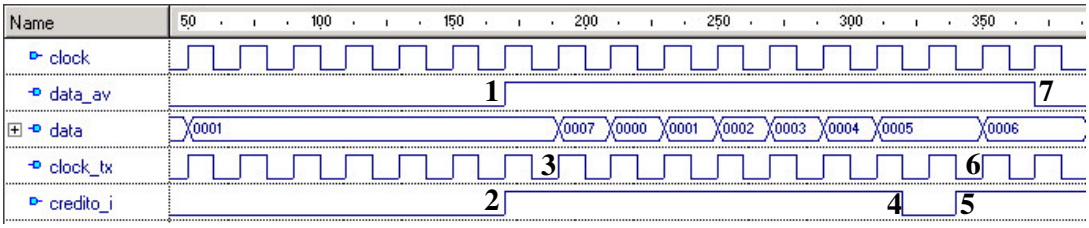


Figura 16 - Diagrama de tempo dos eventos de uma transmissão.

1. O transmissor coloca o sinal *data_av* em nível alto, indicando que tem um dado válido no barramento *data* (0001) para ser transmitido. Esse sinal permanecerá em nível alto enquanto o transmissor tiver dados para enviar.
2. No mesmo ciclo de *clock* o receptor detecta que o transmissor quer enviar um dado. O receptor verifica que tem espaço disponível para o armazenar dados e sinaliza para o transmissor colocando em nível alto seu sinal *credito_o*, o qual está ligado à entrada *credito_i* do transmissor. Esse sinal permanecerá em nível alto enquanto o receptor tiver espaço para receber dados. Aqui, como no caso do protocolo *handshake*, o sinal *credito_i* apresenta um atraso combinacional em relação ao sinal *data_av*, porém isso não é observável na simulação funcional.
3. A cada borda de subida do sinal *clock_tx*, o receptor armazena um *flit* e o transmissor, se tiver um novo dado para enviar, disponibiliza-o no barramento *data*.
4. O transmissor coloca em nível baixo o sinal *credito_o*, indicando que não tem mais espaço para armazenar *flits*. O transmissor mantém o dado 0005 no barramento *data* até que o receptor indique que tem crédito novamente.
5. O receptor volta a indicar que tem crédito novamente.
6. O receptor armazena o *flit* 0005.
7. O transmissor coloca o sinal *data_av* em nível baixo, indicando que não tem mais *flits* para transmitir.

Observar que este mecanismo de protocolo implica sincronização baseada em sinais de *clock* entre o transmissor (*clock_tx*) e o receptor (*clock_rx*) como mostra a Figura 17. O sinal de *clock_tx* do transmissor é usado apenas para controlar a fila de entrada do receptor. Enquanto essa fila tiver espaço disponível para armazenar *flits* o receptor mantém o sinal *credito_o* em nível alto, caso contrário *credito_o* é colocado em nível baixo. Graças a esse mecanismo de sincronização, o transmissor e o receptor podem operar em freqüências diferentes.

Este mecanismo permite uma operação da rede segundo o paradigma *localmente síncrono globalmente assíncrono* (GALS) [ZIP04]. Este modo de operação permite reduzir a potência consumida por um dado circuito integrado, uma vez que não é mais necessário ter uma árvore global de distribuição de *clock*.

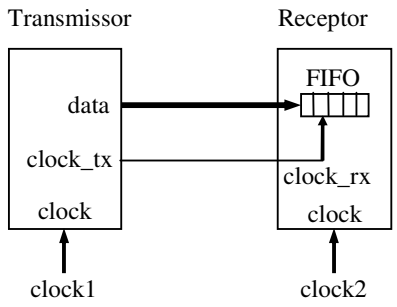


Figura 17 – Sincronização no protocolo baseado em crédito.

Um exemplo de rede que utiliza um protocolo de controle de fluxo baseado em crédito é a QNoC [BOL04]. A Figura 18 ilustra a arquitetura do roteador utilizado na rede QNoC.

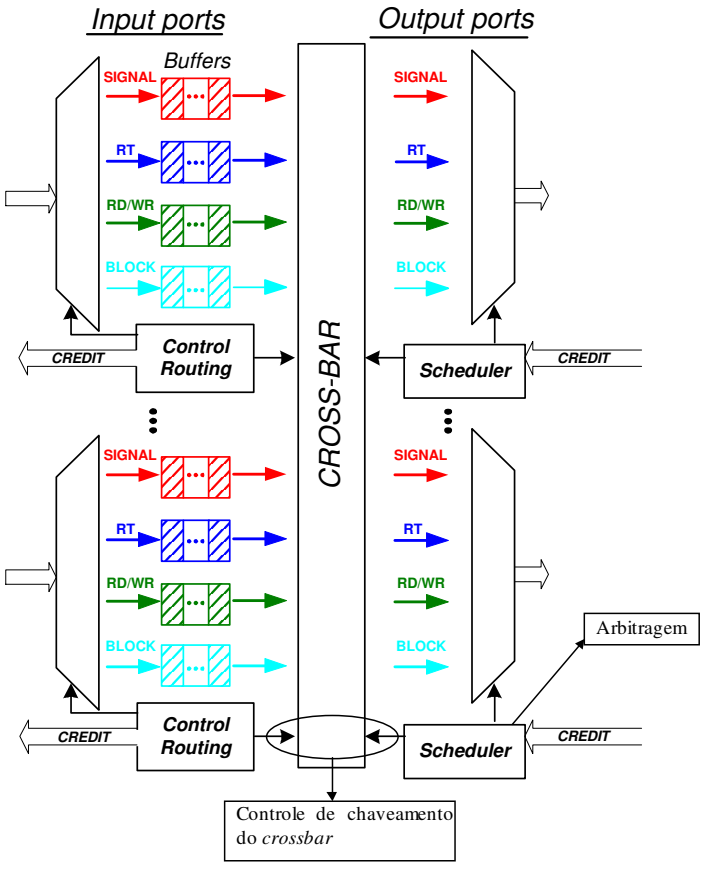


Figura 18 – Roteador da rede QNoC.

A QNoC diferencia a comunicação entre os roteadores em quatro diferentes níveis de serviço:

- *Signalling*: usado para a transmissão de mensagens urgentes e pacotes muito pequenos proporcionando baixa latência (como interrupções de processadores remotos);
- *Real-Time*: garante largura de banda e latência para aplicações em *real-time*, como processamento de som e vídeo;
- *Read/Write (RD/WR)*: dá suporte para o acesso simples a memória e registradores, sem requisitos de latência ou largura de banda;

- *Block-Transfer*: usado para a transferência de mensagens e blocos de dados longos (como preenchimento de um bloco de dados de memória *cache*);

Os *flits* que compõem os pacotes são classificados em três tipos:

- *FP (full packet)*: pacote composto por apenas um *flit*.
- *EP (end of package)*: o último *flit* de um pacote.
- *BDY (body)*: um *flit* qualquer que não seja o último.

Cada porta do roteador possui quatro filas de entrada, uma para cada nível de serviço. Para indicar a disponibilidade de espaço nas filas, é utilizado um barramento de quatro bits (*CREDIT*) que indica se há espaço disponível em cada uma das filas. A Figura 19 mostra as forma de onda de uma transmissão, onde o transmissor indica o tipo do *flit* (*Type*) presente no barramento (*Data_o*) e o nível de serviço (*SL*).

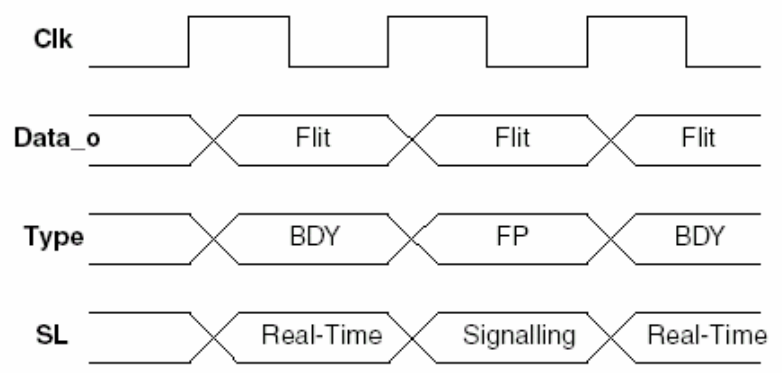


Figura 19 – Transmissão de *flits* na QNoC, ilustrando a preempção de pacotes de mais alta prioridade (*Signalling*) sobre pacotes de menor prioridade(*Real-Time*).

3.3 Slack Buffer

Trata-se de uma abordagem alternativa para o controle de fluxo baseado em crédito. O *buffer* de entrada do receptor é visto como um reservatório com uma marca de nível alto e uma marca de nível baixo como mostra a Figura 20. O conteúdo desse reservatório deve ser mantido na região delimitada por estas duas marcas, a qual é chamada de histerese.

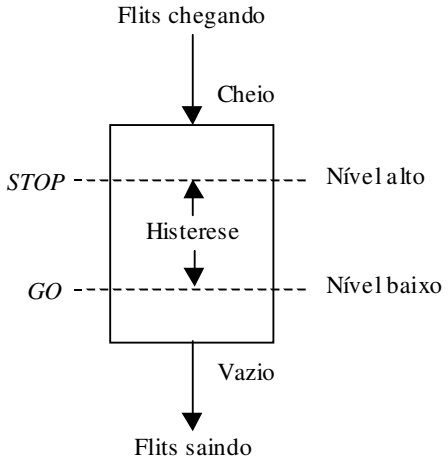


Figura 20 – Operação do slack buffer.

Quando o nível do reservatório está abaixo da marca de nível baixo, o sinal *GO* é enviado para o transmissor, indicando que o receptor está apto a receber dados. De maneira análoga, quando o nível do reservatório ultrapassa a marca de nível alto, o sinal *STOP* é enviado para o transmissor, indicando que o receptor não pode receber dados no momento.

Sinais redundantes de *GO* podem ser enviados em qualquer lugar abaixo da marca de Nível alto e sinais de *STOP* podem ser enviados em qualquer lugar acima da marca de Nível baixo sem nenhum efeito prejudicial. A taxa de sinalização dos sinais *GO* e *STOP* pode ser reduzida simplesmente aumentando a região de histerese.

Este tipo de controle de fluxo é bastante adequado à redes que utilizam chaveamento *wormhole*, segundo [BOD95] .

Este procedimento é muito semelhante ao mecanismo de controle de fluxo baseado em crédito. A principal diferença consiste na ausência do controle por histerese, utilizando-se todo o *buffer* no mecanismo de crédito.

3.4 Comparação Handshake versus Crédito

Abaixo é apresentada uma comparação, baseada em formas de onda, entre os protocolos *handshake* e baseado em crédito visando análise de desempenho. Em ambos os casos o *buffer* do receptor tem capacidade para armazenar até 8 *flits* e encontra-se inicialmente vazio. O sinal de crédito (*credit_i*) é ativado assim que a conexão for estabelecida.

A Figura 21 apresenta a transmissão de 9 *flits* utilizando o protocolo *handshake*. Como mostra o diagrama de tempo abaixo, passaram-se 17 ciclos de *clock* desde o momento em que o transmissor sinalizou o primeiro dado válido (1) até o momento em que os 9 *flits* foram enviados (2).

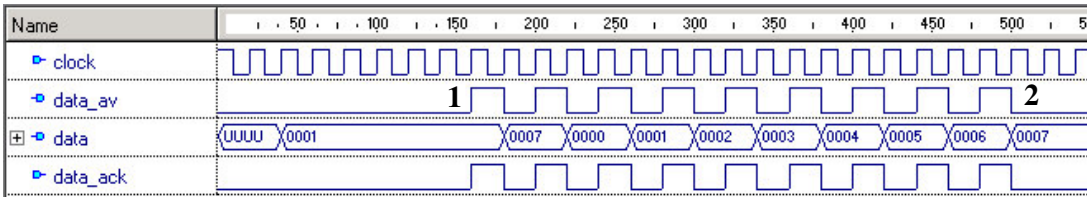


Figura 21 – Transmissão de 9 *flits* utilizando o protocolo *handshake*.

A Figura 22 apresenta a transmissão de 9 *flits* utilizando o protocolo baseado em crédito. Como mostra o diagrama de tempo abaixo, passaram-se apenas 10 ciclos de *clock* desde o momento em que o transmissor sinalizou o primeiro dado válido (1) até o momento em que os 9 *flits* foram enviados (2).

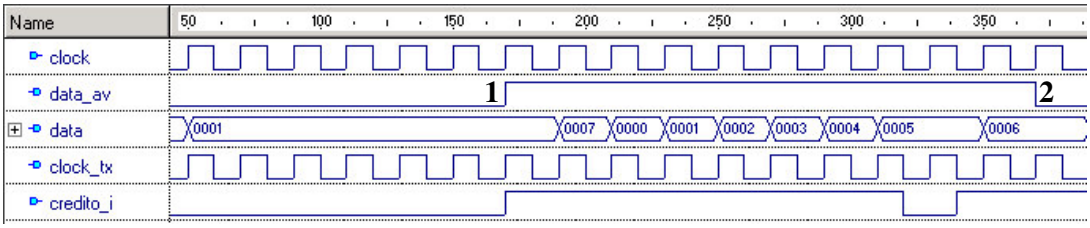


Figura 22 - Transmissão de 9 *flits* utilizando o protocolo baseado em crédito.

Como pode-se observar, o protocolo baseado em crédito é mais rápido, pois nele o transmissor não espera um sinal de *ack* do receptor a cada dado enviado, como é caso do protocolo *handshake*. No protocolo baseado em crédito o transmissor continua a enviar dados enquanto a entrada *credito_i* estiver nível alto, ou seja, enquanto o receptor tiver espaço disponível no seu *buffer*.

A Tabela 2 mostra o resultado de uma comparação entre duas NoCs com diferentes controles de fluxo. A topologia usada foi malha 3x3, com cada roteador enviando 1000 pacotes para diferentes destinos. Ambas NoCs foram geradas pela ferramenta MAIA, trabalham com *flits* de 16 bits e *buffers* de entrada de 8 posições. A simulação foi realizada com pacotes de tamanho 10 *flits* e carga de tráfego igual a 100%. Os resultados extraídos, também utilizando a ferramenta MAIA são: (*i*) latência mínima, máxima e média para a transmissão de um pacote; (*ii*) desvio padrão da latência; (*iii*) tempo total para a transmissão de todos os pacotes. Todos os tempos apresentados são em ciclos de *clock*. O resultado de área foi obtido através da ferramenta de síntese lógica *Lonardo Spectrum* tendo como alvo um dispositivo FPGA XC2V2000 da Virtex.

Tabela 2 - Resultados da comparação *Handshake* versus Créditos.

	Controle de fluxo	
	Handshake	Créditos
Latência mínima	31 ciclos de *clock*	20 ciclos de *clock*
Latência máxima	550 ciclos de *clock*	386 ciclos de *clock*
Latência média	97,62 ciclos de *clock*	70,9 ciclos de *clock*
Desvio padrão da latência	53,52	41,078
Tempo total de transmissão	52562 ciclos de *clock*	38933 ciclos de *clock*
Área	36,67% CLB Slices	39,93% CLB Slices

Os resultados demonstram um significativo ganho de desempenho utilizando-se um protocolo baseado em créditos, sendo que o custo da sua implementação, em termos de área, em relação ao protocolo *Handshake* é pequeno. O protocolo baseado em crédito é um protocolo síncrono e de alto desempenho, que aceita circuitos com diferentes taxas de *clock*. O protocolo *handshake* é mais simples de implementar, entretanto requer que os módulos vizinhos trabalhem na mesma freqüência, sob o risco de ocorrer meta-estabilidade.

4 MEMORIZAÇÃO

A organização dos *buffers* dentro do roteador tem um impacto significativo no seu desempenho. Há quatro estratégias de memorização: (i) apenas *latches* de entrada e saída; (ii) *buffers* na entrada; (iii) *buffers* na saída e (iv) *buffer* centralizado. Poucos *flits* de memorização já são suficientes para um melhor de desempenho. Com o aumento do tamanho e da densidade dos *chips*, maior é a capacidade de memorização que se torna disponível, oferecendo ao projetista a possibilidade de explorar diferentes organizações de *buffers*.

4.1 Memorização na entrada

Memorização na entrada consiste em inserir *buffers* independentes em cada uma das portas de entrada do roteador, como mostra a Figura 23. Essa estratégia é a mais simples e de menor custo em relação as demais estratégias que utilizam *buffers*, pois cada *buffer* possui um espaço de memória fixo onde os dados são lidos na mesma ordem em que são escritos. A operação do roteador (conjunto composto pelas filas, árbitro e lógica de roteamento) é relativamente simples: ele monitora o primeiro elemento de cada uma das filas (cabeçalho do pacote), realiza o roteamento baseado no destino do pacotes, o qual está contido no cabeçalho, e faz uma requisição de chaveamento para cada um deles Tipicamente, a lógica de roteamento está associada a cada uma das portas de entrada para determinar a saída desejada.

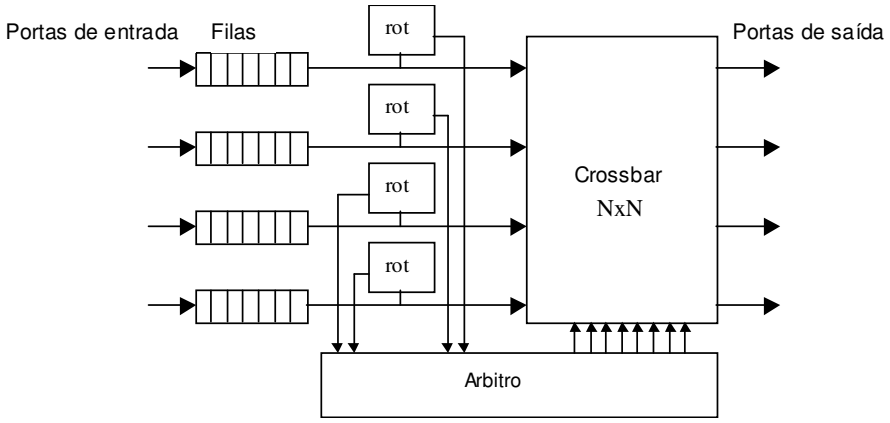


Figura 23 – *Buffers FIFO* na entrada.

O grande inconveniente dessa abordagem é a suscetibilidade ao bloqueio de cabeça de fila (*HOL - head of line*). Supondo duas portas de entrada com pacotes destinados à mesma porta de saída, um deles será transmitido, enquanto que o outro ficará bloqueado na fila. O pacote logo atrás do pacote bloqueado poderia estar destinado a uma outra porta de saída que não está em uso, porém também ficará bloqueado na fila [CUL98]. Consequentemente os *buffers* devem ser dimensionados para minimizar ou evitar esse tipo de bloqueio.

