RAF: ROTEADOR ASSÍNCRONO FLEXÍVEL PARA REDES INTRA-CHIP

Israel Mendonça dos Santos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Victor Mauro Goulart Ferreira

Rio de Janeiro
Setembro de 2014

RAF: ROTEADOR ASSÍNCRONO FLEXÍVEL PARA REDES INTRA-CHIP

Israel Mendonça dos Santos

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Victor Mauro Goulart Ferreira, Ph.D.

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2014

Santos, Israel Mendonça dos

RAF: Roteador Assíncrono Flexível para Redes Intra-chip/Israel Mendonça dos Santos. – Rio de Janeiro: UFRJ/COPPE, 2014.

XVI, 94 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Victor Mauro Goulart Ferreira

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 78 – 82.

1. Roteador Reconfigurável. 2. Roteadores Assíncronos. 3. Redes de Interconexão Intra-Chip. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico este trabalho aos meus pais e à minha irmã, sem cujo apoio total e irrestrito nada teria sido possível.

Agradecimentos

Agradeço, primeiramente, a Deus pela minha vida, dos meus familiares e dos amigos, e por sua infinita misericórdia, permitir que chegasse até aqui.

Aos meus pais, por sempre me apoiarem e me tornarem um homem de caráter. A minha irmã, por ser uma eterna amiga e companheira.

Aos meus orientadores, Felipe França e Victor Goulart, pelo acompanhamento deste trabalho e por estarem sempre disponíveis para tirar dúvidas, mesmo pela internet. Obrigado pela amizade, paciência e companheirismo.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) por todo o suporte fornecido que me permitiu desenvolver este trabalho. Em especial gostaria de agradecer aos professores Valmir e Amorim e aos funcionários da secretaria e suporte (Solange, Gutierrez, Itamar, Adilson, Mercedes, Solange e Sônia).

À Universidade do Estado do Rio de Janeiro (UERJ) pela base sólida de minha formação. Em especial a: Rosa Costa, Alexandre Sztajnberg, Leandro Marzulo, Maria Alice, Galúcio, Paulo Eustáquio, professores que fizeram grande diferença em minha formação.

Aos autores referenciados que, com seus trabalhos, possibilitaram a realização desta pesquisa.

À Capes, pela bolsa fornecida, que permitiu custear minhas despesas durante o Mestrado.

À JASSO pela oportunidade de fazer um intercâmbio e ao professor Kazuaki Murakami e a Universidade Kyushu pelo acolhimento. Em especial, gostaria de agradecer aos meus amigos do laboratório: Jose, Krishna, Said, Fukuyama, Kawabata, Tsuhata, Ogata, Tanaka, Kuboi, Imamura, Yoshida, Egawa e as secretárias Shudo e Takerabe. O intercâmbio foi uma experiência muito enriquecedora e contribuiu tanto para a minha formção acadêmica quanto pessoal. Foram muitas as amizades feitas neste período. Estas pessoas foram muito importantes, pois me fizeram sentir em casa, mesmo estando longe do meu país. São elas: Geisa, Zhivko, David, Hoda, Wenjing, Hanka, Alex, Jimmy, Mariko, Akira, Duo Duo, Hojung, Diogo, Gabriel, Kasumi, Zheng Ying, Li Yao, Wang Xuan, Vlad, Elnaz, Andressa, Rodolfo e principalmente minha namorada Duong.

Além de experiências e amizades, o intercâmbio também me deu alguém para

amar: Duong, a quem agradeço por todo o carinho e compreensão nos momentos em que estive ausente por conta da realização deste trabalho. Obrigado por ser paciente comigo, principalmente nos momentos de ansiedade e estresse, e por sempre apostar no nosso futuro. Te amo!

Aos vários amigos que fiz na UFRJ e UERJ e que me deram apoio através de sua amizade: Pedro, Arthur (Mentira), Brunno Goldstein, Jux, Bruno Brazil, Lucas, Juliana Sobral, Evandro, Moyses, Denilson, Aylton, Fabio Carvalho, Leandro, Ingrid, Nath Folha D'ouro, DiogoS (23 e não 23), Ítalo, Alan, Alex, Daniela, Rebeca, Danilo, Rafael, Alexandre Nery, Saulo, Luneque, Douglas e Hugo. Obrigado a todos pela amizade e agradáveis momentos que me proporcionaram.

Aos amigos de infância Dennis, Gabriel, Bruna, Maria, Ernane, Rodrigo Zander, Carlos André e Maria. Que mesmo não tendo tanto contato, posso contar sempre que preciso.

Ao meu professor de inglês Romulo que sempre me apoiou e acreditou em mim, além de sempre me entreter com suas histórias engraçadas de ex-alunos.

Aos meus amigos e professores de Japonês: Rika, Raphael e Fabio.

Agradeço a Nguyen Sinh Cung e Vladimir Ilyich Ulyanov por ter lutado por seus ideais e ter ajudado a criar um estado livre.

Por fim, agradeço a Steven Paul Jobs por ter me inspirado e continuar me inspirando a ser um "rebelde"e sempre pensar diferente.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

RAF: ROTEADOR ASSÍNCRONO FLEXÍVEL PARA REDES INTRA-CHIP

Israel Mendonça dos Santos

Setembro/2014

Orientadores: Felipe Maia Galvão França
Victor Mauro Goulart Ferreira

Programa: Engenharia de Sistemas e Computação

Roteadores reconfiguráveis para redes de interconexão intra-chip permitem a alocação de *buffers* de forma diferenciada, aumentando assim o desempenho geral da rede. Roteadores Flexíveis são roteadores reconfiguráveis que expandem o espaço de busca de *buffers* das portas. Quando um pacote faz um requerimento de *buffer* à uma porta de entrada, caso esta não possua espaço disponível, ela irá procurar por espaço nas outras portas do roteador, reduzindo assim a probabilidade de bloqueio do pacote. Neste trabalho é feito uma avaliação detalhada, e comparação com o roteador base, de um roteador reconfigurável denominado Roteador Flexível, o qual foi introduzido em outros trabalhos e terá seus mecanismos e avaliações estendidos neste trabalho. Sua implementação é feita em simulador de redes de alto desempenho, o que permite a adição de mecanismos novos de reconfiguração, além de ser possível variar inúmeros parâmetros de configuração. Algo que nunca haviam sido testado antes. Além disto, uma versão assíncrona deste roteador denominada RAF, Roteador Assíncrono Flexível, também é produzida. Esta versão é produzida a partir da versão síncrona do mesmo utilizando a metodologia ASERT - *Asynchronous Scheduling by Edge Reversal Timing*, um algoritmo de temporização descentralizado, utilizado para gerar a sinalização entre as unidades funcionais assíncronas baseado na divisão hierárquica dos blocos funcionais do circuito do roteador síncrono. Ambas as versões do roteador flexível, síncrona e assíncrona, são implementadas em simuladores de *hardware* e tem seus desempenhos testados. Utilizando os mecanismos aplicados no simulador, obtivemos até 21% de melhora de desempenho em relação a *throughput* para a versão síncrona, além de provar que é plausível a construção da versão assíncrona do mesmo.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

AFR: ASYNCHRONOUS FLEXIBLE ROUTER FOR NETWORK ON CHIPS

Israel Mendonça dos Santos

September/2014

Advisors: Felipe Maia Galvão França
Victor Mauro Goulart Ferreira

Department: Systems Engineering and Computer Science

Reconfigurable routers for network-on-chips (NoCs) allows the allocation of buffers in a differently way, which increases the overall performance of the network. Flexible Routers are reconfigurable routers that expand the search space of the input port buffers. Whenever a packet makes a buffer request to an input port, if there is no free buffers at this port, the search will be performed in other ports of the router, reducing the probability of blocking of packets. The Flexible Router was proposed in previous works, but due to technology limitations, it was not possible to perform a full evaluation of its benefits. This works makes a detailed evaluation and comparison with the base router using parameters and mechanisms never tested before. Its implementation is made in a NoC simulator, which allows us to implement new reconfiguration mechanisms for the first time. In addition, it was also built the asynchronous version of this router called RAF - Asynchronous Flexible Router. This version is produced after the application of the synchronous-asynchronous technique ASERT - Asynchronous Scheduling by Edge Reversal Timing, a fully decentralized timing method used to generate signals between the asynchronous functional blocks based on the existing hierarchical division of the original synchronous circuit. Both versions of the flexible router, synchronous and asynchronous, were implemented and tested in simulators. Our router achieved gains in performance related to throughput up to 21% and it was proven that an asynchronous version of the router is totally possible to be made.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os avanços tecnológicos recentes permitem a integração de bilhões de transistores em um único *chip*. Com isso, sistemas completos, possuindo todos os componentes necessários para um projeto de hardware como processador, memória e periféricos, podem ser encapsulados em uma peça única, conhecidos como sistemas em um único chip ou *SoC (System-on-a-Chip)* [1–3]. Com este aumento do número de componentes por *chip*, abordagens convencionais de comunicação, como a criação de um *link* dedicado conectando cada componente, começam a se tornar inviáveis. Devido a isto, o desempenho dos sistemas digitais modernos começa a sofrer influencia cada vez maior da aquitetura ou infraestrutura de comunicação comparada à velocidade dos circuitos lógicos ou memórias. Torna-se então necessário a implementação de uma arquitetura especial de comunicação que quebre as barreiras impostas pelas soluções clássicas.

Network-on-Chips (NoCs) [4–6], ou Rede de Interconexão Intra-chip, é uma nova infra-estrutura de comunicação proposta para sanar os problemas clássicos impostos por esse novo paradigma. NoCs vem ganhando fama devido à sua escalabilidade e eficiência. Em NoCs, os componentes do sistema trocam informações utilizando a rede como um sub-sistema para o tráfego de informações.

NoCs possuem diferentes custos e requerimentos de performance e, dependendo da aplicação alvo, construir uma rede genérica para cobrir todo o espectro de aplicações necessárias significaria criar roteadores caros e ineficientes em termos de área e consumo de energia. Em contrapartida, desenvolver uma NoC específica para cada aplicação do mercado significa que decisões importantes deveriam ser tiradas no momento de design e, NoCs específicas, embora possuam um desempenho otimizado para uma aplicação ou um determinado grupo de aplicações, perdem generalidade,o que pode acabar comprometendo requisitos como escalabilidade e impedir as aplicações de otimizações visando diferentes comportamentos de diferentes demandas de sistemas.

As NoCs são construídas a partir de múltiplos canais de dados ponto-a-ponto

interconectados por diversos roteadores.

1.1 Motivação

De acordo com Dally [7], o papel de um roteador é basicamente transferir pacotes pelos canais da rede. Roteadores de alto-desempenho possuem um papel fundamental para permitir um fluxo de informações que satisfaçam as necessidades da aplicação da NoC. Dentre os componentes dos roteadores, *buffers* são utilizados para armazenar pacotes temporariamente ou que por algum motivo específico ficaram incapacitados de avançar para o próximo roteador. De fato, tem-se observado que armazenar um pacote consome muito mais energia do que transmitir o mesmo. [8]

Buffers são os componentes que mais impactam o desempenho de roteadores. Um roteador ideal deveria ser capaz de se adaptar aos diferentes requisitos de aplicações dinamicamente sem ter sua performance comprometida. Os *buffers* são utilizados para fazer a adaptação da rede quando a quantidade de pacotes que entram em uma porta do roteador é alterada. Utiliza-los para fazer esta adaptabilidade vem com um custo, visto que os *buffers* são os componentes que mais consomem energia. Como pode ser visto em [9], aproximadamente 64% do consumo de energia dos roteadores vêm dos *buffers* e mesmo quando a carga da rede é alta, existem até 85% de *buffers* ociosos. Isto demonstra a necessidade de buscar novas maneiras de otimizar a utilização destes recursos valiosos através de um mecanismo diferenciado de alocação de recursos do roteador.

Além dos *buffers*, existe também um desperdicio de energia inerente de projetos de circuitos síncronos. Dentre os causadores deste desperdício de energia, destacam-se o chaveamento excessivo dos módulos, a imposição da obrigação do tempo de execução de cada módulo ser igual a do módulo mais lento e o *clock skew*. Circuitos assíncronos possuem várias características desejáveis que resolvem, em parte, todos os problemas mencionados anteriormente. Normalmente, projetos de circuitos assíncronos possuem protocolos de sinalização local que reduzem a quantidade de chaveamentos necessários para manter os circuitos operando. Estes protocolos também eliminam a necessidade de todos os módulos possuírem o mesmo tempo de execução, mantendo um tempo de execução heterogeneo para cada bloco funcional. Por último, eles também eliminam a necessidade de um sistema central de sincronismo, desobrigando o projetista a distribuir um ou mais sinais de *clock* a todos os elementos sequenciais e se preocupar com a defasagem entre eles.

Tendo introduzido os problemas relacionados a desempenho e desperdício de energia anteriormente, é então introduzido o conceito de Roteador Assíncrono Flexível (RAF): Uma nova arquitetura de roteador que utiliza de forma mais eficiente os *buffers* disponíveis do roteador.

1.2 Estado da Arte

Em SoCs é normal encontrarmos diferentes demandas de componentes que se conectam através da rede intra-chip. Devido a isto, nota-se que diferentes aplicações possuem diferentes demandas em termos de largura de banda nos roteadores. Na literatura, apresentam-se várias soluções que propõe uma melhoria no sentido de aumentar o desempenho da NoC.

A quantidade e o tamanho do *buffer* são aspectos que estão relacionados diretamente com o desempenho da rede [10]. O tamanho do *buffer* em particular, foi analisado em [10, 11]. Foi descoberto que a área de um roteador de rede de interconexão intra-chip é ocupada em sua grande maioria pelo *buffer*.

Em [12] é proposto um roteador com *clocks* diferentes para os *body* e *head flits*. Os *body flits* operam mais rápido do que os *head flits*. O desempenho do roteador é melhorado na medida que os *body flits* avançam mais rápido pelo caminho já reservado pelos *head flits*.

Em [12–15] também são propostas outras arquiteturas de roteador, porém, em todos estes trabalhos, a configuração do roteador é feita no momento do *design* do mesmo, fazendo com que a solução seja estática. Soluções estáticas são mais simples, baratas e gastam menos energia do que as reconfiguráveis, porém podem sofrer uma perda de desempenho considerável caso os requisitos da aplicação se alterem, além de nem sempre ser possível fazer a escalabilidade neste tipo de roteador. Além disto, customização e atualização dos roteadores estáticos para diferentes demandas não são possíveis sem um custoso trabalho de reengenharia, o que faz com que soluções que melhor se adaptem a estas mudanças mais desejáveis.

1.2.1 Comparação com gerenciamento de *buffer*

Existem muitas propostas com o objetivo de gerenciar os *buffers* para melhorar o desempenho de roteadores.

A arquitetura proposta em [16] modifica a forma que a arbitração é feita a fim de melhorar o pareamento entre as portas de entrada e as portas de saída. Ela faz com que esta arbitração seja feita em duas etapas mais rápidas do que a normalmente utilizada e através deste mecanismo de pareamento melhorado entre portas de entrada e saída, o desempenho é melhorado.

DAMQ [17] e seus respectivos melhoramentos [18–20] utilizam um número fixo e dedicado de *buffers* em cada porta de saída, armazenando tráfego direcionado para uma direção. A vantagem de se utilizar a alocação estática de *buffers*, é que fluxos de todas as direções tem as mesmas condições para competir com o *crossbar*. Esta filosofia também foi aplicada em *Row-Column (RoCo) Decoupled Router [21]*, que separa os canais virtuais em dois grupos: um para o tráfego na direção x e outro

para o tráfego na direção y. O problema de contenção na etapa de arbitração de *switch* é melhorado, como explicado nos trabalhos mencionados anteriormente.

Em [22] e [23] é proposto uma arquitetura de *buffer* distribuído que emula *buffers* na porta de saída. *Buffers* na porta de saída possuem uma maior capacidade de alcançar um maior *throughput* e possuem um atraso menor quando comparado com *buffers* na porta de entrada. Para que a emulação pudesse ser feita, foram utilizados dois *crossbars* com *buffers* entre ambos que simulam o *buffer* de saída, além de uma memória de um *slot* na porta de entrada. Isto gera um aumento excessivo de área no roteador, visto que *buffers* consomem muita área.

ViChaR [24] utiliza uma estrutura de *buffer* unificada (UBS, do inglês *Unified Buffer Structure [25]* em conjunto com um regulador de canais virtuais, que dinamicamente aloca canais virtuais e *buffers* de acordo com as condições do tráfego. O *UBS* é utilizado para criar a ilusão de que a quantidade, assim como a profundidade dos canais virtuais varia de acordo com as demandas do tráfego. Quando a quantidade de pacotes na rede aumenta, ViChaR dispõe mais canais virtuais para estes pacotes a fim de reduzir contenção. Diferentemente do trabalho proposto nesta dissertação, ViChaR utiliza não utiliza uma estrutura de *buffer* distribuída.

Em [26] é proposto uma arquitetura de um roteador reconfigurável similar à proposta neste trabalho. Nela, uma porta empresta/toma emprestado espaços nos *buffers* das portas vizinhas, possibilitando assim, que a profundidade dos *buffers* varie ao longo tempo de acordo com as demandas do tráfego. Para construir o mecanismo que possibilita o empréstimo de *buffers*, foram utilizados multiplexadores, que interligam os *buffers*. Após um tempo, este pacote é encaminhado à porta de entrada que pediu o *buffer* emprestado através destes multiplexadores. A diferença entre este trabalho e o proposto nesta dissertação é a não-existência da necessidade do pacote ser encaminhado para outra porta, uma vez que tenha sido concedido espaço para o pacote ser armazenado em uma porta vizinha.

A ideia do Roteador Flexível foi proposta em [27] e aprimorada em [28]. Foi provado o potencial da ideia através dos resultados, porém, devido a complexidade de se implementar alguns mecanismos em *hardware*, como Canais Virtuais e controle de fluxo *wormhole*, todo o potencial desta arquitetura não pôde ser testado.

1.3 Objetivos e Contribuições

Este trabalho propõe modificações no mecanismo de alocação de *buffers* dos roteadores de NoCs tradicionais síncronos, buscando aumentar o desempenho do roteador, além de propor uma versão assíncrona dos roteadores base e proposto com o objetivo de reduzir o custo energético dos mesmos. O mecanismo vem encapsulado em uma nova arquitetura de roteador denominado Roteador Flexível, que permite a uma

porta de entrada fazer a alocação de *buffers* pertencentes a outras portas. Embora este mecanismo tenha sido proposto em outros textos [27–29], este trabalho se diferencia ao adicionar elementos na arquitetura nunca testados, além de aperfeiçoar alguns já existentes. A versão assíncrona de baixo custo energético do roteador flexível é denominado Roteador Assíncrono Flexível (RAF).

O Roteador Assíncrono Flexível faz alterações no caminho de dados do roteador base. Com o objetivo de avaliar-se os impactos positivos e negativos da arquitetura proposta, variou-se as configurações do mesmo no momento de simulação.

Além do impacto no desempenho da NoC que incorpora tal roteador com o uso de aplicações artificiais, é objetivo desta pesquisa verificar o impacto de área e consumo de energia que a sua versão assíncrona em hardware (*Verilog*), além de apontar as principais diferenças para as versões síncronas e assíncronas. Foram criadas versões em linguagem de descrição de hardware para ambas.Para que a versão assíncrona fosse produzida, foi aplicado o método ASERT [30] de sinalização assíncrona na versão síncrona do roteador.

A implementação do roteador será testada através da implementação de suas funcionalidades no simulador de redes de interconexão (Topaz [31]), responsável por testar suas funcionalidades e obter os resultados de desempenho. O consumo de área e energia de sua versão síncrona e assíncrona, foram testados através de implementações na linguagem de descrição de hardware *Verilog* e ferramentas de síntese FPGA e VLSI foram utilizadas para extrair tais informações em LUTs e tecnologia de 0.90nm.

Em resumo, as contribuições deste trabalho são a arquitetura do roteador flexível mais completa, testado com parâmetros nunca antes testados como *wormhole* e canais virtuais e a conversão síncrona-assíncrona do roteador base e flexível através da metodologia de conversão ASERT.

1.4 Estrutura do Texto

Este trabalho se divide em 7 partes. A primeira parte compreende o presente capítulo introdutório com motivação da pesquisa, trabalhos relacionados, objetivos e contribuições e estrutura do texto. O Capítulo 2 nos dá uma ideia geral sobre redes de interconexão intra-chip, introduz o conceito de NoCs e relembra conceitos básicos, úteis para o entendimento da arquitetura do roteador flexível. O Capítulo 3 fala sobre o mecanismo de conversão síncrono-assíncrono ASERT e dá exemplos de como implementá-lo. O roteador flexível é apresentado no Capítulo 4, a forma de que o mesmo foi implementado em *hardware* e como foi feita a sua versão assíncrona é apresentado no Capítulo 5. A sexta parte é composta pelo Capítulo 6, onde são descritos os experimentos realizados e é feita uma análise dos resultados obtidos.

Finalmente, o Capítulo 7 apresenta as conclusões e possíveis trabalhos futuros.

Capítulo 2

Conceitos Básicos

O desempenho dos sistemas digitais avançados hoje em dia é limitado pela comunicação ou interconexão, não pela lógica ou memória [7]. Isso se deve ao fato que o tamanho dos transistores está em constante decremento e em razão disto, tornando-se cada vez mais comum a associação de múltiplos componentes à um único *chip*. De acordo com G. Moore [32] o número de componentes que pode ser agregado em um único *chip* dobra a cada 18 meses. Com isto, a densidade dos *chips* está cada vez maior. Este avanço nos capacita a ter, em alguns casos, sistemas completos em um único chip.

System-on-chip (SoC) é o conceito em que todos os componentes de um computador estão agregados em um único sistema integrado. Neste tipo de sistema, é comum a existência de muitos componentes heterogêneos que incluem um ou mais processadores, controladores e blocos de memória entre outros. A indústria de SoC está em fase de transição de sistemas *single-core*, ou seja, com apenas um núcleo físico para sistemas *multi-core*, com dezenas para centenas de núcleos em um único *chip* [33]. A integração destes componentes requer um mecanismo de comunicação eficiente em que seja garantido os requisitos de desempenho.

2.1 Emergência das Redes de Interconexões Intra-chip

Diferentes arquiteturas de comunicação foram propostas a fim de interligar diferentes componentes dos SoCs. Antes de serem listadas as mais comuns, torna-se necessário a introdução das medidas de desempenho utilizadas nas redes. Estas medidas são utilizadas de forma a comparar diferentes redes de interconexão intra-chip entre si e que também serão utilizadas nesta dissertação.

- *Throughput*: É a quantidade de pacotes que chegam ao nó destino para um determinado tráfego. Para medi-lo, conta-se a quantidade de pacotes que são

entregues ao seu destino durante um intervalo de tempo para o conjunto total de tráfego na rede.

- Latência: É o tempo que um pacote leva de uma origem ao seu destino. É medido como a diferença entre o instante de tempo que o pacote foi injetado na rede, e o instante de tempo em que ele foi consumido pelo componente conectado ao roteador destino.

- Tolerância a falhas: É a capacidade da rede de continuar operando mesmo após a ocorrência de falhas. Chamamos de *single-point* uma rede que é capaz de continuar operando após 1 falha, seja ela em um canal (um canal cortado por exemplo) ou em um nó(um roteador queimado por exemplo) [7]. Devido a quantidade de falhas que podem ocorrem neste tipo de sistema serem muito pequenas, algo que na maioria dos casos não comprometem o desempenho geral do sistema, as redes *single-point* são consideradas boas.

Tendo introduzido estas medidas, faz-se uma lista das arquiteturas de redes mais comuns:

2.1.1 Arquitetura de Comunicação Ponto-a-Ponto

Esta arquitetura utiliza um ou vários fios dedicados para fazer a conexão entre dois componentes distintos diretamente. Embora seu desempenho seja ótimo em termos de *throughput*, latência e consumo de energia, visto que cada componente terá acesso exclusivo ao recurso, sua implementação torna-se dispendiosa na medida em que o número de componentes do *chip* aumenta. O aumento de componentes tem como consequência o aumento desproporcional do número de fios na rede. Para cada novo componente adicionado a rede, é necessário o acréscimo de um fio dedicado para cada componente que irá se comunicar com este novo componente, o que acaba ocasionando graves consequências no *design* da rede. A primeira é um *overhead* de área devido a quantidade de fios necessários para atender esta arquitetura. A segunda é a dificuldade de rotear fisicamente este material no *chip*, além do fato dela não ser tolerante a falhas. Caso um ou mais fios que conectam dois componentes se danifiquem, estes ficarão incapazes de se comunicar. Uma consequência ainda mais grave é o fato desta arquitetura não ser modular, ou seja, caso haja alguma alteração na aplicação, deverá ser aplicado todo um trabalho de *redesign* da rede. A Figura 2.1 mostra um exemplo de arquitetura de comunicação ponto-a-ponto.

2.1.2 Arquitetura de Comunicação de Barramento

Esta arquitetura introduziu o conceito de links compartilhados. Os componentes estão conectados através de um canal unidirecional compartilhado (barramento ou

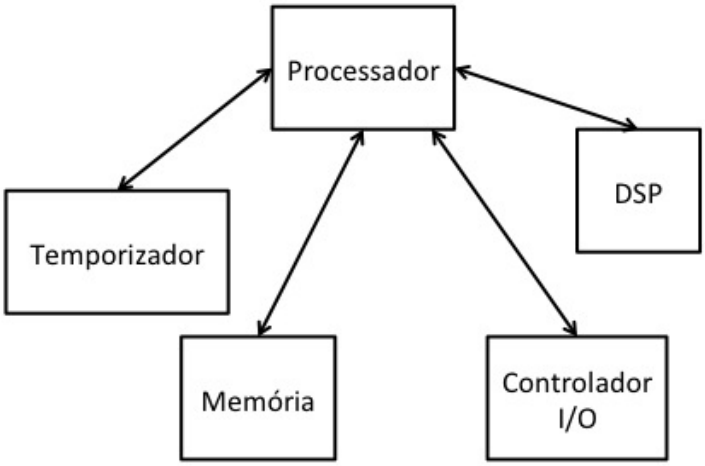


Figura 2.1: Arquitetura de comunicação Ponto-a-Ponto.

bus[34]) e trocam mensagens através deste canal. Todas as mensagens são enviadas a todos os nós no formato *broadcast*. Caso um nó receba uma mensagem ao qual ele não seja o destinatário, ele simplesmente a ignora. Para os nós utilizarem o meio de transmissão (cabo) simultaneamente, torna-se necessário a utilização de uma tecnologia de controle de acesso ao meio, como *Carrier Sense Multiple Access* (CSMA)[35] ou *Bus Master*. Do ponto de vista arquitetural, esta arquitetura é modular. Componentes podem ser adicionados e removidos do barramento sem nenhuma restrição. No entanto, a sincronização destes componentes se torna penosa na medida que seu número aumenta, o que pode comprometer o desempenho da rede. Um outro problema ainda mais grave, é a falta de capacidade de tolerância a falhas desta rede. No caso, se uma parte do canal for rompido, todos os nós ficam isolados e toda a comunicação da rede fica comprometida. Este tipo de rede se torna bem atraente para sistemas que utilizam-se de poucos componentes pois sua implementação é barata (necessita-se apenas de um meio de transmissão) e simples. A Figura 2.2 mostra um exemplo de arquitetura de comunicação de barramento.

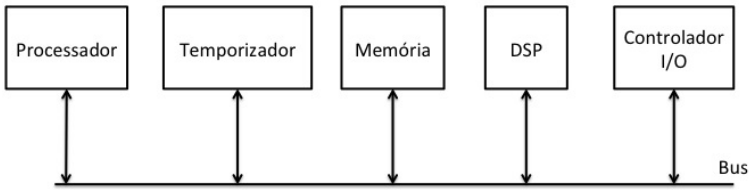


Figura 2.2: Arquitetura de comunicação de Barramento.

2.1.3 Redes de Interconexão Intra-chip

Na medida que o número de componentes aumenta consideravelmente, torna-se inviável a utilização de fios dedicados ou de canais únicos. O próximo passo é na direção de uma rede de troca de mensagens. Em uma rede de interconexão intra-chip, também conhecidas como NoCs (do inglês *Network-on-Chips*), os módulos trocam dados usando a rede como um sub-sistema para tráfego de informações através de mensagens divididas em pacotes. Ela é construída a partir de múltiplos *links* ponto-a-ponto interconectados por roteadores. No *design* destas redes, alto *throughput* e baixa latência são ambos importantes parâmetros de *design* e o roteador é um elemento fundamental para que estes requisitos sejam cumpridos. Roteadores de alto desempenho são requeridos para permitir uma evasão de pacotes que atenda os requerimentos do sistema. Uma mensagem deve passar por diferentes roteadores até chegar ao seu nó destino. A Figura 2.3 mostra um exemplo de arquitetura de redes de interconexão intra-chip.

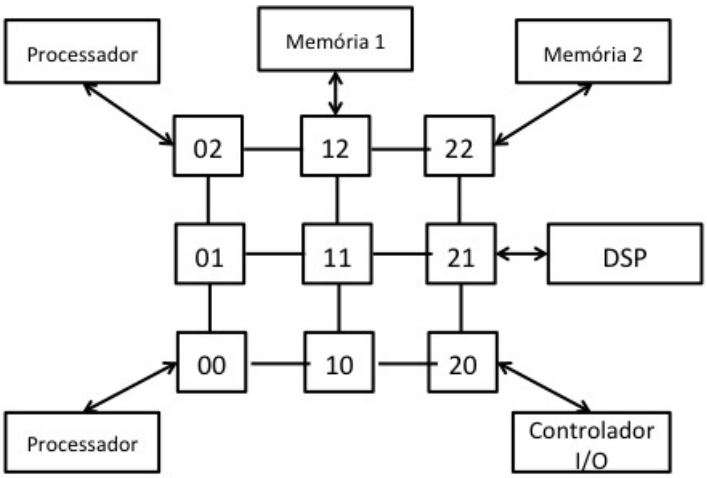


Figura 2.3: Rede de Interconexão Intra-chip.

2.2 Considerações sobre Redes Intra-chip

2.2.1 Componentes da NoC

Uma rede de interconexão intra-chip é composta basicamente por quatro elementos básicos: canais, interfaces de rede, pacotes e roteadores.

- Canais: É o meio de transporte físico que transmite dados de um ponto a outro. Em NoCs, os canais são responsáveis por fazer a ligação física entre dois roteadores, ou um roteador e uma interface de rede. Sua importância se dá na quantidade de fios que possuem para transmitir dados. Quanto maior a

quantidade de pinos, maior será a banda efetiva do canal, e consequentemente, maior será a quantidade de dados transmitidos por ciclo. A quantidade de fios, assim como a de canais em uma NoC é restrita ao *design* da rede.

- Interface de rede: A interface de rede é a conexão entre a rede de interconexão e o cliente da rede (*IP core - processing element* ou memória). Como alguns PEs já possuem uma interface pré-definida, a interface de rede pode servir como *wrapper* para se ajustar ao roteador da NoC. Uma interface bem construída é dita não-obstrutiva, ou seja, permite que o cliente faça total uso da banda da rede. Embora a maioria das interfaces de rede sejam construídas para utilizar os protocolos de rede mais conhecidos, algumas podem ser modificadas para melhor atender o seu cliente. Por exemplo, uma interface que conecta a memória a rede, pode ser adaptada a utilizar apenas protocolos de memória compartilhada, enquanto uma interface que conecta um dispositivo externo, deve utilizar um protocolo diferente.

- Pacotes: Embora não sejam componentes físicos da rede, são componentes criados pela interface de rede. São utilizados para encapsular uma mensagem. Além da mensagem, contém também informações que irão ser utilizadas pelos roteadores para que a mensagem chegue ao seu destino correto e de forma ordenada. Todo pacote é identificado por um identificador único (*packet* ID). Seu tamanho depende da implementação da NoC e é medido em *flits*. Um *flit* é a menor unidade de transmissão de dados entre roteadores na NoC; normalmente é configurado como o mesmo número de bits usados no canal para que seja possível enviar um *flit* por ciclo. *Flits* podem ser definidos basicamente em três tipos: *Head, Body e Tail flits. Head flits* normalmente carregam toda informação do *header* do pacote como informações de roteamento, assim como de alocação de canal virtual. *Body e tail flits* normalmente carregam o *packet load* ou dados em si, sendo que o que diferencia o *tail* de um *body flit* é a informação no *tail flit* dizendo que esta é a ultima parte do pacote, essa informação é utilizada para desalocar *buffers* após o *tail flit* ter sido recebido. A utilização da informação do *tail flit* será explicada no Capítulo 4. A estrutura de um pacote, assim como a de um *flit* pode ser conferida na Figura 2.4.

- Roteadores: A função básica de um roteador é encaminhar pacotes pela rede até seu nó destino. Em NoCs, um roteador comum contem cinco portas de entrada e cinco portas de saída equivalentes: Norte (N), Sul(S), Leste(E), Oeste(W) e Local(L). Todo roteador deve, no melhor caso, ser capaz de ler um pacote que chega em uma de suas cinco portas e, de acordo com o controle de fluxo, encaminhá-lo para uma de suas cinco portas de saída. Em alguns casos, o roteador possui *buffers* em suas portas de entrada, de saída, ou ambas

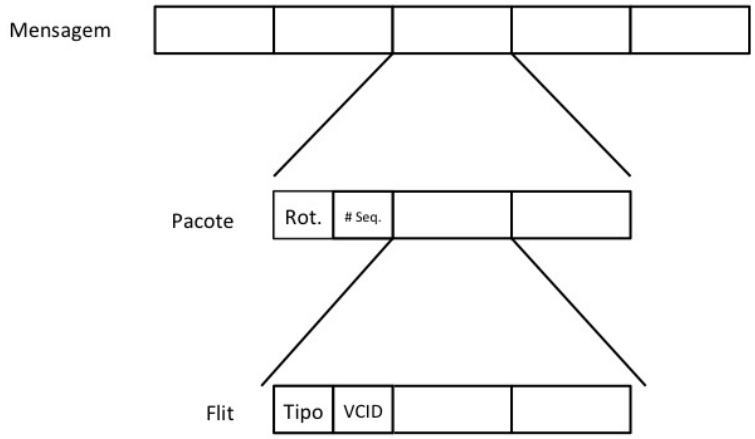


Figura 2.4: Subdivisões de uma mensagem: A primeira divisão é em pacotes. A segunda divisão é em uma unidade menor denominada *flit*.

para armazenar pacotes que não puderam ser encaminhados no ciclo atual. O desempenho dos roteadores tem grande impacto no desempenho da rede. Normalmente o roteador faz a injeção e ejeção de pacotes através de seu nó local, que é conectado a um *Processing Element*. A arquitetura básica do roteador pode ser vista na Figura 2.5, cada componente da figura será explicado na próxima seção.

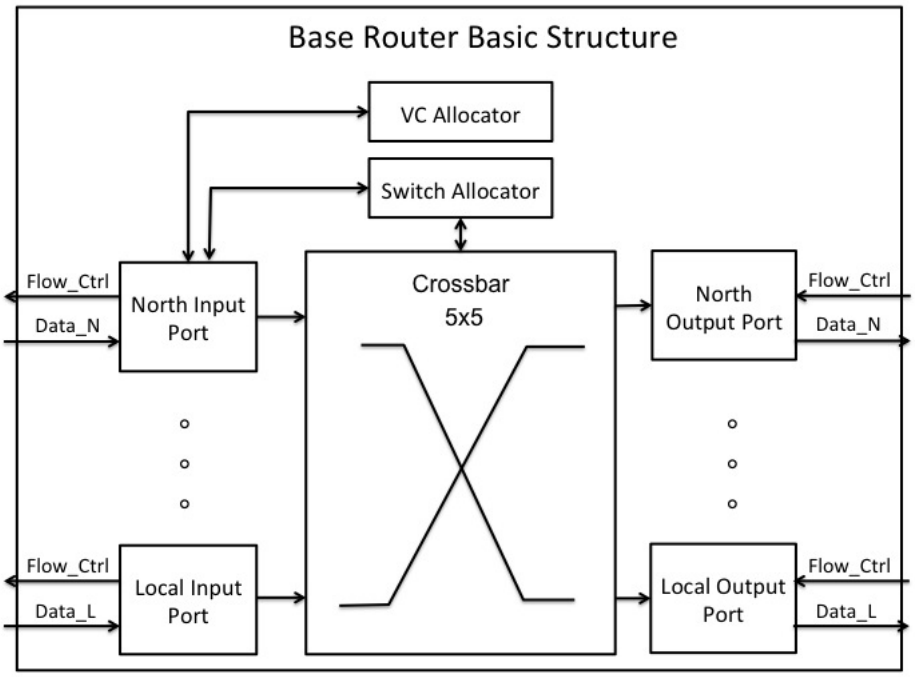


Figura 2.5: Estrutura do Roteador Base.