Uma alternativa para contornar o problema do bloqueio *HOL* é a abordagem *SAFC* (*Statically Allocated, Fully Connected*), a qual consiste em dividir cada *buffer* de entrada em *N* partições com tamanho igual a 1/*N* do tamanho do *buffer* original, conforme é mostrado na Figura 24. Esta alternativa requer o uso de um *crossbar* N^2x*N* ou *N crossbars N*x1 ao invés de um único *crossbar N*x*N*.

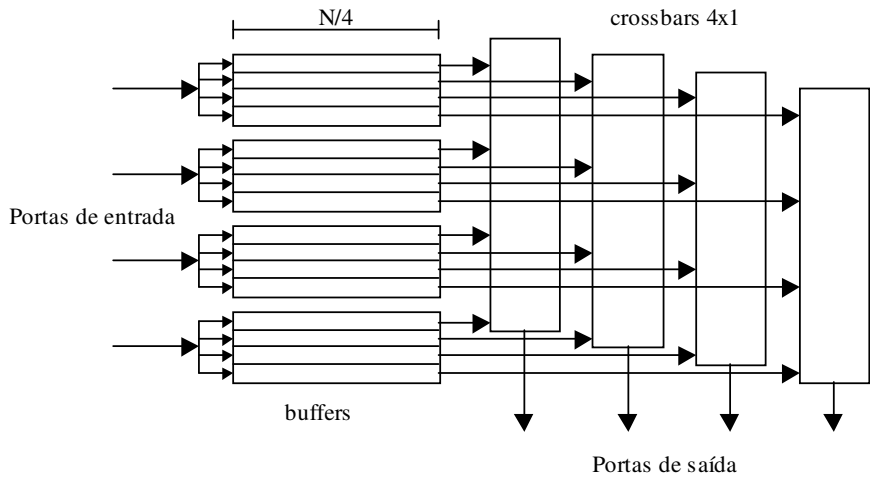


Figura 24 – *Buffers SAFC*.

A desvantagem da *SAFC* é o custo adicional referente ao controle do núcleo de chaveamento e ao controle dos *N buffers* por porta de entrada. Além disso, a taxa de utilização dos *buffers* é limitada a 1/*N* do espaço de armazenamento total, ou seja, pior que a primeira estratégia, e o controle de fluxo é mais complexo, pois deve ser realizado para cada *buffer* de entrada, e exige um pré-roteamento dos pacotes recebidos para que os mesmos sejam direcionados à partição correspondente à saída a ser requisitada. Portanto, se o tráfego não for completamente uniforme, a primeira estratégia se adapta melhor do que *buffers SAFC* [TAM92].

A estratégia *SAMQ* (*Statically Allocated Multi-Queue*), mostrada na Figura 25 visa simplificar o gerenciamento do *crossbar* através da multiplexação das saídas dos *buffers* de entrada atribuídos a uma mesma porta de saída. A estratégia *SAMQ* elimina alguns dos inconvenientes da *SAFC*, pois reduz o custo do *crossbar*. Porém, ela ainda mantém os outros dois problemas relacionados à utilização dos *buffers* e ao controle de fluxo.

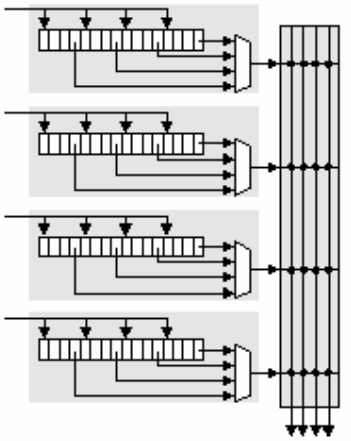


Figura 25 – *Buffers SAMQ*.

A estratégia *DAMQ* (*Dynamically-Allocated, Multi-Queue*) foi proposta em [TAM92] com o objetivo de evitar os problemas das estratégias anteriores. Na *DAMQ*, o espaço no *buffer* é associado a uma porta de entrada e particionado dinamicamente entre as portas de saída conforme a demanda dos pacotes recebidos (Figura 26). Assim, é possível evitar o bloqueio *HOL* dos *buffers FIFO* e aumentar a utilização do espaço de memória disponível. Outra vantagem é que o controle

de fluxo na *DAMQ* é mais simples que nas estratégias *SAFC* e *SAMQ*, não requerendo pré-roteamento. Sua desvantagem é a implementação física complexa do gerenciamento do *buffer DAMQ*, baseado em listas encadeadas.

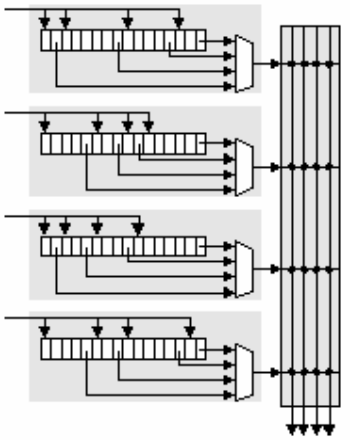


Figura 26 – *Buffers DAMQ*.

4.2 Memorização na saída

Nesta abordagem o espaço de memorização é particionado entre as saídas, e cada uma destas partições podem ser implementadas com *buffers* independentes como mostra a Figura 27. Cada *buffer* deve ser capaz de suportar a demanda simultânea das *N* entradas, podendo ser implementado com *N* portas de escrita ou com uma porta de escrita operando a uma velocidade *N* vezes maior que a das entradas. No entanto a implementação de *buffer* com múltiplas portas de escrita aumenta o seu tamanho e reduz seu desempenho. Outra dificuldade na implementação desta abordagem é que ela requer um controle de fluxo interno entre portas de entrada e de saída do roteador.

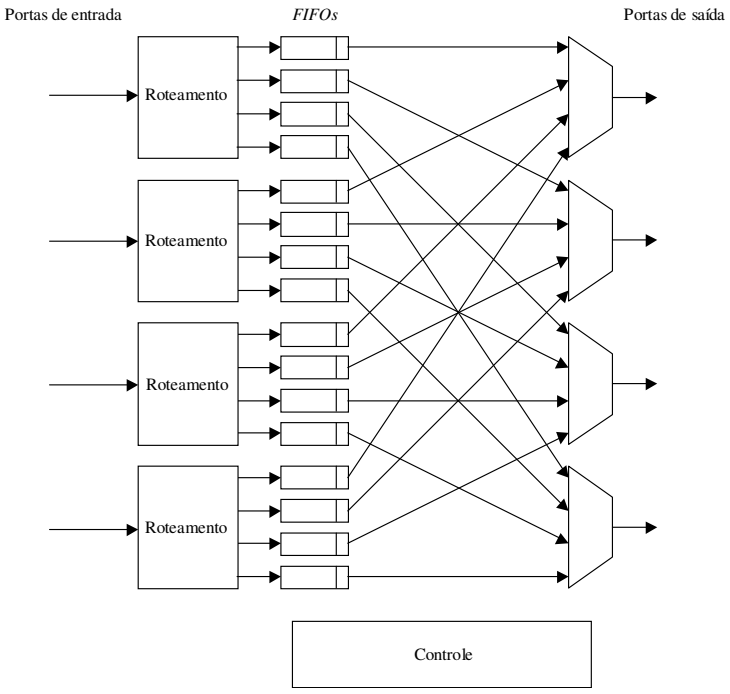


Figura 27 – *Buffers FIFO* na saída.

O tamanho médio das filas de sistemas que utilizam *buffer* na saída é menor que o tamanho médio do sistema equivalente utilizando *buffers* na entrada, no caso de *buffers FIFO*. A razão disso é que sistemas que utilizam *buffers FIFO* na entrada, precisam lidar com o problema do bloqueio *HOL*, o que não ocorre com a utilização de memorização na saída [KAR86].

4.3 Memorização centralizada compartilhada

Neste caso, utiliza-se um *buffer* centralizado no roteador para armazenar os pacotes bloqueados de todas as portas de entrada. Este *buffer* é denominado *CBDA* (*Centrally-Buffered, Dynamically-Allocated*), pois seu espaço de endereçamento é dinamicamente distribuído entre os pacotes bloqueados. Sua largura de banda, no pior caso, deve ser igual à soma das larguras de banda de todas as portas, ou seja, em um roteador *N*x*N*, o *buffer* deve possuir 2*N* portas de modo a permitir *N* acessos simultâneos de leitura e *N* acessos simultâneos de escrita [TAM92].

Esta abordagem oferece uma utilização do espaço de memória melhor do que aquelas proporcionadas pelas abordagens nas quais esse espaço é prévia e estaticamente alocado às portas de entrada. Porém estudos mostram que com compartilhamento completo do *buffer*, se uma saída estiver em uso por uma entrada e outra entrada com pacotes destinados a essa saída continuar a receber dados, isto poderá levar ao preenchimento do *buffer*, afetando as outras comunicações. Esse problema é semelhante ao bloqueio *HOL* e pode ser evitado através da limitação do espaço alocável para cada porta [REE87].

A Figura 28 ilustra o roteador *SPIN*, o qual utiliza memorização centralizada compartilhada, além da memorização na entrada.

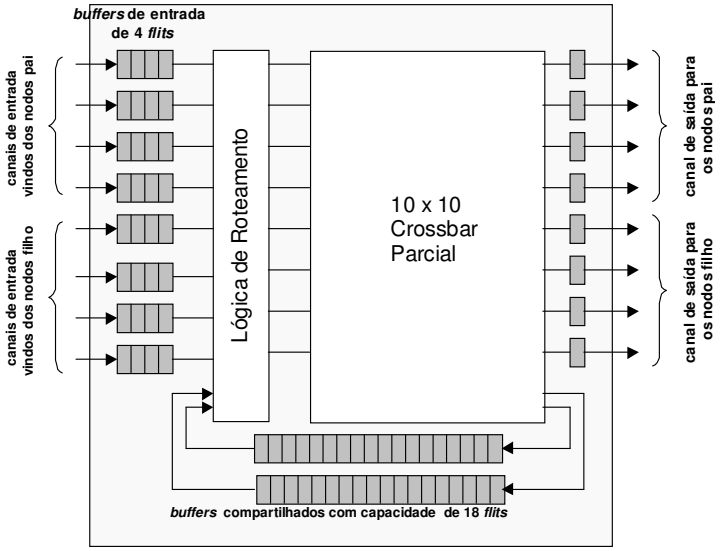


Figura 28 – Arquitetura do roteador *SPIN*.

Esse roteador é utilizado em uma rede com topologia árvore gorda. Os pacotes que são direcionados aos nodos filhos podem utilizar os *buffers* compartilhados com capacidade para 18 *flits* cada. Eles reduzem a contenção de saída, oferecendo uma forma de armazenamento centralizado aos pacotes bloqueados. Tal técnica é mais eficiente em termos de projeto de hardware do que simplesmente aumentar o tamanho de todos os *buffers* de entrada, porque na prática somente algumas das entradas estarão sujeitas à contenção. Segundo o autor, o tamanho de *buffer* escolhido trata eficientemente tanto pacotes pequenos quanto grandes.

5 ROTEAMENTO

O algoritmo de roteamento de uma rede determina quais dos possíveis caminhos de um nodo fonte para um nodo destino são usados como rotas e como a rota seguida por cada pacote em particular é determinada. Um dos objetivos mais importantes em relação ao algoritmo de roteamento, é garantir que ele evite problemas como:

- *Deadlock*: dependência cíclica entre roteadores (Figura 29), os quais esperam pela liberação de uma porta de saída de forma que nenhum dos pacotes atinge progresso algum, ficando bloqueados nas chaves. O capítulo 7 explica detalhadamente o problema de *deadlock.*

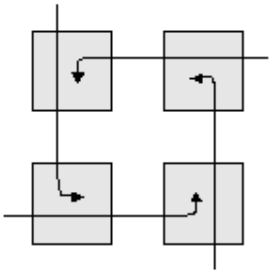


Figura 29 – Exemplo de dependência cíclica ou *deadlock*.

- *Livelock*: situação na qual os pacotes de uma mensagem ficam circulando na rede sem fazer progresso algum em direção ao seu destino. Neste caso não há bloqueio dos pacotes.

A seguir são apresentados os critérios e classificações associados para algoritmos de roteamento.

Quanto ao lugar onde as decisões de roteamento são tomadas

- Roteamento centralizado: os caminhos são estabelecidos por um controlador central.
- Roteamento na origem: a chave origem define todo o caminho a ser percorrido pelo pacote antes de injetá-lo na rede. Nesse roteamento, o cabeçalho do pacote deve carregar toda a informação de roteamento, aumentando o tamanho do pacote [MOH98].
- Roteamento distribuído: cada chave que recebe o pacote decide para onde irá enviá-lo baseado no endereço destino contido no seu cabeçalho.

Quanto à adaptatividade

- Roteamento determinístico: o algoritmo fornece sempre o mesmo caminho entre um determinado par origem-destino.
- Roteamento adaptativo: o caminho é definido em função do tráfego da rede. Este roteamento aumenta o número de caminhos possíveis para um pacote atingir seu destino. Pode ser subdividido em parcialmente adaptativo e totalmente adaptativo. No método parcialmente adaptativo apenas um subconjunto dos caminhos físicos disponíveis é alocado para a

comunicação. No método totalmente adaptativo todos os caminhos físicos da rede podem ser alocados para comunicação. Entretanto, *deadlock* e *livelock* podem ocorrer em algoritmos totalmente adaptativos [NI93], o que acaba limitando seu uso. O uso de algoritmos adaptativos pode tornar a NoC tolerante a falhas.

Quanto ao comprimento relativo do caminho a ser percorrido

- Roteamento mínimo: o algoritmo fornece sempre os canais que certamente levam ao menor caminho entre um par origem-destino, ou seja, a cada chaveamento o pacote se aproxima do seu destino.
- Roteamento não mínimo: o pacote pode seguir qualquer caminho para atingir seu destino. O algoritmo oferece maior flexibilidade em termos de caminhos possíveis, consequentemente diminuindo o bloqueio de pacotes nas chaves. Porém, este tipo de roteamento pode ocasionar *livelock* e o aumento da latência para a entrega das mensagens.

Quanto à implementação

- Baseado em tabela: o roteamento é feito a partir de uma consulta a uma tabela em memória.
- Baseado em máquina de estados: o roteamento é feito a partir da execução de um algoritmo implementado em hardware ou software.

5.1 Algoritmos de roteamento para redes malha

Glass e Ni propuseram algoritmos de roteamento para redes de interconexão que utilizem topologia malha 2D e chaveamento *wormhole* os quais são livres de *deadlock* e *livelock* [GLA94]. Eles foram propostos em versões mínimo e não mínimo quanto ao caminho a ser percorrido.

Em uma malha 2D um pacote pode seguir por uma de quatro possíveis direções: *East, West, North* e *South*. No caminho seguido pelo pacote, oito possíveis mudanças de direção podem ser tomadas (Figura 30a). Como mostrado em [GLA94], se pelo menos duas mudanças de direção forem proibidas (linhas pontilhadas na Figura 30b), torna-se viável a implementação de algoritmos livres de *deadlock*, pois esta condição é suficiente para isto.

a) Oito possíveis mudanças de direção. b) Limitação das mudanças de direção.

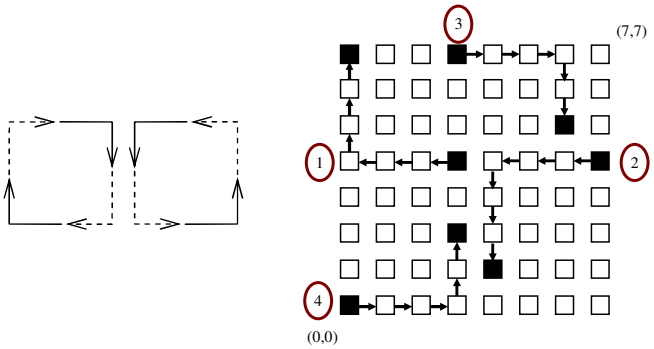
Figura 30 – Mudanças de direção que os *flits* podem tomar no roteamento XY.

A seguir são apresentados quatro algoritmos de roteamento, um determinístico (XY) e três parcialmente adaptativos – *West-first, North-last* e *Negative-first*. Todos os algoritmos apresentados são mínimos, para evitar o aumento da latência na entrega dos pacotes bem como situações de

livelock. As coordenadas relativas aos nodos fonte (*Source*) e destino (*Target*) são representados respectivamente pelas notações (X_S, Y_S) e (X_T, Y_T). A adaptatividade, mesmo parcial, pode reduzir o tempo total para a entrega de um pacote individual, pois em algumas situações ele pode mudar a sua direção, evitando condições de bloqueio.

5.1.1 Algoritmo XY

O algoritmo XY é determinístico. Os *flits* de um pacote são primeiramente roteados na direção X até que cheguem à coordenada X_T, em seguida são roteados na direção Y até que cheguem à coordenada Y_T, como ilustra a Figura 31b. Se alguma porta está em uso por outro pacote, os *flits* permanecem bloqueados na chave até que o caminho seja liberado. Como mostra a Figura 31a, mudanças de direção quando um pacote está na direção Y são proibidas (linhas pontilhadas).



a) Linhas contínuas representam mudanças de direção permitidas.

b) Roteamento XY.

Figura 31 – Algoritmo de roteamento XY.

5.1.2 Algoritmo *West-First*

No algoritmo *West-First* se $X_T <= X_S$, os pacotes são roteados deterministicamente, como no algoritmo XY, (Figura 32,caminhos 1 e 2). Se $X_T > X_S$ os pacotes podem ser roteados de forma adaptativa nas direções *East, North* ou *South* (Figura 32,caminhos 3 e 4). As mudanças de direção proibidas são as duas para *West*.

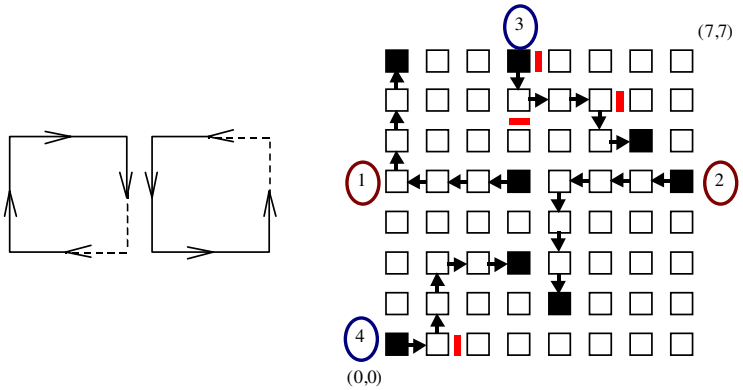


Figura 32 – Algoritmo de roteamento *West-First*.

5.1.3 Algoritmo *North-Last*

No algoritmo *North-Last* se $Y_T \geq Y_S$, os pacotes são roteados deterministicamente (Figura 33, caminhos 1 e 4). Se $Y_T < Y_S$ os pacotes podem ser roteados de forma adaptativa nas direções *West, East,* ou *South* (Figura 33,,caminho 2 e 4). As mudanças de direção proibidas são as duas possíveis quando o pacote está na direção *North*.

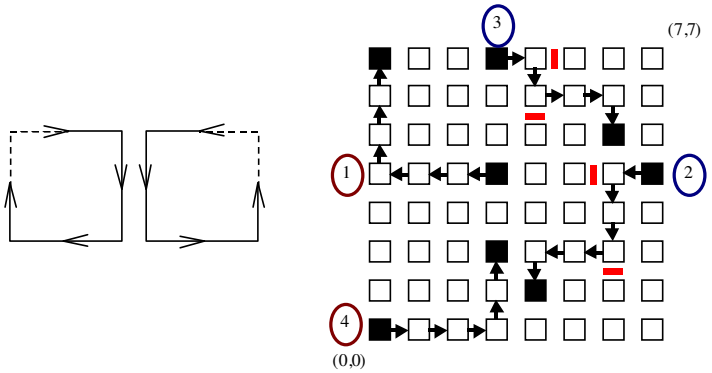


Figura 33 - Algoritmo de roteamento North-Last

5.1.4 Algoritmo *Negative-First*

No algoritmo *Negative-First*, os pacotes são roteados primeiro nas direções negativas, isto é, para as direções *South* ou *West*. Se ($X_T \leq X_S$ e $Y_T \geq Y_S$) ou ($X_T \geq X_S$ and $Y_T \leq Y_S$) então os pacotes são roteados deterministicamente, como mostra a Figura 34, caminho 1 (endereço fonte (3,4) e endereço destino (0,7)) e caminho 3 (endereço fonte (3,7) e endereço destino (6,5)). Todas as demais condições permitem que o roteamento seja adaptativo, como mostram os caminhos 4 e 2. As mudanças de direção proibidas são as duas de uma direção positiva para uma direção negativa.

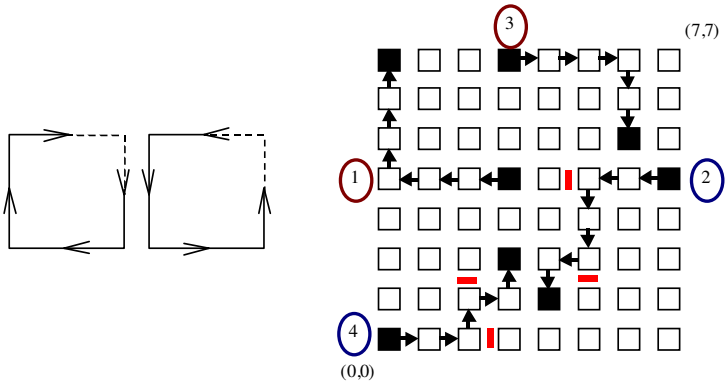


Figura 34 – Algoritmo de roteamento *Negative-First*.

5.2 Avaliação dos algoritmos

A seguir é apresentada uma comparação entre os algoritmos de roteamento apresentados na Seção anterior visando avaliação de desempenho. Os testes foram realizados utilizando-se a ferramenta MAIA para a geração das NoCs, geração de tráfego e análise dos resultados obtidos. Utilizou-se uma topologia malha 5 x 5 com cada roteador enviando 40 pacotes. Todas NoCs

geradas trabalham com *flits* de 16 bits e *buffers* de entrada de 8 posições. As simulações foram realizadas com pacotes de tamanho 10, 100, 1000 e 10000 *flits*, e carga de tráfego igual a 30%, 50%, 70% e 100%.

A carga de tráfego é definida como uma porcentagem da largura de banda do canal usada por cada iniciador da comunicação [AND03]. A Figura 35 ilustra duas cargas de tráfego diferentes, 100% e 50%. Uma carga de 100% é atingida quando todos os núcleos estão enviando dados continuamente, sem interrupção entre pacotes sucessivos. Em situações reais, a carga do sistema é muito menor.

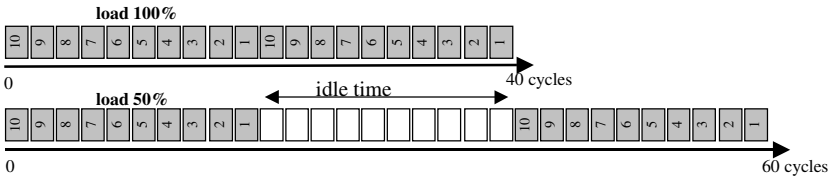


Figura 35 – Interpretação da carga de tráfego.

A Figura a seguir apresenta os resultados obtidos para os diferentes tamanhos de pacotes: (a) 10 *flits*, (b) 100 *flits*, (c) 1000 *flits* e (d) 10000 *flits*.

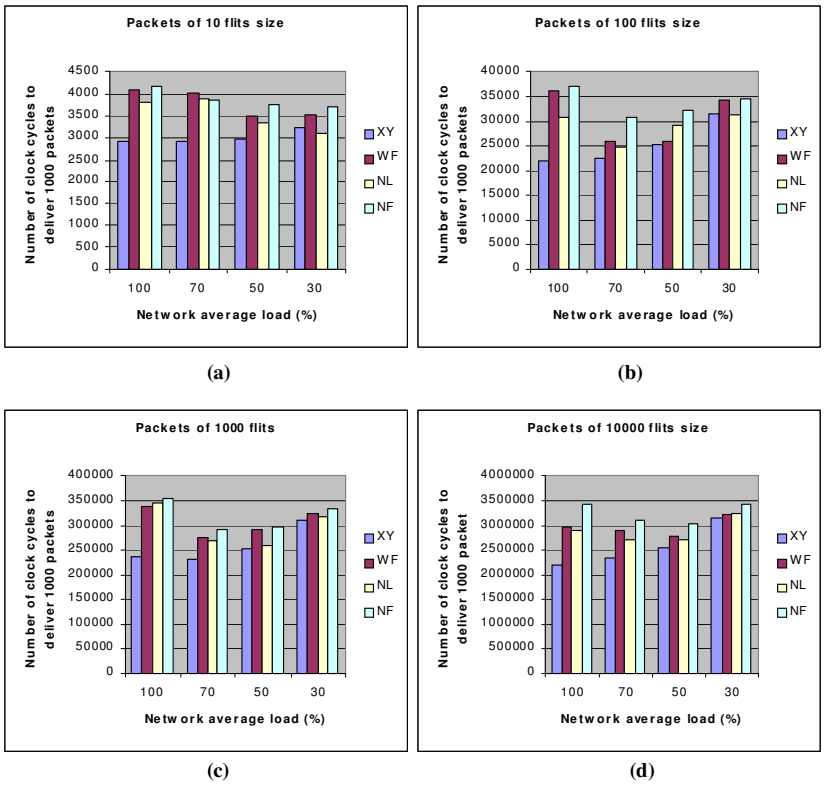


Figura 36 – Comparação dos algoritmos de roteamento.

Os resultados indicam que em termos de tempo total (em ciclos de *clock*) para a entrega de todos pacotes, o algoritmo determinísico XY é mais rápido que os outros três algoritmos parcialmente adaptativos. Os algoritmos adaptativos podem acelerar potencialmente o tempo de entrega de pacotes individuais, mas globalmente resultam em um desempenho inferior ao XY.

Glass e Ni [GLA94], sugerem que reduzir o número de mudanças de direção de uma mensagem pode reduzir o bloqueio e consequentemente melhorar o desempenho. Isto pode ser justificado pelo fato de que o roteamento adaptativo tende a concentrar o tráfego no centro da rede, desta maneira aumentando o número de caminhos bloqueados. O algoritmo *North-Last* apresenta uma pequena vantagem sobre o XY para uma carga de tráfego de 30% e pacotes pequenos(10 e 100 *flits*). Esta situação proporciona um número reduzido de pacotes bloqueados e um tempo maior entre o envio de pacotes. Como o algoritmo XY não pode explorar diferentes caminhos, mesmo que eles estejam disponíveis, o roteamento adaptativo tem vantagem neste caso [MEL04].

6 ARBITRAGEM

Em um roteador de uma rede intra-chip que utiliza chaveamento por pacotes (eg. *store-and-forward* ou *wormhole*), os pacotes chegam pelos seus canais de entrada e são encaminhados aos seus canais de saída. Visto que diferentes pacotes que chegam simultaneamente por diferentes canais de entrada podem requerer um mesmo canal de saída, é preciso escalonar o uso desse canal de saída pelos pacotes requisitantes. Segundo [DUA02], esse escalonamento deve ser realizado de modo a garantir uma utilização balanceada dos canais de saída pelos canais de entrada e assegurar que nenhum pacote fique indefinidamente esperando dentro de um roteador para ser encaminhado ao seu destinatário (problema denominado *starvation*). Essa tarefa é de responsabilidade do árbitro do roteador, sendo que ele exerce um papel importante na maximização do desempenho da rede.

As duas principais formas de implementação de arbitragem são: centralizada ou distribuída. Na forma centralizada, os mecanismos de roteamento e de arbitragem são implementados em um único módulo, enquanto que na forma distribuída o roteamento e arbitragem são realizados de forma independente para cada porta do roteador.

6.1 Arbitragem centralizada

Este esquema consiste de um único módulo que recebe as requisições dos *buffers* das portas de entrada do roteador e seleciona um para que possa ser iniciada a transmissão. Essa organização permite que o árbitro receba requisições de maneira simultânea e independentes. A Figura 37 mostra uma possível organização para esse esquema de arbitragem.

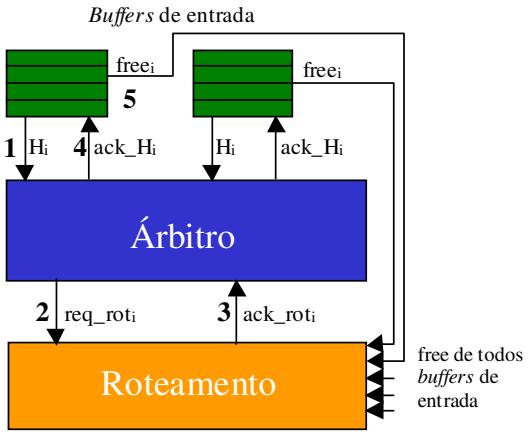


Figura 37 – Arbitragem centralizada.

A seguir é apresentada a operação da arbitragem centralizada baseada na organização apresentada na Figura 37.

1. Cada um dos *buffers* de entrada do roteador, ao receber um novo pacote, indica ao árbitro que está pronto para transmitir.

2. O árbitro seleciona uma das requisições, quando houver mais de uma, e requisita roteamento para o pacote selecionado.

3. Assim que o roteamento é concluído, a conexão é estabelecida e o árbitro é informado.

4. A seguir, o árbitro informa ao *buffer* selecionado no passo 1 que ele pode começar a transmitir os *flits* armazenados do pacote.

5. Após o *buffer* ter enviado todos os *flits* do pacote, ele sinaliza através do sinal *free* e a conexão é encerrada.

Essa abordagem visa minimizar os recursos de conexão entre as portas de entrada e saída. Isto por que ela leva em consideração todos os pacotes que estejam prontos para serem transmitidos nos *buffers* das portas de entrada, bem como o estado atual das portas de saída.

A desvantagem da arbitragem centralizada é que ela impõe maiores restrições quanto à capacidade de roteamento de pacotes no tempo. No entanto, árbitros centralizados são necessários em redes que utilizam roteamento adaptativo ou *buffers* de entrada com múltiplas filas [SAN03]. Árbitros centralizados mantêm a informação sobre o estado de todas as portas de saída (visão global), o que possibilita a adaptatividade, enquanto que os árbitros distribuídos mantêm somente informação sobre o estado da porta a qual ele está associado (visão local).

6.2 Arbitragem distribuída

Na forma distribuída, o roteamento e a arbitragem são realizados de forma independente para cada porta do roteador. Cada porta de entrada possui um módulo de roteamento associado a ela, juntamente com um módulo de arbitragem na sua porta de saída, sendo que as portas são bidirecionais. Cada módulo de arbitragem possui uma visão local, limitada apenas à porta de saída a qual está associado. A Figura 38 ilustra essa organização.

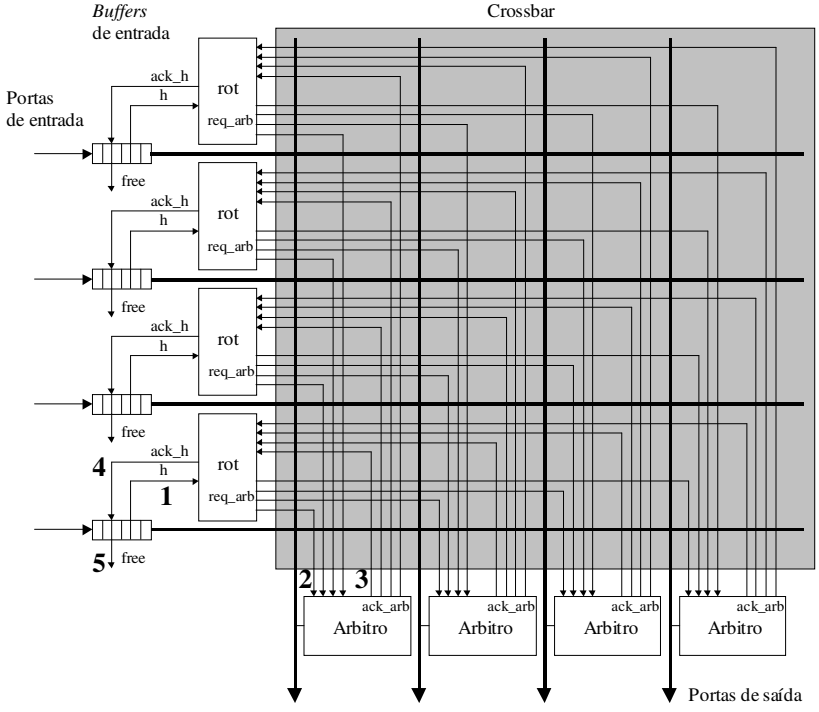


Figura 38 – Arbitragem distribuida.

Associada a cada um dos *buffers* de entrada, tem-se uma lógica de roteamento, a qual possui uma linha de requisição para cada um dos árbitros das portas de saída e recebe uma linha de resposta de cada um deles. A seguir é apresentada a operação da arbitragem distribuída baseada na organização apresentada na Figura 38.

1. Cada um dos *buffers* de entrada, ao receber um novo pacote, indica à lógica de roteamento (rot) que está pronto para transmitir.
2. A lógica de roteamento seleciona uma porta de saída, baseada no cabeçalho do pacote, e então requisita arbitragem para o árbitro dessa porta.
3. O árbitro seleciona uma das requisições, quando houver mais de uma, estabelece a conexão entre o *buffer* e a porta de saída e sinaliza para a lógica de roteamento, indicando que o pacote já pode ser transmitido.
4. A lógica de roteamento repassa para o *buffer* o sinal recebido, o qual inicia a transmissão dos *flits* armazenados do pacote.
5. Após o *buffer* ter enviado todos os *flits* do pacote, ele sinaliza isto através do sinal *free* e a conexão é desfeita.

A abordagem distribuída permite a construção de roteadores mais rápidos, o que proporciona um aumento da capacidade de roteamento dos pacotes. Contudo, ela apresenta limitações quanto à sua aplicação em redes com algoritmos de roteamento adaptativos. Considere um algoritmo de roteamento adaptativo, onde um dado roteador pode enviar solicitação simultânea a mais de um árbitro [CUL98]. Se pelo menos dois dos árbitros requisitados atenderem à solicitação do roteador, uma saída terá que ser escolhida e a outra ficará ociosa, diminuindo a taxa de utilização do *crossbar*. Este problema é resolvido se o árbitro tiver uma visão global, como na forma de implementação centralizada. A arbitragem distribuída é mais adequada a redes que utilizam roteamento determinístico.

6.3 Daisy chain

Neste esquema a linha com a informação da garantia de transmissão (*grant*) parte do árbitro e está ligada serialmente a todas as portas formando uma corrente. A porta de mais alta prioridade é a que está ligada diretamente ao árbitro, e quanto mais longe do árbitro menor é a prioridade. Uma porta de alta prioridade que deseja transmitir, simplesmente intercepta o sinal da garantia de transmissão, não permitindo que ele chegue até as portas de menor prioridade. A Figura 39 mostra como é a organização da arbitragem *daisy chain*.

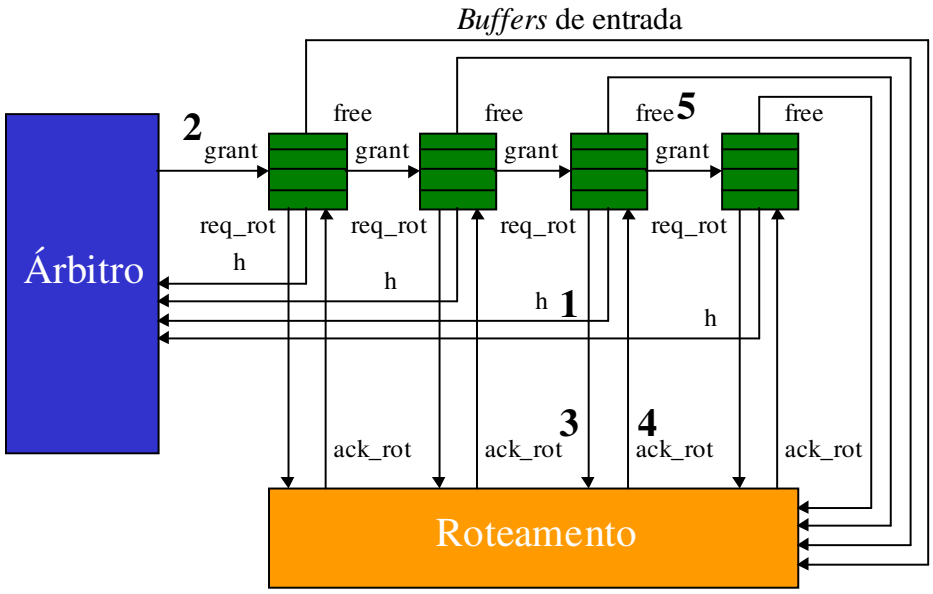


Figura 39 – Arbitragem *daisy chain*.

A seguir é apresentada a operação da arbitragem distribuída baseada na organização apresentada na Figura 39.

1. Cada um dos *buffers* de entrada, ao receber um novo pacote, indica ao árbitro que deseja transmitir e espera pelo sinal da garantia de transmissão (*grant*).
2. O árbitro passa o sinal de garantia para o primeiro *buffer* da corrente.
3. Ao receber o sinal da garantia de transmissão, o *buffer* requisita roteamento para o seu pacote.
4. Assim que o roteamento é concluído, a conexão é estabelecida e o *buffer* é informado de que pode começar a transmissão do pacote.
5. Após o *buffer* ter enviado todos os *flits* do pacote, ele sinaliza através do sinal *free* e a conexão é desfeita.

A vantagem desta organização é a simplicidade enquanto que a desvantagem está no fato de que ela não garante que todas as portas conseguirão transmitir, pois uma porta de baixa prioridade pode ter seu acesso postergado indefinidamente, causando *starvation*. Além disso, o esquema *daisy chain* aumenta o tempo de roteamento dos pacotes, em função da necessidade de propagar o sinal de garantia de uso a todas as portas de entrada. Por exemplo: se a porta de menor prioridade (mais longe do árbitro) for a única que deseja transmitir, ela tem de esperar que o sinal da garantia saia do árbitro e passe por todas as demais portas até que chegue a ela.