2.2.2 Topologia

Topologia é o nome que se dá ao arranjamento dos nós da rede [7]. Estas podem ser separadas em duas categorias: redes diretas e redes indiretas.

Nas redes diretas, ou redes estáticas, cada roteador possui uma instancia ou bloco IP (do inglês *Intellectual Property*) associado. Este par pode ser visto como um elemento único no sistema, comumente denominado nó. A Figura 2.7 mostra um nó de rede direta.

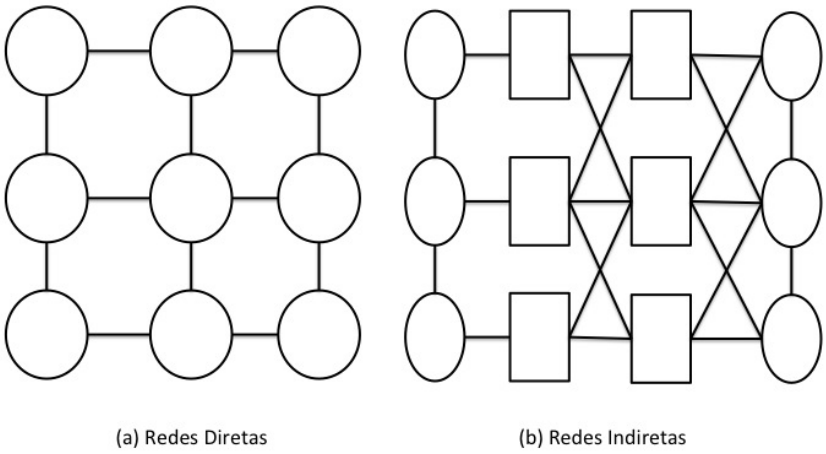


Figura 2.6: (a)Uma rede direta aonde todos os nós são homogeneos conténdo IP e roteador. (b)Rede indireta aonde cada nó é um roteador ou um switch.

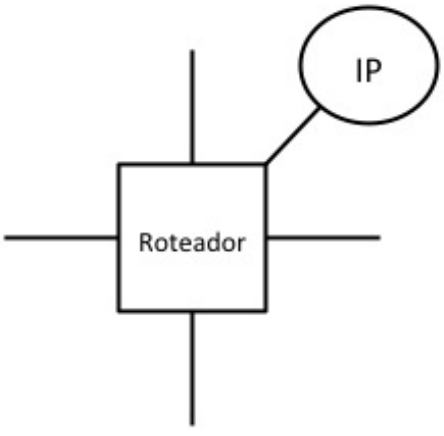


Figura 2.7: Nó contendo IP e roteador em uma rede direta.

Nas redes indiretas, os nós possuem apenas uma função: IP ou *switch*. Os IPs não se conectam diretamente, mas são conectados através de *switches* que fazem o redirecionamento de pacotes entre a origem e o destino. Neste tipo de rede, mensagens são geradas e consumidas apenas por determinados nós (os que são IP) fazendo assim com que os nós não sejam mais homogêneos. A capacidade de um

nó poder ser apenas um *switch* ou IP ou ambos é a característica principal que diferencia as redes diretas das redes indiretas.

2.2.3 Controle de Fluxo

Para que uma mensagem seja transmitida de uma origem a um destino, esta é divida em pacotes (e em alguns casos, sub-dividida novamente em unidades de fluxo - *flits*). O controle de fluxo determina como os recursos da rede, tais como canais e *buffers* serão alocados para os pacotes. Um bom controle de fluxo aloca estes recursos de forma eficiente, fazendo assim com que a banda da rede seja mais eficientemente aproveitada.

O que distingue um mecanismo de controle de fluxo de outro, é para que tipo de recurso que ele aloca espaço de memória (Nenhum espaço na memória, pacotes ou *flits*), além de como eles gerenciam estes recursos. Os mais simples são os que não utilizam *buffers*. Neles, caso um pacote seja bloqueado, ele é descartado ou desviado de sua rota. Com a adição de *buffers* como recurso, foram criados vários mecanismos de controle de fluxo.

Os métodos de controle de fluxo mais conhecidos são *Store-and-Forward*, *Cut-Through* e *Wormhole*. Mais detalhes sobre o funcionamento de cada um são dados a seguir:

- *Store-and-Forward*: Neste método, os *buffers* são alocados em unidades de pacote. Um pacote fica armazenado no *buffer* até que todos os seus *flits* cheguem. A desvantagem deste método é que os *buffers* precisam ser grandes o suficiente para armazenar um pacote inteiro. Se a rede possui tamanhos de pacote variados, o *buffer* precisa ser no mínimo do tamanho do maior pacote da rede, o que pode ocasionar o desperdício de espaço e energia por causa de *buffers*. Um outro problema com *store-and-forward* é o fato de que pode ocorrer um desperdício de banda quando o pacote fica ocioso esperando até que todos os *flits* estejam completamente no *buffer*. Para resolver este tipo de problema foi introduzido o conceito de *Cut–Through*.

- *Cut-Through* [36]: Neste método, os *buffer* também são alocados em unidades de pacote, porém os *flits* são encaminhados logo assim que haja espaço no roteador vizinho, eliminando a necessidade de esperar que o pacote inteiro esteja no *buffer*. Isto causa uma melhor utilização da banda da rede, visto que a mesma fica menos tempo ociosa, e consequentemente melhora o desempenho da mesma. Uma comparação destes dois métodos pode ser vista na Figura 2.8. Um problema que persiste neste método é que a alocação de recursos é feita para pacotes. Assim como no *Store-and-forward*, o *buffer* precisa ser grande

o suficiente para que um pacote inteiro caiba nele. Este problema é resolvido em *wormhole*.

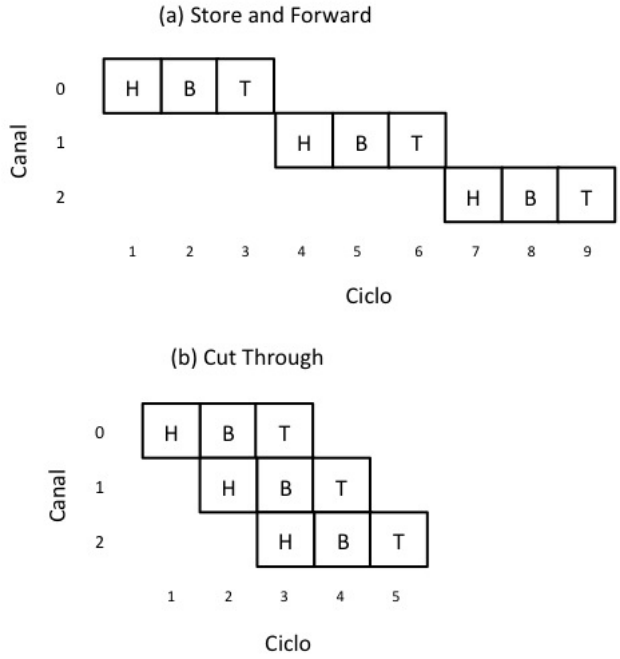


Figura 2.8: Exemplo que mostra como um pacote de três *flits*: Head(H), Body(B) e Tail(T), é transmitido através três 3 nós. Em *Store-and-Forward* (a), o pacote demora 9 ciclos para atravessar os 3 nós. Utilizando o mecanismo de controle de fluxo *Cut-Through* (b), este tempo é reduzido para apenas 5 ciclos.

- *Wormhole* [37]: Neste método, assim como no *Cut-through*, os *flits* não precisam esperar que sua ultima parte chegue no roteador. Eles são encaminhados assim que possível em direção ao seu destino. A grande diferença entre estes dois métodos se dá pela forma que este mecanismo aloca seus *buffers*. Ao invés de utilizar a unidade de medida em pacotes, ele utiliza a medida em *flits*. Isto permite que o tamanho dos *buffers* seja menor do que um pacote, fazendo assim com que os *buffers* consigam acomodar variáveis tamanhos de pacote a um custo reduzido, visto que *buffers* podem ser configurados a um tamanho mínimo de apenas um *flit*. Embora não aumente a banda efetiva, *wormhole* faz uma melhor utilização dos *buffers*, o que ocasiona em economia de energia e área na arquitetura do roteador.

2.2.4 Canais Virtuais

As redes de interconexão possuem dois recursos básicos: Nós e canais. Como já foi discutido anteriormente, um nó é composto basicamente por um roteador, e para cada porta do roteador temos um canal associado. Quando um roteador possui um

buffer em sua porta de entrada, este é utilizado para armazenar pacotes até que os mesmos sejam encaminhados para outros nós. Enquanto um pacote A tem um *buffer* alocado, nenhum outro pacote poderá utilizar o canal associado a ele até que o pacote A o desaloque. Se o pacote A ficar bloqueado, nenhum pacote subsequente poderá utilizar este canal. Este tipo de situação, aonde um pacote não pode avançar devido à um bloqueio imopsto por outro pacote é denominado congestionamento. A Figura 2.9 ilustra o congestionamento causado nesta situação.

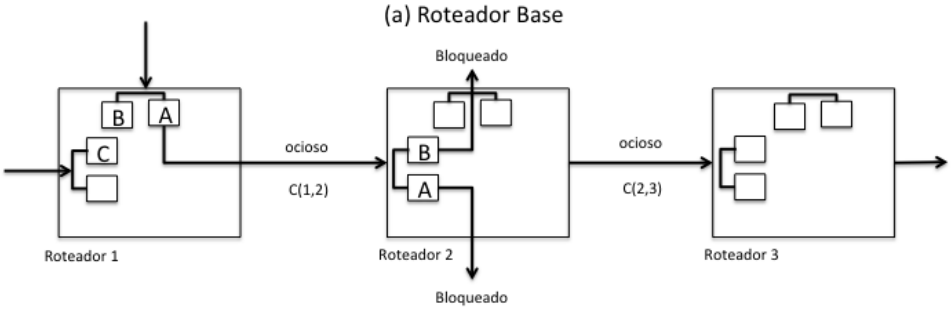


Figura 2.9: Exemplo que mostra como o pacote B fica impossibilitado de alocar um canal ocioso devido a falta de espaço para armazená-lo no roteador vizinho. Tornando assim o canal C(1,2) ocioso.

Para resolver o problema de bloqueio de canais por pacotes ociosos, é introduzido o conceito de canais virtuais [38, 39]. Canais virtuais removem a alocação de um canal por um único *buffer*, fazendo assim com que um único canal seja compartilhado entre diferentes *buffers*. Se um pacote A aloca um canal físico associado a um canal virtual, um outro pacote B pode alocar o mesmo canal físico através de outro canal virtual. A Figura 2.10 mostra a mesma rede da Figura 2.9 com a adição de canais virtuais. Enquanto o pacote A fica bloqueado tentando sair pela porta Sul do Roteador 2, o pacote B é encaminhado para a porta Oeste através de outro canal virtual.

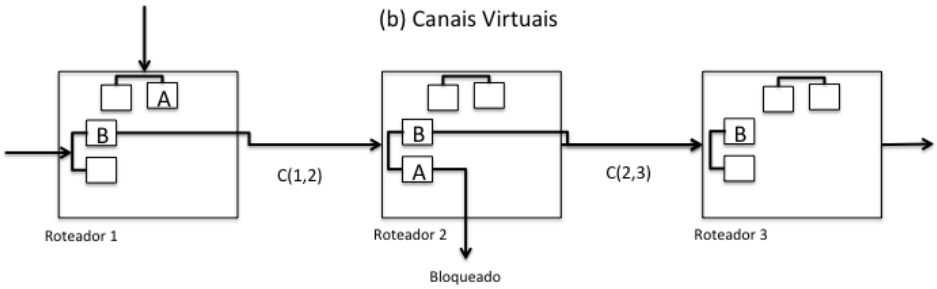


Figura 2.10: Utilizando canais virtuais, o pacote de B foi capaz de utilizar o canal C(1,2) que antes estava ocioso. Isto foi possível devido a adição de *buffers* extras no roteador 2.

Adicionar canais virtuais a uma rede de interconexão é análogo a adicionar pistas

extras a uma estrada. Uma rede sem canais virtuais, é uma rede com uma pista só, aonde se um pacote desta rede fica bloqueado, todos os subsequentes também ficam. Aumentando o número de pistas (canais virtuais), adiciona-se uma alternativa aos pacotes seguintes ao bloqueado, reduzindo assim o congestionamento e aumentando a banda total da rede.

2.2.5 Gerenciamento de *buffer* e *backpressure*

Para gerenciar a utilização dos *buffers* e informar aos nós vizinhos sobre a disponibilidade de seus *buffers*, é implementado o mecanismo de gerência de *buffer*. Desta forma um roteador pode fornecer informações de controles a seu vizinho avisando que seus *buffers* estão cheios e assim não causar um *overflow* de pacotes em seus *buffers*. Atualmente estão em uso três tipos de mecanismos de *backpressure*: créditos, *on/off* e *ack/nack*. Vamos examiná-los a seguir:

- Créditos: Neste mecanismo, para cada *flit* de cada *buffer* é associado um crédito. Toda vez que o nó emissor (roteador *upstream*) quer enviar um *flit*, ele irá verificar a quantidade de créditos no nó receptor (roteador *downstream*). Caso ele possua créditos suficientes, significa que há espaço no *buffer* do *downstream* e não há problemas em enviar o *flit*. Caso não haja créditos, significa que o *buffer* do *downstream* está cheio e não se pode enviar *flits*.

- *On/Off*: Neste mecanismo, antes de enviar um pacote, o roteador *upstream*checa uma *flag* no roteador *downstream* que informa se o os *buffers* desta porta do roteador estão cheios ou não. Quando o *buffer* do *downstream* ficar cheio, ele irá enviar uma mensagem de controle para o *upstream* com esta informação. O *upstream* então irá parar de enviar pacotes. Quando o *downstream* possuir *buffers* disponíveis, ele irá enviar novamente uma mensagem ao *upstream* informando-o que ele está pronto para receber novos *flits*. O *upstream* então retornará a enviar seus pacotes.

  Um cuidado que deve ser tomado ao se implementar este tipo de mecanismo é o tempo de travessia da mensagem de controle. Se o *downstream* esperar seus *buffers* ficarem cheios para enviar a mensagem de controle, durante o tempo que esta mensagem leva para chegar ao *upstream* e ele processá-la, o *upstream* pode enviar pacotes e causar um *overflow* no *downstream*. Para resolver este problema, é implementado um limiar para o *downstream* enviar sua mensagem de cheio. Assim, mesmo que o *upstream* envie pacotes enquanto a mensagem de controle atravessa o canal, o *downstream* nunca sofrerá *overflow*.

- *Ack/Nack*: Neste método, não há um controle sobre a quantidade de *buffers* disponíveis na porta de entrada. O *downstream* apenas informa ao *upstream*

se a mensagem foi processada (ACK) ou se ela foi rejeitada por falta de espaço ($NACK$). Quando a rede está congestionada, este tipo de mecanismo pode ocasionar em muitas retransmissões de *flits*, além do uso excessivo de mensagens de controle.

Para todos os mecanismos de gerenciamento de *buffer* mencionados acima, normalmente utiliza-se um canal especial para transmitir estas informações de controle para que a banda da rede não seja consumida com este tipo de mensagem. Há estudos que propõem a utilização da rede normal de *flits* para a transmissão de mensagens de controle, porém tais estudos estão fora do escopo desta dissertação e podem ser encontrados em [7, 40].

2.3 Roteador Base

Para se ter um entendimento do roteador flexível, faz-se necessária a introdução do roteador base. Este roteador foi utilizado como ponto inicial para serem implementadas as ideias do roteador flexível. Como dito na seção anterior, ele é composto de cinco portas de entradas e cinco portas de saída, onde cada uma é associada a uma direção: Norte(N), Sul (S), Leste(E) e Oeste(W), e uma porta local (L). Também possui um *crossbar* que faz a conexão entre estas portas de entrada e saída, *buffers* para fazer o armazenamento temporário de pacotes, e unidades de roteamento e arbitração. Cada componente do roteador será mais detalhado na seção 2.3.1. A forma como estes componentes se relacionam para fazer a operação de roteamento de pacotes será discutida na Seção 2.3.2. Roteadores modernos tem sua operação dividida em estágios como em *pipeline*. Assim como *pipelines* em CPUs, o *pipeline* do roteador está sujeito a *stalls* que impactam seu desempenho. O *pipeline*, assim como *stalls* serão discutidos na Seção 2.3.2.

2.3.1 Componentes do roteador base

Os componentes do roteador possuem uma função bem específica. Cada um deles será listado e explicado a seguir:

- Portas de entrada e saída: As portas são os canais de comunicação do roteador. É a única parte visível para um observador externo. Tanto as portas de entrada como as de saída possuem finalidades semelhantes e suas implementações são *quasi-idênticas* diferindo apenas no componente a qual são conectadas. As portas de entrada de um roteador estão diretamente conectadas as portas de saídas de seus vizinhos, assim como as portas de saída do mesmo, estão conectadas as portas de entrada de seus vizinhos. Dito isto, sabe-se que sua crucial

diferença é que uma é usada para envio enquanto a outra para recebimento de pacotes.

- FIFO *buffer*: É uma memória utilizada para armazenar pacotes temporariamente e que porventura não podem ser enviados adiante (os motivos serão discutidos em seções a frente). Como dito anteriormente, dependendo da implementação do roteador, os *buffers* podem estar presentes nas portas de entrada ou de saída. Nesta dissertação, consideramos que os *buffers* estão presentes nas portas de entrada.

- Unidade de Roteamento: É a unidade responsável por decidir a porta de saída correta em direção ao seu destino. Esta unidade irá ler o destino do pacote no *head-flit*, e de acordo com a posição atual do roteador irá decidir a porta de saída deste pacote, baseado no algoritmo de roteamento. O roteador base implementado utiliza *XY-Routing* [41]. Este algoritmo, assim como os motivos que levaram à sua escolha são explicados em detalhes na Seção 2.3.3.

- Unidade de Arbitração: Quando mais de uma porta de entrada deseja enviar flits para uma mesma porta de saída, é necessário que seja feita uma arbitração do recurso *crossbar* que fará a movimentação dos *flits* internamente no roteador. A unidade de arbitração é o componente do roteador responsável em colocar uma ordem na alocação deste recurso, selecionando apenas uma porta para utilizar aquele caminho no *crossbar* por um ou mais ciclos. A arbitração é feita utilizando ou não um mecanismo de prioridade pré-definido. Na implementação utilizada nesta dissertação, utiliza-se o mecanismo de prioridade circular *round-robin*, aonde uma porta tem prioridade sobre a outra e a cada ciclo estas prioridades são alteradas de forma que o acesso ao recurso seja feito de maneira justa, ou seja com a utilização igual dos recursos para todos as portas.

- Unidade de Alocação de Canais Virtuais: Esta unidade é a responsável por gerenciar a alocação de canais virtuais em um roteador. Como foi discutido anteriormente, o mecanismo de canais virtuais, fraciona o canal adicionando *buffers* extras ao roteador. Esta unidade faz a alocação destes *buffers* para pacotes. Ela é responsável também por resolver conflitos existentes quando existem mais pacotes do que canais virtuais disponíveis. No roteador base é utilizado um arbitrador *round-robin*.

2.3.2 Operação do Roteador Base

A operação do roteador pode ser separada em quatro estágios distintos de *pipeline* [7] Cálculo de rota, Arbitração de Canal Virtual, Arbitração de *Switch* e Travessia do

Switch. Cada estágio utiliza uma ou mais componentes mencionados anteriormente e serão explicados a seguir:

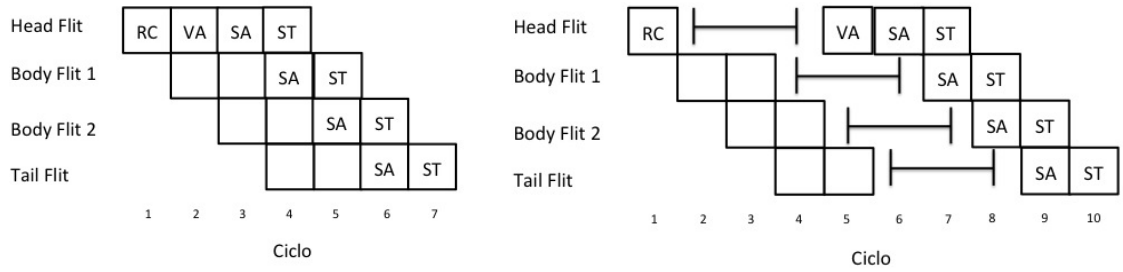
- Calculo da Rota (*Routing Computation - RC*): A unidade de roteamento é o primeiro destino do pacote após ele entrar pela porta de entrada. Informações sobre o destino do pacote são lidas do *head-flit* para que seja definido uma porta de saída para este pacote. A definição da porta de saída é feita pelo algoritmo de roteamento. O algoritmo de roteamento utilizado no roteador base desta dissertação é detalhado na Seção 2.3.3.

- Arbitração de Canal Virtual (*Virtual Channel Arbitration - VA*): Após ser definido uma porta de saída, o *head-flit* é encaminhado para a unidade de arbitração de canal virtual. Nela o pacote irá disputar com outros pacotes destinados para a mesma porta de saída por um canal virtual.

- Arbitração do *Switch* (*Switch Arbitration - SA*): Após ser definido um canal virtual e uma porta de saída, chega o momento em que o pacote irá se dirigir para uma porta de saída. Para fazê-lo, todos os seus *flits* precisam passar pelo *crossbar*, com isso se dá início ao estágio de Arbitração do *Switch*. Este estágio é necessário porque o *crossbar* é um recurso compartilhado e vários pacotes podem estar tentando utilizar o recurso ao mesmo tempo. Neste estágio a unidade de arbitração faz o casamento entre os pedidos dos pacotes e os recursos disponíveis. Assim, para cada porta de saída, é alocada uma porta de entrada. Quando um pacote não consegue avançar neste estágio, ele o irá tentar novamente no próximo ciclo.

- Travessia do *Switch* (*Switch Traversal - ST*): Este é o último estágio de um pacote no roteador. Após o pacote atravessar o *crossbar*, ele é armazenado na unidade de saída e está pronto para ir ao próximo roteador (ou ao consumidor do pacote, caso este seja o último roteador da rota).

Assim como um *pipeline* de processador, o *pipeline* do roteador também pode sofrer *stalls*. Dizemos que ocorreu um *stall* no *pipeline* quando um *flit* fica impossibilitado de avançar para o próximo estágio. Isto causa um atraso não somente neste *flit*, mas também nos *flits* subsequentes ao *flit* parado. O exemplo da Figura 2.11 mostra a travessia de um pacote contendo 4 *flits* atravessando 4 canais. Em 2.11(a), o pacote atravessa os 4 canais sem sofrer nenhum bloqueio. Como pode ser visto, apenas o *head-flit* passa pelas fases de Cálculo da Rota e Alocação de Canal Virtual. Isto se deve ao fato de que estas informações são calculadas apenas uma vez para cada pacote, extraindo informações do *head-flit* e, sendo utilizada para todos os *flits* restantes. Na Figura 2.11(b), o mesmo pacote tenta atravessar os 4



(a) *Pipeline* (b) *Stall*

Figura 2.11: Em (a) o pacote passa por todos os estágios do *pipeline* sem sofrer nenhuma espécie de bloqueio. Em (b) o pacote demora 3 ciclos para conseguir um canal virtual. Devido a este bloqueio, todos os *flits* subsequentes sofrem o mesmo atraso de 4 ciclos.

roteadores, porém fica impossibilitado de alocar um canal virtual durante 3 ciclos, sendo atrasado até que o consiga. Este atraso é propagado para todos os demais *flits* subsequentes do *flit* bloqueado. Neste caso, dizemos que o *pipeline* sofreu um *stall*.

O Roteador Flexível, introduzido no Capítulo 4 tem como objetivo reduzir a quantidade de *stalls* que o *pipeline* sofre devido a impossibilidade de alocação de canais virtuais, aumentando assim o desempenho da rede em termos de *throughput* e latência.

2.3.3 Algoritmo de Roteamento XY

O algoritmo de roteamento DOR-XY (do inglês *Dimension-Order XY*), é determinístico, ou seja, para uma determinada entrada, a saída será sempre a mesma.

Os *flits* de um pacote são primeiramente roteados na coordenada X para então serem roteados na coordenada Y até a chegada ao seu destino. Normalmente a quantidade de travessias de roteadores, ou *hops*, que serão necessários em cada direção, assim como a direção (positiva ou negativa em relação ao eixo) é calculada no nó transmissor e armazenada no *head-flit*. Assim a unidade de roteamento de um roteador só precisa ler este valor e decidir a porta que o pacote deve seguir.

Este algoritmo é capaz de construir sempre o caminho mínimo para um pacote através da checagem dos limites da dimensão do roteador origem e roteador destino em ordem decrescente, fazendo com que não sejam formados ciclos no grafo de dependência dos canais, evitando assim *deadlocks* [42]. Como todos os pacotes que deixam a mesmo nó origem em direção ao mesmo nó destino seguem o mesmo caminho, o algoritmo não precisa lidar com casos complexos como pacotes que chegam em seus destinos fora de ordem. Além disto, esta mesma propriedade o deixa livre de *livelocks*.

Capítulo 3

Metodologia de Conversão Síncrono-Assíncrona - ASERT

Um dos objetivos principais deste trabalho é a construção de um roteador de baixo custo energético. Para isto, foi necessário o estudo de técnicas que reduzem a dissipação de potência nos circuitos síncronos. Desde seu lançamento, circuitos assíncronos têm sido reconhecidos como potenciais provedores de vantagens em relação a sua contraparte síncrona. São inúmeras as propriedades que tornam circuitos assíncronos desejáveis em projetos de *SoCs*, dentre elas a redução do consumo de potência em diversas situações [43–46], frequência de operação dependente do atraso médio dos circuitos ao invés do maior atraso, menor emissão e eletromagnética.

Neste capítulo será apresentado a metodologia de conversão síncrono-assíncrona ASERT. Esta metodologia será utilizada no Capítulo 5 para converter o circuito do Roteador Flexível síncrono para o circuito do Roteador Flexível Assíncrono (RAF). A Seção 3.1 apresenta brevemente a linguagem *Verilog*, a linguagem utilizada para fazer a descrição em hardware dos roteadores. A Seção 3.3 introduz o mecanismo de escalonamento por reversão de arestas, SER, que é usado como base para a metodologia ASERT (Seção 3.4), e finalmente a Seção 3.5 explica detalhadamente a metodologia.

3.1 Linguagens de descrição de hardware

A metodologia ASERT foi criada para ser aplicada em circuitos descritos em *hardware*. Portanto, para que a conversão síncrona-assíncrona possa ser possível, é necessário que o circuito dos roteadores estejam descritos em uma linguagem de descrição de hardware (HDL). Dentre estas, duas são mais conhecidas: Verilog e VHDL.

3.1.1 VHDL

VHDL, que significa Linguagem de Descrição de Hardware de Circuitos Integrados de Alta Velocidade, do ingles *VHSIC Hardware Description Language*, é uma linguagem desenvolvida e otimizada para descrever o comportamento de circuitos e sistemas digitais eletrônicos provinda de um projeto de pesquisa do Departamento de Defesa dos EUA no inicio da década de 80.

VHDL possui diversos mecanismos para descrição do comportamento de componentes eletrônicos, desde simples portas lógicas até microprocessadores completos ou circuitos integrados personalizados. A linguagem permite que características do circuito sejam precisamente definidas, como tempo de transição de subida e descida dos sinais, atrasos intrínsecos das portas lógicas e é claro sua função lógica.

Existem duas estruturas básicas na descrição de um circuito em VHDL: entidade (*entity*) e arquitetura (*architecture*). Comparando com o método de projeto esquemático de circuitos, a entidade é análoga à descrição simbólica, representando o subcircuito através de entradas e saídas, e a arquitetura é análoga à esquemática, onde a verdadeira função lógica do subcircuito é descrita.

3.1.2 Verilog

A linguagem de descrição de hardware *Verilog* foi introduzida em 1985 pela *Gateway Design System Corporation*, agora parte da *Cadence Design Systems*, esta linguagem é largamente utilizada no meio industrial de projetos VLSI. Sua sintaxe é muito similar à linguagem de programação C, o que incentiva seu uso por engenheiros elétricos e de computação, já que muitos a aprendem durante o curso universitário.

A linguagem *Verilog* descreve um sistema digital como um conjunto de módulos, onde cada um possui uma interface para outros módulos a fim de que suas interconexões também possam ser relacionadas na descrição completa do sistema. Os módulos são capazes de operações concorrentes e podem representar desde simples portas lógicas até sistemas completos como um micro-processador.

Verilog foi a linguagem de descrição de hardware escolhida neste trabalho. Isto se deve ao fato de que sua estrutura é mais parecida com programação estrutural, o que tornou mais rápida a aplicação pelo autor.

3.2 Assincronismo global, sincronismo local

Assincronismo global, sincronismo local (GALS - Global asyncrhonous, locally synchronous) é um modelo de computação baseado nos modelos síncronos e assíncronos de computação. Ele permite o relaxamento do modelo síncrono para a construção de sistemas consistindo de diversos blocos denominados "ilhas síncronas"(a parte do

sistema na qual os blocos operam de acordo com o modelo síncrono de comunicação). Estas ilhas se comunicam através de modelos assíncronos de comunicação.

Circuitos GALS consistem de um conjunto de módulos síncronos se comunicando entre si através de *wrappers* assíncronos. Este tipo de modelo possui diversas vantagens dos dois modelos, como baixo consumo energético e interferencia eletromagnética. Devido a isto, GALS tem se popularizado cada vez mais em SoCs.

O modelo de computação GALS é um compromisso entre um sistema totalmente síncrono (com um domínio de clock único) e circuitos totalmente assíncronos (cada módulo pode ser considerado como contendo um domínio de *clock* independente).

3.3 SER - Escalonamento por reversão de arestas

A metodologia de Escalonamento por Reversão de Arestas *(SER - Scheduling by Edge Reversal)* [47] foi elaborada para resolver o problema de ordem de acesso a recursos compartilhados. Esta metodologia foi usada em [48] como uma estrutura de sinalização e sincronismo de processos independentes que compartilham recursos. Esta seção introduz o conceito de SER.

Considerando o grafo orientado conexo G = (N,E), aonde N é um conjunto de processos (nós) e E um conjunto de arestas, para todo processo P1 que compartilha recursos atômicos com outro processo P2, cria-se uma aresta direcionada E(P1,P2) que conecta os dois nós. Uma orientação acíclica de G é uma função no formato ω :E $\rightarrow$ N de forma que não existam ciclos direcionados.

Se ω é uma orientação acíclica de G, então alguns nós possuem todas as arestas incidentes voltadas para eles. Denominamos estes nós sumidouros. Similarmente, os nós em que todas as arestas são voltadas para outros nós são denominados fontes. Todos os nós sumidouros podem operar enquanto os demais nós devem permanecer inativos. Como apenas os nós sumidouros operam, os nós vizinhos, ou seja, os que compartilham recursos com os sumidouros não operam, o que garante a exclusão mútua a qualquer acesso aos recursos compartilhados entre eles.

Após a operação, um nó sumidouro reverterá a orientação de todas as suas arestas, tornando-se um nó fonte, e assim possibilitando o acesso dos recursos aos seus vizinhos. Uma nova orientação acíclica é formada e todo o processo é então repetido para o novo conjunto de sumidouros. Assumindo que ω' = g(ω) simboliza esta operação, SER pode ser considerada como a aplicação "infinita"de g(ω) sobre G. Assumindo que G é finito, é visível que um conjunto de orientações acíclicas se repetirá. Denominamos um período de operação p, o conjunto de estados dos nós resultantes de operações acíclicas aplicadas sobre os mesmos que, dentro de

um período de tempo determinístico, se repetirão ciclicamente. Como em cada orientação acíclica há pelo menos um sumidouro, ou seja, um nó autorizado a operar, garante-se que nenhum *deadlock* ou *starvation* ocorrerá. Além disto, está provado que em qualquer período p, todos os nós operam exatamente m vezes [47, 49]. A Figura 3.1 ilustra a dinâmica do SER.

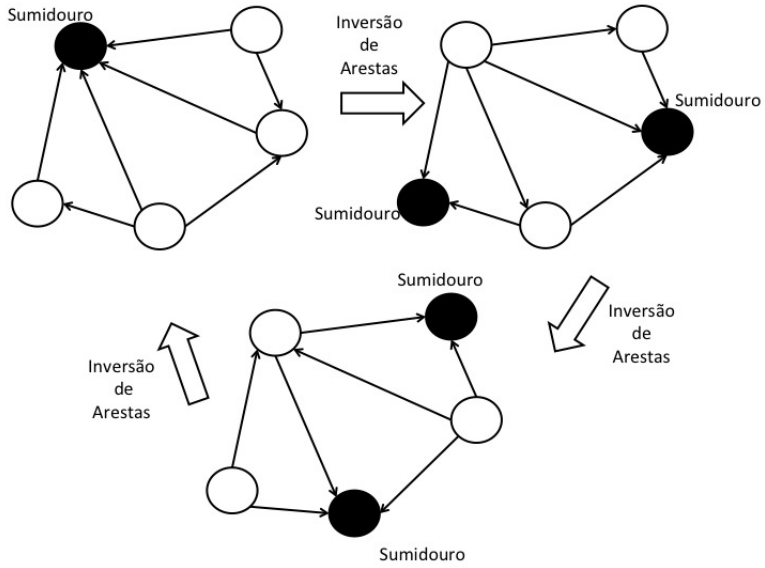


Figura 3.1: Dinâmica do SER. Os nós sumidouros são representados por círculos cheios. A cada ciclo, os nós sumidouros invertem todas as suas arestas.

3.4 ASERT

A metodologia ASERT [30] - *Asynchronous Scheduling by Edge Reversal Timing* descreve um ambiente de trabalho que permite o compartilhamento de recursos comuns entre processos independentes. Todos os processos do sistema operam de forma independente, de forma que as restrições só ocorrem quando relacionadas à exclusão mútua com seus vizinhos e a sua própria temporização. Do ponto de vista arquitetural, não existem restrições quanto ao número de processos ou o tipo de comunicação que ocorre entre eles.

A implementação da metodologia ASERT requer a elaboração de dois circuitos: O controlador de arestas e o controlador de nó. Ambos são explicados a seguir. A direção da aresta é representada por uma ficha. Esta variável é compartilhada por dois nós e quando um dos nós faz a aquisição da ficha, representa que a aresta está voltada em sua direção.

- O **Controlador de Aresta** cuida da sinalização entre dois nós vizinhos, assimilando o nível lógico alto como a direção que a aresta está apontando. A representação de uma aresta sendo apontada é dada por uma ficha, sendo que

se um nó possui a ficha, a aresta está apontando para este. Uma das possíveis implementações deste circuito é uma porta lógica do tipo XOR e outra do tipo INVERSOR como mostra a Figura 3.2, tendo como entrada os sinais de fim de operação (*End-of-operation EOP1* e *EOP2*) de cada um dos nós conectados e como saídas os sinais indicando a direção da aresta (Ficha e ∼Ficha).

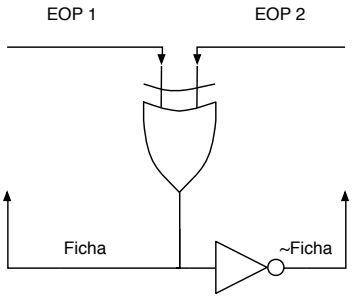


Figura 3.2: Uma possível implementação do Controlador de Aresta.