6.4 Políticas de arbitragem

Durante a operação da NoC, múltiplos canais de entrada em um roteador podem conter pacotes prontos para serem transmitidos. A decisão de qual dos canais será escolhido para iniciar a transmissão é baseada em uma política de arbitragem. Essa política pode se basear em várias características do pacote e do roteador como: tamanho do pacote, localização do nodo fonte do pacote, localização do nodo destino do pacote, o momento em que o pacote chegou no roteador, o momento em que o pacote saiu do nodo fonte, os canais de entrada já escolhidos anteriormente, etc.

Algumas dessas políticas são: prioridade estática, *Round-Robin*, randômica, *Oldest-first*. Cada uma dessas políticas tem diferentes características quanto ao desempenho e à complexidade de implementação.

- *Prioridade estática*: o mais simples de ser implementado, constituído por um circuito codificador de prioridade simples. Cada requisição considerada pelo árbitro tem um nível de prioridade fixo. Desta forma, dependendo do tráfego da rede, uma requisição com menor prioridade pode ficar sempre esperando para ser atendida, vindo a sofrer *starvation*. Essa política é intrínseca à organização *daisy chain*.
- *Round-Robin*: pode ser visto como uma solução para os problemas da prioridade estática. Essa política utiliza um esquema de prioridades dinâmicas, proporcionando um serviço mais justo que a prioridade estática. Em geral, políticas mais justas possuem um melhor desempenho. Pode ser implementado através de uma fila circular, baseado em um codificador de prioridade programável.
- *Randômica*: essa política seleciona um dos canais de entrada de maneira aleatória, ou seja, sem a necessidade de se basear em informação alguma relativa à rede. Tem como principal vantagem a baixa latência na escolha de um canal.
- *Oldest-first*: nessa política, o canal escolhido é o que está há mais tempo querendo transmitir. Essa política tende a ter a mesma latência média na entrega de pacotes que a randômica, porém reduz significantemente a variância (diferença entre tempos mínimo e máximo de entrega de pacotes) [DAL90]. Uma maneira simples de implementar é manter um contador para cada canal de entrada, o canal escolhido será sempre o que tiver o contador com valor maior.

6.5 Arbitragem centralizada versus arbitragem distribuída

Essa Seção apresenta resultados comparativos entre arbitragem centralizada e arbitragem distribuída utilizando controle de fluxo *Handshake* e baseado em créditos. A comparação foi feita utilizando-se a rede HERMES, na qual implementou-se as duas formas de arbitragem. Na arbitragem centralizada utilizou-se a política *Round-Robin* e na distribuída utilizou-se *Round-Robin* e *Oldest-first*.

Os testes foram realizados utilizando-se a ferramenta MAIA para a geração das NoCs, geração de tráfego e análise dos resultados obtidos. A topologia usada foi malha 3 x 3, com cada roteador enviando 1000 pacotes, totalizando 9000 pacotes. As NoCs geradas trabalham com *flits* de 16 bits e *buffers* de entrada de 8 posições. A simulação foi realizada com pacotes de tamanho de 10 *flits* e carga de tráfego igual a 100%.

A Tabela 3 e a Tabela 4 apresentam as seguintes estatísticas geradas pela ferramenta MAIA: (*i*) latência mínima, máxima e média de transmissão de um pacote; (*ii*) desvio padrão da latência; (*iii*) tempo total para a transmissão de todos os pacotes. Todos os tempos apresentados são em ciclos de *clock*. O resultado de área foi obtido através da ferramenta de síntese lógica *Lonardo Spectrum* tendo como alvo um dispositivo FPGA XC2V2000 da Virtex.

Tabela 3 - Resultados da comparação das formas de arbitragem utlizando *Handshake* com controle de fluxo.

	Handshake			
	Arbitragem centralizada e política *Round-Robin*	Arbitragem distribuída e política *Round-Robin*	Arbitragem distribuída e política *Oldest-first*	Arbitragem distribuída e política *Oldest-first* mais *Round-Robin*
Latência mínima	31 ciclos de *clock*	31 ciclos de *clock*	31 ciclos de *clock*	31 ciclos de *clock*
Latência máxima	550 ciclos de *clock*	537 ciclos de *clock*	413 ciclos de *clock*	522 ciclos de *clock*
Latência média	97,62 ciclos de *clock*	94,24 ciclos de *clock*	92,88 ciclos de *clock*	92,45 ciclos de *clock*
Desvio padrão da latência	53,52	47,50	43,51	43,16
Tempo total de transmissão	52562 ciclos de *clock*	50099 ciclos de *clock*	49673 ciclos de *clock*	47958 ciclos de *clock*
Área	36,67% CLB Slices	46,72% CLB Slices	72,01% CLB Slices	80,98% CLB Slices

Tabela 4 - Resultados da comparação das formas de arbitragem utlizando controle de fluxo baseado em créditos.

	Créditos			
	Arbitragem centralizada e política *Round-Robin*	Arbitragem distribuída e política *Round-Robin*	Arbitragem distribuída e política *Oldest-first*	Arbitragem distribuída e política *Oldest-first* mais *Round-Robin*
Latência mínima	20 ciclos de *clock*	20 ciclos de *clock*	20 ciclos de *clock*	20 ciclos de *clock*
Latência máxima	386 ciclos de *clock*	284 ciclos de *clock*	238 ciclos de *clock*	253 ciclos de *clock*
Latência média	70,9 ciclos de *clock*	60,97 ciclos de *clock*	60,83 ciclos de *clock*	61,38 ciclos de *clock*
Desvio padrão da latência	41,078	27,72	27,51	27,53
Tempo total de transmissão	38933 ciclos de *clock*	34027 ciclos de *clock*	33817 ciclos de *clock*	33702 ciclos de *clock*
Área	39,93% CLB Slices	48,53% CLB Slices	73,91% CLB Slices	84,31% CLB Slices

Os resultados indicam que em termos de latência total para a entrega de todos pacotes a arbitragem distribuída é mais rápida que a centralizada, sendo que a versão distribuída com política de *Round-Robin* mais *Oldest-first* foi a que apresentou o melhor desempenho.

Em relação aos demais dados, as versões com arbitragem distribuída também apresentaram melhores resultados de desempenho que a arbitragem centralizada, porém o custo de área aumenta significativamente.

7 CANAIS VIRTUAIS

Como visto na Seção 2.1 as redes de interconexão podem ser agrupadas em duas classes principais, as redes diretas e as redes indiretas. Durante todo esse trabalho, assumiu-se redes diretas, onde cada nodo de chaveamento (roteador) possui um nodo de processamento associado. A topologia mais utilizada nesse tipo de rede é a malha, a qual também tem sido usada durante todo esse trabalho, bem como o modo de chaveamento *wormhole*, o qual possibilita a multiplexação de canais físicos em canais virtuais.

As redes de interconexão são compostas por dois tipos de recursos: nodos de chaveamento e canais físicos. Em geral, cada nodo de chaveamento em uma rede de interconexão possui um conjunto de *buffers* e um *crossbar*. Em redes *wormhole* com memorização na entrada, para cada canal físico, há um *buffer* de entrada associado, o qual armazena os *flits* recebidos para que depois os mesmos sejam posteriormente enviados. Enquanto um pacote A tem um *buffer* alocado, nenhum outro pacote poderá utilizar o canal a ele associado até que o pacote A o libere. Se o pacote A ficar bloqueado na rede enquanto ocupa o *buffer*, o canal associado fica ocupado e nenhum outro pacote pode usá-lo. A Figura 40 ilustra a situação descrita.

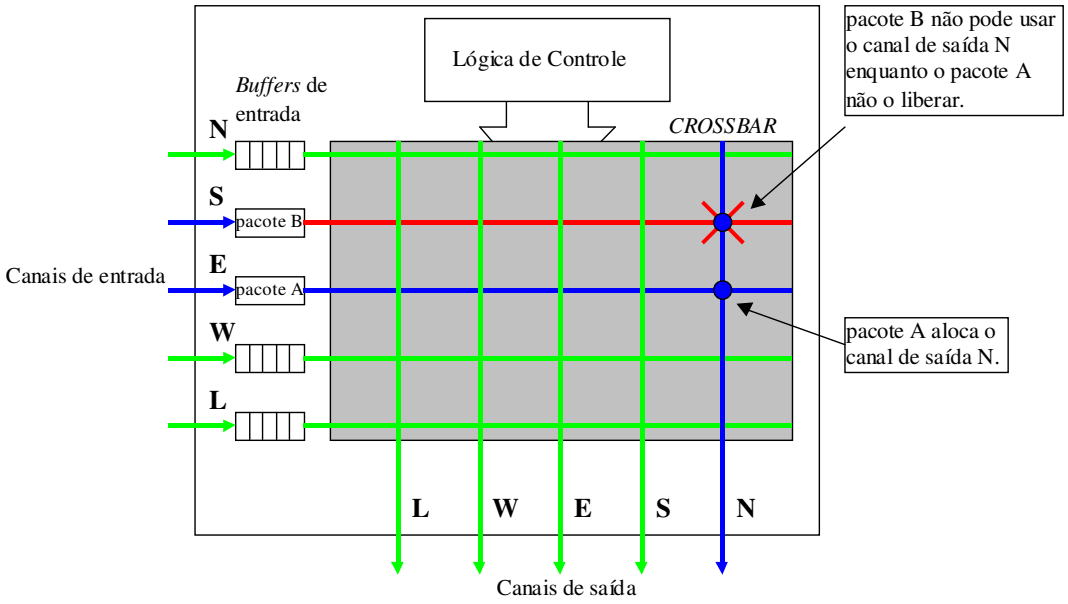


Figura 40 - Situação de bloqueio em uma rede de interconexão.

Canais virtuais eliminam a alocação de um único pacote por canal físico, dividindo o último em vários canais virtuais, um para cada pacote associado ao canal físico. Se um pacote A aloca um um canal virtual, outro pacote pode alocar um outro canal virtual do mesmo canal físico. A Figura 41 ilustra a adição de dois canais virtuais para cada canal físico à rede de interconexão da Figura 40. Enquanto o pacote A é transmitido por um dos canais virtuais do canal físico N, o pacote B pode ser transmitido pelo outro canal virtual do mesmo canal físico.

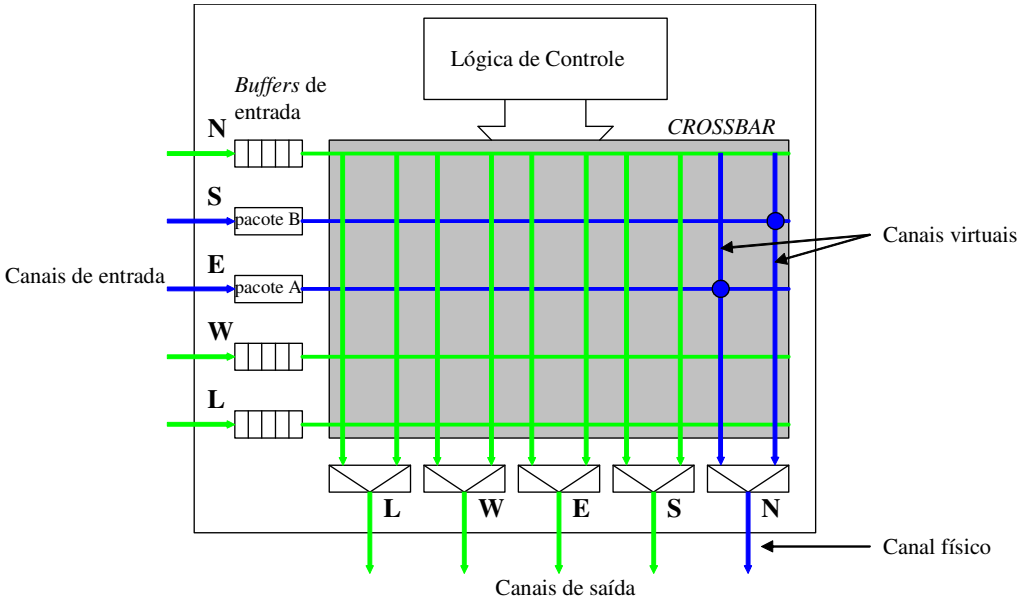


Figura 41 – Canais físicos divididos em canais virtuais.

Adicionar canais virtuais a uma rede de interconexão é análogo a adicionar pistas (*lanes*) em uma estrada. Uma rede sem canais virtuais é formada por estradas de uma só pista. Nesse tipo de rede, um pacote bloqueado ocupando um canal bloqueia todos os demais pacotes que precisam desse canal. Adicionando canais virtuais à rede diminui-se o número de pacotes bloqueados, ou seja, reduz-se o congestionamento na rede.

O objetivo de uma rede de interconexão é prover a maior largura de banda possível, a um custo aceitável em termos de consumo de área. Segundo [DAL90] o custo de adicionar canais virtuais corresponde apenas ao acréscimo de lógica de controle. Ao final deste capítulo esse custo é avaliado.

Os canais virtuais foram propostos pela primeira vez em [DAL86], com o intuito de prevenir *deadlocks* que podem ocorrer em redes que utilizam chaveamento *wormhole* e roteamento que permita qualquer mudança de direção. O *deadlock* ocorre quando existe uma dependência cíclica de recursos na rede, o que é melhor entendido pelo exemplo ilustrado a seguir.

A Figura 42 mostra parte de uma rede de interconexão com quatro roteadores (1, 2, 3 e 4) possuindo cada um quatro portas bidirecionais (N, S, E, e W) conectadas a um núcleo *crossbar*. Cada um dos roteadores tem um pacote a ser transmitido.

- O roteador 1 tem um pacote A, que tem como destino o roteador 4.
- O roteador 2 tem um pacote B, que tem como destino o roteador 3.
- O roteador 3 tem um pacote C, que tem como destino o roteador 2.
- O roteador 4 tem um pacote D, que tem como destino o roteador 1.

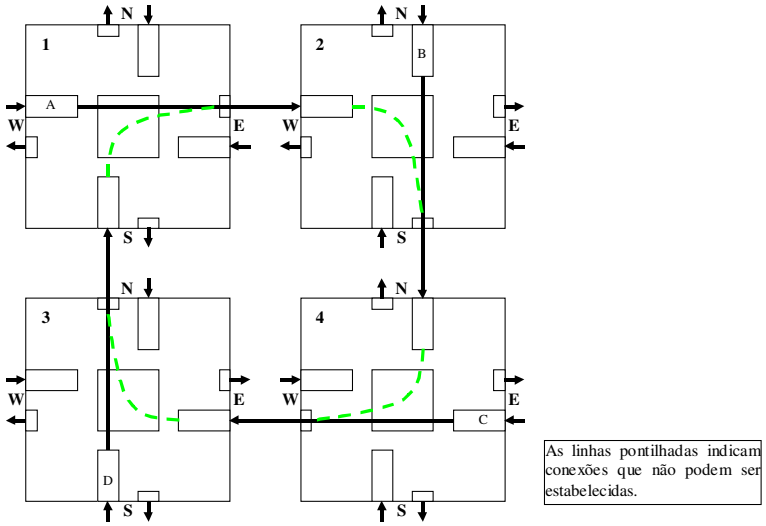


Figura 42 – Dependência cíclica que origina o *deadlock*.

A dependência cíclica ocorre porque existem quatro pacotes sendo transmitidos simultaneamente na rede, e cada um dos pacotes utiliza um canal que é requisitado por outro pacote para que este último atinja seu destino. Por exemplo, os *flits* do pacote A começam a ser transmitidos para o roteador 2 pela porta de saída E. Chegando no roteador 2 pela porta de entrada W, o pacote requisita a porta de saída S para chegar ao seu destino, porém esta porta já está em uso pelo pacote B. O mesmo ocorre com os demais pacotes que estão sendo transmitidos, criando o ciclo que dá origem ao *deadlock*.

A solução para o problema apresentado na Figura 42 pode ser obtida dividindo cada um dos canais físicos em dois canais virtuais e definindo uma ordem total entre os *lanes*, quebrando assim a dependência cíclica, como ilustra a Figura 43. Retomando o exemplo do pacote A, assim que seus *flits* chegam pela porta de entrada W do roteador 2, ele requisita a porta de saída S. Essa porta já está com um dos canais virtuais ocupado pelo pacote B, porém, o outro canal virtual esta disponível, logo o pacote A passa a utilizá-lo para atingir seu destino.

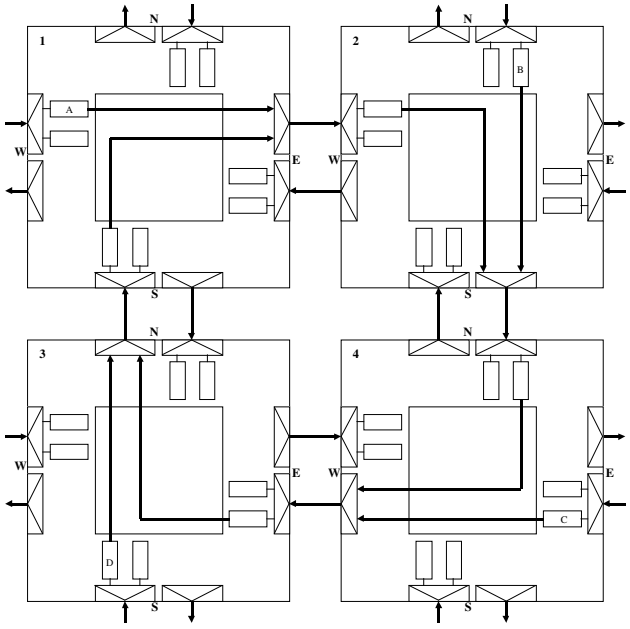


Figura 43 – Quebrando a dependência cíclica com o uso de canais virtuais.

Uma rede convencional organiza os *buffers* de entrada (Seção 4.1) sob a forma de um uma fila (*FIFO*), a qual só pode armazenar pacotes inteiros em seqüência. Se um pacote for bloqueado por conta de uma colisão, o canal físico também ficará bloqueado devido ao bloqueio de cabeça de fila (*HOL*) e nenhum outro pacote poderá utilizar esse canal, como ilustra a Figura 44(a).

Já uma rede que usa canais virtuais, organiza os *buffers* em pequenas filas de profundidade menor como mostra a Figura 44(b). Com essa organização obtém-se uma coleção de *buffers* que podem ser alocados independentemente uns dos outros. Isso aumenta a flexibilidade na alocação de *buffers*, ameniza ou resolve o problema de bloqueio *HOL*, melhora a utilização do canal e aumenta o *throughput*.

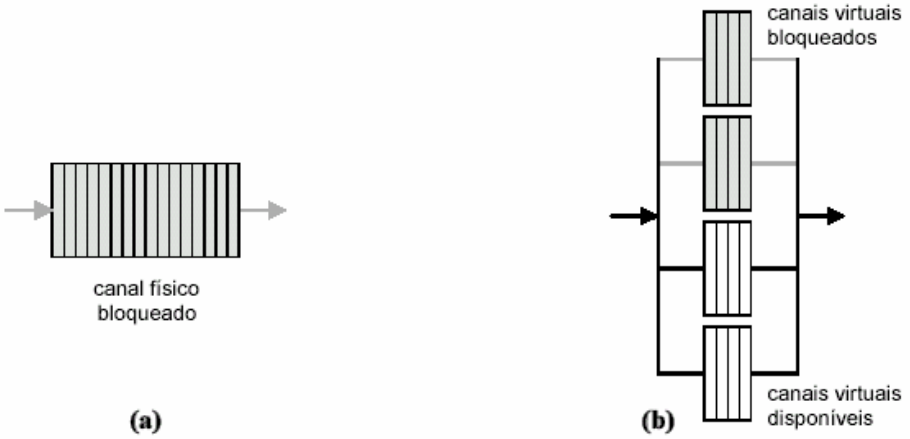


Figura 44 – Organizações de *buffers* de entrada.

O controle de fluxo com canais virtuais é realizado em dois níveis. No nível de pacote, os pacotes são atribuídos aos canais virtuais (ou *lanes*). No nível de *flit*, os *flits* são atribuídos aos *buffers* [DAL90].

A atribuição de *lane* é feita pelo nodo transmissor, o qual monitora todos os canais de saída, mantendo a informação sobre a disponibilidade dos *lanes*. Para um determinado pacote no *buffer* de entrada do nodo transmissor, é selecionado um canal físico de saída particular com base no destino do pacote e no resultado do algoritmo de roteamento. A lógica de controle de fluxo então atribui esse pacote a algum *lane* livre no canal físico selecionado. Se todos os *lanes* estão em uso, o pacote é bloqueado.

O controle no nível de *flit* é usado após a atribuição de um *lane* a um pacote. Tipicamente, em redes com controle de fluxo bloqueante, é necessário que o nodo receptor envie ao nodo transmissor uma informação a respeito da disponibilidade de espaço para armazenar um novo *flit*. Essa informação pode ser transferida através de fios dedicados à essa função, como no caso de um controle de fluxo baseado em créditos (Seção 3.2), ou através de uma banda adicional no canal oposto ao canal de dados [INM89a], o que exige o uso de enlaces bidirecionais. Além disso, no canal de envio no qual os *flits* de dados são transmitidos para o receptor, a informação relativa ao *lane* selecionado deve ser enviada juntamente com esses *flits*.

Em [DAL90] foram realizadas simulações de redes indiretas multiestágio a fim de medir o efeito dos canais virtuais no desempenho da rede. As simulações foram feitas em nível de *flit* e a quantidade de armazenamento total foi mantida constante em 16 *flits* por nodo, enquanto que o número de canais virtuais foi variado de 1 a 16 em potências de dois. O tamanho dos pacotes foi

fixado em 20 *flits*. Os resultados mostraram que o aumento do número de *lanes* pode levar a incrementos da vazão (ou *throughput*) de até 3,5 vezes em relação à redes sem canais virtuais. As simulações também indicaram que o acréscimo de *lanes* tem pouco efeito sobre a latência abaixo da vazão de saturação de uma rede convencional.

7.1 Canais virtuais na rede Hermes

A rede Hermes original (Seção 2.6) possui um único *lane* por canal físico, o que permite a ela transmitir somente um pacote por vez em cada um dos seus canais físicos. A sua lógica de controle associa somente um canal de saída a um *buffer* de entrada, permitindo até cinco conexões simultâneas. A Figura 45 mostra o roteador da rede Hermes transmitindo cinco pacotes simultaneamente.

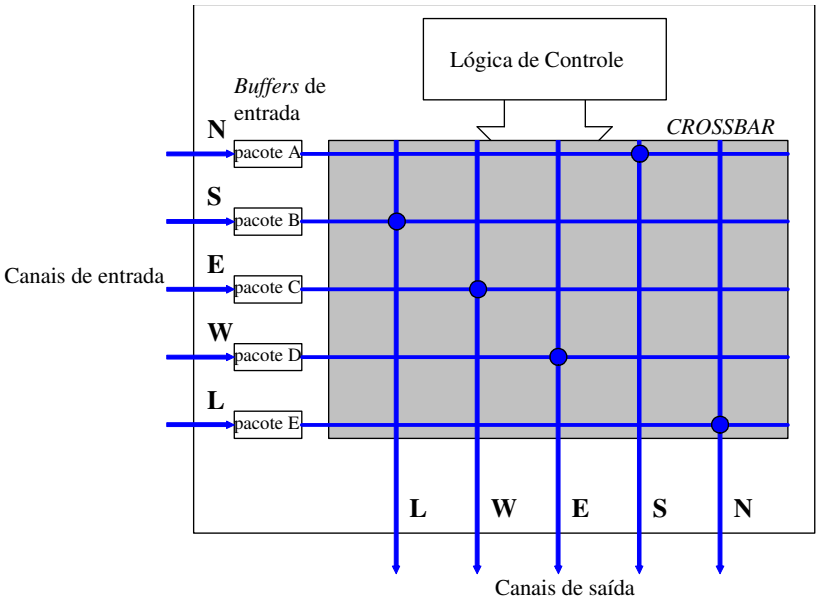


Figura 45 – Cinco pacotes sendo transmitidos simultaneamente na rede Hermes.

Este capítulo apresenta duas implementações de canais virtuais sobre a rede Hermes. Uma mantendo a estrutura de um *buffer* único por entrada enquanto a outra adota uma estrutura de *buffers* multiplexados por entrada. A implementação com *buffer* único apresentou problemas de *deadlock* devido à sua arquitetura de *buffers*, os quais serão explicados na Seção a seguir. Por essa razão optou-se por criar uma nova arquitetura de *buffers* para sanar o problema, após algumas tentativas de tentar resolvê-lo. Mesmo apresentando problemas de *deadlock*, a implementação de canais virtuais com *buffer* único foi incluída com o intuito de enriquecer o estudo a respeito de canais virtuais.

7.1.1 Canais virtuais com *buffer* único

A idéia principal desta implementação é adicionar mais um *lane* por canal físico, mantendo a estrutura original da Hermes de um *buffer* único por entrada (Seção 4.1). Desta maneira, os *buffers* devem estar preparados para armazenar simultaneamente *flits* de até dois pacotes fora de seqüência.

Essa proposta visa realizar poucas modificações na lógica de controle (arbitragem e roteamento) e no roteador para dar suporte ao novo *lane* inserido.

7.1.1.1 Roteador

Os roteadores realizam a transferência de pacotes entre os núcleos. O roteador da Hermes possui uma lógica de controle que engloba arbitragem e roteamento, portas de comunicação para outros roteadores e para um núcleo. As portas de comunicação incluem canais de entrada e de saída. Os canais de entrada possuem *buffers* para o armazenamento temporário de *flits*. As portas possuem ainda um controlador de enlace para a implementação do protocolo físico de comunicação.

O modo de chaveamento w*ormhole*, utilizado pela Hermes, possibilita a multiplexação dos canais físicos. Adicionando-se mais um *lane* ao roteador original da Hermes (Figura 45) e multiplexando os canais físicos, tem-se a arquitetura da Figura 46. Observa-se que os *lanes* foram acrescentados somente nas portas que se conectam com outros roteadores (N, S, E e W). Os endereços dos roteadores são compostos pelas coordenadas XY da rede de interconexão, onde X é a posição horizontal e Y a posição vertical. A distribuição de endereços aos roteadores é necessária para a execução do algoritmo de roteamento.

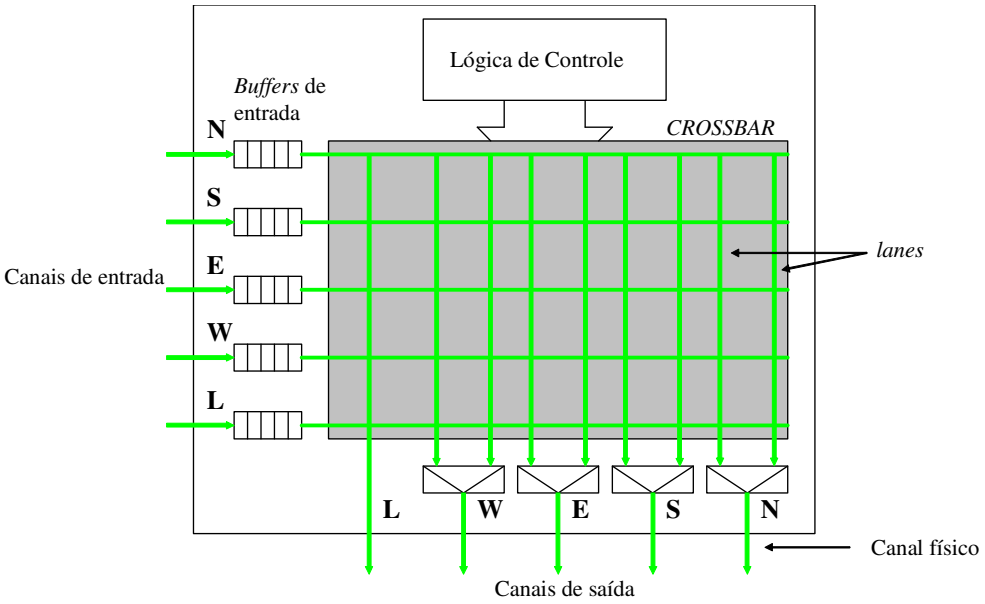


Figura 46 – Roteador com dois *lanes* por canal físico.

7.1.1.2 *Buffers* de entrada

Quando a aplicação do algoritmo de roteamento resulta no bloqueio dos *flits* de um pacote, ocorre uma perda de desempenho em toda rede de interconexão, porque os *flits* são bloqueados não somente no roteador atual, mas em tantos roteadores intermediários quanto forem necessários para armazenar todos os *flits* do pacote. Para diminuir a perda de desempenho, utiliza-se um módulo *Buffer* em cada porta de entrada do roteador, reduzindo o número de roteadores afetados com o

bloqueio dos *flits* de um pacote. O módulo *Buffer*, para dar suporte a canais virtuais com *buffer* único, contém os três *buffers* apresentados a seguir e uma lógica para o controle dos mesmos.

Como mostrou a Figura 46, apesar do roteador possuir dois *lanes*, tem-se apenas um *buffer* por canal físico e não um *buffer* por canal lógico (*lane*). Para que um *buffer* único por canal físico consiga armazenar *flits* de dois pacotes diferentes ao mesmo tempo, dois *buffers* auxiliares foram necessários. Esses *buffers* armazenam informações relativas aos *flits* contidos no *buffer* de *flits* (*flit_buffer*). Os *buffers* auxiliares são:

- *lane_buffer* que armazena informação sobre o *lane* de entrada.
- *type_buffer* que armazena informação sobre o tipo dos *flits*.

Esses *buffers* tem o mesmo tamanho do *flit_buffer* e a informação contida em uma posição X é relativa ao *flit* contido na posição X do *flit_buffer*, como exemplifica a Figura 47.

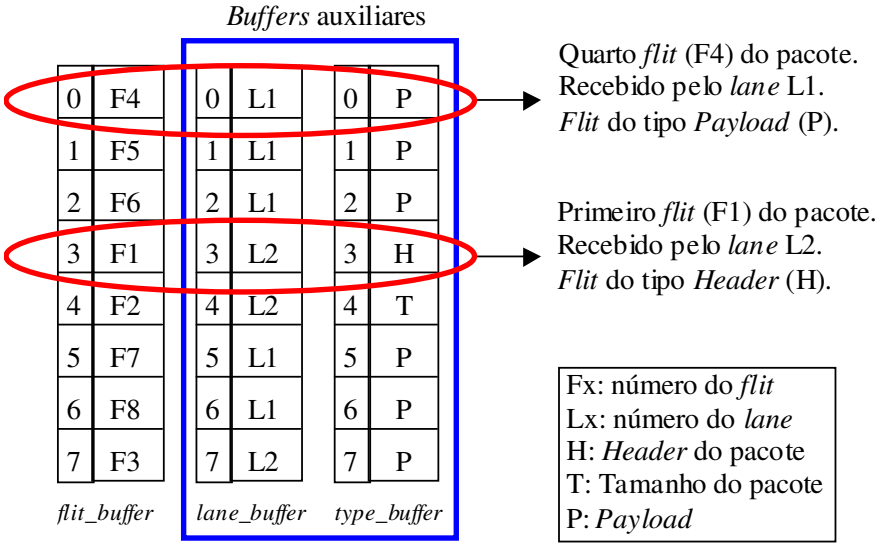


Figura 47 – *Buffers* contidos no módulo *Buffer*.

Esses *buffers* funcionam como *FIFOs* circulares. Para o controle deles utilizam-se três ponteiros: *first*, *prox* e *last*.

- *First* aponta para a posição do *buffer* onde se encontra o *flit* a ser consumido.
- *Prox* aponta para a posição do *buffer* onde se encontra o próximo *flit* a ser consumido (*first*+1).
- *Last* aponta para a posição onde deve ser inserido um novo *flit* recebido.

A conexão entre as portas deve ser finalizada quando todos os *flits* do pacote forem enviados. Por este motivo, o roteador possui um contador (*counter_flit*) para cada canal lógico. Esses contadores estão implementados dentro do módulo *Buffer* e são inicializados quando o segundo *flit* do pacote (T: Tamanho do pacote), que contém o número de *flits* restantes do mesmo, chega no roteador. Esses contadores são decrementados a cada *flit* enviado com sucesso. Quando o valor do contador chega a zero a conexão é finalizada e a tabela *free* é atualizada.

A entrada de *flits* no *buffer* de *flits* é controlada pelo transmissor através do controle de fluxo. Neste caso utilizou-se o controle de fluxo baseado em créditos (Seção 3.2). O envio dos *flits* é controlado pela máquina de estados apresentada na Figura 48.

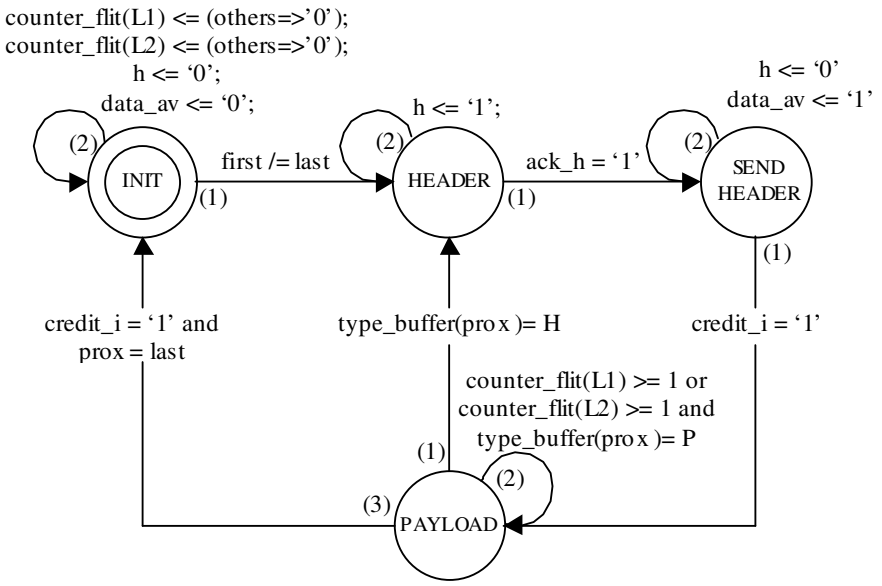


Figura 48 – Máquina de estados do módulo *Buffer*.

- Quando o sinal *reset* é ativado, a máquina de estados avança para o estado INIT. No estado INIT os sinais *counter_flit* (contador de *flits* do corpo do pacote), *h* (que indica requisição de roteamento) e *data_av* (que indica a existência de *flit* a ser transmitido) são inicializados com zero. Se existir algum *flit* na fila, ou seja, os ponteiros *first* e *last* apontarem para posições diferentes, a máquina de estados avança para o estado HEADER.
- No estado HEADER é requisitado o chaveamento (*h<='1'*), porque o *flit* na posição apontada pelo ponteiro *first*, quando a máquina encontra-se nesse estado, é sempre o cabeçalho do pacote. A máquina permanece neste estado até que receba a confirmação do chaveamento (*ack_h='1'*), quando então avança para SENDHEADER.
- Em SENDHEADER é indicado que existe um *flit* a ser transmitido (*data_av='1'*). A máquina de estados permanece em SENDHEADER até receber a confirmação da transmissão (*credit_i='1'*) do cabeçalho. Neste estágio o ponteiro *first* aponta para o segundo *flit* do pacote e avança para o estado PAYLOAD.
- No estado PAYLOAD os sinais *counter_flit* são carregados e a máquina permanece neste estado enviando *flits* até que todos sejam enviados, quando então a máquina volta para o estado INIT, ou até quando o próximo *flit* do *flit_buffer* for do tipo HEADER, fazendo com que a máquina volte para o estado HEADER. Como o *buffer* de *flits* pode armazenar *flits* de dois pacotes simultaneamente, enquanto a máquina está no estado PAYLOAD enviando os *flits* de um pacote A, ela pode ter armazenado também *flits* de um pacote B. Então, quando é detectado o *flit* de cabeçalho do pacote B, a máquina volta para o estado HEADER para que seja realizado o chaveamento para este pacote.

7.1.1.3 Lógica de controle

A lógica de controle inclui dois módulos: árbitro e lógica de roteamento. O roteador, ao receber pacotes, utiliza uma política de arbitragem para determinar qual pacote será roteado. Em

seguida executa um algoritmo de roteamento para determinar a porta de saída e o *lane* (canal lógico) por onde o pacote deve ser enviado.

Quando o algoritmo de roteamento retorna uma porta de saída e um *lane* livres, a conexão entre a porta de entrada e a porta de saída é estabelecida e são preenchidas as tabelas de chaveamento *port_in, port_out, lane_in* e *free*. Todas as tabelas possuem dois níveis de indexação.

A tabela *port_in* é indexada pela porta de entrada e pelo *lane* de entrada, sendo preenchida com a porta de saída da conexão.

A tabela *port_out* é indexada pela porta de saída e pelo *lane* de saída, sendo preenchida com a porta de entrada.

A tabela *lane_in* é indexada pela porta de entrada e pelo *lane* de entrada, sendo preenchida com o *lane* de saída.

A tabela *free* é indexada pela porta de saída e pelo *lane* de saída, sendo preenchida com o estado da porta: livre (1) ou ocupada (0).

A Figura 49 apresenta um possível chaveamento e como as tabelas são preenchidas, considerado que as portas de entrada W,E e L receberam dados do *lane* L1 do roteador anterior. A porta *Local* recebe dados sempre pelo L1, pois ela possui somente um *lane*.