- O **Controlador de Nó** é o circuito utilizado para habilitar ou desabilitar a operação de um nó. O sinal de habilitação e desabilitação do nó é gerado através da observação da presença de fichas. Podendo ser implementado através da utilização de uma porta AND, como pode ser visto na Figura 3.3. Cada entrada do controlador de nó é conectada a uma saída de um controlador de aresta vinda de um dos nós vizinhos, enquanto sua saída *opera* é conectada ao módulo para permitir o início da operação do circuito relativo ao nó. O sinal de *fim de operação EOP* pode ser implementado de diversas formas, a forma utilizada neste trabalho será explicada na Seção 5.2.2. A entrada *run* é usada como um sinal global de permissão de operação e, caso necessário, é utilizado para permitir o bloqueio de operação do nó.

A Figura 3.4 apresenta uma implementação de como são interconectados um controlador de arestas e dois nós utilizando apenas portas lógicas convencionais.

3.5 Metodologia de Conversão de Circuitos Síncrono Assíncrono

A metodologia de conversão de circuitos síncronos para assíncronos é baseada na associação de cada módulo da descrição de hardware em um nó do modelo ASERT. O processo de geração de nós é baseado na extração da informação de interconexão entre módulos. A correlação entre módulos e nós ASERT, assim como conexões entre as arestas ASERT serão discutidas a seguir.

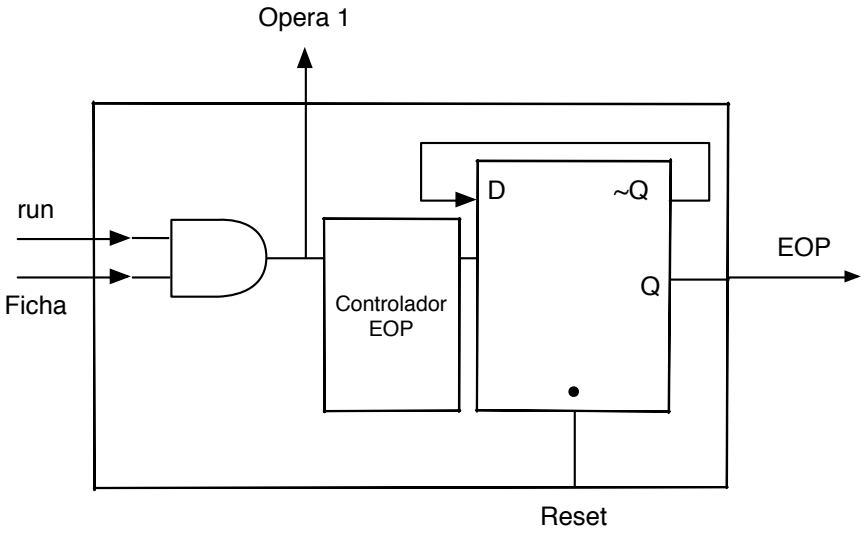


Figura 3.3: Uma possível implementação do Controlador de Nó.

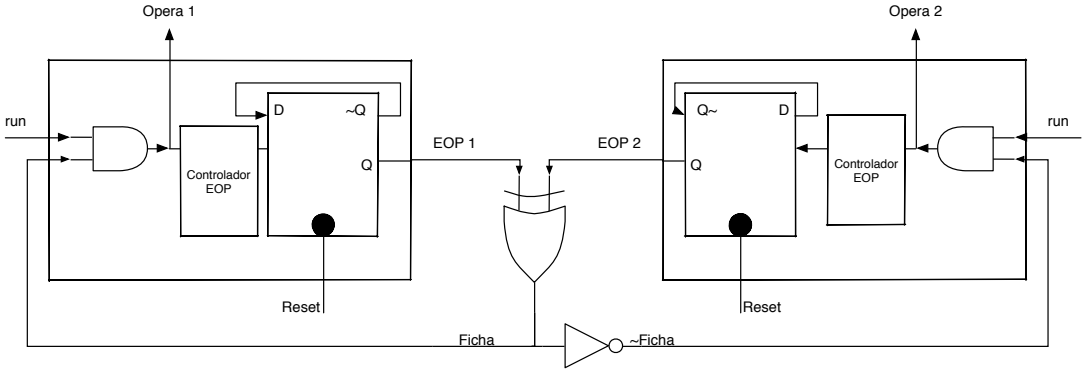


Figura 3.4: Junção de dois controladores de Nó e Arestas.

3.5.1 Correlação entre módulos e nós ASERT

Na metodologia ASERT, um nó é uma unidade básica, representada como uma função básica de funcionamento que depende da permissão de seus vizinhos para operar. Também é dever desta entidade, fazer a comunicação para seus vizinhos, informando-os sobre o término de sua operação. Para criar estas unidades básicas, a metodologia de conversão baseia-se nos blocos originais e hierarquias criadas pelo projetista através da linguagem de descrição de hardware *(HDL - Hardware Description Language)*. Esta seção irá apresentar como os blocos e nós estão associados uns com os outros.

Sistemas são descritos em uma HDL como um conjunto de blocos interconectados. Estes blocos chamados de módulos em *Verilog*, são definidos por uma interface de entradas e saídas e pela descrição da sua funcionalidade. Os projetistas baseiam-se nos módulos para encapsular arquiteturas bem definidas, de forma que a comple-

xidade da estrutura total seja simplificada quando instanciam-se estes elementos de forma hierárquica. O entendimento do projetista sobre a operação geral do sistema é usado no mecanismo de conversão para co-relacionar as funcionalidades comuns dos módulos em nós ASERT. Estes elementos dependem de outros recursos para serem capazes de operar assim como um módulo depende da disponibilidade de entradas válidas para ser capaz de prover uma saída consistente.

Como exemplo, a Figura 3.5 representa o módulo sequencial M que possui duas entradas Entrada 1 e Entrada 2, quatro módulos internos M1, M2, M3 e M4 e uma saída denominada Saída 1. Neste caso, Saída 1 assume o valor exclusivamente dependente do conjunto de entradas Entrada 1 e Entrada 2, que por sua vez são fornecidas por outros módulos.

Assumindo que um sinal é associado indicando que um novo valor está disponível para cada entrada, é possível reconhecer o instante em que o módulo M é capaz de gerar uma saída coerente Saída 1, produzindo um sinal quando alcança o final de sua operação. É importante ressaltar que a expressão "novo valor" não se refere necessariamente a um valor diferente do anterior, mas sim o final do processamento do sinal das entradas do módulo, que poderia ser um valor diferente ou não. A Figura 3.6 mostra o diagrama de dependências dos sinais do módulo M, assim como suas entradas e saídas.

O nó M representa o módulo M, círculos E1 e E2 representam os geradores de entrada Entrada 1 e Entrada 2 do módulo M, e o nó S1 representa o receptor Saída 1 do módulo M. Observe que existe uma relação próxima entre a representação do circuito e a metodologia ASERT. Assim como os nós controlados pelo ASERT, os entradas e saídas do módulo devem possuir todas as suas dependências disponíveis para que seja possível a operação de forma consistente para que um resultado correto seja produzido. Este resultado será então consumido pelo módulo conectado a esta saída.

3.5.2 Correlação entre interconexão de módulos e arestas ASERT

Arestas, assim como nós, tem um papel fundamental na metodologia ASERT para circuitos assíncronos. Elas são responsáveis por identificar o instante adequado de execução de cada um dos dois nós interconectados por elas, baseado na informação provida pelo sinal de fim de operação de cada um dos nós.

Interconexões podem ser vistas como um recurso comum compartilhado por dois blocos. O que provê uma saída e o que lê esta informação. Esta condição ocorre sempre que os módulos produtores fornecem uma saída válida e o modulo consumidor está pronto para receber estes dados, o que pode ser facilmente modelado

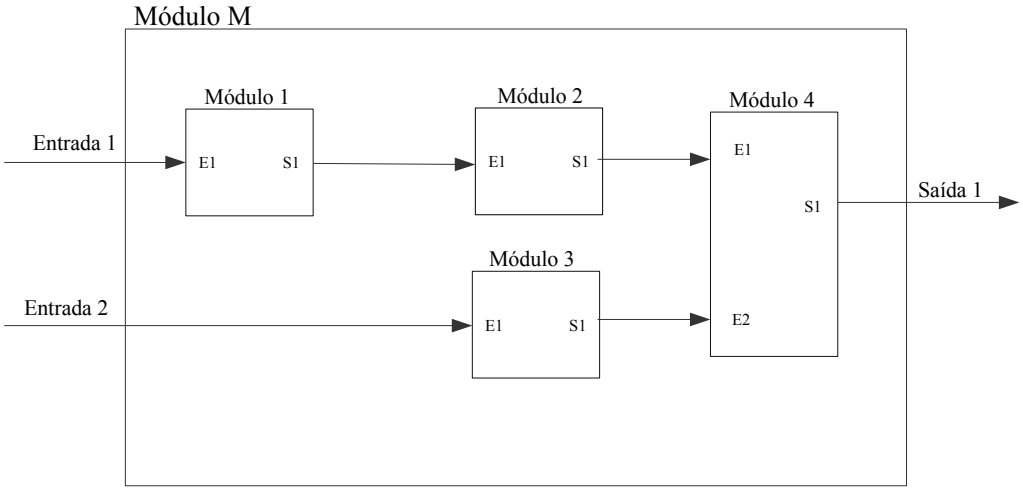


Figura 3.5: Modulo M. Este módulo necessita que Entrada 1 e Entrada 2 possuam valores coerentes para que os mesmos sejam processados pelos seus módulos internos Módulo 1, Módulo 2, Módulo 3 e Módulo 4, para gerar uma saída coerente.

com a metodologia ASERT. A interconexão de um circuito receptor em particular é representada por um nó extra acoplado ao nó receptor e uma aresta oriunda do nó emissor conectada à este nó extra. A estrutura presente permite operações paralelas para ambos os blocos, o emissor e o consumidor até que outra troca de informações ocorra.

O módulo M, Figura 3.5 representa um circuito descrito como quatro módulos interconectados, aonde M1 e M3 são blocos que recebem as entradas Entrada 1 e Entrada 2 para poderem gerar dados. O módulo M2 consome os dados oriundos da saída Saída 1 de M1 e produz dados para M4. O último módulo interno de M, M4,consome os dados oriundos das saídas de M2 e M3, os processa e então produz a saída Saída 1.

M2/S1 terá um valor coerente quando M1/E1 gerar uma saída válida utilizando os valores de E1. Da mesma forma, M3 e M2 devem sinalizar a M4 de que novos valores em M2/S1 e M3/S1 foram calculados de forma que M4 possa fazer uso deste dado. Este mecanismo de compartilhamento de recursos e sinalização é resolvido perfeitamente pela combinação de nós e arestas ASERT, visto que a informação de fim de operação é trocada entre estes dois nós. Figura 3.7 mostra como o circuito da Figura 3.5 seria sinalizado utilizando ASERT.

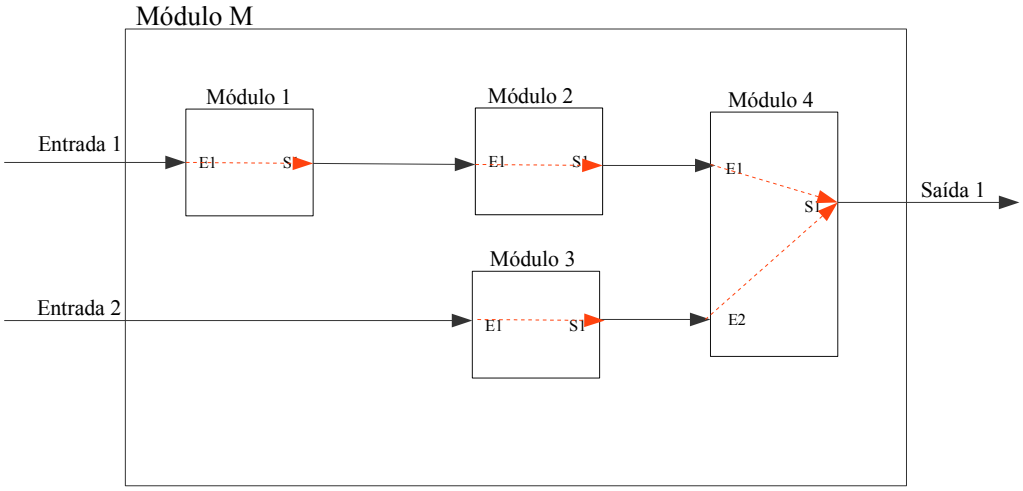


Figura 3.6: Cone de dependências do módulo M.

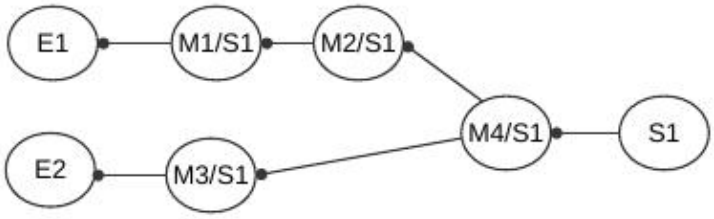


Figura 3.7: Diagrama SER do módulo M utilizando o cone de dependências da Figura 3.6.

De fato, todas as informações necessárias para se criar as arestas que permitem a conversão síncrona-assíncrona estão disponíveis no código que descreve o circuito. Se extrairmos todas as saídas dos módulos e identificarmos os módulos nos quais estas saídas estão conectadas, a tarefa de gerar os nós e arestas ASERT se torna simples.

Associando-se nós ASERT para cada saída registrada dos módulos, assim como para cada entrada primaria e saída do arquivo de descrição de hardware e arestas para cada uma das suas interconexões, o diagrama ASERT pode ser representado como na Figura 3.7.

O último passo é gerar os nós e arestas necessárias, conectar estes blocos e

substituir o sinal de clock pelo sinal de opera dos controladores de nó, como mostrado na Figura 3.8.

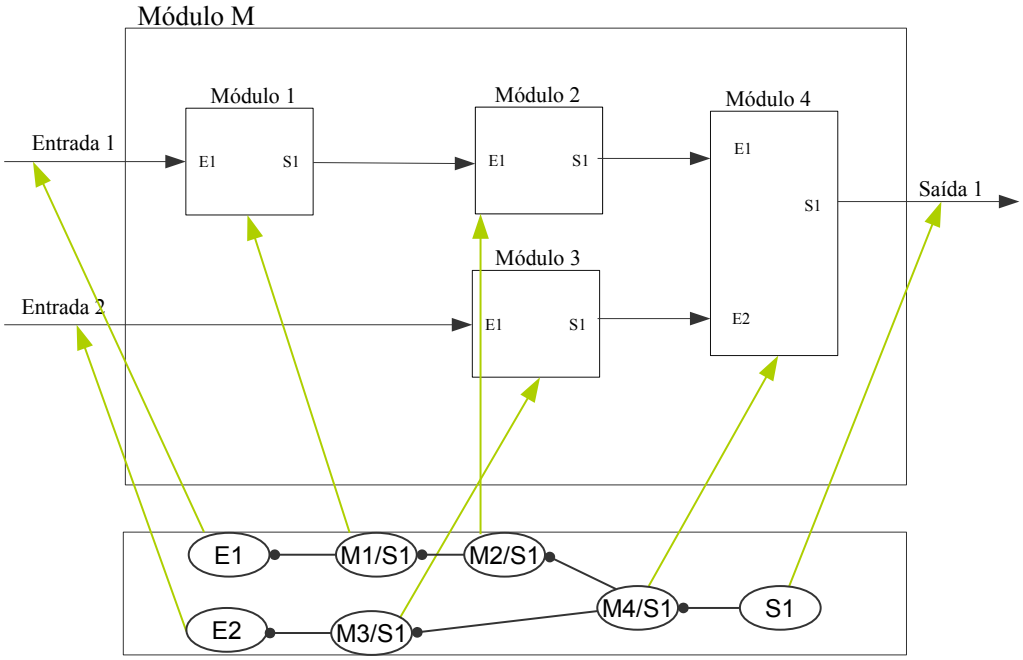


Figura 3.8: Módulo M em conjunto com o módulo do ASERT.

3.6 Resumo do Capítulo

Neste capítulo foi introduzido a linguagem de descrição de hardware (HDL) Verilog, além de explicarmos a metodologia SER. Foi mostrado de que forma a metodologia ASERT utiliza o SER para gerar os sinais de *timming* para os circuitos. Foi mostrado um exemplo descrevendo o passo-a-passo de como aplicar a metodologia ASERT em circuitos.

Capítulo 4

Roteadores Flexíveis

Estatisticamente, nem todos os *buffers* de um roteador de *NoC* são utilizados durante a execução de um sistema [9]. De fato, a utilização dos *buffers* tem relação direta com o desempenho de uma *NoC*. Quando o tráfego na rede está intenso, quanto maior for a taxa de utilização dos *buffers*, maior poderá ser a quantidade de pacotes que um roteador consegue guardar, sem causar o descarte de pacotes. Com a utilização de *buffers* sub-utilizados aumenta-se "virtualmente"o tamanho dos *buffers*, aumentando também a elasticidade dos mesmos (ajustando-se dinamicamente ao tráfico local), permitindo um relaxamento nas condições de contenção de fluxo e elevando o *throughput* da rede, visto que existirá menos contenção.

Em [50] foi visto que armazenar um pacote dispende mais energia do que enviar o pacote em si. Tendo isto em vista é desejável que o projetista do sistema organize os *buffers* de forma a extrair o máximo de desempenho dos mesmos.

Para que seja melhorada a utilização dos *buffers* de um roteador, torna-se necessário fazer certas alterações no mecanismo de alocação dos mesmos, de forma a atender os requisitos do sistema aumentando a utilização média de cada um. Portas de entrada, com baixa frequência de acesso, podem disponibilizar seus recursos para a utilização por outras portas de entrada, que possuem taxa de acesso superior, reduzindo assim a quantidade de *pipeline stalls* no estágio da alocação de canais virtuais do roteador *upstream*.

Um outro ponto que foi levado em consideração é que o mecanismo responsável por esta alocação diferenciada deve ser simples o suficiente de forma que a operação original não seja afetada, e concomitantemente, adicionando o mínimo de lógica extra necessária para sua implementação.

Este capítulo apresenta o roteador flexível e a extensão dos trabalhos anteriores [27**?** , 28]. O objetivo deste trabalho é propor uma extensão dos mecanismos propostos em trabalhos anteriores, que leva em consideração os tópicos mencionados acima e tem como finalidade fazer um melhor aproveitamento dos *buffers* disponíveis no roteador. Para tal, foram implementados os mecanismos de reconfiguração de

buffers do roteador flexível.

O trabalho desenvolvido é baseado no discernimento daquele que projetou o roteador flexível e no modo como o mesmo dividiu os blocos funcionais básicos. Foi utilizado como base o roteador apresentado no capítulo 2, de forma que o funcionamento básico continua sendo mesmo.

A arquitetura do roteador flexível é apresentada na próxima seção. Seu modo de operação é então explicado após a arquitetura. Já os problemas relacionados a *deadlock*, assim como sua solução serão apresentados na Seção 4.4.1 e por último é mostrado o algoritmo de alocação de canais virtuais definitivo.

4.1 Arquitetura e Pipeline do Roteador Flexível

O Roteador Flexível é uma nova arquitetura de roteador que permite uma melhor utilização dos *buffers* em sua totalidade. Através de um mecanismo simples de controle, os *buffers* de uma porta de entrada se tornam disponíveis para pacotes oriundos de qualquer outra porta de entrada. Do ponto de vista arquitetural, não há restrições quanto a seleção de um *buffer* que um pacote pode utilizar. Desta forma o desempenho do roteador em termos de *throughput* e latência é aumentado.

Quando um pacote entra em sua fase de escolha de canal virtual (*virtual channel arbitration*), o mesmo enviará um pedido de canal virtual ao próximo roteador definido na rota do pacote (*downstream*). O *downstream* ao receber este pedido de alocação de canal virtual faz uma busca em seus *buffers* e os aloca caso possua espaços disponíveis. O que torna o roteador flexível diferente do base é o espaço de busca nos *buffers* dos canais virtuais. Enquanto o roteador base limita seu espaço de busca à apenas sua porta de entrada, no roteador flexível, cada porta buscará também nas outras portas de entrada, caso a própria não possua mais *buffers* disponíveis. Utilizando mecanismos especiais, a porta de entrada em que está sendo solicitado um canal virtual, envia pedidos a outras portas de entrada caso não possua o recurso, aumentando assim a probabilidade de alocação de um *buffer* para o pacote. As outras portas ao receberem o pedido, o respondem informando suas respectivas disponibilidades.

Para ilustrar esta situação, a Figura 4.1 mostra três roteadores de uma *Mesh 2D* unidirecional interligados. Na figura, o pacote A entrou pela porta Norte do Roteador 1, adquiriu o canal C(1,2) que interliga os roteadores 1 e 2, e ficou bloqueado no Roteador 2 quando tentava sair pela porta Sul do mesmo. No mesmo instante, um outro pacote, B, entrou pela Norte do Roteador 1, foi roteado pelo canal C(1,2) e também ficou bloqueado no Roteador 2 quando tentava sair pela porta Norte do mesmo. Pouco tempo depois, entra pela porta Oeste do Roteador 1 o pacote C que tem como destino o Roteador 3 e precisa ser roteado pelas portas Leste dos

roteadores 1 e 2 até chegar ao seu ponto final. A Figura 4.1(a) mostra a situação em que utiliza-se o mecanismo do roteador convencional, quando os dois canais virtuais da porta de entrada Oeste do Roteador 2 estão bloqueados. Neste caso, o pacote C ficará bloqueado no Roteador 1 a espera da liberação de um dos canais virtuais da porta Oeste do Roteador 2. Os canais C(1,2) e C(2,3) estão ociosos devido aos bloqueios causados pelo pacote A e B. No entanto, o pacote C não pode utilizar esta banda por não existir *buffers* disponíveis na porta de entrada Oeste do Roteador 2.

A Figura 4.1(b) mostra a mesma configuração de roteadores e a mesma situação de pacotes que ocorreu na Figura 4.1(a). Porém, neste caso, a arquitetura do roteador flexível é utilizada. Como pode ser conferido na figura, quando a porta de entrada Oeste do Roteador 2 não encontra espaço em seus *buffers* locais, ela emite um pedido para as outras portas. A porta Norte então responde o pedido concedendo um de seus *buffers*, acomodando assim o pacote C. Então, o *buffer* Norte do Roteador 2 que estava ocioso, pode ser alocado para o pacote C, fazendo assim com que os canais C(1,2) e C(2,3) que outrora foram deixados ociosos, pudessem ter sua banda utilizada, já que no próximo ciclo o pacote C pode ser transferido para o Roteador 3.

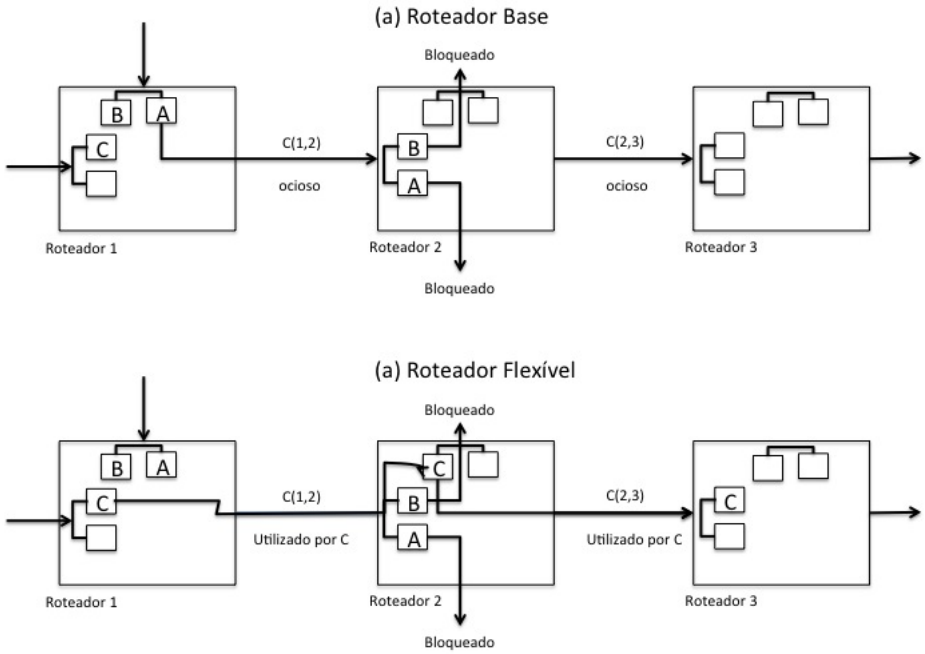


Figura 4.1: (a) Utilizando o roteador base, quando os pacotes A e B bloqueiam, o canal C(1,2) fica ocioso devido a sua associação com os *buffers* da porta Oeste. Embora o pacote C precise utilizar o canal ocioso, ele fica impossibilitado. (b) Com o roteador flexível, o pacote C é capaz de prosseguir ao seu destino sem sofrer nenhuma espécie de bloqueio utilizando o *buffer* Norte do Roteador 2.

O exemplo anterior mostra um caso em que um bloqueio de um pacote no Rote-

ador 1 é evitado graças ao mecanismo de alocação de *buffers* do Roteador Flexível. Como foi evitado o bloqueio do pacote, seu tempo de travessia da rede foi reduzido. O que resulta em uma redução na latência total da rede. Reduzir a latência da rede significa que pacotes estão utilizando menos tempo para chegar ao seu destino, liberando assim, recursos que outrora estariam bloqueados, permitindo então que mais pacotes sejam inseridos na rede e consequentemente aumentando o *throughput* total.

Embora na arquitetura flexível o espectro de busca de *buffers* seja maior quando comparado ao base, o que torna o processamento mais complexo, o circuito que faz a busca de canais virtuais foi projetado de forma a adicionar o mínimo de lógica extra no sistema. Fazendo assim com que seu *overhead* tanto em área, quanto em tempo de execução seja factível.

4.2 Arquitetura do Roteador Flexível

A Figura 4.2 mostra a arquitetura do roteador flexível. Como pode ser visto, a construção do mesmo é similar a de um roteador tradicional. Ambos possuem cinco portas de entrada e cinco portas de saída (Norte, Sul, Leste, Oeste e Local), cada porta contendo um determinado número de *buffers*, um *crossbar* que interliga as portas de entrada às portas de saída, um arbitrador e uma unidade para a alocação de canais virtuais. De fato, a única diferença que pode ser vista arquiteturalmente é na forma em que a porta de entrada é organizada. Adiciona-se elementos extras neste módulo (um multiplexador e o Controlador de Flexibilidade do FIFO - CFF). para que seja possível executar a lógica de alocação diferenciada do roteador proposto em [27**?** , 28].

Alguns elementos arquiteturais, embora sejam os mesmos, possuem uma lógica de controle diferente. Os elementos que compõe o Roteador Flexível são explicados detalhadamente abaixo:

1. **Porta de Entrada**: É o modulo pelo qual os pacotes entram no roteador. Este módulo encapsula o Controlador de Flexibilidade do *FIFO*, a unidade de roteamento e o controlador do *switch*. A única diferença entre a versão flexível e a versão base deste módulo é a substituição do controlador de canais virtuais padrão pelo módulo responsável pela alocação de *buffers* diferenciada, além da adição de um multiplexador que irá redirecionar os pacotes entrantes para outras portas. A porta de entrada modificada pode ser conferida na Figura 4.3.

2. **Gerenciador da *FIFO***: Este é o módulo que faz o controle de leitura e escrita do *buffer*, além de sinalizar quando ele esta cheio ou vazio. Embora

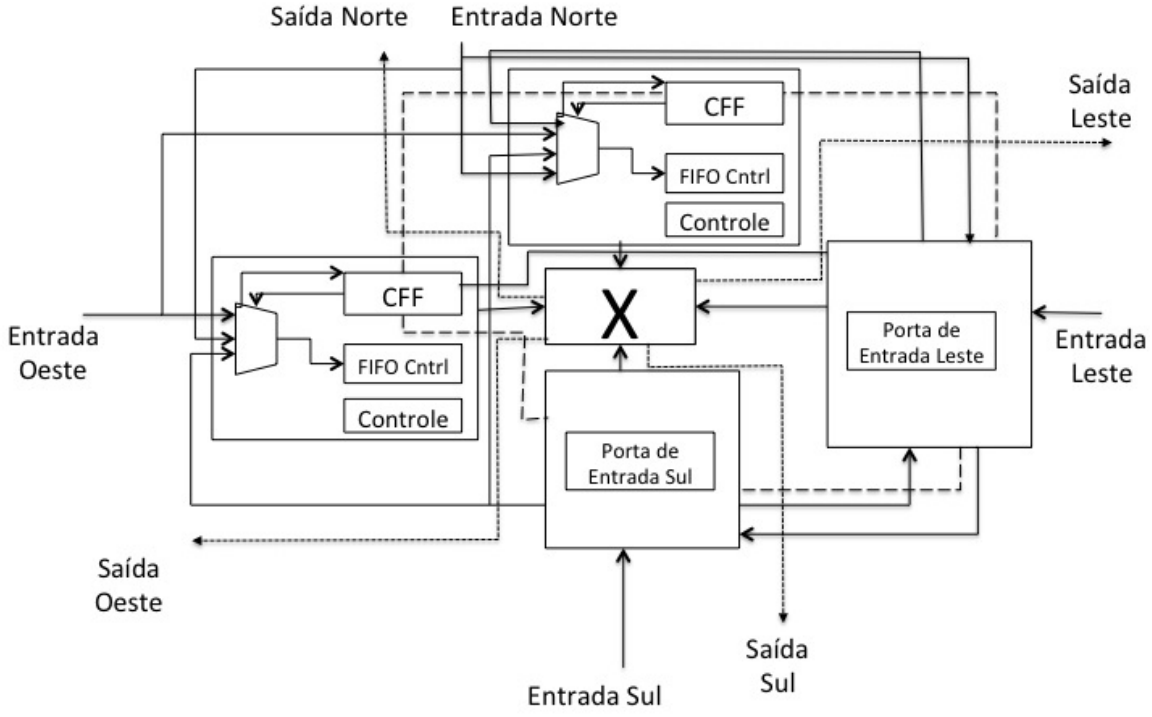


Figura 4.2: Arquitetura do roteador flexível.

do ponto de vista lógico, este *buffer* seja inalterado, a diferença entre a versão encontrada no Roteador Flexível para a versão do roteador base é a fonte pelo qual os pacotes armazenados se originam. Na arquitetura flexível, o *buffer* pode guardar pacotes oriundos de todas as portas (Norte, Sul, Leste, Oeste e Local). Neste trabalho, diferentemente dos anteriores sobre o roteador flexível, foi testado o controle de fluxo *wormhole*, ou seja, o controle com precisão a nível de *flits* ao invés de pacotes como nos trabalhos anteriores.

3. **Controlador de Flexibilidade do *FIFO* (CFF)** (do inglês *Fifo Flexibility Controller - FFC)* [27]: Este é o módulo que substitui o alocador de canais virtuais no roteador flexível. Este módulo foi introduzido com o intuito de fazer a alocação dos *buffers* de forma diferencial, alocando *buffers* de forma global, ou seja, cada porta de entrada pode utilizar qualquer *buffer* pertencente a qualquer outra porta de entrada dentro do mesmo roteador. A arquitetura assim como o mecanismo de funcionamento deste módulo serão explicados na Seção 4.3 mais a frente.

4. **Unidade de Roteamento**: Esta unidade irá definir uma porta de saída para o pacote. Como foi explicado anteriormente, esta decisão é tomada tendo como base o algoritmo de roteamento XY. Diferentemente do roteador base, o Roteador Flexível precisa de informações sobre o próximo *hop* do pacote, por isto o roteamento é calculado um ciclo antes.

5. **Alocador de *Switch***: Não foram necessárias alterações neste módulo, visto que ele apenas aloca o *switch* para os canais virtuais requerendo a mesma porta de saída.

6. ***Crossbar***: Como a quantidade de canais virtuais fisicamente continua sendo a mesma, este módulo não precisou ser alterado.

7. **Porta de Saída**: Assim como os módulos de alocação de *switch* e o *crossbar*, não foram necessárias alterações neste módulo.

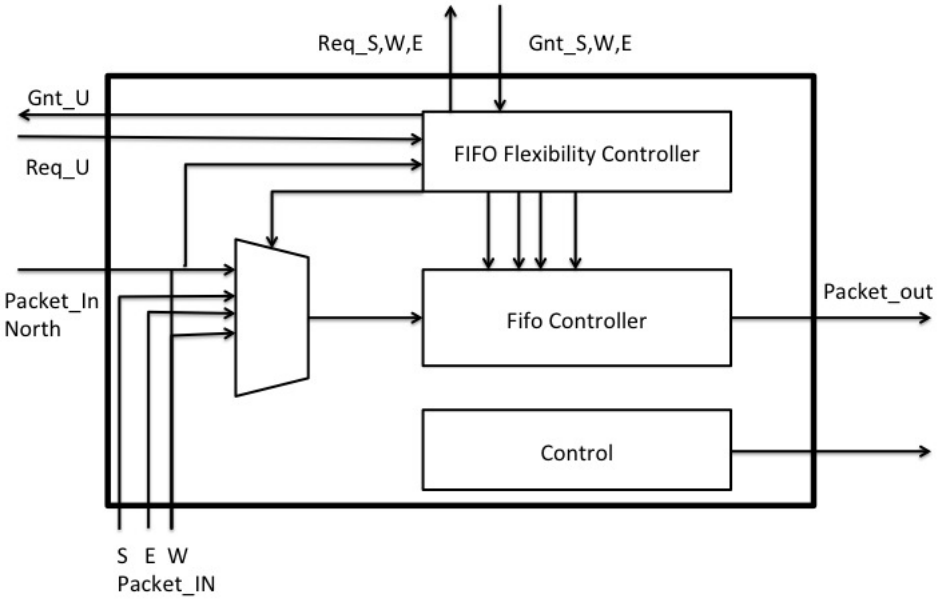


Figura 4.3: Porta de Entrada modificada com CFF - Controlador de Flexibilidade do FIFO.

A junção dos elementos mencionados anteriormente compõe o Roteador Flexível. Sua unidade principal, o CFF - Controlador de Flexibilidade do FIFO, será explicado detalhadamente na próxima seção.

4.3 Controlador de Flexibilidade do FIFO

Para que seja feita uma alocação de *buffers* global sem afetar de forma crítica o desempenho do roteador, se faz necessário a criação de um mecanismo de execução rápida e simples, com a adição mínima de *overhead* em termos de área, consumo de energia e tempo de execução.

O desenvolvimento de tal mecanismo foi pensado também de forma que fosse fácil de ser implantado a partir do roteador base, sem adicionar muito custo de

lógica extra. Tendo em vista as limitações inerentes à construção de roteadores para NoC criou-se o Controlador de Flexibilidade do *FIFO*.

Este módulo, o responsável por fazer a alocação de *buffers* diferenciada no roteador flexível, possui um algoritmo de busca de *buffers* simples que faz com que a busca completa em todos os *buffers* internos, assim como a emissão de uma resposta ao requerente seja concluída em um ciclo. Isto é possível devido a forma de como a checagem de disponibilidade de *buffers* é feita.

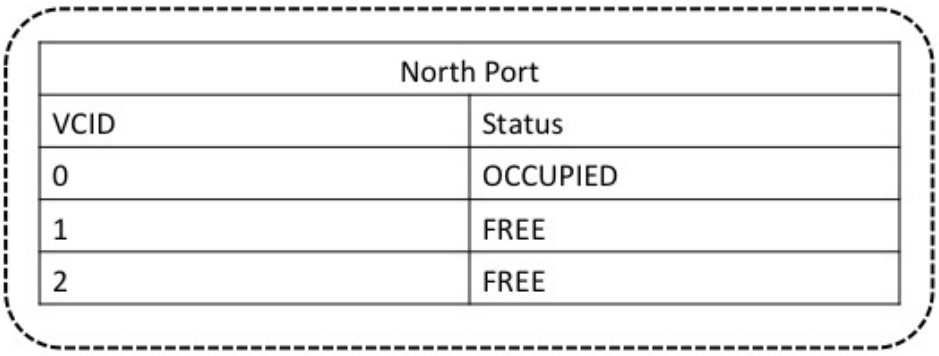
Para explicar a operação do CFF, deve ser definida a noção de canal virtual nativo e canal virtual ádvena. Considera-se um canal virtual nativo, o relativo a uma determinada porta de entrada, e que no roteador base é de uso exclusivo da mesma. Considera-se um canal virtual ádvena, o que pertence a outras portas dentro do mesmo roteador, e que é temporariamente alocado para uma porta de entrada diversa da original. É importante verificar que a rotulação de um *buffer* como nativo ou ádvena, varia do ponto de vista de cada porta de entrada. Por exemplo: Os *buffers* da porta de entrada Norte, são considerados nativos para a porta de entrada Norte e ádvenas para as demais.

O trabalho do CFF se resume em fazer a alocação destes canais virtuais ádvenas caso os nativos estejam indisponíveis. Isto é feito de forma que quando um determinado roteador *upstream* faz uma requisição de canal virtual ao roteador *downstream*, o CFF, que se encontra no *downstream*, verifica com o controlador FIFO se existe um canal virtual nativo disponível para armazenar o novo pacote, caso não seja encontrado, ele irá emitir um pedido de canal virtual para um outro CFF em outra porta de entrada. Feito isto, ele irá aguardar a resposta do CFF em outras portas: Caso exista algum canal virtual disponível, ele irá retransmitir o VCID para o *upstream* e, caso não exista, ele irá emitir uma mensagem de *buffer* cheio.

Para verificar a disponibilidade dos *buffers* em outras portas de forma eficiente, faz-se necessário o emprego de uma estrutura auxiliar chamado de **Tabela de Disponibilidade Interna de *Buffers* (TDIF)**. Esta estrutura auxiliar pode ser implementada como uma tabela que guarda a informação da disponibilidade de todos os *buffers* do roteador. É importante que esta tabela seja acessível para consulta por qualquer CFF. Quando não for possível encontrar um canal virtual local, o CFF irá emitir o pedido de canal virtual àdvena. Antes de um pedido de canal virtual ádvena ser emitido, o CFF irá checar na TDIF a disponibilidade de *buffers* em outras portas. Isto é feito para que um pedido não seja emitido para uma porta que também não possui canais virtuais disponíveis. O formato da tabela pode ser visto na Figura 4.4. Ela consiste basicamente do identificador dos canais virtuais e suas respectivas disponibilidades.

Como também pode ser visto na Figura 4.3, cada CFF necessita de canais auxiliares para fazer a troca de informações com os outros CFFs do roteador. Estes

canais irão transportar as mensagens necessárias para fazer a alocação diferenciada dos *buffers* no roteador.



North Port	
VCID	Status
0	OCCUPIED
1	FREE
2	FREE

Figura 4.4: Tabela de Disponibilidade Interna de Buffers - TDIF: A tabela armazena o identificador do canal virtual e seu *status*. As informações da tabela são atualizadas assíncronamente.

Quando um *flit* chega a uma porta de entrada, é lido o seu identificador de canal virtual (VCID) e através desta informação, o *flit* é multiplexado para a porta correta. Por exemplo: Um *flit* chega na porta Norte do roteador, porém ele possui um VCID da porta Oeste. Quando esta informação for lida no multiplexador, ele irá ser encaminhado para o controlador do FIFO da porta Oeste, aonde irá ser armazenado. Todo *flit* passa por este processo independentemente de ser local ou ádvena.

A Figura 4.5 mostra de que forma os *buffers* do roteador podem ser reconfigurados. Primeiramente, a profundidade dos roteadores deve ser definida durante o projeto, neste exemplo foi usado um *buffer* de 4 (quatro) *slots* e o tamanho dos pacotes foram definidos como 2 (dois) *flits*. No exemplo da Figura 4.5(a) cada canal virtual possui 4 *slots* de *buffer* locais exclusivos para suas respectivas portas de entrada. Agora suponha que o tráfego se torna desbalanceado e as portas Leste e Sul se tornam mais requisitadas. Suponha também que as portas Oeste e Norte tenham suas requisições reduzidas. O roteador flexível irá tentar distribuir a carga das portas leste e sul nas outras portas. Desta forma, pacotes que na versão básica do roteador ficariam bloqueados no *upstream*, podem ser transmitidos e alocados no roteador *downstream*. Uma possível configuração de canais virtuais pode ser conferida na Figura 4.5(b), quando *buffers* de outras portas de entrada foram alocados temporariamente.

É importante observar que mesmo que a porta de entrada Oeste armazene pacotes da porta de entrada Leste, isto não impede esta de armazenar pacotes de outras portas. Não existe restrições quanto a mistura de pacotes no mesmo *buffer*, desde que os pacotes estejam ordenados e armazenados completamente. Isto somente é possível porque o mecanismo básico do canal virtual não foi alterado. A informação

de roteamento que é extraída do *header* do pacote é utilizada para rotear todos os *flits* subsequentes até que seja encontrado um *tail flit*. Este mecanismo só é possível devido a não-intercalação de pacotes. De forma que quando é definida uma rota para os *flits* de um canal virtual, sabe-se que todos os *flits* subsequentes estão se direcionando para a mesma porta de saída. Para que isto seja possível, também é proibido que pacotes da mesma mensagem sejam alocados em múltiplos diferentes canais virtuais.

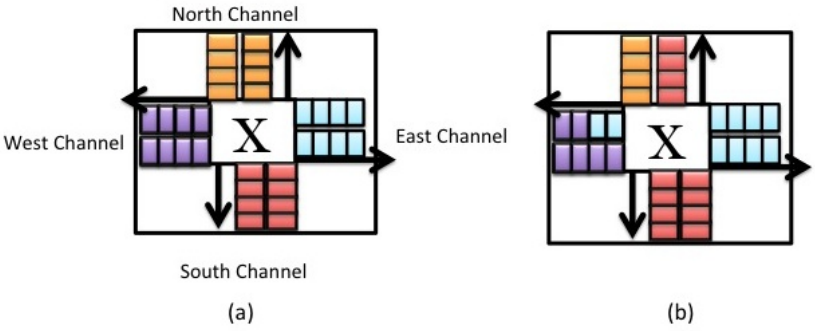


Figura 4.5: (a) Roteador base construído com dois canais virtuais com espaço para 4 (quatro) *flits*; (b) Roteador Flexível com os *buffers* reconfigurados para atender a demanda do tráfego.

O *tail flit* se torna especialmente importante neste cenário porque é através dele que o roteador se informa sobre o término do pacote.

4.4 Deadlock

Embora do ponto de vista arquitetural não existam restrições no mecanismo, a alocação de canais virtuais de maneira irrestrita cria uma dependência de recursos. Esta dependência de recursos, quando não tratada, pode resultar em situações de *deadlock*.

Um exemplo é mostrado na Figura 4.6. Suponha dois roteadores de uma Mesh 2D unidirecional interligados. Suponha agora que todos os pacotes do Roteador 1 tenham como destino o Roteador 2, e desejam utilizar o Canal C(1,2) que interliga a porta Leste do Roteador 1 à porta Oeste do Roteador 2. Ao mesmo tempo, todos os pacotes do Roteador 2 tenham como destino o Roteador 1 e desejam utilizar o canal C(2,1) que interliga a porta Oeste do Roteador 2 à porta Leste do Roteador 1. Os canais C(1,2) e C(2,1) se tornam bloqueados devido a não-existência de recursos (*buffers* disponíveis nos roteadores 1 e 2. Pode-se dizer que o Roteador 1 depende da liberação de recursos do Roteador 2 para ser desbloqueado, e que simultaneamente o Roteador 2 depende da liberação de recursos do Roteador 1 para ser desbloqueado. Forma-se então uma dependência cíclica que configura uma situação de *deadlock*.

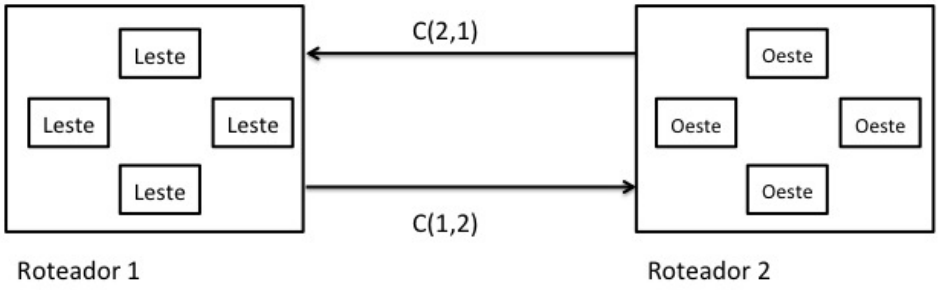


Figura 4.6: Dois roteadores bloqueados devido a dependência cíclica causando *deadlocks*.

Deadlocks são catastróficos para uma rede [7]. Após alguns recursos serem ocupados por pacotes em *deadlock*, outros pacotes dependentes deste recurso também se tornam bloqueados. Dentro de alguns ciclos toda a rede pode ficar paralisada, o que em uma rede de interconexão intra-chip significaria o congelamento do sistema.

Soluções clássicas para problemas de *deadlock* são variados. Existem técnicas que focam em sua prevenção, enquanto outras assumem que estes são eventos raros e de tempos em tempos executam mecanismos para descobrir se há *deadlock* na rede, caso suspeitem da existencia do mesmo [49].

Prevenção de *deadlocks* é baseada em aplicar restrições no *design* do roteador de forma a eliminar uma ou mais condições para que ocorra *deadlock*.

A detecção de *deadlocks* se baseia num pensamento mais simplista, não é aplicado nenhuma restrição ao roteador, ele simplesmente deixa que *deadlocks* ocorram ocasionalmente. Um mecanismo de detecção é aplicado de tempos em tempos quando há a suspeita de *deadlock*, e sendo confirmada a existência do mesmo, um ou mais recursos que estão bloqueados são liberados.

Embora a segunda solução seja menos restritiva, mecanismos de detecção de *deadlocks* podem causar um *overhead* de trocas de mensagens na rede, além de ter-se a necessidade de que sejam implementados canais dedicados para que a comunicação de detecção seja possível, visto a impossibilidade de utilizar-se canais normais, pois os mesmos se encontram bloqueados. Outro fator que torna a detecção de *deadlocks* menos atrativa para NoCs é o fato de que a solução para liberar recursos normalmente se baseia no descarte de pacotes que estão em roteadores bloqueados, algo que nem sempre é possível, ou bastante complexo de ser feito em NoCs.

Prevenir *deadlocks* é uma solução mais restritiva em termos de utilização de recursos, porém a garantia da não-ocorrência de *deadlocks* elimina a necessidade de *hardware* especializado de detecção, tornando assim o gerenciamento dinâmico do circuito mais simples. Além de evitar o desperdício de processamento que o sistema necessitaria para detectar e recuperar-se de *deadlocks*.

Este trabalho se baseia no *turn-model* para prevenir *deadlocks* e utiliza as mesmas restrições aplicadas em [27] para tratar o problema.

4.4.1 Turn-Model

Prevenir *deadlocks* requer que sejam eliminadas as dependências no grafo de alocação de recursos. Isto pode ser feito adicionando-se ordem parcial na alocação dos mesmos. *Deadlocks* ocorrem quando pelo menos um agente está prendendo um recurso de prioridade alta e está aguardando pela liberação de um outro recurso com prioridade mais baixa. Por sua vez quem está prendendo este recurso de prioridade mais baixa está aguardando a liberação do recurso de prioridade mais alta, configura-se então um cenário em que existe um ciclo de dependências, causando assim o *deadlock*. Quando os recursos são ordenados, este tipo de situação será prevenida. Um agente que detém um recurso de alta prioridade, só irá fazer pedidos de recursos com prioridades maiores eliminando assim qualquer possibilidade de formação de ciclos.

O *turn-model* [51], como pode ser visto na Figura 4.7, separa as oito curvas que um pacote pode fazer (-x para +y (1), -y para -x (2), +x para -y (3), +y para +x (4), -y para +x (5), +x para +y (6), +y para -x (7) e -x para -y(8)) em dois ciclos abstratos. Quando se elimina uma curva em cada um destes dois ciclos, ciclos são eliminados e por consequência, esta rede se torna livre de *deadlocks*.

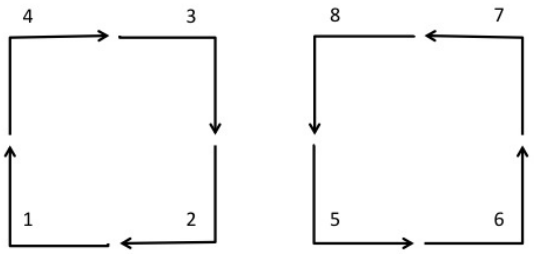


Figura 4.7: Oito curvas num roteador abstrato. Caso todas as curvas sejam permitidas, a rede poderá sofrer deadlock.

A aplicação do *turn-model* no roteador flexível foi feita através de aplicações de restrições na alocação de *buffers* para um pacote. É levado em consideração a porta de saída que o pacote irá se destinar no roteador. Através desta informação restringe-se qual canal virtual ádvena o pacote pode utilizar.

As restrições aplicadas no roteador foram as mesmas utilizadas pelo algoritmo de roteamento *Dimension-Order XY* [27]. Como pode ser visto na Figura 4.8, foram removidos dois turnos em cada um dos ciclos abstratos (num total de quatro), os que vão da direção y para x. Desta forma, pacotes oriundos da direção x (positiva ou negativa) e que estejam indo para a direção y, podem alocar *buffers* pertencentes a qualquer uma das portas, porém, pacotes oriundos da direção y (positiva ou

negativa) são proibidos de fazerem a alocação de *buffers* que pertencem a portas da direção x (leste e oeste). Após a aplicação destas restrições, o roteador flexível fica livre de *deadlocks*.

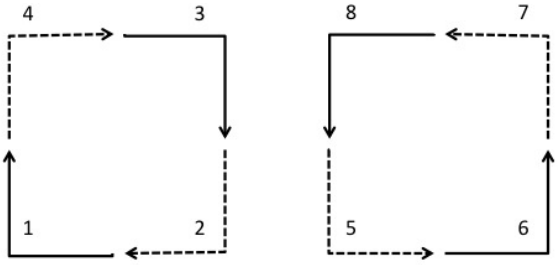


Figura 4.8: Remoção de dois turnos abstratos em cada ciclo.

4.5 Método de Alocação de Canais Virtuais

Tendo em vista a prevenção de *deadlocks* mencionadas na seção anterior, torna-se necessário a criação de um método de alocações de canais virtuais que obedeça as restrições necessárias para que o roteador seja livre de *deadlocks*.

O mecanismo proposto consiste em pegar emprestado canais virtuais de portas de entrada diferentes de acordo com a disponibilidade dos *buffers*. Se um pacote deseja ser transmitido e a porta de entrada do *downstream* não contêm canais virtuais disponíveis, o FFC irá buscar na tabela por um canal virtual livre em outra porta e emitir um pedido de canal virtual para um FFC da porta selecionada. A porta selecionada então responde caso tenha a disponibilidade de alocar o canal virtual selecionado para a outra porta. Mesmo que um canal virtual esteja disponível, não é garantido que ele será reservado neste ciclo mediante pedido. Isto se deve ao fato de que uma porta de entrada pode receber mais de um pedido de alocação de canal virtual simultaneamente, neste caso, é utilizado um arbitrador *round-robin* para escolher uma porta vencedora.

4.6 Resumo do Capítulo

Neste capítulo foi definido o que são roteadores flexíveis, em que ponto a pesquisa está atualmente e o que foi implementado de diferente nesta dissertação. Foram introduzidos detalhadamente a arquitetura tradicional do roteador flexível, assim como as alterações feitas neste trabalho e também como foram alterados os mecanismos do roteador base para construir o roteador flexível. O Controlador de Flexibilidade do FIFO (CFF) foi introduzido e seu funcionamento e arquitetura detalhados. Foram analisados os problemas de dependência que podem ocorrer caso a

alocação de *buffers* se torne irrestrita no roteador flexível, assim como a solução para este tipo de problema através de mecanismos de prevenção de *deadlock* utilizando-se do *turn-model*.

Capítulo 5

RAF: Roteador Assíncrono Flexível

O mecanismo de alocação de *buffers* diferenciado apresentado no capítulo anterior foi projetado com o objetivo de aumentar o desempenho em termos de *throughput* e latência. No entanto, problemas relacionados ao consumo de energia ainda persistem. Torna-se então necessário a criação de versões *low-power* dos roteadores tradicionais.

É proposto então o Roteador Assíncrono Flexível (RAF), uma versão assíncrona do Roteador Flexível visto no Capítulo 4. Este roteador tem o objetivo de ser uma alternativa de baixo custo energético ao roteador tradicional.

Para que fosse analisado os benefícios proporcionados pela arquitetura assíncrona de roteadores em NoCs, foram implementadas em hardware versões assíncronas dos roteadores base, visto no Capítulo 2, e flexível, visto no Capítulo 4. Para isto, foi necessário também a implementação das versões síncronas dos mesmos na linguagem de descrição de hardware *Verilog* explicada na Seção 3.1. Foi então aplicada a metodologia de conversão síncrono-assíncrono baseado no ASERT, explicada em detalhes na Seção 3.5. Este capítulo mostra de que forma foi organizada a arquitetura dos roteadores implementados em hardware, assim como a forma que foi utilizada a metodologia ASERT para que fosse produzido a versão assíncrona dos mesmos.

A Seção 5.1 explica como foram organizados os blocos funcionais das arquiteturas dos roteadores base e flexível em hardware. A Seção 5.2 explica de que forma foi aplicada a metodologia ASERT para converter os roteadores.

5.1 Arquitetura do Roteador em Hardware

O estudo da versão assíncrona do Roteador Flexível requer a elaboração do mesmo em uma linguagem de descrição de hardware, e devido a complexidade necessária

para a implementação de certos mecanismos em hardware, como canais virtuais e controle de fluxo *wormhole*, a versão em HDL dos roteadores tiveram que sofrer alterações no design: Ao invés de vários canais virtuais, cada roteador possui apenas um *buffer* por porta de entrada, além de implementarem o mecanismo de controle de fluxo *Store-and-Forward*. Embora estas versões sejam mais simples, quando comparados com as versões explicadas em outros capítulos, a versão apresentada neste capitulo, nos dá base suficiente para que se tenha ideia de quais são os benefícios e custos inerentes da utilização de um roteador assíncrono. Para as versões síncronas, utilizou-se o roteador flexível desenvolvido em [29]. O roteador foi construído utilizando a linguagem de descrição de *hardware* Verilog. Este roteador pode ser dividido em duas partes principais: A porta de entrada e a porta de saída. Os módulos que são encapsulados em cada uma destas partes são listados abaixo:

- Porta de Entrada: Este módulo compõe os módulos que fazem o processamento do pacote antes de o encaminhar para uma determinada porta de saída. Este módulo contém o controlador do FIFO, responsável por coordenar a escrita e leitura nos *buffers* do roteador, assim como a fazer a sinalização da capacidade disponível do *buffer*. Também está contido neste módulo a unidade de roteamento, assim como a de alocação de *buffer* no *downstream*. No caso do roteador flexível, o CFF também está contido nesta unidade e é utilizado em conjunto com o controlador do FIFO para alocar os *buffers*.

- Porta de saída: Este módulo compõe os módulos que fazem o encaminhamento dos pacotes que estão nas portas de entrada para os roteadores vizinhos ou para a porta local, caso este seja o último *hop* do pacote, além de co-ordenar o acesso as portas (estágio SA do pipeline do roteador). Nele estão contidos o arbitrador do *switch* para garantir ordem no acesso do *crossbar* e o controlador da porta de saída, que seleciona e encaminha os pacotes do *crossbar* para o próximo roteador. Este módulo também é responsável por receber, processar e informar a porta de entrada sobre a disponibilidade dos *buffers* nos roteadores vizinhos.

As figuras 5.1 e 5.2 ilustram de que forma foram organizados os módulos para a construção dos roteadores base e flexível.

Como pode ser visto nas Figuras 5.1 e 5.2, como o roteador é construído para operar em NoCs 2D, ele possui 5 portas de entrada e 5 portas de saída (Norte (N), Sul (S), Leste (E), Oeste (W) e Local(L)). E cada porta de entrada possui uma ligação com um multiplexador de cada porta de saída. A partir de agora, as figuras só ilustrarão a ligação entre uma porta de entrada e uma porta de saída para fins de simplificação do entendimento da arquitetura dos roteadores.

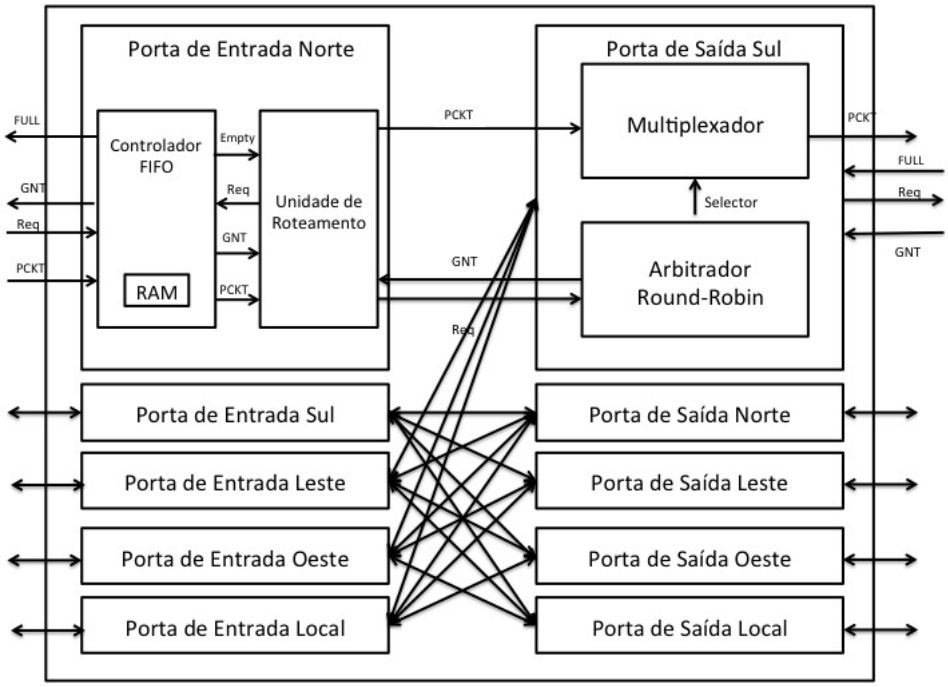


Figura 5.1: Módulos do Roteador Base.

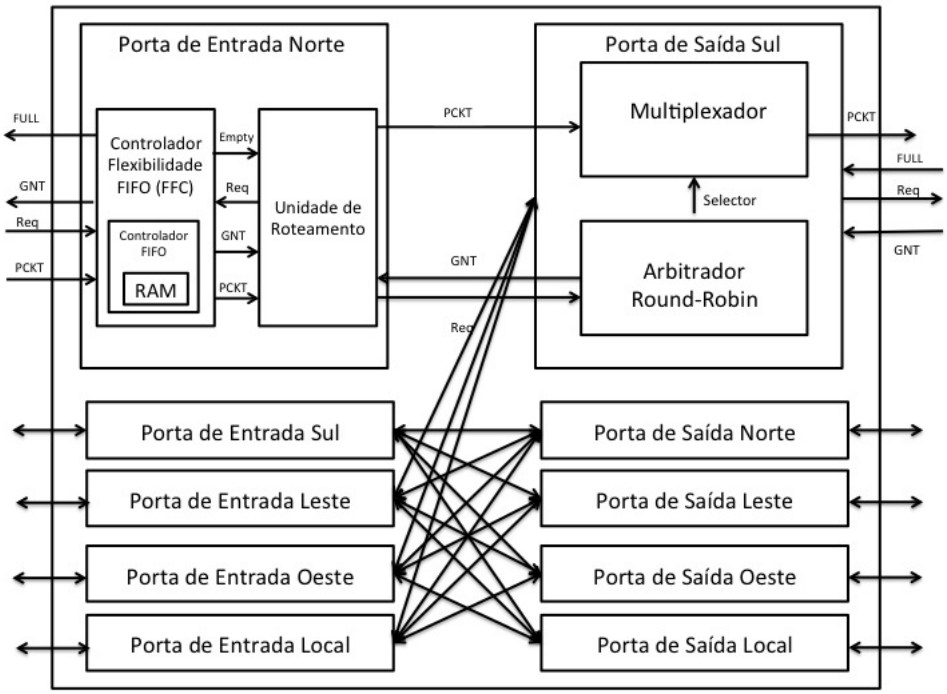


Figura 5.2: Módulos do Roteador Flexível. No roteador flexível, o controlador do FIFO fica encapsulado dentro do Controlador de Flexibilidade do FIFO (CFF).

5.1.1 Operação dos Roteadores Base e Flexível em Hardware

Diferentemente da versão apresentada no capítulo anterior, a modelagem dos roteadores na versão em hardware teve quer ser feita com detalhes a nível de sinal. Para fazer a comunicação entre roteadores, foi escolhido o protocolo de comunicação de *(Request/Grant)*.

Quando algum nó vizinho deseja enviar uma mensagem para o roteador, este irá enviar um pedido através do sinal *Request* (Req) de sua porta de saída e aguardar o roteador *downstream* liberar o envio do pacote através de seu sinal de *Grant* (Gnt). Caso o sinal seja 1, significa que o pacote pode ser transferido normalmente. O roteador emissor irá enviar seu sinal de *request* continuamente até que o roteador receptor lhe responda com o *grant*.

Após a recepção do pacote no roteador *downstream* através da porta de entrada, esta irá encaminhar, através do controlador da *FIFO*, o pacote para o seu respectivo *buffer* para que este seja armazenado para a próxima etapa.

A Unidade de Roteamento, a cada ciclo, se estiver ociosa, verifica se existe um pacote armazenado no FIFO. Caso exista, este o lê e determina a porta de saída pela qual o pacote deverá ser encaminhado. Após a determinação da rota, este irá enviar um *request* para o arbitrador da porta de saída determinada e irá competir com as outras portas pelo uso do *crossbar*.

A cada ciclo, o arbitrador seleciona uma porta de entrada, dentre as solicitantes, para emitir o *grant* de utilização da porta de saída no próximo ciclo.

No próximo ciclo, os multiplexadores das portas de saída, irão transferir os pacotes do fio do *crossbar*, selecionado pelo arbitrador, para no registrador da porta de saída. No ciclo seguinte, esta porta então irá enviar um *request* para o roteador vizinho e reiniciar todo ciclo.

Assim como o roteador explicado no Capítulo 4, a diferença principal entre a versão do roteador base e flexível é a adição do Controlador de Flexibilidade do *FIFO*. A diferença em hardware é que o sinal de *Grant* emitido pelo roteador receptor irá ser emitido caso exista espaço em algum *buffer* de outra porta de entrada, respeitando as limitações explicadas na Seção 4.4. Outra diferença em *hardware* é o controle de fluxo empregado, que em software foi *wormhole* e em *hardware* foi *Store-and-Forward*.

5.2 Conversão síncrona-assíncrona do roteador

O processo de conversão de roteadores síncrono-assíncrono utilizado foi o desenvolvido em [48] e explicado na Seção 3.5. Embora no trabalho de [48] tenha sido

proposto uma forma de conversão automática, em vista da indisponibilidade da obtenção das ferramentas necessárias para a conversão automatizada (o sistema construído em [48]), a conversão feita neste trabalho foi feito de forma manual, respeitando todas as regras impostas pela metodologia.

5.2.1 Aplicação da metodologia

A aplicação da metodologia começa com a análise do código de descrição do circuito original síncrono em *verilog*, a fim de identificar as entradas e saídas primárias. As Figuras 5.3 e 5.4 mostram os pontos em que foram identificados a necessidade de um controlador de nó nos circuitos de ambos os roteadores. Os locais mostrados nas figuras para a adição do sinal dos controladores de nós foram escolhidos porque são os pontos de entrada do sinal de *clock* original. Vale a pena lembrar que a metodologia ASERT é não-invasiva, ou seja, o circuito original não foi alterado, e embora outras técnicas invasivas poderiam ter sido utilizadas, para este trabalho foi escolhido o ASERT por ser uma metodologia simples de ser aplicada.

Na Figura 5.4, o circuito do roteador flexível necessita de um controlador de nó extra para controlar a unidade de flexibilidade do *FIFO*. O impacto da adição deste controlador extra, assim como da adição do módulo de sinalização ASERT, serão analisados no Capítulo 6.

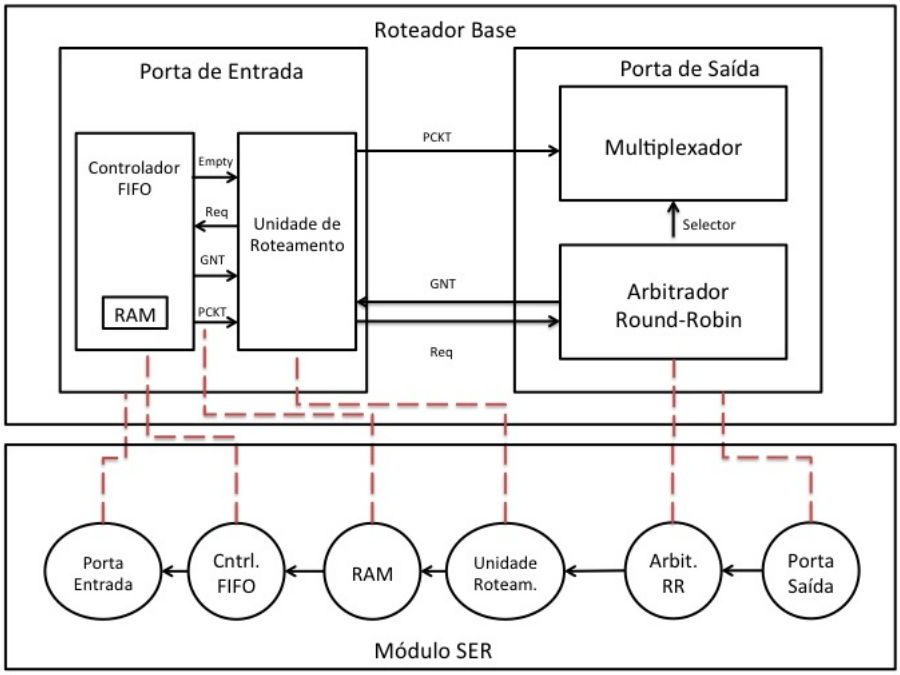


Figura 5.3: Modelagem do local aonde serão inseridos os controladores de nó ASERT no roteador base.

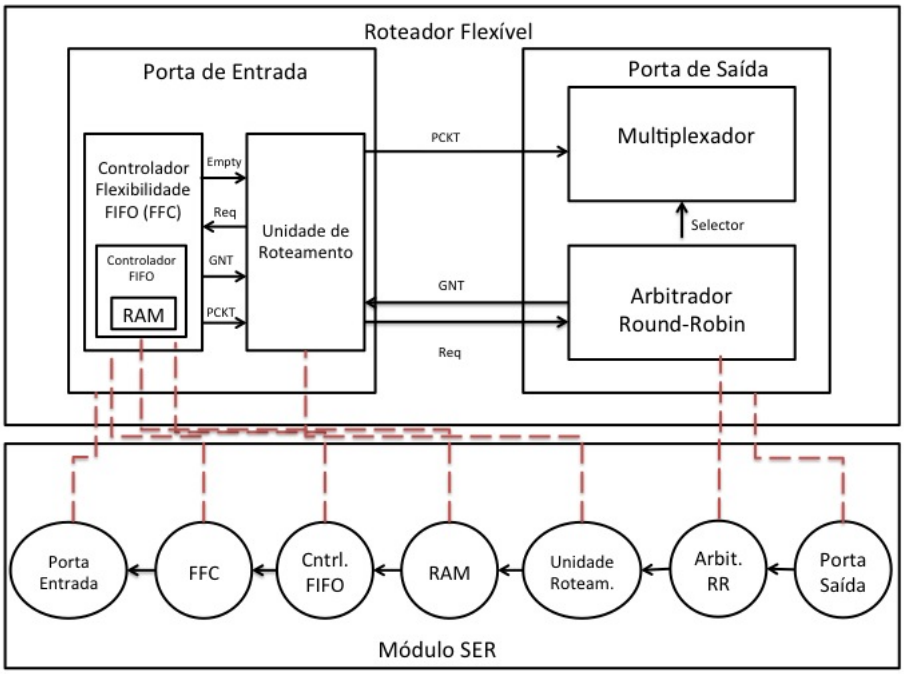


Figura 5.4: Modelagem do local aonde serão inseridos os controladores de nó ASERT no roteador flexível.

5.2.2 Sinalização de EOP

A viabilidade do emprego de qualquer método de sinalização assíncrona se baseia substancialmente na disponibilidade da comunicação do término de operação das unidades funcionais a serem sincronizadas. Como todo sistema GALS, a metodologia necessita de um método de sincronização local dentro das ilhas. A metodologia aplicada propõe uma forma de sinalização denominada *diet clock*.

O *diet clock* é um procedimento alternativo de geração do sinal de fim de operação aonde é empregado um sinal temporal de referência, o *clock*, aplicado a contadores de valores fixos que produzem intervalos pré-programados correspondentes aos tempos de duração de cada nó de interconexão estimados no momento de design dos roteadores. Cada contador associado a um controlador de nó possui um sinal de habilitação da contagem diretamente conectado ao sinal de início de operação do nó. Dessa forma, o contador se encontra em estado estacionário até que seja requisitado o começo da temporização. Ele foi escolhido por ser um *clock* mais simples, ele é chamado *diet* por ser aplicado nos contadores e operar a uma frequência mais rápida.

O conjunto de portas lógicas, que constituem os circuitos contadores responsáveis pela geração dos sinais de fim de operação, deve ser contido em uma região física devidamente concentrada.

A técnica do *diet clock* foi baseada em um incrementador síncrono de 3 bits com

período de operação de 1 ns. Os contadores foram configurados para enviar um sinal de troca de sinal a cada período de tempo, sendo este período de tempo o necessário para a execução do controlador de nó acoplado no contador. O *diet clock* é uma técnica considerada boa porque é simples de ser implementada e atende todos os requisitos para possibilitar a avaliação do método. Porém, caso a implementação seja feita em um circuito real, ele não seria utilizado.

5.2.3 Construção dos Roteadores Assíncronos

Com os tempos de execução de cada módulo calculados, através da obtenção do tempo do caminho crítico dos módulos, e os contadores calibrados, admitindo-se que cada contador demora 1 ciclo para exibir novos valores, gera-se então os elementos de sinalização necessários para a metodologia e, envolve-se então os mesmos em um módulo topo pronto para ser utilizado.

Aplicando-se a técnica ASERT, gera-se um novo módulo com novos sinais de entrada e saída substituirão o sinal de *clock* original do circuito. Cada conexão entre dois módulos resulta na criação de dois controladores de arestas: um que conecta o nó fonte ao nó intermediário e um que conecta o nó intermediário ao nó destino. Como explicado no Capítulo 3, os nós intermediários são necessários para que não seja criada uma falsa dependência entre dois módulos que podem operar em paralelo.

O próximo passo é a junção deste módulo *ASERT* ao módulo dos roteadores originais. São criados os fios necessários para interconectar os dois módulos e ambos são englobados em um módulo único. As Figuras 5.5 e 5.6 mostram o circuito final dos roteadores assíncronos base e flexível, respectivamente. É importante frisar que os nós intermediários não são mostrados nas figuras por questões de simplicidade. Como explicado anteriormente, para cada controlador de aresta, existe um nó intermediário.