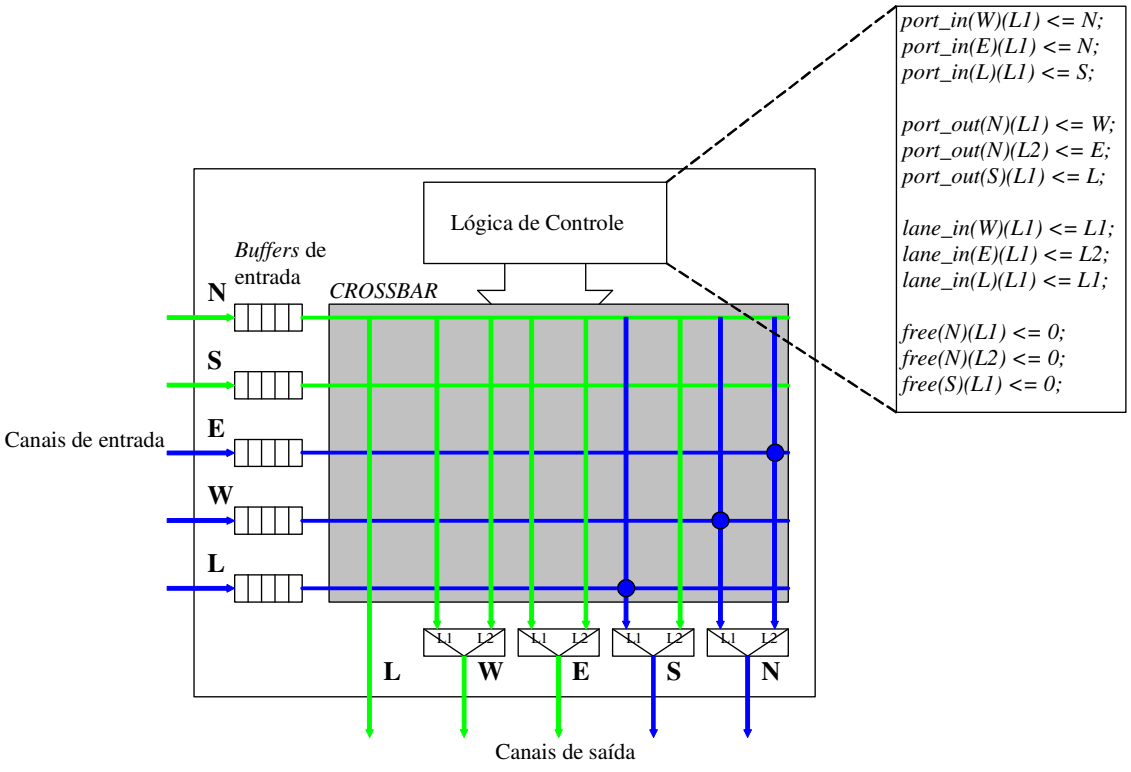


Figura 49 –Chaveamento e as tabelas correspondentes.

Como ilustra a Figura 49:

- A porta de saída *North* está com os dois *lanes* ocupados (*free(N)(L1) = 0* e *free(N)(L2) = 0*), recebendo dados da porta *West* pelo *lane* L1 (*port_out(N)(L1) = W*) e da *East* pelo *lane* L2 (*port_out(N)(L2) = E*).

- A porta de entrada *West* utiliza o *lane* L1 (*lane_in(W)(L1) = L1*) da porta de saída *North* (*port_in(W)(L1) = N*) enquanto que a porta de entrada *East* utiliza o *lane* L2 (*lane_in(E)(L1) = L2*) da mesma porta de saída (*port_in(E)(L1) = N*).
- A porta de entrada *Local* utliliza o *lane* L1 (*lane_in(L)(L1) = L1*) da porta de saída *South* (*port_in(L)(L1) = S*), logo a porta de saída *South* tem o *lane* L1 ocupado (*free(S)(L1) = 0*) pela porta de entrada *Local* (*port_out(S)(L1) = L*).

O *crossbar* realiza os chaveamentos correspondentes às tabelas contidas na lógica de controle.

7.1.1.4 Arbitragem

O roteador suporta até nove conexões, por isso existe a necessidade de um árbitro para determinar qual o pacote deve ser roteado quando mais de um cabeçalho chegar ao roteador em um mesmo instante de tempo.

Após atender uma solicitação, a arbitragem aguarda até que o algoritmo de roteamento seja executado e somente após este período volta a atender solicitações. Se o algoritmo de roteamento não consegue estabelecer uma conexão, a porta de entrada volta a solicitar roteamento ao árbitro.

A Figura 50 ilustra a parte da máquina de estados da lógica de controle correspondente à arbitragem, pois a lógica de controle contém uma máquina de estados única que controla a arbitragem e o roteamento.

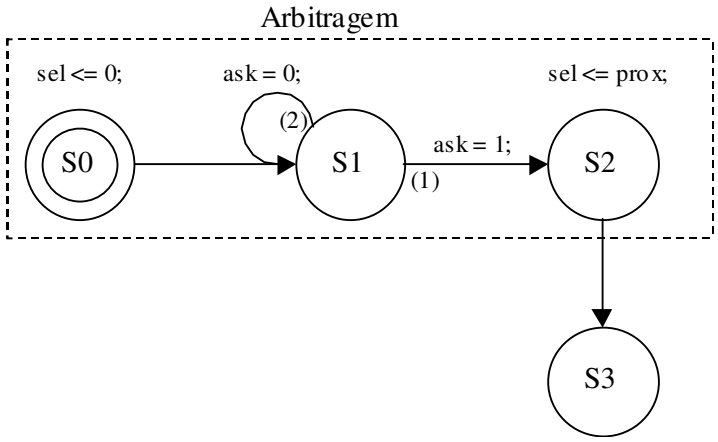


Figura 50 – Parte da máquina de estados da lógica de controle correspondente à arbitragem.

- O estado S0 é o estado de inicialização da máquina. Este estado somente é atingido quando o sinal reset é ativado.
- O estado S1 é o estado de espera por requisição de roteamento. Quando a arbitragem recebe uma ou mais requisições o sinal *ask* é ativado fazendo a máquina avançar para o estado S2.
- No estado S2 a porta de entrada que solicitou roteamento é selecionada. Se houver mais de uma, aquela com maior prioridade é a selecionada. Em seguida a máquina avança para o estado S3 onde é executado o algoritmo de roteamento. O estado S3 é o primeiro estado na máquina da Lógica de controle correspondente ao roteamento.

O exemplo de código fonte em linguagem VHDL para a seleção da porta a ter permissão de

roteamento utilizando a política *Round Robin* é apresentado na Figura 51.

- O sinal *h* corresponde aos pedidos de roteamento das portas de entrada.
- O sinal *sel* corresponde à porta selecionada pelo *Round-Robin*.
- O sinal *prox* corresponde à próxima porta que deve ser a selecionada (*sel*).
- O sinal *header* armazena o cabeçalho do pacote para o qual a porta selecionada requisita roteamento.

```
01 ask <= '1' when h(LOCAL)='1' or h(EAST)='1' or h(WEST)='1' or h(NORTH)='1' or
02         h(SOUTH)='1' else '0';
03
04 incoming <= CONV_VECTOR(sel);
05 header <= data(sel);
06
07        process(sel,h)
08        begin
09                case sel is
10                        when LOCAL=>
11                                if h(EAST)='1' then prox<=EAST;
12                                elsif h(WEST)='1' then  prox<=WEST;
13                                elsif h(NORTH)='1' then prox<=NORTH;
14                                elsif h(SOUTH)='1' then prox<=SOUTH;
15                                else prox<=LOCAL; end if;
16
17                        when EAST=>
18                                if h(WEST)='1' then     prox<=WEST;
19                                elsif h(NORTH)='1' then prox<=NORTH;
20                                elsif h(SOUTH)='1' then prox<=SOUTH;
21                                elsif h(LOCAL)='1' then prox<=LOCAL;
22                                else prox<=EAST; end if;
23
24                        when WEST=>
25                                if h(NORTH)='1' then prox<=NORTH;
26                                elsif h(SOUTH)='1' then prox<=SOUTH;
27                                elsif h(LOCAL)='1' then prox<=LOCAL;
28                                elsif h(EAST)='1' then prox<=EAST;
29                                else prox<=WEST; end if;
30
31                        when NORTH=>
32                                if h(SOUTH)='1' then prox<=SOUTH;
33                                elsif h(LOCAL)='1' then prox<=LOCAL;
34                                elsif h(EAST)='1' then prox<=EAST;
35                                elsif h(WEST)='1' then prox<=WEST;
36                                else prox<=NORTH; end if;
37
38                        when SOUTH=>
39                                if h(LOCAL)='1' then prox<=LOCAL;
40                                elsif h(EAST)='1' then prox<=EAST;
41                                elsif h(WEST)='1' then prox<=WEST;
42                                elsif h(NORTH)='1' then prox<=NORTH;
43                                else prox<=SOUTH; end if;
44                end case;
45        end process;
```

Figura 51 – Exemplo de código fonte em linguagem VHDL para a seleção da porta de entrada que terá permissão de chaveamento.

7.1.1.5 Roteamento

O roteamento é o passo executado logo após a arbitragem, pois ele determina por qual porta de saída o pacote selecionado pela arbitragem será enviado. A maneira como a porta de saída é selecionada depende do algoritmo de roteamento utilizado.

A Figura 52 ilustra a parte da máquina de estados da lógica de controle correspondente o roteamento. Esta máquina de estados implementa o algoritmo de roteamento XY determinístico, o qual tem sido o algoritmo utilizado neste trabalho.

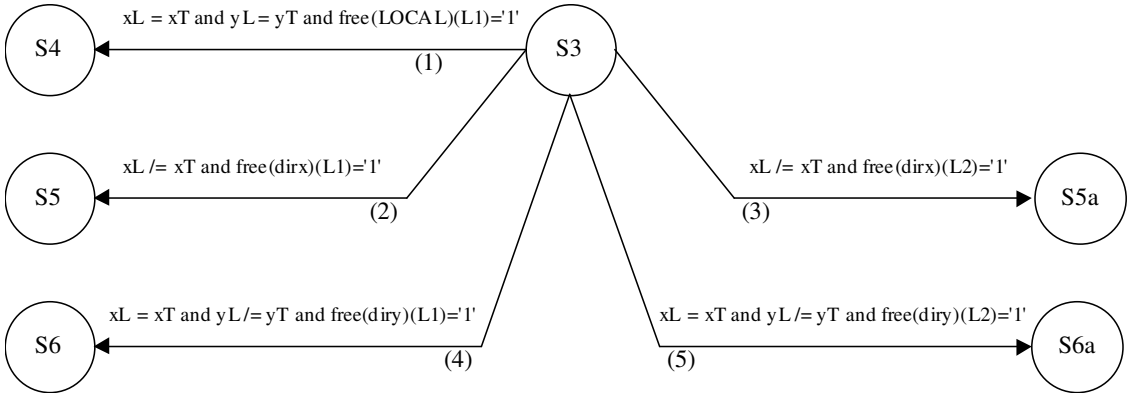


Figura 52 – Parte da máquina de estados da lógica de controle correspondente ao roteamento.

- No estado S3 é efetivamente realizado roteamento. O algoritmo XY faz a comparação entre o endereço do roteador atual e o endereço do roteador destino do pacote (armazenado no primeiro *flit* do pacote) e a máquina avança para um dos 5 possíveis estados.
- Nos demais estados é estabelecida a conexão da porta de entrada com a de saída através do preenchimento das tabelas *port_in*, *port_out*, *lane_in*. O estado atual das portas também é atualizado através do preenchimento da tabela *free*.

A Figura 53 mostra o código fonte em linguagem C de uma possível implentação do algoritmo XY determinístico, bem como o preenchimento das tabelas *port_in*, *port_out*, *lane_in* e *free*.

```
01  if (xL > xT) dirx = WEST; else dirx = EAST;
02
03  if (yL > yT) diry = NORTH; else diry = SOUTH;
04
05  if (xL == xT && yL == yT) && free[LOCAL][L1] == 1 {
06      port_in[dir_incoming] = LOCAL;
07      lane_in[dir_incoming][lane_incoming] = L1;
08      port_out[LOCAL][L1] = dir_incoming;
09      free[LOCAL][L1] = 0;
10      ack_h = TRUE; }
11
12  else if (xL != xT && free[dirx][L1] == 1) {
13      port_in[dir_incoming] = dirx;
14      lane_in[dir_incoming][lane_incoming] = L1;
15      port_out[dirx][L1] = dir_incoming;
16      free[dirx][L1] = 0;
17      ack_h = TRUE; }
18
19  else if (xL != xT && free[dirx][L2] == 1) {
20      port_in[dir_incoming] = dirx;
21      lane_in[dir_incoming][lane_incoming] = L2;
22      port_out[dirx][L2] = dir_incoming;
23      free[dirx][L2] = 0;
24      ack_h = TRUE; }
25
26  else if (xL == xT and yL != yT && free[diry][L1] == 1) {
27      port_in[dir_incoming] = diry;
28      port_out[diry][L1] = dir_incoming;
29      free[diry][L1] = 0;
30      ack_h = TRUE; }
31
32  else if (xL == xT and yL != yT && free[diry][L2] == 1) {
33      port_in[dir_incoming] = diry;
34      port_out[diry][L2] = dir_incoming;
35      free[diry][L2] = 0;
36      ack_h = TRUE; }
37
38  else
39      ack_h = 0;
```

Figura 53 – Exemplo de código fonte em linguagem C do algoritmo de roteamento XY determinístico.

O algoritmo de roteamento XY determinístico faz a comparação do endereço do roteador atual com o endereço do roteador destino do pacote (armazenado no primeiro *flit* do pacote). O pacote deve ser enviado para a porta local do roteador quando o endereço xLyL[2] do roteador atual for igual ao endereço xTyT[3] do roteador destino do pacote. Caso contrário, é realizada, primeiramente, a comparação horizontal de endereços. A comparação horizontal determina se o pacote deve ser enviado para o leste (xL<xT), para o oeste (xL>xT), ou se o mesmo já está horizontalmente alinhado ao roteador destino (xL=xT). Caso esta última condição seja verdadeira é realizada a comparação vertical que determina se o pacote deve ser enviado para o sul (yL<yT), para o norte (yL>yT), ou se o mesmo já está verticalmente alinhado ao roteador destino (yL=yT). Caso esta última condição seja verdadeira, ou a porta vertical escolhida esteja com os dois *lanes* ocupados, é realizado o bloqueio dos *flits* do pacote em todos os roteadores intermediários até que algum *lane* da porta de saída escolhida seja liberado.

7.1.1.6 *Deadlock*

Como mencionado anteriormente, essa implementação apresentou problemas de *deadlock*. A Figura 54 ilustra uma possível situação de *deadlock* que pode ocorrer nessa implementação, seguida de uma descrição.

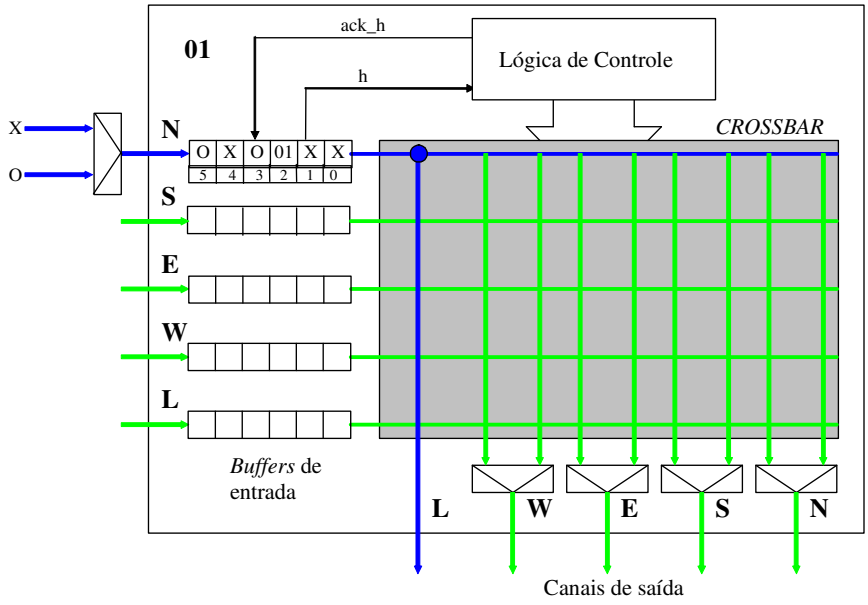


Figura 54 – Situação 1 de *deadlock*.

A porta de entrada N está recebendo dois pacotes com destino para a porta L. O pacote contendo apenas *flits* X chegou primeiro, já foi chaveado e está sendo transmitido para a porta de saída L. Em seguida começam a chegar *flits* O, os quais pertencem ao segundo pacote, juntamente com *flits* X do primeiro pacote. O *flit* 01 da posição 2 do *buffer* é o cabeçalho do segundo pacote. O cabeçalho do primeiro pacote, o qual também era 01, já foi transmitido.

Quando o próximo *flit* a ser transmitido for o cabeçalho do segundo pacote (01), o *buffer* faz

[2] xLyL é o endereço da chave atual, onde xL é o endereço horizontal e yL o endereço vertical.
[3] xTyT é o endereço da chave destino dos dados, onde xT é o endereço horizontal e yT o endereço vertical.

um novo pedido de chaveamento para a lógica de controle através do sinal *h* e fica esperando a confirmação que vem pelo sinal *ack_h*. Enquanto o *buffer* espera pela confirmação da lógica de controle, nenhum *flit* é transmitido.

A lógica de controle consulta a tabela *free* e verifica que a porta L já está em uso, pois ainda restam *flits* do primeiro pacote a serem enviados. Sendo assim ela não envia a confirmação para o *buffer* enquanto existam *flits* do primeiro pacote a serem enviados. Como os *flits* restantes do primeiro pacote estão atrás do cabeçalho do segundo pacote, eles não podem ser enviados (houve um bloqueio de cabeça de fila), pois o *buffer* está trancado esperando por um *ack_h*. Essa situação caracteriza uma dependência cíclica ocasionando o *deadlock*.

Como mostrou o situação acima, o *deadlock* ocorreu pelo fato de o *buffer* da porta N ter armazenado dois pacotes com o mesmo cabeçalho. Uma tentativa para solucionar o problema foi acrescentar uma restrição à lógica de roteamento. Essa restrição não permitia que dois pacotes com o mesmo cabeçalho fossem chaveados para a mesma porta de saída. Dessa maneira não há como um *buffer* armazenar *flits* de dois pacotes com o mesmo cabeçalho. Não entanto essa restrição não eliminou totalmente o *deadlock*. A Figura 55 ilustra uma nova situção de *deadlock* oriunda da restrição inserida, seguida de uma descrição.

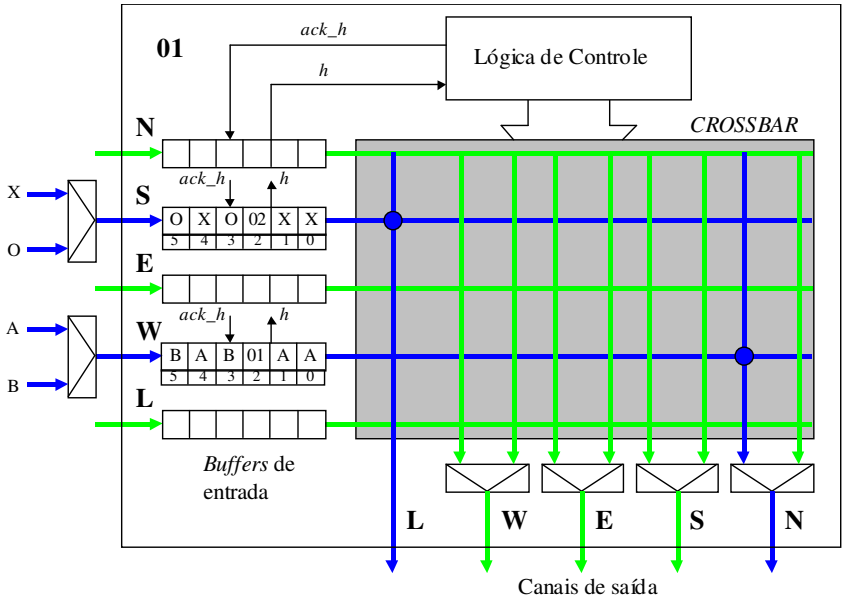


Figura 55 – Situação 2 de *deadlock*.

Como no caso anterior, agora a porta de entrada S também recebe dois pacotes. O pacote contendo apenas *flits* X chegou primeiro, já foi chaveado e está sendo transmitido para a porta de saída L. Em seguida começam a chegar *flits* O, os quais pertencem ao segundo pacote, juntamente com *flits* X do primeiro pacote. O *flit* 02 da posição 2 *buffer* é o cabeçalho do segundo pacote. O cabeçalho do primeiro pacote, o qual era 01, já foi transmitido.

A porta de entrada W também está recebendo dois pacotes. O primeiro pacote a chegar contendo apenas *flits* A, já foi chaveado e está sendo transmitido pelo *lane* 1 da porta de saída N. Em seguida começam a chegar *flits* B, os quais pertencem ao segundo pacote, juntamente com *flits* A do primeiro pacote. O *flit* 01 da posição 2 do *buffer* é o cabeçalho do segundo pacote. O

cabeçalho do primeiro pacote era 02 e já foi transmitido.

Quando o *buffer* da porta de entrada W for transmitir o cabeçalho do segundo pacote (01), ele fará um pedido de chaveamento à lógica de controle e vai ficar esperando pela confirmação. Essa confirmação virá somente quando a porta de entrada S enviar todos os *flits* do seu primeiro pacote, pois ela está ocupando a porta de saída L, a qual é solicitada pelo *buffer* de W.

Em seguida o *buffer* da porta S faz um pedido de chaveamento para o seu segundo pacote, cujo cabeçalho é 02. Devido à restrição adicionada à lógica de controle, ela não pode chavear o pacote para o *lane* 2 da porta de saída N porque o *lane* 1 já está em uso pelo primeiro pacote da porta de entrada W, o qual também tinha cabeçalho 02.

Nessa situação ambos *buffers* encontram-se bloqueados em uma dependência cíclica. O *buffer* de W espera que o *buffer* de S libere a porta de saída L enquanto ocupa o *lane* 1 da porta de saída N, e o *buffer* de S espera que o *buffer* W libere o *lane* 1 da porta de saída N enquanto ocupa a porta de saída L.

Note que ambas as situações apresentadas têm como raiz do *deaklock* o problema de bloqueio de cabeça de fila. A Seção a seguir apresenta uma segunda implementação livre de *deadlock*.

7.1.2 Canais virtuais com filas multiplexadas

Nesta implementação, o objetivo é eliminar o problema do *deadlock* causado pelo bloqueio de cabeça de fila. Uma nova arquitetura de *buffers* foi criada, onde agora o módulo *buffer* contém dois *buffers* de *flits* independentes, um para cada *lane*. Dessa maneira os *flits* recebidos por diferentes *lanes* são armazenados no *buffer* correspondente. Essa proposta visa modificar somente o módulo *buffer*, mantendo sua interface. Sendo assim a arbitragem e o roteamento permanecem inalterados em relação à versão de canais virtuais com *buffer* único.

7.1.2.1 Módulo *buffer*

Um novo módulo *buffer* foi criado, baseado na estratégia *SAMQ* (Seção 4.1), contendo dois *buffers* independentes para o armazenamento dos *flits* dos pacotes recebidos. Ao chegar em uma das portas de entrada do roteador, o *flit* é demultiplexado e armazenado no *buffer* correspondente ao *lane* de saída da porta que o está enviando. Por exemplo, se o *flit* é enviado pelo *lane* 2 de uma porta de saída, ele será armazenado no *buffer* relativo ao *lane* 2. Assim quando o roteador está recebendo dois pacotes simultaneamente pela mesma porta de entrada, os *flits* de cada pacote são armazenados em *buffers* diferentes. A Figura 56 ilustra a nova arquitetura do módulo *buffer*. Note que a porta de entrada L continua com a mesma estrutura de *buffer* único, pois nela não há canais virtuais.

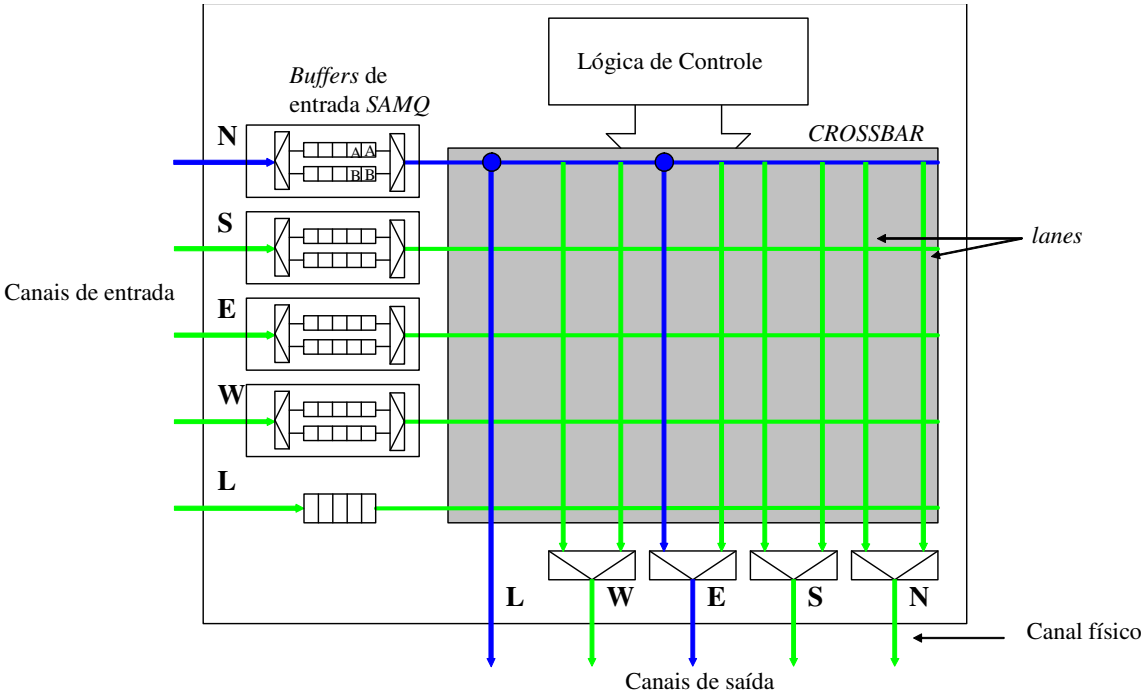


Figura 56 – Roteador com *buffers* multiplexados.

Os *buffers* continuam sendo organizados como *FIFOs* circulares. Também possuem os três ponteiros de controle *first*, *prox* e *last* e o contador de *flits counter_flit*, o qual determina o fim da conexão.

A interface de saída do módulo *buffer* é compartilhada pelos *buffers* internos (*buffer_l1 e buffer_l2*), portanto somente um dos *buffers* internos pode enviar *flits* ou requisitar chaveamento em um determinado instante de tempo. O armazenamento dos *flits* continua a ser controlado pelo controle de fluxo. No entanto, para o envio de *flits* tem-se uma máquina de estados que controla o compartilhamento da interface de saída do módulo *buffer*. A Figura 57 mostra a máquina de estados seguida de uma discussão do seu funcionamento. Nos estados HEADER_L1 e SEND_L1 a interface de saída do módulo *buffer* está associada ao *buffer_l1*, enquanto que nos estados HEADER_L2 e SEND_L2 está associada ao *buffer_l2*.

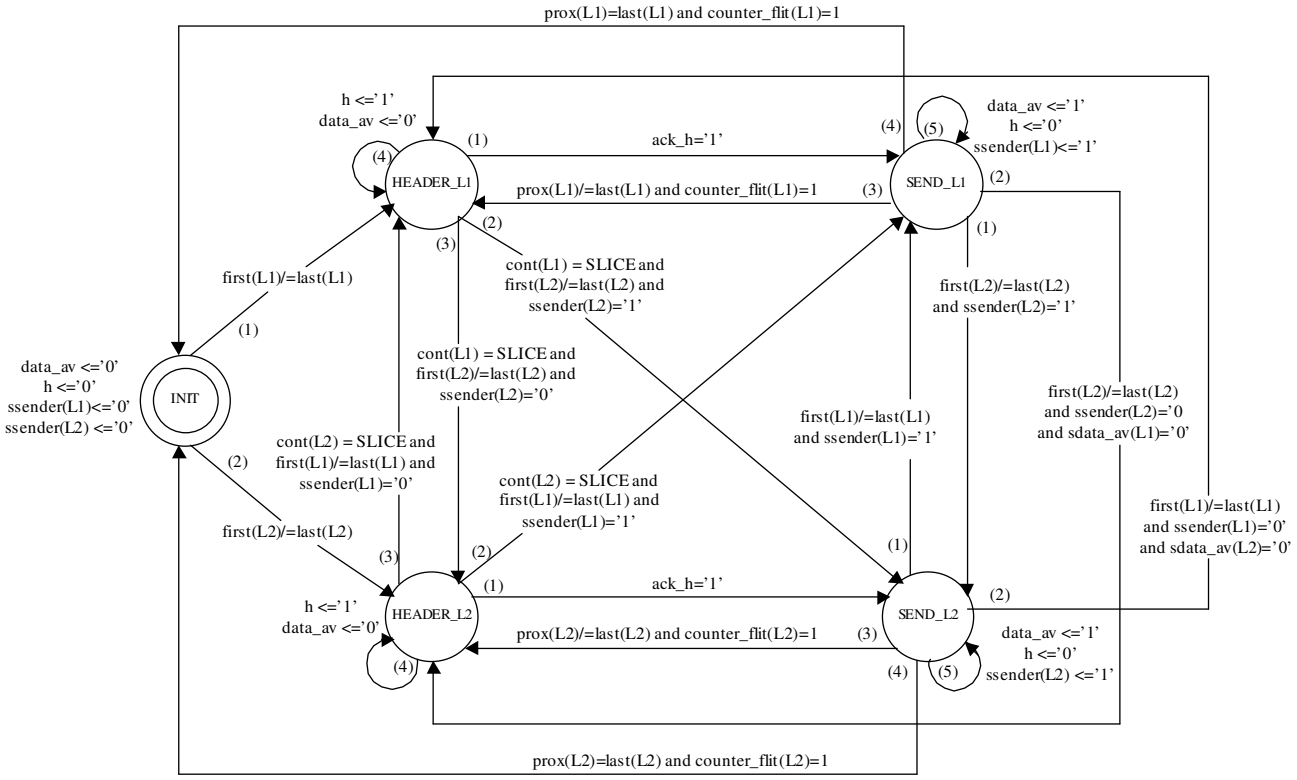


Figura 57 – Máquina de estados para o envio dos *flits*.

- Quando o sinal *reset* é ativado, a máquina de estados avança para o estado INIT. A máquina permanece neste estado até que exista algum *flit* em algum dos *buffers*, quando então avança para HEADER_L1 ou HEADER_L2.
- No estado HEADER_L1 é requisitado o chaveamento para o *buffer_l1* (*h='1'*). A máquina permanece nesse estado por *SLICE*[4] ciclos de *clock* esperando pela confirmação (*ack_h='1'*). Caso receba a confirmação antes do tempo excedido, a máquina avança para SEND_L1. Depois de excedido o tempo, ela avança para SEND_L2 se o *buffer_l2* tiver *flits* para serem enviados, para HEADER_L2 se o *buffer_l2* deseja requisitar chaveamento ou permanece nesse estado caso nenhuma das condições anteriores seja satisfeita.
- No estado HEADER_L2 é requisitado o chaveamento para o *buffer_l2* (*h='1'*). A máquina permanece nesse estado por *SLICE* ciclos de *clock* esperando pela confirmação (*ack_h='1'*). Caso receba a confirmação antes do tempo excedido, a máquina avança para SEND_L2. Depois de excedido o tempo, ela avança para SEND_L1 se o *buffer_l1* tiver *flits* para serem enviados, para HEADER_L1 se o *buffer_l1* deseja requisitar chaveamento ou permanece nesse estado caso nenhuma das condições anteriores seja satisfeita.
- Em SEND_L1 é indicado que existe um *flit* a ser transmitido (*data_av='1'*). A cada *flit* enviado é verificado o estado do *buffer_l2*. Se o *buffer_l2* tiver *flits* para serem enviados, a máquina avança para SEND_L2. Se o *buffer_l2* deseja requisitar chaveamento e o *buffer_l1* não tem mais dados para enviar (*sdata_av(L1)='0'*) a máquina avança para HEADER_L2. Se nenhuma das condições anteriores for satisfeita, a máquina permanece em SEND_L1 até

[4] Constante de tempo.

que próximo *flit* do *buffer_l1* seja um cabeçalho, quando então a máquina volta para HEADER_L1, ou e se não há mais *flits* em ambos *buffers*, quando então a máquina volta para INIT.

- Em SEND_L2 é indicado que existe um *flit* a ser transmitido (*data_av='1'*). A cada *flit* enviado é verificado o estado do *buffer_l1*. Se o *buffer_l1* tiver *flits* para serem enviados, a máquina avança para SEND_L1. Se o *buffer_l1* deseja requisitar chaveamento e o *buffer_l2* não tem mais dados para enviar (*sdata_av(L2)='0'*) a máquina avança para HEADER_L1. Se nenhuma das condições anteriores for satisfeita, a máquina permanece em SEND_L2 até que próximo *flit* do *buffer_l2* seja um cabeçalho, quando então a máquina volta para HEADER_L2, ou e se não há mais *flits* em ambos *buffers*, quando então a máquina volta para INIT.

7.2 Validação funcional

A Figura 58 ilustra um exemplo de transmissão onde o roteador 20 envia dados para o roteador 21. O roteador 20 envia os dados pela porta de saída *North* (índice 2) e o roteador 20 os recebe pela porta *South*.

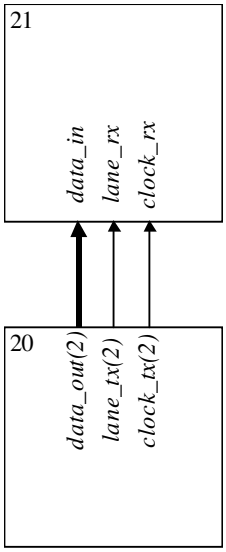


Figura 58 – Transmissão de dados.

A Figura 59 mostra as formas de onda do módulo *buffer* da porta entrada *South* recebendo os dados enviados pela porta de saída *North* do roteador 20. O roteador 20 está enviando dois pacotes simultaneamente utilizando os dois *lanes* de saída da porta *North*.

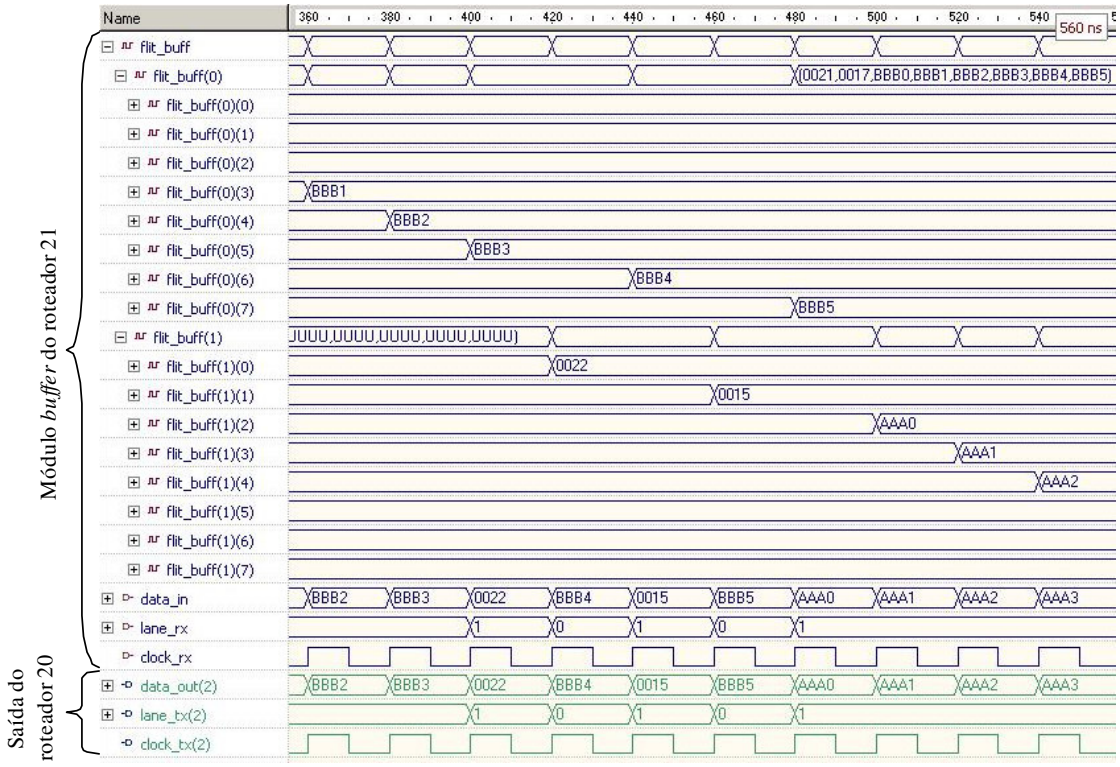


Figura 59 -Entrada de dados no módulo buffer do roteador 21 e saída de dados do roteador 20

A entrada *lane_rx* (conectada a *lane_tx(2)*) indica em qual dos *buffers* (*flit_buff*) o dado contido em *data_in* deve ser armazenado. O roteador 20 controla a entrada dos dados nos *buffers* através do sinal *clock_tx(2)* (conectado a *clock_rx*), pois trata-se de um controle de fluxo baseado em créditos (Seção 3.2). Os dados são armazenados na borda de subida do do sinal *clock_tx(2)*.