5.3 Resumo do Capítulo

Neste capítulo foi apresentado o Roteador Assíncrono Flexível. Introduziu-se a metodologia ASERT e mostrou-se de que maneira a mesma foi utilizada para realizar-se a conversão síncrono-assíncrono dos roteadores. Foi apresentado de que forma estes roteadores são organizados em *hardware* como módulos, além de ser mostrado que partes destes roteadores precisaram ser modificadas para a conversão. Foi apresentado também a maneira pela qual as informações de *timming* dos circuitos foram obtidas e utilizadas com o *diet clock*.

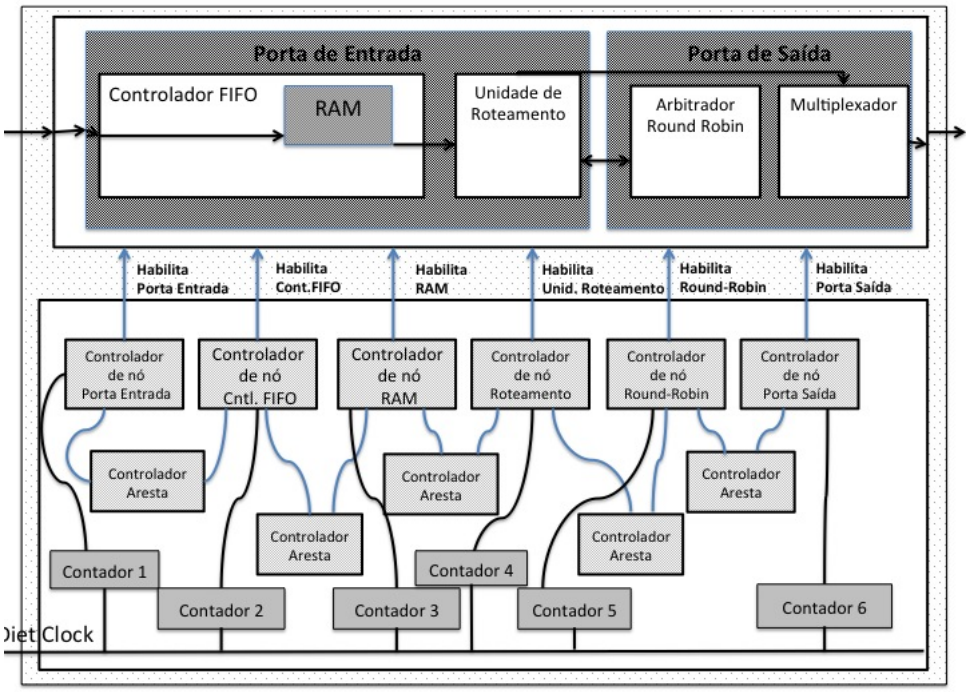


Figura 5.5: Circuito final do roteador assíncrono base utilizando o diet clock.

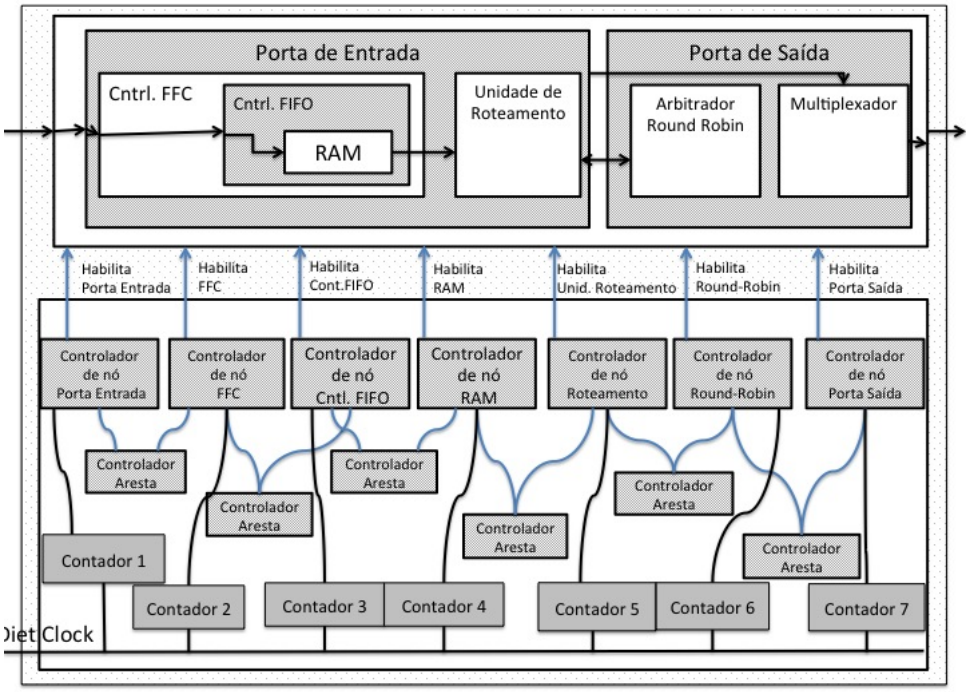


Figura 5.6: Circuito final do roteador assíncrono flexível utilizando o diet clock.

Capítulo 6

Experimentos e Resultados

Para avaliar o roteador flexível são feitos dois conjuntos de experimentos. O primeiro conjunto visa avaliar o desempenho do roteador flexível em relação a *throughput* e latência. São construídas redes formadas apenas por roteadores flexíveis e compara-se com redes de configuração idênticas construídas a partir de roteadores base. O segundo conjunto tem o objetivo de avaliar a diferença entre área e potência dissipada dos dois roteadores. Além de serem implementadas versões assíncronas utilizando o método de conversão ASERT de cada roteador para que seja feita a comparação com as versões síncronas.

Nesse capítulo são apresentadas as configurações das redes e roteadores, os procedimentos experimentais adotados, e os resultados obtidos.

6.1 Avaliação do desempenho do Roteador Flexível em relação à rede

Para avaliar o desempenho do roteador flexível, foram realizados experimentos na forma de simulação utilizando o simulador com precisão a nível de ciclo, TOPAZ [31]. TOPAZ é um simulador para redes de interconexão, implementado em C++, modela os *pipelines* do roteador e opera na granularidade de componentes arquiteturais independentes. O simulador foi estendido para implementar a arquitetura do roteador flexível. Configurou-se 3 redes NoC onde foram substituídos os roteadores pelas arquiteturas base e flexível. Para cada rede são variados também os parâmetros de configuração dos roteadores (número de VCs, número de *buffers* e tamanhos de pacotes em número de *flits*) a fim de avaliar as diferenças de desempenho de ambos.

6.1.1 Parâmetros Arquiteturais

Para verificar quais parâmetros arquiteturais têm mais impacto no desempenho do roteador flexível foram variados o tamanho da rede, o tamanho dos pacotes, o tamanho dos *buffer* assim como a quantidade de canais virtuais e a frequência que pacotes são injetados na rede. A importância de cada um destes parâmetros é explicada a seguir:

O tamanho da rede influencia na quantidade de nós que um pacote necessita atravessar para chegar ao seu destino. Quanto maior for o diâmetro da rede, maior é a quantidade média de *hops* que um pacote necessitará para chegar ao seu destino, aumentando assim a possibilidade deste pacote ficar bloqueado a espera de recursos. O tamanho da rede também influencia na largura de banda média no núcleo da rede, quanto maior for a rede, maior será sua largura de banda.

O tamanho do pacote influencia na utilização dos *buffers* de um roteador. Conforme o tamanho dos pacotes aumenta, é reduzida a quantidade de pacotes que podem ser armazenados em um único roteador, podendo existir casos em que apenas um *buffer* não é suficiente para armazenar o pacote inteiro, fazendo assim com que um pacote seja distribuído por vários roteadores ao longo do caminho de rota do pacote.

O tamanho do *buffer* influencia na quantidade de dados que um roteador será capaz de armazenar antes de necessitar bloquear pacotes. Como mencionado anteriormente, seu tamanho também tem influência sobre a quantidade de potência dissipada no roteador.

A quantidade de canais virtuais influencia no fracionamento do canal. Conforme aumenta-se o número de canais virtuais, mais fracionado fica o canal físico, porém este fracionamento se torna útil em casos onde existiria o bloqueio do pacote na cabeça da fila *(Head-of-Line blocking - HoL blocking)*. Quanto mais canais virtuais forem adicionados, menor será a probabilidade deste evento ocorrer.

A frequência com que cada pacote é injetado na rede influencia diretamente no congestionamento da rede. Se os pacotes são injetados muito rapidamente, os *buffers* dos roteadores tendem a encher na mesma frequência, podendo ocasionar bloqueios mais frequentemente. Caso os pacotes sejam injetados com uma frequência menor, a probabilidade de um pacote ficar bloqueado diminui.

A Tabela 6.1 disponibiliza a variação de cada um destes parâmetros utilizados nas simulações.

Tabela 6.1: Variação dos parâmetros de Simulação.

Parâmetro	**Valores**
Tamanho da Rede	4x4, 6x6 e 8x8
Tamanho do Pacote	4, 8, 12 e 16 flits
Tamanho do Buffer	4, 8 e 16 flits
# de Canais Virtuais	2 e 4
Variação de Frequência de Injeção	0.3 à 0.9 pacotes por ciclo

É importante ressaltar que todas as redes utilizadas são construídas utilizando-se a topologia *mesh* [7], aonde os nós são encapsulados em um grid n-dimensional (sendo que nesta dissertação, n = 2). O algoritmo de roteamento utilizado é o DOR-XY 2.3.3, onde os pacotes são primeiramente roteados na dimensão X e só então na dimensão Y. Este algoritmo foi utilizado devido ao fato do foco principal deste trabalho é elucidar as melhorias de performance providas pelo roteador flexível, e algoritmos de roteamento mais complexos elevam o tempo de voo (*flight time*) do pacote no roteador, embaçando esse efeito. Ou seja, utilizando DOR-XY torna mais propício a tarefa de avaliar o maior nível de injeção e pressão sobre os *buffers*. O controle de fluxo *wormhole* é utilizado para controlar os pacotes ao longo de sua travessia pela rede. O padrão de tráfego utilizado foi o Uniforme Aleatório, onde cada nó injeta pacotes na rede em intervalos regulares, e a probabilidade de um nó ser origem ou destino é igual para todos os nós da rede.

6.1.2 Procedimentos Experimentais

Para cada tamanho de rede, foi executada uma simulação para que fosse gerado um arquivo contendo as informações de cada roteador fonte e cada roteador destino para cada pacote, assim como seu tempo de inserção e ejeção na rede. Então é feita uma nova simulação para cada arquitetura de roteador utilizando como *trace* o arquivo contendo os tempos e roteadores fonte e destino gravados anteriormente. Para cada rede são inseridos 50.000 pacotes variando a frequência de injeção de pacotes, em passos de 0.03 flits/ciclo, conforme mencionada na Tabela 6.1. Para cada execução, o tamanho do pacote e do *buffer* de cada roteador é fixo. A variação destes parâmetros é feita entre uma execução e outra. A rede é construída de forma homogênea, ou seja, todos os roteadores são idênticos.

Para rodar os experimentos foi usada uma estações de trabalho *DELL PowerEdge 2900* com um processador *Xeon* de 4 núcleos de processamento, cada um

com *simultaneous multi-threading* (e portanto 2 núcleos lógicos em cada núcleo físico). A máquina possui 2GB de memória RAM. O sistema operacional utilizado foi GNU/Linux (*CentOS 6.5 64 bits*).

6.1.3 Resultados

Devido a quantidade de resultados a serem analisados, nesta seção é mostrado apenas os resultados para o tamanho de rede 8x8. Esta rede foi escolhida porque os resultados em redes menores, como 4x4 e 6x6, não possuem diferenças de desempenho tão expressivas como as da rede 8x8. Conforme aumenta-se o tamanho da rede, de 4x4 para 8x8, a quantidade de roteadores que um pacote precisa atravessar aumenta, assim é aumentada a probabilidade de ocorrer congestionamento devido a concorrência de dois ou mais pacotes pelo mesmo recurso. O congestionamento em redes menores é menor, visto que, devido ao diâmetro reduzido da mesma, cada pacote precisa atravessar em média menos roteadores para chegar ao seu destino.Os gráficos são mostrados para diferentes configurações de tamanho de pacote. Cada linha representa o tamanho de cada pacote. Por exemplo Flex 4 significa "roteador flexível com tamanho de pacote 4 (quatro) *flits*"e Base 8 significa "roteador base com tamanho de pacote 8 (oito) *flits*". Os resultados das simulações com os tamanhos de rede não mostrados nesta seção podem ser encontrados no Apêndice A.

Começamos assumindo que os roteadores possuem a mesma quantidade de canais virtuais. A partir das Figuras 6.1 à 6.5 pode-se verificar que a diferença em *throughput* cresce à medida que a taxa de injeção aumenta. Isto se deve ao fato de que quando a taxa de injeção é baixa, não existe muito congestionamento, o que faz com que o desempenho dos roteadores sejam equivalentes. A medida que mais pacotes são injetados na rede, a demanda por recursos é aumentada. Com isto, o roteador que possuir a melhor forma de atender a esta demanda por recursos proverá o melhor desempenho. Nos experimentos com dois canais virtuais, o roteador flexível tem *throughput* até 21% maior (4 *flits* por *buffer* e pacotes com 4 *flits*, Figura 6.1) quando comparado com o roteador base. Nos casos em que o tamanho do *buffer* é maior, esta diferença em desempenho é reduzida, ainda assim o roteador flexível tem 9% (8 flits por *buffer* e pacotes com 8 *flits*, Figura 6.3) e 11% (16 *flits* por *buffer* e pacotes com 16 *flits*, Figura 6.5) de melhora de desempenho quando comparado ao roteador base.

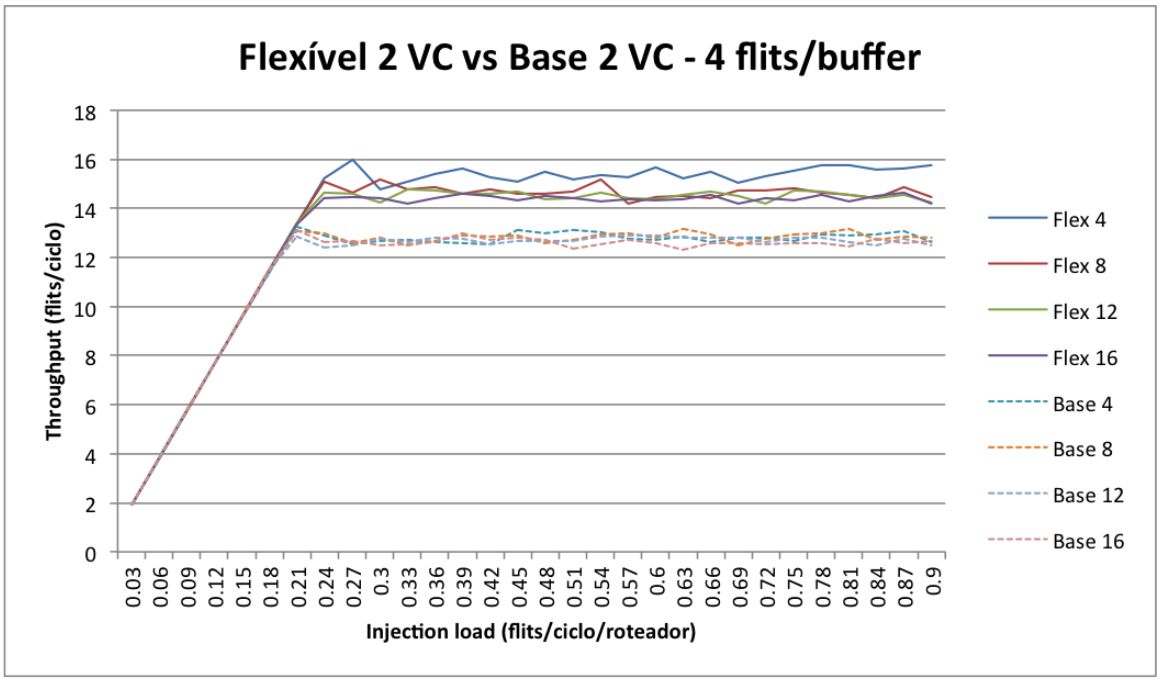


Figura 6.1: Throughput - 2 Canais Virtuais e 4 flits por buffer.

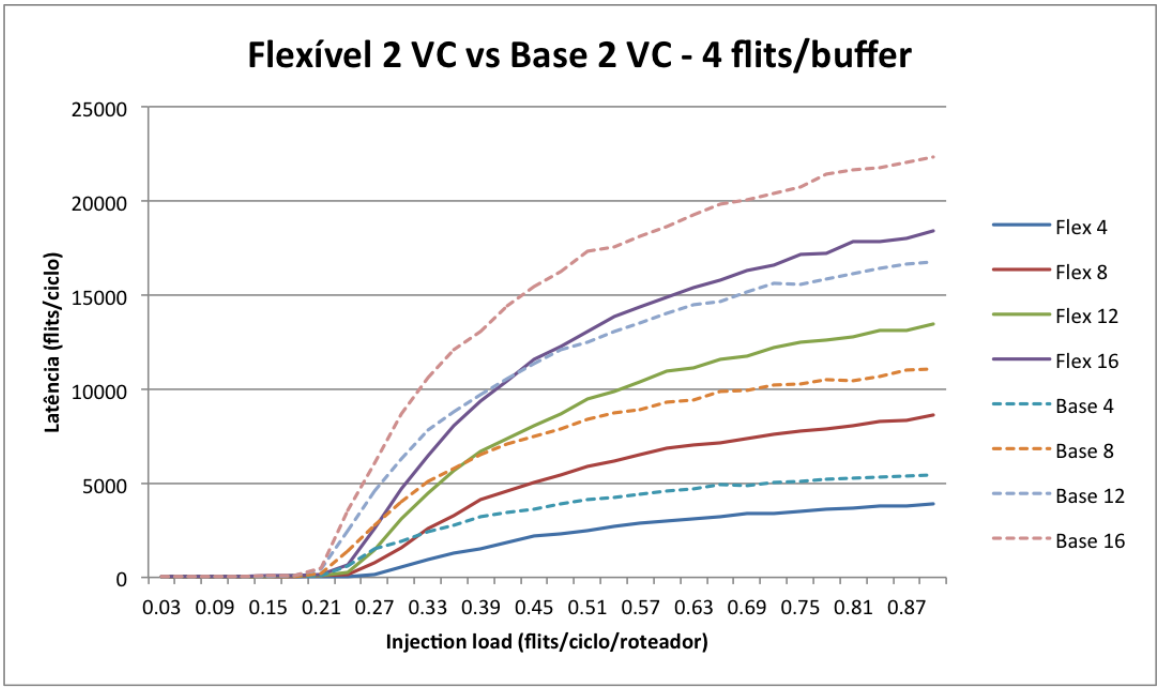


Figura 6.2: Latência - 2 Canais Virtuais e 4 flits por buffer.

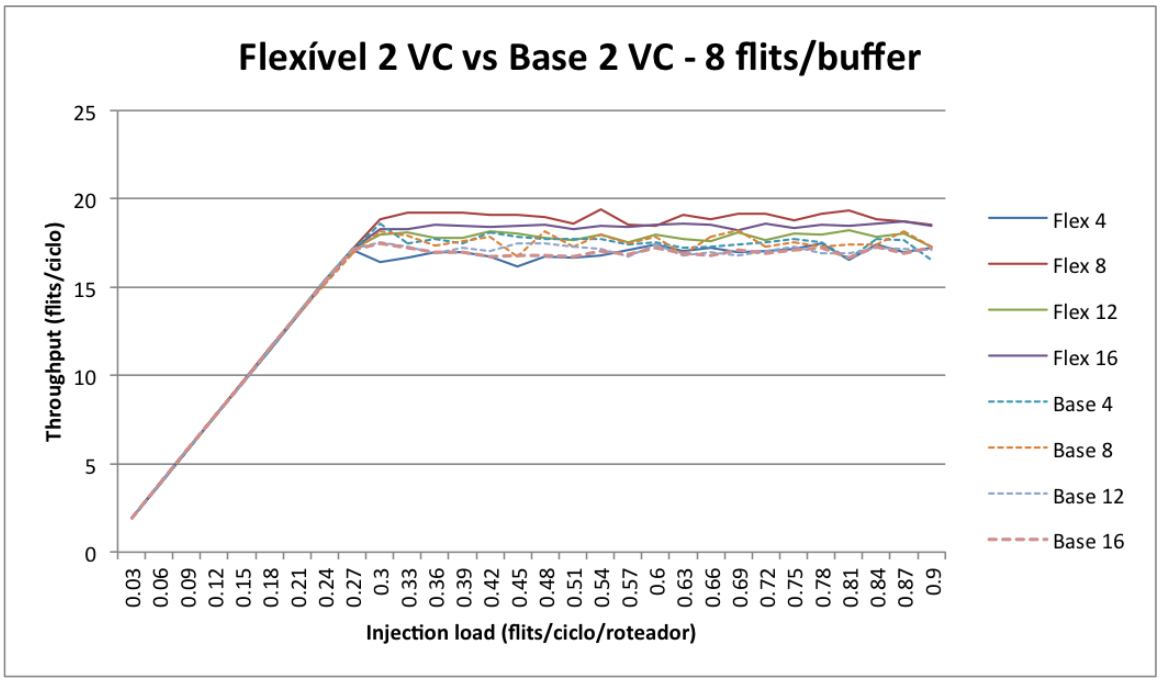


Figura 6.3: Throughput - 2 Canais Virtuais e 8 flits por buffer.

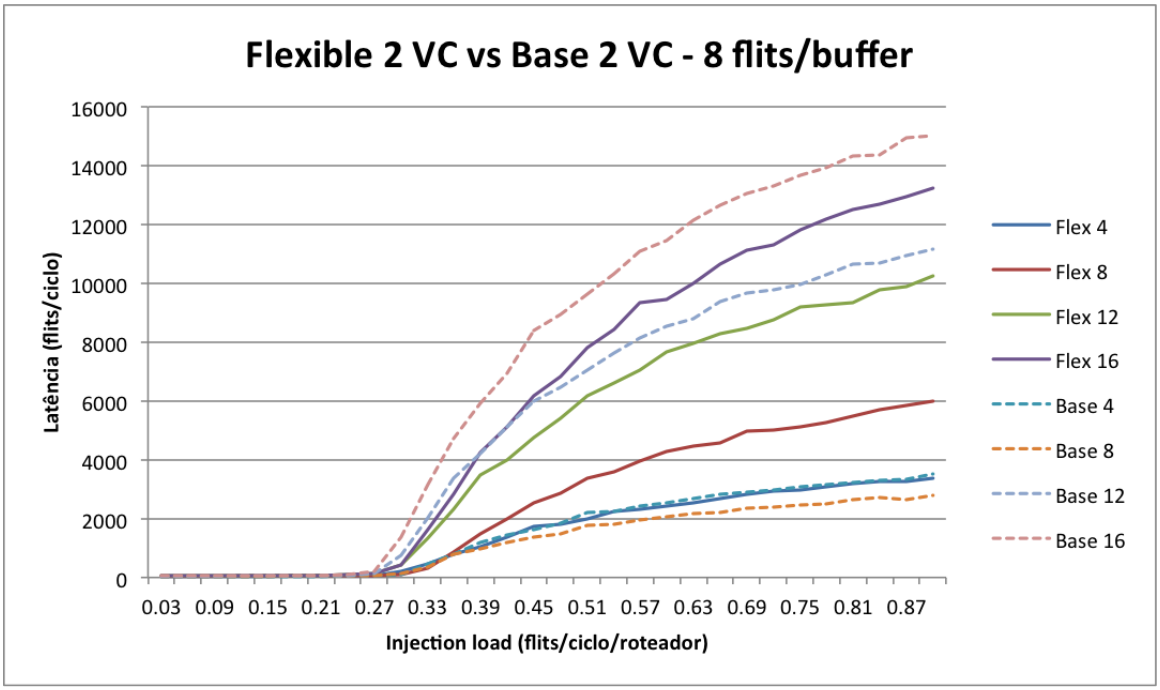


Figura 6.4: Latência - 2 Canais Virtuais e 8 flits por buffer.

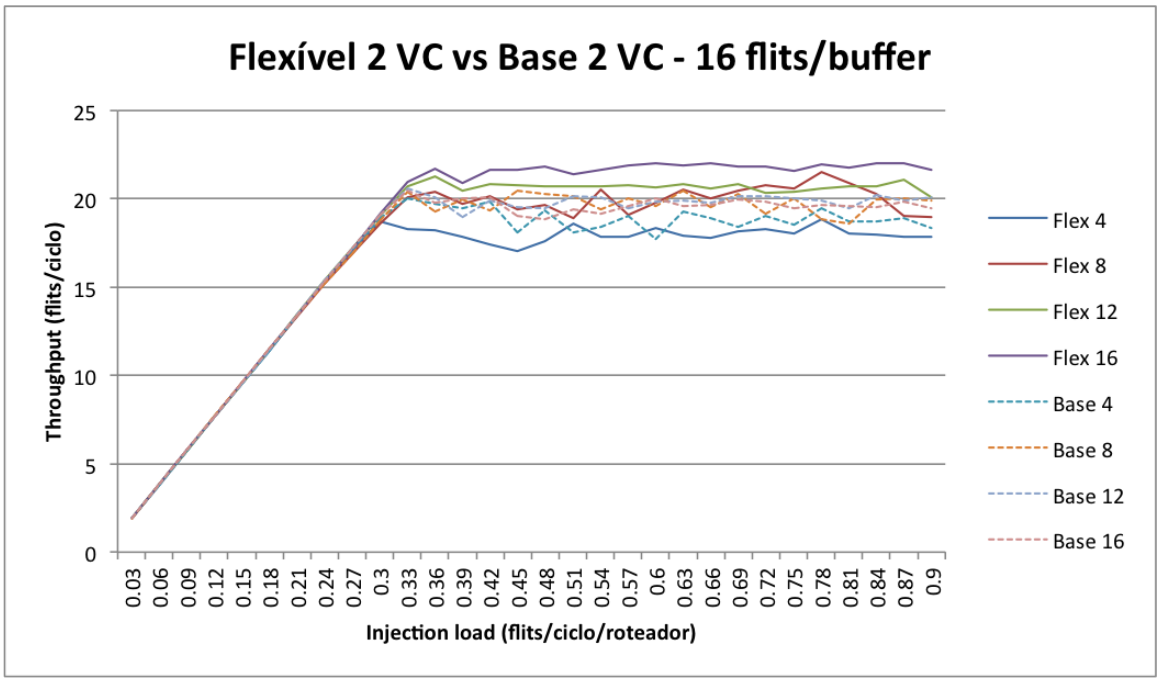


Figura 6.5: Throughput - 2 Canais Virtuais e 16 flits por buffer.

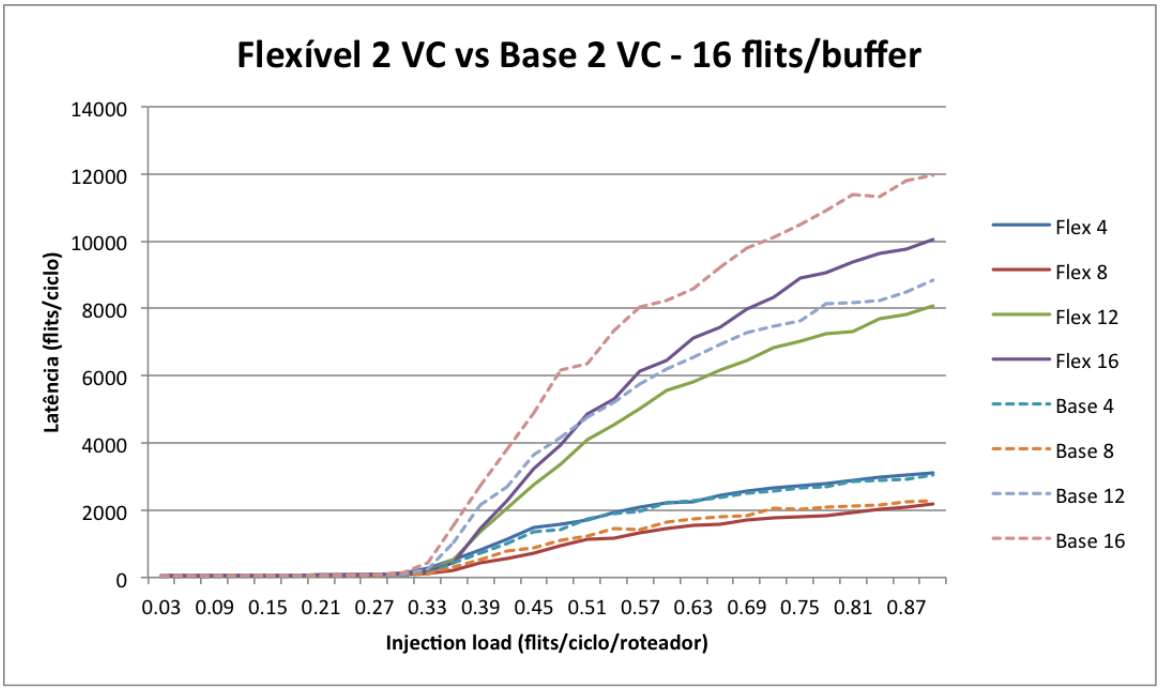


Figura 6.6: Latência - 2 Canais Virtuais e 16 flits por buffer.

Conforme aumenta-se a quantidade de recursos nos roteadores, a diferença de desempenho entre as duas arquiteturas é reduzida. Isto se deve ao fato de que a arquitetura flexível ter sido desenvolvida para trabalhar quando os recursos são escassos. À medida que aumenta-se o número de recursos, o *overhead* do mecanismo flexível acaba impactando o desempenho do roteador. Para roteadores com quatro canais virtuais, a diferença de desempenho entre os dois roteadores é pequena.

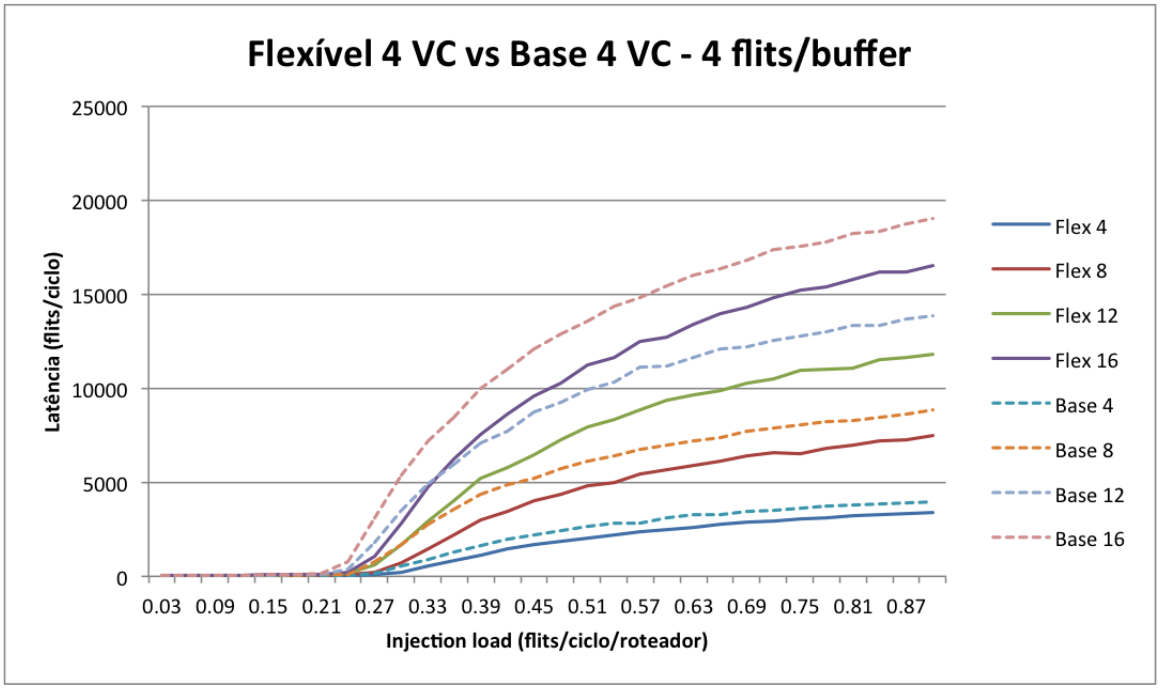


Figura 6.7: Throughput - 4 Canais Virtuais e 4 flits por buffer.

Utilizando-se pacotes de 16 *flits*, o roteador flexível tem melhor desempenho para tamanhos de *buffer* de 4, 8 e 16 *flits* (6% para 4 *flits*/*buffer* e 8 *flits*/*buffer* e 3% para 16 *flits* por *buffer*, Figuras 6.8, 6.9 e 6.11). É importante observar também que o roteador flexível, nos casos mencionados anteriormente, possui ponto de saturação maior do que roteador base. Isto faz com que a rede consiga processar mais pacotes simultaneamente. Embora o roteador base tenha *throughput* menor nos casos mostrados anteriormente, para os casos de pacotes com 4 *flits* e *buffers* de tamanho 8 e 16 *flits*, o roteador base obteve melhor performance (10% e 8% respectivamente. Figuras 6.9 e 6.11). Quando um pacote é grande, ele necessita de mais recursos para ser alocado, podendo ser necessário que a alocação do mesmo seja feita em até mais de um roteador (no caso de *buffers* pequenos). Isto faz com que a alocação diferenciada do roteador flexível resulte em melhores resultados, porém para pacotes pequenos, como o caso de pacotes de 4 *flits*, menos recursos são necessários para alocar pacotes, fazendo com que o base tenha um desempenho melhor devido a sua simplicidade de operação.

Como podemos ver nos gráficos, os melhores resultados são os que o tamanho do pacote se encaixa perfeitamente com o tamanho do *buffer* (a quantidade de *flits* que um canal virtual consegue conter). Uma explicação para isto é que quando o pacote cabe inteiramente no canal virtual, ele pode ser movimentado inteiramente de um roteador a outro, além dele ter acesso exclusivo ao recurso, ou seja, um pacote nunca fica bloqueado esperando o pacote anterior ser liberado. Conforme os pacotes aumentam de tamanho, será necessário mais de um roteador para alocar este pacote,

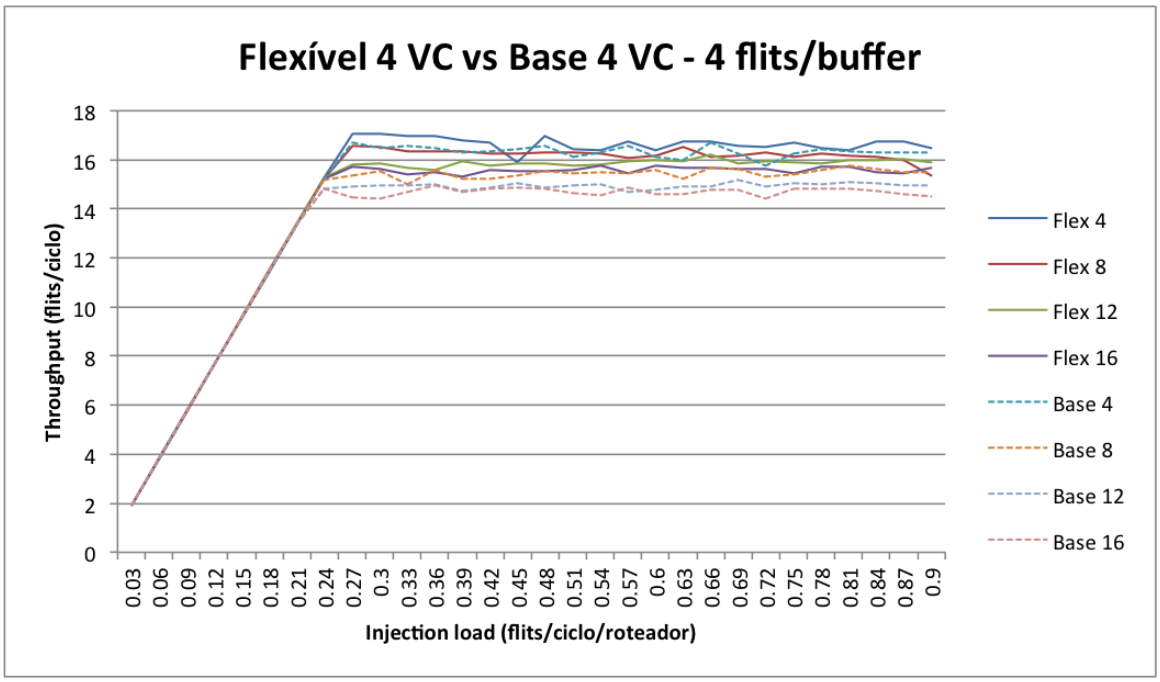


Figura 6.8: Latência - 4 Canais Virtuais e 4 flits por buffer.