A Figura 60 mostra as formas de onda da lógica de controle do roteador 20 realizando a arbitragem e o roteamento seguida de uma descrição.

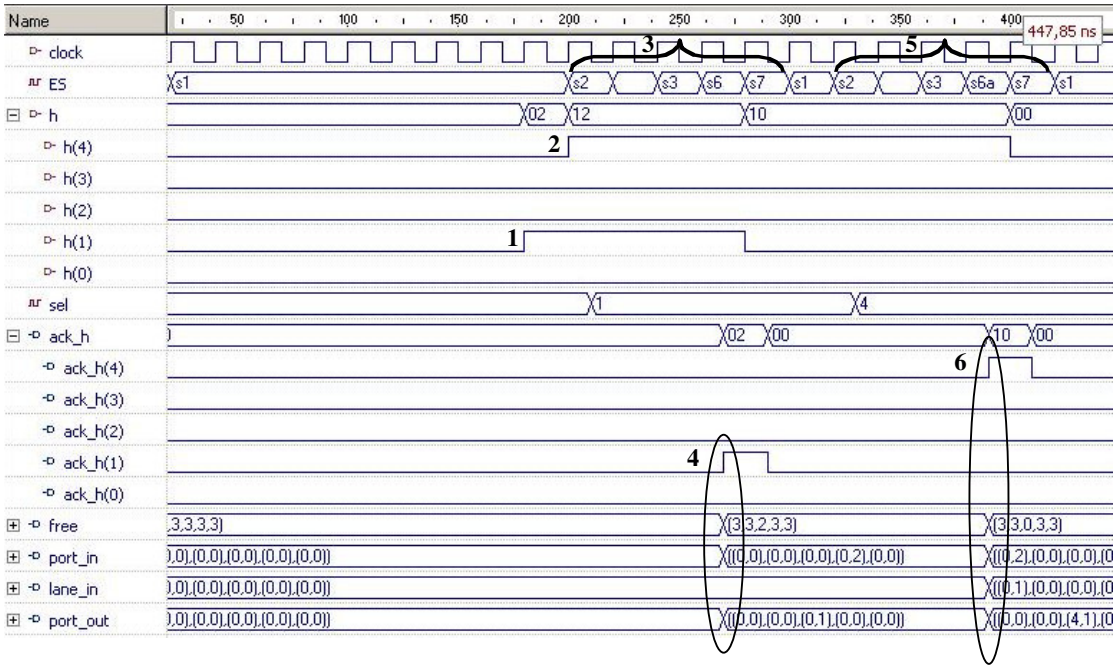


Figura 60 – Arbitragem.

1. A lógica de controle recebe um pedido de chaveamento da porta de entrada *West*.
2. A lógica de controle recebe outro pedido de chaveamento, agora da porta de entrada *Local*.
3. É realizada a arbitragem e o roteamento para a porta de entrada *West*.
4. As tabelas de chaveamento são preenchidas e a confirmação de chaveamento é sinalizada para a porta de entrada *West*.
5. É realizada a arbitragem e o roteamento para a porta de entrada *Local*.
6. As tabelas de chaveamento são preenchidas e a confirmação de chaveamento é sinalizada para a porta de entrada *Local*.

Como resultado, tem-se o chaveamento mostrado na Figura 61.

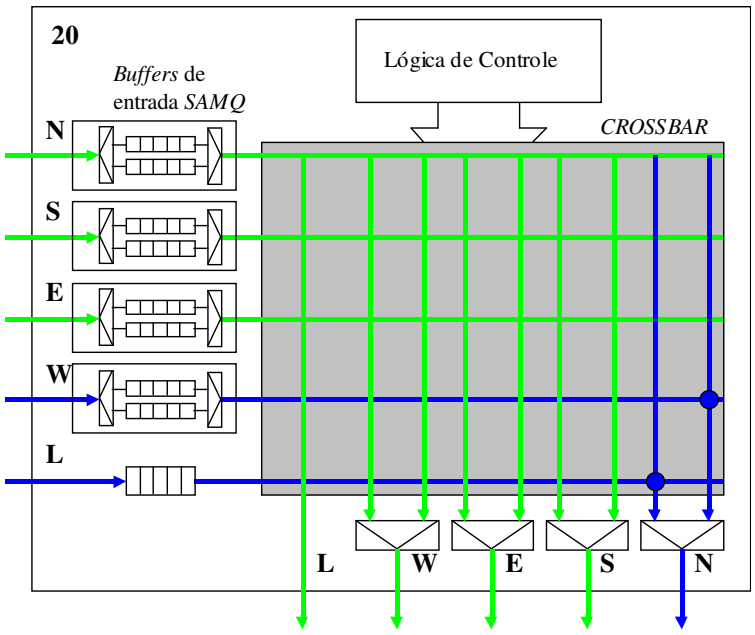


Figura 61 – Chaveamento do roteador 20.

7.3 Comparação 1 *lane* versus 2 *lanes*

Essa Seção apresenta resultados preliminares comparativos entre uma NoC com dois canais virtuais e uma sem canais virtuais. A comparação foi feita utilizando-se a rede HERMES, na qual foram implementados os canais virtuais.

Os testes foram realizados utilizando-se a ferramenta MAIA para a geração das NoCs, geração de tráfego e análise dos resultados obtidos. A topologia usada foi malha 5 x 5, com 5 roteadores enviando 100 pacotes cada, totalizando 500 pacotes. As NoCs geradas trabalham com *flits* de 16 bits e *buffers* de entrada de 8 posições. A simulação foi realizada com pacotes de tamanho igual a 100 *flits* e carga de tráfego igual a 33%. O resultado de área foi obtido através da ferramenta de sintese lógica *Lonardo Spectrum* tendo como alvo um dispositivo FPGA XC2V6000 da Virtex. A Tabela 5 apresenta os resultados obtidos e a Figura 62 ilustra um gráfico comparativo em relação à latência média para cargas de tráfego de 10%, 20%, 30%, 33%, 35%, 40% e 50%.

Tabela 5 - Resultados da comparação.

	Hermes Malha 5x5	
	1 *lane* por canal físico	2 *lanes* por canal físico
Latência média	1306.106 ciclos de *clock*	1054.258 ciclos de *clock*
Latência mínima	123 ciclos de *clock*	125 ciclos de *clock*
Latência máxima	5247 ciclos de *clock*	4212 ciclos de *clock*
Desvio padrão da latência	1690.36 ciclos de *clock*	1300.76 ciclos de *clock*
Throughput médio	31.76%	32.14%
Throughput mínimo	29.62%	30.51%
Throughput máximo	33.22%	33.27%
Desvio padrão do *Throughput*	1.75%	1.32%
Tempo total	34014 ciclos de *clock*	33054 ciclos de *clock*
Àrea	26,92% LUTs	78.08% LUTs

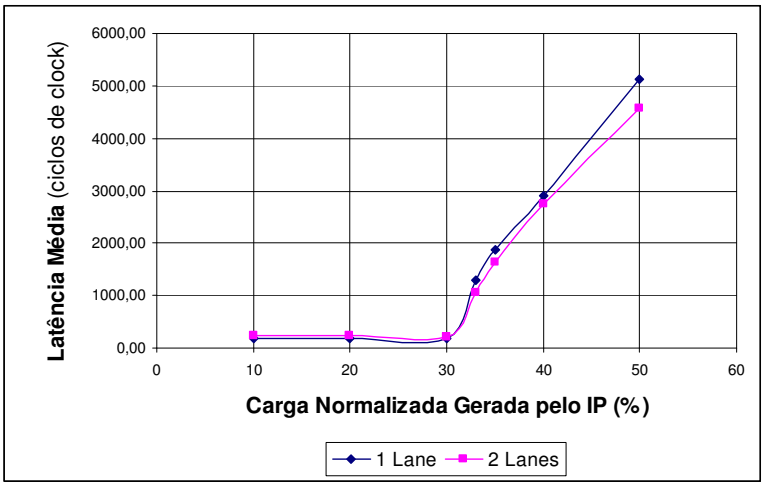


Figura 62 – Gráfico comparativo.

Como mostra a tabela, a versão com 2 *lanes* foi superior em todos os parâmetros de desempenho, mas ainda a um custo muito elevado em área. Em relação ao gráfico pode-se observar que enquanto a rede não está saturada (até 30% de carga) as duas versões têm uma latência média muito semelhante, com uma pequena vantagem para a versão com 1 *lane*. A partir dos 30% de carga a versão com 2 *lanes* começa a se mostrar superior mantendo a latência média inferior a versão com 1 *lane*.

8 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou diferentes possibilidades de implementação dos principais módulos que compõem a arquitetura de roteadores para redes intra-chip com topologia malha e modo de chaveamento *wormhole*, bem como avaliações preliminares de desempenho. Todo o trabalho de implementação e avaliação foi desenvolvido sobre a rede HERMES, utilizando-se a ferramenta MAIA para sua geração.

A rede HERMES, existente antes deste trabalho [MOR04b], possuía diversas escolhas arquiteturais definidas em função da complexidade dos módulos e da experiência do grupo de pesquisa na qual ela foi desenvolvida. Este trabalho de conclusão permitiu avaliar cada um dos módulos componentes da rede, e justificar ou não sua adoção. A Tabela 6 mostra os módulos que compunham a rede Hermes original e as contribuições do presente trabalho.

Tabela 6 – Módulos da Hermes original e contribuições.

Algoritmos	HERMES Original	***Contribuição do Presente Trabalho***
Controle de fluxo	*Handshake*	Avaliação e implementação do controle de fluxo baseado em créditos. Mostrou-se que o mesmo possui desempenho superior, permitindo também comunicação assíncrona (GALS).
Memorização	*Buffers de entrada*	Discutiu-se as diversas formas de memorização utilizadas em NoCs, justificando-se a escolha de *bufferização* de entrada.
Roteamento	XY determinístico	Mostrou-se que o roteamento XY determinístico apresenta melhor desempenho frente aos algoritmos parcialmente adaptativos, corroborando a adoção do mesmo para a rede Hermes. Entretanto, deve-se ainda avaliar como os algoritmos parcialmente adaptativos operam quando canais virtuais são utilizados.
Arbitragem	*Round-Robin* centralizado	*Round-Robin* centralizado apresenta desempenho inferior às arbitragens distribuídas. Três algoritmos distribuídos foram implementados a avaliados. A política de arbitragem *Oldest-first* e sua integração com a política *Round-Robin* apresentou o melhor desempenho elétrico.
Canais Virtuais	Ausente	Duas implementações avaliadas. Mostrou-se que o uso de *buffers* simples conduz ao problema de bloqueio *HOL*, impedindo seu uso em aplicações com tráfego aleatório. A segunda implementação, com filas S*AMQ*, apresentou (resultados preliminares) ganho de desempenho a um custo ainda muito elevado em área. Canais virtuais com filas S*AMQ* apresentam potencial para uso em algoritmos com qualidade de serviço (*QoS*), dado o baixo desvio padrão na vazão.

O presente trabalho consolidou os conceitos básicos relativos a redes intra-chip, e através de inúmeras simulações avaliou o desempenho dos diversos algoritmos e arquiteturas.

A continuidade deste Trabalho de Conclusão envolve as seguintes tarefas:

1. Otimizar o código de canais virtuais de forma a minimizar o consumo de área.
2. Realizar avaliações mais profundas no que se refere ao desempenho de canais virtuais.
3. Avaliar os algoritmos de roteamento parcialmente adaptativos quando canais virtuais são utilizados.
4. Realizar novas avaliações em relação à arbitragem distribuída diminuindo os tamanhos dos *buffers* de entrada.
5. Prototipar aplicações utilizando canais virtuais.

Trabalhos de pesquisa referentes à continuidade deste Trabalho de Conclusão, a serem desenvolvido a médio e a longo prazo compreendem:

1. Definição da arquitetura de roteadores que permitem alocação dinâmica da banda passante por pacotes, em função de restrições de latência e vazão. Esta arquitetura de roteador permitirá atender a requisitos de qualidade de serviço.
2. Desenvolvimento de sistemas multi-processados conectados por NoC. O objetivo deste trabalho é pesquisar arquiteturas *sea-of-processors*. Os paradigmas empregados em multicomputadores deverão ser adaptados para arquiteturas SoC, onde área é uma restrição maior.
3. Pesquisa e avaliação de arquiteturas GALS visando o baixo consumo de potência. Considerando que a distribuição do sinal de relógio ao longo de todo um SoC é praticamente impossível de ser realizada, dado o comprimento das conexões e a potência consumida, a avaliação de arquiteturas baseadas em *ilhas síncronas* permitirá o desenvolvimento de sistemas complexos aliado ao baixo consumo de potência.
4. Geração de redes heterogêneas. Em função de uma dada aplicação otimizar a rede, utilizando uma biblioteca de componentes (desenvolvidos no presente trabalho), gerar a rede que apresente o melhor compromisso de área *versus* desempenho.

9 REFERÊNCIAS BIBLIOGRÁFICAS

[AND03] Andriahantenaina, A.; et al. "*SPIN: a Scalable, Packet Switched, On-Chip Micro-network"*. In: Design Automation and Test in Europe Conference (DATE), 2003, pp. 70-73.

[BEN01] Benini, L.; De Micheli, G. *"Powering Networks on Chips"*. In: 14th International Symposium on System Synthesis, 2001, pp. 33-38.

[BEN02] Benini, L.; De Micheli, G. *"Networks on chips: a new SoC paradigm"*. Computer, v.35(**1**), 2002, pp. 70-78.

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

[BOD95] Boden, N. J.; et al. "*Myrinet: a Gigabit-per-second Local Area Network"*. IEEE Micro, v.15(**1**), 1995, pp. 29-36.

[BOL04] Bolotin, E.; Cidon, I; Ginosar, R.; Kolodny, A. "*QNoC: QoS architecture and design process for Network on Chip"*. The Journal of Systems Architecture, v.20(**2-3**), 2004, pp. 105-128.

[CUL98] Culler, D.; Singh, J. P. *"Parallel Computer Architecture: a Hardware Software Approach"*. Los Altos, California: Morgan Kaufmann, 1998, 1100 p.

[DAL01] Dally, W. J.; Towles, B. "*Route packets, not wires: on-chip interconnection networks"*. In: Design Automation Conference (DAC), 2001, pp. 684-689.

[DAL86] Dally, W. J.; Seitz, C. L. "*The Tosus Routing Chip* ". Distributed Computing, v.1, 1986, pp. 187-196.

[DAL90] Dally, W. J. *"Virtual-Channel Flow Control"*. In: 17th International Symposium on Computer Architecture (ISCA), 1990, pp. 60-68.

[DUA02] Duato, J.; Ni, L. *"Interconnection Networks"*. Morgan Kaufmann, 2002, 624 p.

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

[GUE00] Guerrier, P.; Greiner, A. "*A generic architecture for on-chip packet-switched interconnections"*. In: Design Automation and Test in European Conference (DATE), 2000, pp. 250-256.

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

[HEN96] Hennessy, J. L.; Patterson, D. "*Computer Architecture: A Quantitative Approach*". San Francisco, California : Morgan Kaufmann, 1996, 760 p.

[HU02] Hu, J.; Deng, Y.; Marculescu, R. "*System-Level Point-to-Point Communication Synthesis Using Floorplanning Information"*. In: Int. Conference on VLSI Design, 2002, pp. 297-301.

[HWA93] Hwang, K. "Advanced Computer Architecture: Parallelism, Scalability, Programmability". New York : McGraw-Hill, 1993. 770p.

[INM89a] INMOS. "*IMS C004 Programmable Link Switch"*. In: INMOS Engineering Data, 1989, pp. 479-501.

[KAR86] Karol, M. J.; Hluchyj, M. G.; Morgan, S. P. "*Input vs. Output queuing on a space-division packet switch"*. In: Proc. IEEE Global Telecommun. Conf., 1986, pp. 659-655.

[KUM02] Kumar, S.; Jantsch, A.; Soininen, J.; Forsell, M.; Millberg, M.; Öberg, J.; Tiensyrjä, A. *"A Network on Chip Architecture and Design Methodology"*. In: Int. Symposium on Very Large Integration Scale, 2002, pp. 105-112.

[LAN00] Langen, D.; Brinkmann, A.; Ruckert, U. "*High Level Estimation of the Area and Power Consumption of on-Chip Interconnects"*. In: ASIC/SOC Conference, 2000, pp. 297-301.

[LIA00] Liang, J.; et al. "*aSOC: A Scalable, Single-Chip Communication Architecture"*. In: International Conference on Parallel Architectures and Compilation Techniques, 2000, pp. 37-46.

[MEL04] Mello, A.; Ost, L.; Calazans, N.; Moraes, F. *"Evaluation of Routing Algorithms on Mesh Based NoCs"*. Technical Report TR042, PPGCC-PUCRS, 2004, 17p.

[MOH98] Mohapatra, P. "Wormhole routing techniques for directly connected multicomputer systems". ACM Computing Surveys, v.30(**3**), 1998, pp. 374-410.

[MOR04a] Moraes, F.; Ost, L.; Mello, A.; Palma, J.; Calazans, N. "*NOCGEN - Uma Ferramenta para Geração de Redes Intra-Chip Baseada na Infra-Estrutura HERMES"*. In: X WORKSHOP IBERCHIP, 2004, pp. 210-216.

[MOR04b] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "*HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip"*. Integration: the VLSI Journal, v. 38(**1**), 2004, pp. 69-93.

[NI93] Ni, L.; McKinley, P. "*A Survey of Wormhole Routing Techniques in Direct Networks"*. IEEE Computer Magazine, v.26(**2**), 1993, pp. 62-76.

[PAT98] Patterson, D.; Hennessy, J. "*Organização e Projeto de Computadores – A Interface Hardware/Software*". Morgan Kaufmann (LTC Editora), 2000, 551 p.

[REE87] Reed, D.; Fujimoto, R. "*Multicomputer Networks: Message-Based Parallel Processing*". Cambrige, MA: MIT Press, 1987, 400p.

[SAN03] Santos, F.; Zeferino, C.; Susin, A.. "*Modelos Parametrizáveis de Árbitros Centralizados para a Síntese de Redes-em-Chip*". Hífen. Urugaiana, 2003, pp. 91-96.

[SCH97] Schaller, R. "*Moore´s Law: Past, Present and Future*". IEEE Spectrum, v.34(**6**), 1997, pp. 52-59.

[TAM92] Tamir, Y.; Frazier, G. "*Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches"*. IEEE Transactions on Computers, v.41(**6**), 1992, pp.725-737.

[ZEF03] Zeferino, C. *"Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho"*. Tese de Doutorado. PPGC, UFRGS, 2003, 123p.

[ZIP04] Zipf, P.; Hinkelmann, H.; Ashraf, A.; Glesner, M.*"A Switch Architecture and Signal Synchronized for GALS System-on-Chips"*. In: 17th symposium on Integrated circuits and system design table of contents, 2004, pp. 210-215.