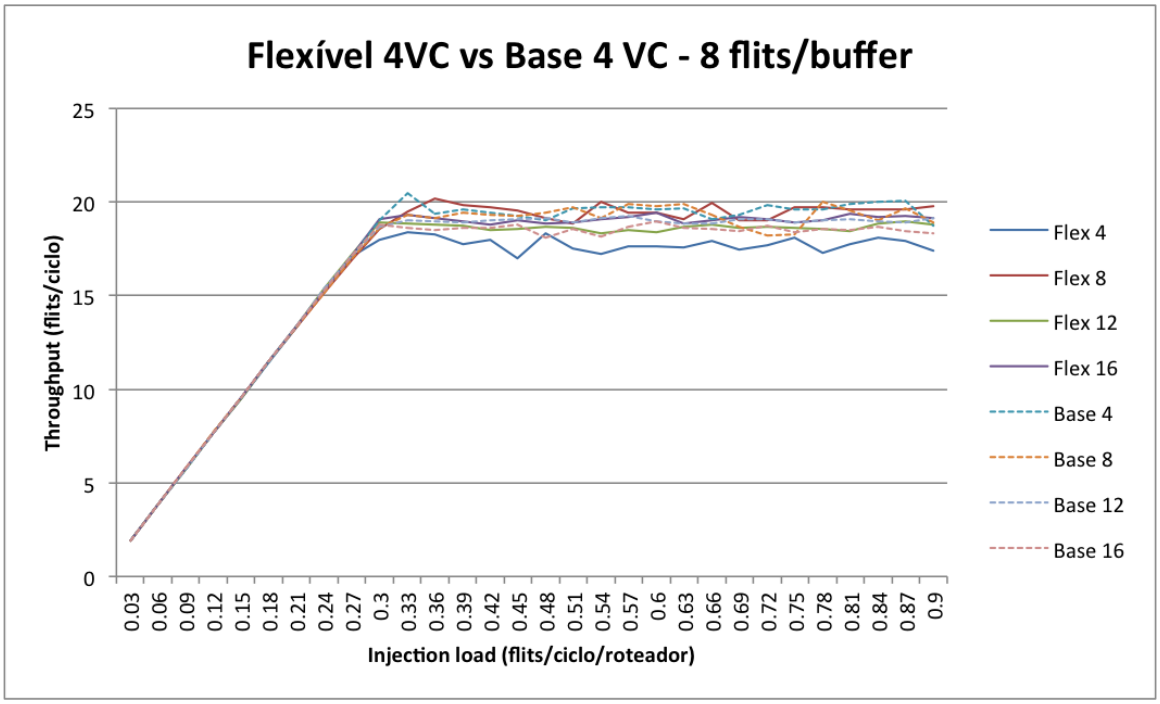


Figura 6.9: Throughput - 4 Canais Virtuais e 8 flits por buffer.

podendo ser necessário até cinco roteadores para isto (4 *flits* por *buffer* e pacotes de 16 *flits*), o que acaba gerando congestionamento.

Uma outra observação feita neste trabalho e em [52] é a comparação de desempenho em relação a *throughput* entre o roteador flexível com dois canais virtuais e o roteador base com 4 canais virtuais, ambos com 4 *flits* por *buffer*. Como podemos ver na Figura 6.13, embora a quantidade *buffers* no roteador flexível seja a metade

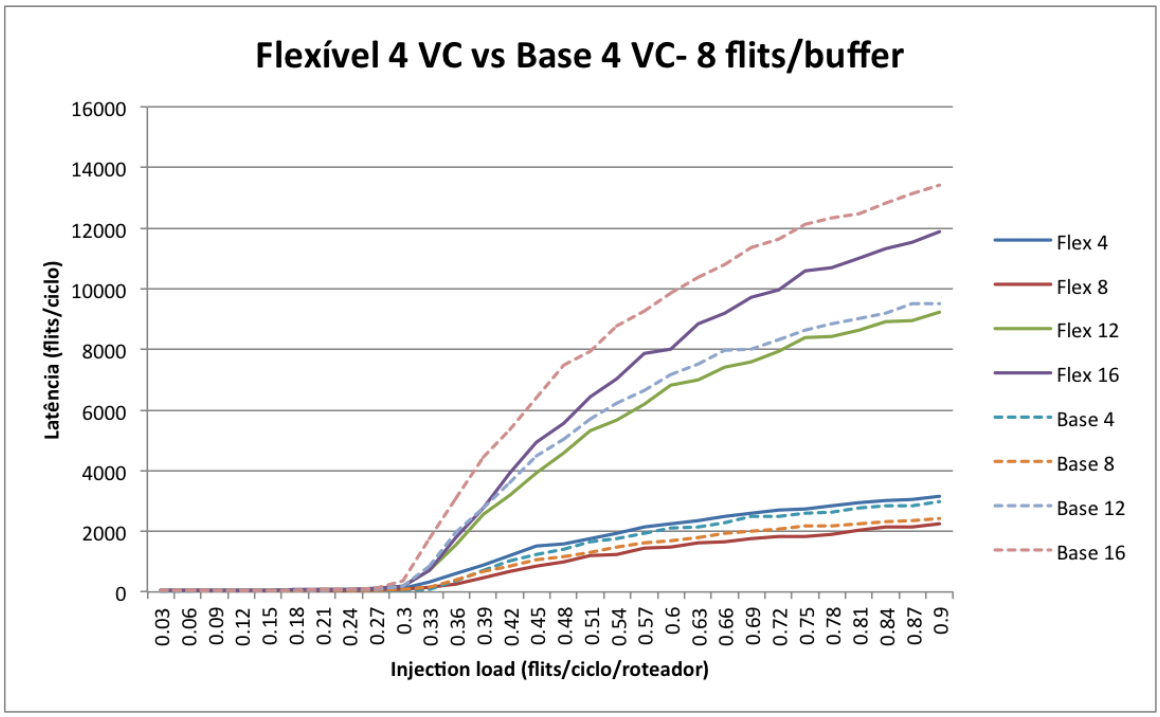


Figura 6.10: Latência - 4 Canais Virtuais e 8 flits por buffer.

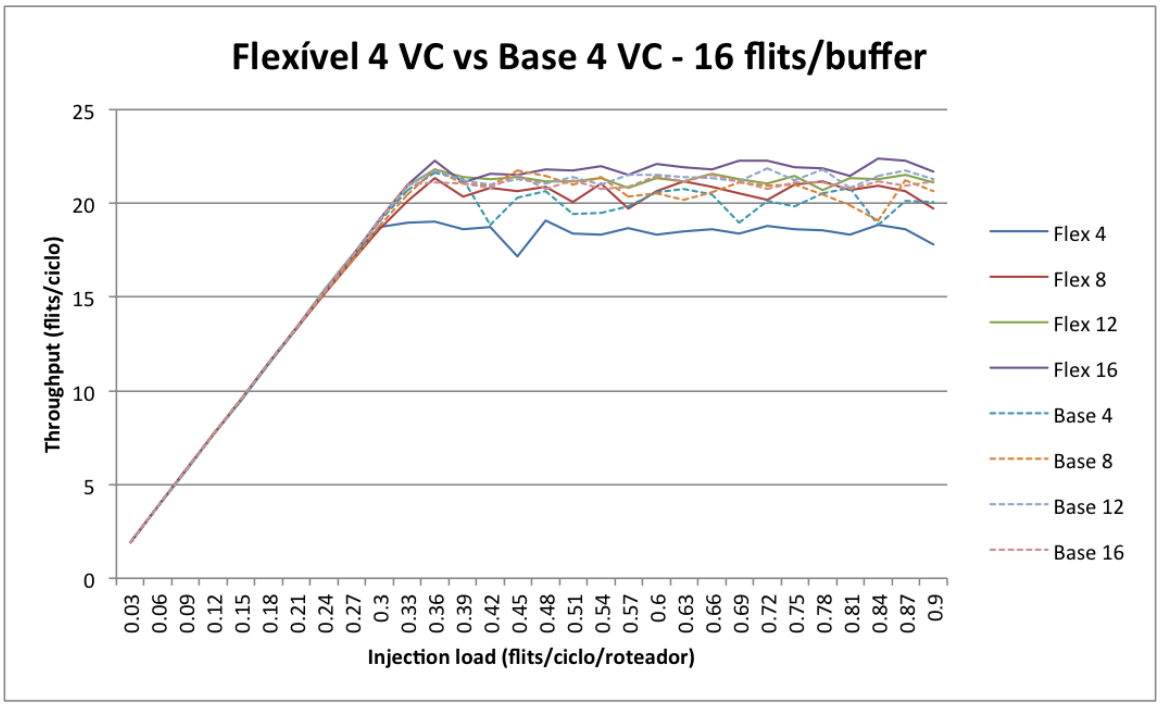


Figura 6.11: Throughput - 4 Canais Virtuais e 16 flits por buffer.

da quantidade no roteador base, o *throughput* deste roteador é apenas 3% inferior. Quando foram adicionados mais dois canais virtuais ao roteador base, o problema de contenção foi amenizado, aumentando seu *throughput*. No entanto, o roteador flexível obteve o mesmo desempenho do roteador base utilizando apenas metade dos *buffers*. O que gera uma economia no *design* do projeto, visto que são necessários

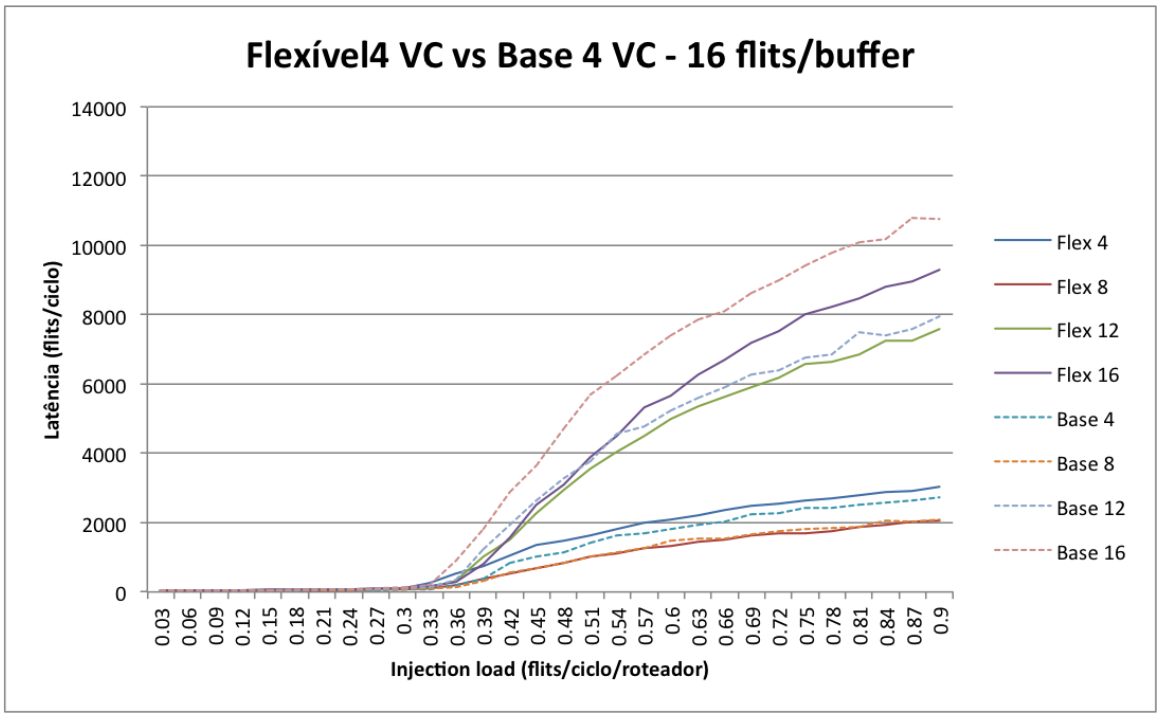


Figura 6.12: Latência - 4 Canais Virtuais e 16 flits por buffer.

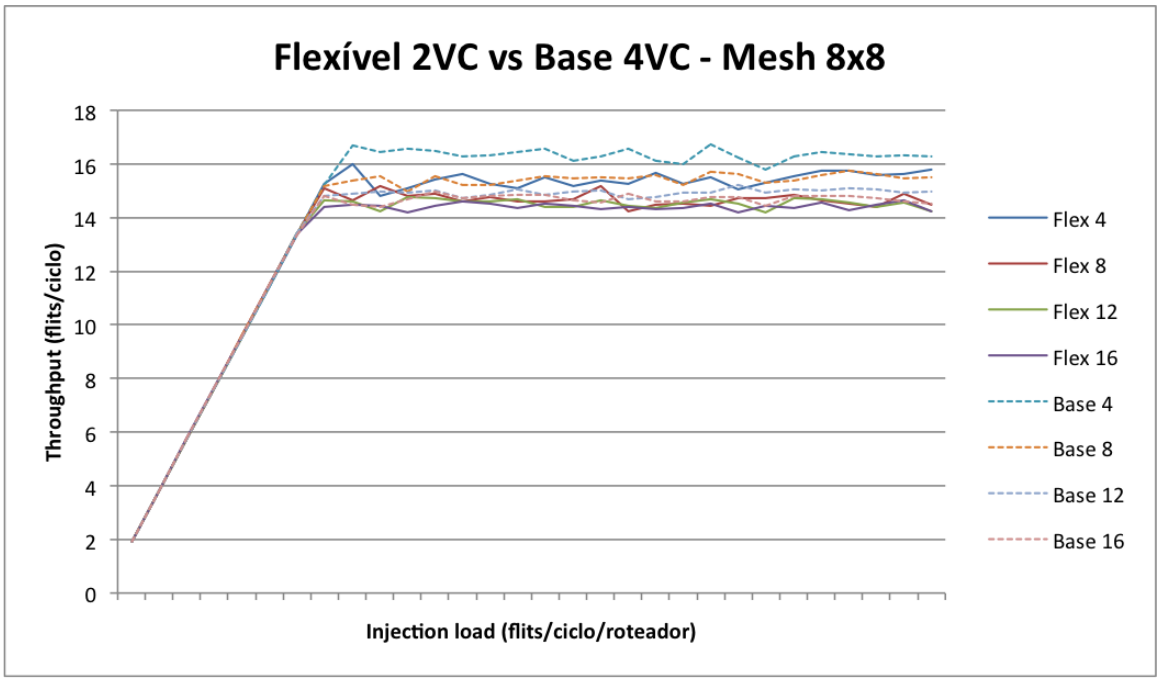


Figura 6.13: Throughput - Roteador Flexível com 2 Canais Virtuais e Roteador Base com 4 Canais Virtuais.

apenas metade dos recursos. Além disto, como mencionado no Capítulo 2, *buffers* ocupam a maior parte da área dos roteadores e são componentes que consomem uma quantidade considerável de energia. Assim, devido a esta redução na quantidade de *buffers*, o roteador flexível se torna uma alternativa menos dispendiosa energeticamente.

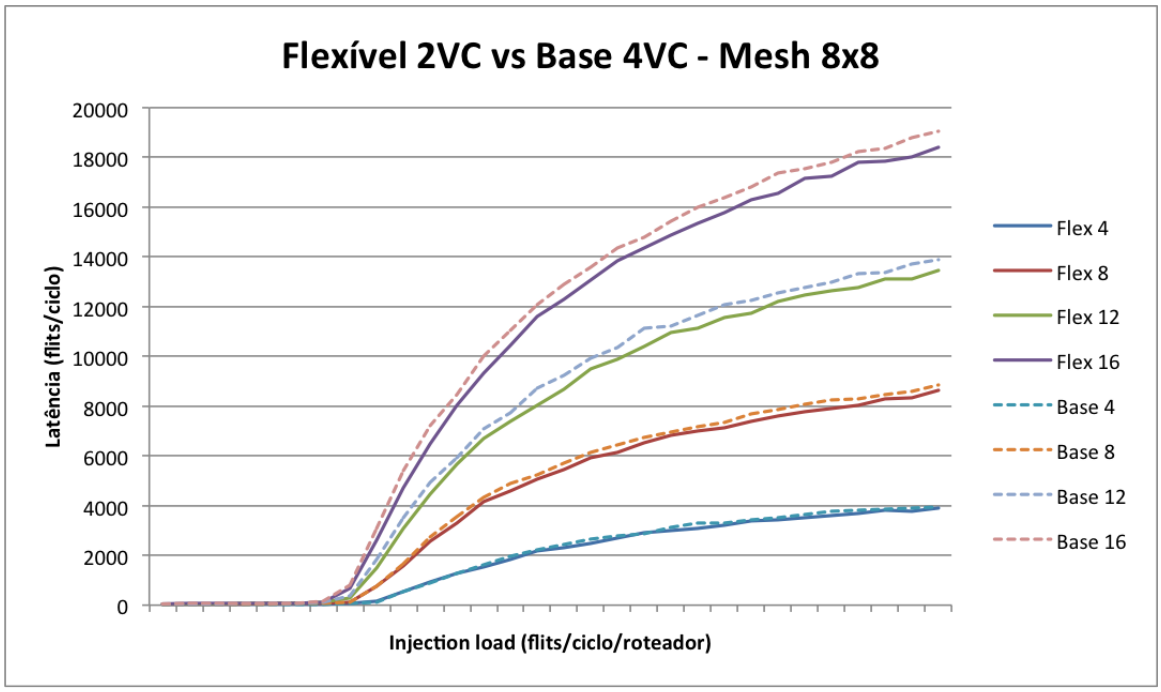


Figura 6.14: Latência - Roteador Flexível com 2 Canais Virtuais e Roteador Base com 4 Canais Virtuais.

Para encontrar o motivo da melhora de desempenho do roteador flexível, foi feita uma análise do uso de cada *buffer* dos roteadores ao decorrer de um período. Foram considerados apenas os roteadores nas diagonais do primeiro quadrante da rede conforme mostra a Figura 6.15. Pode se dizer que esta porção da NoC representa o mesmo comportamento para todos os outros roteadores da rede em outros quadrantes porque o padrão de tráfego utilizado é o uniforme. Os roteadores dos quadrantes tem comportamento parecido com seus vizinhos, por isto não foi necessário ser feito a análise de todos os roteadores do quadrante. Foi analisado o uso individual dos *buffers* da rede para saber como ele está sendo utilizado por cada *buffer*.

Como o objetivo era analisar as razões da melhoria de desempenho, o estudo foi feito apenas para o caso em que o roteador exibe os melhores resultados: *Buffer* de tamanho 4 e Pacotes de tamanho 8. O número de pacotes injetados na rede foi reduzido de 50.000 para 25.000 para obter-se simulações mais curtas, foi visto empiricamente que este número, embora seja menor do que o utilizado nos experimentos anteriores, possui comportamento similar ao dos mesmos. O *throughput* do roteador para este caso não foi muito afetado. Foram tomadas vinte amostras em dois períodos de simulação: Uma amostra a cada mil ciclos de simulação e uma amostra após a injeção de mil pacotes. As amostras em relação aos períodos de simulação são tiradas para que seja visto a utilização dos *buffers* ao longo do tempo. Em relação a quantidade de pacotes injetados para que seja visualizado a ocupação dos *buffers* sabendo-se que um determinado número de pacotes já foi inserido na rede.

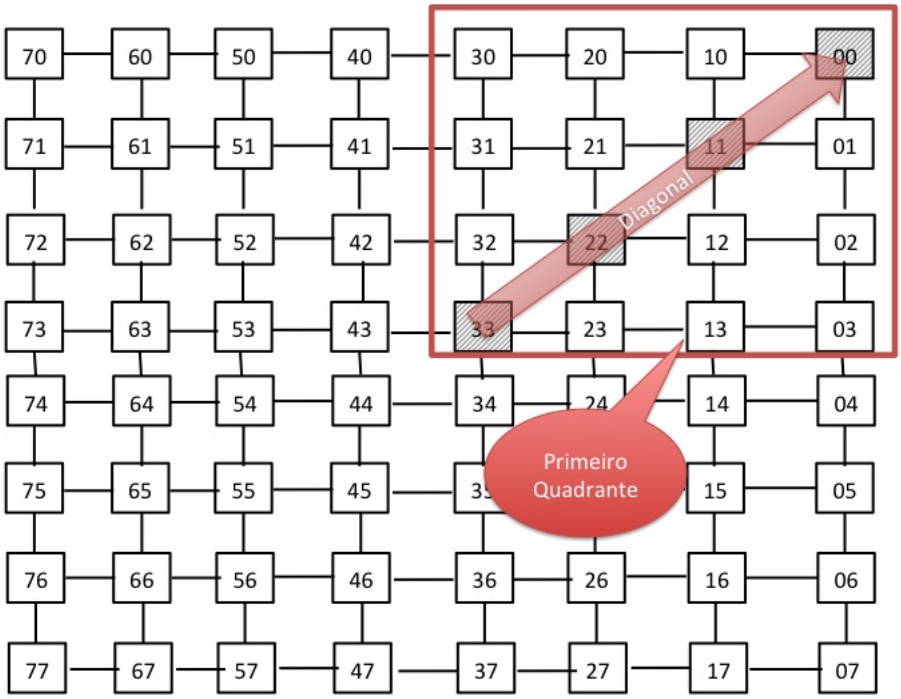


Figura 6.15: Primeiro quadrante da rede. Foram analisados os roteadores (0,0),(1,1),(2,2) e (3,3).

A análise começa com os resultados em relação aos pacotes. Esta simulação objetiva visualizar a variação do uso dos *buffers* à medida que são injetados pacotes na rede. As figuras 6.16 e 6.17 mostra a variação da utilização dos *buffers* dos roteadores base e flexível respectivamente durante o intervalo de injeção de 20.000 pacotes.

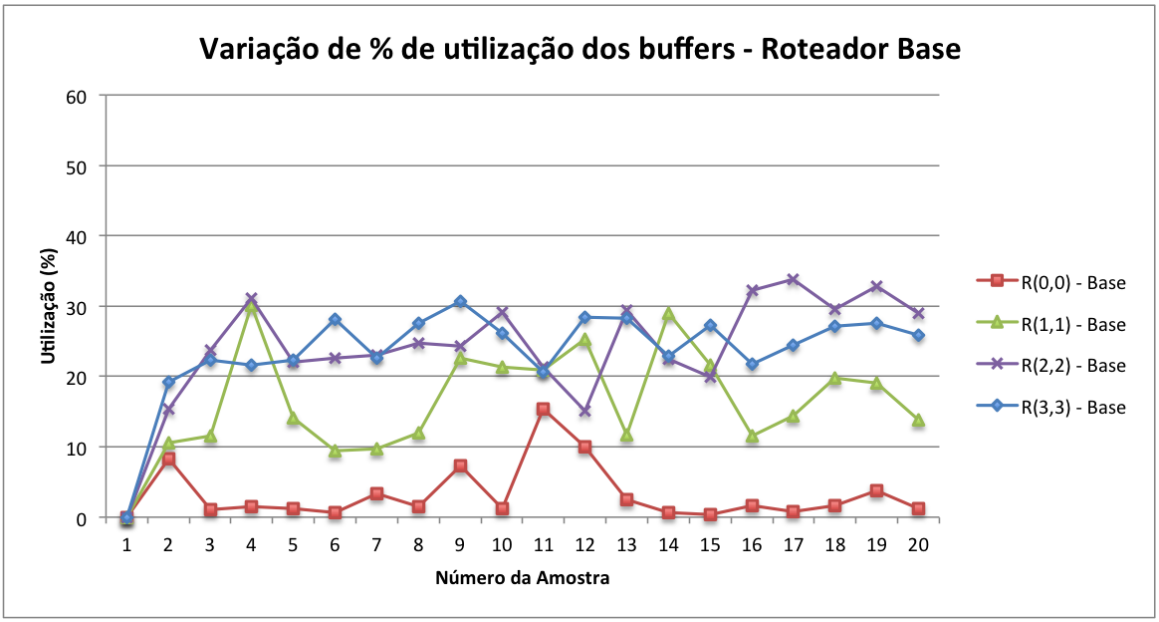


Figura 6.16: Taxa de utilização dos buffers em relação à quantidade de pacotes - Roteador Base.

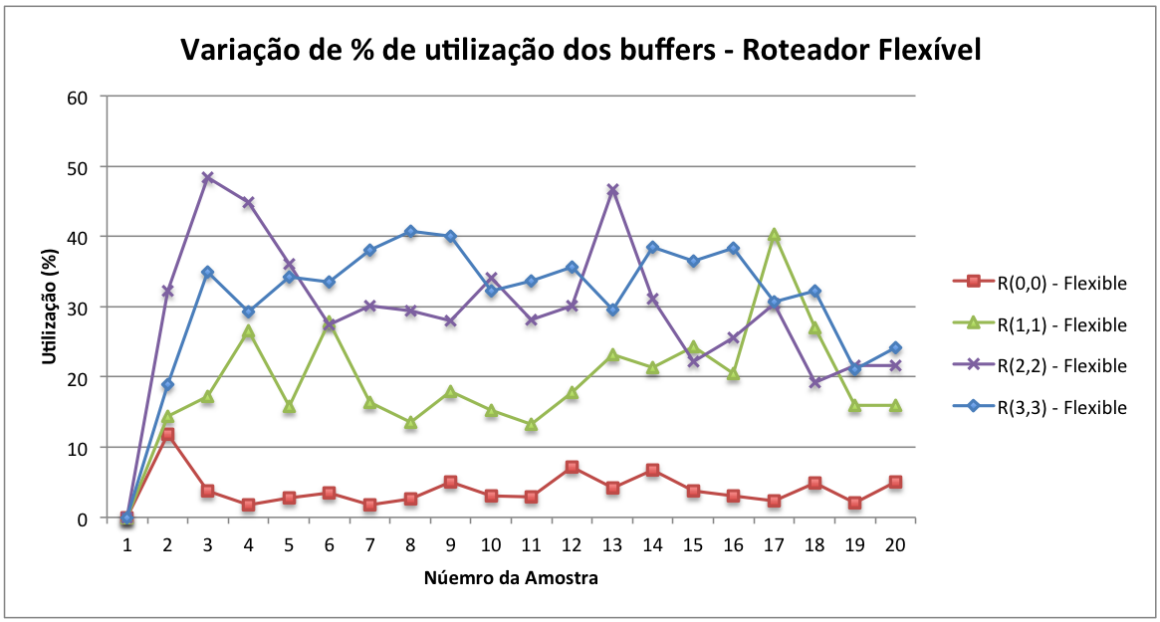


Figura 6.17: Taxa de utilização dos buffers em relação a injeção de pacotes - Roteador Flexível.

Como pode ser visto, a utilização dos *buffers* dos roteadores da arquitetura flexível são maiores do que os da arquitetura base. Para os roteadores localizados perto do centro da rede, roteadores (2,2) e roteadores (3,3), a diferença de utilização chega a ser de 42%. Isto se deve ao fato do mecanismo de alocação de *buffers* do roteador flexível distribuiu melhor o tráfego entre os *buffers* aumentando assim a quantidade de carga que este roteador consegue suportar.

A segunda parte da análise é em relação ao tempo. Mediu-se a utilização dos *buffers* retirando amostras a cada período de 1000 ciclos. As figuras 6.18 e 6.19 mostra a evolução da utilização dos *buffers* dos dois roteadores através de 20.000 ciclos.

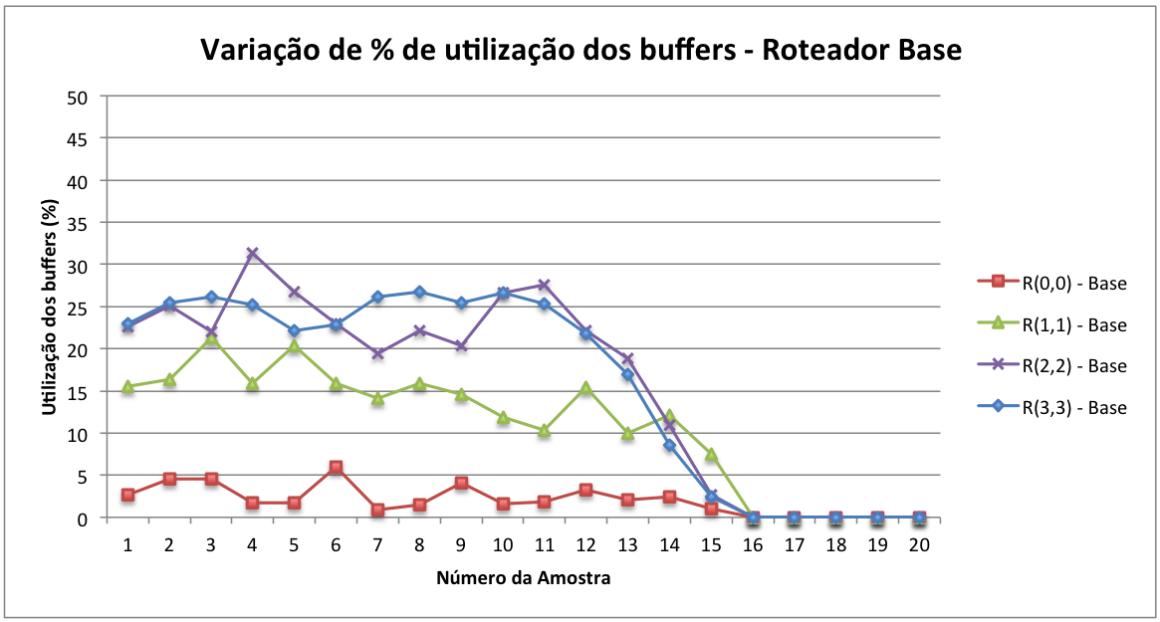


Figura 6.18: Taxa de utilização dos buffers em relação ao tempo - Roteador Base.

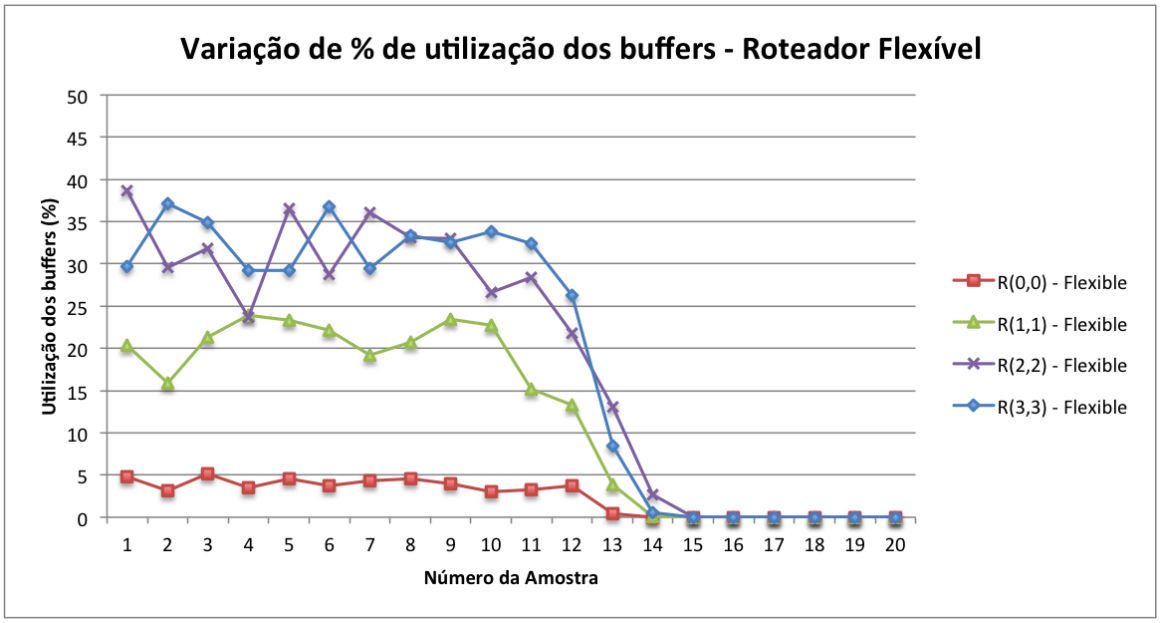


Figura 6.19: Taxa de utilização dos buffers em relação ao tempo - Roteador Flexível.

Como esperado, assim como nos gráficos de utilização em relação a pacotes, a utilização de *buffers* do roteador flexível em relação ao tempo é maior que do base. Como mais pacotes são transmitidos e consequentemente ejetados da rede, a execução do flexível termina aproximadamente 1000 ciclos antes do base. Um outro benefício da utilização dos *buffers* ser mais uniforme é a questão de confiabilidade do sistema. Com a distribuição do tráfego nos *buffers*, a probabilidade de um ficar mais sobrecarregado do que o outro diminui, o que consequentemente diminui o risco deste recurso quebrar, aumentando assim a confiabilidade do sistema.

6.2 Avaliação da Área e Potência

Neste segundo conjunto de experimentos, o objetivo é fazer a análise da diferença de área e potência dissipada do roteador flexível e do roteador base em relação às suas contrapartes assíncronas. Para gerar os arquivos necessários, foram utilizados casos de teste executados através da ferramenta de simulação *iSIM* [53]. Para extrair as informações de área e potência dissipada, foram utilizadas duas ferramentas de síntese:

A primeira é a ferramenta de lógica programável *Xilinx ISE Synthesis Tool (XST) 14.7*, com o circuito implementado em uma FPGA (do inglês *Field Programable Gate Array*) Virtex-5 com dispositivo alvo *xc5vfx50t-1ff1136*. Esta ferramenta gera resultados de ocupação de área para o dispositivo selecionado, além de fornecer uma ferramenta auxiliar, denominada *xPowerAnalyzer*, para que seja feita a estimativa de dissipação de potência.

A segunda ferramenta é a de síntese *Cadence Encounter Digital Implementation System* para VLSI - *Very Large Scane Integration* usando células-padrão da tecnologia de transistores de *90 nm* para a obtenção dos resultados de área e potência dissipada.

6.2.1 Parâmetros arquiteturais

Ambos os roteadores, flexível e base, utilizados nesta seção utilizam o mecanismo de controle de fluxo *Store-and-Forward* e não possuem canais virtuais. Isto é devido a complexidade de se implementar canais virtuais e controle de fluxo *wormhole* em hardware. O código dos dois roteadores utilizados, em suas versões síncronas, foram os implementados em [54]. Ambos implementados na linguagem de descrição de hardware *Verilog*. Neste trabalho criou-se a versão assíncrona dos mesmos para comparação.

Nos experimentos a seguir foram criados quatro roteadores, organizados como uma topologia *Mesh* 2x2, para que fosse analisada a diferença entre as duas versões: síncrona e assíncrona, adicionados aos *overheads* inerentes à criação de uma rede. Foi escolhida uma rede pequena porque o dispositivo alvo utilizado não suporta redes maiores devido a sua limitação de recursos. A rede é organizada como é mostrado na Figura 6.20.

A aplicação do procedimento de conversão de circuitos síncrono-assíncrono, apresentado no Capítulo 3, foi realizada em cada um dos dois roteadores para obter suas versões assíncronas.

Controladores de nós e controladores de arestas foram implementados em circuitos lógicos conforme apresentado na Seção 3.5. Um controlador de aresta é formado

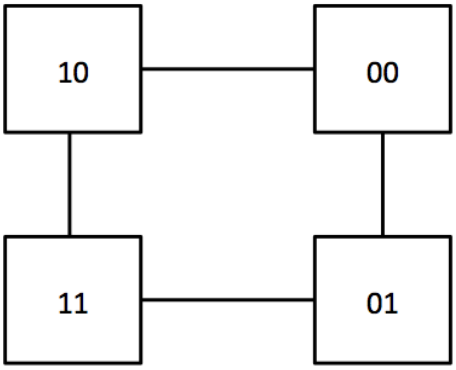


Figura 6.20: NoC - Tamanho 2x2.

por uma porta lógica do tipo *XOR* de duas entradas e outra do tipo inversor. O controlador de nó, por sua vez, é constituído por portas do tipo *AND*, cujo número de entradas depende diretamente do número de arestas que estejam conectadas ao nó, além de um registrador para sinalizar o estado de final de operação.

É importante informar também que a frequência utilizada pela tecnologia de VLSI foi de 200Mhz. Esta foi a maior frequência que foi possível de ser sintetizada no circuito desejado utilizando as ferramentas disponíveis.

6.2.2 Procedimentos Experimentais

A fim de realizar a extração dos valores da dissipação de potência, são necessários que vetores de excitação sejam aplicados coerentemente às entradas primárias dos roteadores. Para isto, foram utilizados ambientes de teste em que são inseridos um determinado número de pacotes nas portas de cada roteador e aguarda-se a saída de todos estes pacotes nas portas de saída dos roteadores. As variáveis da execução são gravadas em um arquivo *VCD (Value Change Dump)* durante a simulação. Feito isto, este arquivo é carregado juntamente com o arquivo de síntese resultando em uma base de dados que indicará à ferramenta de análise de potência como é o comportamento dinâmico dos circuitos dos roteadores. Os resultados obtidos através desta análise podem ser conferidos na próxima seção.

6.2.3 Resultados

A implementação do roteador flexível gera um aumento de área devido a inclusão do módulo CFF e da adição de fios e multiplexadores ao circuito necessários para sua implementação.

Como a metodologia utilizada para a conversão dos circuitos síncrono-assíncrono não é invasiva, ou seja, nenhuma das portas originais do circuito é mudada, nenhum caminho de dados é perturbado. Assim, após a criação dos nós sinalizadores, este foi

apenas adicionado ao circuito original e seus sinais de sinalização foram conectados ao circuito original.

Os elementos de sinalização gerados utilizados na versão assíncrona, como os controladores de nós e os controladores de arestas, estão relacionados na Tabela 6.2.

Tabela 6.2: Número de controladores de nós e arestas dos roteadores e redes.

Circuito	Controladores de Nós	Controladores de Nós de interconexão	Total de controladores de Nós	Controladores de arestas
Roteador Base	30	25	55	70
Roteador Flexível	34	29	64	78
Rede 2x2 (Base)	120	100	220	280
Rede 2x2 (Flexível)	136	116	252	312

A adição destes elementos sinalizadores impacta na área e na potência de cada circuito. Como o roteador flexível possui mais lógica do que o roteador base, é esperado que ele necessite de mais circuitos sinalizadores. A área resultante de cada circuito convertido assim como a área original do circuito síncrono são relacionadas na Tabelas 6.3 e 6.4.

Tabela 6.3: Comparação da área ocupada pelas versões síncronas e assíncronas dos roteadores base, flexível, além das redes construídas a partir dos mesmos através da tecnologia FPGA.

Circuito	Síncrono			Assíncrono			**Diferença (%)**
-	FF	LUT	PAR	FF	LUT	PAR	-
Roteador Base	615	928	1.146	720	1.161	1.356	18,3
Roteador Flexível	647	1.223	1.394	765	1.459	1.660	19,1
Rede 2x2 (Base)	1.768	3.172	3.787	2.192	3.320	3.981	5,1
Rede 2x2 (Flexível)	2.247	3.446	4.313	2.695	4.015	4.770	10,5

Tabela 6.4: Comparação da área ocupada pelas versões síncronas e assíncronas dos roteadores base, flexível, além das redes construídas a partir dos mesmos através da tecnologia VLSI (tecnologia 90nm).

Circuito	Síncrono	Assíncrono	Diferença (%)
Roteador Base	23820,5	25025,5	5,0
Roteador Flexível	24886,5	26448,2	6,0
Rede Base	95232,7	100077,7	5,0
Rede Flexível	99507,7	104740,7	5,0

As FPGAs utilizam *lookup tables (LUTs)* para fazer a reconfiguração de seu circuito e implementar o hardware descrito na linguagem HDL. O circuito dos controladores de nós e arestas usam muitos contadores, o que não é otimizado para FPGAs e acabam gerando um *overhead* de LUTs, principalmente no módulo da porta de entrada. O que não se conforma facilmente com a arquitetura interna dos FPGAs, visto que esta é baseada em LUTs. Quando os circuitos foram sintetizados utilizando tecnologia VLSI, utilizando *NAND Gates*, este *overhead* foi reduzido e a diferença entre os dois circuitos foi suavizada. De forma que esta diferença passou de 19% (Roteador Flexível) para um máximo de 6%(Roteador Flexível). No caso das redes, a diferença entre a rede de roteadores flexível também caiu pela metade, de 10% para 5% utilizando-se VLSI.

O último ponto analisado na comparação entre as versões síncronas e assíncronas dos roteadores, e redes, foi a quantidade de potência dissipada ao serem postos a executar a mesma tarefa. Novamente, através do simulador *iSIM*, foram inseridos pacotes em cada porta de entrada dos roteadores e aguardou-se que o último pacote inserido deixasse o roteador, comprovando que os circuitos equivalentes realizaram tarefas idênticas. Foi então gravada a excitação das portas durante a execução em um arquivo de formato *VCD*. Este arquivo foi então utilizado em conjunto com o mapeamento feito pelas ferramentas de síntese. Para calcular-se a potência na tecnologia de FPGA, utilizou-se o *VCD* como entrada para a ferramenta de análise de potência *xPowerAnalyser*.

Como nenhuma porta lógica existente na versão síncrona é alterada quando convertida para a versão assíncrona, a potência estática dissipada pela versão assíncrona deve ser maior do que a versão síncrona, visto que esta é diretamente proporcional à quantidade de CLBs (do inglês *Configurable Logic Blocks*) utilizadas para implementar o circuito na FPGA. As Tabelas 6.5 e 6.6 apresentam a variação das potências das duas implementações. O que se apresenta a seguir é a soma da potência dissipada acumulada através de toda a simulação.

Tabela 6.5: Comparação da energia utilizada pelas versões síncronas e assíncronas dos roteadores base, flexível, além das redes construídas a partir dos mesmos utilizando tecnologia FPGA.

Circuito	Síncrono μW	Assíncrono μW	Diferença (%)
Roteador Base	0,575	0,597	3
Roteador Flexível	0,631	0,625	-1
Rede Base	0,856	0,838	-2
Rede Flexível	0,903	0,968	7

Tabela 6.6: Comparação da energia utilizada pelas versões síncronas e assíncronas dos roteadores base, flexível, além das redes construídas a partir dos mesmos utilizando tecnologia VLSI.

Circuito	Síncrono μW	Assíncrono μW	Diferença (%)
Roteador Base	1493902,6	1591418,4	6
Roteador Flexível	1651817,2	332889473,8	100
Rede Base	6042893,0	6441646,0	6
Rede Flexível	5992440,9	6558522,9	9

Quando analisado apenas os roteadores, o roteador flexível apresenta um aumento da quantidade de potência dissipada, porém quando é criada uma rede, ainda que pequena, é possível visualizar que existe uma economia de energia com a utilização do roteador flexível. Isto se deve ao fato de a rede estar operando de forma mais eficiente, reduzindo o congestionamento que aumentaria a potência dissipada pela mesma, visto que os *buffers* seriam utilizados por mais tempo.

Quanto à versão assíncrona dos mesmos, pode ser visto que houve um aumento de potência dissipada. Isto se deve ao fato de proporcionalmente, o *overhead* dos circuitos ASERT serem muito grandes, em relação ao circuito do roteador. Além disto, foi utilizado o *diet clock* para gerar os sinais de fim de operação, o que não é ideal quando se trata de nós de sinalização assíncronos. O ideal é que se seja utilizada uma forma de eliminar completamente o *clock* para que o consumo de energia do circuito possa ser corretamente avaliado. No mais, as ferramentas de síntese que este autor possui acesso não permitem a criação de circuitos completamente assíncronos, de forma que deve ser fornecido um sinal de *clock* para cada módulo, o que acaba influenciando no consumo de potência.

O aumento do consumo de potencia é maior no roteador flexível do que no roteador base devido a forma como os mesmos foram implementados. Enquanto o roteador base utiliza um sinal único de *clock*, que é então substituído pelo sinal dos

controladores de nós, o circuito do flexível utiliza 5 sinais de *clock* com frequências diferentes devido ao seu mecanismo de realocação de canais virtuais que deve ser executado em paralelo com a execução normal do circuito. Como apenas o sinal que vai aos módulos principais, o que compõe o caminho crítico do sistema, é alterado pela metodologia *ASERT*, isto acaba afetando os outros sinais de *clock* fazendo com que a quantidade de energia gasta seja maior. Além disto, devido ao *diet clock*, os contadores utilizados para gerar os sinais de fim de operação operam a uma frequência muito rápida, fazendo chaveamento a todo tempo, e devido a isto, consumindo uma energia muito grande.

Os resultados de comparação de área entre os roteadores base e flexível síncronos não são abordados nesta dissertação visto que os mesmos foram avaliados anteriormente em [54].

6.3 Resumo do Capítulo

Neste capítulo foram realizados dois conjuntos de experimentos para avaliar o desempenho das versões síncronas e assíncronas dos roteadores base e flexível. Na sua versão síncrona, foi mostrado que quando os recursos da rede são limitados (como *buffers* com pouco espaço e pouco número de canais virtuais), o roteador flexível tem um *throughput* até 21% maior e latência até 62% em comparação com o roteador base. Quando a quantidade de recursos aumenta, esta diferença começa a diminuir e existem casos em que o roteador flexível tem desempenho pior em comparação com o roteador base. Na comparação de roteadores síncrono e assíncrono, foi visto que embora a metodologia ASERT adicione um *overhead* ao circuito original, este ocupa no máximo 5% da área do circuito, o que torna a alternativa viável. Quanto a economia de energia, não foi possível ainda obtê-la, porém mesmo com a adição de circuitos extras foi visto que a diferença de energia chegou a ser de 6%.

Capítulo 7

Discussões e trabalhos futuros

O trabalho nesta dissertação apresenta novas ideias para o desenvolvimento de roteadores para *NoCs* que utilizam seus recursos disponíveis de forma mais eficiente. Contudo, aspectos novos foram abordados e possibilidades futuras apontadas. A pesquisa aqui apresentada buscou explorar além do que já havia sido feito, porém sem perder a conexão com os trabalhos predecessores.

Esta seção encerra este trabalho, consolidando, nas duas seções a seguir, as conclusões e ponderações mais importantes (Seção 7.1) e investigando novas oportunidades de melhorias que podem ser implementadas (Seção 7.3).

7.1 Conclusão

O trabalho exposto nesta dissertação, apresenta o RAF - Roteador Assíncrono Flexível, uma nova arquitetura de roteadores para *NoCs*. Arquitetura assíncrona pela qual, através da reconfiguração dos *buffers* de forma dinâmica em tempo de execução, consegue obter, em determinados casos, desempenho superior a arquitetura padrão utilizada pelo roteador base. O mecanismo que torna esta reconfiguração de *buffers* possível, foi pensado de forma a ter um fluxo de execução simples, e que pudesse reduzir o problema de bloqueio de pacotes, sem a necessidade da adição de *buffers* adicionais ao roteador. Além disto, este roteador é assíncrono, o que teoricamente o permitirá reduzir custos energéticos, assim que as limitações explicadas no capítulo anterior forem resolvidas.

Quando é solicitado um canal virtual a uma porta do roteador flexível, o espectro de busca de *buffers* desta porta não é limitado a apenas *buffers* contidos em sua porta de entrada local. Quando esta porta não possui canais virtuais disponíveis, é feito uma busca em todas as portas do roteador, de forma que caso exista alguma porta com canais virtuais ociosos, os mesmos podem ser alocados para atender a solicitação. Para que fosse demonstrado o potencial do roteador flexível, sua arquitetura foi implementada no simulador TOPAZ. O simulador não possui meca-

nismos de medida de área e nem de consumo de energia. Para avaliar os mesmos, foi utilizado uma versão mais básica do roteador implementado em [54] na linguagem de descrição de hardware *Verilog*. Embora esta versão seja um pouco diferente da implementada no simulador TOPAZ, ela atende a necessidade pois implementa os mecanismos essenciais para a funcionalidade da arquitetura do roteador flexível. Técnicas de dessincronização foram aplicadas aos roteadores base e flexível de forma que fosse possível analisar o consumo de energia em uma versão *low-power* de ambos.

Nos experimentos apresentados no Capítulo 6, diferentes parâmetros foram explorados, objetivando encontrar casos em que a arquitetura roteador flexível se torna vantajosa em relação à arquitetura do roteador base nos quesitos de *throughput* e latência e também caracterizar a diferença em termos de consumo de energia das versões síncronas e assíncronas dos mesmos. Em relação a *throughput* e latência, na maioria dos casos o problema de contenção foi amortizado e ambos foram melhorados, chegando a resultados 21% para *throughput* e 62% para latência melhores do que o roteador base. Atribui-se estes resultados à simplicidade e agilidade com que o roteador flexível trata a reassociação virtual dos *buffers*. Em relação a conversão síncrono-assíncrono do sistema, notou-se que *overhead* de área do circuito de sinalização foi de tamanho razoavel, tendo seu máximo em 19% para FPGAs e 6% para VLSI. Foi visto também que embora economia de energia fosse um resultado esperado, isto se deve ao tamanho do circuito ser muito pequeno e ao sinal de *clock* utilizado que acabou gerando chaveamentos desnecessários durante a execução do circuito, o que impossibilitou que as vantagens da aplicação da metodologia ASERT se tornassem visíveis.

Nos cenários em que os recursos do roteador são escassos, o *overhead* de reconfiguração é baixo, quando comparado com a contenção que pode ser causada pelo bloqueio de pacotes. Nessa situação, verificou-se que os resultados do roteador flexível foram melhores. À medida que a quantidade de recursos aumenta, foi observado que a arquitetura proposta sofreu penalidades devido ao *overhead* intrínseco de seu mecanismo de funcionamento.

Embora o *overhead* de área dos controladores ASERT não sejam fatores impeditivos para a construção do roteador assíncrono, o mecanismo de geração de sinal de final de operação, o *diet clock*, se mostrou desvantajoso para este tipo de circuito. Foi observado que será necessário implementar uma forma que consuma menos energia para gerar este sinal.

Por tudo que foi dito acima, conclui-se que o Roteador Flexível Assíncrono é uma arquitetura poderosa, e deve ser considerado como uma alternativa ao roteador base quando a rede possuir poucos recursos disponíveis.

7.2 Lições Aprendidas

Nestes dois anos e meio de mestrado, muitas lições foram aprendidas, nem todas relacionadas ao conteúdo técnico apresentado neste trabalho, na verdade, muitas delas foram de engrandecimento pessoal e amadurecimento. Farei aqui um pequeno resumo ilustrando as principais: A primeira coisa ao termino desta dissertação que me vem a mente é o sentimento de que eu deveria ter começado a escreve-la antes. De fato, sempre usei a desculpa de não ter resultados suficientes para começar a escrever porém, nem tudo que está escrito são resultados. Talvez, se quando estava ainda estudando sobre os tópicos dos capítulos de *background* também escrevesse, os mesmos estariam muito mais didáticos do que os atuais. Deveria ter tentado escrever um pouco diariamente, mesmo que apenas uma frase, para que as pequenas lições diárias não fossem perdidas. Um outro fator que me ajudaria bastante seria aumentar a frequência de participação de defesas de mestrado e doutorado. Com certeza isto teria me ajudado a evitar alguns erros na hora de fazer a minha, além de ver de que forma eram apresentadas e quais eram os tipos de perguntas que eu poderia esperar. Outro grande ponto a ser discutido é a questão da capacidade de resolver problemas, principalmente os de implementação. Se eu tivesse utilizado mais minha capacidade analítica de resolver problemas, ao invés de tentar resolve-los da forma mais simples e rápida possível, eu teria evitado falácias que no final só atrasaram meu processo de geração de resultados. Dentre todas as lições aprendidas, destaco como sendo a lição mais importante a capacidade de analisar sempre um problema com os pés no chão, considerando todas as possibilidades (não somente as que vem na mente) e tentando os reduzir em problemas menores. Um bom exemplo disto foi a descoberta que minha implementação do roteador flexível era passível de *deadlock*. No inicio, quando os pacotes ficavam presos nos *buffers*, achava que tinha algum erro na memória, no *pipeline*,etc. Nunca desconfiei de *deadlock*. Somente quando estava cansado de não achar respostas foi que parei e pensei em todas as possibilidades. Quando revi minha lógica, encontrei o erro (que não tinha absolutamente nada a ver com o que tinha pensado) e fui capaz de resolvê-lo em pouco tempo. Se tivesse analisado friamente anteriormente, certamente não teria gastado dias (incluindo vários finais de semana) atrás deste problema. Por isto que além dos conhecimentos técnicos obtidos durante o processo de elaboração desta dissertação, destaco este como sendo o mais importante.

7.3 Trabalhos Futuros

Apesar de apresentar resultados promissores, a arquitetura do roteador flexível ainda possui muitas questões a serem trabalhadas. Podemos enumerar os trabalhos a serem

desempenhados futuramente nos seguintes pontos:

1. Problema de dependência entre *buffers* de dois roteadores: Este problema foi apresentado na Seção 4.3 e pode ser reduzido ou extinto através de mecanismos inteligentes no *buffer*. Um destes mecanismos pode ser informar aos roteadores vizinhos sobre sua carga atual, fazendo assim com que outros canais virtuais tenham preferência na hora da alocação.

2. Mecanismo de Tratamento de *Deadlock*: A forma atual em que os *deadlocks* são tratados, restringe dois ciclos do *turn-model* causando restrições no mecanismo de reconfiguração e consequentemente embarreirando possíveis ganhos em desempenho. Uma forma de lidar com este problema é a implementação de um mecanismo de prevenção de *deadlock* que não restrinja tanto o roteador, como a eliminação de menos ciclos do *turn-model*. Ou, remover todas as restrições e implementar mecanismos de detecção ou recuperação de *deadlocks*.

3. Utilização de Traces Reais: Para obter-se resultados do desempenho do roteador flexível em padrões de tráfego de sistemas reais. A fim de obter resultados mais reais, devem ser utilizados traces de aplicações que rodam em *System-on-Chips* a fim de obter resultados mais realísticos.

4. Dessincronização do roteador flexível: A versão assíncrona do roteador flexível, mostrou um consumo de energia maior em relação a síncrona. Isto pode estar relacionado com a forma que o mecanismo de reconfiguração de *buffers* foi implementado em hardware. A metodologia *ASERT* é não-invasiva, ou seja, a arquitetura original do roteador flexível foi mantida, sendo substituída apenas os sinais de *clocks*. Sinais estes que eram utilizados como input em cada módulo e, foram substituídos por sinais de controles de operação gerados pelos controladores nós da metodologia *ASERT*. Se utilizarmos uma forma invasiva, ou seja, alterando o projeto do circuito, o mesmo pode ser re-projetado e aperfeiçoado para que a sua versão assíncrona obtenha ganhos melhores em termos de energia, sem comprometer o desempenho.

5. Eliminação do *diet clock*: O circuito utilizado para gerar os sinais de final de operação acabaram adicionando um *overhead* extra em relação a consumo de energia ao sistema. Uma forma de lidar com este problema é a elaboração de um novo mecanismo que tenha um consumo de energia menor.

Referências Bibliográficas

[1] HUNT, M., ROWSON, J. "Blocking in a System on a Chip", *IEEE Spectrum*, v. 33, n. 11, pp. 35–41, November 1996.

[2] GEPPERT, L. "Solid State [trends/developments]", *IEEE Spectrum*, v. 33, n. 1, pp. 51–55, January 1996.

[3] CHANDRAMOULI, R., PATERAS, S. "Testing Systems on a Chip", *IEEE Spectrum*, v. 33, n. 11, pp. 42–47, November 1996.

[4] BENINI, L., DE MICHELI, G. "Networks on Chips: A New SoC Paradigm", *IEE Computer*, v. 35, n. 1, pp. 70–78, January 2002.

[5] HEMANI, A., JANTSCH, A., KUMAR, S., et al. "Networks on Chip: An architecture for billion transistor era", *Proceedings of the IEEE NorChip Conference*, 2000.

[6] DALLY, W. J., TOWLES, B. "Route Packets, Not Wires: On-Chip Interconnection Networks", *Design Automation Conference (DAC)*, pp. 684–689, 2001.

[7] DALLY, W. J., TOWLES, B. *Principle and Practices of Interconnection Networks*. Morgan Kaufman, 2004.

[8] YE, T. T., BENINI, L. "Analysis of power consumption on switch fabrics in network routers", *Proceedings of the 39th Design Automation Conference (DAC)*, pp. 524–529, 2002.

[9] XUNING, C., PEH, L. S. "Leakage Power Modeling and Optimization in Interconnection Networks", *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 90–95, August 2003.

[10] VARATKAR, G., MARCULESCU, R. "Traffic analysis for on-chip networks design of multimedia applications", *39th Design Automation Conference (DAC)*, pp. 795–200, 2002.

[11] JINGCAO, H., MARCULESCU, R. "Application-specific buffer space allocation for networks-on-chip router design", *International Conference on Computer Aided Design (ICCAD)*, pp. 354–361, November 2004.

[12] AHMAD, B., AHMADINIA, A., ARSLAN, T. "Dinamically reconfigurable NOC with bus based interface for ease integration and reduced designed time", *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 309–314, June 2008.

[13] AHONEN, T., NURMI, J. "Hierarchically heterogeneous network-on-chip", *International Conference Computer as a Tool (EUROCON)*, pp. 2580–2586, September 2007.

[14] VESTIAS, M., NETO, H. "Router design for application specific network-on-chip on reconfigurable systems", *Field Programable Logic Applications*, pp. 389–394, August 2007.

[15] BOUHRAOUA, A., ELRABAA, M. "Addressing heterogeneous bandwidth requirements in modified fat-tree network-on-chip", *International Symposium Eletronic Design Test Application (DELTA)*, pp. 486–490, 2008 2008.

[16] XU, Y., ZHAO, B., ZHANG, Y., et al. "Simple Virtual Channel Allocation for High Throughput and High Frequency On-Chip Routers", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–11, January 2010.

[17] TAMIR, Y., FRAZIER, G., L. "High-performance multiqueue buffers for VLSI communication switches", *International Symposium on Computer Architecture*, pp. 343–354, May-June 1988.

[18] NI, N., PIRVU, M., BHUYAN, L. "Circular buffered switch design with wormhole routing and virtual channels", *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pp. 466–473, October 1998.

[19] PARK, J., O'KRAFKA, B. W., V.-S., DELGADO-FRIAS, J. "Design and evalation of a DAMQ multiprocessor network with self-compacting buffers", *Conference on Supercomputing*, pp. 713–722, November 1994.

[20] LIU, J., DELGADO-FRIAS, J. "A DAMQ shared buffer scheme for Network-on-Chip", *Fifth IASTED International Conference*, pp. 54–58, 2007.

[21] NICOPOULOS, C., PARK, D., KIM, J., et al. "A gracefully degrading and energy-efficient modular router architecture for on-chip networks", *International Symposium on Computer Architecture*, pp. 4–15, 2006.

[22] RAMANUJAM, R., SOTERIOU, V., LIN, B., et al. "Design of a High-Throughput Distributed Shared-Buffer NoC Router", *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pp. 69–78, 2010.

[23] RAMANUJAM, R., SOTERIOU, V., LIN, B., et al. "Extending the Effective Throughput of NoCs With Distributed Shared-Buffer Routers", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 548–561, 2011.

[24] NICOPOULOS, C., PARK, D., KIM, J., et al. "ViChaR: A Dynamic Virtual Channel Regulator for Network-on-Chip Routers", *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 333–346, 2006.

[25] TAMIR, Y., FRAZIER, G. "High-performance multiqueue buffers for VLSI communication switches", *Proceeding of the 15th Annual International Symposium on Computer Architecture, ISCA*, pp. 343–354, 1988.

[26] MATOS, D., CONCATTO, C., KASTENSMIDT, F., et al. "Reconfigurable Routers for Low Power and High Performance", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 19, n. 11, pp. 2045–2057, 2010.

[27] SAYED, M. S., SHALABY, A., EL-SAYED, M., et al. "Flexible router architecture for network-on-chip", *Computers & Mathematics with Applications*, v. 64, n. 5, pp. 1301–1310, 2012.

[28] SAYED, M., SHALABY, A., RAGAB, M.-S., et al. "Congestion mitigation using flexible router architecture for Network-on-Chip", *Communications and Computers JEC-ECC*, pp. 182–187, March 2012.

[29] EL-SAYED, H., RAGAB, M., SAYED, M. S., et al. "Hardware Implementation and Evaluation of Flexible Router Architecture for NoCs", *20th IEEE Intl. Conf. on Electronics, Circuits and Systems*, pp. 621–624, 2013.

[30] FRANCA, F., ALVES, V., GRANJA, E. "Edge Reversal-Based Asynchronous Timing Synthesis", *International Symposium on Circuits and Systems*, v. 2, pp. 45–48, May - Jun 1998.

[31] ABAD, P., PRIETO, P., MENEZO, L., et al. "TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers", *NOCS*, pp. 99–106, May 2012.

[32] MOORE, G. "Cramming More Components on to Integrated Circuits", *Eletronics*, v. 38, n. 8, pp. 114, April 1965.

[33] DALLY, W. J., OWENS, J. D. "Research Challenges for on-Chip Interconnection Networks", *Micro IEEE*, v. 27, n. 5, pp. 96–108, September-October 2007.

[34] CEDER, A., NIGEL, H. M. W. "Bus Network Design", *Transportation Research Part B: Methodological*, v. 20, n. 4, pp. 331–334, 1986.

[35] TANNEMBAUM, A. S. *Computer Networks*. Prentice Hall, 2003.

[36] KERMANI, P., KLEINROCK, L. "Virtual cut-through a new computer communication switching technique." *Computer Networks*, v. 3, pp. 267–286, 1979.

[37] DALLY, W. J., SEITZ, C. "The torus routing chip", *Journal of Distributed Computing*, v. 1, pp. 187–196, 1986.

[38] DALLY, W. J. "Virtual-channel flow control." *Proc. of the International Symposium on Computer Architecture (ISCA)*, pp. 60–68, May 1990.

[39] DALLY, W. J. "Virtual-channel flow control." *IEEE Transactions on Parallel and Distributed Systems*, p. 194–205, March 1992.

[40] BENINI, L., MICHELI, G. D. *Networks on Chips: Technology and Tools*. Morgan Kaufman, 2006.

[41] DUATO, J. "A new theory of deadlock-free adaptive routing in wormhole networks", *IEEE Transactions on Parallel Distributed Systems*, v. 4, n. 12, pp. 1320–1331, 1993.

[42] DUATO, J. "A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks", *IEEE Transactions on Parallel Distributed Systems*, v. 6, n. 10, pp. 1055–1067, October 1995.

[43] MARTIN, A. J. "Remarks on Low-Power Advantages of Asynchronous Circuits", *ESSCIRC: Low- Power Systems on a Chip*, September 1998.

[44] VAN BERKEL, K. E. A. "Asynchronous Circuits for Low Power: A DCC Error Corrector", *Design & Test of Computers, IEEE*, v. 11, n. 2, pp. 22–32, 1994.

[45] PIGUET, C. "Low-Power and Low-Voltage CMOS Digital Design", *Microelectronic Engineering*, v. 39, n. 1, pp. 179–208, 1997.

[46] NIELSEN, L. E. A. "Low-Power Operation Using Self-timed Circuits and Adaptive Scaling of the Supply Voltage", *IEEE Transactions on VLSI*, v. 2, n. 4, pp. 391=397, 1994.

[47] BARBOSA, V., GAFNI, E. "Concurrency in Heavily loaded Neighborhood-Constrained Systems", *ACM Transactions on Programming Language and Systems (TOPLAS)*, v. 11, n. 4, pp. 562–584, October 1989.

[48] CASSIA, R. D. F. *Uma Metodologia Automatizável para Conversão Síncrono-Assíncrono de Circuitos Digitais*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, COPPE, 2007.

[49] VALMIR, C. B. *An Introduction to Distributed Algorithms*. The MIT Press, 2003.

[50] KAROL, M. J., HLUCHYJ, M. G., MORGAN, S. P. "Input versus Output queuing on a space-division packet switch", *IEEE Trans. Communication.*, v. 35, n. 12, pp. 1347–1356, December 1987.

[51] GLASS, C., NI, L. "The turn model for adaptive routing", *Proceeding of the 15th Annual International Symposium on Computer Architecture*, pp. 278–287, 1992.

[52] SANTOS, I. M., FRANÇA, F. M. G., GOULART, V. M. "Flexible Router - A new approach for buffer utilization", *To be published at 9th International Conference on Systems and Networks Communications*, October 2014.

[53] "ISE Simulator (ISim)". Disponível em: <`http://www.xilinx.com/tools/isim.htm`>.

[54] HOSSAM, E.-S. A.-F. H. *Hardware Implementation and Evaluation of the Flexible Router for NoC*. Tese de Mestrado, Egypt-Japan University of Science and Technology (E-JUST), 2014.

Apêndice A

Gráficos de *Throughput* e Latência

Neste apêndice são exibidos os gráficos com os resultados de *throughput* e latência para as configurações não mostradas no Capítulo 6. Este apêndice está dividido em duas seções: A primeira exibe os resultados para redes com 2 canais virtuais e a segunda exibe os resultados para redes com 4 canais virtuais.

A.1 Gráficos de redes com 4 canais virtuais

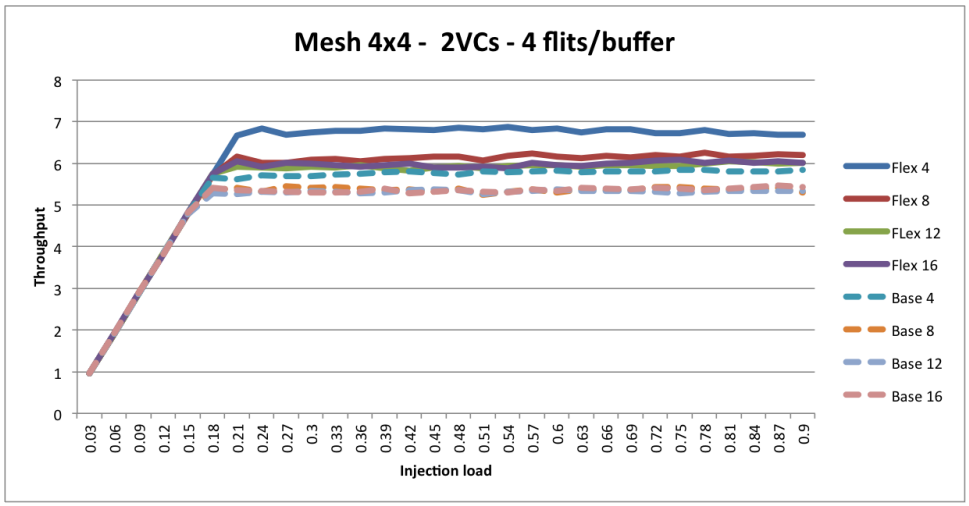


Figura A.1: Throughput - Mesh 4x4 com 2 Canais Virtuais e 4 flits por buffer.

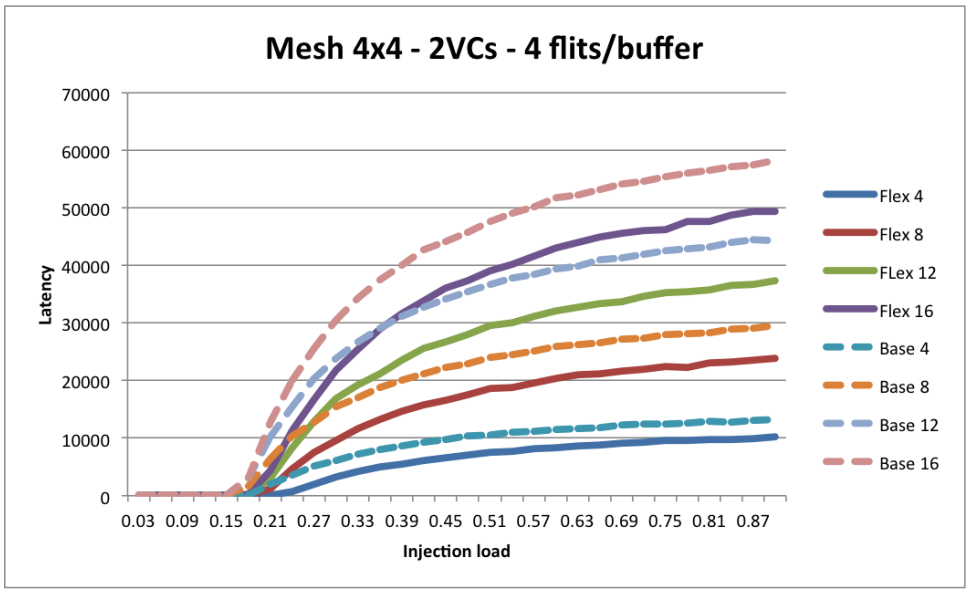


Figura A.2: Latência - Mesh 4x4 com 2 Canais Virtuais e 4 flits por buffer.

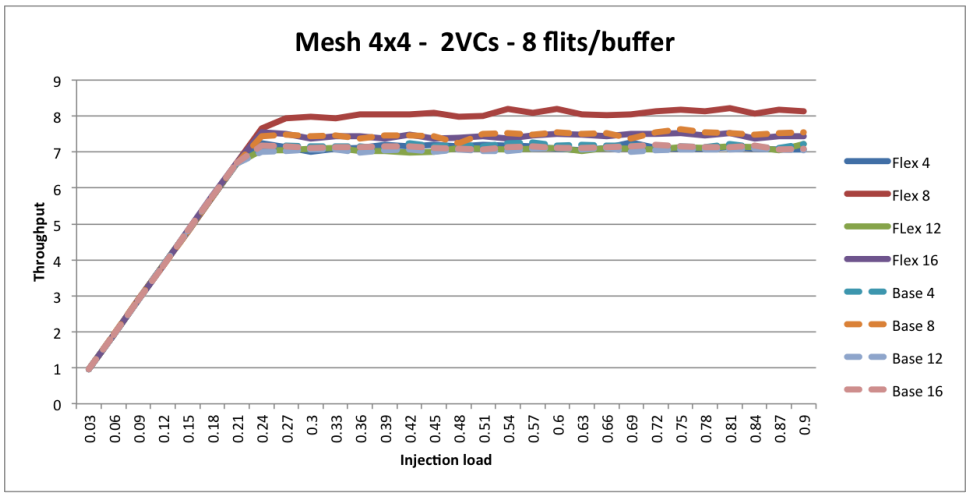


Figura A.3: Throughput - Mesh 4x4 com 2 Canais Virtuais e 8 flits por buffer.

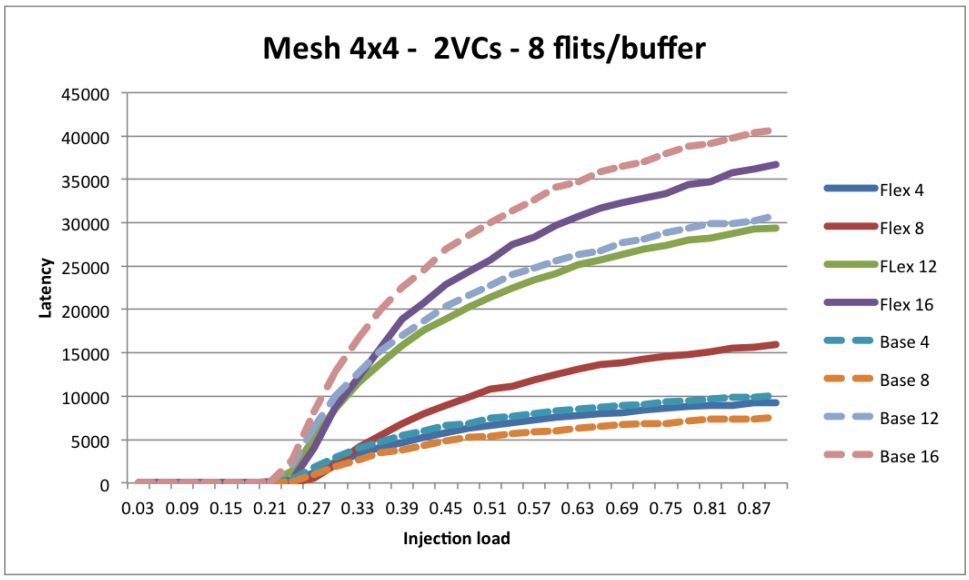


Figura A.4: Latência - Mesh 4x4 com 2 Canais Virtuais e 8 flits por buffer.

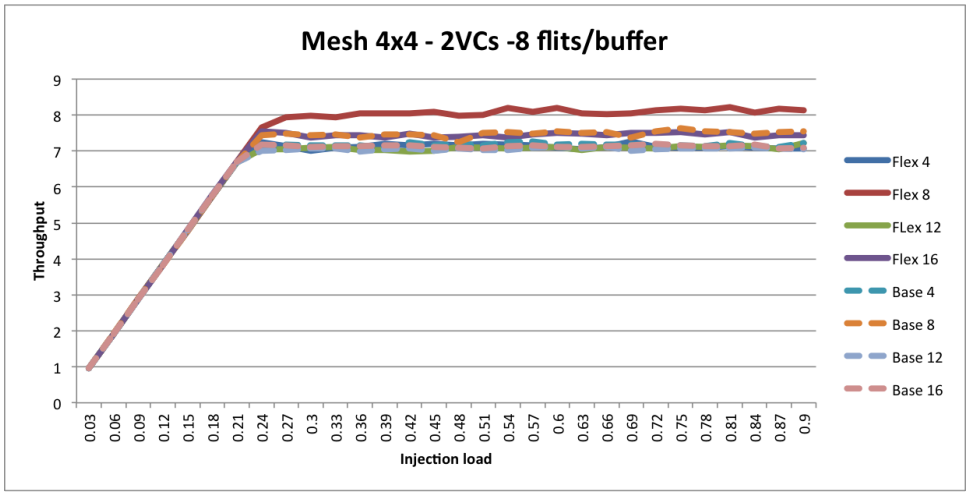


Figura A.5: Throughput - Mesh 4x4 com 2 Canais Virtuais e 16 flits por buffer.

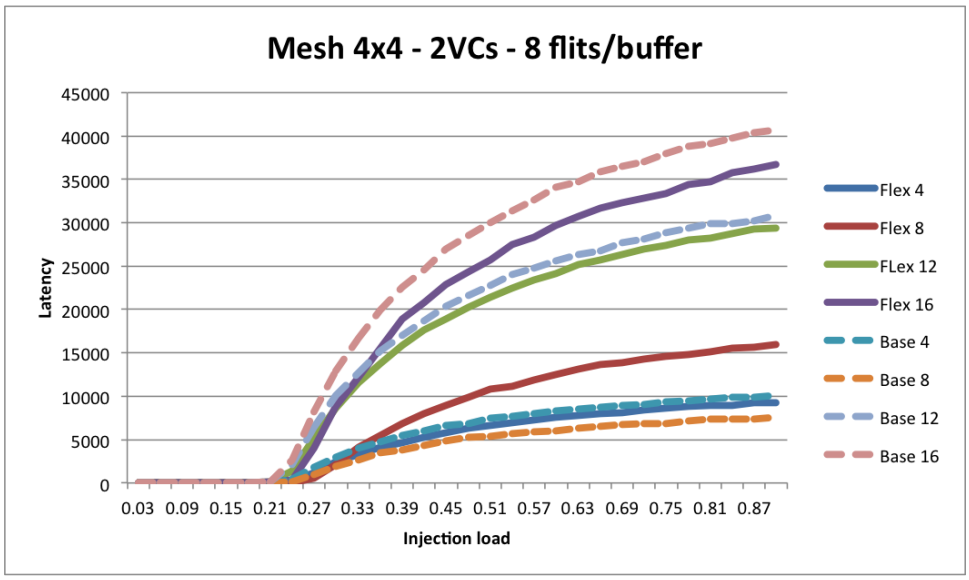


Figura A.6: Latência - Mesh 4x4 com 2 Canais Virtuais e 16 flits por buffer.

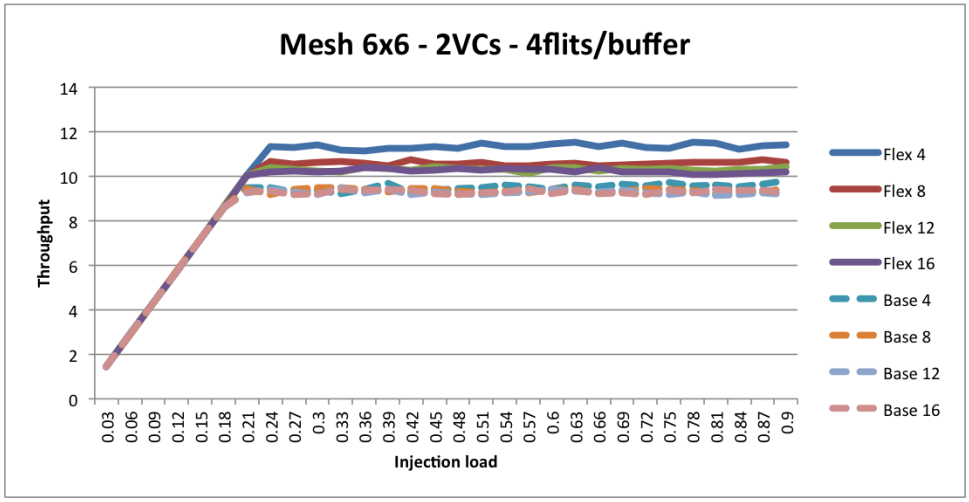


Figura A.7: Throughput - Mesh 6x6 com 2 Canais Virtuais e 4 flits por buffer.

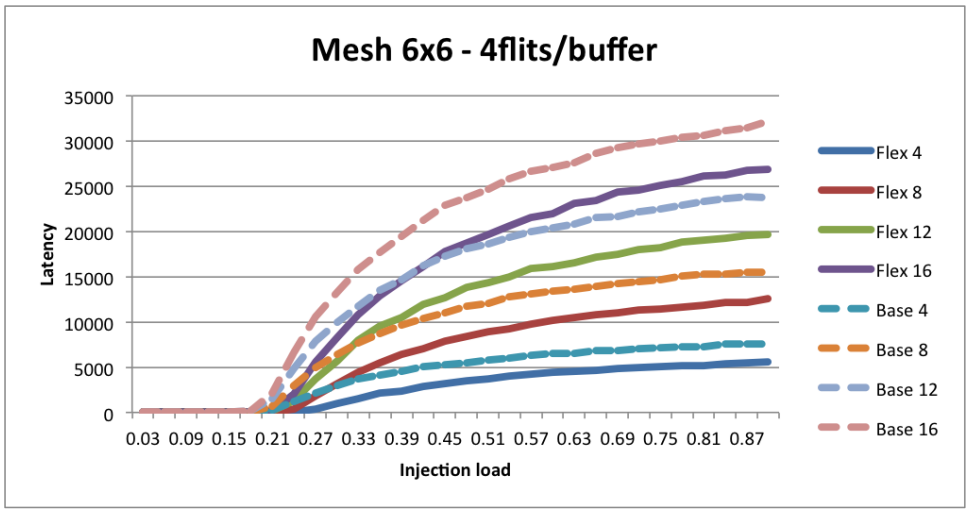


Figura A.8: Latência - Mesh 6x6 com 2 Canais Virtuais e 4 flits por buffer.

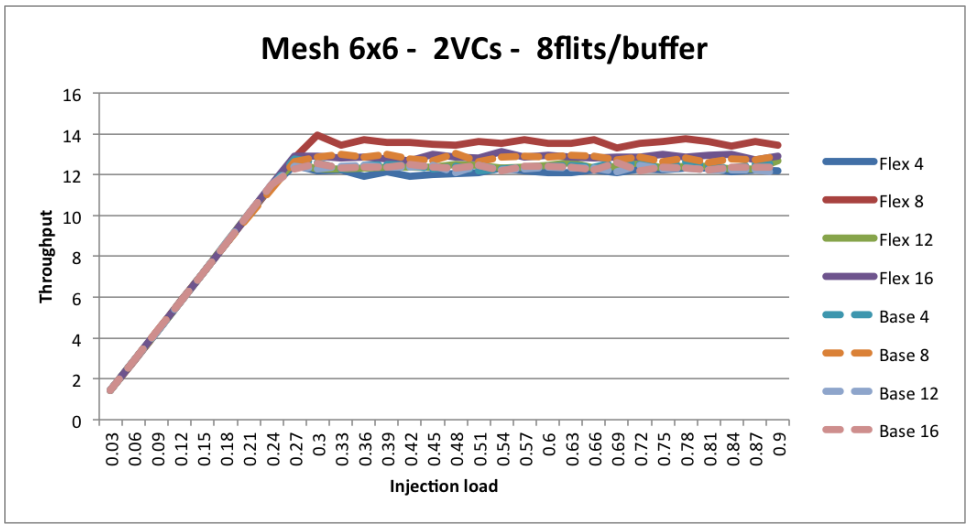


Figura A.9: Throughput - Mesh 6x6 com 2 Canais Virtuais e 8 flits por buffer.

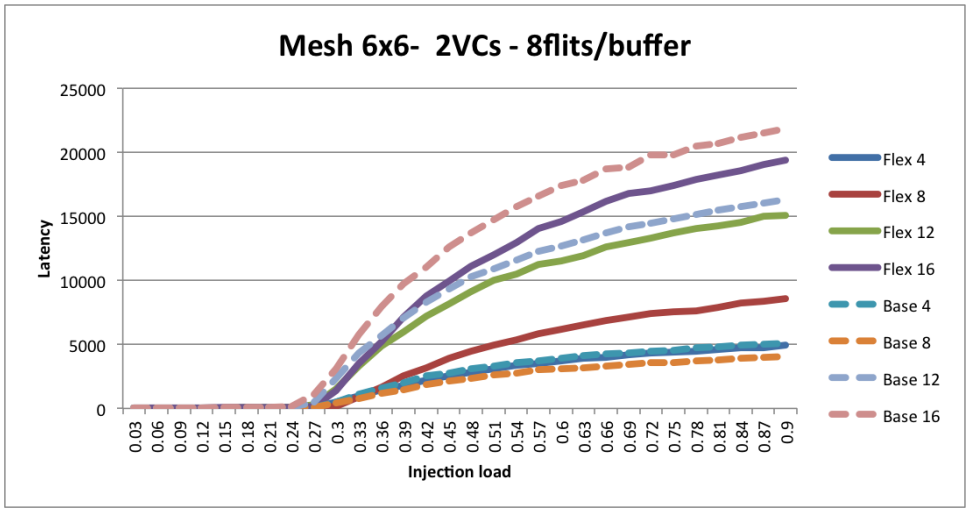


Figura A.10: Latência - Mesh 6x6 com 2 Canais Virtuais e 8 flits por buffer.

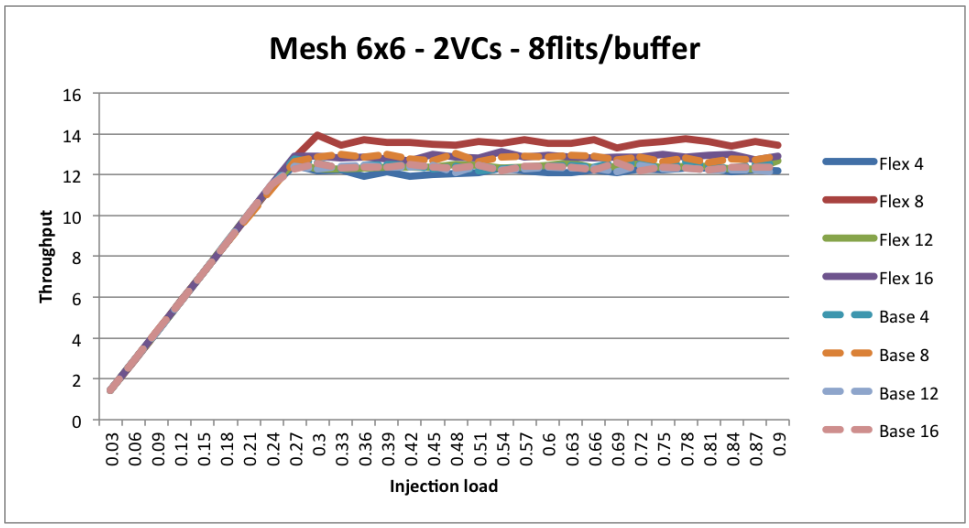


Figura A.11: Throughput - Mesh 6x6 com 2 Canais Virtuais e 16 flits por buffer.

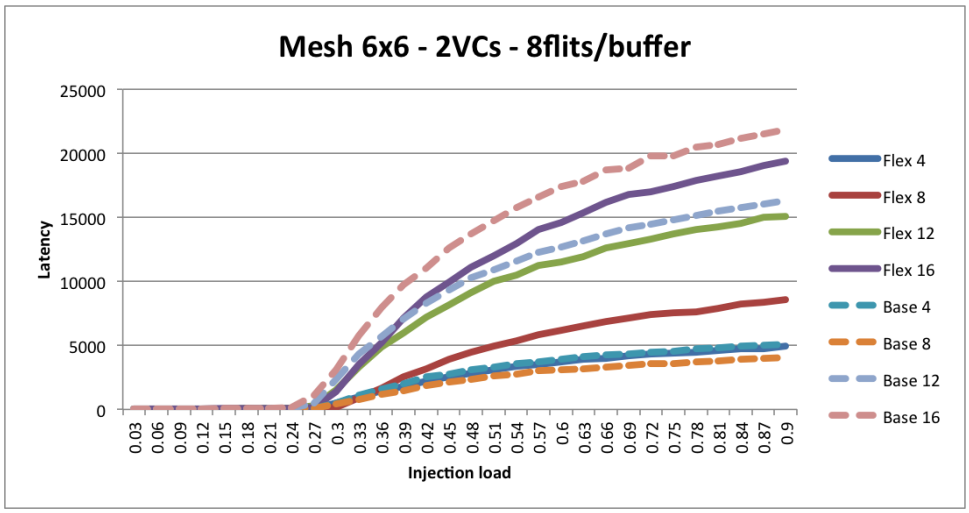


Figura A.12: Latência - Mesh 6x6 com 2 Canais Virtuais e 16 flits por buffer.

A.2 Gráficos de redes com 4 canais virtuais

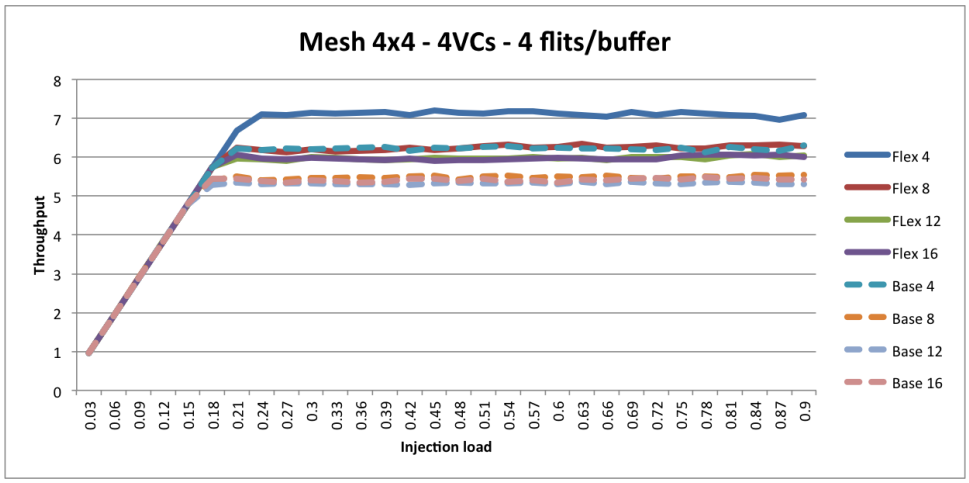


Figura A.13: Throughput - Mesh 4x4 com 4 Canais Virtuais e 4 flits por buffer.

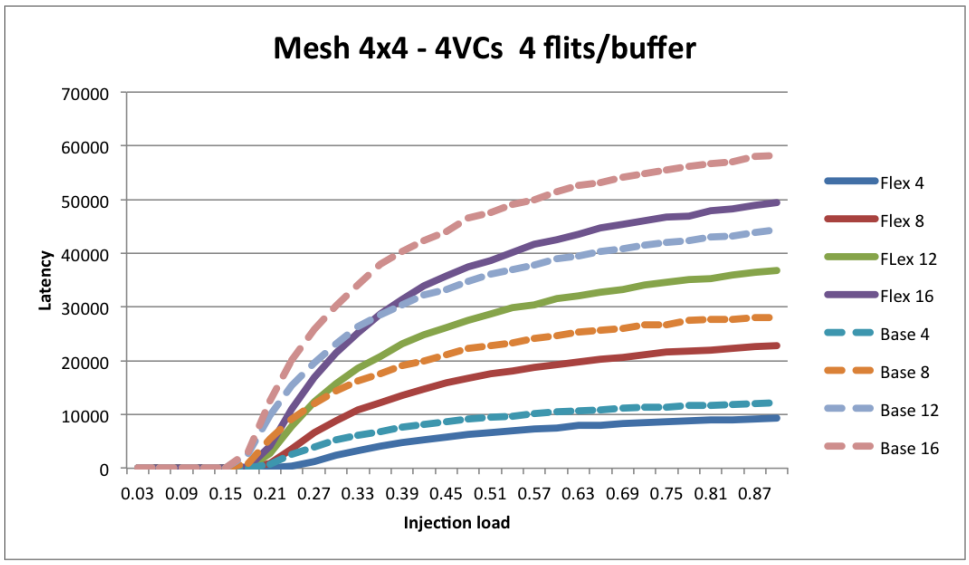


Figura A.14: Latência - Mesh 4x4 com 4 Canais Virtuais e 4 flits por buffer.

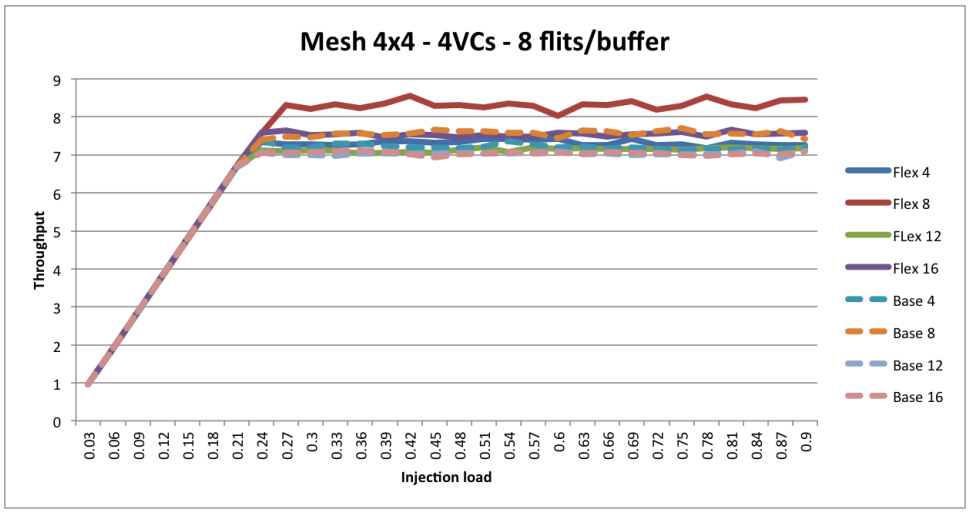


Figura A.15: Throughput - Mesh 4x4 com 4 Canais Virtuais e 8 flits por buffer.

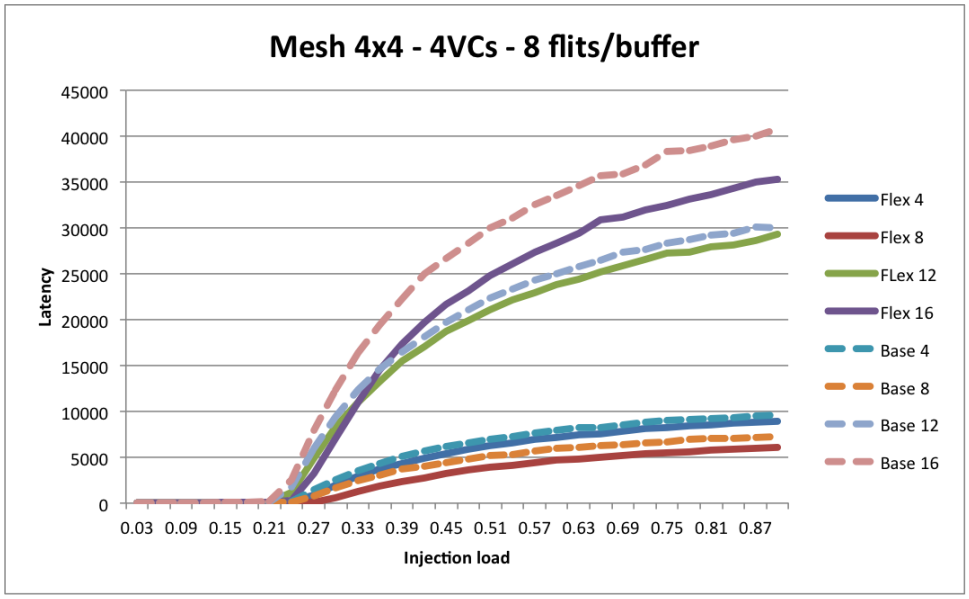


Figura A.16: Latência - Mesh 4x4 com 4 Canais Virtuais e 8 flits por buffer.

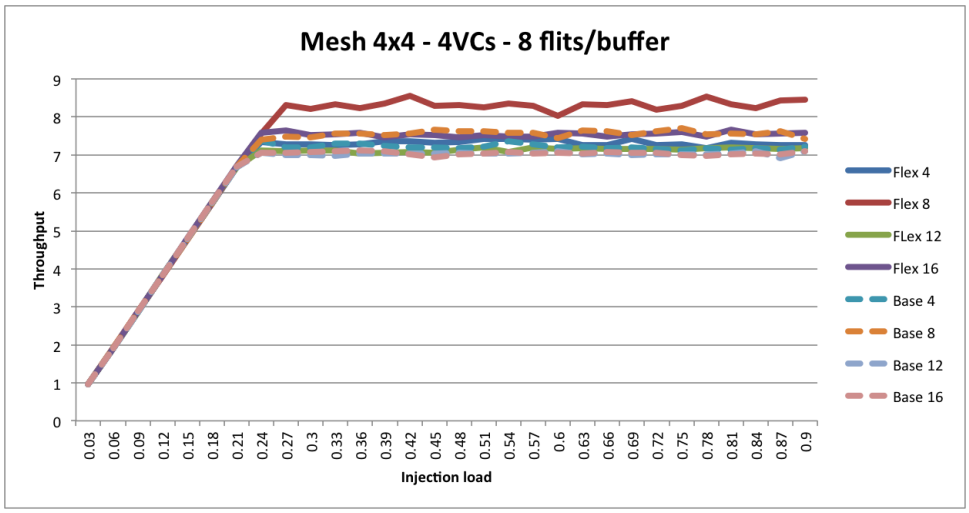


Figura A.17: Throughput - Mesh 4x4 com 4 Canais Virtuais e 16 flits por buffer.

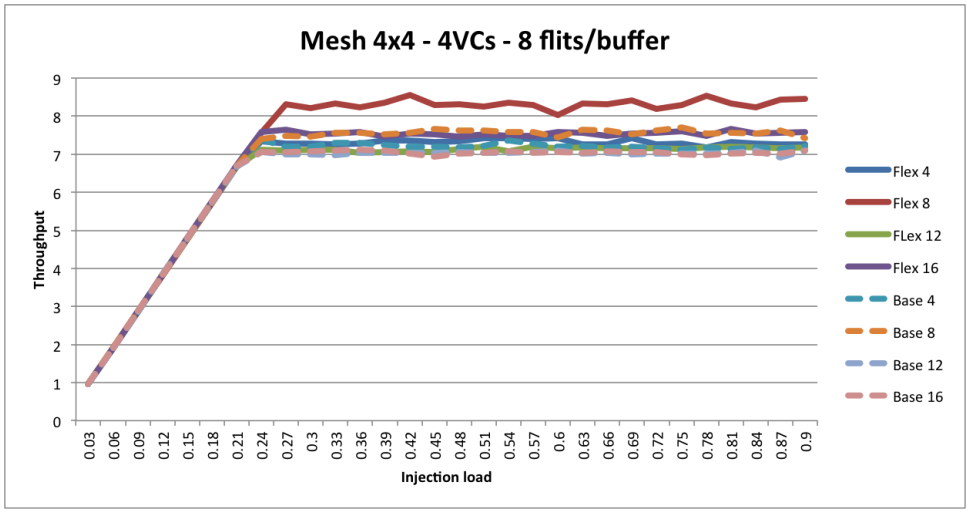


Figura A.18: Latência - Mesh 4x4 com 4 Canais Virtuais e 16 flits por buffer.

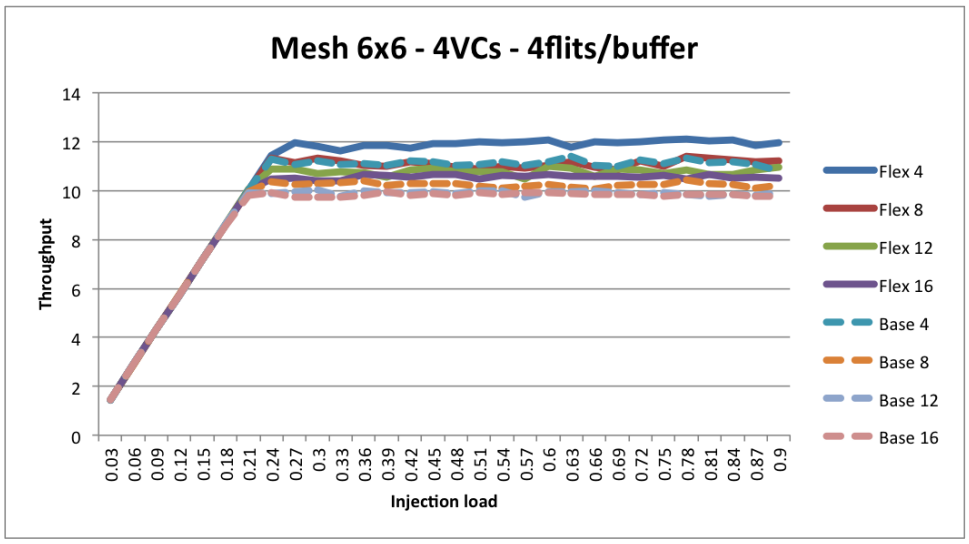


Figura A.19: Throughput - Mesh 6x6 com 4 Canais Virtuais e 4 flits por buffer.

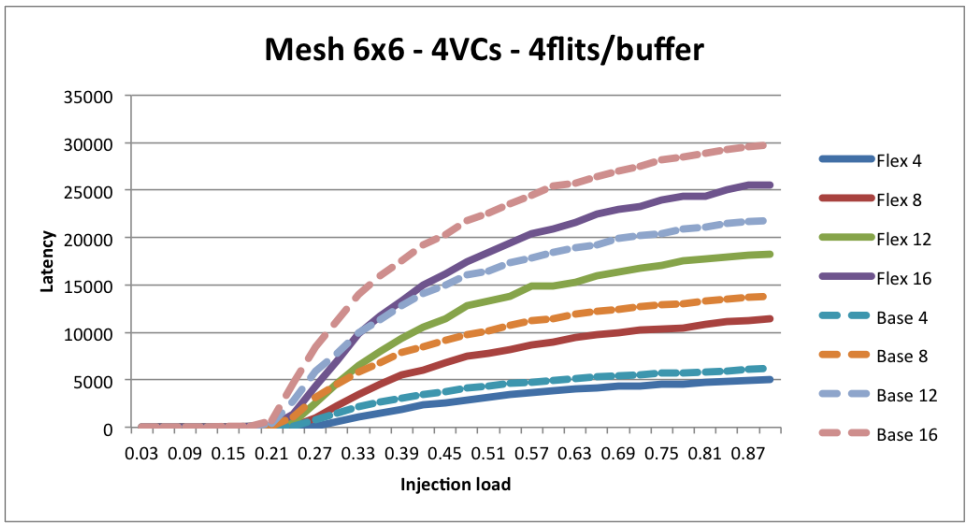


Figura A.20: Latência - Mesh 6x6 com 4 Canais Virtuais e 4 flits por buffer.

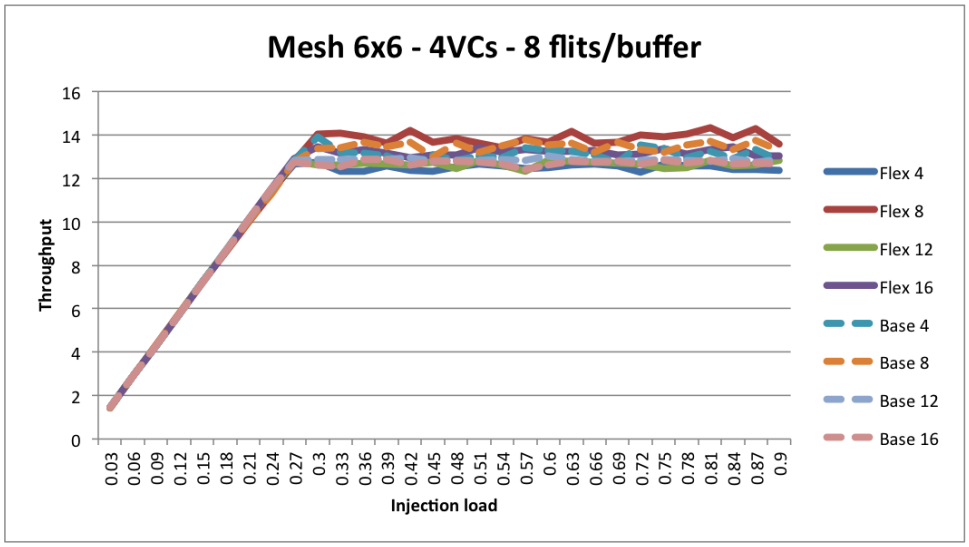


Figura A.21: Throughput - Mesh 6x6 com 2 Canais Virtuais e 8 flits por buffer.

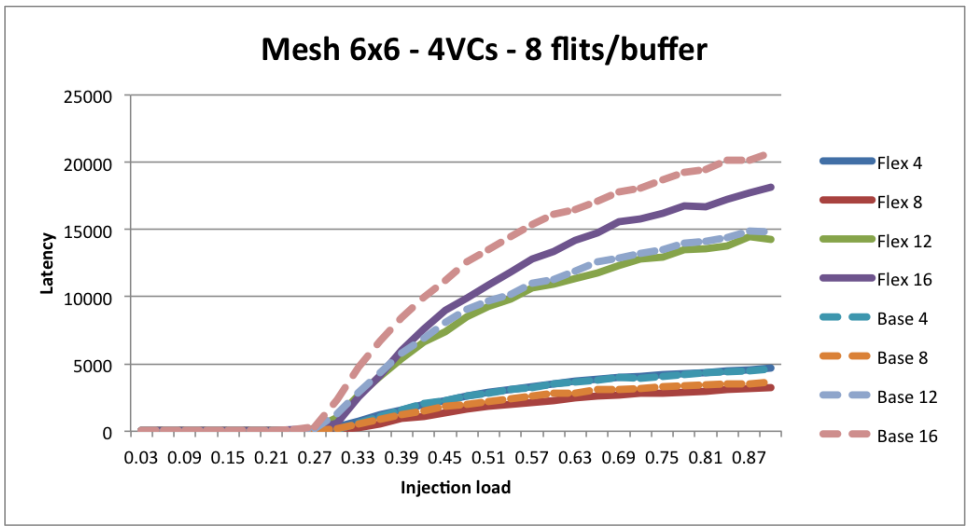


Figura A.22: Latência - Mesh 6x6 com 2 Canais Virtuais e 8 flits por buffer.

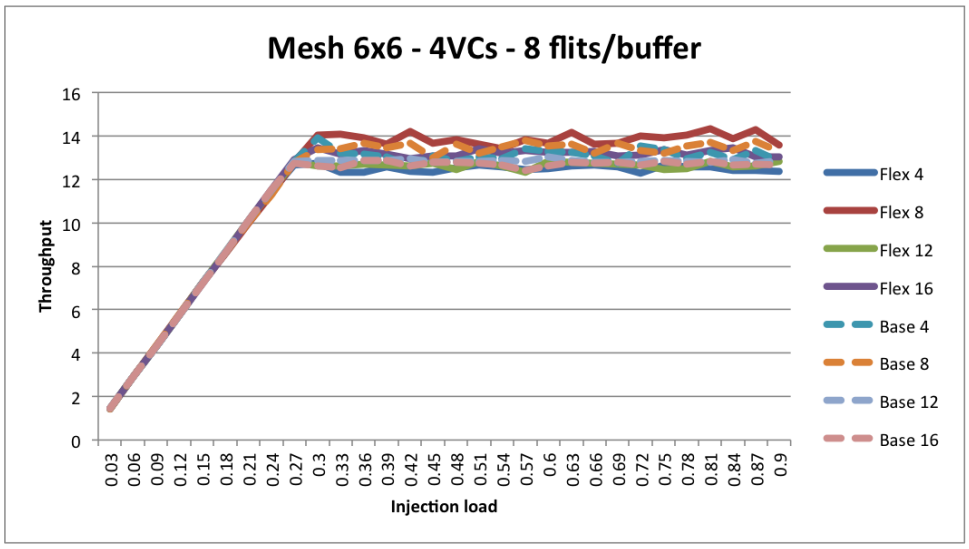


Figura A.23: Throughput - Mesh 6x6 com 2 Canais Virtuais e 16 flits por buffer.

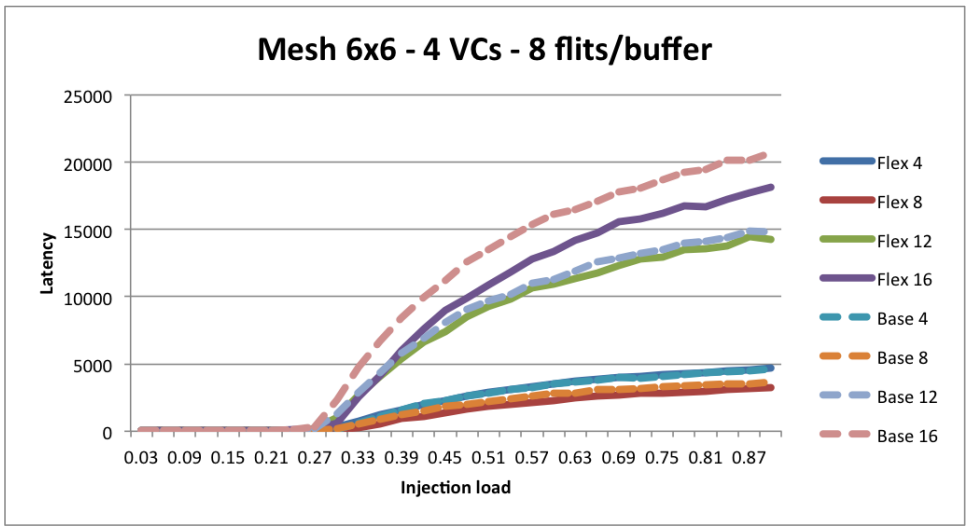


Figura A.24: Latência - Mesh 6x6 com 2 Canais Virtuais e 16 flits por buffer.